

INTRO TO MySQL FOR DATA ANALYSIS

★★★★★ With Expert SQL Instructor **John Pauler**



COURSE STRUCTURE



This is a **project-based course**, for students looking for a *practical, hands-on*, and *highly engaging* approach to querying and analyzing databases with MySQL

Additional resources include:



Downloadable Ebook to serve as a helpful reference when you're offline or on the go



Quizzes & Homework Exercises to test and reinforce key concepts, with step-by-step solutions



Bonus Projects to test your abilities and apply the skills developed throughout the course

COURSE OUTLINE

1	SQL Intro & Setup	<i>Explore the SQL landscape, download Community Server and Workbench, and create the project database</i>
2	Database Fundamentals (Part 1)	<i>Review basic database concepts, including table structure, columns and records, and primary vs. foreign keys</i>
3	Analyzing Data from Single Tables	<i>Use SQL queries to select data, apply filters and sorting rules, analyze segments of data, and evaluate conditional logic</i>
[MID-COURSE PROJECT]		
4	Database Fundamentals (Part 2)	<i>Understand table relationships, dimension vs. fact tables, and the basics of normalization and cardinality</i>
5	Analyzing Multiple Tables via Joins	<i>Learn the different types of joins, when to use each of them, and how to analyze data across multiple tables</i>

[FINAL PROJECT]

INTRODUCING THE COURSE PROJECT

THE SITUATION

You and your rich Uncle Jimmy just purchased **Maven Movies**, a brick and mortar DVD Rental business. Uncle Jimmy put up the money, and you're in charge of the day-to-day operations.

THE BRIEF

As a new owner, you'll need to learn *everything* you can about your business: your product inventory, your staff, your customer purchase behaviors, etc.

You have access to the entire Maven Movies SQL database, but the remaining employees are not able to give you much direction. **You'll need to analyze everything on your own.**

THE OBJECTIVE

Use MySQL to:

- *Access and explore the Maven Movies database*
- *Develop a firm grasp of the 16 database tables and how they relate to each other*
- *Analyze all aspects of the company's data, including transactions, customers, staff, etc.*

SETTING EXPECTATIONS

- 1 You'll be learning **MySQL**, and practicing that using **MySQL Workbench****
 - In your career, you may end up using other "flavors" of SQL (T-SQL, PL/SQL, PostgreSQL, etc.)*
 - Each flavor is very similar, with only minor syntax changes; the concepts you learn will apply universally*
- 2 This course is designed to get you **up & running** with SQL for analysis**
 - The goal is to provide a **foundational understanding** of SQL and relational databases; some concepts may be covered at a high level, and we won't cover everything that SQL can do in the scope of this course*
- 3 The course focuses on **understanding, extracting, and analyzing data****
 - This course is ideal for people who want a firm understanding of analyzing data in relational databases*
 - We start with the basics, so feel free to skip ahead as you see fit if you have had some SQL exposure in the past*
- 4 We will **NOT** cover **building & maintenance of databases** in this course**
 - This course is more oriented toward analysis, and will not get into typical database administrator tasks*
 - We will **NOT** cover database or table creation, altering or dropping tables, or managing user permissions*

INTRODUCING SQL

WHY LEARN SQL?

- ★ SQL is the **standard language** for relational database management
- ★ You can **load, query** and **analyze massive data sets** using SQL
- ★ Learning SQL is **fun, intuitive** and **surprisingly easy**
- ★ Companies **always** need employees who know SQL
- ★ Becoming a database + SQL expert has **never-ending practical benefits**

BRIEF HISTORY OF SQL

- SQL stands for **Structured Query Language**, and was designed in the early 1970s at IBM to manipulate and retrieve data stored in a **relational database management system** (*RDMS*)
- There is a **universal standard** for SQL set by the International Organization for Standards and the American National Standards Institute (*ANSI*), with updates released every **~3-5 years**
- Vendors are constantly adding new features on top of the standards, which creates different “*flavors*” of SQL (***MySQL, PostgreSQL, SQLite, etc.***)

COMMON FLAVORS OF SQL



HEY THIS IS IMPORTANT!

These flavors of SQL are much more *similar* than they are different – all are based on the same universal standard, with slight variations in syntax.

POPULAR MySQL EDITORS



Mac, Windows, Linux



Web App (works on any OS)



Windows only



Sequel Pro

Mac only



HEY THIS IS IMPORTANT!

We'll be practicing **MySQL** using **MySQL Workbench**, but rather than thinking "*I learned MySQL*" or "*I can use Workbench*", you should think "*I'm learning SQL*". Period. You'll be able to use the concepts and syntax you learn here universally.

DOWNLOAD & SETUP

MySQL DOWNLOAD & SETUP – OVERVIEW

Step 1	Download Community Server	<i>This allows SQL to run on your machine</i>
Step 2	Download MySQL Workbench	<i>This is the program you'll use to write and run SQL queries (it's intuitive, and works across operating systems)</i>
Step 3	Connect Workbench to Server	<i>We'll get you connected to the server so you can use Workbench to start running your own SQL queries</i>
Step 4	Review Workbench Interface	<i>We'll take a quick tour of the Workbench interface to get you familiar with the layout and key components</i>
Step 5	Create the Database	<i>We'll run the SQL code to build the 16 table database which we'll be exploring throughout the course (this part is easy!)</i>

SERVER

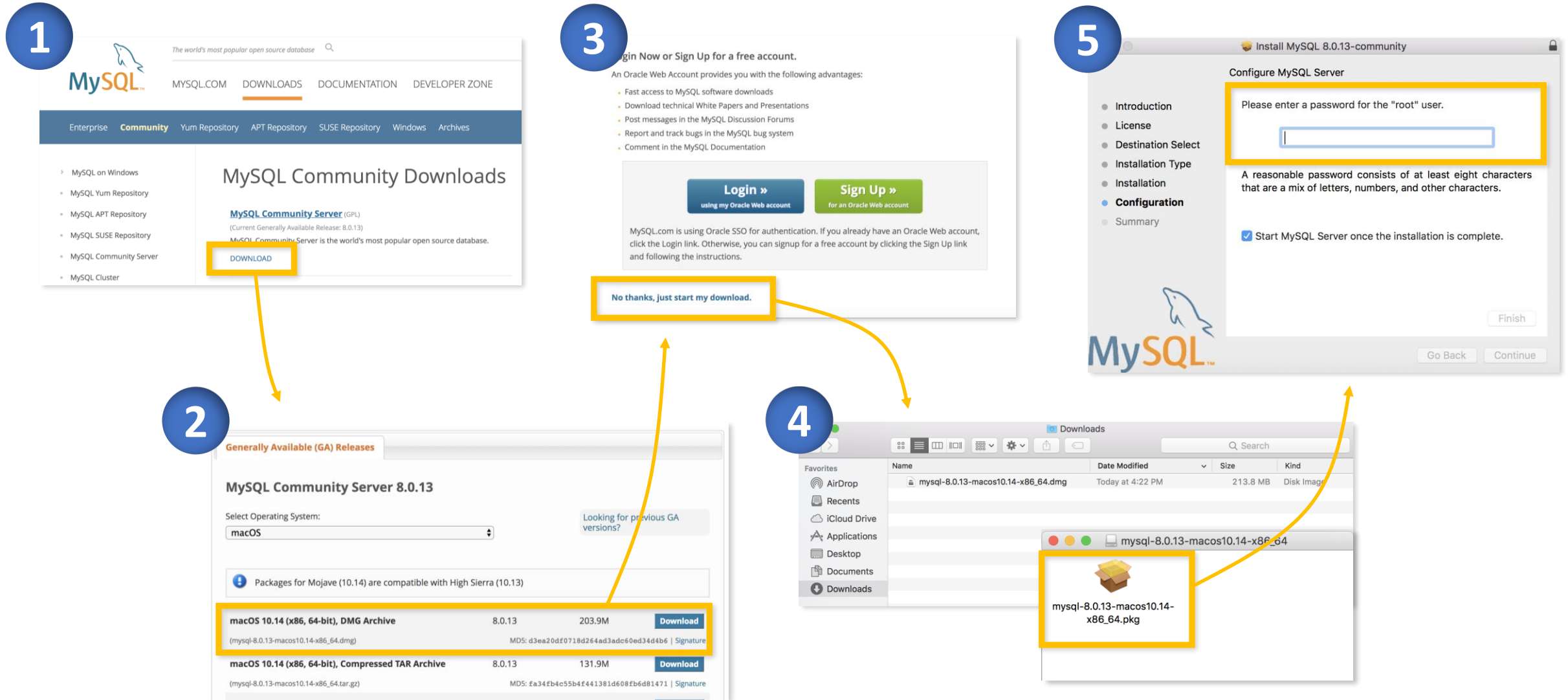
STEP 1: COMMUNITY SERVER (MAC)

MySQL COMMUNITY SERVER – MAC DOWNLOAD GUIDE



- 1 Go to <https://dev.mysql.com/downloads> and download **MySQL Community Server**
- 2 Select the **MacOS** operating system, and download the **DMG Archive** version
 - **Note:** *you'll likely see a later version than the one shown (just download the latest)*
- 3 No need to Login or Sign Up, just click “***No thanks, just start my download***”
- 4 Find the install file in your downloads, then double click to run the installer package
- 5 Click through each install step, leaving defaults unless you need customized settings
 - **Note:** *Make sure to store your **root password** somewhere, you'll need this later!*

MySQL COMMUNITY SERVER – MAC DOWNLOAD GUIDE



MySQL

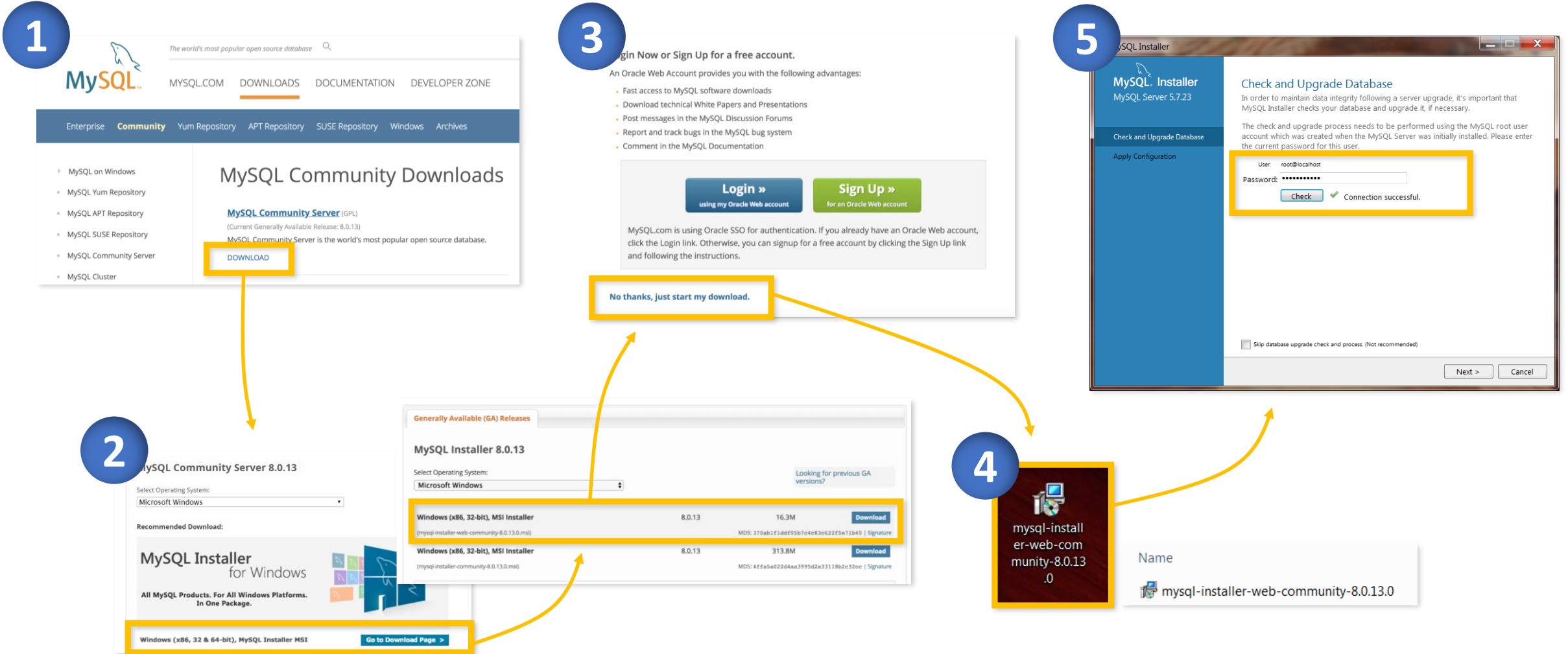
STEP 1: COMMUNITY SERVER (PC)

MySQL COMMUNITY SERVER – PC DOWNLOAD GUIDE



- 1 Go to <https://dev.mysql.com/downloads> and download **MySQL Community Server**
- 2 Select the **Microsoft Windows** operating system, and the **Installer MSI** download
 - ***Note:** On the download page you may see two versions: select **mysql-installer-web-community** if you are connected to the internet, and keep in mind that you may see a later version than the one shown (just download the latest)*
- 3 No need to Login or Sign Up, just click “***No thanks, just start my download***”
- 4 Find the install file in your downloads, then double click to run the installer package
- 5 Click through each install step, leaving defaults unless you need customized settings
 - ***Note:** Make sure to store your **root password** somewhere, you’ll need this later!*

MySQL COMMUNITY SERVER – PC DOWNLOAD GUIDE



STEP 2: MySQL WORKBENCH (MAC)

MySQL WORKBENCH – MAC DOWNLOAD GUIDE



- 1 Go to <https://dev.mysql.com/downloads/workbench>, scroll down to **Generally Available (GA) Releases**, and select the **MacOS** operating system
- 2 We'll be using version **8.0.16** for this course, so you can either click "***Looking for previous GA versions?***" to search for the same one, or simply download the latest available
- 3 No need to Login or Sign Up, just click "***No thanks, just start my download***"
- 4 Find the install file in your downloads, click the MySQL Workbench logo (*with the dolphin*) and drag it into your **Applications** folder
- 5 Look for MySQL workbench in your list of applications, double click to launch, then proceed to ***Step 3: Connecting to the server***

MySQL WORKBENCH – MAC DOWNLOAD GUIDE



WORKBOOK

STEP 2: MySQL WORKBENCH (PC)

MySQL WORKBENCH – PC DOWNLOAD GUIDE

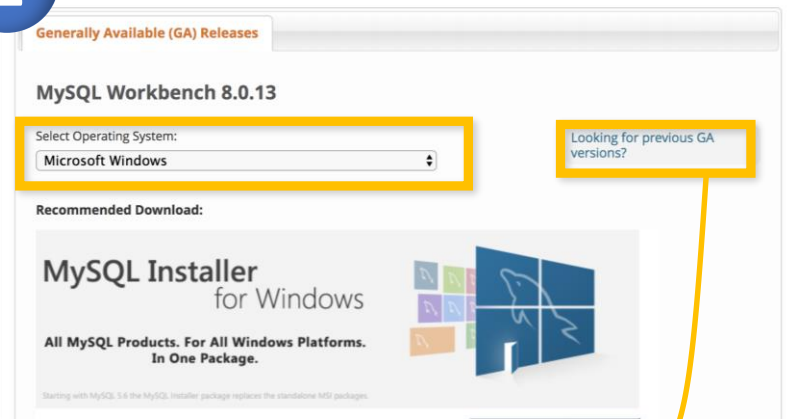


- 1 Go to <https://dev.mysql.com/downloads/workbench>, scroll down to **Generally Available (GA) Releases**, and select the **Microsoft Windows** operating system
- 2 We'll be using version **8.0.13** for this course, so you can either click ***"Looking for previous GA versions?"*** to search for the same one, or simply download the latest available
- 3 No need to Login or Sign Up, just click ***"No thanks, just start my download"***
- 4 Find the install file in your downloads, double click to run the installation process, and stick with default settings unless you need a custom configuration
- 5 Look for MySQL workbench in your list of programs, double click to launch, then proceed to ***Step 3: Connecting to the server***
 - **Note:** You may see a warning if you aren't on **Windows 10+**, but most older systems (i.e. Windows 7) should be compatible

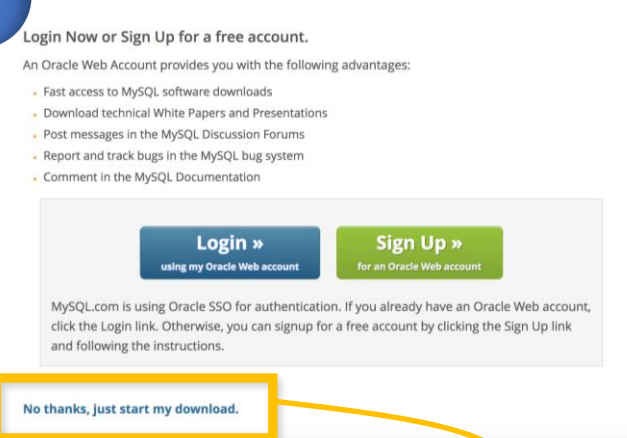
MySQL WORKBENCH – PC DOWNLOAD GUIDE



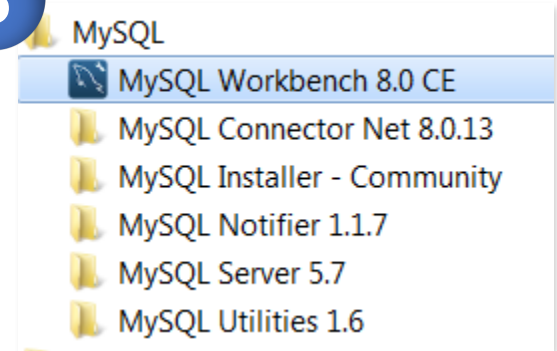
1



3

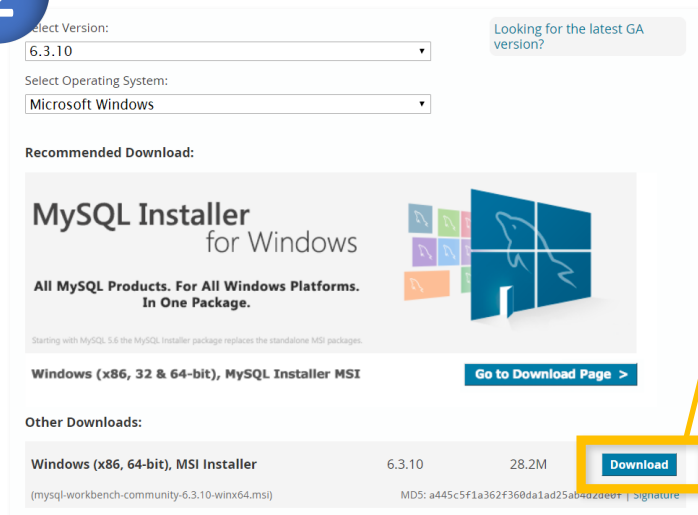


5

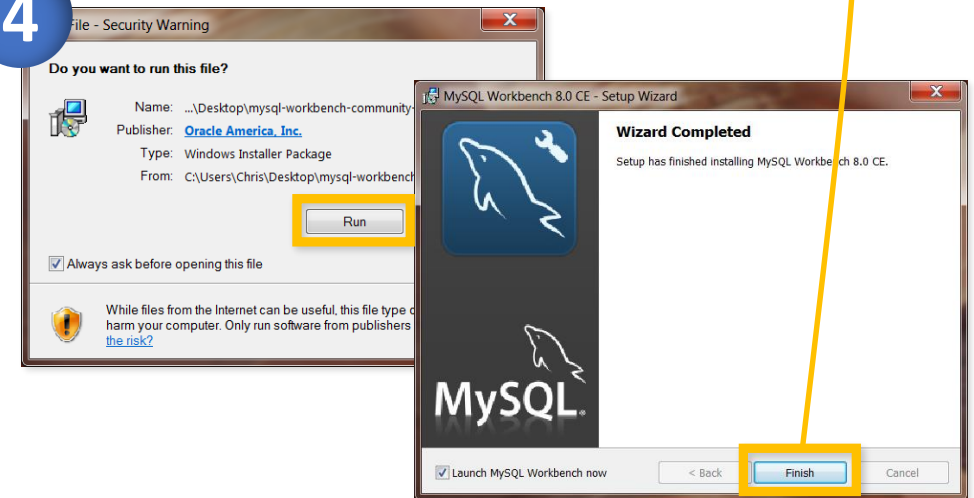


2

Look for version **8.0.13**, or download the latest



4



STEP 3: CONNECTING TO THE SERVER

phpMyAdmin

Current server:

phpMyAdmin demo - My

rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
1	2005-05-24 22:53:30	367	130	2005-05-26 22:04:30	1	2006-02-15 21:30:53
2	2005-05-24 22:54:33	1525	459	2005-05-28 19:40:33	1	2006-02-15 21:30:53
3	2005-05-24 21:03:39	1711	409	2005-06-01 22:13:39	1	2006-02-15 21:30:53
4	2005-05-24 10:44:22	2422	251	2005-05-01 10:41:41	2	2006-02-15 21:30:53
5	2005-05-24 05:52:20	2307	222	2005-06-02 04:21:21	2	2006-02-15 21:30:53
6	2005-05-24 08:09:09	2718	549	2005-06-27 01:07:07	1	2006-02-15 21:30:53
7	2005-05-24 11:58:29	2999	291	2005-06-29 20:05:33	2	2006-02-15 21:30:53
8	2005-05-24 23:31:46	2346	239	2005-05-27 23:33:46	2	2006-02-15 21:30:53
9	2005-05-25 00:00:40	2580	126	2005-05-26 00:22:40	1	2006-02-15 21:30:53
10	2005-05-25 00:02:21	1824	399	2005-05-31 22:44:21	2	2006-02-15 21:30:53

