

# CSC 261/461 – Database Systems

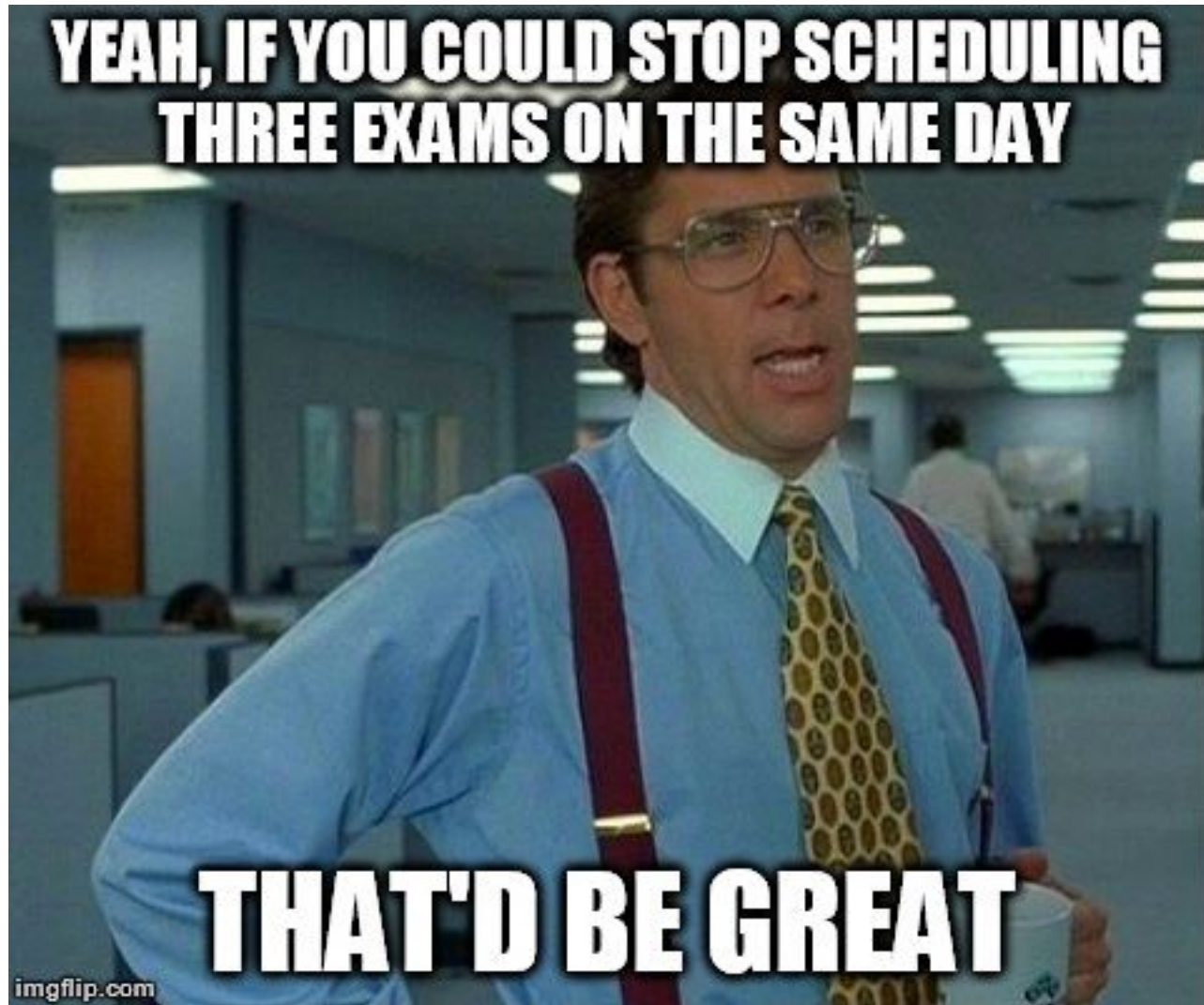
## Lecture 14

Fall 2017

# Announcement

- Midterm on this Wednesday
- Please arrive 5 minutes early; We will start at 12:30 pm sharp
- I will post a sitting arrangement today.
- Bluehive (CIRC) account
- Start studying for MT

Are you ready?



## Night before the exam

# Night before an exam



What my teacher thinks i do



What my parents think i do



What society thinks I do



What my mates think i do



What I think I do



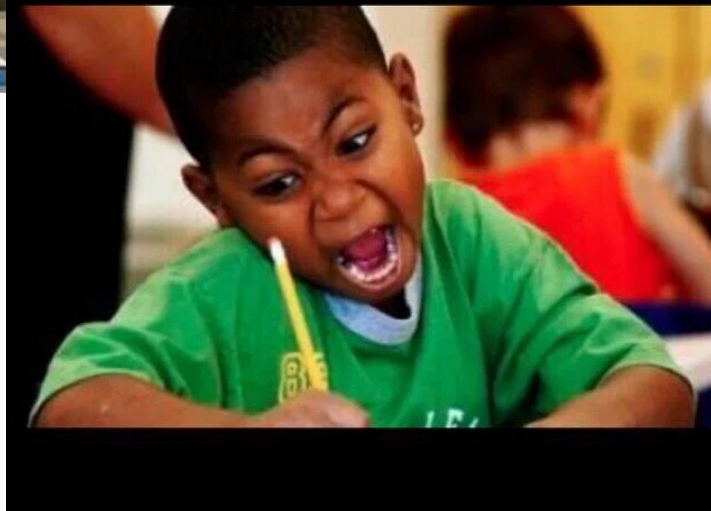
What I actually do



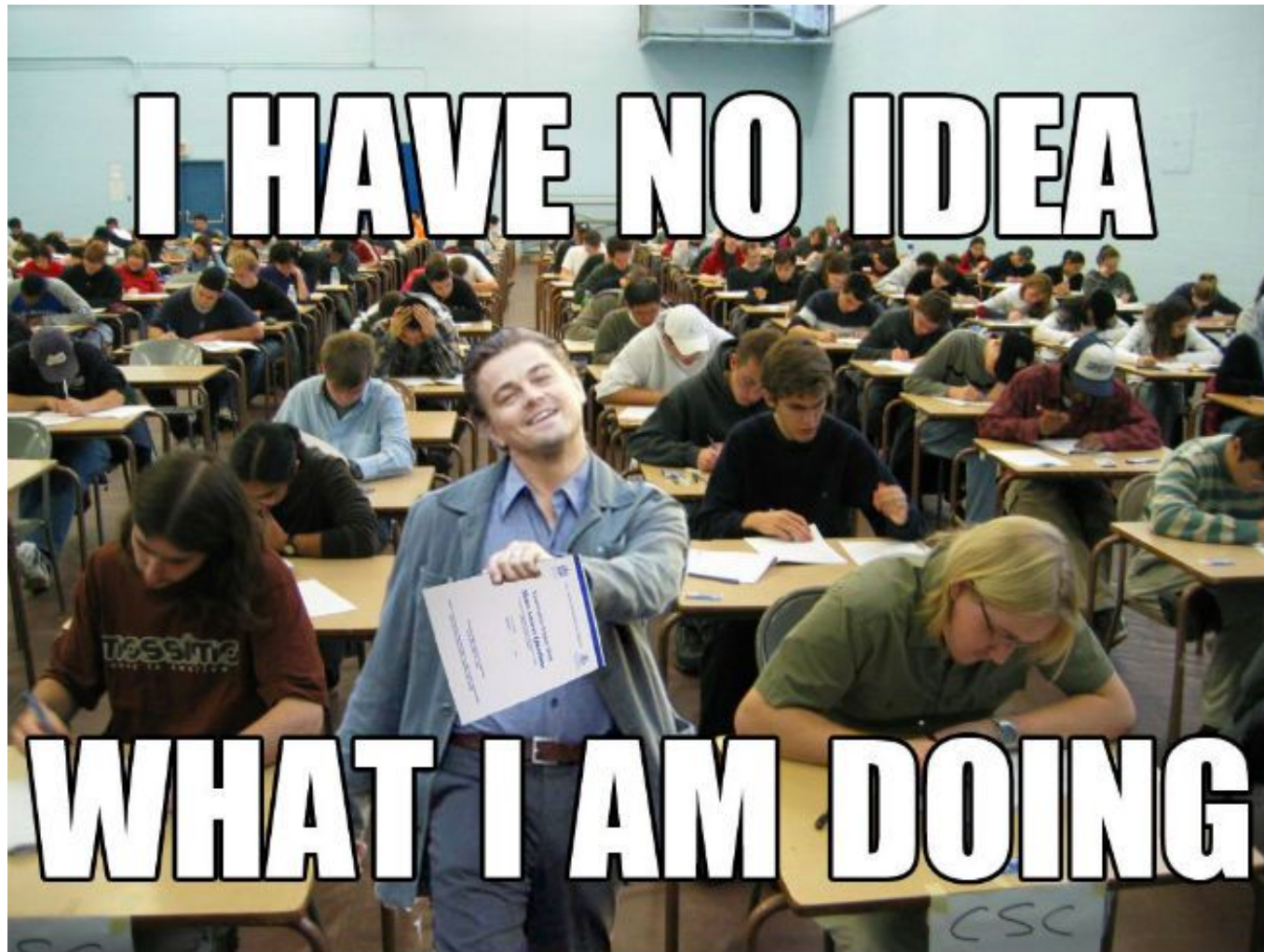
# During the exam



**THE LAST 5 MINUTES OF EXAM**



Or Finishing too early



We Know Memes

# MIDTERM REVIEW

# Chapters to Read

- Chapter 1
- Chapter 2
- Chapter 3
- Chapter 4 (Just the basics. Only ISA relationships. Even studying the slides is fine)
- Chapter 5, 6, 7 (SQL)
- Chapter 8 (8.1-8.3)
- Chapter 9
- Chapter 14 (14.1-14.5)
- Chapter 15 (15.1-15.3)

# Exam Structure

- Problem 1 (20 pts)
  - Short answers, True/False, or One liner
- Other problems (40 pts)
- Total: 60 pts
- 2 bonus points for writing the class id correctly.

