Data 101/Info 258: Data Engineering
Final Exam Solutions

UC Berkeley, Spring 2025

May 14, 2025

Name: ______________________________

Email: ______________________________@berkeley.edu

Student ID: ______________________________

Examination room: ______________________________

Name of the student on your left: ______________________________

Name of the student on your right: ______________________________

Instructions

Do not open the examination until you are instructed to do so.

This exam consists of **101** ~~99~~ points spread over **5 questions** (including the Honor Code), and must be completed in the 170-minute time period on May 14, 2025, 11:30am – 2:30pm unless you have pre-approved accommodations otherwise.

For multiple-choice questions, select **one choice** for circular bubble options, and select **all choices that apply** for box bubble options. In either case, please indicate your answer(s) by **fully** shading in the corresponding circle/box.

Make sure to write your SID on each page to ensure that your exam is graded.

Honor Code **[1 pt]**

As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others. I am the person whose name is on the exam, and I completed this exam in accordance with the Honor Code.

Signature: ______________________________

Chapter 1: New Sport - Bouldering [15 pt]

You are a data engineer working at a company with software that helps climbing gyms track climbers' progress through their bouldering routes. Using data analytics, you help gyms spot promising climbers and create better route challenges.

The company stores information in the following relations.

Climbers Table (`climbers`)

```
CREATE TABLE climbers (
    id INT PRIMARY KEY,
    name VARCHAR(30) NOT NULL,
    join_date DATE,
    level INT
);
```

Routes Table (`routes`)

```
CREATE TABLE routes (
    id INT PRIMARY KEY,
    color VARCHAR(20) NOT NULL,
    difficulty INT
);
```

Attempts Table (`attempts`)

```
CREATE TABLE attempts (
    id INT PRIMARY KEY,
    climber_id INT REFERENCES climbers(id),
    route_id INT REFERENCES routes(id),
    attempt_date DATE,
    time_spent INTERVAL,
    is_successful BOOLEAN
);
```

1.1. [5 pt] We want to track how a climber improves over time. After every attempt, calculate their **running success rate**, defined as the proportion of successful attempts out of all attempts made up to and including that attempt.

Return the following columns: `climber_id`, `attempt_date`, `running_success_rate`.

Order the results by `climber_id` and `attempt_date` in ascending order.

Example Output:

climber_id	attempt_date	running_success_rate
101	2024-01-01	1.00
101	2024-01-03	0.50
101	2024-01-04	0.67
202	2024-02-02	0.00
202	2024-02-05	0.50
303	2024-03-01	1.00
303	2024-03-05	1.00
303	2024-03-09	0.67

Available relations (repeated here for your convenience):

```
climbers(id, name, join_date, level)
routes(id, color, difficulty)
attempts(id, climber_id, route_id, attempt_date, time_spent, is_successful)
```

```
SELECT

 ______________________________________________________________,

 __________ (CASE WHEN __________ THEN __________ ELSE __________ END)

OVER (PARTITION BY __________________ORDER BY __________________)

AS running_success_rate

FROM ____________________________________________________________

ORDER BY _______________________________________________________;
```

Solution:

```
SELECT
  climber_id,
  attempt_date,
  AVG(CASE WHEN is_successful = TRUE THEN 1 ELSE 0 END)
    OVER (PARTITION BY climber_id ORDER BY attempt_date)
  AS running_success_rate
FROM attempts
ORDER BY climber_id, attempt_date;
```

1.2. [5 pt] We want to study bouldering climbers' progress by finding attempts where a climber moves on to a harder route than their current attempt. Write a SQL query to **find all attempts where the next attempt has a higher difficulty level than the current attempt for the same climber**.

Return the following columns: `climber_id`, `attempt_id`, `next_attempt_id`, `difficulty`, `next_difficulty`.

Order the results by `climber_id` and `attempt_id` in ascending order.

Example Output:

climber_id	attempt_id	next_attempt_id	difficulty	next_difficulty
101	4001	4002	2	3
101	4005	4006	3	4
202	5002	5003	5	7
303	6003	6004	3	4
303	6004	6005	4	5

Available relations (repeated here for your convenience):

```
climbers(id, name, join_date, level)
routes(id, color, difficulty)
attempts(id, climber_id, route_id, attempt_date, time_spent, is_successful)
```

To complete the task, first, create a view that finds all the next attempts for each attempt per climber. Below, feel free to add `WHERE TRUE` if the corresponding blank doesn't need a filter.

```
CREATE OR REPLACE VIEW next_attempts AS

SELECT ____________________________________________________________

_______________________________________________AS next_attempt_id,

________________________________________________AS next_difficulty

FROM ______________________________________________________________

WHERE ____________________________________________________________;

SELECT ____________________________________________________________

___________________________________________________________________

FROM next_attempts

WHERE _____________________________________________________________

ORDER BY _________________________________________________________;
```

Solution:

```
CREATE OR REPLACE VIEW next_attempts AS
  SELECT
    a.id AS attempt_id,
    a.climber_id,
    r.difficulty,
    LEAD(a.id, 1) OVER (PARTITION BY a.climber_id
    ORDER BY a.attempt_date) AS next_attempt_id,
    LEAD(r.difficulty, 1) OVER (PARTITION BY a.climber_id
    ORDER BY a.attempt_date) AS next_difficulty
  FROM attempts a, routes r
  WHERE a.route_id = r.id
  ;

SELECT
  climber_id,
  attempt_id,
  next_attempt_id,
  difficulty,
  next_difficulty
```

```
FROM next_attempts
WHERE next_difficulty > difficulty
ORDER BY climber_id, attempt_id;
```

