Data 101: Data Engineering
Final Exam

UC Berkeley, Fall 2023

December 15, 2023

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 **125 points** spread over **10 questions** (including the Honor Code), and must be completed in the 170-minute time period on December 15, 2023, 8:10am – 11am unless you have pre-approved accommodations otherwise.

For 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.

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: Research Is a JOIN-t Venture [22 pt]

The Sky Computing Lab at UC Berkeley works on research projects related to cloud computing. The following two tables describe the projects and their student researcher assignments.

Projects

projectId	title
1	Cirrus
2	Stratus
3	Cumulus
4	Cumulonimbus

Students

studentId	name	projectId
1	Frank	1
2	Abhay	1
3	Nikita	2
4	Jordan	3
5	Bridget	NULL

1.1. [2 pt] Consider the following query:

```
SELECT * FROM Projects NATURAL JOIN Students;
```

What are the attribute names in the resulting output schema? Select all that apply.

- ☐ `projectId`
- ☐ `studentId`
- ☐ `title`
- ☐ `name`
- ☐ `Projects.projectId`
- ☐ `Students.studentId`
- ☐ `Projects.title`
- ☐ `Students.name`
- ☐ `Students.projectId`
- ☐ None of the choices

1.2. [2 pt] Express the natural join query in Question 1.1 as an equivalent *inner join* query. *Note*: Not all blanks may be needed.

```
SELECT ______________________________________________________

_____________________________________________________________

_____________________________________________________________

FROM ________________________________________________________

_____________________________________________________________

____________________________________________________________;
```

Solution:

```
SELECT Projects.projectId AS projectId, title, studentId, name
FROM Projects INNER JOIN Students
ON Projects.projectId = Students.projectId;
```

Rubric: 1 point for the correct join condition; 1 point for the correct column names.

1.3. [2 pt] Write a SQL query to find the titles of the projects that **no** students are assigned to. You must use the common table expression `projectIdsWithStudents` in this query.

```
WITH projectIdsWithStudents AS ( ______________________________

______________________________________________________________

______________________________________________________________

) SELECT ______________________________________________________

FROM __________________________________________________________

_____________________________________________________________;
```

Solution:

```
WITH projectIdsWithStudents AS (
  SELECT DISTINCT projectId FROM Students
)
SELECT title FROM Projects
WHERE projectId NOT EXISTS projectIdsWithStudents;
```

Note: `NOT IN` and `NOT EXISTS` are equivalent for all intents and purposes.

Suppose that the same database now stores *millions* of projects and students across the nation.

1.4. [6 pt] Creating a view versus creating a materialized view has consequences on query performance. As the database administrator, you would like to consider creating `CirrusMembers`, the list of students working on project ID 1, in one of two ways:

-
  ```
  CREATE VIEW CirrusMembers AS
      (SELECT * FROM Students WHERE projectId = 1);
  ```

-
  ```
  CREATE MATERIALIZED VIEW CirrusMembers AS
      (SELECT * FROM Students WHERE projectId = 1);
  ```

For each of the following queries, will the query run faster if `CirrusMembers` is declared as a view or if it instead is declared as a materialized view? Select (A) if view is faster, (B) if materialized view is faster, or (C) if it does not matter.

		(A)	(B)	(C)
i.	`SELECT * FROM CirrusMembers;`	◯	◯	◯
ii.	`SELECT * FROM Students WHERE projectId = 1;`	◯	◯	◯
iii.	`INSERT INTO Projects (...);`	◯	◯	◯
iv.	`INSERT INTO Students (...);`	◯	◯	◯
v.	`DELETE FROM Projects WHERE projectId = 5;`	◯	◯	◯
vi.	`DELETE FROM Students WHERE projectId = 1;`	◯	◯	◯

Solution: (B); (C); (C); (A); (C); (A)

Now, suppose that instead of building `CirrusMembers`, you create the following indexes to improve query performance:

(A) `CREATE INDEX ProjectsProjectId ON Projects(projectId);`
(B) `CREATE INDEX StudentsStudentId ON Students(studentId);`
(C) `CREATE INDEX StudentsProjectId ON Students(projectId);`

1.5. [4 pt] Which of the above indexes will be accessed (i.e., read/modified) when executing the following queries? For each query below, fill in the letter(s) corresponding to each index accessed. Select all that apply or None.

		(A)	(B)	(C)	None
i.	`SELECT * FROM Projects WHERE projectId = 337;`	☐	☐	☐	☐
ii.	`SELECT * FROM Projects WHERE title LIKE '%cloud%';`	☐	☐	☐	☐
iii.	`SELECT * FROM Students WHERE projectId = 337;`	☐	☐	☐	☐
iv.	`SELECT * FROM Projects NATURAL JOIN Students;`	☐	☐	☐	☐

Solution: (A); None; (C); (A), (C)

1.6. [6 pt] Indexes may speed up some queries, but maintaining indexes also has a cost. For each of the following queries, compare its performance before and after creating the indexes described above. Will each query run (A) faster with indexes, (B) slower, or (C) about the same?

		(A)	(B)	(C)
i.	`SELECT * FROM Projects;`	○	○	○
ii.	`SELECT * FROM Projects WHERE projectId = 337;`	○	○	○
iii.	`SELECT * FROM Projects WHERE title LIKE '%cloud%';`	○	○	○
iv.	`SELECT COUNT(*) FROM Students GROUP BY projectId;`	○	○	○
v.	`INSERT INTO Projects (...);`	○	○	○
vi.	`DELETE FROM Projects WHERE projectId = 5;`	○	○	○

Solution: (C); (A); (C); (A); (B); (B);

Chapter 2: The Bay Area Bridge Maze [12 pt]

Welcome to the San Francisco Bay Area! In this section, we will look at all the bridges that span the Bay, along with their associated hourly traffic data records and incident data records. The data is stored in the following schema:

```
Bridges (id INT, name VARCHAR, openYear INT, length FLOAT, type VARCHAR)
Traffic (id INT, bridgeId INT, date DATE, hour INT, vehicles INT)
Incidents (id INT, bridgeId INT, date DATE, severity VARCHAR, injuries INT)
```

Write a PostgreSQL query to accomplish each specified task below.
Note: For each question below, not all blanks may be needed.

