Instructor Guide: Pedal Works — Service and Sales Analysis

CONFIDENTIAL — FOR INSTRUCTOR USE ONLY

Assignment Overview

This assignment assesses students' ability to query a multi-table relational database (bikeShop) using progressively complex SQL techniques. Questions 1–4 test foundational SELECT, WHERE, LIKE, and ORDER BY skills on single tables. Questions 5–7 introduce JOINs, GROUP BY, aggregate functions, and HAVING clauses. Question 8, worth 32 points, is the capstone: students must chain four tables, apply COUNT(DISTINCT ...) to avoid row inflation, filter with HAVING, and sort aggregated results. By the end, students should demonstrate comfort with filter logic, multi-table joins, aggregate correctness, and the WHERE-vs-HAVING distinction.

Partial Credit Policy

Partial credit is awarded when a student demonstrates understanding of the core concept but makes a recoverable error. In general: correct table selection and join logic earns 50–60% of points even if filters or aggregates are wrong. A correct filter/aggregate with a wrong join earns 40–50%. A structurally correct query with only a sort direction or alias error earns 85–90%. Queries that produce 0 rows due solely to a string literal case mismatch (e.g., 'green bay' vs 'Green Bay') should receive 70–80% credit if all other logic is sound, with a written note to the student. No credit is awarded for queries that return all rows from a table without any meaningful filtering or aggregation when the question requires it. For Question 8 specifically, use the point-breakdown rubric in its grading_tips to award granular partial credit across the five key components.

General Grading Tips

- Run the result_hash verification first: compare student query output hashes against the provided hashes before reading the code — this quickly separates fully correct submissions from those needing partial credit review.
- Check column names and aliases in student output: questions specify exact output column names (e.g., totalRepairOrders, totalRevenue) — minor alias differences (e.g., 'total_revenue' vs 'totalRevenue') should receive only a cosmetic 1-point deduction, not a full deduction.
- For Questions 5–8, watch for students who write technically valid SQL with the right output but use implicit join syntax (comma-separated tables in FROM with WHERE conditions) — verify correctness but note that explicit JOIN syntax is the expected standard for this course level.
- Question 8 is worth 40% of total points — spend proportionally more time reviewing it and consider running a second check pass on borderline submissions.

- If a student's query returns the right rows but in a different order than expected (e.g., ties in ORDER BY resolved differently), do not penalize — order within ties is implementation-dependent.
- Encourage students to verify row counts against expected counts listed in the assignment — a simple sanity check they should perform before submitting.

Question-by-Question Guide

Question 1

Common Student Mistakes

- Using city = 'green bay' (wrong case) and getting 0 rows if the DB is case-sensitive
- Forgetting ORDER BY or ordering by firstName instead of lastName
- Selecting extra columns like customerId or city in the output when only three were asked for
- Using SELECT * instead of specifying columns
- Omitting the schema prefix 'bikeshop.' if it is required in the course environment

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: Which table stores information about customers and where they live?
- If a student is stuck, ask: How do you tell SQL to only return rows where a specific column matches a given value?
- If a student is stuck, ask: Once you have the right rows, how do you control the order in which they appear?

Solution Walkthrough

1. Identify the source table: bikeshop.customers contains firstName, lastName, email, and city. 2. Write SELECT customers.firstName, customers.lastName, customers.email to project only the requested columns. 3. Add FROM bikeshop.customers. 4. Add WHERE customers.city = 'Green Bay' to restrict rows to Green Bay residents — note the exact capitalization. 5. Add ORDER BY customers.lastName ASC to sort alphabetically by last name. Final query: SELECT customers.firstName, customers.lastName, customers.email FROM bikeshop.customers WHERE customers.city = 'Green Bay' ORDER BY customers.lastName ASC; Expected: 8 rows.

Grading Tips

Award full 8 points for a correct result set (8 rows, 3 columns, correct order). Deduct 2 points if ORDER BY is missing or wrong direction. Deduct 2 points if extra columns appear in output. Deduct 1 point for cosmetic issues like missing schema prefix if the student's environment does not require it but the instructor's rubric does. Give 0 points if WHERE clause is missing entirely (wrong data returned).