1.3. [5 pt] For some climbers, their skill level is missing. However, we can impute their missing level using their **most recent successful climb**. Specifically, we define the **imputed level** as the **difficulty of the most recent successful route** the climber completed. Write a SQL query to return the level, or impute the level field if it is NULL, based on this definition. You are required to use a subquery for this one. Return the following columns: `climber_id`, `climber_name`, `imputed_level`

Note: `coalesce_agg` is a custom aggregate function introduced in class to return the first non-null value in a group based on a given order.

Example Output:

climber_id	climber_name	imputed_level
101	Michelle	1
102	Li	10
103	Aditi	4
104	Zack	0
105	Christy	2

Available relations (repeated here for your convenience):

```
climbers(id, name, join_date, level)
routes(id, color, difficulty)
attempts(id, climber_id, route_id, attempt_date, time_spent, is_successful)
```

```
SELECT

  c.id AS climber_id,

  c.name AS climber_name,

  CASE

    WHEN ____________________ IS NOT NULL THEN ____________________

    ELSE (

      SELECT coalesce_agg( ______________ ORDER BY ______________ )

      FROM ______________________________________________________

      JOIN __________________________ ON __________________________

      WHERE _________________________ AND _________________________

    )

  END AS imputed_level

FROM climbers c;
```

Solution:

```
SELECT
  c.id AS climber_id,
  c.name,
  CASE
    WHEN c.level IS NOT NULL THEN c.level
    ELSE (
      SELECT coalesce_agg(r.difficulty ORDER BY a.attempt_date DESC)
      FROM attempts a
      JOIN routes r ON a.route_id = r.id
      WHERE a.climber_id = c.id AND a.is_successful = TRUE
    )
  END AS imputed_level
FROM climbers c;
```

Chapter 2: Class of 2025! [19 pt]

As part of graduation preparations, the commencement committee at Berkeley is coordinating ceremony logistics for each department. Graduating student data is stored in the `logistics` table, a portion of which is shown below. The full table contains additional rows not displayed here.

student_id	student_name	ceremony_id	venue	department	date	ticket_count
101	Aditi	7	Greek Theatre	Computer Science	2025-05-14	3
102	Manas	4	Zellerbach Hall	Public Policy	2025-05-13	2
104	Nehal	3	Zellerbach Hall	Public Health	2025-05-14	3
103	Christy	7	Greek Theatre	Computer Science	2025-05-14	3
106	Zack	2	Wheeler Auditorium	Data Science	2025-05-16	4
102	Manas	2	Wheeler Auditorium	Data Science	2025-05-16	2

2.1. [3 pt] Which of the following functional dependencies **cannot be true**? Select all that apply.

☐ A. student_name → student_id

☐ **B. student_name → ceremony_id**

☐ **C. date → department**

☐ D. department → date

☐ **E. student_id → (student_name, department)**

☐ F. (student_name, venue) → ticket_count

2.2. [2 pt] To eliminate data redundancy, the committee decides to normalize the logistics table according to the following functional dependencies:

- student_id → student_name
- ceremony_id → (venue, department)
- ceremony_id → date

Select the columns that appear in each of the normalized tables:

i. `students`	ii. `ceremonies`	iii. `tickets`
☐ **A. student_id**	☐ A. student_id	☐ **A. student_id**
☐ **B. student_name**	☐ B. student_name	☐ B. student_name
☐ C. ceremony_id	☐ **C. ceremony_id**	☐ **C. ceremony_id**
☐ D. venue	☐ **D. venue**	☐ D. venue
☐ E. department	☐ **E. department**	☐ E. department
☐ F. date	☐ **F. date**	☐ F. date
☐ G. ticket_count	☐ G. ticket_count	☐ **G. ticket_count**

2.3. [3 pt] Based on the previous two questions, which of the following statements are true? You may assume that each statement is independent of the others. Select all that apply.

- ☐ **A.** The original `logistics` table cannot be recovered once the data has been normalized into the `students`, `ceremonies`, and `tickets` tables.
- ☐ **B.** Adding a maximum capacity to each ceremony (via the insertion of a new column) would be more expensive (in terms of time) under the normalized tables (`students`, `ceremonies`, and `tickets`) than under the `logistics` table.
- [✓] **C. If the venue for a particular ceremony changed in the `logistics` table, we may need to update multiple rows.**
- [✓] **D. If additional students were added to the `logistics` table, some functional dependencies could be potentially violated.**
- ☐ **E.** If some students were deleted from the `logistics` table, some functional dependencies could be potentially violated.

2.4. [3 pt] Next, the committee decides to assign a commencement speaker to each ceremony. To keep track of all attending speakers, one member proposes introducing a new table with the following schema:

`speakers(speaker_id, speaker_name, ceremony_id, venue)`.