2.1. [3 pt] Create a view with two columns, `bridgeId` and `incidentsCount`, in which each row contains the total number of incidents on that bridge.

```
CREATE VIEW IncidentsSummary AS (

  SELECT ______________________________________________

  FROM ________________________________________________

  _____________________________________________________);
```

Solution:

```
    SELECT bridgeId, COUNT(*) AS incidentsCount
    FROM Incidents GROUP BY bridgeId;
```

2.2. [4 pt] Find the bridges with more incidents than average. Return the bridge IDs and their incident counts. Assume the `IncidentsSummary` view in Question 2.1 is correctly implemented.

```
SELECT ________________________________________________

FROM __________________________________________________

_______________________________________________________

______________________________________________________;
```

Solution:

```
    SELECT bridgeId, incidentsCount
    FROM IncidentsSummary
    WHERE incidentsCount >
      (SELECT AVG(incidentsCount) FROM IncidentsSummary);
```

or

```
    SELECT bridgeId, COUNT(*) incidentsCount
    FROM Incidents
    GROUP BY bridgeId
    HAVING COUNT(*) >
      (SELECT AVG(incidentsCount) FROM IncidentsSummary);
```

For your convenience, the relational schema from the previous page is copied below:

```
Bridges (id INT, name VARCHAR, openYear INT, length FLOAT, type VARCHAR)
Traffic (id INT, bridgeId INT, date DATE, hour INT, vehicles INT)
Incidents (id INT, bridgeId INT, date DATE, severity VARCHAR, injuries INT)
```

2.3. [5 pt] **Roll down those windows.** For each record in the `Traffic` table, compute a running total of vehicles for each bridge on each day. Return each record's bridge ID, date, number of vehicles, and the running total (as `runningTotal`), in that order. Sort the result by date in ascending order. *Hint*: The `date` column contains the timestamp of each record.

```
SELECT ______________________________________________

_____________________________________________________

_____________________________________________________

_____________________________________________________

________________________________________ AS runningTotal

FROM ________________________________________________

____________________________________________________;
```

Solution:

```
SELECT
    bridgeId, date, vehicles,
    SUM(vehicles) OVER (PARTITION BY bridgeId ORDER BY date) AS runningTotal
FROM
    Traffic
ORDER BY
    date;
```

Chapter 3: Query Performance of the Bourgeoisie [18 pt]

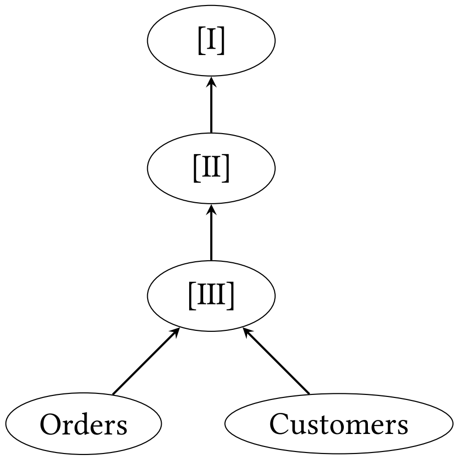
Mackenzie works at a boutique store whose headquarters are in Ulaanbaatar, the capital city of Mongolia. The store maintains the following database schema:

```
Orders (id, date, customerId, productId)
Customers (id, name, country)
```

3.1. [4 pt] Suppose that Cassandra writes the below query:

```
SELECT O.id, C.name
FROM Orders O, Customers C
WHERE O.customerId = C.id
AND C.country = 'Mongolia'
AND O.date >= '2023-01-01';
```

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



i. 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. 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.

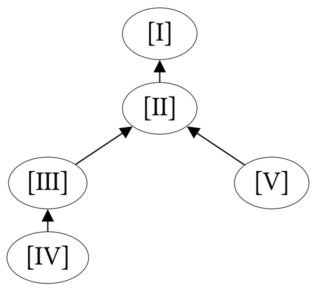
[I] `O.id, C.name`

[II] `date >= '2023-01-01'` $\wedge$ `country = 'Mongolia'`

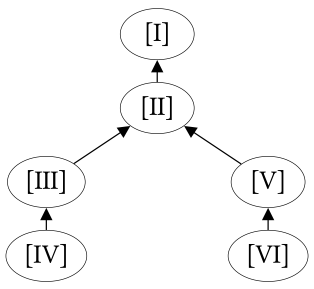
[III] `O.customerId=C.id`

3.2. [7 pt] Suppose the query optimizer then optimizes the query plan in Question 3.1 with **predicate pushdown**. What is the resulting optimized query plan?

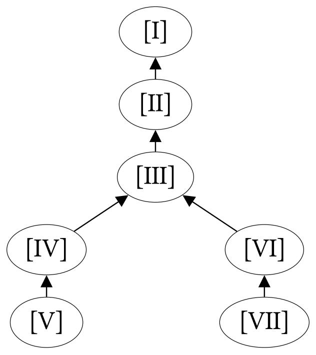
i. First, select the tree shape that **most closely** resembles the optimized query plan:



○ (A)



○ (B)



○ (C)

Solution: (B)

ii. 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 `O.id, C.name`

[II] $\bowtie$ `O.customerId = C.id`

[III] σ `O.date >= '2023-01-01'`

[IV] `Orders`

[V] σ `C.country = 'Mongolia'`

[VI] `Customers`

[VII] N/A

3.3. [3 pt] Next, consider the following query:

```
SELECT COUNT(*)
FROM Orders
WHERE date > '2023-10-01'
GROUP BY productId;
```

