1 **(a)** $\mathrm{R1} \coloneqq \mathrm{PROJECT\_GROUP},\ \mathrm{R2} \coloneqq \mathrm{PROJECT\_GROUP}$

$\mathrm{StudentPairs} \coloneqq \Pi_{\mathrm{R1.sID\ a\ sID1,\ R2.sID\ as\ sID2,\ courseID}} (\mathrm{R1} \bowtie_{\mathrm{R1.groupID=R2.groupID\ AND\ R1.sID<R2.sID}} R2)$

$\mathrm{PairCount} \coloneqq \gamma_{\mathrm{sID1,\ sID2,\ COUNT(courseID)\rightarrow pairCount}}\ \mathrm{StudentPairs}$

$\mathbf{Answer} \coloneqq \boldsymbol{\sigma}_{\mathbf{pairCount\geq 3}} \boldsymbol{PairCount}$

(b) Continuing from 1(a),

$$\mathrm{StudentsList} \coloneqq \delta\big((\Pi_{\mathrm{sID1\rightarrow sID}}\mathrm{Answer}) \cup (\Pi_{\mathrm{sID2\rightarrow sID}} Answer)\big)$$

$$\mathrm{StudentInternships} \coloneqq \mathrm{StudentsList} \bowtie_{\mathrm{L}} \mathrm{TAKE\_INTERNSHIP} \bowtie \mathrm{INTERNSHIP}$$

$$\mathrm{R3} \coloneqq \gamma_{\mathrm{sID,\ aID,\ salary,\ startDate,\ endDate,\ country,\ COUNT(uID)\rightarrow uIDCount}}\ \mathrm{StudentInternships}$$

$$R4 \coloneqq \gamma_{\mathrm{sID,\ COUNT(aID)\rightarrow uniqueInternCount}}\ \mathrm{R3}$$

$$\mathbf{Answer} \coloneqq \boldsymbol{\sigma}_{\mathbf{uniqueInternCount\ =1}} \mathbf{R4}$$

Explanation:
We get the list of students from 1(a) by using union, then combine it with their respective internships. Then, we group these StudentInternships based on their student ID, company ID (aID), salary, startDate, endDate, and country.

If this student does have internships with different company, salary, startDate, endDate, or country, then there should be more than one row in R4 with the same sID. Then, we will find the sID that only appears once in R4.

2 **(a)** The candidate keys for this relation are AC and BC (with these key, we can determine the remaining keys).

The rule for BCNF is that for each functional dependency (FD):

- It is a trivial FD, or
- The LHS is a superkey for schema R.

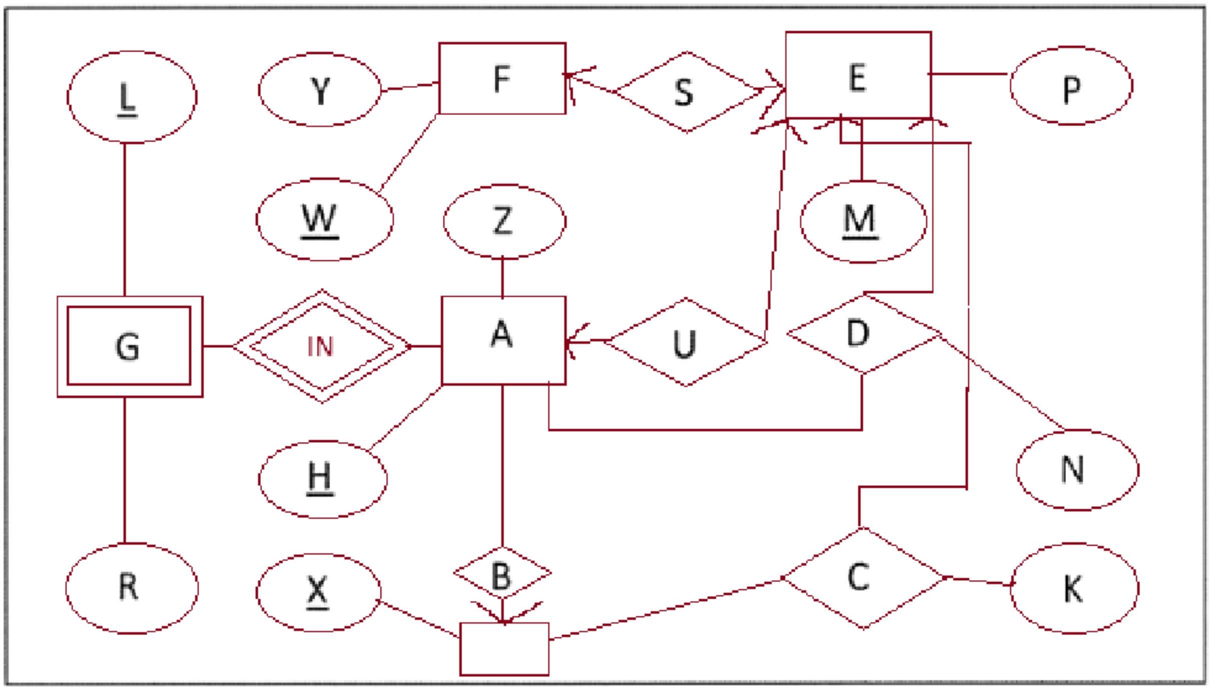
The FDs that broke these laws are $A \rightarrow B$, $BD \rightarrow A$, $CD \rightarrow E$, $AB \rightarrow D$.
Using the decomposition algorithm provided in the lecture,

