Instructor Guide: Cross-Grain Brewing — Distribution Analysis

CONFIDENTIAL — FOR INSTRUCTOR USE ONLY

Assignment Overview

This assignment covers foundational to advanced SQL querying against the craftBrewery relational database, which models a small craft brewery's operations including beers, styles, batches, taproom sales, wholesale accounts, orders, and territories. Students are expected to demonstrate mastery of SELECT with filtering and sorting (Q1), INNER JOIN across two tables (Q2), LEFT JOIN with aggregation for zero-count inclusion (Q3), WHERE+GROUP BY+HAVING for filtered aggregation (Q4), multi-table join chains with aggregation and status filtering (Q5), correlated subquery with IN (Q6), conditional aggregation using CASE WHEN inside SUM for pivoting (Q7), and scalar subquery inside HAVING for cross-group comparison (Q8). Grading should reward correct logic and result accuracy, with meaningful partial credit for syntactically sound but logically incomplete attempts.

Partial Credit Policy

Partial credit should reflect how close the student's logic is to the correct solution, not just whether results match. A query that uses the right technique with a minor error (wrong column name, missing ORDER BY, off-by-one in HAVING threshold) should receive 70–85% of points. A query that demonstrates the right approach but has a fundamental logical flaw (wrong join type, missing required filter, wrong aggregate) should receive 40–60%. A query that queries the right tables but uses incorrect technique (e.g., JOIN instead of required subquery) should receive 30–50%. No credit should be awarded for queries that do not execute or that query entirely wrong tables. Syntactically valid queries that return zero rows due to a filter error should receive no more than 40% — the output is meaningless even if the structure is partially right.

General Grading Tips

- Run the solution SQL against the actual database before grading each question to verify expected row counts and result hashes — data drift from earlier sessions can sometimes change expected results.
- Use the result_hash values to do a quick programmatic check of student submissions if your platform supports it; manual row-count verification is a good fallback.
- Watch for students who get the correct row count by accident through compensating errors (e.g., wrong join type but also wrong filter that cancels out). Always review the SQL logic, not just the output.

- Encourage students to run subqueries in isolation before embedding them — this is a debugging habit worth reinforcing in feedback.
- Be consistent about column alias requirements — if the question specifies column names (e.g., totalOrders, avgUnitPrice), deduct 1 point for missing or misnamed aliases that would cause the result hash to differ.
- For hard questions (Q6, Q7, Q8), consider awarding a 'technique demonstration' point even if the result is wrong but the required technique (subquery, CASE WHEN, scalar HAVING) is clearly and correctly attempted.

Question-by-Question Guide

Question 1

Common Student Mistakes

- Using isActive = true or isActive = 'true' instead of isActive = 1 (tinyint comparison)
- Selecting extra columns not asked for (e.g., beerId, isActive) which may not affect logic but shows inattention to requirements
- Ordering by ABV ascending instead of descending
- Omitting the WHERE clause entirely and returning all beers including inactive ones
- Using SELECT * instead of specifying the three required columns

Socratic Questions (When Students Are Stuck)

- If a student is stuck on filtering: What does the isActive column represent, and what value indicates a beer is currently active according to the schema?
- If a student is stuck on ordering: The question asks for highest ABV first — which ORDER BY keyword produces that arrangement?
- If a student returns too many rows: How many beers did you get back? How could you verify that only active beers are included in your result?

Solution Walkthrough

1. Identify the target table: craftbrewery.beers contains all required columns (beerName, abv, ibu, isActive). 2. Write SELECT specifying exactly beerName, abv, ibu. 3. Add WHERE beers.isActive = 1 to filter out inactive beers. 4. Add ORDER BY beers.abv DESC to sort highest ABV first. Final query: SELECT beers.beerName, beers.abv, beers.ibu FROM craftbrewery.beers WHERE beers.isActive = 1 ORDER BY beers.abv DESC; Expected: 77 rows.

Grading Tips

Award full 8 points for correct columns, correct filter, and correct descending order. Deduct 2 points if ORDER BY is ASC instead of DESC. Deduct 2 points if WHERE clause is missing (returns inactive beers). Deduct 1 point if extra unrequested columns are selected but logic is otherwise correct. Award 5/8 if columns and filter are correct but ordering is entirely absent.

