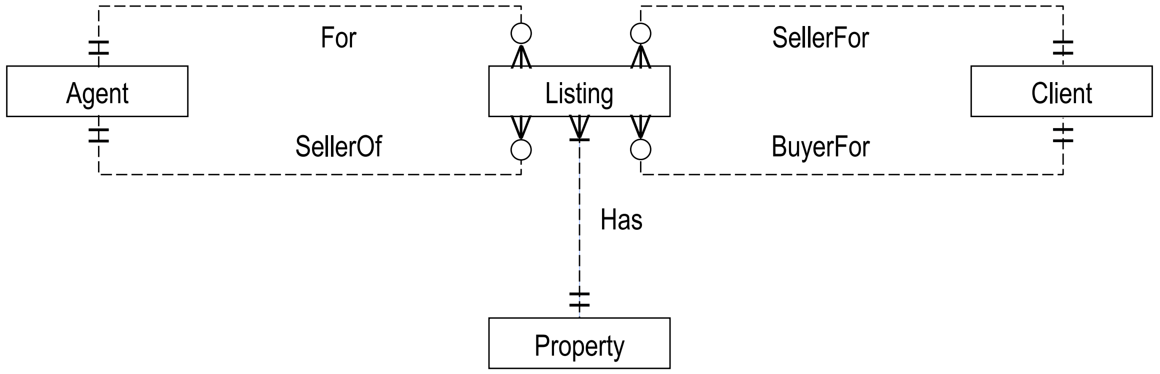
MSBD 6000L: Database Systems

Final Exam — Sample Solutions

1.



1 mark for each named relationship (0.5 if unnamed)
0.25 marks each cardinality constraint
0.25 marks each participation constraint
Deduct
0.25 marks each unnecessary/incorrect constraint
0.5 marks each unnecessary relationship
1 mark for adding XOR constraint for Listing or Client

Why is there not an exclusive OR (**XOR**) constraint between the SellerFor and BuyerFor relationships?

a) Suppose we put the **XOR** constraint on Client (i.e., a Client instance can only have a SellerFor or a BuyerFor relationship). In this case, it is not possible for a client to be both a seller and a buyer, contradicting the problem description.

b) Suppose we put the **XOR** constraint on Listing (i.e., a Listing instance can only have a SellerFor or a BuyerFor relationship). In this case, it is not possible for a listing to be related to a client by both a SellerFor and a BuyerFor relationship, again contradicting the problem description.

2. Athlete(<u>id</u>, name, birthdate, country)

Event(<u>eventName</u>, date, time)

Trial(<u>*id, eventName*</u>, score)
  id references Athlete(id) on delete cascade
  eventName references Event(eventName) on delete cascade

CompetesIn(<u>*id, eventName*</u>)
  id references Athlete(id) on delete cascade
  eventName references Event(eventName) on delete cascade

1 mark for each foreign key attribute
1 mark for each new key

For each referential integrity constraint
  0.5 marks for referencing correct relation
  0.5 marks for correct delete action

3. a) i. The file is a heap file (not ordered), so the index must be dense.
Therefore, the leaf level needs to point to 3,200,000 records. The fanout of all B^+-tree nodes is 20.
Height of the B^+-tree: $\lceil \log20\ 3,200,000 \rceil = 5$ [1 mark]

ii. Number of 1st level (leaf) nodes (pages): $\lceil 3,200,000/20 \rceil = 160,000$ [1 mark]
Number of 2nd level nodes (pages): $\lceil 160,000/20 \rceil = 8,000$ [1 mark]
Number of 3rd level nodes (pages): $\lceil 8,000/20 \rceil = 400$ [1 mark]
Number of 4th level nodes (pages): $\lceil 400/20 \rceil = 20$ [1 mark]
Number of 5th level (root) pages: $\lceil 20/20 \rceil = 1$ [1 mark]

b) Page I/Os needed to add the record to the heap file: 1 read + 1 write = <u>2</u> [1 mark]

Page I/Os needed to search B^+-tree to determine where to add the new value: <u>3</u> reads (1 read for each level of the B^+-tree below the root; the root is already in the buffer) [1 mark]

Since all nodes except the root have maximum occupancy, nodes below the root on the path from the root to the leaves must be split to insert the new value and then need to be written to secondary storage. Moreover, even though the root node is always kept in the buffer, it needs to be written to secondary storage when the B^+-tree is updated to ensure the integrity of the B^+-tree in case of system failure.