Discussion Prompt

Why might a production database store city names in a normalized lookup table (e.g., a cities table with a foreign key) rather than a plain varchar column, and how would that change this query?

Edge Cases

- Case sensitivity: If the student writes 'green bay' or 'GREEN BAY' and gets 0 rows, clarify that string literals must match stored data exactly in most SQL engines unless LOWER()/ILIKE is used

— award full credit if the logic is correct and only the literal case is off, but note the error.

• Students who include city in the SELECT list: the column was used for filtering, not output — deduct 1–2 points per rubric but the logic is otherwise sound.

Question 2

Common Student Mistakes

• Ordering ASC instead of DESC for purchase price

• Filtering on bikeType = 'mountain' (wrong case) yielding 0 rows

• Selecting all columns with SELECT * rather than the four specified

• Confusing bikeType with brand or model column names

Socratic Questions (When Students Are Stuck)

• If a student is stuck, ask: Which column in the bikes table tells you what type of bike it is?

• If a student is stuck, ask: You want the most expensive bikes first — which keyword controls ascending vs. descending order?

• If a student is stuck, ask: How many columns does the question ask you to display, and can you list them by name?

Solution Walkthrough

1. Source table: bikeshop.bikes. 2. SELECT bikes.brand, bikes.model, bikes.bikeType, bikes.purchasePrice. 3. FROM bikeshop.bikes. 4. WHERE bikes.bikeType = 'Mountain' — exact capitalization required. 5. ORDER BY bikes.purchasePrice DESC for highest to lowest. Final query: SELECT bikes.brand, bikes.model, bikes.bikeType, bikes.purchasePrice FROM bikeshop.bikes WHERE bikes.bikeType = 'Mountain' ORDER BY bikes.purchasePrice DESC; Expected: 59 rows.

Grading Tips

Full 8 points for correct 59-row result with DESC ordering. Deduct 2 points for wrong sort direction (ASC). Deduct 2 points for missing ORDER BY entirely. Deduct 1–2 points for extra or missing columns in SELECT. If 0 rows are returned due to case mismatch on 'Mountain', award partial credit (3–4 points) if the logic structure is otherwise correct.

Discussion Prompt

If the database contained 'mountain', 'Mountain', and 'MOUNTAIN' as different bikeType values, how would you write a case-insensitive filter in standard SQL? Does this differ between MySQL, PostgreSQL, and SQL Server?

Edge Cases

• Students who use LIKE '%Mountain%' instead of = 'Mountain': technically returns the same rows if no other bikeType contains the substring, so accept it — but note it is less precise and deduct 1 point.

• Ties in purchasePrice: rows with identical prices may appear in any order relative to each other — do not penalize for secondary ordering differences.

Question 3

Common Student Mistakes

- Writing LIKE 'bundle' without wildcards, matching only exact value 'bundle'
- Using LIKE 'bundle%' or LIKE '%bundle' instead of '%bundle%', missing matches where 'bundle' is in the middle
- Forgetting ORDER BY unitCost ASC or reversing to DESC
- Using = 'bundle' instead of LIKE
- Case-sensitivity issues: writing LIKE '%Bundle%' and missing lowercase entries (depends on DB collation)

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: When you want to match a word that could appear anywhere within a longer string, which SQL operator do you use?
- If a student is stuck, ask: What does the percent sign (%) mean as a wildcard in a LIKE pattern?
- If a student is stuck, ask: If 'bundle' could appear at the beginning, middle, or end of the partName, where do you need to place your wildcards?

Solution Walkthrough

1. Source table: bikeshop.parts. 2. SELECT parts.partName, parts.partNumber, parts.category, parts.unitCost. 3. FROM bikeshop.parts. 4. WHERE parts.partName LIKE '%bundle%' — the wildcards on both sides ensure any partName containing the substring 'bundle' (anywhere) is matched. The question states case-insensitive is acceptable, so '%bundle%' is fine if the collation is case-insensitive; otherwise use LOWER(parts.partName) LIKE '%bundle%'. 5. ORDER BY parts.unitCost ASC. Final query: SELECT parts.partName, parts.partNumber, parts.category, parts.unitCost FROM bikeshop.parts WHERE parts.partName LIKE '%bundle%' ORDER BY parts.unitCost ASC; Expected: 15 rows.