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

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 HashAggregate  (cost=29.73..31.68 rows=195 width=12)
                (actual time=0.014..0.015 rows=3 loops=1)
   Group Key: productid
   ->  Bitmap Heap Scan on orders  (cost=8.93..26.65 rows=617 width=4)
                                    (actual time=0.006..0.006 rows=3 loops=1)
         Recheck Cond: (date > '2023-10-01'::date)
         Heap Blocks: exact=1
         ->  Bitmap Index Scan on idx_orders_date
                                    (cost=0.00..8.78 rows=617 width=0)
                                    (actual time=0.002..0.003 rows=3 loops=1)
               Index Cond: (date > '2023-10-01'::date)
```

i. What is the name of the index used in this query? **idx_orders_date**

ii. What is the optimizer's estimate on how many rows the index scan will return? **617**

iii. What is the actual number of rows returned by the index scan? **3**

3.4. [4 pt] Finally, consider the following query:

```
SELECT COUNT(*) FROM Customers C
INNER JOIN Orders O ON C.id = O.customerId
WHERE C.country = 'Mongolia' AND O.date > '2023-10-01';
```

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

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Aggregate  (cost=48.96..48.97 rows=1 width=8)
            (actual time=0.027..0.029 rows=1 loops=1)
   ->  Hash Join  (cost=29.61..48.95 rows=3 width=0)
                  (actual time=0.025..0.026 rows=0 loops=1)
         Hash Cond: (o.customerid = c.id)
         ->  Bitmap Heap Scan on orders o
                                    (cost=8.93..26.65 rows=617 width=4)
                                    (actual time=0.005..0.005 rows=1 loops=1)
               Recheck Cond: (date > '2023-10-01'::date)
               Heap Blocks: exact=1
               ->  Bitmap Index Scan on idx_orders_date
                                    (cost=0.00..8.78 rows=617 width=0)
                                    (actual time=0.003..0.003 rows=3 loops=1)
                     Index Cond: (date > '2023-10-01'::date)
         ->  Hash  (cost=20.62..20.62 rows=4 width=4)
                   (actual time=0.005..0.006 rows=0 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 8kB
               ->  Seq Scan on customers c  (cost=0.00..20.62 rows=4 width=4)
                                    (actual time=0.005..0.005 rows=0 loops=1)
                     Filter: ((country)::text = 'Mongolia'::text)
                     Rows Removed by Filter: 3
```

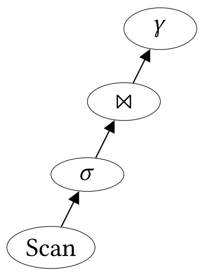
i. What join method was used in this query? **Hash join**

ii. Which table was used to build the hash table? [✓ **Customers**] Orders

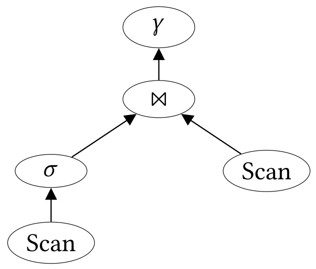
iii. Which table scan was estimated to return more rows? Customers [✓ **Orders**]

iv. Which of the following trees **most closely** resembles this query plan?

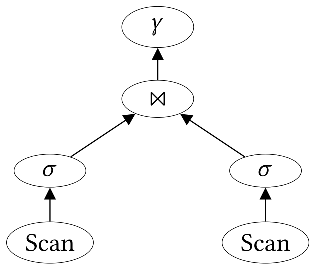
Solution: (C)



○ (A)



○ (B)



○ (C)

Chapter 4: Normalizing Snackpass Usage [5 pt]

Consider the following table `UserRestaurantVisits`, which logs each user's visit to a particular restaurant, including details like the visit's date and Snackpass points earned for the visit.

UserID	UserName	RestaurantID	RestaurantName	Cuisine	Date	Points
1	Napoleon	101	Racha Café	Thai	2023-01-15	10
1	Napoleon	102	Sizzling Lunch	Fusion	2023-02-05	5
2	Cassandra	103	Tacos'n More	Mexican	2023-03-20	8
3	Ron	104	Tender Greens	Salad	2023-04-10	12
5	Dana	105	Imm Thai	Thai	2023-05-25	7
2	Cassandra	105	Imm Thai	Thai	2023-04-25	10

4.1. [2 pt] Assume the table only contains data as shown above. Which of the following functional dependencies are true for this table? *Select all that apply.*

☐ **A. UserID → UserName.**

☐ B. UserID → Date.

☐ C. UserID → Points.

☐ **D. RestaurantID → Cuisine.**

☐ **E. RestaurantName → RestaurantID.**

☐ F. None of the above.

4.2. [3 pt] We want to perform normalization in order to reduce data redundancy. What columns should be in each of the tables below? *Select all that apply.*

i. `Users`

- ☐ **`A.` UserID**
- ☐ **`B.` UserName**
- ☐ `C.` RestaurantID
- ☐ `D.` RestaurantName
- ☐ `E.` Cuisine
- ☐ `F.` Date
- ☐ `G.` Points

ii. `Restaurants`

- ☐ `A.` UserID
- ☐ `B.` UserName
- ☐ **`C.` RestaurantID**
- ☐ **`D.` RestaurantName**
- ☐ **`E.` Cuisine**
- ☐ `F.` Date
- ☐ `G.` Points

iii. `Visits`

- ☐ **`A.` UserID**
- ☐ `B.` UserName
- ☐ **`C.` RestaurantID**
- ☐ `D.` RestaurantName
- ☐ `E.` Cuisine
- ☐ **`F.` Date**
- ☐ **`G.` Points**

Chapter 5: Music Structure and Semi-Structure [14 pt]

Yuto is comparing different data models for his collection of classical music.

