# CSC343 Worksheet 6 Solution

June 22, 2020

1. **Exercise 6.6.1:**

a)
```
SET TRANSACTION READONLY;
BEGIN TRANSACTION;
    SELECT model, price FROM PC
    WHERE speed = speed AND
    ram=ram
COMMIT;

```

**Notes:**

- Transactions
  - is a collection of one or more operations that must be executed atomically
  - COMMIT causes the transaction to end successfully
  - ROLLBACK causes the transaction to abort. Any changes are undone
  - SET TRANSACTION READ ONLY
    * tells the database that it will not be modified
    * Must be declared before transaction
    * Is useful when one user is running multiple queries while other is updating the same table

**Example:**

```
BEGIN TRANSACTION;

UPDATE accounts
SET balance = balance - 1000
WHERE account_no = 100;

UPDATE accounts
    SET balance = balance + 1000
WHERE account_no = 200;

INSERT INTO account_changes(account_no,flag,amount,
changed_at)
```

```
    VALUES(100,'-',1000,datetime('now'));

    COMMIT;

    // Example - SET TRANSACTION READONLY
    SET TRANSACTION READONLY;
    BEGIN TRANSACTION;
        ...
    COMMIT;

```

b)
```
    BEGIN TRANSACTION;
    DELETE FROM PC
    WHERE model=<model number>

    DELETE FROM Product
    WHERE model=<model number>

    COMMIT;

```

c)
```
    BEGIN TRANSACTION;

    UPDATE PC
    SET price=price - 100
    WHERE model=<model number>

    COMMIT;

```

d)
```
    BEGIN TRANSACTION;

    IF (<model> IN (
        SELECT <model> FROM Product
        NATURAL JOIN PC)

        PRINT 'Error occured';
    ELSE
        INSERT INTO PC
        VALUES (<model>, <speed>, <ram>, <hd>, <price>)

        INSERT INTO Product
        VALUES (<maker>, <model>, <type>)
    COMMIT;

```

2. **Exercise 6.6.2:**

For all cases, when system crashes, the operations in transaction are aborted and database is reverted back to pre-transaction state.

3. **Exercise 6.6.3:**

   **Notes:**

   - Isolation Levels
     - SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
     - SET TRANSACTION ISOLATION LEVEL REPEATIBLE READ;
     - SET TRANSACTION ISOLATIO LEVEL SERIALIZABLE

4. **Exercise 8.1.1:**

   a)
```
    CREATE VIEW RichExec AS
        SELECT * FROM MovieExec
        WHERE netWorth >= 10000000;

```

   **Notes:**

   - Virtual Views
     - **Syntax:** CREATE VIEW < view-name > AS < view-definition >
     - Contrasts to database that exists in physical storage
     - Exists in RAM
     - Is created using query
     - can be used like a relation

     **Notes:**

```
        CREATE VIEW ParamountMovies AS
            SELECT title, year
            FROM Movies
            WHERE studioName = 'Paramount';

```

   b)
```
    CREATE VIEW StudioPres AS
        SELECT * FROM Movies
        INNER JOIN Studio ON cert# = presC#;

```

   c)
```
    CREATE VIEW ExecutiveStar AS
        SELECT * FROM MovieExec
        NATURAL JOIN MovieStar;

```

5. **Exericse 8.1.2:**

a)
```sql
SELECT name, gender FROM ExecutiveStar;
```

b)
```sql
SELECT name FROM RichExec WHERE netWorth > 10000000;
```

c)
```sql
SELECT name FROM StudioPres
NATURAL JOIN ExecutiveStar
WHERE netWorth > 50000000
```

6. **Exericse 8.2.1:**

*RichExec* is updatable.

**Notes:**

- Updatable View Conditions
  - The WHERE cluase in CREATE VIEW must not be a subquery
  - The FROM clause has only one occurence of R
  - The SELECT clause must include enough attributes
  - NOT NULL attributes must have default values
    - A solution to this is by including the attribute without default value in CREATE VIEW

  **Example:**

```
Movies(title, year, length, genre, studioName, producerC#)
Suppose studioName is NOT NULL but has no default value.
Then, a fix is:

CREATE VIEW Paramount AS
    SELECT studioName, title, year
    FROM Movies
    WHERE studioName = 'Paramount';
```

7. **Exericse 8.2.2:**

a) No. It is not updatable. Since,

   1. studioName attribute in Movies is NOT NULL without default value

b)
```sql
CREATE TRIGGER DisneyComediesInsert
INSTEAD OF INSERT ON DisneyComedies
REFERENCING
    NEW ROW AS NewTuple
FOR EACH ROW
INSERT INTO Movies(title, year, length, genre, studioName)
VALUES(NewTuple.title, NewTuple.year, NewTuple.length, 'comedy',
'Disney');
```

**Notes:**

- Using Trigger in VIEW
  - Uses INSTEAD OF in place of BEFORE or AFTER
  - When event causes the trigger, the trigger is done instead of the event