Grading Tips

Full 8 points for 15 correct rows in ASC order. Deduct 3 points for using LIKE 'bundle%' or LIKE '%bundle' (missing rows). Deduct 4 points for using = 'bundle' (almost certainly 0 rows). Deduct 2 points for wrong sort direction. Award 5–6 points if LIKE pattern is correct but ORDER BY is missing. If the student uses LOWER() or ILIKE for case insensitivity and gets correct rows, award full credit.

Discussion Prompt

How does a full-text index differ from a LIKE '%bundle%' pattern match in terms of performance, and when would you recommend each approach in a production system?

Edge Cases

- If the database collation is case-sensitive and a student uses '%bundle%' but some records store 'Bundle' with a capital B, they may get fewer than 15 rows — check whether the stored data uses consistent casing before penalizing; if the data is all lowercase, '%bundle%' is correct.
- Students using LOWER(partName) LIKE '%bundle%' or ILIKE '%bundle%' (PostgreSQL): accept as correct.

Question 4

Common Student Mistakes

- Filtering role = 'technician' (wrong case) getting 0 rows
- Ordering hourlyRate ASC instead of DESC
- Including storeId or employeeId in the SELECT which were not requested
- Using SELECT * instead of named columns
- Forgetting that the table is 'employees' and querying a non-existent table name

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: Which table contains information about employees, their roles, and their pay rates?
- If a student is stuck, ask: How do you restrict results so only rows with a specific role value are returned?
- If a student is stuck, ask: The question asks for highest to lowest hourly rate — which ORDER BY direction achieves that?

Solution Walkthrough

1. Source table: bikeshop.employees. 2. SELECT employees.firstName, employees.lastName, employees.role, employees.hourlyRate. 3. FROM bikeshop.employees. 4. WHERE employees.role = 'Technician' — exact capitalization. 5. ORDER BY employees.hourlyRate DESC. Final query: SELECT employees.firstName, employees.lastName, employees.role, employees.hourlyRate FROM bikeshop.employees WHERE employees.role = 'Technician' ORDER BY employees.hourlyRate DESC; Expected: 114 rows.

Grading Tips

Full 8 points for 114 correct rows, 4 columns, DESC sort. Deduct 2 points for ASC sort. Deduct 2 points for missing ORDER BY. Deduct 1–2 points for extra columns. Award 3–4 points if WHERE clause is correct but ORDER BY is entirely missing. Give 0 if WHERE is absent (all employees returned).

Discussion Prompt

In a real HR system, what are the security implications of allowing all application users to query employee hourly rates, and how might you use SQL views or row-level security to restrict this?

Edge Cases

- Ties in hourlyRate: multiple technicians at the same rate may appear in any order relative to each other — do not penalize secondary ordering.
- If a student adds a secondary ORDER BY lastName or firstName after hourlyRate, this is harmless and still correct — full credit.

Question 5

Common Student Mistakes

- Using a CROSS JOIN or omitting the ON clause, producing a Cartesian product
- Joining on the wrong keys (e.g., repairOrders.storeId = employees.employeeId)
- Forgetting the WHERE status = 'In Progress' filter
- Using WHERE instead of JOIN syntax (implicit join) — acceptable if result is correct, but note it
- Misspelling 'In Progress' (e.g., 'In progress' or 'InProgress') causing 0 rows

- Ordering by dropOffDate DESC instead of ASC

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: What column in repairOrders links it to the employees table, and what is the matching column in employees?
- If a student is stuck, ask: After you join the two tables, how do you filter so only 'In Progress' orders appear?
- If a student is stuck, ask: The question says ascending by drop-off date — what does that mean for the earliest dates: do they come first or last?

