

# Introduction to Database

Jianjun Chen

# Contents

- Database introduction
- Relational model
- Relational Keys

# What is Data?

- Example 1: An array that stores some numbers, which can be retrieved sometime later.

```
private int workloadWeekly[] = {2, 3, 5, 2, 1, 9};

public int getWorkload(String dayOfWeek) {
    switch (dayOfWeek.toLowerCase()) {
        case "monday":
            return workloadWeekly[0];
        case "tuesday":
            return workloadWeekly[1];
        ...
    }
}

public void increaseAllWorkload() {
    ...
}
```

# What is Data?

- Example 2: A piece of data inside CPU register

```
int main(void)
{
    int x = 0;
    x = 1;
    return x;
}
```

```
main:
.LFB0:
    pushq    %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe    %rbp, 0
    subq    $48, %rsp
    .seh_stackalloc 48
    .seh_endprologue
    call    main
    movl    $0, -4(%rbp)
    movl    $1, -4(%rbp)
    movl    -4(%rbp), %eax
    addq    $48, %rsp
    popq    %rbp
    ret
```

"`-4(%rbp)`" is where the variable `x` is stored

# What is Data?

We can see that:

- Data is only meaningful under its designed scenario.
  - In example 1: The array `workloadWeekly` is private, thus cannot be used outside of its class.
  - In example 2: The data stored in “-4(%rbp)” is invalid once the function returns.
- Must have ways to create/modify data.
- Must have ways to access data.

# What is Database?

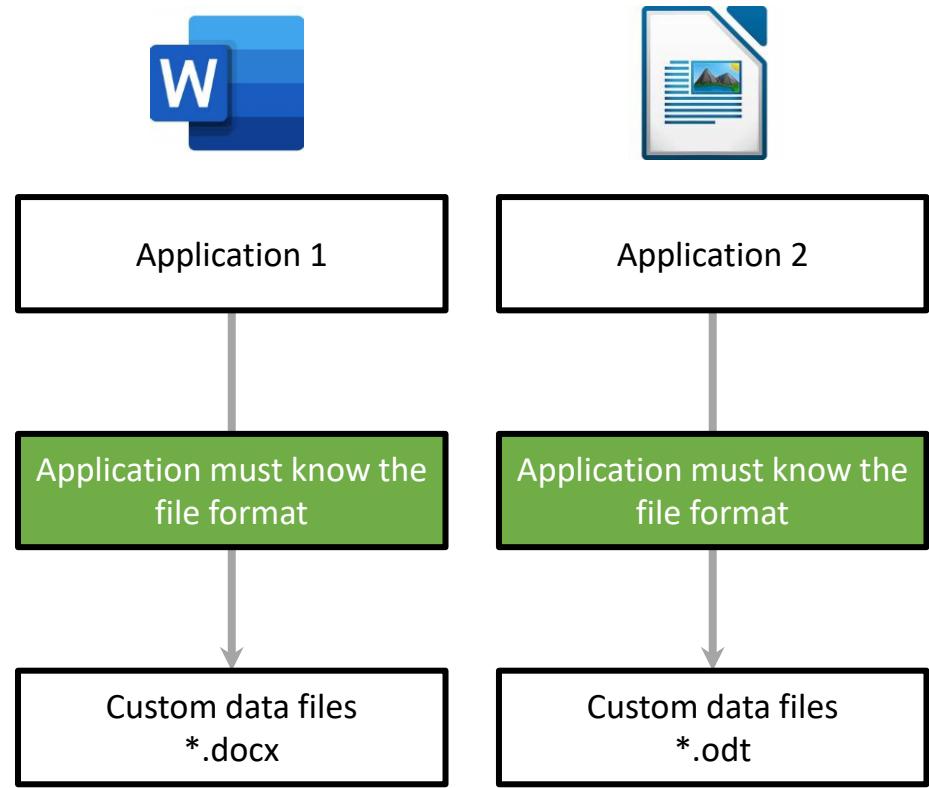
Database: “Organised collection of data. Structured, arranged for ease and speed of search and retrieval.”

Database Management System (DBMS):

- **Designed scenario:** Software that is designed to enable users and programs to store, retrieve and update data from database.
- **Create/Modify/Access:** A software must have a set of standard functions to be called DBMS.
  - We will learn about these functions soon!
- But why do we need DBMS?

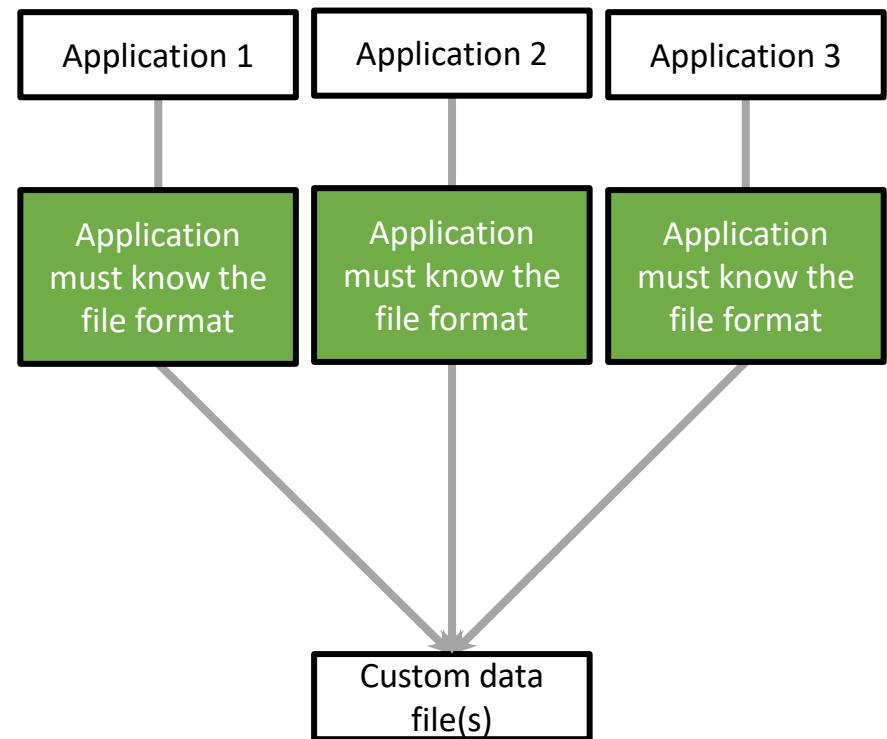
# Pre-DBMS Methods

- Applications store data as files.
- Each application uses its own format.
- Other applications need to understand that specific format.
  - Leads to duplicated code and wasted effort.
  - Compatibility issues.



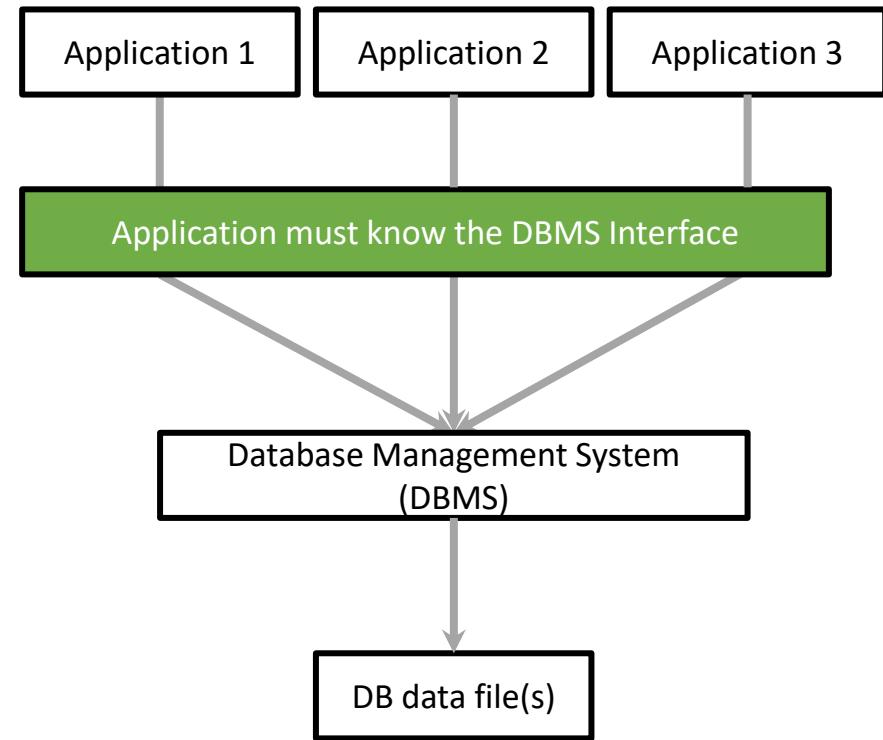
# Pre-DBMS Methods

- How about using a common data format?
- Still need to write duplicated code for reading this file format.
- Synchronisation issues.
  - Accessed simultaneously?
  - Very hard to coordinate operations from different apps.