Page I/Os needed to update B^+-tree: <u>7</u> writes (3 original nodes + 3 new split nodes + 1 root node) [2 marks]

Total Page I/Os needed: 2 + 3 + 7 = <u>12</u>

4. a) i. Fanclub X MemberOf → Result A (Cartesian product operation)

Size of one Result A tuple <u>in bytes</u>: 210+35 = <u>245</u> [0.5 marks]

Total <u>number of tuples</u> in Result A: 100*21,000 = <u>2,100,000</u> [0.5marks]

Read page I/O cost: 25+750 =<u>775</u> [1 mark]
(Fanclub fits in the buffer)

Write page I/O cost: $\lceil 2{,}100{,}000/\lfloor 1000/245 \rfloor \rceil$ = <u>525,000</u> [1 mark]

ii. Result A X Clubmember → Result B (Cartesian product operation)

Size of one Result B tuple <u>in bytes</u>: 245+125 = <u>370</u> [0.5 marks]

Total number of Result B tuples: 2,100,000*10,000 = <u>21,000,000,000</u> [0.5 marks]

Read page I/O cost: $\lceil 1250/50-2 \rceil$*525,000+1250 = <u>14,176,250</u> [1 mark]

Write page I/O cost: $\lceil 21{,}000{,}000{,}000/\lfloor 1000/370 \rfloor \rceil$ = <u>10,500,000,000</u> [1 mark]

iii. $\sigma_{\text{F.clubId=M.clubId} \wedge \text{M.username=C.username} \wedge \text{clubName='Fab 4'} \wedge \text{gender='M'} \wedge (\text{educationLevel='tertiary'} \vee \text{educationLevel='post tertiary'})}$Result B

Total number of tuples in the query result: <u>42</u> [1 mark]

<u>Explanation</u>

Since there are 10,000*2.1=21,000 MemberOf tuples, on average each fan club has 21,000/100=210 club members. Consequently, on average, fan club 'Fab 4' has 210 members (since club names are unique). [1 mark]

Since 50% of club members are male and (3000+1000)/10000 (i.e., 40%) of club members have education level of either tertiary or post tertiary, therefore, on average, the query will return .5*.4*210=42 tuples. [2 marks]

b) **<u>Step 1</u>:** <u>Index lookup</u> to evaluate $\pi_{\text{clubId}}(\sigma_{\text{clubName='Fab 4'}}\text{Fanclub})$ ⟹ Step 1 Result [0.5 marks]

Since Fanclub has a 2-level B$^+$-tree index on clubName, 2 page I/Os are required to search the B$^+$-tree and 1 page I/O to retrieve the Fanclub record. Since club names are unique, only one record will be retrieved. [1 mark]

Only clubId, 5 bytes, is kept in the buffer. [0.5 marks]

Average page I/O cost: <u>3</u> [1 mark] **Number of result tuples:** <u>1</u> [1 mark]

<u>Step 2</u>: **<u>Join using indexed nested-loop</u>** to evaluate π_{username}(Step 1 Result ⋈ MemberOf) ⟹ Step 2 Result [0.5 marks]

The clubId value from Step 1 is used to search the 5-level B$^+$-tree index on MemberOf to find the first page with the given club id requiring 5 page I/Os. The username part of the search key is ignored. Since there is only one club id from Step 1 to match with MemberOf, a total of 5 page I/Os are needed to search the B$^+$-tree. [1 mark]

From a) we know that each Fanclub tuple is related to, on average, 21000/100=210 MemberOf tuples and $\lfloor 1000/35 \rfloor$=28 MemberOf tuples fit on a page. Thus, since the file is ordered on (clubId, username), in the best case, all the MemberOf tuples will be on $\lceil 210/28 \rceil$=8 pages requiring 8 page I/Os, while in the worst case they may span 9 pages requiring 9 page I/Os. It is not possible that they span more than 9 pages. [2 marks]