Showing rows 0 - 24 (599 total, Query took 0.0212 seconds.)

```
1 SELECT
2   customer.first_name,
3   first_name,
4   COUNT(DISTINCT rental_id) AS rentals
5 FROM rental
6 LEFT JOIN customer
7   ON customer.customer_id = rental.customer_id
8 GROUP BY
9   customer.first_name,
10  first_name
11 ORDER BY
12  COUNT(DISTINCT rental_id) DESC
```

Table	Action	Index	Type	Collation	Size	Overhead
actor	Browse Structure Search Insert Empty Drop	PRIMARY	MyISAM	utf8_general_ci	10.0 K B	-
actor_info	Browse Structure Search Insert Empty Drop	PRIMARY	MyISAM	utf8_general_ci	10.0 K B	-
address	Browse Structure Search Insert Empty Drop	PRIMARY	MyISAM	utf8_general_ci	10.0 K B	-
category	Browse Structure Search Insert Empty Drop	PRIMARY	MyISAM	utf8_general_ci	10.0 K B	-
equipment	Browse Structure Search Insert Empty Drop	PRIMARY	MyISAM	utf8_general_ci	10.0 K B	-
film	Browse Structure Search Insert Empty Drop	PRIMARY	MyISAM	utf8_general_ci	10.0 K B	-
film_actor	Browse Structure Search Insert Empty Drop	PRIMARY	MyISAM	utf8_general_ci	10.0 K B	-
film_category	Browse Structure Search Insert Empty Drop	PRIMARY	MyISAM	utf8_general_ci	10.0 K B	-
film_text	Browse Structure Search Insert Empty Drop	PRIMARY	MyISAM	utf8_general_ci	10.0 K B	-
inventory	Browse Structure Search Insert Empty Drop	PRIMARY	MyISAM	utf8_general_ci	10.0 K B	-

CONNECTING TO THE SERVER

- 1 After launching Workbench, check the **MySQL Connections** section on the welcome page
 - *If you see a connection already, right-click to **Edit Connection**, otherwise click the **plus sign (+)** to add a new one*
- 2 Name the connection “**MavenMovies**”, confirm that the Username is “**root**”, and click **OK**
- 3 Once you see the **MavenMovies** connection on your welcome screen, simply click the tile and enter your **root password** to complete the connection

CONNECTING TO THE SERVER

1 Welcome to MySQL Workbench

MySQL Workbench is the official graphical user interface (GUI) tool for MySQL. It allows you to design, create and browse your database schemas, work with database objects and insert data as well as design and run SQL queries to work with stored data. You can also migrate schemas and data from other database vendors to your MySQL database.

MySQL Connection

MySQL Workbench could not detect any MySQL server running. This means that MySQL is not installed or is not running. [Rescan servers](#)

2 Setup New Connection

Connection Name: **MavenMovies** Type a name for the connection

Connection Method: Standard (TCP/IP) Method to use to connect to the RDBMS

Parameters SSL Advanced

Hostname: 127.0.0.1 Port: 3306 Name or IP address of the server host - and TCP/IP port.

Username: **root** Name of the user to connect with.

Password: Store in Keychain ... Clear The user's password. Will be requested later if it's not set.

Default Schema: The schema to use as default schema. Leave blank to select it later.

Configure Server Management... Test Connection Cancel **OK**

3 Welcome to MySQL Workbench

MySQL Connections

MavenMovies
root
127.0.0.1:3306

4 Connect to MySQL Server

Please enter password for the following service:

Service: Mysql@127.0.0.1:3306

User: root

Password:

☐ Save password in keychain

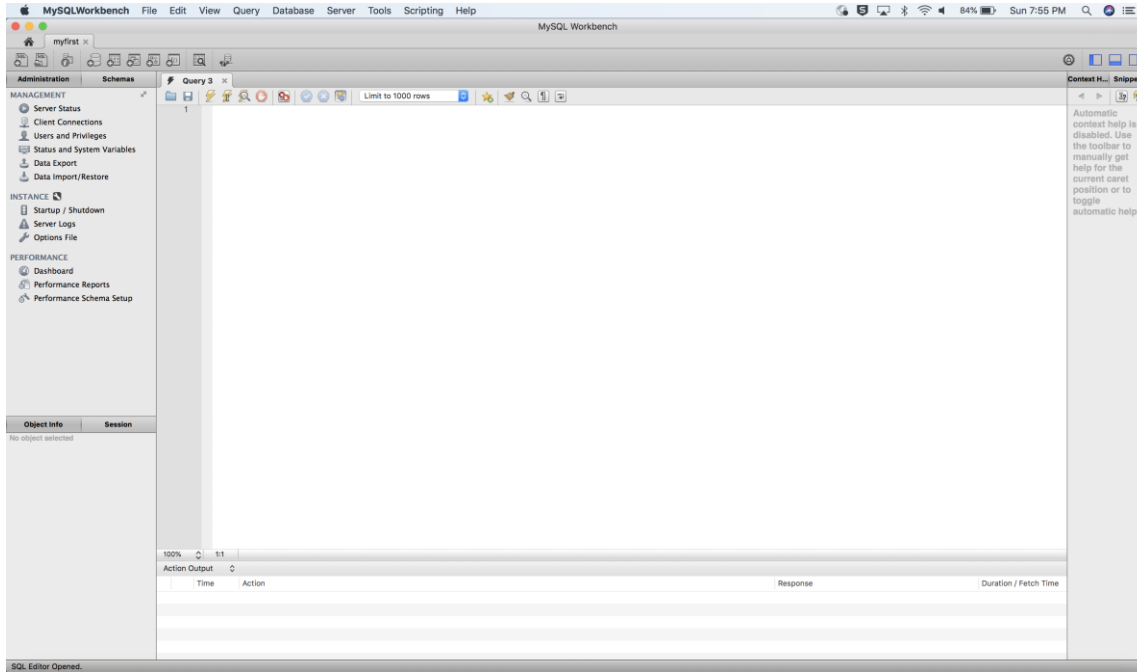
Cancel OK

STEP 4: MySQL WORKBENCH INTERFACE

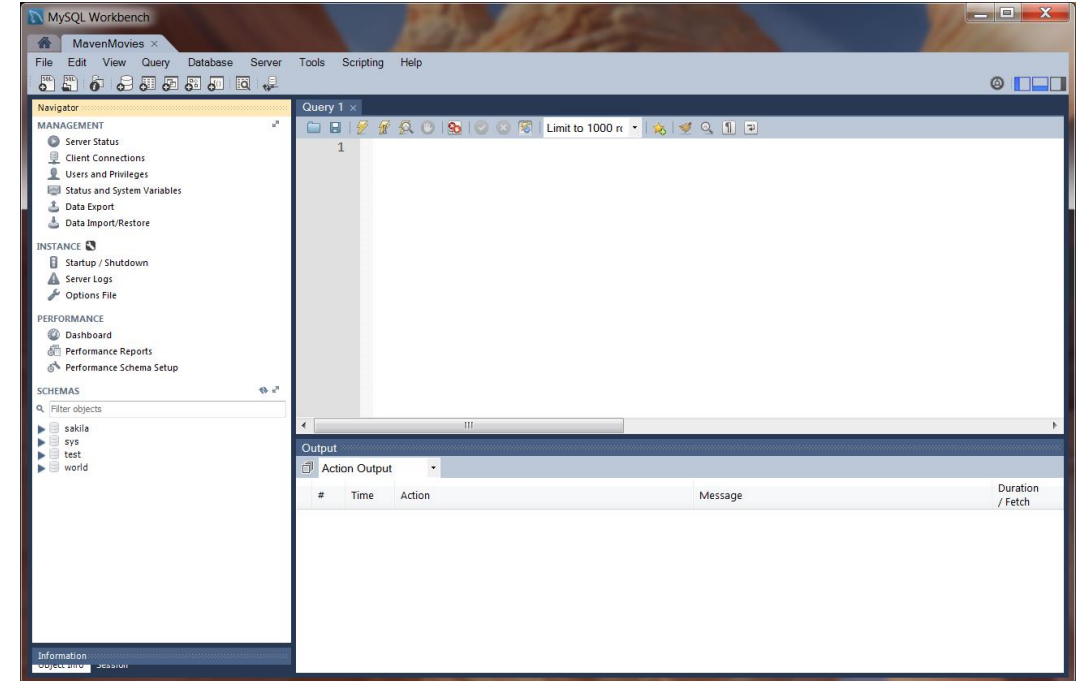
MySQL WORKBENCH INTERFACE (MAC VS. PC)



Mac interface



PC interface



HEY THIS IS IMPORTANT!

Workbench looks slightly different on **Mac** vs. **PC**, but everything you need is found in the same place. While the course is recorded on a Mac, but you should have no problem keeping up on a PC

QUICK TOUR: THE WORKBENCH INTERFACE

The screenshot shows the MySQL Workbench interface with the following components and annotations:

- Query Editor Window:** An arrow points to the central area where the SQL query `SELECT * FROM rental;` is written. The text below it says: *This is where you write and run your code*.
- Schemas Tab:** An arrow points to the left-hand sidebar showing the database schema hierarchy (Administration, Schemas, mavenmovies, Tables, Views, Stored Procedures, Functions, sys). The text below it says: *Here you can view tables and views in your database*.
- Result Grid:** An arrow points to the table of results displayed below the query editor. The text to its right says: *After running your SQL queries, your results appear here*.
- Action Output:** An arrow points to the bottom section showing the execution log. The text to its right says: *This is a summary of actions taken by the server (TIP: the 'Response' column is great for troubleshooting errors!)*.

rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
1	2005-05-24 22:53:30	367	130	2005-05-26 22:04:30	1	2006-02-15 21:30:53
2	2005-05-24 22:54:33	1525	459	2005-05-28 19:40:33	1	2006-02-15 21:30:53
3	2005-05-24 23:03:39	1711	408	2005-06-01 22:12:39	1	2006-02-15 21:30:53
4	2005-05-24 23:04:41	2452	333	2005-06-03 01:43:41	2	2006-02-15 21:30:53
5	2005-05-24 23:05:21	2079	222	2005-06-02 04:33:21	1	2006-02-15 21:30:53
6	2005-05-24 23:08:07	2792	549	2005-05-27 01:32:07	1	2006-02-15 21:30:53
7	2005-05-24 23:11:53	3995	269	2005-05-29 20:34:53	2	2006-02-15 21:30:53
8	2005-05-24 23:31:46	2346	239	2005-05-27 23:33:46	2	2006-02-15 21:30:53
9	2005-05-25 00:00:40	2580	126	2005-05-28 00:22:40	1	2006-02-15 21:30:53
10	2005-05-25 00:02:21	1824	399	2005-05-31 22:44:21	2	2006-02-15 21:30:53
11	2005-05-25 00:09:02	4443	142	2005-06-02 20:56:02	2	2006-02-15 21:30:53
12	2005-05-25 00:19:27	1584	261	2005-05-30 05:44:27	2	2006-02-15 21:30:53

Time	Action	Response	Duration / Fetch Time
15:34:40	SELECT * FROM rental LIMIT 0, 50000	16044 row(s) returned	0.015 sec / 0.015 sec

STEP 5: CREATING THE DATABASE

The screenshot displays the phpMyAdmin web interface. On the left, a sidebar shows the database structure with a tree view. The main area is divided into three panes. The top pane shows the 'rental' table structure with columns: rental_id, rental_date, inventory_id, customer_id, return_date, staff_id, and last_update. The middle pane displays a table of 10 rows of data. The bottom pane shows a SQL query editor with a query that selects customer information and counts distinct rental IDs. The query is:

```
SELECT customer.first_name, last_name, COUNT(DISTINCT rental_id) FROM rental LEFT JOIN customer ON customer.customer_id = rental.customer_id GROUP BY last_name, first_name ORDER BY COUNT(DISTINCT rental_id) DESC
```

rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
1	2005-05-24 22:53:30	367	130	2005-05-26 22:04:30	1	2006-02-15 21:30:53
2	2005-05-24 22:54:33	1525	459	2005-05-28 19:40:33	1	2006-02-15 21:30:53
3	2005-05-24 23:03:39	1241	130	2005-06-01 22:13:39	1	2006-02-15 21:30:53
4	2005-05-24 23:04:41	2462	130	2005-06-01 22:13:39	1	2006-02-15 21:30:53
5	2005-05-24 23:05:41	2079	130	2005-06-01 22:13:39	1	2006-02-15 21:30:53
6	2005-05-24 23:06:41	1592	130	2005-06-01 22:13:39	1	2006-02-15 21:30:53
7	2005-05-24 23:07:41	1595	130	2005-06-01 22:13:39	1	2006-02-15 21:30:53
8	2005-05-24 23:31:46	2346	239	2005-05-27 23:33:46	2	2006-02-15 21:30:53
9	2005-05-25 00:00:40	2580	126	2005-05-26 00:22:40	1	2006-02-15 21:30:53
10	2005-05-25 00:02:21	1824	399	2005-05-31 22:44:21	2	2006-02-15 21:30:53

CREATING THE DATABASE

- 1 In MySQL Workbench, click **File** from the top menu, then select **Open SQL Script**
- 2 Navigate to the ***create_mavenmovies.sql*** file provided in the course resources
 - *This code will automatically generate the entire database that we'll be exploring throughout the course, modeling a real-world DVD rental business*
- 3 Click anywhere in the **SQL Query Editor window** (*without highlighting any code*), and click the **lightning bolt icon** to run all of the code and create the database
- 4 After running the code, confirm the following:
 1. You see a list of results in the **Action Output** window, with **green check marks** and no errors in the **Response** column
 2. When you refresh the **Schemas** list, you should see a new database called **mavenmovies**, containing **16 tables**

