Midterm: DSC 208 Data Management for Analytics

Questions and Explanations

Question 1. Consider the following chart showing Pareto tradeoffs of 4 different ML models on two metrics of interest. Suppose you are told that model D is Pareto-optimal. For which of the following metrics on the X axis will that make sense?
Chart description:
Axes: y-axis Prediction accuracy, x-axis Metric TBD
Points: A(1,2), B(1,1), C(2,1), D(2,2)

a) Training throughput (examples per second)

b) Training cost (dollars)

c) Space footprint (bytes)

d) Inference latency (seconds)

Answer: a) Training throughput (examples per second)

- *Explanation:* Pareto optimality in this context means that a model is optimal if you cannot improve one metric without worsening another. The Y-axis is "Prediction accuracy," which is a "higher-is-better" metric. For a model D(2,2) to be Pareto-optimal when compared to A(1,2), B(1,1), C(2,1):
  - If X-axis is "Training cost" (lower is better): D (cost 2) is worse than A (cost 1) in cost for the same accuracy (accuracy 2). So D would not be Pareto-optimal.
  - If X-axis is "Space footprint" (lower is better): Similar to cost, D (footprint 2) is worse than A (footprint 1) for the same accuracy. So D would not be Pareto-optimal.
  - If X-axis is "Inference latency" (lower is better): Similar again, D (latency 2) is worse than A (latency 1) for the same accuracy. So D would not be Pareto-optimal.
  - If X-axis is "Training throughput (examples per second)" (higher is better):
    - Compared to A(1,2): D (throughput 2, accuracy 2) is better than A (throughput 1, accuracy 2) in throughput, and equal in accuracy. So A is dominated by D.
    - Compared to B(1,1): D (throughput 2, accuracy 2) is better than B (throughput 1, accuracy 1) in both. So B is dominated by D.
    - Compared to C(2,1): D (throughput 2, accuracy 2) is better than C (throughput 2, accuracy 1) in accuracy, and equal in throughput. So C is dominated by D.

    In this case, D is not dominated by any other model, making it Pareto-optimal. This makes sense only if the X-axis metric is also "higher-is-better".

Question 2. Which structured data model(s) support(s) in-place edits to the data?

a) Relation

b) DataFrame

c) Matrix

d) All of the three

Answer: d) All of these

- *Explanation:*
  - 'Relation' (in a relational database): Relational databases are designed for transactional workloads, which include 'UPDATE' operations that perform in-place edits to data.

– 'DataFrame' (e.g., in Pandas or R): DataFrames are mutable objects in memory. You can directly assign new values to cells or columns, effectively performing in-place edits.
– 'Matrix' (e.g., in NumPy or R): Matrices, when stored in mutable data structures, allow for direct modification of individual elements (e.g., $matrix[row, col] = new_value$).

Therefore, all three data models, when implemented in typical software systems, support in-place edits to data.

Question 3. Which of the following best describes the relational data model?

a) A data model that organizes data into a hierarchy of entities and relationships
b) A data model that organizes data into a tree structure
c) A data model that organizes data into a set of tables with columns and rows
d) A data model that organizes data into a graph structure

Answer: c) A data model that organizes data into a set of tables with columns and rows

- *Explanation:* This is the fundamental definition of the relational data model, as introduced by Edgar Codd. Data is organized into two-dimensional 'tables' (also called 'relations'), where each table consists of named 'columns' (attributes) and unordered 'rows' (tuples). Relationships between tables are established through shared attributes (keys).

Question 4. Here is a table representing a relation S Table:

StudentID	Name	Age	Gender
1	John Smith	22	Male
2	Jane Smith	20	Female
3	Zhang San	21	Male

Identify:

1. The attributes of S.
2. The schema of S.
3. The tuples of S.
4. The components of the tuples for each attribute of S.

Which of the following is NOT a true statement about relation S?

a) Age is an attribute of S
b) S has four attributes
c) (3, Zhang San, 21, Male) is a tuple of R
d) S has four tuples