# DBMS Approach

- Work as a delegate for this common collection of data.
- Applications use a common API for accessing database.
  - A header file in C, plus the dynamic/static library file.
  - A Java Interface, plus a set of class files or a single “.jar” file



We will see

# Commonly Seen DBMS

- Oracle
- DB2
- MySQL
- Ingres
- PostgreSQL
- Microsoft SQL Server
- MS Access

# Databases: Applications

- Example 1: Member cards of a chain store

Phone No.	Name	Points
233333	Vincent	1000
233334	Matt	1231

- Example 2: Banking service, account balance.

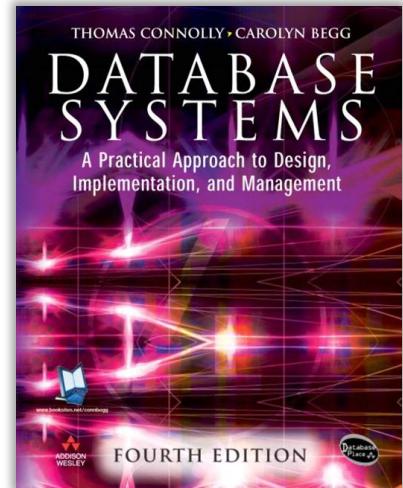
Card ID	Holder ID	Name	Balance
0933 1223 0001 4321	12360	Daryl XXXX	-50
0963 1245 0291 0177	78799	Jessie XXXX	233333

# DBMS Functions / Must Haves

- Allow users to store, retrieve and update data
- Ensure either that all the updates corresponding to a given action are made or that none of them is made (Atomicity)
- Ensure that DB is updated correctly when multiple users are updating it concurrently
- Recover the DB in the event it is damaged in any way
- Ensure that only authorised users can access the DB
- Be capable of integrating with other software

# About this Module

- Module convener's email address:
  - [Jianjun.Chen@xjtu.edu.cn](mailto:Jianjun.Chen@xjtu.edu.cn)
- Textbook:
  - Database Systems: a practical approach to design, implementation, and management - Connolly, Thomas M., Begg, Carolyn E.
- Assessments:
  - A two-stage coursework (10% + 15%)
  - 1 In-class exam at the end of this semester (15%)
  - One final exam.



# The Relational Model

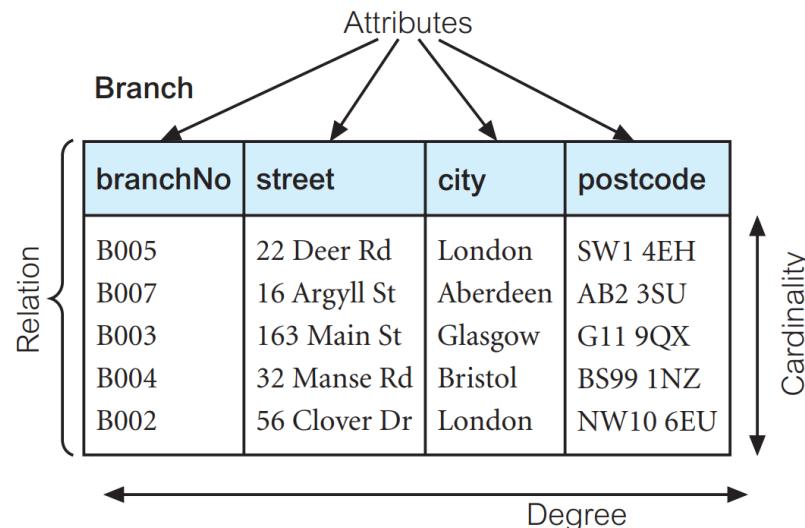
And the relational database management systems (RDBMS)

# The Relational Model

- The relational model is **one approach** to managing data. Originally Introduced by E.F. Codd in his paper “A Relational Model of Data for Large Shared Databanks”, 1970.
  - An earlier model is called the navigational model.
  - RDBMS are based on the relational model.
- The model uses a structure and language that is consistent with **first-order predicate logic**
  - Provides a declarative method for specifying data and queries
  - Details are covered in the Chapter 4 of the textbook.

# Relational Model: Terminologies\*

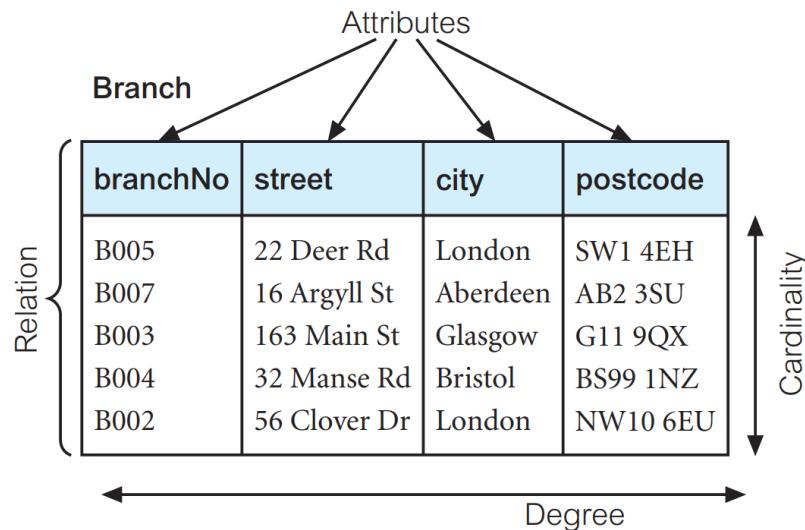
- A **relation** is a mathematical concept. The physical form of a relation is a **table** with columns and rows.
- An **attribute** is a named **column** of a relation.
- A **domain** is the set of **allowable values** for attributes.
  - Age must be a positive integer.
  - Postcodes have length limit.



\*A red title indicates that the knowledge is needed throughout the module

# Terminologies

- Attribute
- Domain
- **Tuple**: a tuple is a **row** of a relation. (order of tuples does not matter)
- The **degree** of a relation is the **number of attributes** it contains
- **Cardinality**: the number of tuples in a relation.



# Terminologies

- **Relation schema:**

- The definition of a relation, which contains the name and domain of each attribute.
- **Formally** (See Chapter 4.2.3): “A named relation defined by a set of attribute and domain name pairs”

- **Relational database schema:**

- A set of relation schemas, each with a distinct name

branchNO	Character: size 4, range B001-B999
street	Character: size 25
city	Character: size 15
postcode	Character: size 8

# Alternative Terminologies

<b>Formal Terms</b>	<b>Alternative #1</b>	<b>Alternative #2</b>
Relation	Table	File
Tuple	Row	Record
Attribute	Column	Field

Table 4.1 in the textbook

Relation schema:

relation\_name(ID: Char, Name: Char, Salary: Monetary, Department: Char)

Attributes are: ID, Name, Salary & Department

The degree of the relation is 4

ID	Name	Salary	Department
M139	John Smith	18000	Marketing
M140	Mary Jones	22000	Marketing
A368	Jane Brown	22000	Accounts
P222	Mark Brown	24000	Personnel
A367	David Jones	20000	Accounts

Tuples, e.g.  
{  
  (ID, A368),  
  (Name, Jane Brown),  
  (Salary, 22,000),  
  (Department, Accounts)  
}

The cardinality of the relation is 5

# Relation: Properties

- A relation also has the following properties:
  - Its name is unique in the relational database scheme.
  - Each cell contains exactly one atomic value.
  - Each attribute of a relation must have a distinct name.
  - The values of an attribute are from the same domain.
  - No duplicate tuples.
  - The order of attributes has no significance.
  - The order of tuples has no significance.

# Relational Keys

Super key, candidate key, primary key, foreign key

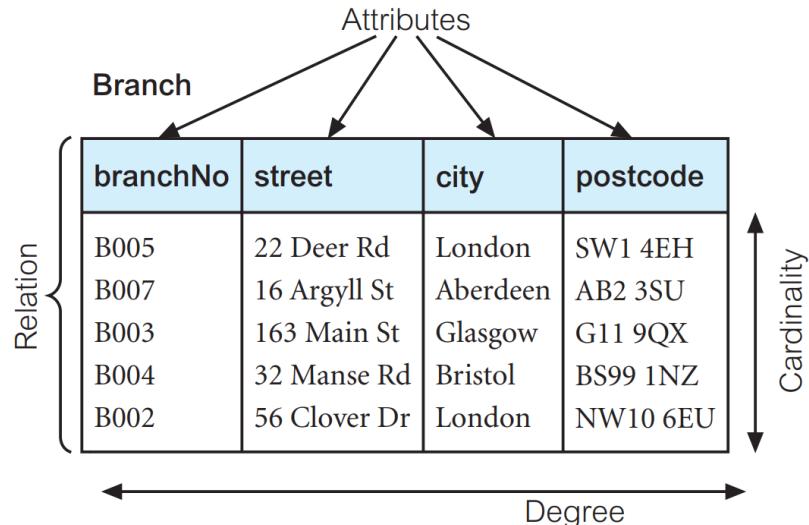
NULL

# Data Integrity

- In this section, you will be introduced to a very important part of the relational model called relational keys.
- In relational databases, related information are often grouped together to form tables.
  - For example, in a company that has many branch offices:
    - The information related to staff is put together.
    - The information related to office is saved in another table.
- Also, there will be a lot of referencing among tables.
  - For example, given an ID of a staff, you should be able to locate the address of his office.