CREATING THE DATABASE

1 MySQL Workbench File menu: Open SQL Script...

2 File explorer showing the downloaded file: create_mavenmovies.sql (3.4 MB, SQL File)

3 MySQL Workbench: Open the SQL script in a new tab (create_db.x).

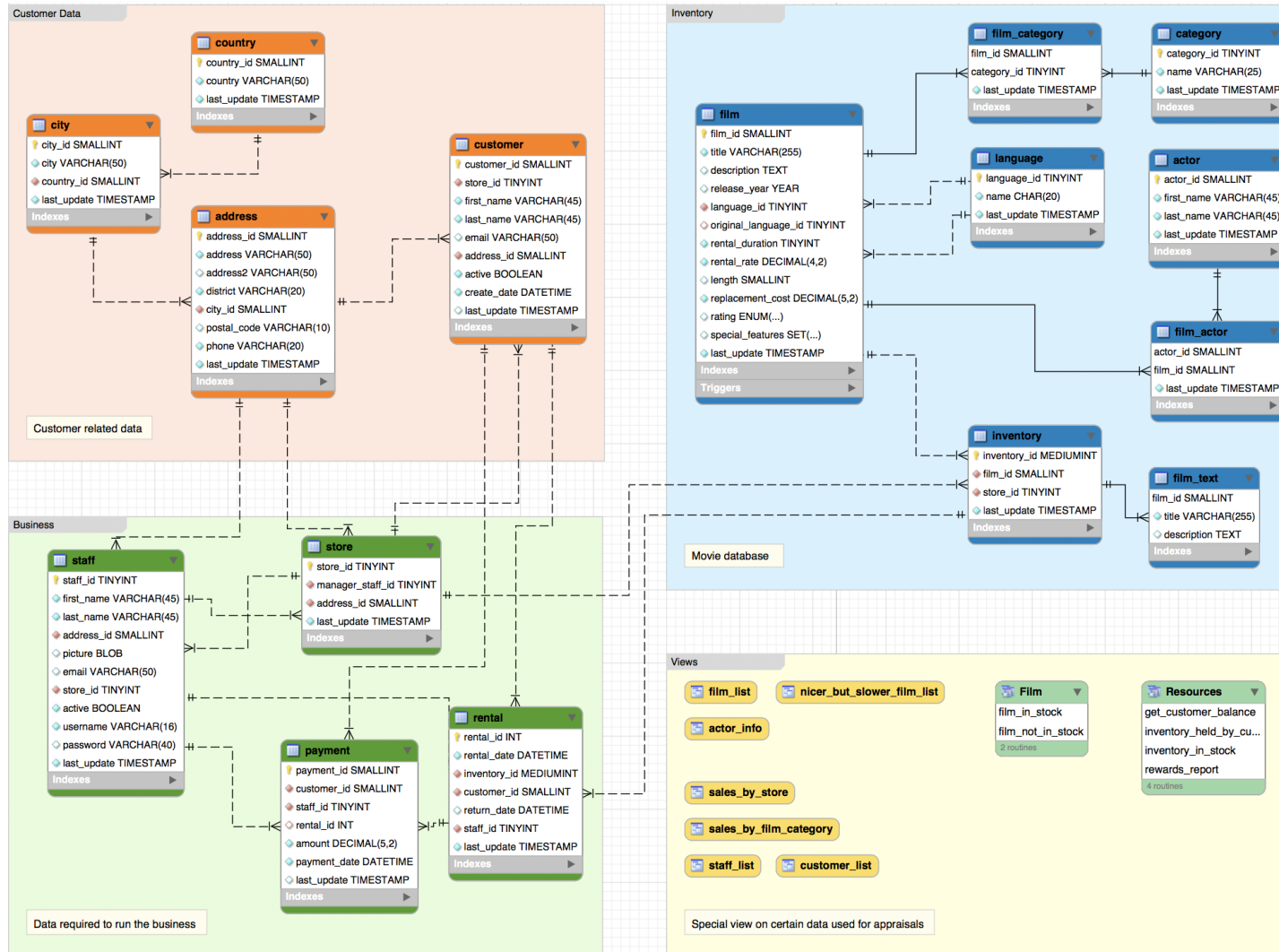
4 MySQL Workbench: Execute the SQL script. The output shows the successful creation of the mavenmovies database and tables.

```
-- Sakila Sample Database Schema
-- Version 1.0
-- Copyright (c) 2006, 2015, Oracle and/or its affiliates.
-- All rights reserved.
-- Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
-- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
-- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
-- * Neither the name of Oracle nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
-- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, USE mavenmovies;
-- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message
CREATE SCHEMA mavenmovies;
-- Table structure for table 'actor'
CREATE TABLE actor (
  actor_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  first_name VARCHAR(45) NOT NULL,
  last_name VARCHAR(45) NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (actor_id),
  KEY idx_actor_last_name (last_name)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
-- Table structure for table 'address'
CREATE TABLE address (
  address_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  address VARCHAR(45) NOT NULL,
  address2 VARCHAR(45) NOT NULL,
  district VARCHAR(20) NOT NULL,
  city VARCHAR(20) NOT NULL,
  state_province VARCHAR(20) NOT NULL,
  country VARCHAR(20) NOT NULL,
  PRIMARY KEY (address_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

Action	Time	Action	Response	Duration / Fetch Time
98	19:05:35	SET AUTOCOMMIT=0	0 row(s) affected	0.00013 sec
99	19:05:35	LOCK TABLES 'staff' WRITE	0 row(s) affected	0.0015 sec
100	19:05:35	INSERT INTO 'staff' VALUES (1,'Mike','Hillyer',3,0x89504E470D0A1A0A000000004948445200000000)	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0	0.0022 sec
101	19:05:35	UNLOCK TABLES	0 row(s) affected	0.0016 sec
102	19:05:35	COMMIT	0 row(s) affected	0.00011 sec
103	19:05:35	SET AUTOCOMMIT=0	0 row(s) affected	0.00012 sec
104	19:05:35	INSERT INTO store VALUES (1,1,2006-02-15 04:57:12), (2,2,2006-02-15 04:57:12)	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0	0.00024 sec

DATABASE FUNDAMENTALS (PART 1)

A DATABASE CAN CONTAIN MANY RELATED TABLES

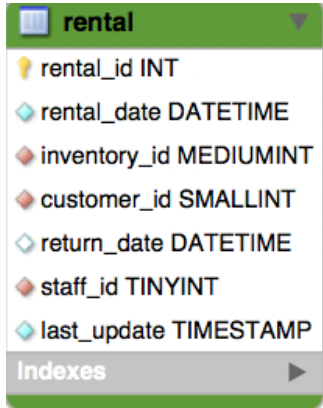


In this case our database includes **16 related tables**, containing information about:

- **Customers** (Name, Address, etc.)
- **Business** (Staff, Rentals, etc.)
- **Inventory** (Films, Categories, etc.)

We'll start by using MySQL to explore **individual tables**, then discuss table relationships and multi-table joins later in the course

EACH TABLE CONTAINS ROWS & COLUMNS

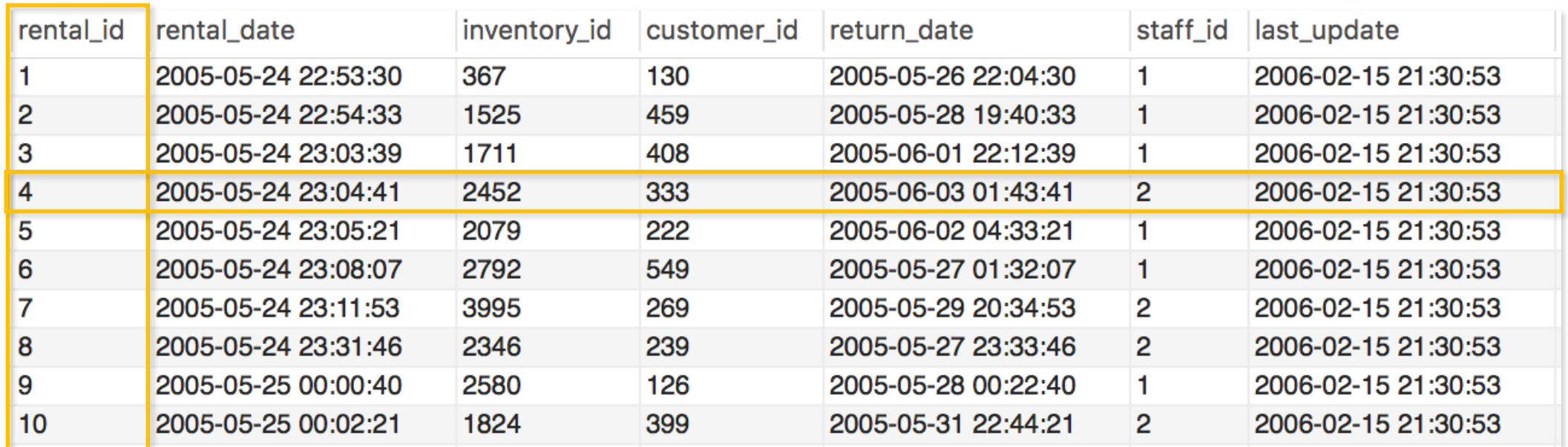


rental						
rental_id	INT					
rental_date	DATETIME					
inventory_id	MEDIUMINT					
customer_id	SMALLINT					
return_date	DATETIME					
staff_id	TINYINT					
last_update	TIMESTAMP					
Indexes						

Tables contain information organized into **columns** (*or fields*) and **rows** (*or records*)

In this case, our **Rental** table contains **7 columns** and **10 rows**:

- Each **column** contains an attribute related to our film rentals (*rental/return date, customer ID, etc.*)
- Each **row** corresponds to **one specific rental** (*which film was rented, when, who rented it, etc.*)



rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
1	2005-05-24 22:53:30	367	130	2005-05-26 22:04:30	1	2006-02-15 21:30:53
2	2005-05-24 22:54:33	1525	459	2005-05-28 19:40:33	1	2006-02-15 21:30:53
3	2005-05-24 23:03:39	1711	408	2005-06-01 22:12:39	1	2006-02-15 21:30:53
4	2005-05-24 23:04:41	2452	333	2005-06-03 01:43:41	2	2006-02-15 21:30:53
5	2005-05-24 23:05:21	2079	222	2005-06-02 04:33:21	1	2006-02-15 21:30:53
6	2005-05-24 23:08:07	2792	549	2005-05-27 01:32:07	1	2006-02-15 21:30:53
7	2005-05-24 23:11:53	3995	269	2005-05-29 20:34:53	2	2006-02-15 21:30:53
8	2005-05-24 23:31:46	2346	239	2005-05-27 23:33:46	2	2006-02-15 21:30:53
9	2005-05-25 00:00:40	2580	126	2005-05-28 00:22:40	1	2006-02-15 21:30:53
10	2005-05-25 00:02:21	1824	399	2005-05-31 22:44:21	2	2006-02-15 21:30:53

TABLES CAN CONTAIN PRIMARY & FOREIGN KEYS

The ***rental_id*** column is known as a **Primary Key**, which serves as a **unique identifier** for each record in the rental table

rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
1	2005-05-24 22:53:30	367	130	2005-05-26 22:04:30	1	2006-02-15 21:30:53
2	2005-05-24 22:54:33	1525	459	2005-05-28 19:40:33	1	2006-02-15 21:30:53
3	2005-05-24 23:03:39	1711	408	2005-06-01 22:12:39	1	2006-02-15 21:30:53
4	2005-05-24 23:04:41	2452	333	2005-06-03 01:43:41	2	2006-02-15 21:30:53
5	2005-05-24 23:05:21	2079	222	2005-06-02 04:33:21	1	2006-02-15 21:30:53
6	2005-05-24 23:08:07	2792	549	2005-05-27 01:32:07	1	2006-02-15 21:30:53
7	2005-05-24 23:11:53	3995	269	2005-05-29 20:34:53	2	2006-02-15 21:30:53
8	2005-05-24 23:31:46	2346	239	2005-05-27 23:33:46	2	2006-02-15 21:30:53
9	2005-05-25 00:00:40	2580	126	2005-05-28 00:22:40	1	2006-02-15 21:30:53
10	2005-05-25 00:02:21	1824	399	2005-05-31 22:44:21	2	2006-02-15 21:30:53

inventory_id, ***customer_id***, and ***staff_id*** are **Foreign Keys**, which **reference primary keys** from other tables



HEY THIS IS IMPORTANT!

Primary Keys **cannot** contain duplicates, and **cannot** be NULL; Foreign Keys **can** repeat, and **can** be NULL

USE MAVENMOVIES



HEY THIS IS IMPORTANT!

The USE statement identifies the schema you will be selecting data from in Workbench.

Example: *USE mavenmovies;*

If you encounter an error that says 'no database selected, you'll need to select your database with a USE statement

ANALYZING SINGLE TABLES

THE “BIG 6” ELEMENTS OF A SQL SELECT STATEMENT

START OF
STATEMENT

SELECT

Identifies the column(s) you want your query to select for your results

SELECT columnName

FROM

Identifies the table(s) your query will pull data from

FROM tableName

WHERE

(Optional) Specifies record-filtering criteria for filtering your results

WHERE logicalCondition

GROUP BY

(Optional) Specifies how to group the data in your results

GROUP BY columnName

HAVING

(Optional) Specifies group-filtering criteria for filtering your results

HAVING logicalCondition

ORDER BY

(Optional) Specifies the order in which your query results are displayed

ORDER BY columnName

END OF
STATEMENT

THE **SELECT** STATEMENT

SELECT

Identifies the column(s) you want your query to select for your results

SELECT columnName, otherColumnName

This lets the SQL server know you are about to specify the column(s) to select for your query result

This list is what SQL will select into your query result. You can list a single column, or specify multiple columns separated by a comma.



HEY THIS IS IMPORTANT!

SELECT is one of the two things you will need to include in every SQL query you write (**FROM** is the other)



PRO TIP:

SELECT tells the SQL server to retrieve **all rows** for the columns specified. Don't worry about filtering for now (we'll do that later by adding a **WHERE** clause)

THE FROM CLAUSE

FROM

Identifies the table(s) your query will select data from

FROM tableName

This lets the SQL server know you are about to specify the table(s) to select from for your query result

This is the table SQL will select from



HEY THIS IS IMPORTANT!

FROM is one of the two things you will need to include in every SQL query you write (**SELECT** is the other)



PRO TIP:

*Unlike specifying multiple columns in your **SELECT**, you cannot just list multiple tables separated by commas in your **FROM**. To use multiple tables, you'll need to use a **JOIN** (we'll cover how to do this later).*

SELECT * FROM

- **SELECT * FROM** tells SQL to select *all* columns from the specified table
- Running **SELECT * FROM {table}** without using a WHERE clause will return the entire table (*all columns, all rows*)

PRO TIP:

SELECT * FROM is a great way to quickly see what data a table contains

MySQL QUERY IN ACTION:

```
SELECT *  
FROM rental
```

QUERY RESULTS:

Result Grid		Filter Rows:	Search	Edit:	Export/Import:	
rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
1	2005-05-24 22:53:30	367	130	2005-05-26 22:04:30	1	2006-02-15 21:30:53
2	2005-05-24 22:54:33	1525	459	2005-05-28 19:40:33	1	2006-02-15 21:30:53
3	2005-05-24 23:03:39	1711	408	2005-06-01 22:12:39	1	2006-02-15 21:30:53
4	2005-05-24 23:04:41	2452	333	2005-06-03 01:43:41	2	2006-02-15 21:30:53
5	2005-05-24 23:05:21	2079	222	2005-06-02 04:33:21	1	2006-02-15 21:30:53
6	2005-05-24 23:08:07	2792	549	2005-05-27 01:32:07	1	2006-02-15 21:30:53

Response	Duration / Fetch Time
16044 row(s) returned	0.012 sec / 0.013 sec

SELECTING COLUMNS

- You can return one or more specific columns in your results by naming them in your **SELECT** statement
- To select data from multiple columns, separate them with commas

PRO TIP:

Use line breaks and indentation to make your code more human-readable.

MySQL QUERY IN ACTION:

```
SELECT
  customer_id,
  rental_date
FROM rental;
```

QUERY RESULTS:

Result Grid		Filter Rows
customer_id	rental_date	
130	2005-05-24 22:53:30	
459	2005-05-24 22:54:33	
408	2005-05-24 23:03:39	
333	2005-05-24 23:04:41	
222	2005-05-24 23:05:21	
549	2005-05-24 23:08:07	
269	2005-05-24 23:11:53	
239	2005-05-24 23:31:46	

ERROR MESSAGE

Error Code: 1064. You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'FROM rental' at line 4

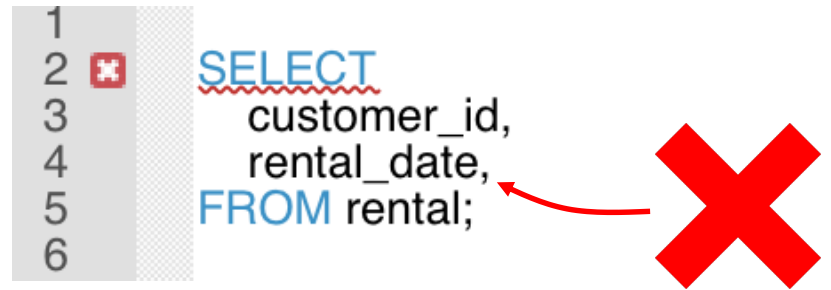
WHAT IT MEANS

Error Code 1064 means you're not following SQL syntax. The most common culprits are an extra comma after your final column of your SELECT statement, or a missing comma between two columns.

HOW TO DEBUG LIKE A PRO

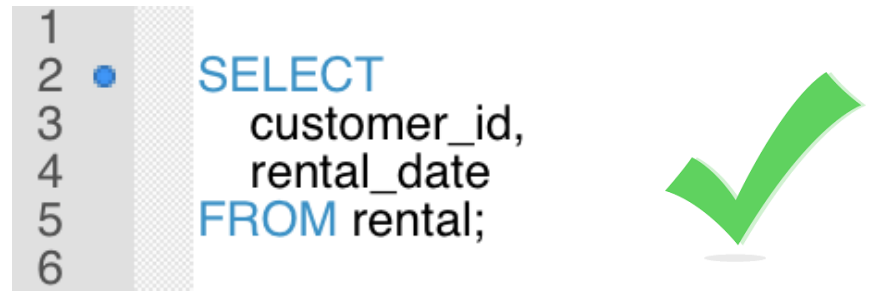
Look for the line number at the end of your Error Response, in this case line 4. The Response also tells you where your syntax error is near, in this case 'FROM rental'. Find the row, and that part of your code. These are your best clues.

ERROR TYPE MISSING/EXTRA COMMA



```
1  
2  
3  
4  
5  
6
```

`SELECT
customer_id,
rental_date,
FROM rental;`



```
1  
2  
3  
4  
5  
6
```



`SELECT
customer_id,
rental_date
FROM rental;`

YOUR ASSIGNMENT:

“I’m going to send an email letting our customers know there has been a management change.

*Could you pull a list of the **first name**, **last name**, and **email** of each of our customers?”*

Result Preview

Result Grid			Filter Rows:	<input type="text" value="Search"/>	Expo
	first_name	last_name	email		
▶	MARY	SMITH	MARY.SMITH@sakilacustomer.org		
	PATRICIA	JOHNSON	PATRICIA.JOHNSON@sakilacustomer.org		
	LINDA	WILLIAMS	LINDA.WILLIAMS@sakilacustomer.org		
	BARBARA	JONES	BARBARA.JONES@sakilacustomer.org		
	ELIZABETH	BROWN	ELIZABETH.BROWN@sakilacustomer.org		
	JENNIFER	DAVIS	JENNIFER.DAVIS@sakilacustomer.org		
	MARIA	MILLER	MARIA.MILLER@sakilacustomer.org		
	SUSAN	WILSON	SUSAN.WILSON@sakilacustomer.org		
	MARGARET	MOORE	MARGARET.MOORE@sakilacustomer.org		
	DOROTHY	TAYLOR	DOROTHY.TAYLOR@sakilacustomer.org		
	LISA	ANDERS...	LISA.ANDERSON@sakilacustomer.org		
	NANCY	THOMAS	NANCY.THOMAS@sakilacustomer.org		
	KAREN	JACKSON	KAREN.JACKSON@sakilacustomer.org		
	BETTY	WHITE	BETTY.WHITE@sakilacustomer.org		
	HELEN	HARRIS	HELEN.HARRIS@sakilacustomer.org		
	SANDRA	MARTIN	SANDRA.MARTIN@sakilacustomer.org		

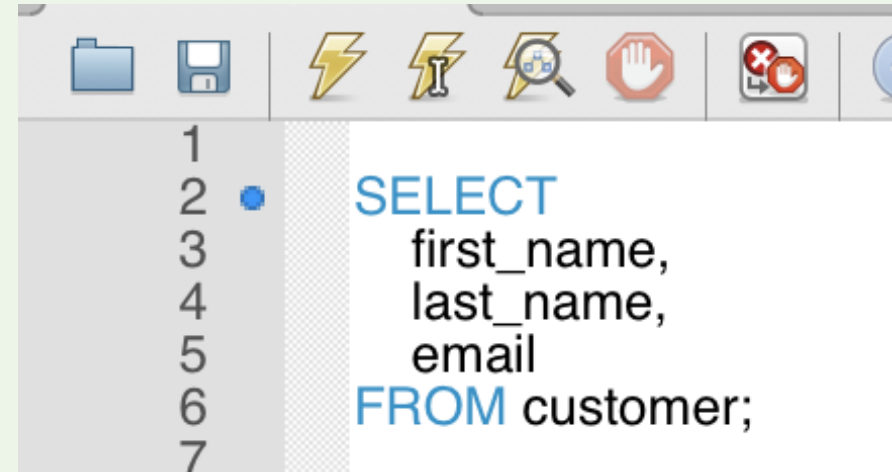
TEST YOUR SKILLS: QUERYING TABLES

YOUR ASSIGNMENT:

"I'm going to send an email letting our customers know there has been a management change.

*Could you pull a list of the **first name**, **last name**, and **email** of each of our customers?"*

Solution Query



The screenshot shows a SQL query editor window. The toolbar at the top includes icons for file operations (folder, save), execution (lightning bolt), search (magnifying glass), and other functions. On the left, a list of line numbers from 1 to 7 is visible, with line 2 selected. The query text in the editor is:

```
SELECT  
  first_name,  
  last_name,  
  email  
FROM customer;
```

TEST YOUR SKILLS: QUERYING TABLES

SELECT DISTINCT

- When you use **SELECT DISTINCT**, your result set will return just the **distinct (or unique)** values in those columns
- If you include multiple columns, your result set will return all **distinct combinations** of values across those columns



PRO TIP:

SELECT DISTINCT is a great way to quickly see all values in a column

MySQL QUERY IN ACTION:

```
SELECT DISTINCT  
rating  
FROM film
```

QUERY RESULTS:

Result Grid


rating	
PG	
G	
NC-17	
PG-13	
R	

YOUR ASSIGNMENT:

*“My understanding is that we have titles that we rent for durations of **3, 5, or 7** days.*

*Could you pull the records of our films and see if there are any other **rental durations**?”*

Result Preview

Result Grid		
	rental_duration	
▶	6	
	3	
	7	
	5	
	4	

TEST YOUR SKILLS: SELECT DISTINCT

YOUR ASSIGNMENT:

*“My understanding is that we have titles that we rent for durations of **3, 5, or 7** days.*

*Could you pull the records of our films and see if there are any other **rental durations**?”*

Solution Query

```
SELECT DISTINCT  
    rental_duration  
FROM film
```

TEST YOUR SKILLS: SELECT DISTINCT

THE WHERE CLAUSE

WHERE

(Optional) Specifies criteria for filtering the records of your result set

WHERE logicalCondition

This lets the SQL server know you are about to specify a logical condition for which rows to include

This is where you prescribe the logical conditions used to filter your result set

Examples:

- **WHERE** category = 'Sci-Fi'
- **WHERE** amount > 5.99
- **WHERE** rental_date BETWEEN '2006-01-01' AND '2006-06-01'



HEY THIS IS IMPORTANT!

WHERE is an **optional clause**, and always comes *after* your FROM clause and *before* any GROUP BY, HAVING, or ORDER BY clauses (if included)



PRO TIP:

*Specify multiple filter criteria using **AND/OR statements** within your WHERE clause*

WHERE OPERATORS

- **WHERE** clauses can filter records using any of these logical operators:

Operator	What it Means
=	Equals
<>	Does NOT Equal
>	Greater Than
<	Less Than
>=	Greater Than Or Equal To
<=	Less Than Or Equal To
BETWEEN	A Range Between Two Values
LIKE	Matching a Pattern Like This
IN()	Equals One of These Values

MySQL QUERY IN ACTION:

```
SELECT
  customer_id,
  rental_id,
  amount,
  payment_date
FROM payment
WHERE amount = 0.99
```

QUERY RESULTS:

Result Grid		Filter Rows:	Search
customer_id	rental_id	amount	payment_date
1	1422	0.99	2005-06-15 16:02:55
1	2363	0.99	2005-06-18 13:33:59
1	8074	0.99	2005-07-28 17:33:39
1	8116	0.99	2005-07-28 19:20:07
1	11367	0.99	2005-08-02 18:01:38
1	12250	0.99	2005-08-18 03:57:29
1	13068	0.99	2005-08-19 09:55:16
1	14762	0.99	2005-08-21 23:33:57
2	9248	0.99	2005-07-30 14:14:11

WHERE EXAMPLES

1. Pulling records before or after a date, or between two dates
2. Limiting your result set to include only transactions above a certain payment amount
3. Filtering your result set to just records related to certain customers, or to certain lines of business

MySQL QUERY IN ACTION:

```
SELECT
customer_id,
rental_id,
amount,
payment_date
FROM payment
WHERE payment_date > '2006-01-01'
```

QUERY RESULTS:

Result Grid		Filter Rows: <input type="text" value="Search"/>		
	customer_id	rental_id	amount	payment_date
	5	13209	0.99	2006-02-14 15:16:03
	9	15813	4.99	2006-02-14 15:16:03
	11	11646	0.99	2006-02-14 15:16:03
	14	13780	4.99	2006-02-14 15:16:03
	15	13798	3.98	2006-02-14 15:16:03
	15	13968	0.00	2006-02-14 15:16:03
	21	14933	2.99	2006-02-14 15:16:03
	22	12222	4.99	2006-02-14 15:16:03

ERROR MESSAGE

Error Code: 1064. You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'FROM payment' at line 7

WHAT IT MEANS

Again, the Error Code 1064 means you're not following SQL syntax. In this case, we've tried to apply the WHERE clause before the FROM, forgetting that WHERE always comes after FROM.

HOW TO DUBUG LIKE A PRO

Look for the line number at the end of your Error Response, in this case line 7. The Response also tells you where your syntax error is near, in this case 'FROM payment'. Find the row, and that part of your code. These are your best clues.

ERROR TYPE WHERE BEFORE FROM

```
1 SELECT
2   customer_id,
3   rental_id,
4   amount,
5   payment_date
6   WHERE payment_date > '2006-01-01'
7 FROM payment
8
```



```
1 SELECT
2   customer_id,
3   rental_id,
4   amount,
5   payment_date
6 FROM payment
7 WHERE payment_date > '2006-01-01'
8
```





YOUR ASSIGNMENT:

*“I’d like to look at **payment records** for our **long-term customers** to learn about their purchase patterns.*

*Could you pull all **payments** from our **first 100 customers** (based on customer ID)?”*

Result Preview

Result Grid			Filter Rows:	<input type="text" value="Search"/>
customer_id	rental_id	amount	payment_date	
1	76	2.99	2005-05-25 11:30:37	
1	573	0.99	2005-05-28 10:35:23	
1	1185	5.99	2005-06-15 00:54:12	
1	1422	0.99	2005-06-15 18:02:53	
1	1476	9.99	2005-06-15 21:08:46	
1	1725	4.99	2005-06-16 15:18:57	
1	2308	4.99	2005-06-18 08:41:48	
1	2363	0.99	2005-06-18 13:33:59	
1	3284	3.99	2005-06-21 06:24:45	
1	4526	5.99	2005-07-08 03:17:05	
1	4611	5.99	2005-07-08 07:33:56	

TEST YOUR SKILLS: WHERE CLAUSES

YOUR ASSIGNMENT:

*“I’d like to look at **payment records** for our **long-term customers** to learn about their purchase patterns.*

*Could you pull all **payments** from our **first 100 customers** (based on customer ID)?”*

Solution Query

```
SELECT
    customer_id,
    rental_id,
    amount,
    payment_date
FROM payment
WHERE customer_id < 101;
```

ALTERNATIVE OPTIONS:

```
WHERE customer_id <= 100
```

```
WHERE customer_id BETWEEN 1 AND 100
```

TEST YOUR SKILLS: WHERE CLAUSES

WHERE & AND

- You can include **multiple logical conditions** in your **WHERE** clause using an **AND** statement
- Use **AND** to return records which satisfy *all* criteria

MySQL QUERY IN ACTION:

```
SELECT
customer_id,
rental_id,
amount,
payment_date
FROM payment
WHERE amount = 0.99
AND payment_date > '2006-01-01'
```

QUERY RESULTS:

Result Grid		Filter Rows:		Search
	customer_id	rental_id	amount	payment_date
5	13209	0.99	2006-02-14 15:16:03	
11	11646	0.99	2006-02-14 15:16:03	
29	15577	0.99	2006-02-14 15:16:03	
33	12277	0.99	2006-02-14 15:16:03	
58	15326	0.99	2006-02-14 15:16:03	
69	11995	0.99	2006-02-14 15:16:03	
99	11593	0.99	2006-02-14 15:16:03	
100	15021	0.99	2006-02-14 15:16:03	

PRO TIP:

Add and remove filtering logic and compare results to quickly master this

ERROR MESSAGE

Error Code: 1064. You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near "2006-01" at line 9

WHAT IT MEANS


Error Code 1064 means you're not following SQL syntax. In this case, a very common example, we have forgotten to put a quotation mark after the date.

HOW TO DUBUG LIKE A PRO


Look for the line number at the end of your Error Response, in this case line 9. The Response also tells you where your syntax error is near, in this case the date. Find the row, and that part of your code. These are your best clues.

ERROR TYPE MISSING QUOTATION MARK

```
1 ❌ SELECT
2     customer_id,
3     rental_id,
4     amount,
5     payment_date
6 FROM payment
7
8 WHERE amount = 0.99
9 ❌ AND payment_date > '2006-01-01'
```



```
1 • SELECT
2     customer_id,
3     rental_id,
4     amount,
5     payment_date
6 FROM payment
7
8 WHERE amount = 0.99
9     AND payment_date > '2006-01-01'
```



YOUR ASSIGNMENT:

“The payment data you gave me on our first 100 customers was great – thank you!

*Now I’d love to see **just payments over \$5** for those same customers, since January 1, 2006.”*

Result Preview

Result Grid				Filter Rows:	Search
customer_id	rental_id	amount	payment_date		
42	13351	5.98	2006-02-14 15:16:03		
53	11657	7.98	2006-02-14 15:16:03		
60	12489	9.98	2006-02-14 15:16:03		
75	13534	8.97	2006-02-14 15:16:03		

TEST YOUR SKILLS: WHERE & AND

YOUR ASSIGNMENT:

“The payment data you gave me on our first 100 customers was great – thank you!

*Now I’d love to see **just payments over \$5** for those **same customers, since January 1, 2006.**”*

Solution Query

```
SELECT
  customer_id,
  rental_id,
  amount,
  payment_date
FROM payment
WHERE customer_id < 101
  AND amount > 5
  AND payment_date > '2006-01-01' ;
```

ALTERNATIVE OPTIONS:

*You can add the three filters in any order without changing the outcome, and can define the **customer_id** filter in a number of different ways.*

As long as you see the 4 records shown on the previous slide, you’re good to go!

TEST YOUR SKILLS: WHERE & AND

WHERE & OR

- You can include **multiple logical conditions** in your **WHERE** clause using an **OR** statement
- Use **OR** to return records which satisfy *any* criteria

MySQL QUERY IN ACTION:

```
SELECT
    customer_id,
    rental_id,
    amount,
    payment_date
FROM payment
WHERE customer_id = 5
    OR customer_id = 11
    OR customer_id = 29;
```

QUERY RESULTS:

Result Grid		Filter Rows:		Search
customer_id	rental_id	amount	payment_date	
5	731	0.99	2005-05-29 07:25:16	
5	1085	6.99	2005-05-31 11:15:43	
5	1142	1.99	2005-05-31 19:46:38	
5	1502	3.99	2005-06-15 22:03:14	
5	1631	2.99	2005-06-16 08:01:02	
5	2063	4.99	2005-06-17 15:56:53	
5	2570	2.99	2005-06-19 04:20:13	

PRO TIP:



You can use **AND** and **OR** together in the same **WHERE** clause. This is powerful!

YOUR ASSIGNMENT:

“The data you shared previously on customers 42, 53, 60, and 75 was good to see.

*Now, could you please write a query to pull **all payments from those specific customers**, along with **payments over \$5**, from any customer?”*

Result Preview

Result Grid				Filter Rows:	Search
customer_id	rental_id	amount	payment_date		
1	1185	5.99	2005-06-15 00:54:12		
1	1476	9.99	2005-06-15 21:08:46		
1	4526	5.99	2005-07-08 03:17:05		
1	4611	5.99	2005-07-08 07:33:56		
1	6163	7.99	2005-07-11 10:13:46		
1	15315	5.99	2005-08-22 20:03:46		
2	5755	6.99	2005-07-10 12:38:56		
2	7376	5.99	2005-07-27 15:23:02		
2	7459	5.99	2005-07-27 18:40:20		
2	8230	5.99	2005-07-29 00:12:59		

TEST YOUR SKILLS: WHERE & OR

YOUR ASSIGNMENT:

“The data you shared previously on customers 42, 53, 60, and 75 was good to see.

*Now, could you please write a query to pull **all payments from those specific customers**, along with **payments over \$5**, from any customer?”*

Solution Query

```
SELECT
    customer_id,
    rental_id,
    amount,
    payment_date
FROM payment
WHERE amount > 5
    OR customer_id = 42
    OR customer_id = 53
    OR customer_id = 60
    OR customer_id = 75
```

ALTERNATIVE OPTIONS:

The logic in this WHERE can be written in any order.

TEST YOUR SKILLS: WHERE & OR

WHERE & IN

- If you find yourself writing multiple **OR** conditions that reference different values in the *same column*, you can use **IN()** to save some time
- The two queries shown at the right produce identical results, but the second version is easier to read (*and write!*)



PRO TIP:

Using **IN** saves time, and will make it easier to read your query later

MySQL QUERY IN ACTION:

```
SELECT
  customer_id,
  rental_id,
  amount,
  payment_date
FROM payment
WHERE amount > 5
OR customer_id = 42
OR customer_id = 53
OR customer_id = 60
OR customer_id = 75
```

These two queries do the exact same thing

```
SELECT
  customer_id,
  rental_id,
  amount,
  payment_date
FROM payment
WHERE amount > 5
OR customer_id IN (42,53,60,75)
```

ERROR MESSAGE

*Error Code: 1054. Unknown column
'cusotmer_id' in 'where clause'*

WHAT IT MEANS

Error Code 1054 means you're telling the SQL server to use a column that it cannot find. The most common reasons:

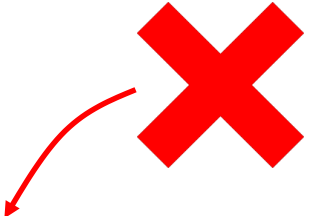
- You've misspelled the column name*
- That column is not in this table*

HOW TO DUBUG LIKE A PRO


Read the column name provided in the Response. If it looks like a misspelling, find the misspelling in your code and fix it. If the column appears spelled correctly, check to make sure that column exists in the table you are using.

ERROR TYPE UNKNOWN COLUMN

```
SELECT
  customer_id,
  rental_id,
  amount,
  payment_date
FROM payment
WHERE cusotmer_id IN(5,11,29);
```



```
SELECT
  customer_id,
  rental_id,
  amount,
  payment_date
FROM payment
WHERE customer_id IN(5,11,29);
```



THE LIKE OPERATOR

LIKE

Allows you to use **pattern** matching in your logical operators (instead of **exact** matching)

LIKE '%patternToLookFor%'

LIKE tells SQL that you're prescribing a **pattern** instead of an exact number or string of characters

This is where you define the pattern. The "%" before and after the text is a type of **wildcard** (along with "_")

Examples :

- **WHERE** name **LIKE** 'Denise%' -- records where name starts with 'Denise', with **any number of characters after**
- **WHERE** description **LIKE** '%fancy%' -- records that contains 'fancy', with **any characters before OR after**
- **WHERE** name **LIKE** '%Johnson' -- records that end with 'Johnson', with **any number of characters before**
- **WHERE** first_name **LIKE** '_erry' -- records that end with 'erry', with **exactly one character before** (i.e. Terry, Jerry)



HEY THIS IS IMPORTANT!

Capitalization matters. When you specify a value to look for, the server will consider the capitalization you provided in your SQL statement.



PRO TIP:

NOT LIKE can also be used to **filter out** (rather than keep) records where values match a pattern you provide, and uses the same wildcards and capitalization rules as **LIKE**

WILDCARD EXAMPLES

Syntax	How SQL Evaluates Logic
column = 'value'	TRUE if the value in the column matches your value exactly (<i>nothing before or after</i>)
column LIKE 'pattern%'	TRUE if the first characters in the column match your pattern, even if they are followed by others
column LIKE '%pattern'	TRUE if the last characters in the column match your pattern, even if they are preceded by others
column LIKE '%pattern%'	TRUE if any characters in the column match your pattern, even if there are extra characters before or after
column LIKE 'pattern_'	TRUE if the first characters in the column match your pattern, even if they are followed by exactly 1 other character
column LIKE '_pattern'	TRUE if the last characters in the column match your pattern, even if they are preceded by exactly 1 other character
column LIKE '_pattern_'	TRUE if any characters in the column match your pattern, even if there is exactly 1 character before or after

MySQL QUERY IN ACTION:

```
SELECT
    title,
    description
FROM film
WHERE description LIKE '%Dentist%'
```

QUERY RESULTS:

Result Grid	Filter Rows:	Search	Export:
title	description		
AFRICAN EGG	A Fast-Paced Documentary of a Pastry Chef And a Dentist who must Pursue a Forensic Psychologist in The Gulf of Mexico		
ALI FOREVER	A Action-Packed Drama of a Dentist And a Crocodile who must Battle a Feminist in The Canadian Rockies		
AMISTAD MIDSUMMER	A Emotional Character Study of a Dentist And a Crocodile who must Meet a Sumo Wrestler in California		
ANACONDA CONFESSIONS	A Lacklustre Display of a Dentist And a Dentist who must Fight a Girl in Australia		
ARSENIC INDEPENDENCE	A Fanciful Documentary of a Mad Cow And a Womanizer who must Find a Dentist in Berlin		
AUTUMN CROW	A Beautiful Tale of a Dentist And a Mad Cow who must Battle a Moose in The Sahara Desert		
BLADE POLISH	A Thoughtful Character Study of a Frisbee And a Pastry Chef who must Fight a Dentist in The First Manned Space Station		
BLINDNESS GUN	A Touching Drama of a Robot And a Dentist who must Meet a Hunter in A Jet Boat		
BOILED DARES	A Awe-Inspiring Story of a Waitress And a Dog who must Discover a Dentist in Ancient Japan		
BOWFINGER GABLES	A Fast-Paced Yarn of a Waitress And a Composer who must Outgun a Dentist in California		
BROOKLYN DESERT	A Beautiful Drama of a Dentist And a Composer who must Battle a Sumo Wrestler in The First Manned Space Station		
CENTER DINOSAUR	A Beautiful Character Study of a Sumo Wrestler And a Dentist who must Find a Dog in California		
CHOCOLAT HARRY	A Action-Packed Epistle of a Dentist And a Moose who must Meet a Mad Cow in Ancient Japan		
CIRCUS YOUTH	A Thoughtful Drama of a Pastry Chef And a Dentist who must Pursue a Girl in A Baloon		
CLONES PINOCCHIO	A Amazing Drama of a Car And a Robot who must Pursue a Dentist in New Orleans		
COLDBLOODED DARLING	A Brilliant Panorama of a Dentist And a Moose who must Find a Student in The Gulf of Mexico		
CONNECTICUT TRAMP	A Unbelievable Drama of a Crocodile And a Mad Cow who must Reach a Dentist in A Shark Tank		
CRUELTY UNFORGIVEN	A Brilliant Tale of a Car And a Moose who must Battle a Dentist in Nigeria		
CURTAIN VIDEOTAPE	A Boring Reflection of a Dentist And a Mad Cow who must Chase a Secret Agent in A Shark Tank		
DARES PLUTO	A Fateful Story of a Robot And a Dentist who must Defeat a Astronaut in New Orleans		
DATE SPEED	A Touching Saga of a Composer And a Moose who must Discover a Dentist in A MySQL Convention		

YOUR ASSIGNMENT:

*“We need to understand the **special features** in our films. Could you pull a list of films which include a **Behind the Scenes** special feature?”*

Result Preview

Result Grid



Filter Rows:

Search

Export:



title	special_features
ACADEMY DINOSAUR	Deleted Scenes,Behind the Scenes
AFFAIR PREJUDICE	Commentaries,Behind the Scenes
ALAMO VIDEOTAPE	Commentaries,Behind the Scenes
ALI FOREVER	Deleted Scenes,Behind the Scenes
ALICE FANTASIA	Trailers,Deleted Scenes,Behind the Scenes
ALIEN CENTER	Trailers,Commentaries,Behind the Scenes
ALONE TRIP	Trailers,Behind the Scenes
ALTER VICTORY	Trailers,Behind the Scenes
AMADEUS HOLY	Commentaries,Deleted Scenes,Behind the Sce...
AMELIE HELLFIGHTERS	Commentaries,Deleted Scenes,Behind the Sce...
AMERICAN CIRCUS	Commentaries,Behind the Scenes
AMISTAD MIDSUMMER	Commentaries,Behind the Scenes
ANALYZE HOOSIERS	Trailers,Behind the Scenes
ANONYMOUS HUMAN	Deleted Scenes,Behind the Scenes
ANTHEM LUKE	Deleted Scenes,Behind the Scenes
ANYTHING SAVANNAH	Trailers,Deleted Scenes,Behind the Scenes

TEST YOUR SKILLS: LIKE W/ WILDCARDS

YOUR ASSIGNMENT:

*“We need to understand the **special features** in our films. Could you pull a list of films which include a **Behind the Scenes** special feature?”*

Solution Query

```
SELECT
    title,
    special_features
FROM film
WHERE special_features LIKE '%Behind The Scenes%'
```

TEST YOUR SKILLS: LIKE W/ WILDCARDS

THE GROUP BY CLAUSE

GROUP BY

(Optional) Specifies how to group the data in your results

GROUP BY columnName, otherColumnName

This lets the SQL server know you are about to specify how you want your result set to be grouped

This is where you prescribe which column(s) the server will use to group your data. You can use multiple columns for grouping if needed.

Examples:

- **GROUP BY** category
- **GROUP BY** store_location
- **GROUP BY** store_location, category



HEY THIS IS IMPORTANT!

GROUP BY is optional, and comes *after* any WHERE, and *before* any HAVING or ORDER BY clauses in your query



PRO TIP:

*GROUP BY is great for **comparing different segments** of your data (similar to Pivot Tables in Excel)*

GROUP AGGREGATION

- **GROUP BY** allows us to perform segment analysis on our data at various levels of granularity
- Combine **GROUP BY** with aggregate functions (like **COUNT** or **SUM**) to specify how values are summarized for each group

PRO TIP:

Aggregate functions serve the same purpose as **summarization modes** in a Pivot Table

MySQL QUERY IN ACTION:

```
SELECT  
rating,  
COUNT(film_id)  
FROM film  
GROUP BY  
rating
```

QUERY RESULTS:

Result Grid	
rating	COUNT(film_id)
PG	194
G	178
NC-17	210
PG-13	223
R	195

ERROR MESSAGE

Error Code: 1140. In aggregated query without GROUP BY, expression #1 of SELECT list contains nonaggregated column

WHAT IT MEANS

Error Code 1140 means you're using an aggregate function, but also included some non-aggregated column (in this case ratings) and haven't included it in a GROUP BY

HOW TO DUBUG LIKE A PRO

Look at the expression # provided in the error response to quickly identify the column that is causing the error. Add that column to a GROUP BY. Think "Dimensions" vs "Metrics". Make sure all Dimensions are in your GROUP BY.

ERROR TYPE NON-AGGREGATED COLUMN

