# January 22, 2018

Quiz 1 on January 29th with iClicker

*(suite of last class) (slide 63)*

***Aggregation – Grouping***

5 diff ops that you might want to do with queries

- might need to group/combine tuples
  - Ex if John took 3 classes, you will need to combine his grades to have a GPA

Relation Schema: **Movie**(title, year, length, filmType, studioName, producer#)

Query: what is total length in min produced by each studio?

Query in SQL:

```
SELECT studioName, SUM(length)
FROM Movie
GROUP BY studioName;
```

*Sum is aggregation**

If you don't have GROUP BY, there is no partitioning.

- The entire relation will be a partition = only one output.
- you will get the sum of ALL movie tuples and normally you will get the first tuple in the studioName.

***When you do aggregation, the only attributes not aggregated will be the ones in the GROUP BY***.

Can we use GROUP BY without using aggregation?

So if you don't have SUM(length).=> GROUP BY it will create a partition for everything; so without using aggregation, it will only output 1 of those studio names. **You CAN'T have GROUP BY without Aggregation**.

Relation Schema:

**Movie**(title, year, length, filmType, studioName, producerC#)

**Exec**(name, address, cert#, netWorth)

Query: for reach producer (name), list total length of films produced?

Query SQL:

```
SELECT Exec.name, SUM(Movie.length)
FROM Exec, Movie
WHERE Movie.producerC# = Exec.cert#
GROUP BY Exec.name;
```

SQL = declarative language

Imperative = when you say how to proceed (Java, C)

Declarative = when you say what you want

***Rule about null values***
If studio activity has a null value, what to do?
Nulls are counted **when grouping but ignored when aggregating**
= they are taking into consideration when partitioning
Ex: Consider instance of R(A,B)
How many different tuples : 4 for A (null, 1, 2, 3), so this partition is divided in 4.

| A | B |
|---|---|
| - | 1 |
| - | - |
| 1 | - |
| 2 | 1 |
| 2 | 3 |
| 3 | 2 |

tuples:

| | |
|---|---|
| - | 1 |
| 1 | - |
| 2 | 4 |
| 3 | 2 |

***What would not be in the output? (null, null)***

Ex: R(A,B) containing: (null, 1), (2,1), (null, null), (3,2), (2,3), (1, null)

| A | B | **is B different from 2** |
|---|---|---|
| - | 1 | yes |
| - | - | not counted |
| 1 | - | not counted |
| 2 | 1 | yes |
| 2 | 3 | yes |
| 3 | 2 | not diff from 2 |

Now Group By A values

| | |
|---|---|
| - | 1 |
| 1 | - |
| 2 | 4 |
| 3 | 2 |

answer: (2,4)

***Aggregating –HAVING clause***
can follow group by clause with a HAVING clause HAVING is followed by some conditions about the group
Ex:

relation schema:

**Movie**(title, year, length, filmType, studioName, producerC#)
**Exec**(name, address, cert#, netWorth)

**Query**: for producers who made at least one film prior to 1930 list total length of films produced?

**Query in SQL**:

```
SELECT Exec.name, SUM(Movie.length)
FROM Exec, Movie
WHERE producer# = cert#
GROUP BY Exec.name
HAVING MIN(Movie.year) < 1930;
```

**Query** which chooses the movies based on property of **each movie tuple**:

```
SELECT Exec.name, SUM(Movie.length)
FROM Exec, Movie
WHERE producer# = cert# AND Movie.year <1930
GROUP BY Exec.name
```

By putting it in the WHERE clause it will **make it single tuple**. A set of tuples in each conditions VS A tuple based conditions

***Order By***

If you want your output to have a certain order.
Ex:

**Movie**(title, year, length, filmType, studioName, producerC#)

```
SELECT Exec.name, SUM(Movie.length)
FROM Exec, Movie
WHERE product# = cert#
GROUP BY Exec.name
HAVING MIN(Movie.year) < 1930
ORDER BY Exec.name ASC;
```

In general: **ORDER BY** A **ASC**, B **DESC**, C **ASC**;
ASC: smallest to largest  DESC: largest to smallest

**Databases modifications**

How to change the state (content) of the database?

- 3 types of ***transactions/statements***:
  - **Insert** tuples into a relation
  - **Delete** certain tuples from a relation
  - **Update** values of certain attributes of certain existing tuples

***Insertion***

Consist of:

- **INSERT INTO**
- name of relation **R** (table in database)
- parenthesized list of attributes of relation R
- **VALUES**
- *turple* expression

Format: **INSERT INTO R(A1, …, An)VALUES(V1,…, Vn)**
if A1 is a string, V1 is a string. = must be same type
So VALUES will be insert into the table
There is another way to declare:

**INSERT INTO R VALUES(1, 'CSE' 25);**

suppose R(A,B,C,D) and then say INSERT INTO R(C, D) VALUES('a', 23);
So you are choosing the attributes you want in R, but R should have 4 components, so A, B have not been given values, they will by default by added null. Another way equivalent: INSERT INTO R(A,B,C,D) VALUES(null, null, 'a', 23);
BUT IF YOU DO INSERT INTO R(A,B,C,D) VALUES('a', 23) it will give an error.
Ex:
Relation schema: **StarsIn**(title, year, starName)
Update the database: Add "Sydney Greenstreet" to the list of stars of The Maltese Falcon
in SQL:

**INSERT INTO** StarsIn(title, year, starName)
**VALUES**('The Maltese Falcon', 1942, 'Sydney Greenstreet');

***THE KEY ATTRIBUTES CANNOT BE NULL***

ANOTHER FORMULATION:

**INSERT INTO StarsIn**
**VALUES**('The Maltese Falcon', 1942, 'Sydney Greenstreet');

These where simple insertion statement.

Ex:
Database schema:

**Studio**(name, address, presC#)
**Movie**(title, year, length, filmType, studioName, producer#)

Update the database

Add to **Studio**, all studio names mentioned in **Movie** relation

In SQL:

```
INSERT INTO Studio(name)
SELECT DISTINCT  studioName
FROM Movie
WHERE studioName NOT IN(SELECT name FROM studio)
```

NOT IN will provide set of values
*So we are saying*: if studioName is NOT in that set of values, then I want it.
When you do an insert, it will depends of the implementation, usually it should be at the bottom.

***Deletion***
Consist of:

- **DELETE FROM**
- name of relation **R**
- **WHERE**
- a condition

Format: **DELETE FROM R WHERE** <condition>;
*If you want to delete a table**: DROP table Movie**; for example.*
**Effect** : every tuple in relation **R** satisfying the condition will be deleted from R
*Note: unlike INSERT, we MAY need a WHERE clause here*

If you say **DELETE FROM R WHERE** < >
YOU WILL GET A SYNTAX ERROR because there is no condition.
If you don't want any condition you need to write **DELETE FROM R**;
DELETE FROM should not be use to delete an entire table/schema.

Ex:
Relation schema: **Exec**(name, address, cert#, netWortk)
Update: delete every movie executive whose netWorth < 10,000,000
In SQL:

```
DELETE FROM Exec
WHERE netWorth < 10,000,000;
```

Anything wrong here? Do not put the commas for the numbers.

Ex:
Relation Schema:
**Studio**(name, address, presC#)
**Movie**(title, year, length, filmType, studioName, producer#)
Update: delete **from** Studio, those studios not mentioned in **Movie**
In SQL:

```
DELETE FROM Studio
WHERE name NOT IN(SELECT StudioName FROM Movie);
```

You can have a constant relation:

```
SELECT *
FROM R
WHERE a NOT IN (1,2);
```

it's a constant relation as the set of values are fixed.
Another version of it:

```
SELECT *
FROM R
WHERE (a,c) IN (('R'), (3,25));
```

For every tuple, it will check if a,c are in set R and in set (3,25)

***Update***

Consist of:

- **UPDATE**
- name of relation **R**
- **SET**
- list of formulas, each of which will assign a value to attribute of R
- **WHERE**
- a condition

Format: **UPDATE** R **SET**<new-value assignments> **WHERE** <condition>;

Ex:
Database schema:

**Studio**(name, address, presC#)
**Exec**(name, address, cert#, netWorth)

Update: modify table **Exec** by attaching the title '**Pres.**' in front of name of every movie executive who is also president of some studio
In SQL:

```
UPDATE Exec
SET name = 'Pres.' || name
WHERE  cert# IN (SELECT presC# FROM Studio);
```

---

## January 24, 2018

**Quiz 0 : Frequence DB**

*Q1 In relational DB model, a relation instance is a…*

- **A. set of tuple**
- B. list of tuple => order is important so can't be
- C. *collection of tuples* => can contain multiple duplicate, order etc.
- D. bag of tuples => duplicates are allowed, which shouldn't be. relation is a set doesn't have a duplicate.

*Q2 Let R(a) be a relation schema in which the domain of A includes all integers. Then which of following statements is NOT correct about an instance r of R?*

- A. r can have any nb of tuples
- ***B. r can be infinite* (limited space in DB)**
- C. r can be empty

Q3 *Let R(A,B) be a relation that has n tuples. Suppose null is allowed for A. Then of the following statements, which ones describes best the number of tuples returned by the query:*

```
SELECT A FROM R GROUP BY A;
```

- *A. At least n*
- B. Exactly n (if there no null, then exactly n)
- C. **At most n** (if there is at most n as null are counted)

## *Defining DB Schema*

SQL includes 2 types of statements:

- DML and DDL

DDL allows us to define/modify the relation/DB schemas.

To ***create a table***

```
CREATE TABLE name (list of elements);
```

elements => <u>attributes</u> and their <u>types</u> but declarations of <u>key</u> and <u>constraints</u> may also appear.

Ex:

```
CREATE TABLE Star (
name CHAR(30),
address VARCHAR(255),
genre CHAR(1),
birthdate DATE
);
```

**CHAR** vs **VARCHAR**: char is exactly 30 character when VARCHAR means it can go until 255 characters.

To ***delete a table from DB***:

```
DROP TABLE name;
```

Ex:

```
DROP TABLE Star;
```

***Data types***

**INT** or **INTEGER**
**REAL** or **FLOAT**
**DECIMAL**(n, d) – **NUMERIC**(n, d)
- Ex: `DECIMAL(6,2)`, e.g. 0123.45

**CHAR**(n)/**BIT**(B)
- fixed length character/bit string

**VARCHAR**(n)/**BITVARYING**(n)
- variable-length strings up to n characters

**SQL2 Syntax for:**
-- **Time**: format 'hh:mm:ss[.ss...]'
-- **Date**: format 'yyyy-mm-dd' (m =0 or 1)
- Example:
- `CREATE TABLE Days(d DATE);`
- `INSERT INTO Days VALUES('2012-12-23');`

*Note 1: In Oracle, the default format of date is 'dd-mon-yy',*
e.g., `INSERT INTO Days VALUES('22-jan-18');`

*Note 2: The Oracle function to date converts a specified format into default,*
e.g., `INSERT INTO Days VALUES (to_date('2018-01-22', 'yyyy-mm-dd'));`

***Altering Relation Schemas***

- Adding columns
  `ALTER TABLE R ADD <column declaration>;`
  Ex: Add attribute phone to table Star
  `ALTER TABLE Star ADD phone CHAR(16);`
- Removing Columns
  Remove an attribute from a relation R using DROP:
  `ALTER TABLE R DROP COLUMN <column_name>;`
  Ex: Remove column phone from Star
  `ALTER TABLE Star DROP COLUMN phone;`
  *Note: Can't drop a column, if it is the only column*

If you want to modify the schema:
But schema changes is less frequent than content change.

***Attribute Properties***

We can assert that value of attribute A to be:

- **NOT NULL**

- then every tuple must have a "real" value (not null) for this attribute

- **DEFAULT** value
  - Null is the default value for every attribute
  - however, you can define any value desired as default

Ex

```
CREATE TABLE Star(
    name CHAR(30),
    address VARCHAR(255),
    gender CHAR(1) DEFAULT '?',
    birthday DATE NOT NULL);

ALTER TABLE Star ADD phone CHAR(16) DEFAULT
'unlisted';
INSERT INTO Star(name, birthdate) VALUES ('SALLY',
'0000-00-00');
```

| name | address | gender | birthdate | phone |
|---|---|---|---|---|
| Sally | NULL | ? | 0000-00-00 | unlisted |

```
INSERT INTO Star(name, phone) VALUES ('SALLY', '333-
2255');
```

- this insertion fails since value for birthdate is not given, and null is not allowed.

To add default value after an attribute is defined:

```
ALTER TABLE Star ALTER phone SETDEFAULT 'no-phone';
```

In Oracle:

```
ALTER TABLE Star MODIFY phone DEFAULT 'no-phone';
```

---

## *New Chapter: Entity/relationship (E/R) Model*

### *DB Design*

*Requirements collection and analysis*

- Determine what information the database must hold (what is in the DB – what data is in the DB )
- Determine the *relationships* among the components of that information
  - **Conceptual database design** using some data model

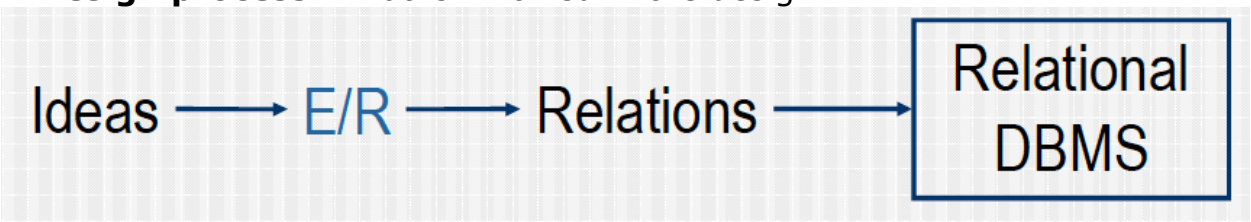
A **data model** is a **collection of concepts** *for describing*:

- 1. data and relationships among data
- 2. data semantics and constraints (ex the keys, fact that student ID and course ID in enrolment are connected)

***Design Approaches & Notations***

Entity-Relationship Model (E/R)
Object-Oriented (e.g., Object Definition Language -- ODL)
Semi-structured data (e.g., XML)

**A Design process**: what is involved in the design ?

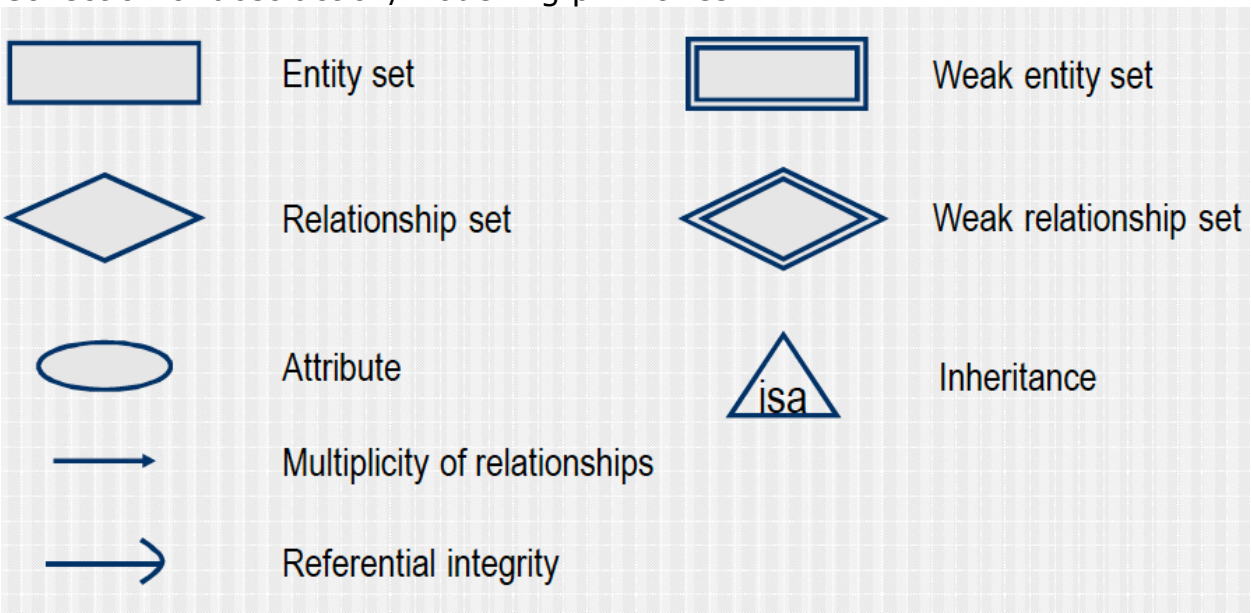



1) ideas 2) ideas into a high level concept 3) mysql relations 4) convert design into implementation

***E/R Model***

E/R model (Chen 1976) is a graphical approach to database modelling
It grew out of modelling applications
It is widely used in database design
There is no one standard for the E/R language/notation

Collection of abstraction/modelling primitives

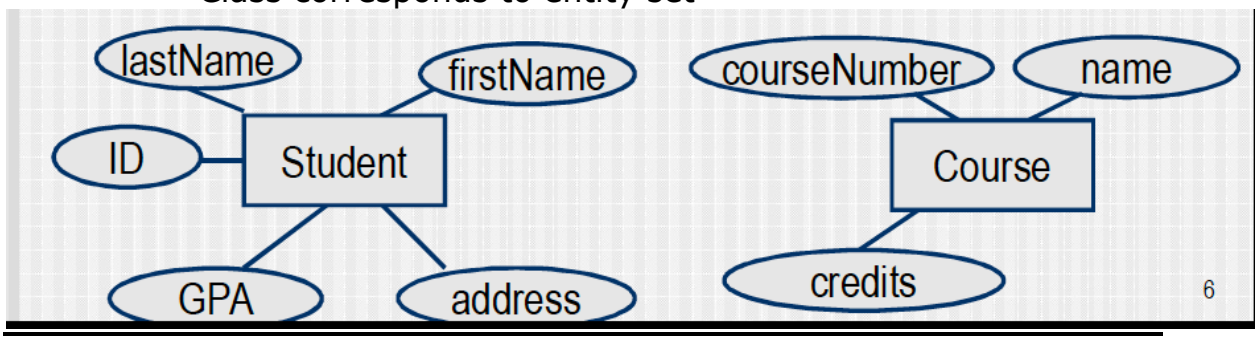



****ATTENTION TO ARROWS****

***Entities and Entity sets***

***Entity*** – A real world object that is "distinguishable" from other objects

***Entity Set*** -- A collection of similar entities
All entities in an entity set have the same set of attributes
In ODL:
Object corresponds to entity
Class corresponds to entity set

