Q1 Q2	**Q1. Write SQL Statements** **Assume a given schema. Write a set of SQL statements that will include JOIN and pattern matching operations.** **Q2. Write SQL Statements** **Write a set of advanced SQL statements that will include UNION and sub-queries.**

Exercise 6.1.2: Write the following queries, based on our running movie database example
Movies(title, year, length, genre, studioName, producerC#) StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
 	
a) Find the address of MGM studios.
 	
b) Find Sandra Bullock's birthdate.
 	
c) Find all the stars that appeared either in a movie made in 1980 or a movie with "Love" in the title.
	6.1.2
	
	a)
	`SELECT address AS Studio_Address`
	`FROM Studio`
	`WHERE NAME = 'MGM';`
	
	b)
	`SELECT birthdate AS Star_Birthdate`
	`FROM MovieStar`
	`WHERE name = 'Sandra Bullock';`
	
	c)
	`SELECT starName`
	`FROM StarsIn`
	`WHERE movieYear = 1980`
	`OR movieTitle LIKE '%Love%';`
	
	However, above query will also return words that have the substring Love e.g. Lover. Below query will only return movies that have title containing the word Love.
	
	`SELECT starName`
	`FROM StarsIn`
	`WHERE movieYear = 1980`
	`OR movieTitle LIKE 'Love %'`
	`OR movieTitle LIKE '% Love %'`
	`OR movieTitle LIKE '% Love'`
	`OR movieTitle = 'Love';`

d) Find all executives worth at least $10,000,000.

e) Find all the stars who either are male or live in Malibu (have string Malibu as a part of their address).

```
d)
SELECT  name AS Exec_Name
FROM    MovieExec
WHERE   netWorth >= 10000000;

e)(pattern matching)
SELECT  name AS Star_Name
FROM    movieStar
WHERE   gender = 'M'
     OR address LIKE '% Malibu %';
```

Exercise 6.1.4: Write the following queries based on the database schema of Exercise 2.4.3:
Classes(class, type, country, numGuns, bore, displacement) Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
and show the result of your query on the data of Exercise 2.4.3.

a. Find the class name and country for all classes with at least 10 guns.

b. Find the names of all ships launched prior to 1918, but call the resulting column shipName.

```
6.1.4

a)
SELECT  class,
        country
FROM    Classes
WHERE   numGuns >= 10 ;

CLASS              COUNTRY
------------------ ------------
Tennessee          USA

  1 record(s) selected.

b)
SELECT  name AS shipName
FROM    Ships
WHERE   launched < 1918 ;

SHIPNAME
------------------
Haruna
Hiei
Kirishima
Kongo
Ramillies
Renown
Repulse
Resolution
Revenge
Royal Oak
Royal Sovereign
```

	11 record(s) selected.
c. Find the names of ships sunk in battle and the name of the battle in which they were sunk.	c) SELECT ship AS shipName, battle FROM Outcomes WHERE result = 'sunk' ; SHIPNAME BATTLE ---------- ---------- Arizona Pearl Harbor Bismark Denmark Strait Fuso Surigao Strait Hood Denmark Strait Kirishima Guadalcanal Scharnhorst North Cape Yamashiro Surigao Strait 7 record(s) selected.
d. Find all ships that have the same name as their class.	d) SELECT name AS shipName FROM Ships WHERE name = class ; SHIPNAME ---------- Iowa Kongo North Carolina Renown Revenge Yamato 6 record(s) selected.
e. Find the names of all ships that begin with the letter "R."	**e) (pattern matching)** **SELECT name AS shipName** **FROM Ships** **WHERE name LIKE 'R%';** SHIPNAME ---------- Ramillies Renown Repulse Resolution Revenge Royal Oak Royal Sovereign 7 record(s) selected.

```
SELECT  name AS shipName
FROM    Ships
WHERE   name LIKE 'R%'

UNION

SELECT  ship AS shipName
FROM    Outcomes
WHERE   ship LIKE 'R%';
```

f. Find the names of all ships whose name consists of three or more words (e.g., King George V).

```
f) Only using a filter like '% % %'
will incorrectly match name such as '
a b  '
since % can match any sequence of 0
or more characters.


SELECT  name AS shipName
FROM    Ships
WHERE   name LIKE '_% _% _%' ;

SHIPNAME
------------------

  0 record(s) selected.

Note: As in (e), UNION with results
from Outcomes.

SELECT  name AS shipName
FROM    Ships
WHERE   name LIKE '_% _% _%'

UNION

SELECT  ship AS shipName
FROM    Outcomes
WHERE   ship LIKE '_% _% _%' ;



SHIPNAME
------------------
Duke of York
King George V
Prince of Wales

  3 record(s) selected.
```

Exercise 6.2.2: Write the following queries, based on the database schema

```
Product(maker,model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen,
price) Printer(model, color, type,
price)
```

of Exercise 2.4.1, and evaluate your queries using the data of that exercise.

a. Give the manufacturer and speed of laptops with a hard disk of at least thirty gigabytes.