# Time distribution

- Time crunch
  - Not really
  - Should take more than 60 minutes to finish.
  - Not as relaxed as the quiz

# **MATERIALS COVERED**



# Questions to ponder

- Why not Lists? Why database?
- How related tables avoid problems associated with lists?

# Problems with Lists

- Multiple Concepts or Themes:
  - Microsoft Excel vs Microsoft Access
- Redundancy
- Anomalies:
  - Deletion anomalies
  - Update anomalies
  - Insertion anomalies

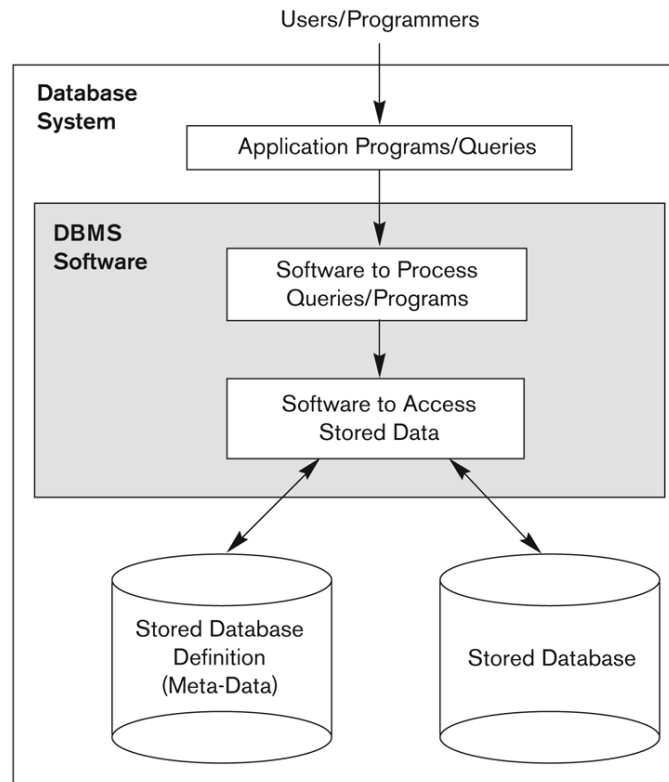
# List vs Database

- Lists do not provide information about relations!
- Break lists into tables
- Facilitates:
  - Insert
  - Delete
  - Update
- Input and Output interface (Forms and Reports)
- Query!

# Again, Why database?

- To store data
- To provide structure
- Mechanism for querying, creating, modifying and deleting data.
- CRED (Create, Read, Update, Delete)
- Store information and relationships
- Database Schema vs. Database State

# Simplified Database System Environment



**Figure 1.1**  
A simplified database  
system environment.

# SQL

# General form SQL

SELECT	S
FROM	$R_1, \dots, R_n$
WHERE	$C_1$
GROUP BY	$a_1, \dots, a_k$
HAVING	$C_2$

Evaluation steps:

1. Evaluate **FROM-WHERE**: apply condition  $C_1$  on the attributes in  $R_1, \dots, R_n$
2. **GROUP BY** the attributes  $a_1, \dots, a_k$
3. **Apply condition  $C_2$  to each group (may have aggregates)**
4. Compute aggregates in S and return the result



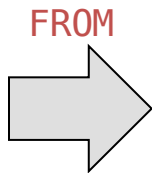
# Grouping and Aggregation

## Semantics of the query:

1. Compute the **FROM** and **WHERE** clauses
2. Group by the attributes in the **GROUP BY**
3. Compute the **SELECT** clause: grouped attributes and aggregates

# 1. Compute the **FROM** and **WHERE** clauses

```
SELECT  product, SUM(price*quantity) AS TotalSales
FROM    Purchase
WHERE   date > '10/1/2005'
GROUP BY product
```



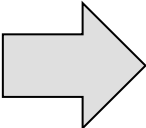
Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

## 2. Group by the attributes in the **GROUP BY**

```
SELECT    product, SUM(price*quantity) AS TotalSales
FROM      Purchase
WHERE     date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

**GROUP BY**



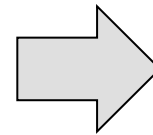
Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

### 3. Compute the **SELECT** clause: grouped attributes and aggregates

```
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

SELECT



Product	TotalSales
Bagel	50
Banana	15

# HAVING Clause

```
SELECT product, SUM(price*quantity)
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING SUM(quantity) > 100
```

Same query as before, except that we consider only products that have more than 100 buyers

HAVING clauses contains conditions on **aggregates**

*Whereas WHERE clauses condition on **individual tuples...***

# Null Values

Unexpected behavior:

```
SELECT *  
FROM Person  
WHERE age < 25 OR age >= 25
```

Some Persons are not included !

# Null Values

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT *  
FROM Person  
WHERE age < 25 OR age >= 25  
      OR age IS NULL
```

Now it includes all Persons!



# Inner Joins

By default, joins in SQL are “inner joins”:

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM   Product
       JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

Both equivalent:  
Both INNER JOINS!

# Inner Joins + NULLS = Lost data?

By default, joins in SQL are “inner joins”:

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM   Product
       JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

However: Products that never sold (with no Purchase tuple) will be lost!

# INNER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, Purchase.store
FROM   Product
       INNER JOIN Purchase
       ON Product.name = Purchase.prodName
```



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Note: another equivalent way to write an INNER JOIN!