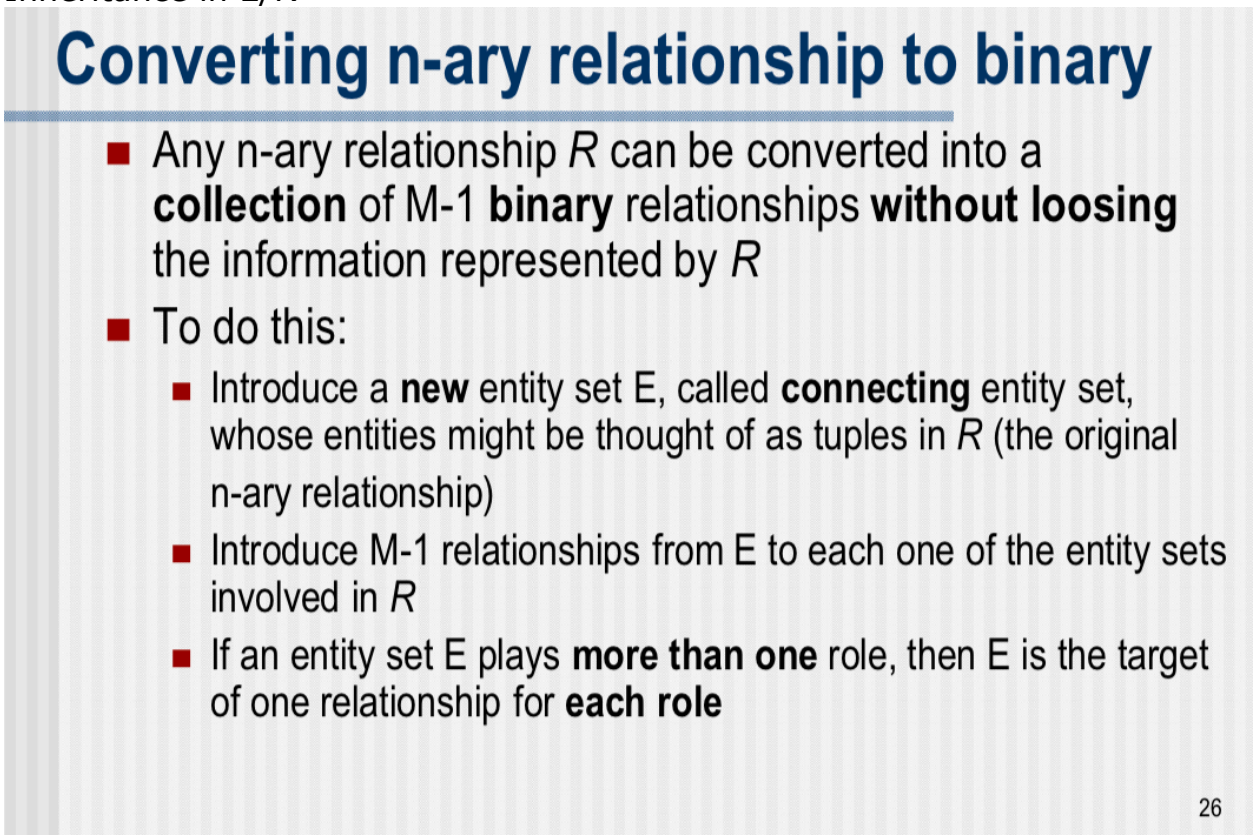


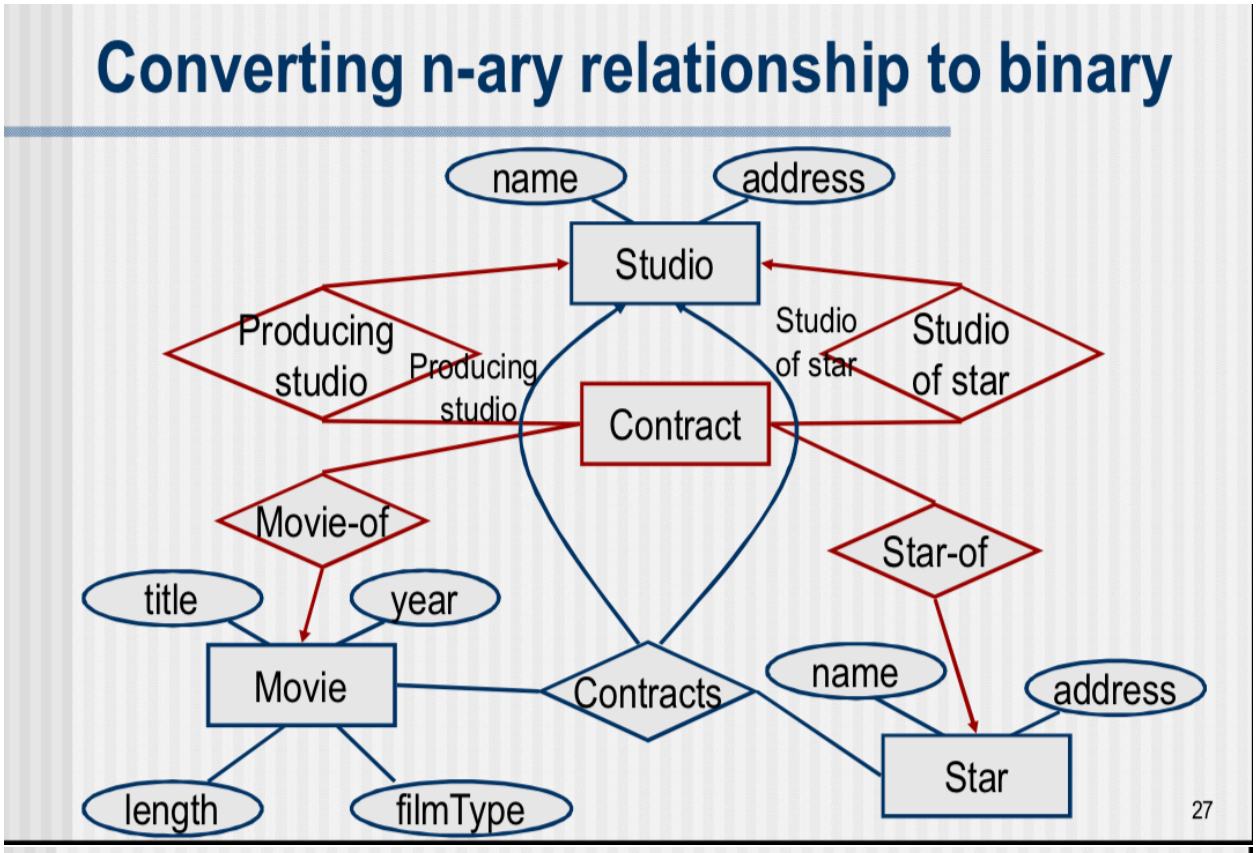

## January 31, 2018

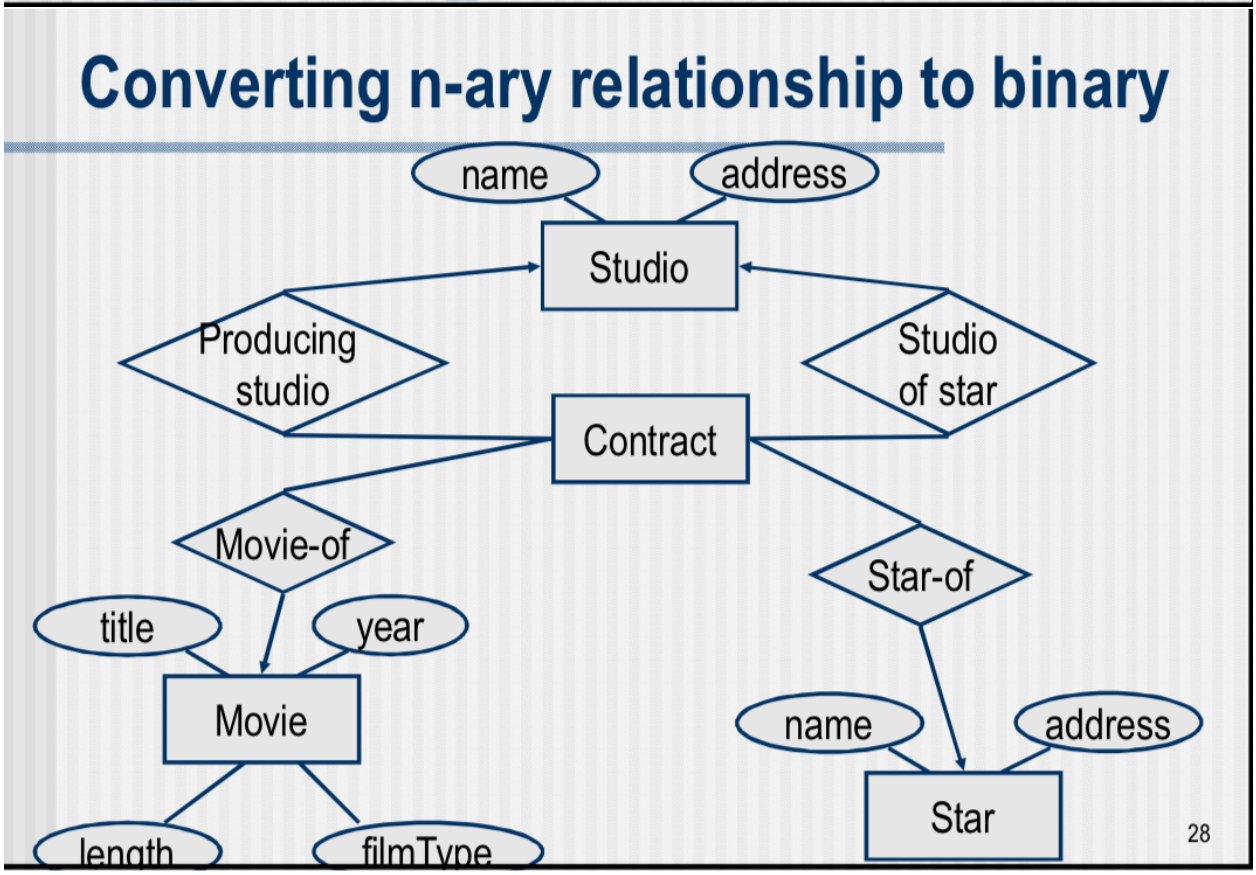
***Converting***
Inheritance in E/R

## February 5, 2018

***Converting***
Inheritance in E/R

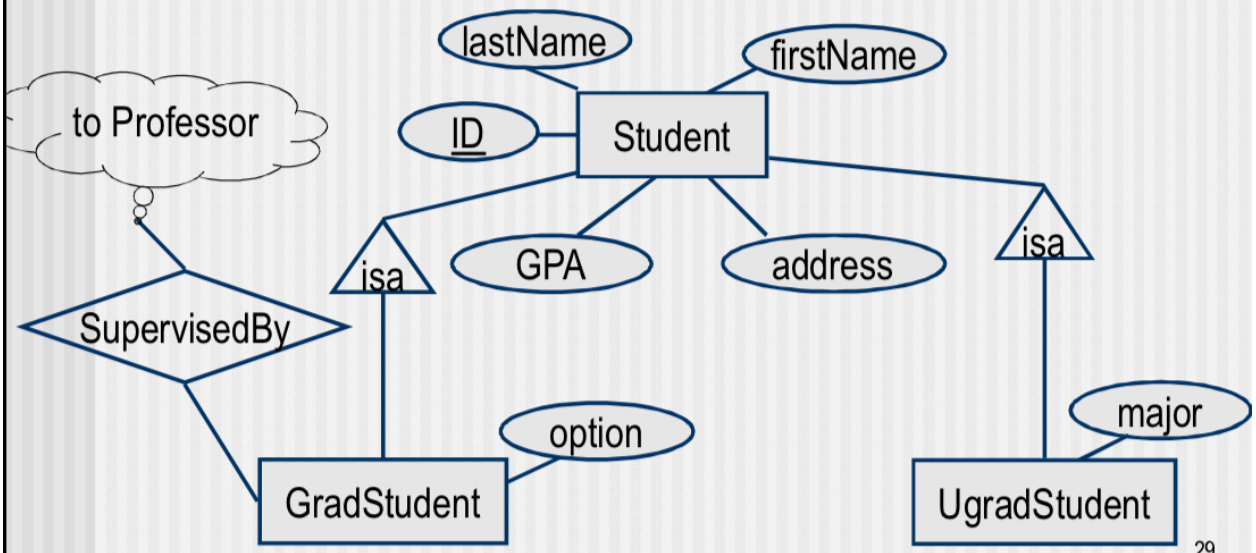



Converting n-ary relationship to binary
name
address
Studio
Producing studio
Producing studio
Studio of star
Studio of star
Contract
Movie-of
Star-of
title
year
Movie
Contracts
name
address
Star
length
filmType
27


Converting n-ary relationship to binary
name
address
Studio
Producing studio
Studio of star
Contract
Movie-of
Star-of
title
year
Movie
name
address
Star
length
filmType
28

# Inheritance in E/R

- Inheritance in E/R is expressed by ***isa*** relationship



# Inheritance in E/R

- There is a subtle difference between the concept of inheritance in an OO approach and in E/R
- In OO, an object must be a member of exactly one class
- In E/R
  - We consider an entity as having "components" belonging to several entity sets that are "part of" a single **isa**-hierarchy
  - The "components" are connected into a single entity by the **isa** relationships
  - The entity has whatever attributes any of its components has, and participates in whatever relationships its components participate in
    - ➔ **In an E/R diagram, we represent an entity set (e.g., CartoonMurderMystery) only if it has attributes and/or relationships of its own**

31

**Constraints**

There are some important aspects of the real world that cannot be represented using the ODL or E/R model introduced so far
The additional information about these aspects often takes the form of constraints on the data

**Classification of Constraints**

**Key** is a subset of Attributes
it's a set of attribute(s) that **uniquely** identifies a object or entity within its entity set

**Single-value constraints** are requirements that the value in a certain context/role be unique.

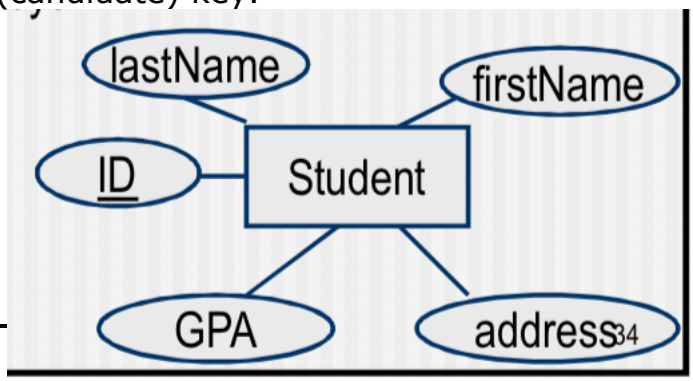
An entity (or object) may have at most one value for a given attribute or relationship.
Person: name, social-security number
Company: stock price
How do we do this in ODL?
In E/R, every attribute has at most one value.
Arrows tell us about multiplicity of relations.
If we have a single-valued constraint, we can either:
1. Require that the value exist (see referential integrity shortly)
2. Allow null values.

**Referential integrity constraints** are requirements that a value referred to by some object/entity must actually exist in the database

relationship has one value and the value must exist.
Example:
Product madeBy Company: company must exist.
How do we enforce referential integrity constraints? (otherwise, we get dangling pointers)
- forbid to delete a reference object, or
- delete the objects that reference an object we're deleting.
Company
Product
makes

**Domain constraints** require value of attribute must be drawn from specific set of values (attribute domain) or lies within specific range
**General constraints** arbitrary assertions that mush hold on the DB

**Keys**
**Superkey** is a set of attributes whose values uniquely identify an entity (object) in the entity set (class); this set may not be minimal.
**minimal superkey** is called a (candidate) key.




ID is key of **Student**

## February 7, 2018

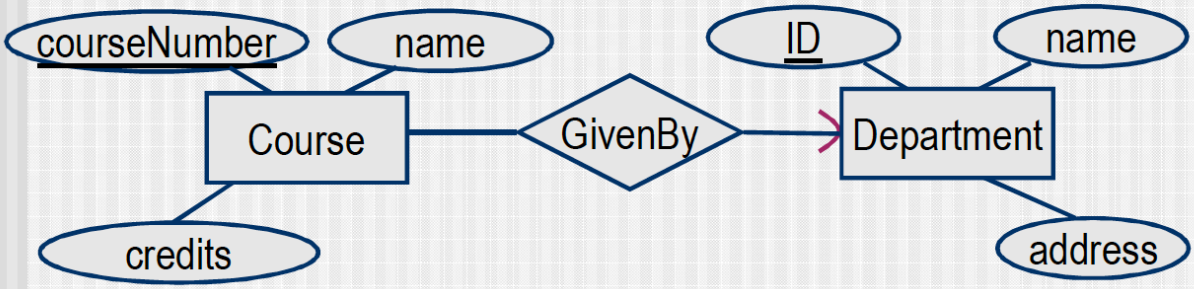
**Single-Value Constraints**
in E/R:
- attributes are atomic (first normal form, 1NF)
- an arrow can be used to express multiplicity
- what about multi-valued or structure in E/R?

Suppose A is a single-valued attribute.
Using the E/R model introduced so far, we can't:
- Express that the value for A must be present (e.g., when A is the key or part of a key), or
- Express that the value for A may be present (e.g., when A's value is optional, for which we use NULL)

If the choice is not explicit, we use the following "defaults":

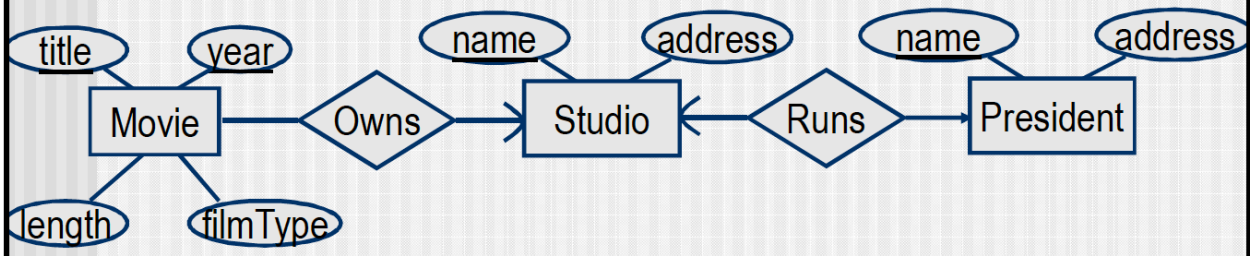
- The value for A must exist if A is part of the key
- The value for A is optional, otherwise

- For relationships:
  - **Single-value** + **Existence** = **Referential Integrity Constraint**
- We extend the **arrow** notation to indicate a reference is mandatory (to support referential integrity)




- This means, there is no course listed in the database unless there is a department giving the course.

42




- The studio for a movie must always be present in **Studio**
- If someone is a president, then the studio that he/she runs must exist in **Studio**
- However, the model allows studios without presidents (temporarily)

**Domain constraints**

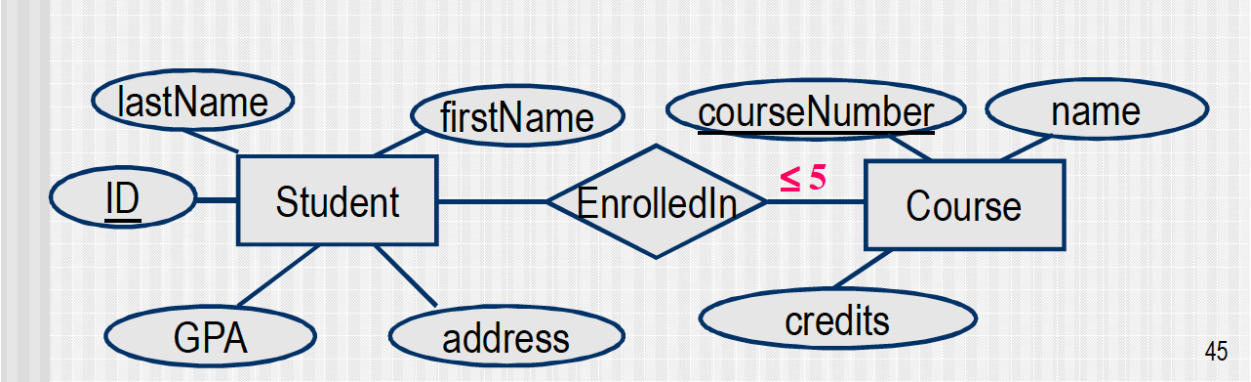
Domain constraints restrict **values of an attribute** to be drawn from a set

- E/R does not support imposing domain constraints

- ODL allows using types to limit/control possible values of the attributes
- ODL does not support restricting the "range" of values allowed for an attribute

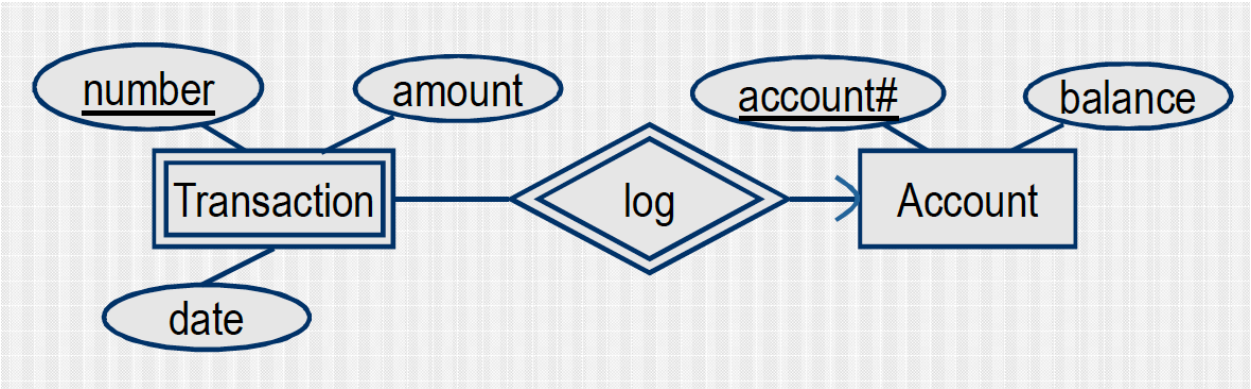
**Relationship degree constraints**
Relationship degree constraints: restrict the number of entities in the entity sets involved in a relationship.

*Ex*: can impose constraint that: *a student cannot be enrolled in more than 5 courses*
in E/R, we may attach a **bounding number** to corresponding link
in ODL, set of references or an **array** of size 5 (of reference type) will do



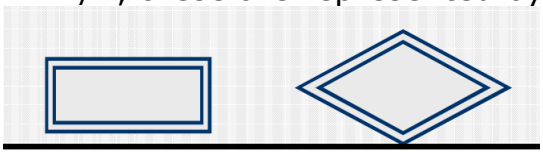

**Weak Entity / Relationship Sets:**
*Example*

- Log records the transactions initiated through an ATM
- Each transaction has a number, a date, and an amount
- Different accounts might have transactions by the same number, on the same date, and for the same amount
- Duplicate because we are not connecting account number: transaction # number is called a ***discriminator*** and the key is account#



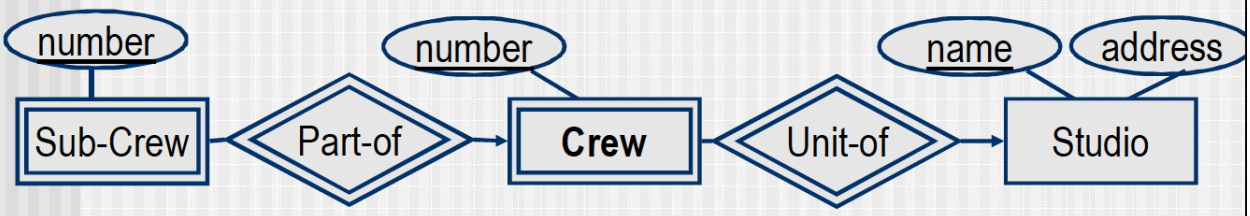

A **strong** entity set has a key
A **weak** entity set does ***not have sufficient attributes of its own to form a key***. It participates in a M-1 relationship (with no descriptive attributes) with a strong entity set
**Discriminator** of a weak entity set is a ***set of attributes that distinguishes among the entities corresponding to a strong entity***.
**key of a weak entity set** = key of the strong entity together with the discriminator of the weak entity

- In E/R, these are represented by:



# Sources of Weak Entity Sets

- Entity sets fall into a hierarchy
  - The entities in set **Crew** are sub-units of entities in set Studio
    - The "numbers" used for crew entities are not unique until we take into account the name of Studio with which she/he is associated (subordinate)
  - It is possible to have a chain of "weak" entity sets/relationships




# Sources of Weak Entity Sets

- Connecting entity sets (CES):
  - CES's often have no attributes of their own
  - Their key is formed from the key attributes of the entity sets which they connect
- A connecting entity set is always weak

# Design Principles

- Design should
  - Reflect the "reality" we are trying to model – faithful/realistic
  - Avoid redundancy -- minimal
    - Redundant information takes space
    - Could cause inconsistency
  - Be as simple as possible
- Be careful when choosing between using attributes and using classes or entity sets. Remember that:
  - An attribute is **simpler** to implement than a class/entity set or a relationship
  - If something has more information associated with it than just its name, it should probably be modeled as an entity set or a class



# A quick test!

- Which of the following statements is **NOT** correct?

  A. A data model is a collection of concepts for describing data and their relationships.

  B. A data model is a language for describing the data semantics and the constraints.

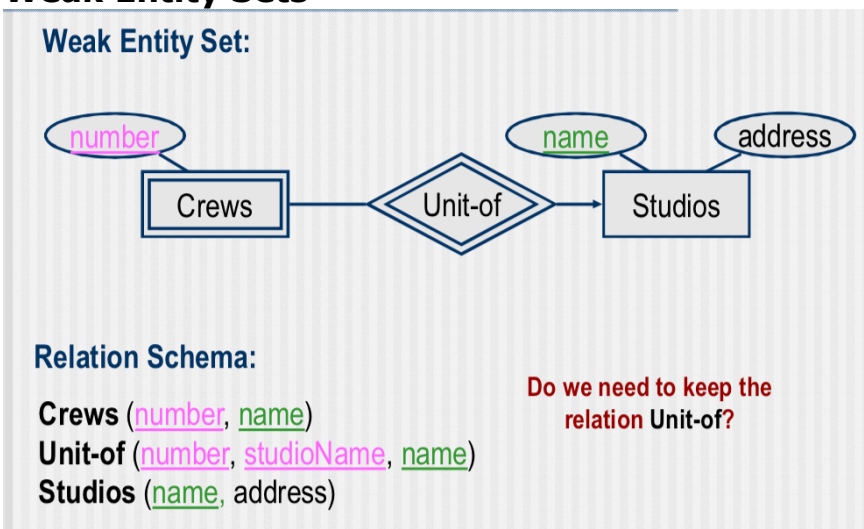
  C. A data model is a language for expressing queries and transactions over a database.

  D. A data model is an abstract representation of the structure of the data.

**C is not correct.**

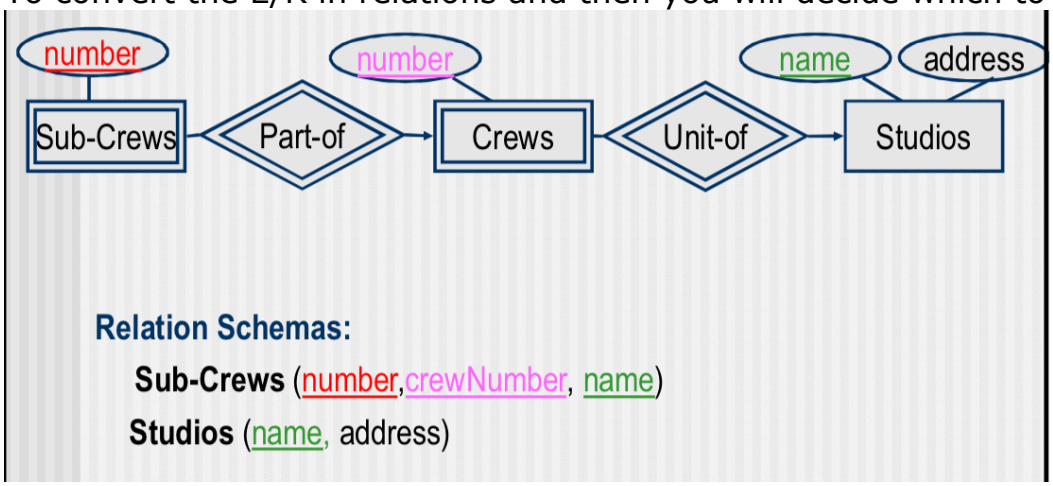
# February 12, 2018

## Weak Entity Sets




Need to get key attribute . Unit-of is not really needed as it will be two tables.

