Data 101: Data Engineering
Final Exam Solutions

UC Berkeley, Fall 2024

December 17, 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 **101** ~~125~~ points spread over **6 questions** (including the Honor Code), and must be completed in the 170-minute time period on December 17, 2024, 3:10pm – 6:00pm 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: What's the weather? [27 pt]

The United States Federal Emergency Management Agency (FEMA) maintains a database of federally declared disasters. We explore the two tables in this database over the next few questions.

The `incidents` table contains incident records for all federally declared disasters. A sample record is shown on the right.

Note: See the **exam reference packet** for a full description of the FEMA database schema **(Sec. 1)**.

```
-[ RECORD 1 ]-----+-----------------
id                | fbb556
disaster_num      | 4829
incident_type     | Hurricane
begin_date        | 2024-09-24
end_date          |
fips_state        | 45
declaration_title | HURRICANE HELENE
```

1.1. [2 pt] Suppose there are 70,000 records in `incidents` occupying a total of 5.6MB, ignoring metadata. What is the size of each record in **bytes**? Show your work.

_____ bytes

Solution: $\frac{5.6\text{MB}}{70,000\text{records}} = \frac{5.6 \times 10^6\text{B}}{70 \times 10^3\text{records}} = \mathbf{80\ bytes}$

1.2. [2 pt] Consider the following attribute description:

> `fips_state CHAR(2)`: FIPS two-digit numeric code used to identify the United States, the District of Columbia, US territories, outlying areas of the US and freely associated states. FIPS codes range from 00 to 99.

The attribute `fips_state` has been declared as `CHAR(2)`, a fixed-length, 3-byte string type that can represent two characters. Your friend argues that a 1-byte numeric type (specifically, an integer with range -128 to 127) is more appropriate.

Based on the description above, would be the more appropriate type (string or 1-byte numeric) to choose for the `fips_state` attribute? Fill in the appropriate bubble and **justify** your choice in **one sentence**.

A ◯ **string** / ◯ **1-byte numeric** data type is more appropriate for `fips_state`.

Reason:

Solution: A string data type is more space, but could be appropriate because it communicates that a FIPS state code is not an attribute which you can add or subtract like an integer. Otherwise, a (1-byte) integer may be more appropriate simply because it is a more efficient storage mechanism.

Note: Please read the full database schema in the exam reference packet (Sec. 1) before continuing. We have included relevant attribute descriptions below:

- `disaster_num INTEGER`: Sequentially assigned number used to designate an event or incident declared as a disaster.
- `incident_type TEXT`: The primary or official type of incident, e.g., fire, flood, etc.
- `begin_date DATE`: Date the incident began.

1.3. You are asked to design dbt-style tests. For each statement below, write a query that returns an empty table if the statement is true for all records; otherwise, return the `id`s of records for which the statement is **false**. You may not need all blanks.

i. [2 pt] *All incidents have a begin date.*
Write a query that returns the IDs of incidents that have a NULL begin date, or an empty table if no records match.

```
SELECT id

FROM incidents

WHERE _____;
```

Solution:

```
SELECT id
FROM incidents
WHERE begin_date IS NULL;
```

ii. [6 pt] *All incidents with the same disaster number have the same begin date.*
Write a query that returns the IDs of incidents that have the same disaster number but *different* begin dates, or an empty table if no records match.

Note: Equality operators (=, `!=`) work with `date` type attributes.

```
WITH min_dates AS (

  SELECT _____

         _____

          OVER ( _____ ) AS val

  FROM incidents
)
SELECT id
FROM incidents
JOIN min_dates

  ON _____

WHERE _____;
```

Solution:

```
WITH min_dates AS (
  SELECT DISTINCT disaster_num, -- DISTINCT optional but more efficient
    MIN(begin_date)
      OVER (PARTITION BY disaster_num) AS val
  FROM incidents
)
SELECT id
FROM incidents
JOIN min_dates ON min_dates.disaster_num = incidents.disaster_num
WHERE begin_date != min_dates.val;
```

1.4. [2 pt] Consider the functional dependency $X \rightarrow Y$. Specify X and Y such that this functional dependency is equivalent to the statement in Question 1.3(ii).

