

DATA 101/CS 187: DATA ENGINEERING

MIDTERM EXAM 1 SOLUTIONS

UC Berkeley, Fall 2026

October 02, 2026

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 **90** points spread over **7 questions** (including the Honor Code), and must be completed in the 110-minute time period on October 02, 2026, 7:10pm – 9:00pm 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: Potpourri [8 pt]

1.1. This question uses the table menu, which lists drinks sold at Berkeley cafes.

```
CREATE TABLE menu (
  drink      VARCHAR(50) PRIMARY KEY,
  cafe       VARCHAR(50) NOT NULL,
  price      INT NOT NULL,
  rating     INT          -- NULL if the drink has no ratings yet
);
```

The table contains exactly six rows shown here:

drink	cafe	price	rating
Karak Chai	Elaichi	6	9
Adeni Chai	Qahwa	6	8
Oat Chai	Elaichi	8	NULL
Matcha Latte	8 Grams	8	9
Pumpkin Latte	Starbucks	10	4
Vanilla Latte	Mind	5	NULL

i. [1 pt] How many rows does this query return?

```
SELECT * FROM menu WHERE rating != 9;
```

Solution: 2 rows

ii. [2 pt] The query below is meant to return the last word of each drink name, for example 'Latte' for 'Matcha Latte'.

```
SELECT REGEXP_REPLACE(drink, [[REGEX]], '\1') AS drink_type
FROM menu;
```

Which regular expression will **not** work in place of [[REGEX]]?

- ☒ A. `'.* ([A-Za-z+])$'`
- ☐ B. `'.* ([^\s]+)$'`
- ☐ C. `'.* (\w+)$'`
- ☐ D. `'.* (\w{1,})$'`

iii. [1 pt] What does this query return?

```
SELECT PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY price)
FROM menu;
```

☒ A. 6

☐ B. 7

☐ C. 7.17

☐ D. 8

iv. [2 pt] How many rows does this query return?

```
(SELECT cafe FROM menu WHERE price < 7
 UNION ALL
 SELECT cafe FROM menu WHERE rating = 9)
EXCEPT
SELECT cafe FROM menu WHERE drink = 'Vanilla Latte';
```

☐ A. 2

☒ B. 3

☐ C. 4

☐ D. 5

v. [2 pt] The following statements are run in order:

```
UPDATE menu SET price = price + 1 WHERE cafe = 'Elaichi';
DELETE FROM menu WHERE rating IS NULL;
```

What does the following query return now?

```
SELECT COUNT(*) AS n, SUM(price) AS total FROM menu;
```

☐ A. n = 4, total = 30

☒ B. n = 4, total = 31

☐ C. n = 6, total = 43

☐ D. n = 6, total = 45

Chapter 2: Campus Beats On Repeat [18 pt]

Campus Beats is a music streaming service and has the following tables:

```
CREATE TABLE users (  
    uid          INT PRIMARY KEY,  
    name         VARCHAR(100),  
    plan         VARCHAR(10) CHECK (plan IN ('free', 'student', 'premium')),  
    join_date    DATE  
);
```

```
CREATE TABLE artists (  
    aid          INT PRIMARY KEY,  
    name         VARCHAR(100)  
);
```

```
CREATE TABLE tracks (  
    tid          INT PRIMARY KEY,  
    title        VARCHAR(200),  
    aid          INT,  
    duration     INT,  
    release_date DATE,  
    FOREIGN KEY (aid) REFERENCES artists(aid)  
);
```

```
CREATE TABLE streams (  
    sid          INT PRIMARY KEY,  
    uid          INT,  
    tid          INT,  
    played_at    DATE,  
    seconds_played INT,      -- how long the user listened to a song  
    FOREIGN KEY (uid) REFERENCES users(uid),  
    FOREIGN KEY (tid) REFERENCES tracks(tid)  
);
```

2.1. Campus Beats is building Semester Wrapped, a feature that shows every Cal student their listening stats at the end of the semester. The feature is powered by the following raw data sources:

- Streaming events that the mobile app logs on each user's mobile phone.
- A nightly snapshot of the application database (user profiles, tracks, artists).
- Spreadsheets exported from an advertising platform.
- Transactions from any plan purchases.

i. [1 pt] To power Semester Wrapped, Campus Beats copies raw log files into staging tables inside its data warehouse. Scheduled SQL jobs inside the warehouse then de-duplicate events, convert timestamps to UTC, and populate various analytical tables that feed the Semester Wrapped feature. Which term best describes this approach?

- ☐ A. ETL, because the events are de-duplicated and converted to UTC before they reach the analytics tables
- ☐ B. A data lake, because raw files are kept in their original form
- ☒ C. **ELT, because the raw files are loaded into the warehouse before any transformation happens**
- ☐ D. A lakehouse, because raw and modeled data live in one system with SQL on top

ii. [1 pt] Beyond Semester Wrapped, the Campus Beats finance team must report quarterly revenue to investors. Every team must be able to work with identical numbers, and the schema of the revenue tables changes rarely. Which approach should the data engineering team adopt to store this data?

- ☐ A. A data lake of raw files transformed on demand by each team
- ☒ B. **A data warehouse with a schema designed in advance**
- ☐ C. A schema-less NoSQL store
- ☐ D. Separate spreadsheets to be maintained by each team

- 2.2. Semester Wrapped needs a campus-wide leaderboard of the top 10 artists, ranked by total hours streamed across all of their tracks during August 2026. If the total number of hours are identical, you must break ties by sorting artist names in alphabetical order. For each of these artists, report the total hours streamed and the number of distinct tracks played (as `n_tracks`).