Discussion Prompt

Why might a brewery store isActive as a tinyint (0/1) rather than a boolean or a varchar like 'Yes'/'No'? What are the trade-offs in terms of storage, readability, and portability across database systems?

Edge Cases

- If a student uses WHERE isActive = TRUE — many SQL dialects accept this and it evaluates correctly for tinyint; accept as correct if results match the expected 77 rows.
- Ties in ABV: if two beers share the same ABV value, their relative order is non-deterministic. Do not penalize for differing sub-order among tied ABV values unless a secondary sort was explicitly required.

Question 2

Common Student Mistakes

- Using LEFT JOIN instead of INNER JOIN (or JOIN), which may include beers with no matching style — contradicts the 'only include beers that have a matching style' requirement
- Joining on the wrong column (e.g., beers.beerId = styles.styleId instead of beers.styleId = styles.styleId)
- Reversing the ORDER BY — ordering by beerName first, then styleName second
- Forgetting the secondary sort on beerName (only sorting by styleName)
- Selecting styles.styleId instead of styles.description, or omitting description entirely

Socratic Questions (When Students Are Stuck)

- If a student used LEFT JOIN: The question says 'only include beers that have a matching style' — which type of JOIN returns only rows that match on both sides?
- If a student got the wrong row count: How many total beers are in the beers table? How many styles exist? Would every beer necessarily have a matching style, and what does that imply for the join type?
- If a student has incorrect ORDER BY: The question lists style name first, then beer name — how does that translate to the ORDER BY clause?

Solution Walkthrough

1. Both beers and styles tables are needed. 2. The join key is beers.styleId = styles.styleId (FK relationship). 3. Use INNER JOIN (or JOIN) because only beers with a matching style should appear. 4. SELECT beers.beerName, styles.styleName, styles.description. 5. ORDER BY styles.styleName ASC, beers.beerName ASC — style name is the primary sort, beer name is the tiebreaker. Final query: SELECT beers.beerName, styles.styleName, styles.description FROM craftbrewery.beers JOIN craftbrewery.styles ON beers.styleId = styles.styleId ORDER BY styles.styleName ASC, beers.beerName ASC; Expected: 260 rows.

Grading Tips

Full 11 points for correct join type, correct ON clause, correct columns, and correct two-level ORDER BY. Deduct 3 points for using LEFT JOIN (returns extra rows). Deduct 2 points if ORDER BY has only one level or is reversed. Deduct 1 point for wrong join key. Award 6/11 if the correct tables and intent are shown but join condition and ordering both have errors.

Discussion Prompt

In this brewery context, would it ever make sense to have a beer with no associated style? How does the database design (FK constraint) reflect a business rule, and how does JOIN type selection enforce that rule at query time?

Edge Cases

- If the database has beers with NULL styleId or styles with no associated beers, a LEFT JOIN from beers to styles would still return the same 260 rows only if all beers have a matching style. If a student uses LEFT JOIN and gets 260 rows, the logic is technically wrong but the result matches — deduct 2 points for incorrect join type regardless of row count match.
- ASC keyword is optional in ORDER BY and should not be penalized if omitted.

Question 3

Common Student Mistakes

- Using INNER JOIN instead of LEFT JOIN, which excludes accounts that have never placed an order
- Using COUNT(*) instead of COUNT(orders.orderId) — with a LEFT JOIN, COUNT(*) will return 1 for accounts with no orders instead of 0 because the row still exists
- Grouping only by accountName without also grouping by accountId, which can cause ambiguity or errors if two accounts share a name
- Forgetting the GROUP BY clause entirely and getting an aggregation error or incorrect result
- Ordering ascending instead of descending

Socratic Questions (When Students Are Stuck)

- If a student used INNER JOIN and got fewer than 260 rows: The question says 'include accounts that have never placed an order' — what does an INNER JOIN do to rows that don't have a match on the right side?
- If a student used COUNT(*) and got 1 instead of 0 for zero-order accounts: When there is no matching order row (NULL from the LEFT JOIN), what does COUNT(*) count versus COUNT(orders.orderId)?
- If a student is missing GROUP BY: You're trying to count orders per account — what clause tells SQL how to form the groups before aggregating?