```
SELECT
  rating,
  COUNT(film_id)
FROM film
```



```
SELECT
  rating,
  COUNT(film_id)
FROM film
GROUP BY
  rating
```



COMMENTS & ALIASES

- **Comments** can be added to your code using “--” or “/*”, which tells SQL to ignore those lines
- Comments can apply to **entire lines, portions of a line, or multiple lines**
- **Aliases** allow you to assign a custom name to a field in your result set, using an **AS** statement

PRO TIP:

*Commenting make queries much more **human readable** -- use them often!*

MySQL QUERY IN ACTION:

These are **comments**.
SQL skips these lines

```
-- SQL single line comment
SELECT
rating,
COUNT(film_id)
COUNT(film_id) AS films_with_this_rating
FROM film
GROUP BY
rating
```

This is an example of an “**alias**”

```
/*
Above starts a multi-line comment.
Below ends a multi-line comment.
*/
```

QUERY RESULTS:

Result Grid		Filter Rows:	Search
rating	COUNT(film_id)	films_with_this_rating	
PG	194	194	
G	178	178	
NC-17	210	210	
PG-13	223	223	
R	195	195	



Assigning an alias does **NOT** change the resulting values

YOUR ASSIGNMENT:

“I need to get a quick overview of how long our movies tend to be rented out for.

*Could you please pull a **count of titles sliced by rental duration?**”*

Result Preview

Result Grid				Filter Rows:	<input type="text" value="Search"/>
rental_duration	films_with_this_rental_duration				
6	212				
3	203				
7	191				
5	191				
4	203				

BONUS: USE AN ALIAS TO MATCH THE COLUMN NAMES

TEST YOUR SKILLS: GROUP BY

YOUR ASSIGNMENT:

“I need to get a quick overview of how long our movies tend to be rented out for.

*Could you please pull a **count of titles sliced by rental duration?**”*

Solution Query

```
SELECT
    rental_duration,
    COUNT(film_id) AS films_with_this_rental_duration
FROM film
GROUP BY
    rental_duration
```

TEST YOUR SKILLS: GROUP BY

MULTIPLE GROUP BY

- You can use **GROUP BY** with *multiple* columns at once, by listing the columns and separating them with a comma
- This allows you to create groups and sub-groups in your result set, (just like specifying *multiple row or column labels in a PivotTable!*)

PRO TIP:

Group by customer segment, and add a time dimension to create **cohort trends**

MySQL QUERY IN ACTION:

```
SELECT
rating,
rental_duration,
COUNT(film_id) AS count_of_films
FROM film
GROUP BY
rating,
rental_duration
```

QUERY RESULTS:

Result Grid		Filter Rows:
rating	rental_duration	count_of_films
PG	6	39
G	3	49
NC-17	7	40
G	5	33
G	6	39
PG	3	36
PG-13	6	50
R	6	27
PG-13	3	39
NC-17	6	57

AGGREGATE FUNCTIONS

- The powerful functions below can all be used with **GROUP BY** to provide group-level summaries

Function	Purpose
COUNT()	Count of Records <i>Skips NULL, except COUNT(*)</i>
COUNT DISTINCT()	Count of Distinct Values <i>Skips NULL values</i>
MIN()	Finds the Smallest Value <i>Skips NULL values</i>
MAX()	Finds the Largest Value <i>Skips NULL values</i>
AVG()	Average of All Values <i>Skips NULL values</i>
SUM()	SUM of All Values <i>Treats NULL values as Zero</i>




MySQL QUERY IN ACTION:

```
SELECT
rating,
COUNT(film_id) AS count_of_films,
MIN(length) AS shortest_film,
MAX(length) AS longest_film,
AVG(length) AS average_length_of_film,
-- SUM(length) AS total_minutes
AVG(rental_duration) AS average_rental_duration

FROM film
GROUP BY
rating
```

Note the application of **aliases** to make the output easier to interpret

QUERY RESULTS:

Result Grid		 	Filter Rows:	<input type="text" value="Search"/>	Export: 
rating	count_of_films	shortest_film	longest_film	average_length_of_film	average_rental_duration
PG	194	46	185	112.0052	5.0825
G	178	47	185	111.0506	4.8371
NC-17	210	46	184	113.2286	5.1429
PG-13	223	46	185	120.4439	5.0538
R	195	49	185	118.6615	4.7744

YOUR ASSIGNMENT:

“I’m wondering if we charge more for a rental when the replacement cost is higher.

*Can you help me pull a **count of films**, along with the **average**, **min**, and **max rental rate**, grouped **by replacement cost**?”*

Result Preview

Result Grid		Filter Rows:	Search	Export:
replacement_cost	number_of_films	cheapest_rental	most_expensive_rental	average_rental
29.99	53	0.99	4.99	2.537170
28.99	41	0.99	4.99	2.990000
27.99	53	0.99	4.99	2.839057
26.99	46	0.99	4.99	2.990000
25.99	43	0.99	4.99	2.850465
24.99	38	0.99	4.99	3.147895
23.99	45	0.99	4.99	2.723333
22.99	55	0.99	4.99	3.062727
21.99	55	0.99	4.99	2.990000
20.99	57	0.99	4.99	2.919825
19.99	50	0.99	4.99	3.110000

BONUS: USE ALIASES TO MATCH THE COLUMN NAMES

TEST YOUR SKILLS: AGGREGATE FUNCTIONS

YOUR ASSIGNMENT:

“I’m wondering if we charge more for a rental when the replacement cost is higher.

*Can you help me pull a **count of films**, along with the **average, min, and max rental rate**, grouped **by replacement cost**?”*

Solution Query

```
SELECT
  replacement_cost,
  COUNT(film_id) AS number_of_films,
  MIN(rental_rate) AS cheapest_rental,
  MAX(rental_rate) AS most_expensive_rental,
  AVG(rental_rate) AS average_rental
FROM film
GROUP BY
  replacement_cost
```

TEST YOUR SKILLS: AGGREGATE FUNCTIONS

THE HAVING CLAUSE

HAVING

(Optional) Specifies group-filtering criteria for filtering your results

HAVING logical condition

This is where you specify the filtering logic that you want applied to your group-level aggregated metrics.

Examples:

- **HAVING COUNT**(*) > 1
- **HAVING SUM**(payment) > 10
- **HAVING MIN**(rental_date) < '2005-05-25'



HEY THIS IS IMPORTANT!

HAVING can only be used with GROUP BY. If you are trying to filter your results, but aren't grouping with **GROUP BY**, then you should use a **WHERE** clause instead.



PRO TIP:

*HAVING is a great way to limit your results to the **most important groups** for your business. For example, you can limit results to customers with total payments above a certain amount, or to your most-rented films.*

HAVING

- **HAVING** is an optional clause you can use with **GROUP BY** to limit your result set to groups which satisfy certain logical criteria
- **HAVING** comes *after* the **GROUP BY** clause and *before* **ORDER BY**, if you are sorting your results (*we'll cover this next*)

MySQL QUERY IN ACTION:

```
SELECT
    customer_id,
    COUNT(*) AS total_rentals
FROM rental
GROUP BY
    customer_id
HAVING COUNT(*) >= 30
```

QUERY RESULTS:



Result Grid		Filter
customer_id	total_rentals	
1	32	
5	38	
7	33	
15	32	
20	30	
21	35	
23	30	
26	34	
27	31	
28	32	
29	36	
30	34	
35	32	
38	34	

YOUR ASSIGNMENT:

“I’d like to talk to customers that have not rented much from us to understand if there is something we could be doing better.

*Could you pull a list of **customer_ids** with **less than 15 rentals** all-time?”*

Result Preview

Result Grid				Filter
	customer_id	total_rentals		
▶	61	14		
	110	14		
	281	14		
	318	12		

TEST YOUR SKILLS: HAVING

YOUR ASSIGNMENT:

“I’d like to talk to customers that have not rented much from us to understand if there is something we could be doing better.

*Could you pull a list of **customer_ids** with **less than 15 rentals** all-time?”*

Solution Query

```
SELECT
    customer_id,
    COUNT(*) AS total_rentals
FROM rental
GROUP BY
    customer_id
HAVING
    COUNT(*) < 15
```

TEST YOUR SKILLS: HAVING

THE ORDER BY CLAUSE

ORDER BY

(Optional) Specifies the order in which your query results are displayed

ORDER BY columnName, otherColumnName

This lets the SQL server know you are about to specify how you want to order (or sort) the records returned in your result set

*This is where you prescribe which column(s) to use to sort your data (default is **ascending** order)*

Examples:

- **ORDER BY** amount
- **ORDER BY** created_date **DESC**
- **ORDER BY** created_date **DESC**, amount **DESC**



HEY THIS IS IMPORTANT!

ORDER BY is **optional**, and is always the **last clause** of “The Big 6”. It comes *after* any WHERE, GROUP BY, or HAVING clauses (if included)



PRO TIP:

*When you use ORDER BY with **multiple criteria** (see the third example above), the server will prioritize sorting the data based on first column specified, then use additional columns as tiebreakers*

ORDER BY

- SQL lets you sort your results by including **ORDER BY** after your **FROM** clause (and after your **WHERE**, **GROUP BY**, and **HAVING** clauses, if applicable)
- ORDER BY** defaults to ascending order (low to high), but can be modified to sort in descending order by adding **DESC** after the column reference

MySQL QUERY IN ACTION:

```
SELECT
customer_id,
rental_id,
amount,
payment_date
FROM payment
ORDER BY amount DESC;
```

QUERY RESULTS:

customer_id	rental_id	amount	payment_date
195	16040	11.99	2005-08-23 22:19:33
13	8831	11.99	2005-07-29 22:37:41
305	2166	11.99	2005-06-17 23:51:21
116	14763	11.99	2005-08-21 23:34:00
237	11479	11.99	2005-08-02 22:18:13
196	106	11.99	2005-05-25 18:18:19
592	3973	11.99	2005-07-06 22:58:31
204	15415	11.99	2005-08-22 23:48:56
591	4383	11.99	2005-07-07 20:45:51

YOUR ASSIGNMENT:

“I’d like to see if our longest films also tend to be our most expensive rentals.

*Could you pull me a list of **all film titles** along with their **lengths** and **rental rates**, and **sort them from longest to shortest**?”*

Result Preview

Result Grid			Filter Rows: <input type="text"/>
title	length	rental_rate	
CHICAGO NORTH	185	4.99	
CONTROL ANTHEM	185	4.99	
DARN FORRESTER	185	4.99	
GANGS PRIDE	185	2.99	
HOME PITY	185	4.99	
MUSCLE BRIGHT	185	2.99	
POND SEATTLE	185	2.99	
SOLDIERS EVOLUTION	185	4.99	

TEST YOUR SKILLS: ORDER BY

YOUR ASSIGNMENT:

“I’d like to see if our longest films also tend to be our most expensive rentals.

*Could you pull me a list of **all film titles** along with their **lengths** and **rental rates**, and **sort them from longest to shortest**?”*

Solution Query

```
SELECT
  title,
  length,
  rental_rate
FROM film
ORDER BY length DESC;
```

TEST YOUR SKILLS: ORDER BY

RECAP: THE “BIG 6” ELEMENTS OF A SQL SELECT STATEMENT

START OF
STATEMENT

SELECT

Identifies the column(s) you want your query to select for your results

SELECT columnName

FROM

Identifies the table(s) your query will pull data from

FROM tableName

WHERE

(Optional) Specifies record-filtering criteria for filtering your results

WHERE logicalCondition

GROUP BY

(Optional) Specifies how to group the data in your results

GROUP BY columnName

HAVING

(Optional) Specifies group-filtering criteria for filtering your results

HAVING logicalCondition

ORDER BY

(Optional) Specifies the order in which your query results are displayed

ORDER BY columnName

END OF
STATEMENT

THE CASE STATEMENT

CASE

Allows you to process a series of IF/THEN logical operators in a specific order

CASE WHEN logic1 **THEN** value1 **WHEN** logic2 **THEN** value2 **ELSE** value3 **END**

Every CASE statement begins with **CASE**, ends with **END**, and contains at least one **THEN/WHEN** pair.

This is where you define your logical operators and the values you want assigned when met.

Example:

```
CASE  
  WHEN category IN ('horror', 'suspense') THEN 'too scary'  
  WHEN length > 90 THEN 'too long'  
  ELSE 'we should see it'  
END
```



HEY THIS IS IMPORTANT!

CASE statements execute in the order they appear; if a record satisfies more than one logical condition, the record will be assigned by the **first THEN statement**



PRO TIP:

I often use my **ELSE** condition as a catch all, and write something like **"Oops...check logic!"**
This helps me quickly see if I forgot to include any conditions in my CASE statement (examples to follow)

CASE STATEMENTS

- CASE Statements allow you to use conditional logic to specify how your results should be calculated for various cases
- One of the most common application for CASE is to “bucket” values, as shown here →

PRO TIP:



When values feel too granular or noisy, **CASE** can help you roll them up to a higher level

MySQL QUERY IN ACTION:

```
SELECT DISTINCT
```

```
length,
```

```
CASE
```

```
  WHEN length < 60 THEN 'under 1 hr'
```

```
  WHEN length BETWEEN 60 and 90 THEN '1-1.5 hrs'
```

```
  WHEN length > 90 THEN 'over 1.5 hrs'
```

```
  ELSE 'uh oh...check logic!'
```

```
END As length_bucket
```

```
FROM film
```

QUERY RESULTS:

Result Grid

	length	length_bucket
▶	86	1-1.5 hrs
	48	under 1 hr
	50	under 1 hr
	117	over 1.5 hrs
	130	over 1.5 hrs
	169	over 1.5 hrs
	62	1-1.5 hrs
	54	under 1 hr

CASE STATEMENTS

- The top **WHEN/THEN** pair executes first. If true, the **CASE** is complete. If not, it continues testing each condition until the **END** is reached

- If multiple **WHEN/THEN** conditions are true, the **upper-most** condition determines the value, since it's the first to be evaluated (*top to bottom*)

- If no conditions are satisfied when the **CASE** reaches the **END**, a **NULL** value will be returned

MySQL QUERY IN ACTION:

```
SELECT DISTINCT
  length,
  CASE
    WHEN length < 60 THEN 'under 1 hr'
    WHEN length BETWEEN 60 and 90 THEN '1-1.5 hrs'
    WHEN length > 90 THEN 'over 1.5 hrs'
    ELSE 'uh oh...check logic!'
  END As length_bucket
FROM film
```

CASE STATEMENT EXECUTION PROCEDURE:

LENGTH	CASE STATEMENT EVALUATION PROCESS ILLUSTRATED
52	1. length < 60 = TRUE → return ' under 1 hr '
65	1. length < 60 = FALSE → try next condition 2. length BETWEEN 60 AND 90 = TRUE → return ' 1-1.5hrs '
125	1. length < 60 = FALSE → try next condition 2. length BETWEEN 60 AND 90 = FALSE → try next 3. length > 90 = TRUE → return ' over 1.5hrs '

CASE WITH ELSE

- In this **CASE**, our buckets are not mutually exclusive or collectively exhaustive
- Notice how **48** and **50** get assigned to **'under 1 hr'** by the first condition, even though they also satisfy the second, and that **117** doesn't satisfy **any** of the given criteria



PRO TIP:

*Include an error message using an **ELSE** statement to see if you missed any logic*

MySQL QUERY IN ACTION:

```
SELECT DISTINCT
  length,
  CASE
    WHEN length < 60 THEN 'under 1 hr'
    WHEN length < 90 THEN '1-1.5 hrs'
    WHEN length > 120 THEN 'over 2 hrs'
    ELSE 'uh oh...check logic!'
  END As length_bucket
FROM film
```

QUERY RESULTS:

Result Grid		Filter
	length	length_bucket
▶	86	1-1.5 hrs
	48	under 1 hr
	50	under 1 hr
	117	uh oh...check logic!
	130	over 2 hrs
	169	over 2 hrs

CASE OPERATORS

- CASE can work with the same set of logical operators that we used with our WHERE statements:

Operator	What it Means
=	Equals
<>	Does NOT Equal
>	Greater Than
<	Less Than
>=	Greater Than Or Equal To
<=	Less Than Or Equal To
BETWEEN	A Range Between Two Values
LIKE	Matching a Pattern Like This
IN()	Equals One of These Values

MySQL QUERY IN ACTION:

```
SELECT DISTINCT
  title,
  CASE
    WHEN rental_duration <= 4 THEN 'rental_too_short'
    WHEN rental_rate >= 3.99 THEN 'too_expensive'
    WHEN rating IN ('NC-17','R') THEN 'too_adult'
    WHEN length NOT BETWEEN 60 AND 90 THEN 'too_short_or_too_long'
    WHEN description LIKE '%Shark%' THEN 'nope_has_sharks'
    ELSE 'great_reco_for_my_niece'
  END AS fit_for_recommendation
FROM film
```

QUERY RESULTS:

title	fit_for_recommendation
ACADEMY DINOSAUR	great_reco_for_my_niece
ACE GOLDFINGER	rental_too_short
ADAPTATION HOLES	too_adult
AFFAIR PREJUDICE	too_short_or_too_long
AFRICAN EGG	too_short_or_too_long
AGENT TRUMAN	rental_too_short
AIRPLANE SIERRA	too_expensive
AIRPORT POLLOCK	too_expensive
ALABAMA DEVIL	rental_too_short
ALADDIN CALENDAR	too_expensive
ALAMO VIDEOTAPE	too_short_or_too_long
ALASKA PHANTOM	too_short_or_too_long
ALI FOREVER	rental_too_short

YOUR ASSIGNMENT:

“I’d like to know which store each customer goes to, and whether or not they are active.

*Could you pull a list of **first and last names of all customers**, and label them as either ‘store 1 active’, ‘store 1 inactive’, ‘store 2 active’, or ‘store 2 inactive’?”*

Result Preview

Result Grid		  Filter Rows: <input type="text"/>
first_name	last_name	store_and_status
▶ MARY	SMITH	store 1 active
PATRICIA	JOHNSON	store 1 active
LINDA	WILLIAMS	store 1 active
BARBARA	JONES	store 2 active
ELIZABETH	BROWN	store 1 active
JENNIFER	DAVIS	store 2 active
MARIA	MILLER	store 1 active
SUSAN	WILSON	store 2 active
MARGARET	MOORE	store 2 active
DOROTHY	TAYLOR	store 1 active
LISA	ANDERS...	store 2 active
NANCY	THOMAS	store 1 active
KAREN	JACKSON	store 2 active
BETTY	WHITE	store 2 active
HELEN	HARRIS	store 1 active
SANDRA	MARTIN	store 2 inactive
DONNA	THOMPS...	store 1 active

TEST YOUR SKILLS: CASE STATEMENTS

YOUR ASSIGNMENT:

“I’d like to know which store each customer goes to, and whether or not they are active.

*Could you pull a list of **first and last names** of all customers, and label them as either ‘store 1 active’, ‘store 1 inactive’, ‘store 2 active’, or ‘store 2 inactive’?”*

Solution Query