# LEFT OUTER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, Purchase.store
FROM   Product
       LEFT OUTER JOIN Purchase
       ON Product.name = Purchase.prodName
```



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

# Other Outer Joins

- Left outer join:
  - Include the left tuple even if there's no match
- Right outer join:
  - Include the right tuple even if there's no match
- Full outer join:
  - Include the both left and right tuples even if there's no match

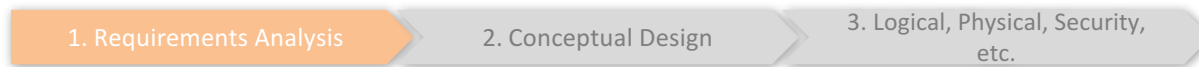
## Also read

- Triggers
- Views
- Operations on Databases
  - Create, Drop, and Alter
- Operations of Tables
  - Insert, Delete and Update
- Constraint:
  - Key constraint
  - Foreign key
  - On Delete cascade, Set Null, Set Default;

# **DATABASE DESIGN**



# Database Design Process

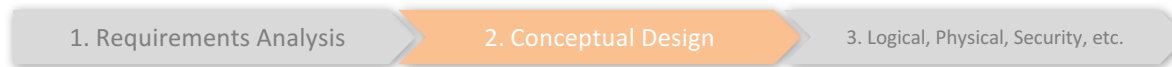


## 1. Requirements analysis

- What is going to be stored?
- How is it going to be used?
- What are we going to do with the data?
- Who should access the data?

Technical and non-technical people are involved

# Database Design Process

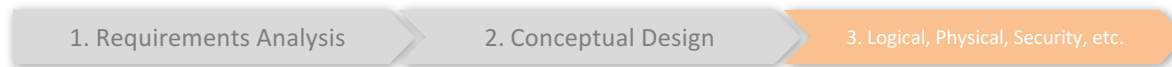


## 2. Conceptual Design

- A high-level description of the database
- Sufficiently precise that technical people can understand it
- But, not so precise that non-technical people can't participate

This is where E/R fits in.

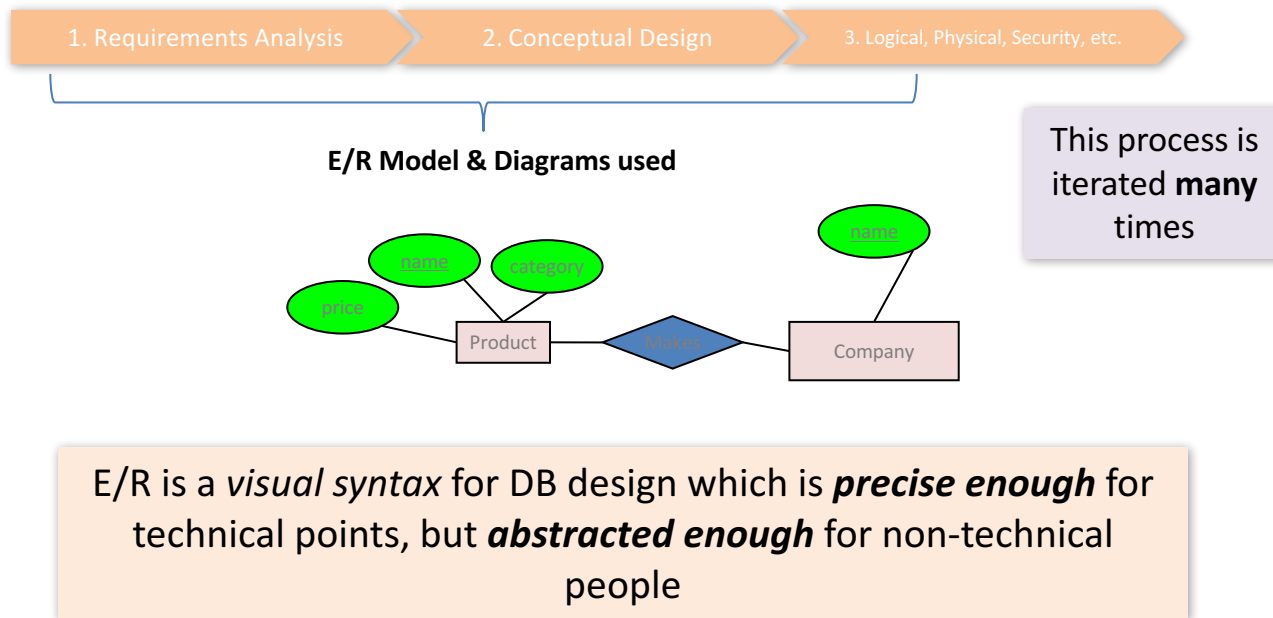
# Database Design Process



## 3. Implementation:

- Logical Database Design
- Physical Database Design
- Security Design

# Database Design Process



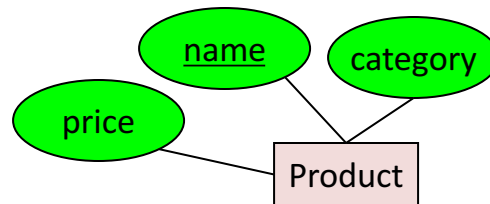
# Requirements Become the E-R Data Model

- After the requirements have been gathered, they are transformed into an Entity Relationship (E-R) Data Model
- E-R Models consist of
  1. Entities
  2. Attributes
    - a) Identifiers (Keys)
    - b) Non-key attributes
  3. Relationships

# **1. E/R BASICS: ENTITIES & RELATIONS**

# Entities and Entity Sets

- An entity set has **attributes**
  - Represented by ovals attached to an entity set

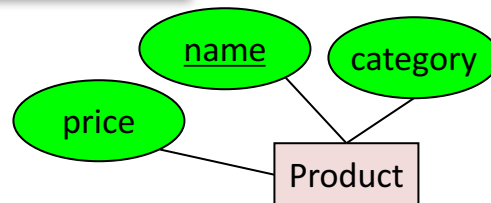


Shapes are important. Colors are not.

# Keys

- A key is a **minimal** set of attributes that uniquely identifies an entity.

Denote elements of the primary key by underlining.



Here, {name, category} is not a key (it is not *minimal*).

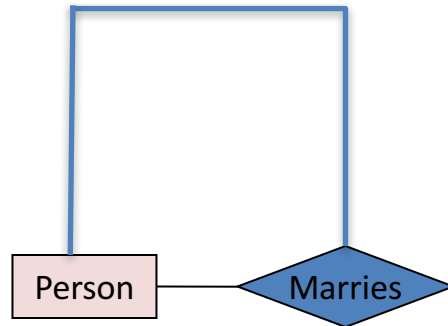
The E/R model forces us to designate a single **primary** key, though there may be multiple candidate keys



# Relationships

- A **relationship** connects two or more entity sets.
- It is represented by a **diamond**, with lines to each of the entity sets involved.
- The *degree* of the relationship defines the number of entity classes that participate in the relationship
  - Degree 1 is a **unary** relationship
  - Degree 2 is a **binary** relationship
  - Degree 3 is a **ternary** relationship

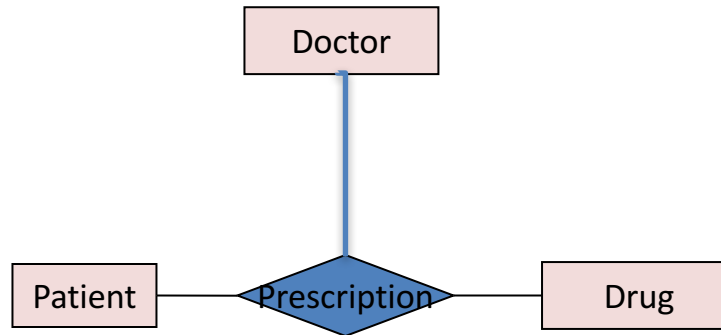
# Conceptual Unary Relationship



# Conceptual Binary Relationship

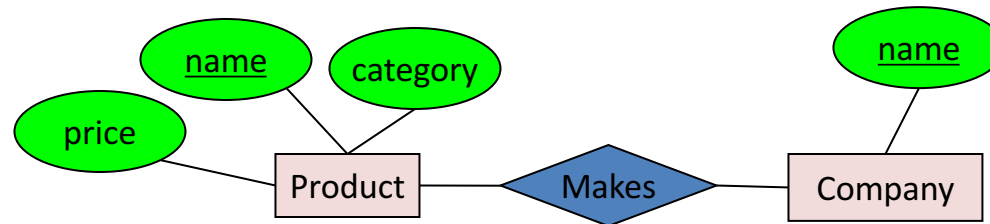


# Conceptual Ternary Relationship

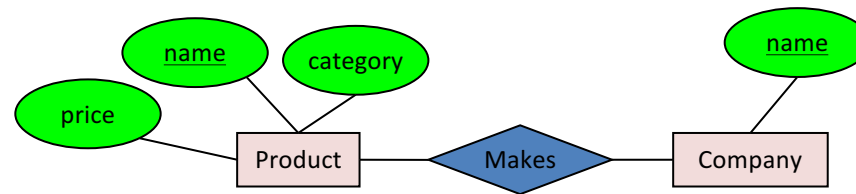


# The R in E/R: Relationships

- E, R and A together:



# What is a Relationship?



A relationship between entity sets **P** and **C** is a *subset of all possible pairs of entities in P and C*, with tuples uniquely identified by **P and C's keys**

# What is a Relationship?

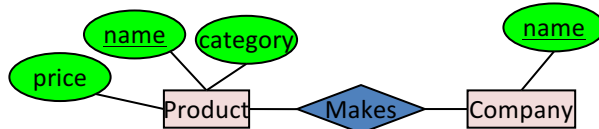
Company	
name	
GizmoWorks	
GadgetCorp	

Product			
name	category	price	
Gizmo	Electronics	\$9.99	
GizmoLite	Electronics	\$7.50	
Gadget	Toys	\$5.50	



Company C × Product P

C.name	P.name	P.category	P.price
GizmoWorks	Gizmo	Electronics	\$9.99
GizmoWorks	GizmoLite	Electronics	\$7.50
GizmoWorks	Gadget	Toys	\$5.50
GadgetCorp	Gizmo	Electronics	\$9.99
GadgetCorp	GizmoLite	Electronics	\$7.50
GadgetCorp	Gadget	Toys	\$5.50



A **relationship** between entity sets **P** and **C** is a **subset of all possible pairs of entities in P and C**, with tuples uniquely identified by **P and C's keys**

# What is a Relationship?

**Company**

<u>name</u>
GizmoWorks
GadgetCorp

**Product**

<u>name</u>	category	price
Gizmo	Electronics	\$9.99
GizmoLite	Electronics	\$7.50
Gadget	Toys	\$5.50



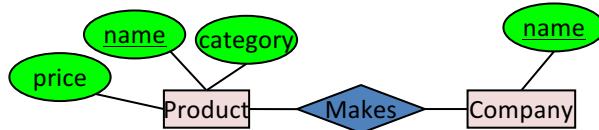
**Company C × Product P**

<u>C.name</u>	<u>P.name</u>	P.category	P.price
GizmoWorks	Gizmo	Electronics	\$9.99
GizmoWorks	GizmoLite	Electronics	\$7.50
GizmoWorks	Gadget	Toys	\$5.50
GadgetCorp	Gizmo	Electronics	\$9.99
GadgetCorp	GizmoLite	Electronics	\$7.50
GadgetCorp	Gadget	Toys	\$5.50



**Makes**

<u>C.name</u>	<u>P.name</u>
GizmoWorks	Gizmo
GizmoWorks	GizmoLite
GadgetCorp	Gadget

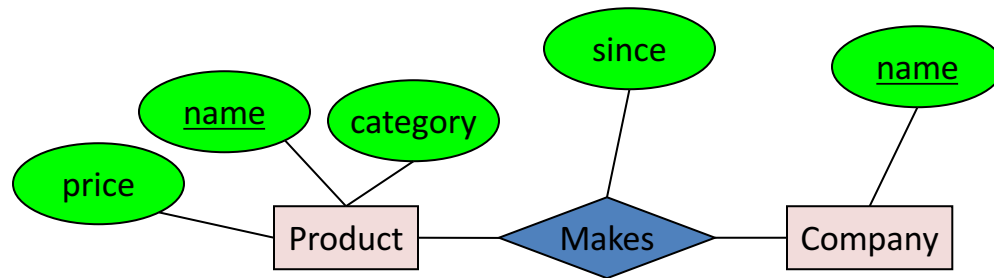


A **relationship** between entity sets **P** and **C** is a **subset of all possible pairs of entities in P and C**, with tuples uniquely identified by **P and C's keys**



# Relationships and Attributes

- Relationships may have attributes as well.








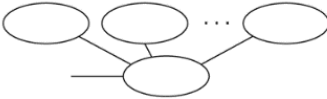



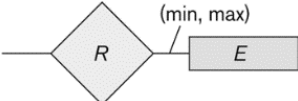


For example: “since” records when company started making a product

Note: “since” is implicitly unique per pair here! Why?

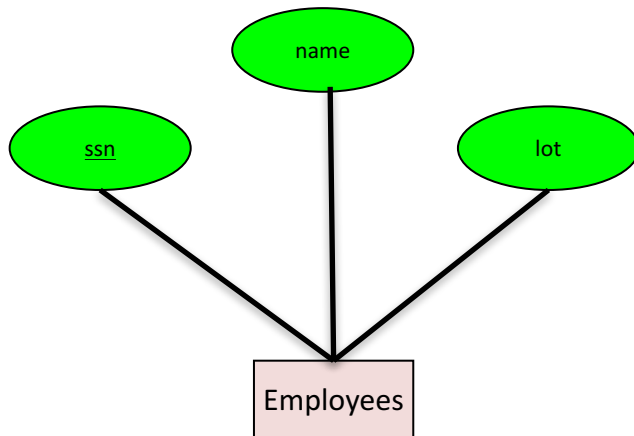
- Summary

**Figure 3.14**  
Summary of the  
notation for ER  
diagrams.

Symbol	Meaning
	Entity
	Weak Entity
	Relationship
	Identifying Relationship
	Attribute
	Key Attribute
	Multivalued Attribute
	Composite Attribute
	Derived Attribute
	Total Participation of $E_2$ in $R$
	Cardinality Ratio 1: N for $E_1:E_2$ in $R$
	Structural Constraint (min, max) on Participation of $E$ in $R$

# Design Theory (ER model to Relations)

# Entity Sets to Tables



ssn	name	lot
123-22-3333	Alex	23
234-44-6666	Bob	44
567-88-9787	John	12