Solution Walkthrough

1. Start from accounts (the 'many' side we want to preserve). 2. LEFT JOIN orders ON accounts.accountId = orders.accountId to keep accounts with no orders. 3. Use COUNT(orders.orderId) — this correctly returns 0 when orderId is NULL (no orders). COUNT(*) would incorrectly return 1. 4. GROUP BY accounts.accountId, accounts.accountName, accounts.accountType — include accountId to avoid name collision issues. 5. ORDER BY totalOrders DESC. Final query: SELECT accounts.accountName, accounts.accountType, COUNT(orders.orderId) AS totalOrders FROM craftbrewery.accounts LEFT JOIN craftbrewery.orders ON accounts.accountId = orders.accountId GROUP BY accounts.accountId, accounts.accountName, accounts.accountType ORDER BY totalOrders DESC; Expected: 260 rows.

Grading Tips

Full 11 points for correct LEFT JOIN, COUNT(orders.orderId), proper GROUP BY, and correct ORDER BY. Deduct 4 points for INNER JOIN (fundamentally wrong — excludes zero-order accounts). Deduct 3 points for COUNT(*) with LEFT JOIN (zero-order accounts show 1 instead of 0). Deduct 1 point for missing accountId in GROUP BY if database still runs but is technically ambiguous. Award partial credit of 6/11 for correct structure with both COUNT and JOIN type wrong.

Discussion Prompt

Why is COUNT(column) different from COUNT(*) specifically in the context of outer joins? Can you think of other aggregate functions where NULL handling from a LEFT JOIN could lead to subtly wrong results?

Edge Cases

- COUNT(DISTINCT orders.orderId) is also acceptable and produces the same result assuming no duplicate orderId entries per account join; award full credit.
- Students who group only by accountName — if accountName is guaranteed unique in the data, results may still be correct; award full credit only if the query is logically sound (i.e., they include enough columns to disambiguate groups).

Question 4

Common Student Mistakes

- Putting the isActive = 1 filter in the HAVING clause instead of WHERE (logically incorrect — filters should happen before grouping when applied to row-level attributes)
- Using HAVING COUNT(*) > 5 without the WHERE isActive = 1 filter, counting all beers (active and inactive) per style
- Using WHERE COUNT(beers.beerId) > 5 — aggregate functions cannot appear in a WHERE clause
- Forgetting to join styles and beers, attempting to query from one table only
- Using LEFT JOIN, which would include styles with no beers (showing 0 counts incorrectly)

Socratic Questions (When Students Are Stuck)

- If a student put isActive in HAVING: At what stage of query execution does WHERE filter rows versus when does HAVING filter groups? Which is more efficient for a row-level condition?
- If a student used HAVING COUNT > 5 without the WHERE filter: Can you trace through a style that has 4 active and 3 inactive beers — would your query include or exclude it? Is that the intended behavior?
- If a student is confused about WHERE vs HAVING for aggregates: Can you put an aggregate function like COUNT() inside a WHERE clause? Why or why not?

Solution Walkthrough

1. Join styles to beers on styleId. Use INNER JOIN because we only care about styles that have active beers. 2. Add WHERE beers.isActive = 1 to filter to active beers BEFORE grouping. 3. GROUP BY styles.styleId, styles.styleName. 4. Add HAVING COUNT(beers.beerId) > 5 to keep only styles with more than 5 active beers. 5. SELECT styles.styleName and COUNT(beers.beerId) AS activeBeerCount. 6. ORDER BY activeBeerCount DESC. Final query: SELECT styles.styleName, COUNT(beers.beerId) AS activeBeerCount FROM craftbrewery.styles JOIN craftbrewery.beers ON styles.styleId = beers.styleId WHERE beers.isActive = 1 GROUP BY styles.styleId, styles.styleName

HAVING COUNT(beers.beerId) > 5 ORDER BY activeBeerCount DESC; Expected: 9 rows.

Grading Tips

Full 12 points for correct JOIN, WHERE for isActive, GROUP BY, HAVING threshold, and ORDER BY. Deduct 3 points if WHERE isActive = 1 is in HAVING instead — logic is wrong even if results accidentally match on this dataset. Deduct 4 points if isActive filter is missing entirely. Deduct 1 point if styleId is missing from GROUP BY but query runs. Award 7/12 for correct JOIN and HAVING structure but wrong filter placement or missing isActive filter.