# Integrity Consideration 1

- We want to minimise data redundancy within a relation.
  - The database should be able to check for any duplicate tuples before tuples are added or modified
- This is enforced by something called **Primary key**.



# Relational Keys: Super Key

- **Super key:** one or more attributes that uniquely identifies a tuple within a relation.

Passport ID	Student ID	Name

- {Passport ID} {Name} alone does not uniquely identify a tuple.
- {Passport ID, Name} Because there are people with the same name.
- {Student ID}
- {Student ID, Name}
- {Passport ID, Student ID}
- {Passport ID, Student ID, Name}

# Relational Keys: Candidate Key & Primary Key

- **Candidate key:** a super key such that no proper subset is a super key within the relation
  - Every tuple has a unique value for that set of attributes: uniqueness
  - No proper subset of the set has the uniqueness property: minimality
- **Primary key:** The candidate key that is selected to identify tuples uniquely within the relation.

Passport ID	Student ID	Name

Available choices:

- {Passport ID}
- {Passport ID, Name}
- {Student ID}
- {Student ID, Name}
- {Passport ID, Student ID}
- {Passport ID, Student ID, Name}

# Choosing Candidate Keys

- You can't necessarily infer the candidate keys based solely on the data in your table
  - More often than not, an instance of a relation will only hold a small subset of all the possible values
  - E.g. Restaurants' booking number might reset to 1 after a large number.



Queue No. A31  
Table type: up to 4  
  
31 People are waiting  
ahead of you.

A1, A2, A3... A99, A999 -> A1

- You must use knowledge of the real-world to help.

# Choosing Candidate Keys

- What are the possible candidate keys of the following relation?

OfficeID	Name	Country	Postcode	Phone
O1001	Headquarters	UK	W1 1AA	0044 20 1545 3241
O1002	R&D Labs	UK	W1 1AA	0044 20 1545 4984
O1003	US West	USA	94130	001 415 665981
O1004	US East	USA	10201	001 212 448731
O1005	Telemarketing	UK	NE5 2GE	0044 1909 559862
O1006	Telemarketing	USA	84754	001 385 994763

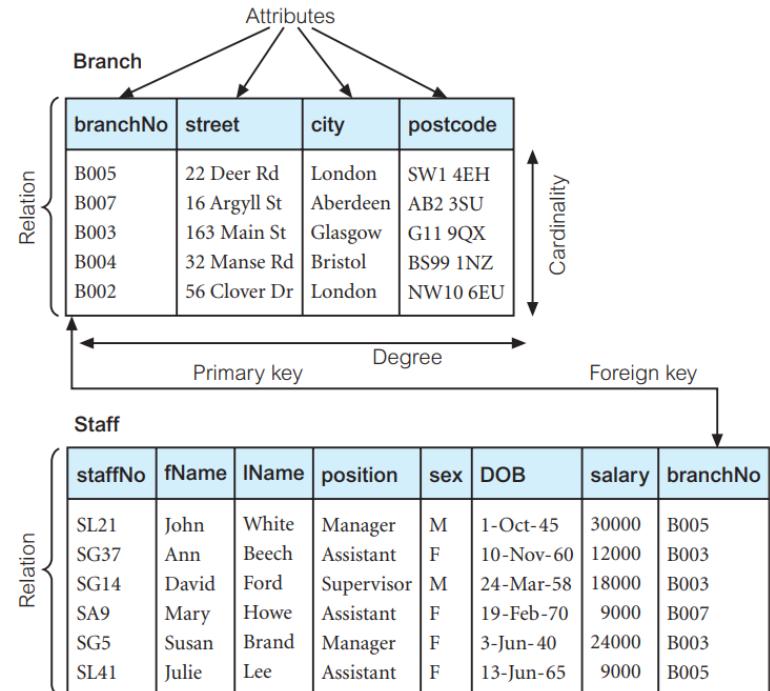
The candidate keys are : {OfficeID}, {Phone}, {Name, Postcode/Zip}, {Name, Country}

- Note: Keys like {Name, Postcode/Zip, Phone} satisfy uniqueness, but not minimality

OfficeID	Name	Country	Postcode	Phone
O1001	Headquarters	UK	W1 1AA	0044 20 1545 3241
O1002	R&D Labs	UK	W1 1AA	0044 20 1545 4984
O1003	US West	USA	94130	001 415 665981
O1004	US East	USA	10201	001 212 448731
O1005	Telemarketing	UK	NE5 2GE	0044 1909 559862
O1006	Telemarketing	USA	84754	001 385 994763

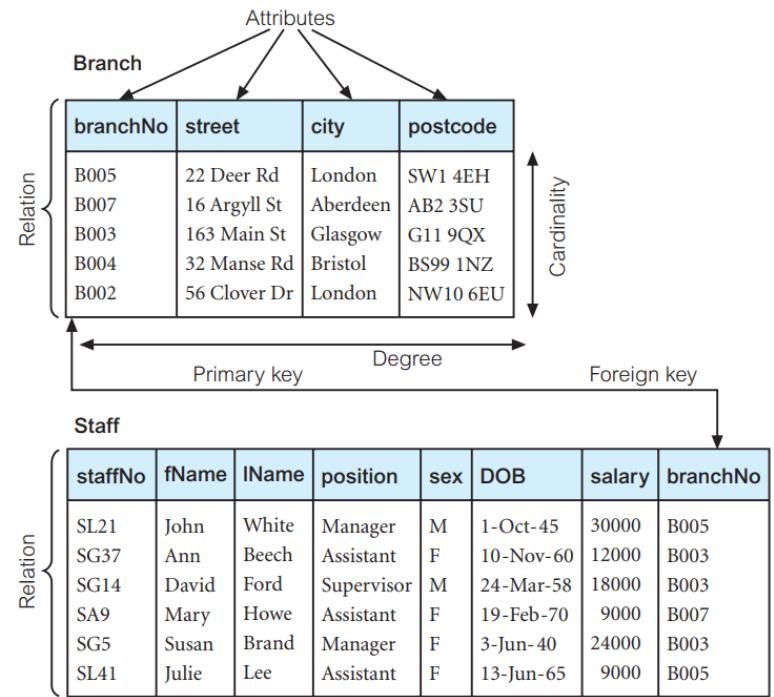
# Integrity Consideration 2

- It is also very common that tuples in one relation references data from another relation.
  - As a result, a database should provide such mechanism to ensure correct references.
- This is enforced by something called **foreign key**



# Relational Keys: Foreign Key

- **Foreign key:**
  - One or more attributes within one relation that **must match** the candidate key of some (possibly the same) relation
  - Example: We want the values of the ‘branchNo’ in relation staff to be one of the ‘branchNo’ in relation Branch.

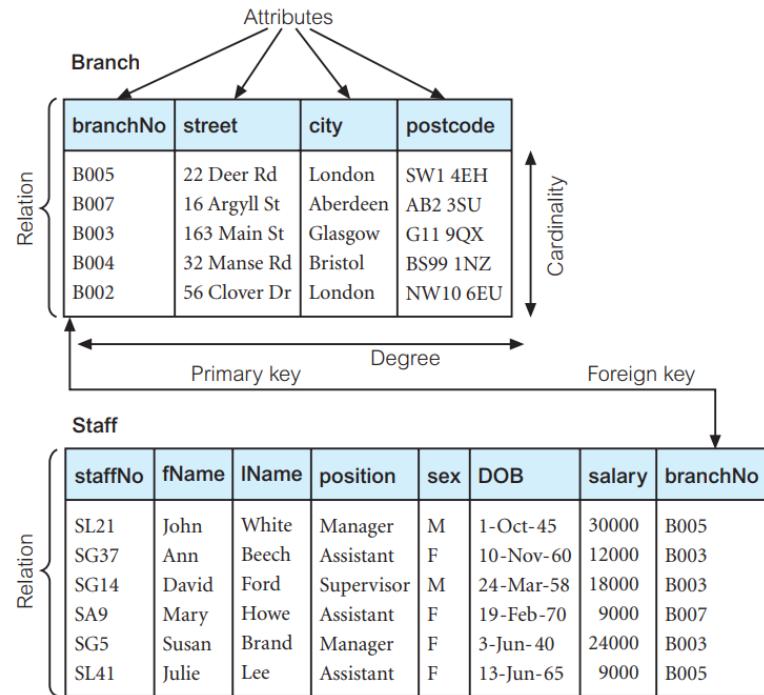


# Relational Keys and Integrity Constraints

- Primary Key enforces **entity integrity**
- Foreign Key enforces **referential integrity**
- Why they are important? Read [here](#). You need more lectures to understand that, though.