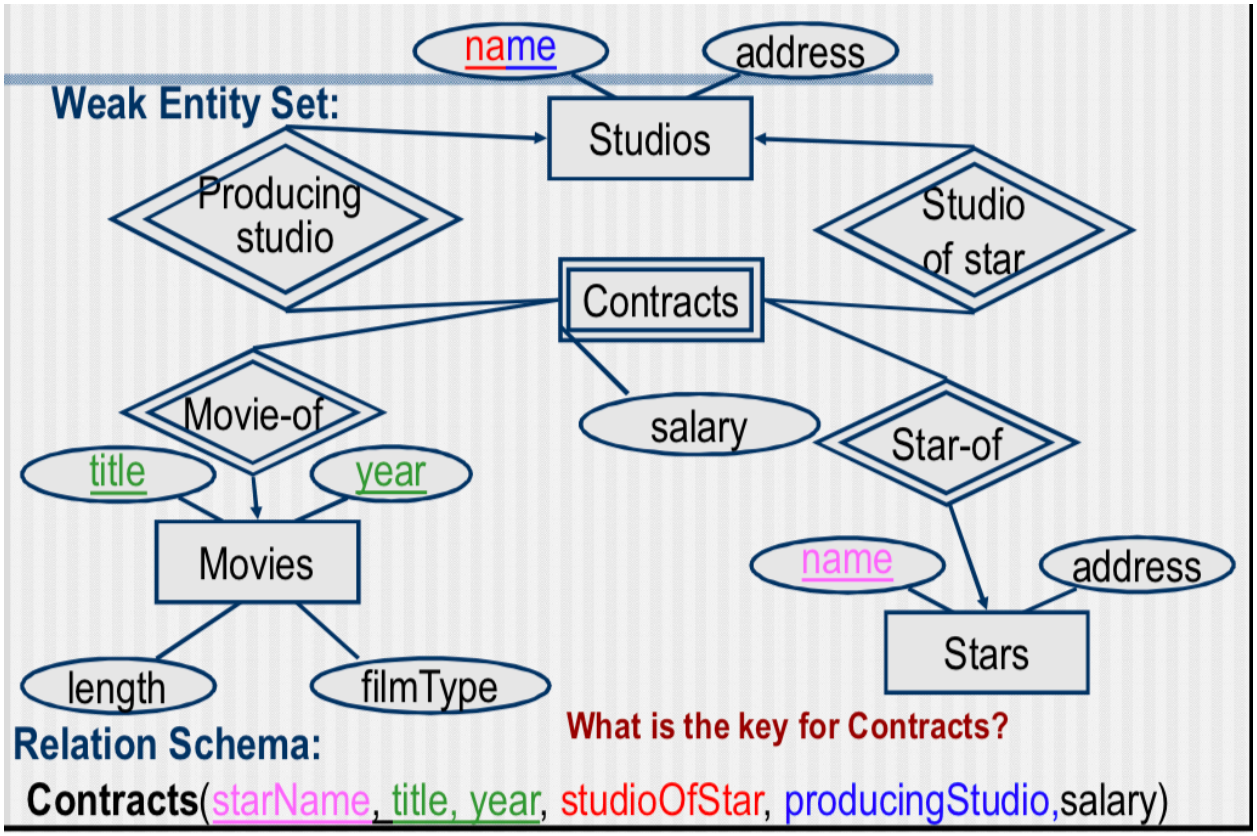
To convert the E/R in relations and then you will decide which to keep




***For this diagram how many tables do we get?***

4 tables: 1) movies 2) stars 3) studios 4) Contracts

*For Contracts*: key attributes: so we get the key from Stars and from Movies, and Studios= 5 attributes: Movie.title, Movie.year, Stars.name, salary, Studios.name



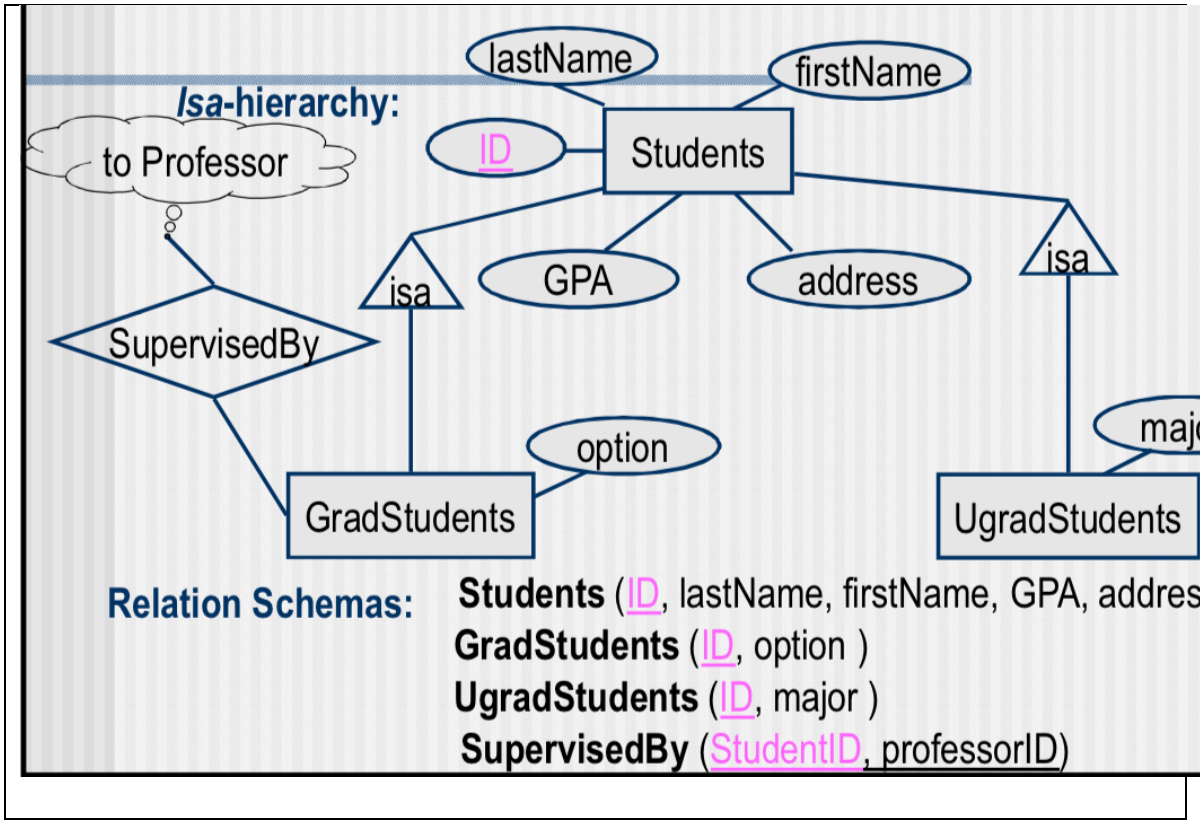

***Converting isa-Hierarchies to relations:***

***1. Straight-E/R style method***

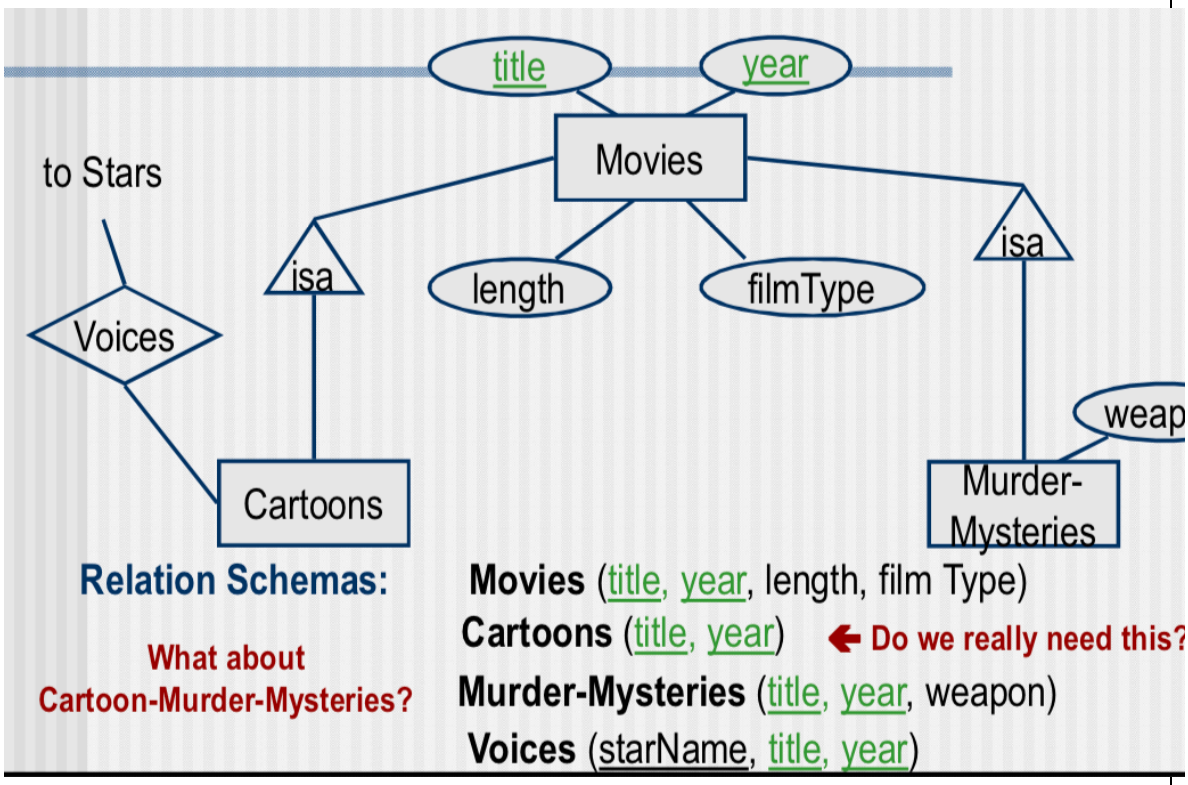
For each entity set ***E***, create a relation (table) ***e***, and give it attribute(s) ***A***, whenever:

- ***A*** belongs to ***E***
- ***A*** is the ***key attribute of the parent(s) relation***

No relation is created for the ***isa-relationship***




**2. object oriented method**

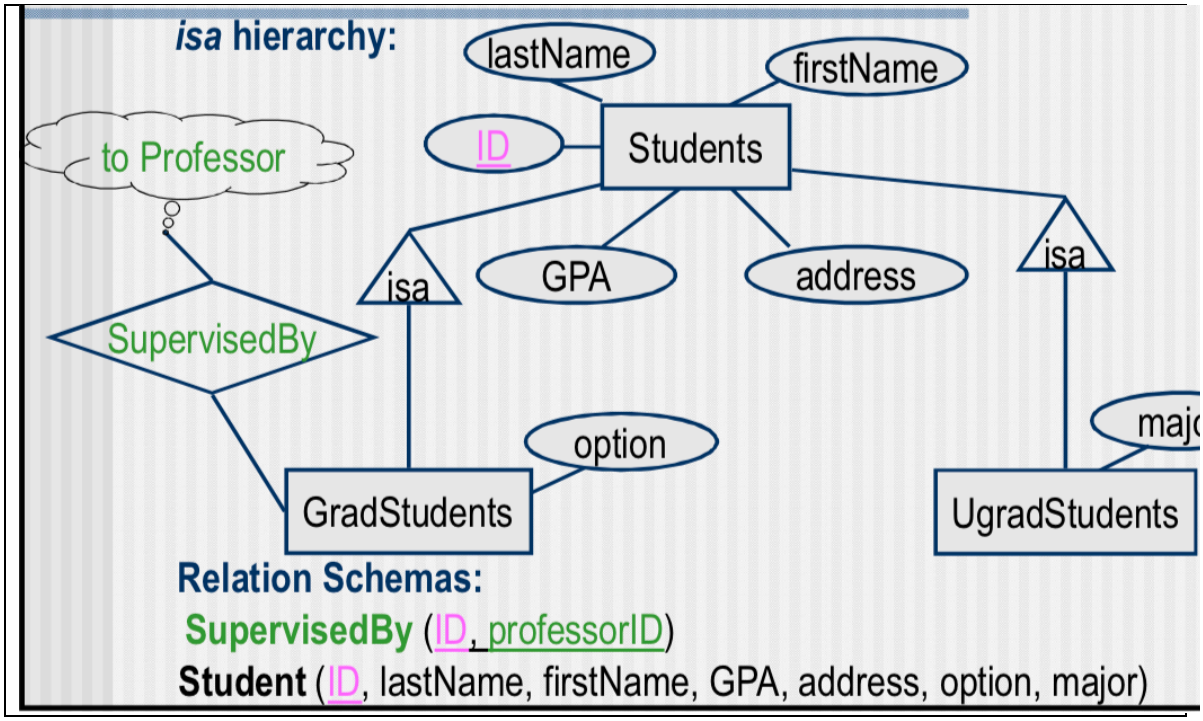



***3. nulls method***

If we are ***allowed*** to use ***NULL*** as a value in tuples, we can handle a hierarchy of entity sets (classes) with a ***single relation***

This relation has all the attributes belonging to any entity set(class) of the hierarchy.

An entity/object is represented by a single tuple that has NULL in each attribute that is not defined for that entity/object.




# A quick test!

- Suppose R is a M-1 relationship from entity set E1={a1,a2} to E2={b1,b2}. Which of the following is NOT a ***valid instance*** of R?
  - R = {(a1, b1), (a1, b2)}.
  - R = {(a1, b1)}.
  - R = {(a2, b1)}.
  - R = {(a1, b1), (a2, b1)}.
  - R = { }

***No null in this kind of this question.***

**Functional dependencies**

Functional dependency (FD): kind of constraint

- Suppose **R** is a relation schema and **X,Y** $\subseteq$ **R**.

  A FD on **R** is a statement of the form **X** $\rightarrow$ **Y**, which asser
  "For every "legal/valid" instance **r** of **R**, and for all pairs of tuples t1 and t2 in **r**, if t1 and t2 **agree** on the values in **X**, then t1 and t2 **agree** also on the values in **Y**."

  In symbols: $\forall$ t1,t2 $\in$ **r**: t1[**X**] = t2[**X**] $\Rightarrow$ t1[**Y**] = t2[**Y**].
- We read **X** $\rightarrow$ **Y** as:

  **X (functionally) determines Y (or Y is determined by X)**
- We say that the FD: **X** $\rightarrow$ **Y** is *relevant* to **R** if **X**U**Y** $\subseteq$ **R**.

So if you fix these attributes under these columns, then the tuple will be unique.
If a movie that the same title, they could have different feedback, different type and they would not be the same. So here the title doesn't imply the file type.
If X and Y are to subset of schema that you have and it is valid: it means you take any 2 tuples of any time, if these two tuples have the same attributes under these attributes they will have the same values under these two attributes too. You cannot have two tuples with the same name and year.

A up-side-down means, take any movies if they agree on X then they will agree on Y.

---

## February 14, 2018

***Trivial FD's***

- An FD **X** $\rightarrow$ **Y** is said to be **trivial** if **Y** $\subseteq$ **X**.
  - For example: {**title, year**} $\rightarrow$ **title** is a trivial FD
- Otherwise, the FD is called **nontrivial**
  - For example: {**title, year**} $\rightarrow$ **length** is a nontrivial FD

R = <A,B,C> R= <A,B> F != <A -> B>
Valid input:

| *A* | *B* | *C* |
|---|---|---|
| 1 | 2 | 3 |

Invalid input:

| *A* | *B* | *C* |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 4 | 5 |

R= <A,B> A -> ø

| *A* | *B* |
|---|---|
| 1 | 2 |
| 1 | 3 |

not a trivial FD because the right handsight is not a subset of the left handsight

R=<A,B> ø -> A

---

# February 28, 2018

## Equivalence of two sets of FD's

- Let **R** be a relation schema, and **S, T** be sets of FDs on **R.**
- Defn: we say **S** ***covers*** **T** (**S** $\models$**T**) if for every instance **r** of **R**, whenever **r** satisfies (every FD in) **S**, **r** also satisfies **T**.
- Defn: **T** and **S** are ***equivalent*** (**S** $\equiv$ **T**) iff **S** $\models$**T** and **T** $\models$ **S**.
- Note: **F** and **F**$^+$ are equivalent.

Example: Suppose **R** = {A,B,C}, and

**S** = {A $\rightarrow$ B, B $\rightarrow$ C, A $\rightarrow$ C}

**T** = {A $\rightarrow$ B, B $\rightarrow$ C}

We can show that **S** $\equiv$ **T**.

19

if you say a set is closed, what does it mean: the set of integers is set under the close.

<u>Closure of a set of FD's</u> means <u>set that includes every FD's that is implied in that FD's</u>

how are they related: unrelated in sense 2 sets A and B if A != B or related A subset of B. **S covers T (S |= T)** or **T and S are equivalent (S≡T)**
if S and T agree on A, they can agree on B and they can agree on C.

## Armstrong Axioms

- **R** is a relation schema, and **X**, **Y**, **Z** are subsets of **R**.
- **Reflexivity**
  - If **Y** ⊆ **X**, then **X** → **Y** (trivial FDs)
- **Augmentation**
  - If **X** → **Y**, then **XZ** → **YZ**, for every **Z**
- **Transitivity**
  - If **X** → **Y** and **Y** → **Z**, then **X** → **Z**
- These are **sound** and **complete** inference rules for FDs

<u>Reflexivity</u>: if Y is a subset of X then X → Y is trivial
AB → CDE
{A,B} → {B,C,D,E}
AB → A that is trivial
<u>Augmentation</u>: if X → Y, then XZ → YZ for every Z
A → B
AC → BC

## Additional Rules/axioms

Other useful rules that follow from Armstrong Axioms:

Suppose **X, Y, Z,** and **W** are sets of attributes.

- **Union (Combining) Rule**
  - If **X** → **Y** and **X** → **Z**, then **X** → **YZ**
- **Decomposition (Splitting) Rule**
  - If **X** → **YZ**, then **X** → **Y** and **X** → **Z**
- **Pseudotransitivity Rule**
  - If **X** → **Y** and **WY** → **Z**, then **XW** → **Z**

AC → BC

can I get this? : A → B **NO**
It means that AC controls B, AC → B and AC controls C, AC → C
You can add things to both sides but not remove because it won't be the same.
Decomposition: A → BCD It means A → B; A → C; A → D
Pseudotransitivity rule: X controls Y, WY controls Z then XW → Z

**Example: Discovering hidden FD's**

- Consider **R** = {**A**, **B**, **C**, **G**, **H**, **I**} with the FDs:

  **F** = { **A** → **B**, **A** → **C**, **CG** → **H**, **CG** → **I**, **B** → **H** }
- Using Armstrong's rules, we can derive more FDs
  - Since **A** → **B** and **B** → **H**, then **A** → **H**, by transitivity
  - Since **CG** → **H** and **CG** → **I**, then **CG** → **HI**, by union
  - Since **A** → **C** then **AG** → **CG**, by augmentation

    Now, since **AG** → **CG** and **CG** → **I**, then **AG** → **I**, by transitivity (and in a similar way, we get **F** ⊨ **AG** → **H**)
  - Many trivial dependencies can be derived(!) by augmentation

**Implication problem**

- Given a set **F** of FDs, does **X** → **Y** follow from **F**?
- In other words: is **X** → **Y** in the closure of **F**?

  (In symbols, does **F** ⊨ **X**→**Y** hold, or is

  **X**→**Y** ∈ **F**$^+$ true?
- How to answer this question?
  - Compute the closure of **F** & check if it includes **X**→**Y**
- What is the problem with this approach?
  - Computing **F**$^+$ is *expensive*! Is there a better solution?

**EXAM QUESTION: Is X → Y in close of F?**

## Closure of a Set of Attributes

- Given <**R**, **F**>; Let **X** $\subseteq$ **R**.

  The closure of **X** under **F** is the set of all attributes **Y** in **R** that are determined by **X**. This yields **X** $\rightarrow$ **Y**, i.e., every valid instance of **R** (that satisfies **F**) also satisfies **X** $\rightarrow$ **Y**
  - We denote the closure of a set of attributes **X** under **F** by $X^+_F$
    - When **F** is known , we simply write $X^+$ (and omit **F**)
    - Closure of $\{A1, A2, \ldots, A_n\}$ is denoted $\{A1, A2, \ldots, A_n\}^+$
  - Note that **X** $\subseteq$ $X^+$, for any set **X** of attributes (because **X** $\rightarrow$ **X**)

## Computing the Closure of Attributes

- Given a set **F** of FD's and a set **X** of attributes, how to compute the closure of **X** w.r.t. **F**?
  - Starting with set $X^+=X$, we repeatedly expand $X^+$ by adding the RHS Z for every FD: **W**$\rightarrow$**Z** in **F**, if the LHD **W** is already in $X^+$.
  - This process terminates when $X^+$ could not be expanded further.
  - This process is expressed as an algorithm in the next slide.

### An Algorithm to Compute X+ under F

$X^+ \leftarrow X$ (initialization step)

*repeat*

  *for each* FD **W** $\rightarrow$ **Z** in **F** *do*:

    *if* **W** $\subseteq$ $X^+$ *then*

      $X^+ \leftarrow X^+ \cup Z$ // add **Z** to the result

*until* $X^+$ does not change

**Complexity?** In the worst case, how many times the "repeat" statement may be executed?

## Examples

- Consider a relation schema **R** = { **A**, **B**, **C**, **D**, **E**, **H** } with the FD's **F** = { **AB** → **C**, **BC** → **AD**, **D** → **E**, **CH** → **B** }
- Suppose **X**={**A**,**B**}. Compute **X**$^+$
- Execution result at each iteration:
  - Initially, **X**$^+$ = {**A**, **B**}
  - Using **AB** → **C**, we get **X**$^+$ = {**A**, **B**, **C**}
  - Using **BC** → **AD**, we get **X**$^+$ = {**A**, **B**, **C**, **D**}
  - Using **D** → **E**, we get **X**$^+$ = {**A**, **B**, **C**,**D**, **E**}
  - No more change to **X**$^+$ is possible.
    - ➔ **X**$^+$ = {**A**, **B**}$^+$ = {**A**, **B**, **C**, **D**, **E**}

Does the order in which FD's appear in F affect the computation?
It does not. It only affects the number of iterations.

## Implication Problem revisited

- Given a set of FD's **F**, does an FD: **X** → **Y** follow from **F**?
  - That is, is FD **X** → **Y** in **F**$^+$?
- To answer this, we can compute **X**$^+$ under **F**, and check if **Y** is in **X**$^+$ or not
  - If yes, then the answer is positive! (**F** $\models$ **X** → **Y** ☺)
  - Otherwise, it is negative (**F** $\not\models$ **X** → **Y** ☹)

## Examples

- Consider <R,F> where R = { A, B, C, D, E, H } and
  F = { AB → C, BC → AD, D → E, CH → B }
- Does AB → D follow from F?
- Two steps:
  1. Compute $\{A, B\}^+ = \{A, B, C, D, E\}$
  2. Check if $D \in \{A,B\}^+$
  - So, here we conclude that AB → D is implied by F

- Consider a relation schema R = { A, B, C, D, E, H } with FDs: F = { AB → C, BC → AD, D → E, CH → B }
- True/False: Does D → A follow from F?
- Two steps:
  1. Compute $X^+ = \{D\}^+ = \{D, E\}$
  2. Check if $A \in X^+$
  - Since $A \notin \{D, E\}$, the answer is NO, i.e., $F \not\models D \rightarrow A$

D+ = {D, E} is D → A implied by D+, <u>NO</u>. So D does not control A.