```
CREATE TABLE Employees (    ssn char(11),
                             name varchar(30),
                             lot Integer,
                             PRIMARY KEY (ssn))
```

# Other Conversions (ER model to Tables)

- Relationships:
- Many to Many
- One to Many
- One to One

# Scenario

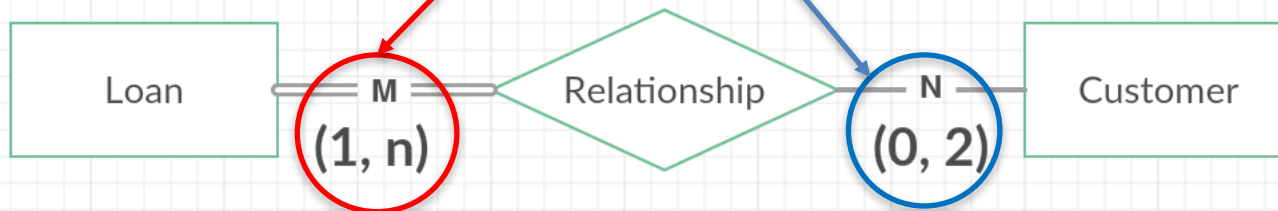
- One customer can have **at max 2 loans**. One loan can be given to **multiple** customers.

What it really means:

- One customer can have **(0,2)** loans
- One loan can be given to **(1,n)** customer
- This is a many to many scenario



Crow's foot Notation



Chen Notation

# FUNCTIONAL DEPENDENCIES



# Prime and Non-prime attributes

- A **Prime** attribute must be a member of *some* candidate key
- A **Nonprime** attribute is not a prime attribute—that is, it is not a member of any candidate key.

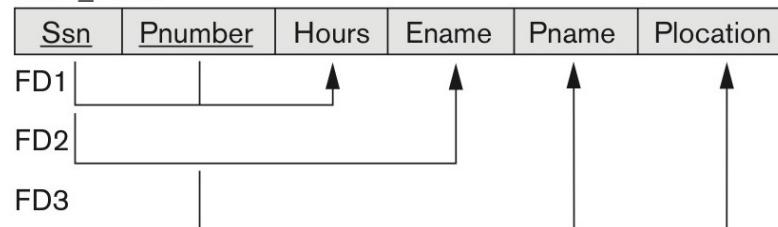
(a)

EMP\_DEPT



(b)

EMP\_PROJ



# Back to Conceptual Design

Now that we know how to find FDs, it's a straight-forward process:

1. Search for “bad” FDs
2. If there are any, then *keep decomposing the table into sub-tables* until no more bad FDs
3. When done, the database schema is *normalized*

# Boyce-Codd Normal Form (BCNF)

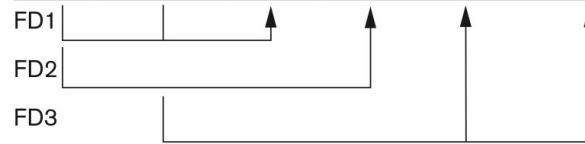
- Main idea is that we define “good” and “bad” FDs as follows:
  - $X \rightarrow A$  is a “*good FD*” if  $X$  is a (*super*)*key*
    - In other words, if  $A$  is the set of all attributes
  - $X \rightarrow A$  is a “*bad FD*” otherwise
- We will try to eliminate the “bad” FDs!
  - Via normalization

# Normalizing into 2NF and 3NF

(a)

**EMP\_PROJ**

<u>Ssn</u>	<u>Pnumber</u>	Hours	Ename	Pname	Plocation
------------	----------------	-------	-------	-------	-----------



2NF Normalization

**EP1**

<u>Ssn</u>	<u>Pnumber</u>	Hours
------------	----------------	-------



**EP2**

<u>Ssn</u>	Ename
------------	-------



**EP3**

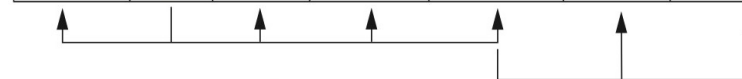
<u>Pnumber</u>	Pname	Plocation
----------------	-------	-----------



(b)

**EMP\_DEPT**

Ename	<u>Ssn</u>	Bdate	Address	<u>Dnumber</u>	Dname	Dmgr_ssn
-------	------------	-------	---------	----------------	-------	----------



3NF Normalization

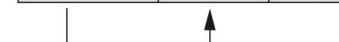
**ED1**

Ename	<u>Ssn</u>	Bdate	Address	<u>Dnumber</u>
-------	------------	-------	---------	----------------



**ED2**

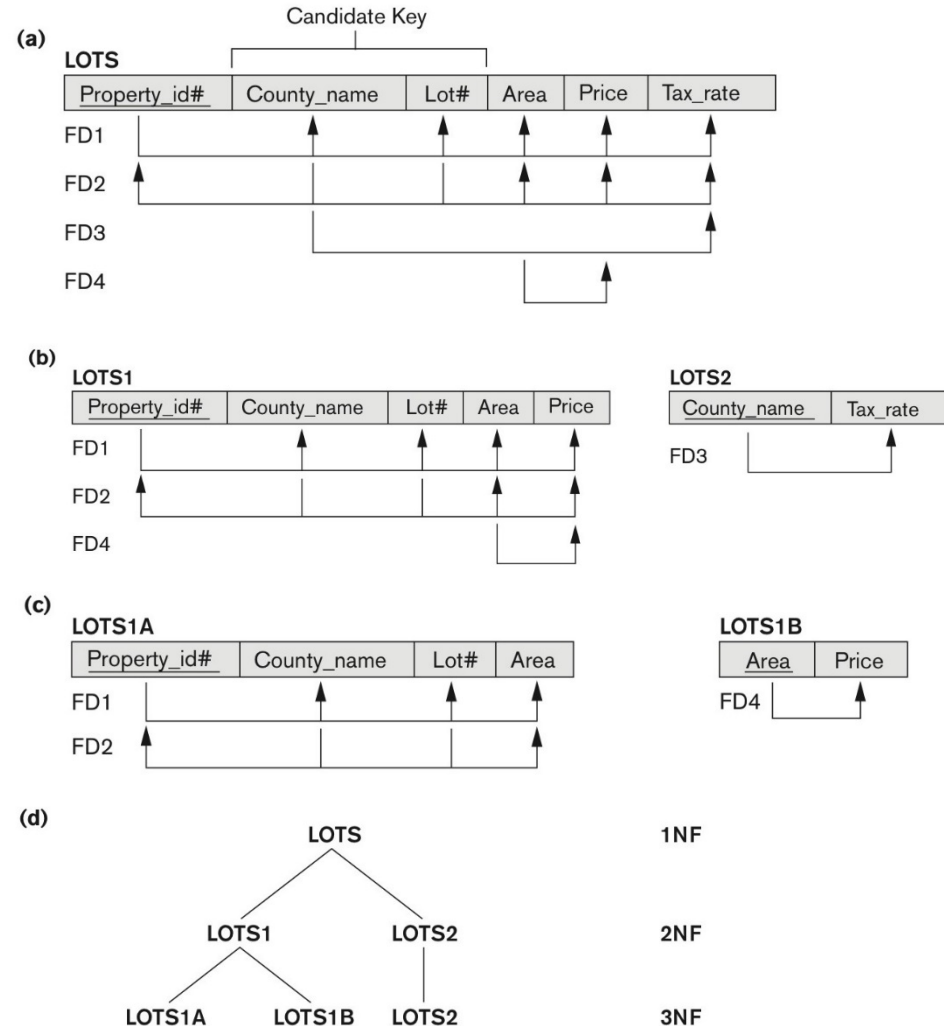
<u>Dnumber</u>	Dname	Dmgr_ssn
----------------	-------	----------



# Figure 14.12 Normalization into 2NF and 3NF

**Figure 14.12**

Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Progressive normalization of LOTS into a 3NF design.



# Normal Forms Defined Informally

- 1<sup>st</sup> normal form
  - All attributes depend on **the key**
- 2<sup>nd</sup> normal form
  - All attributes depend on **the whole key**
- 3<sup>rd</sup> normal form
  - All attributes depend on **nothing but the key**

## General Definition of 2NF and 3NF (For Multiple Candidate Keys)

- A relation schema  $R$  is in **second normal form (2NF)** if every non-prime attribute  $A$  in  $R$  is fully functionally dependent on *every* key of  $R$
- A relation schema  $R$  is in **third normal form (3NF)** if it is in 2NF *and* no non-prime attribute  $A$  in  $R$  is transitively dependent on *any* key of  $R$

# 1. BOYCE-CODD NORMAL FORM



## Figure 14.14 A relation TEACH that is in 3NF but not in BCNF

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

- Two FDs exist in the relation TEACH:
  - fd1: { student, course}  $\rightarrow$  instructor
  - fd2: instructor  $\rightarrow$  course
- {student, course} is a candidate key for this relation
- So this relation is in 3NF *but not in* BCNF
- A relation **NOT** in BCNF should be decomposed so as to meet this property,
  - while possibly forgoing the preservation of all functional dependencies in the decomposed relations.

# Achieving the BCNF by Decomposition (2)

- Three possible decompositions for relation TEACH
  - D<sub>1</sub>: {student, instructor} and {student, course}
  - D<sub>2</sub>: {course, instructor} and {course, student}
  - D<sub>3</sub>: {instructor, course} and {instructor, student} ✓

