Instructor Guide: Sakila Video — Rentals and Revenue

CONFIDENTIAL — FOR INSTRUCTOR USE ONLY

Assignment Overview

This assignment covers foundational to advanced SQL querying skills using the Sakila sample database. Students are expected to demonstrate proficiency in SELECT with filtering (WHERE, LIKE), single and multi-table JOINs, aggregate functions (COUNT, SUM, AVG, ROUND), GROUP BY, HAVING for post-aggregation filtering, ORDER BY, and LIMIT. Questions 1–4 are straightforward single-table queries testing basic syntax and clause ordering. Questions 5–7 introduce multi-table joins with aggregation and HAVING. Question 8 is a capstone item requiring students to chain six tables together and combine multiple aggregation and filtering techniques. By the end, students should be able to trace a data path through a normalized schema, choose the correct aggregation strategy, and distinguish between WHERE and HAVING.

Partial Credit Policy

In general, award partial credit when a student demonstrates understanding of the correct technique but makes a minor implementation error. For easy questions (Q1–Q4, 8 points each): full credit for correct results, 6/8 for minor errors (wrong sort direction, boundary condition), 4/8 for correct table and columns but wrong filtering logic, 2/8 for correct SELECT structure but missing WHERE or ORDER BY entirely. For medium questions (Q5–Q7, 12–14 points): break points down by clause correctness — JOIN (30%), GROUP BY (20%), aggregate (20%), HAVING/ORDER BY (20%), output format (10%). For the hard question (Q8, 28 points): use the per-component breakdown described in Q8 grading tips. Never award zero to a student who shows a reasonable attempt using the correct tables — at minimum, credit the FROM clause and any correct JOINs. Do not penalize for stylistic differences (alias names, capitalization, whitespace, schema prefix usage) as long as the query logic is correct.

General Grading Tips

- Always run each student's query against the actual Sakila database before grading — some syntactically creative solutions (subqueries, CTEs, different join orders) may be logically equivalent to the model answer.
- Check the WHERE clause in the code, not just the output — especially for Questions 1, 3, and 4 where boundary conditions (> vs >=) produce nearly identical result sets.
- For Questions 5–8, inspect the GROUP BY clause carefully. MySQL's default mode (without ONLY_FULL_GROUP_BY) may allow sloppy GROUP BY usage that still produces correct results — penalize the SQL logic, not just the output.

• Watch for students who hardcode values (e.g., manually listing actor IDs instead of using HAVING) — these solutions are fragile and should receive partial credit only.

• Encourage students to use table aliases in complex queries (Q8 especially) — while not required, it signals good SQL habits and improves readability.

• For Q8, it is worth giving a brief in-class walkthrough of the join chain before grading so students understand the schema path. Confusion about the inventory table is the most common conceptual blocker.

• If students use USE sakila; at the top of their script instead of fully qualifying table names, this is acceptable — do not penalize.

Question-by-Question Guide

Question 1

Common Student Mistakes

• Using >= 3.00 instead of > 3.00, returning films at exactly $3.00

• Omitting the schema prefix (sakila.) if the default database is not set

• Forgetting ORDER BY or ordering ASC instead of DESC

• Selecting extra columns not requested (e.g., film_id) without understanding that this is usually acceptable but should match the spec

• Using WHERE rental_rate > 3 without the decimal, which works in MySQL but reflects imprecision in understanding data types

Socratic Questions (When Students Are Stuck)

• If a student is stuck, ask: What clause do you use to filter rows in SQL, and where does it appear relative to SELECT and FROM?

• If a student is stuck, ask: The question says 'greater than $3.00' — what comparison operator expresses 'greater than' (not 'greater than or equal to')?

• If a student is stuck, ask: How would you tell SQL which direction to sort — and what keyword controls ascending vs. descending?

Solution Walkthrough

Step 1: Identify the source table — only sakila.film is needed. Step 2: Specify the three required columns in SELECT: title, rental_rate, rating. Step 3: Add a WHERE clause filtering rental_rate > 3.00 (strict greater-than). Step 4: Add ORDER BY rental_rate DESC to sort highest rates first. Final query: SELECT film.title, film.rental_rate, film.rating FROM sakila.film WHERE film.rental_rate > 3.00 ORDER BY film.rental_rate DESC;

Grading Tips

Award full credit if the correct rows are returned in correct order with the three specified columns. Deduct 2 points for using >= instead of > (wrong boundary). Deduct 2 points for missing or incorrect ORDER BY direction. Deduct 1 point for minor style issues (no table prefix) if results are still correct. Do not penalize for including additional non-requested columns unless the assignment rubric explicitly forbids it.