Suppose he first builds the following MongoDB collection `music`, of which a sample is below:

```
[
 {
  "title": "Beethoven's Symphony No. 5",
  "composer": "Beethoven",
  "compositionYear": 1808,
  "genre": "Classical",
  "tonic_keys": ["C Minor"],
  "performances": [
   {
    "orchestra": "Vienna Philharmonic",
    "conductor": "Leonard Bernstein",
    "year": 1978
   }
  ]
 },
 {
  "title": "The Four Seasons",
  "composer": "Vivaldi",
  "compositionYear": 1723,
  "genre": "Baroque",
  "tonic_keys": ["E Major", "G Minor", "F Major", "F Minor"]
 },
 {
  "title": "Swan Lake",
  "composer": "Tchaikovsky",
  "compositionYear": 1876,
  "genre": "Ballet",
  "tonic_keys": ["B Minor", "A Major", "E Minor"],
  "performances": [
   {
    "balletCompany": "Mariinsky Ballet",
    "year": 1989
   }
  ]
 },
 {
  "title": "Pierrot Lunaire",
  "composer": "Schoenberg",
  "compositionYear": 1912,
  "genre": "Atonal"
 }
]
```

For each of the questions below, write a MongoDB query on the `music` collection to accomplish the specified task. Both MongoDB and PyMongo syntax are acceptable.

5.1. [3 pt] Find all works written in the *C Major* tonic key.

Solution:

```
music.find({"tonic_keys": "C Major"})
```

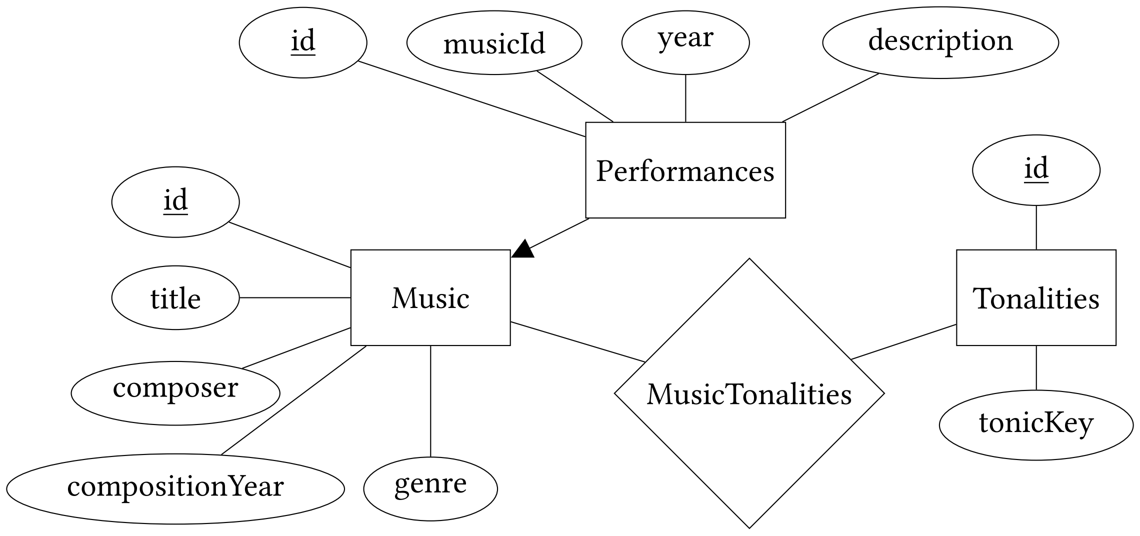
5.2. [4 pt] Find the number of works per each genre. The output should be a list of documents with only the following fields: `genre` and `count`.

```
music.aggregate([
    {




    },
    {




    }
])
```

Solution:

```
music.aggregate([
    {
        $group: {
            _id: "$genre",
            count: { $sum: 1 }
        }
    },
    {
        $project: {
            _id: 0,
            genre: "$_id",
            count: 1
        }
    }
])
```

Yuto is also considering a PostgreSQL solution. He looks at his existing database and constructs the below ER diagram, with which he then declares a relational schema:



5.3. [3 pt] Use the above ER diagram to complete the below DDL statements. In particular, declare the `MusicTonalities` table, and include any applicable foreign key constraints.

```
CREATE TABLE Music(
  id INT PRIMARY KEY, title VARCHAR, genre VARCHAR,
  composer VARCHAR, compositionYear INT
);

CREATE TABLE Performances(
  id INT PRIMARY KEY, musicId INT,
  year INT, description VARCHAR,
  FOREIGN KEY (musicId) REFERENCES Music(id),
);

CREATE TABLE Tonalities(
    id INT PRIMARY KEY,
    tonicKey VARCHAR,
);

CREATE TABLE MusicTonalities( ____________________________________

________________________________________________________________

________________________________________________________________

______________________________________________________________);
```

Solution:

```
CREATE TABLE MusicTonalities(
    id INT PRIMARY KEY,  -- optional
    musicId INT,
    tonalityId INT,
    FOREIGN KEY (musicId) REFERENCES Music(id),
    FOREIGN KEY (tonalityId) REFERENCES Tonalities(id)
);
```

5.4. i. [1 pt] Given the schema defined in Question 5.3, write a SQL query to find the number of performances in the year 1978.

```
SELECT ______________________________________________

FROM ________________________________________________

 ___________________________________________________;
```

Solution:

```
SELECT COUNT(*)
FROM Performances
WHERE year = 1978;
```

ii. [1 pt] Compare query performance between a PostgreSQL RDBMS and a MongoDB database. Is the SQL query you wrote faster or slower than the equivalent MongoDB query?

[✓ **Faster**] ○ Slower ○ Comparable

iii. [2 pt] In no more than 3 sentences, justify your answer to the previous part.

Solution: Faster. Because in SQL we can directly query the performances table. In MongoDB we have to scan over the music collection which contains a lot of irrelevant data.