Assuming that the original data was normalized (across the `students`, `ceremonies`, and `tickets` tables), would introducing the `speakers` table to this set make the data susceptible to any of the following anomalies? Justify your answer in no more than one sentence.

i. Update anomalies [✓ **Yes**] No

Solution:
Since `venue` is redundantly listed in both `ceremonies` and `speakers`, any changes would need to be reflected in both tables for consistency.

ii. Delete anomalies Yes [✓ **No**]

Solution:
Deleting a record from `speakers` would not lead to unintentional data loss as venues are already stored in ceremonies.

2.5. [4 pt] This semester, Data 101 staff has been hired to assist the committee with diploma distribution for many different departments. The distribution system works as follows:

- Each distribution center is uniquely identified by an ID. All centers have an associated address.
- Each distribution center must be staffed by at least one staff member.
- A staff member may be assigned to manage at most one distribution center, but not all staff members are managers. Each distribution center must have exactly one manager.
- Some staff members are not currently assigned to any distribution center, and some work multiple shifts across multiple centers.

Select the appropriate connection between each of the components of the above ER diagram.

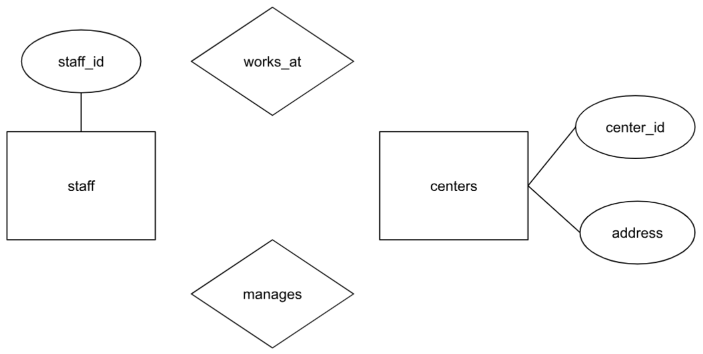

Figure 1: ER Diagram

i. `centers` and `manages`
◯ `A.` Thin line
◯ `B.` Bolded line
◯ `C.` Thin arrow
[✓ **`D.` Bolded arrow**]

ii. `centers` and `works_at`
◯ `A.` Thin line
[✓ **`B.` Bolded line**]
◯ `C.` Thin arrow
◯ `D.` Bolded arrow

iii. `staff` and `works_at`
[✓ **`A.` Thin line**]
◯ `B.` Bolded line
◯ `C.` Thin arrow
◯ `D.` Bolded arrow

iv. `staff` and `manages`
◯ `A.` Thin line
◯ `B.` Bolded line
[✓ **`C.` Thin arrow**]
◯ `D.` Bolded arrow

2.6. [4 pt] Next, imagine the ER diagram is updated to replace the two separate relationships — `works_at` and `manages` — with a single relationship `works_at`, which includes an attribute `role` (e.g., "employee", "manager").

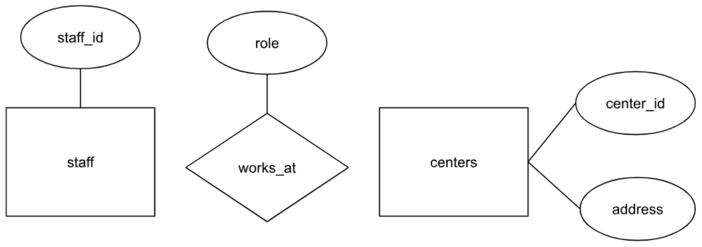

Figure 2: Modified ER Diagram

i. Identify **one advantage** of the new design (Figure 2) compared to the original (Figure 1). Your answer should include a brief justification.

Solution:
- Reduces redundancy by combining two relationships into one, making queries and maintenance easier.
- Easier to update a staff member's role by changing an attribute instead of modifying relationships.

ii. Identify **one drawback** of the new design (Figure 2) compared to the original (Figure 1). Your answer should include a brief justification.

Solution:

- Harder to enforce rules like “exactly one manager per center” or “a staff member may manage at most one center” within a single relationship table.
- Using a single relationship for both general staffing and management can make it less clear in the ER diagram what specific business rules apply to managers versus regular staff.

Chapter 3: Invited to a Potwatch? [19 pt]

Data 101 course staff is holding a potluck + watch party ("potwatch") to celebrate the end of the semester and watch the NBA Playoffs! Each staff member is responsible for buying certain grocery items for the dish they are bringing to the potwatch.

All shopping data is recorded in the MongoDB collection `db.purchases`, and item information is stored in another MongoDB collection `db.items`. A sample from those collections is below. When completing the MongoDB queries, use MQL syntax, and assume that all documents have been loaded into both collections and that each staff member will purchase at least one item.

A Purchase Document

```
{
    "_id": ObjectId("6634b8e7a1d1c2a4ef9e1000"),
    "staff_name": "Christy",
    "cart_name": "Weeknight Pasta",
    "items_bought": [
        { item_id: 1590, quantity: 1 },
        { item_id: 2031, quantity: 2 },
        { item_id: 3112, quantity: 3 },
        { item_id: 4075, quantity: 1 },
        { item_id: 5060, quantity: 1 }
    ],
    "date": "2025-04-30"
}
```

An Item Document

```
{
    "_id": 3112,
    "item_name": "Pancetta",
    "category": "Meat",
    "price_per_unit": 3.49
}
```

3.1. [5 pt] For each cart, calculate the total number of items purchased by each cart by summing up all the quantities in the `$items_bought` array.