Fill in each section of the query below, using the answer boxes that follow the query. **Note: you may not need all the sections; leave the corresponding answer box empty if this is the case.** *Hint:* the `played_at` column is in the form 'YYYY-MM-DD'.

```
SELECT a.name,  
       _____ A _____ AS n_tracks,  
       _____ B _____ AS total_hours  
FROM streams AS s  
     JOIN tracks AS t ON _____ C _____  
     JOIN artists AS a ON _____ D _____  
WHERE _____ E _____  
GROUP BY _____ F _____  
HAVING _____ G _____  
ORDER BY _____ H _____  
LIMIT 10;
```

- i. [1 pt] Fill in section A

Solution: `COUNT(DISTINCT t.tid)` or `COUNT(DISTINCT s.tid)`

- ii. [1 pt] Fill in section B

Solution: `SUM(s.seconds_played) / 3600.0`
or `SUM(s.seconds_played / 3600.0)`

- iii. [1 pt] Fill in section C

Solution: `s.tid = t.tid`

iv. [1 pt] Fill in section D

Solution: `t.aid = a.aid`

v. [1 pt] Fill in section E

Solution:

```
s.played_at >= '2026-08-01' AND s.played_at < '2026-09-01'  
or s.played_at >= '2026-08-01' AND s.played_at <= '2026-08-31'  
or s.played_at > '2026-07-31' AND s.played_at < '2026-09-01'  
or s.played_at > '2026-07-31' AND s.played_at <= '2026-08-31'  
or s.played_at BETWEEN '2026-08-01' AND '2026-08-31'
```

vi. [1 pt] Fill in section F

Solution: `a.aid, a.name`
or `a.aid`

vii. [1 pt] Fill in section G

Solution: Not required for this query

viii. [1 pt] Fill in section H

Solution: `total_hours DESC, a.name ASC`
or `SUM(s.seconds_played) / 3600.0 DESC, a.name ASC`

2.3. The Campus Beats home page features a “Top 50” playlist. Answer the following questions.

i. [1 pt] Assume that Campus Beats uses a query *Q* to compute the current “Top 50” tracks. At 3:00 pm, Campus Beats creates the following three objects, each using results from query *Q*:

- `top_tracks_table`, using `CREATE TABLE ... AS`
- `top_tracks_view`, using `CREATE VIEW ... AS`
- `top_tracks_mv`, using `CREATE MATERIALIZED VIEW ... AS`

At 3:10 pm, a newly released track called “Affogato” goes viral. It would now appear in the Top 50 playlist if query *Q* were evaluated using the current contents of the `streams` table. Campus Beats is using PostgreSQL and has not run any other statements on the three objects since their creation. Which of the following statements are true? **Select all that apply.**

☐ A. The query `SELECT * FROM top_tracks_table;` will include “Affogato”.

☐ B. The query `SELECT * FROM top_tracks_view;` will include “Affogato”.

- ☐ C. The query `SELECT * FROM top_tracks_mv;` will include “Affogato”.
- ii. [1 pt] Refreshing the “Top 50” playlist ends up being slow, and Campus Beats now decides to restrict `streams` to the most recent 7 days, which is a very small fraction of rows in the table. Which option is likely to most help speed up their query?
- ☐ A. A hash index on `streams(played_at)`
- ☒ B. A B+ tree index on `streams(played_at)`
- ☐ C. A B+ tree index on `streams(sid)`
- ☐ D. A sequential scan on `streams`

- iii. [1 pt] For this part, assume there exists a table named `top_tracks`, containing 50 rows, all stored in a single page. A data engineer creates a B+ tree index on `top_tracks(n_streams)`, and then executes the following query:

```
SELECT *
FROM top_tracks
ORDER BY n_streams DESC;
```

However, the query plan reveals that the planner chose a sequential scan followed by a sort, rather than an index scan. Which of the following best explains why?

- ☐ A. An index can only be used when the query contains a WHERE clause
- ☐ B. A B+ tree index cannot be used for a query requiring sorting
- ☐ C. Multiple tracks may have the same `n_streams`, requiring a sequential scan
- ☒ D. Scanning the page and sorting 50 rows may cost less than traversing the index

- 2.4. [3 pt] Consider the instances below, assuming the tables **contain no other rows**.

users		
uid	name	plan
1	Ana	student
2	Ben	free
3	Cy	premium
4	Dani	free

streams			
sid	uid	tid	seconds_played
501	1	10	180
502	1	11	30
503	1	10	200
504	2	10	15
505	3	12	240

Fill in the **complete output** of the following query, **including the correct row order**:

```
SELECT u.plan,
       COUNT(DISTINCT s.tid) AS n_tracks,
       SUM(s.seconds_played) AS total_seconds
FROM users AS u
LEFT JOIN streams AS s ON s.uid = u.uid
GROUP BY u.plan
HAVING SUM(s.seconds_played) > 0
ORDER BY total_seconds DESC;
```

plan	n_tracks	total_seconds
_____	_____	_____
_____	_____	_____
_____	_____	_____

Solution:

plan	n_tracks	total_seconds
student	2	410
premium	1	240
free	1	15

- 2.5. [2 pt] The growth team wants to know how many users on each subscription plan who joined in 2026 have streamed at least one track. The following query is intended to return one row per plan with the count of such users.

```
SELECT u.plan, COUNT(*) AS n_users
FROM users AS u JOIN streams AS s ON s.uid = u.uid
WHERE u.join_date >= '2026-01-01' AND u.join_date < '2027-01-01'
GROUP BY u.plan, u.uid;
```

Which of the following are errors in the query? **Select all that apply.**

- ☒ A. **COUNT(*) counts streams rather than users**
- ☐ B. The condition on join_date must be placed in a HAVING clause
- ☐ C. The join must be a LEFT JOIN so that users with no streams are counted
- ☒ D. **u.uid must be removed from the GROUP BY clause**
- ☐ E. u.plan must be removed from the SELECT clause

Chapter 3: TicketMaestro is Sold Out [21 pt]

TicketMaestro sells tickets for arena and stadium tours.

```
CREATE TABLE performers (  
    id            INT PRIMARY KEY,  
    name          VARCHAR(100)  
);  
  
CREATE TABLE shows (  
    id            INT PRIMARY KEY,  
    performer_id  INT,  
    city          VARCHAR(50),  
    show_year     INT,  
    FOREIGN KEY (performer_id) REFERENCES performers(id)  
);  
  
CREATE TABLE fans (  
    id            INT PRIMARY KEY,  
    name          VARCHAR(100),  
    tier           VARCHAR(10) CHECK (tier IN ('standard', 'verified', 'vip')),  
    join_year     INT  
);  
  
CREATE TABLE tickets (  
    id            INT PRIMARY KEY,  
    fan_id        INT,  
    show_id       INT,  
    purchased_at  TIMESTAMP,  
    price         DECIMAL(10, 2),  
    resold        BOOLEAN,  
    FOREIGN KEY (fan_id) REFERENCES fans(id),  
    FOREIGN KEY (show_id) REFERENCES shows(id)  
);
```

3.1. After a chaotic presale, TicketMaestro's operations team is running a series of analyses on the data. Identify correct relational algebra expressions or SQL queries for each analysis.