Only username, 10 bytes, is kept for each tuple. The 210 usernames occupy 10*210=2,100 bytes or $\lceil 2{,}100/1000 \rceil$=3 pages which are kept in the buffer. [0.5 marks]

Average page I/O cost: <u>13 or 14</u> [1 mark] **Number of result tuples:** <u>210</u> [1 mark]

<u>Step 3</u>: **<u>On-the-fly</u>** to evaluate $\pi_{\text{username, firstName, lastName, email}}(\sigma_{\text{gender='M'} \wedge (\text{educationLevel='tertiary'} \vee \text{educationLevel='post tertiary'})}\text{Clubmember})$ ⟹ Step 3 Result [0.5 marks]

Since neither gender nor educationLevel are indexed, the evaluation of the conditions gender='M' ∧ (educationLevel='tertiary' ∨ educationLevel='post tertiary') are deferred to do with the join in Step 4. [0.5 marks]

Average page I/O cost: <u>0</u> **Number of result tuples:** <u>0</u>

<u>Step 4</u>: **<u>Hash file search</u>** to evaluate $\pi_{\text{firstName, lastName, email}}$(Step 2 Result ⋈ Step 3 Result) ≡ $\pi_{\text{firstName, lastName, email}}(\text{Step 2 Result} \bowtie \sigma_{\text{gender='M'} \wedge (\text{educationLevel='tertiary'} \vee \text{educationLevel='post tertiary'})}\text{Clubmember})$ ⟹ Query Result [0.5 marks]

<u>Note</u>: There is no hash index on username for the Clubmember relation!

Each username from the Step 2 Result is hashed to find the corresponding Clubmember tuple which requires 1 page I/O for each username since Clubmember is a hash file. Since there are 210 usernames, a total of 210 page I/Os are required. [1 mark]

As stated in Step 3, the conditions gender='M' ∧ (educationLevel='tertiary' ∨ educationLevel='post tertiary') are checked in each retrieved tuple in the buffer. [0.5 marks]

Average page I/O cost: <u>210</u> [1 mark] **Number of result tuples:** <u>42</u> [mark]

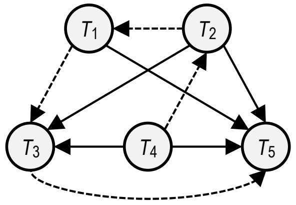
Total average page I/O cost: 3 + (13 or 14) + 210 = <u>226</u> or <u>227</u>

5. a) i. Is the schedule <u>serializable</u>? Yes

The schedule is <u>serializable</u>.

The equivalent serial schedule is $T_4 \longrightarrow T_2 \longrightarrow T_1 \longrightarrow T_3 \longrightarrow T_5$ (shown as dashed arrows),which is the only possible one.

Precedence Graph



ii. Is the schedule <u>recoverable</u>? Yes

T_4 does not read any data item written by another transaction

T_2 reads A written by $T_4 \Rightarrow T_4$ must commit before T_2

T_3 reads A written by $T_2 \Rightarrow T_2$ must commit before T_3

T_5 reads A written by $T_3 \Rightarrow T_3$ must commit before T_5

T_3 reads B written by $T_1 \Rightarrow T_1$ must commit before T_3

T_5 reads B written by $T_3 \Rightarrow T_3$ must commit before T_5

T_1 reads C written by $T_4 \Rightarrow T_4$ must commit before T_1

Commit order: $T_4 \longrightarrow T_1 \longrightarrow T_2 \longrightarrow T_3 \longrightarrow T_5$

or

$T_4 \longrightarrow T_2 \longrightarrow T_1 \longrightarrow T_3 \longrightarrow T_5$

<u>Serializability</u>
0.5 marks for each correct precedence graph arc
2.5 marks for correct serial schedule
(All transactions must be in the correct order to get 2.5 marks.)

<u>Recoverability</u>
0.5 marks for each correct commit order

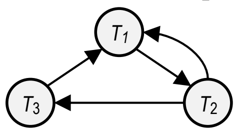
b) Is the schedule serializable according to the given timestamp ordering?
No, it fails at write(A) of T_1.