Solution Walkthrough

1. Identify tables: repairOrders (for order info) and employees (for name). 2. Join condition: repairOrders.employeeId = employees.employeeId. 3. SELECT repairOrders.repairOrderId, repairOrders.dropOffDate, repairOrders.status, employees.firstName, employees.lastName. 4. FROM bikeshop.repairOrders JOIN bikeshop.employees ON repairOrders.employeeId = employees.employeeId. 5. WHERE repairOrders.status = 'In Progress'. 6. ORDER BY repairOrders.dropOffDate ASC. Final query: SELECT repairOrders.repairOrderId, repairOrders.dropOffDate, repairOrders.status, employees.firstName, employees.lastName FROM bikeshop.repairOrders JOIN bikeshop.employees ON repairOrders.employeeId = employees.employeeId WHERE repairOrders.status = 'In Progress' ORDER BY repairOrders.dropOffDate ASC; Expected: 250 rows.

Grading Tips

Full 12 points for 250 correct rows with correct columns and ASC date order. Deduct 4 points if WHERE filter is missing (all statuses returned). Deduct 2 points for DESC instead of ASC ordering. Deduct 3 points for wrong JOIN key producing inflated or incorrect rows. Deduct 2 points for missing ORDER BY. Award 6–7 points if JOIN is correct but WHERE filter is absent. Award 8–9 points if everything is correct except sort direction.

Discussion Prompt

What would happen to the result set if an employee was deleted from the employees table but their repairOrders records remained? How could you detect or prevent orphan records like this using database constraints?

Edge Cases

- Students using implicit join syntax (FROM repairOrders, employees WHERE repairOrders.employeeId = employees.employeeId AND repairOrders.status = 'In Progress'): award full credit if result is correct, but note that explicit JOIN syntax is preferred for readability.
- Students who use LEFT JOIN instead of INNER JOIN: because repairOrders has a FK to employees (non-null assumed), result should be identical — award full credit if rows match.

Question 6

Common Student Mistakes

- Using INNER JOIN instead of LEFT JOIN, causing stores with zero repair orders to be excluded
- Forgetting GROUP BY, causing aggregate errors or grouping incorrectly

- Using COUNT(*) instead of COUNT(repairOrders.repairOrderId) — for LEFT JOIN, COUNT(*) would count the NULL row as 1 for stores with no orders
- Grouping only by storeName or city without storeId, which could cause issues if two stores share the same name or city
- Ordering ASC instead of DESC

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: The question says 'include all stores, even those with zero repair orders' — which type of JOIN ensures rows from the left table always appear even when there is no match on the right?
- If a student is stuck, ask: When you want to count how many repair orders belong to each store, what SQL clause do you use to group rows by store?
- If a student is stuck, ask: If a store has no repair orders, the repairOrderId column will be NULL after a LEFT JOIN — what does COUNT(repairOrders.repairOrderId) return for NULL values compared to COUNT(*)?

Solution Walkthrough

1. Tables: stores (always show) LEFT JOIN repairOrders (may have 0 matches). 2. Join condition: stores.storeId = repairOrders.storeId. 3. SELECT stores.storeName, stores.city, COUNT(repairOrders.repairOrderId) AS totalRepairOrders. Using COUNT on the FK column means NULL values (stores with no orders) count as 0. 4. GROUP BY stores.storeId, stores.storeName, stores.city — include storeId to uniquely identify each store. 5. ORDER BY totalRepairOrders DESC. Final query: SELECT stores.storeName, stores.city, COUNT(repairOrders.repairOrderId) AS totalRepairOrders FROM bikeshop.stores LEFT JOIN bikeshop.repairOrders ON stores.storeId = repairOrders.storeId GROUP BY stores.storeId, stores.storeName, stores.city ORDER BY totalRepairOrders DESC; Expected: 12 rows.

Grading Tips

Full 12 points for 12 rows with correct counts, including any zero-count stores, in DESC order. Deduct 4 points for using INNER JOIN (missing zero-order stores). Deduct 3 points for COUNT(*) if it inflates zero-count stores to 1. Deduct 2 points for missing GROUP BY (query likely errors). Deduct 2 points for wrong sort order. Award 7–8 points if JOIN type is wrong but aggregation and grouping are otherwise correct.

