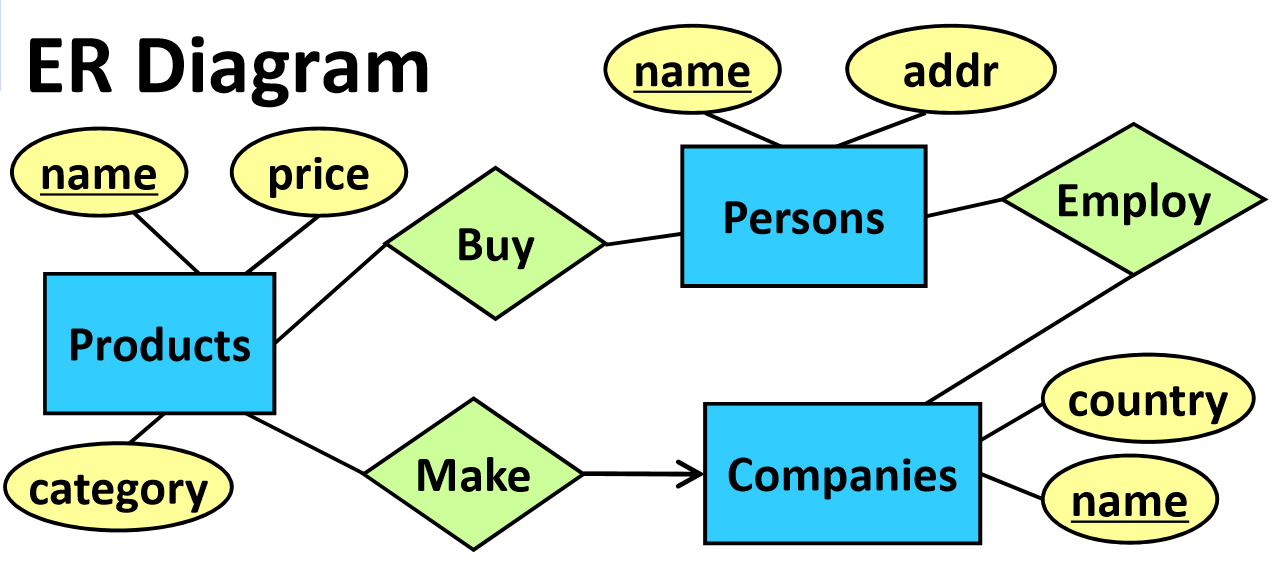
CZ2007 Lecture Notes Part 1

# Database Systems

* **Database**: collection of data organized for efficient retrieval by a computer
* **Database Management System (DBMS)**: software that manages and retrieves information from a database
* **Relational model**: all data is stored in relation to each other

# Entity-Relationship (ER) Diagrams

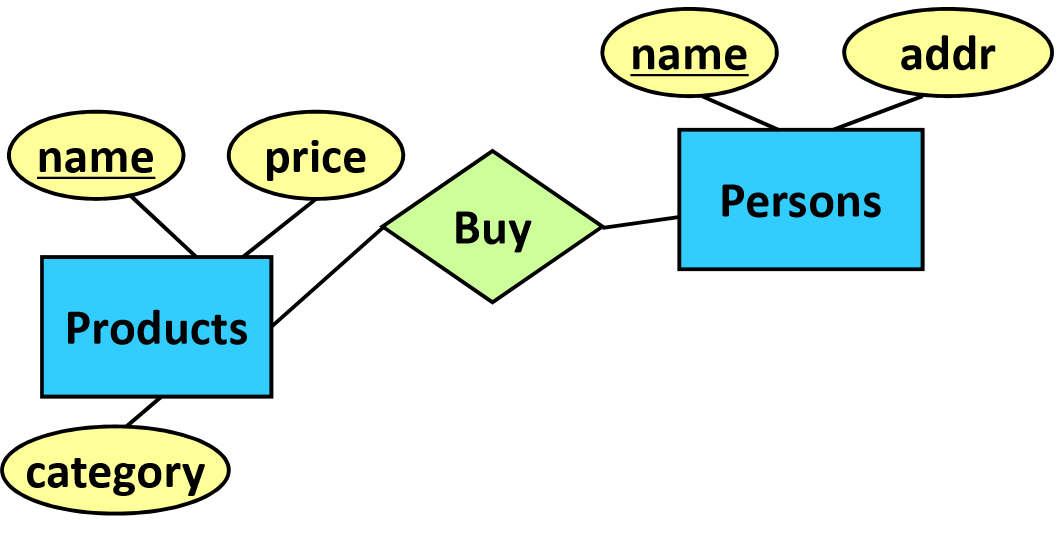
* Models the requirements of an application from concepts to tables



* **Entity Sets** (rectangles): groups of similar real-world objects/data
* **Attributes** (ovals): property of an entity set
* **Relations** (diamonds): connection between two entity sets

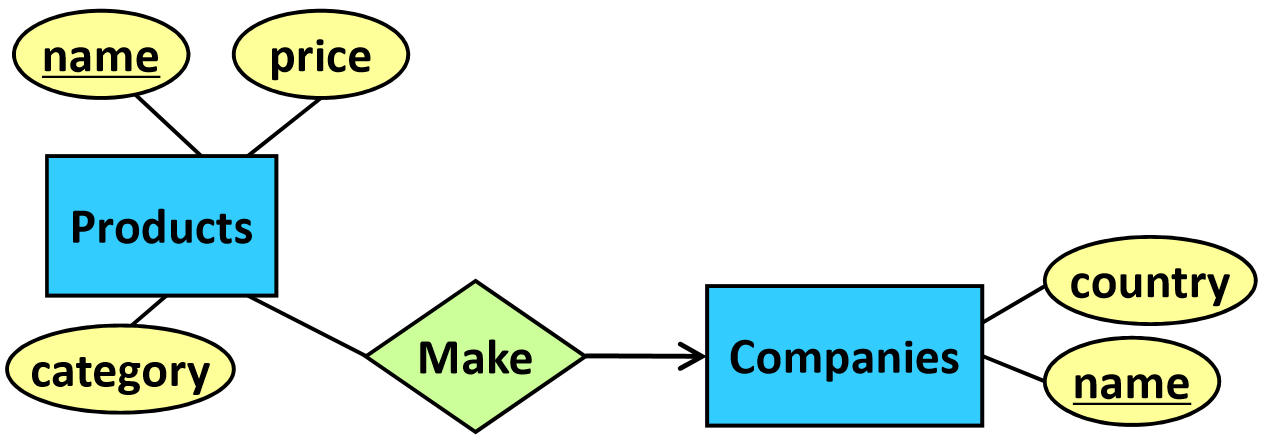
## Types of Relationships

### Many-to-many



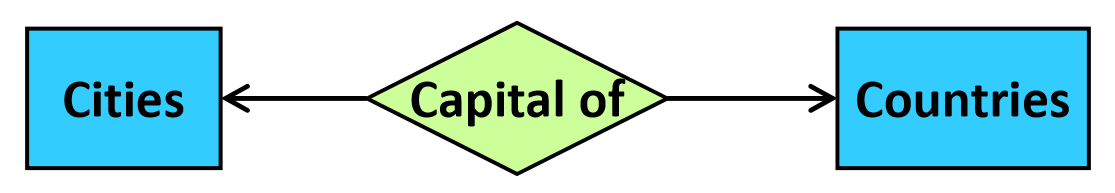
* One product can be bought by more than one person
* One person can buy more than one product
* No arrows

### Many-to-one



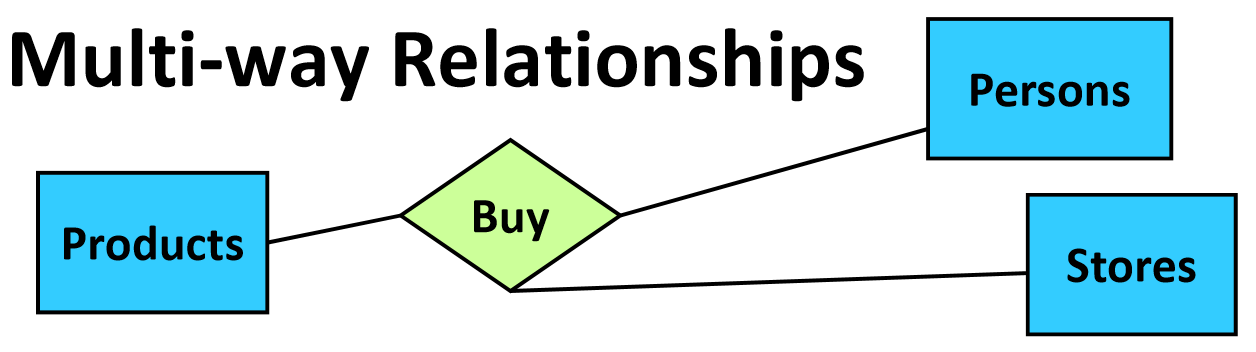
* One company can make multiple products
* Each product can only be made by one company
* Arrow points on side of one

### One-to-one



* Each country can only have one capital city
* Each city can only be the capital of one country
* Arrows on both sides