The documents returned by this query should have the following structure:

Total Number of Items Purchased per Cart Result

```
[
  { "_id": "Weeknight Pasta", "totalItems": 5 },
  { "_id": "Chicken Parmigiana", "totalItems": 10 },
  { "_id": "Chicken with Parsley", "totalItems": 11 },
  { "_id": "Sweet and Spicy Scallops", "totalItems": 23 },
  // and so on...
]
```

Fill in the blanks below so that the query runs correctly. You may not need all of the lines provided, and include necessary syntax (ex: {}, "") if needed.

Total Number of Items Purchased per Cart Query

```
db.purchases.aggregate([
  {
    ____________________________________________________________________

    ____________________________________________________________________
  },
  {
    ____________________________________________________________________

    ____________________________________________________________________

    ____________________________________________________________________

  }
])
```

Solution:

```
db.purchases.aggregate([
  { $unwind: "$items_bought" },
  { $group: {
      _id: "$cart_name",
      totalItems: { $sum: "$items_bought.quantity" }
  }}
])
```

3.2. i. [6 pt] Staff has access to `db.purchases` and `db.items`, but it's a bit annoying and tricky navigating through both collections. Particularly, each cart contains the item IDs and quantities, but doesn't include the item name or price.

They want to create a new collection, `db.cart_item_details`, that contains the `cart_name`, `staff_name`, `item_name`, `quantity`, `price_per_unit`, and the total cost of that item line. Assume that each document will be a unique `(cart, item)` pair.

The documents returned by this query should have the following structure:

Cart Details Result

```
[
  {
    "cart_name": "Weeknight Pasta",
    "staff_name": "Christy",
    "item_name": "Pasta",
    "quantity": 1,
    "price_per_unit": 2.99,
    "item_total": 2.99
  }, ...
  {
    "cart_name": "Chicken Parmigiana",
    "staff_name": "Wesley",
    "item_name": "Garlic",
    "quantity": 2,
    "price_per_unit": 0.75,
    "item_total": 1.50
  },
  {
    "cart_name": "Chicken with Parsley",
    "staff_name": "Chenxi",
    "item_name": "Parsley",
    "quantity": 3,
    "price_per_unit": 0.49,
    "item_total": 1.47
  },
  {
    "cart_name": "Sweet and Spicy Scallops",
    "staff_name": "Aldrin",
    "item_name": "Scallops",
    "quantity": 20,
    "price_per_unit": 2.27,
    "item_total": 45.40
  },
  // and so on...
]
```

Fill in the blanks below so that the query runs correctly. You may not need all of the lines provided, and include necessary syntax (ex: {}, "") if needed.

Cart Details Query

```
db.purchases.aggregate([

  { _______________: ________________________________________________},

  { _____________: {

      ______________: ________________________________________________,

      ______________: ________________________________________________,

      ______________: ________________________________________________,

      as: "item_info"
  }},

  { _______________: ________________________________________________},

  { _____________: {

        cart_name: ___________________________________________________,

        staff_name: __________________________________________________,

        item_name: ___________________________________________________,

        quantity: ____________________________________________________,

        price_per_unit: ______________________________________________,

        item_total: {

            __________________________________________________________

            __________________________________________________________
        }
      }},
  { $out: "cart_item_details" }
])
```

Solution:

```
db.purchases.aggregate([
  { $unwind: "$items_bought" },
  { $lookup: {
      from: "items",
      localField: "items_bought.item_id",
      foreignField: "_id",
      as: "item_info"
  }},
  { $unwind: "$item_info" },
  { $project: {
      cart_name: 1,
      staff_name: 1,
      item_name: "$item_info.item_name",
      quantity: "$items_bought.quantity",
      price_per_unit: "$item_info.price_per_unit",
      item_total: {
        $multiply: ["$items_bought.quantity", "$item_info.price_per_unit"]
      }
  }},
  { $out: "cart_item_details" }
])
```

ii. [4 pt] There's a lot of similarities between MongoDB and SQL. To start, using `db.cart_item_details` we want to find the total amount spent per cart (`totalSpent`). Assume additional fees like sales tax are not included in this total.

The documents returned by this query should have the following structure:

Total Amount Spent per Cart Result

```
[
  {
    "_id": "Chicken with Parsley",
    "totalSpent": 26.52
  },
  {
    "_id": "Weeknight Pasta",
    "totalSpent": 27.28
  },
  {
    "_id": "Chicken Parmigiana",
    "totalSpent": 36.26
  },
  {
    "_id": "Sweet and Spicy Scallops",
    "totalSpent": 51.83
  },
  // and so on...
]
```

Fill in the blanks below so that the query runs correctly. You may not need all of the lines provided, and include necessary syntax (ex: {}, "") if needed.

Total Amount Spent per Cart Query

```
db.cart_item_details.aggregate([
  {
    __________: {

      _______________: ______________________________________________,

      _______________: ______________________________________________
    }
  }
])
```

Solution:

```
db.cart_item_details.aggregate([
  {
    $group: {
      _id: "$cart_name",
      total: { $sum: "$item_total" }
    }
  }
])
```

iii. [4 pt] Let's do this in SQL now! Our output should contain the cart name and total spent (`total_spent`), where rows are ordered in by `total_spent` in non-descending order.