- i. [2½ pt] The marketing team needs to send promotional emails to all performers who are touring in 2026. **Select all** expressions that return the names of these performers.

- ☐ A. $\pi_{\text{performers.name}}(\text{performers} \times \sigma_{\text{show_year}=2026}(\text{shows}))$
- ☐ B. $\pi_{\text{performers.name}}(\sigma_{\text{show_year}=2026}(\text{performers}))$
- ☐ C. $\pi_{\text{name}}(\text{performers} \bowtie \sigma_{\text{show_year}=2026}(\text{shows}))$
- ☒ D. $\pi_{\text{performers.name}}(\text{performers} \bowtie_{\text{performers.id=shows.performer_id}} (\sigma_{\text{show_year}=2026}(\text{shows})))$
- ☒ E. $\pi_{\text{performers.name}}(\sigma_{\text{show_year}=2026}(\text{performers} \bowtie_{\text{performers.id=shows.performer_id}} \text{shows}))$

- ii. [2 pt] TicketMaestro wants to recruit performers who are not yet touring on the platform. **Select all** expressions that return the names of performers who have no shows.

- ☐ A. $\pi_{\text{name}}(\sigma_{\text{performers.id} \neq \text{shows.performer_id}}(\text{performers} \times \text{shows}))$
- ☐ B. $\pi_{\text{id}}(\text{performers}) - \pi_{\text{performer_id}}(\text{shows})$
- ☐ C. $\pi_{\text{name}}(\text{performers} \bowtie_{\text{performers.id=shows.performer_id}} \text{shows})$
- ☒ D. $\pi_{\text{name}}(\text{performers} \bowtie (\pi_{\text{id}}(\text{performers}) - \rho_{\text{performer_id} \rightarrow \text{id}}(\pi_{\text{performer_id}}(\text{shows}))))$
- ☐ E. None of the above

- iii. [2 pt] Which of the SQL queries below return the same names as the correct expression in the previous part, i.e., return the names of performers with no shows? Assume that `shows.performer_id` is never NULL. **Select all that apply.**

- ☒ A. **SELECT name
FROM performers
WHERE id NOT IN (SELECT performer_id FROM shows);**
- ☐ B. **SELECT a.name
FROM performers AS a
JOIN shows AS s ON a.id != s.performer_id;**
- ☒ C. **SELECT a.name
FROM performers AS a
LEFT JOIN shows AS s ON a.id = s.performer_id
WHERE s.id IS NULL;**
- ☒ D. **SELECT name
FROM performers NATURAL JOIN (
 (SELECT id FROM performers)
 EXCEPT
 (SELECT performer_id AS id FROM shows)**

) AS x;

- iv. [2 pt] The operations team wants to identify shows with unusually high demand. Write a relational algebra expression that returns the `show_id` of every show for which more than 50,000 tickets were purchased.

Hint: Use γ to GROUP BY.

Solution: $\pi_{\text{show_id}}(\sigma_{\text{n_tickets} > 50,000}(\rho_{\text{show_id, n_tickets}}(\gamma_{\text{show_id, COUNT}^*}(\text{tickets}))))$

Alternative solutions (others were also accepted if correct):

$\pi_{\text{show_id}}(\sigma_{\text{COUNT}^* > 50,000}(\gamma_{\text{show_id, COUNT}^*}(\text{tickets})))$

$\pi_{\text{show_id}}(\sigma_{\text{COUNT}^* > 50,000}(\gamma_{\text{show_id, COUNT}^*}(\text{tickets} \bowtie_{\text{tickets.show_id=shows.id}} \text{shows})))$

- 3.2. Unfortunately, TicketMaestro's system only supports primitive relational operators. The compliance team needs any new queries to only use primitive operators $\{\sigma, \pi, \rho, \times, \cup, -\}$ going forward. Help TicketMaestro complete the following expression that returns the names of fans in the 'verified' tier who have purchased at least one ticket to a show by a performer named 'The Weekday':

$\pi_{[A]}(\sigma_{[B] \wedge [C] \wedge [D] \wedge [E] \wedge [F]}(\text{fans } [G] \text{ tickets } [G] \text{ shows } [G] \text{ performers}))$

- i. [1/2 pt] Which operator belongs in each position marked [G]?

☐ A. $-$

☐ B. $\cup$

☒ C. $\times$

☐ D. $\bowtie$

- ii. [2 pt] Which conditions must appear among the blanks [B] to [F]? **Select all that apply.**

☒ A. `fans.id = tickets.fan_id`

☒ B. `tickets.show_id = shows.id`

☒ C. `shows.performer_id = performers.id`

☐ D. `fans.id = performers.id`

☐ E. `fans.tier = 'verified'`

☐ F. `performers.name = 'The Weekday'`

iii. [1 pt] Which attribute belongs in [A]?

