Instructor Guide: Chinook Music Store — Sales Analysis

CONFIDENTIAL — FOR INSTRUCTOR USE ONLY

Assignment Overview

This assignment covers foundational to intermediate SQL querying skills using the Chinook music store database. Students are expected to demonstrate proficiency in SELECT with filtering and sorting (Q1–Q2), multi-table JOINs (Q3, Q6), LEFT JOIN with aggregation (Q4), HAVING-based group filtering (Q5), subquery-based anti-join patterns (Q7), and conditional aggregation with CASE WHEN (Q8). By the end, students should be able to construct queries that span multiple tables, apply aggregate functions correctly, filter at both row and group levels, and perform conditional classification of aggregated results. Watch for whether students understand the semantic difference between WHERE and HAVING, when to use LEFT vs INNER JOIN, and how NOT IN behaves with NULLs.

Partial Credit Policy

Partial credit should reflect demonstrated understanding of the relevant SQL concepts, not just whether the final output is correct. A query that uses the right technique (e.g., LEFT JOIN, HAVING, NOT IN) but has a minor syntactic or key-mapping error should receive 60–75% credit. A query that retrieves from the correct tables with correct columns but is missing a critical clause (WHERE, ORDER BY, HAVING) should receive 50–65% credit. A query that shows no understanding of the primary technique being tested (e.g., using WHERE instead of HAVING for Q5, using INNER JOIN instead of LEFT JOIN for Q4, or completely missing the subquery for Q7) should receive no more than 40% credit even if the remaining logic is correct. Queries that produce correct output through an approach not taught in the course (e.g., procedural workarounds) should receive credit for output correctness but the instructor should note the preferred approach. Always award 0 points for plagiarized queries, but consult institutional policy before assigning academic integrity violations.

General Grading Tips

- Run each student query against the actual Chinook database before grading to verify row counts — many questions have specific expected outputs (e.g., Q7 should return 0 rows in the standard Chinook dataset).
- For multi-table JOINs, check not just whether the student joined tables but whether they joined on the correct keys — a plausible-looking but wrong join (e.g., joining on Name instead of ID) can accidentally produce correct-looking results with incorrect logic.
- Pay attention to whether students qualify column names with table names or aliases — in multi-table queries, unqualified columns can cause ambiguity errors or silent bugs if column names

overlap (especially 'Name').

• Distinguish between syntactic errors (query fails to run) and semantic errors (query runs but produces wrong results) — semantic errors typically warrant heavier deductions because they reflect conceptual misunderstanding.

• Watch for students who hard-code specific IDs or values that happen to work for this dataset but are not generalizable — deduct points and comment on query reusability.

• When a student's query produces an empty result set, verify whether the empty set is actually the correct answer (as in Q7) before assuming the student made an error.

• Accept logically equivalent alternative approaches (NOT EXISTS vs NOT IN, LEFT JOIN IS NULL vs NOT IN, subquery vs join-based aggregation) unless the question explicitly requires a specific technique.

Question-by-Question Guide

Question 1

Common Student Mistakes

• Using 'United States' or 'US' instead of the exact string 'USA' in the WHERE clause, returning zero rows

• Omitting one of the ORDER BY columns, or reversing the sort priority (ordering by FirstName before LastName)

• Including the Country column in the SELECT output when the question only asks for FirstName, LastName, and Email

• Using a case-insensitive comparison that happens to work in MySQL but may fail in other engines — not a grading issue, but worth noting pedagogically

• Forgetting the schema prefix 'chinook.' if the database requires it

Socratic Questions (When Students Are Stuck)

• If a student is stuck: What column in the Customer table tells you where a customer lives? How would you compare that to a specific text value?

• If a student returns 0 rows: What values are actually stored in the Country column — have you checked a few sample rows? Does your string match exactly?

• If a student sorts incorrectly: The question says 'alphabetically by last name, then by first name' — how does SQL evaluate multiple columns in ORDER BY, and which one comes first in your clause?