6.2.2

a)

```
SELECT  R.maker AS manufacturer,
        L.speed AS gigahertz
FROM    Product R,
        Laptop L
WHERE   L.hd   >= 30
    AND R.model = L.model ;
```

```
MANUFACTURER GIGAHERTZ
------------ ----------
A                  2.00
A                  2.16
A                  2.00
B                  1.83
E                  2.00
E                  1.73
E                  1.80
F                  1.60
F                  1.60
G                  2.00

  10 record(s) selected.
```

b. Find the model number and price of all products (of any type) made by manufacturer *B*.

```
b)
SELECT  R.model,
        P.price
FROM    Product R,
        PC P
WHERE   R.maker = 'B'
    AND R.model = P.model

UNION

SELECT  R.model,
        L.price
FROM    Product R,
        Laptop L
WHERE   R.maker = 'B'
    AND R.model = L.model

UNION

SELECT  R.model,
        T.price
FROM    Product R,
        Printer T
WHERE   R.maker = 'B'
    AND R.model = T.model ;

MODEL PRICE
----- ------
1004     649
1005     630
1006    1049
2007    1429

  4 record(s) selected.
```

c. Find those manufacturers that sell Laptops, but not PC's.

```
c)
SELECT  R.maker
FROM    Product R,
        Laptop L
WHERE   R.model = L.model

EXCEPT

SELECT  R.maker
FROM    Product R,
        PC P
WHERE   R.model = P.model ;

MAKER
-----
F
G

  2 record(s) selected.
```

d. Find those hard-disk sizes that occur in two or more PC's.

```
d)
SELECT DISTINCT P1.hd
FROM    PC P1,
        PC P2
WHERE   P1.hd    =P2.hd
    AND P1.model > P2.model ;

Alternate Answer:

SELECT DISTINCT P.hd
FROM    PC P
GROUP BY P.hd
HAVING COUNT(P.model) >= 2 ;
```

e. Find those pairs of PC models that have both the same speed and RAM. A pair should be listed only once; e.g., list (*i , j*) but not *(j,i)*.

```
e)
SELECT  P1.model,
        P2.model
FROM    PC P1,
        PC P2
WHERE   P1.speed = P2.speed
    AND P1.ram   = P2.ram
    AND P1.model < P2.model ;

MODEL MODEL
----- -----
1004  1012

  1 record(s) selected.
```

f. Find those manufacturers of at least two different computers (PC's or laptops) with speeds of at least 3.0

```
f)
SELECT  M.maker
FROM
        (SELECT maker,
                R.model
        FROM    PC P,
                Product R
        WHERE   SPEED >= 3.0
            AND P.model=R.model

        UNION

        SELECT  maker,
                R.model
        FROM    Laptop L,
                Product R
        WHERE   speed >= 3.0
            AND L.model=R.model
        ) M
GROUP BY M.maker
HAVING COUNT(M.model) >= 2 ;

MAKER
-----
B
```

	1 record(s) selected.
Exercise 6.2.3: Write the following queries, based on the database schema Classes(class, type, country, numGuns, bore, displacement) Ships(name, class, launched) Battles(name, date) Outcomes(ship, battle, result) of Exercise 2.4.3, and evaluate your queries using the data of that exercise. a. Find the ships heavier than 35,000 tons. b. List the name, displacement, and number of guns of the ships engaged in the battle of Guadalcanal.	6.2.3 a) SELECT S.name FROM Ships S, Classes C WHERE S.class = C.class AND C.displacement > 35000; NAME ------------------ Iowa Missouri Musashi New Jersey North Carolina Washington Wisconsin Yamato 8 record(s) selected. b) SELECT S.name , C.displacement, C.numGuns FROM Ships S , Outcomes O, Classes C WHERE S.name = O.ship AND S.class = C.class AND O.battle = 'Guadalcanal' ; NAME DISPLACEMENT NUMGUNS ------------------ ------------ ----- -- Kirishima 32000 8 Washington 37000 9 2 record(s) selected.

c. List all the ships mentioned in the database. (Remember that all these ships may not appear in the Ships relation.)

Note:South Dakota was also engaged in battle of Guadalcanal but not chosen since it is not in Ships table(Hence, no information regarding it's Class is available).

c)

```
SELECT  name shipName
FROM    Ships

UNION

SELECT  ship shipName
FROM    Outcomes ;
```

```
SHIPNAME
------------------
Arizona
Bismark
California
Duke of York
Fuso
Haruna
Hiei
Hood
Iowa
King George V
Kirishima
Kongo
Missouri
Musashi
New Jersey
North Carolina
Prince of Wales
Ramillies
Renown
Repulse
Resolution
Revenge
Rodney
Royal Oak
Royal Sovereign
Scharnhorst
South Dakota
Tennesee
Tennessee
Washington
West Virginia
Wisconsin
Yamashiro
Yamato
```

```
  34 record(s) selected.
```

d. Find those countries that have both battleships and battlecruisers.

```
d)
SELECT  C1.country
FROM    Classes C1,
        Classes C2
WHERE   C1.country = C2.country
    AND C1.type    = 'bb'
    AND C2.type    = 'bc' ;

COUNTRY
------------
Gt. Britain
Japan

  2 record(s) selected.
```