## Closure and Keys

- Consider a case where $X^+$ includes **all** the attributes of a relation R
  → Clearly, X is a (super) **key** of R
- → To check if X is a candidate key of R, we should check 2 things:
  1. If $X^+$ is a superkey R, i.e., when $X^+ = R$, and
  2. If **no** proper subset of X is a key, i.e., $\forall A \in X: (X-\{A\})^+ \neq R$
- To find the keys of a relation, we can use the algorithm on slide 26
- This would be exponential in the number of attributes! Can do better?
- Knowledge about keys is essential to understand "Normal forms."

If X+ include all attributes then X is a <u>superkey</u>.
What is the complexity of this? 2^n.

(DB5)

# Schema Refinement: Minimal Bases (Canonical Covers)

## Bases

- Note that FD's on a relation may be represented in different but equivalent ways.
- Recall that, given two sets of FD's **F** and **G** on **R,** we say that: "**G** *follows from* **F** (**F** ⊨ **G**)", provided for every instance **r** of **R,** if **r** satisfies **F**, then **r** satisfies **G**. In this case, we may also say: "**F** *implies* **G**", or "**F** covers **G**", *or* "**G** *is implied by* **F**".
- If **F** ⊨ **G** and **G** ⊨ **F** both hold, then we say that **G** and **F** are equivalent and denote this by **F** ≡ **G.** In this case, **we may** also say that **F** and **G** are covers for each other.
- Note that **F** ≡ **G** iff $F^+ \equiv G^+$

if two set are equivalent their closure will be equivalent.

## Canonical Cover

- Let **F** be a set of FD's. A canonical cover of **F** is a set **G** of FD's that satisfies the following conditions:
  1. **G** is equivalent to **F**, that is, **G** ≡ **F**
  2. Every FD in **G** is has a single attribute on the right hand side.
  3. **G** is minimal, that is, if we obtain a set **H** of FD's from **G** by deleting some FD's in **G** or by reducing the left hand side of some FD's, then **H** won't be equivalent to **F** (that is, **H** ≢ **F**)

➔ A canonical cover **G** is minimal in two respects:

1. Every FD in **G** is "required" in order for **G** to be equivalent to **F**
2. Every FD in **G** is as "small" as possible, that is,
   - each attribute on the left hand side **X** is necessary.
   - Recall: the RHS of every FD in **G** is a single attribute

## Computing Canonical Cover

Given a set **F** of FD's, how to compute *a* canonical cover **G** of **F**?

- **Step 1:** Put the FD's in the **simple** form (i.e., one attribute on the RHS)
  - Initialize **G := F**
  - Replace each FD $X \longrightarrow A_1A_2 \ldots A_k$ in **G** with $X \longrightarrow A_1$, $X \longrightarrow A2$, ..., $X \longrightarrow A_k$
- **Step 2: Minimize** the left hand side **X** of every FD
  - E.g., if $AB \longrightarrow C$ is in **G**, check if A or B on the LHS is redundant , i.e.,

    $(G - \{AB \rightarrow C\} \cup \{A \rightarrow C\})^+ \equiv F^+$ or

    $(G - \{AB \rightarrow C\} \cup \{B \rightarrow C\})^+ \equiv F^+$ ?
- **Step 3: Delete** redundant FD's, if any
  - For each FD $X \longrightarrow A$ in **G**, check if $(G - \{X \rightarrow A\})^+ \equiv F^+$?

6

## Example:

- R = { A, B, C, D, E, H}
- F = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
- Step 1 – put FD's in the simple form
  - All present FD's are simple

➔ G = {A→B, DE→A, BC → E, AC → E, BCD → A, AED → B}

- R = { A, B, C, D, E, H }
- F = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
- Step 2 – Check every FD to see if it is left reduced
  - For every FD **X → A** in **G,** check if the closure of a subset of **X** determines **A**. If so, remove the redundant attribute(s) from **X**

- R = { A, B, C, D, E, H }
- F = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
- G = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
  - A → B
    ➔ obviously OK (no left redundancy)
  - DE → A
    - $D^+ = D$
    - $E^+ = E$

    ➔ OK (no left redundancy)

9

- R = { A, B, C, D, E, H }
- F = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
- G = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
  - BC → E
    - $B^+ = B$
    - $C^+ = C$

    ➔ OK (no left redundancy)

- R = { A, B, C, D, E, H }
- F = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
- G = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
  - AC → E
    - $A^+$ = AB
    - $C^+$ = C

      ➔ OK (no left redundancy)

- R = { A, B, C, D, E, H }
- F = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
- G = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
  - BCD → A
    - $B^+$ = B
    - $C^+$ = C
    - $D^+$ = D
    - $\{BC\}^+$ = BCE
    - $\{CD\}^+$ = CD
    - $\{BD\}^+$ = BD

    ➔OK (no left redundancy)

- R = { A, B, C, D, E, H }
- F = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
- G = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
  - AED → B
    - $A^+$ = AB
    - E & D are redundant
      ➔ we can remove them from AED → B
  - G = { ~~A → B~~, DE → A, BC → E, AC → E, BCD → A, A → B }

➔ G = { DE → A, BC → E, AC → E, BCD → A, A → B }

13

- R = { A, B, C, D, E, H}
- F = { A → B, DE → A, BC → E, AC → E, BCD → A, AED → B }
- Step 3 – Find and remove redundant FDs
  - For every FD **X → A** in **G**
    - Remove **X → A** from **G**; call the result **G'**
    - Compute $X^+$ under **G'**
    - If **A ∈ $X^+$**, then **X → A** is redundant and hence we remove the FD **X → A** from **G** (that is, we rename **G'** to **G**)

## March 5, 2018

A → BE
BD → E
B → C
E → A
*<u>Find all the key (minimal)</u>*
AD+={A,D,B,E,C} candidate key
BD+={B,D,E,A,C} candidate key
CD+={C,D, }

DE+ = {D,E,A} candidate key

**Computing Canonical Cover**

## March 7, 2018

# Decomposition

- A **decomposition** of a relation schema **R** consists of replacing **R** by two or more non-empty relation schemas such that each one is a subset of **R** and together they include all attributes of **R.** Formally, $\underline{R} = \{R_1,\ldots,R_m\}$ is a decomposition if all conditions below hold:
  1. $R_i \neq \emptyset$, for all i in {1,…,m}
  2. $R_1 \cup \ldots \cup R_m = R$
  3. $R_i \neq R_j$, when $i \neq j$
  - When **m = 2**, the decomposition $\underline{R} = \{R_1, R_2\}$ is called ***binary***
- Not every decomposition of R is "desirable". Why?
- Properties of a decomposition?
  (1) Lossless-join – this is a **must**
  (2) Dependency-preserving – this is **desirable**

Explanation follows …

25

Example

**Relation Instance:**

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 2 | 5 |

**Decomposed into:**

| A | B |
|---|---|
| 1 | 2 |
| 4 | 2 |

| B | C |
|---|---|
| 2 | 3 |
| 2 | 5 |

**To "recover" information, we join the relations:**

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 2 | 5 |
| 4 | 2 | 3 |
| 1 | 2 | 5 |

**Why do we got new tuples?**

When we **join** the two decomposed tables:
Now you get 4 tuples! The red ones does not exist and we do not want this to happen.
**THIS IS A BAD EXAMPLE OF DECOMPOSITION**: we lost something: the semantic, the correctness is lost. Not every decomposition is desirable.
This example is a **lossy decomposition**.
We want operation to be **lossless –join** (we do not want to loss correctness)

## Lossless Join Decomposition

- **R** is a relation schema and **F** is a set of FD's over **R**.

  A binary decomposition of **R** into relation schemas $\mathbf{R_1}$ and $\mathbf{R_2}$ with attribute sets **X** and **Y** is said to be a **lossless-join decomposition with respect to F**, if for every instance ***r*** of **R** that satisfies **F**, we have $\pi_X(r) \bowtie \pi_Y(r) = r$
- **Thm**: Let **R** be a relation schema and **F** a set of FD's on **R**. A binary decomposition of **R** into $\mathbf{R_1}$ and $\mathbf{R_2}$ with attribute sets **X** and **Y** is lossless **if** $X \cap Y \rightarrow X$ or $X \cap Y \rightarrow Y$, i.e., this binary decomposition is lossless if the common attributes of **X** and **Y** form a key of $\mathbf{R_1}$ or $\mathbf{R_2}$

27

Example:

**Relation Instance:**

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 2 | 3 |

**Decomposed into:**

| A | B |
|---|---|
| 1 | 2 |
| 4 | 2 |

| B | C |
|---|---|
| 2 | 3 |

$F = \{ B \rightarrow C \}$

**To recover the original relation r, we join the two relations:**

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 2 | 3 |

**No new tuples !**

If **common attribute B is key to first relation or second relation** (one of them) => allows multiple matches **but B being a key will assure** that it's value can only match with one other tuple => **uniqueness**.

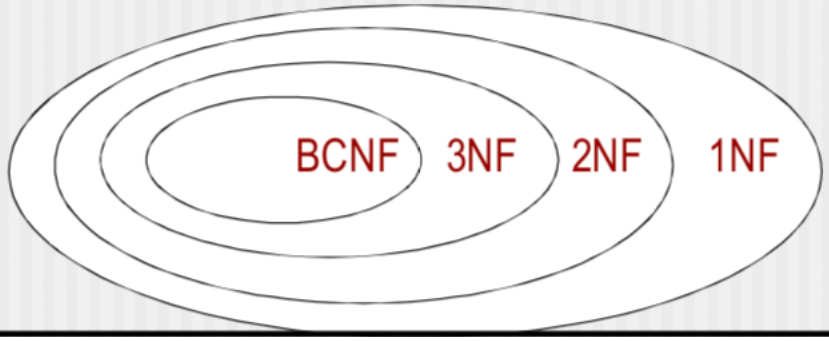
B controls C, so if you have table B and C, B is the key. Under B you'll not see repeated value, so when you join two tables, they will be based on B

New Chapter

**Normal Forms**

- If a relation schema is in a normal forms, we know that it is in some particular shape/health in the sense that *certain kinds of problems (related to redundancy) cannot arise*
- Given a relation schema **R**, we need to be able to check if it is in certain normal form. If not, we need to be able to decompose it into smaller such normal relations. How?
- To address these issues, we need to study **normal forms**

- The normal forms as defined and captured by FD's:
  - First normal form (1NF)
  - Second normal form (2NF)
  - √ Third normal form (3NF)
  - √ Boyce-Codd normal form (BCNF)
- These normal forms form the hierarchy:




**3NF:**

Given: A relation schema **R** with a set of FD's **F** on **R.**

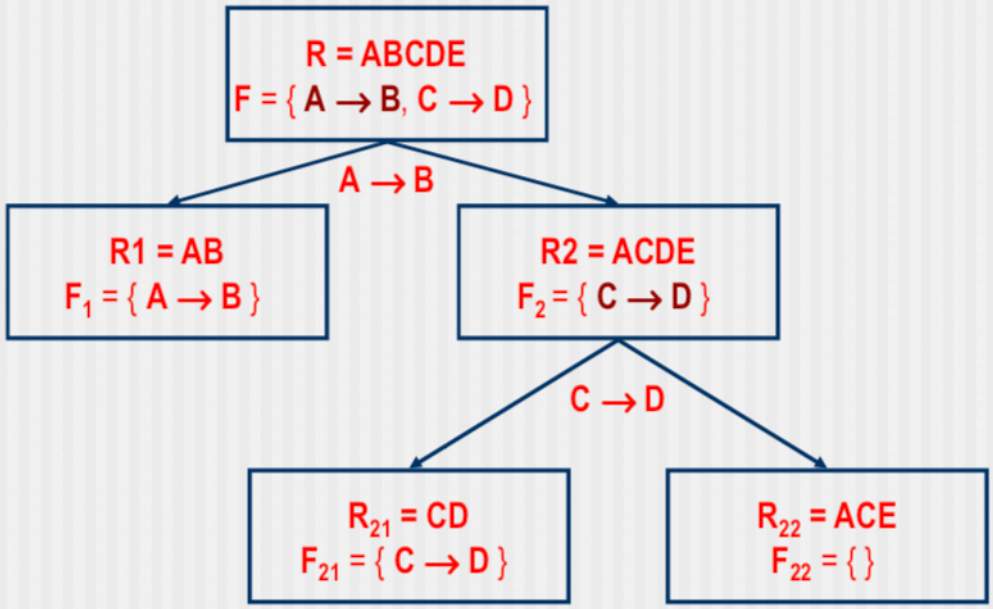
- We say **R** w.r.t. **F** is in 3NF (**third normal form**), if for every FD **X** → **A** in **F**, at least one of the following conditions holds:
  - **X** → **A** is a trivial, i.e., **A** ∈ **X,** or
  - **X** is a superkey, or
  - If **X** is not a key, then **A** is part of some key of **R**

➔ To determine if **R** with FD **F** is in 3NF:

- Check if the LHS of each nontrivial FD in **F** is a superkey
- If not, check if its RHS is part of any key of **R**

3

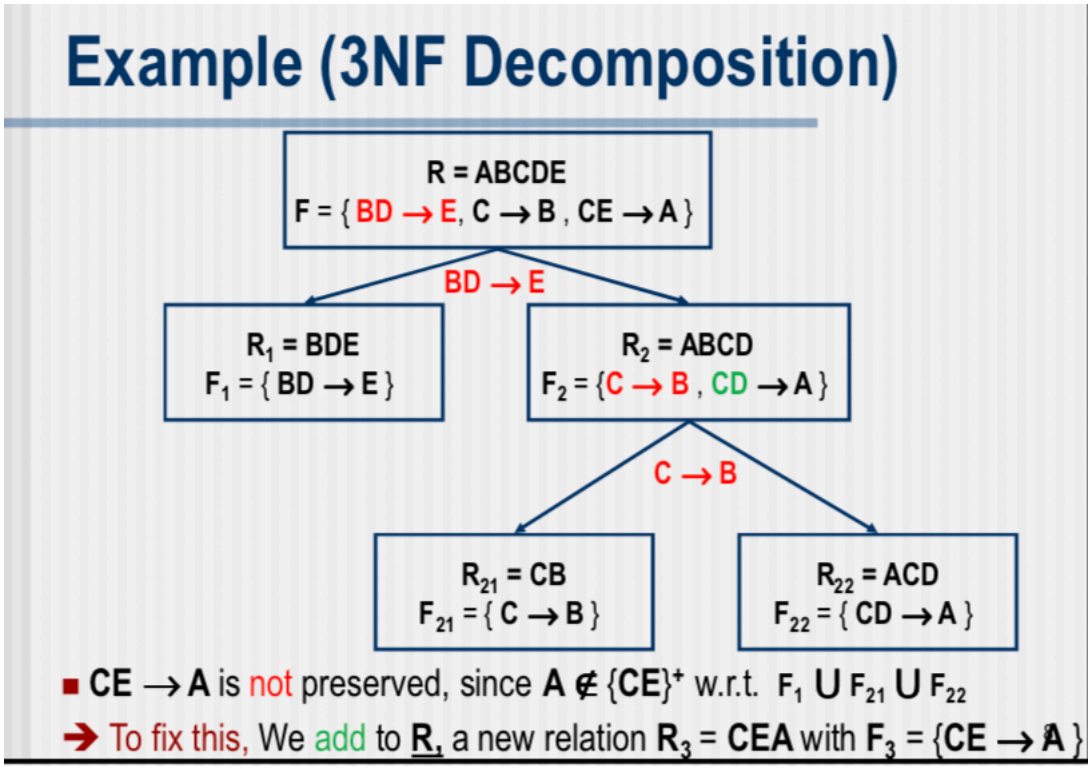
# Example (Decomposition into BCNF relations)



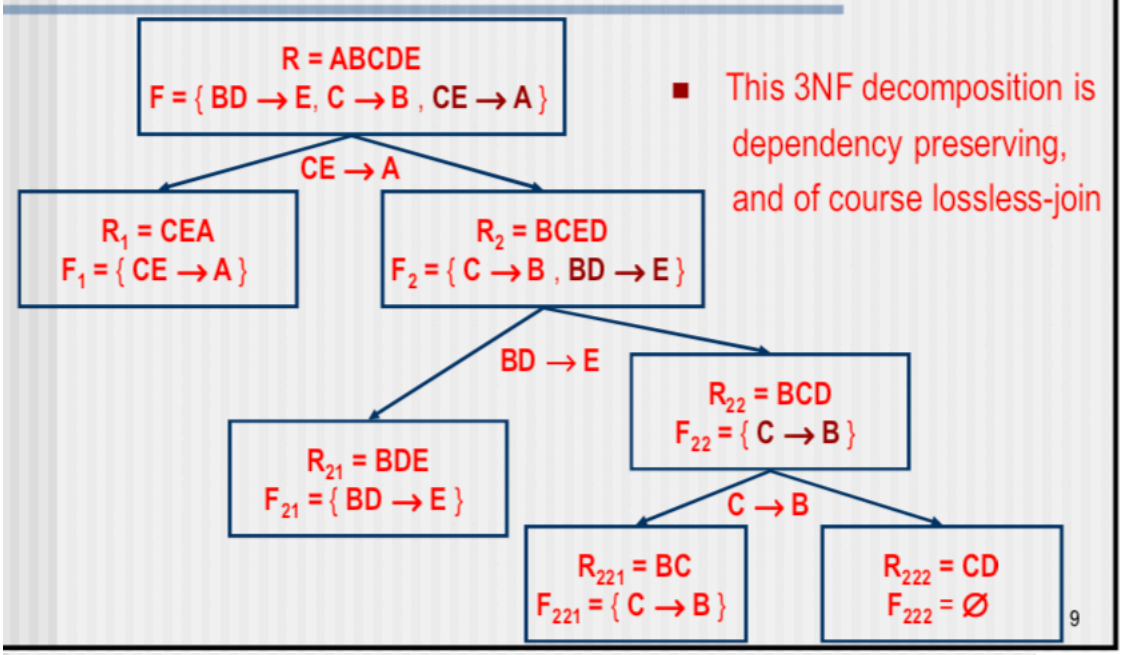

ACE is the key.
A→B is A is not a super key => violation this is a FD violated.
The order you choose the FD violation, it can change your answer.

R1 is a super key for AB.
R2: no BCNF because C+ gives you CD, it is violating FD.
So you get R21 = CD which gives you super key which is BCNF normal form
R22= ACE which means F() is empty => it's **trivially BCNF**
If there is a violation then not BCNF, if no violation, then BCNF.

## March 12, 2018



## Example (using a *different order*)




- This 3NF decomposition is dependency preserving, and of course lossless-join

9

## Decomposition into 3NF

- 1ST approach (binary decomposition):
  - Lossless-join √
  - May not be dependency preserving. If so, then add extra relations **XA**, for every FD **X** → **A** we lost
- **Approach 2:** the *synthesis* approcah
  - Dependency preservation √
  - However, may not be lossless-join. If so, we must add to **R**, one extra relation that includes whose attributes form a key of **R**

    What would be the FDs on this newly added relation?

## An Algorithm to Check Lossless join

Suppose relation $R\{A_1, \ldots, A_k\}$ is decomposed into $R_1, \ldots, R_n$
To determine if this decomposition is lossless, we use a table,
$L[1 \ldots n][1 \ldots k]$

**Initializing the table:**

**for** each relation $R_i$ **do**
  **for** each attribute $A_j$ **do**
    **if** $A_j$ is an attribute in $R_i$
      **then** $L[i][j] \leftarrow a_j$
      **else** $L[i][j] \leftarrow b_{ij}$

If this decomposition is lossy, you can convert what you have into a counter example.

| **R/A** | **A1** | **A2** | **…** | **Ak-1** | **Ak** |
|---|---|---|---|---|---|
| **R1** | a1 | a2 | b1 | ak-1 | b2 |
| **R2** | b3 | a2 | a3 | b4 | ak |
| **…** | a1 | b25 | | | |
| **Rn** | | | | | |

## Algorithm to Check Lossless (cont'd)

**repeat**
  **for each** FD $X \rightarrow Y$ in **F do:**
    **if** ∃ rows i and j such that $L[i] == L[j]$, for each attribute in **X**,
      **then** for ∀ column $t$ corresponding to an attribute $A_t$ in **Y do:**
        **if** $L[i][t] == a_t$
          **then** $L[j][t] \leftarrow a_t$
        **else if** $L[j][t] == a_t$
          **then** $L[i][t] \leftarrow a_t$
        **else** $L[j][t] \leftarrow L[i][t]$
**until** *no change*

The decomposition is lossless if, after performing this algorithm, L contains a row of all a's. That is, if there exists a row $i$ in L such that: $L[i][j] == a_j$ for every column $j$ corresponding to each attribute $A_j$ in **R**

## Examples

- Given <R,F>, where **R** = ( **A**, **B**, **C**, **D** ), and **F** = { **A** → **B**, **A** → **C**, **C** → **D** } is a set of FD's on **R**
- Is the decomposition **R** = {$R_1$, $R_2$} lossless, where $R_1$ = ( **A**, **B**, **C** ) and $R_2$ = ( **C** , **D**)?
  - To be discussed in class
- Now consider **S** = ( **A**, **B**, **C**, **D**, **E** ) and the set G of FD's on S, where **G** = { **AB** → **CD**, **A** → **E**, **C** → **D** }
- Is decomposition of **S** = {$S_1$, $S_2$, $S_3$} lossless, where $S_1$ = ( **A**, **B**, **C** ), $S_2$ = ( **B**, **C**, **D** ), and $S_3$ = ( **C**, **D**, **E** )?
  - To be discussed in class

15

| **R/A** | **A** | **B** | **C** | **D** |
|---|---|---|---|---|
| **ABC** | a1 | a2 | a3 | b1 a4 |
| **CD** | b2 | b3 | a3 | a4 |
| | | | | |

A controls b. A controls C. C controls D.
If these 2 tuples agrees on C then they agree on D. ABC b1 becomes a4

| S/S | A | B | C | D | E |
|---|---|---|---|---|---|
| **ABC** | a1 | a2 | a3 | ~~1~~ a4 | 2 |
| **BCD** | 3 | a2 | a3 | a4 | 4 |
| **CDE** | 5 | 6 | a3 | a4 | a5 |
| | | | | | |

2 tuples with same a and b values? No.
2 tuples with c values? yes, so those 2 tuples must have same D values.