Solution Walkthrough

Step 1: Identify the source table — only chinook.Customer is needed. Step 2: Select the three required columns: FirstName, LastName, Email. Step 3: Add a WHERE clause filtering Country = 'USA' (exact string match). Step 4: Add ORDER BY LastName ASC, FirstName ASC to satisfy the sort requirement. Final query: SELECT Customer.FirstName, Customer.LastName, Customer.Email FROM chinook.Customer WHERE Customer.Country = 'USA' ORDER BY Customer.LastName ASC, Customer.FirstName ASC;

Grading Tips

Award full 5 points for a correct query producing the right columns, correct filter, and correct sort order. Deduct 1 point for wrong sort order (e.g., FirstName before LastName). Deduct 1–2 points for the wrong country string if the student can articulate they understood the concept. Deduct 1 point for including extra columns beyond what was asked. Award 3/5 if the correct table and filter logic are present but the ORDER BY is entirely missing. Award 1–2/5 if only a basic SELECT with no WHERE is present.

Discussion Prompt

Why might storing country names as free-text strings (like 'USA') rather than ISO country codes cause problems in real-world databases? How would a database designer mitigate this?

Edge Cases

- Some students may use LIKE 'USA' or LIKE '%USA%' — the latter is incorrect because it could match 'USA Territory'; deduct 1 point for the broader LIKE but give credit if they explain the intent.
- Students who omit the chinook schema prefix: accept if the database context is set correctly; note it as a best-practice issue rather than a correctness issue.

Question 2

Common Student Mistakes

- Using >= 0.99 instead of > 0.99, which incorrectly includes tracks priced at exactly $0.99
- Reversing the ORDER BY direction — ordering UnitPrice ascending instead of descending, or Name descending instead of ascending
- Comparing UnitPrice to the string '0.99' instead of the numeric literal 0.99 (may work in MySQL but reflects poor understanding)
- Including extra columns like TrackId or AlbumId that were not requested
- Omitting the secondary sort by track name entirely

Socratic Questions (When Students Are Stuck)

- If a student is stuck: Which column in the Track table stores the price? What SQL operator means 'strictly greater than'?
- If a student uses >= 0.99: The question says 'greater than $0.99' — does greater than include the value itself, or only values above it?
- If a student reverses sort order: Walk me through what 'ordered by unit price descending' means — should the highest prices appear first or last?

Solution Walkthrough

Step 1: Source table is chinook.Track. Step 2: Select Name and UnitPrice only. Step 3: WHERE UnitPrice > 0.99 — strictly greater than, not greater-than-or-equal. Step 4: ORDER BY UnitPrice DESC, Name ASC — price highest first, then alphabetically within same price tier. Final query: SELECT Track.Name, Track.UnitPrice FROM chinook.Track WHERE Track.UnitPrice > 0.99 ORDER BY Track.UnitPrice DESC, Track.Name ASC;

Grading Tips

Deduct 2 points for using >= instead of > (this is a semantic error, not just stylistic). Deduct 1 point for missing the secondary sort by Name. Deduct 1 point for wrong sort direction on either column. Award 4/5 if the filter and primary sort are correct but secondary sort is missing. Award 2/5 if the student retrieves from the right table with the right columns but the filter logic is entirely wrong.

Discussion Prompt

In a real e-commerce database, why might you avoid storing prices as plain NUMERIC columns and instead use a separate pricing table with effective dates? What business scenarios make this important?

Edge Cases

- If UnitPrice values in the actual Chinook dataset are only 0.99 and 1.99, a student may be surprised by how many rows qualify — confirm with them that 1.99 > 0.99 is true.
- Students who use WHERE UnitPrice <> 0.99: This is logically wrong (includes prices below 0.99); treat as a conceptual misunderstanding and deduct 2–3 points.