Chapter 6: "I Volunteer as Data Transformation!" [9 pt]

Natalie and Cassandra are analyzing data on the Hunger Games and are looking to apply advanced EDA and data transformation. Consider the following array, `tributes`:

```
tributes = [100 120 200 300 500 1200]
```

6.1. [5 pt] Use the Hampel X84 method to detect outliers in `tributes`. Recall that the Hampel X84 method trims outliers that are $2 \cdot k \cdot MAD$ from the median, where MAD is the Median Absolute Deviation, and k is a scalar multiplier.

i. What is the median of `tributes`?

○ 50 ○ 100 ○ 140 ○ 200 [✓ **250**] ○ 300

Solution: $(200 + 300)/2 = 250$.

ii. What is the MAD of `tributes`?

○ 50 ○ 100 [✓ **140**] ○ 200 ○ 250 ○ 300

Solution: The distances are: 150, 130, 50, 50, 250, 950.

The median is $(130 + 150)/2 = 140$.

iii. Using the Hampel X84 method, circle all outlier(s) in this data. For the purposes of the exam, use $k = 1.5$ (instead of 1.4826) as the MAD multiplier. Show your work.

Circle all outliers: 100 / 120 / 200 / 300 / 500 / 1200

Solution: The radius is $2 \times 1.4826 \times 140 \approx 2 \times 1.5 \times 140 = 420$.

The bounds are: $(250 - 420, 250 + 420) = (-170, 670)$.

Hence 1200 is the outlier.

6.2. [2 pt] Suppose we instead apply a 20% winsorization of `tributes` (i.e., 20% tails). What is the resulting 20% winsorized array? The 20th percentile and 80th percentile of `tributes` are 120 and 500, respectively.

- ◯ A. `[120 200 300 500]`
- ◯ B. `[200 200 200 300 300 300]`
- ◯ C. `[120 120 200 300 500 500]`
- ◯ D. `[100 100 200 300 1200 1200]`
- ◯ E. `[100 120 200 300 500 1200]`
- ◯ F. None of the above

6.3. [2 pt] What is the Levenshtein distance between the strings *Katniss* and *Catnip*?

Levenshtein distance: ____3____

Chapter 7: ACID is a Basic Principle of Concurrency [10 pt]

7.1. [1 pt] Suppose one transaction Tx 1 is allowed to **read** a new value **written** by another concurrent transaction Tx 2, before Tx 2 encounters an error and rolls back. Which ACID property might the database violate?

- ◯ Atomicity
- ◯ Consistency
- [✓ **Isolation**]
- ◯ Durability

7.2. [3 pt] Which of the following is/are true if two actions conflict? Select all that apply.

- ☐ A. They must be from the same transaction.
- [✓ **B. They must be from different transactions.**]
- [✓ **C. They must operate on the same data object.**]
- ☐ D. One of them must be a read operation (i.e., read action).
- [✓ **E. One of them must be a write operation (i.e., write action).**]
- ☐ F. Both of them must be write operations (i.e., write actions).

Consider the following transaction schedule of 3 concurrent transactions on data objects A, B, C:

	1	2	3	4	5	6	7	8	9
Tx 1	$R_1(A)$	$W_1(A)$						$R_1(C)$	$W_1(C)$
Tx 2			$R_2(B)$		$W_2(A)$				
Tx 3				$R_3(C)$		$R_3(A)$	$W_3(C)$		

7.3. [3 pt] Consider the conflict graph for this schedule, which we will not ask you to draw out. Instead, select all edges that exist in this conflict graph, where Tx 1 → Tx 2 means there is a directed edge that starts at Tx 1 and ends at Tx 2.

- [✓ **Tx 1 → Tx 2**]
- ☐ Tx 2 → Tx 1
- [✓ **Tx 2 → Tx 3**]
- ☐ Tx 3 → Tx 2
- [✓ **Tx 1 → Tx 3**]
- [✓ **Tx 3 → Tx 1**]
- ☐ None

7.4. [3 pt] Is this schedule serializable? If the answer is yes, write an equivalent serial transaction schedule. If the answer is no, explain why.

Serializable? (circle) Yes / No

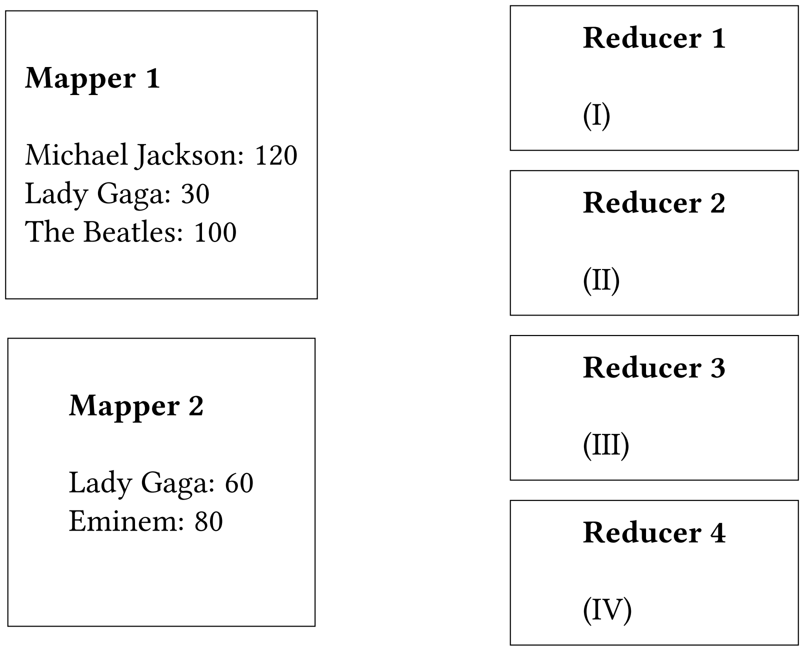
Solution: No, this schedule is not serializable because there is a cycle in the conflict graph.