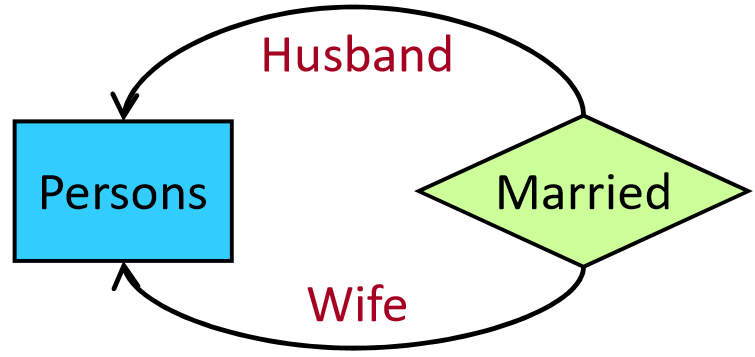
## Multi-Way Relationships



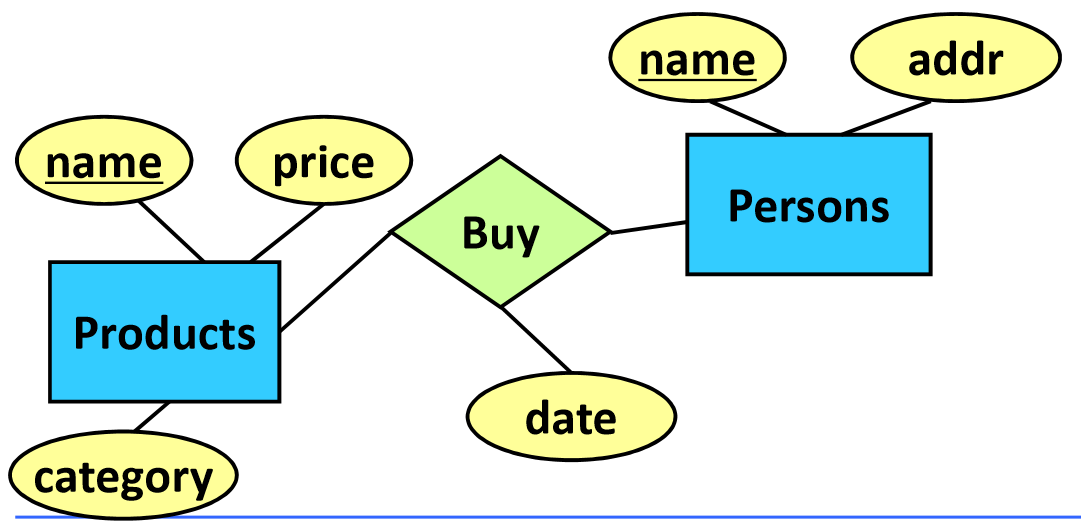


* One (person, store) pair can correspond to only one product
* Each product can correspond to multiple (person, store) pairs

## Roles



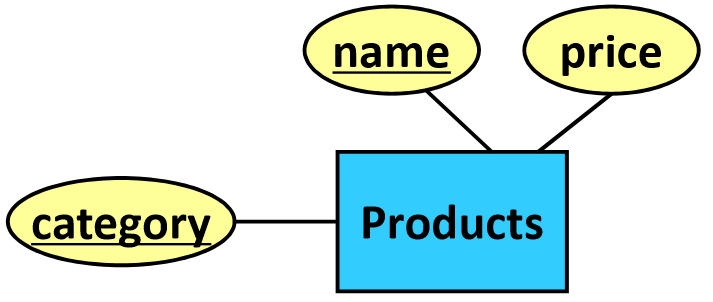
* When the same entity set can appear multiple times in a relationship, specify the role of each relationship
* Relationships can also have their own attributes
  + E.g. want to store the date of purchase of products



## Constraints

### Key constraints

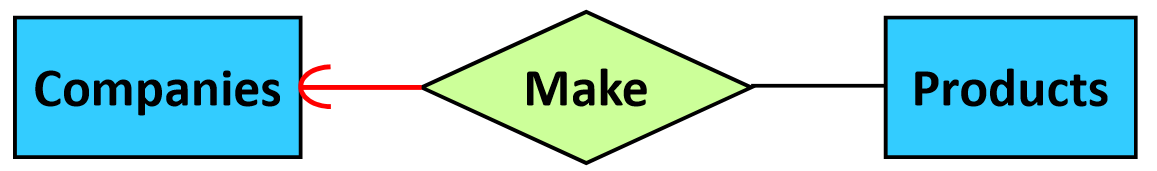
* Each entity set must have a **unique identifier** – can be one or more attributes
* With multiple attributes, each set must be unique, not individual attributes



* Underline key attribute(s)

### Referential integrity constraints

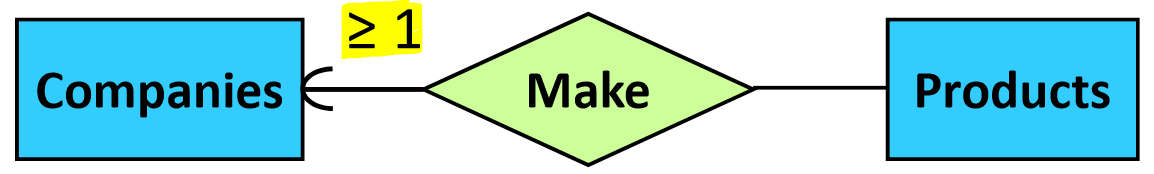
* Every member of an entity set must have a relationship to another entity set
  + E.g. every product must be made by one company – no products can exist that don’t have a make relationship with a company



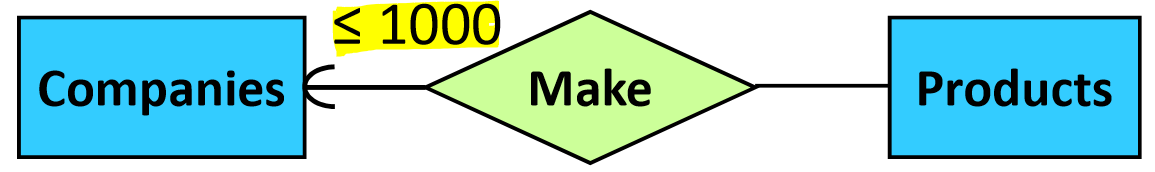
* Rounded arrow

### Degree constraints (not on exam)

* Every member of an entity set must have a certain number of relationships to another entity set
  + E.g. every company must make at least one product

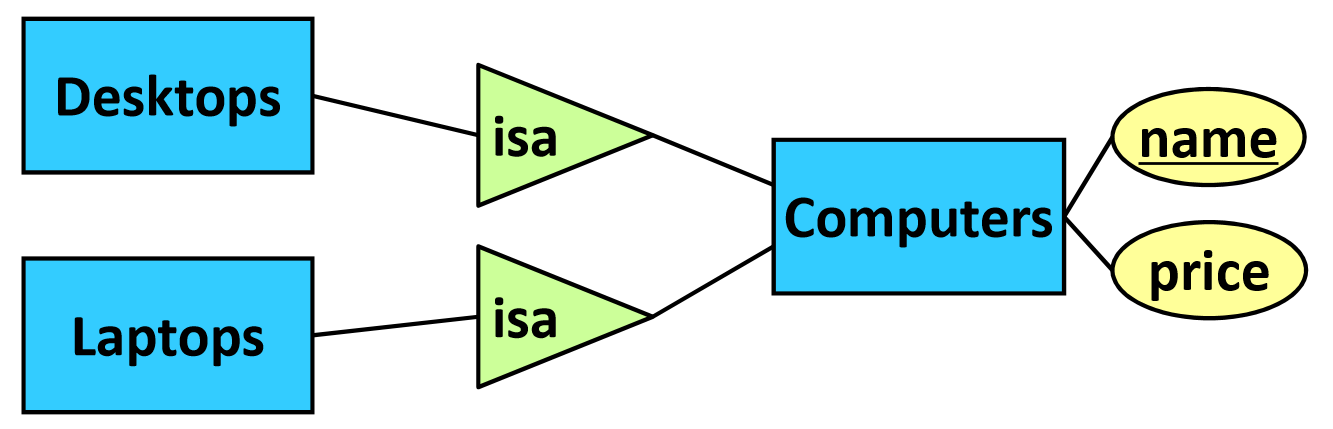


* + E.g. every company can make at most 1000 products



* Degree constraints are hard to enforce in a DBMS

## Subclasses

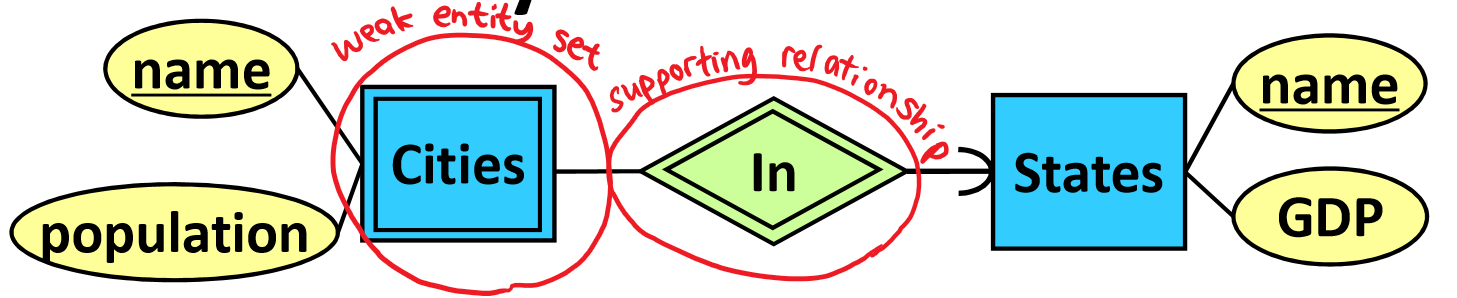


* Special type of an entity set, with additional attributes
  + Key of subclass = key of superclass
* An entity set can have multiple subclasses
* Isa triangle

### When to use subclasses?

* When subclass has an attribute missing from the superclass
* When subclass has its own relationships to other entity sets

## Weak Entity Sets



* Entity sets that **cannot be uniquely identified** by their own attributes
  + Instead, they are uniquely identified based on attributes from other entities
  + Key of Cities = (Cities.name, States.name)
* Double border around weak entity set and supporting relationship
* Add referential integrity to supporting entity set

## ER Design Principles

1. Be faithful to the application requirements/specifications
2. Avoid redundancy

* Don’t repeat information – can waste space or lead to inconsistent data

1. Keep it simple

* Don’t repeat the same data
* Sometimes subclasses can be differentiated by an attribute

1. Don’t over-use weak entity sets

# From ER Diagrams to Relational Schemas

* **Relation schema**: database table – name and attributes
* **Database schema**: set of relation schemas

## Entity Sets

* Each entity set becomes a relation
  + Key of relation = key of **entity set**

Example:

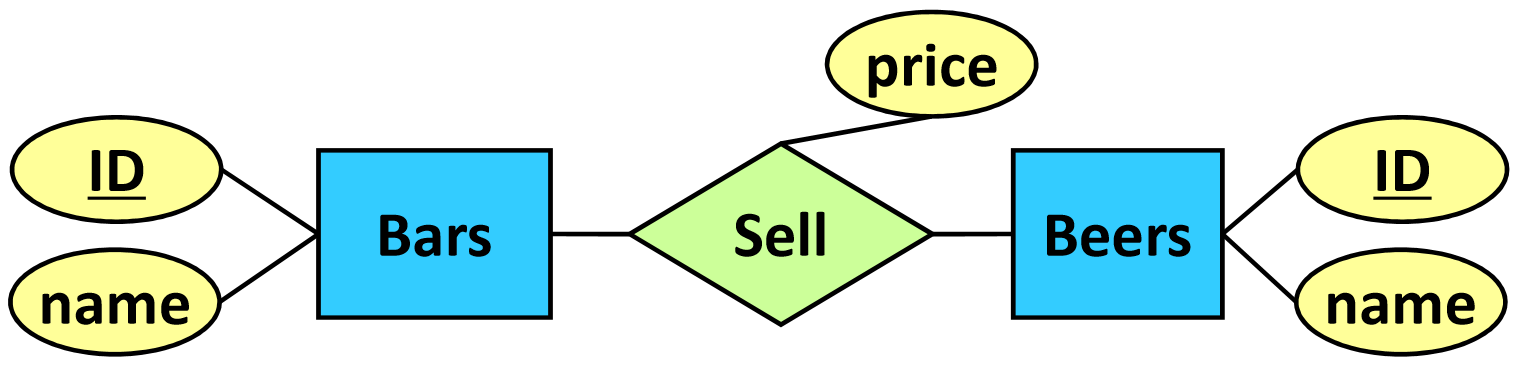
### 

|  |  |  |  |
| --- | --- | --- | --- |
| **Bars** | | | |
| ID | name | addr | license |
| … | … | … | … |

## Many-to-Many Relations

* Each many-to-many relationship becomes a relation
  + Key of relation = keys of **participating entity sets**

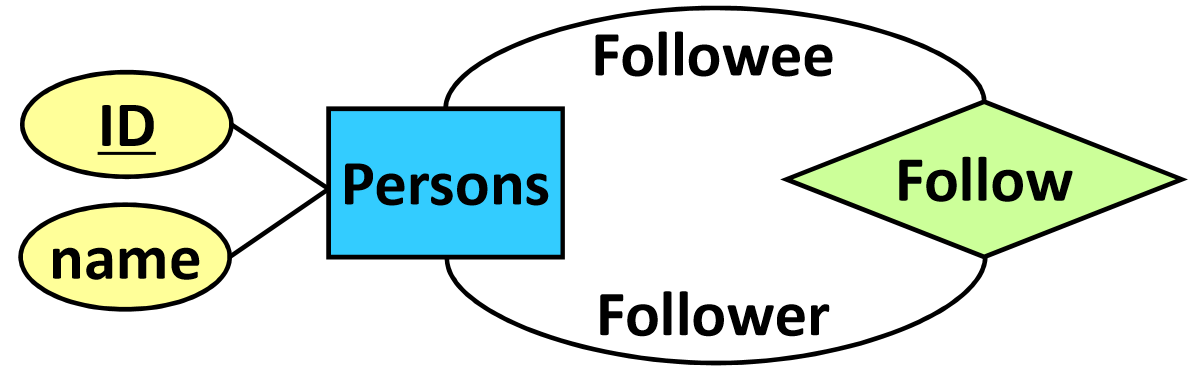
Example:



|  |  |  |
| --- | --- | --- |
| **Sell** | | |
| Bars-ID | Beers-ID | price |
| … | … | … |

* If an entity set is referred to multiple times in the relation, then use its **role** instead of entity set name

Example:

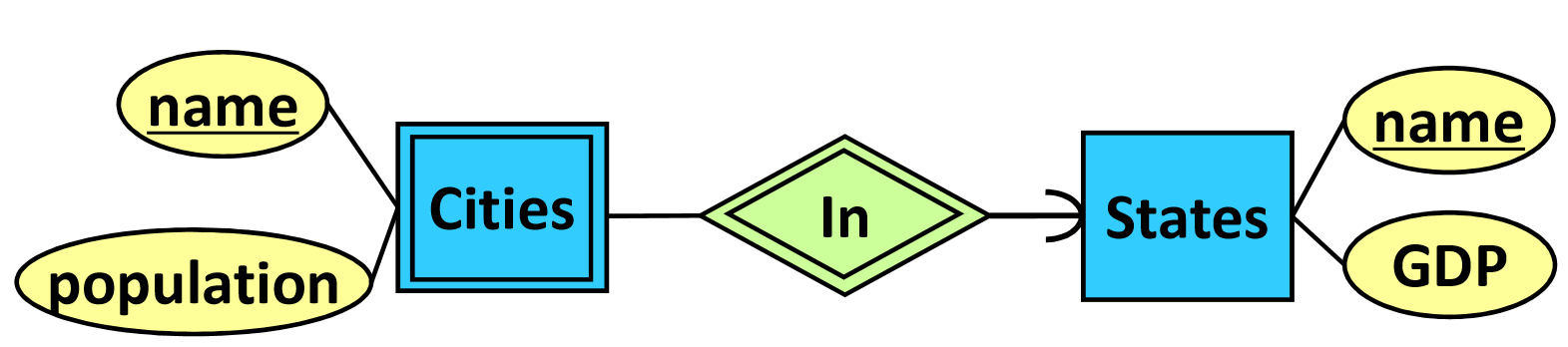


|  |  |
| --- | --- |
| **Follow** | |
| Follower-ID | Followee-ID |
| … | … |

## Weak Entity Sets

* Create a relation for the weak entity set, with all of its attributes and the **key(s) of the supporting entity set**
* Ignore the supporting relationship

Example:



|  |  |  |
| --- | --- | --- |
| **Cities** | | |
| States-name | name | population |
| … | … | … |

## Subclasses

### ER Approach

* Create a relation for each entity set, with only the attributes of that set
* Subclasses may be stored in multiple relations
  + Results in redundant records

### OO Approach

* Create a relation for each possible combination of subclasses
  + Each record only appears in one relation
  + But this results in many relations if there are many combinations

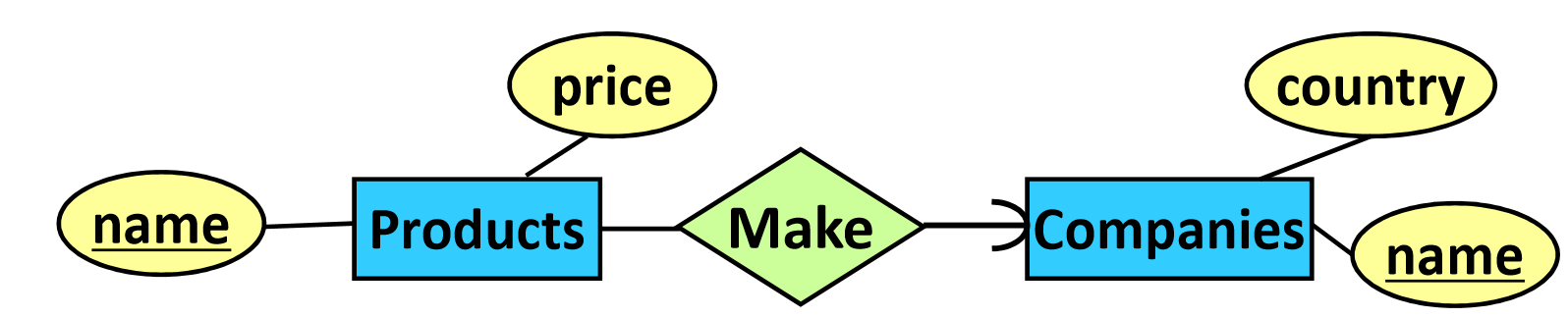
### NULL Approach

* Create one relation with all attributes of subclasses, use NULL for values that don’t apply
  + This can result in a sparse table with many NULL values

## Many-to-One Relationships

* Create a relation for each entity set
* Add the **key of the “one” side** as an attribute to the relation of the “many” side
* Do not need to create a relation for the relationship

Example:



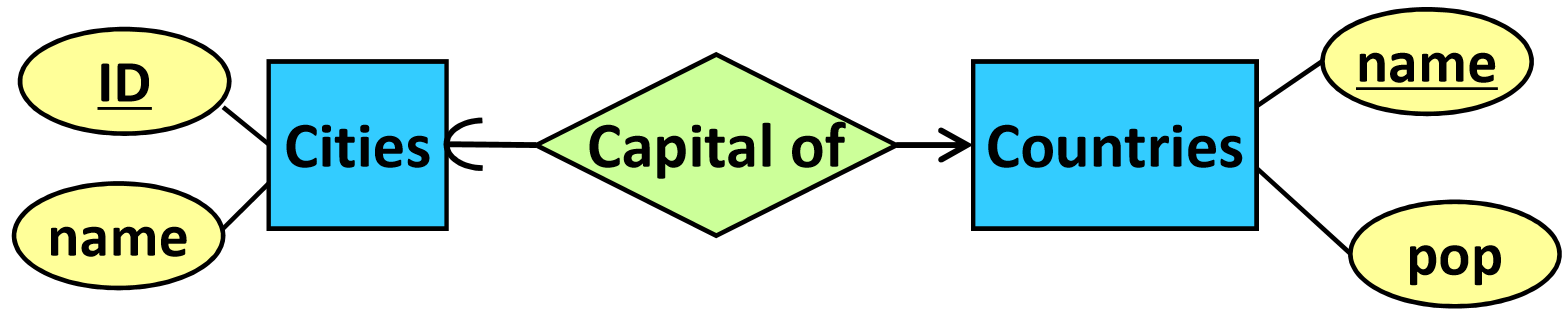
|  |  |
| --- | --- |
| **Companies** | |
| name | country |
| … | … |

|  |  |  |
| --- | --- | --- |
| **Products** | | |
| name | price | company-name |
| … | … | … |

## One-to-One Relationships

* Create a relation for each entity set
* Add the **key of one of the “one” sides** as an attribute to the relation of the other “one” side
* Do not need to create a relation for the relationship
* If there is a side with **referential integrity**, then should put that side’s key into the other side – because it will never be null

Example:

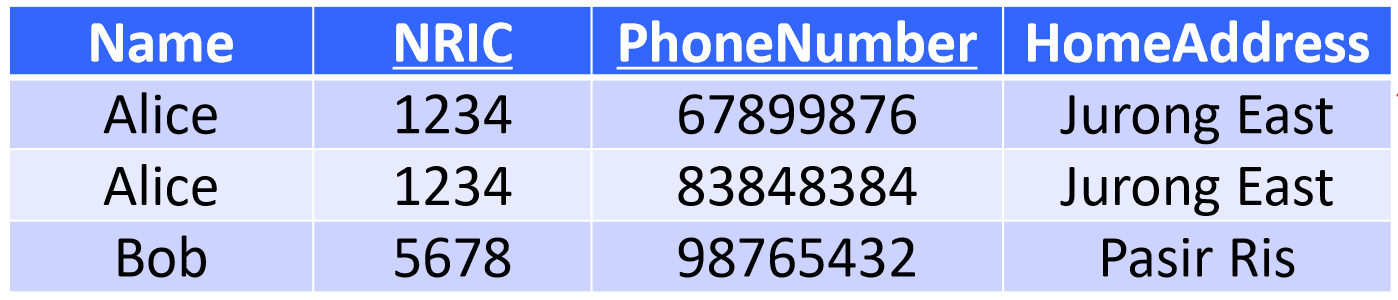


|  |  |
| --- | --- |
| **Cities** | |
| CID | cname |
| … | … |

|  |  |  |
| --- | --- | --- |
| **Countries** | | |
| name | pop | CID |
| … | … | … |

# Database Design

## Database Anomalies



* **Redundancy**: address has duplicate entries
* **Update anomalies**: it’s possible to update one of Alice’s address records but not the other
  + Leads to inconsistencies in the data
* **Deletion anomalies**: if Bob no longer has a phone number, impossible to delete phone number as it’s part of the key (cannot be NULL)
* **Insertion anomalies**: cannot insert a record without a phone number, since key attributes cannot be NULL

## Normalization

|  |  |  |
| --- | --- | --- |
| **Name** | **NRIC** | **HomeAddress** |
| Alice | 1234 | Jurong East |
| Bob | 5678 | Pasir Ris |

|  |  |
| --- | --- |
| **NRIC** | **PhoneNumber** |
| 1234 | 67899876 |
| 1234 | 83848384 |
| 5678 | 98765432 |

* To fix data anomalies, we can decompose the table
  + No redundancy – Alice’s address isn’t duplicated
  + No update anomalies – only one record needs to be updated
  + No deletion anomalies – can remove Bob’s phone number without affecting other data
  + No insertion anomalies – can insert record without phone number

## Functional Dependencies

* Describes **correlations** between attributes in a relation
* Given attributes and , if there do not exist two objects with the same values for but different values for , then we say
  + The same values of will always produce the same

Examples: Purchase(CustomerID, ProductID, ShopID, Price, Date)

* Each shop can sell at most one product
  + ProductID 🡪 ShopID
    - Same shop can sell different products, but each product can only be sold by one shop
* No shop will sell the same product to the same customer on the same date at different prices
  + CustomerID, ShopID, ProductID, Date 🡪 Price
    - Same shop can sell same product to different customers on same day at different prices, just not to the same customer

## Armstrong’s Axioms

1. **Axiom of Reflexivity**

Set of attributes 🡪 subset of its attributes

* E.g. ABCD 🡪 AB

1. **Axiom of Augmentation**

Given A 🡪 B, AC 🡪 BC for any C

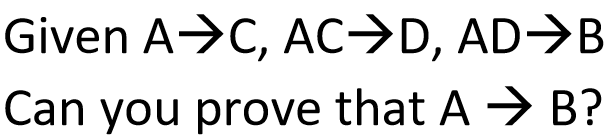
* E.g. if NRIC 🡪 Name, then NRIC, Address 🡪 Name, Address

1. **Axiom of Transitivity**

Given A 🡪 B and B 🡪 C, A 🡪 C

* E.g. if Name 🡪 Address and Address 🡪 Postal Code, then Name 🡪 Postal Code

Example:

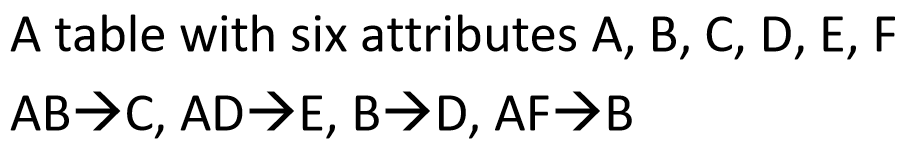


* A 🡪 AC (Augmentation)
* A 🡪 D (Transitivity)
* A 🡪 AD (Augmentation)
* A 🡪 B (Transitivity)

## Closures (Activated Sets)

* Let be a set of attributes. The **closure** of , is a set of attributes that can be decided by
* To find the closure of an attribute/attributes, given functional dependencies:
  + Start with the attribute(s) in the activated set
  + Add all attributes that can be decided by the attributes in the activated set
  + Add all attributes that can be decided by any group of attributes in the activated set; repeat

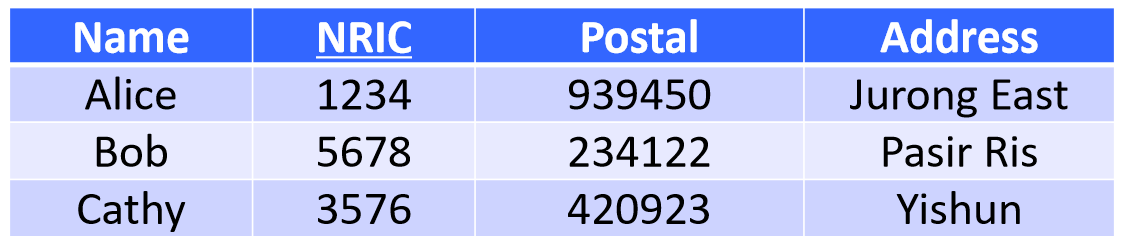
Example:



### Closures and Functional Dependencies

* To prove that a FD A 🡪 B holds, we need to prove that  **contains**
* To prove that a FD A 🡪 B does not hold, we need to prove that  **does not contain**

## Keys in a Table



### Superkeys

A set of attributes in a table that **decide all other attributes**

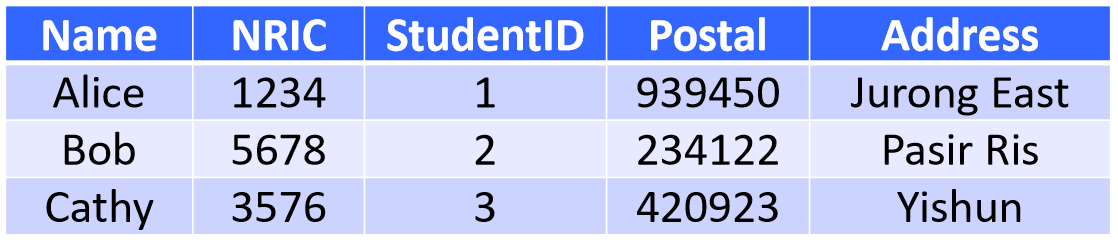
* E.g. NRIC is a superkey because NRIC 🡪 Name, Postal, Address
* E.g. {NRIC, Name} is a superkey because {NRIC, Name} 🡪 Postal, Address

### Keys

A superkey that, if an attribute is removed from the set, will **no longer be a superkey**

* E.g. NRIC is a key because NRIC 🡪 Name, Postal, Address and cannot be removed from the set
* E.g. {NRIC, Name} is not a superkey because removing Name leaves a superkey NRIC 🡪 Name, Postal, Address

### Candidate Keys



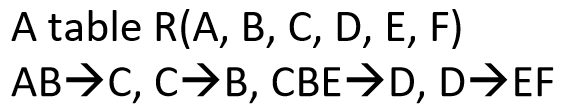
If a table has **multiple keys**, all keys are candidate keys

* E.g. NRIC 🡪 StudentID, Name, Postal, Address and StudentID 🡪 NRIC, Name, Postal, Address, so both are candidate keys
* One key is chosen as the **primary key**; the rest are **secondary keys**

## Finding the Keys in a Table

* Check all possible combinations of attributes in a table by computing its closure
  + If a closure **contains all attributes**, it is a superkey and might be a key
  + If an attribute is a key, then can skip finding closures of all sets containing
  + If an attribute does not appear in the RHS of any FDs, then  **must be in every key**

Example:



* A must be in every key, since it does not appear in the RHS of any FDs

# Normal Form

**Normal form** involves some conditions that a “good” table must satisfy

## Boyce-Codd Normal Form (BCNF)

* A table R is in BCNF if and only if the **LHS** of every **non-trivial FD** contains a **key of R**
  + A **non-trivial FD** is one where the RHS is not included in the LHS (not implied by the Axiom of Reflexivity)
* BCNF forbids a non-trivial X 🡪 Y where X isn’t a key
  + The intuition behind this is if X is not a key, then the FD suggests that X should be unique, but it is not a key of the table – the table can contain redundancies

Example: R(A, B)  
Given FDs: A 🡪 B  
Key: A  
All FDs on R: A 🡪 B, AB 🡪 A, AB 🡪 B, AB 🡪 AB

Since AB 🡪 A, AB 🡪 B, AB 🡪 AB are all trivial FDs, only need to consider A 🡪 B, which contains a key on the LHS, so R is in BCNF.

Example: R(A, B, C)  
Given FDs: A 🡪 B  
Key: AC  
All FDs on R: A 🡪 B, AB 🡪 B, AC 🡪 C, …

Since A 🡪 B is a non-trivial FD that doesn’t contain a key on the LHS, R is not in BCNF.