e. Find those ships that were damaged in one battle, but later fought in another.

```
e)
SELECT  O1.ship
FROM    Outcomes O1,
        Battles B1
WHERE   O1.battle = B1.name
    AND O1.result = 'damaged'
    AND EXISTS
        (SELECT B2.date
        FROM    Outcomes O2,
                Battles B2
        WHERE   O2.battle=B2.name
            AND O1.ship  = O2.ship
            AND B1.date  < B2.date
        ) ;

SHIP
------------------

  0 record(s) selected.
```

f. Find those battles with at least three ships of the same country.

```
f)
SELECT  O.battle
FROM    Outcomes O,
        Ships S   ,
        Classes C
WHERE   O.ship  = S.name
    AND S.class = C.class
GROUP BY C.country,
        O.battle
HAVING COUNT(O.ship) > 3;
SELECT  O.battle
FROM    Ships S  ,
        Classes C,
        Outcomes O
WHERE   C.Class = S.class
    AND O.ship  = S.name
GROUP BY C.country,
        O.battle
HAVING COUNT(O.ship) >= 3;
```

Exercise 6.3.1: Write the following queries, based on the database schema

```
Product(maker,model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen,
price) Printer(model, color, type,
price)
```

of Exercise 2.4.1. You should use at least one subquery in each of your answers and write each query in two significantly different ways (e.g., using different sets of the operators `EXISTS`, `IN`, `ALL`, and `ANY`).

```
6.3.1
```

a. Find the makers of PC's with a speed of at least 3.0.

```
a)
SELECT DISTINCT maker
FROM    Product
WHERE   model IN
        (SELECT model
        FROM    PC
        WHERE   speed >= 3.0
        );
SELECT DISTINCT R.maker
FROM    Product R
WHERE   EXISTS
        (SELECT P.model
        FROM    PC P
        WHERE   P.speed >= 3.0
            AND P.model  =R.model
        );
```

b. Find the printers with the highest price.

```
b)
SELECT  P1.model
FROM    Printer P1
WHERE   P1.price >= ALL
        (SELECT P2.price
        FROM    Printer P2
        ) ;
SELECT  P1.model
FROM    Printer P1
WHERE   P1.price IN
        (SELECT MAX(P2.price)
        FROM    Printer P2
        ) ;
```

c. Find the laptops whose speed is slower than that of any PC.

```
c)
SELECT  L.model
FROM    Laptop L
WHERE   L.speed < ANY
        (SELECT P.speed
        FROM    PC P
        ) ;
SELECT  L.model
FROM    Laptop L
WHERE   EXISTS
        (SELECT P.speed
        FROM    PC P
        WHERE   P.speed >= L.speed
        ) ;
```

d. Find the model number of the item (PC, laptop, or printer) with the highest price.

```
d)
SELECT  model
FROM
        (SELECT model,
                price
        FROM    PC

        UNION

        SELECT  model,
                price
        FROM    Laptop

        UNION

        SELECT  model,
                price
        FROM    Printer
        ) M1
WHERE   M1.price >= ALL
        (SELECT price
        FROM    PC

        UNION

        SELECT  price
        FROM    Laptop
```

```
        UNION

        SELECT  price
        FROM    Printer
        ) ;


(d) - contd --

SELECT  model
FROM
        (SELECT model,
                price
        FROM    PC

        UNION

        SELECT  model,
                price
        FROM    Laptop

        UNION

        SELECT  model,
                price
        FROM    Printer
        ) M1
WHERE   M1.price IN
        (SELECT MAX(price)
        FROM
                (SELECT price
                FROM    PC

                UNION

                SELECT  price
                FROM    Laptop

                UNION

                SELECT  price
                FROM    Printer
                ) M2
        ) ;
```

e. Find the maker of the color printer with the lowest price.

```
e)
SELECT  R.maker
FROM    Product R,
        Printer T
WHERE   R.model  =T.model
    AND T.price <= ALL
        (SELECT MIN(price)
        FROM    Printer
        );
SELECT  R.maker
FROM    Product R,
        Printer T1
WHERE   R.model   =T1.model
    AND T1.price IN
        (SELECT MIN(T2.price)
        FROM    Printer T2
        );
```

f. Find the maker(s) of the PC(s) with the fastest processor among all those PC's that have the smallest amount of RAM.

```
f)

SELECT  R1.maker
FROM    Product R1,
        PC P1
WHERE   R1.model=P1.model
    AND P1.ram IN
        (SELECT MIN(ram)
        FROM    PC
        )
    AND P1.speed >= ALL
        (SELECT P1.speed
        FROM    Product R1,
                PC P1
        WHERE   R1.model=P1.model
            AND P1.ram IN
                (SELECT MIN(ram)
                FROM    PC
                )
        );
SELECT  R1.maker
FROM    Product R1,
        PC P1
WHERE   R1.model=P1.model
    AND P1.ram  =
        (SELECT MIN(ram)
        FROM    PC
        )
    AND P1.speed IN
        (SELECT MAX(P1.speed)
        FROM    Product R1,
                PC P1
        WHERE   R1.model=P1.model
            AND P1.ram IN
                (SELECT MIN(ram)
                FROM    PC
                )
        );
```