1. $A \rightarrow B$
   $\{A\}^+ = \{A, B, D\}$
   Split R into R1(A, B, D) and R2(A, C, E)

   For both R1 and R2, none of the FDs breaks the rule.

Thus, the decomposition would result in R1(A, B, D) and R2(A, C, E).
However, it does not preserve all functional dependencies, specifically $CD \rightarrow E$, because the decomposed relations does not have C, D and E together.

(b)



3 (a) (i)

```
WITH MaxCid AS (
      SELECT DISTINCT cid FROM Owns
      WHERE aid IN (
          SELECT aid FROM Account
          WHERE amount = (
              SELECT MAX(amount) FROM Account
          )
    )
)

SELECT c.name, cc.address FROM PremierCustomer c
JOIN MaxCid mc ON c.cid = mc.cid
JOIN Customer cc ON cc.cid = c.cid;
```

(ii)

```
WITH OwnsType AS (
      SELECT o.cid AS cid, a.type AS type, COUNT(a.aid) AS count
FROM Owns o
      JOIN Account a ON o.aid=a.aid
      GROUP BY o.cid, a.type
)

SELECT cid FROM OwnsType
WHERE type='Checking' AND count>=2
INTERSECT
SELECT cid FROM OwnsType
```

```
WHERE type='Saving' AND count>=1;
```

(iii)

```
WITH OtherTransactions AS (
      SELECT al.cid AS src, o.cid AS target FROM ActivityLog al
      JOIN Owns o ON al.aid2=o.aid
      WHERE optype='transfer' AND al.cid != o.cid
)

SELECT name FROM Customer
WHERE cid NOT IN (
      SELECT src AS cid FROM OtherTransactions
      UNION
      SELECT target AS cid FROM OtherTransactions
);
```

(b) **(i)** 1, 1, 2

(ii) James, Anna

4 **(a)** Views are query over the base relation to produce another table, but it is virtual (does not exist until it is being executed or called).
Pro: Does not take up space.
Cons: Queries defined on the view needs to be executed on-the-fly.

Materialized views are like views, except that the table does exist and being kept as a temporary table.
Pro: Can be used to answer queries efficiently as its result are being stored physically.
Cons: Consumes space and needs to be updated when the base table is modified.

Temporary view only exists within the lifespan of the query, unlike views/materialized views that can be reused for other queries.
Pro: Can improve query readability and maintainability in complex scripts, as well as not cluttering the database with permanent objects.
Cons: Cannot be shared between different sessions/queries

(b) Trigger for insertions:

```
CREATE TRIGGER trg_ActivityLog_Insert
AFTER INSERT ON ActivityLog
REFERENCING NEW ROW AS NewTuple
FOR EACH ROW
BEGIN
    IF NewTuple.optype = 'deposit' THEN
        UPDATE Account
        SET amount = amount + NewTuple.amount
        WHERE aid = NewTuple.aid1;
```

```
    ELSEIF NewTuple.optype = 'withdrawal' THEN
        UPDATE Account
        SET amount = amount - NewTuple.amount
        WHERE aid = NewTuple.aid1;
    ELSEIF NewTuple.optype = 'transfer' THEN
        UPDATE Account
        SET amount = amount - NewTuple.amount
        WHERE aid = NewTuple.aid1;

        UPDATE Account
        SET amount = amount + NewTuple.amount
        WHERE aid = NewTuple.aid2;
    END IF;
END;
```