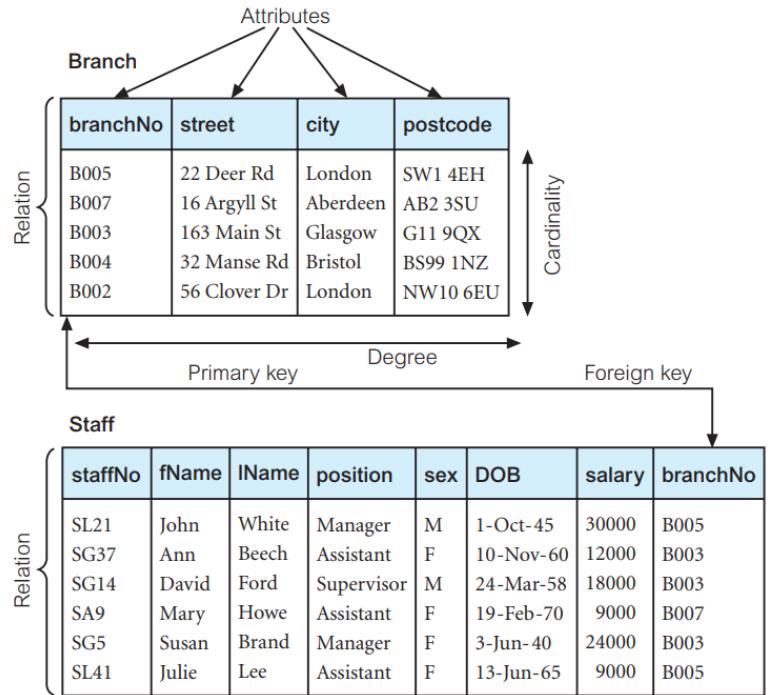
# A Small Challenge

- Assume that the two tables on the right are stored in 2-dimensional arrays:
  - String[][] Branch
  - String[][] Staff
- How would you find out the postcode of the branch where Julie Lee works in? (In Java code)



# A Small Challenge

- Now, write a function that allows the caller to find out the branch information of any Staff.
- Doing so really helps understand future contents.



```
String find(staffID, branchAttributeName)
```

# Extended Reading

- “Database Systems: A Practical Approach ...” by Connolly and Begg.
- Go through Chapter 1 to Chapter 4 quickly.
- Focus more on chapter 4.

# Software Preparation For Labs

- Prepare your personal laptop. Install the necessary software for this course:
  - MySQL Database or MariaDB.
  - A visual database administration tool like phpMyAdmin or MySQL workbench.
- A tutorial will be available on ICE.

# Relational Algebra

Jianjun Chen

# Contents

- Basic operators in relational algebra.
- Cartesian product and Joins.
- Aggregations, division and grouping.

# Relational Algebra

DBMS: Software that is designed to enable users and programs to store, retrieve and update data.

- We need a language to describe these operations.
- The language is called Structured Query Language (SQL)
- What is relational algebra?
  - Theoretical foundation of (part of) SQL.

# Relational Algebra

- “Find all universities with > 20000 students”
  - Relational Algebra:
    - $\pi_{uName}(\sigma_{Enrollment>20000}(University))$
  - SQL:
    - `SELECT uName FROM University WHERE University.Enrollment > 20000`
- Relational Algebra and SQL are declarative (Not procedural like C/Java)
  - No need to specify the steps of data processing.

# Relational Algebra: Operator Properties

- Like functions in Java and C, operators in relational algebra requires operands and returns results.
  - One of the operands must be a relation.
  - The “return result” is a temporary relation (called **View**) that has been processed.
- The returned relation can be then processed by another operator.
  - This property is called **closure**: the ability that allows expressions to be nested. (Textbook: Chapter 5.1)

# Relational Algebra VS SQL

- Relational algebra is set-based, that means duplicate tuples in results are always eliminated.
- Duplicates are kept in SQL results.

# Relational Algebra: Basic Operators

# Selection: Sigma ( $\sigma$ )

- Usage:  $\sigma_{predicate}(R)$
- The Selection operation works on a single relation R and defines a relation that contains only those tuples of R that satisfy the specified condition (predicate).
- *List all staffs with a salary greater than £10,000*
  - $\sigma_{\text{salary}>10000}(\text{staff})$
  - $\sigma_{\text{staff.salary}>10000}(\text{staff})$
  - Useful when a column name appears in two tables
- Predicate also supports logical operators  $\wedge$  (AND),  $\vee$  (OR) and  $\sim$  (NOT).
  - $\sigma_{\text{salary}>10000 \wedge \text{salary}<20000}(\text{staff})$

Predicate: a function with parameters that either returns a true or false.

## Staff

staffNo	fName	IName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

$\sigma_{\text{salary} > 10000}(\text{staff})$



staffNo	fName	IName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003

# Projection: PI ( $\pi$ )

- Usage:  $\pi_{a_1, a_2, a_3 \dots, a_n}(R)$
- The Projection operation works on a single relation R and defines a relation that contains a vertical subset of R, extracting the values of specified attributes and eliminating duplicates.
- Produce a list of salaries for all staff, showing only the staffNo, fName, lName, and salary details.

$$\pi_{\text{staffNo}, \text{fName}, \text{lName}, \text{salary}}(R)$$

## Staff

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

$\pi_{\text{staffNo}, \text{fName}, \text{lName}, \text{salary}}(\text{Staff})$



staffNo	fName	lName	salary
SL21	John	White	30000
SG37	Ann	Beech	12000
SG14	David	Ford	18000
SA9	Mary	Howe	9000
SG5	Susan	Brand	24000
SL41	Julie	Lee	9000

# Union: $\cup$

- Usage:  $R_1 \cup R_2$
- The union of two relations  $R_1$  and  $R_2$  defines a relation that contains all the tuples of  $R_1$ , or  $R_2$ , or both  $R_1$  and  $R_2$ , duplicate tuples being eliminated.  $R_1$  and  $R_2$  must be union-compatible.
- List all cities where there is either a branch office or a property for rent

$$\pi_{\text{city}}(\text{Branch}) \cup \pi_{\text{city}}(\text{PropertyForRent})$$

Branch

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

PropertyForRent

propertyNo	street	city	postcode
PA14	16 Holhead	Aberdeen	AB7 5SU
PL94	6 Argyll St	London	NW2
PG4	6 Lawrence St	Glasgow	G11 9QZ
PG36	2 Manor Rd	Glasgow	G32 4QZ
PG21	18 Dale Rd	Glasgow	G12
PG16	5 Novar Dr	Glasgow	G12 9AZ

$\pi_{\text{city}}(\text{Branch}) \cup \pi_{\text{city}}(\text{PropertyForRent})$

city
London
Aberdeen
Glasgow
Bristol

# Union Compatible

- UNION-compatible means that the numbers of attributes must be the same and their corresponding data types also match
- **Union compatible:**  
A: (First\_name (char), Last\_name(char), Date\_of\_Birth(date))  
B: (FName(char), LName(char), DOB(date))  
**Both table have 3 attributes and of same date type.**
- **Not union compatible:**  
A: (First\_name (char), Last\_name(char), Date\_of\_Birth(date))  
B: (FName(char), LName(char), PhoneNumber(number))  
  
A: (First\_name (char), Date\_of\_Birth(date), Last\_name(char))  
B: (FName(char), LName(char), DOB(date))

# Set Difference: Minus -

- Usage:  $R - S$
- The Set difference operation defines a relation consisting of the tuples that are in relation R, but not in S. R and S must be union-compatible.
- List all cities where there is a branch office but no properties for rent.

$$\pi_{\text{city}}(\text{Branch}) - \pi_{\text{city}}(\text{PropertyForRent})$$

Branch

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

PropertyForRent

propertyNo	street	city	postcode
PA14	16 Holhead	Aberdeen	AB7 5SU
PL94	6 Argyll St	London	NW2
PG4	6 Lawrence St	Glasgow	G11 9QX
PG36	2 Manor Rd	Glasgow	G32 4QX
PG21	18 Dale Rd	Glasgow	G12
PG16	5 Novar Dr	Glasgow	G12 9AX



$\pi_{\text{city}}(\text{Branch}) - \pi_{\text{city}}(\text{PropertyForRent})$



city
Bristol

# Intersection $\cap$

- Usage:  $R \cap S$
- The Intersection operation defines a relation consisting of the set of all tuples that are in both R and S. R and S must be union-compatible.
- Same as:  $R - (R - S)$
- List all cities where there is both a branch office and at least one property for rent.

$$\pi_{\text{city}}(\text{Branch}) \cap \pi_{\text{city}}(\text{PropertyForRent})$$

Branch

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

PropertyForRent

propertyNo	street	city	postcod
PA14	16 Holhead	Aberdeen	AB7 5SU
PL94	6 Argyll St	London	NW2
PG4	6 Lawrence St	Glasgow	G11 9QX
PG36	2 Manor Rd	Glasgow	G32 4QX
PG21	18 Dale Rd	Glasgow	G12
PG16	5 Novar Dr	Glasgow	G12 9AX



$$\pi_{\text{city}}(\text{Branch}) \cap \pi_{\text{city}}(\text{PropertyForRent})$$


city
Aberdeen
London
Glasgow

# Cartesian Product and Joins

# Cartesian Product

- Usage:  $R \times S$
- The Cartesian product operation defines a relation that is the concatenation of every tuple of relation R with every tuple of relation S.
- E.g:

$$(\Pi_{\text{clientNo}, \text{fName}, \text{iName}}(\text{Client})) \times (\Pi_{\text{clientNo}, \text{propertyNo}, \text{comment}}(\text{Viewing}))$$

- The result is presented in the next slide.