☐ A. `fans.id`

☐ B. `performers.name`

☐ C. `tickets.fan_id`

☐ D. `fans.name`

3.3. After a presale controversy, TicketMaestro's analytics team is building several reports using subqueries and CTEs.

i. [3 pt] The analytics team wants to identify show ids for which no fans of the 'vip' tier purchased a ticket. Assume `tickets.show_id` and `tickets.fan_id` are never NULL. Which of the following queries return the required show ids? **Select all that apply.**

☐ A.

```
SELECT s.id
FROM shows AS s
WHERE NOT EXISTS (
    SELECT * FROM tickets AS t JOIN fans AS f ON t.fan_id = f.id
    WHERE t.show_id = s.id AND f.tier = 'vip');
```

☐ B.

```
SELECT DISTINCT s.id
FROM shows AS s
    LEFT JOIN tickets AS t ON t.show_id = s.id
    LEFT JOIN fans AS f ON f.id = t.fan_id AND f.tier = 'vip'
WHERE f.id IS NULL;
```

☐ C.

```
SELECT DISTINCT s.id
FROM shows AS s
    JOIN tickets AS t ON t.show_id = s.id
    JOIN fans AS f ON f.id = t.fan_id
WHERE f.tier != 'vip';
```

☐ D.

```
SELECT s.id
FROM shows AS s
WHERE s.id NOT IN (
    SELECT t.show_id FROM tickets AS t
    JOIN fans AS f ON t.fan_id = f.id
    WHERE f.tier = 'vip');
```

- ii. TicketMaestro wants to identify its biggest spenders in 2026 for a loyalty reward. Fill in each section, using the answer boxes that follow the query, so that the query returns the name and the total amount spent on tickets purchased in 2026 as `total` for every fan whose total expense in 2026 is greater than the average of the total expense in 2026. The average is taken over fans with at least one purchase in 2026.

```
WITH spend AS (  
    SELECT _____ A _____ AS fan_id, _____ B _____ AS total  
    FROM tickets  
    WHERE _____ C _____  
    GROUP BY _____ D _____  
)  
SELECT f.name, s.total  
FROM spend AS s JOIN fans AS f ON _____ E _____  
WHERE s.total > (SELECT _____ F _____ FROM spend);
```

α) [1 pt] Fill in section A

Solution: `fan_id` or `tickets.fan_id`

β) [1 pt] Fill in section B

Solution: `SUM(price)` or `SUM(tickets.price)`

γ) [1 pt] Fill in section C

Solution:
`purchased_at >= '2026-01-01' AND purchased_at < '2027-01-01'`
or
`EXTRACT (YEAR FROM purchased_at) = 2026`

δ) [1 pt] Fill in section D

Solution: `fan_id` or `tickets.fan_id`

ε) [1 pt] Fill in section E

Solution: `s.fan_id = f.id`

ζ) [1 pt] Fill in section F

Solution: `AVG(total)` or `AVG(spend.total)` or equivalent.

Chapter 4: Cal Enroll [14 pt]

Cal Enroll is a simplified course enrollment system. Each course can have one or more sections. In each section, a student is either enrolled or waitlisted.

```
CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    dept      VARCHAR(10) NOT NULL,
    number    VARCHAR(10) NOT NULL,
    title     TEXT,
    units     INT CHECK (units BETWEEN 1 AND 4),
    UNIQUE (dept, number)
);

CREATE TABLE sections (
    section_id INT PRIMARY KEY,
    course_id  INT NOT NULL REFERENCES courses(course_id) ON DELETE CASCADE,
    term       VARCHAR(6) NOT NULL,
    capacity   INT NOT NULL CHECK (capacity > 0)
);

CREATE TABLE enrollments (
    student_id INT NOT NULL,
    section_id INT REFERENCES sections(section_id)
                ON DELETE CASCADE ON UPDATE CASCADE,
    status     VARCHAR(10) NOT NULL
                CHECK (status IN ('enrolled', 'waitlisted')),
    added_at   TIMESTAMP NOT NULL,
    PRIMARY KEY (student_id, section_id)
);
```

Each of the tables contains the following rows:

courses

course_id	dept	number	title	units
1	DATA	101	Data Engineering	4
2	CS	186	Intro to DB Systems	4
3	STAT	133	Concepts in Computing	3

sections

section_id	course_id	term	capacity
101	1	FA26	250
102	2	FA26	200
103	1	SP26	180

enrollments

student_id	section_id	status	added_at
3031	101	enrolled	2026-08-20 09:00
3031	102	waitlisted	2026-08-20 09:05
3032	101	enrolled	2026-08-21 10:00
3033	103	enrolled	2026-01-10 08:00

Answer the following questions about the constraints of the Cal Enroll schema.

- 4.1. [3 pt] Each statement below is run independently on the instance shown above. Which statements **fail** because they violate a constraint (as opposed to performing a cascading change)? **Select all that apply.**

- ☐ A. INSERT INTO courses VALUES (4, 'DATA', '101', 'Data Engineering', 4);
- ☐ B. INSERT INTO courses VALUES (5, 'DATA', '102', 'Data Systems', 5);
- ☐ C. INSERT INTO courses (course_id, dept, number) VALUES (6, 'INFO', '258');
- ☐ D. INSERT INTO sections VALUES (110, 9, 'FA26', 300);
- ☐ E. INSERT INTO sections VALUES (111, 1, 'FA26', 300);
- ☐ F. INSERT INTO enrollments VALUES (3031, 103, 'dropped', '2026-09-16 08:00');

- 4.2. Starting from the instance shown above, the following two statements are executed in order.

```
UPDATE sections SET section_id = 201 WHERE section_id = 101;
DELETE FROM courses WHERE course_id = 2;
```

- i. [1 pt] How many rows does the enrollments table contain now?

Solution: 3 rows

student_id	section_id	status	added_at
3031	201	enrolled	2026-08-20 09:00
3032	201	enrolled	2026-08-21 10:00
3033	103	enrolled	2026-01-10 08:00

- ii. [1 pt] What is the value of section_id in the row where student 3031 has status 'enrolled' now?

Solution: 201

- iii. [1 pt] What is the state of the row where student 3031 had status as 'waitlisted'?
- ☐ A. The DELETE on `courses` was rejected, so it remains unchanged
 - ☐ B. It still exists, but the `section_id` is NULL
 - ☒ C. It was deleted due to the cascade on `sections` and `enrollments`
 - ☐ D. None of the above

4.3. [1 pt] Which of the following statements successfully adds a room column to the sections table?