Exercise 6.3.7: For these relations from our running movie database schema `StarsIn(movieTitle, movieYear, starName) MovieStar(name, address, gender, birthdate) MovieExec(name, address, cert#, netWorth) Studio(name, address, presC#)` describe the tuples that would appear in the following SQL expressions: a) `Studio CROSS JOIN MovieExec;` b) `Starsln NATURAL FULL OUTER JOIN MovieStar;` c) `Starsln FULL OUTER JOIN MovieStar ON name = starName;`	**6.3.7** **(a)** **n*m tuples are returned where there are n studios and m executives. Each studio will appear m times; once for every exec.** **(b)** **There are no common attributes between StarsIn and MovieStar; hence no tuples are returned.** **(c)** **There will be at least one tuple corresponding to each star in MovieStar. The unemployed stars will appear once with null values for StarsIn. All employed stars will appear as many times as the number of movies they are working in. In other words, for each tuple in StarsIn(starName), the correspoding tuple from MovieStar(name)) is joined and returned. For tuples in MovieStar that do not have a corresponding entry in StarsIn, the MovieStar tuple is returned with null values for StarsIn columns.**
Exercise 6.3.8: Using the database schema `Product(maker,model, type) PC(model, speed, ram, hd, rd, price) Laptop(model, speed, ram, hd, screen, price) Printer(model, color, type, price)` write a SQL query that will produce information about all products — PC's, laptops, and printers — including their manufacturer if available, and whatever information about that product is relevant (i.e., found in the relation for that type of product).	**6.3.8** **Since model numbers are unique, a full natural outer join of PC, Laptop and Printer will return one row for each model. We want all information about PCs, Laptops and Printers even if the model does not appear in Product but vice versa is not true. Thus a left natural outer join between Product and result above is required. The type attribute from Product must be renamed since Printer has a type attribute as well and the two attributes are different.** **(SELECT maker,** **model,** **type AS productType** **FROM Product** **) RIGHT NATURAL OUTER JOIN ((PC FULL NATURAL OUTER JOIN Laptop) FULL NATURAL OUTER JOIN Printer);** **Alternately, the Product relation can be joined individually with each of PC,Laptop and Printer and the three**

	results can be Unioned together. For attributes that do not exist in one relation, a constant such as 'NA' or 0.0 can be used. Below is an example of this approach using PC and Laptop. SELECT R.MAKER , R.MODEL , R.TYPE , P.SPEED , P.RAM , P.HD , 0.0 AS SCREEN, P.PRICE FROM PRODUCT R, PC P WHERE R.MODEL = P.MODEL UNION SELECT R.MAKER , R.MODEL , R.TYPE , L.SPEED , L.RAM , L.HD , L.SCREEN, L.PRICE FROM PRODUCT R, LAPTOP L WHERE R.MODEL = L.MODEL;

Exercise 6.3.9: Using the two relations
 	
`Classes(class, type, country, numGuns, bore, displacement)`
 	
`Ships(name, class, launched)`
 	
from our database schema of Exercise 2.4.3, write a SQL query that will produce all available information about ships, including that information available in the `Classes` relation. You need not produce information about classes if there are no ships of that class mentioned in `Ships`.
	6.3.9
	SELECT *
	FROM Classes RIGHT NATURAL OUTER JOIN Ships ;

Exercise6.3.10: Repeat Exercise 6.3.9, but also include in the result, for any class C that is not mentioned in`Ships`, information about the ship that has the same name *C* as its class. You may assume that there is a ship with the class name, even if it doesn't appear in `Ships`.

```
6.3.10
SELECT  *
FROM    Classes RIGHT NATURAL
        OUTER JOIN Ships

UNION
        (SELECT C2.class       ,
                C2.type        ,
                C2.country     ,
                C2.numguns     ,
                C2.bore        ,
                C2.displacement,
                C2.class NAME  ,
                0
        FROM    Classes C2,
                Ships S2
        WHERE   C2.Class NOT IN
                (SELECT Class
                FROM    Ships
                )
        ) ;
```

Exercise6.3.11: The join operators (other than outerjoin) we learned in this section are redundant, in the sense that they can always be replaced by select-from-where expressions. Explain how to write expressions of the following forms using select-from-where:

a) `R CROSS JOIN S;`

b) `R NATURAL JOIN S;`

c) `R JOIN S ON` *C* where *C* is a SQL condition.

```
6.3.11
(a)
SELECT  *
FROM    R,
        S ;

(b)
Let Attr consist of
AttrR = attributes unique to R
AttrS = attributes unique to S
AttrU = attributes common to R and S
 Thus in Attr, attributes common to R
and S are not repeated.

SELECT  Attr
FROM    R,
        S
WHERE   R.AttrU1 = S.AttrU1
    AND R.AttrU2 = S.AttrU2 ...
    AND R.AttrUi = S.AttrUi ;

(c)
SELECT  *
FROM    R,
        S
WHERE   C ;
```

Q3

Exercise 7.1.1: Our running example movie database of Section 2.2.8 has keys defined for all its relations.

