

# CSC343 Worksheet 6 Solution

June 22, 2020

## 1. Exercise 6.6.1:

```
a) SET TRANSACTION READONLY;  
2 BEGIN TRANSACTION;  
3     SELECT model, price FROM PC  
4     WHERE speed = speed AND  
5         ram=ram  
6 COMMIT;  
7
```

### 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:

```
1 BEGIN TRANSACTION;  
2  
3 UPDATE accounts  
4 SET balance = balance - 1000  
5 WHERE account_no = 100;  
6  
7 UPDATE accounts  
8 SET balance = balance + 1000  
9 WHERE account_no = 200;  
10  
11 INSERT INTO account_changes(account_no,flag,amount,  
    changed_at)
```

```
12     VALUES (100, '-', 1000, datetime('now'));
13
14     COMMIT;
15
16     // Example - SET TRANSACTION READONLY
17     SET TRANSACTION READONLY;
18     BEGIN TRANSACTION;
19         ...
20     COMMIT;
21
```

```
b) BEGIN TRANSACTION;
2  DELETE FROM PC
3  WHERE model=<model number>
4
5  DELETE FROM Product
6  WHERE model=<model number>
7
8  COMMIT;
9
```

```
c) BEGIN TRANSACTION;
2
3  UPDATE PC
4  SET price=price - 100
5  WHERE model=<model number>
6
7  COMMIT;
8
```

```
d) BEGIN TRANSACTION;
2
3  IF (<model> IN (
4      SELECT <model> FROM Product
5      NATURAL JOIN PC)
6
7      PRINT 'Error occurred';
8  ELSE
9      INSERT INTO PC
10     VALUES (<model>, <speed>, <ram>, <hd>, <price>)
11
12     INSERT INTO Product
13     VALUES (<maker>, <model>, <type>)
14 COMMIT;
15
```

## 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

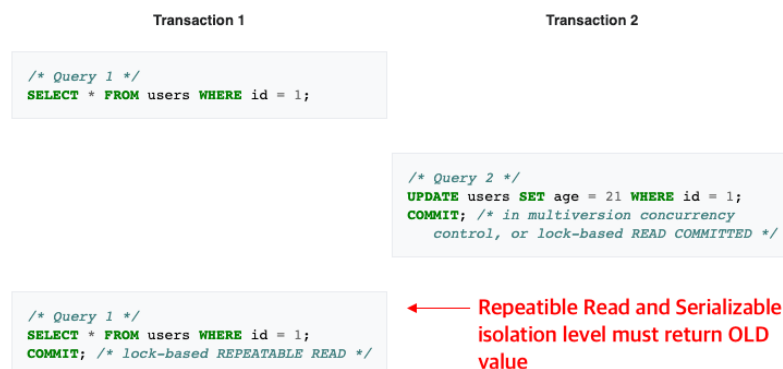
- Reading data modified by another transaction (Dirty Read)
- Repeated retrieval of rows resulting in different values (Non-repeatable read)
- Insertion/deletion of data (Phantom)

#### 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-repeatable 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 REPEATABLE READ;
  - \* Is the higher level of isolation
  - \* Guarantees everything of READ COMMITTED level
  - \* Can read unchanged data in subsequent reads
- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
  - \* Is the highest level of isolation
  - \* Guarantees everything of READ REPEATABLE 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 committed' and 'repeatable read', [link](#)

\* Wikipedia: Isolation (database systems), link

#### 4. Exercise 8.1.1:

```
a) CREATE VIEW RichExec AS
2   SELECT * FROM MovieExec
3   WHERE netWorth >= 100000000;
4
```

##### 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:

```
1 CREATE VIEW ParamountMovies AS
2   SELECT title, year
3   FROM Movies
4   WHERE studioName = 'Paramount';
5
```

```
b) CREATE VIEW StudioPres AS
2   SELECT * FROM Movies
3   INNER JOIN Studio ON cert# = presC#;
4
```

```
c) CREATE VIEW ExecutiveStar AS
2   SELECT * FROM MovieExec
3   NATURAL JOIN MovieStar;
4
```

#### 5. Exercise 8.1.2:

```
a) SELECT name, gender FROM ExecutiveStar;
2
```

```
b) SELECT name FROM RichExec WHERE netWorth > 100000000;
2
```

```

c)  SELECT name FROM StudioPres
    2  NATURAL JOIN ExecutiveStar
    3  WHERE netWorth > 50000000
    4

```

### 6. Exercise 8.2.1:

*RichExec* is updatable.

#### Notes:

- Updatable View Conditions
  - The WHERE clause in CREATE VIEW must not be a subquery
  - The FROM clause has only one occurrence 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:

```

1  Movies(title, year, length, genre, studioName, producerC#)
2  Suppose studioName is NOT NULL but has no default value.
   Then, a fix is:
3
4  CREATE VIEW Paramount AS
5      SELECT studioName, title, year
6      FROM Movies
7      WHERE studioName = 'Paramount';
8

```

### 7. Exercise 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
    2  INSTEAD OF INSERT ON DisneyComedies
    3  REFERENCING
    4      NEW ROW AS NewTuple
    5  FOR EACH ROW
    6  INSERT INTO Movies(title, year, length, genre, studioName)
    7  VALUES(NewTuple.title, NewTuple.year, NewTuple.length, 'comedy',
    8  '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:

```

1      CREATE VIEW ParamountMovies AS
2          SELECT title, year
3          FROM Movies
4          WHERE studioName = 'paramount';
5
6      CREATE TRIGGER ParamountInsert
7      INSTEAD OF INSERT ON ParamountMovies
8      REFERENCING NEW ROW AS NewRow
9      FOR EACH ROW
10         INSERT INTO Movies(title, year, studioName)
11         VALUES(NewRow.title, NewRow.year, 'Paramount');
12