Question 3

Common Student Mistakes

- Using only one JOIN and forgetting to also join the second lookup table (Genre or MediaType)
- Joining on the wrong foreign key columns (e.g., joining Genre to MediaType directly instead of both to Track)
- Selecting Genre.Name and MediaType.Name without aliasing them, then being confused by duplicate column names in the result
- Using a LEFT JOIN when an INNER JOIN is appropriate here (all tracks are expected to have a genre and media type)
- Reversing the ORDER BY priority — ordering by track name before genre name

Socratic Questions (When Students Are Stuck)

- If a student is stuck: How many tables do you need to connect? Draw the path: Track has a GenreId — where does that lead? Track also has a MediaTypeId — where does that lead?
- If a student joins incorrectly: What column in Track links it to Genre? What column in Track links it to MediaType? Are you joining each pair on the right keys?
- If a student gets ambiguous column name errors: When two tables both have a column called 'Name', how does SQL know which one you mean? How can you make it explicit?

Solution Walkthrough

Step 1: Start from chinook.Track as the central table. Step 2: JOIN chinook.Genre ON Track.GenreId = Genre.GenreId — this brings in Genre.Name. Step 3: JOIN chinook.MediaType ON Track.MediaTypeId = MediaType.MediaTypeId — this brings in MediaType.Name. Step 4: SELECT Track.Name, Genre.Name, MediaType.Name — qualify all three to avoid ambiguity. Step 5: ORDER BY Genre.Name ASC, Track.Name ASC. Recommended to alias Genre.Name AS GenreName and MediaType.Name AS MediaTypeName for clarity. Final query: SELECT Track.Name, Genre.Name, MediaType.Name FROM chinook.Track JOIN chinook.Genre ON Track.GenreId = Genre.GenreId JOIN chinook.MediaType ON Track.MediaTypeId = MediaType.MediaTypeId ORDER BY Genre.Name

ASC, Track.Name ASC;

Grading Tips

Award full 10 points for correct joins on correct keys, correct columns selected, and correct sort order. Deduct 3 points for missing one of the two JOINs entirely. Deduct 2 points for joining on incorrect keys (e.g., GenreId = MediaTypeId). Deduct 1 point for missing or incorrect sort order. Deduct 1 point for unqualified column names that would cause ambiguity errors. Award partial credit of 5/10 if the student correctly joins two of the three tables and demonstrates understanding of the JOIN concept.

Discussion Prompt

This query produces a Cartesian-product-style concern if joins are done incorrectly. How does specifying the correct ON condition prevent a cross join? What would happen to the row count if you forgot the ON clause entirely?

Edge Cases

- Some students may use table aliases (t, g, mt) — this is perfectly acceptable and should be credited fully as long as the logic is correct.
- Students who use comma-separated tables with WHERE conditions instead of JOIN syntax (old-style implicit joins): logically equivalent if correct — award full credit but note the preferred modern syntax.

Question 4

Common Student Mistakes

- Using INNER JOIN instead of LEFT JOIN, which silently excludes artists with zero albums
- Counting the wrong column — COUNT(*) instead of COUNT(Album.AlbumId); COUNT(*) will return 1 instead of 0 for artists with no albums when using LEFT JOIN
- Forgetting to GROUP BY or grouping only by Artist.Name when Artist.ArtistId should also be included (or used as the primary group key) to handle potential name duplicates
- Not aliasing the COUNT result, making ORDER BY on it fail or be unclear
- Ordering by the alias string name with quotes, causing a literal string sort rather than numeric sort

Socratic Questions (When Students Are Stuck)

- If a student uses INNER JOIN: The question says 'include artists who have no albums' — what kind of JOIN keeps rows from the left table even when there's no match on the right?
- If a student gets 1 instead of 0 for artists with no albums: What does COUNT(*) count? What does COUNT(Album.AlbumId) count when Album.AlbumId is NULL due to a LEFT JOIN with no match?
- If a student forgets GROUP BY: What error do you get when you mix aggregate functions with non-aggregated columns without a GROUP BY? What does GROUP BY actually do to the rows?

