Data 101: Data Engineering
Midterm Exam Solutions

UC Berkeley, Fall 2024

October 16, 2024

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 **75 points** spread over **4 questions** (including the Honor Code), and must be completed in the 110-minute time period on October 16, 2024, 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: I didn't sign up for all this reading! [28 pt]

We explore a simplified version of a local library system. Broadly, libraries track books which are stored at various locations. Our library database stores data about the books, such as their titles, authors, and ISBNs (a unique number for each each book). **Note:** The schema for this question is included in the exam reference packet, but you do not need to review it until Question 1.4.

1.1. The ISBN of each book is a *unique* 13 digit number, formatted as `978-6-543-21012-3`. The `books` table contains an attribute called `isbn` which is currently a variable length string (the `TEXT` type).

i. [1 pt] If we store the ISBN as a **string**, how many bytes would we need to store this value, including the 4 dashes (`-`)? **Round up to the nearest multiple of 4 bytes**, e.g., 9 bytes would round up to 12.

________ bytes

Solution: 13 + 4 + 1 = 18 bytes, where 1 byte is used for the null terminator of strings (beyond the scope of the course). We asked for multiples of 4 bytes, so the answer is **20 bytes**.

ii. [1 pt] Suppose that we did not need the fancy formatting and instead used a **numeric data type** to store the ISBNs. Which **numeric** data type would be most appropriate? Consider that 10^{13} is much larger than 2^{32}, and $2^{10} = 1024 \approx 10^3$.

○ `int16` ○ `int32` [✓ `int64`] ○ `float32` ○ `double64`

iii. [1 pt] Given your responses above, which would be the most appropriate data type (strings or numeric) to choose for the `isbn` attribute? Fill in the appropriate bubble and **justify** your choice in **one sentence**.

A ○ **string** / ○ **numeric** data type is more appropriate for `isbn`.

Solution: Either answer is correct as long as the justification matches. A string data type is more space, but could be appropriate because it communicates that an ISBN is not an attribute which you can add or subtract like an integer. Otherwise, an integer may be more appropriate simply because it is a more efficient storage mechanism.

iv. [2 pt] Complete the function that converts the ISBN string to an **INTEGER** data type.

You can assume that all ISBNs follow the form: XXX-X-XXX-XXXXX-X where each X is a digit, stored in a column called `isbn`, e.g., `'978-6-543-21012-3'`. You conversion should ensure that `'978-6-543-21012-3'` is coerced to `9786543210123`.

Notes: You should use `SUBSTRING`, `REPLACE`, `REGEXP_REPLACE`, `SUBSTR` or other valid string manipulation functions. It may be helpful to review the exam reference packet **(Sections 6–7)** for string functions. Recall that `::dtype` casts to data type `dtype`.

____::INTEGER

Solution:

```
REPLACE(isbn, '-', '')::INTEGER
REGEXP_REPLACE(isbn, '-', '', 'g')::INTEGER
```

For the next two parts, consider this abbreviated schema for our library. The `users` and `checkouts` tables respectively describe who is using the library and which books they have currently or previously checked out.

```
users(id, first_name, last_name, email, phone_number, joined_date)
checkouts(id, user_id, book_id, location_id,
    checkout_date, return_date, due_date)
```

Users can check out and return books from a specific location. A book which is actively checked out will have a `checkout_date` and a `due_date`, but a `NULL` `return_date`. All three of these attributes are the SQL `DATE` type.

1.2. Write a query that will return the users who have **3 or more active checkouts**, sorted by the user who has the most currently checked out books. Your query should compute the user (`user_id`, `first_name`, `last_name`), `n_active_checkouts` (the total number of books the user currently has checked out), and `next_due_date` (the earliest due date for any book that the user currently has checked out; technically this could be in the past for an overdue book). Example output:

user_id	first_name	last_name	n_active_checkouts	next_due_date
1	Jonathan	Doe	4	2024-09-15
2	Rebecca	Malik	4	2024-09-16
3	CW	Cano	3	2024-09-17

i. [2 pt] First, write a CTE `active_checkouts` that returns the rows representing currently checked out books.

```
WITH active_checkouts AS (

  SELECT * FROM checkouts

  WHERE ____________________________________________________
) ...;
```

Solution:

```
WITH active_checkouts AS (
    SELECT *
    FROM checkouts c
    WHERE c.return_date IS NULL
) ...;
```

ii. [7 pt] Next, complete the rest of the query. Assume that your CTE from the previous part is correct. You do not need to worry about breaking ties in the ordering.