## Client

clientNo	fName	IName	telNo	prefTy
CR76	John	Kay	0207-774-5632	Flat
CR56	Aline	Stewart	0141-848-1825	Flat
CR74	Mike	Ritchie	01475-392178	House
CR62	Mary	Tregeare	01224-196720	Flat

## Viewing

clientNo	propertyNo	viewDate	comment
CR56	PA14	24-May-04	too small
CR76	PG4	20-Apr-04	too remote
CR56	PG4	26-May-04	
CR62	PA14	14-May-04	no dining room
CR56	PG36	28-Apr-04	

$$(\prod_{\text{clientNo}, \text{fName}, \text{IName}} (\text{Client})) \times (\prod_{\text{clientNo}, \text{propertyNo}, \text{comment}} (\text{Viewing}))$$

	client.clientNo	fName	IName	Viewing.clientNo	propertyNo	comment
Client[0] + Viewing[0]	CR76	John	Kay	CR56	PA14	too small
Client[0] + Viewing[1]	CR76	John	Kay	CR76	PG4	too remote
Client[0] + Viewing[2]	CR76	John	Kay	CR56	PG4	
Client[0] + Viewing[3]	CR76	John	Kay	CR62	PA14	no dining room
Client[0] + Viewing[4]	CR76	John	Kay	CR56	PG36	
Client[1] + Viewing[0]	CR56	Aline	Stewart	CR56	PA14	too small
Client[1] + Viewing[1]	CR56	Aline	Stewart	CR76	PG4	too remote
Client[1] + Viewing[2]	CR56	Aline	Stewart	CR56	PG4	
Client[1] + Viewing[3]	CR56	Aline	Stewart	CR62	PA14	no dining room
Client[1] + Viewing[4]	CR56	Aline	Stewart	CR56	PG36	
Client[2] + Viewing[0]	CR74	Mike	Ritchie	CR56	PA14	too small
Client[2] + Viewing[1]	CR74	Mike	Ritchie	CR76	PG4	too remote
Client[2] + Viewing[2]	CR74	Mike	Ritchie	CR56	PA14	

client.clientNo	fName	lName	Viewing.clientNo	propertyNo	comment
CR76	John	Kay	CR56	PA14	too small
CR76	John	Kay	CR76	PG4	too remote
CR76	John	Kay	CR56	PG4	
CR76	John	Kay	CR62	PA14	no dining room
CR76	John	Kay	CR56	PG36	
CR56	Aline	Stewart	CR56	PA14	too small
CR56	Aline	Stewart	CR76	PG4	too remote
CR56	Aline	Stewart	CR56	PG4	
CR56	Aline	Stewart	CR62	PA14	no dining room
CR56	Aline	Stewart	CR56	PG36	
CR74	Mike	Ritchie	CR56	PA14	too small
CR74	Mike	Ritchie	CR76	PG4	too remote
CR74	Mike	Ritchie	CR56	PG4	
CR74	Mike	Ritchie	CR62	PA14	no dining room
CR74	Mike	Ritchie	CR56	PG36	
CR62	Mary	Tregear	CR56	PA14	too small
CR62	Mary	Tregear	CR76	PG4	too remote
CR62	Mary	Tregear	CR56	PG4	
CR62	Mary	Tregear	CR62	PA14	no dining room
CR62	Mary	Tregear	CR56	PG36	

# Cartesian Product

- The result of the previous operation is not very meaningful.
- But based on that, we can filter out rows and obtain some useful information.
- For example, how to list the names and comments of all clients who have viewed a property for rent?
  - Hint: use selection with a predicate

We want a result like this



client.clientNo	fName	IName	Viewing.clientNo	propertyNo	comment
CR76	John	Kay	CR76	PG4	too remote
CR56	Aline	Stewart	CR56	PA14	too small
CR56	Aline	Stewart	CR56	PG4	
CR56	Aline	Stewart	CR56	PG36	
CR62	Mary	Tregear	CR62	PA14	no dining room

client.clientNo	fName	lName	Viewing.clientNo	propertyNo	comment
CR76	John	Kay	CR56	PA14	too small
CR76	John	Kay	CR76	PG4	too remote
CR76	John	Kay	CR56	PG4	
CR76	John	Kay	CR62	PA14	no dining room
CR76	John	Kay	CR56	PG36	
CR56	Aline	Stewart	CR56	PA14	too small
CR56	Aline	Stewart	CR76	PG4	too remote
CR56	Aline	Stewart	CR56	PG4	
CR56	Aline	Stewart	CR62	PA14	no dining room
CR56	Aline	Stewart	CR56	PG36	

client.clientNo	fName	lName	Viewing.clientNo	propertyNo	comment
CR76	John	Kay	CR76	PG4	too remote
CR56	Aline	Stewart	CR56	PA14	too small
CR56	Aline	Stewart	CR56	PG4	
CR56	Aline	Stewart	CR56	PG36	
CR62	Mary	Tregear	CR62	PA14	no dining room

$\sigma_{\text{Client.clientNo} = \text{Viewing.clientNo}}((\Pi_{\text{clientNo}, \text{fName}, \text{lName}}(\text{Client})) \times (\Pi_{\text{clientNo}, \text{propertyNo}, \text{comment}}(\text{Viewing})))$

# Theta Join $\bowtie_F$

- Usage:  $R \bowtie_F S$
- The Theta join defines a relation that contains tuples satisfying the predicate  $F$  from  $R \times S$ . The predicate  $F$  is of the form " $R.a_i \theta S.b_i$ " where  $\theta$  may be one of the comparison operators ( $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $=$ ,  $\neq$ ).
  - If  $F$  contains only equality ( $=$ ), it is instead called **Equijoin**.
- The previous example can be simply represented by a **Theta Join**:  $R \bowtie_F S$ 
  - $R \bowtie_F S = \sigma_F (R \times S)$

# Theta Join $\bowtie_F$

- List the names and comments of all clients who have viewed a property for rent:

$$(\Pi_{\text{clientNo}, \text{fName}, \text{IName}}(\text{Client})) \bowtie_{\text{Client.clientNo} = \text{Viewing.clientNo}} (\Pi_{\text{clientNo}, \text{propertyNo}, \text{comment}}(\text{Viewing}))$$

client.clientNo	fName	IName	Viewing.clientNo	propertyNo	comment
CR76	John	Kay	CR76	PG4	too remote
CR56	Aline	Stewart	CR56	PA14	too small
CR56	Aline	Stewart	CR56	PG4	
CR56	Aline	Stewart	CR56	PG36	
CR62	Mary	Tregear	CR62	PA14	no dining room

Same result as the previous one ;)

# Natural Join $\bowtie$

- Usage:  $R \bowtie S$
- The Natural join is an Equijoin of the two relations  $R$  and  $S$  over all common attributes  $x$ .
- One occurrence of each common attribute is eliminated from the result. (see clientNo in the next slide)
- List the names and comments of all clients who have viewed a property for rent.

$$(\Pi_{\text{clientNo}, \text{fName}, \text{lName}}(\text{Client})) \bowtie (\Pi_{\text{clientNo}, \text{propertyNo}, \text{comment}}(\text{Viewing}))$$

Client

clientNo	fName	IName	telNo	prefTy
CR76	John	Kay	0207-774-5632	Flat
CR56	Aline	Stewart	0141-848-1825	Flat
CR74	Mike	Ritchie	01475-392178	House
CR62	Mary	Tregear	01224-196720	Flat

Viewing

clientNo	propertyNo	viewDate	comment
CR56	PA14	24-May-04	too small
CR76	PG4	20-Apr-04	too remote
CR56	PG4	26-May-04	
CR62	PA14	14-May-04	
CR56	PG36	28-Apr-04	no dining room

$$(\Pi_{\text{clientNo}, \text{fName}, \text{IName}}(\text{Client})) \bowtie (\Pi_{\text{clientNo}, \text{propertyNo}, \text{comment}}(\text{Viewing}))$$

The extra  
clientNo is  
removed.

See theta  
join for  
comparison

clientNo	fName	IName	propertyNo	comment
CR76	John	Kay	PG4	too remote
CR56	Aline	Stewart	PA14	too small
CR56	Aline	Stewart	PG4	
CR56	Aline	Stewart	PG36	
CR62	Mary	Tregear	PA14	no dining room

# Natural Join

- The natural Join works based on attributes names and their types (domain)
- If two tables have multiple attributes with the same names and data types, then all these attributes will be selected. For example:

A: (**FName (char)**, LName(char), Date\_of\_Birth(date))

B: (**FName(char)**, LName(char), PhoneNumber(number))

# Natural Join $\bowtie$ : A Question

What should the natural join of these two tables look like?