Discussion Prompt

In SQL, the logical execution order is: FROM → JOIN → WHERE → GROUP BY → HAVING → SELECT → ORDER BY. How does understanding this order help you decide whether to place a condition in WHERE versus HAVING? Are there performance implications?

Edge Cases

- HAVING COUNT(beers.beerId) > 5 vs HAVING COUNT(*) > 5: After the WHERE filter, COUNT(*) and COUNT(beers.beerId) produce the same result since all grouped rows are non-null active beers. Accept either.
- Students who use >= 6 instead of > 5 produce the same result for integer counts; award full credit.

Question 5

Common Student Mistakes

- Missing one of the intermediate joins (e.g., skipping the accounts table and trying to join territories directly to orders)
- Filtering by orders.status in HAVING instead of WHERE
- Forgetting the WHERE orders.status = 'Delivered' filter entirely, summing revenue from all order statuses
- Using SUM(orders.lineTotal) — lineTotal is on orderLines, not orders
- Misspelling 'Delivered' (case sensitivity may matter depending on database collation)
- Not grouping by territoryId, causing incorrect aggregation when territory names might not be unique

Socratic Questions (When Students Are Stuck)

- If a student is missing a join: Walk me through the path from territories to orderLines in the schema — which tables sit between them, and how are they linked?
- If a student forgot the status filter: The question says 'only include orders with a status of Delivered' — where in your query do you enforce that condition, and at what point in execution does it apply?
- If a student is using the wrong table for lineTotal: Which table in the schema contains the lineTotal column? Where does revenue at the line item level live?

Solution Walkthrough

1. The join chain must be: territories → accounts (on territoryId) → orders (on accountId) → orderLines (on orderId). 2. Add WHERE orders.status = 'Delivered' to restrict to delivered orders only. 3. SELECT

territories.territoryName, territories.salesRep, SUM(orderLines.lineTotal) AS totalRevenue. 4. GROUP BY territories.territoryId, territories.territoryName, territories.salesRep. 5. ORDER BY totalRevenue DESC. Final query: SELECT territories.territoryName, territories.salesRep, SUM(orderLines.lineTotal) AS totalRevenue FROM craftbrewery.territories JOIN craftbrewery.accounts ON territories.territoryId = accounts.territoryId JOIN craftbrewery.orders ON accounts.accountId = orders.accountId JOIN craftbrewery.orderLines ON orders.orderId = orderLines.orderId WHERE orders.status = 'Delivered' GROUP BY territories.territoryId, territories.territoryName, territories.salesRep ORDER BY totalRevenue DESC; Expected: 12 rows.

Grading Tips

Full 12 points for complete join chain, correct WHERE filter, correct SUM target, and correct GROUP BY and ORDER BY. Deduct 4 points for missing the status filter. Deduct 3 points for a broken join chain (missing one table). Deduct 2 points for SUM on wrong column. Deduct 1 point for missing territoryId in GROUP BY. Award 6/12 for correct intent and partial join chain with missing filter.

Discussion Prompt

This query requires chaining four tables. How does the order in which you write joins affect readability versus performance? Does SQL's query optimizer rearrange join order, and if so, why might a developer still care about the written order?

Edge Cases

- Territories that have accounts but no 'Delivered' orders will be excluded — this is correct per the question requirements, but some students may question this. Confirm that INNER JOINs throughout are appropriate here since we only want territories with revenue.
- If a student uses LEFT JOINs throughout and gets 12 rows with the WHERE filter, the WHERE implicitly converts the outer joins to inner joins — accept if results match.

Question 6

Common Student Mistakes

- Using a JOIN instead of a subquery with IN — question explicitly requires a subquery approach
- Writing the subquery against the beers table instead of beerTags (the tag data is in beerTags)
- Forgetting WHERE beerTags.tagName = 'Hoppy' inside the subquery, returning all beer IDs
- Misspelling 'Hoppy' (check case sensitivity of the data)
- Ordering by brewDate DESC instead of ASC
- Selecting columns from beerTags in the outer query (e.g., including tagName), which is not accessible without a join

Socratic Questions (When Students Are Stuck)

- If a student used a JOIN: The question specifically asks you to use a subquery with IN — can you rewrite the JOIN version using a WHERE beerId IN (SELECT ...) pattern instead?
- If a student's subquery is returning wrong results: What does your subquery return when you run it by itself? Is it returning a list of beerIds for beers tagged 'Hoppy'?
- If a student is confused about what to put inside IN: The outer query needs to filter batches by beerId — what list of values does the beerId column in batches need to be checked against?

