1.

a) This query does not correspond to a range query on the search key as the condition on the first attribute if the search key is a comparison condition. It looks up records which have the value of A between 10 and 50. However, each record is likely to be in a different block, because of the ordering of records in the file, leading to many I/O operations. In the worst case, for each record, it needs to traverse the whole tree (cost is h), so the total cost is n1*h.

b) This query can be answered by using an ordered index on the search key (A,B). For each value of A this is between 10 and 50, the system located records with B value between 5 and 10. However, each record could be in a different disk block. This amounts to executing the query based on the condition on A, this costs n1*h. Then these records are checked to see if the condition on B is satisfied. So, the total cost in the worst case is n1*h

c) n1 records satisfy the first condition and n2 records satisfy the second condition. When both the conditions are queried, n1 records are output in the first stage. So, in the case where n1=n2, no extra records are output in the first stage. Otherwise, the records which don't satisfy the second condition are also output with an additional cost of ***h*** each (worst case).

2.

a) Since the index is a primary dense index, there are as many data entries in the B+tree as records in the heap file. An index page consists of at most 2d keys and 2d+1 pointers. So we have to maximized under the condition that $2d \cdot 40+(2d+1) \cdot 10 \leq 1000$. The solution isd= 9, which means that we can have 18 keys and 19 pointerson an index page. A record on a leaf page consists of the key field and a pointer.Its size is 40+10=50 bytes. Therefore a leaf page has space for (1000/50)=20 data entries. The resulting tree has $\log_{19}(20000/20) + 1 = 4$ levels

b) Since the nodes at each level are filled as much as possible, there are 20000/20=1000 leaf nodes (on level 4). (A full index node has 2d+1 = 19 children.) Therefore There are 1000/19= 53 index pages on level 3,53/19= 3 index pages on level2, and there is one index page on level 1 (the root of the tree)

c) Here the solution is similar to part 1, except the key is of size 10 instead of size40. An index page consists of at most 2d keys and 2d+ 1 pointers. So we have to maximized under the condition that $2d \cdot 10 + (2d+ 1) \cdot 10 \leq 1000$. The solution is d= 24, which means that we can have 48 keys and 49 pointers on an index page. A record on a leaf page consists of the key field and a pointer. Its size is10+10=20 bytes. Therefore a leaf page has space for (1000/20)=50 data entries.The resulting tree haslog49(20000/50) + 1= 3 levels.

d) Since each page should be filled only 70 percent, this means that the usable size of a page is $1000 \cdot 0.70 = 700$ bytes. Now the calculation is the same as in part 1 but using pages of size 700 instead of size 1000. An index page consists of at most 2 keys and 2d+ 1 pointers. So we have to maximized under the condition that $2d \cdot 40+(2d+1) \cdot 10 \leq 700$. The solution isd= 6, which means that we can have 12 keys and 13 pointers on an index page. A record on a leaf page consists of the key field and a pointer. Its size is 40+10=50 bytes. Therefore a leaf page has space for (700/50)=14 data entries. The resulting tree haslog13(20000/14) + 1= 4 levels

3)

r1: n_{r1}=20000 tuples needs b_{r1} =800 blocks, and

r2: n_{r2}=45000 tuples needs b_{r2} =1500 blocks.

a. Nested-loop join:

Using r1 as the outer relation we need 20000 ∗ 1500 + 800 = 30, 000, 800 disk accesses,

if r2 is the outer relation we need 45000 ∗800 + 1500 = 36, 001, 500 disk accesses.

Total number of block transfers in worst case = $n_{r1} * b_{r2} + b_{r1}$

Total number of seeks required in worst case = $n_{r1} + b_{r1}$

Total number of block transfers in best case = $b_{r1} + b_{r2}$

Total number of seeks required in best case= $2(n_{r1} + b_{r1})$

b. Block nested-loop join:

If r1 is the outer relation, we need 800 ∗1500+800 disk accesses,

if r2 is the outer relation we need 1500 ∗800+1500 disk accesses.