FName	LName	Date of Birth
John	Smith	2000-1-1
Lily	Adam	2001-1-11

FName	LName	Phone No
Jason	Smith	111
Lily	Adam	222

# Left Outer Join:



- Usage:  $R \bowtie S$
- The (left) Outer join is a join in which tuples from R that do not have matching values in the common attributes of S are also included in the result relation. Missing values in the second relation are set to null.
  - “Left (Outer numbers) right”.
  - Like natural join, but use Nulls for missing values in table S.
- Produce a status report on property viewings.

$$(\Pi_{\text{propertyNo, street, city}}(\text{PropertyForRent})) \bowtie \text{Viewing}$$

## PropertyForRent

propertyNo	street	city	postcc
PA14	16 Holhead	Aberdeen	AB7 5S
PL94	6 Argyll St	London	NW2
PG4	6 Lawrence St	Glasgow	G11 9C
PG36	2 Manor Rd	Glasgow	G32 4C
PG21	18 Dale Rd	Glasgow	G12
PG16	5 Novar Dr	Glasgow	G12 9A

## Viewing

clientNo	propertyNo	viewDate	comment
CR56	PA14	24-May-04	too small
CR76	PG4	20-Apr-04	too remote
CR56	PG4	26-May-04	
CR62	PA14	14-May-04	no dining room
CR56	PG36	28-Apr-04	

$(\Pi_{\text{propertyNo}, \text{street}, \text{city}}(\text{PropertyForRent})) \bowtie \text{Viewing}$

propertyNo	street	city	clientNo	viewDate	comment
PA14	16 Holhead	Aberdeen	CR56	24-May-04	too small
PA14	16 Holhead	Aberdeen	CR62	14-May-04	no dining room
PL94	6 Argyll St	London	null	null	null
PG4	6 Lawrence St	Glasgow	CR76	20-Apr-04	too remote
PG4	6 Lawrence St	Glasgow	CR56	26-May-04	
PG36	2 Manor Rd	Glasgow	CR56	28-Apr-04	
PG21	18 Dale Rd	Glasgow	null	null	null
PG16	5 Novar Dr	Glasgow	null	null	null

## PropertyForRent[0]

PA14	16 Holhead	Aberdeen	AB7 5S
------	------------	----------	--------

## Viewing

clientNo	propertyNo	viewDate	comment
CR56	PA14	24-May-04	too small
CR76	PG4	20-Apr-04	too remote
CR56	PG4	26-May-04	
CR62	PA14	14-May-04	no dining room
CR56	PG36	28-Apr-04	

Found two matches:

PA14	16 Holhead	Aberdeen	CR56	24-May-04	too small
PA14	16 Holhead	Aberdeen	CR62	14-May-04	no dining room

## PropertyForRent[1]

PL94	6 Argyll St	London	NW2
------	-------------	--------	-----

## Viewing

clientNo	propertyNo	viewDate	comment
CR56	PA14	24-May-04	too small
CR76	PG4	20-Apr-04	too remote
CR56	PG4	26-May-04	
CR62	PA14	14-May-04	no dining room
CR56	PG36	28-Apr-04	

No matches found, add Null

PL94	6 Argyll St	London	null	null	null
------	-------------	--------	------	------	------

# Left Outer Join: R $\bowtie$ S

- Q: What if “Viewing” has more “PropertyNo” than “PropertyForRent”?
- A: Check the steps in the previous slide.
- A: Or check this [Wikipedia page](#): can you work it out by yourself? ☺
- Right Outer Join and full outer join also exist.
  - Right outer join ( $\bowtie$ ): Just the opposite of left outer Join.
  - Full outer join ( $\bowtie\!\!\bowtie$ ): Find it out!

# Rename Operator: rho $\rho$

- $\rho_s(E)$  or  $\rho_{s(a_1, a_2, \dots, a_n)}(E)$
- The Rename operation provides a new name S for the expression E, and optionally names the attributes as  $a_1, a_2, \dots, a_n$ .
- Why rename operator?
  - E.g. Natural Join relies on the attribute name to work properly.
  - Some times the attributes of two different tables have different names, but they actually refer to the same kind of information.
- A: (First\_name (char), Last\_name(char), Date\_of\_Birth(date))  
B: (FName(char), LName(char), DOB(date))

# Rename Operator: rho $\rho$

- $\rho_s(E)$  or  $\rho_{s(a_1, a_2, \dots, a_n)}(E)$
- $\rho_{uni}(University)$  changes “University” table to “uni”
- $\rho_{b(col1,col2,col3,col4)}(Branch)$  changes “Branch” table to “b” without changing its tuples.

col1	col2	col3	col4
Tuples	Stay	The	Same

# Division, Aggregation, Grouping

# Division: $\div$

- Usage:  $R \div S$
- Division is used when we wish to express queries with “all”:
  - “Which persons have a bank account at ALL the banks in the country?”
  - “Which students are registered on ALL the courses”
- Assume relation R is defined over the attribute set A and relation S is defined over the attribute set B such that  $B \subseteq A$  (B is a subset of A).

Identify all clients who have viewed all properties with three rooms:

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003



$\Pi_{\text{clientNo}, \text{propertyNo}}(\text{Viewing})$

$\Pi_{\text{propertyNo}}(\sigma_{\text{rooms}=3}(\text{PropertyForRent}))$

clientNo	propertyNo
CR56	PA14
CR76	PG4
CR56	PG4
CR62	PA14
CR56	PG36

propertyNo
PG4
PG36

B: (propertyNo)

A: (ClientNo, propertyNo)

$(\Pi_{\text{clientNo}, \text{propertyNo}}(\text{Viewing})) \div (\Pi_{\text{propertyNo}}(\sigma_{\text{rooms}=3}(\text{PropertyForRent})))$

$$\Pi_{\text{clientNo}, \text{propertyNo}}(\text{Viewing}) \quad \Pi_{\text{propertyNo}}(\sigma_{\text{rooms}=3}(\text{PropertyForRent}))$$

clientNo	propertyNo
CR56	PA14
CR76	PG4
CR56	PG4
CR62	PA14
CR56	PG36

propertyNo
PG4
PG36

$$(\Pi_{\text{clientNo}, \text{propertyNo}}(\text{Viewing})) \div (\Pi_{\text{propertyNo}}(\sigma_{\text{rooms}=3}(\text{PropertyForRent})))$$


RESULT

clientNo
CR56

clientNo	propertyNo	propertyNo
CR56	PA14	PG4
CR76	PG4	PG36
CR56	PG4	
CR62	PA14	
CR56	PG36	

# Division: $\div$

Formal definition as in our textbook:

- Assume relation R is defined over the attribute set A and relation S is defined over the attribute set B such that  $B \subseteq A$  (B is a subset of A).
- Let  $C = A - B$ , that is, C is the set of attributes of R that are not attributes of S (**clientNo**). The Division operation defines a relation over the attributes C that consists of the set of tuples from R that match the combination of every tuple (**pg4, pg36**) in S.

# Aggregate:



- Usage:  $\mathfrak{I}_{AL}(R)$
- Applies the aggregate function list,  $AL$ , to the relation  $R$  to define a relation over the aggregate list.  $AL$  contains one or more ( $<\text{aggregate\_function}>$ ,  $<\text{attribute}>$ ) pairs.
  - The symbol is “capital I” (according to the system symbol table).
- Aggregate functions:
  - COUNT: returns the number of values in the associated attribute.
  - SUM: returns the sum of the values in the associated attribute.
  - AVG: returns the average of the values in the associated attribute.
  - MIN: returns the smallest value in the associated attribute.
  - MAX: returns the largest value in the associated attribute.

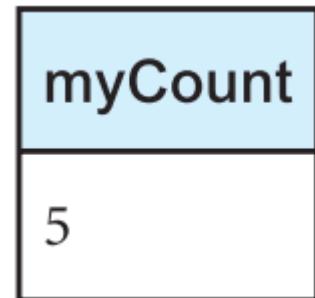
# How many properties cost more than £350 per month to rent?

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003



$\rho_r(myCount)(\text{COUNT } propertyNo(\sigma_{rent > 350}(\text{PropertyForRent})))$



# Find the minimum, maximum, and average staff salary

Staff

staffNo	fName	IName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005



$\rho_r(myMin, myMax, myAverage)(\textcolor{red}{\exists MIN \ salary, MAX \ salary, AVERAGE \ salary}(\text{Staff}))$



myMin	myMax	myAverage
9000	30000	17000

# Grouping

- Usage:  ${}_{GA}\tilde{\mathfrak{I}}_{AL}(R)$ 
  - Groups the tuples of relation R by the grouping attributes, GA.
  - And then applies the aggregate function list AL to define a new relation. AL contains one or more (`<aggregate_function>`, `<attribute>`) pairs.
  - The resulting relation contains the grouping attributes, GA, along with the results of each of the aggregate functions.
- Simplified: find aggregation result for each different value in GA.