Solution Walkthrough

1. The outer query selects from batches: batchCode, brewDate, volumeLitres, status. 2. The filter is WHERE batches.beerId IN (...). 3. The subquery selects beerTags.beerId FROM craftbrewery.beerTags WHERE beerTags.tagName = 'Hoppy'. 4. This returns all beerIds associated with the 'Hoppy' tag. 5. ORDER BY batches.brewDate ASC. Final query: SELECT batches.batchCode, batches.brewDate, batches.volumeLitres, batches.status FROM craftbrewery.batches WHERE batches.beerId IN (SELECT beerTags.beerId FROM craftbrewery.beerTags WHERE beerTags.tagName = 'Hoppy') ORDER BY batches.brewDate ASC; Expected: 551 rows.

Grading Tips

Full 15 points for correct subquery structure, correct tagName filter inside subquery, correct outer columns, and correct ORDER BY. Deduct 4 points for using a JOIN instead of IN subquery (technique not demonstrated). Deduct 3 points if subquery is missing the tagName = 'Hoppy' filter. Deduct 2 points for wrong ORDER BY direction. Award 10/15 for correct IN subquery structure with tagName filter missing or misspelled.

Discussion Prompt

This query could also be written as a JOIN between batches and beerTags. What are the advantages and disadvantages of each approach (IN subquery vs JOIN)? Are there cases where one significantly outperforms the other?

Edge Cases

- A beer may have the 'Hoppy' tag multiple times in beerTags (if no UNIQUE constraint). The subquery with IN handles duplicates gracefully (IN doesn't care about duplicates in the list). A JOIN without DISTINCT would produce duplicate batch rows — this is a reason the subquery approach is safer here.
- Case sensitivity: if 'Hoppy' vs 'hoppy' differs in the database, a student using 'hoppy' would return 0 rows. Check the actual data casing before penalizing.

Question 7

Common Student Mistakes

- Using CASE WHEN inside SUM but forgetting the ELSE 0, causing NULL propagation and incorrect totals
- Placing CASE WHEN outside SUM (e.g., SELECT CASE WHEN ... THEN SUM(...)) which is syntactically invalid
- Misspelling serving type values — 'Growler Fill', 'Crowler', 'Half Pint', 'Flight' must match exactly
- Using AVG instead of SUM inside the CASE expressions
- Using GROUP BY beerName only (without beerId), causing issues if two beers share a name
- Forgetting the JOIN between beers and taproomSales, querying only one table

Socratic Questions (When Students Are Stuck)

- If a student has NULL in totals: What happens when a CASE WHEN condition is not met and there is no ELSE clause? What value does SQL use by default, and how does that affect SUM?

- If a student is trying to write the CASE outside SUM: Can you aggregate a CASE expression directly? What if you think of it as: for each row, CASE produces a value (or 0), and SUM adds those values up across the group?
- If a student is missing serving types in a category: Let's look at the question again — which serving types belong to 'Small'? Make sure your IN list matches those exactly.

Solution Walkthrough

1. Join beers to taproomSales on beerId. 2. GROUP BY beers.beerId, beers.beerName — one row per beer. 3. For smallTotal: SUM(CASE WHEN taproomSales.servingType IN ('Flight', 'Half Pint') THEN taproomSales.saleAmount ELSE 0 END). 4. For standardTotal: SUM(CASE WHEN taproomSales.servingType = 'Pint' THEN taproomSales.saleAmount ELSE 0 END). 5. For largeTotal: SUM(CASE WHEN taproomSales.servingType IN ('Growler Fill', 'Crowler') THEN taproomSales.saleAmount ELSE 0 END). 6. ORDER BY beers.beerName ASC. The INNER JOIN ensures only beers that have taproom sales appear. Final query as specified in solution_sql. Expected: 234 rows.