- ☐ A. UPDATE sections SET room = 'Wheeler';
- ☒ B. ALTER TABLE sections ADD COLUMN room VARCHAR(20);
- ☐ C. ALTER TABLE sections SET room = 'Wheeler';
- ☐ D. UPDATE TABLE sections ADD COLUMN room VARCHAR(20);

4.4. The advising office wants to identify students who were unable to enroll in any of the 'FA26' sections they tried to join. Fill in each section, using the answer boxes that follow the query, so that the query returns the student_id of every student who is 'waitlisted' for all 'FA26' sections they tried to enroll in.

```
SELECT e1.student_id
FROM enrollments AS e1
JOIN sections AS s1 ON _____ A _____
WHERE _____ B _____
GROUP BY _____ C _____
HAVING COUNT(*) = (
    SELECT COUNT(*)
    FROM enrollments AS e2
    JOIN sections AS s2 ON _____ D _____
    WHERE _____ E _____
);
```

i. [1/2 pt] Fill in section A

Solution: e1.section_id = s1.section_id

ii. [1/2 pt] Fill in section B

Solution: s1.term = 'FA26'
Alternate: s1.term = 'FA26' AND e1.status = 'waitlisted'

iii. [1 pt] Fill in section C

Solution: `e1.student_id`

iv. [½ pt] Fill in section D

Solution: `e2.section_id = s2.section_id`

v. [1½ pt] Fill in section E

Solution: `s2.term = 'FA26' AND e2.student_id = e1.student_id
AND e2.status = 'waitlisted'`
Alternate: `s2.term = 'FA26' AND e2.student_id = e1.student_id`
(valid if alternate solution was used in 4.4ii)

4.5. [2 pt] Which of the following situations will the Cal Enroll schema (as defined at the start) **fail to prevent? Select all that apply.**

- ☐ A. A student remains enrolled in a section whose course has been deleted from the courses table
- ☐ B. A student has both an 'enrolled' and a 'waitlisted' row for the same section
- ☐ **C. A section has more enrolled students than its capacity**
- ☐ **D. A student enrolls in two different sections of the same course in the same term**

4.6. [1 pt] A developer runs the following UPDATE on the enrollments table:

```
UPDATE enrollments
SET status = 'enrolled'
WHERE status = 'waitlisted' AND added_at < '2026-08-21';
```

Which of the following best describes how PostgreSQL executes this statement?

- ☐ A. It scans each row, and if the row matches the WHERE condition, it updates the row immediately before moving on to the next row.

- ☐ B. It creates a new table with the updated rows and replaces the original enrollments table.
- ☐ C. It locks the entire table, preventing any reads until the UPDATE finishes.
- ☒ D. It first identifies all rows matching the WHERE condition, then updates all of them in a second pass.

Chapter 5: Pedal to the Metal [11 pt]

BearBikes is Berkeley's bike share app. Each ride is one row in the trips relation below.

```
CREATE TABLE trips (  
  id          INT PRIMARY KEY,  
  bike_id     INT,  
  start_station_id INT,  
  end_station_id INT,  
  started_at  TIMESTAMP,  
  ended_at    TIMESTAMP  
);
```

The table `trips` has 18,000,000 rows. Assume that the only indexes are the ones PostgreSQL creates automatically for primary keys and `UNIQUE` attributes. Unless stated otherwise, no `CREATE INDEX` statements have been executed.

5.1. The team must estimate how much disk space `trips` occupies and whether an index is worth creating. Assume each record of `trips` is 120 bytes, and each page is 8,000 bytes and holds 800 bytes of metadata.

i. [1 pt] How many records can fit on one page? (Answer as an integer only)

Solution: 60

Usable space per page = $8000 - 800 = 7200$ bytes

Records per page = $7200 / 120 = 60$

ii. [1 pt] Let us assume your answer for the previous question to be the variable x . How many pages does the table `trips` occupy in terms of x ?

Solution: $18,000,000 / x$

- iii. [2 pt] Assume that `start_station_id` has exactly 600 distinct values and that trips are spread evenly across them (i.e. roughly the same number of trips start at each station), and that rows in `trips` are stored in the order of insertion. An index exists on `start_station_id` and is being used to answer the following query:

```
SELECT *  
FROM trips  
WHERE start_station_id = 42;
```

In the worst case, approximately how many pages will be fetched to answer this query? Explain your answer in under 3 sentences.

Solution: 30,000 pages

Since trips are uniformly distributed over 600 stations, station 42 has $18,000,000 / 600 = 30,000$ matching rows. Given that the table is physically stored in insertion order, the relevant trips are scattered across the table. In the worst case, each trip resides on a different page.

- iv. [2 pt] For this part only, assume that each page holds 80 rows, and the following query returns 24,000 rows:

```
SELECT *  
FROM trips  
WHERE start_station_id = 42;
```

Assume that trips are stored by physically clustering on the `start_station_id`, and a B+ tree index exists on `start_station_id`. Approximately how many pages will be fetched for the query now? Explain your answer in under 3 sentences.

Solution: 300 pages

With physical clustering on the `start_station_id`, all trips for station 42 are stored contiguously on disk. So the number of pages fetched is approximately $24,000 \text{ rows} / 80 \text{ records per page} = 300 \text{ pages}$.