Note: You may not need all blanks.

```
WITH active_checkouts AS (...)
SELECT
  u.id AS user_id,

   ___________________________________________________________

   ___________________________________________________________

   ___________________________________________________________

  FROM  users u

  ________ JOIN active_checkouts ac ON _________________________

GROUP BY _____________________________________________________

HAVING _______________________________________________________

ORDER BY ____________________________________________________;
```

Solution:

```
WITH active_checkouts AS (
    SELECT *
    FROM checkouts c
    WHERE c.return_date IS NULL
)
SELECT
    u.id AS user_id,
    u.first_name,
    u.last_name,
    COUNT(ac.id) AS n_active_checkouts,
    MIN(ac.due_date) AS next_due_date
FROM users u
JOIN active_checkouts ac ON ac.user_id = u.id
GROUP BY u.id, u.first_name, u.last_name
HAVING COUNT(ac.id) >= 3
ORDER BY n_active_checkouts DESC;
```

1.3. [8 pt] We'd now like to determine the most popular days of the week each book is checked out. Now, consider the `books` table, in addition to the previous `checkouts` table.

```
books(id, title, author, isbn, publication_year)
checkouts(id, user_id, book_id, location_id,
    checkout_date, return_date, due_date)
```

Complete the following query to show each book and its **rank**, by the number of checkouts on each **day of week** the book was checked out. The table should be sorted by the number of checkouts (largest first) on a given day of week and include the following columns:

- `id` and `title` of the book
- `dow`, a numeric day of the week the book was checked out.
  - `dow` is a **number** representing the day of week (0 to 6) that a book was checked out. `EXTRACT(DOW FROM '2024-10-16')` will return 3 for Wednesday.
- `num_checkouts`, the number of times the book was checked out on the day `dow`
- `ranking`, the rank of the book by number of checkouts, where 1 means most checkouts on the day `dow`.

id	title	dow	num_checkouts	ranking
9	Brave New World	3	2	1
12	The Grapes of …	1	2	1
		…		
13	The Adventures	0	1	1
18	Don Quixote	1	1	2
2	To Kill a Mocki…	1	1	2

Note: You may not need all blanks.

```
SELECT  b.id, b.title AS title,

   EXTRACT(DOW FROM c.checkout_date) AS dow,

   ______________________________________________ AS num_checkouts,

   ___________ OVER ( ____________________________________________

   ___________________________________________________ ) AS ranking
FROM books b

JOIN ____________________________________________________________

GROUP BY ________________________________________________________

ORDER BY ______________________________________________________ ;
```

Solution:

```
SELECT
    b.id,
    b.title,
    EXTRACT(DOW FROM c.checkout_date) AS dow,
    COUNT(c.id) AS num_checkouts,
    RANK() OVER (PARTITION BY EXTRACT(DOW FROM c.checkout_date)
                    ORDER BY COUNT(c.id) DESC) AS ranking
FROM Books b
JOIN Checkouts c ON b.id = c.book_id
GROUP BY b.id, b.title, dow
ORDER BY num_checkouts DESC;
```

For the final few parts of this question, consider this expanded schema for our library. The `books`, `checkouts`, and `users` tables are included again for clarity.

```
locations(id, name, address, phone_number)
book_locations(id, book_id, location_id,
  total_copies, available_copies)
books(id, title, author, isbn, publication_year)
checkouts(id, user_id, book_id, location_id,
    checkout_date, return_date, due_date)
users(id, first_name, last_name, email, phone_number, joined_date)
```

Note: Please read the full database schema in the exam reference packet (Sec. 1) before continuing. It includes information about the data types and constraints. We have intentionally removed all primary key, foreign key, and indexes from this schema.

Considering the provided schema, answer the questions below. Note: `a.b` is column `b` on table `a`.

1.4. [2 pt] Which columns **could** be marked as **primary keys**? Select all that apply.