### Check if a table R is in BCNF:

Given a table R and a set of FDs on R:

1. Derive the keys of R from the given FDs
2. Determine all given non-trivial FDs on R
3. For each non-trivial FD, check if its LHS contains a key

### BCNF Decomposition

1. Identify an FD X 🡪 Y on R that violates BCNF
2. Compute the closure of X
3. Decompose R into two tables: R1 contains **all attributes in the closure**, R2 contains **X and all attributes not in the closure**
4. Check if R1 and R2 are in BCNF, and decompose if necessary

Notes:

* BCNF decomposition may not be unique
* Tables with two attributes are always in BCNF

Example: R(A, B, C, D)  
Given FDs: A 🡪 B, B 🡪 C, C 🡪 D  
Keys: A

For R(A, B, C, D):  
B 🡪 C is a violation  
{B}+ = {BCD}  
Decompose R into R1(B, C, D) and R2(A, B)  
 - R2 is in BCNF since it only contains two attributes

For R1(B, C, D):  
Key: B   
C 🡪 D is a violation  
{C}+ = {CD}  
Decompose R1 into R3(C, D) and R4(B, C)  
 - R2 and R3 are in BCNF since they only contain two attributes

Final result: R2(A, B), R3(C, D), R4(B, C)

**Properties of BCNF Decomposition**

* No update or deletion anomalies
* Very small redundancy
* **Lossless join property**: original table can be reconstructed from decomposed tables
  + BCNF decomposition guarantees that when a table R is deconstructed into R1 and R2, because the **common attributes** in R1 and R2 **constitute a superkey of R1 or R2**
* It may not preserve all FDs
  + We want to preserve all FDs because it makes it easier to avoid inappropriate updates
  + E.g. Two tables R1(A, C) and R2(B, C), original FDs C 🡪 B and AB 🡪 C
    - Due to AB 🡪 C, we aren’t supposed to have two tuples (a1, b1, c1) and (a1, b1, c2)
    - But when storing them separately, hard to check if the two tuples exist at the same time, so hard to enforce AB 🡪 C

A tricky decomposition case occurs when:

* There is an FD X 🡪 Y such that **X contains an attribute in T**, **but** **Y contains an attribute not in T**
* If this occurs, we have to use closures to check whether the table is in BCNF
  + Check whether there is a closure {X}+ = {Y} such that:
    - Y does not contain all attributes in T, and
    - Y contains more attributes than X

Example: R(A, B, C, D, E)  
Given FDs: A 🡪 B, BC 🡪 D  
Keys: ACE

For R(A, B, C, D, E):  
A 🡪 B is a violation  
{A}+ = {A, B}  
Decompose R into R1(A, B) and R2(A, C, D, E)  
 - R1 is in BCNF since it only contains two attributes

For R2(A, C, D, E):  
For A 🡪 B, A is in R2 but B is not   
Check closures of table:   
{A}+ = {A, B} => since B is not in R2, it becomes {A}+ = {A}  
{C}+ = {C}, {D}+ = {D}, {E}+ = {E}  
{AB}+ = {AB}  
{AC}+ = {ABCD} => {AC}+ = {ACD}   
 - A 🡪 B => AC 🡪 BC => AC 🡪 D  
 - since AC 🡪 D is a non-trivial FD but AC is not a key, R2 is not in BCNF  
Decompose R2 into R3(A, C, D) and R4(A, C, E)  
 - R4 is in BCNF

For R3(A, C, D):  
{A}+ = {AB} => {A}+ = {A} since B not in R3  
{C}+ = {C}, {D}+ = {D}  
{AB}+ = {AB}  
{AC}+ = {ABCD} => {AC}+ = {ACD} since B not in R3  
{AD}+ = {ABD} => {AD}+ = {AD} since B not in R3  
{CD}+ = {CD}  
 - no closures indicate a violation of BCNF, so R3 is in BCNF  
  
Example: R(A, B, C, D, E, F)  
Given FDs: B 🡪 D, C 🡪 E, DE 🡪 A  
Keys: BCF

For R(A, B, C, D, E, F):

## Third Normal Form (3NF)

* A normal form that is less strict than BCNF, but ensures preservation of FDs
* A table R satisfies 3NF if and only if, for every non-trivial FD X 🡪 Y:
  + Either **X contains a key**, or
  + Every attribute in Y is either **contained in a key** or **contained in X**

Example: Given FDs: C 🡪 B, AB 🡪 C, BC 🡪 C  
Keys: AB, AC

C 🡪 B is okay, because B is contained in a key  
AB 🡪 C is okay, because AB is a key  
BC 🡪 C is okay, because it is non-trivial (and C is contained in BC)

So, this table is in BCNF.

Example: Given FDs: A 🡪 B, B 🡪 C  
Keys: A

A 🡪 B is okay, since A is a key  
B 🡪 C is not okay, since B is not a key and C is not contained in B or key A

So, this table is not in BCNF.

### 3NF Decomposition:

Given a table R and a set of FDs S:

1. Derive a **minimal basis** of S
2. In the minimal basis, combine the FDs whose LHS are the same
3. Create a table for each FD
4. If none of the tables contain a key of R, create a table that contains a key of R
5. Remove redundant tables

### Minimal Basis

The **minimal basis** of a set of FDs S, is a simplified version of S (not always unique):

1. Any FD in the minimal basis only has **one attribute** on the RHS
2. No FD in the minimal basis **can be derived from any other FDs**
3. For any FD in the minimal basis, **none of the attributes on the LHS are redundant** – i.e. removing an attribute, the resulting FD can be derived from other FDs

**To find the minimum basis, given a set of FDs S:**

1. Transform all FDs so that each FD only has one attribute on the RHS
2. Remove redundant FDs – to check an FD, remove it from S, and find the closure of its LHS from remaining FDs
3. Remove redundant attributes on the LHS of each FD

Example: S = {A 🡪 BD, AB 🡪 C, C 🡪 D, BC 🡪 D}

Step 1: S = {A 🡪 B, A 🡪 D, AB 🡪 C, C 🡪 D, BC 🡪 D}

Step 2:

* Check A 🡪 B: removing it, {A}+ = {AD} – not redundant
* Check A 🡪 D: removing it, {A}+ = {ABCD} – redundant, S = {A 🡪 B, AB 🡪 C, C 🡪 D, BC 🡪 D}

Check AB 🡪 C: removing it, {AB}+ = {ABD} – not redundant

* Check C 🡪 D: removing it, {C}+ ={C} – not redundant
* Check BC 🡪 D: removing it, {BC}+ = {BCD} – redundant, S = {A 🡪 B, AB 🡪 C, C 🡪 D}

Step 3: only AB 🡪 C has more than one attribute on LHS

* Check without A: {B}+ = {B} – not redundant
* Check without B: {A}+ = {ABC} – redundant, S = {A 🡪 B, A 🡪 C, C 🡪 D}

Example: S = {A 🡪 C, AC 🡪 D, AD 🡪 B}

Step 1: S = {A 🡪 C, AC 🡪 D, AD 🡪 B}

Step 2:

* Check A 🡪 C: removing it, {A}+ = {A} – not redundant
* Check AC 🡪 D: removing it, {AC}+ = {AC} – not redundant
* Check AD 🡪 B: removing it, {AD}+ = {AD} – not redundant

Step 3:

* Check AC 🡪 D:
  + Check without A: {C}+ = {C} – not redundant
  + Check without C: {A}+ = {ACD} – redundant, S = {A 🡪 C, A 🡪 D, AD 🡪 B}
* Check AD 🡪 B:
  + Check without A: {D}+ = {D} – not redundant
  + Check without D: {A}+ = {ABCD} – redundant, S = {A 🡪 C, A 🡪 D, A 🡪 B}

Example: 3NF decomposition given R(A, B, C, D) and S = {A 🡪 B, A 🡪 C, C 🡪 D}

* Step 1: combine FDs with same LHS: S = {A 🡪 BC, C 🡪 D}
* Step 2: create table for each FD: R1(A, B, C), R2(C, D)
* Step 3: add a table that contains a key of the original table R, if necessary
* Step 4: remove redundant tables, if necessary

Example: 3NF decomposition given R(A, B, C, D) and S = {A 🡪 B, C 🡪 D}

Key: AC

* Step 1: combine FDs with same LHS: S = {A 🡪 B, C 🡪 D}
* Step 2: create table for each FD: R1(A, B), R2(C, D)
* Step 3: add a table that contains a key of the original table R: R3(A, C)
* Step 4: remove redundant tables, if necessary

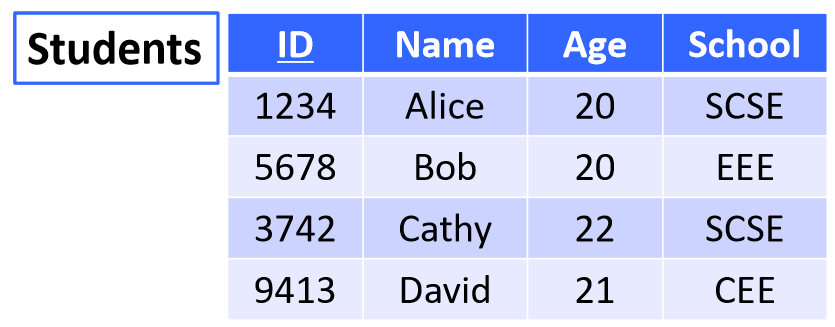
## BCNF vs. 3NF

* **BCNF**: for every non-trivial FD, its LHS must be a key
* **3NF**: for every non-trivial FD, either its LHS is a key, or every attribute on its RHS appears in a key or the LHS
* **BCNF is stricter than 3NF**:
  + Any table that satisfies BCNF satisfies 3NF, but not vice versa
  + Any table that violates 3NF violates BCNF, but not vice versa
* BCNF decomposition eliminates more redundancies, but may not preserve all FDs
* 3NF decomposition may have more redundancies than BCNF, but preserves all FDs

# Relational Algebra

* Provides a mathematical way of formulating queries on relations/tables
* Comprised of numerous operators for query formation

## Selection

* Selects **rows** from a table that satisfy the condition
* Example:   
  
  + Find all students age 20:

Result:

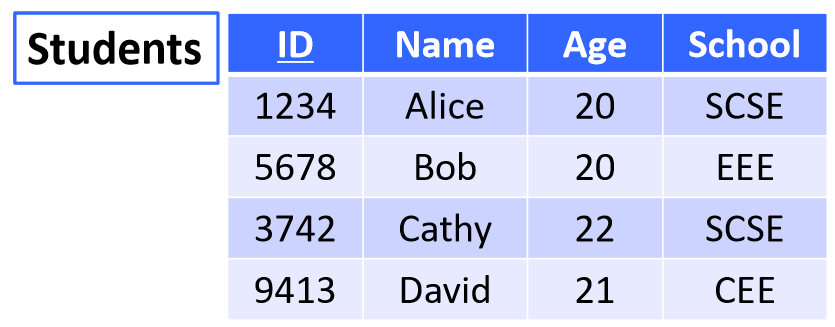
|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Name** | **Age** | **School** |
| 1234 | Alice | 20 | SCSE |
| 5678 | Bob | 20 | EEE |