# Grouping

- Find the number of staff working in each branch and the sum of their salaries.

$$\rho_r(branchNo, myCount, mySum)(branchNo \Join COUNT staffNo, SUM salary(Staff))$$

Staff

staffNo	fName	IName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

Staff

staffNo	fName	IName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005



$$\rho_r(branchNo, myCount, mySum)(branchNo \exists COUNT staffNo, SUM salary(Staff))$$


branchNo	myCount	mySum
B003	3	54000
B005	2	39000
B007	1	9000

# Extended Reading

- Find out the information about the full outer join.
- Can you describe how full outer join works?
- Use pseudo code to describe the process of all operators covered in this lecture.
- Read our textbook, chapter 5.1

# SQL 1: Defining & Modifying Tables

Jianjun Chen

# Contents

- SQL stands for “Structured Query Language”.
- SQL consists of two parts:
  - Data definition language (DDL).
  - Data manipulation language (DML).
- Today, we will cover the DDL and a small portion of DML:
  - Create tables and their constraints.
  - Changing columns/constraints of tables.
  - Adding/Updating/Removing tuples.

# SQL Format in the Slides

- SQL statements will be written in
  - **BOLD COURIER FONT**
- SQL keywords are not case-sensitive, but we'll write SQL keywords in uppercase for clarity.
- However, table names, column names etc. are case sensitive. For example:
  - **SELECT (sName) FROM Student;**

# Database Containment Hierarchy

- A computer may have one or more **clusters**.
  - A cluster is a database server.
  - = a computer can run multiple database servers.
- Each cluster contains one or more **catalogs**.
  - Catalog is just another name for “database”.
- Each catalog consists of set of **schemas**.
  - Schema is a namespace of tables, and security boundary.
- A schema consists of tables, views, domains, assertions, collations, translations, and character sets. All have same owner.

# More about Catalog and Schema

- A very clear discussion is available [here](#).
  - But It can be confusing at the current stage because you haven't learned so much about database.
- Feel free to come back later.

# Creating a Database

- First, we need to create a schema
  - **CREATE SCHEMA** name ; }
  - **CREATE DATABASE** name ; }
- Same effect
- If you want to create tables in this schema, you need to tell MySQL to “enter” into this schema, type:
  - **USE** name ;
- After that, if you create tables, they will be created in this schema.

**phpMyAdmin**

Recent Favorites

- New
- cse103
- information\_schema
- mysql
- performance\_schema
- phpmyadmin

Server: 127.0.0.1

Databases SQL Status User accounts Export More

Run SQL query/queries on server “127.0.0.1”:

```
CREATE SCHEMA mySchema;
```

Bind parameters

Bookmark this SQL query:

[ Delimiter  ]  Show this query here again  Retain query box  Rollback when finished  
 Enable foreign key checks



**phpMyAdmin**

Recent Favorites

- New
- cse103
- information\_schema
- myschema**
- mysql
- performance\_schema
- phpmyadmin

Server: 127.0.0.1

Databases SQL Status User accounts Export More

Show query box

MySQL returned an empty result set (i.e. zero rows). (Query took 0.0019 seconds.)

```
CREATE SCHEMA mySchema
```

[\[Edit inline\]](#) [\[ Edit \]](#) [\[ Create PHP code \]](#)

Error: #1046 No database selected

# SQL: Table Definition

Syntax of “CREATE TABLE”

Data types of SQL

# Create Tables

```
CREATE TABLE name (
    col-name datatype [col-options] ,
    :
    col-name datatype [col-options] ,
    [constraint-1] ,
    :
    [constraint-n]
) ;
```

[xxx]: something optional.

# Common Data Types (MySQL)

DataType	keyword
Boolean	BOOLEAN
Character	CHAR(size)    VARCHAR(size_limit)    TEXT    TINYTEXT
Number	INTEGER(size)    FLOAT(size, d)    DOUBLE(size,d)
Date	DATE    DATETIME    TIME

**FLOAT(size,d)**    d  
12345666666.7777  
size

**CHAR** has fixed string length  
**VARCHAR** has variable string length

**Date:** 1-oct-2015  
**Time:** 21:05:02

col-name **datatype** [col-options]

# Strings In SQL

- Strings in SQL are surrounded by single quotes:
  - '**I AM A STRING**'
- Single quotes within a string are doubled or escaped using \
  - '**I''M A STRING**'
  - '**I\'M A STRING**'
  - '' - is an empty string
- In MySQL, double quotes also work (Not a standard)

# Column Options

**col-name datatype [col-options]**

- **NOT NULL:**
  - values of this column cannot be null.
- **UNIQUE:**
  - each value must be unique (candidate key on a single attribute)
- **DEFAULT** value:
  - Default value for this column if not specified by the user.
  - Does not work in MS Access.

# Column Options

- **AUTO\_INCREMENT** = baseValue:

```
CREATE TABLE Persons (
    Id INT AUTO_INCREMENT,
    ...
```

You cannot use  
“AUTO\_INCREMENT = 2”  
inside CREATE TABLE

- a value (usually  $\text{max}(\text{col}) + 1$ ) is automatically inserted when data is added.
- You can also manually provide values to override this behaviour:

```
ALTER TABLE Persons AUTO_INCREMENT = 100;
```

- Read the official manual [here](#).

# Create Tables: Full Example

```
CREATE TABLE Persons (
    ID INT UNIQUE NOT NULL,
    LastName VARCHAR(255) NOT NULL,
    FirstName VARCHAR(255) ,
    Age INT ,
    City VARCHAR(255)
) ;
```

# Try It Yourself

- Convert the following relation into SQL query.
- You need to choose appropriate data types as well.
- No need to add tuples right now.

Branch

branchNo	street	city	postCode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

# Constraints

Domain, Primary key, unique key, foreign key

# Domain Constraints

- You can limit the possible values of an attribute by adding a **domain constraint**.
- A domain constraint can be defined along with the column or separately:

```
CREATE TABLE People (
    id INTEGER PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    sex CHAR NOT NULL CHECK (sex IN ('M', 'F'))
);
```

- Unfortunately, MySQL does not support domain constraints, these constraints will be ignored.

# Constraints

- General Syntax:
  - **CONSTRAINT** name **TYPE details;**
  - If you don't provide a name, one will be generated
- MySQL provides following constraint types
  - **PRIMARY KEY**
  - **UNIQUE**
  - **FOREIGN KEY**
  - **INDEX**

# UNIQUE

- Usage:
  - **CONSTRAINT** name **UNIQUE** (col1, col2, ...)
- Same effect as the one specified with column options but can be applied to multiple columns and make them one candidate key.
- The following candidate keys are different
  - One candidate key (a, b, c):
    - Tuples (1, 2, 3) and (1, 2, 2) are allowed
  - Separate candidate keys (a) (b) (c):
    - Tuples (1, 2, 3) and (1, 2, 2) are NOT allowed

# Primary Key

- Usage:
  - **CONSTRAINT** name **PRIMARY KEY** (col1, col2, ...)
- **PRIMARY KEY** also automatically adds **UNIQUE** and **NOT NULL** to the relevant column definition
- Extended reading:
  - [Difference](#) between Primary Key and Unique.
  - [More underlying mechanism.](#)

# Primary Key: Example

```
CREATE TABLE Branch (
    branchNo CHAR(4) ,
    street VARCHAR(100) ,
    city VARCHAR(25) ,
    postcode VARCHAR(7) ,
    CONSTRAINT branchPK
        PRIMARY KEY (branchNo)
)
```

# Foreign Key

- To apply a foreign key, you need to provide
  - The columns which make up the foreign key
  - The referenced table
  - The columns which are referenced by the foreign key
  - You can optionally provide reference options

Usage:

```
CONSTRAINT name  
FOREIGN KEY  
    (col1, col2, ...)  
REFERENCES  
table-name  
    (col1, col2, ...)  
[ON UPDATE ref_opt  
ON DELETE ref_opt]  
  
ref_opt: RESTRICT |  
CASCADE | SET NULL |  
SET DEFAULT
```

# Foreign Key

- Important:

- When a foreign key is applied, the value must either reference the other table or set to NULL.
- In the following tables, the Staff .branchNo must either use a value from the branch table or set to null.
- **The referenced column must be a candidate key (or primary key)**

Staff

staffNo	fName	IName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

Branch

branchNo	street	city	postCode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

## Try it Yourself:

- Write a SQL query to create the table for Staff. (optional)
- Create a foreign key for Staff.branchNo that references Branch.branchNo.

```
CONSTRAINT name
  FOREIGN KEY
    (col1, col2, ...)
  REFERENCES
    table-name
    (col1, col2, ...)
```

Staff

staffNo	fName	IName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

Branch

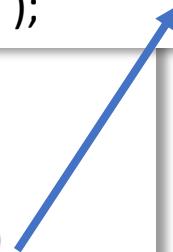
branchNo	street	city	postCode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

# Foreign Key

- The referenced column **must be** a candidate key (or primary key)

```
CREATE TABLE staff (
    staffNo VARCHAR(8),
    fName VARCHAR(20),
    lName VARCHAR(20),
    position VARCHAR(20),
    sex VARCHAR(10),
    DOB DATE,
    salary INTEGER,
    branchNo CHAR(4),
    CONSTRAINT staffPK PRIMARY KEY (staffNo),
    CONSTRAINT staffFK FOREIGN KEY (branchNo)
        REFERENCES Branch (branchNo)
);
```

```
CREATE TABLE Branch (
    branchNo CHAR(4),
    street VARCHAR(100),
    city VARCHAR(25),
    postcode VARCHAR(7),
    CONSTRAINT branchPK PRIMARY KEY  
(branchNo)
);
```



# Foreign Keys and Tuple Updates

- We know that adding non-existing branchNo to Staff will be rejected by DBMS.
- But what happens when we change/delete existing branchNo in Branch that are being referenced by Staff?
- What strategies can you think of if you were the designer?