```
SELECT
  first_name,
  last_name,
  CASE
    WHEN store_id = 1 AND active = 1 THEN 'store 1 active'
    WHEN store_id = 1 AND active = 0 THEN 'store 1 inactive'
    WHEN store_id = 2 AND active = 1 THEN 'store 2 active'
    WHEN store_id = 2 AND active = 0 THEN 'store 2 inactive'
    ELSE 'oops...check logic!'
  END AS store_and_status
FROM customer
```

ALTERNATIVE OPTIONS:

There are a number of ways to write a valid CASE statement that produces the same results. If you can produce the correct values for these 4 buckets, you’re good to go!

TEST YOUR SKILLS: CASE STATEMENTS

PRO TIP: “PIVOTING” DATA WITH COUNT & CASE

CASE “PIVOTS”

Excel’s ability to pivot to columns can be replicated in SQL using **COUNT** and **CASE**

Excel makes it very easy to “pivot” data on two dimensions.

Here we’re breaking down the count of **inventory_id** by **film_id** (rows) and **store_id** (columns) to quickly see how many copies of each film we have at each store:

	inventory...	film_id	store_id	last_update
1	1	1	1	2006-02-15 05:09:17
2	1	1	1	2006-02-15 05:09:17
3	1	1	1	2006-02-15 05:09:17
4	1	1	1	2006-02-15 05:09:17
5	1	1	2	2006-02-15 05:09:17
6	1	1	2	2006-02-15 05:09:17
7	1	1	2	2006-02-15 05:09:17
8	1	1	2	2006-02-15 05:09:17
9	2	2	2	2006-02-15 05:09:17
10	2	2	2	2006-02-15 05:09:17
11	2	2	2	2006-02-15 05:09:17
12	3	2	2	2006-02-15 05:09:17
13	3	2	2	2006-02-15 05:09:17
14	3	2	2	2006-02-15 05:09:17
15	3	2	2	2006-02-15 05:09:17
16	4	1	1	2006-02-15 05:09:17
17	4	1	1	2006-02-15 05:09:17
18	4	1	1	2006-02-15 05:09:17
19	4	1	1	2006-02-15 05:09:17

Row Labels	1	2	Grand Total
1	4	4	8
2	3	3	6
3	4	4	8
4	4	3	7
5	3	3	6
6	3	3	6
7	2	3	5
8	4	4	8
9	3	2	5
10	4	3	7
11	4	3	7
12	3	4	7
13	4	4	8
14	2	4	6
15	2	2	4
16	3	3	6
17	3	3	6
18	4	2	6
19	3	3	6
20	3	3	6
21	2	4	6
22	4	3	7
23	3	2	5
24	4	4	8
25	4	2	6
26	2	3	5
27	4	4	8
28	3	3	6
29	2	2	4
30	2	2	4
31	4	4	8
32	2	2	4
33	4	4	8
34	4	4	8
35	4	3	7
36	4	3	7
37	4	3	7

Both methods yield identical results

MySQL can do the same thing using **COUNT** functions inside of a **CASE** statement:

film_id	store_1_copies	store_2_copies	total_copies
1	4	4	8
2	0	3	3
3	0	4	4
4	4	3	7
5	0	3	3
6	3	3	6
7	2	3	5
8	0	4	4
9	3	2	5
10	4	3	7
11	4	3	7
12	3	4	7
13	0	4	4
14	2	4	6
15	2	2	4
16	2	2	4
17	3	3	6
18	3	3	6
19	4	2	6
20	3	0	3
21	2	4	6
22	4	3	7

CASE & COUNT

- Excel makes it easy to pivot data into columns and rows
- We can do the same thing in MySQL by using GROUP BY, combined with the “CASE Pivot”
- When CASE Pivoting, we use COUNT() and only count records that match a certain criteria

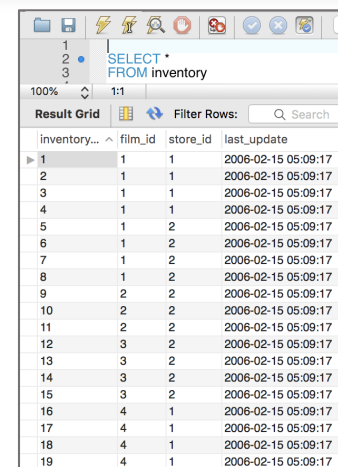
PRO TIP:

Use **GROUP BY** to define your row labels, and **CASE** to pivot to columns

MySQL QUERY IN ACTION:

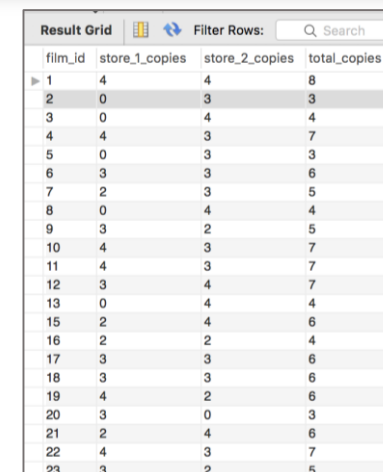
```
SELECT
  film_id,
  COUNT(CASE WHEN store_id = 1 THEN inventory_id ELSE NULL END) AS store_1_copies,
  COUNT(CASE WHEN store_id = 2 THEN inventory_id ELSE NULL END) AS store_2_copies,
  COUNT(inventory_id) AS total_copies
FROM inventory
GROUP BY
  film_id
ORDER BY
  film_id;
```

ORIGINAL TABLE:



inventory...	film_id	store_id	last_update
1	1	1	2006-02-15 05:09:17
2	1	1	2006-02-15 05:09:17
3	1	1	2006-02-15 05:09:17
4	1	1	2006-02-15 05:09:17
5	1	2	2006-02-15 05:09:17
6	1	2	2006-02-15 05:09:17
7	1	2	2006-02-15 05:09:17
8	1	2	2006-02-15 05:09:17
9	2	2	2006-02-15 05:09:17
10	2	2	2006-02-15 05:09:17
11	2	2	2006-02-15 05:09:17
12	3	2	2006-02-15 05:09:17
13	3	2	2006-02-15 05:09:17
14	3	2	2006-02-15 05:09:17
15	3	2	2006-02-15 05:09:17
16	4	1	2006-02-15 05:09:17
17	4	1	2006-02-15 05:09:17
18	4	1	2006-02-15 05:09:17
19	4	1	2006-02-15 05:09:17

QUERY RESULTS:





film_id	store_1_copies	store_2_copies	total_copies
1	4	4	8
2	0	3	3
3	0	4	4
4	4	3	7
5	0	3	3
6	3	3	6
7	2	3	5
8	0	4	4
9	3	2	5
10	4	3	7
11	4	3	7
12	3	4	7
13	0	4	4
15	2	4	6
16	2	2	4
17	3	3	6
18	3	3	6
19	4	2	6
20	3	0	3
21	2	4	6
22	4	3	7
23	3	2	5

YOUR ASSIGNMENT:

“I’m curious how many inactive customers we have at each store.

*Could you please create a table to **count the number of customers** broken down by **store_id** (in rows), and **active status** (in columns)?”*

Result Preview

Result Grid				Filter
	store_id	active	inactive	
▶	1	318	8	
	2	266	7	

TEST YOUR SKILLS: CASE & COUNT

YOUR ASSIGNMENT:

“I’m curious how many inactive customers we have at each store.

*Could you please create a table to **count the number of customers** broken down by **store_id** (in rows), and **active status** (in columns)?”*

Solution Query

```
SELECT
  first_name,
  last_name,
  CASE
    WHEN store_id = 1 AND active = 1 THEN 'store 1 active'
    WHEN store_id = 1 AND active = 0 THEN 'store 1 inactive'
    WHEN store_id = 2 AND active = 1 THEN 'store 2 active'
    WHEN store_id = 2 AND active = 0 THEN 'store 2 inactive'
    ELSE 'oops...check logic!'
  END AS store_and_status
FROM customer
```

TEST YOUR SKILLS: CASE & COUNT

INTRODUCING THE MID COURSE PROJECT

THE SITUATION

The company's insurance policy is up for renewal and the insurance company's underwriters need some updated information from us before they will issue a new policy.

THE OBJECTIVE

Use MySQL to:

Leverage your SQL skills to extract and analyze data from various tables in the Maven Movies database to answer the underwriters' questions. Each question can be answered by querying just one table. Part of your job as an Analyst is figuring out which table to use.

INTRODUCING THE MID COURSE PROJECT

THE LETTER

Dear Maven Movies Management,

In our review of your policy renewal application, we have realized that your business information has not been updated in a number of years.

In order to accurately assess the risk and approve your policy renewal, we will need you to provide all of the following information.

*Sincerely,
Joe Scardycat, Lead Underwriter*

MID COURSE PROJECT QUESTIONS

- 1** We will need a list of all staff members, including their first and last names, email addresses, and the store identification number where they work.
- 2** We will need separate counts of inventory items held at each of your two stores.
- 3** We will need a count of active customers for each of your stores. Separately, please.
- 4** In order to assess the liability of a data breach, we will need you to provide a count of all customer email addresses stored in the database.

MID COURSE PROJECT QUESTIONS

- 5** We are interested in how diverse your film offering is as a means of understanding how likely you are to keep customers engaged in the future. Please provide a count of unique film titles you have in inventory at each store and then provide a count of the unique categories of films you provide.
- 6** We would like to understand the replacement cost of your films. Please provide the replacement cost for the film that is least expensive to replace, the most expensive to replace, and the average of all films you carry.
- 7** We are interested in having you put payment monitoring systems and maximum payment processing restrictions in place in order to minimize the future risk of fraud by your staff. Please provide the average payment you process, as well as the maximum payment you have processed.
- 8** We would like to better understand what your customer base looks like. Please provide a list of all customer identification values, with a count of rentals they have made all-time, with your highest volume customers at the top of the list.

DATABASE FUNDAMENTALS (PART 2)

DATABASE NORMALIZATION

Normalization is the process of structuring the tables and columns in a relational database to **minimize redundancy** and **preserve data integrity**. Benefits of normalization include:

- **Eliminating duplicate data** (*this makes storage and query processing more efficient*)
- **Reducing errors and anomalies** (*restrictions around data structure help to prevent human errors*)

inventory_id	title	release_year	store_address	store_district
1	ACADEMY DINOSAUR	2006	47 MySakila Drive	Alberta
2	ACADEMY DINOSAUR	2006	47 MySakila Drive	Alberta
3	ACADEMY DINOSAUR	2006	47 MySakila Drive	Alberta
4	ACADEMY DINOSAUR	2006	47 MySakila Drive	Alberta
5	ACADEMY DINOSAUR	2006	28 MySQL Boulevard	QLD
6	ACADEMY DINOSAUR	2006	28 MySQL Boulevard	QLD
7	ACADEMY DINOSAUR	2006	28 MySQL Boulevard	QLD
8	ACADEMY DINOSAUR	2006	28 MySQL Boulevard	QLD
9	ACE GOLDFINGER	2006	28 MySQL Boulevard	QLD
10	ACE GOLDFINGER	2006	28 MySQL Boulevard	QLD
11	ACE GOLDFINGER	2006	28 MySQL Boulevard	QLD
12	ADAPTATION HOLES	2006	28 MySQL Boulevard	QLD
13	ADAPTATION HOLES	2006	28 MySQL Boulevard	QLD
14	ADAPTATION HOLES	2006	28 MySQL Boulevard	QLD
15	ADAPTATION HOLES	2006	28 MySQL Boulevard	QLD
16	AFFAIR PREJUDICE	2006	47 MySakila Drive	Alberta
17	AFFAIR PREJUDICE	2006	47 MySakila Drive	Alberta
18	AFFAIR PREJUDICE	2006	47 MySakila Drive	Alberta
19	AFFAIR PREJUDICE	2006	47 MySakila Drive	Alberta
20	AFFAIR PREJUDICE	2006	28 MySQL Boulevard	QLD

HEY THIS IS IMPORTANT!

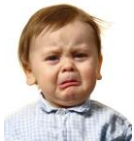
If you **don't normalize** your database, you end up with tables that look like this, with lots and lots of duplicate values.

This may not seem like a lot for the 20 records shown, but can have a *significant* impact as your database scales.

NORMALIZATION: MULTIPLE RELATED TABLES

In practice, normalization involves breaking out data from a **single merged table** into **multiple related tables**

- Instead of storing redundant information about each store and film in a single table (*like the one on the left*), we create **new tables containing a *single record* for each unique value**, and link to those tables using a simple id



Not normalized:

inventory_id	title	release_year	store_address	store_district
1	ACADEMY DINOSAUR	2006	47 MySakila Drive	Alberta
2	ACADEMY DINOSAUR	2006	47 MySakila Drive	Alberta
3	ACADEMY DINOSAUR	2006	47 MySakila Drive	Alberta
4	ACADEMY DINOSAUR	2006	47 MySakila Drive	Alberta
5	ACADEMY DINOSAUR	2006	28 MySQL Boulevard	QLD
6	ACADEMY DINOSAUR	2006	28 MySQL Boulevard	QLD
7	ACADEMY DINOSAUR	2006	28 MySQL Boulevard	QLD
8	ACADEMY DINOSAUR	2006	28 MySQL Boulevard	QLD
9	ACE GOLDFINGER	2006	28 MySQL Boulevard	QLD
10	ACE GOLDFINGER	2006	28 MySQL Boulevard	QLD
11	ACE GOLDFINGER	2006	28 MySQL Boulevard	QLD
12	ADAPTATION HOLES	2006	28 MySQL Boulevard	QLD
13	ADAPTATION HOLES	2006	28 MySQL Boulevard	QLD
14	ADAPTATION HOLES	2006	28 MySQL Boulevard	QLD
15	ADAPTATION HOLES	2006	28 MySQL Boulevard	QLD
16	AFFAIR PREJUDICE	2006	47 MySakila Drive	Alberta
17	AFFAIR PREJUDICE	2006	47 MySakila Drive	Alberta
18	AFFAIR PREJUDICE	2006	47 MySakila Drive	Alberta
19	AFFAIR PREJUDICE	2006	47 MySakila Drive	Alberta
20	AFFAIR PREJUDICE	2006	28 MySQL Boulevard	QLD



Normalized!

inventory_id	film_id	address_id
1	1	1
2	1	1
3	1	1
4	1	1
5	1	2
6	1	2
7	1	2
8	1	2
9	2	2
10	2	2
11	2	2
12	3	2
13	3	2
14	3	2
15	3	2
16	4	1
17	4	1
18	4	1
19	4	1
20	4	2

film_id	title	release_year
1	ACADEMY DINOSAUR	2006
2	ACE GOLDFINGER	2006
3	ADAPTATION HOLES	2006
4	AFFAIR PREJUDICE	2006

address_id	address	district
1	47 MySakila Drive	Alberta
2	28 MySQL Boulevard	QLD

We now have **single records** containing all of the information about our films and addresses – no more redundancy!

TABLE RELATIONSHIPS & CARDINALITY

Cardinality refers to the **uniqueness of values** in a column (*or attribute*) of a table, and is commonly used to describe how two tables relate (***one-to-one***, ***one-to-many***, or ***many-to-many***). For now, here are the key points to grasp:

inventory_id	film_id	address_id
1	1	1
2	1	1
3	1	1
4	1	1
5	1	2
6	1	2
7	1	2
8	1	2
9	2	2
10	2	2
11	2	2
12	3	2
13	3	2
14	3	2
15	3	2
16	4	1
17	4	1
18	4	1
19	4	1
20	4	2

film_id	title	release_year
1	ACADEMY DINOSAUR	2006
2	ACE GOLDFINGER	2006
3	ADAPTATION HOLES	2006
4	AFFAIR PREJUDICE	2006

address_id	address	district
1	47 MySakila Drive	Alberta
2	28 MySQL Boulevard	QLD

- **Primary keys are unique**
 - They **cannot** repeat, so there is only **one** instance of each primary key value in a column
- **Foreign keys are non-unique**
 - They **can** repeat, so there may be **many** instances of each foreign key value in a column
- We can create a **one-to-many** relationship by connecting a **foreign key** in one table to a **primary key** in another

RELATIONSHIP DIAGRAMS

Consider the two tables shown below:

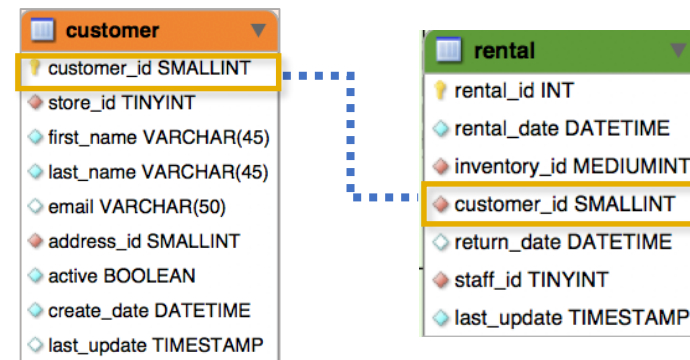
- The **Customer** table below contains details about each customer, identified by a unique **customer_id** (the table's **primary key**)
- The **Rental** table contains records of each rental, and includes a non-unique **customer_id** field since customers may rent films on multiple occasions (this is one of the table's **foreign keys**)

Customer table:

customer_id	store_id	first_name	last_name	email	address_id	active	create_date
1	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5	1	2006-02-14 22:04:36
2	1	PATRICIA	JOHNSON	PATRICIA.JOHNSON@sakilacustomer.org	6	1	2006-02-14 22:04:36
3	1	LINDA	WILLIAMS	LINDA.WILLIAMS@sakilacustomer.org	7	1	2006-02-14 22:04:36
4	2	BARBARA	JONES	BARBARA.JONES@sakilacustomer.org	8	1	2006-02-14 22:04:36
5	1	ELIZABETH	BROWN	ELIZABETH.BROWN@sakilacustomer.org	9	1	2006-02-14 22:04:36
6	2	JENNIFER	DAVIS	JENNIFER.DAVIS@sakilacustomer.org	10	1	2006-02-14 22:04:36
7	1	MARIA	MILLER	MARIA.MILLER@sakilacustomer.org	11	1	2006-02-14 22:04:36
8	2	SUSAN	WILSON	SUSAN.WILSON@sakilacustomer.org	12	1	2006-02-14 22:04:36
9	2	MARGARET	MOORE	MARGARET.MOORE@sakilacustomer.org	13	1	2006-02-14 22:04:36

Rental table:

rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
1	2005-05-24 22:53:30	367	130	2005-05-26 22:04:30	1	2006-02-15 21:30:53
2	2005-05-24 22:54:33	1525	459	2005-05-28 19:40:33	1	2006-02-15 21:30:53
3	2005-05-24 23:03:39	1711	408	2005-06-01 22:12:39	1	2006-02-15 21:30:53
4	2005-05-24 23:04:41	2452	333	2005-06-03 01:43:41	2	2006-02-15 21:30:53
5	2005-05-24 23:05:21	2079	222	2005-06-02 04:33:21	1	2006-02-15 21:30:53
6	2005-05-24 23:08:07	2792	549	2005-05-27 01:32:07	1	2006-02-15 21:30:53
7	2005-05-24 23:11:53	3995	269	2005-05-29 20:34:53	2	2006-02-15 21:30:53
8	2005-05-24 23:31:46	2346	239	2005-05-27 23:33:46	2	2006-02-15 21:30:53
9	2005-05-25 00:00:40	2580	126	2005-05-28 00:22:40	1	2006-02-15 21:30:53



We can diagram table relationships in Workbench to understand how the records in each table relate

NOTE: This isn't required, but serves as a helpful reference



PRO TIP:

When you explore a database for the first time, **diagram your relationships** to understand your table structure

USING JOINS FOR MULTI-TABLE QUERYING

The whole point of table relationships is to enable **multi-table querying** (*i.e. pulling data from multiple tables at once*)

- In SQL, we use **JOIN statements** to do this, by **writing these table relationships directly into our queries**

inventory	film
inventory_id MEDIUMINT	film_id SMALLINT
film_id SMALLINT	title VARCHAR(255)
store_id TINYINT	description TEXT
last_update TIMESTAMP	release_year YEAR
Indexes	language_id TINYINT
	original_language_id TINYINT
	rental_duration TINYINT
	rental_rate DECIMAL(4,2)
	length SMALLINT
	replacement_cost DECIMAL(5,2)
	rating ENUM(...)
	special_features SET(...)
	last_update TIMESTAMP
	Indexes
	Triggers

For example, what if you need a table showing **all film titles in each store's inventory**?

Since **title** and **store_id** live in separate tables, we can't use single-table queries; ***we'll need to use a JOIN!***

SELECT DISTINCT

inventory.inventory_id,
inventory.store_id,
film.title,
film.description

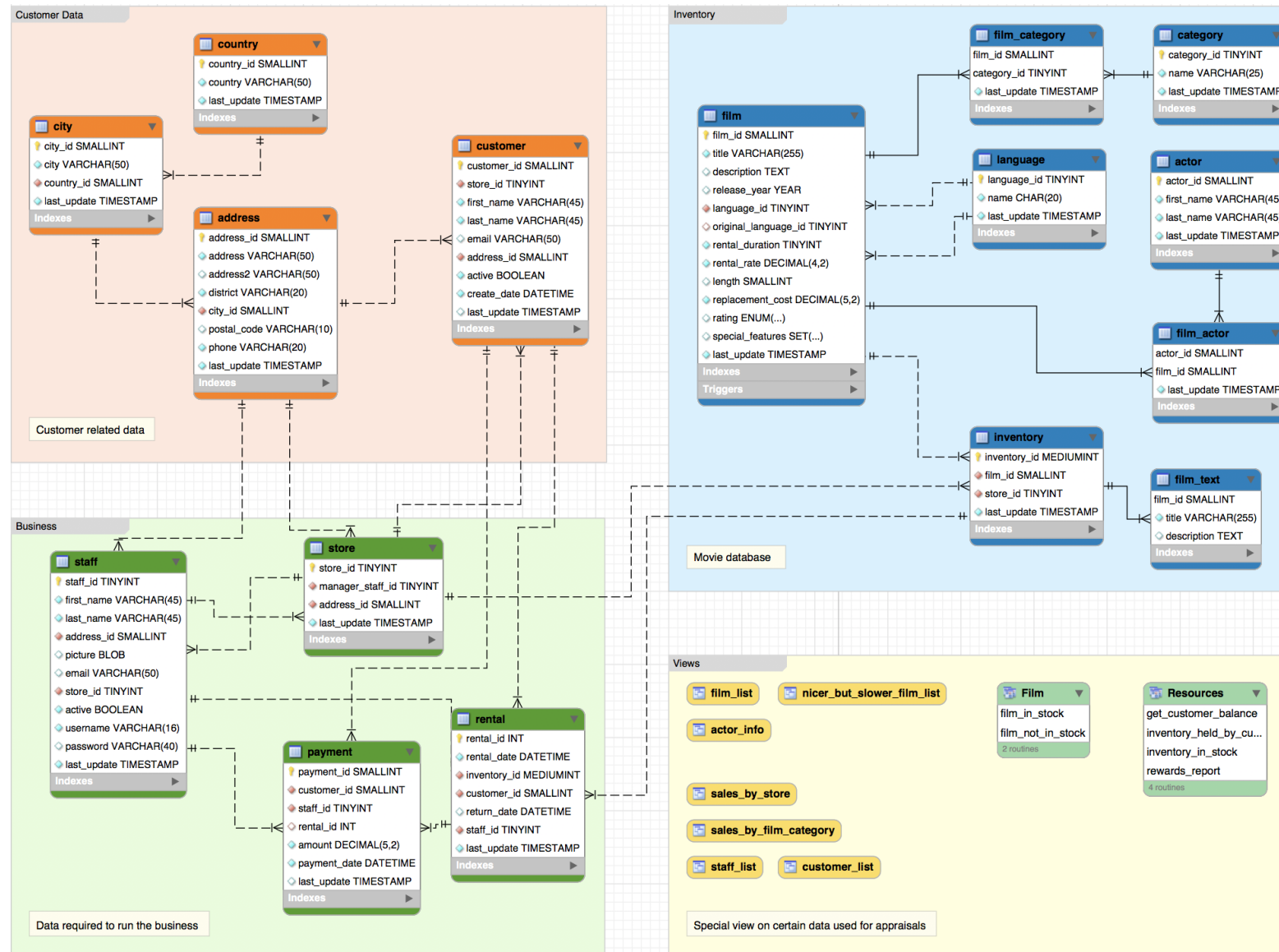
FROM film

INNER JOIN inventory

ON film.film_id = inventory.film_id

Sneak peek at a MySQL JOIN query

FULL MAVEN MOVIES DATABASE



The image shows a screenshot of the phpMyAdmin web interface. On the left, there's a sidebar with a database schema diagram. The main area displays a table with columns: rental_id, rental_date, inventory_id, customer_id, return_date, staff_id, and last_update. The table contains 10 rows of data. On the right, a SQL query window is open, showing a query that selects rental_id, rental_date, inventory_id, customer_id, return_date, staff_id, and last_update from the rental table, grouped by rental_id. The query is: `SELECT rental_id, rental_date, inventory_id, customer_id, return_date, staff_id, last_update FROM rental GROUP BY rental_id`. The results show 10 rows of data. The text "JOINING MULTIPLE TABLES" is overlaid in large white letters across the center of the image.

rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
1	2005-05-24 22:53:30	367	130	2005-05-26 22:04:30	1	2006-02-15 21:30:53
2	2005-05-24 22:54:33	1525	458	2005-05-28 19:40:33	1	2006-02-15 21:30:53
3	2005-05-24 22:54:33	1878	258	2005-06-01 14:12:30	1	2006-02-15 21:30:53
4	2005-05-24 22:54:33	1433	2	2005-06-01 15:21:30	2	2006-02-15 21:30:53
5	2005-05-24 22:54:33	1433	1	2005-06-01 15:21:30	1	2006-02-15 21:30:53
6	2005-05-24 22:54:33	1433	1	2005-06-01 15:21:30	1	2006-02-15 21:30:53
7	2005-05-24 22:54:33	1433	1	2005-06-01 15:21:30	1	2006-02-15 21:30:53
8	2005-05-24 22:54:33	1433	1	2005-06-01 15:21:30	1	2006-02-15 21:30:53
9	2005-05-25 00:00:40	2580	126	2005-05-26 00:22:40	1	2006-02-15 21:30:53
10	2005-05-25 00:02:21	1804	388	2005-05-31 22:44:21	2	2006-02-15 21:30:53

COMMON JOIN TYPES

INNER JOIN

Returns records that exist in **BOTH** tables, and excludes unmatched records from either table

FROM leftTableName
INNER JOIN rightTableName

LEFT JOIN

Returns ALL records from the **LEFT** table, and any matching records from the **RIGHT** table

FROM leftTableName
LEFT JOIN rightTableName

RIGHT JOIN

Returns ALL records from the **RIGHT** table, and any matching records from the **LEFT** table

FROM leftTableName
RIGHT JOIN rightTableName

FULL OUTER JOIN

Returns ALL records from **BOTH** tables, including non-matching records

FROM leftTableName
FULL JOIN rightTableName

UNION

Returns all data from one table, with all data from another table **appended to the end**

SELECT FROM firstName
UNION
SELECT FROM secondTableName

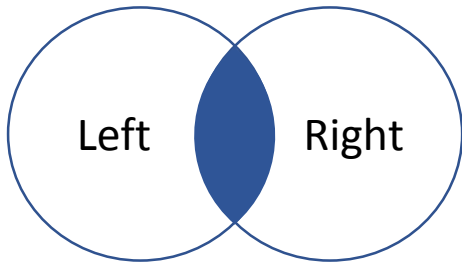
INNER JOIN

INNER JOIN

Returns records from BOTH tables when there is a match, and excludes unmatched records

INNER JOIN rightTableName **ON** leftTable.columnName = rightTable.columnName

Tells SQL to return only the **overlap**



This is where you name your right table, and tell SQL which column to match on by specifying the column name from each table

Example:

```
FROM rental
INNER JOIN customer
ON rental.customer_id = customer.customer_id
```



HEY THIS IS IMPORTANT!

INNER JOIN is one of the two join types you'll likely use most (**LEFT JOIN** is the other). Make sure you understand the differences between the two types!



PRO TIP:

A good way to remember how **INNER JOIN** works is to think of it as only returning the **inner/middle** part of a Venn diagram (where the two tables match/overlap)

INNER JOIN

- **INNER JOIN** lets us pull records from two tables at once, and only returns records when the “JOIN ON” logic produces a match
- After the **FROM** clause, we add **INNER JOIN**, followed by the name of the table we want to join to, followed by logic for how the server should perform the join
- When you write queries using multiple tables, you need to specify both the table and column name (i.e. **inventory.inventory_id**)

MySQL QUERY IN ACTION:

The 1st query pulls
DISTINCT inventory_id
values from **inventory**

The 2nd query joins the
rental and **inventory**
tables to return
matching **inventory_ids**

1 **SELECT DISTINCT** inventory_id
2 **FROM** inventory;
3
4 **SELECT DISTINCT**
5 inventory.inventory_id,
6 rental.inventory_id
7 **FROM** inventory
8 **INNER JOIN** rental
9 **ON** inventory.inventory_id = rental.inventory_id
10

QUERY RESULTS:

Result Grid		Filter Rows:
inventory_id	inventory_id	
1	1	
2	2	
3	3	
4	4	
6	6	
7	7	
8	8	
9	9	
10	10	
11	11	
12	12	

Notice how the 1st query returns **4,581** rows and the 2nd query returns **4,580** rows.

Inventory_id = 5 *does not* appear in the **rental** table, so the record is filtered out of the result set even though it *does* appear in the **inventory** table (must be a *really* bad movie...)

Action Output			
	Time	Action	Response
1	17:39:35	SELECT DISTINCT inventory_id FROM inventory LIMIT...	4581 row(s) returned
2	17:39:35	SELECT DISTINCT inventory.inventory_id, rental.inve...	4580 row(s) returned

ERROR MESSAGE

*Error Code: 1052. Column 'inventory_id' in
field list is ambiguous*

WHAT IT MEANS


*Error Code 1052 means you are
referencing a column name which appears
in more than one of your joined tables,
and you have not specified which table
you want to use.*

HOW TO DUBUG LIKE A PRO

*Read the column name and the location in
the response to find the ambiguous
column name. Specify a table and re-run.
In this case, the location is 'field list'.
Other times, you could see 'where clause',
'on clause', etc.*


ERROR TYPE

COLUMN IS AMBIGUOUS



```
SELECT DISTINCT
  inventory_id
FROM inventory
INNER JOIN rental
ON inventory.inventory_id = rental.inventory_id;
```

A red arrow points from the 'inventory_id' column in the 'FROM' clause to the 'inventory_id' column in the 'ON' clause, highlighting the ambiguity.



```
SELECT DISTINCT
  inventory.inventory_id
FROM inventory
INNER JOIN rental
ON inventory.inventory_id = rental.inventory_id;
```

YOUR ASSIGNMENT:

“Can you pull for me a list of each film we have in inventory?”

*I would like to see the film’s **title**, **description**, and the **store_id** value associated with each item, and its **inventory_id**. Thanks!”*

Result Preview

Result Grid

Filter Rows:

Search

Export:

Fetch rows:

	inventory_id	store_id	title	description
▶	1	1	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Bat
	2	1	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Bat
	3	1	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Bat
	4	1	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Bat
	5	2	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Bat
	6	2	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Bat
	7	2	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Bat
	8	2	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Bat
	9	2	ACE GOLDFINGER	A Astounding Epistle of a Database Administrator And a Explor
	10	2	ACE GOLDFINGER	A Astounding Epistle of a Database Administrator And a Explor
	11	2	ACE GOLDFINGER	A Astounding Epistle of a Database Administrator And a Explor
	12	2	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car who must :
	13	2	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car who must :
	14	2	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car who must :
	15	2	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car who must :

TEST YOUR SKILLS: JOINING TABLES

YOUR ASSIGNMENT:

“Can you pull for me a list of each film we have in inventory?”

*I would like to see the film’s **title**, **description**, and the **store_id** value associated with each item, and its **inventory_id**. Thanks!”*

Solution Query

```
SELECT DISTINCT
    inventory.inventory_id,
    inventory.store_id,
    film.title,
    film.description
FROM film
    INNER JOIN inventory
        ON film.film_id = inventory.film_id
```

TEST YOUR SKILLS: JOINING TABLES

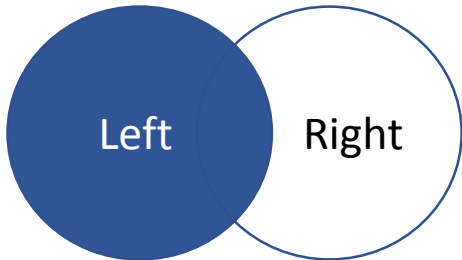
LEFT JOIN

LEFT JOIN

Returns all records from the LEFT table, and any matched records from the RIGHT table

LEFT JOIN rightTableName **ON** leftTable.columnName = rightTable.columnName

Tells SQL to return the overlap and everything from the **left** table



This is where you name your right table, and tell SQL which column to match on by specifying the column name from each table

Example:

```
FROM rental
LEFT JOIN customer
ON rental.customer_id = customer.customer_id
```



HEY THIS IS IMPORTANT!

LEFT JOIN is what you'll use when you want additional data from a second table, and you want to keep ALL records from your first table (even if there isn't a match)



PRO TIP:

This is the join type that I use most often. When you're working with a data set and want to add data from another table **while keeping all of your current records**, LEFT JOIN is the way to go!

LEFT JOIN

- In this example, we're looking for a **COUNT** of all films in which each actor appeared
- We use **LEFT JOIN** here because we want to return *all* of the actors' names, even if they don't match with any film records

MySQL QUERY IN ACTION:

```
SELECT
    actor.first_name,
    actor.last_name,
    COUNT(film_actor.film_id) AS number_of_films
FROM actor
LEFT JOIN film_actor
    ON actor.actor_id = film_actor.actor_id
GROUP BY
    actor.first_name,
    actor.last_name
```

QUERY RESULTS:

Result Grid		Filter Rows:	
first_name	last_name	number_of_films	
SUSAN	DAVIS	54	
GINA	DEGENERES	42	
WALTER	TORN	41	
MARY	KEITEL	40	
MATTHEW	CARREY	39	
SANDRA	KILMER	37	
SCARLETT	DAMON	36	
GROUCHO	DUNST	35	
VAL	BOLGER	35	
ANGELA	WITHERSPOON	35	
UMA	WOOD	35	
HENRY	BERRY	35	

LEFT vs INNER JOIN

- While **INNER JOIN** returns results *only* when there is a match, **LEFT JOIN** returns any matches plus all records from the left table (*the first table named*)
- You can think of **INNER JOIN** as being more restrictive, and **LEFT JOIN** being a little looser

PRO TIP:

Compare your query with **LEFT** and **INNER** joins to quickly master the differences

MySQL QUERY IN ACTION:

```
1 SELECT DISTINCT
2   inventory.inventory_id,
3   rental.inventory_id
4 FROM inventory
5 INNER JOIN rental
6   ON inventory.inventory_id = rental.inventory_id;
7
8
9 SELECT DISTINCT
10  inventory.inventory_id,
11  rental.inventory_id
12 FROM inventory
13 LEFT JOIN rental
14   ON inventory.inventory_id = rental.inventory_id
15
```

INNER JOIN
vs.
LEFT JOIN

QUERY RESULTS:

Result Grid		Filter Rows:
inventory_id	inventory_id	
1	1	
2	2	
3	3	
4	4	
6	6	
7	7	
8	8	
9	9	
10	10	
11	11	
12	12	

INNER JOIN = 4,580 ROWS

Result Grid		Filter Rows:
inventory_id	inventory_id	
1	1	
2	2	
3	3	
4	4	
5	NULL	
6	6	
7	7	
8	8	
9	9	
10	10	
11	11	
12	12	



LEFT JOIN: 4,581 ROWS

YOUR ASSIGNMENT:

*“One of our investors is interested in the films we carry and **how many actors** are listed **for each film title**.*

*Can you pull a list of all titles, and figure out **how many actors** are **associated with each title**?”*

Result Preview

Result Grid   Filter Rows: <input type="text" value="Search"/>		
	title	number_of_actors
▶	ACADEMY DINOSAUR	10
	ACE GOLDFINGER	4
	ADAPTATION HOLES	5
	AFFAIR PREJUDICE	5
	AFRICAN EGG	5
	AGENT TRUMAN	7
	AIRPLANE SIERRA	5
	AIRPORT POLLOCK	4
	ALABAMA DEVIL	9
	ALADDIN CALENDAR	8
	ALAMO VIDEOTAPE	4
	ALASKA PHANTOM	7

TEST YOUR SKILLS: LEFT JOIN

YOUR ASSIGNMENT:

*“One of our investors is interested in the films we carry and **how many actors** are listed **for each film title**.*

*Can you pull a list of all titles, and figure out **how many actors** are **associated with each title**?”*

Solution Query

```
SELECT
    film.title,
    COUNT(film_actor.actor_id) AS number_of_actors
FROM film
    LEFT JOIN film_actor
        ON film.film_id = film_actor.film_id
GROUP BY
    film.title
```

TEST YOUR SKILLS: LEFT JOIN

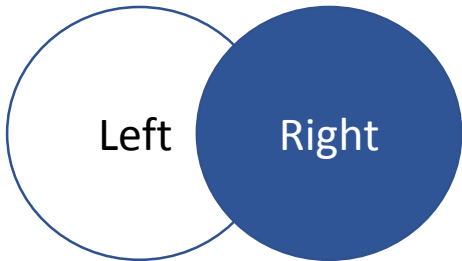
RIGHT JOIN

RIGHT JOIN

Returns all records from the *RIGHT* table, and any matched records from the *LEFT* table

RIGHT JOIN rightTableName **ON** leftTable.columnName = rightTable.columnName

Tells SQL to return the overlap and everything from the **right** table



This is where you tell SQL how to look for a match by specifying columns from each table

Example:

```
FROM rental  
RIGHT JOIN customer  
ON rental.customer_id = customer.customer_id
```



HEY THIS IS IMPORTANT!

RIGHT JOIN is like the *opposite* of **LEFT JOIN**; instead of keeping all records from the first (*left*) table named, it keeps all records from the second (*right*)



PRO TIP:

I write SQL queries every day in my professional career, and I've never needed to use **RIGHT JOIN**.
To keep things simple, just use LEFT JOIN (I'm only teaching you about **RIGHT JOIN** for completeness)

LEFT vs INNER vs RIGHT

LEFT JOIN

```
SELECT
  actor.actor_id,
  actor.first_name AS actor_first,
  actor.last_name AS actor_last,
  actor_award.first_name AS award_first,
  actor_award.last_name AS award_last,
  actor_award.awards
FROM actor
LEFT JOIN actor_award
  ON actor.actor_id = actor_award.actor_id
ORDER BY actor_id
```

Returns **all values** from the **left (first)** table and any records from the right (**second**) table which match the JOIN criteria. Returns **NULL** when no match is found.

actor_id	actor_first	actor_last	award_first	award_last	awards
1	PENELOPE	GUINNESS	PENELOPE	GUINNESS	Emmy, Oscar
2	NICK	WAHLBERG	NICK	WAHLBERG	Emmy
3	ED	CHASE	NULL	NULL	NULL
4	JENNIFER	DAVIS	NULL	NULL	NULL
5	JOHNNY	LOLLOBRIGIDA	JOHNNY	LOLLOBRIGIDA	Emmy
6	BETTE	NICHOLSON	BETTE	NICHOLSON	Oscar
7	GRACE	MOSTEL	NULL	NULL	NULL
8	MATTHEW	JOHANSSON	NULL	NULL	NULL
9	JOE	SWANK	NULL	SWANK	Oscar
10	CHRISTIAN	GABLE	CHRISTIAN	GABLE	Tony
11	ZERO	CAGE	ZERO	CAGE	Emmy
12	KARL	BERRY	KARL	BERRY	Emmy, Oscar, Tony
13	UMA	WOOD	NULL	NULL	NULL
14	VIVIEN	BERGEN	VIVIEN	BERGEN	Emmy
15	CUBA	OLIVIER	NULL	NULL	NULL
16	FRED	COSTNER	NULL	NULL	NULL

INNER JOIN

```
SELECT
  actor.actor_id,
  actor.first_name AS actor_first,
  actor.last_name AS actor_last,
  actor_award.first_name AS award_first,
  actor_award.last_name AS award_last,
  actor_award.awards
FROM actor
INNER JOIN actor_award
  ON actor.actor_id = actor_award.actor_id
ORDER BY actor_id
```

Returns any **matching values** from **both tables**. For records with no match, INNER JOIN does not return values from either table.

actor_id	actor_first	actor_last	award_first	award_last	awards
1	PENELOPE	GUINNESS	PENELOPE	GUINNESS	Emmy, Oscar
2	NICK	WAHLBERG	NICK	WAHLBERG	Emmy
5	JOHNNY	LOLLOBRIGIDA	JOHNNY	LOLLOBRIGIDA	Emmy
6	BETTE	NICHOLSON	BETTE	NICHOLSON	Oscar
9	JOE	SWANK	JOE	SWANK	Oscar
10	CHRISTIAN	GABLE	CHRISTIAN	GABLE	Tony
11	ZERO	CAGE	ZERO	CAGE	Emmy
12	KARL	BERRY	KARL	BERRY	Emmy, Oscar, Tony
14	VIVIEN	BERGEN	VIVIEN	BERGEN	Emmy
17	HELEN	VOIGHT	HELEN	VOIGHT	Tony
18	DAN	TORN	DAN	TORN	Emmy, Oscar
19	BOB	FAWCETT	BOB	FAWCETT	Oscar, Tony
20	LUCILLE	TRACY	LUCILLE	TRACY	Emmy, Tony
21	KIRSTEN	PALTROW	KIRSTEN	PALTROW	Emmy, Oscar, Tony
22	ELVIS	MARX	ELVIS	MARX	Tony
23	SANDRA	KILMER	SANDRA	KILMER	Emmy, Tony

RIGHT JOIN