## 4.3 Interpreting the General Definition of Third Normal Form (2)

■ **ALTERNATIVE DEFINITION of 3NF:** We can restate the definition as:

A relation schema  $R$  is in **third normal form (3NF)** if, whenever a nontrivial FD  $X \rightarrow A$  holds in  $R$ , either

- a)  $X$  is a superkey of  $R$  or
- b)  $A$  is a prime attribute of  $R$

The condition (b) takes care of the dependencies that “slip through” (are allowable to) 3NF but are “caught by” BCNF which we discuss next.

# 1. BOYCE-CODD NORMAL FORM

# What you will learn about in this section

1. Boyce-Codd Normal Form
2. The BCNF Decomposition Algorithm

## 5. BCNF (Boyce-Codd Normal Form)

- **Definition of 3NF:**
- A relation schema  $R$  is in **3NF** if, whenever a nontrivial FD  $X \rightarrow A$  holds in  $R$ , either
  - a)  $X$  is a superkey of  $R$  or
  - b)  $A$  is a prime attribute of  $R$
- A relation schema  $R$  is in **Boyce-Codd Normal Form (BCNF)** if whenever an FD  $X \rightarrow A$  holds in  $R$ , then
  - a)  $X$  is a superkey of  $R$
  - ~~b) There is no b~~
- Each normal form is strictly stronger than the previous one
  - Every 2NF relation is in 1NF
  - Every 3NF relation is in 2NF
  - Every BCNF relation is in 3NF

# Boyce-Codd normal form

(a)

LOTS1A

<u>Property_id#</u>	County_name	Lot#	Area
---------------------	-------------	------	------



BCNF Normalization

LOTS1AX

<u>Property_id#</u>	Area	Lot#
---------------------	------	------

LOTS1AY

<u>Area</u>	County_name
-------------	-------------

(b)

R

<u>A</u>	<u>B</u>	C
----------	----------	---



**Figure 14.13**

Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF due to the f.d.  $C \rightarrow B$ .

# A relation TEACH that is in 3NF but not in BCNF

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

- Two FDs exist in the relation TEACH:

X

→ A

- {student, course} → instructor
- instructor → course

- {student, course} is a candidate key for this relation
- So this relation is in 3NF *but not in BCNF*
- A relation NOT in BCNF should be decomposed



# Achieving the BCNF by Decomposition

- Three possible decompositions for relation TEACH
  - D1: {student, instructor} and {student, course}
  - D2: {course, instructor} and {course, student}
  - ✓ D3: {instructor, course} and {instructor, student}

# Boyce-Codd Normal Form

BCNF is a simple condition for removing anomalies from relations:

A relation  $R$  is in BCNF if:

if  $\{X_1, \dots, X_n\} \rightarrow A$  is a *non-trivial* FD in  $R$   
then  $\{X_1, \dots, X_n\}$  is a **superkey** for  $R$

In other words: there are no “bad” FDs

# Example

Name	SSN	PhoneNumber	City
Fred	123-45-6789	206-555-1234	Seattle
Fred	123-45-6789	206-555-6543	Seattle
Joe	987-65-4321	908-555-2121	Westfield
Joe	987-65-4321	908-555-1234	Westfield

$\{SSN\} \rightarrow \{Name, City\}$

This FD is *bad*  
because it is **not** a  
superkey

$\Rightarrow$  **Not** in BCNF

*What is the key?*  
 $\{SSN, PhoneNumber\}$

# Example

Name	<u>SSN</u>	City
Fred	123-45-6789	Seattle
Joe	987-65-4321	Madison

<u>SSN</u>	<u>PhoneNumber</u>
123-45-6789	206-555-1234
123-45-6789	206-555-6543
987-65-4321	908-555-2121
987-65-4321	908-555-1234

$\{SSN\} \rightarrow \{Name, City\}$

This FD is now  
*good* because it is  
the key

Let's check anomalies:

- Redundancy ?
- Update ?
- Delete ?

Now in BCNF!

# BCNF Decomposition

BCNFDecomp(R):

If  $X \rightarrow A$  causes BCNF violation:

Decompose R into

$R_1 = XA$

$R_2 = R - A$

(Note: X is present in both  $R_1$  and  $R_2$ )

Return BCNFDecomp( $R_1$ ), BCNFDecomp( $R_2$ )

# Example

BCNFDecomp(R):

If  $X \rightarrow A$  causes BCNF violation:

Decompose R into

$R_1 = XA$

$R_2 = R - A$

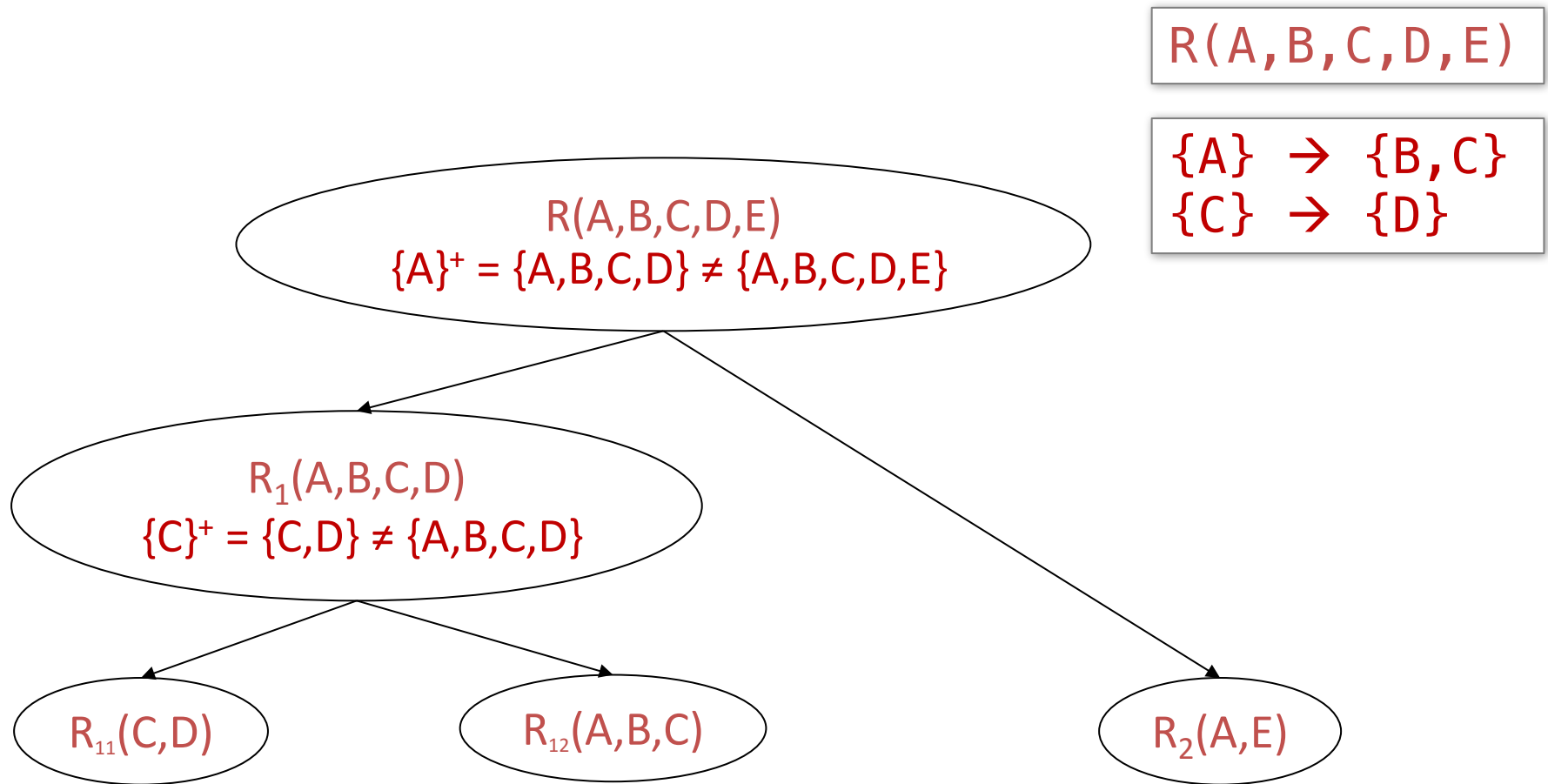
(Note: X is present in both  $R_1$  and  $R_2$ )

Return BCNFDecomp( $R_1$ ),  
BCNFDecomp( $R_2$ )

$R(A, B, C, D, E)$

$\{A\} \rightarrow \{B, C\}$   
 $\{C\} \rightarrow \{D\}$

# Example



## **2. DECOMPOSITIONS**

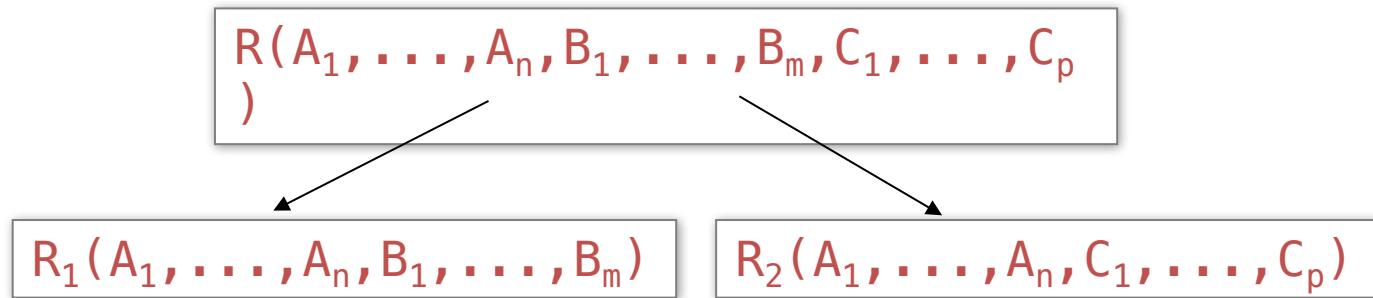


# Recap: Decompose to remove redundancies

1. We saw that **redundancies** in the data (“bad FDs”) can lead to data anomalies