Total number of block transfers in worst case = $b_{r1} * b_{r2} + b_{r1}$

Total number of seeks required = $2 * b_{r1}$

Total number of block transfers in best case = $b_{r1} + b_{r2}$

Total number of seeks required = $2(n_{r1} + b_{r1})$

4)

- The relation resulting from the join of r_1, r_2, and r_3 will be the same no matter which way we join them, due to the associative and commutative properties of joins. So we will consider the size based on the strategy of $((r_1 \bowtie r_2) \bowtie r_3)$. Joining r_1 with r_2 will yield a relation of at most 1000 tuples, since C is a key for r_2. Likewise, joining that result with r_3 will yield a relation of at most 1000 tuples because E is a key for r_3. Therefore the final relation will have at most 1000 tuples.
- An efficient strategy for computing this join would be to create an index on attribute C for relation r_2 and on E for r_3. Then for each tuple in r_1, we do the following:
  a. Use the index for r_2 to look up at most one tuple which matches the C value of r_1.
  b. Use the created index on E to look up in r_3 at most one tuple which matches the unique value for E in r_2.

5)

a)

$E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.
Let us rename $(E_1 \bowtie_\theta (E_2 - E_3))$ as R_1, $(E_1 \bowtie_\theta E_2)$ as R_2 and $(E_1 \bowtie_\theta E_3)$ as R_3. It is clear that if a tuple t belongs to R_1, it will also belong to R_2. If a tuple t belongs to R_3, $t[E_3$'s attributes] will belong to E_3, hence t cannot belong to R_1. From these two we can say that

$$\forall t,\ t \in R_1 \Rightarrow t \in (R_2 - R_3)$$

It is clear that if a tuple t belongs to $R_2 - R_3$, then $t[R_2$'s attributes$] \in E_2$ and $t[R_2$'s attributes$] \notin E_3$. Therefore:

$$\forall t,\ t \in (R_2 - R_3) \Rightarrow t \in R_1$$

The above two equations imply the given equivalence.
This equivalence is helpful because evaluation of the right hand side join will produce many tuples which will finally be removed from the result. The left hand side expression can be evaluated more efficiently.

b)

$\sigma_\theta(E_1 \rtimes E_2) = \sigma_\theta(E_1) \rtimes E_2$ where θ uses only attributes from E_1.
θ uses only attributes from E_1. Therefore if any tuple t in the output of $(E_1 \rtimes E_2)$ is filtered out by the selection of the left hand side, all the tuples in E_1 whose value is equal to $t[E_1]$ are filtered out by the selection of the right hand side. Therefore:

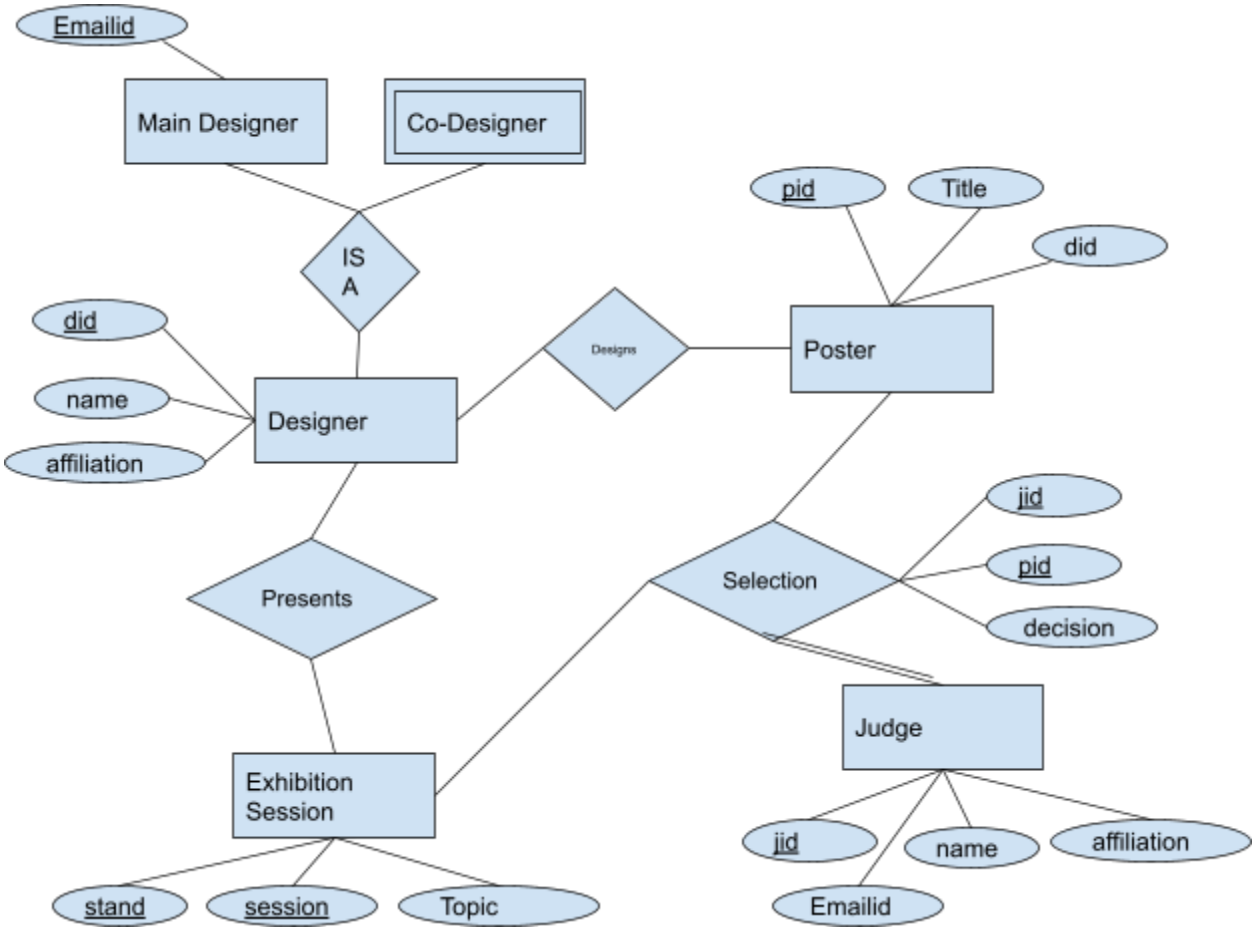
$$\forall t,\ t \notin \sigma_\theta(E_1 \rtimes E_2) \Rightarrow t \notin \sigma_\theta(E_1) \rtimes E_2$$

Using similar reasoning, we can also conclude that

$$\forall t,\ t \notin \sigma_\theta(E_1) \rtimes E_2 \Rightarrow t \notin \sigma_\theta(E_1 \rtimes E_2)$$

The above two equations imply the given equivalence.
This equivalence is helpful because evaluation of the right hand side avoids producing many output tuples which are anyway going to be removed from the result. Thus the right hand side expression can be evaluated more efficiently than the left hand side expression.

6)



7)
The following SQL statement creates
Table A:
CREATE TABLE A (a1 CHAR(10),
a2 CHAR(10),
PRIMARY KEY (a1))

Tables B and C are similar to A.

CREATE TABLE R (a1 CHAR(10),
b1 CHAR(10) NOT NULL ,
c1 CHAR(10) ,
PRIMARY KEY (a1),
UNIQUE (b1)
FOREIGN KEY (a1) REFERENCES A,
FOREIGN KEY (b1) REFERENCES B)

FOREIGN KEY (c1) REFERENCES C)
We cannot capture the total participation constraint of A in R. This is because we cannot ensure that every key a1 appears in R without the use of checks.

8)

Given two relations $R1$ and $R2$, where $R1$ contains $N1$ tuples and $R2$ contains $N2$ tuples, and $N2 > N1 > 0$, give the maximum and minimum possible sizes (in tuples) for the result relation produced by each of the following relational algebra expressions. In each case, state any assumptions about the schemas for $R1$ and $R2$ that are needed to make the expression meaningful.