`purchases`: Tracks items purchased per cart

```
CREATE TABLE purchases (
    staff_name VARCHAR(100),
    cart_name VARCHAR(100),
    item_id INT,
    quantity INT,
    purchase_date VARCHAR(100)
);
```

`items`: Tracks metadata (id, name, category, price) per item

```
CREATE TABLE items (
    item_id INT,
    item_name VARCHAR(100),
    category VARCHAR(100),
    price_per_unit FLOAT
);
```

Example Output:

cart_name	total_spent
Chicken with Parsley	26.52
Weeknight Pasta	27.28
Chicken Parmigiana	36.26
Sweet and Spicy Scallops	51.83
......	

```
SELECT ____________________________________________

___________________________________________________

FROM ______________________________________________

JOIN ______________________ON _____________________

GROUP BY __________________________________________

ORDER BY _________________________________________;
```

Solution:

```
Note: Students should assume that cart names are unique.

SELECT
  p.cart_name,
  SUM(p.quantity * i.price_per_unit) AS total_spent
FROM purchases p
JOIN items i ON p.item_id = i.item_id
GROUP BY p.cart_name
ORDER BY total_spent;
```

Chapter 4: Into the ~~Unknown~~ Jungle [20 pt]

Data 101 course staff is prepping for a trip to the jungle. While doing research on what type of animals they may run into, they stumble upon a food web: a diagram that displays which animals in an ecosystem eat what! After further research, they compiled the following tables:

`species` Table

```
CREATE TABLE species (
    species_id INT PRIMARY KEY,
    name VARCHAR(100) UNIQUE NOT NULL,
    habitat VARCHAR(100), -- optional, extra info
);
```

`ts_predation` Table

```
CREATE TABLE ts_predation (
    predator_id INT,
    prey_id INT,
    relationship TEXT,
    weight FLOAT,
    PRIMARY KEY (predator_id, prey_id),
    FOREIGN KEY (predator_id) REFERENCES Species(species_id),
    FOREIGN KEY (prey_id) REFERENCES Species(species_id)
);
```

You might notice that the `ts_predation` table is like a triplestore storing relationships between predator and prey. During data collection, staff were a bit sloppy, so the value in the `weight` column can represent one of two values:

- If the relationship column is 'EATS', `weight` represents how much of the prey the predator consumes (i.e., if the value is 3, a lion will eat 3 rabbits to feel satiated).
- If the relationship column is 'EATEN BY', `weight` represents how much a single prey will fill up the corresponding predator (i.e., if the value is 1/3, a rabbit will fill up 1/3 of a lion's appetite).

You can assume that the table does not have duplicate (`predator_id`, `prey_id`) combinations.

4.1. [4 pt] Using `ts_predation`, obtain the `predation` table below. You can assume the `weight` column is > 0.

```
predation (
    predator_id INT,
    prey_id INT,
    consumption_weight FLOAT
);
```

`consumption_weight` should tell us how much the corresponding type of prey the predator should consume to feel satiated.

Assume that the food web/chain is acyclic (i.e. there are no cycles). Note that a predator may have many sources of prey, and a prey may have many predators.

Additionally, assume in this question that a predator will only eat the prey explicitly listed in the table, and won't eat the prey of its prey. You also may not need all the lines provided.

```
CREATE TABLE predation AS

SELECT
  ______________________________________
  ______________________________________
  ______________________________________
  ______________________________________
  ______________________________________

FROM ts_predation;
```

Solution:

```
CREATE TABLE predation AS
SELECT predator_id,
       prey_id,
       CASE
           WHEN relationship = 'EATS' THEN weight
           WHEN relationship = 'EATEN BY' THEN 1 / weight
       END AS consumption_weight
FROM ts_predation;
```

4.2. [4 pt] **From this question onwards**, assume we have access to the `predation` table introduced in the previous question.

Write a query that returns `predator_id`, `grandprey_id`, and `same_habitat`. `same_habitat` is a column of type `BOOLEAN` that indicates whether both the predator and grandprey (the prey of a predator's prey) share a habitat. Make sure to include all the grandprey corresponding to a predator in your answer. You may not need all the lines provided.

Available relations (repeated here for your convenience):

- `species (species_id, name, habitat)`
- `predation (predator_id, prey_id, consumption_weight)`

```
WITH predator_grandprey_ids AS (
  ____________________________________________________________
  ____________________________________________________________
  ____________________________________________________________
  ____________________________________________________________
)
  SELECT pgi.*, _______________________________________________
  ____________________________________________________________
  FROM predator_grandprey_ids AS pgi, species AS predator_species,
  species AS grandprey_species
  ____________________________________________________________
  ____________________________________________________________
  ____________________________________________________________
  ___________________________________________________________;
```

Solution:

```
WITH predator_grandprey_ids AS (
  SELECT p1.predator_id, p2.prey_id AS grand_prey_id
  FROM predation AS p1, predation AS p2
  WHERE p1.prey_id = p2.predator_id
)
SELECT pgi.*,
       grandprey_species.habitat = predator_species.habitat AS same_habitat
FROM predator_grandprey_ids AS pgi,
     species AS predator_species,
     species AS grandprey_species
WHERE predator_species.predator_id = pgi.predator_id
  AND grandprey_species.prey_id = pgi.grand_prey_id;
```

4.3. [6 pt] Write a query that outputs, **across all predators in the food web**, the highest number of prey (including the prey inside that prey, etc.) inside a given predator if it eats its required amount of prey to feel full. Name this new column `total_prey_consumed`, and include the `predator_id`.