Answer: d) S has four tuples

- *Explanation:* Let's evaluate each statement:
  – **a) Age is an attribute of S:** True. 'Age' is clearly listed as a column header, meaning it's an attribute (or schema element) of the relation.
  – **b) S has four attributes:** True. The attributes are 'StudentID', 'Name', 'Age', and 'Gender'. There are indeed four distinct attributes.
  – **c) (3, Zhang San, 21, Male) is a tuple of R:** True. This is exactly one of the rows (tuples) in the provided table. (Note: The question says "a tuple of R" but the table is named S. Assuming it refers to relation S.)
  – **d) S has four tuples:** False. The table explicitly shows three rows of data: 1. (1, John Smith, 22, Male) 2. (2, Jane Smith, 20, Female) 3. (3, Zhang San, 21, Male) Therefore, relation S has three tuples, not four.

Question 5. Suppose relations R(A,B) and S(B,C,D) have the tuples shown below: Tables: relation R(A,B):

A	B
1	2
3	4
5	6

relation S(B,C,D):

B	C	D
2	4	6
4	6	8
4	7	9

SQL query:

```
SELECT A, R.B, S.B, C, D
FROM R, S
WHERE R.A < S.C AND R.B < S.D
```

Then, identify one of the tuples in the result from the list below.

a) (5,6,2,4,6)

b) (3,4,5,7,9)

c) (3,4,4,7,8)

d) (1,2,2,4,6)

Answer: d) (1,2,2,4,6)

- *Explanation:* We are performing a theta-join with the condition 'R.A ¡ S.C AND R.B ¡ S.D'. Let's systematically check each R tuple against each S tuple:
  From R: (1,2)
  - Against S: (B=2, C=4, D=6)
  - Condition 1: R.A ¡ S.C (1 ¡ 4)? True.
  - Condition 2: R.B ¡ S.D (2 ¡ 6)? True.
  - Both True. Result tuple: (A=1, R.B=2, S.B=2, C=4, D=6) → **(1,2,2,4,6)**. This matches option (d).
  - Against S: (B=4, C=6, D=8)
  - Condition 1: R.A ¡ S.C (1 ¡ 6)? True.
  - Condition 2: R.B ¡ S.D (2 ¡ 8)? True.
  - Both True. Result tuple: (1,2,4,6,8). (Not an option)
  - Against S: (B=4, C=7, D=9)
  - Condition 1: R.A ¡ S.C (1 ¡ 7)? True.
  - Condition 2: R.B ¡ S.D (2 ¡ 9)? True.
  - Both True. Result tuple: (1,2,4,7,9). (Not an option)

  Since we found a match, we can stop here. The tuple (1,2,2,4,6) is a valid result.

Question 6. Suppose R(A,B) has a tuple t: (A = 2, B = NULL). The following query outputs t. SQL query:

```
SELECT *
FROM R
WHERE (((A IS NULL) OR (B = 2)) AND (B IS NULL))
```

a) True

b) False

Answer: b) False

- *Explanation:* Let's evaluate the 'WHERE' clause for tuple t: (A=2, B=NULL). The condition is '(((A IS NULL) OR (B = 2)) AND (B IS NULL))'
  1. 'A IS NULL': Is 2 IS NULL? False. 2. 'B = 2': Is NULL = 2? Unknown (in SQL, comparison with NULL yields Unknown). 3. 'B IS NULL': Is NULL IS NULL? True.
  Now substitute these truth values into the overall expression: '((False OR Unknown) AND True)'
  - 'False OR Unknown' evaluates to 'Unknown'.
  - 'Unknown AND True' evaluates to 'Unknown'.

  In SQL, a 'WHERE' clause only returns rows where the condition evaluates to 'TRUE'. Since the condition evaluates to 'UNKNOWN' (not 'TRUE'), the tuple t will 'NOT' be output by this query. Therefore, the statement is False.

