Module: CSM040 Data Management
Coursework: July to September 2023 study session
Student: Andrew Davis 220491901

1)

a)

i) $\Pi_{\text{MEMBER\_NAME, DOB}}$ (σ (MEMBER_ID IN ($\Pi_{\text{BIDDER\_ID}}$ (σ $_{\text{(BID\_AMOUNT > RESERVE\_PRICE)}}$) (ITEM ⋈ BID)))) (MEMBER)

```
SELECT MEMBER_NAME, DOB
FROM MEMBER
WHERE MEMBER_ID IN
    (SELECT BIDDER_ID
    FROM BID, ITEM
    WHERE BID.ITEM_ID = ITEM.ITEM_ID
    AND BID_AMOUNT > RESERVE_PRICE);
```

ii) $\Pi_{\text{DESCRIPTION}}$ (σ (ITEM_ID IN ($\Pi_{\text{ITEM\_ID}}$ (σ $_{\text{(MEMBER\_TYPE = 'GOLD' ∧ MEMBER\_TYPE = 'SILVER')}}$) (MEMBER ⋈ BID)))) (ITEM)

```
SELECT DESCRIPTION
FROM ITEM
WHERE ITEM_ID IN
    (SELECT ITEM_ID
    FROM BID, MEMBER
    WHERE BIDDER_ID = MEMBER_ID
    AND MEMBER_TYPE = 'GOLD'
    AND MEMBER_TYPE = 'SILVER');
```

iii) $\Pi_{\text{MEMBER\_NAME}}$ (σ (MEMBER_ID IN ($\Pi_{\text{SELLER\_ID}}$ (σ (ITEM_ID IN ($\Pi_{\text{ITEM\_ID}}$ (σ $_{\text{(RATING = 10)}}$) (REVIEW)))) (ITEM)))) (MEMBER)

```
SELECT MEMBER_NAME
FROM MEMBER
WHERE MEMBER_ID IN
    (SELECT SELLER_ID
    FROM ITEM
    WHERE ITEM_ID IN
            (SELECT ITEM_ID
            FROM REVIEW
            WHERE RATING = 10));
```

iv) $\Pi_{\text{DESCRIPTION}}$ (σ (ITEM_ID IN ($\Pi_{\text{ITEM\_ID}}$ (σ $_{\text{(MEMBER\_TYPE = 'GOLD')}}$) (MEMBER ⋈ BID)))) (ITEM)

```
SELECT DESCRIPTION
FROM ITEM
WHERE ITEM_ID IN
    (SELECT ITEM_ID
    FROM BID, MEMBER
    WHERE BIDDER_ID = MEMBER_ID
    AND MEMBER_TYPE = 'GOLD');
```

b)

i)
```
SELECT MEMBER_NAME, AVG(r.RATING) AS “RATING”
FROM MEMBER, REVIEW r, ITEM i
WHERE r.ITEM_ID = i.ITEM_ID
AND i.SELLER_ID = MEMBER_ID
AND START_DATE BETWEEN '2021-12-31' AND '2023-01-01'
GROUP BY MEMBER_NAME
HAVING COUNT(i.SELLER_ID) >= 10;
```

ii) This one is a bit complicated for me and really don’t understand how to get the 50% of successful bids. I think this is where everything boils down to me not being use to this little information (been working with databases for a while but usually have way more columns so there isn’t ambiguity).

I understand that I need use COUNT() to add all the bids from a specific person, but then when it comes to determining the winning bid, it gets complicated (need to figure out both MAX(BID_AMOUNT) plus MIN(BID_DATE) to make sure the person was the earliest, highest bidder, but all we can go off of is the ITEM_ID in the REVIEW table)

c)
```
DELETE r
FROM REVIEW r
WHERE r.ITEM_ID IN
    (SELECT i.ITEM_ID
    FROM ITEM i
    WHERE i.START_DATE < '2019-12-31');
```

2)

a) Every Primary Key that is listed are unique and must not be null in the table. This eliminates duplicates as well as rows that may have a couple columns in common (such as prices, auction lengths, date of births, etc) but not all columns can be the same across two rows. Foreign Keys are used to unique identify those primary keys in connecting tables, thus also eliminating duplicate values and/or mistyped values. The foreign key must exist in the table it is linked to as the primary key.