Solution Walkthrough

Step 1: Start from chinook.Artist — this is the LEFT side because we want all artists. Step 2: LEFT JOIN chinook.Album ON Artist.ArtistId = Album.ArtistId — for artists with no albums, Album columns will be NULL. Step 3: GROUP BY Artist.ArtistId, Artist.Name — group per artist. Step 4: COUNT(Album.AlbumId) — counting a nullable column from the right side of a LEFT JOIN correctly

returns 0 when there are no matching albums. Step 5: Alias the count as AlbumCount. Step 6: ORDER BY AlbumCount DESC, Artist.Name ASC. Final query: SELECT Artist.Name, COUNT(Album.AlbumId) AS AlbumCount FROM chinook.Artist LEFT JOIN chinook.Album ON Artist.ArtistId = Album.ArtistId GROUP BY Artist.ArtistId, Artist.Name ORDER BY AlbumCount DESC, Artist.Name ASC;

Grading Tips

The LEFT JOIN vs INNER JOIN distinction is the core learning objective here — deduct 4 points if INNER JOIN is used (the result is logically wrong). Deduct 2 points for COUNT(*) instead of COUNT(Album.AlbumId) (partial conceptual understanding). Deduct 1 point for missing secondary sort. Award 6/10 if the student uses LEFT JOIN correctly but counts incorrectly. Award 4/10 if the student uses INNER JOIN but otherwise has correct GROUP BY and COUNT logic.

Discussion Prompt

Why does COUNT(column) behave differently from COUNT(*) when used with a LEFT JOIN? Can you think of a scenario where COUNT(*) would actually give you the wrong answer compared to COUNT(column)?

Edge Cases

- Some students may GROUP BY Artist.Name only — this is technically acceptable in MySQL (which allows grouping by non-primary-key columns) but is ambiguous if two artists share the same name; note this but do not heavily penalize unless it produces incorrect results.
- Students who use a subquery to count albums per artist and then LEFT JOIN back to Artist: logically correct alternative approach — award full credit.

Question 5

Common Student Mistakes

- Using WHERE COUNT(...) > 50 instead of HAVING COUNT(...) > 50 — WHERE cannot reference aggregate functions
- Using LEFT JOIN instead of INNER JOIN, which could include genres with no tracks (count of 0) and never qualify the HAVING filter anyway — functionally benign but conceptually imprecise
- Referencing the alias TrackCount in the HAVING clause (not supported in all databases — MySQL allows it, but standard SQL requires repeating the aggregate expression)
- Forgetting to GROUP BY before applying HAVING, resulting in a single aggregate over all tracks
- Not ordering the results at all, or ordering ascending instead of descending

Socratic Questions (When Students Are Stuck)

- If a student uses WHERE COUNT(...) > 50: At what point in query execution does WHERE filter rows? At what point does COUNT run? Can WHERE see aggregate results?
- If a student is unsure about HAVING: If WHERE filters individual rows before grouping, what clause would you use to filter groups after the aggregate is calculated?
- If a student forgets GROUP BY: What does COUNT(Track.TrackId) without a GROUP BY return? One number or many — and why?

Solution Walkthrough

Step 1: Source tables are chinook.Genre and chinook.Track. Step 2: JOIN chinook.Track ON Genre.GenreId = Track.GenreId — INNER JOIN is appropriate since we only care about genres that have tracks. Step 3: GROUP BY Genre.GenreId, Genre.Name — one row per genre. Step 4: HAVING COUNT(Track.TrackId) > 50 — filter groups where track count exceeds 50. Step 5: SELECT Genre.Name, COUNT(Track.TrackId) AS TrackCount. Step 6: ORDER BY TrackCount DESC. Final query: SELECT Genre.Name, COUNT(Track.TrackId) AS TrackCount FROM chinook.Genre JOIN chinook.Track ON Genre.GenreId = Track.GenreId GROUP BY Genre.GenreId, Genre.Name HAVING COUNT(Track.TrackId) > 50 ORDER BY TrackCount DESC;

Grading Tips

The WHERE vs HAVING distinction is the primary learning objective — if a student uses WHERE COUNT(...) > 50, that is a syntactic/conceptual error; deduct 5 points but award credit for correct GROUP BY and JOIN. Award 10/15 if HAVING is used correctly but ORDER BY is missing. Award 7/15 if the student demonstrates correct grouping and aggregation but applies the filter incorrectly. Award full credit if a student uses a subquery to filter (e.g., wrapping the grouped query) even if HAVING was not used directly — it demonstrates equivalent understanding.