Discussion Prompt

Why might a business analyst care specifically about films priced above a threshold rather than at-or-above? How does the choice of > versus >= affect downstream business decisions, such as promotional pricing?

Edge Cases

- Films with rental_rate = 3.00 exactly should NOT appear — if a student's result includes them, they used >= and should lose points.
- If the student does not set sakila as the default database and omits the schema prefix, the query will error; accept queries that use either fully qualified names or a USE sakila; statement at the top.

Question 2

Common Student Mistakes

- Using LIKE 'S_' instead of LIKE 'S%', which only matches last names exactly two characters long starting with S
- Using LIKE '%S%' instead of 'S%', returning all last names containing S anywhere
- Forgetting the secondary sort on first_name — only sorting by last_name
- Case sensitivity concerns: MySQL LIKE is case-insensitive by default on most collations, but students may write 's%' thinking it matters
- Sorting DESC instead of ASC (the default is ASC, but students sometimes explicitly write DESC)

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: What SQL operator is used to match patterns in strings, and what wildcard character matches any sequence of characters?
- If a student is stuck, ask: If you wanted last names that begin with 'S', where in the pattern string would you place the wildcard — before or after the letter?
- If a student is stuck, ask: The question asks to sort by last name and then by first name — how do you specify more than one sort column in ORDER BY?

Solution Walkthrough

Step 1: Source table is sakila.customer. Step 2: SELECT first_name, last_name, email. Step 3: Add WHERE last_name LIKE 'S%' — the pattern 'S%' matches any string beginning with S followed by zero or more characters. Step 4: ORDER BY last_name ASC, first_name ASC — the secondary sort on first_name handles ties where multiple customers share the same last name. Final query: SELECT customer.first_name, customer.last_name, customer.email FROM sakila.customer WHERE customer.last_name LIKE 'S%' ORDER BY customer.last_name ASC, customer.first_name ASC;

Grading Tips

Award full credit for correct pattern matching and both sort keys. Deduct 3 points for incorrect LIKE pattern (e.g., '%S%' or 'S_'). Deduct 2 points for missing the secondary sort on first_name. Deduct 1 point for DESC instead of ASC. The ASC keyword is optional (it is the default), so do not penalize students who omit it.

Discussion Prompt

Why is LIKE generally slower than an equality check on large tables? What database feature could speed up prefix searches like 'S%', and would it help with '%S%'?

Edge Cases

- Customers with last name exactly 'S' (one character) should be included — the % wildcard matches zero or more characters, so 'S%' does match 'S'. Verify students understand this.
- If a student uses REGEXP '^S' or SUBSTRING, the logic may be correct; award full credit if results match.

Question 3

Common Student Mistakes

- Using AND incorrectly — writing OR instead of AND, returning films that satisfy either condition
- Using >= 120 instead of > 120 for length, or <= 20.00 instead of < 20.00 for replacement_cost
- Confusing the sort column — ordering by replacement_cost instead of length
- Forgetting ORDER BY entirely
- Selecting the wrong columns (e.g., including rental_rate instead of replacement_cost)

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: The question has two conditions that must both be true — what logical operator in SQL requires all listed conditions to be satisfied simultaneously?
- If a student is stuck, ask: 'Longer than 120 minutes' — does 'longer than' include 120 itself, or only values strictly above it?
- If a student is stuck, ask: Which column should drive the sort order, and in which direction?

Solution Walkthrough

Step 1: Source table is sakila.film. Step 2: SELECT title, length, replacement_cost. Step 3: WHERE length > 120 AND replacement_cost < 20.00 — both conditions must hold simultaneously, so AND is correct. Step 4: ORDER BY length ASC to show shortest qualifying films first. Final query: SELECT film.title, film.length, film.replacement_cost FROM sakila.film WHERE film.length > 120 AND film.replacement_cost < 20.00 ORDER BY film.length ASC;

Grading Tips

Award full credit for correct two-condition filtering and ascending sort on length. Deduct 3 points for using OR instead of AND (fundamentally wrong logic, result set will be much larger). Deduct 2 points for boundary errors (>= vs >). Deduct 2 points for wrong sort column or direction. Deduct 1 point for selecting wrong fields.

Discussion Prompt

How would the result set change if OR were used instead of AND? Can you think of a real-world business scenario where OR would be the correct choice for these two conditions instead?

Edge Cases

- Films with length exactly 120 should NOT appear (strict >). If a student's result includes them, penalize for boundary error.
- Films with replacement_cost exactly $20.00 should NOT appear (strict <). Same penalty applies.