Question 7. Suppose the relation R(a,b,c) has the tuples: Table: Table R:

a	b	c
0	1	2
0	1	3
4	5	6
4	6	3

Compute the generalized projection $\pi_{B,A+C,B}(R)$, and then identify from the list below one of the tuples in this projection.

a) (10,5,10)

b) (1,1,3)

c) (1,3,0)

d) (6,7,6)

Answer: d) (6,7,6)

- *Explanation:* A generalized projection applies expressions to the attributes of each tuple and then projects the results. For each tuple (a, b, c) in R, we need to compute (b, a+c, b).
  1. Tuple (0, 1, 2):
     - b = 1
     - a+c = 0+2 = 2
     - b = 1
     - Result: (1, 2, 1)
  2. Tuple (0, 1, 3):
     - b = 1
     - a+c = 0+3 = 3
     - b = 1
     - Result: (1, 3, 1)
  3. Tuple (4, 5, 6):
     - b = 5
     - a+c = 4+6 = 10
     - b = 5
     - Result: (5, 10, 5)
  4. Tuple (4, 6, 3):

- b = 6
- a+c = 4+3 = 7
- b = 6
- Result: (6, 7, 6)

The resulting bag of tuples from the projection is $\{(1,2,1),(1,3,1),(5,10,5),(6,7,6)\}$. From the given options, (6,7,6) is present in this result.

Question 8. TRUE/FALSE: A left outer join can always be written using a right outer join

a) False

b) True

Answer: b) True

- *Explanation:* Yes, this is true. A 'LEFT OUTER JOIN' (e.g., 'A LEFT OUTER JOIN B') keeps all records from the left table (A) and the matching records from the right table (B). If there's no match, NULLs are placed for B's columns. A 'RIGHT OUTER JOIN' (e.g., 'A RIGHT OUTER JOIN B') keeps all records from the right table (B) and matching records from the left table (A). Crucially, 'A LEFT OUTER JOIN B' is logically equivalent to 'B RIGHT OUTER JOIN A'. You just swap the tables around the 'RIGHT OUTER JOIN' keyword.

Question 9. Consider the following query on the Netflix database schema discussed in the lectures with Ratings relation alias R and Movies relation alias M. Which of the queries listed afterward is logically equivalent to this query? $\sigma_{\text{Year}=2021 \wedge \text{Stars}=5.0}(R \bowtie M)$

a) $\sigma_{\text{Year}=2021}(R) \bowtie \sigma_{\text{Stars}=5.0}(M)$

b) $\sigma_{\text{Year}=2021 \wedge \text{Stars}=5.0}(R) \bowtie M$

c) $\sigma_{\text{Year}=2021 \wedge \text{Stars}=5.0}(M) \bowtie R$

d) $\sigma_{\text{Year}=2021}(M) \bowtie \sigma_{\text{Stars}=5.0}(R)$

Answer: d) $\sigma_{\mathbf{Year}=2021}(M) \bowtie \sigma_{\mathbf{Stars}=5.0}(R)$

- *Explanation:* This question tests the "pushdown" optimization rule in relational algebra. Selection (σ) operations can be pushed down before a join ($\bowtie$) if the selection condition only involves attributes from one of the relations being joined. The original query is $\sigma_{\text{Year}=2021 \wedge \text{Stars}=5.0}(R \bowtie M)$.
  - 'Year' is an attribute of 'Movies (M)'.
  - 'Stars' is an attribute of 'Ratings (R)'.

  Therefore, we can push down the 'Year = 2021' selection to 'M' and the 'Stars = 5.0' selection to 'R' *before* performing the join. This results in: $(\sigma_{\text{Year}=2021}(M)) \bowtie (\sigma_{\text{Stars}=5.0}(R))$ This is exactly option (d). This optimization is crucial for performance as it reduces the size of the relations before the potentially expensive join operation.

Question 10. A primary index is necessarily also the following type of index?

a) None of the three

b) Composite

c) Unique

d) Secondary

Answer: c) Unique

- *Explanation:*
  - A 'primary index' is an index built on the primary key of a table.
  - A 'primary key' by definition must contain unique values and uniquely identify each row in the table.

– Therefore, an index built on a primary key (a primary index) must necessarily enforce 'uniqueness'.
– It is not necessarily a 'composite' index (it can be on a single attribute).
– It is also not necessarily a 'secondary' index; in fact, a primary index is often the main or clustering index that dictates the physical storage order of data.