Discussion Prompt

What is the logical order of SQL clause execution (FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY)? Why does understanding this order matter when deciding whether to use WHERE or HAVING?

Edge Cases

- Students who use HAVING TrackCount > 50 (referencing the alias): This works in MySQL but not in standard SQL. Accept it but note the portability concern.
- If a student uses > 50 vs >= 50: '> 50' means strictly more than 50, which is correct. A student using >= 51 produces the same result for integer counts — accept this.

Question 6

Common Student Mistakes

- Filtering by title using HAVING instead of WHERE, leading to unnecessary computation
- Joining Employee to Invoice directly without going through Customer, missing the intermediary link
- Forgetting to JOIN Invoice at all and trying to SUM a column that doesn't exist in Customer
- Not grouping by all non-aggregated SELECT columns (EmployeeId, FirstName, LastName, Title), causing errors
- Using COUNT instead of SUM for revenue calculation
- Omitting the WHERE Title = 'Sales Support Agent' filter and returning all employees

Socratic Questions (When Students Are Stuck)

- If a student is stuck on the join path: Look at the tables — Employee connects to Customer how? And Customer connects to Invoice how? Can you trace a path from Employee to Invoice through those foreign keys?

- If a student filters by title using HAVING: HAVING filters after grouping — is there any reason we'd want to first aggregate rows for non-Sales-Support-Agent employees and then discard them? What's more efficient?
- If a student uses COUNT(Invoice.Total) instead of SUM: The question asks for total revenue — does COUNT give you a sum of values, or a count of rows? Which function adds the values together?

Solution Walkthrough

Step 1: Start from chinook.Employee. Step 2: JOIN chinook.Customer ON Employee.EmployeeId = Customer.SupportRepId — connects each employee to their supported customers. Step 3: JOIN chinook.Invoice ON Customer.CustomerId = Invoice.CustomerId — connects customers to their invoices. Step 4: WHERE Employee.Title = 'Sales Support Agent' — pre-filter to relevant employees before aggregation. Step 5: GROUP BY Employee.EmployeeId, Employee.FirstName, Employee.LastName, Employee.Title — one row per employee. Step 6: SUM(Invoice.Total) AS TotalRevenue — sum all invoice totals for that employee's customers. Step 7: ORDER BY TotalRevenue DESC. Final query: SELECT Employee.FirstName, Employee.LastName, Employee.Title, SUM(Invoice.Total) AS TotalRevenue FROM chinook.Employee JOIN chinook.Customer ON Employee.EmployeeId = Customer.SupportRepId JOIN chinook.Invoice ON Customer.CustomerId = Invoice.CustomerId WHERE Employee.Title = 'Sales Support Agent' GROUP BY Employee.EmployeeId, Employee.FirstName, Employee.LastName, Employee.Title ORDER BY TotalRevenue DESC;

Grading Tips

Award full 15 points for correct three-table join path, correct filter, correct aggregation, and correct sort. Deduct 4 points for a missing JOIN (e.g., Employee joined directly to Invoice). Deduct 3 points for COUNT instead of SUM for revenue. Deduct 2 points for missing the WHERE Title filter. Deduct 1 point for missing ORDER BY or wrong direction. Deduct 1 point for incomplete GROUP BY. Award 8/15 for a query that demonstrates correct aggregation and grouping logic but has the wrong join path.

Discussion Prompt

In a real business scenario, why might attributing revenue to a support representative be a useful metric? What are the limitations of this approach — for example, what if a customer's support rep changed over time?

Edge Cases

- Students who use HAVING Employee.Title = 'Sales Support Agent' instead of WHERE: Functionally equivalent in most cases but semantically wrong; deduct 1 point and explain the difference.
- Students who hard-code specific employee IDs instead of filtering by title: Penalize 2 points for non-generalizable approach, but award credit if results are correct.