Question 4

Common Student Mistakes

- Using active = TRUE or active = 'true' instead of active = 1, which may still work in MySQL but shows conceptual uncertainty
- Using active = 0 instead of active = 1, returning inactive staff
- Forgetting ORDER BY last_name or ordering DESC instead of ASC
- Selecting too many or too few columns — the question asks for first_name, last_name, and email only
- Querying the customer table instead of the staff table

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: The schema says active is a BOOLEAN — what integer values does MySQL use to represent TRUE and FALSE in a BOOLEAN/TINYINT column?
- If a student is stuck, ask: You want staff who ARE active, not inactive — which value, 0 or 1, represents an active record?
- If a student is stuck, ask: Both customer and staff tables have an active column — which table does this question ask about?

Solution Walkthrough

Step 1: Source table is sakila.staff. Step 2: SELECT first_name, last_name, email. Step 3: WHERE active = 1 — in MySQL, BOOLEAN is stored as TINYINT(1), where 1 = TRUE (active) and 0 = FALSE (inactive). Step 4: ORDER BY last_name ASC. Final query: SELECT staff.first_name, staff.last_name, staff.email FROM sakila.staff WHERE staff.active = 1 ORDER BY staff.last_name ASC; Note: The Sakila staff table has only a few rows, so the result set will be small — this is normal.

Grading Tips

Award full credit for correct table, correct active filter (1 or TRUE), correct columns, and correct ORDER BY. Deduct 3 points for filtering active = 0 (returns inactive staff — inverted logic). Deduct 1 point for using active = TRUE vs active = 1 — both work in MySQL, but accept either. Deduct 2 points for querying the wrong table (customer instead of staff). Note: Because the staff table is small, a student who prints all staff without filtering will have results that look almost correct — check the WHERE clause explicitly.

Discussion Prompt

Why might a database store a boolean as an integer (TINYINT) rather than a native BOOLEAN type? Are there situations where this could cause confusion or bugs in application code?

Edge Cases

- The Sakila staff table contains very few rows (typically 2). If a student returns both rows but the active filter is wrong, their output may coincidentally look correct if both happen to be active — always inspect the SQL, not just the output.
- Using WHERE active = TRUE is valid in MySQL and should receive full credit.

Question 5

Common Student Mistakes

- Joining in the wrong direction or joining on the wrong key (e.g., joining category_id to film_id)
- Forgetting GROUP BY and getting a single aggregated row or a MySQL implicit grouping error
- Using COUNT(*) instead of COUNT(film_category.film_id) — both give the same result here but show different understanding
- Not including category.name in GROUP BY (MySQL may allow this with ONLY_FULL_GROUP_BY disabled, but it is non-standard)
- Ordering ASC instead of DESC
- Not aliasing the COUNT column, making the output column name unclear

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: You need data from two tables — what SQL clause connects them, and which columns do they share that you can link on?
- If a student is stuck, ask: You want one row per category — what clause groups rows together so you can count within each group?
- If a student is stuck, ask: After grouping, how do you count the number of films within each group?

Solution Walkthrough

Step 1: Identify tables — category and film_category, linked by category_id. Step 2: JOIN sakila.film_category ON category.category_id = film_category.category_id. Step 3: SELECT category.name and COUNT(film_category.film_id) AS total_films. Step 4: GROUP BY category.category_id, category.name — grouping by the PK is safest; including name is required in strict SQL mode. Step 5: ORDER BY total_films DESC. Final query: SELECT category.name, COUNT(film_category.film_id) AS total_films FROM sakila.category JOIN sakila.film_category ON category.category_id = film_category.category_id GROUP BY category.category_id, category.name ORDER BY total_films DESC;

Grading Tips

Award full credit for correct join, correct GROUP BY, correct COUNT, and correct ORDER BY. Deduct 3 points for missing GROUP BY (fundamentally broken query). Deduct 2 points for joining on wrong columns. Deduct 1 point for missing alias on the count column. Deduct 1 point for ordering in the wrong direction. COUNT(*) vs COUNT(film_category.film_id) — both produce identical results here since film_id is NOT NULL in film_category; award full credit for either.

Discussion Prompt

Why is it important to GROUP BY category.category_id in addition to category.name? What problem could arise if two categories had the same name but different IDs?

Edge Cases

- If a category has zero films (no matching rows in film_category), an INNER JOIN will exclude it. In Sakila, all categories have films, so this won't affect results — but if a student uses LEFT JOIN to be safe, that is also correct and should receive full credit.
- Some students may GROUP BY category.name only — this is technically wrong in strict SQL mode and conceptually fragile; deduct 1 point but accept if results are correct in the test

environment.