Table: MEMBER
Primary Key: MEMBER_ID

Table: ITEM
Primary Key: ITEM_ID
Foreign Key: SELLER_ID
SELLER_ID links to MEMBER_ID of MEMBER

Table: BID
Primary Key: BID_ID
Foreign Keys: ITEM_ID, BIDDER_ID
ITEM_ID links to ITEM_ID of ITEM
BIDDER_ID links to MEMBER_ID of MEMBER

Table: REVIEW
Primary/Foreign Key: ITEM_ID
ITEM_ID links to ITEM_ID of BID
Explanation: I am just unsure it if is best to reference the ITEM_ID from the BID table or the ITEM table. I think referencing the ITEM table is best since that is the original Primary Key (ITEM_ID from BID will also be a Foreign Key), but there would be no review without a bid, so I went with the BID reference. Since the REVIEW table is a one-to-one relationship with the selected table, both the primary key and the foreign key for this table are the same.

b)

i) Having a Foreign Key assigned to ITEM_ID in the REVIEW table will solve this issue. see explanation from part i about the REVIEW table to see why BID was the chosen table.

For the REVIEW table:

FOREIGN KEY (ITEM_ID) REFERENCES BID(ITEM_ID)

ii) This is the same issue as part i. Foreign Keys are used so that there is always a reference to a Primary Key, and thus the identifier must exist

For the BID table:

FOREIGN KEY (ITEM_ID) REFERENCES ITEM(ITEM_ID)

FOREIGN KEY (BIDDER_ID) REFERENCES MEMBER(MEMBER_ID)

iii) This one is throwing me off a bit only because I am uncertain how the MEMBER_ID is being recorded. Since it is a unique string (aka not auto populated), I believe that those have to be put in manually for each member. Thus, the MEMBER_ID field has to be UNIQUE. But because this field is

entered manually, there could be two MEMBER_IDs for the same person but with different MEMBER_TYPE statuses. Hence making this question a little open ended. With having so few columns, I do not want to limit if another name exists but with a different birthday or another birthday with a different name. In the end, you would have to have a CHECK on each field that makes sure the row doesn't already exist (which I do not think is possible). So I think in the end I am just going with a UNIQUE MEMBER_ID.

For the MEMBER table:

```
UNIQUE (MEMBER_ID)
```

iv) Making the MEMBER_TYPE column to be NOT NULL will resolve the issue of having this field not being recorded. This one has to be done when designated column names (not having a constraint after the fact).

```
CREATE TABLE MEMBER (
MEMBER_TYPE VARCHAR(50)  NOT NULL,
…)
```

c) Since there are multiple tables that have the same column labeled ITEM_ID, having an index for some or all of those tables could be useful so that confusion is cut out or there is no need to use table names in the statements (aka BID.ITEM_ID, ITEM.ITEM_ID).

```
CREATE INDEX bid_item_idx
ON BID (ITEM_ID)
```

I am not quite sure any other index may be useful (probably because I have some of the queries wrong). Maybe having one on DESCRIPTION and ITEM_ID of the ITEM table would work since we look for that in two different questions.

```
CREATE INDEX item_desc_idx
ON ITEM (ITEM_ID, DESCRIPTION)
```

3)

a)

i) Due to the conflicts with R2, R3, and R4, there will have to be some conflict resolution that needs to happen. R1 has no WRITE conflicts, thus all of the READs are non-conflict. R5 only happens on TC, thus does not have conflicts. Because the locks on TB are required for TA and TC to complete, but the COMMIT does not happen until last, this schedule is not possible.