```

```

c)    CREATE TRIGGER DisneyComediesInsert
2      INSTEAD OF INSERT ON DisneyComedies
3      REFERENCING
4          NEW ROW AS NewTuple
5          OLD ROW AS OldTuple
6      FOR EACH ROW
7      UPDATE Movies
8      SET length=NewTuple.length
9      WHERE title=OldTuple.title AND year=OldTuple.year;
10

```

## 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
2      INSTEAD OF INSERT ON NewPC
3      REFERENCING
4          NEW ROW AS NewTuple
5          OLD ROW AS OldTuple
6      FOR EACH ROW
7      INSERT INTO PC(model speed, ram, hd ,price)
8      VALUES (NewTuple.model, NewTuple.speed, NewTuple.ram, NewTuple.hd
, NewTuple.price);
9
10     INSERT INTO Product(maker, model, type)
11     VALUES (NewTuple.maker, NewTuple.model, 'pc');
12

```

c)

```
1 CREATE TRIGGER NewPCUpdate
2   INSTEAD OF INSERT ON NewPC
3   REFERENCING
4     NEW ROW AS NewTuple
5   FOR EACH ROW
6   UPDATE PC
7   SET model=NewTuple.model
8     speed=NewTuple.speed,
9     ram=NewTuple.ram,
10    hd=NewTuple.hd,
11    price=NewTuple.price;
12
13 UPDATE Product
14 SET maker=NewTuple.maker,
15     model=NewTuple.model,
16     type='pc';
17
```

**Correct Solution:**

```
1 CREATE TRIGGER NewPCUpdate
2   INSTEAD OF UPDATE ON NewPC
3   REFERENCING
4     NEW ROW AS NewTuple
5   FOR EACH ROW
6   UPDATE PC
7   SET model=NewTuple.model
8     speed=NewTuple.speed,
9     ram=NewTuple.ram,
10    hd=NewTuple.hd,
11    price=NewTuple.price;
12
13 UPDATE Product
14 SET maker=NewTuple.maker,
15     model=NewTuple.model,
16     type='pc';
17
```

d)

```
1 CREATE TRIGGER NewPCDelete
2   INSTEAD OF DELETE ON NewPC
3   REFERENCING
4     NEW ROW AS NewTuple
5   FOR EACH ROW
6   DELETE FROM PC
7   WHERE model=NewTuple.model;
8
9   DELETE FROM Product
10  WHERE model=NewTuple.model;
11
```



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

```
1 CREATE INDEX KeyIndex ON Movies(title, year);
2
```

- e.g. Dropping index

```
1 CREATE INDEX KeyIndex ON Movies(title, year);
2
```

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 speeds 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:

```

1  UPDATE MovieProd
2  SET name='New Name'
3  WHERE (title, year) IN
4  (
5      SELECT title, year FROM Movies
6      INNER JOIN MovieExecs
7      ON Movies.productC# = MovieExec.cert#
8      WHERE cert# = '4567'
9  );
10

```

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

```

1  UPDATE NewPC
2  SET maker='new-maker'
3      model='new-model-number'
4      speed='new-speed'
5      ram='new-ram'
6      hd='new-hd'
7      price='new-price'
8  WHERE model = 'old-model-number';
9

```

- Delete

```

1  DELETE FROM NewPC WHERE model = 'old-model-number'
2

```

**Notes:**

- Materialized view of NewPC

```

1  CREATE MATERIALIZED VIEW NewPC AS
2      SELECT maker, model, speed, ram, hd, price
3      FROM Product, PC
4      WHERE Product.model = PC.model AND type = 'pc';
5

```

**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

```

1  INSERT INTO ShipStats
2  VALUES (
3      SELECT country, AVG(displacement), COUNT(*)
4      FROM Classes, Ships
5      WHERE Classes.class = ships.class
6      GROUP BY country
7      HAVING country='name-of-country'
8  )
9

```

- Updates

```

1  DELETE FROM ShipStats WHERE country = 'name-of-country';
2  INSERT INTO ShipStats
3  VALUES (
4      SELECT country, AVG(displacement), COUNT(*)
5      FROM Classes, Ships
6      WHERE Classes.class = ships.class
7      GROUP BY country
8      HAVING country='name-of-country'
9  )
10

```

- Delete

```

1  DELETE FROM ShipStats WHERE country = 'country'
2

```

**15. Exercise 8.5.4**

Query Q #1: The title of all movies created by movie exec 'Max Bialystock'

```
1  SELECT title
2  FROM Movies, MovieExec
3  WHERE producerC# = cert# name = 'Max Bialystock'
4
```

Query Q #2: The title of all movies created in 2019

```
1  SELECT title
2  FROM Movies, MovieExec
3  WHERE producerC# = cert# year = 2019
4
```

Query Q #3: The name of all producers whose movie was created in 2019

```
1  SELECT name
2  FROM Movies, MovieExec
3  WHERE producerC# = cert# year = 2019
4
```

Query Q #4: The name of all producers whose movie was created in 2019

```
1  SELECT name
2  FROM Movies, MovieExec
3  WHERE producerC# = cert# year = 2019
4
```

Query Q #5: The name of star in movie created by producer 'John Wilks'

```
1  SELECT starName
2  FROM Movies, MovieExec
3  WHERE producerC# = cert# name = 'John Wilks'
4
```

Query Q #6: The title of movie created by producer 'John Wilks' in 2019

```
1  SELECT title,
2  FROM Movies, MovieExec
3  WHERE producerC# = cert# name = 'John Wilks' year = 2019
4
```

### 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
- Key point: use queries  $C$  and relations in FROM related to materialized view