Grading Tips

Full 16 points for correct JOIN, correct three CASE WHEN expressions with ELSE 0, correct GROUP BY, and correct ORDER BY. Deduct 3 points for missing ELSE 0 in any CASE (produces NULLs in totals). Deduct 2 points per category with wrong serving type mapping. Deduct 2 points for grouping by beerName only. Deduct 3 points for using AVG instead of SUM. Award 10/16 for correct structure with one or two serving type mismatches.

Discussion Prompt

The CASE WHEN inside SUM pattern is often called 'conditional aggregation' or a 'pivot.' SQL has no native PIVOT syntax in MySQL. When would you choose conditional aggregation in SQL versus pivoting the data in application code or a BI tool? What are the scalability trade-offs?

Edge Cases

- A beer with taproom sales but none of the named serving types would appear in results with all three totals as 0 (due to ELSE 0). This is acceptable and expected behavior.
- If a student uses SUM(CASE WHEN ... THEN saleAmount END) without ELSE — MySQL treats missing ELSE as ELSE NULL. SUM ignores NULLs, so if a beer has ANY matching rows, the sum is correct; if ALL rows are in a different category, the column will be NULL not 0. Deduct 2 points for this because results will differ from expected.

Question 8

Common Student Mistakes

- Writing the global average as a separate query instead of embedding it as a scalar subquery in HAVING
- Using WHERE AVG(...) > (...) — aggregate functions cannot be used in WHERE
- Not joining all three required tables (accounts, orders, orderLines), trying to reach orderLines from accounts directly
- Forgetting ROUND(..., 2) on the displayed average unit price

• Grouping only by accountName without accountId, risking collisions on same-named accounts

• Using the HAVING value (unrounded AVG) but displaying it rounded, leading to inconsistency between filter and display — this is actually correct behavior and should not be penalized

Socratic Questions (When Students Are Stuck)

• If a student is trying to use WHERE with an aggregate: Can SQL evaluate AVG() during the WHERE phase? What clause is designed specifically to filter on aggregate results?

• If a student wrote the global average as a separate hardcoded number: What if the data changes — would your query still be correct? How could you make the global average dynamic so it recalculates automatically?

• If a student is confused about where the subquery goes: The HAVING clause compares each account's average to another value — where exactly in the HAVING clause would you place the expression that computes the overall average?

Solution Walkthrough

1. Join accounts → orders (on accountId) → orderLines (on orderId). 2. GROUP BY accounts.accountId, accounts.accountName, accounts.city, accounts.accountType. 3. SELECT accounts.accountName, accounts.city, accounts.accountType, ROUND(AVG(orderLines.unitPrice), 2) AS avgUnitPrice. 4. In HAVING: AVG(orderLines.unitPrice) > (SELECT AVG(orderLines.unitPrice) FROM craftbrewery.orderLines) — the scalar subquery computes the global average across ALL order lines. Note: use the unrounded AVG in HAVING for precision. 5. ORDER BY avgUnitPrice DESC. Final query as in solution_sql. Expected: 90 rows.

Grading Tips

Full 15 points for correct three-table join, correct GROUP BY with accountId, scalar subquery in HAVING, ROUND on display, and ORDER BY. Deduct 4 points for hardcoding the global average as a literal number. Deduct 4 points for missing the HAVING altogether or using WHERE for the aggregate comparison. Deduct 2 points for missing ROUND. Deduct 1 point for missing accountId in GROUP BY. Award 8/15 for correct join chain and HAVING structure but missing scalar subquery (uses hardcoded or no global average).

Discussion Prompt

A scalar subquery in HAVING is evaluated once per group (or optimized to run once total by most engines). How does this differ from a correlated subquery? Could this query be rewritten using a CTE or a join to a derived table for the global average? What might be the readability or performance trade-offs?

Edge Cases

• Accounts with no order lines will not appear due to INNER JOIN — this is correct and acceptable since there is no average to compare.

• The scalar subquery computes the global average including ALL accounts, even those that will be filtered out — this is the standard interpretation and is correct.

• ROUND in HAVING vs HAVING without ROUND: using ROUND(AVG(...), 2) in the HAVING comparison is technically valid but introduces rounding in the comparison itself, which can cause borderline accounts to shift in or out. Award full credit either way as long as the subquery comparison is present.