Question 11. Consider the following schema:

```
Student (snum: integer, sname: string, major: string, level: string, age: integer)
Class (cname: string, meets_at: time, room: string, fid: integer)
Faculty (fid: integer, fname: string, depid: integer)
Enrolled (snum: integer, cname: string)
```

a) Write the SQL statements required to create all of the above relations, including all primary and foreign keys

```
Answer:
CREATE TABLE Student (
snum INT AS PRIMARY KEY,
sname VARCHAR(50),
major VARCHAR(50),
level VARCHAR(50),
age INTEGER)

CREATE TABLE Class (
fid INTEGER AS PRIMARY KEY
cname VARCHAR(50),
meets_at TIME,
room VARCHAR(50))

Create Table Faculty (
fid INT AS PRIMARY KEY,
fname VARCHAR(50),
depid INT)

CREATE TABLE Enrolled (
snum INT AS PRIMARY KEY,
cname VARCHAR(50))
```

Evaluation and Correction for 11a):

- **Student table:** 'snum INT PRIMARY KEY' is the correct syntax. 'AS' is not needed.
- **Class table:** The primary key should be 'cname' (class name), not 'fid'. 'fid' is a foreign key referencing 'Faculty'. Also, 'fid' is a foreign key, not a primary key.
- **Faculty table:** Looks correct.
- **Enrolled table:** This is a many-to-many relationship, and its primary key should be a composite key of both 'snum' and 'cname'. Both 'snum' and 'cname' are foreign keys.

  Corrected SQL for 11a):

```
CREATE TABLE Student (
    snum INTEGER PRIMARY KEY,
    sname VARCHAR(50),
    major VARCHAR(50),
    level VARCHAR(50),
    age INTEGER
);

CREATE TABLE Faculty (
    fid INTEGER PRIMARY KEY,
    fname VARCHAR(50),
    depid INTEGER
);
```

```
CREATE TABLE Class (
    cname VARCHAR(50) PRIMARY KEY,
    meets_at TIME,
    room VARCHAR(50),
    fid INTEGER,
    FOREIGN KEY (fid) REFERENCES Faculty(fid)
);

CREATE TABLE Enrolled (
    snum INTEGER,
    cname VARCHAR(50),
    PRIMARY KEY (snum, cname),
    FOREIGN KEY (snum) REFERENCES Student(snum),
    FOREIGN KEY (cname) REFERENCES Class(cname)
);
```

b) Write the SQL statements that finds the names of all senior students (Student.level = "Junior") who have less than 25 years old (Student.age¡25) and enrolled in a class taught by Prof. Gupta (Faculty.fname= "Gupta").

```
Answer:
SELECT DISTINCT s.name, s.level, s.age
FROM Student AS s, Faculty as f
WHERE s.level='Junior' AND s.age > 25 AND f.name ='Gupta'
```

Evaluation and Correction for 11b):

- The join conditions between 'Student', 'Enrolled', 'Class', and 'Faculty' are missing.
- The 'age' condition is 's.age ¿ 25' in the answer, but the question asks for 'Student.age ¡ 25'.
- The question asks for 'Student.level = "Junior"', which is correctly used.
- The 'DISTINCT' keyword is good as a student might be in multiple classes taught by Prof. Gupta. **Corrected SQL for 11b):**

```
SELECT DISTINCT S.sname
FROM Student AS S
JOIN Enrolled AS E ON S.snum = E.snum
JOIN Class AS C ON E.cname = C.cname
JOIN Faculty AS F ON C.fid = F.fid
WHERE S.level = 'Junior'
  AND S.age < 25
  AND F.fname = 'Gupta';
```

c) Write the SQL statements that finds the names of faculty members for whom the combined enrollment of the courses that they teach is less than three.

```
Answer:
SELECT DISTINCT f.name
FROM Faculty as f, Enrolled as e
WHERE SUM(e.snum) < 3
```

Evaluation and Correction for 11c):