## March 14, 2018

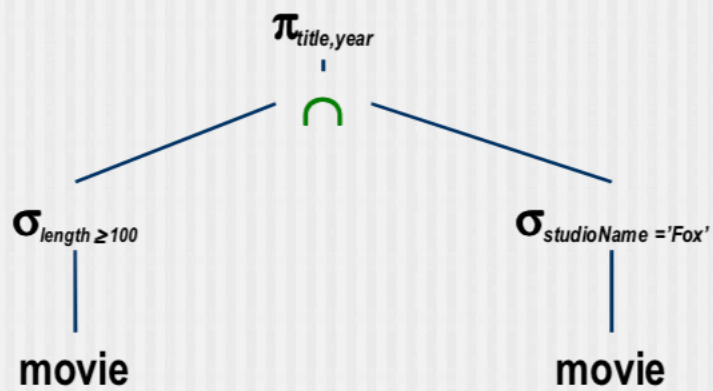
Missed 45min of the class all slides of DB7 before this one:

### Example: RA Query

- Relation schema:
  **Movie** (title, year, length, filmType, studioName)
- Query: List the *title* and *year* of every movie made by **Fox** studio whose length is at least 100 minutes?
- One way to express this query in RA is:
  $\pi_{title,year}(\sigma_{studioName = 'Fox' \text{ and } length >= 100}(\text{movie}))$
- Another way:
  - Select those **movie** tuples that have *length* ≥100
  - Select those **movie** tuples that have *studioName* = 'Fox'
  - Find the intersection of the above two results
  - Project on the attributes ***title*** and ***year***

## Example: RA Query

$\pi_{title,year}(\sigma_{studioName = 'Fox' \text{ and } length >= 100}(movie))$




$\pi_{title,year}(\sigma_{length \geq 100}(movie) \cap \sigma_{studioName = 'Fox'}(movie))$

## Example: RA Query

- Relation schema:
  **Movie** (title, year, length, filmType, studioName)
  **StarsIn** (title, year, starName)
- Query: List the **names** of the stars of movies of length ≥ 100 minutes long.
- One expression in RA for this query:
  - Select **movie** tuples of ***length* ≥ 100**
  - Join the result with relation **StarsIn**
  - Project on the attribute ***starName***
- ***Exp1***: $\pi_{starName}(\sigma_{length \geq 100}(movie) \bowtie starsIn)$
- ***Another solution***: $\pi_{starName}(\sigma_{length \geq 100}(movie \bowtie starsIn))$

in SQL
SELECT *
FROM Movies m1, Movies m2 *(It gives you access to 2 copies of Movies)*

WHERE …. *(condition)*
You will need to refer to m1 or m2 and not Movies.

## Renaming Operator ($\rho$)

- To **control** the names of the attributes of the relations used by algebra operations, we may need to explicitly **rename** relations. Can do this for *convenience* as well
- **Renaming Operator** is denoted as $\rho_{s(A1,A2,\ldots,An)}(r)$
- The result is a copy of the input relation instance **r**, but named as **s,** and the attributes A1, A2, …, An, in that order.
- Use $\rho_s(r)$ to give relation **r**, a new name **s**

  In this case, the schema of **s** will be the same as that of **r**.

$\rho_{m2}$ (Movies) renaming Movies to m2 and m2 will take its attributes if not specified.

# Example

- Query: $\pi_{starName}(\sigma_{length \geq 100}(movie) \bowtie starsIn)$
- This query can be rewritten in 2 steps as follows:

1. $\rho_{M(title, year, length, filmType, studioName)}(\sigma_{length \geq 100}(movie))$

   or even simpler as: $\rho_M(\sigma_{length \geq 100}(movie))$ if used in the same formula

2. Or use $M := \sigma_{length \geq 100}(movie)$ as a separate formula and then formulate the query as: $\pi_{starName}(M \bowtie starsIn)$

- Consider **takes**(sid, cid, grade)
- Query: Find ID of every student who has taken at least 2 courses.
- $\pi_{takes.sid}(\sigma_{(takes.sid = T.sid) \text{ and } (takes.cid \neq T.cid)}(takes \times \rho_T(takes)))$

M:= $\sigma_{length>=100}$(movie)

T1 (TABLE ONE)

| Sid | Cid |
|---|---|
| 123 | 335 |
| 123 | 353 |
| 111 | 335 |

T2 (TABLE TWO)

| Sid | Cid |
|---|---|
| 123 | 335 |
| 123 | 353 |
| 111 | 335 |

We want the cross product of T1 and T2: so we do the $\pi$ takes sid (….) like on the slide. $\rho_T$ (takes) is the table two

## Dependent and Independent Operations

- Some RA operations can be expressed based on other operations. Examples include:
  - $r \cap s = r - (r - s)$
  - $r \bowtie_C s = \sigma_C (r \times s)$
  - $r \bowtie s = \pi_L(\sigma_{r.A1 = s.A1 \text{ AND} \ldots \text{ AND } r.An = s.An} (r \times s))$,

    where **L** is the list of attributes in **R** *followed* by those attributes in **S** that are not in **R,** and **A1**,…, **An** are the common attributes of **R** and **S**

Minimal set to express minimal : 3

# Relational Algebra with Bag Semantics

- Relations stored in DB are called **base** relations/tables.
- Base relations are normally sets; no duplicates.
- In some situations, e.g., during query processing, it is allowed for relations to have duplicate tuples.
- If duplicates are allowed in a collection, it is called ***bag/multiset***.

Instance r of R:

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 6 | 5 | 8 |
| 6 | 5 | 8 |
| 1 | 2 | 3 |
| 9 | 7 | 11 |

Here, r is a bag

# Why Bags?

- **1. Faster projection operations**
  - Bag projection is faster, since otherwise returning distinct values is expensive (as we need sorting for duplicate elimination.

    Another example: *Computing the bag union* ($r \cup^B s$) *is much cheaper* than computing the standard set union $r \cup s$.

    Formally, if **r** and **s** have *n* and m tuples, then the bag and set union operations will cost $O(n+m)$ and $O(n*m)$, respectively.
- **2. Correct computation** with some aggregation
  - For example, to compute the *average* of values for attribute A in the previous relation, we must consider the bag of those values

# Set Operations on Bags

- $\mathbf{r} \cup^B \mathbf{s}$, the **bag union** of **r** and **s**, is the bag of tuples that are in **r**, in **s,** or in both. If a tuple *t* appears *n* times in **r**, and *m* times in **s**, then *t* appears *n+m* times in bag $\mathbf{r} \cup^B \mathbf{s}$

$$\mathbf{r} \cup^B \mathbf{s} = \{ t{:}k \mid t{:}n \in \mathbf{r} \wedge t{:}m \in \mathbf{s} \wedge k = n+m \}$$

- $\mathbf{r} \cap^B \mathbf{s}$, the **bag intersection** of **r** and **s**, is the bag of tuples that appear in both **r** and **s.** If a tuple *t* appears *n* times in **r**, and *m* times in **s**, then the number of occurrences of t in bag $\mathbf{r} \cap^B \mathbf{s}$ is *min(n,m)*

$$\mathbf{r} \cap^B \mathbf{s} = \{ t{:}k \mid t{:}n \in \mathbf{r} \wedge t{:}m \in \mathbf{s} \wedge k = min(n,m) \}$$

- $\mathbf{r} -^B \mathbf{s}$, the **bag difference** of **r** and **s** is defined as follows:

$$\mathbf{r} -^B \mathbf{s} = \{ t{:}k \mid t{:}n \in \mathbf{r} \wedge t{:}m \in \mathbf{s} \wedge k = max(0, n-m) \}$$

$$\mathbf{s} -^B \mathbf{r} = \{ t{:}k \mid t{:}n \in \mathbf{r} \wedge t{:}m \in \mathbf{s} \wedge k = max(0, m-n) \}$$

# Example

**Bag r:**

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 1 | 2 |
| 1 | 2 |

**Bag s:**

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 3 | 4 |
| 5 | 6 |

**$\mathbf{r} \cup^B \mathbf{s}$:**

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 1 | 2 |
| 1 | 2 |
| 1 | 2 |
| 3 | 4 |
| 3 | 4 |
| 5 | 6 |

# Example

Bag r:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 1 | 2 |
| 1 | 2 |

Bag s:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 3 | 4 |
| 5 | 6 |

$r \cap^B s$:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |

# Example

Bag r:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 1 | 2 |
| 1 | 2 |

Bag s:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 3 | 4 |
| 5 | 6 |

$r -^B s$:

| A | B |
|---|---|
| 1 | 2 |
| 1 | 2 |

# Example

Bag r:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 1 | 2 |
| 1 | 2 |

Bag s:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 3 | 4 |
| 5 | 6 |

$s -^B r$:

| A | B |
|---|---|
| 3 | 4 |
| 5 | 6 |

# Bag Projection $\pi^B$

- Let **R** be a relation scheme, and **r** be a collection of tuples over **R**, which could have duplicates.

  The **bag projection** operator is used to produce, from **r,** a bag of tuples over some of **R.**
- Even when **r** does not have duplicates, we may get duplicates when projecting on some attributes of **R**.

  That is, $\pi^B$ does not eliminate the duplicates and hence corresponds exactly to the SELECT clause in SQL.

# Example

Bag r:

| A | B | C |
|---|---|---|
| 1 | 2 | 5 |
| 3 | 4 | 6 |
| 1 | 2 | 7 |
| 1 | 2 | 8 |

$\pi^B_{A, B}$ (r):

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 1 | 2 |
| 1 | 2 |

# Example

**Relation Schema:** **movie**(title, year, length, filmType, studioName, producer)

**Instance:**

| title | year | length | filmType | studioName | producer |
|---|---|---|---|---|---|
| Star wars | 1977 | 124 | color | Fox | 12345 |
| Mighty Ducks | 1991 | 104 | color | Disney | 67890 |
| Wayne's World | 1992 | 95 | color | Paramount | 99999 |

$\pi^{B}_{filmType}$(movie):

| filmType |
|---|
| color |
| color |
| color |

# Selection on Bags

- The selection operator $\sigma_c$ applied to an instance **r** of relation **R**, will return a subset of **r**
  - The tuples in the output relation are those that satisfy condition **C**, which involves attributes of **R**
  - Duplicates are **not** eliminated from the result of a bag-selection

Note: The selection operation $\sigma$ in RA is different from the **SELECT** clause in SQL

# Example

Bag r:

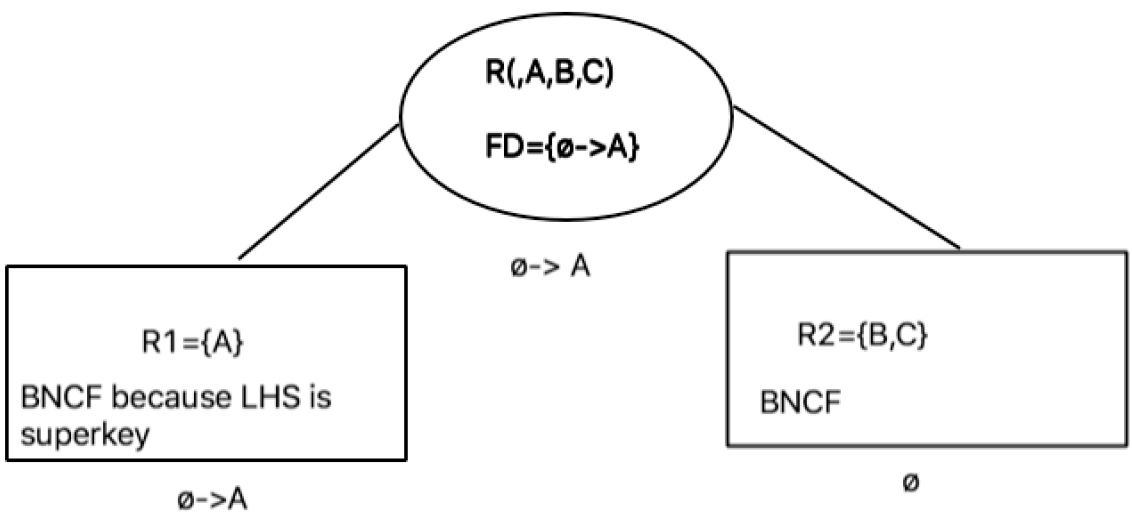
| A | B | C |
|---|---|---|
| 1 | 2 | 5 |
| 3 | 4 | 6 |
| 1 | 2 | 7 |
| 1 | 2 | 7 |

$\sigma_{C \geq 6}(r)$:

| A | B | C |
|---|---|---|
| 3 | 4 | 6 |
| 1 | 2 | 7 |
| 1 | 2 | 7 |

## March 19, 2018

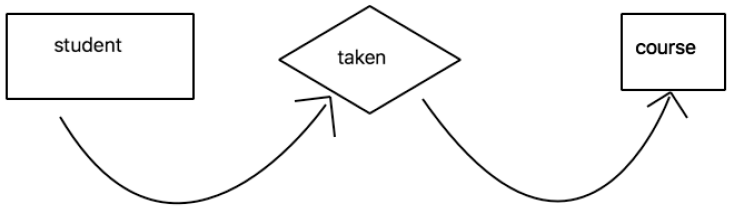
R(A,B,C) ø→ A We conclude that column A has identical values
In what normal form is R? FD={ø→A}
ø+= A this not a super key , so not BNCF, not 3NF because keys are B,C
BC+= {A,B,C}
R(A,B,C) ø→A what is the violating FD is ø→ A



# Constraints on Relations

- RA offers a convenient way to express a wide variety of constraints, e.g., referential integrity and FD's.
- There are **two ways** to express constraints in RA
  1. If **r** is an expression in RA, then the constraint **r** = Ø says:
     *"**r** has no tuples, i.e., or **r** is empty"*
  2. If **r** and **s** are RA *expressions*, then the constraint **r** ⊆ **s** says:
     "*every tuple in (the result of) **r** is in (the result of) **s***"

These constraints hold also when **r** and **s** are bags.




r ⊆ s then r – s =ø
r: taken and s : course

# Referential Integrity Constraints

- Referential integrity in relational data model means:
  - if we have a value ***v*** in a tuple *t* in a relation **r**,

    then we also expect that ***v*** appears in a particular component of some tuple *s* in relation **s**

  E.g., if we have a tuple *(s,c,g)* in ***takes****(sid,cid,grade)*, then there must be a **student** with *sid = s* and a **course** with *cid = c* such that *s* has taken *c*

  *The mentions of values s and c in **takes** "refers" to some values outside this relation, and these values **must** exist*

# Example

- Relation schemas:
  **Movie** (title, year, length, filmType)
  **StarsIn** (title, year, starName)
- Constraint:

  the **title** and **year** of every movie that appears in relation **starsIn** *must* appear also in **movie**; otherwise there is a violation in referencing in **starsIn**
- Query in RA:
  - $\pi_{\text{title, year}}(\text{starsIn}) \subseteq \pi_{\text{title, year}}(\text{movie})$

  or equivalently
  - $\pi_{\text{title, year}}(\text{starsIn}) - \pi_{\text{title, year}}(\text{movie}) = \varnothing$

Movie tuple from Movie? π title, year(Movie)

# Functional Dependencies

- **Any** functional dependency X ⟶ Y can be expressed as an expression in RA
- Example:
  Consider the relation schema:
  **Star** (name, address, gender, birthdate)
- How to express the FD: **name → address** in RA?

**SO WE WANT CROSS PRODUCT**

- Relation schema:
  **Star** (name, address, birthdate)
- With the FD: **name → address**
- The **idea** is that if we construct all pairs of **star** tuples, we **must not** find a pair that agree on **name** but disagree on **address**
- To "construct" the pairs in RA, we use **Cartesian product**, and to find pairs that violate this FD, we use **selection**
- We are then ready to express this **FD** by equating the result to **ø**, as follows…

**3 copies of star**

# Example

**Star:**

| Name | Address | Birthdate |
|---|---|---|
| Carrie Fisher | 123 Maple | 9/9/99 |
| Mark Hamill | 456 Oak rd. | 8/8/88 |
| Harrison Ford | 789 Palm rd. | 7/7/77 |

$\rho_{S1(name,\ address,\ birthdate)}(star)$

| Name | Address | Birthdate |
|---|---|---|
| Carrie Fisher | 123 Maple | 9/9/99 |
| Mark Hamill | 456 Oak rd. | 8/8/88 |
| Harrison Ford | 789 Palm rd. | 7/7/77 |

$\rho_{S2(name,\ address,birthdate)}(star)$

| Name | Address | Birthdate |
|---|---|---|
| Carrie Fisher | 123 Maple | 9/9/99 |
| Mark Hamill | 456 Oak rd. | 8/8/88 |
| Harrison Ford | 789 Palm rd. | 7/7/77 |

**The cross product of s1xs2 where name is the same and address is different: there is no single star that has a different address**

**s1 × s2:**

| S1.Name | S1.Address | S1.Birthdate | S2.Name | S2.Address | S2.Birthda |
|---|---|---|---|---|---|
| Carrie Fisher | 123 Maple | 9/9/99 | Carrie Fisher | 123 Maple | 9/9/99 |
| Carrie Fisher | 123 Maple | 9/9/99 | Mark Hamill | 456 Oak rd. | 8/8/88 |
| Carrie Fisher | 123 Maple | 9/9/99 | Harrison Ford | 789 Palm rd. | 7/7/77 |
| Mark Hamill | 456 Oak rd. | 8/8/88 | Carrie Fisher | 123 Maple | 9/9/99 |
| Mark Hamill | 456 Oak rd. | 8/8/88 | Mark Hamill | 456 Oak rd. | 8/8/88 |
| Mark Hamill | 456 Oak rd. | 8/8/88 | Harrison Ford | 789 Palm rd. | 7/7/77 |
| Harrison Ford | 789 Palm rd. | 7/7/77 | Carrie Fisher | 123 Maple | 9/9/99 |
| Harrison Ford | 789 Palm rd. | 7/7/77 | Mark Hamill | 456 Oak rd. | 8/8/88 |
| Harrison Ford | 789 Palm rd. | 7/7/77 | Harrison Ford | 789 Palm rd. | 7/7/77 |

$\sigma_{S1.name=S2.name\ \mathbf{AND}\ S1.address \neq S2.address}(s1 \times s2) = \varnothing$

# Functional Dependencies

- Relation schema:
  **Star** (name, address, birthdate)
- With the FD: **name → address**
- In RA:

$$\sigma_{S1.name=S2.name \text{ AND } S1.address \neq S2.address}(\rho_{S1}(star) \times \rho_{S2}(star)) = \varnothing$$

# Domain Constraints

- Relation schema:
  **Star** (name, address, gender, birthdate)
- How to express the following constraint?
  Valid values for **gender** are **'F'** and **'M'**
- In RA:
  - $\sigma_{gender \neq 'F' \text{ AND } gender \neq 'M'}(star) = \varnothing$
  - This is an example of ***domain constraints***

**Salaries has a range. No tuple has a salary over 150000**

# Domain Constraints

- Relation schema:
  **Employee** (eid, name, address, salary)
- How to express the constraint:
  Maximum employee salaries is $150,000
- In RA:
  - $\sigma_{salary > 150000}$ (employee) = ∅

Enrolled will get students that took a class so students with no class won't show, and will get classes in which students a registered but if a course has no students enrolled it will not show up.

# "For All" Queries (1)

- Given the database schema:
  Student(Sid, Sname, Addr)
  Course(Cid, Cname, Credits)
  Enrolled (Sid, Cid)
- Consider the query:
  "*Find students enrolled in **all** the courses.*"
- A first attempt (below) fails!

$$\pi_{Sid}(\text{Enrolled})$$

- This RA query returns students enrolled in some courses.
- So, how to correctly express "For All" types of queries?

# "For All" Queries (2)

- A *solution strategy* would be to:
  - start with the list of all students (**all guys**), from which we then subtract those who have not taken some courses (**bad guys**)
- That is, to find all the "good guys", we need to find "all guys" from which we then remove the "bad guys", i.e.,

**Answer (Good guys) = All guys – Bad guys**

{sid} x {cid} - enrolled (sid, cid)

| |
|---|
| (s1,c1) |
| (s1,c2) |
| (s2,c1) |
| (s2,c2) |

| | |
|---|---|
| ~~s1~~ | ~~c1~~ |
| ~~s1~~ | c2 |
| s2 | c2 |

| | |
|---|---|
| s1 | c1 |
| s1 | c2 |
| s2 | c2 |

= {(s2, c2)}

# "For All" Queries (3)

- Set of all students which we need to consider:

  All Courses $\leftarrow \pi_{Cid}$ (Course)

  All Students $\leftarrow \pi_{Sid}$ (Student)

- Steps to find students not enrolled in all the courses
  1. Create all possible "student-course" pairs:

     SC-Pairs $\leftarrow \pi_{Sid}$ (Student) $\times \pi_{Cid}$ (Course)

  2. Get all "actual" student-course pairs -- take them from Enrolled
  3. Students who are not enrolled in all the courses:

     B $\leftarrow \pi_{Sid}(\pi_{Sid}$(Student) $\times \pi_{Cid}$(Course) − Enrolled)

- Answer : All Students − "Bad"

Division operation properties:
1) $\pi$ cid(course) must be a subset of Enrolled

# The Division Operation (÷)

- The previous query can be conveniently and expressed in RA using the ***division*** operator ÷
  - Divide Enrolled by $\pi_{Cid}$ (Course)

    that is, Enrolled ÷ $\pi_{Cid}$ (Course)
  - Schema of the result is {Sid, Cid} − {Cid}
- R ÷ S requires that the attributes of S to be a subset of R.
  - The schema of the output would be R − S

# Example: Enrolled (student, sport)

**Find students enrolled in all sports {Hockey, Football}.**

**Enrolled (Student, sport)**

| | |
|---|---|
| Jim | Hockey |
| Joe | Football |
| Jim | Football |
| Sue | Hockey |

# Example: Enrolled(student, sport)

$\pi_{student}$(Enrolled) $\times$ $\pi_{sport}$(Enrolled) – Enrolled =

| | |
|---|---|
| Jim | Hockey |
| Jim | Football |
| Joe | Hockey |
| Joe | Football |
| Sue | Hockey |
| Sue | Football |

–

| | |
|---|---|
| Jim | Hockey |
| Joe | Football |
| Jim | Football |
| Sue | Hockey |

=

| | |
|---|---|
| Joe | Hockey |
| Sue | Football |

$\pi_{student}$(Enrolled) –
$\pi_{student}$( $\pi_{student}$(Enrolled) $\times$ $\pi_{sport}$(Enrolled) – Enrolled )

**All** **Bad**

| |
|---|
| Jim |
| Joe |
| Sue |

–

| |
|---|
| Joe |
| Sue |

= ? ➔ Jim is the only student enrolled in all sports

With division operator:

Enrolled(sid, sports) ÷ π sports (Enrolled)

# Another Example

- $r \div s = \pi_{R-S}(r - (\pi_{R-S}(r) \times s - r))$
- Given the DB schema:
  - Customer(cid, name)
  - Branch(bid, district)
  - Account(cid, bid)
- Query: "*Find the names of all customers who have an account in every branch located in the Westmount area*"
- Solution?
  - $\pi_{name}$ (Customer ⋈ Account ⋈ ($\sigma_{district = "Westmount"}$ (Branch)))?
- **No**, this returns the names of all customers who have an account at some branch in Westmount, but not necessarily at every such branch.