- 5.2. The dashboard's "Yesterday's Rides" panel runs the query below. With no index on `started_at`, query execution took over 10 seconds:

```
SELECT * FROM trips
WHERE started_at >= '2026-09-01' AND started_at < '2026-09-02';
```

This worried the BearBikes team, and they created a B+ tree index named `trips_started_at_idx` on `trips(started_at)`. An engineer now runs the query with `EXPLAIN ANALYZE` to check how it is performing:

```
Index Scan using trips_started_at_idx on trips
  (cost=0.56..4127.90 rows=41000 width=120)
  (actual time=0.052..142.330 rows=38512 loops=1)
    Index Cond: ((started_at >= '2026-09-01 00:00:00'::timestamp)
                  AND (started_at < '2026-09-02 00:00:00'::timestamp))
Planning Time: 0.402 ms
Execution Time: 148.977 ms
```

- i. [1 pt] Which kind of scan does the optimizer use to read `trips`?
- ☐ A. Sequential scan
- ☒ B. Index scan
- ☐ C. Bitmap heap scan
- ii. [2 pt] The engineer now adds `ORDER BY started_at` to the query above and runs `EXPLAIN ANALYZE` again. Which of the following statements hold about the new plan? **Select all that apply.**
- ☒ A. The plan still includes an Index scan using `trips_started_at_idx`.
- ☐ B. The plan will now include a separate Sort step.
- ☐ C. The plan will now include a Sequential scan on `trips`.
- ☒ D. The plan still reports `rows=38512` as the total rows returned.
- ☐ E. The execution time will increase substantially.

iii. [1 pt] Returning to the original query (without `ORDER BY started_at`), the engineer widens the condition in the `WHERE` clause to: `WHERE started_at >= '2020-01-01'`, which now matches nearly all of the 18 million rows. Which query plan would perform best?

- ☐ A. Index Scan using `trips_started_at_idx` on `trips`
Index Cond: (`started_at >= '2020-01-01 00:00:00'::timestamp`)
- ☐ B. Bitmap Heap Scan on `trips`
Recheck Cond: (`started_at >= '2020-01-01 00:00:00'::timestamp`)
-> Bitmap Index Scan on `trips_started_at_idx`
Index Cond: (`started_at >= '2020-01-01 00:00:00'::timestamp`)
- ☒ C. Seq Scan on `trips`
Filter: (`started_at >= '2020-01-01 00:00:00'::timestamp`)
- ☐ D. Sort
Sort Key: `started_at`
-> Seq Scan on `trips`
Filter: (`started_at >= '2020-01-01 00:00:00'::timestamp`)

5.3. [1 pt] For this part only, assume the team has created the four indexes listed in the options below. They realize that inserting new rows into the `trips` table has become slow, and decide to drop indexes that will not affect the following three queries:

Q1: `SELECT * FROM trips WHERE bike_id = 1234;`

Q2: `SELECT * FROM trips`

`WHERE ended_at >= '2026-09-01' AND ended_at < '2026-09-02';`

Q3: `SELECT * FROM trips ORDER BY ended_at DESC LIMIT 10;`

Which indexes should they drop? **Select all that apply.**

- ☐ A. A B+ tree index on `bike_id`
- ☐ B. A B+ tree index on `ended_at`
- ☒ C. A hash index on `ended_at`
- ☒ D. A B+ tree index on `start_station_id`

Chapter 6: Houston, We Have a Query [17 pt]

NASA's Planetary Defense Coordination Office has hired you to build a platform that tracks near-Earth asteroids. Every time an observatory spots an asteroid, a row is added to `sightings`.

```
CREATE TABLE observatories (
  id          INT PRIMARY KEY,
  code        VARCHAR(10) UNIQUE,    -- e.g., "VLT", "ALMA"
  country     VARCHAR(50)
);
```

```
CREATE TABLE asteroids (
  id          INT PRIMARY KEY,
  name        VARCHAR(100),
  diameter_m  INT
);
```

```
CREATE TABLE sightings (
  id          INT PRIMARY KEY,
  asteroid_id INT NOT NULL,
  observatory_id INT NOT NULL,
  seen_at     TIMESTAMP,
  brightness  INT,
  status      VARCHAR(10) CHECK (status IN ('verified', 'pending')),
  FOREIGN KEY (asteroid_id) REFERENCES asteroids(id),
  FOREIGN KEY (observatory_id) REFERENCES observatories(id)
);
```

- 6.1. [2 pt] Let a represent the `asteroids` relation, and s represent the `sightings` relation. Consider the following query:

$$\pi_{a.name, s.brightness}(\sigma_{a.diameter_m > 1000 \wedge s.status = 'verified'}(a \bowtie_{a.id = s.asteroid_id} s))$$

Which of the following rewrites are equivalent to this query? **Select all that apply.**

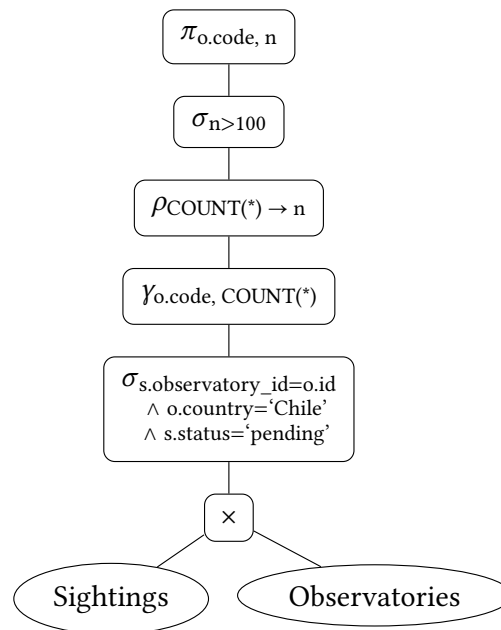
- ☐ A. $\sigma_{a.diameter_m > 1000 \wedge s.status = 'verified'}(\pi_{a.name, s.brightness}(a \bowtie_{a.id = s.asteroid_id} s))$
- ☒ B. $\pi_{a.name, s.brightness}(\sigma_{a.diameter_m > 1000}(a) \bowtie_{a.id = s.asteroid_id} \sigma_{s.status = 'verified'}(s))$
- ☐ C. $\pi_{a.name, s.brightness}(\sigma_{a.diameter_m > 1000 \wedge s.status = 'verified'}(a) \bowtie_{a.id = s.asteroid_id} s)$
- ☒ D. $\pi_{a.name, s.brightness}(\sigma_{s.status = 'verified'}(\sigma_{a.diameter_m > 1000}(a) \bowtie_{a.id = s.asteroid_id} s))$

- 6.2. The table `observatories` has 500 rows; `asteroids` has 40,000 rows; and `sightings` has 2,000,000 rows. Assume that the only indexes are the ones PostgreSQL creates automatically for primary keys and UNIQUE attributes. Unless stated otherwise, no `CREATE INDEX` statements have been executed.