  **Example:**

```sql
CREATE VIEW ParamountMovies AS
    SELECT title, year
    FROM Movies
    WHERE studioName = 'paramount';

CREATE TRIGGER ParamountInsert
INSTEAD OF INSERT ON ParamountMovies
REFERENCING NEW ROW AS NewRow
FOR EACH ROW
INSERT INTO Movies(title, year, studioName)
VALUES(NewRow.title, NewRow.year, 'Paramount');
```

c)
```sql
CREATE TRIGGER DisneyComediesInsert
INSTEAD OF INSERT ON DisneyComedies
REFERENCING
    NEW ROW AS NewTuple
    OLD ROW AS OldTuple
FOR EACH ROW
UPDATE Movies
SET length=NewTuple.length
WHERE title=OldTuple.title AND year=OldTuple.year;
```

8. **Exercise 8.2.3**

a) No. the view is not updatable. Because for it to be updatable, only one relation must exist in FROM

b)
```sql
CREATE TRIGGER NewPCInsert
INSTEAD OF INSERT ON NewPC
REFERENCING
```

```
        NEW ROW AS NewTuple
        OLD ROW AS OldTuple
    FOR EACH ROW
    INSERT INTO PC(model speed, ram, hd ,price)
    VALUES (NewTuple.model, NewTuple.speed, NewTuple.ram, NewTuple.hd
   , NewTuple.price);

    INSERT INTO Product(maker, model, type)
    VALUES (NewTuple.maker, NewTuple.model, 'pc');

```

c)

```
    CREATE TRIGGER NewPCUpdate
    INSTEAD OF INSERT ON NewPC
    REFERENCING
        NEW ROW AS NewTuple
    FOR EACH ROW
    UPDATE PC
    SET model=NewTuple.model
        speed=NewTuple.speed,
        ram=NewTuple.ram,
        hd=NewTuple.hd,
        price=NewTuple.price;

    UPDATE Product
    SET maker=NewTuple.maker,
        model=NewTuple.model,
        type='pc';

```

**Correct Solution:**

```
    CREATE TRIGGER NewPCUpdate
    INSTEAD OF UPDATE ON NewPC
    REFERENCING
        NEW ROW AS NewTuple
    FOR EACH ROW
    UPDATE PC
    SET model=NewTuple.model
        speed=NewTuple.speed,
        ram=NewTuple.ram,
        hd=NewTuple.hd,
        price=NewTuple.price;

    UPDATE Product
    SET maker=NewTuple.maker,
        model=NewTuple.model,
        type='pc';

```

d)

```
    CREATE TRIGGER NewPCDelete
    INSTEAD OF DELETE ON NewPC
    REFERENCING
        NEW ROW AS NewTuple
    FOR EACH ROW
    DELETE FROM PC
    WHERE model=NewTuple.model;

    DELETE FROM Product
    WHERE model=NewTuple.model;

```

9. a)

```
    CREATE INDEX studioNameIndex Studio(name)

```

**Notes:**

- Indexes
  - **Syntax (Create Index):**
    CREATE INDEX $<$ index-name $>$ $R(<$ attributes $>)$
  - **Syntax (Drop Index):**
    DROP INDEX $<$ index-name $>$
  - Used to find tuples in a very large database
    - Is efficient
  - Can be thought as (key, value) pair in a binary search tree
  - e.g. Declaring Index

```
    CREATE INDEX KeyIndex ON Movies(title, year);

```

  - e.g. Dropping index

```
    CREATE INDEX KeyIndex ON Movies(title, year);

```

b)

```
    CREATE INDEX movieExecAddressIndex MovieExec(address)

```

c)

```
    CREATE INDEX movieKeyIndex Movies(genre, length)

```

10. **Exercise 8.4.1:**

| Action | No Index | Star Index | Movie Index | Both Indexes |
|---|---|---|---|---|
| $Q_1$ | 100 | 4 | 100 | 4 |
| $Q_2$ | 100 | 100 | 4 | 4 |
| $I$ | 2 | 4 | 4 | 6 |
| Average | $2 + 100p_1 + 100p_2$ | $4 + 96p_2$ | $4 + 96p_1$ | $6 - 2p_1 - 2p_2$ |

**Notes:**

- Database Tuning
  - Index sppeds up queries that can use it
  - Index should NOT be created when modifications are the frequent choice of action

11. **Exercise 8.4.2:**

    Omitted for the time being

12. **Exercise 8.5.1:**

```
    UPDATE MovieProd
    SET name='New Name'
    WHERE (title, year) IN
    (
        SELECT title, year FROM Movies
        INNER JOIN MovieExecs
        ON Movies.productC# = MovieExec.cert#
        WHERE cert# = '4567'
    );

```

<u>**Notes:**</u>

- Materialized Views
  - Is also known as a summary
  - Is also known as black-box abstraction
  - Stores view in physical storage
  - Useful when storing expensive operation like AVG or COUNT