Movies(title , year, length, genre, studioName, producerC#)
StarsIn(movieTitle. movieYear. starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)

Declare the following referential integrity constraints for the movie database as in Exercise 7.1.1.

a) The producer of a movie must be someone mentioned in MovieExec. Mod- ifications to MovieExec that violate this constraint are rejected.

a)

```
CREATE TABLE Movies (
title           CHAR(100),
year            INT,
length          INT,
genre           CHAR(10),
studioName      CHAR(30),
producerC#      INT,
PRIMARY KEY (title, year),
FOREIGN KEY (producerC#)  REFERENCES
MovieExec(cert#)
);
```

b) Repeat (a), but violations result in the producerC# in Movie being set to NULL.

b)

```
CREATE TABLE Movies (
title           CHAR(100),
year            INT,
length          INT,
genre           CHAR(10),
studioName      CHAR(30),
producerC#      INT     REFERENCES
MovieExec(cert#)
ON DELETE SET NULL
ON UPDATE SET NULL,
PRIMARY KEY (title, year)
);
```

c) Repeat (a), but violations result in the deletion or update of the offending Movie tuple.

```
c)

CREATE TABLE Movies (
title           CHAR(100),
year            INT,
length          INT,
genre           CHAR(10),
studioName      CHAR(30),
producerC#      INT     REFERENCES
MovieExec(cert#)
ON DELETE CASCADE
ON UPDATE CASCADE,
PRIMARY KEY (title, year)
);
```

d) A movie that appears in StarsIn must also appear in Movie. Handle violations by rejecting the modification.

```
d)
CREATE TABLE StarsIn (
movieTitle      CHAR(100)
REFERENCES  Movie(title),
movieYear       INT,
starName        CHAR(30),
PRIMARY KEY (movieTItle, movieYear,
starName)
);
```

e) A star appearing in StarsIn must also appear in MovieStar. Handle violations by deleting violating tuples.

```
e)

CREATE TABLE StarsIn (
movieTitle      CHAR(100)
REFERENCES  Movie(title)
        ON DELETE CASCADE,
movieYear       INT,
starName        CHAR(30),
PRIMARY KEY (movieTItle, movieYear,
starName)
);
```

Exercise 7.1.5: Write the following referential integrity constraints for the battleships database as in Exercise 7.1.4. Use your assumptions about keys from that exercise, and handle all violations by setting the referencing attribute value to NULL.

7.1.5

a) Every class mentioned in Ships must be mentioned in Classes.

```
a)
ALTER TABLE Ships
        ADD FOREIGN KEY (class)
REFERENCES Classes (class)
                                        ON
DELETE SET NULL
                                        ON
UPDATE SET NULL;
```

b) Every battle mentioned in Outcomes must be mentioned in Battles.

c) Every ship mentioned in Outcomes must be mentioned in Ships.

```
In addition to the above declaration,
class must be declared the primary
key for Classes.

b)
ALTER TABLE Outcome
        ADD FOREIGN KEY (battle)
REFERENCES Battles (name)
                                    ON
DELETE SET NULL
                                    ON
UPDATE SET NULL;

c)
ALTER TABLE Outcomes
        ADD FOREIGN KEY (ship)
REFERENCES Ships (name)
                                  ON
DELETE SET NULL
                                  ON
UPDATE SET NULL;
```

Exercise 7.2.3: Write the following constraints as tuple-based CHECK con- straints on one of the relations of our running movies example:

Movies(title, year, length, genre, studioName, producerC#) StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)

If the constraint actually involves two relations, then you should put constraints in both relations so that whichever relation changes, the constraint will be checked on insertions and updates. Assume no deletions; it is not always pos- sible to maintain tuple-based constraints in the face of deletions.

a. A star may not appear in a movie made before they were born.

```
a)
CREATE TABLE StarsIn (
  …
  starName    CHAR(30)
      CHECK (starName IN (SELECT name
FROM MovieStar

WHERE YEAR(birthdate) > movieYear))
  …
)

b)
```

b. No two studios may have the same address.	CREATE TABLE Studio (
	  …
	  address CHAR(255) CHECK (address IS UNIQUE)
	  …
	);

c. A name that appears in MovieStar must not also appear in MovieExec.
 	
d. A studio name that appears in Studio must also appear in at least one Movies tuple.
 	
e. If a producer of a movie is also the president of a studio, then they must be the president of the studio that made the movie.
	c)
	CREATE TABLE MovieStar (
	  …
	  name CHAR(30) CHECK (name NOT IN (SELECT name FROM MovieExec))
	  …
	);
	
	d)
	CREATE TABLE Studio (
	  …
	  Name CHAR(30) CHECK (name IN (SELECT studioName FROM Movies))
	  …
	  );
	
	e)
	CREATE TABLE Movies (
	  …
	  CHECK (producerC# NOT IN (SELECT presC# FROM Studio) OR
	  studioName IN (SELECT name FROM Studio
	  WHERE presC# = producerC#))
	  …
	);

Q4	Q4. Write SQL Triggers Write triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint.

