Week 5

Theory

Views

- Table which does not physically exist
- Queried as normal relations, sometimes updatable
- CREATE VIEW … AS …

Modifying views

- Possible for "Updatable views"
    - Translating action into equivalent action on base table
    - Or using Instead-of trigger
- Updatable Views
    - Selecting <attrs> from R
    - R could be updatable view itself
    - Where clause must not involve R in subquery
    - From clause consist of one occurrence of R and no other relation
    - <attrs> include enough attributes, to be able to insert into R, using either NULL or defaults.
- Instead of triggers
    - Intercepts action to modify view, performs action on relation itself

```
CREATE TRIGGER ParamountInsert
INSTEAD OF INSERT ON ParamountMovies
REFERENCING NEW ROW AS NewRow
FOR EACH ROW
INSERT INTO Movies(title, year, studioName)
VALUES(NewRow.title, NewRow.year, 'Paramount');
```

Indexes

- Structure which makes it efficient to find tuples that have fixed value for some attribute
- Expensive to scan for tuples matching some condition
- CREATE INDEX <name> on <relation>(<attr>)

Materialized views

- Maintaining view at all times
- Should be recomputed when underlying base tables change
- Stored like any base table
- Never have to reconstruct entire view from scratch
- Typically maintained periodically (e.g. each night)

2.1 (page 341)

Exercise 2.1 : Which of the views of Exercise 1.1 are updatable?

Exercise 1.1 : From the following base tables of our running example

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Construct the following views:

a) A view `RichExec` giving the name, address, certificate number and net worth of all executives with a net worth of at least $10,000,000.

b) A view `StudioPres` giving the name, address, and certificate number of all executives who are studio presidents.

c) A view `ExecutiveStar` giving the name, address, gender, birth date, certificate number, and net worth of all individuals who are both executives and stars.

a. …

Yes, all from same relation…

b. …

No, we are joining two tables to get this.

c. …

No, joining again.

2.2

Exercise 2.2: Suppose we create the view:

```
CREATE VIEW DisneyComedies AS
    SELECT title, year, length FROM Movies
    WHERE studioName = 'Disney' AND genre = 'comedy';
```

a) Is this view updatable?

b) Write an instead-of trigger to handle an insertion into this view.

c) Write an instead-of trigger to handle an update of the length for a movie (given by title and year) in this view.

a. …

Yes

b. …

```
CREATE TRIGGER DisneyInserts
INSTEAD OF INSERT ON DisneyComedies
REFERENCING NEW ROW AS newRow
FOR EACH ROW
INSERT INTO Movies (title, year, length, studioName, genre)
VALUES(newRow.title, newRow.year, newRow.length, ‘Disney’, ‘comedy’);
```

c. …

```
CREATE TRIGGER DisneyInserts
INSTEAD OF UPDATE length ON DisneyComedies
REFERENCING NEW ROW AS newRow
FOR EACH ROW
UPDATE Movies
SET length = newRow.length
WHERE title = newRow.title AND year = newRow.year;
```

2.3

Exercise 2.3: 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';
```

Notice that we have made a check for consistency: that the model number not only appears in the PC relation, but the type attribute of Product indicates that the product is a PC.

a) Is this view updatable?

b) Write an instead-of trigger to handle an insertion into this view.

c) Write an instead-of trigger to handle an update of the price.

d) Write an instead-of trigger to handle a deletion of a specified tuple from this view.

a. …

No, we are joining two tables

b. …

```
CREATE TRIGGER NewPCInserts
INSTEAD OF INSERT ON NewPC
REFERENCING NEW ROW AS newRow
FOR EACH ROW
BEGIN
        INSERT INTO Products (model, maker, type)
        VALUES(newRow.model, newRow.maker, 'pc');
        INSERT INTO PC (model, speed, ram, hd, price)
        VALUES(newRow.model, newRow.speed, newRow.ram, newRow.hd,
        newRow.price);
END;
```

c. …