2. We developed mechanisms to **detect and remove redundancies by decomposing tables into BCNF**
  1. BCNF decomposition is *standard practice*- very powerful & widely used!
3. However, sometimes decompositions can lead to **more subtle unwanted effects...**

When does this happen?

# Decompositions in General



$R_1$  = the *projection* of  $R$  on  $A_1, \dots, A_n, B_1, \dots, B_m$


$R_2$  = the *projection* of  $R$  on  $A_1, \dots, A_n, C_1, \dots, C_p$

# Theory of Decomposition


Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

Sometimes a decomposition is “correct”

I.e. it is a **Lossless decomposition**



Name	Price
Gizmo	19.99
OneClick	24.99
Gizmo	19.99




Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

# Lossy Decomposition

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

*However  
sometimes it isn't*

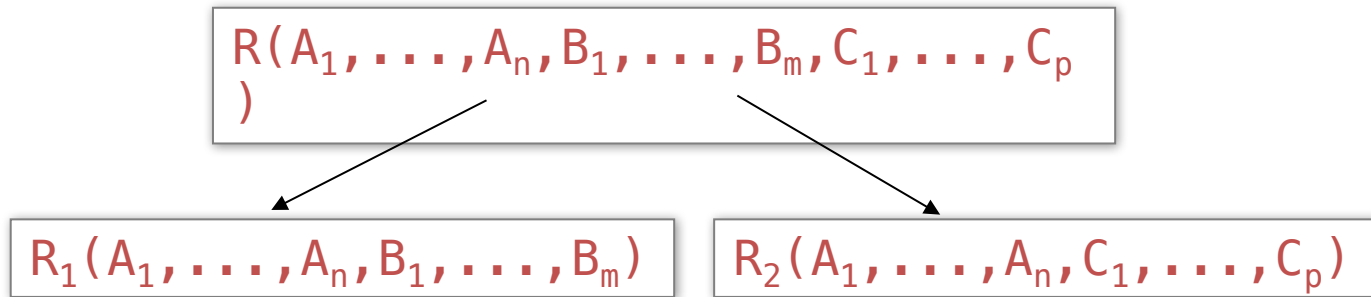
What's wrong  
here?



Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

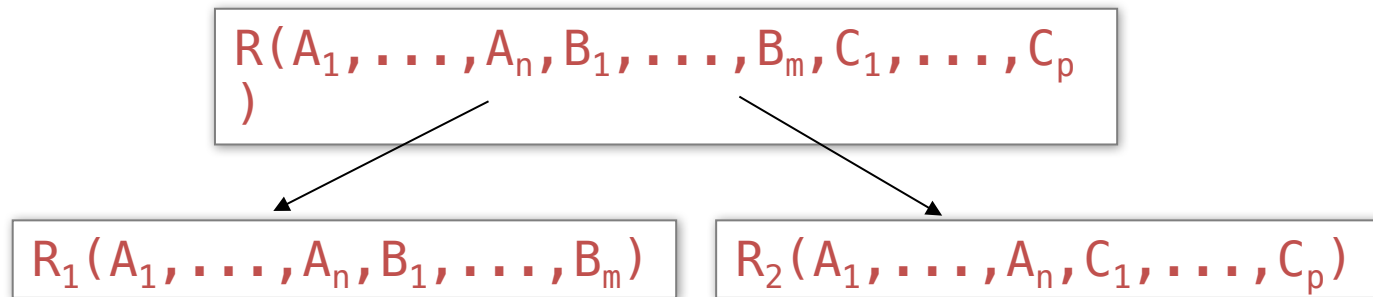
Price	Category
19.99	Gadget
24.99	Camera
19.99	Camera

# Lossless Decompositions



A decomposition  $R$  to  $(R_1, R_2)$  is **lossless** if  $R = R_1 \text{ Join } R_2$

# Lossless Decompositions



If  $\{A_1, \dots, A_n\} \rightarrow \{B_1, \dots, B_m\}$   
Then the decomposition is lossless

Note: don't need  $\{A_1, \dots, A_n\} \rightarrow \{C_1, \dots, C_p\}$

BCNF decomposition is always lossless. Why?

# A relation TEACH that is in 3NF but not in BCNF

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

- Two FDs exist in the relation TEACH:

**X**

**→ A**

- {student, course} → instructor
- instructor → course

- {student, course} is a candidate key for this relation
- So this relation is in 3NF *but not in* BCNF
- A relation **NOT** in BCNF should be decomposed

# Achieving the BCNF by Decomposition (2)

- Three possible decompositions for relation TEACH
  - D1: {student, instructor} and {student, course}
  - D2: {course, instructor} and {course, student}
  - ✓ D3: {instructor, course} and {instructor, student}



## A problem with BCNF

Problem: To enforce a FD, must reconstruct original relation—*on each insert!*

# A Problem with BCNF

Unit	Company	Product
...	...	...

<u>Unit</u>	Company
...	...

Unit	Product
...	...

$\{\text{Unit}\} \rightarrow \{\text{Company}\}$

$\{\text{Unit}\} \rightarrow \{\text{Company}\}$   
 $\{\text{Company}, \text{Product}\} \rightarrow \{\text{Unit}\}$

We do a BCNF decomposition  
on a “bad” FD:  
 $\{\text{Unit}\}^+ = \{\text{Unit}, \text{Company}\}$

We lose the FD  $\{\text{Company}, \text{Product}\} \rightarrow \{\text{Unit}\}!!$

## So Why is that a Problem?

<u>Unit</u>	Company
Galaga99	UW
Bingo	UW

Unit	Product
Galaga99	Databases
Bingo	Databases

No problem so far.  
All *local* FD's are satisfied.

$\{\text{Unit}\} \rightarrow \{\text{Company}\}$

Unit	Company	Product
Galaga99	UW	Databases
Bingo	UW	Databases

Let's put all the data back into a single table again:

Violates the FD  $\{\text{Company}, \text{Product}\} \rightarrow \{\text{Unit}\}!!$

# The Problem

- We started with a table  $R$  and FDs  $F$
- We decomposed  $R$  into BCNF tables  $R_1, R_2, \dots$  with their own FDs  $F_1, F_2, \dots$
- We insert some tuples into each of the relations—which satisfy their local FDs but when reconstruct it violates some FD **across** tables!

Practical Problem: To enforce FD, must reconstruct  $R$ —*on each insert!*

# Possible Solutions

- Various ways to handle so that decompositions are all lossless / no FDs lost
  - For example 3NF- stop short of full BCNF decompositions.
- Usually a tradeoff between redundancy / data anomalies and FD preservation...

BCNF still most common- with additional steps to keep track of lost FDs...

# Other Topics

- **Problem Set 5** (Really important)
  - Cover
  - Minimal Cover
  - BCNF violations and Decomposition

# RELATIONAL ALGEBRA & CALCULUS

# 1. Selection ( $\sigma$ )

Students(sid, sname, gpa)

- Returns all tuples which satisfy a condition
- Notation:  $\sigma_c(R)$
- Examples
  - $\sigma_{\text{Salary} > 40000}(\text{Employee})$
  - $\sigma_{\text{name} = \text{"Smith"}}(\text{Employee})$
- The condition c can be =, <, ≤, >, ≥, <>

SQL:

```
SELECT *  
FROM Students  
WHERE gpa > 3.5;
```



RA:

$\sigma_{gpa > 3.5}(\text{Students})$



Another example:

SSN	Name	Salary
1234545	John	200000
5423341	Smith	600000
4352342	Fred	500000

$\sigma_{\text{Salary} > 40000}$  (Employee)



SSN	Name	Salary
5423341	Smith	600000
4352342	Fred	500000

## 2. Projection ( $\Pi$ )

- Eliminates columns, then removes duplicates
- Notation:  $\Pi_{A_1, \dots, A_n}(R)$
- Example: project social-security number and names:
  - $\Pi_{SSN, Name}(Employee)$
  - Output schema: Answer(SSN, Name)

Students(sid, sname, gpa)

SQL:

```
SELECT DISTINCT  
    sname,  
    gpa  
FROM Students;
```



RA:

$\Pi_{sname, gpa}(Students)$

Another example:

SSN	Name	Salary
1234545	John	200000
5423341	John	600000
4352342	John	200000

$\Pi_{\text{Name,Salary}}$  (Employee)



Name	Salary
John	200000
John	600000

# Note that RA Operators are Compositional!

Students(sid, sname, gpa)

```
SELECT DISTINCT  
  sname,  
  gpa  
FROM Students  
WHERE gpa > 3.5;
```

How do we represent  
this query in RA?



$\Pi_{sname, gpa}(\sigma_{gpa > 3.5}(Students))$



$\sigma_{gpa > 3.5}(\Pi_{sname, gpa}(Students))$

Are these logically  
equivalent?

### 3. Cross-Product (×)

- Each tuple in  $R_1$  with each tuple in  $R_2$
- Notation:  $R_1 \times R_2$
- Example:
  - Employee  $\times$  Dependents
- Rare in practice; mainly used to express joins

```
Students(sid, sname, gpa)  
People(ssn, pname, address)
```

SQL:

```
SELECT *  
FROM Students, People;
```



RA:

*Students  $\times$  People*

Another example: People

ssn	pname	address
1234545	John	216 Rosse
5423341	Bob	217 Rosse



Students

sid	sname	gpa
001	John	3.4
002	Bob	1.3

*Students × People*



ssn	pname	address	sid	sname	gpa
1234545	John	216 Rosse	001	John	3.4
5423341	Bob	217 Rosse	001	John	3.4
1234545	John	216 Rosse	002	Bob	1.3
5423341	Bob	216 Rosse	002	Bob	1.3

# Renaming ( $\rho$ )

Students(sid, sname, gpa)

- Changes the schema, not the instance
- A ‘special’ operator- neither basic nor derived
- Notation:  $\rho_{B_1, \dots, B_n} (R)$
- Note: this is shorthand for the proper form (since names, not order matters!):
  - $\rho_{A_1 \rightarrow B_1, \dots, A_n \rightarrow B_n} (R)$

SQL:

```
SELECT
  sid AS studId,
  sname AS name,
  gpa AS gradePtAvg
FROM Students;
```



RA:

$\rho_{studId, name, gradePtAvg}(Students)$

We care about this operator *because* we are working in a *named perspective*

Another example:

Students

sid	sname	gpa
001	John	3.4
002	Bob	1.3

$\rho_{studId,name,gradePtAvg}(Students)$



Students

studId	name	gradePtAvg
001	John	3.4
002	Bob	1.3



# Natural Join ( $\bowtie$ )

Note: Textbook notation is \*

Students(sid, name, gpa)  
People(ssn, name, address)

- Notation:  $R_1 \bowtie R_2$
- Joins  $R_1$  and  $R_2$  on *equality of all shared attributes*
  - If  $R_1$  has attribute set  $A$ , and  $R_2$  has attribute set  $B$ , and they share attributes  $A \cap B = C$ , can also be written:  $R_1 \bowtie_C R_2$
- Our first example of a *derived* RA operator:
  - Meaning:  $R_1 \bowtie R_2 = \Pi_{A \cup B}(\sigma_{C=D}(\rho_{C \rightarrow D}(R_1) \times R_2))$
  - Where:
    - The rename  $\rho_{C \rightarrow D}$  renames the shared attributes in one of the relations
    - The selection  $\sigma_{C=D}$  checks equality of the shared attributes
    - The projection  $\Pi_{A \cup B}$  eliminates the duplicate common attributes

SQL:

```
SELECT DISTINCT
  sid, S.name, gpa,
  ssn, address
FROM
  Students S,
  People P
WHERE S.name = P.name;
```



RA:  
 $Students \bowtie People$

Another example:

Students S

sid	S.name	gpa
001	John	3.4
002	Bob	1.3



People P

ssn	P.name	address
1234545	John	216 Rosse
5423341	Bob	217 Rosse

*Students* ⋈ *People*



sid	S.name	gpa	ssn	address
001	John	3.4	1234545	216 Rosse
002	Bob	1.3	5423341	216 Rosse

# Natural Join

- Given schemas  $R(A, B, C, D)$ ,  $S(A, C, E)$ , what is the schema of  $R \bowtie S$  ?
- Given  $R(A, B, C)$ ,  $S(D, E)$ , what is  $R \bowtie S$  ?
- Given  $R(A, B)$ ,  $S(A, B)$ , what is  $R \bowtie S$  ?

# Example: Converting SFW Query -> RA

Students(sid,name,gpa)  
People(ssn,name,address)

```
SELECT DISTINCT
  gpa,
  address
FROM Students S,
     People P
WHERE gpa > 3.5 AND
      S.name = P.name;
```



$\Pi_{gpa,address}(\sigma_{gpa>3.5}(S \bowtie P))$

How do we represent  
this query in RA?

# Logical Equivalence of RA Plans

- Given relations  $R(A,B)$  and  $S(B,C)$ :
  - Here, projection & selection commute:
    - $\sigma_{A=5}(\Pi_A(R)) = \Pi_A(\sigma_{A=5}(R))$
  - What about here?
    - $\sigma_{A=5}(\Pi_B(R)) \neq \Pi_B(\sigma_{A=5}(R))$

# Relational Algebra (RA)

- Five basic operators:

1. Selection:  $\sigma$
2. Projection:  $\Pi$
3. Cartesian Product:  $\times$
4. Union:  $\cup$
5. Difference:  $-$

We'll look at these

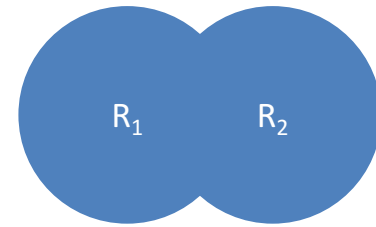
*And also at some of  
these derived  
operators*

- Derived or auxiliary operators:

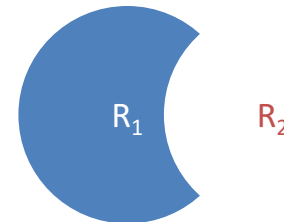
- Intersection
- Joins (natural, equi-join, theta join, semi-join)
- Renaming:  $\rho$
- Division

# 1. Union ( $\cup$ ) and 2. Difference ( $-$ )

- $R_1 \cup R_2$
- Example:
  - $\text{ActiveEmployees} \cup \text{RetiredEmployees}$

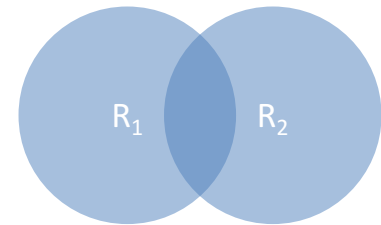


- $R_1 - R_2$
- Example:
  - $\text{AllEmployees} - \text{RetiredEmployees}$



## What about Intersection ( $\cap$ ) ?

- It is a derived operator
- $R_1 \cap R_2 = R_1 - (R_1 - R_2)$
- Also expressed as a join!
- Example
  - `UnionizedEmployees`  $\cap$  `RetiredEmployees`





# Theta Join ( $\bowtie_{\theta}$ )

Students(sid, sname, gpa)  
People(ssn, pname, address)

- A join that involves a predicate
- $R_1 \bowtie_{\theta} R_2 = \sigma_{\theta}(R_1 \times R_2)$
- Here  $\theta$  can be any condition

Note that natural join is a theta join + a projection.

SQL:

```
SELECT *  
FROM  
    Students, People  
WHERE  $\theta$ ;
```



RA:  
 $Students \bowtie_{\theta} People$

# Equi-join ( $\bowtie_{A=B}$ )

- A theta join where  $\theta$  is an equality
- $R_1 \bowtie_{A=B} R_2 = \sigma_{A=B} (R_1 \times R_2)$
- Example:
  - $\text{Employee} \bowtie_{\text{SSN}=\text{SSN}} \text{Dependents}$

Most common join  
in practice!

```
Students(sid, sname, gpa)  
People(ssn, pname, address)
```

SQL:

```
SELECT *  
FROM  
    Students S,  
    People P  
WHERE sname = pname;
```



RA:

$S \bowtie_{sname=pname} P$

# Semijoin ( $\bowtie$ )

- $R \bowtie S = \Pi_{A_1, \dots, A_n} (R \Join S)$
- Where  $A_1, \dots, A_n$  are the attributes in  $R$
- Example:
  - Employee  $\bowtie$  Dependents

Students(sid, sname, gpa) People(ssn, pname, address)
--

SQL:

SELECT DISTINCT sid, sname, gpa FROM Students, People WHERE sname = pname;
---



RA:

*Students  $\bowtie$  People*

# Division ( $\div$ )

- $T(Y) = R(Y,X) \div S(X)$
- $Y$  is the set of attributes of  $R$  that are not attributes of  $S$ .
- For a tuple  $t$  to appear in the result  $T$  of the Division, the values in  $t$  must appear in  $R$  in combination with *every* tuple in  $S$ .

# Example

R(Y,X)

÷

S(X)

=

T(Y)

```
PilotSkills
pilot_name    plane_name
=====
'Celko'       'Piper Cub'
'Higgins'     'B-52 Bomber'
'Higgins'     'F-14 Fighter'
'Higgins'     'Piper Cub'
'Jones'       'B-52 Bomber'
'Jones'       'F-14 Fighter'
'Smith'       'B-1 Bomber'
'Smith'       'B-52 Bomber'
'Smith'       'F-14 Fighter'
'Wilson'      'B-1 Bomber'
'Wilson'      'B-52 Bomber'
'Wilson'      'F-14 Fighter'
'Wilson'      'F-17 Fighter'
```

```
Hangar
plane_name
=====
'B-1 Bomber'
'B-52 Bomber'
'F-14 Fighter'
```

```
PilotSkills DIVIDED BY Hangar
pilot_name
=====
'Smith'
'Wilson'
```

```
SELECT PS1.pilot_name
      FROM PilotSkills AS PS1, Hangar AS H1
     WHERE PS1.plane_name = H1.plane_name
    GROUP BY PS1.pilot_name
   HAVING COUNT(PS1.plane_name) =
 (SELECT COUNT(plane_name) FROM Hangar);
```

<https://www.simple-talk.com/sql/t-sql-programming/divided-we-stand-the-sql-of-relational-division/>

# Complete Set of Relational Operations

- The set of operations including
  - Select  $\sigma$ ,
  - Project  $\pi$
  - Union  $\cup$
  - Difference  $-$
  - Rename  $\rho$ , and
  - Cartesian Product  $\bowtie$
- is called a *complete set*
- because any other relational algebra expression can be expressed by a combination of these five operations.
- For example:
  - $R \cap S = (R \cup S) - ((R - S) \cup (S - R))$
  - $R \bowtie_{\langle \text{join condition} \rangle} S = \sigma_{\langle \text{join condition} \rangle} (R \bowtie S)$

# Table 8.1 Operations of Relational Algebra

**Table 8.1** Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation $R$ .	$\sigma_{\langle \text{selection condition} \rangle}(R)$
PROJECT	Produces a new relation with only some of the attributes of $R$ , and removes duplicate tuples.	$\pi_{\langle \text{attribute list} \rangle}(R)$
THETA JOIN	Produces all combinations of tuples from $R_1$ and $R_2$ that satisfy the join condition.	$R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$
EQUIJOIN	Produces all the combinations of tuples from $R_1$ and $R_2$ that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$ , OR $R_1 \bowtie_{(\langle \text{join attributes } 1 \rangle), (\langle \text{join attributes } 2 \rangle)} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of $R_2$ are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 \star_{\langle \text{join condition} \rangle} R_2$ , OR $R_1 \star_{(\langle \text{join attributes } 1 \rangle), (\langle \text{join attributes } 2 \rangle)} R_2$ OR $R_1 \star R_2$

*continued on next slide*

# Table 8.1 Operations of Relational Algebra (continued)

**Table 8.1** Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
UNION	Produces a relation that includes all the tuples in $R_1$ or $R_2$ or both $R_1$ and $R_2$ ; $R_1$ and $R_2$ must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both $R_1$ and $R_2$ ; $R_1$ and $R_2$ must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in $R_1$ that are not in $R_2$ ; $R_1$ and $R_2$ must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of $R_1$ and $R_2$ and includes as tuples all possible combinations of tuples from $R_1$ and $R_2$ .	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in $R_1$ in combination with every tuple from $R_2(Y)$ , where $Z = X \cup Y$ .	$R_1(Z) \div R_2(Y)$



# Examples of Queries in Relational Algebra : Procedural Form

- Q1: Retrieve the name and Address of all Employees who work for the 'Research' department.

Research\_dept  $\leftarrow \sigma_{Dname='Research'}(Department)$

Research\_emps  $\leftarrow (RESEARCH\_DEP \bowtie_{DNumber=Dno} Employee)$

Result  $\leftarrow \pi_{Fname, Lname, Address}(Research\_emps)$

As a single expression, this query becomes:

$\pi_{Fname, Lname, Address}(\sigma_{Dname='Research'}(Department) \bowtie_{Dnumber=Dno} Employee)$

# Examples of Queries in Relational Algebra : Procedural Form

- Q6: Retrieve the names of Employees who have no dependents.

$ALL\_EMPS \leftarrow \pi_{SSN}(Employee)$

$EMPS\_WITH\_DEPS(SSN) \leftarrow \pi_{Essn}(DEPENDENT)$

$EMPS\_WITHOUT\_DEPS \leftarrow (ALL\_EMPS - EMPS\_WITH\_DEPS)$

$RESULT \leftarrow \pi_{Lname, Fname} (EMPS\_WITHOUT\_DEPS * Employee)$

As a single expression, this query becomes:

$\pi_{Lname, Fname} ((\pi_{Ssn}(Employee) - \rho_{Ssn}(\pi_{Essn}(Dependent))) * Employee)$

# Division

- $T(Y) = R(Y,X) \div S(X)$
- The complete division expression:
- $R \div S = \pi_Y R - \pi_Y((\pi_Y(R) \times S) - R)$

Ignoring the projections, there are just three steps:

- Compute all possible attribute pairings
- Remove the existing pairings
- Remove the non-answers from the possible answers

<https://www2.cs.arizona.edu/~mccann/research/divpresentation.pdf>

# Division Example

$$R(Y,X)$$

Y	X
y1	x1
y1	x2
y2	x1
y3	x1
y3	x2
y3	x3

$$S(X)$$

X
x1
x2

$$\pi_Y(R) \bowtie S$$

Y	X
y1	x1
y1	x2
y2	x1
y2	x2
y3	x1
y3	x2

$$(\pi_Y(R) \bowtie S) - R$$

Y	X
y2	x2

$$\pi_Y((\pi_Y(R) \bowtie S) - R) \quad R \div S = \pi_Y R - \pi_Y((\pi_Y(R) \bowtie S) - R)$$

Y
y2

Y
y1
y3

# Acknowledgement

- Some of the slides in this presentation are taken from the slides provided by the authors.
- Many of these slides are taken from cs145 course offered by Stanford University.
- Thanks to YouTube, especially to [Dr. Daniel Soper](#) for his useful videos.