Question 7

Common Student Mistakes

- Using NOT EXISTS instead of NOT IN — this is actually a valid and often preferable approach; do not penalize it
- Using an INNER JOIN with a NULL filter instead of a proper anti-join pattern, getting the logic backwards
- Using LEFT JOIN ... WHERE Invoice.CustomerId IS NULL — this is a valid alternative anti-join pattern; award full credit
- Confusing NOT IN with NOT LIKE — a fundamental misunderstanding of set operations
- Not handling the fact that in this dataset, all customers have invoices (the result set may be empty) — students may think their query is wrong when it is correct

Socratic Questions (When Students Are Stuck)

- If a student is stuck: You need to find customers whose ID does NOT appear in another table — what kind of SQL construct lets you check whether a value is absent from a set of values?
- If a student gets an empty result set and is confused: Is it possible that every customer in the Chinook database has made at least one purchase? How would you verify that? Does an empty result necessarily mean your query is wrong?
- If a student tries to use a JOIN: If you join Customer to Invoice on CustomerId, what rows do you get? What kind of join — and what additional filter — would you need to find customers with NO matching invoice rows?

Solution Walkthrough

Step 1: The goal is to find Customer rows with no corresponding Invoice rows. Step 2 (NOT IN approach): Write a subquery SELECT Invoice.CustomerId FROM chinook.Invoice to get all customer IDs that appear in the Invoice table. Step 3: In the outer query, WHERE Customer.CustomerId NOT IN (subquery) — this filters to customers whose ID is not in that set. Step 4: SELECT FirstName, LastName, Email. Step 5: ORDER BY LastName ASC, FirstName ASC. Alternative (LEFT JOIN): SELECT Customer.FirstName, Customer.LastName, Customer.Email FROM chinook.Customer LEFT JOIN chinook.Invoice ON Customer.CustomerId = Invoice.CustomerId WHERE Invoice.CustomerId IS NULL ORDER BY Customer.LastName ASC, Customer.FirstName ASC. Both approaches are fully correct. Note: In the Chinook sample dataset, all customers have invoices, so the expected result is an empty set — this is correct behavior.

Grading Tips

Award full 20 points for either NOT IN with subquery, NOT EXISTS with correlated subquery, or LEFT JOIN + IS NULL anti-join pattern. Deduct 5 points if the student uses NOT IN but puts the wrong column in the subquery (e.g., InvoiceId instead of CustomerId). Deduct 5 points if a student uses an INNER JOIN and tries to filter — they likely have the logic inverted. Award 14/20 if the anti-join concept is correct but the student misidentifies the joining key. Award 5/20 if the student demonstrates awareness of the anti-join concept but cannot execute it correctly. Note on empty result: do not penalize students for returning zero rows if their query is logically correct.

Discussion Prompt

What are the performance implications of NOT IN vs NOT EXISTS vs LEFT JOIN IS NULL for anti-join patterns? In particular, what dangerous behavior does NOT IN exhibit when the subquery contains NULL values, and how does NOT EXISTS avoid this?

Edge Cases

- NOT IN with NULLs: If Invoice.CustomerId could be NULL (it cannot due to FK constraints in Chinook, but worth noting), NOT IN would return no rows at all. Students who mention this caveat should receive bonus acknowledgment.
- Students who use a correlated subquery with NOT EXISTS: SELECT ... FROM Customer c WHERE NOT EXISTS (SELECT 1 FROM Invoice i WHERE i.CustomerId = c.CustomerId) — award full credit, this is the most robust approach.
- Empty result set in Chinook: Award full credit if the query is logically correct. If a student submits a query that filters to only a subset of customers 'to prove it works,' note the misunderstanding.