- The 'SUM(e.snum)' is incorrect; 'snum' is a student ID, not an enrollment count. To count enrollment, you need to count distinct 'snum' values for each class, then sum those counts per faculty. Or, simpler, 'COUNT(E.snum)' for all enrollments associated with a faculty member.
- Missing joins between 'Faculty', 'Class', and 'Enrolled'.
- Missing 'GROUP BY' clause for aggregation.
  Corrected SQL for 11c):

```sql
SELECT F.fname
FROM Faculty AS F
JOIN Class AS C ON F.fid = C.fid
JOIN Enrolled AS E ON C.cname = E.cname
GROUP BY F.fid, F.fname
HAVING COUNT(E.snum) < 3; -- COUNT(E.snum) counts the number of enrollments
                          -- if distinct students are required, use COUNT(
                              DISTINCT E.snum)
```

d) For each level, print the level and the average age of students for that level.

```sql
Answer:
SELECT AVERAGE(s.age) as avg_age, s.level
FROM STUDENTS as s
GROUP BY s.level
```

Evaluation and Correction for 11d):

- 'AVERAGE' should be 'AVG'.
- Otherwise, this query is correct. **Corrected SQL for 11d):**

```sql
SELECT S.level, AVG(S.age) AS avg_age
FROM Student AS S
GROUP BY S.level;
```

e) Write the SQL statements that, for each faculty member that has taught classes only in room 'R2010', prints the faculty member's name and the total number of classes she or he has taught.

```sql
Answer:
SELECT DISTINCT f.name, COUNT(c.name)
FROM Faculty AS f, Classes AS c
JOIN f.fid on c.fid
WHERE c.name='R2010'
```

Evaluation and Correction for 11e):

- The join syntax 'JOIN f.fid on c.fid' is incorrect. It should be 'ON f.fid = c.fid'.
- The 'WHERE c.name='R2010'' condition only selects classes in 'R2010'. To find faculty who *only* taught in 'R2010', you need a more complex condition (e.g., using 'NOT EXISTS' or 'EXCEPT' or 'GROUP BY' with 'HAVING').
- 'COUNT(c.name)' should be 'COUNT(DISTINCT c.cname)' if counting unique classes.
- Missing 'GROUP BY'.
  Corrected SQL for 11e):

```sql
SELECT F.fname, COUNT(DISTINCT C.cname) AS num_classes
FROM Faculty AS F
JOIN Class AS C ON F.fid = C.fid
GROUP BY F.fid, F.fname
HAVING EVERY(C.room = 'R2010'); -- PostgreSQL/SQL:2003 "EVERY" or equivalent
    logic
                               -- For more portable SQL:
-- HAVING COUNT(CASE WHEN C.room = 'R2010' THEN 1 END) = COUNT(C.cname);
-- OR, using NOT EXISTS:
/*
SELECT F.fname, COUNT(C.cname) AS num_classes
FROM Faculty AS F
JOIN Class AS C ON F.fid = C.fid
WHERE NOT EXISTS (
    SELECT 1
    FROM Class AS C2
    WHERE C2.fid = F.fid AND C2.room != 'R2010'
)
GROUP BY F.fid, F.fname;
*/
```

f) Write the SQL statements that computes the average age of all students who are majoring in "Data Science" and enrolled in "DSC 80" and "DSC 100"

```
Answer:
SELECT s.major
FROM Students AS s, Classes AS c
WHERE c.name ='DSC80' AND c.name='DSC100' AND s.major='Data Science'
```

Evaluation and Correction for 11f):

- The query should 'SELECT AVG(s.age)', not 's.major'.
- The condition 'c.name ='DSC80' AND c.name='DSC100'' is logically impossible for a single class name. A student needs to be enrolled in *both* classes. This requires either two separate join paths or using subqueries/'HAVING' with 'GROUP BY'.
- Missing joins with 'Enrolled'.

  Corrected SQL for 11f):

```
SELECT AVG(S.age) AS average_age
FROM Student AS S
WHERE S.major = 'Data Science'
  AND S.snum IN (SELECT E.snum FROM Enrolled AS E WHERE E.cname = 'DSC 80')
  AND S.snum IN (SELECT E.snum FROM Enrolled AS E WHERE E.cname = 'DSC 100');
```

  (Note: Assumes "DSC 80" and "DSC 100" are exact class names including the space).

g) Write the SQL statements that find the names of students not enrolled in any class.

```
Answer:
SELECT s.name
FROM Students as s, Enrolled as e
JOIN s.snum ON e.snum
WHERE e.snum =0
```

Evaluation and Correction for 11g):

- The join 'JOIN s.snum ON e.snum' is incorrect syntax. It should be 'ON S.snum = E.snum'.
- 'WHERE e.snum = 0' is an incorrect way to find students not enrolled. It implies there's an enrollment record with 'snum=0' for non-enrolled students, which is not standard.
- The correct way to find non-enrolled students is using 'LEFT JOIN' and checking for 'NULL' in the joined table, or using 'NOT EXISTS' or 'NOT IN'.