Exercise 7.5.2: Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The database schema is from the "PC" example of Exercise 2.4.1:

Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)

a. When updating the price of a PC, check that there is no lower priced PC with the same speed.

```
a)
CREATE TRIGGER LowPricePCTrigger
AFTER UPDATE OF price ON PC
REFERENCING
        OLD ROW AS OldRow,
        OLD TABLE AS OldStuff,
        NEW ROW AS NewRow,
        NEW TABLE AS NewStuff
FOR EACH ROW
WHEN (NewRow.price < ALL
        (SELECT PC.price FROM PC
         WHERE PC.speed =
NewRow.speed))
BEGIN
        DELETE FROM PC
        WHERE (model, speed, ram, hd,
price) IN NewStuff;
        INSERT INTO PC
                (SELECT * FROM
OldStuff);
END;
```

b. When inserting a new printer, check that the model number exists in Product.

```
b)
CREATE TRIGGER NewPrinterTrigger
AFTER INSERT ON Printer
REFERENCING
        NEW ROW AS NewRow,
        NEW TABLE AS NewStuff
FOR EACH ROW
WHEN (NOT EXISTS (SELECT * FROM
Product
                WHERE
Product.model = NewRow.model))
DELETE FROM Printer
        WHERE (model, color, type,
price) IN NewStuff;
```

c. When making any modification to the Laptop relation, check that the average price of laptops for each manufacturer is at least $1500.

```
c)
CREATE TRIGGER AvgPriceTrigger
AFTER UPDATE OF price ON Laptop
REFERENCING
        OLD TABLE AS OldStuff,
        NEW TABLE AS NewStuff
FOR EACH STATEMENT
WHEN (1500 > (SELECT AVG(price) FROM
Laptop))
BEGIN
        DELETE FROM Laptop
        WHERE (model, speed, ram, hd,
screen, price) IN NewStuff;
        INSERT INTO Laptop
                (SELECT * FROM
OldStuff);
END;
```

d. When updating the RAM or hard disk of any PC, check that the updated PC has at least 100 times as much hard disk as RAM.

```
d)
CREATE TRIGGER HardDiskTrigger
AFTER UPDATE OF hd, ram ON PC
REFERENCING
        OLD ROW AS OldRow,
        OLD TABLE AS OldStuff,
        NEW ROW AS NewRow,
        NEW TABLE AS NewStuff
FOR EACH ROW
WHEN (NewRow.hd < NewRow.ram * 100)
BEGIN
        DELETE FROM PC
        WHERE (model, speed, ram, hd,
price) IN NewStuff;
        INSERT INTO PC
                (SELECT * FROM
OldStuff);

END;
```

e. When inserting a new PC, laptop, or printer, make sure that the model number did not previously appear in any of PC, Laptop, or Printer.

```
e)
CREATE TRIGGER DupModelTrigger
BEFORE INSERT ON PC, Laptop, Printer
REFERENCING
        NEW ROW AS NewRow,
        NEW TABLE AS NewStuff
FOR EACH ROW
WHEN (EXISTS (SELECT * FROM NewStuff
NATUAL JOIN PC)
             UNION ALL
             (SELECT * FROM NewStuff
NATUAL JOIN Laptop)
             UNION ALL
             (SELECT * FROM NewStuff
NATUAL JOIN Printer))
BEGIN
```

	SIGNAL SQLSTATE '10001' ('Duplicate Model – Insert Failed'); END;
Exercise 7.5.3: Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The database schema is from the battleships example of Exercise 2.4.3. Classes(class, type, country, numGuns, bore, displacement) Ships(name, class, launched) Battles(name, date) Outcomes(ship, battle, result) a. When a new class is inserted into Classes, also insert a ship with the name of that class and a NULL launch date.	a) CREATE TRIGGER NewClassTrigger AFTER INSERT ON Classes REFERENCING NEW ROW AS NewRow FOR EACH ROW BEGIN INSERT INTO Ships (name, class, lunched) VALUES (NewRow.class, NewRow.class, NULL); END
b. When a new class is inserted with a displacement greater than 35,000 tons, allow the insertion, but change the displacement to 35,000.	b) CREATE TRIGGER ClassDisTrigger BEFORE INSERT ON Classes REFERENCING NEW ROW AS NewRow, NEW TABLE AS NewStuff FOR EACH ROW WHEN (NewRow.displacement > 35000) UPDATE NewStuff SET displacement = 35000;
c. If a tuple is inserted into Outcomes, check that the ship and battle are listed in Ships and B attles, respectively, and if not, insert tuples into one or both of these relations, with NULL components where necessary.	c) CREATE TRIGGER newOutcomesTrigger AFTER INSERT ON Outcomes REFERENCING NEW ROW AS NewRow FOR EACH ROW WHEN (NewRow.ship NOT EXISTS (SELECT name FROM Ships))

	INSERT INTO Ships (name, class, lunched) VALUES (NewRow.ship, NULL, NULL); CREATE TRIGGER newOutcomesTrigger2 AFTER INSERT ON Outcomes REFERENCING NEW ROW AS NewRow FOR EACH ROW WHEN (NewRow.battle NOT EXISTS (SELECT name FROM Battles)) INSERT INTO Battles (name, date) VALUES (NewRow.battle, NULL);

d. When there is an insertion into Ships or an update of the class attribute of Ships, check that no country has more than 20 ships.
(solution for d is weird !)
	d)
	CREATE TRIGGER changeShipTrigger
	AFTER INSERT ON Ships
	REFERENCING
	NEW TABLE AS NewStuff
	FOR EACH STATEMENT
	WHEN (20 < ALL
	(SELECT COUNT(name) From Ships NATURAL JOIN Classes
	GROUP BY country))
	DELETE FROM Ships
	WHERE (name, class, launched) IN NewStuff;
	