Question 8

Common Student Mistakes

- Placing the CASE WHEN expression in the WHERE clause instead of the SELECT clause
- Using CASE WHEN on individual Invoice.Total rows instead of on the aggregated SUM, leading to row-level classification before grouping
- Getting the boundary conditions wrong — using > 100 and > 50 without ensuring 'Medium' correctly captures the 50–100 range (forgetting WHEN SUM >= 50 after the first condition already excludes > 100)
- Not using ROUND() on TotalRevenue, or applying ROUND inside the CASE WHEN instead of in the SELECT
- Referencing TotalRevenue alias inside the CASE WHEN expression (not valid in most SQL engines — must repeat SUM(Invoice.Total))
- Forgetting to GROUP BY BillingCountry before applying CASE WHEN

Socratic Questions (When Students Are Stuck)

- If a student is stuck on CASE WHEN placement: Where in the query do you compute the SUM per country — in SELECT or WHERE? If CASE WHEN needs to evaluate the SUM, where should the CASE WHEN expression live?
- If a student applies CASE WHEN to individual rows: Think about when CASE WHEN runs versus when GROUP BY runs. If you want to classify the total for a whole country, does the classification happen before or after the rows are grouped?
- If a student gets boundary conditions wrong: If the first WHEN says SUM > 100, and the second says SUM > 50, what range does the second WHEN actually cover at that point in the CASE statement? Does SQL evaluate CASE conditions in order, stopping at the first true one?

Solution Walkthrough

Step 1: Only chinook.Invoice is needed. Step 2: GROUP BY Invoice.BillingCountry — one row per country. Step 3: SELECT BillingCountry, COUNT(Invoice.InvoiceId) AS InvoiceCount, ROUND(SUM(Invoice.Total), 2) AS TotalRevenue. Step 4: Write CASE WHEN expression operating on SUM(Invoice.Total) — the aggregate, not individual rows: CASE WHEN SUM(Invoice.Total) > 100 THEN 'High' WHEN SUM(Invoice.Total) >= 50 THEN 'Medium' ELSE 'Low' END AS RevenueTier. Note: because CASE evaluates top-to-bottom and stops at first match, the second condition WHEN SUM >= 50 implicitly means 50 <= SUM <= 100. Step 5: ORDER BY TotalRevenue DESC. Final query:

SELECT Invoice.BillingCountry, COUNT(Invoice.InvoiceId) AS InvoiceCount, ROUND(SUM(Invoice.Total), 2) AS TotalRevenue, CASE WHEN SUM(Invoice.Total) > 100 THEN 'High' WHEN SUM(Invoice.Total) >= 50 THEN 'Medium' ELSE 'Low' END AS RevenueTier FROM chinook.Invoice GROUP BY Invoice.BillingCountry ORDER BY TotalRevenue DESC;

Grading Tips

Award full 20 points for correct GROUP BY, correct COUNT and SUM aggregation, correct ROUND, correct CASE WHEN with proper boundary conditions, and correct ORDER BY. Deduct 5 points for CASE WHEN applied to non-aggregated Invoice.Total instead of SUM(Invoice.Total). Deduct 3 points for incorrect boundary conditions in CASE WHEN (e.g., both thresholds use > instead of the correct mix of > and >=). Deduct 2 points for missing ROUND. Deduct 2 points for missing ORDER BY. Award 12/20 if GROUP BY and aggregation are correct but CASE WHEN is missing entirely. Award 8/20 if the student correctly groups and counts but applies CASE WHEN at the wrong level.

Discussion Prompt

CASE WHEN evaluates conditions in order and stops at the first match — this is called 'short-circuit evaluation.' How does this behavior affect how you write your conditions? Would the query produce different results if you swapped the order of the WHEN clauses (putting the >= 50 condition first)?

Edge Cases

- Boundary condition for 'Medium': The question says '$50 to $100 inclusive' — students must use >= 50 AND the upper bound is handled by the first WHEN (> 100 already excluded). Students who write WHEN SUM BETWEEN 50 AND 100 THEN 'Medium' are also correct if they handle the 'High' condition first.
- Students who reference TotalRevenue alias in CASE WHEN: This fails in standard SQL (and most engines). If the student notes this as an intentional alias reuse and it errors, guide them to repeat the expression.
- Students who use ORDER BY 3 DESC (positional reference): Valid SQL, but note it as fragile practice; do not penalize.
- Students who round inside the CASE WHEN on SUM: e.g., WHEN ROUND(SUM(...), 2) > 100 — functionally equivalent; award full credit.