In other words, your query should return the maximum number of prey objects consumed along any "path," along with the `predator_id`.

For example: Say a lion needs to eat 3 rabbits to feel full, and a rabbit can either eat 2 carrots or 5 lettuce to feel full. Meanwhile, a bear needs to eat 20 berries to feel full.

Then, the most prey "objects" in the lion is 15 (3 rabbits × 5 lettuce per rabbit), not 6 (3 rabbits × 2 carrots). Furthermore, the most prey "objects" in the bear is 20. Therefore, the bear has the most prey "objects" consumed along any path.

Expected output for this example (assume the `predator_id` of a bear is 5):

predator_id	total_prey_consumed
5	20

You may not need all the lines provided. Available relations (for your convenience):

- `species (species_id, name, habitat)`
- `predation (predator_id, prey_id, consumption_weight`

```
WITH RECURSIVE prey_chain(predator_id, prey_id, consumption_weight)
AS ( SELECT  -- path length 1

  ________________________________________________

FROM ____________________________________________

UNION ALL

SELECT

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

  ________________________________________________

)
-- After building the full prey_chain, get the max consumption weight
-- (number of animals eaten by a predator) across all predators.

SELECT __________________________________________

FROM prey_chain

  ________________________________________________

  ________________________________________________

  ________________________________________________;!
```

Solution:

```
WITH RECURSIVE prey_chain(predator_id, prey_id, consumption_weight) AS (
    SELECT predator_id, prey_id, consumption_weight -- path length 1
    FROM predation

    UNION ALL

    SELECT
        pc.predator_id,
        p.prey_id,
        pc.consumption_weight * p.consumption_weight AS consumption_weight
    FROM prey_chain AS pc
    JOIN predation AS p ON pc.prey_id = p.predator_id
)
-- After building the full prey_chain, get the max consumption weight
-- (number of animals eaten by a predator) across all predators.
SELECT
    predator_id,
    consumption_weight AS total_prey_consumed
FROM prey_chain
ORDER BY total_prey_consumed DESC
LIMIT 1;
```

4.4. [6 pt] **Note: This is a challenging question; only attempt it when you are done with other questions.**

We will now be writing a MapReduce program that gives us the number of "grand-prey" (prey of a predator's prey) for a given predator. The data is organized as multiple `(predator, prey)` tuples, with each map operation operating on one tuple at a time, and emitting key-value pairs.

You will now explain in words what the map and reduce steps will do. Read through the entire question before filling in the blanks.

For the map step, describe, given a tuple `(predator pd, prey py)`, what key-value pairs will be emitted. You will need to emit two key-value pairs per input tuple, where the value is the same as the input tuple `(pd, py)` itself; fill in the two keys below.

```
Key 1: ________________________________Value 1: (pd, py)

Key 2: ________________________________Value 2: (pd, py)
```

Solution:

```
Key 1:  pd                    Value 1: (pd, py)
Key 2:  py                    Value 2: (pd, py)
```

As is typical in MapReduce, all key-value pairs with the same key `k` will end up at the same reduce node. Suppose `S = {(pd_1, py_1) ... (pd_n, py_n)}` are the set of values for the given key `k`. Our goal is to output pairs `(predator pd, grandprey gpy)`.

Describe the procedure we can use to construct the output pairs by filling in the blanks below (you can use pseudocode, as long as it's interpretable):

```
L = {}; R = {};
For all (pd_i, py_i) in S where pd_i = k:
    R = R U {py_i}
For all (pd_i, py_i) in S where py_i = k:
    L = L U {pd_i}

Return ______________________________________________

 ____________________________________________________
```

Solution:

```
L = {}; R = {};
```

```
For all (pd_i, py_i) in S where pd_i = k:
    R = R U {py_i}
For all (pd_i, py_i) in S where py_i = k:
    L = L U {py_i}
Return (l, r) for all l in L and r in R
```

Chapter 5: What is Potpourri?! [29 pt]

5.1. [1 pt] What are the differences between spreadsheets and relations? Select all that apply.

☐ **A. In spreadsheets, there may have missing or inconsistent column headers, whereas in relations, every column (attribute) must be named and typed.**

☐ **B. Spreadsheets allow mixing different types (e.g., text, number, date) within a single column, but relations require each column to have a consistent data type.**

☐ **C. Spreadsheets support ad hoc cell-level operations like copy-paste and formulae in place of values, while relational systems enforce integrity through schemas and constraints.**

☐ D. Spreadsheets support formulae, whereas relational databases do not support any form of computed or derived attributes.

5.2. [2 pt] You are working with the following spreadsheet:

	A	B	C	D
1	Day 1	Day 2	Day 3	Day 4
2	5	8	6	7

- Row 2 contains sales on each day.
- You want Row 3 to compute the **cumulative sales up to that day** (i.e., `=A2` in column A, `=A2+B2` in column B, `=A2+B2+C2` in column C, etc.).