  Corrected SQL for 11g):

```
SELECT S.sname
FROM Student AS S
LEFT JOIN Enrolled AS E ON S.snum = E.snum
WHERE E.snum IS NULL;
```

  (Alternatively, using 'NOT EXISTS' or 'NOT IN' is also valid).

h) Find the names of students that are either junior (Student.level = "Junior") or enrolled in at most 2 courses.

```
Answer:
SELECT s.level
FROM Students AS s, Enrolled as e
JOIN s.snum ON e.sum
WHERE s.level='Junior' AND e.snum<=2
```

Evaluation and Correction for 11h):

- The query should 'SELECT s.sname', not 's.level'.
- The join condition is again syntactically incorrect ('ON e.sum' should be 'ON S.snum = E.snum').

- The condition ‘e.snum ¡= 2‘ is nonsensical for counting courses; ‘e.snum‘ is a student ID. You need to count the number of courses per student and then filter.
- The ‘AND‘ should be an ‘OR‘ as the question states ”either... or...”.

Corrected SQL for 11h):

```
SELECT DISTINCT S.sname
FROM Student AS S
WHERE S.level = 'Junior'
   OR S.snum IN (
       SELECT E.snum
       FROM Enrolled AS E
       GROUP BY E.snum
       HAVING COUNT(E.cname) <= 2
   );
```

Question 12. Consider the following schema: Supplier (sid: integer, sname: string, city: string , state: string) Part (pid: integer, pname: string, size: string, color: string) Supply (sid: integer, pid: integer, cost: real)

The keys are underlined, and the domain of each attribute is listed after the attribute name. The “Supply” relation consists of the prices for parts supplied by the Suppliers. ($\rho_R(E)$ renames the result of expression E as R) Write the SQL queries corresponding to the following RA expressions:

a)

$$\pi_{\mathbf{sid}}(\pi_{\mathbf{pid}}(\sigma_{\mathbf{city='La\ Jolla'} \wedge \mathbf{state='CA'}}\mathbf{Supplier})) \bowtie \sigma_{\mathbf{cost}>100}\mathbf{Supply}$$

```
Answer:
SELECT DISTINCT Supply.sdi
FROM Supply, Supplier
WHERE Supplier.sdi =Supply.sdi
AND Supplier.city ='La Jolla'
AND Supplier.state ='CA'
AND Supply.cost >100
```

Evaluation and Correction for 12a):

- The ‘WHERE Supplier.sdi =Supply.sdi‘ is a typo and should be ‘Supplier.sid = Supply.sid‘.
- The ‘FROM Supply, Supplier‘ implies a cross-join, and the ‘WHERE‘ clause acts as an implicit join condition. This is acceptable, but explicit ‘JOIN‘ syntax is preferred.
- The relational algebra expression suggests a slightly different flow: 1. Select Suppliers in La Jolla, CA. 2. Project ‘sid‘ from those suppliers. 3. Select ‘Supply‘ records where ‘cost ¿ 100‘. 4. Join the results of steps 2 and 3 on ‘sid‘. 5. Project ‘sid‘ from the final join. Your SQL correctly captures the join and selection logic to get the ‘sid‘s that match both conditions. The structure of the RA expression with an intermediate projection on ‘pid‘ then ‘sid‘ on the ‘Supplier‘ side is a bit unusual if only ‘sid‘ is ultimately needed for the join. However, the SQL achieves the correct final set of SIDs.