## Database:

**Customer(cid, name), Branch(bid, district), Account(cid, bid)**

- We can apply the division operator ÷
  - Find all customer-branch pairs **(cid, bid)** for which customer (cid) has an account at branch (bid):
    $\pi_{cid,bid}$ (customer ⋈ account)
  - Divide the above by all bid's of branches in Westmount
    $\pi_{bid}$ ($\sigma_{district = "Westmount"}$ (branch))
- $\pi_{name}$ ((customer ⋈ account) ÷ $\pi_{bid}$ ($\sigma_{district = "Westmount"}$ (branch)))
- So, the division operation $r \div s$ is defined as:
  $r \div s = \pi_{R-S}(r - (\pi_{R-S}(r) \times s - r))$

# DB8

## SQL Queries: Review

- SQL query has a form
  **SELECT** . . .
  **FROM** . . .
  **WHERE** . . . ;
- The **SELECT** clause says which **A**ttribute(s) we are interested in
- The **FROM** clause says which **R**elation(s) we refer to
- The **WHERE** clause says which **T**uple(s) we refer to

## Case Insensitivity

- SQL is *case insensitive*
- So, keyword FROM maybe written as:
  - FROM or
  - From or
  - FrOm
- Only in ***strings,*** SQL distinguishes between uppercase and the lowercase letters
  - So, the following are different strings:
    - 'FROM'
    - 'From'
    - 'FrOm'

## Select Clause

- In place of * in the **SELECT** clause, we can put any attribute "we wish" to project on
- In the **SELECT** clause, we can also do **renaming**

**SELECT** title **AS** name, length **AS** duration
**FROM** Movie;

➔ the structure of the query output "*appears as*": (name, duration)

**AS will rename attributes**

## Select Clause

- We can also use a *formula* in place of an attribute

**SELECT** title **AS** name, length/60 **AS** lengthInHours
**FROM** Movie;

➔ the structure of the output: (name, lengthInHours)

**You can use mathematical expression length/60**

## Select Clause

- SQL even allows using a constant as an item in the **SELECT** clause, as shown below.

```
SELECT title AS name, length/60 AS length, 'hrs.' AS inHours
FROM Movie;
```

➔The structure of the output:

| (name, | length, | inHours) |
|---|---|---|
| Gone with the wind | 1.98 | hrs. |
| King Kong | 0.75 | hrs. |

- Why? To put some "useful" words into the output that SQL displays

## March 21, 2018

WHERE R.A= t.b AND s LIKE p are two constraints
give me the year of the title "Gone …" then expressed:
WHERE title LIKE 'Gone%'
Special characters

# Comparison of Strings

- Two strings are equal if they have the same sequence of characters/symbols
- Strings are compared alphabetically
  - 'fodder' < 'foo'
  - 'bar' < 'bargain'
- **WHERE** R.A = T.B **AND** s LIKE p
  - **s** is an attribute of type *string* and **p** is a pattern; e.g. WHERE title LIKE 'Gone%'
  - "*Ordinary*" characters in **p** matches ordinary characters in **s**
  - What about "*Special*" characters in **p**: % , _
  - "%" in **p** matches **any** sequence of zero or more characters in **s**
  - " " in **p** matches any **one** character in **s**

# Comparison of Strings

- Suppose we remember a movie "Star *something*", and we do remember that "the something" has four letters

```
SELECT title
FROM Movie
WHERE title LIKE 'Star _ _ _ _';

SELECT title
FROM Movie
WHERE title LIKE 'Star%';
```

# Comparison of Strings

What if the pattern p includes ', %, or _?

- Find all movies with a possessive ( 's ) in their title

  **LIKE '%'s%'**

  **SELECT** title
  **FROM** Movie
  **WHERE** title **LIKE** '%''s%';

  - The convention is that two apostrophes '' in a string represent **one** single apostrophe ('), and not the end of string

# Comparison of Strings

- What if **p** involves the *special characters* **%** or _?
  - We should "escape" their special meaning using "some" **escape character**
  - **SQL** allows using *any* character as escape character
- s **LIKE** 'x%%x%' **ESCAPE** 'x';
  - Here, x is the escape character ➔ x% means the character %, and not its usual meaning (the special character)
  - The pattern 'x%%x%' matches strings: %whatever%

# Ordering the Ouput

- We may wish the output of a query to be displayed in some order. This could be done using the SQL clause:
  - **ORDER BY** <list of attributes>
- E.g., List Disney movies in 1990 by their length, shortest first and then by the alphabetical order of the titles:

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length, title;
```

  - Default ordering is ASCending, unless we use the **DESC** keyword
  - Ties are broken by the "next" attribute in the **ORDER BY** list.

# Products and Joins

- SQL has a simple way to couple relations in a query
  - How? Simply list each relation in the **FROM** clause
- All the relations in the **FROM** clause are coupled through **Cartesian product**
- Then we can put conditions in the **WHERE** clause in order to get a desired kind of **join**

# Join (Example, Recall)

- Relation schemas:

  **Movie** (title, year, length, filmType)

  **Owns** (title, year, studioName)

- Query:

  Find titles and lengths of all movies produced by Disney

- Query in SQL:

```
SELECT Movie.title, Movie.length
FROM Movie, Owns
WHERE Movie.title = Owns.title AND Movie.year = Owns.year
        AND Owns.studioName = 'Disney';
```

# Union, Intersection, and Difference

- We can apply the common set operations of **union, intersection,** and **difference** to relations **R** and **S,** if they are *compatible*.
- When the output of two or more SQL queries are compatible, we may "combine" the queries using:
  - **UNION**
  - **INTERSECT**
  - **EXCEPT** (or **MINUS** in Oracle)

# Union, Intersection, and Difference

- Relation schemas:

  **Movie** ( title, year, length, filmType)

  **StarsIn** (title, year, starName)

- Query:

  Find titles and years of movies that appeared in either **Movie** or **StarsIn** relations

- Query in SQL:

```
SELECT title, year
FROM Movie
    UNION
SELECT title, year
FROM StarsIn;
```

# Union, Intersection, and Difference

- Relation schemas:

  **Star(**name, address, gender, birthdate**)**

  **Exec**(name, address, cert#, netWorth)

- Query:

  Find names and addresses of all female movie stars who are also movie executives with a net worth of over $10,000,000

- Query in SQL:

```
SELECT name, address
FROM Star
WHERE gender = 'F'
    INTERSECT
SELECT name, address
FROM Exec
WHERE netWorth > 10000000;
```

# Union, Intersection, and Difference

- Relation schemas:

  **Star** (name, address, gender, birthdate)

  **Exec** (name, address, cert#, netWorth)

- Query:

  Find names and addresses of movie stars who are **not** movie executives

- Query in SQL:

```
SELECT name, address
 FROM Star
    EXCEPT              //or MINUS in Oracle//
SELECT name, address
 FROM Exec;
```

# Duplicate Elimination

***Note that in SQL:***

- The **union**, **intersection**, and **difference** operations normally eliminate duplicates (the set semantics)
- To retain duplicates, hence preventing duplicate elimination, we must use the keyword **ALL** after the operator **UNION**, **INTERSECT**, and **EXCEPT**
  - *R* **UNION ALL** *S (the only bag operation supported in Oracle)*
  - *R* **INTERSECT ALL** *S*
  - *R* **EXCEPT ALL** *S*

# Retaining Duplicates

- ***R* UNION ALL *S***
  - The bag of elements that are in **R**, **S**, or in both. If **R** is a bag in which tuple *t* appears ***n*** times, and **S** is a bag in which *t* appears ***m*** times, then the number of occurrences of tuple *t* in bag **R** ∪ **S** is ***n*** + ***m***
- ***R* INTERSECT ALL *S***
  - The bag of elements that are in both **R** and **S.** If **R** is a bag in which tuple *t* appears ***n*** times, and **S** is a bag in which *t* appears ***m*** times, then the number of occurrences of *t* in bag **R** ∩ **S** is *min*(***n,m***)
- ***R* EXCEPT ALL *S***
  - The bag of elements that are in **R** but not in **S.** If **R** is a bag in which tuple *t* appears ***n*** times, and **S** is a bag in which *t* appears ***m*** times, then the number of occurrences of *t* in bag **R – S** is *max*(**0, *n* - *m***)

# Retaining Duplicates in Union

- Relation schemas:

  **Movie** ( title, year, length, filmType)

  **StarsIn** (title, year, starName)

- Query:

  List the title and year of every movie that appears in **Movie** or **StarsIn**

- Query in SQL:

```
SELECT title, year
FROM Movie
    UNION ALL
SELECT title, year
FROM StarsIn;
```

**DB9**

# Scalar Values

- An SQL query is an expression that evaluates to a collection of tuples, i.e., it produces a relation/bag
- This "collection" may have only one attribute
- It is also possible that there will be only one single value produced for that attribute
- If all these hold, then we say that the query produces a **scalar** value
  - **Scalar values** – example include simple values such as integers, reals, strings, dates, etc.

2

# Queries that Produce Scalar Values

- Relation schema:

  **Movie**( title, year, length, filmType, studioName, producerC# )
- Query:

  Find certificate number of the producer of "Star Wars"
- Query in SQL:

```
SELECT producerC#
FROM Movie
WHERE title = 'Star Wars';
```

*Assuming that we have only one such movie.*

# Subqueries

- Conditions in the **WHERE** clause may have comparisons that involve scalar values
- A SQL query can produce a scalar value
- If so, we can use such SELECT-FROM-WHERE expression, surrounded by parentheses, **as if it were a constant**
- **Subquery** – a query within a query

  The result of a SQL subquery is a collection (relation/bag)

# Example

- Relation schemas:

  **Movie** (title, year, length, filmType, studioName, producerC#)
  **Exec** (name, address, cert#, netWorth)
- Query:

  Find the name of the producer of "Star Wars"
- Query in SQL:

```
SELECT Exec.name
FROM Movie, Exec
WHERE Movie.title = 'Star Wars' AND
        Movie.producerC# = Exec.cert#;
```

# Example

- Relation schemas:

  **Movie** (title, year, length, filmType, studioName, producerC#)
  **Exec** (name, address, cert#, netWorth)

- Query:

  Find the name of the producer of "Star Wars"

- Query with Subquery:

```
SELECT name
FROM Exec
WHERE cert# = ( SELECT producerC#
                FROM Movie
                WHERE title = 'Star Wars' );
```

# Conditions Involving Relations

- There are a number of SQL checks/conditions that can be done on a relation **R** and produce a **boolean** value
- These conditions can be **negated** by putting a **NOT** before them
- Typically, **R** above is the result of an SQL subquery, shown as: **(R)**
- If such a condition involves a scalar value ***s*** or a tuple, we should make sure its type matches **R**.

# Conditions Involving Relations

- "**EXISTS (*R*)**" is a condition that is true iff ***R*** is not empty
- "***s* IN (*R*)**" is true iff ***s*** is equal to **one** of the values in ***R***
  - "***s* NOT IN (*R*)**" is true iff ***s*** is not equal to any value in ***R***
- "***s* > ALL (*R*)**" is true iff ***s*** is greater than **every** value in ***R***
  - "**>**" could be replaced by other operators with the analogous meaning
  - **Note:** "***s* <> ALL (*R*)**" is the same as "***s* NOT IN *R***"
- "***s* > ANY (*R*)**" is true iff ***s*** is **> at least one** value in ***R***
  - "**>**" could be replaced by any of the other 5 comparison operators with the analogous meaning
  - **Note:** "***s* = ANY (*R*)**" is the same as "***s* IN *R***"

8

# Conditions Involving Tuples

- A tuple in SQL is represented by a parenthesized list of scalar values; the concept "tuple" can be viewed as an *extension* of the concept of scalar;
  - (123, 'foo')
- Mixing of constants and attributes is also permitted in tuples
  - (123, Movie.title)
- If a tuple ***t*** has the same number of components as a relation ***R***, then it makes sense to compare ***t*** and ***R*** like:
  - ***t* IN (*R*)** -- this is true iff ***t*** *is in* ***R***
  - ***t* <> ANY (*R*)** -- *this is true* ***R*** *includes a tuple other than* ***t***

We can get one relation at a time and get subqueries.

# Example

- Relation schemas:

  **Movie** (title, year, length, filmType, studioName, producerC#)
  **Exec** (name, address, cert#, netWorth)
  **StarsIn** (title, year, starName)
- Query: Find the names of the producers of Harrison Ford's movies
- Query in SQL:

```
SELECT name
FROM Exec
WHERE cert#  IN (SELECT producerC#
                 FROM Movie
                 WHERE (title, year) IN (SELECT title, year
                                         FROM StarsIn
                                   WHERE starName = 'Harrison Ford'));
```

10

# Example

- Relation schemas:

  **Movie**(title, year, length, filmType, studioName, producerC#)
  **Exec**(name, address, cert#, netWorth)
  **StarsIn**(title, year, starName)
- Query: Find names of the producers of Harrison Ford's movies
- Query in SQL:

```
SELECT Exec.name
FROM Exec, Movie, StarsIn
WHERE Exec.cert# = Movie.producerC# AND
      Movie.title = StarsIn.title AND
      Movie.year = StarsIn.year AND
      starName = 'Harrison Ford';
```

# Correlated Subqueries

- Simple subqueries can be evaluated once and the result be used in a higher level (calling) query
- A more complex use of nested subquery requires the subquery to be evaluated many times, once for each assignment of a value (to some term in the subquery) that comes from a tuple variable in the calling query
- A subquery of this type is called ***correlated subquery***

# Correlated Subqueries

- Relation schema:

  **Movie**(title, year, length, filmType, studioName, producerC#)
- Query:

  Find movie titles that appear more than once
- Query in SQL:

```
SELECT title
FROM Movie Old
WHERE year < ANY (SELECT year
                  FROM Movie
                  WHERE title = Old.title);
```

Note the scopes of the variables in this query.

# Correlated Subqueries

- Query in SQL

```
SELECT title
FROM Movie Old
WHERE year < ANY (SELECT year
                  FROM Movie
                  WHERE title = Old.title);
```

- The condition in the outer WHERE is true only if there is a movie with sam title as *Old.title* that has a **later** year
  - ➔ The query will produce a title **one fewer times** than there are movies with that title
- What would be the result if we used "<>", instead of "<" ?
  - ➔ For a movie title appearing 3 times, we would get 3 copies of the title in the output

# March 26, 2018

**2 relations are compatible**: same number of attributes and same type.

## Views

- View is a table/relation defined in a database but has no tuples explicitly stored for it in the database but rather computed, when needed, from the view definition

2 application:

1) **provide logical data independence**
   If the schema of a relation is changed, we can define a view with the old schema so that applications that use the old schema can continue using it
2) **Security**
   Views can be used to restricts the users access only the information they are allowed to "see and operate on"

- Relation schema:
  **Movie**(title, year, length, filmType, studioName, producerC#)
- View:
  Create the Paramount's movies (title and year)
- View in SQL:

```
CREATE VIEW ParamountMovie AS
SELECT title, year
FROM Movie
WHERE studioName = 'Paramount';
```

View is ***very expensive***.
A view can be used in defining new queries/views in exactly the same way as an explicitly stored table may be used:

- Example to query the (virtual) relation ParamountMovie

```
SELECT title
FROM ParamountMovie
WHERE year = 1979;
```

- This query is translated, by the query processor, into:

```
SELECT title
FROM Movie
WHERE studioName = 'Paramount' AND year = 1979;
```

If it is cached , the system may also  keep it for some time.

- Relation schema:

  **ParamountMovie** (title, year )

  **StarsIn**(title, year, starName)

- Query:

  List the stars of the movies made by Paramount.

- Query in SQL

```
SELECT DISTINCT StarsIn.starName
FROM ParamountMovie, StarsIn
WHERE ParamountMovie.title = StarsIn.title AND
        ParamountMovie.year = StarsIn.year;
```

- Relation schema:

  **Movie** (title, year, length, filmType, studioName, producerC#)
  **Exec** (name, address, cert#, netWorth)

- View:

  Define a view of Movie (titles and executives/producers)

- View in SQL:

```
CREATE VIEW MovieProd AS
SELECT Movie.title, Exec.name
FROM Movie, Exec
WHERE Movie.producerC# = Exec.cert#;
```

- Relation schema:

  **MovieProd** (title, name)

- Query:

  Find the name of the producer of 'Gone With the Wind'?

- Query in SQL:

```
SELECT name
FROM MovieProd
WHERE title = 'Gone With the Wind';
```

**Renaming attributes used in view definitions**

- Example:

```
CREATE VIEW MovieProd (MovieTitle, ProducerName) AS
SELECT Movie.title, Exec.name
FROM Movie, Exec
WHERE Movie.producerC# = Exec.cert#;
```

- Relation schema:

  **MovieProd** (MovieTitle, ProducerName)
- Query:

  Find the name of the producer of 'Gone With the Wind'?
- Query in SQL:

```
SELECT ProducerName
FROM MovieProd
WHERE MovieTitle = 'Gone With the Wind';
```

# Updating Views?

- We saw that a view can appear in a query in exactly the same way as a "base" table may appear.
- What about modifications/updates?
- What does it mean to update a view?
  - Translate modification of the view to the corresponding modification on the base tables used in the view definition
- Should we allow updates on views?
  - Yes, in principle, but some problems may arise
- Some "simple" views can be updated
  - Such views are called **updatable views**
- Many views cannot be updated
  - This is due to the so called **view-update anomaly**

# Insertion into Views?

- Relation schema:

  **Movie**(title, year, length, filmType, studioName, producerC#)

- View: Recall the definition of ParamountMovie

```
CREATE VIEW ParamountMovie AS
SELECT title, year
FROM Movie
WHERE studioName = 'Paramount';
```

- Update statement:

```
INSERT INTO ParamountMovie (title,year) VALUES('KK', 2002);
```

- Result: the following tuple being added to Movie

  ('KK', 2002, NULL, NULL, NULL, NULL) What's the problem?

# Insertion into Views?

- Relation schema:

  **Movie**(title, year, length, filmType, studioName, producerC#)
- An updatable view:

```
CREATE VIEW ParamountMovie AS
SELECT title, year, studioName
FROM Movie
WHERE studioName = 'Paramount';
```
- Update statement:

```
INSERT INTO ParamountMovie VALUES('KK',2002,'Paramount');
```
- Result: the following tuple is being added to Movie

  ('KK', 2002, NULL, NULL, 'Paramount', NULL) Problem solved!



# Insertion into Views?

- Relation schemas:

  **Movie**(title, year, length, filmType, studioName, producerC# )

  **Exec**(name, address, cert#, netWorth)
- View in SQL:

```
CREATE VIEW MovieProd AS
SELECT Movie.title, Exec.name
FROM Movie, Exec
WHERE Movie.producerC# = Exec.cert#;
```
- Update statement

```
INSERT INTO MovieProd (title,name) VALUES('The Movie', 'J. Smith');
```
- Result: these tuples are added to the corresponding relations:

**Movie**('The Movie', NULL, NULL, NULL, NULL, NULL)

**Exec**('J. Smith', NULL, NULL, NULL) Problems? The insertion command will fail !

# Deletion from Views?

- Relation schema:

  **Movie**(title, year, length, filmType, studioName, producerC#)

- View: Recall the definition :

```
CREATE VIEW ParamountMovie AS
SELECT title, year, studioName
FROM Movie
WHERE studioName = 'Paramount';
```

- Delete statement:

```
DELETE FROM ParamountMovie WHERE title LIKE '%K%';
```

- Translated query:

```
DELETE FROM Movie
WHERE studioName = 'Paramount' AND title LIKE '%K%';
```

# Updating Views?

- Relation schema:

  **Movie**(title, year, length, filmType, studioName, producerC#)
- View:

  **CREATE VIEW** ParamountMovie **AS**

  **SELECT** title, year,studioName

  **FROM** Movie

  **WHERE** studioName = 'Paramount';
- The view update statement:

  **UPDATE** ParamountMovie **SET** year = 1797 **WHERE** title ='KK';

  - We may drop a view: **DROP VIEW** ParamountMovie;

# Updating Views?

- Recall: updating views includes insertion, deletion, and changing data
- SQL provides a formal definition of when modifications to a view are permitted
- Roughly, this is permitted when the view is defined by selecting some attributes from one relation **R**, which could be an "updatable" view itself
  - The list in the **SELECT** clause includes "enough" attributes that for every tuple inserted into the view, the tuple inserted into the base relation will "yield" the inserted tuple of the view
  - The **NOT NULL** constraints on the base table will not be violated
  - The view definition uses **SELECT** (but not **SELECT DISTINCT**)
  - The **WHERE** clause does not involve **R** in a subquery

**DB10**

Update of salaries → cannot have more salaries than chair

# Domains

- SQL allows user defined data types - **domains**
- We can define a domain as follows:
  - The keyword ***CREATE DOMAIN***
  - The name of the domain
  - The keyword ***AS***
  - Type description
  - Optional default value, constraints
- Example

**CREATE DOMAIN** <name> **AS** <type description>

# Domains

- To create a domain:
  - **CREATE DOMAIN** MovieDomain **AS VARCHAR(50);**
- Example:

```
CREATE TABLE Movie (
    title MovieDomain,
    year DATE,
    .......
    .......
);
```

# NULLs

- We use NULL in place of a value in a tuple's component when:
  - The value is *unknown* -- don't know
  - The value is *inapplicable* -- NA
  - The exact value *does not matter* -- don't care
- There could be many reasons why a value is not present in a relation, e.g., when *inserting* a tuple into a relation, we don't have/wish to specify all the values for the attributes.

# Arithmetic operations on NULLs

- Result of an arithmetic operator, when at least one of the operands has a value of NULL, is NULL
- Example:
  - Suppose the value of attribute x is NULL
    ➔ The value x+3 is also NULL
  - Note however that NULL is not a constant!
    ➔ NULL + 3 is ***illegal***

# Arithmetic operations on NULLs

- Some key laws in math fail to hold with NULLs
- Suppose x is an attribute with numeric value
  - Example 1:
    - We know that $x * 0 = 0$, but
    - If $x$ is NULL → $x * 0$ is NULL
  - Example 2:
    - We also know that $x - x = 0$, but
    - If $x$ is NULL → $x - x$ is NULL

# Comparison operations on NULLs

- The result of a comparison is "usually" **TRUE** or **FALSE**
- That is, 2 possible values or ***2-valued logic***
- Comparisons involving NULLs give rise to a **3rd** truth value, **UNKNOWN,** and hence we are dealing with a 3-valued logic
- In this case, the 3 possible values are **true, false, unknown**

# 3-Valued Logic

- We may assume that:
  - **TRUE** = 1
  - **FALSE** = 0
  - **UNKNOWN** = 1/2
- Then:
  - $x$ **AND** $y$ = min ( $x$, $y$ )
  - $x$ **OR** $y$ = max ( $x$, $y$ )
  - **NOT** $x$ = 1–$x$

# Truth table for 3-Valued Logic

| *X* | *Y* | *X* AND *Y* | *X* OR *Y* | NOT *X* |
|---|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE | FALSE |
| TRUE | UNKNOWN | UNKNOWN | TRUE | FALSE |
| TRUE | FALSE | FALSE | TRUE | FALSE |
| UNKNOWN | TRUE | UNKNOWN | TRUE | UNKNOWN |
| UNKNOWN | UNKNOWN | UNKNOWN | UNKNOWN | UNKNOWN |
| UNKNOWN | FALSE | FALSE | UNKNOWN | UNKNOWN |
| FALSE | TRUE | FALSE | TRUE | TRUE |
| FALSE | UNKNOWN | FALSE | UNKNOWN | TRUE |
| FALSE | FALSE | FALSE | FALSE | TRUE |

# 3-Valued Logic

- Some key laws in logic fail to hold with NULLs
- Example:
  - Law of the excluded middle
    - *x* **OR NOT** *x* = **TRUE**
  - For 3-valued logic
    - if *x* = **UNKNOWN** → *x* **OR (NOT** *x***)** =

      max(1/2,(1−1/2)) = 1/2 = **UNKNOWN** ≠ **(TRUE)**
    - **Note: Do not treat NULL as a "value"**

      (e.g., see the next example)

## March 28, 2018

**Final is MC only**

# Example

- Relation schema:
  **Movie** ( title, year, length, filmType, studioName, producerC#)
- Consider query:

```
SELECT *
FROM Movie
WHERE length <= 120 OR length > 120;
```

- Don't we expect to get a copy of the **Movie** relation?
- Yes, if there is no Movie tuple whose length is NULL
- This query returns a subset of **Movie** tuples in general.
  It returns only each **Movie** tuple whose length is not NULL.

# Example

- Read the box on page 254 in the textbook for some rules on NULLs
- The value NULL is ignored in any aggregation.
  - Query:

```
SELECT COUNT(A) FROM R;
```

    returns the number of non-null values under attribute A in R.
  - Query:

```
SELECT COUNT(*) FROM R;
```

    returns the number of tuples in R.
- NULL is treated as an ordinary value in a "group by" attribute.
  - Query:

```
SELECT A, AVG(B) FROM R GROUP BY A;
```

  produces a tuple for each distinct value A, including the null, if exists

# Joins in SQL

- How to express the Cartesian product in SQL?
  - List the relation names in the **FROM** clause
- How do we express various joins in SQL?
  - Follow the Cartesian Product with conditions in the **WHERE** clause for the desired join

# Joins in SQL2

- In SQL2, there are other forms for expressing $\times$ and $\bowtie$
- Cartesian Product of Movie and StarsIn
  - Movie **CROSS JOIN** StarsIn; (Movie $\times$ StarsIn, in RA)
- Theta- (or equi-) join of Movie and StarsIn
  - Movie **JOIN** StarsIn **ON**
    StarsIn.title = Movie.title **AND** StarsIn.year = Movie.year;

  **Note**: The result may have some redundant columns;
  We can use the above expression as a subquery in a **FROM** clause and use **SELECT** to remove these undesired attributes
- Natural join of Movie and StarsIn
  - Movie **NATURAL JOIN** StarsIn;

# Example

- Relation schemas:

  **Exec**(name, address, cert#, netWorth)

  **Star** (name, address, gender, birthdate)
- Query:

  Find all info on all stars who are also movie executives
- Query in SQL:

  **SELECT** *

  **FROM** Star **NATURAL JOIN** Exec;
  - The join expression appears in a **FROM** clause
  - Parenthesized **SELECT-FROM-WHERE** is also allowed in a **FROM** clause

# Example

- Relation schemas:

  **Exec**(name, address, cert#, netWorth)

  **Star** (name, address, gender, birthdate)
- Query: Find "all" information on all stars
- Query in SQL:

  **SELECT** *

  **FROM** Star;
  - This query does not return all information on stars who are also movie executives
- Query in SQL

  **SELECT** *

  **FROM** Star **NATURAL JOIN** Exec ;
  - This query does NOT return stars who are not movie executives

# Outer joins

- ***R* OUTER JOIN *S*** -- computes the join of ***R*** and ***S*** with **dangling** tuples padded with **NULL**s
- A tuple in ***R*** is **dangling** if it doesn't join with any tuple in ***S***
- Outer join could be
  - **FULL OUTER JOIN**
    - It pads, with nulls, the dangling tuples of ***R*** and ***S***
  - **LEFT OUTER JOIN**
    - It pads dangling tuples of ***R*** only
  - **RIGHT OUTER JOIN**
    - It pads dangling tuples of ***S*** only

# Example

Instance R:

| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |

Instance S:

| B | C |
|---|---|
| 2 | 5 |
| 2 | 6 |
| 7 | 8 |

SELECT * FROM R NATURAL FULL OUTER JOIN S:

| A | B | C |
|---|---|---|
| 1 | 2 | 5 |
| 1 | 2 | 6 |
| 2 | 3 | NULL |
| NULL | 7 | 8 |

# Example

**Instance R:**

| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |

**Instance S:**

| B | C |
|---|---|
| 2 | 5 |
| 2 | 6 |
| 7 | 8 |

R NATURAL LEFT OUTER JOIN S:

| A | B | C |
|---|---|---|
| 1 | 2 | 5 |
| 1 | 2 | 6 |
| 2 | 3 | NULL |

# Example

**Instance R:**

| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |

**Instance S:**

| B | C |
|---|---|
| 2 | 5 |
| 2 | 6 |
| 7 | 8 |

R NATURAL RIGHT OUTER JOIN S:

| A | B | C |
|---|---|---|
| 1 | 2 | 5 |
| 1 | 2 | 6 |
| NULL | 7 | 8 |

# Outer joins in SQL2

*R* [**NATURAL**] [**LEFT** | **RIGHT** | **FULL**] **OUTER JOIN** *S* [**ON** ...]

- Can do either a natural join ⋈ or a theta-join ⋈c
- Use **NATURAL** for ⋈, and use **ON** for theta-outer join, but not both (NATURAL and ON cannot be used together)
- Can use any one of **LEFT**, **RIGHT**, or **FULL**

- Examples:
  - *R* **NATURAL FULL OUTER JOIN** *S*;
  - *Star* **NATURAL LEFT OUTER JOIN** *Exec*; (for slide #17)
  - Movie **RIGHT OUTER JOIN** StarsIn **ON** StarsIn.title = Movie.title **AND** StarsIn.year = Movie.year;

22

# Constraints and Triggers (Chapter 7)

- SQL provides a variety of ways for expressing *integrity constraints* as part of the database schema
- Constraints' checking, in essence, provide users with more control over the database content
- An *active* element is a statement that we write once, store it in the database, and "expect" it to be executed at "appropriate" times
- The *time of action* might be when certain events occur (e.g., insertion of a tuple into a relation or any change(s) made to the database so that certain condition(s) becomes true

23

# Constraints

- Declaration of primary keys
- Foreign key constraints (also see referential integrity constraints)
  - E.g., if in relation **StarsIn**, it says that a star has a role in a movie **m,** then there should be a movie tuple **m** in **Movie**
- Constraints on attributes, tuples, and relations
- SQL2 Assertions = global/general constraints (inter-relations)
  - Not supported in Oracle
- Triggers
  - are substitutes for "general" assertions
- SQL3 triggers and assertions

24

**Fct dependencies are relation constraints.**

# Primary Key

- A key constraint is declared by the DDL command **CREATE TABLE**
- Can use the keywords **PRIMARY KEY** or **UNIQUE**
  - Oracle treats them as synonyms
  - A table can only have one primary key but any number of "unique" declarations
- Two ways to declare a primary key in the **CREATE TABLE** statement:
  - After an attribute type, if the attribute is a key by itself
  - As a separate line, for any number of attributes forming the key:

    **PRIMARY KEY ( list of attribute(s) )**
    - **Should use the second way if the key is not a singleton**

# Primary Key

- Example:

```
CREATE TABLE Star(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE);
```

Three consequences of declaring a primary key:

1. Repeated values of the key attributes will not be allowed
   Any violation will be rejected/failed by the DBMS
2. NULLs are not allowed for the key attribute(s)
3. (Another possible consequence) Creating an index on the primary key, or alternatively keeping the table sorted on the key attribute(s).

# Primary Key

- Example:

```
CREATE TABLE Star (
        name CHAR(30) UNIQUE,
        address VARCHAR(255) UNIQUE,
        gender CHAR(1),
        birthdate DATE);
```

This is to say: *no two movie stars have the same address*

# Primary Key

- Example:

```
CREATE TABLE Star(
        name CHAR(30),
        address VARCHAR(255),
        gender CHAR(1),
        birthdate DATE,
        UNIQUE (name)
);
```

Note the distinction between **UNIQUE** and **PRIMARY KEY**:
When using **UNIQUE**, NULLs are allowed for (all or some of) the key attribute(s).

28

Primary key constraints→ relation case of constraints

# Primary Key

- Example:

```
CREATE TABLE Movie (
        title CHAR(20),
        year INT,
        length INT,
        PRIMARY KEY(title, year)
);
```

Here, the key declaration must appear on a separate line, since the key consists of more than 1 attribute (title, year)

# Foreign Keys

- Referential integrity constraints:

  Intuitively, values for certain attributes must "make sense"
  - That is, every non-Null value in attribute **A** of relation **R** must appear in attribute **B** of relation **S -- Inclusion dependency**
- In SQL, we declare an attribute (or a set K of attributes) in a relation (**R**) to be a *foreign key*, if they reference/point to some attribute(s) G of some related relation **S**.

  That is, t[K] = s[G] if tuple *t* in **R** refers to tuple *s* in **S**
  - The set of attributes G must be declared as the *primary key* of **S**
  - The notion of referential integrity constraint is the one that connects or relates tuples in different relations.

# Foreign Keys

- Two ways to declare foreign keys:
  - If the foreign key is a single attribute A, we may write the following after the attribute *name* and its *type*:

(1) **REFERENCES** < referenced-table > (A)

  - When the foreign key includes more than one attribute, write it as a separate line in the **CREATE TABLE** declaration:

(2) **FOREIGN KEY** (<attributes>) **REFERENCES** <table> (<attributes>)

    - Use form (2) if the foreign key includes 2 or more attributes

# Foreign Keys

- Relation Schemas:

**Exec**(eName, address, cert#, netWorth)

**Studio** (sName, address, presidentC#)

Here, Studio(presidentC#) refers to Exec(cert#).

```
CREATE TABLE Studio(
        sName CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presidentC#  INT REFERENCES Exec(cert#)
);
```

# Foreign Keys

- Relation Schemas:

  **Exec**(eName, address, cert#, netWorth)
  **Studio** (sName, address,presidentC#)

```
CREATE TABLE Studio(
    sName CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presidentC#  INT,
    FOREIGN KEY presidentC#  REFERENCES  Exec(cert#) );
```

In either declaration form, when a value ***v*** appears for presidentC# in a Studio tuple, **v** MUST already exist for an Exec tuple.

Note: *An exception to this requirement is when **v** is NULL.*

# Maintaining Referential Integrity

**Exec**(eName, address, cert#, netWorth)
**Studio** (sName, address,presidentC#)

- How to maintain referential integrity when the database is modified?
- Possible situations **violating** foreign key constraints:
  - Insert:
    - 1. Insert a new Studio tuple with presidentC# that is not cert# of any Exec tuple
  - Update:
    - 2. Update a Studio tuple to change its presidentC# to a non-Null value that is not the cert# of any tuple in table Exec
    - 3. Update an Exec tuple **e** that changes **e**.cert#, when the old cert# was presidentC# of some Studio tuple **s** (i.e., **e**.cert# is referenced by **s**)
  - Delete:
    - 4. Delete an Exec tuple when its cert# is presidentC# of 1 or more Studio tuples Recall that cert# is the key of the Exec table

34

When this will be violated ( 2 first)
4 cases said upper

# Maintaining Referential Integrity

- There are *three* policies, when there is a transaction that violates a referential integrity:
  - The ***reject*** policy *(default)*

    The system will reject any transaction violating referential integrity constraints -- a run-time error will be generated and the database state will *not* change.
  - If we update (3) or delete (4) a referenced item:
    - The ***cascade*** policy: changes to the referenced attributes are "mimicked" at the foreign key (e.g. presidentC#)
    - The "***set null***" policy: set the referencing attribute to NULL (e.g., presidentC# in Studio)

# Maintaining Referential Integrity

**Exec**(eName, address, cert#, netWorth)

**Studio** (sName, address,presidentC#)

Example: For the (default) reject policy, the system will **reject** the following modifications/transactions:

- Insert a new Studio tuple with presidentC# that is not in any Exec tuple
- Update a Studio tuple to change the presidentC# to a non-null value that is not in any Exec tuple
- Update an Exec tuple to change cert#, when the old cert# value is presidentC# in some Studio tuple(s)
- Delete an Exec tuple when its cert# is presidentC# in some Studio tuple(s)

36

## April 4, 2018

# Maintaining Referential Integrity

**Exec**(eName, address, cert#, netWorth)
**Studio** (sName, address,presidentC#)

- How to maintain referential integrity when the database is modified?
- Possible situations **violating** foreign key constraints:
  - Insert:
    - 1. Insert a new Studio tuple with presidentC# that is not cert# of any Exec tuple
  - Update:
    - 2. Update a Studio tuple to change its presidentC# to a non-Null value that is not the cert# of any tuple in table Exec
    - 3. Update an Exec tuple **e** that changes **e**.cert#, when the old cert# was presidentC# of some Studio tuple **s** (i.e., **e**.cert# is referenced by **s**)
  - Delete:
    - 4. Delete an Exec tuple when its cert# is presidentC# of 1 or more Studio tuples Recall that cert# is the key of the Exec table

34

**3 types of policies: reject, cascade, set null.**

# Maintaining Referential Integrity

- There are *three* policies, when there is a transaction that violates a referential integrity:
  - The ***reject*** policy *(default)*

    The system will reject any transaction violating referential integrity constraints -- a run-time error will be generated and the database state will *not* change.
  - If we update (3) or delete (4) a referenced item:
    - The ***cascade*** policy: changes to the referenced attributes are "mimicked" at the foreign key (e.g. presidentC#)
    - The "***set null***" policy: set the referencing attribute to NULL (e.g., presidentC# in Studio)

# Maintaining Referential Integrity

**Exec**(eName, address, cert#, netWorth)

**Studio** (sName, address,presidentC#)

Example: For the (default) reject policy, the system will **reject** the following modifications/transactions:

- Insert a new Studio tuple with presidentC# that is not in any Exec tuple
- Update a Studio tuple to change the presidentC# to a non-null value that is not in any Exec tuple
- Update an Exec tuple to change cert#, when the old cert# value is presidentC# in some Studio tuple(s)
- Delete an Exec tuple when its cert# is presidentC# in some Studio tuple(s)

# Maintaining Referential Integrity

**Exec**(eName, address, cert#, netWorth)

**Studio** (sName, address,presidentC#)

- The cascade policy:
  - In case of **deleting** (**updating)** a tuple in a referenced relation (Exec), the DBMS will delete (update) the referencing tuple(s) in Studio
- Example:
  - The system will cascade the following modifications:
    - Update Exec tuple to change cert#, when its cert# is presidentC# in some Studio tuple
      - The system will update presidentC# in corresponding Studio tuple(s)
    - Delete an Exec tuple when its cert# is presidentC# in some Studio tuple
      - The system will delete corresponding tuple(s) in Studio

# Maintaining Referential Integrity

- The set-null policy:
  - In case of **deleting** or **updating** a tuple in a referenced relation (Exec), the DBMS will set to NULL the corresponding values in the referencing tuples (in Studio)
- Example:
  - The system will do the following modifications:
    - Update Exec tuple to change cert#, when its cert# is presidentC# in some Studio tuple
      - The DBMS will set to NULL presidentC# in corresponding Studio tuple(s)
    - Delete an Exec tuple when its cert# is presidentC# in some Studio tuple(s)
      - The DBMS will set to NULL presidentC# in the corresponding Studio tuple(s)

38

# Selecting a Policy

- Reaction options/policies may be chosen (by ?) for deletes and updates, in an *independent* way
- **ON [DELETE | UPDATE] [CASCADE | SET NULL]**

# Selecting a Policy

- Example:

```
CREATE TABLE Studio(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presidentC#  INT,
    FOREIGN KEY presidentC# REFERENCES Exec(cert#)
        ON DELETE SET NULL   (or SET DEFAULT)
        ON UPDATE CASCADE
);
```

# Selecting a Policy

- "Correct" or "right" policy is a design decision
- Example:
  - If a studio president retires (and its tuple gets deleted), the studio should exist with a NULL value for the president and not be deleted along with the president
  - If the certificate number cert# of a studio president was changed (by an update), it should be updated in all corresponding places (presidentC# in Studio, in our case); *we should not lose the information on who is the current president of a given studio*

**SQL & INTEGRITY**
Constraints
- Checks
- Assertions
- Triggers

2 consequences: 1) 2) cannot use set-null policy to fix violation

## Not-Null Constraints

- We can assert that the value of an attribute may not be NULL
- Example:

  presidentC# **INT REFERENCES** Exec(cert#) **NOT NULL**

**Two consequences:**

1. We can't insert a tuple into Studio by just specifying name and address; *the value presidentC# must also be present.*
2. We can't use the "set-null" policy to fix foreign key violations by changing presidentC# to be NULL

# Attribute-Based Checks

- Aside from "referencing", more complex constraints can be attached to an attribute declaration, followed by

  **CHECK (condition-on-attribute)**
- That is, we want the condition to hold for that attribute in every tuple in the relation
- Condition may involve the checked attribute
- Other attributes and relations may be involved, but only in subqueries (see slide #46)
  - **Oracle** : No subqueries allowed in the **condition**

# Attribute-Based Checks

- Example:

```
CREATE TABLE Studio(
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presidentC#  INT
       CHECK (presidentC# >= 100000)
);
```

This requires certificate numbers to be at least 6 digits.

# Attribute-Based Checks

When an attribute-based check/condition is (not) done?

- The condition is checked
  - When the associated attribute changes, i.e., an insert or update occurs
- The condition is **NOT** checked
  - When the relations involved in the subquery of the condition are changed; see the next slide

# Attribute-Based Checks

- Example:

```
CREATE TABLE Studio(
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presidentC#  INT
    CHECK (presidentC# IN (SELECT cert# FROM Exec))
);
```

- Is this check the same as a foreign-key constraint?
  - Not really! The above check is done only when we insert a tuple in **Studio** or change the presidentC# in an existing tuple in **Studio**, **not** when deleting or update a tuple in **Exec**

# Attribute-Based Checks

- Example:

```
CREATE TABLE Star(
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        birthdate DATE,
        gender CHAR(1) CHECK (gender IN ('F', 'M'))
);
```

The above condition uses an explicit set/relation with two tuples, providing possible values for attribute *gender*

# Tuple-Based Checks

- We restrict some components of the tuples of a relation by a **tuple-based check**
- Tuple-based check must appear on a separate element/part in a table declaration command
- Format:

  **CHECK** (condition)
  - Condition may involve any attribute(s) of the table
  - Other attributes and tables may be involved, but only in subqueries
    - **Oracle**: Does not support subqueries in condition

# Modifications of Constraints

- We can add, modify, or delete constraints at any time
- Give names to constraints so we can "refer" to them later

Examples of *defining* constraints:

name **CHAR**(30) **CONSTRAINT** Name-Is-Key **PRIMARY KEY**,

OR

**CONSTRAINT** RightTitle **CHECK** (gender = 'F' **OR** name **NOT LIKE** 'Ms.%')

Examples of *modifying* (deleting/adding) constraints:

**ALTER TABLE** Star **DROP CONSTRAINT** Name-Is-Key;

OR

**ALTER TABLE** Star **ADD CONSTRAINT** NameIsKey **PRIMARY KEY** (*name*);

# Assertions

- *Assertions*, also called "*general constraints*", are boolean-valued SQL expressions that must *always* be satisfied
- Sometimes we need a constraint that involves a relation as a whole or part of the database schema

  (This is not supported in Oracle)
- Assertions are checked when a specified relation changes
- Syntax:

  **CREATE ASSERTION** < assertion-name > **CHECK** (< condition >);

  **Note:** Unlike attribute/tuple based checks, assertions are defined outside table declarations.

51

# Assertions

- Relation schemas:

  **Exec (name, address, cert#, netWorth)**
  **Studio (name, address, presidentC#)**
- Assertion in SQL:

```
CREATE ASSERTION RichPresident CHECK
    (NOT EXIST (SELECT *
                FROM Studio, Exec
                WHERE Studio.presidentC# = Exec.cert# AND
                      Exec.netWorth < 10000000
            )
);
```

- This constraint is checked when **Studio** and/or **Exec** tables change

52

## Triggers

- Also called **event-condition-action** (ECA) rules
- *Event*
  - DB transactions (Insertion, Deletion, Update)
- *Condition*
  - A condition to check if a trigger applies
- *Action*
  - One or more SQL statements to be executed if a triggering event occurred and the condition(s) holds.

53

## Triggers

- Triggers are not supported in SQL2
- Oracle's version and SQL3 version differ from Checks or SQL2 assertions in that:
  - For triggers, event is "programmable", rather than **implied** by the kind of check
  - For checks, we don't have conditions specified
  - (Re)action could be any sequence of database operations

➔Active Database Management Systems (ADBMS)

54

# Triggers

Relation Schema: **Exec(name, address, cert#, networth)**

Example of an SQL trigger:

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec                ← Event
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)    ← [Condition]
   UPDATE Exec                                  ← Action
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#;
```

# Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE Exec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#;
```

- AFTER could be changed to either:
  - BEFORE
  - INSTEAD OF

# Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE Exec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#;
```

- **OF** in **'UPDATE OF'** is optional
  - If present, it defines the event to be only an update of the attribute(s) listed after the keyword **OF**

# Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE Exec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#;
```

- UPDATE (OF whatever) could be changed to:
  - INSERT
  - DELETE

# Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER INSERT ON Exec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE Exec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#;
```

← OLD makes no sense!