You plan to write **one formula** in cell A3 and then **drag it to the right** to fill B3, C3, D3. Which of the following formulas will correctly compute the cumulative total in Row 3 when written in cell **A3** and dragged across? Select all that apply.

☐ **A. `=SUM($A$2:A2)`**

☐ B. `=SUM($A2:A2)`

☐ C. `=SUM($A$2:$A2)`

☐ **D. `=SUM(A$2:A2)`**

5.3. [2 pt] Which of the following is typically true when optimizing for performance? Select all that apply.

◯ A. Indexes are helpful for write-heavy workloads.

◯ B. Normalization can help avoid expensive joins.

◯ **C. Predicates and projections should be pushed down.**

◯ D. We should try joining tables in decreasing order of size.

5.4. [1 pt] Assuming you, as a business analyst, are performing a join between two relations on `c_id` (the unique identifier of each customer), using tables:

- `Customers` (sorted by `c_id`)
- `Transactions` (sorted by `c_id`)

Which type of join is likely to be the most efficient to use?

◯ A. Nested-loop join

◯ B. Cartesian product

◯ C. Hash join

◯ **D. Sort-merge join**

5.5. [2 pt] You are working with a large `Orders` table with the following schema:

`Orders (order_id, customer_id, order_date, status, total_amount)`

You frequently run the following query to generate dashboards:

```
SELECT customer_id, SUM(total_amount)
FROM Orders
WHERE status = 'completed'
GROUP BY customer_id
ORDER BY SUM(total_amount) DESC
LIMIT 10;
```

Which of the following actions would likely improve the performance of this query? Select all that apply.

☐ **A. Building an index on `status`.**

☐ **B. Building an index on `customer_id`.**

☐ **C. Clustering the relation on `customer_id`.**

☐ D. Building an index on `total_amount`.

5.6. [2 pt] You are working with the relation `Employees(empl_id, state, salary)`, and you construct a stratified sample based on the state attribute. The population sizes and [Exam correction] ~~average~~ total salaries per stratum are as follows:

- CA: 200 employees, sample of 10, total salary for the sample = $600,000
- WA: 200 employees, sample of 10, total salary for the sample = $500,000
- NY: 400 employees, sample of 10, total salary for the sample = $700,000

You want to estimate the result of the following query using your stratified sample:

```
SELECT AVG(salary) FROM Employees;
```

Which of the following is the correct **estimated average salary of all employees** based on the stratified sample? Feel free to leave your answer as an expression, if needed.

____

Solution:

`62,500 or correct equivalent expression`

5.7. [1 pt] A bank database processes a transaction that transfers $50 from Christy's account to Zack's. Midway through the transaction, the system crashes after withdrawing from Christy's account, but before depositing it to Zack's account. When the system restarts, Zack's account shows no change, but Christy has already lost $50. Which of the following ACID properties are violated in this scenario? Select all that apply.

☐ **A. Atomicity**

☐ B. Isolation

☐ C. Consistency

☐ D. Durability

5.8. [2 pt] When a student enrolls in a particular course, a transaction T_i performs the following actions in order:

- $R_i(C)$: Read from courses (to check current status).
- $W_i(C)$: Write to courses (to update the opening seats).
- ~~$R_i(E)$~~ $W_i(E)$ [Exam correction]: Write to enrollment (to update the students' enrollment).

Suppose that two students are trying to simultaneously enroll in the same course, resulting in concurrent transactions T_1 and T_2.

The DBMS generates the following transaction schedule:

	1	2	3	4	5	6
T_1	$R_1(C)$		$W_1(C)$			$W_1(E)$
T_2		$R_2(C)$		$W_2(C)$	$W_2(E)$	

Table 5: Transaction Schedule for T_1 and T_2

i. [1 pt] Is this serializable?

A. Yes [✓ B. No] C. Cannot be determined

ii. [1 pt] Does this schedule provide isolation between T_1 and T_2?

A. Yes [✓ B. No] C. Cannot be determined

iii. [2 pt] Justify your answer to questions i) and ii) in no more than two sentences.

Solution: The schedule is not serializable because it contains a cycle in the conflict graph: T_1 reads from courses before T_2 writes, and T_2 reads before T_1 writes — creating a write–read conflict in both directions. Isolation is violated because both transactions access and modify the same data concurrently, leading to a possible lost update or overbooking of the course.

5.9. [1 pt] You are analyzing sales data using the following SQL query:

```
SELECT region, SUM(amount)
FROM sales
GROUP BY region;
```

Later, a stakeholder asks you to modify the query to:

```
SELECT region, store_id, SUM(amount)
FROM sales
GROUP BY region, store_id;
```

Which of the following best describes what happened to the output relation and OLAP operation in this query change?

○ **A. The data became more fine-grained; this is a drilldown operation.**

○ B. The data remained at the same level of detail; this is just a formatting change.

○ C. The data became more coarse-grained; this is a **drilldown** operation.

○ D. The data became more fine-grained; this is a **rollup** operation.