Chapter 8: Wikipedia is Distributed Media [4 pt]

8.1. [4 pt] Consider the physical execution of a MapReduce program that aggregates (i.e. sums up) the daily page views of Wikipedia pages of popular music artists. Specifically:

- Each mapper task reads in a partition, and outputs article titles and their partial sum of views in this partition.
- Each reducer task takes in all partial sums of a particular article, and outputs the total page views of that article.

Suppose the mapper outputs are as given below, in the form *key: value*:



Example execution of a MapReduce program.

Suppose that the keys are then assigned in alphabetical order to the reducers. What are the **outputs** of the reducers? Write each in the form of *key: value*.

(I) **Eminem: 80**

(II) **Lady Gaga: 90**

(III) **Michael Jackson: 120**

(IV) **The Beatles: 100**

Chapter 9: Rolling a Data Cube [13 pt]

Consider the following sales records relation `data`:

Year	Month	Category	Sales
2022	1	Electronics	150
2022	1	Clothing	500
2022	2	Electronics	120
2022	2	Clothing	400
…	…	…	…
2022	1	Electronics	180
2023	1	Clothing	450
…	…	…	…

9.1. [5 pt] Write a SQL query to perform a roll up that gets the **approximate annual** sales data.

- For each year, get the total sales, average sales per month in that year, and the standard deviation of sales across all months in that year.
- Compute these statistics by sampling 20% of the records using `BERNOULLI`.

Hint: `STDDEV(values)` gives the standard deviation of a column named `values`.

```
SELECT ________________________________________________________,

_____________________________________________________ AS totalSales,

___________________________________________ AS monthlyAverageSales,

_________________________________________ AS salesStandardDeviation

FROM data TABLESAMPLE ______________________________________________

______________________________________________________________________;
```

Solution:

```
SELECT
    Year,
    Month,
    SUM(sales) AS totalSales,
    AVG(sales) AS monthlyAverageSales,
    STDDEV(sales) AS salesStandardDeviation
FROM data TABLESAMPLE BERNOULLI(20)
GROUP BY ROLLUP(Year, Month);
```

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 attached as reference to this exam** and is sufficient for you to answer the following questions.

9.2. [3 pt] Lisa wants to create monthly and yearly aggregates of the sales records using the following PostgreSQL query:

```
SELECT year, month, SUM(sales) FROM data GROUP BY ROLLUP(year, month);
```

The same aggregation result can also be created with multiple group-by queries, each producing a subset of the aggregation rows in the roll-up query. Which of the following queries will produce rows that will also appear in the roll-up query result? *Select all that apply.*

☐ **A. `SELECT year, month, SUM(sales) FROM data GROUP BY year, month;`**

☐ **B. `SELECT year, NULL as month, SUM(sales) FROM data GROUP BY year;`**

☐ C. `SELECT NULL as year, month, SUM(sales) FROM data GROUP BY month;`

☐ D. `SELECT SUM(sales) FROM data;`

☐ **E. `SELECT NULL AS year, NULL AS month, SUM(sales) FROM data;`**

9.3. [3 pt] Lisa now wants to try out the `CUBE()` aggregations using the following PostgreSQL query:

```
SELECT year, month, SUM(sales) FROM data GROUP BY CUBE(year, month);
```

Which of the following queries will produce rows that will also appear in the cube query result? *Select all that apply.*

☐ **A. `SELECT year, month, SUM(sales) FROM data GROUP BY year, month;`**

☐ **B. `SELECT year, NULL as month, SUM(sales) FROM data GROUP BY year;`**

☐ **C. `SELECT NULL as year, month, SUM(sales) FROM data GROUP BY month;`**

☐ D. `SELECT SUM(sales) FROM data;`

☐ **E. `SELECT NULL AS year, NULL AS month, SUM(sales) FROM data;`**

9.4. [2 pt] Suppose A is a set of n attributes, e.g., $A = \{year, month\}$. Both `ROLLUP(A)` and `CUBE(A)` can be constructed by combining a set of multiple group-by queries, each on a distinct attribute set.

i. How many distinct attribute sets will `ROLLUP(A)` produce? $n + 1$

ii. How many distinct attribute sets will `CUBE(A)` produce? 2^n

Chapter 10: Grab Bag [17 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.

10.1. [2 pt] Which of the following statements about PostgreSQL primary keys (PKs) are true?

☐ A. Specifying a column as PK will build a hash index on that column, by default.

☐ B. Specifying a column as PK in a relational schema will also ensure that the relation's records are clustered on that column's values, i.e., stored in PK order on disk.

☐ **C. Specifying a column as PK will enforce both a unique constraint and a non-null constraint on that column.**

☐ **D. Primary keys can span more than one column.**

10.2. [3 pt] Which of the following statements about sampling methods are true? The lecture code for reservoir sampling is attached as reference to this exam.

☐ A. `ORDER BY RANDOM() LIMIT` n does **not** produce a fixed number of output rows.

☐ **B. `TABLESAMPLE BERNOULLI` does not produce a fixed number of output rows.**

☐ C. `TABLESAMPLE BERNOULLI` supports stratified sampling out of the box, i.e.., one can specify that the output sample has (on average) n rows with a specific attribute value.

☐ **D. Reservoir sampling is an algorithm that produces a simple random sample of rows with runtime linear in the number of rows of the original table.**

☐ **E. Reservoir sampling supports sampling from streams of data, where the total number of rows to sample from is not known in advance.**

☐ F. Joining samples of two tables A and B will produce the same number of rows, on average, as first joining A and B, then sampling the joined result.

10.3. [2 pt] Which of the following statements about Entity Resolution is **most** correct?

○ A. Entity Resolution is the process of normalizing a relational schema using ER Diagram Software.

○ **B. Entity Resolution is the process of standardizing data into distinct real-world entities.**

○ C. Both of the above statements are true.

○ D. None of the above statements are true.

10.4. [4 pt] Which rectangular data models exhibit each property below? Assume PostgreSQL relations, pandas DataFrames, scipy matrices, and Google Sheets. Select all that apply.

i. All values in an instance of this model must be of the same data type.