	CREATE TRIGGER changeShipTrigger2
	AFTER UPDATE ON Ships
	REFERENCING
	OLD TABLE AS OldStuff,
	NEW TABLE AS NewStuff
	FOR EACH STATEMENT
	WHEN (20 < ALL
	SELECT COUNT(name) From Ships NATURAL JOIN Classes
	GROUP BY country))
	BEGIN
	DELETE FROM Ships
	WHERE (name, class, launched) IN NewStuff;
	INSERT INTO Ships
	(SELECT * FROM OldStuff);
	END;

Check, under all circumstances that could cause a violation, that no ship fought in a battle that was at a later date than another battle in which that ship was sunk.	e) CREATE TRIGGER sunkShipTrigger AFTER INSERT ON Outcomes REFERENCING NEW ROW AS NewRow NEW TABLE AS NewStuff

	FOR EACH ROW
	WHEN ((SELECT date FROM Battles
	WHERE name = NewRow.battle)
	< ALL
	(SELECT date FROM Battles
	WHERE name IN (SELECT
	battle FROM Outcomes
	
	WHERE ship = NewRow.ship AND
	
	result = "sunk"
	)
	)
	)
	
	DELETE FROM Outcomes
	WHERE (ship, battle, result) IN
	NewStuff;
	
	CREATE TRIGGER sunkShipTrigger2
	AFTER UPDATE ON Outcomes
	REFERENCING
	NEW ROW AS NewRow,
	NEW TABLE AS NewStuff
	FOR EACH ROW
	FOR EACH ROW
	WHEN ((SELECT date FROM Battles
	WHERE name = NewRow.battle)
	< ALL
	(SELECT date FROM Battles
	WHERE name IN (SELECT
	battle FROM Outcomes
	
	WHERE ship = NewRow.ship AND
	
	result = "sunk"
	)
	)
	)
	BEGIN
	DELETE FROM Outcomes
	WHERE (ship, battle, result)
	IN NewStuff;
	INSERT INTO Outcomes
	(SELECT * FROM
	OldStuff);
	END;

Exercise 7.5.4: Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The problems are based on our running movie example:	

Movies(title, year, length, genre, studioName, producerC#) StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
You may assume that the desired condition holds before any change to the database is attempted. Also, prefer to modify the database, even if it means inserting tuples with NULLor default values, rather than rejecting the attempted modification.

a) Assure that at all times, any star appearing in StarsIn also appears in MovieStar

a.

```
CREATE TRIGGER changeStarsInTrigger
AFTER INSERT ON StarsIn
REFERENCING
        NEW ROW AS NewRow,
FOR EACH ROW
WHEN (NewRow.starName NOT EXISTS
                (SELECT name FROM
MovieStar))
INSERT INTO MovieStar(name)

VALUES(NewRow.starName);

CREATE TRIGGER changeStarsInTrigger2
AFTER UPDATE ON StarsIn
REFERENCING
        NEW ROW AS NewRow,
FOR EACH ROW
WHEN (NewRow.starName NOT EXISTS
                (SELECT name FROM
MovieStar))
INSERT INTO MovieStar(name)

VALUES(NewRow.starName);
```

b) Assure that at all times every movie executive appears as either a studio producer of a movie, or both.

```
b)
CREATE TRIGGER changeMovieExecTrigger
AFTER INSERT ON MovieExec
REFERENCING
        NEW ROW AS NewRow,
FOR EACH ROW
WHEN (NewRow.cert# NOT EXISTS
                (SELECT presC# FROM
Studio)
              UNION ALL
                SELECT producerC#
FROM Movies)
     )
INSERT INTO Movies(procucerC#)
              VALUES(NewRow.cert#);
```

c) Assure that every movie has at least one male and one female star. d) Assure that the number of movies made by any studio in any year is no more than 100.	* insert into the relation Movies rather than Studio since there's no associated info with Studio. CREATE TRIGGER changeMovieExecTrigger2 AFTER UPDATE ON MovieExec REFERENCING NEW ROW AS NewRow, FOR EACH ROW WHEN (NewRow.cert# NOT EXISTS (SELECT presC# FROM Studio) UNION ALL SELECT producerC# FROM Movies)) INSERT INTO Movies(procucerC#) VALUES(NewRow.cert#); c) CREATE TRIGGER changeMovieTrigger AFTER DELETE ON MovieStar REFERENCING OLD TABLE AS OldStuff, FOR EACH STATEMENT WHEN (1 > ALL (SELECT COUNT(*) FROM StarIn s, MovieStar m WHERE s.starName = m.name GROUP BY s.movieTitle, m.gendar)) INSERT INTO MovieStar (SELECT * FROM OldStuff); d) CREATE TRIGGER numMoviesTrigger AFTER **INSERT** ON Movies REFERENCING NEW TABLE AS NewStuff FOR EACH STATEMENT WHEN (100 < ALL (SELECT COUNT(*) FROM Movies GROUP BY studioName, year)) DELETE FROM Movies WHERE (title, year, length, genre, StudioName, procedureC#)IN NewStuff; CREATE TRIGGER numMoviesTrigger2 AFTER **UPDATE** ON Movies REFERENCING