Corrected SQL for 12a) (Minor fix and explicit join):

```
SELECT DISTINCT S.sid
FROM Supplier AS S
JOIN Supply AS Su ON S.sid = Su.sid
WHERE S.city = 'La Jolla'
  AND S.state = 'CA'
  AND Su.cost > 100;
```

b)

$$\pi_{\mathbf{city}}(\sigma_{\mathbf{c}>1000}(\gamma_{\mathbf{city},\mathbf{count(*)\ c}}(\mathbf{Part} \bowtie (\sigma_{\mathbf{cost}<100}\mathbf{Supply}) \bowtie (\sigma_{\mathbf{state='CA'}}\mathbf{Supplier})))$$

```
Answer:
SELECT Supplier.city
FROM Part as p
JOIN Supply ON p.pid = Supply.pid
JOIN Supplier ON supply.sid =Supplier.sid
WHERE Supply.cost < 100
AND Supplier.state= 'CA'
AND COUNT(Supplier.city)> 1000
```

Evaluation and Correction for 12b):

- The relational algebra expression uses $\gamma_{\text{city,count(*) c}}$ which means 'GROUP BY city' and then count how many tuples (after the joins and selections) fall into each city. The 'c ¿ 1000' is then a 'HAVING' clause.
- The provided SQL puts 'COUNT(Supplier.city)¿ 1000' in the 'WHERE' clause, which is a syntax error because aggregate functions cannot be used directly in 'WHERE'. It needs a 'GROUP BY' and 'HAVING'.
- The final 'SELECT Supplier.city' needs to be part of the 'GROUP BY' clause.

  Corrected SQL for 12b):

```
SELECT S.city
FROM Part AS P
JOIN Supply AS Su ON P.pid = Su.pid
JOIN Supplier AS S ON Su.sid = S.sid
WHERE Su.cost < 100
  AND S.state = 'CA'
GROUP BY S.city
HAVING COUNT(*) > 1000;
```

c)

$$\rho_{R1}(\pi_{\textbf{sid}}((\pi_{\textbf{pid}}(\sigma_{\textbf{size}>200}\textbf{Part})) \bowtie \textbf{Supply}))$$

$$\rho_{R2}(\pi_{\textbf{sid}}(\sigma_{\textbf{state='WA'} \vee \textbf{state='CA'}}\textbf{Supplier}))$$

$$R1 \cup R2$$

```
Answer:
SELECT DISTINCT Supply.sdi
FROM Part AS p
JOIN Supply ON p.pid=Supply.pid
WHERE p.size > 100
UNION
SELECT sid
FROM Supplier AS s
WHERE s.state= 'WA' OR s.state= 'CA'
```

Evaluation and Correction for 12c):

- In the first part, 'p.size ¿ 100' is used in SQL, but RA has 'size ¿ 200'. Let's assume the RA is the source of truth, so it should be '¿ 200'.
- The 'Supply.sdi' typo should be 'Supply.sid'.
- The 'UNION' operator is correct for relational algebra union. 'DISTINCT' is implied by 'UNION' in SQL, but 'UNION ALL' would correspond to bag union. Given the RA 'union' symbol, 'UNION' (which removes duplicates) is appropriate.
- The 'SELECT sid FROM Supplier AS s' is correct.

  Corrected SQL for 12c):

```
SELECT DISTINCT Su.sid
FROM Part AS P
JOIN Supply AS Su ON P.pid = Su.pid
WHERE P.size > 200 -- Corrected based on RA expression
UNION
```

```sql
SELECT S.sid
FROM Supplier AS S
WHERE S.state = 'WA' OR S.state = 'CA';
```

Question 13. Consider the Netflix database schema discussed in the lectures: Schema: R (RatingID, Stars, RateDate, UID, MID) U (UID, Name, Age, JoinDate) M (MID, Name, Year, Director)

For each query given below, write a CREATE INDEX statement that can help speed up that query. Make sure to clearly mention the index type and SearchKey.

a) SELECT Director FROM M WHERE Year ¿= 2022;

Answer:

```sql
CREATE INDEX index_movie_year ON M USING BTREE (Year)
```

Evaluation for 13a):

- Correct. A B+ tree index is excellent for range queries (‘¿=‘). The search key is ‘Year‘.

b) SELECT Name FROM U WHERE Age = 18;

Answer:

```sql
CREATE INDEX index_user_age_hash ON U USING HASH (Age)
```

Evaluation for 13b):