Question 6

Common Student Mistakes

- Using LEFT JOIN and including customers with zero payments (the question says 'at least one payment', so INNER JOIN is appropriate — though in practice every customer in Sakila has payments)
- Forgetting LIMIT 10 and returning all customers
- Using AVG instead of SUM for total amount paid
- Not grouping by customer_id, leading to incorrect aggregation when first/last names could theoretically repeat
- Concatenating first and last name into a single column (CONCAT) when the question asks for them separately
- Ordering ASC instead of DESC

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: To get one row per customer showing their total spending, what two clauses do you need — one to combine rows and one to sum a value within each group?
- If a student is stuck, ask: The question asks for the TOP 10 customers — what SQL clause restricts the number of rows returned?
- If a student is stuck, ask: Which aggregate function gives you a running total of all payment amounts — SUM or COUNT?

Solution Walkthrough

Step 1: Tables needed — customer and payment, linked by customer_id. Step 2: JOIN sakila.payment ON customer.customer_id = payment.customer_id (INNER JOIN ensures only customers with payments). Step 3: SELECT customer.first_name, customer.last_name, SUM(payment.amount) AS total_paid. Step 4: GROUP BY customer.customer_id, customer.first_name, customer.last_name — grouping by PK prevents issues with name collisions. Step 5: ORDER BY total_paid DESC. Step 6: LIMIT 10. Final query: SELECT customer.first_name, customer.last_name, SUM(payment.amount) AS total_paid FROM sakila.customer JOIN sakila.payment ON customer.customer_id = payment.customer_id GROUP BY customer.customer_id, customer.first_name, customer.last_name ORDER BY total_paid DESC LIMIT 10;

Grading Tips

Award full credit for correct JOIN, SUM aggregation, GROUP BY, ORDER BY DESC, and LIMIT 10. Deduct 4 points for missing LIMIT 10 (major requirement missed). Deduct 3 points for using AVG instead of SUM. Deduct 2 points for missing GROUP BY or grouping by name only. Deduct 1 point for ordering ASC. If a student uses a subquery approach to get top 10, accept it if results are correct.

Discussion Prompt

If two customers had identical first and last names, how would grouping by customer_id vs. grouping by name only affect the results? Why is it a best practice to group by the primary key when grouping by a named entity?

Edge Cases

- In Sakila, all active customers have payment records, so LEFT JOIN vs INNER JOIN produces the same result — accept either, but note the conceptual difference in grading feedback.
- Some students may round the SUM to 2 decimal places — this is not required but not wrong; accept it.

Question 7

Common Student Mistakes

- Using WHERE COUNT(...) > 30 instead of HAVING COUNT(...) > 30 — WHERE cannot filter on aggregate functions
- Forgetting to JOIN the film_actor table and trying to count from actor alone
- Putting the HAVING condition before GROUP BY in the written query (syntax error)
- Using HAVING total_films > 30 with the alias — MySQL allows this, but standard SQL requires repeating the aggregate expression; accept either
- Not grouping by actor_id, leading to inaccurate counts if two actors share a name
- Confusing COUNT(film_actor.film_id) with COUNT(film_actor.actor_id)

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: You want to filter based on a COUNT — can you put aggregate functions in a WHERE clause? If not, what clause is designed for filtering after aggregation?
- If a student is stuck, ask: You need the number of films per actor — what table connects actors to films, and what column in that table identifies a film?
- If a student is stuck, ask: In what order do WHERE, GROUP BY, and HAVING appear in a SQL query?

Solution Walkthrough

Step 1: Tables — actor and film_actor, linked by actor_id. Step 2: JOIN sakila.film_actor ON actor.actor_id = film_actor.actor_id. Step 3: SELECT actor.first_name, actor.last_name, COUNT(film_actor.film_id) AS total_films. Step 4: GROUP BY actor.actor_id, actor.first_name, actor.last_name. Step 5: HAVING COUNT(film_actor.film_id) > 30 — this filters groups after aggregation. Step 6: ORDER BY total_films DESC. Final query: SELECT actor.first_name, actor.last_name, COUNT(film_actor.film_id) AS total_films FROM sakila.actor JOIN sakila.film_actor ON actor.actor_id = film_actor.actor_id GROUP BY actor.actor_id, actor.first_name, actor.last_name HAVING COUNT(film_actor.film_id) > 30 ORDER BY total_films DESC;

Grading Tips

Award full credit for correct JOIN, GROUP BY, HAVING with correct threshold, and ORDER BY. Deduct 5 points for using WHERE instead of HAVING on the aggregate (this is a key conceptual error and the query will fail or produce wrong results). Deduct 2 points for missing GROUP BY. Deduct 1 point for wrong ORDER BY direction. Accept HAVING total_films > 30 (using alias) — MySQL permits this even though it is non-standard.