This is the revised schedule:

	TA		TB		TC	
(1)	READ R1	S lock obtained - record read				
(2)			READ R2	S lock obtained - record read		
(3)					READ R3	S lock obtained - record read
(4)			WRITE R2	X lock obtained - record written		
(5)	READ R3	S lock obtained - record read				
(6)	READ R2	S lock requested - wait				
(7)			READ R1	S lock obtained - record read		
(8)			READ R4	S lock obtained - record read		
(9)					WRITE R3	X lock requested - wait
(10)			COMMIT	locks released		
(11)		S lock obtained - record read				
	COMMIT	locks released				
(12)						X lock obtained - record written
					READ R2	S lock obtained - record read
(13)					READ R4	S lock obtained - record read
(14)					READ R5	S lock obtained - record read
(15)					WRITE R4	X lock obtained - record written
(16)					WRITE R5	X lock obtained - record written
(17)					COMMIT	locks released

ii) TA's attempt at reading R2 causes a late read, and thus creating a rollback for TA. There are no other conflicts.

This is the revised schedule (with notes):

	TA	TB	TC	R1, R, W	R2, R, W	R3, R, W	R4, R, W	R5, R, W
(0)	t1	t2	t3	t0, t0	t0, t0	t0, t0	t0, t0	t0, t0
(1)	READ R1			t1, t0				
(2)		READ R2			t2, t0			
(3)			READ R3			t3, t0		
(4)		WRITE R2			t2, t2			
(5)	READ R3					t3, t0		
(6)	READ R2				late read			
(7)			WRITE R3			t3, t3		
(8)			READ R2		t3, t2			
(9)		READ R1		t2, t0				
(10)		READ R4					t2, t0	
(11)			READ R4				t3, t0	
(12)			READ R5					t3, t0
(13)			WRITE R4				t3, t3	
(14)	ROLLBACK							
(15)			WRITE R5					t3, t3
(16)			COMMIT					
(17)		COMMIT						

iii) The table below has notes that describes what is happening on each line, but this is the quick brief:
(1) TA is read-only and has a timestamp of t1, thus all READs are of the original version (v1)
(2) TB is read-write and has a timestamp of t2, thus all READs are of the original version (v1) and the WRITE creates a new version (v2)
(3) TC is read-write and has a timestamp of t3. The READ of R2 is of v1 although TB has a WRITE beforehand, the lock is not released since the COMMIT has not happened.

This is the revised schedule (with notes):

	TA	TB	TC	R1	R2	R3	R4	R5
(0)	read-only	read-write	read-write	version	version	version	version	version
	t1	t2	t3	v1	v1	v1	v1	v1
(1)	READ R1			reads v1				

(2)		READ R2			reads v1			
(3)			READ R3			reads v1		
(4)		WRITE R2			write v2			
(5)	READ R3					reads v1		
(6)	READ R2				reads v1			
(7)			WRITE R3			writes v2		
(8)			READ R2		reads v1 (t2 commit hasn't happened so X lock is not released)			
(9)		READ R1		reads v1				
(10)		READ R4					reads v1	
(11)			READ R4				reads v1	
(12)			READ R5					reads v1
(13)			WRITE R4				writes v2	
(14)	COMMIT							
(15)			WRITE R5					writes v2
(16)			COMMIT					
(17)		COMMIT						

b) I did not read this question ahead of doing part a.iii, but I have already included this in my answer above.

c) For part i: the serialized schedule would be TB, TA, TC. This is due to the lock release schedule in which TB needs to finish first for TA to read R2, then TA needs to finish so that TC can write R3.
For part ii: the serialized schedule would be TB, TC, TA. Because of the rollback that happens on TA, that shows that TA eventually because a new timestamped transaction after TB and TC. TB is first because of the original timestamp order given to each transaction.
For part iii: the serialized schedule would be TA, TC, TB. TA is read-only and thus reads all of the original version of each record (and can therefore be first in the serial order). TC is second because the write on R2 is not committed, so all of the reads on TC are also the original versions until TC writes a new update. Because there is no write conflict of TC onto TB, then TB is ok to finish last (all reads are also original versions and the write creates a new one).

d) The original schedule is conflict serializable based on the write-read conflict of TB to TA and TB to TC on R2, read-write conflict of TA to TC on R3, and the read-write conflict of TB to TC on R4.