X:
- ◯ A. id
- ◯ **B. disaster_num**
- ◯ C. begin_date

Y:
- ◯ A. id
- ◯ B. disaster_num
- ◯ **C. begin_date**

1.5. [2 pt] According to the National Oceanic and Atmospheric Administration (NOAA), incidents like Hurricanes and Tropical Storms are given short, distinctive names to avoid confusion and streamline communications. What is the Levenshtein distance between the strings *Irene* and *Helene*?

Levenshtein distance: 3

1.6. [4 pt] The incident type (`incident_type` attribute) is the officially designated type of a given incident, such as fire, flood, and so on. Find the number of unique disasters (i.e., disaster numbers) per incident type, sorted by most unique disasters first, *[exam clarification]* meaning sorted by the highest number of unique disasters with that incident type.

incident_type	count
Fire	1648
Severe Storm	1097
...	...

Sample output on right. You may not need all blanks.

```
SELECT _____
       _____
FROM incidents
_____
_____;
```

Solution:

```
SELECT incident_type, COUNT(DISTINCT disaster_num)
FROM incidents
GROUP BY incident_type
ORDER BY COUNT(DISTINCT disaster_num) DESC;
```

1.7. [2 pt] There are 26 incident types (Fire, Flood, Hurricane, Tropical Storm, etc.) Assuming that all records have non-NULL incident types, what is theoretically the minimum number of bits needed to encode (i.e., represent) the `incident_type` attribute? Ignore any practical limits of SQL.

Note: See the **exam reference packet** for a base-2 logarithm lookup table **(Sec. 10.1)**.

◯ 4 bits ◯ **5 bits** ◯ 8 bits ◯ 26 bits ◯ None of these

1.8. [5 pt] Suppose that all incidents have an begin date, and all incidents with the same disaster number have the same begin date (see Question 1.3). However, not all incidents have an incident **end date**.

You decide to **impute missing end dates using the average duration (in days)** of disasters by incident type, which has been already computed for you in a view, `avg_durations`.

incident_type	avg
Fire	15
Flood	13
Hurricane	13
…	…

`avg_durations`

Write a query that uses `avg_durations` to create a column, `impute_end_date`, that takes the end date (if it exists) or imputes the end date as begin date plus average duration of incident per type:

disaster_num	incident_type	begin_date	end_date	impute_end_date
4829	Hurricane	2024-09-24		2024-10-06
4837	Tropical Storm	2024-09-15	2024-09-19	2024-09-19
5534	Fire	2024-09-01		2024-09-15
5534	Fire	2024-09-01		2024-09-15
4037	Hurricane	2011-08-24	2011-08-30	2011-08-30

Fill in the blanks such that the below query outputs records like those above.

Notes: You may not need all blanks. See the **exam reference packet** for an example of `CASE` **(Sec. 6.2)**. The addition (+) operator supports `date` types: `10` plus `2024-10-01` is `2024-10-11`.

```
SELECT disaster_num, incidents.incident_type, begin_date, end_date,
       (CASE _____
             _____
             _____
        END) AS impute_end_date
FROM incidents
_____
_____;
```

Solution:

```
SELECT disaster_num, incidents.incident_type, begin_date, end_date,
  (CASE WHEN end_date IS NULL
        THEN (begin_date + avg_durations.avg - 1)
        ELSE end_date
   END
  ) AS impute_end_date
FROM incidents
JOIN avg_durations
ON avg_durations.incident_type = incidents.incident_type;
```

Chapter 2: Outliers Gonna Outlie [24 pt]

We continue our exploration of the FEMA dataset with the `grants` table, which contains financial assistance information about disasters. Consider the following attribute description:

> `ia_approved_num INTEGER`: The number of disaster assistance applications that were approved for Individual Assistance (IA).