e) Assure that the average length of all movies made in any year is no more than 120.	OLD TABLE AS OldStuff
	NEW TABLE AS NewStuff
	FOR EACH STATEMENT
	WHEN (100 < ALL
	(SELECT COUNT(*) FROM Movies
	GROUP BY studioName, year))
	BEGIN
	DELETE FROM Movies
	WHERE (title, year, length, genre, StudioName, procedureC#)
	IN NewStuff;
	INSERT INTO Movies
	(SELECT * FROM OldStuff);
	END;
	
	e)
	CREATE TRIGGER avgMovieLenTrigger
	AFTER **INSERT** ON Movies
	REFERENCING
	NEW TABLE AS NewStuff
	FOR EACH STATEMENT
	WHEN (120 < ALL
	(SELECT AVG(length) FROM Movies
	GROUP BY year))
	DELETE FROM Movies
	WHERE (title, year, length, genre, StudioName, procedureC#)IN NewStuff;
	
	CREATE TRIGGER avgMovieLenTrigger2
	AFTER **UPDATE** ON Movies
	REFERENCING
	OLD TABLE AS OldStuff
	NEW TABLE AS NewStuff
	FOR EACH STATEMENT
	WHEN (120 < ALL
	(SELECT AVG(length) FROM Movies
	GROUP BY year))
	BEGIN
	DELETE FROM Movies
	WHERE (title, year, length, genre, StudioName, procedureC#)
	IN NewStuff;
	INSERT INTO Movies
	(SELECT * FROM OldStuff);
	END;

chapter 8 Q5. Write materialized views	Q5. Write materialized views Given set of base tables and materialized view that is based on the given base tables.

	What modifications to the base tables that would require changes to the Materialized View and how do you propagate the changes incrementally to the materialized view?
Exercise 8.5.2: Suppose the view NewPC of Exercise 8.2.3 were a materialized view. What modifications to the base tables Product and PC would require a modification of the materialized view? How would you implement those modi- fications incrementally? Using the base tables Product(maker, model, type) PC(model, speed, ram, hd, price) suppose we create the view: CREATE VIEW NewPC AS SELECT maker, model, speed, ram, hd, price FROM Product, PC WHERE Product.model = PC.model AND type = 'pc';	**Exercise 8.5.2** Insertions, deletions, and updates to the base tables Product and PC would require a modification of the materialized view. Insertions into Product with type equal to 'pc': INSERT INTO NewPC SELECT maker, model, speed, ram, hd, price FROM Product, PC WHERE Product.model = newModel and Product.model = PC.model; Insertions into PC: INSERT INTO NewPC SELECT maker, 'newModel', 'newSpeed', 'newRam', 'newHd', 'newPrice' FROM Product WHERE model = 'newModel'; Deletions from Product with type equal to 'pc': DELETE FROM NewPC WHERE maker = 'deletedMaker' AND model='deletedModel'; Deletions from PC: DELETE FROM NewPC WHERE model = 'deletedModel'; Updates to PC: Update NewPC SET speed=PC.speed, ram=PC.ram, hd=PC.hd, price=PC.price FROM PC where model=pc.model;

	Update to the attribute 'model' needs to be treated as a delete and an insert. Updates to Product: Any changes to a Product tuple whose type is 'pc' need to be treated as a delete or an insert, or both
Exercise 8.5.3: This exercise explores materialized views that are based on aggregation of data. Suppose we build a materialized view on the base tables Classes(class, type, country, numGuns, bore, displacement) Ships(name, class, launched) from our running battleships exercise, as follows: CREATE MATERIALIZED VIEW ShipStats AS SELECT country, AVG(displacement), COUNT(*) FROM Classes, Ships WHERE Classes.class = Ships.class GROUP BY country; What modifications to the base tables Classes and Ships would require a modification of the materialized view? How would you implement those modi- fications incrementally?	Modifications to the base tables that would require a modification to the materialized view: inserts and deletes from Ships, deletes from class, updates to a Class' displacement. Deletions from Ship: UPDATE ShipStats SET displacement=((displacement * count) – (SELECT displacement FROM Classses WHERE class = 'DeletedShipClass')) / (count – 1), count = count – 1 WHERE country = (SELECT country FROM Classes WHERE class='DeletedShipClass'); Insertions into Ship: Update ShipStat SET displacement=((displacement*count) + (SELECT displacement FROM Classes WHERE class='InsertedShipClass')) / (count + 1), count = count + 1 WHERE

	country = (SELECT country FROM Classes WHERE classes='InsertedShipClass); Deletes from Classes: NumRowsDeleted = SELECT count(*) FROM ships WHERE class = 'DeletedClass'; UPDATE ShipStats SET displacement = (displacement * count) - (DeletedClassDisplacement * NumRowsDeleted)) / (count – NumRowsDeleted), count = count – NumRowsDeleted WHERE country = 'DeletedClassCountry'; Update to a Class' displacement: N = SELECT count(*) FROM Ships where class = 'UpdatedClass'; UPDATE ShipsStat SET displacement = ((displacement * count) + ((oldDisplacement – newDisplacement) * N))/count WHERE country = 'UpdatedClassCountry';