- ☐ **A.** `users.id`
- ☐ B. `users.name`
- ☐ **C.** `books.id`
- ☐ **D.** `books.isbn`
- ☐ E. `book_locations.book_id`
- ☐ **F.** `checkouts.id`
- ☐ G. `checkouts.user_id`
- ☐ H. None of the above

1.5. [2 pt] Which columns **could** be marked as **foreign keys**? Select all that apply.

- ☐ A. `users.id`
- ☐ B. `users.name`
- ☐ C. `books.id`
- ☐ D. `books.isbn`
- ☐ **E.** `book_locations.book_id`
- ☐ F. `checkouts.id`
- ☐ **G.** `checkouts.user_id`
- ☐ H. None of the above

1.6. [2 pt] Consider these hypothetical scenarios that make use of the library database. What **additional** columns should we create an index for? Assume we have correctly made an index for all columns that are a primary key. For each scenario write the appropriate column name as `table_name.column_name`.

i. Searching for books by a title ____**`books.title`**____

ii. Easily show a user all the books they currently have checked out ____**`checkouts.user_id`**____

Chapter 2: Timestamping Reviews [20 pt]

Let's consider how to store combined date and time values (or simply, "date/time values") in two choices of attributes: the 8-byte SQL `TIMESTAMP` data type, in Coordinated Universal Time (UTC), and a 4-byte **epoch time** `INTEGER` data type. **Note**: Please see the **exam reference packet (Sec. 3)** for relevant SQL `TIMESTAMP` functions and conversions.

Your developer teammate has designed and implemented the `reviews_epoch` table below for storing user reviews of businesses on Yelp, a social networking business review site. They store `created_at` as a 4-byte integer, representing the **epoch time** at which the review was written.

```
CREATE TABLE reviews_epoch (
  review_id CHAR(22) PRIMARY KEY,
  user_id CHAR(22), business_id CHAR(22), stars INTEGER,
  created_at INTEGER,   /* epoch time */
  text TEXT
);
```

review_id	user_id	business_id	stars	created_at	text
SZp6…	wl5f…	jBpF…	4	1451611911	…
9USm…	rEYm…	5EpF…	5	1510870342	…
5DbC…	w9R6…	jkGQ…	1	1451611911	…

Sample rows of `reviews_epoch`

2.1. [3 pt] Convert the epoch times above to SQL timestamps by constructing the `dt_conversions` view with two columns: each epoch time in `created_at` and its corresponding conversion to a SQL timestamp (UTC). **Do not include duplicates**.

epoch	timestamp_utc
1451611911	2016-01-01 01:31:51
1510870342	2017-11-16 22:12:22

(output for sample rows above)

Note: Use the specific SQL `TIMESTAMP` functions provided in the **exam reference packet (Sec. 3)**. You may not need all blanks.

```
CREATE VIEW dt_conversions AS (

  SELECT

     ____________________________________________________________

     ____________________________________________________________

  FROM reviews_epoch

);
```

Solution:

```
CREATE VIEW dt_conversions AS (
  SELECT DISTINCT created_at AS epoch,
       TO_TIMESTAMP(created_at) AT TIME ZONE 'UTC' AS timestamp_utc
  FROM reviews_epoch
);
```

Your teammate argues that epoch times are more efficient than any other representation for date/time—for both storage and query performance reasons. Let's explore this claim.

2.2. [1 pt] Suppose there are 7 million records in `reviews_epoch`. How much storage does it take to store the table where date/time values are SQL timestamps, compared to epoch times? Below, fill in the bubble and blank that make the statement true. Additionally, show your work for computing storage size in **megabytes (MB)**.

Storing the `created_at` column as SQL timestamps takes a total of __________ MB
◯ **more** / ◯ **less** space than the equivalent column as 4-byte integers.

Solution: (7×10^6) records $\times (8-4)$ B $\times 10^{-6}$(MB/B) $= 28$

Storing the `created_at` column as 8-byte SQL timestamps takes a total of **28 MB more** space than the equivalent column as 4-byte integers.