```
SELECT
  actor.actor_id,
  actor.first_name AS actor_first,
  actor.last_name AS actor_last,
  actor_award.first_name AS award_first,
  actor_award.last_name AS award_last,
  actor_award.awards
FROM actor
RIGHT JOIN actor_award
  ON actor.actor_id = actor_award.actor_id
ORDER BY actor_id
```

Returns **all values** from the **right (second)** table, and any records from the left (**first**) table which match the JOIN criteria. Returns **NULL** when no match is found.

actor_id	actor_first	actor_last	award_first	award_last	awards
NULL	NULL	NULL	JILLIAN	DEMAMP	Emmy, Tony
NULL	NULL	NULL	THOMAS	BARLEY	Tony
NULL	NULL	NULL	SPIKE	THOMPSON	Emmy, Oscar
NULL	NULL	NULL	ALICE	DANGERFI...	Emmy, Tony
NULL	NULL	NULL	MINDY	WANDERS	Tony
NULL	NULL	NULL	ADAM	HOLMVICK	Emmy, Oscar
NULL	NULL	NULL	ASHLEY	SLAGEN	Tony
NULL	NULL	NULL	CAITLIN	HOFFMAN	Tony
1	PENELOPE	GUINNESS	PENELOPE	GUINNESS	Emmy, Oscar
2	NICK	WAHLBE...	NICK	WAHLBERG	Emmy
5	JOHNNY	LOLOB...	JOHNNY	LOLOBRI...	Emmy
6	BETTE	NICHOL...	BETTE	NICHOLSON	Oscar
9	JOE	SWANK	JOE	SWANK	Oscar
10	CHRISTIAN	GABLE	CHRISTIAN	GABLE	Tony
11	ZERO	CAGE	ZERO	CAGE	Emmy
12	KARL	BERRY	KARL	BERRY	Emmy, Oscar, Tony

FULL OUTER JOIN (aka FULL JOIN)

FULL JOIN

Returns all records from BOTH tables when there is a match in either one of the tables

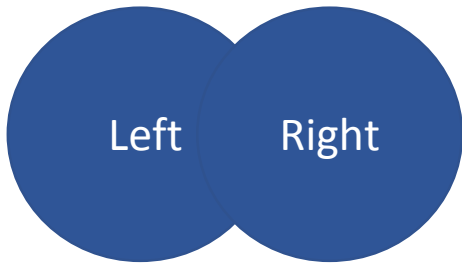
FULL JOIN rightTableName ON leftTable.columnName = rightTable.columnName

Tells SQL to return records from both tables

This is where you tell SQL how to look for a match, by specifying columns from each table

Example:

```
FROM rental
FULL JOIN customer
ON rental.customer_id = customer.customer_id
```



HEY THIS IS IMPORTANT!

Use a **FULL JOIN** when you want records from both tables **even when there isn't a match for the column you're joining on** (this may yield lots of records!)



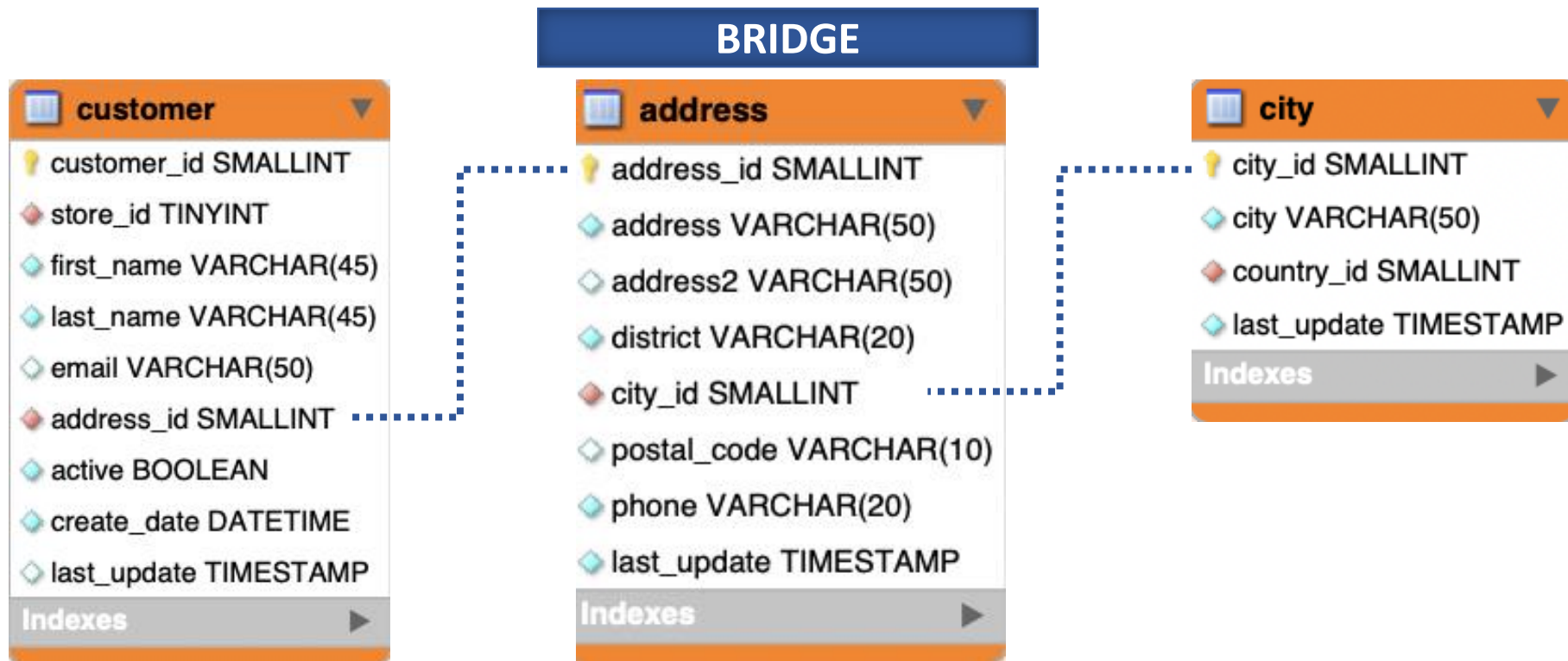
PRO TIP:

FULL JOIN likely isn't something you'll use every day, but can come in handy if you ever need to merge ALL records from two tables

PRO TIP: “BRIDGING” UNRELATED TABLES

When you need to connect two tables which **do not directly relate**, look for a third table containing keys common to both; this can serve as a “**bridge**” to join your tables together

- Here we have no key to connect the **customer** table directly to **city**, but we can join **customer** to **address** (using *address_id*), and **address** to **city** (using *city_id*). In this case the **address** table serves as our bridge.



PRO TIP: BRIDGING

- In this example, we need the **film.title** and **category.name** fields, but the **film** and **category** tables don't have a matching key
- We'll need to find a "**bridge**" table, which relates to both of the tables we want to pull data from
- In this example, **film_category** has keys to both **film** and **category** tables, and can serve as our bridge

PRO TIP:



If you aren't sure if two tables can connect, list the tables each one is joined to. Find a bridge!

MySQL QUERY IN ACTION:

```
SELECT
    film.film_id,
    film.title,
    film_category.category_id,
    category.name

FROM film
    INNER JOIN film_category
        ON film.film_id = film_category.film_id
    INNER JOIN category
        ON film_category.category_id = category.category_id

ORDER BY film_id
```

QUERY RESULTS:

Result Grid			
Filter Rows: <input type="text" value="Q Search"/>			
film_id	title	category_id	name
1	ACADEMY DINOSAUR	6	Documentary
2	ACE GOLDFINGER	11	Horror
3	ADAPTATION HOLES	6	Documentary
4	AFFAIR PREJUDICE	11	Horror
5	AFRICAN EGG	8	Family
6	AGENT TRUMAN	9	Foreign
7	AIRPLANE SIERRA	5	Comedy
8	AIRPORT POLLOCK	11	Horror
9	ALABAMA DEVIL	11	Horror
10	ALADDIN CALENDAR	15	Sports
11	ALAMO VIDEOTAPE	9	Foreign
12	ALASKA PHANTOM	12	Music
13	ALI FOREVER	11	Horror
14	ALICE FANTASIA	4	Classics
15	ALIEN CENTER	9	Foreign

Using two JOINS, we can produce a result set containing values from 3 tables:

- **film**
- **film_category**
- **category**

YOUR ASSIGNMENT:

“Customers often ask which films their favorite actors appear in.

*It would be great to have a list of **all actors**, with each **title** that they appear in. Could you please pull that for me?”*

Result Preview

Result Grid			Filter Rows: <input type="text" value="Search"/>
	first_name	last_name	title
▶	CHRISTIAN	AKROYD	BACKLASH UNDEFEAT...
	CHRISTIAN	AKROYD	BETRAYED REAR
	CHRISTIAN	AKROYD	CAPER MOTIONS
	CHRISTIAN	AKROYD	CATCH AMISTAD
	CHRISTIAN	AKROYD	CHANCE RESURRECTI...
	CHRISTIAN	AKROYD	CONFUSED CANDLES
	CHRISTIAN	AKROYD	CUPBOARD SINNERS
	CHRISTIAN	AKROYD	DIVIDE MONSTER
	CHRISTIAN	AKROYD	DOOM DANCING
	CHRISTIAN	AKROYD	DOORS PRESIDENT
	CHRISTIAN	AKROYD	DRIVER ANNIE
	CHRISTIAN	AKROYD	FEATHERS METAL
	CHRISTIAN	AKROYD	FIRE WOLVES
	CHRISTIAN	AKROYD	HILLS NEIGHBORS
	CHRISTIAN	AKROYD	HOME PITY
	CHRISTIAN	AKROYD	INNOCENT USUAL

TEST YOUR SKILLS: BRIDGING

YOUR ASSIGNMENT:

“Customers often ask which films their favorite actors appear in.

*It would be great to have a list of **all actors**, with each **title** that they appear in. Could you please pull that for me?”*

Solution Query

```
SELECT
  actor.first_name,
  actor.last_name,
  film.title

FROM actor
  INNER JOIN film_actor
    ON actor.actor_id = film_actor.actor_id
  INNER JOIN film
    ON film_actor.film_id = film.film_id

ORDER BY
  actor.last_name,
  actor.first_name
```

TEST YOUR SKILLS: BRIDGING

MULTI-CONDITION JOINS

- The top query joins **film** to **category** (using **film_category** as a bridge), then uses a **WHERE** clause to filter down the result set to only include 'horror' films
- The bottom query makes being a 'horror' film part of the **JOIN** criteria, by adding an **AND** statement within the **ON** clause (which makes the join itself more selective)
- Both methods are valid, and both filter the result set to 'horror' films, producing identical results

MySQL QUERY IN ACTION:

```
SELECT
  film.film_id,
  film.title,
  film.rating,
  category.name

FROM film
  INNER JOIN film_category
    ON film.film_id = film_category.film_id
  INNER JOIN category
    ON film_category.category_id = category.category_id
WHERE category.name = 'horror'

ORDER BY film_id
```



These two queries produce the exact same results

```
SELECT
  film.film_id,
  film.title,
  film.rating,
  category.name

FROM film
  INNER JOIN film_category
    ON film.film_id = film_category.film_id
  INNER JOIN category
    ON film_category.category_id = category.category_id
  AND category.name = 'horror'

ORDER BY film_id
```


YOUR ASSIGNMENT:

“The Manager from Store 2 is working on expanding our film collection there.

*Could you pull a list of **distinct titles** and their **descriptions**, currently available in inventory at **store 2**?”*

Result Preview

Result Grid		Filter Rows:	Search	Export:
title	description			
ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in The Canadian Rockies			
ACE GOLDFINGER	A Astounding Epistle of a Database Administrator And a Explorer who must Find a Car in Ancient China			
ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car who must Sink a Lumberjack in A Baloon Factory			
AFFAIR PREJUDICE	A Fanciful Documentary of a Frisbee And a Lumberjack who must Chase a Monkey in A Shark Tank			
AFRICAN EGG	A Fast-Paced Documentary of a Pastry Chef And a Dentist who must Pursue a Forensic Psychologist in The Gulf of Mexico			
AGENT TRUMAN	A Intrepid Panorama of a Robot And a Boy who must Escape a Sumo Wrestler in Ancient China			
AIRPLANE SIERRA	A Touching Saga of a Hunter And a Butler who must Discover a Butler in A Jet Boat			
AIRPORT POLLOCK	A Epic Tale of a Moose And a Girl who must Confront a Monkey in Ancient India			
ALABAMA DEVIL	A Thoughtful Panorama of a Database Administrator And a Mad Scientist who must Outgun a Mad Scientist in A Jet Boat			
ALADDIN CALENDAR	A Action-Packed Tale of a Man And a Lumberjack who must Reach a Feminist in Ancient China			
ALAMO VIDEOTAPE	A Boring Epistle of a Butler And a Cat who must Fight a Pastry Chef in A MySQL Convention			
ALASKA PHANTOM	A Fanciful Saga of a Hunter And a Pastry Chef who must Vanquish a Boy in Australia			
ALI FOREVER	A Action-Packed Drama of a Dentist And a Crocodile who must Battle a Feminist in The Canadian Rockies			
ALIEN CENTER	A Brilliant Drama of a Cat And a Mad Scientist who must Battle a Feminist in A MySQL Convention			
ALLEY EVOLUTION	A Fast-Paced Drama of a Robot And a Composer who must Battle a Astronaut in New Orleans			
ALONE TRIP	A Fast-Paced Character Study of a Composer And a Dog who must Outgun a Boat in An Abandoned Fun House			

TEST YOUR SKILLS: MULTI-CONDITION JOINS

YOUR ASSIGNMENT:

“The Manager from Store 2 is working on expanding our film collection there.

*Could you pull a list of **distinct titles** and their **descriptions**, currently available in inventory at **store 2**?”*

Solution Query

```
SELECT DISTINCT
    film.title,
    film.description
FROM film
    INNER JOIN inventory
        ON film.film_id = inventory.film_id
        AND inventory.store_id = 2
```

TEST YOUR SKILLS: MULTI-CONDITION JOINS

UNION: A DIFFERENT WAY TO COMBINE TABLES

UNION

Returns all data from the *FIRST* table, with all data from the *SECOND* table appended to the end

UNION SELECT sameColumnName **FROM** secondTableName

UNION tells SQL to combine the results of one **SELECT** statement with another

+

Instead of a JOIN to another table, we write a second **SELECT** statement here:

Example:

```
-- yields one column with all names (first+last)
SELECT first_name, last_name FROM advisor
UNION
SELECT first_name, last_name FROM investor
```



HEY THIS IS IMPORTANT!

UNION will **deduplicate** records, and keep only *distinct* values in your result set. If you want to keep duplicate records as well, use **UNION ALL**.



PRO TIP:

UNION is an easy one to run into errors with. Make sure that **A)** your two **SELECT** statements have the same number of columns, **B)** columns are in the same order, and **C)** columns in each table have similar data types.

UNION

- **UNION** allows you to combine the results of two or more **SELECT** statements into a single result set
- Think of **UNION** as “stacking” two result sets on top of each other to make one longer result set

- Each **SELECT** statement must include the same number of columns, and stacked columns must share compatible formats



MySQL QUERY IN ACTION:

```
SELECT
    'advisor' AS type,
    first_name,
    last_name
FROM advisor

UNION

SELECT
    'investor' AS type,
    first_name,
    last_name
FROM investor
```

QUERY RESULTS:



Result Grid   Filter Rows: <input type="text"/>			
	type	first_name	last_name
▶	advisor	Barry	Beenthere
	advisor	Cindy	Smartypants
	advisor	Mary	Moneybags
	advisor	Walter	White
	investor	Montgomery	Burns
	investor	Anthony	Stark
	investor	William	Wonka

YOUR ASSIGNMENT:

“We will be hosting a meeting with all of our staff and advisors soon.

*Could you pull one list of **all staff and advisor names**, and include a column **noting whether they are a staff member or advisor**? Thanks!”*

Result Preview

Result Grid   Filter Rows:			
	type	first_name	last_name
	advisor	Barry	Beenthere
	advisor	Cindy	Smartyants
	advisor	Mary	Moneybags
	advisor	Walter	White
	staff	Mike	Hillyer
	staff	Jon	Stephens

TEST YOUR SKILLS: UNION

YOUR ASSIGNMENT:

“We will be hosting a meeting with all of our staff and advisors soon.

*Could you pull one list of **all staff and advisor names**, and include a column **noting whether they are a staff member or advisor**? Thanks!”*

Solution Query

```
SELECT
    'advisor' AS type,
    first_name,
    last_name
FROM advisor

UNION

SELECT
    'staff' AS type,
    first_name,
    last_name
FROM staff
```

TEST YOUR SKILLS: UNION

INTRODUCING THE FINAL COURSE PROJECT

THE SITUATION

You and your business partner were recently approached by another local business owner who is interested in purchasing Maven Movies. He primarily owns restaurants and bars, so he has lots of questions for you about your business and the rental business in general. His offer seems very generous, so you are going to entertain his questions.

THE OBJECTIVE

Use MySQL to:

Leverage your SQL skills to extract and analyze data from various tables in the Maven Movies database to answer your potential Acquirer's questions. Each question will require you to write a multi-table SQL query, joining at least two tables.

INTRODUCING THE FINAL COURSE PROJECT

THE LETTER

Dear Maven Movies Management,

I am excited about the potential acquisition and learning more about your rental business.

Please bear with me as I am new to the industry, but I have a number of questions for you. Assuming you can answer them all, and that there are no major surprises, we should be able to move forward with the purchase.

*Best,
Martin Moneybags*

FINAL COURSE PROJECT QUESTIONS

- 1** My partner and I want to come by each of the stores in person and meet the managers. Please send over the managers' names at each store, with the full address of each property (street address, district, city, and country please).
- 2** I would like to get a better understanding of all of the inventory that would come along with the business. Please pull together a list of each inventory item you have stocked, including the store_id number, the inventory_id, the name of the film, the film's rating, its rental rate and replacement cost.
- 3** From the same list of films you just pulled, please roll that data up and provide a summary level overview of your inventory. We would like to know how many inventory items you have with each rating at each store.
- 4** Similarly, we want to understand how diversified the inventory is in terms of replacement cost. We want to see how big of a hit it would be if a certain category of film became unpopular at a certain store. We would like to see the number of films, as well as the average replacement cost, and total replacement cost, sliced by store and film category.

FINAL COURSE PROJECT QUESTIONS

- 5** We want to make sure you folks have a good handle on who your customers are. Please provide a list of all customer names, which store they go to, whether or not they are currently active, and their full addresses – street address, city, and country.
- 6** We would like to understand how much your customers are spending with you, and also to know who your most valuable customers are. Please pull together a list of customer names, their total lifetime rentals, and the sum of all payments you have collected from them. It would be great to see this ordered on total lifetime value, with the most valuable customers at the top of the list.
- 7** My partner and I would like to get to know your board of advisors and any current investors. Could you please provide a list of advisor and investor names in one table? Could you please note whether they are an investor or an advisor, and for the investors, it would be good to include which company they work with.
- 8** We're interested in how well you have covered the most-awarded actors. Of all the actors with three types of awards, for what % of them do we carry a film? And how about for actors with two types of awards? Same questions. Finally, how about actors with just one award?