```
CREATE TRIGGER NewPCPriceUpdate
INSTEAD OF UPDATE price ON NewPC
REFERENCING NEW ROW AS newRow
FOR EACH ROW
UPDATE PC
SET price = newRow.price
WHERE model = newRow.model;
```

d. …

```
CREATE TRIGGER NewPCDeletion
INSTEAD OF DELETE ON NewPC
REFERENCING OLD ROW AS oldRow
FOR EACH ROW
BEGIN
        DELETE FROM Products
        WHERE model = oldRow.model;
        DELETE FROM PC
        WHERE model = oldRow.model;
END;
```

3.1 (page 344)

Exercise 3.1: For our running movies example:

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

Declare indexes on the following attributes or combination of attributes:

a) `studioName`.

b) `address` of `MovieExec`.

c) `genre` and `length`.

a. …

```
CREATE INDEX studioNameIndex ON Movies(studioName)
```

b. …

```
CREATE INDEX address ON MovieExec(address)
```

c. …

```
CREATE INDEX genreLength ON Movies(genre, length)
```

4.1 (page 351)

Exercise 4.1: Suppose that the relation `StarsIn` discussed in Example 14 required 100 pages rather than 10, but all other assumptions of that example continued to hold. Give formulas in terms of p_1 and p_2 to measure the cost of queries Q_1 and Q_2 and insertion I, under the four combinations of index/no index discussed there.

Action	No Index	Star Index	Movie Index	Both Indexes
$\boldsymbol{Q_1}$	100	4	100	4
$\boldsymbol{Q_2}$	100	100	4	4
$\boldsymbol{I}$	2	4	4	6
$\boldsymbol{Average}$	$98 \cdot p_1 + 98 \cdot p_2 + 2$	$96 \cdot p_2 + 4$	$96 \cdot p_1 + 4$	$6 - 2p_1 - 2p_2$

4.2

! Exercise 4.2: In this problem, we consider indexes for the relation

`Ships(name, class, launched)`

from our running battleships exercise. Assume:

i. `name` is the key.

ii. The relation `Ships` is stored over 50 pages.

iii. The relation is clustered on `class` so we expect that only one disk access is needed to find the ships of a given class.

iv. On average, there are 5 ships of a class, and 25 ships launched in any given year.

v. With probability p_1 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE name =` n.

vi. With probability p_2 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE class =` c.

vii. With probability p_3 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE launched =` y.

viii. With probability $1 - p_1 - p_2 - p_3$ the operation on this relation is an insertion of a new tuple into `Ships`.

You can also make the assumptions about accessing indexes and finding empty space for insertions that were made in Example 14.

Consider the creation of indexes on `name`, `class`, and `launched`. For each combination of indexes, estimate the average cost of an operation. As a function of p_1, p_2, and p_3, what is the best choice of indexes?

Action	**No Index**	***name* Index**	***class* Index**	***launched* index**	***name class* index**	***name launched* index**	***class launched* index**	**All Indexes**
Q_1	50	2	50	50	2	2	50	2
Q_2	50	50	2	50	2	50	2	2
Q_3	50	50	50	26	50	26	26	26
I	51	53	4	53	6	55	6	8
Avg	$51 - p_1 - p_2 - p_3$	$53 - 51p_1 - 3p_2 - 3p_3$	$4 + 46p_1 - 2p_2 + 46p_3$	$53 - 3p_1 - 3p_2 - 27p_3$	$6 - 4p_1 - 4p_2 + 44p_3$	$55 - 53p_1 - 5p_2 - 31p_3$	$6 + 44p_1 - 4p_2 + 18p_3$	$8 - 6p_1 - 6p_2 + 16p_3$

5.1 (page 257)

Exercise 5.1: Complete Example 15 by considering updates to either of the base tables.