* + Find all SCSE students under 21:

Result:

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Name** | **Age** | **School** |
| 1234 | Alice | 20 | SCSE |

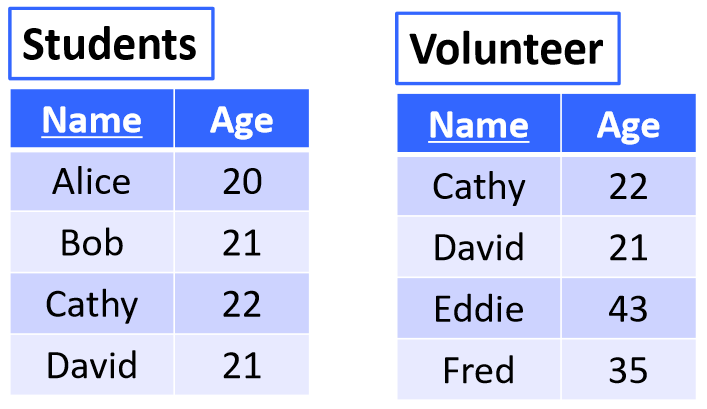
## Projection

* Selects **columns** from a table
* Example:  
  
  + Select the IDs and Names of all students:

Result:

|  |  |
| --- | --- |
| **ID** | **Name** |
| 1234 | Alice |
| 5678 | Bob |
| 3742 | Cathy |
| 9413 | David |

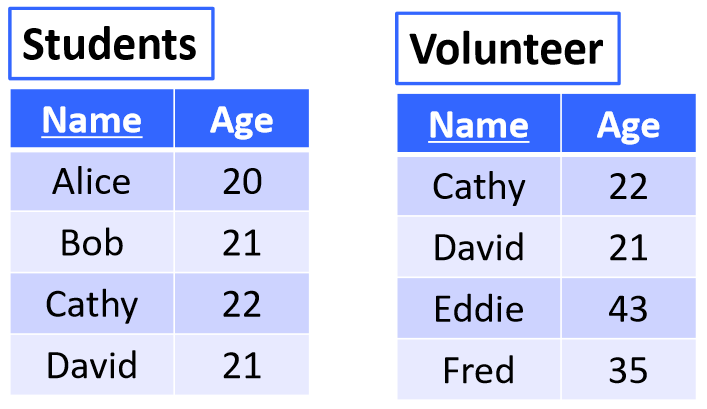
## Union

* Selects **rows in multiple tables**, automatically removing duplicates
  + Two tables must have the same schema
* Example: Find the names of persons who are either students or volunteers  
  

Result:

|  |
| --- |
| **Name** |
| Alice |
| Bob |
| Cathy |
| David |
| Eddie |
| Fred |

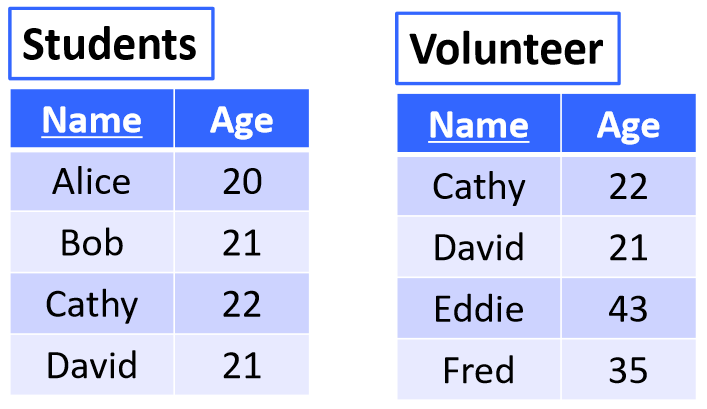
## Intersection

* Find **common rows** between two tables, automatically removing duplicates
  + Two sides must have the same schema
* Example: Find the names of persons who are both students and volunteers  
  

Result:

|  |
| --- |
| **Name** |
| Cathy |
| David |

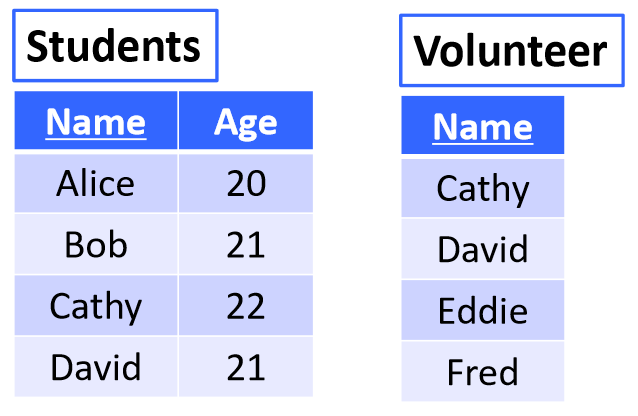
## Difference

* Selects rows in Table1 **that are not in Table2**, automatically removing duplicates
  + Two sides must have the same schema
* Example: Find the persons who are students but not volunteers  
  

Result:

|  |  |
| --- | --- |
| **Name** | **Age** |
| Alice | 20 |
| Bob | 21 |

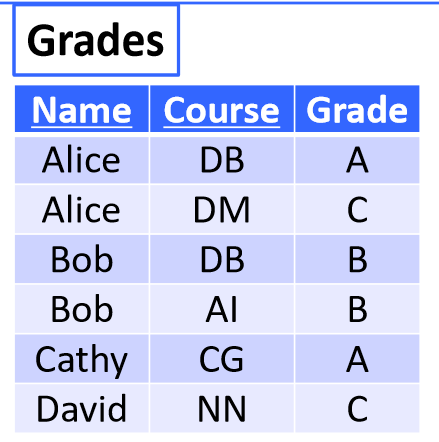
* Example: Find the persons who are students but not volunteers



* + Result:

|  |
| --- |
| **Name** |
| Alice |
| Bob |

**Exercise**:



* Find the students who have taken DB and DM, but not AI or CG

Result:

|  |
| --- |
| **Name** |
| Alice |

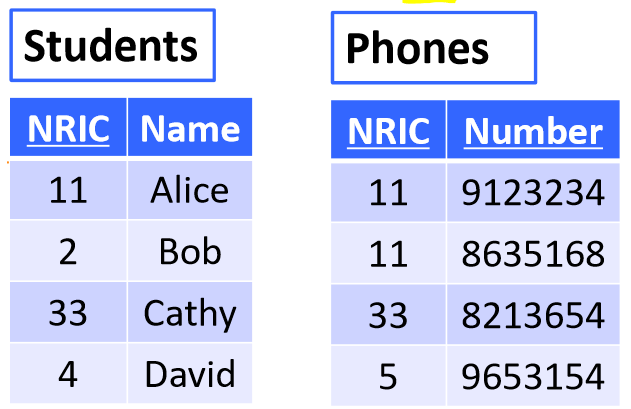
* Find the students who have never taken DM

Result:

|  |
| --- |
| **Name** |
| Bob |
| Cathy |
| David |

## Natural Join

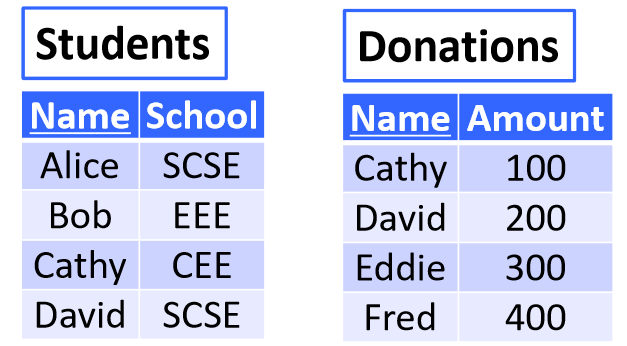
* Combine the rows of two tables based on a **common attribute** in the tables
  + Useful when information about the same item is stored in multiple tables
* Example: Find the Name, NRIC, and Phone of each student with a phone



* + Result:

|  |  |  |
| --- | --- | --- |
| **NRIC** | **Name** | **Number** |
| 11 | Alice | 9123234 |
| 11 | Alice | 8635168 |
| 33 | Cathy | 8213654 |

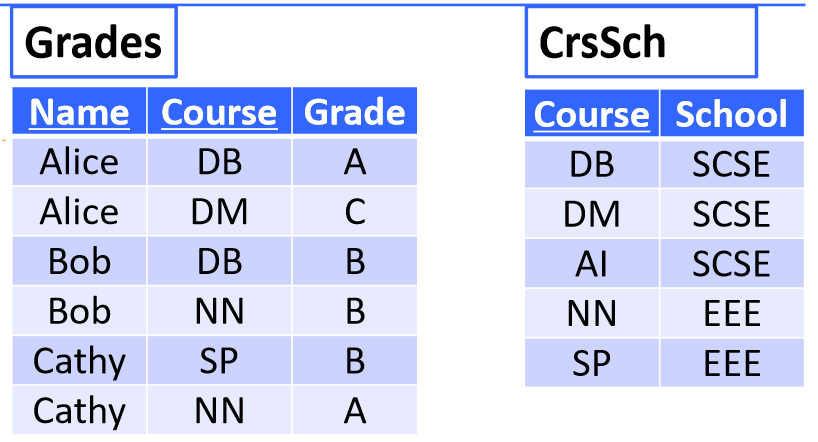
* Example: Find the names, schools, and amounts of SCSE students who have made a donation



* + Result:

|  |  |  |
| --- | --- | --- |
| **Name** | **School** | **Amount** |
| David | SCSE | 200 |