☐ A. Relation ☐ B. Dataframe [✓ **C. Matrix/Tensor**] ☐ D. Spreadsheet

ii. Columns must have labels, e.g., an attribute name.

[✓ **A. Relation**] [✓ **B. Dataframe**] ☐ C. Matrix/Tensor ☐ D. Spreadsheet

iii. The user interface supports direct manipulation.

☐ A. Relation ☐ B. Dataframe ☐ C. Matrix/Tensor [✓ **D. Spreadsheet**]

iv. In any instance of this model, a row can be referenced by row address or primary key.

☐ A. Relation [✓ **B. Dataframe**] [✓ **C. Matrix/Tensor**] [✓ **D. Spreadsheet**]

Solution: iv. Relations do not necessarily have primary keys (e.g., junction tables). Matrices/tensors have rows that can be referenced by row address/index, whether mathematically or in scipy. For this part only, full credit was awarded if at least three boxes were selected.

10.5. [2 pt] Which of the following statements about OLAP are true? Select all that apply.

☐ A. OLAP systems are designed to support a high throughput of database updates by many simultaneous users.

[✓ **B. OLAP systems are most often deployed in a data warehouse.**]

☐ C. The majority of OLAP systems are implemented as multidimensional OLAP (MOLAP) systems and do not support any SQL interfaces.

[✓ **D. Cross-tabs in OLAP systems are functionally equivalent to pivot tables in that they can summarize data across multiple variable categories.**]

10.6. [4 pt] True/False: Fall 2023 Data 101 Guest Lectures Edition

i. Databricks provides a unified platform that integrates different business needs like storage, governance, data science, and analytics on top of Spark.

[✓ **True**] ○ False

ii. Databricks provides significantly different algorithms and implementations for the same set of features available in the open-source Spark.

○ True [✓ **False**]

iii. Modin is a Python package that implements a dataframe model with parallel processing support.

[✓ **True**] ○ False

iv. For the Data 101 DataHub, every user logs in to their own machine, whose computational resources are not shared with other users.

○ True [✓ ○ **False**]

Chapter 11: 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 your paper and leave.
- If ≤ 10 minutes remain, please sit quietly until the exam concludes.

Congrats on finishing the class! We're so happy to have spent this semester with you.

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

This page is intentionally left blank.

Excerpts from PostgreSQL Documentation (You can tear off this page)

7.2.4. GROUPING SETS, CUBE, and ROLLUP

More complex grouping operations than those described above are possible using the concept of grouping sets. The data selected by the FROM and WHERE clauses is grouped separately by each specified grouping set, aggregates computed for each group just as for simple GROUP BY clauses, and then the results returned. For example:

```
=> SELECT * FROM items_sold;  => SELECT brand, size, sum(sales)
                              FROM items_sold
                              GROUP BY GROUPING SETS ((brand), (size), ());
 brand | size | sales          brand | size | sum
-------+------+-------        -------+------+-----
 Foo   | L    |  10            Foo   |      |  30
 Foo   | M    |  20            Bar   |      |  20
 Bar   | M    |  15                  | L    |  15
 Bar   | L    |  5                   | M    |  35
                                     |      |  50
```

Each sublist of GROUPING SETS may specify zero or more columns or expressions and is interpreted the same way as though it were directly in the GROUP BY clause. An empty grouping set means that all rows are aggregated down to a single group (which is output even if no input rows were present), as described above for the case of aggregate functions with no GROUP BY clause.

A shorthand notation is provided for specifying two common types of grouping set. A clause of the form
`ROLLUP ( e1, e2, e3, ... )` represents the given list of expressions and all prefixes of the list including the empty list; thus it is equivalent to

```
GROUPING SETS (
    ( e1, e2, e3, ... ), ... ( e1, e2 ), ( e1 ), ( )
)
```

This is commonly used for analysis over hierarchical data; e.g., total salary by department, division, and company-wide total. A clause of the form `CUBE ( e1, e2, ... )` represents the given list and all of its possible subsets (i.e., the power set). Thus `CUBE ( a, b, c )` is equivalent to

```
GROUPING SETS (
    ( a, b, c ), ( a, b    ), ( a,    c ), ( a       ),
    (    b, c ), (    b    ), (       c ), (         )
)
```

The CUBE and ROLLUP constructs can be used either directly in the GROUP BY clause, or nested inside a GROUPING SETS clause. If one GROUPING SETS clause is nested inside another, the effect is the same as if all the elements of the inner clause had been written directly in the outer clause.

Lecture 25 Reservoir Sampling Algorithm (You can tear off this page)

```python
from random import randrange

def reservoir_sample(data, n, k):
  # fill the reservoir array
  r = []
  for i in range(k):
    r.append(data[i])

  # replace elements with gradually decreasing probability
  for i in range(k, n-1):
    # randrange(a) generates a uniform integer in [0, a)
    j = randrange(i+1)
    if j < k:
        r[j] = data[i]

  return(r)
```

PostgreSQL

```
[ WITH with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ]
                          [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start ]

where from_item can be one of:
    table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
                [ TABLESAMPLE sampling_method ( argument [, ...] ) ]
    [ LATERAL ] ( select ) [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    from_item join_type from_item { ON join_condition |
                                USING ( join_column [, ...] ) [ AS join_using_alias ] }
    from_item NATURAL join_type from_item
    from_item CROSS JOIN from_item

and grouping_element can be one of:   ( )    expression    ( expression [, ...] )

and with_query is:
    with_query_name [ ( column_name [, ...] ) ] AS ( select | values )
```

PostgreSQL, cont.

```
<window or agg_func> OVER (
  [PARTITION BY <…>]
  [ORDER BY <…>]
  [RANGE BETWEEN <…> AND <…>])
```

<window or agg_func>: aggregate functions: `AVG`, `SUM`, …, or:

- `RANK()` ordering within the window
- `LEAD/LAG(exp, n)` value of exp that is n ahead/behind in the window
- `PERCENT_RANK()` relative rank of current row as a %
- `NTH_VALUE(exp, n)` value of exp @ position n in window

```
range_start/range_end:
  UNBOUNDED PRECEDING
  UNBOUNDED FOLLOWING
  CURRENT ROW
  offset PRECEDING
  offset FOLLOWING
```

```
SELECT id, location, age,
  AVG(age) OVER ()
    AS avg_age
FROM Residents;

SELECT id, location, age,
  SUM(age) OVER (
    PARTITION BY location
    ORDER BY age
    RANGE BETWEEN
      UNBOUNDED PRECEDING
      AND
      1 PRECEDING )
      AS a_sum
FROM Residents
ORDER BY location, age;
```

```
REGEXP_REPLACE(source, pattern,
      replacement)
SELECT levenshtein(str1, str2) FROM Strings;
SELECT 'Hello' || 'World',
       STRPOS('Hello', 'el'),
       SUBSTRING('Hello', 2, 3);
```

```
CREATE TABLE <relation name> AS (
    <subquery> );
CREATE TABLE Zips (
    location VARCHAR(20) NOT NULL,
    zipcode INTEGER,
    in_district BOOLEAN DEFAULT False,
    PRIMARY KEY (location),
    UNIQUE (location, zipcode)
);
DROP TABLE [IF EXISTS] <relation name>;
ALTER TABLE Zips
     ADD avg_pop REAL,
     DROP in_district;
CREATE TABLE Cast_info (
  person_id INTEGER,
  movie_id INTEGER,
  FOREIGN KEY (person_id)
    REFERENCES Actor (id)
    ON DELETE SET NULL
    ON UPDATE CASCADE,
  FOREIGN KEY (movie_id)
    REFERENCES Movie (id)
    ON DELETE SET NULL);
```

Entity Resolution Diagrams (ER Diagrams)

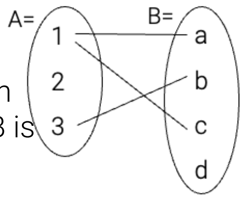
Entity set (rectangles)

- Entities: things, objects, etc.;
- Entity sets: sets entities w/commonalities.
- Every entity set is required to have a primary key (underlined attribute).

Attributes (ovals)
Atomic features connected to entity sets or relationships.

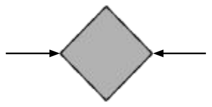
Relationships (diamonds)

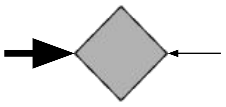
- Connects entity sets.
- A relationship between the entity sets A and B is a subset of A x B.

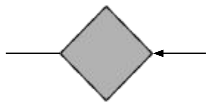


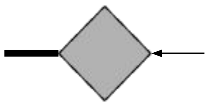
Edges in ER Diagrams can be directed/undirected and represent constraints on subset A x B.

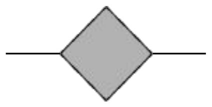
- Undirected edge (with no arrows): no constraints
- Directed edge (arrow): constrains, or determines, the relation to be at most one.
- Bolded edge determines the relation to be at least one.

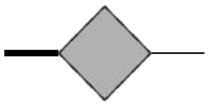

One-one: One on LHS connected to at most one of RHS, and vice-versa


One-one: One on LHS connected to exactly one of RHS (≤ 1 & ≥ 1); one on RHS connected to at most one on LHS


Many-one: One on LHS connected to many on RHS


Many-one: One on LHS connected to at least one on RHS; one on RHS connected to at most one on LHS


Many-many: One on LHS connected to many/few on the RHS, and vice versa


Many-many: One on LHS connected to at least one on RHS; RHS unconstrained

MongoDB

```
db.prizes.find({category: "peace"},
     {_id: 0, category: 1, year: 1,
        laureates.firstname: 1,
        laureates.surname: 1})
      .sort({year: 1, category: -1})
      .limit(2))
collection.find({})
collection.findOne({})
collection.aggregate ( [
  { stage: {…} },
  …
  { stage: {…} }
] )
```

where ***stage*** is one of

```
  $match
  $project
  $sort/$limit
  $group, e.g., { "$group" :
      { "_id" : "$item",
        "totalqty" :
           {"$sum" : "$instock.qty"}}}
  $unwind, e.g., { $unwind: "$instock" }
  $lookup, e.g., { $lookup :
     {from : "inventory",
      localField : "instock.loc",
      foreignField : "instock.loc",
      as :"otheritems"}
     }
```

Odds and Ends

For a dataset X with median $\tilde{X} = \text{median}(X)$, the Median Absolute Deviation (MAD) is $\text{MAD}(X) = \text{median}(|X_i - \tilde{X}|)$.

The Minimum Description Length (MDL) for encoding a set of values c in a set of types H is $\text{MDL} = \min_{T \in H} \sum_{v \in c} (I_T(v) log(|T|) + (1 - I_T(v)) len(v))$
where $I_T(v)$ is an indicator for if v "fits" in type T (with |T| distinct values), log is base 2, and len(v) is the cost for encoding a value v in some default type.

A **functional dependency** (FD) is a form of constraint between 2 sets of attributes in a relation. For a relational instance with attributes X, Y, and Z:

- The FD X → Y is satisfied if for every pair of tuples t1 and t2 in the instance, if t1.X = t2.X, then t1.Y = t2.Y.
- The FD AB → C is satisfied if for every pair of tuples t1 and t2 in the instance, if t1.A = t2.A and t1.B = t2.B, then t1.C = t2.C.

`Map``(k, v) → <k', v'>*`

- Takes a key-value pair and outputs a set of key-value pairs
- There is one `Map` function call for each `(k,v)` pair

`Reduce``(k', <v'>*) → <k', v''>*`

- All values `v'` with same key `k'` are reduced together and processed in `v'` order
- There is one `Reduce` function call for each unique key `k'`