- If the event is **INSERT**:
  - **REFERENCING**
    **NEW ROW AS** NewTuple

# Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER DELETE ON Exec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple          ← NEW makes no sense!
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth )
   UPDATE Exec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#;
```

- If event is **DELETE**:
  - **REFERENCING**
    **OLD ROW AS** OldTuple

# Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
 WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE Exec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert# ;
```

- The action could be any sequence of SQL statements, separated by semicolons and embeded in a pair of **BEGIN** and **END**, **e.g.**, **WHEN condition BEGIN S1; … Sk; END;**

# Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE Exec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#;
```

- This trigger is **row-level trigger**
- If we omit "**FOR EACH ROW**", the trigger becomes a **statement-level trigger,** which is the default

62

# Triggers

- If we update an "entire" table with an SQL statement
  - A **row-level trigger** will be executed once for each tuple
  - A **statement-level trigger** will be executed only once for the entire update
- In a statement-level trigger:
  - We can not refer to old and new tuples
  - Instead, we can/should refer to
    - The collection of old tuples as **OLD TABLE**
    - The collection of new tuples as **NEW TABLE**

# Triggers

- **Ex: Constraint: "The *average* net worth of executives may NOT drop below $500,000." This could be violated by updating netWorth, or deleting or inserting tuples from/into Exec.**
- **Below is a trigger for update; similar triggers must be written for delete and insert.**

```
CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
    OLD TABLE AS OldStuff,
    NEW TABLE AS NewStuff
FOR EACH STATEMENT
WHEN (500000 > (SELECT AVG(networth) FROM Exec))
BEGIN
   DELETE FROM Exec
   WHERE (name, address, cert#, netWorth) IN NewStuff ;
   INSERT INTO Exec(SELECT * FROM OldStuff) ;
END;
```

64

# Triggers

**Relation scheme:** Employee(name, <u>empId</u>, salary, dept, supervisorId)

**Constraint:** *No employee gets a salary more than his/her supervisor.*

```
CREATE TRIGGER Inform_supervisor
BEFORE INSERT OR UPDATE OF salary, supervisorId ON Employee
NEW ROW AS new
FOR EACH ROW
WHEN (new.salary > (SELECT salary
                    FROM Employee
                    WHERE empId=new.supervisorId))
Begin
    ROLLBACK;
    Inform_Supervisor(new.supervisorId, new.empId);
End;
```

65

# Instead-Of Triggers

**Relation Scheme: Movie**(title, year, length, filmType, studioName, producerC#)

```
CREATE VIEW ParamountMovie AS
SELECT title, year
FROM Movie
WHERE studioName = 'Paramount';
```

The following *trigger* replaces an insertion on the view (ParamountMovie) with an insertion on its underlying base table (Movie)

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

# A test on Triggers!

Suppose relation **rel(a, b)** has no tuples. Consider the trigger:

```
CREATE TRIGGER T
   AFTER INSERT ON rel
   REFERENCING NEW ROW AS newRow
   FOR EACH ROW
   WHEN (newRow.a * newRow.b > 10)
   INSERT INTO rel VALUES(newRow.a - 1, newRow.b + 1);
```

Inserting which of the following tuples into relation **rel** results in having exactly 2 tuples in **rel**?

1. (3, 5)
2. (4, 3)
3. (3, 4)
4. (5, 3)

67

## April 11, 2018

**DB11** (missed last class)
Datalog is a programming : logical DB query language
Every rule is referred as a query
P(x,y) ← P(x,z), P(z,y) this is a query and it is a view:
We are defining view is defined as already existing views.

Datalog program is a collection of rules/queries

# Datalog

**AlongMovie(Title, Year) ← movie(Title, Year, Length, Type), Length >= 100.**

- The **head** – the left hand side of the arrow/implication
- The **body** – the right hand side is a conjunction (AND) of predicates (called subgoals)
- NOTE: The book uses **AND** in the rule bodies instead of "commas."
- The head is a "positive" predicate (atom) and the subgoals in the rule body are *literals* (an atom or a negated atom)
  - Atom - a formula of the form $p(T_1,\ldots,T_n)$, where **p** is a predicate name and $T_i$'s are *terms*
  - Predicate – **normal** (ordinary) relation name (e.g., movie, p) or **built_in** predicates (e.g., **>=** in the above example)
  - Terms (arguments) – In Datalog, $T_i$ is either a *variable* or a *constant*
  - Subgoals in the rule body may be negated using NOT

# Datalog

longMovie(Title, Year) ← movie(Title, Year, Length, Type), Length >= 100.

- A variable in a rule body is called *local* if it appears only in the rule body, e.g., Length and Type
- The head is true if there are values for local variables that make every subgoal (in the rule body) true
- If the body includes no negation, then the rule can be viewed as a join of relations in the rule body followed by a projection on the head variable(s)

# Datalog

longMovie(Title, Year) ← movie(Title, Year, Length, Type), Length >= 100.

- This rule may be expressed in RA as:

$$\rho_{\text{longMovie}}(\pi_{\text{Title,Year}}(\sigma_{Length \geq 100}(\text{movie})))$$

# Variable-Based Interpretations of Rules

- In principle, given the rule
- $r$ : H ← B1,…,Bk.

  we consider all possible assignments $\mathcal{I}$ of values (constants in the domain) to the variables in the rule.
- Such assignments $\mathcal{I}$ are called interpretations.
- For every interpretation $\mathcal{I}$ of *the rule*, if the body is true under $\mathcal{I}$, we add to the head relation, the tuple defined by H under $\mathcal{I}$.

  (we only consider *ground* interpretations/substitutions).
  - That is, if $\mathcal{I}$(Bi) is true, $\forall$ **i** $\in$ {1,…,k}, then $\mathcal{I}$(H) is true.
  - In this case, we say that "$\mathcal{I}$ *satisfies r" or "*$\mathcal{I}$ *is a model for r"*, *(this is denoted as* $\mathcal{I} \models r$*)*

# Example

s(X, Y) ← r(X, Z), r(Z, Y), **NOT** r(X, Y)**.**

**Instance r:**

| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |

**Instance s:**

| A | B |
|---|---|
| 1 | 3 |

- The only assignments that make the first subgoal true are:

  1. $\mathcal{I}_1$: X → 1, Z → 2
  2. $\mathcal{I}_2$: X → 2, Z → 3.
- In case (1),
  - Y → 3 makes the second subgoal r(Z,Y) true
  - Since (1, 3) $\notin$ r, then "**NOT** r(X,Y)" is also true
  - Thus, we infer tuple (1, 3) for the head relation, s
- In case (2),
  - No value of "Y" makes the second subgoal true

Binary relation it has 2 columns, joining the relation with itself.
2 tuples in r that respects 2 conditions and is NOT r(X,Y) then I am interested in the value X,Y here.

# Tuple-Based Interpretations of Rules

- Consider tuple variables for each positive normal subgoals that range over their relations
  - For each assignment of tuples to each of these subgoals, we determine the implied assignment θ of values to variables
  - If the assignment $\mathcal{I}$ is:
    - consistent and also
    - satisfies all the subgoals (normal and built-ins) in the body

    then we add to the head relation, the tuple defined by the head H under $\mathcal{I}$

# Example

s(X, Y) ← r(X, Z), r(Z, Y), **NOT** r(X, Y).

**Instance r:**

| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |

**Instance s:**

| A | B |
|---|---|
| 1 | 3 |

- Have 4 assignments of tuples to subgoals:

  **r(X,Z) r(Z,Y)**

  1. (1, 2) (1, 2)
  2. (1, 2) (2, 3)
  3. (2, 3) (1, 2)
  4. (2, 3) (2, 3)

- Only the second assignment
  - is consistent for the value assigned to Z and satisfies the negative subgoal "**NOT** r(X,Y)"

  ➔ (1,3) is the only tuple we get for s

# Datalog Programs

- A datalog program is a finite collection of rules
- Note: while standard datalog does not allow negation, in our presentation here, the programs and rules are actually in datalog extended with negation and built-in predicates.
- Predicates/relations can be divided into two classes
  - **EDB** Predicates (input relations), also called ***FACTS***
    - Extensional database = relations stored explicitly in DB
  - **IDB** Predicates (derived/output relations), defined by rule(s)
    - Intensional database
    - They are similar to views in relational databases
- **Note: EDB** predicates appear only in the rule body and **IDB**s appear in the head and possibly in the body

# Operations in Datalog

- The usual set operations
- **Consider relation schemas r(X,Y) and s(X,Y)**
  - **Intersection**
    - RA: $Q = r \cap s$
    - Datalog: $q(X,Y) \leftarrow r(X,Y), s(X,Y).$
  - **Union**
    - RA: $Q = r \cup s$
    - Datalog: the following two rules:
      1. $q(X,Y) \leftarrow r(X, Y)$ .
      2. $q(X,Y) \leftarrow s(X, Y)$ .
  - **Difference**
    - RA: $Q = r - s$
    - Datalog: $q(X,Y) \leftarrow r(X,Y)$, **NOT** $s(X,Y)$.

# Operations in Datalog

- Projection operation
  - RA: $p = \pi_x (r)$
  - Datalog: $p(X) \leftarrow r(X,Y).$

# Operations in Datalog

- Selection operation
  - RA: $s = \sigma_{X > 10 \text{ AND } y = 5}(r)$
  - Datalog: $s(X,Y) \leftarrow r(X,Y), X > 10, Y = 5.$

# Operations in Datalog

- Selection operation. Recall the schema of r(X,Y).
  - RA: $s = \sigma_{X > 10 \text{ OR } y = 5}(r)$
  - Datalog: the following two rules:
    1. $s(X, Y) \leftarrow r(X, Y), X > 10.$
    2. $s(X, Y) \leftarrow r(X, Y), Y = 5.$

    Note: the following Datalog program is equivalent to the above.
    1. $s(A,B) \leftarrow r(A,B), A > 10.$
    2. $s(C,D) \leftarrow r(C,D), D = 5.$

# Operations in Datalog

- Cartesian Product operation
- Consider relation schemas r(A, B) and s(C, D)
  - RA: $Q = r \times s$
  - Datalog: $q(X, Y, Z, W) \leftarrow r(X,Y), s(Z,W).$

# Operations in Datalog

- Join operation
  - Theta-join with an AND condition, e.g., "$c_1$ AND $c_2$"
    - RA: $tj1 = r \bowtie_{X > Z \text{ AND } Y < W} s$
    - Datalog: $tj1(X, Y, Z, W) \leftarrow r(X,Y), s(Z,W), X > Z, Y < W.$
  - Theta-join with an OR condition, e.g., "$c_1$ OR $c_2$"
    - RA: $tj2 = r \bowtie_{X > Z \text{ OR } Y < W} s$
    - Datalog: the following two rules:

$$tj2(X, Y, Z, W) \leftarrow r(X,Y), s(Z, W), X > Z.$$
$$tj2(X, Y, Z, W) \leftarrow r(X,Y), s(Z,W), Y < W.$$

# Operations in Datalog

- Join operation
  - Equi-join
    - RA: $ej3 = r \bowtie_{Y=Z} s$
    - Datalog: ej3(X,Y,Z,W) ← r(X,Y), s(Z,W), Y = Z.

      OR even better (simpler):

      ej3(X,Y,Y,W) ← r(X,Y), s(Y,W).

# Operations in Datalog

- Join operation
  - Natural join
    - RA: $nj4 = r \bowtie s$
    - Datalog: nj4(X,Y,W) ← r(X,Y), s(Y,W).

# Example:Datalog Queries/Programs

- Database schema:
  **movie**( Title, Year, Length, FilmType, StudioName)
  **starsIn**( Title, Year, StarName)
- Query: Find the **names** of stars of movies that are at least 100 minutes long
- Relational Algebra Expression:
  $Q = \pi_{starName} (\sigma_{length \geq 100} (movie) \bowtie starsIn)$
- Datalog program:
  r1(Title, Year, Length, Type, Studio) ←
  movie(Title, Year, Length, Type, Studio), Length >= 100.
  r2(Title, Year, Length, Type, Studio, Name) ←
  r1(Title, Year, Length, Type, Studio), starsIn(Title, Year, Name).
  q(Name) ← r2(Title, Year, Length, Type, Studio, Name).

  As in RA case, we could express this query using just one rule, as follows:
  q(Name) ← movie(Title, Year, Length, Type, Studio), Length >= 100,
  starsIn(Title, Year, Name).

# Expressive Power of Datalog

- Relational algebra = Nonrecursive Datalog+ negation
- Datalog can express SQL **SELECT-FROM-WHERE** statements that do **not** use aggregation and/or grouping
- The SQL-99 standard supports recursion but it is not part of the "core" SQL-99 standard that every DBMS should support
- Some DBMS implementations, e.g. DB2, support *linear recursion*

# Example

trainSchedule(From,To)

- Datalog:
  - fromMontreal(C) ← trainSchedule('Montreal', C).
  - fromMontreal(TC) ← fromMontreal(C), trainSchedule(C,TC).

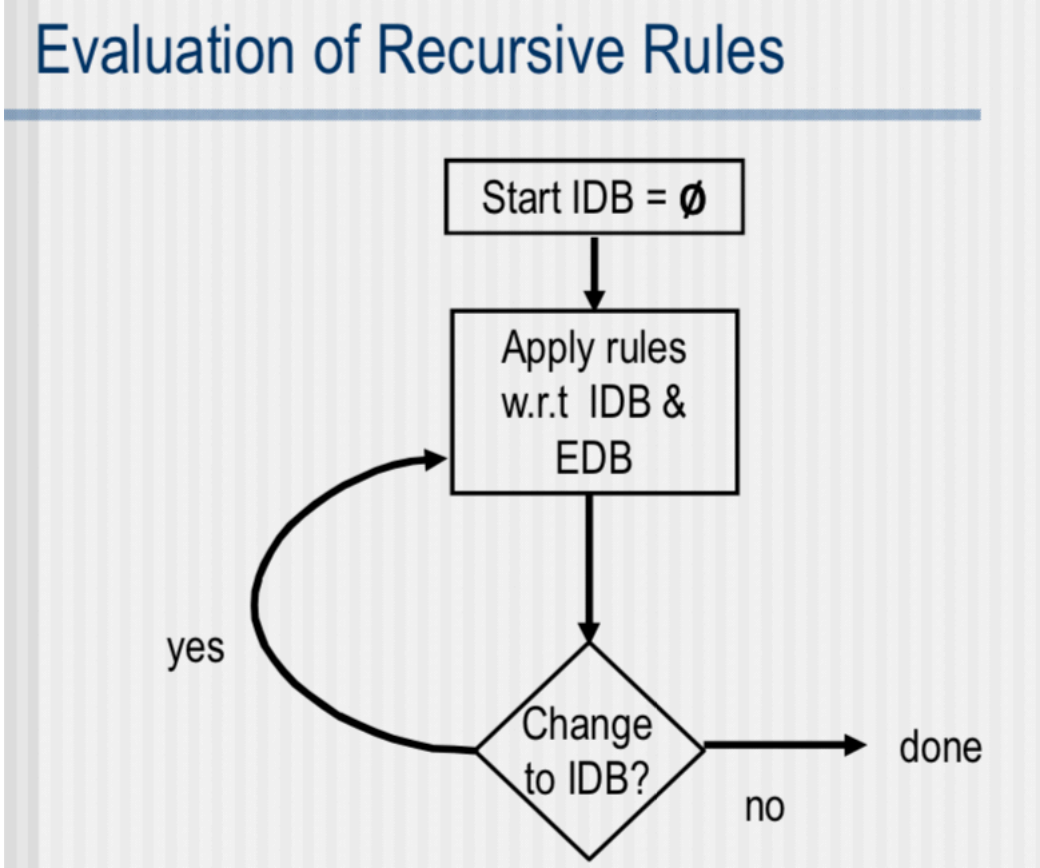
- Find every cities that can be reached from Montreal by train

**trainSchedule:**

| From | To |
|---|---|
| Toronto | Calgary |
| NYC | Boston |
| NYC | Albany |
| Chicago | Detroit |
| Montreal | Toronto |
| Montreal | NYC |
| Boston | Quebec City |

**fromMontreal:**

| |
|---|
| Toronto |
| NYC |
| Calgary |
| Boston |
| Albany |
| Quebec City |


Evaluation of Recursive Rules
Start IDB = ø
Apply rules w.r.t IDB & EDB
Change to IDB?
yes
no
done

# Example

**Relation Schema:** **sequelOf**(Movie,Sequel)

**Instance:**

| Movie | Sequel |
|---|---|
| Star wars | Star wars II |
| Naked Gun | Naked Gun 2 |
| Naked Gun 2 | Naked Gun 2 ½ |
| Star wars II | Star wars III |
| Naked Gun 2 ½ | Naked Gun 3 |

**For a given movie, find all the follow-up movies, i.e., a sequel, a sequel of a sequel, and so on**

## April 16, 2018

## Safety

- It is possible to write a rule that makes "no sense".
- Example of such rules:
  - s(X) ← r(Y).
  - s(X) ← **NOT** r(X).
  - s(X) ← r(Y), X < Y.
- In each of these rules, the IDB relation **s** (output relation) could be infinite, even if (the input) relation ***r*** is finite
- Such rules are said to be not ***SAFE***

## Safety

- For a rule to be safe, the following conditions must hold:
  - If a variable **X** appears in the rule head, then **X** must appear in an "ordinary" predicate in the body or be equal to such a variable (directly or indirectly), e.g., **X=Y**, and **Y** appears in an ordinary predicate in the rule body.

  Recall: the predicates could be ordinary or built-in.

## DB12

**OLD**: standard text-based language or describing the structure of DB
ODL is extension of IDL

## ODL

- **ODL** (Object Definition Language) is a standard text-based language for describing the structure of databases
- **ODL** is an extension of **IDL** (Interface Description Language), a component of **CORBA** (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture)

everything thing is an objet: every account, string, etc.
Object ID (OID) is mainly a unique key to connect objects

## Object Oriented World

- In an object oriented design, the "**world**" we want to model is thought of as being **composed of objects**
- Everything is an **object**
  - *people*
  - *bank accounts*
  - *airline flights*
- Every object has a unique object id (OID)
- Every **object** is an **instance** of a **class**
- A **class** simply represents a grouping of **similar objects**
- All objects that are instances of the same class have the same **properties** and **behaviors**

Keyword Class, Name, and properties

## Class Declarations

- A declaration of a **class** in ODL consists of:
  - The keyword **class**
  - The **name** of the class
  - A bracketed { …} list of **properties** of the class

```
class <name> {
    <list of properties>
  };
class Movie {
    …
  };
```

3 types of properties:

- attributes
- relationships (refer to an obj or collection of obj)
- methods (functions that you can use to access obj of a class and manipulate it)

## Properties of ODL classes

- ODL classes can have three kinds of properties:
  - **Attributes**
    - properties whose types are built from **primitive/basic types** such as integers, strings,…
  - **Relationships**
    - properties whose type is either a **reference** to an object or a **collection** of such references
  - **Methods**
    - **functions** that may be applied to objects of the class

## Attributes in ODL

- Attributes are the **simplest kinds** of properties
- An attribute **describes some aspect of an object** by associating, with the object, a value of some simple **type**
- For example, attributes of a **Student** object
  - Student ID
  - Name
  - Address
  - E-mail

the key is given to the collection of object
so we use Movie**s** to refer to the collection of movie objects
title and year are the keys

## Keys in ODL

- In **ODL**, we declare keys using the keyword **key**
  - If a key has more than one attribute, we surround them by (…)
    - Example: (two attributes forming a key)

```
class Movie
        (extent  Movies key (title, year) ) {
        attribute string title;
        …
    };
```

  - If a class has > one key, we may list them all, separated by commas
    - Example: (A class with two keys)

```
class Employee
        (extent  Employees key empID, SIN) {…};
```

Primary key so no 2 tuples with the same value.

So no 2 movie tuples has the same title and year

## Single-Value Constraints in ODL

- Often, we should *enforce* properties in the database saying that there is **at most one** value playing a particular role
  - For example:
    - that a movie object has a **unique** title, year, length, etc
    - that a movie is owned by a **unique** studio

***Basic attribute***: int, date, float, → primitive attribute
You can have ***collection***: array of integers, set of strings, bag, dictionary, set of movie tuples
Relationship: eg which studio is owning this movie, a studio I am using is a Class and not a primitive type. Collection → sets of Movie which is a class

## Single-Value Constraints

- In **ODL:**
  - An attribute is **not** of a collection type
    (Set, Bag, Array, List, Dictionary are **collection types**.)
  - A relationship is either a class type or (a single use of) a collection type constructor applied to a class type.
- Recall that in the **E/R** notation:
  - attributes are **atomic**
  - an arrow (→) can be used to express the multiplicity of relationships (1:1), (1:M), and (N:M)

Basic types: integer, float, string → attributes
Constructors → tuples connected together

# Type system

A **type system** consists of

- **Basic types**
- **Type constructors**
  - recursive rules whereby **complex types** are built from simpler ones

# Basis of types in ODL

- **Primitive types (atomic)**
  - Integer
  - Float
  - Char
  - Character String
  - Boolean
  - Date
  - Enumeration (a **list of names** declared to be **synonyms for integers**
- **Class types**
  - Movie

Dictionary usually used for index
Cannot be repeatedly: cannot have a lists of sets or a sets of bags.
You can only list, set, bags of primitive types.

# Type constructors in ODL

- **Set**
  - Set <integer>
  - Set <Movie>
- **Bag**
  - Bag <integer>
  - Bag <Movie>
- **Array**
  - Array <integer, 10>
  - Array <Movie, 3>
- **Structure**
  - Struct *Address* {string *street*, string *city*}
- **List**
  - List <integer>
  - List <Student>
- **Dictionary** <keyType, valueType>
  - Dictionary<Student, string>

- **Note:**
  - Set, Bag, Array, List and Dictionary are called **collection types**
  - Collection type cannot be applied repeatedly (nested)
    - E.g., it is **illegal** to write Set<Array<integer,10>>

Class Movie
then its properties: attribute type name

# Example

```
class Movie {
    attribute string title;
    attribute integer year;
    attribute integer length;
    attribute enum Film {color, blackAndWhite} filmType;
    };
```

("Gone with the Wind", 1939, 231, color) is a Movie object.

## Example (non-atomic type)

```
class Star {
    attribute string name;
    attribute Struct Address {
        string street,
        Array<char, 10> city
        } homeAddress;
     attribute Address officeAddress;
    };
```

## Example

```
class Student {
    attribute string ID;
    attribute string lastName;
    attribute string firstName;
    attribute date dob;     /* date is a basic type in ODL */
    attribute string program;
    attribute Struct Address {
        string street,
        string city
         } homeAddress;
    };
```

## Example

```
class Course {
    attribute string courseNumber;
    attribute string courseName;
    attribute integer noOfCredits;
    attribute string department;
    };
```

## Relationships in ODL

- If we are designing a database about **Movies** and **Stars**, what are we missing? The relationships….
- How are **Movies** and **Stars** related?
- Every movie has a star (or stars)

# Example

- Can we write " **attribute Star starOf;** " ?

```
class Movie {
    attribute string title;
    attribute integer year;
    attribute integer length;
    attribute enum Film {color, blackAndWhite} filmType;
    attribute Star starOf;
    };
```

- **No**, the attribute types **must not** be classes

# Example

- **starOf** is a relationship between **Movie** and **Star**

```
class Movie {
    attribute string title;
    attribute integer year;
    attribute integer length;
    attribute enum Film {color, blackAndWhite} filmType;
    relationship Star starOf;
    };
```

## Inverse Relationships

- How are **Movies** and **Stars** related?
- Not only every movie has a star but also every star has a role in some movie(s)
- To fix this in the **Star** class, we add the line:

  **relationship Movie starredIn;**

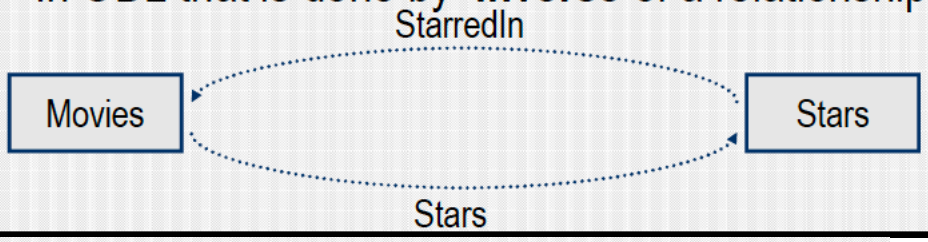
## Example

```
class Star {
    attribute string name;
    attribute Struct Address {
      string street,
      string city
      } address;
    relationship Movie starredIn;
    };
```

- What is the problem here?

# Inverse Relationships

- We are omitting a very important aspect of the relationship between movies and stars
- We need a way to ensure that if a star **S** is connected to a movie **M** via ***stars***, then conversely, **M** is connected to **S** via ***starredIn***
- In ODL that is done by **inverse** of a relationship




# Example

```
class Movie {
    attribute string title;
    attribute integer year;
    attribute integer length;
    attribute enum Film {color, blackAndWhite} filmType;
    relationship Star stars
                    inverse Star::starredIn;
};
```

## Example

```
class Star {
    attribute string name;
    attribute Struct Address {
        string street,
        string city
    } address;
    relationship Movie starredIn
                inverse Movie::stars;
};
```

## Relationships in ODL

- Our design is missing another important point!
- A movie typically has several stars
- A star usually plays in more than one movie
- To fix this, we write:

**relationship Set<**Star**>** stars**;**

## Example

```
class Movie {
    attribute string title;
    attribute integer year;
    attribute integer length;
    attribute enum Film {color, blackAndWhite} filmType;
    relationship Set<Star> stars
                    inverse Star::starredIn;
};
```

## Example

```
class Star {
    attribute string name;
    attribute Struct Address {
        string street,
        string city
    } address;
    relationship Set<Movie> starredIn
                    inverse Movie::stars;
};
```

## Example

- Suppose we introduce another class, **Studio**, representing the studios, i.e., companies that produce movies

```
class Studio {
    attribute string name;
    attribute string address;
    };
```

## Example

- How are **Movies** and **Studios** related?
- Every **Studio** owns several **Movies**

```
class Studio {
    attribute string name;
    attribute string address;
    relationship Set<Movie> owns
                inverse Movie::ownedBy;
    };
```

# Example

- What about inverse?
- Every **Movie** is owned by some **Studio**

```
class Movie {
    attribute string title;
    attribute integer year;
    attribute integer length;
    attribute enum Film {color, blackAndWhite} filmType;
    relationship Set<Star> stars inverse Star::starredIn;
    relationship Studio ownedBy inverse Studio::owns;
    };
```