Percentile statistics for this attribute are shown to the right. You would like to detect outliers on this attribute.

percentile	value
0	1
5	160.8
10	262.6
50	1984
90	28022.6
95	77026.8
100	774691

2.1. Use the percentile table to determine each summary statistic below. If impossible, fill in the "Cannot be determined" bubble and leave the line blank.

i. [1 pt] Mean

N/A

◯ **Cannot be determined**

ii. [1 pt] Median

1984

◯ Cannot be determined

iii. [1 pt] Minimum

1

◯ Cannot be determined

2.2. [3 pt] Suppose that the Median Absolute Deviation (MAD) of `ia_approved_num` is $MAD = 1690$. The Hampel X84 outlier defines outliers as values that are $2k \cdot MAD = 5011.188$ from the median, where $k = 1.4826$ and MAD is the Median Absolute Deviation. With this method, which of the below `ia_approved_num` values are outliers?

☐ A. 13
☐ B. 196
☐ C. 1984
☐ D. 5956
☐ E. 6750
☐ **F. 8084**
☐ **G. 49267**
☐ **H. 108986**
☐ I. Cannot be determined

2.3. Suppose we instead would like to apply a 10% winsorization of `ia_approved_num` (i.e., 10% tails on each side). For each of the below values, what would be its corresponding transformed value in the resulting 10% winsorized array? If the correct value is not listed, fill in the "Another value" bubble.

i. [2 pt] 13

◯ 13
◯ 160.8
◯ **262.6**
◯ Another value

ii. [2 pt] 5956

◯ 1984
◯ **5956**
◯ 28022.6
◯ Another value

iii. [2 pt] 49267

◯ 49267
◯ **28022.6**
◯ 77026.8
◯ Another value

Finally, we consider the relationship between the two tables in the FEMA dataset:

- The `incidents` table contains incident records for all federally declared disasters.
- The `grants` table contains financial assistance information about disasters.

Both tables have the attribute `disaster_num`, which is a sequentially assigned number ("disaster number") used to designate an event or incident declared as a **disaster**.

2.4. You use `psql` to explore the relationships between entities: incidents, grants, and disasters).

```
fema=# SELECT COUNT(*),
fema-#   COUNT(DISTINCT disaster_num)
fema-# FROM grants;
 count | count
-------+-------
  3500 |  3500

fema=# SELECT COUNT(*),
fema-#   COUNT(disaster_num),
fema-#   COUNT(DISTINCT disaster_num)
fema-# FROM incidents;
 count | count | count
-------+-------+-------
 70000 | 70000 |  5000
```

```
fema=# SELECT COUNT(*),
fema-#   COUNT(DISTINCT g.disaster_num),
fema-#   COUNT(DISTINCT i.disaster_num)
fema-# FROM incidents AS i
fema-# JOIN grants AS g
fema-# ON g.disaster_num
fema-#     = i.disaster_num;
 count | count | count
-------+-------+-------
 50000 |  3500 |  3500
```

Given the above output, mark each of the below statements as True or False.

i. [1 pt] Each grant is associated with one disaster. ◯ **True** ◯ False

ii. [1 pt] Each disaster has an associated record in the `grants` table. ◯ True ◯ **False**

iii. [1 pt] One disaster number can be associated with many incidents. ◯ **True** ◯ False

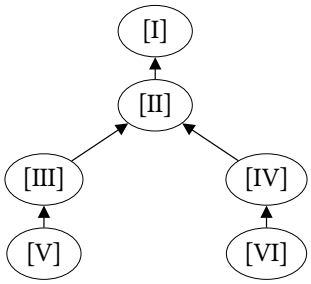
iv. [1 pt] The `disaster_num` attribute can be a primary key for the `grants` table. ◯ **True** ◯ False

2.5. Suppose we write the query to the right.

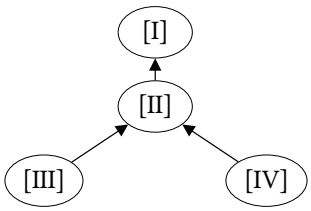
A query optimizer then produces an optimized query plan according to SQL query semantics and **projection pushdown**. What is the resulting optimized query plan?

```
SELECT incidents.id,
    grants.id,
    incident_type,
    ia_approved_num
FROM incidents
JOIN grants
ON incidents.disaster_num
    = grants.disaster_num;
```