```
CREATE MATERIALIZED VIEW MovieProd AS
    SELECT title, year, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#
```

We leave as an exercise the consideration of how updates to `Movies` that involve `title` or `year` are handled, and how updates to `MovieExec` involving `cert#` are handled. □

Updating title, year from Movies, e.g. from (‘Dumb & Dumber’, 1994) to (‘Dumb & Dumber 2’, 1996):

```
UPDATE MovieProd

SET title = ‘Dumb & Dumber 2’, year = 1996

Where title = ‘Dumb & Dumber’ AND year = 1995
```

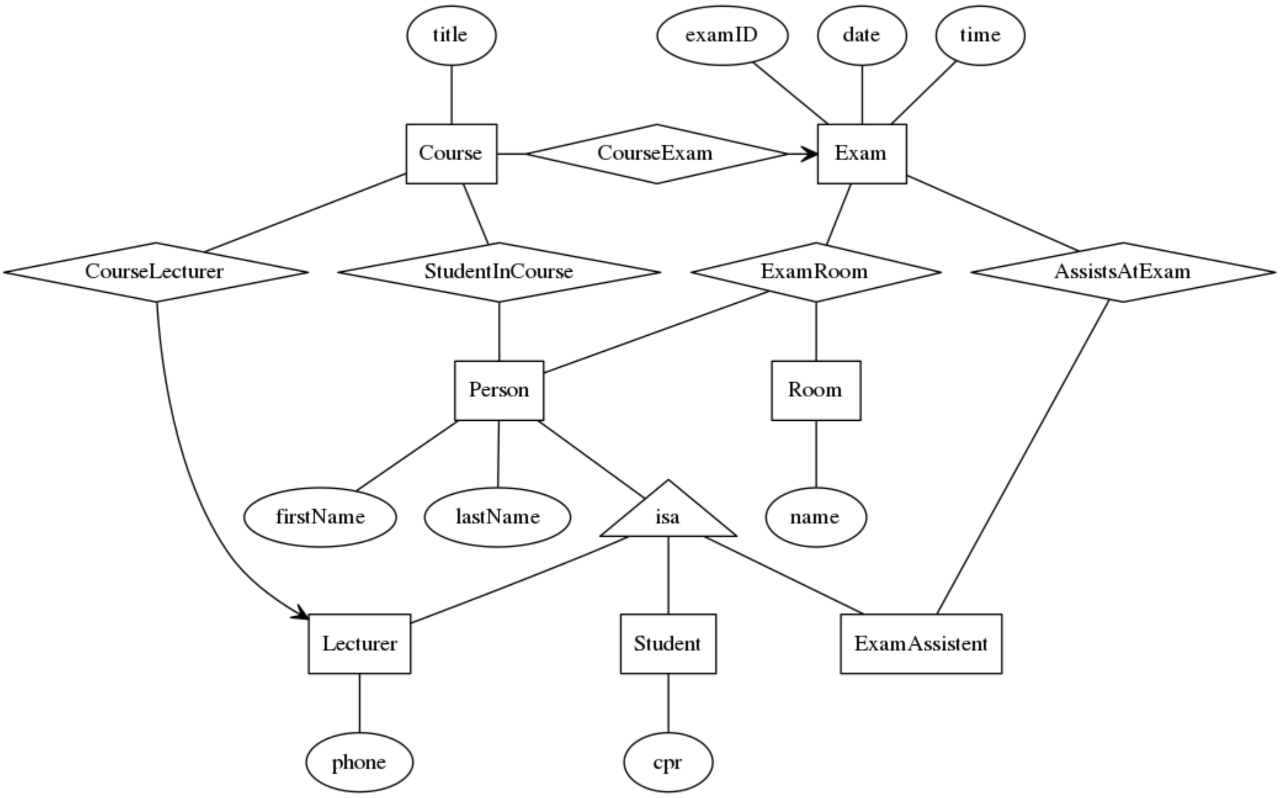
Updating #cert, e.g. from 45678 to 45679