   This is the revised schedule:

	TA	TB	TC
(1)		READ R2	
(2)		WRITE R2	
(3)		READ R1	
(4)		READ R4	
(5)		COMMIT	
(6)	READ R1		
(7)	READ R3		
(8)	READ R2		
(9)	COMMIT		
(10)			READ R3
(11)			WRITE R3
(12)			READ R2
(13)			READ R4
(14)			READ R5
(15)			WRITE R4
(16)			WRITE R5
(17)			COMMIT

e) The fixed schedule that I have provided for the locking schedule (part a.i) is two-phase. This is because the first phase locks at certain points, and the second phase is when the COMMITs happen to release the locks for the other transactions to exit the wait state. If there were no WRITEs within the transactions, then it would only be one-phase (no conflicts arise with only READs and thus COMMITs can happen at any point).

4)

i)
```
CREATE VIEW MOD_ENROL_VIEWER (YEAR, MOD_CODE, STUDENT)
AS
SELECT YEAR, MOD_CODE, STUDENT
FROM MOD_ENROL
WHERE MOD_CODE IN
    (SELECT DISTINCT MOD_CODE
    FROM MOD_ENROL
    WHERE SUBSTRING_INDEX(USER(),'@',1) = STUDENT)
GROUP BY YEAR, MOD_CODE, STUDENT
UNION
```

```
SELECT YEAR, MOD_CODE, STUDENT
FROM MOD_ENROL
WHERE SUBSTRING_INDEX(USER(),'@',1) IN
    (SELECT LEADER
    FROM MODULE)
GROUP BY YEAR, MOD_CODE, STUDENT;
```

ii)
```
CREATE VIEW MOD_QUEST_RESPONSE (YEAR, MOD_CODE, Q_NO,
Q_TEXT, SCORE) AS
SELECT r.YEAR, r.MOD_CODE, q.Q_NO, q.Q_TEXT, r.SCORE
FROM MOD_QUESTIONNAIRE q
JOIN MOD_RESPONSE r
ON q.Q_NO = r.Q_NO
WHERE r.MOD_CODE IN
    (SELECT MOD_CODE
    FROM MODULE
    WHERE SUBSTRING_INDEX(USER(),'@',1) = LEADER)
UNION
SELECT r.YEAR, r.MOD_CODE, q.Q_NO, q.Q_TEXT, r.SCORE
FROM MOD_QUESTIONNAIRE q
JOIN MOD_RESPONSE r
ON q.Q_NO = r.Q_NO
WHERE r.STUDENT = SUBSTRING_INDEX(USER(),'@',1);
```

iii)
```
CREATE VIEW MOD_QUEST_SUMMARY(YEAR, MOD_CODE, Q_NO,
Q_TEXT, MIN_SCORE, MAX_SCORE, AVG_SCORE) AS
SELECT q.YEAR, r.MOD_CODE, q.Q_NO, q.Q_TEXT, MIN(r.SCORE),
            MAX(r.SCORE), ROUND(AVG(r.SCORE), 1)
FROM MOD_RESPONSE r
JOIN MOD_QUESTIONNAIRE q
ON r.Q_NO = q.Q_NO
WHERE r.MOD_CODE =
    (SELECT DISTINCT MOD_CODE
    FROM MODULE
    WHERE SUBSTRING_INDEX(USER(),'@',1) = LEADER)
GROUP BY q.YEAR, r.MOD_CODE, q.Q_NO, q.Q_TEXT;
```

```
CREATE VIEW MOD_OVERALL_SUMMARY(YEAR, MOD_CODE,
AVG_SCORE) AS
SELECT YEAR, MOD_CODE, ROUND(AVG(SCORE), 1)
FROM MOD_RESPONSE
GROUP BY YEAR, MOD_CODE;
```