**Exercise**: Find students who have taken SCSE courses but not EEE courses

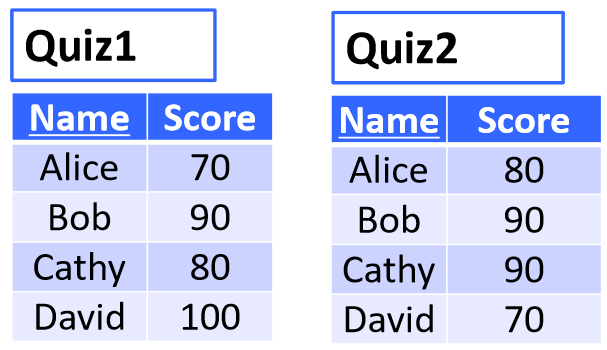


* Result:

|  |
| --- |
| **Name** |
| Alice |
| Bob |

## Theta Join

* When keys don’t have the same name in a table, specify which attribute should match
* Condition can also be **inequality**
  + Duplicates will not be removedfrom results
* Example: Find students who scored higher on quiz 2 than quiz 1

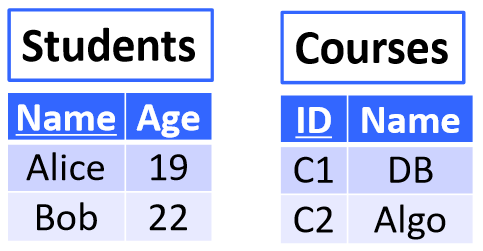


* + Result:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Score** | **Name** | **Score** |
| Alice | 70 | Alice | 80 |
| Cathy | 80 | Cathy | 90 |

## Cartesian Product

* Produces **all possible combinations** of the rows in two tables
* Example: Find all student-course combinations



* + Result:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Age** | **ID** | **Name** |
| Alice | 19 | C1 | DB |
| Alice | 19 | C2 | Algo |
| Bob | 22 | C1 | DB |
| Bob | 22 | C2 | Algo |

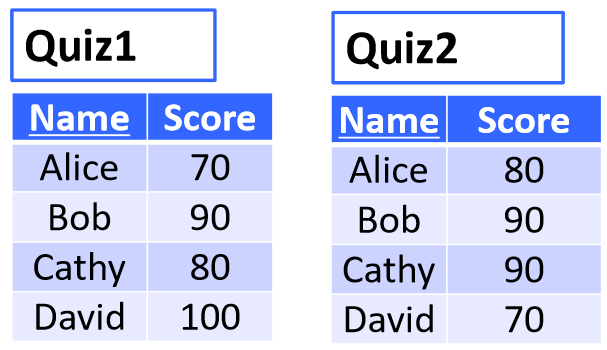
## Assignment

* Assign an intermediate step a name, to break steps down
* E.g.

## Rename

* Changes table and attribute names of result table
* Rename table:
* Rename table and attributes:

**Exercise**:



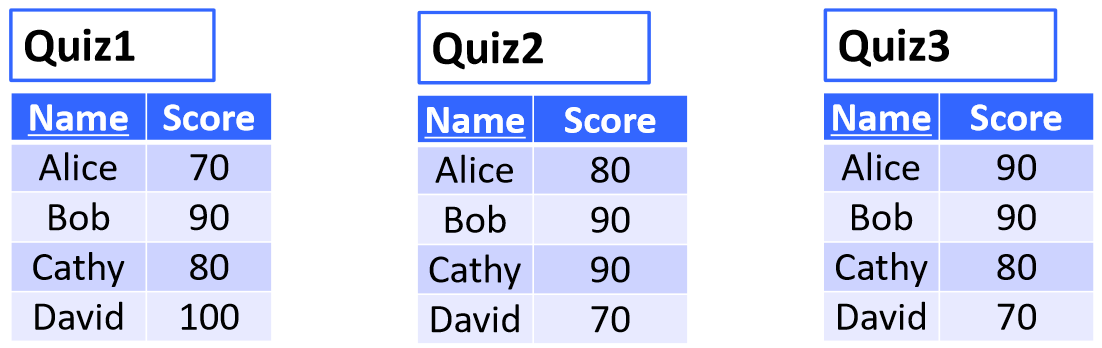
* Find the students who scored higher than Cathy on Quiz 1

Result:

|  |  |
| --- | --- |
| **Name1** | **Score1** |
| Bob | 90 |
| David | 100 |

* Find the student who scored the highest on Quiz 1

**Exercise**: Find the students whose scores in quizzes keep increasing



## Duplicate Elimination

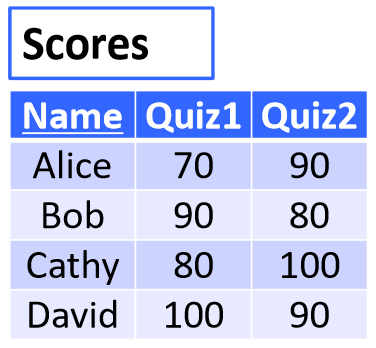
* Eliminates **duplicate tuples**
* Example: Find the list of products sold on 2017.01.01



* + )

## Extended Projection

* Allows the **creation of new attributes through arithmetic**
* Example: Find the total score from Quiz 1 and Quiz 2, for each student

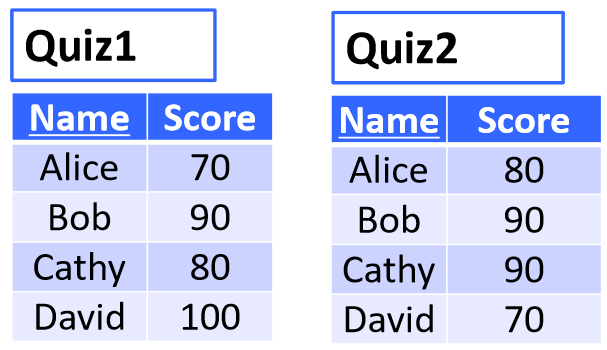


Result:

|  |  |
| --- | --- |
| **Name** | **Total** |
| Alice | 160 |
| Bob | 170 |
| Cathy | 180 |
| David | 190 |

## Grouping/Aggregation

* Create table with attribute using MAX, MIN, AVG, SUM, COUNT functions **on a column**
* Example: Find the highest score in Quiz 1

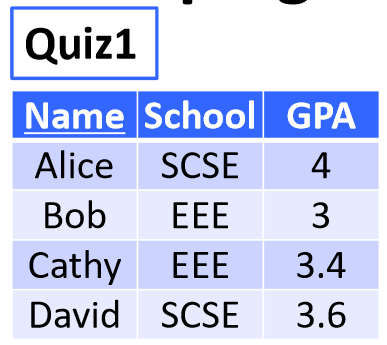


* + Result:

|  |
| --- |
| **MaxScore** |
| 100 |

## Grouping/Aggregation

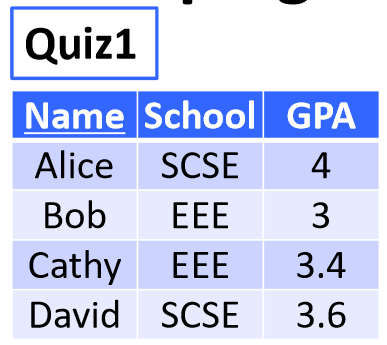
* Divide rows into separate groups based on a column, then compute the arithmetic function on each group
* Example: Find the average GPA in each school



Result:

|  |  |
| --- | --- |
| **School** | **AvgGPA** |
| SCSE | 3.8 |
| EEE | 3.2 |

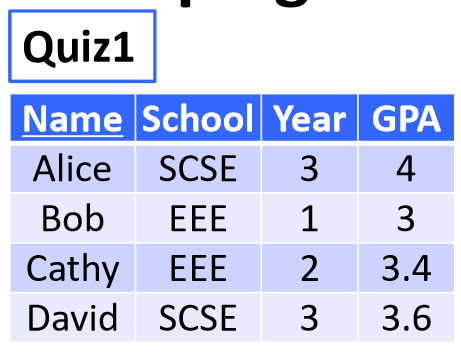
* Example: Find the average and highest GPA in each school



Result:

|  |  |  |
| --- | --- | --- |
| **School** | **AvgGPA** | **MaxGPA** |
| SCSE | 3.8 | 4 |
| EEE | 3.2 | 3.4 |

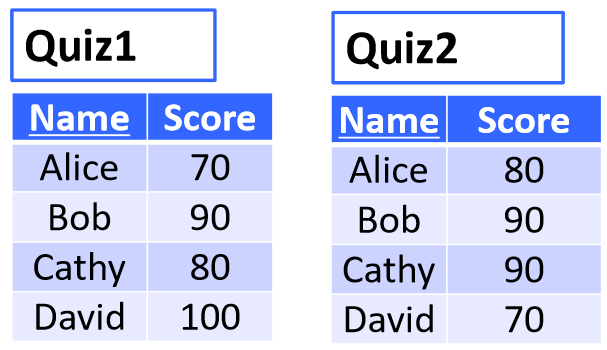
* Example: Find the average GPA in each year in each school



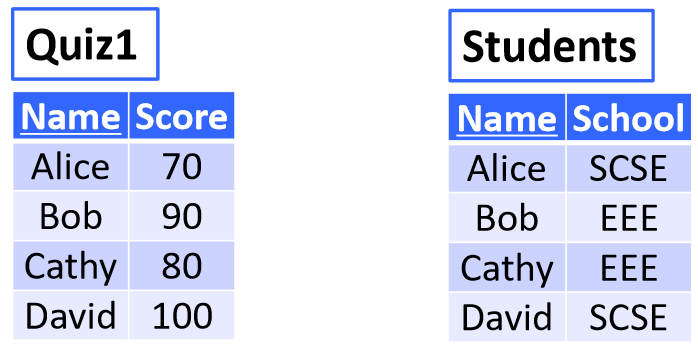
Result:

|  |  |  |
| --- | --- | --- |
| **School** | **Year** | **AvgGPA** |
| SCSE | 3 | 3.8 |
| EEE | 1 | 3 |
| EEE | 2 | 3.4 |