```
DELETE FROM MovieProd
Where #cert = 45678

INSERT INTO MovieProd
        Select title, year, name
        From Movies, MovieExec
        Where producer# = 45679
```

Old Exam Task1

E/R diagram



Obs. This is modelled like described, should probably have some CPR attribute for the Person, relation, otherwise firstName and lastName is the key.

Converting into relation schemas

Exam(<u>examID</u>, date, time, cource_title)

(course title is added to exam)

Person(firstName, lastName, <u>CPR</u>, Phone, examAss, Student, Lecturer)

(converting above by using nulls, and including role bool for each of the roles)

CourseTaughtBy(<u>CPR</u>, <u>examID</u>)

isStudentIn(<u>CPR</u>, <u>ExamID,</u> examRoom)

(putting exam room in this as well)

isAssisting(<u>CPR</u>, <u>ExamID</u>)

Old Exam Task2

Relation	**FD's**	**Keys**	**BCNF Violations**	**BCNF Decomposition**
$R(A,B,C,D,E)$	$AB \to C$ $AC \to D$ $D \to B$ $E \to D$ $AB \to D$	AE	All FD's	Decompose on: $AB \to C$ $R1(A,B,C,D)$ $R2(A,B,E)$
$R1(A,B,C,D)$	$AB \to C$ $AC \to D$ $D \to B$ $AB \to D$	AD AB AC	$D \to B$	Decompose on: $D \to B$ $R11(D,B)$ $R12(A,C,D)$
$R11(D,B)$	$D \to B$	D	None	
$R12(A,C,D)$	$AC \to D$	AC	None	
$R2(A,B,E)$	None	ABE	None	

Meaning we get:

$$R11(D,B)$$

$$R12(A,C,D)$$

$$R2(A,B,E)$$

Relation	**FD's**	**Keys**	**3NF Violations**	**Minimal basis**	**3NF Decomposition**
$R(A,B,C,D,E)$	$AB \to C$ $AC \to D$ $D \to B$ $E \to D$ $AB \to D$	AE	$AB \to C$ $AC \to D$ $D \to B$ $E \to D$ $AB \to D$	$AB \to C$ $AC \to D$ $D \to B$ $E \to D$	$R1(A,B,C)$ $R3(A,C,D)$ $R4(D,B)$ $R5(A,E)$

Old Exam Task3

(a) Events always start and end on the same day at exact hours, i.e., “start” is always a smaller integer than “end”. Give a `CREATE TABLE` statement for creating the table *Event* that respects this.

```
CREATE TABLE EVENT (
        title       CHAR(20),
        date        DATE,
        start       INT,
        end         INT
        PRIMARY KEY(title),
        CHECK (start < end)

);
```

(b) Write one `SELECT FROM WHERE` query that lists the names and phone numbers of all volunteers helping at the event ‘Rock around the Clock’.

```
SELECT DISTINCT name, phone

FROM Volunteer, Helps

WHERE

        Helps.title = ‘Rock around the Clock’
        AND
        Volunteer.name = Helps.name

;
```

(c) Write one `UPDATE` statement that increases the "premeet" time of all organizer-event pairs, where there are no volunteers associated to the same event, by 2 hours.

```
UPDATE Organizes

SET premeet = premeet + 2

Where title in (

        SELECT DISTINCT title
        From Event

        DIFFERENCE

        SELECT DISTINCT title
        FROM helps

)

;
```

(d) Write one `SELECT FROM WHERE` query that computes the ratio of unpaid hours to paid hours for each event together with the amount paid to the main organizer according to his hourly "salary". Unpaid hours are defined as the duration of the event times the number of volunteers while paid hours are defined as the duration of the event plus the number of premeet hours.

```
SELECT (end-start*count(*)) / (end-start+premeet) as ratio, salary

FROM Event, Helps, Organizes, Organizer

WHERE

        Event.title = Helps.title
        AND
        Event.title = Organizes.title
        AND
        Organizes.name = Organizer.name

GROUP BY Helps.title;
```