2.3. [4 pt] Your teammate ran `CREATE INDEX epoch_idx ON reviews_epoch(created_at);` then ran the below query, which finds the reviews that were written on January 19, 2022:

```
SELECT * FROM reviews_epoch
WHERE created_at >= EXTRACT('EPOCH' FROM '2022-01-19'::TIMESTAMP)
    AND created_at < EXTRACT('EPOCH' FROM '2022-01-20'::TIMESTAMP);
```

Use the below `EXPLAIN ANALYZE` output of this query to answer the following questions.

```
                             QUERY PLAN
------------------------------------------------------------------------------
 Gather  (cost=1000.00..170284.29 rows=34950 width=93)
         (actual time=74.018..1231.257 rows=1190 loops=1)
   Workers Planned: 2,  Workers Launched: 2
   ->  Parallel Seq Scan on reviews_epoch
         (cost=0.00..165789.29 rows=14562 width=93)
        (actual time=66.666..1222.807 rows=397 loops=3)
         Filter: (((created_at)::numeric >= 1642550400.000000
                 AND ((created_at)::numeric < 1642636800.000000))
         Rows Removed by Filter: 2329603
 Planning Time: 0.145 ms       Execution Time: 1231.320 ms
```

i. What kind(s) of table scan does the optimizer use? Select all that apply.

- [✓ **Sequential Scan**]
- ☐ Heap Scan
- ☐ Index Scan

ii. What is the name of the index used by the optimizer? Write N/A if not applicable.

N/A

iii. What is the actual number of rows in the result?

1190

iv. Are the rows in the result sorted by the `created_at` attribute?

- ◯ Yes
- [✓ **Not necessarily**]

Given the above analysis, you wonder how a table that uses SQL timestamps may compare.

2.4. [5 pt] Fill in the below lines to create the `reviews_ts` table as a copy of `reviews_epoch` with all date/time epoch times converted to SQL timestamps (UTC). Assume you have access to the `dt_conversions` view from Question 2.1. Finally, specify the equivalent primary key constraint on `review_id`.

Note: You may not need all blanks.

```
CREATE __________________ reviews_ts AS (
  SELECT
     review_id, user_id, business_id, stars, text,
     ______________________________________________  AS created_at
  FROM reviews_epoch AS r, dt_conversions AS dtc
  WHERE ____________________________________________________________
);
ALTER __________________ review_ts
  ADD PRIMARY KEY ( ____________________________________________ );
```

Solution:

```
CREATE TABLE reviews_ts AS (
    SELECT review_id, user_id, business_id, stars, text,
            dtc.timestamp_utc AS created_at
    FROM reviews_epoch AS r, dt_conversions AS dtc
    WHERE r.created_at = dtc.epoch
);

ALTER TABLE review_ts ADD PRIMARY KEY (review_id);
```

2.5. [4 pt] After constructing an index on this new SQL timestamp column, you revisit query performance for the task in Question 2.3. You write a query that finds the reviews in `review_ts` that were written on January 19, 2022:

```
SELECT * FROM reviews_ts
WHERE created_at >= '2022-01-19 00:00:00'::TIMESTAMP
    AND created_at < '2022-01-20 00:00:00'::TIMESTAMP;
```

Use the below `EXPLAIN ANALYZE` output of this query to answer the following questions.

```
                                   QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on reviews_ts  (cost=30.66..5081.32 rows=1388 width=85)
                            (actual time=1.214..124.090 rows=1190 loops=1)
   Recheck Cond:
        ((created_at >= '2022-01-19 00:00:00'::TIMESTAMP)
        AND (created_at < '2022-01-20 00:00:00'::TIMESTAMP))
   Heap Blocks: exact=1169
   ->  Bitmap Index Scan on created_at_ts_idx
       (cost=0.00..30.31 rows=1388 width=0)
       (actual time=0.831..0.832 rows=1190 loops=1)
         Index Cond: ((created_at >= '2022-01-19 00:00:00'::TIMESTAMP)
                     AND (created_at < '2022-01-20 00:00:00'::TIMESTAMP))
 Planning Time: 3.428 ms        Execution Time: 124.200 ms
```

i. What kind(s) of table scans does the optimizer use? Select all that apply.

☐ Sequential Scan
☐ **Heap Scan**
☐ **Index Scan**

ii. What is the name of the index used by the optimizer? Write N/A if not applicable.

created_at_ts_idx

iii. What is the actual number of rows in the result? **1190**

iv. Are the rows in the result sorted by the `created_at` attribute?