- Correct. A hash index is highly efficient for equality lookups (‘=‘). The search key is ‘Age‘. A B+ tree would also work, but hash is typically faster for exact matches.

c) SELECT * FROM R, U WHERE R.UID = U.UID;

Answer:

```sql
CREATE INDEX index_rating_uid_hash ON R USING HASH (UID);
```

Evaluation for 13c):

- Partially correct. An index on ‘R.UID‘ (the foreign key) is good. An index on ‘U.UID‘ (the primary key) is also crucial. For join conditions, it’s generally beneficial to index both sides of the join. Either hash or B-tree would work. **Improved answer for 13c):**

```sql
CREATE INDEX index_rating_uid_hash ON R USING HASH (UID);
-- AND
CREATE INDEX index_user_uid_hash ON U USING HASH (UID);
```

d) SELECT * FROM R WHERE RateDate ¿= 01/01/2021;

```
Answer:
CREATE INDEX index_rating_date ON R USING BTREE (RateDate)
\end{lstlisting>
\textbf{Evaluation for 13d):}
\begin{itemize}
    \item Correct. A B+ tree index is ideal for range queries on dates. The search key
        is RateDate.
\end{itemize}

\item Given the following schema:
    schema:
    Product($\underline{\text{ProductID}}$, Brand, Type, Price)
    Orders($\underline{\text{OrderID}}$, ProductID, OrderDate, Amount)

    The primary key columns are underlined, and the foreign key ProductID in the Product
        table references the ProductID column in the Product table.
    Write a SQL query that calculates the total sales for each brand and type including
        brands and types that have no sales, and orders for which the product information
        is missing, i.e., their product ID is NULL. The query must be executable on SQLite
        , which means you can only use SQL constructs that are supported by SQLite
    \begin{lstlisting}
```

```
Answer:
SELECT outer_q.Brand, outer_q.Type, SUM(outer_q.sales) AS total_sales
FROM (
/* 1. Products (LEFT JOIN) -> keeps rows with zero sales */
    SELECT p.Brand AS Brand,
           p.Type AS Type,
           COALESCE(p.Price * o.Amount, 0) AS sales
      FROM Product AS p
      LEFT JOIN Orders AS o
           ON o.ProductID = p.ProductID

     UNION ALL
/* 2. Orders whose ProductID IS NULL -> treat as "unknown" */
    SELECT '(unknown)' AS Brand, -- label of your choice
           '(unknown)' AS Type,
            SUM(o2.Amount) AS sales -- no price info
      FROM Orders AS o2
     WHERE o2.ProductID IS NULL
     GROUP BY Brand, Type -- collapses to one row
     ) AS outer_q
GROUP BY outer_q.Brand, outer_q.Type
ORDER BY outer_q.Brand, outer_q.Type;
```

Evaluation for 14):

- This is a very well-structured and complex query that correctly addresses all parts of the requirement:
  - ‘LEFT JOIN Product‘ with ‘Orders‘ correctly includes products with no sales (by using ‘COALESCE(p.Price * o.Amount, 0)‘ to handle NULL sales as 0).
  - The ‘UNION ALL‘ with the second subquery ‘(SELECT ... WHERE o2.ProductID IS NULL)‘ correctly handles orders with missing product information, labeling them as ’(unknown)’ and summing their amounts (as price info is missing).
  - The outer ‘GROUP BY Brand, Type‘ aggregates the sales for all categories, including the ’unknown’ one.
  - ‘ORDER BY Brand, Type‘ provides a structured output.
  - The use of ‘COALESCE‘ and ‘UNION ALL‘ is standard SQL and supported by SQLite.
- Minor point: The second subquery has ‘GROUP BY Brand, Type‘ but selects ‘’(unknown)’ AS Brand, ’(unknown)’ AS Type‘, so it will always collapse to a single row for ’(unknown)’ brand and type. This is functionally correct for the goal.

  Overall Assessment for 14): This is an excellent solution to a challenging SQL problem, demonstrating a strong grasp of ‘JOIN‘ types, aggregation, ‘UNION‘, and handling ‘NULL‘ values.