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.

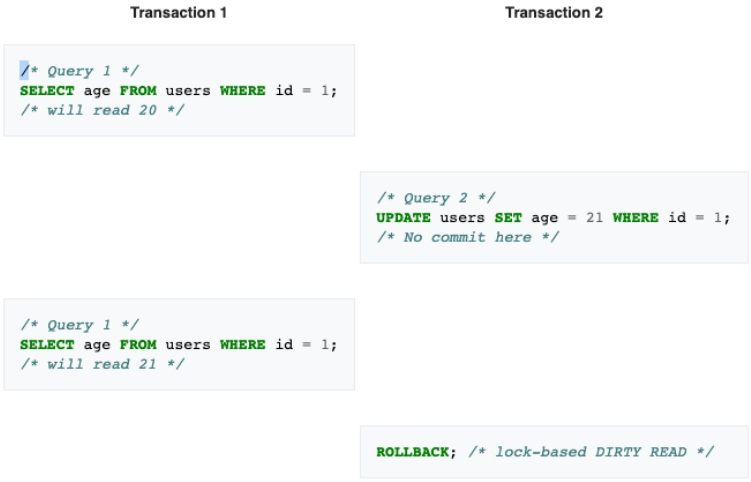
3. **Exercise 6.6.3:**

The following would be observed

- Reading data modified by another transaction (Dirty Read)
- Repeated retrieval of rows resulting in different values (Non-repeatible read)
- Insertion/deletion of data (Phantom)

Notes:

- Dirty Reads
  - A dirty read occurs when a transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed.



- Non-repeatible Reads
  - A non-repeatable read occurs when, during the course of a transaction, a row is retrieved twice and the values within the row differ between reads.

- Phantom Reads
  - A phantom read occurs when, in the course of a transaction, new rows are added or removed by another transaction to the records being read.

- Isolation Levels
  - SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
    - is the lowest isolation level
    - allows to read a transaction that's not yet committed
    - transactions are not isolated from each other
  - SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
    - Does not allow to read a transaction that's not yet committed
    - Prevents other transactions from reading, updating or deleting while commit
  - SET TRANSACTION ISOLATION LEVEL REPEATIBLE READ;
    - Is the higher level of isolation
    - Guarentees everything of READ COMMITTED level
    - Can read unchanged data in subsequent reads
  - SET TRANSACTION ISOLATIO LEVEL SERIALIZABLE
    - Is the highest level of isolation
    - Guarentees everything of READ REPEATIBLE READ;
    - No new data can be seen by a subsequent read.

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not Allowed	Allowed	Allowed
Repeatable Read	Not Allowed	Not Allowed	Allowed
Serializable	Not Allowed	Not Allowed	Not Allowed

References:

- Stack Overflow: Difference between 'read commited' and 'repeatable read', link

* Wikipedia: Isolation (database systems), link

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)
```
SELECT name, gender FROM ExecutiveStar;

```

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

```

c)
```
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)
```
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:

```
    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)

```
    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)

```
    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'
    );

```

Notes:

- 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

13. **Exercise 8.5.3**

The following modifications to base tables require the modification of the materialized view

- PC: Updates(model, speed, ram, hd, price), Delete
- Product: Updates(maker, model, type), Delete

Implementing modifications

- Updates

```
    UPDATE NewPC
    SET maker='new-maker'
        model='new-model-number'
        speed='new-speed'
        ram='new-ram'
        hd='new-hd'
        price='new-price'
    WHERE model = 'old-model-number';

```

- Delete

```
    DELETE FROM NewPC WHERE model = 'old-model-number'

```

Notes:

- Materialized view of NewPC

```
CREATE MATIERLIZED VIEW NewPC AS
    SELECT maker, model, speed, ram, hd, price
    FROM Product, PC
    WHERE Product.model = PC.model AND type = 'pc';

```

14. **Exercise 8.5.3**

The following modifications to base tables require the modification of the materialized view

- Classes: Insert, Updates(class, country, displacement), Delete
- Ships: Insert, Updates(class), Delete

- Insert

```
INSERT INTO ShipStats
VALUES (
    SELECT country, AVG(displacement), COUNT(*)
    FROM Classes, Ships
    WHERE Classes.class = ships.class
    GROUP BY country
    HAVING country='name-of-country'
)

```

- Updates

```
DELETE FROM ShipStats WHERE country = 'name-of-country';
INSERT INTO ShipStats
VALUES (
    SELECT country, AVG(displacement), COUNT(*)
    FROM Classes, Ships
    WHERE Classes.class = ships.class
    GROUP BY country
    HAVING country='name-of-country'
)

```

- Delete

```
DELETE FROM ShipStats WHERE country = 'country'

```

15. **Exercise 8.5.4**

Query Q #1: The address of movie producers whose movie was created in 2019

```
// Example 1
SELECT address
FROM Studio, Movies, MovieExec
WHERE producerC# = cert# year=2019

```

Query Q #2: The networth of movie producers whose movie was created in 2019

```
// Example 2
SELECT netWorth
FROM Studio, Movies, MovieExec
WHERE producerC# = cert# year=2019

```

Notes:

- Rewriting Queries to use Materialized View
  - Conditions under which we can replace part of the query Q by the view V
    (a) The relations in list R_v all appear in the list R_Q
    (b) The condition C_Q is equivalent to C_V AND C for some condition C. As a special case, C_Q could be equivalent to C_V, in which case the "AND C" is unnecessary.
  - Once the conditions are met, Q can be re-written to V as follows
    * Replace the list R_Q by V and the relations that are on list R_Q but not on R_V
    * Replace C_Q by C. If C is not needed (i.e. $C_V = C_Q$), then there is no WHERE clause