○ Yes
[✓ **Not necessarily**]

2.6. [1 pt] Is the query in Question 2.3 faster or slower than the query in Question 2.5?

○ Faster [✓ **Slower**] ○ Comparable

2.7. [2 pt] Without mentioning the storage or query performance reasons above, list **one** additional reason to prefer the SQL timestamp data type over the epoch time integer.

Solution: Possible answers:

- Access to timestamp SQL functions;
- Timestamps are more human-readable.

Chapter 3: Ed Discussion, but for Algebra [14 pt]

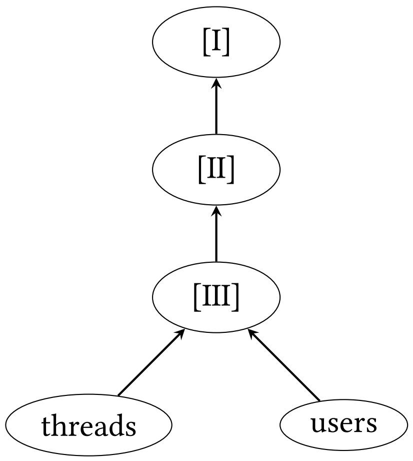
Consider a simplified version of Ed Discussion, where users start new threads, write comments on existing threads, and make hearts (i.e., likes or upvotes) on threads or comments within threads. We share the relevant parts of the database schema below; please see the **exam reference packet (Sec. 2)** for the full description.

```
users(id, email, display_name, hearts_made)
threads(id, thread_type, hearts, user_id, title, text)
comments(id, thread_id, hearts, user_id)
```

3.1. Suppose we write the below query:

```
SELECT display_name AS user_name,
       COUNT(*) AS threads_posted
FROM threads, users
WHERE users.id = threads.user_id
GROUP BY display_name;
```

The query optimizer then produces the execution plan on the right, according to SQL query semantics. Fill in the blanks below.



i. [3 pt] What extended relational operators should be in the nodes marked [I], [II], and [III]?

[I] ○ π [✓ ρ] ○ σ ○ $\bowtie$ ○ $\times$ ○ γ

[II] ○ π ○ ρ ○ σ ○ $\bowtie$ ○ $\times$ [✓ γ]

[III] ○ π ○ ρ ○ σ [✓ $\bowtie$] ○ $\times$ ○ γ

ii. [3 pt] For each relational operator you selected above, write its respective subscript according to the original SQL query, e.g. join conditions, selected attributes, etc. If there are no subscripts, write N/A.

[I] user_name, threads_posted

[II] display_name, COUNT(*)

[III] users.id=threads.user_id

3.2. Select all relational algebra expressions that satisfy each description.

i. [2 pt] Get the titles of threads that do not have any comments.

☐ A. $\pi_{\text{title}}(\text{threads}) \cap \pi_{\text{title}}(\text{threads} \bowtie_{\text{threads.id = thread\_id}} \text{comments})$

☐ **B. $\pi_{\textbf{title}}(\textbf{threads}) - \pi_{\textbf{title}}(\textbf{threads} \bowtie_{\textbf{threads.id = thread\_id}} \textbf{comments})$**

☐ C. $\pi_{\text{title}}(\text{threads} - \text{threads} \bowtie_{\text{threads.id = thread\_id}} \text{comments})$

☐ D. $\pi_{\text{title}}(\text{threads} \bowtie_{\text{threads.id != thread\_id}} \text{comments})$

☐ E. None of the above

ii. [2 pt] Get the display name of the "original poster" for each thread with more than 5 hearts (ignore comment hearts). An original poster for a thread is the user who started the thread.

☐ **A. $\pi_{\textbf{display\_name}}\left(\sigma_{\textbf{hearts>5}}\left(\textbf{threads} \bowtie_{\textbf{user\_id = users.id}} \textbf{users}\right)\right)$**

☐ **B. $\pi_{\textbf{display\_name}}\left(\sigma_{\textbf{hearts>5 AND user\_id = users.id}}\left(\textbf{threads} \times \textbf{users}\right)\right)$**

☐ **C. $\pi_{\textbf{display\_name}}\left(\left(\sigma_{\textbf{hearts>5}}(\textbf{threads}) \bowtie_{\textbf{user\_id = users.id}} \textbf{users}\right)\right)$**