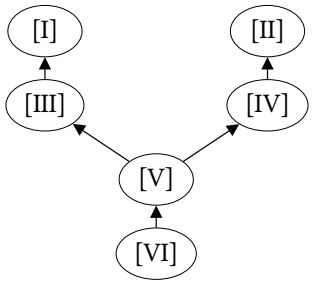
i. [2 pt] First, select the tree shape that **most closely** resembles an optimized plan (for this query) that leverages projection pushdown:



○ (A)



○ (B)



○ (C)

Solution: (A)

ii. [6 pt] For the query plan you selected above, define each node by filling in the blanks:

- For relational operators, write in the relational operator and its subscript.
- For scans, write the table name.
- Depending on the tree shape you selected, you may not need all blanks below. Write *N/A* on the blanks that you do not need.

[I] π `incidents.id, grants.id, incident_type, ia_approved_num`

[II] $\bowtie$ `incidents.disaster_num = grants.disaster_num`

[III] π `id, incident_type, disaster_num`

[IV] π `id, disaster_num, ia_approved_num`

[V] `incidents`

[VI] `grants`

Chapter 3: I'm Craving Mongolian Food! [19 pt]

For this question, we'll be using the Yelp dataset, looking just at the user and business entities. We'll start by using MongoDB to find some prolific Yelp users who might have good reviews or recommendations. When completing the MongoDB queries, use MQL syntax, and assume that all documents have been loaded into the `db.user` collection.

Consider the following properties of a user document. For this dataset we will assume all users have the same possible fields, but some fields may be null if the data does not apply to that user. You can review more about the dataset if you wish in the **exam reference packet (Sec. 2)**.

A User Document

```
{
    "_id": ObjectId("6758dca1091ea6eafadeb758"),
    "user_id": "q_QQ5kBBwlCcbL1s4NVK3g",
    "name": "Jane",
    "review_count": 1220,
    "yelping_since": "2005-03-14 20:26:35",
    "useful": 15038,
    "funny": 10030,
    "cool": 11291,
    "elite": [2011 2012, 2013, 2014], // or null
    "friends": ["friend_id_1", "..." ], // or null
    "fans": 1357,
    "average_stars": 3.85
}
```

3.1. Find the total number of elite Yelp users for each year.

In order to complete this query, we will need to group users by the years in which a user is `"elite"`. Yelp has some criteria by which a user is considered "elite". A user is an "elite" user for 2023 if the year "2023" appears in the `elite` field. The value `"eliteCount"` will be the total number of users who are elite in that year. (We will ignore any users who are not elite in any year, which are represented by a NULL value for the `elite` field.)

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

Count of Elite Users by Year Result

```
[
  { "_id": 2023, "eliteCount": xxx },
  { "_id": 2022, "eliteCount": yyy },
  // "_id" is the year, one entry per year.
  // and so on...
]
```

Fill in the blanks below so that the query runs correctly.

Count of Elite Users by Year Query

```
db.user.___(i)___([
  {
    ___(ii)___: {
            $elite: {
            $exists: true,
            $ne: []
      }
    }
  },
  { ___(iii)___: "$elite" },
  {
    $group: {
      _id: {
        $toInt: ___(iv)___ },
        eliteCount: { ___(v)___: 1
      }
    }
  }
])
```

Solution:

```
db.users.aggregate([
  {
    $match: {
      elite: {
        $exists: true,
        $ne: []
      }
    }
  },
  { $unwind: "$elite" },
  {
    $group: {
      _id: { $toInt: "$elite" },
      eliteCount: { $sum: 1 }
    }
  }
])
```

i. [1 pt] ◯ `find` ◯ `aggregate` ◯ `groupBy` ◯ `collect`

ii. [2 pt]

Solution: `$match`

iii. [1 pt] ◯ `$unwind` ◯ `$lateral` ◯ `$match` ◯ `$group` ◯ `$project`

iv. [2 pt]

Solution: `"$elite"`

v. [1 pt] ◯ `$count` ◯ `$total` ◯ `$max` ◯ `$add` ◯ `$sum`

3.2. For this question, consider the business entity.

The complete Yelp dataset includes millions of businesses. However, we are interested in exploring alternative ways of storing data about businesses that might be more efficient. Consider the following properties of one business:

A Business Document


```
{ "_id": ObjectId("6758dc70091ea6eafadc26c3"),
 "business_id": "6iYb2HFDywm3zjuRg0shjw",
 "name": "Bear's Lair",
 "city": "Berkeley",
 "state": "CA",
 "postal_code": "94720",
 "stars": 3.3,
 "review_count": 102,
 "is_open": 1,
 "attributes": {
    "RestaurantsTableService": "True",
    "BusinessAcceptsCreditCards": "True",
    ...
    "RestaurantsDelivery": "GrubHub"
  },
 "categories": ["Gastropubs", "Food", ...],
 "hours": {
    "Monday": "11:00-23:00",
      "Tuesday": "11:00-23:00",
      "Wednesday": "11:00-23:00",
      "Thursday": "11:00-23:00",
      "Friday": "11:00-23:00",
      "Saturday": "11:00-23:00",
      "Sunday": "11:00-23:00"
    }
}
```


i. [3 pt] Which fields for a given business would need to be adapted to fit into a purely relational model? i.e., without modification, these fields would not represent atomic data. (Select all that apply)

- ☐ **A.** hours
- ☐ B. stars
- ☐ C. review_count
- ☐ **D.** attributes
- ☐ E. is_open
- ☐ F. postal_code
- ☐ **G.** categories

ii. [3 pt] Which of these fields represent data which is denormalized? (Select all that apply)

- ☐ A. hours
- ☐ **B.** stars
- ☐ **C.** review_count
- ☐ D. attributes
- ☐ E. is_open
- ☐ F. postal_code
- ☐ G. categories

3.3. [6 pt] We want to model the `hours` field as a relational table, called `business_hours`. It will store the hours of each business, where **each row represents the hours a business is open for a one day of the week, with columns for the opening and closing times**

stored separately.

Fill in the following table with the column name, data type, and whether the column is a primary key, foreign key, or neither. You should fill all the rows. We have provided some values, but you should expect to fill all blanks.

- `PK` means "primary key", `FK` means "foreign key"
- Remember each row needs a primary key
- You can assume that a `businesses` table contains and ID for each business in the typical format (e.g., `"6iYb2HFDywm3zjuRg0shjw"`).
- There are multiple valid responses for column names and types.

Column Name	Column Type	Constraints
	`integer`	◯ **PK** ◯ **FK** ◯ N/A
	◯ `BOOLEAN` ◯ `INTEGER` ◯ `TEXT` ◯ `TIMESTAMP`	◯ **PK** ◯ **FK** ◯ N/A
`day_of_week`	◯ `BOOLEAN` ◯ `INTEGER` ◯ `TEXT` ◯ `TIMESTAMP`	◯ **PK** ◯ **FK** ◯ N/A
	◯ `BOOLEAN` ◯ `INTEGER` ◯ `TEXT` ◯ `TIMESTAMP`	◯ **PK** ◯ **FK** ◯ N/A
	◯ `BOOLEAN` ◯ `INTEGER` ◯ `TEXT` ◯ `TIMESTAMP`	◯ **PK** ◯ **FK** ◯ N/A

Chapter 4: Oh! That's Lots of Books! [28 pt]

We're headed back to the library! Since the midterm, our library has gotten very busy. Due to all the traffic, the schema for our library has evolved a bit:

```
books (id, title, author, isbn, publication_year)
book_copies (id, book_id, location_id, format, acquisition_date, status);
users (id, first_name, last_name, email, phone_number, joined_date, is_ya)
locations (id, name, address, phone_number)
checkouts (id, user_id, book_copy_id, location_id, checkout_date, due_date)
book_returns (id, checkout_id, location_id, return_date)
```

In this schema, each user checks out a particular `book_copy` which creates an entry in the `checkouts` table. When a user returns a book, an entry is created in the `book_returns` table. **Note:** The full library schema is included in the exam reference packet **(Sec. 3)**.

4.1. [8 pt] **Checked out books revisited....** Write a query to return the `checkout_id`, user's `first_name`, book `title`, `book_copy_id`, and `due_date` for books which **have not yet been returned**. (You may not need all blanks.)