(a) $R1 \cup R2$
Assumption: $R1$ and $R2$ are union compatible
Min: $N2$
Max: $N1 + N2$

(b) $R1 \cap R2$
Assumption: $R1$ and $R2$ are union compatible
Min: 0
Max: $N1$

(c) $R1 - R2$
Assumption: $R1$ and $R2$ are union compatible
Min: 0
Max: $N1$

(d) $R1 \times R2$
Min: $N1 * N2$
Max: $N1 * N2$

(e) $\sigma_{a=5}(R1)$
Assumption: $R1$ has an attribute named a
Min: 0
Max: $N1$

(f) $\pi_a(R1)$
Assumption: $R1$ has an attribute named a
Min: 1
Max: N1

9) a.

CREATE TABLE Emp (eid INTEGER, ename CHAR(10), age INTEGER , salary REAL, PRIMARY KEY (eid), CHECK (salary >= 10000))

OR

Alter table Emp add CHECK (salary >= 10000)

OR

ALTER TABLE Emp
ADD CONSTRAINT CHK_Salary CHECK (salary >= 10000);

B.

CREATE TABLE Dept (did INTEGER, buget REAL, managerid INTEGER , PRIMARY KEY (did), FOREIGN KEY (managerid) REFERENCES Emp, CHECK ((SELECT E.age FROM Emp E, Dept D) WHERE E.eid = D.managerid) > 30)

OR

Alter table Dept add CHECK (SELECT E.age FROM Emp E, Dept D WHERE E.eid = D.managerid) > 30)

OR

ALTER TABLE Emp ADD CONSTRAINT CHK_manager_age CHECK ((SELECT E.age FROM Emp E, Dept D) WHERE E.eid = D.managerid) > 30);

c.

DELETE FROM Emp E WHERE E.eid IN (SELECT W.eid FROM Work W, Emp E2, Dept D WHERE W.did = D.did AND D.managerid = E2.eid AND E.salary > E2.salary)

10.

a) f ={ AB → C,

A → D,

B → F,

F → GH,

D → IJ }

To find the key, check the RHS of the given functional dependencies, and find the closure of the attributes not present at the RHS.

Here,ABE are not present in the RHS,so

(ABE)+ = ABEC (since AB → C)

= ABECD(since A → D)

=ABECDF(since B → F)

=ABECDFGH(since F → GH)

=ABECDFGHIJ(since D → IJ)

=ABCDEFGHI

Hence from this we get all the attributes (holds attribute preservation property),so **AB** is the candidate key of the relation R(ABCDEFGHIJ)

S~o the non prime attributes are (C,D,F,G,H,I,J)

Prime attributes being (A,B,E).

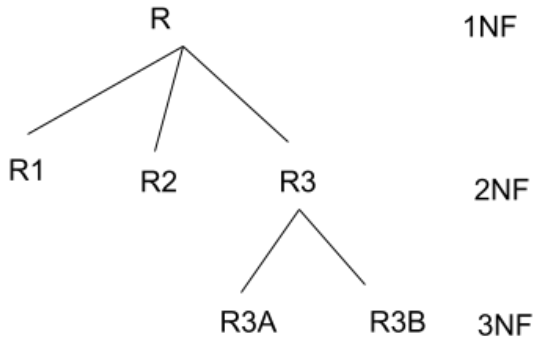
For Relation to be in 2NF, there should be no partial dependency AB → C,A → D, and B → F violates the constraints for 2NF hence we will decompose R into 3 relations,which are:

R1, R2, R3 containing R1(A,B,C), R2(A,D,I,J),R3(B,F,G,H)

Now in R3 ∃ B → F and F → GH which gives non-prime attributes G,H transitively dependent on prime attribute B through F hence we will decompose it into 2 relations:

R3A(B,F) and R3B(F,G,H)

Now the given relation R is in 3NF.



b)

ABD is a key. This can also be determined by calculating ABD+ with respect to the set F.
2NF: R1 (A, B, C) R2 (B,D, E, F) R3 (A, D, G, H, J) R4 (A, I)
3NF: R1 (A, B, C) R2 (B,D, E, F) R3.1 (A, D, G, H) R4 (A, I) R3.2 (H, J)

11. The user of SQL has no idea how the data is physically represented in the machine. He or she relies entirely on the relation abstraction for querying. Physical data independence is therefore assured. Since a user can define views, logical data independence can also be achieved by using view definitions to hide changes in the conceptual schema.