Triggers

```
DELIMITER $$

CREATE TRIGGER update_community_score
AFTER INSERT ON Rating
FOR EACH ROW
BEGIN
    DECLARE countRatings INT;
    DECLARE averageRating DECIMAL(10, 2);

    -- Calculate the count of ratings for the given address
    SELECT COUNT(*) INTO countRatings
    FROM Rating
    WHERE address = NEW.address;

    -- Check if the count is greater than or equal to 2
    IF countRatings >= 2 THEN
        -- Calculate the average rating
        SELECT AVG(rating) INTO averageRating
        FROM Rating
        WHERE address = NEW.address;

        -- Insert or update the aptCommunityScore table
        INSERT INTO aptCommunityScore (address, communityScore)
        VALUES (NEW.address, averageRating)
        ON DUPLICATE KEY UPDATE communityScore = averageRating;
    END IF;
END $$

DELIMITER ;
```

```
DELIMITER $$

CREATE TRIGGER update_community_score_2
AFTER UPDATE ON Rating
FOR EACH ROW
BEGIN
    DECLARE countRatings INT;
    DECLARE averageRating DECIMAL(10, 2);
```

```
     -- Calculate the count of ratings for the given address
     SELECT COUNT(*) INTO countRatings
     FROM Rating
     WHERE address = NEW.address;

     -- Check if the count is greater than or equal to 2
     IF countRatings >= 2 THEN
         -- Calculate the average rating
         SELECT AVG(rating) INTO averageRating
         FROM Rating
         WHERE address = NEW.address;

         -- Insert or update the aptCommunityScore table
         INSERT INTO aptCommunityScore (address, communityScore)
         VALUES (NEW.address, averageRating)
         ON DUPLICATE KEY UPDATE communityScore = averageRating;
     END IF;
 END $$

 DELIMITER ;
```

```
DELIMITER $$

CREATE TRIGGER update_community_score_3
AFTER DELETE ON Rating
FOR EACH ROW
BEGIN
    DECLARE countRatings INT;
    DECLARE averageRating DECIMAL(10, 2);

    -- Calculate the count of ratings for the given address
    SELECT COUNT(*) INTO countRatings
    FROM Rating
    WHERE address = OLD.address;

    -- Check if the count is greater than or equal to 2
    IF countRatings >= 2 THEN
        -- Calculate the average rating
        SELECT AVG(rating) INTO averageRating
        FROM Rating
        WHERE address = OLD.address;
        -- Insert or update the aptCommunityScore table with the average
rating
        INSERT INTO aptCommunityScore (address, communityScore)
        VALUES (OLD.address, averageRating)
        ON DUPLICATE KEY UPDATE communityScore = averageRating;
    ELSE
        -- Set the communityScore to -1 if there are less than 2 ratings
        INSERT INTO aptCommunityScore (address, communityScore)
        VALUES (OLD.address, -1)
        ON DUPLICATE KEY UPDATE communityScore = -1;
    END IF;
END $$

DELIMITER ;
```

Procedure

```
DELIMITER //

CREATE PROCEDURE aptCommunityScore()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE currapt VARCHAR(255);
    DECLARE rating_count INT;
    DECLARE aptcur CURSOR FOR SELECT DISTINCT address FROM Apartment;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    DROP TABLE IF EXISTS aptCommunityScore;
    CREATE TABLE aptCommunityScore (
        address VARCHAR(255) PRIMARY KEY,
        communityScore REAL
    );

    OPEN aptcur;

    REPEAT
        FETCH aptcur INTO currapt;

        IF NOT done THEN

            SELECT COUNT(*) INTO rating_count
            FROM Rating
            WHERE address = currapt;

            IF rating_count < 2 THEN
                INSERT INTO aptCommunityScore (address, communityScore)
                VALUES (currapt, -1);
            ELSE
                INSERT INTO aptCommunityScore (address, communityScore)
                SELECT address, AVG(rating)
                FROM Rating
                WHERE address = currapt
GROUP BY address;
            END IF;
        END IF;

    UNTIL done
    END REPEAT;

    CLOSE aptcur;
END //

DELIMITER ;
```

Transaction

```
router.post('/filter', async (req, res) => {
    const { filters } = req.body;

    if (!Array.isArray(filters) || filters.length === 0) {
        return res.status(400).send('Invalid input. Provide an array of
filters.');
    }

    const tempTables = [];
    try {
        // Set isolation level
        await new Promise((resolve, reject) => {
            connection.query(
                'SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;',
                (err) => {
                    if (err) return reject(err);
                    resolve();
                },
            );
        });

        // Start a transaction
        await new Promise((resolve, reject) => {
            connection.beginTransaction((err) => {
                if (err) return reject(err);
                resolve();
            });
        });

        // Create temporary tables
        for (let i = 0; i < filters.length; i++) {
            const filter = filters[i];
            const tempTableName = `t${i}`;
            tempTables.push(tempTableName);

            // Drop the temporary table if it already exists
            await new Promise((resolve, reject) => {
                const dropTempTableQuery = `DROP TEMPORARY TABLE IF EXISTS
${tempTableName};`;
                connection.query(dropTempTableQuery, (err) => {
                    if (err) return reject(err);
                    resolve();
                });
            });

            let createTempTableQuery = `
                CREATE TEMPORARY TABLE ${tempTableName} AS
            `;
            let condition =
                filter.bound_type === 'AT_LEAST'
                    ? `>= ${filter.threshold}`
                    : `<= ${filter.threshold}`;
            if (filter.subject === 'STREETLIGHTS') {
                createTempTableQuery += `
                    SELECT a.address, a.safestay_score, a.latitude,
a.longitude, a.block
                    FROM Apartment a
                    JOIN Streetlight s ON a.block = s.block
                    GROUP BY a.address, a.safestay_score, a.latitude,
a.longitude, a.block
                    HAVING COUNT(s.streetlight_id) ${condition};
                `;
            } else if (filter.subject === 'CRASHES') {
                createTempTableQuery += `
```

```
                    SELECT a.address, a.safestay_score, a.latitude,
a.longitude, a.block
                    FROM Apartment a
                    JOIN Pedestrian_Crash p ON a.block = p.block
                    GROUP BY a.address, a.safestay_score, a.latitude,
a.longitude, a.block
                    HAVING COUNT(p.crash_id) ${condition};
                `;
            } else if (filter.subject === 'SAFESTAY_SCORE') {
                createTempTableQuery += `
                    SELECT * FROM Apartment
                    WHERE safestay_score ${condition};
                `;
            } else {
                createTempTableQuery += `
                    SELECT a.address, a.safestay_score, a.latitude,
a.longitude, a.block
                    FROM Apartment a
                    JOIN aptCommunityScore cs ON a.address = cs.address
                    WHERE cs.communityScore ${condition};
                `;
            }

            // Execute query
            await new Promise((resolve, reject) => {
                connection.query(createTempTableQuery, (err) => {
                    if (err) return reject(err);
                    resolve();
                });
            });
        }

        // Intersect all temporary tables
        const intersectQuery = tempTables.reduce((acc, tableName, index) => {
            if (index === 0) return tableName;
            return `${acc} INNER JOIN ${tableName} USING (address)`;
        });

        const finalQuery = `SELECT * FROM ${intersectQuery};`;
        const rows = await new Promise((resolve, reject) => {
            connection.query(finalQuery, (err, results) => {
                if (err) return reject(err);
                resolve(results);
            });
        });
        // Commit the transaction
        await new Promise((resolve, reject) => {
            connection.commit((err) => {
                if (err) return reject(err);
                resolve();
            });
        });

        // Return results
        res.json(rows);
    } catch (error) {
        console.error('Error during transaction:', error);

        // Rollback the transaction
        await new Promise((resolve) => {
            connection.rollback(() => {
                resolve();
            });
        });

        res.status(500).send('An error occurred while executing the queries.');
    }
});
```