The resulting table should look like this:

```
 checkout_id | first_name |      title      | book_copy_id |  due_date
-------------+------------+-----------------+--------------+------------
           4 | Jonathan   | Basic Chemistry |         2802 | 2020-06-13
           8 | Cairo      | Basic Chemistry |         1001 | 2021-03-09
```

```
SELECT  c.id as checkout_id,
    _____________________________________________________________
    _____________________________________________________________
    _____________________________________________________________
 FROM checkouts c
 JOIN users u ON c.user_id = u.id
 JOIN book_copies bc
    ON c.book_copy_id = bc.id
 JOIN books b ON bc.book_id = b.id
 WHERE NOT EXISTS (
    _____________________________________________________________
    _____________________________________________________________
    _____________________________________________________________ );
```

Solution:

```
    SELECT
 c.id as checkout_id,
    u.first_name, b.title, c.book_copy_id, c.due_date
    FROM checkouts c
    JOIN users u ON c.user_id = u.id
    JOIN book_copies bc ON c.book_copy_id = bc.id
    JOIN books b ON bc.book_id = b.id
    WHERE NOT EXISTS (
        SELECT id FROM book_returns r
        WHERE r.checkout_id = c.id
    );
```

4.2. [6 pt] You are tasked with **finding the most popular books**, when they are checked, and at which location, which will add the library in the purchasing of new copies of books.

We want to create a view which will aggregate all books (`book_id`), the location (`location_id`), `year`, `month` (of the checkout date) as well as the total `copies_checked_out` in that month. The view will have a schema that looks like this:

```
 book_id | location_id | year | month | copies_checked_out
---------+-------------+------+-------+--------------------
      28 |           3 | 2020 |     1 |                  8
      28 |           3 | 2020 |     2 |                  9
```

Fill in the blanks so that `checkouts_olap` will match this description. **Note:** The SQL function `EXTRACT` returns just the year, month, day, etc. part from a `date` datatype, e.g., `EXTRACT(YEAR FROM '1995-09-05')` returns 1995.

```
CREATE OR REPLACE VIEW checkouts_olap AS
SELECT

    _____

    _____
    EXTRACT(YEAR FROM c.checkout_date) as year,
    EXTRACT(MONTH FROM c.checkout_date) as month,

    _____ as copies_checked_out

FROM  checkouts c
JOIN book_copies bc ON c.book_copy_id = bc.id
JOIN books b ON bc.book_id = b.id
GROUP BY

    _____

    _____

    EXTRACT(YEAR FROM c.checkout_date) as year,
    EXTRACT(MONTH FROM c.checkout_date) as month
ORDER BY book_id DESC;
```

Solution:

```
CREATE OR REPLACE VIEW checkouts_olap AS
SELECT
    b.id as book_id,
    c.location_id,
    EXTRACT(YEAR FROM c.checkout_date) as year,
    EXTRACT(MONTH FROM c.checkout_date) as month,
    COUNT(*) as copies_checked_out
FROM checkouts c
JOIN book_copies bc ON c.book_copy_id = bc.id
JOIN books b ON bc.book_id = b.id
GROUP BY
    b.id,
    c.location_id,
    EXTRACT(YEAR FROM c.checkout_date),
    EXTRACT(MONTH FROM c.checkout_date)
ORDER BY book_id DESC;
```

Note: As part of the course content, we did not cover `CUBE()` and `ROLLUP()`, and so we do not expect you to know how to use these PostgreSQL functions right off-the-bat. However, one of the learning goals of this course is to read SQL documentation to use new functions. **The SQL documentation for `CUBE()` and `ROLLUP()` is in the reference packet (Sec. 4)** and is sufficient for you to answer the below question.

4.3. [5 pt] Now we'd like to use the `checkouts_olap` view to roll up data, exploring all possible combinations of `book_id`, `location_id` and `year`, along with the sum of all checkouts in that group.

The resulting table will look like this:

```
 book_id | location_id | year | total_checkouts
---------+-------------+------+-----------------
 28      | 3           | 2020 |             113
                       ...
 28      | 3           | All  |             300
 28      | All         | All  |             300
 All     | All         | All  |             300
                       ...
 All     | All         | 2020 |             113
                       ...
 28      | All         | 2020 |             113
 All     | 3           | All  |             300
 All     | 3           | 2020 |             113
                       ...
```

Fill in the blanks of the query below. You can receive full credit for this query even if the previous query is not correct.