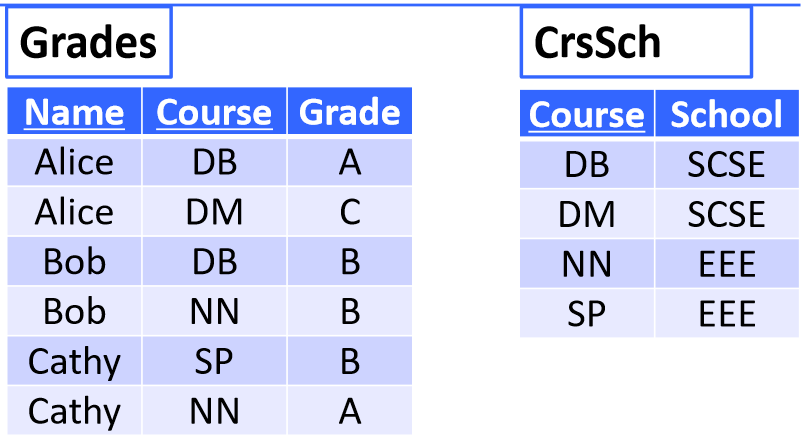
**Exercise**: Find the student that scored the second highest on Quiz 1



**Exercise**: For each school, find the student that scored the highest on Quiz 1



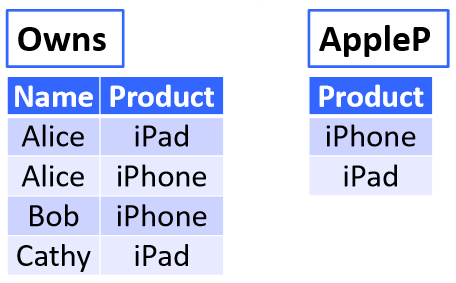
**Exercise**:



* Find the students who have taken all courses from SCSE
* Find the students who have taken all courses from each school

## Division

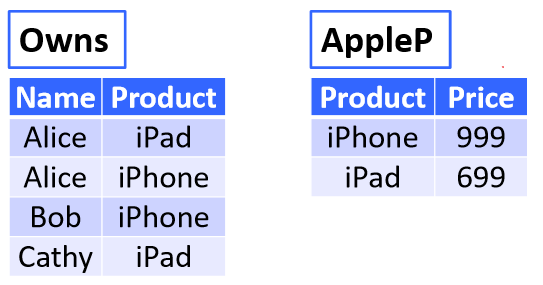
* Faster way to find columns in one table that **contain all values of a column** in another table
  + will return a table only containing
* Example: Find the people who own all Apple products



Result:

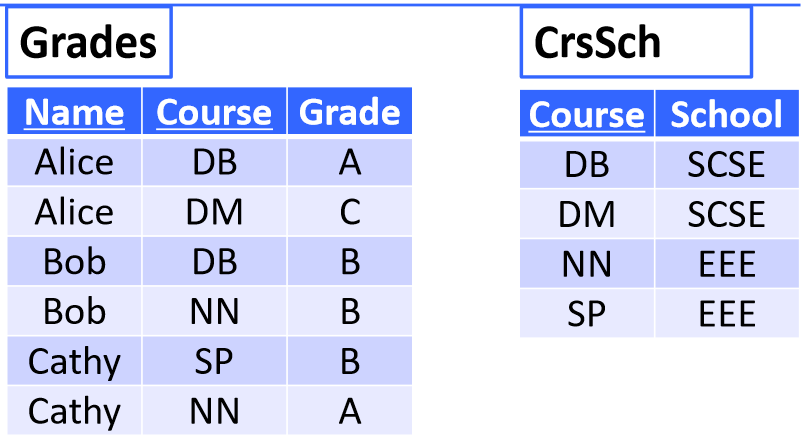
|  |
| --- |
| **Name** |
| Alice |

* Example: Find the people who own all Apple products

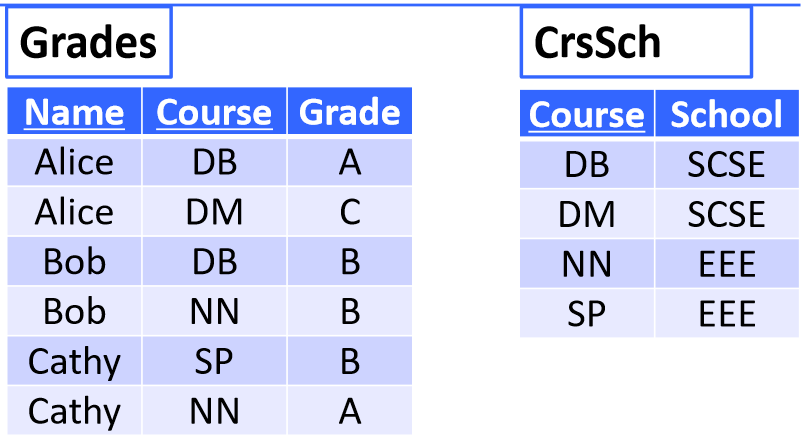


* + Need to eliminate Price column in AppleP first –

**Exercise**: Find the students who have taken all courses from SCSE



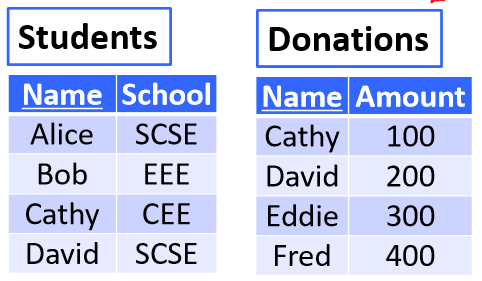
**Exercise**: Find the students who have taken all courses from SCSE but not EEE



* First, find students who have taken all courses from SCSE:
* Next, find students who have taken all courses from EEE:
* Finally, find the difference:

## Left Outerjoin

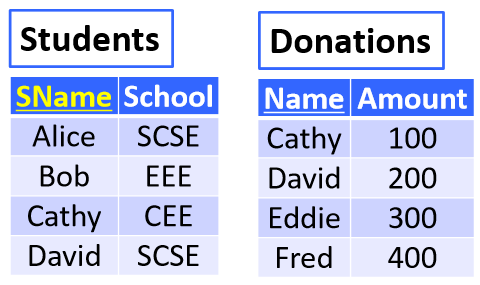
* Retains all attributes of the **left table** in the results, with NULL values for attributes of the right table
  + Similar to theta join, all attributes are retained
* Example: Find the donation information for each student



Result:

|  |  |  |
| --- | --- | --- |
| **Name** | **School** | **Amount** |
| Alice | SCSE | NULL |
| Bob | EEE | NULL |
| Cathy | CEE | 100 |
| David | SCSE | 200 |

* Example: Find the donation information for each student

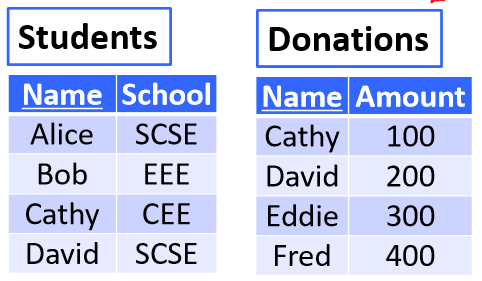


Result:

|  |  |  |  |
| --- | --- | --- | --- |
| **SName** | **School** | **Name** | **Amount** |
| Alice | SCSE | NULL | NULL |
| Bob | EEE | NULL | NULL |
| Cathy | CEE | Cathy | 100 |
| David | SCSE | David | 200 |

## Right Outerjoin

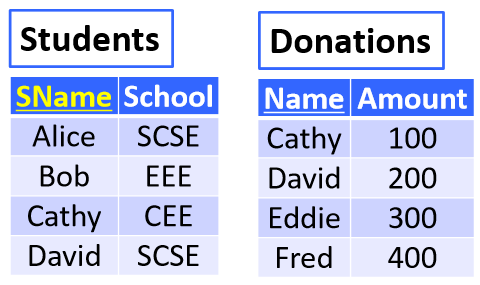
* Retains all attributes of the **right table** in the results, with NULL values for attributes of the left table
  + Similar to theta join, all attributes are retained
* Example: Find the school of each student who made a donation



Result:

|  |  |  |
| --- | --- | --- |
| **Name** | **Amount** | **School** |
| Cathy | 100 | CEE |
| David | 200 | SCSE |
| Eddie | 300 | NULL |
| Fred | 400 | NULL |

* Example: Find the school of each student who made a donation

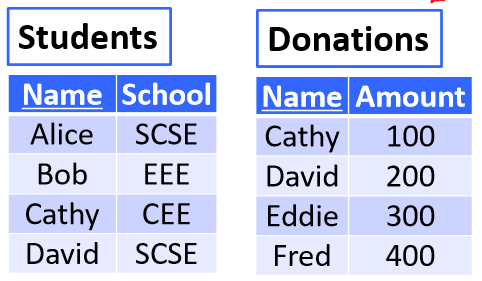


Result:

|  |  |  |  |
| --- | --- | --- | --- |
| **SName** | **School** | **Name** | **Amount** |
| Cathy | CEE | Cathy | 100 |
| David | SCSE | David | 200 |
| NULL | NULL | Eddie | 300 |
| NULL | NULL | Fred | 400 |

## Full Outerjoin

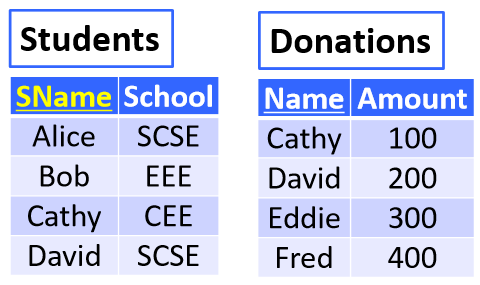
* Retains all attributes of **both tables** in the results, with NULL values for missing attributes
  + Similar to theta join, all attributes are retained
* Example:



* + Result:

|  |  |  |
| --- | --- | --- |
| **Name** | **School** | **Amount** |
| Alice | SCSE | NULL |
| Bob | EEE | NULL |
| Cathy | CEE | 100 |
| David | SCSE | 200 |
| Eddie | NULL | 300 |
| Fred | NULL | 400 |

* Example:



* + Result:

|  |  |  |  |
| --- | --- | --- | --- |
| **SName** | **School** | **Name** | **Amount** |
| Alice | SCSE | NULL | NULL |
| Bob | EEE | NULL | NULL |
| Cathy | CEE | Cathy | 100 |
| David | SCSE | David | 200 |
| NULL | NULL | Eddie | 300 |
| NULL | NULL | Fred | 400 |