5.10. [2 pt] Suppose there are 10 unique regions and 25 store IDs, so the original query above therefore has 10 output tuples. What are the best case and the worst case numbers of output tuples in the modified query (repeated here for convenience)?

```
SELECT region, store_id, SUM(amount)
FROM sales
GROUP BY region, store_id;
```

○ A. Best case = 10, worst case = 25

○ **B. Best case = 25, worst case = 250**

○ C. Best case = 10, worst case = 35

○ D. Best case = 25, worst case = 35

5.11. [2 pt] What makes OLAP fundamentally different from OLTP? Select all that apply.

☐ A. OLAP typically operates on small amounts of data, while OLTP operates on large amounts of data.

☐ **B. OLAP typically operates on large amounts of data, while OLTP operates on small amounts of data.**

☐ **C. OLTP has higher response time requirements, while OLAP can tolerate staleness.**

☐ D. OLAP has higher response time requirements, while OLTP can tolerate staleness

5.12. [1 pt] You are building a data warehouse for a university's enrollment analytics platform.

The team is currently using a star schema, where the fact `Enrollment` table connects directly to dimension tables such as `Student` and `Course`. Now, one team member suggests **normalizing the `Course` table** by splitting off attributes like `department` and `instructor_name` into separate related tables. Which of the following consequences are likely to occur? Select all that apply.

☐ A. The warehouse will contain multiple fact tables for each dimension, increasing query complexity.

☐ **B. The overall data model will become more normalized, reducing storage redundancy.**

☐ **C. Query performance may decrease due to the additional joins required between normalized dimension tables.**

☐ D. The structure will become flatter, improving analytical query performance.

5.13. [2 pt] You are given the following relations:

- `Books (bid, title, genre)`
- `Sales (bid, store_id, copies_sold)`

Which of the following relational algebra expressions correctly return the **titles and genres of all Science Fiction books that were *only* sold in store 42**? Select all that apply.

☐ A. $\pi_{\text{title, genre}}\big(\sigma_{\text{store\_id}=42}(\text{Sales}) \cap \sigma_{\text{genre}='\textit{Science Fiction}'}(\text{Books})\big)$

☐ **B.** $\boldsymbol{\pi_{\text{title, genre}}\big(\sigma_{\text{store\_id}=42 \wedge \text{genre}='\textit{Science Fiction}'}(\text{Books} \bowtie \text{Sales}) - \pi_{\text{title, genre}}\big(\sigma_{\text{store\_id}\neq 42}(\text{Books} \bowtie \text{Sales})\big)}$

☐ C. $\sigma_{\text{genre}='\textit{Science Fiction}'}\big(\big(\pi_{\text{title, genre}}(\text{Books}) \bowtie \sigma_{\text{store\_id}=42}(\text{Sales})\big)\big)$

☐ D. $\pi_{\text{title, genre}}\big(\sigma_{\text{genre}='\textit{Science Fiction}'}(\text{Books}) \bowtie \big(\sigma_{\text{store\_id}=42}(\text{Sales}) - \sigma_{\text{store\_id}\neq 42}(\text{Sales})\big)\big)$

5.14. [2 pt] You are given the following relations:

- `Students (sid, name)`
- `TA (sid, course_id)`

Which of the following relational algebra expressions correctly return the **names of students who are *not* TAs**? Select all that apply.

☐ A. $\pi_{\text{name}}(\text{Students} - \text{TA})$

☐ B. $\pi_{\text{name}}(\text{Students}) - \pi_{\text{name}}(\text{TA})$

☐ C. $\pi_{\text{name}}(\text{Students}) \cap \pi_{\text{name}}(\sigma_{\text{Student.sid}\neq\text{TA.sid}}(\text{Students} \bowtie \text{TA}))$

☐ **D.** $\boldsymbol{\pi_{\text{name}}(\text{Students}) - \pi_{\text{name}}(\text{Students} \bowtie \text{TA})}$

5.15. [1 pt] What of the following are true about DBT (Data Build Tool)? Select all that apply.

☐ **A. It helps set up derived views based on base relations.**

☐ **B. It can define unit tests to validate the resulting relations.**

☐ C. Operations in DBT are defined in python.

☐ D. DBT uses a message passing framework, like RabbitMQ

5.16. [1 pt] You are given a table recording daily sales for a store. Using **linear interpolation**, what are the correct filled-in values for the NULLs? Select the correct set of imputed values.

Day	**Sales**
1	100
2	NULL
3	NULL
4	160
5	190
6	NULL
7	250

Table 6: Daily Sales with Missing Values

◯ A. Day 2 = 115, Day 3 = 145, Day 6 = 220

◯ B. Day 2 = 120, Day 3 = 140, Day 6 = 230

◯ **C. Day 2 = 120, Day 3 = 140, Day 6 = 220**

◯ D. Day 2 = 120, Day 3 = 150, Day 6 = 220

Chapter 6: Congratulations! [0 pt]

Congratulations! You have completed this exam.

- Make sure that you have written your Student ID number on every other page of the exam. You may lose points on pages where you have not done so.
- Also ensure that you have signed the Honor Code on the cover page of the exam.
- If more than 10 minutes remain in the exam period, you may hand in the exam **and** the reference packet and leave.
- If $\leq$ 10 minutes remain, please sit quietly until the exam concludes.

[Optional, 0 pts] Use this page to draw your favorite Data 101/Info 258 moment!