Note: The function `COALESCE(x, y)` replaces a `NULL` values of `x` with the value `y`, so that it is more clear what the resulting group represents.

```
SELECT
 COALESCE(book_id::text, 'All') as book_id,
 COALESCE(location_id::text, 'All') as location_id,
 COALESCE(year::text, 'All') as year,

  _____ AS total_checkouts

FROM _____

GROUP BY _____ ;
```

Solution:

SELECT

```
    COALESCE(book_id::text, 'All') as book_id,
    COALESCE(location_id::text, 'All') as location_id,
    COALESCE(year::text, 'All') as year,
    SUM(copies_checked_out) as total_checkouts
FROM checkouts_olap
GROUP BY CUBE(book_id, location_id, year);
```

4.4. [2 pt] Assuming `checkouts_olap` contains 4 unique `book_ids`, 3 unique `location_ids`, and 4 unique `years`, how many total rows will be in the resulting table from Question 4.3? Briefly, show how you calculated this value.

_____ rows

Solution: When using CUBE we have the number of rows plus one more for the cumulative total. So for 4 `book_ids`, 3 `location_ids`, and 4 `years`, we have $5 \times 4 \times 5 = 100$ total rows.

4.5. Starting with the data in `checkouts_olap`, which of the types of roll up or drill down operations would be possible?

i. [1 pt] Roll up total checkouts for a given book across all library locations. ◯ **Yes** ◯ No

ii. [1 pt] Roll up total checkouts by the user's `joined_date` after joining with the users table. ◯ Yes ◯ **No**

iii. [1 pt] Drill down into the day each book copy was checked out. ◯ Yes ◯ **No**

4.6. When a user checks out a particular `book_copy`, a transaction T_i is executed with the following ordered actions:

1. $R_i(B_c)$: Read from `book_copies` (to check status available).
2. $W_i(B_c)$: Write to `book_copies` (to update status to checked out).
3. $W_i(C)$: Write to `checkouts` (to insert a new entry).

Suppose that two users simultaneously check out the same `book_copy`, resulting in concurrent transactions T_1 and T_2. The DBMS generates the following transaction schedule:

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

i. [2 pt] Is this schedule serializable?

- ◯ **Serializable**
- ◯ Not Serializable
- ◯ Cannot be determined

ii. [2 pt] Does this schedule provide isolation between T_1 and T_2?

- ◯ **Yes**
- ◯ No
- ◯ Cannot be determined

Chapter 5: Teenagers! At The Library [12 pt]

One of the librarians wants to understand how many books "young adults" (i.e., teenagers) are checking out at the library in the month of December. They construct a **spreadsheet**:

	A	*B*	*C*	*D*	*E*	*F*
1	`id`	`date`	`user_id`	`num_out`	`is_ya`	*ya_tot*
2	001	2024-12-01	101	1	TRUE	*1*
3	002	2024-12-01	980	2	TRUE	*3*
4	003	2024-12-02	103	1	FALSE	*3*
5	004	2024-12-14	980	2	TRUE	*5*

- The topmost row and leftmost column are the column and row addresses, respectively, of the spreadsheet (e.g., cell *B*4 has the value 2024-12-02).
- Each row refers to the **total** number of checkouts `num_out` made by a user with ID `user_id` on a specific date `date` in December 2024. The `is_ya` flag is TRUE only if the user is a young adult. `id` is a primary key.
- Rows are sorted by earliest `date` first; ties are broken by `user_id`.

The librarian has already pre-computed columns A–E but would like your help with computing the **young adult total** (column *F*, *italics*): a cumulative sum of the number of checkouts (column *D*) made by young adults (column *E*) up through the current row.

5.1. [2 pt] Use a the `SUMIF` formula to compute Column *F*, rows 2 onwards. From the Google Sheets documentation, `SUMIF(range, criterion, [sum_range])` returns a conditional sum across a range. **Note**: See the exam reference packet for more detailed documentation **(Sec. 5)**.