☐ D. $\pi_{\text{display\_name}}\left((\sigma_{\text{hearts>5}}(\text{threads}) \bowtie \text{users})\right)$

☐ E. None of the above

3.3. [4 pt] Consider the following relational algebra expression, where the expression **cond** is defined as threads.user_id = users.id AND threads.id = comments.thread_id.

$$\gamma_{\text{users.id,threads.id,COUNT}(*)}\left(\sigma_{\textbf{cond}}\left(\text{users} \times \sigma_{\text{threads.hearts>0}}(\text{threads}) \times \text{comments}\right)\right)$$

What is this relational algebra expression doing? Write **one sentence in plain English** using the provided Ed database schema. Do **not** use any relational algebra terminology.

Note: Please see the **exam reference packet (Sec. 2)** for the full database schema.

Solution: For threads that have at least one heart, find the number of comments and the user who created the thread.

Chapter 4: Hearts Branch from Branches [12 pt]

We continue our exploration of a simplified version of Ed Discussion. When comments are written in reply to other comments, they create a comment tree. We share the relevant parts of the database schema below; please see the **exam reference packet (Sec. 2)** for the full description.

```
threads(id, thread_type, hearts, user_id, title, text)
comments(id, thread_id, hearts, user_id)
child_comments(comment_id, child_id)
```

4.1. [1 pt] Each thread must be tagged with a `thread_type`, which is one of three possible text strings: `Question`, `Post`, and `Megathread`. Assuming all threads have non-NULL thread types, what is theoretically **the minimum number of bits** needed to encode (i.e., represent) the `thread_type` attribute? Ignore any practical limits of SQL.

○ 1 bit [✓ 2 bits] ○ 3 bits ○ 8 bits ○ 16 bits

4.2. [3 pt] The below recursive query gets the **comment tree** for comment ID 22.

Note: See the **exam reference sheet** for a full description of comment trees **(Sec. 2)** and an excerpt from the PostgreSQL recursive query reference **(Sec. 4)** .

comment_id	child_id
12	13
21	22
22	23
22	24

child_comments

comment_id
22
23
24

(query output)

```
WITH RECURSIVE tree(comment_id) AS (
  ( SELECT comment_id FROM child_comments WHERE comment_id = 22 )
  UNION
  ( SELECT ____(i)____ FROM child_comments
    JOIN tree ON child_comments.___(ii)____ = tree.___(iii)___ )
)
SELECT * FROM tree;
```

For each of the blanks above, select the correct option so that running the recursive query on the example `child_comments` table produces the corresponding output table.

i. ○ `id` ○ `comment_id` [✓ `child_id`]

ii. ○ `id` [✓ `comment_id`] ○ `child_id`

iii. ○ `id` [✓ `comment_id`] ○ `child_id`

4.3. [8 pt] Users can make hearts (i.e., likes or upvotes) on threads and comments. Write a query that computes, for each thread ID, the total number of hearts made on that thread, counting both the hearts on the original post text and hearts on all comments on the thread. An example output with sample tables is shown below.

Note: You do not need to/**should not** write a recursive query. You may not need all blanks.

id	thread_id	hearts	…
11	100	5	…
12	100	3	…
13	100	2	…
21	200	0	…
22	200	1	…
23	200	2	…
24	200	4	…

comments

id	thread_type	hearts	…
100	…	0	…
200	…	20	…

threads

thread_id	sum
200	27
100	10

(query output)

```
WITH thread_hearts AS (

    ( SELECT ______________________________________

       ______________________________________

       ______________________________________ )

    UNION ALL

    ( SELECT ______________________________________

       ______________________________________

       ______________________________________ )
)

SELECT ______________________________________

______________________________________

FROM thread_hearts

______________________________________ ;
```

Solution:

```
WITH thread_hearts AS (
  SELECT id AS thread_id, hearts FROM threads
UNION ALL
  SELECT thread_id, comments.hearts
  FROM comments, threads
  WHERE comments.thread_id = threads.id
)
SELECT thread_id, SUM(hearts)
FROM thread_hearts
GROUP BY thread_id;
```

Chapter 5: 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 for 1 point.
- 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 design a new Data 101 course logo!

This page is intentionally left blank.