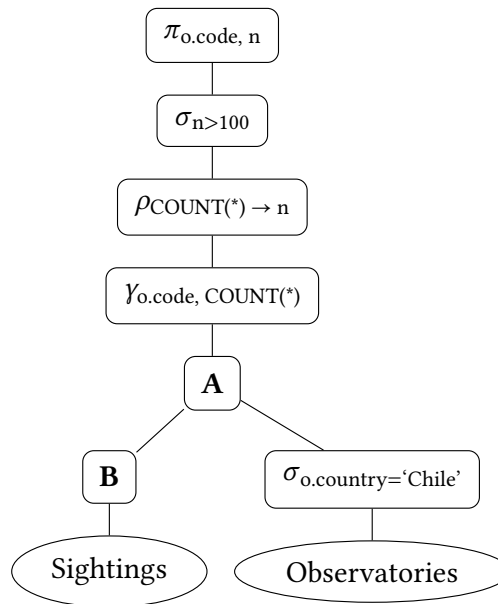
The query below powers a weekly backlog report for observatories in Chile:

```
SELECT o.code, COUNT(*) AS n
FROM sightings AS s, observatories AS o
WHERE s.observatory_id = o.id
      AND o.country = 'Chile'
      AND s.status = 'pending'
GROUP BY o.code
HAVING COUNT(*) > 100;
```

The following is an unoptimized logical query plan:



- i. To optimize the above query plan, we can perform **predicate pushdown**. Fill in the blanks A and B in the optimized plan below:



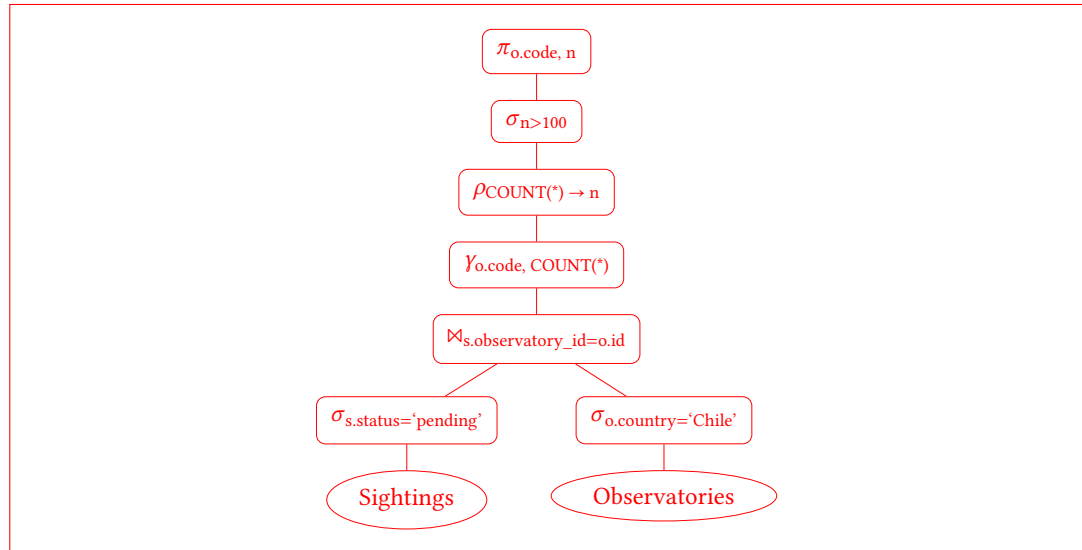
- α) [1 pt] Fill in the blank for A.

Solution: $\bowtie_{s.observatory_id=o.id}$

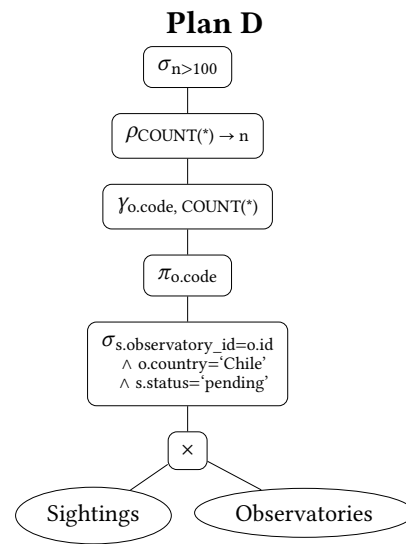
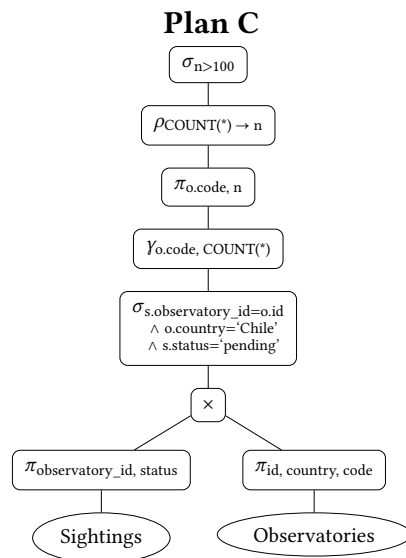
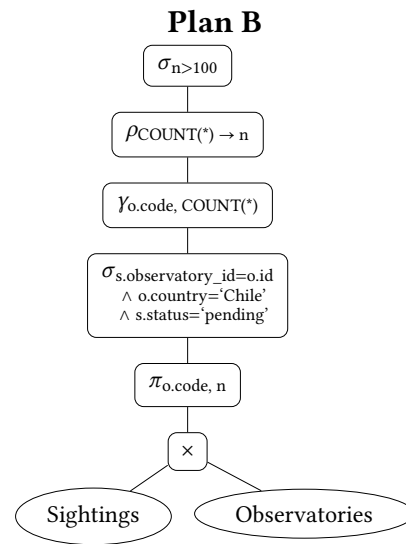
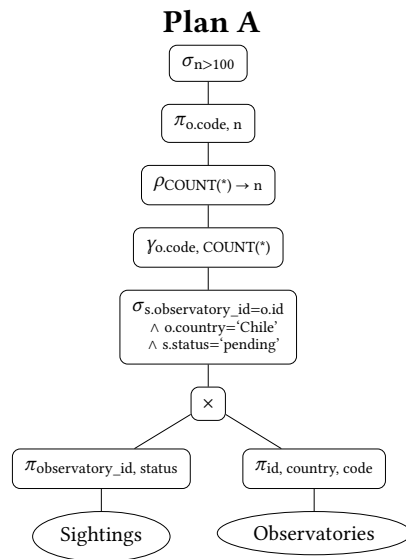
- β) [1 pt] Fill in the blank for B.