Discussion Prompt

Why is it important to COUNT a column from the right (optional) side of a LEFT JOIN rather than using COUNT(*) when you want to count related records? Can you construct a small example that shows the difference in output?

Edge Cases

- If no stores actually have zero repair orders in the data, INNER JOIN and LEFT JOIN produce the same result — verify the data first; if all stores have at least one order, accept INNER JOIN with a note but still require LEFT JOIN conceptually per the question wording.
- Students grouping only by storeName and city without storeId: if all store names are unique, the result is still correct — award full credit but note the best practice of including the PK in GROUP BY.

Question 7

Common Student Mistakes

- Using WHERE COUNT(bikeId) > 30 instead of HAVING — WHERE cannot reference aggregate functions
- Forgetting GROUP BY entirely, causing an error or aggregating all bikes into one row
- Using HAVING brand > 30 or HAVING totalBikes > 30 referencing an alias that the SQL engine doesn't support in HAVING
- Ordering ASC instead of DESC for average purchase price
- Calculating AVG on purchaseYear instead of purchasePrice

Socratic Questions (When Students Are Stuck)

- If a student is stuck, ask: You want to filter based on the COUNT of bikes per brand — does that filter happen before or after grouping, and which SQL clause handles post-grouping filters?
- If a student is stuck, ask: What is the difference between WHERE and HAVING in terms of when they are applied during query execution?
- If a student is stuck, ask: To find the average purchase price per brand, what two SQL elements do you need: one to group rows and one to calculate the average?

Solution Walkthrough

1. Source table: bikeshop.bikes only — no join needed. 2. GROUP BY bikes.brand to create one group per brand. 3. SELECT bikes.brand, COUNT(bikes.bikeId) AS totalBikes, AVG(bikes.purchasePrice) AS avgPurchasePrice. 4. HAVING COUNT(bikes.bikeId) > 30 to keep only brands with more than 30 bikes — this filters groups, not individual rows. 5. ORDER BY avgPurchasePrice DESC. Final query: SELECT bikes.brand, COUNT(bikes.bikeId) AS totalBikes, AVG(bikes.purchasePrice) AS avgPurchasePrice FROM bikeshop.bikes GROUP BY bikes.brand HAVING COUNT(bikes.bikeId) > 30 ORDER BY avgPurchasePrice DESC; Expected: 3 rows.

Grading Tips

Full 12 points for exactly 3 rows with correct brand names, counts, and averages in DESC avg price order. Deduct 5 points for using WHERE instead of HAVING (query may error or return wrong rows). Deduct 3 points for missing HAVING filter (returns all brands). Deduct 2 points for wrong sort direction. Deduct 2 points for missing GROUP BY. Award 6 points if GROUP BY and aggregates are correct but HAVING is missing. Award 8 points if HAVING threshold is correct but sort direction is wrong.

Discussion Prompt

Explain 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 for a given filter?

Edge Cases

- Some SQL engines (MySQL) allow referencing a SELECT alias in HAVING (e.g., HAVING totalBikes > 30) — accept this if it produces the correct result, but note it is not standard SQL.

• Students who use COUNT(*) instead of COUNT(bikes.bikeId): since bikeId is the PK and presumably never NULL, these are equivalent — award full credit.

Question 8

Common Student Mistakes

• Using COUNT(repairOrders.repairOrderId) instead of COUNT(DISTINCT repairOrders.repairOrderId), inflating the count due to multiple repairItems per order

• Stopping the join chain at repairOrders and not joining repairItems, missing the lineTotal column

• Joining in the wrong order or using the wrong FK columns between tables

• Forgetting HAVING SUM(repairItems.lineTotal) > 500, returning all customers

• Using WHERE SUM(...) > 500 instead of HAVING

• Missing GROUP BY on customerId, causing aggregate errors or grouping only by name (risky if duplicate names exist)

• Using SUM(repairOrders.laborHours) as a proxy for revenue instead of SUM(repairItems.lineTotal)

Socratic Questions (When Students Are Stuck)

• If a student is stuck, ask: The data about revenue (lineTotal) lives in repairItems — trace the foreign key path from customers to that table: what intermediate tables do you need to pass through?