Discussion Prompt

Explain in your own words why WHERE cannot be used to filter aggregate results, while HAVING can. At what stage of query processing does each clause execute?

Edge Cases

- Actors with exactly 30 films should NOT appear (strict >). Check boundary condition in student results.
- If a student uses a subquery to pre-aggregate and then filters with WHERE in an outer query, this is logically equivalent and should receive full credit if results are correct.

Question 8

Common Student Mistakes

- Missing one or more joins in the chain (e.g., skipping inventory and trying to join film directly to rental)
- Joining tables in an incorrect order or on wrong keys (e.g., joining rental on film_id instead of inventory_id)
- Forgetting ROUND() on the AVG or rounding to wrong decimal places
- Using WHERE SUM(payment.amount) > 5000 instead of HAVING SUM(payment.amount) > 5000
- Grouping only by category.name instead of category.category_id and category.name
- Forgetting to include payment in the join chain and trying to sum film.rental_rate instead of payment.amount
- Double-counting revenue because of multiple inventory copies — not understanding that each rental row ties to one payment

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: The hint gives you the join path: category → film_category → film → inventory → rental → payment. Starting from category, what column links it to film_category?
- If a student is stuck, ask: You have two aggregates needed — AVG of rental_rate and SUM of payment amount. Both need to be calculated after grouping by category. What clause groups your data, and what clause lets you filter on the grouped totals?
- If a student is stuck, ask: Why do you need the inventory table between film and rental? What column does the rental table use to identify what was rented?

Solution Walkthrough

Step 1: Identify the full join chain — category → film_category (on category_id) → film (on film_id) → inventory (on film_id) → rental (on inventory_id) → payment (on rental_id). Step 2: SELECT category.name, ROUND(AVG(film.rental_rate), 2) AS avg_rental_rate, SUM(payment.amount) AS total_revenue. Step 3: Build the JOIN chain: FROM sakila.category JOIN sakila.film_category ON category.category_id = film_category.category_id JOIN sakila.film ON film_category.film_id = film.film_id JOIN sakila.inventory ON film.film_id = inventory.film_id JOIN sakila.rental ON inventory.inventory_id = rental.inventory_id JOIN sakila.payment ON rental.rental_id = payment.rental_id. Step 4: GROUP BY category.category_id, category.name. Step 5: HAVING SUM(payment.amount) > 5000. Step 6: ORDER BY total_revenue DESC. Final query: SELECT category.name, ROUND(AVG(film.rental_rate), 2) AS avg_rental_rate, SUM(payment.amount) AS

total_revenue FROM sakila.category JOIN sakila.film_category ON category.category_id = film_category.category_id JOIN sakila.film ON film_category.film_id = film.film_id JOIN sakila.inventory ON film.film_id = inventory.film_id JOIN sakila.rental ON inventory.inventory_id = rental.inventory_id JOIN sakila.payment ON rental.rental_id = payment.rental_id GROUP BY category.category_id, category.name HAVING SUM(payment.amount) > 5000 ORDER BY total_revenue DESC;

Grading Tips

This question is worth 28 points — allocate partial credit carefully. Suggested breakdown: 8 points for correct complete join chain (2 per correct join link after the first), 4 points for correct AVG with ROUND, 4 points for correct SUM, 4 points for correct GROUP BY, 4 points for correct HAVING (not WHERE), 2 points for correct ORDER BY, 2 points for correct column aliasing and output. Deduct 4 points per missing join link. Deduct 3 points for using WHERE instead of HAVING. Deduct 2 points for missing ROUND or wrong precision. Award partial credit even if only 4 of 6 joins are present, if the query demonstrates understanding of the chain concept.

Discussion Prompt

Why is the inventory table necessary in this join chain? Could you ever join film directly to rental — and what would be wrong with that? How does the presence of multiple inventory copies of the same film affect the SUM of payments?

Edge Cases

- Some categories may fall exactly at $5,000 in total revenue — they should be excluded (strict >). Verify student results against the exact threshold.
- The AVG(film.rental_rate) in this context averages rental rate across rental transactions, not unique films — a film rented 100 times contributes its rental_rate 100 times to the average. This is statistically meaningful (transaction-weighted average) and is the correct interpretation given the join chain. If a student flags this in a comment, acknowledge it as good critical thinking.
- A student might try to use a CTE or subquery approach — accept it if results are correct, and note it as a sophisticated solution.