Specify the correct formula to insert into cell *F*2 such that, when dragged down (i.e., filled down), cells *F*2 downwards will achieve the desired behavior.

◯ A. `=SUMIF(D$2:D2,E$2:E2)`
◯ B. `=SUMIF(E$2:E2,D$2:D2)`
◯ C. `=SUMIF(D$2:D2, TRUE, E$2:E2)`
◯ D. `=SUMIF(E$2:E2, TRUE, D$2:D2)`
◯ E. None of the above

5.2. [6 pt] Consider how this view would be implemented in different data models. In particular, spreadsheet models support both row addressing and column addressing. Which of the following models support each type of addressing? Select all that apply.

i. PostgreSQL tables ☐ Row addressing ☐ Column addressing ☐ **Neither**

ii. pandas DataFrames ☐ **Row addressing** ☐ **Column addressing** ☐ Neither

iii. 2-D tensors ☐ **Row addressing** ☐ **Column addressing** ☐ Neither

You consider instead computing the "young adult total" column using the original PostgreSQL library database schema.

You first construct a view named `dec_checkouts` that computes the attributes `id`, `date`, `user_id`, `num_out`, and `is_ya` for checkouts in December 2024. The view `dec_checkouts` is the SQL equivalent of columns A–E in the librarian's spreadsheet.

5.3. [4 pt] Your friend who has not taken Data 101 reads the PostgreSQL documentation and writes the query to the right:

This query computes the librarian's requested `ya_tot` column with a window function.

```
SELECT id, date, user_id,
  num_out, is_ya,
  SUM(num_out) OVER (
    PARTITION BY is_ya
    ORDER BY id ASC) AS ya_tot
FROM dec_checkouts
ORDER BY date ASC;
```

Your friend's query produces the following output:

id	date	user_id	num_out	is_ya	ya_tot
001	2024-12-01	101	1	t	1
002	2024-12-01	980	2	t	3
003	2024-12-02	103	1	f	1
004	2024-12-14	980	2	t	5

The output looks **almost** correct but has **an incorrect value of 1** for `ya_tot` in record ID `003` (see spreadsheet cell *F*4, which has value 3).

In no more than **two sentences**, explain to your friend why their query produces this incorrect value.

Solution: Window functions can only compute (aggregate) functions over their specified partition. Your friend's code is effectively computing *two* cumulative sums, for both young adults and non-young adults.

Chapter 6: A Not-So-Random Sampling of Topics [13 pt]

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

6.1. [1 pt] MongoDB supports JOINing data across different ~~documents~~ collections *[exam clarification]*.

◯ **True**
◯ False

6.2. [1 pt] MongoDB does not support indexing data.

◯ True ◯ **False**

6.3. [1 pt] MongoDB is an example of a document store.

◯ **True**
◯ False

6.4. [1 pt] PostgreSQL cannot support semi-structured data like JSON.

◯ True ◯ **False**

6.5. [2 pt] Suppose we would like to distribute our data across multiple servers. An RDBMS like PostgreSQL is preferred over a NoSQL system like MongoDB when:

☐ **A. Your data schema is normalized.**

☐ B. Your data requirements and structure are constantly changing.

☐ C. You have a large amount of sparse data.

☐ **D. It is imperative that all data follow very strict constraints.**

6.6. [1 pt] It is possible to shard (partition) any database in a **horizontal** model, regardless of the structure of the data.

◯ **True**
◯ False

6.7. [1 pt] It is possible to shard (partition) any database in a **vertical** model, regardless of the structure of the data.

◯ True ◯ **False**

6.8. [1 pt] Memory and disk refer to the same physical storage element on a computer.

◯ True ◯ **False**

6.9. [2 pt] In a highly parallel MapReduce system, each **mapper** process gains performance by read/writing data from other mapper processes.

◯ True ◯ **False**

6.10. [2 pt] Which ACID property is best illustrated by this situation: In a highly concurrent system, many users read from and write to the same bank account statements simultaneously.

◯ Atomicity ◯ Consistency ◯ **Isolation** ◯ Durability

Chapter 7: 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 moment!