T_1 [TS=1]	T_2 [TS=2]	T_3 [TS=3]
T_1 [TS=1]	T_2 [TS=2]	T_3 [TS=3]
		read(C) RTS(C)=3; WTS(C)=0
		read(A) RTS(A)=3; WTS(A)=0
read(B) RTS(B)=1; WTS(B)=0		
	read(C) RTS(C)=3; WTS(C)=0	
	read(A) RTS(A)=3; WTS(A)=0	
		write(C) RTS(C)=3; WTS(C)=3
	write(B) RTS(B)=1; WTS(B)=2	
read(A) RTS(A)=3; WTS(A)=0		
write(B) RTS(B)=1; WTS(B)=2	TS(T_1)=1 < WTS(B)=2 ⟹ **ignore**	
write(A)	TS(T_1)=1 < RTS(A)=3 ⟹ **rollback** (since TS(T_1)<RTS(A))	

Is there an equivalent serial schedule? No.

There is a cycle in the precedence graph.

Precedence Graph



0.2 marks for each correct timestamp
1 mark for correct rollback location
0.8 marks for no equivalent serial schedule

6. a) All the write operations after the <checkpoint {}> record, namely,

<T_2 write, B, 8, 12>	[.5 marks]
<T_4 write, B, 12, 15>	[.5 marks]
<T_3 write, A, 5, 30>	[.5 marks]
<T_4 write, A, 30, 20>	[.5 marks]

b) None [1 mark]

c) Yes [1 mark]

Even though T_4 appears in the redo list, it reads a value written by both T_2 (B) and T_3 (A). Since T_2 and T_3 are rolled back, T_4 also needs to be rolled back (cascading rollback) and *resubmitted* rather than redone since the values of A and B that it read that were written by T_2 and T_3 are no longer valid since T_2 and T_3 are rolled back, and their writes are undone. Note that the commit of T_4 is a partial commit since as stated in the question its buffer pages have not been written to disk (i.e., no buffer log pages written to disk between the checkpoint and system failure).

[With Yes answer: 1 mark for correct explanation; 0.5 marks for partially correct explanation]

7. a)

```
create table Rents(
   .
   .
   .
   foreign key (memberId) references Member(memberId) on delete cascade,
   foreign key (movieId) references Movie(movieId) on delete cascade);
```

$\boxed{\text{1 mark}}$ $\boxed{\text{1 mark}}$

```
create table Prefers(
   .
   .
   .
   foreign key (memberId) references Member(memberId) on delete cascade);
```

$\boxed{\text{1 mark}}$

Note: There cannot be a referential integrity constraint from Movie(genre) to Prefers(genre) since genre is not the primary key of Prefers.

$\boxed{\text{2 marks}}$

Table	Possible create order				
Member	1	1	1	2	2
Movie	2	2	3	1	1
Prefers	3	4	2	3	4
Rents	4	3	4	4	3

b) $\pi_{\text{memberId}}(\underbrace{\pi_{\text{memberId, movieId}}(\text{Rents} \bowtie \text{Prefers})}_{\text{movies whose genres members rented}} - \underbrace{\pi_{\text{memberId, movieId}}(\text{Prefers} \bowtie \text{Movie})}_{\text{movies whose genres members prefer}})$ $\boxed{\text{5 marks}}$

Note: The first join eliminates members from the result who have not indicated any preferences.

c) Find the names of the members who have rented a movie but have not indicated any genre preferences. $\boxed{\text{4 marks}}$

d) $\boxed{\text{6 marks}}$

```
select name, count(distinct genre)
from member, Rents left outer join Prefers on Rents.memberId=Prefers.memberId
where member.memberId=Rents.memberId
group by member.memberId, name
order by count(distinct genre) desc, name;
```

Note: An outer join is required to include those members who have rented a movie but have not specified any preferences.

e) $\boxed{\text{5 marks}}$

```
select memberId, name, count(*)
from Member natural join Rents
group by memberId, name
having count(*)=(select max(count(*))
                 from Rents
                 group by memberId);
```

Note: Other equivalent queries can also get full marks.