Solution: $\sigma_{s.status='pending'}$

The full optimized plan:



- ii. [2 pt] Starting from the unoptimized plan, which of the following query plans correctly apply **projection pushdown**? **Select all that apply.**



☒ A. Plan A

☐ B. Plan B

☐ C. Plan C

☒ D. Plan D

6.3. For each query below, select the join algorithm the optimizer is most likely to choose. For reference, the table sizes are: `observatories` (500 rows), `asteroids` (40,000 rows), `sightings` (2,000,000 rows). Assume that the only indexes are the ones PostgreSQL creates automatically for primary keys and UNIQUE attributes, and that the tables `observatories` and `asteroids` fit in memory, while `sightings` does not.

i. [1 pt] `SELECT o.code, s.brightness`
`FROM sightings AS s`
`INNER JOIN observatories AS o ON s.observatory_id = o.id`
`WHERE s.id = 987654;`

☐ A. Hash join

☐ B. Merge join

☒ C. Index nested loop join

ii. [1 pt] `SELECT s.seen_at, s.brightness`
`FROM sightings AS s`
`INNER JOIN observatories AS o ON s.observatory_id = o.id`
`WHERE o.code = 'VLT';`

☒ A. Hash join

☐ B. Merge join

☐ C. Index nested loop join

iii. [1 pt] `SELECT s.seen_at, s.brightness`
`FROM sightings AS s`
`INNER JOIN observatories AS o ON s.observatory_id = o.id`
`LIMIT 10;`

☐ A. Hash join

☐ B. Merge join

☒ C. Index nested loop join

iv. [1 pt] `SELECT a.name, s.brightness`
`FROM asteroids AS a`
`INNER JOIN sightings AS s ON a.id = s.asteroid_id`
`ORDER BY a.id;`

☐ A. Hash join

☒ B. Merge join

☐ C. Index nested loop join

v. [1 pt] `SELECT a.name, s.brightness
FROM asteroids AS a
INNER JOIN sightings AS s ON a.id = s.asteroid_id
ORDER BY a.name;`

☒ A. Hash join

☐ B. Merge join

☐ C. Index nested loop join

6.4. You are trying to estimate the intermediate table sizes different queries will produce. We use the same table sizes: `observatories` (500 rows), `asteroids` (40,000 rows), `sightings` (2,000,000 rows).

i. [1 pt] Consider the following query:

```
SELECT DISTINCT o.code  
FROM observatories o  
JOIN sightings s ON o.id = s.observatory_id;
```

What is the maximum number of rows this query can return?

Solution: 500 rows

ii. [1 pt] Consider the following query:

```
SELECT DISTINCT a.name  
FROM asteroids AS a  
JOIN sightings AS s ON a.id = s.asteroid_id;
```

What is the minimum number of rows this query can return?

☐ A. 0

☒ B. 1

☐ C. 500

☐ D. 40,000

iii. Consider the following query:

```
SELECT o.code, a.name, s.brightness
FROM sightings AS s
JOIN observatories AS o ON s.observatory_id = o.id
JOIN asteroids AS a ON s.asteroid_id = a.id
WHERE o.country = 'Chile' AND a.diameter_m > 1000;
```

Suppose 25 of the 500 observatories are in 'Chile', and 400 of the 40,000 asteroids have `diameter_m > 1000`. Assume sightings are uniformly distributed across all observatories and asteroids (i.e., each observatory contributes 4,000 sightings, and each asteroid is sighted 50 times).

α) [1 pt] If the optimizer first joins `sightings` with the observatories in Chile, how many rows are in the intermediate result (before joining with asteroids)?

- ☐ A. 1,000
- ☐ B. 20,000
- ☒ C. 100,000
- ☐ D. 1,000,000

β) [2 pt] Which join order produces the smaller intermediate result before the final join?

- ☐ A. $(\text{sightings} \bowtie \sigma_{\text{country}='Chile'}(\text{observatories})) \bowtie \sigma_{\text{diameter_m}>1000}(\text{asteroids})$
- ☒ B. $(\text{sightings} \bowtie \sigma_{\text{diameter_m}>1000}(\text{asteroids})) \bowtie \sigma_{\text{country}='Chile'}(\text{observatories})$
- ☐ C. Both orders produce the same intermediate size
- ☐ D. Intermediate sizes cannot be determined with the available information

γ) [1 pt] What is the maximum number of rows the three-way join query can return?

- ☐ A. 1,000
- ☒ B. 20,000
- ☐ C. 100,000
- ☐ D. 200,000

Chapter 7: Congratulations! [0 pt]

Congratulations! You have completed this exam.

- Make sure that you have written your Student ID number on every 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 ≤ 10 minutes remain, please sit quietly until the exam concludes.

Assumptions

[Optional, 0 pts] Were there any ambiguities in the exam? If so, record the ambiguity and the assumptions that you made in the box below. If you submit a regrade request regarding any ambiguities, we will refer to the box below.

Comments, Concerns, etc.

[Optional, 0 pts] Did you notice anything suspicious during the exam? Please note in the box below. Where were you sitting, and what did you notice?

[Optional, 0 pts] Any other comments, concerns, qualms, queries, or suggestions?

For Fun

[Optional, 0 pts] Who would win in a 100 meter race, prime Usain Bolt or a silverback gorilla? Justify your answer if you want.

[Optional, 0 pts] Use this space to draw your favorite Data 101/CS 187 moment!