Trigger for update (reversing old operation and apply new one):

```
CREATE TRIGGER trg_ActivityLog_Update
AFTER UPDATE ON ActivityLog
REFERENCING OLD ROW AS OldTuple NEW ROW AS NewTuple
FOR EACH ROW
BEGIN
    IF OldTuple.optype = NewTuple.optype THEN
        IF OldTuple.optype = 'deposit' THEN
            UPDATE Account
            SET amount = amount - OldTuple.amount + NewTuple.amount
            WHERE aid = NewTuple.aid1;
        ELSEIF OldTuple.optype = 'withdrawal' THEN
            UPDATE Account
            SET amount = amount + OldTuple.amount - NewTuple.amount
            WHERE aid = NewTuple.aid1;
        ELSEIF OldTuple.optype = 'transfer' THEN
            UPDATE Account
            SET amount = amount + OldTuple.amount - NewTuple.amount
            WHERE aid = NewTuple.aid1;

            UPDATE Account
            SET amount = amount - OldTuple.amount + NewTuple.amount
            WHERE aid = NewTuple.aid2;
        END IF;
    ELSE
        -- Handle optype change (more complex scenario)
        -- Reverse the old operation
        IF OldTuple.optype = 'deposit' THEN
            UPDATE Account
```

```
            SET amount = amount - OldTuple.amount
            WHERE aid = OldTuple.aid1;
        ELSEIF OldTuple.optype = 'withdrawal' THEN
            UPDATE Account
            SET amount = amount + OldTuple.amount
            WHERE aid = OldTuple.aid1;
        ELSEIF OldTuple.optype = 'transfer' THEN
            UPDATE Account
            SET amount = amount + OldTuple.amount
            WHERE aid = OldTuple.aid1;

            UPDATE Account
            SET amount = amount - OldTuple.amount
            WHERE aid = OldTuple.aid2;
        END IF;

        -- Apply the new operation
        IF NewTuple.optype = 'deposit' THEN
            UPDATE Account
            SET amount = amount + NewTuple.amount
            WHERE aid = NewTuple.aid1;
        ELSEIF NewTuple.optype = 'withdrawal' THEN
            UPDATE Account
            SET amount = amount - NewTuple.amount
            WHERE aid = NewTuple.aid1;
        ELSEIF NewTuple.optype = 'transfer' THEN
            UPDATE Account
            SET amount = amount - NewTuple.amount
            WHERE aid = NewTuple.aid1;

            UPDATE Account
            SET amount = amount + NewTuple.amount
            WHERE aid = NewTuple.aid2;
        END IF;
    END IF;
END;
```

Trigger for deletion:

```
CREATE TRIGGER trg_ActivityLog_Delete
AFTER DELETE ON ActivityLog
REFERENCING OLD ROW AS OldTuple
FOR EACH ROW
BEGIN
    IF OldTuple.optype = 'deposit' THEN
```

```
        UPDATE Account
        SET amount = amount - OldTuple.amount
        WHERE aid = OldTuple.aid1;
    ELSEIF OldTuple.optype = 'withdrawal' THEN
        UPDATE Account
        SET amount = amount + OldTuple.amount
        WHERE aid = OldTuple.aid1;
    ELSEIF OldTuple.optype = 'transfer' THEN
        UPDATE Account
        SET amount = amount + OldTuple.amount
        WHERE aid = OldTuple.aid1;

        UPDATE Account
        SET amount = amount - OldTuple.amount
        WHERE aid = OldTuple.aid2;
    END IF;
END;
```

(c) **(i)** It will speed up any queries that involves finding on attribute Price, which includes finding the highest price in the table. However, it will not speed up finding product with the highest amount as the calculation (the product of Quantity and Price) is not being indexed.

(ii) Increases the time needed to perform insertion, update and deletion on the table (as it needs to update the index in addition to the table).

(iii) One possible composite index would be:

```
ALTER TABLE Purcahse
ADD INDEX date_price (Date, Price)
```

Which can speed up the following query:

```
SELECT Pname
FROM Purchase WHERE Date BETWEEN '2023-01-01' AND '2023-12-31' AND
Price > 1.00
```

(d) **(i)** Users (<u>UserID</u>, Name, Age, Sex)
Tweets(<u>TweetID</u>, UserID, Timestamp, Content)

(ii)

```
<!ELEMENT twitter_data (users, tweets)>

<!ELEMENT users (user+)>
<!ELEMENT user (name, age, sex)>
<!ATTLIST user id ID #REQUIRED>

<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA)>
```

```
<!ELEMENT sex (#PCDATA)>

<!ELEMENT tweets (tweet+)>
<!ELEMENT tweet (content)>
<!ATTLIST tweet
    id ID #REQUIRED
    user_id IDREF #REQUIRED
    timestamp CDATA #REQUIRED>

<!ELEMENT content (#PCDATA)>
```

Solver: Clayton Fernalo (cfernalo001@e.ntu.edu.sg)