• If a student is stuck, ask: Each repair order can have multiple repair items, so if you COUNT repairOrderId without DISTINCT, what problem might that create for your order count?

• If a student is stuck, ask: You want to filter customers based on the sum of their revenue — does that filter apply to individual rows or to groups of rows? Which SQL clause handles that?

Solution Walkthrough

1. Identify the join chain: customers → bikes (customers.customerId = bikes.customerId) → repairOrders (bikes.bikeId = repairOrders.bikeId) → repairItems (repairOrders.repairOrderId = repairItems.repairOrderId). 2. SELECT customers.firstName, customers.lastName, customers.city, COUNT(DISTINCT repairOrders.repairOrderId) AS totalRepairOrders, SUM(repairItems.lineTotal) AS totalRevenue. COUNT(DISTINCT ...) is required because each repair order may have multiple repairItems rows; without DISTINCT the count would be inflated. 3. FROM bikeshop.customers JOIN bikeshop.bikes ON customers.customerId = bikes.customerId JOIN bikeshop.repairOrders ON bikes.bikeId = repairOrders.bikeId JOIN bikeshop.repairItems ON repairOrders.repairOrderId = repairItems.repairOrderId. 4. GROUP BY customers.customerId, customers.firstName, customers.lastName, customers.city — include customerId to uniquely identify each customer even if names collide. 5. HAVING SUM(repairItems.lineTotal) > 500. 6. ORDER BY totalRevenue DESC. Final query: SELECT customers.firstName, customers.lastName, customers.city, COUNT(DISTINCT repairOrders.repairOrderId) AS totalRepairOrders, SUM(repairItems.lineTotal) AS totalRevenue FROM bikeshop.customers JOIN bikeshop.bikes ON customers.customerId = bikes.customerId JOIN bikeshop.repairOrders ON bikes.bikeId = repairOrders.bikeId JOIN bikeshop.repairItems ON repairOrders.repairOrderId = repairItems.repairOrderId GROUP BY customers.customerId,

customers.firstName, customers.lastName, customers.city HAVING SUM(repairItems.lineTotal) > 500 ORDER BY totalRevenue DESC; Expected: 47 rows.

Grading Tips

This question is worth 32 points — distribute partial credit carefully. Suggested breakdown: correct 4-table join chain (10 pts), correct GROUP BY with customerId (5 pts), COUNT(DISTINCT ...) for repair orders (5 pts), SUM(lineTotal) for revenue (5 pts), HAVING with correct threshold (5 pts), ORDER BY DESC (2 pts). Deduct 5 points for missing DISTINCT in COUNT (inflated order counts). Deduct 8 points for stopping at repairOrders and not reaching repairItems (wrong revenue column). Deduct 5 points for using WHERE instead of HAVING. Award 15–18 points for a query that has the right structure but wrong aggregate or missing DISTINCT. Award 20–22 points for a correct result set with a wrong but reasonable column alias.

Discussion Prompt

Why does COUNT(repairOrders.repairOrderId) give a different result than COUNT(DISTINCT repairOrders.repairOrderId) in this context? Describe a real-world business scenario where failing to use DISTINCT would lead to a misleading report for management.

Edge Cases

- Students who produce exactly 47 rows but with slightly wrong totalRepairOrders values (due to missing DISTINCT) while revenue is correct: award partial credit — full points for revenue logic, deduct only the DISTINCT-related points.
- Students who group only by firstName and lastName without customerId: if two customers share the same full name, their data would be merged — deduct 2 points but do not zero-out if the rest is correct.
- Students who include customers with exactly $500.00 revenue (using >= instead of >): deduct 2 points for the threshold error if row count changes.
- Students who use LEFT JOINs throughout and include customers with no repairs (NULL lineTotal, 0 orders): deduct points for incorrect HAVING logic that may include $0 revenue customers.