# Reference Options

- There are several options when this occurs:
  - **RESTRICT** – stop the user from doing it
    - The default option
  - **CASCADE** – let the changes flow on
  - **SET NULL** – make referencing values null
  - **SET DEFAULT** – make referencing values the default for their column
- These options can be applied to one or both kinds of the table updates:
  - **ON DELETE**
  - **ON UPDATE**

# On Update/Delete Set NULL

- Assume we delete in Branch table.
  - All 'B005' in the Staff table will be set to null.
- If we change 'B005' to 'B006'.
  - All 'B005' will be set to null...Good decision?

Staff

staffNo	fName	IName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

A blue arrow points from the word "NULL" to the last two occurrences of "B005" in the "branchNo" column of the table.

# On Update/Delete Cascade

- Assume we change ‘B005’ to ‘B006’ in Branch table
  - All ‘B005’ in Staff table will be changed to ‘B006’.
  - Seems reasonable
- Assume we delete ‘B005’ in Branch table.
  - All tuples with ‘B005’ in Staff table will also be deleted!
  - Good decision?

# On Update/Delete Set Default

- Assume we delete or change ‘B005’ in Branch table.
- All ‘B005’ in Staff table will be changed to the default value of Staff.branchNo.
- This feature is not available in MySQL.

# Final FK Definition in Staff Table

```
CONSTRAINT staffFK  
  FOREIGN KEY (branchNo)  
  REFERENCES Branch (branchNo)  
    ON DELETE SET NULL  
    ON UPDATE CASCADE
```

# Alternative Ways of Writing

- Primary keys, unique keys and foreign keys can also be directly defined along with attributes.
- But this form has some limitations. **You should find them out.**

```
CREATE TABLE majors (
    majorID VARCHAR(5) PRIMARY KEY
);

CREATE TABLE Students (
    id INTEGER PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    majorID VARCHAR(5) REFERENCES majors(majorID),
    sex CHAR NOT NULL CHECK (sex IN ('M', 'F'))
);
```

# Deleting Tables

- You can delete tables with the **DROP** keyword
  - **DROP TABLE [IF EXISTS] table-name1 , table-name2...;**
- All tuples will be deleted as well.
- Undoing is sometimes impossible.
- Foreign Key constraints will prevent DROPS under the default RESTRICT option
  - To overcome this, either remove the constraint or drop the tables in the correct order (referencing table first)

# Try It!

- Try deleting our previously created tables (Branch, Staff)
- Try both:
  - **DROP TABLE IF EXISTS** Branch, Staff;
  - **DROP TABLE IF EXISTS** Staff, Branch;

# Altering Tables

Keyword ALTER

# Change Tables: ALTER

- ALTER is used to add, delete, or modify columns in an existing table.
- Add column:

```
ALTER TABLE table_name ADD column_name datatype  
[options like UNIQUE ...];
```

- Drop column:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

# ALTER: Modify Columns

- Modify column name and definition:

```
ALTER TABLE table_name  
    CHANGE COLUMN  
        col_name new_col_name datatype [col_options];
```

- Modify column definition only:

```
ALTER TABLE table_name  
    MODIFY COLUMN  
        column_name datatype [col_options];
```

# ALTER: Add Constraints

- To add a constraint:

```
ALTER TABLE table-name  
  ADD CONSTRAINT name definition;
```

- For instance:

```
ALTER TABLE branch  
  ADD CONSTRAINT ck_branch UNIQUE (street);
```

Changes street column of Branch to a candidate key.

# ALTER: Remove Constraints

- To remove a constraint:

**ALTER TABLE** table-name

**DROP INDEX** name | **DROP FOREIGN KEY**  
name | **DROP PRIMARY KEY**

- Separator | means OR

- For example:

**ALTER TABLE** staff **DROP PRIMARY KEY;**

Removes the primary key of staff

# Try It Yourself

- Change the data type of Branch.postcode to VARCHAR(12).
- And then set (city, postcode) as a candidate key.
- And then remove the previously added candidate key.
  - Hint: use **drop index**

# SQL: Tuple Operations

# INSERT

Inserts rows into the database with the specified values:

```
INSERT INTO table-name (col1, col2, ...)  
    VALUES (val1, val2, ...) ,  
            :  
            (val1, val2, val3);
```

If you are adding a value to every column, you don't have to list them:

```
INSERT INTO table-name VALUES (val1, val2, ...);
```

But then the ordering of values must match the ordering of attributes in the table.

# INSERT: Example

```
INSERT INTO Employee  
  (ID, Name, Salary)  
VALUES (2, 'Mary', 26000);
```

```
INSERT INTO Employee  
  (Name, ID)  
VALUES ('Mary', 2);
```

```
INSERT INTO Employee  
VALUES (2, 'Mary', 26000),  
        (3, 'Max', 233333);
```

Employee		
ID	Name	Salary

# INSERT: Example

```
INSERT INTO Employee  
(ID, Name, Salary)  
VALUES (2, 'Mary', 26000);
```

```
INSERT INTO Employee  
(Name, ID)  
VALUES ('Mary', 2);
```

```
INSERT INTO Employee  
VALUES (2, 'Mary', 26000),  
        (3, 'Max', 233333);
```

Employee		
ID	Name	Salary
2	Mary	26000
2	Mary	
2	Mary	26000
3	Max	233333

# Exercise:

- Add ‘B005’ and ‘B002’ to Branch table
  - List columns when inserting ‘B005’.
  - Skip columns when inserting ‘B002’.
  - Can you add a tuple WITHOUT a branchNo?

Branch

branchNo	street	city	postCode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

# UPDATE

- Changes values in specified rows based on WHERE conditions

**UPDATE** table-name

```
SET col1 = val1 [, col2 = val2...]  
[WHERE condition]
```

- All rows where the condition is true have the columns set to the given values
- If no condition is given all rows are changed.
- Values are constants or can be computed from columns

# UPDATE: Example

Employee		
ID	Name	Salary
1	John	25000
2	Mary	26000
3	Mark	18000
4	Anne	22000

```
UPDATE Employee  
SET  
    Salary = 15000,  
    Name = 'Jane'  
WHERE ID = 4;
```

Employee		
ID	Name	Salary
1	John	25000
2	Mary	26000
3	Mark	18000
4	Jane	15000

```
UPDATE Employee  
SET Salary =  
    Salary * 1.05;
```

Employee		
ID	Name	Salary
1	John	26250
2	Mary	27300
3	Mark	18900
4	Anne	23100

# UPDATE: Exercise

- Change the salary of all Staffs whose salary is less than 15000 to 16000.

Staff

staffNo	fName	IName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

# **DELETE**

- Removes all rows, or those which satisfy a condition

**DELETE FROM**

table-name

**[WHERE condition]**

- If no condition is given then ALL rows are deleted.

# DELETE: Example

Employee		
ID	Name	Salary
1	John	25000
2	Mary	26000
3	Mark	18000
4	Anne	22000

**DELETE FROM**  
Employee **WHERE**  
Salary > 20000;

Employee		
ID	Name	Salary
3	Mark	18000

**DELETE FROM**  
Employee;

Employee		
ID	Name	Salary

Questions?

# Additional Exploration

- Check the following reference manuals:
- Create table:
  - <https://dev.mysql.com/doc/refman/5.7/en/create-table.html>
- Data types:
  - <https://dev.mysql.com/doc/refman/5.7/en/data-types.html>

# Some other issues

- Can you create a foreign key on an attribute of INT type that references a CHAR type?
- Can you create multiple primary keys/unique keys on a same relation?
- Please experiment these on your computer.

# SQL 2.1: SELECT

Jianjun

# Contents

- The next three lectures will cover a good portion of “Data manipulation language (DML)” of RDBMS.

# SELECT: Overview

**SELECT [DISTINCT | ALL]**

column-list **FROM** table-names

**[WHERE condition]**

**[ORDER BY column-list]**

**[GROUP BY column-list]**

**[HAVING condition]**

# Examples in This Lecture

Student

ID	First	Last
S103	John	Smith
S104	Mary	Jones
S105	Jane	Brown
S106	Mark	Jones
S107	John	Brown

Grade

ID	Code	Mark
S103	DBS	72
S103	IAI	58
S104	PR1	68
S104	IAI	65
S106	PR2	43
S107	PR1	76
S107	PR2	60
S107	IAI	35

Course

Code	Title
DBS	Database Systems
PR1	Programming 1
PR2	Programming 2
IAI	Introduction to AI

# SELECT

In its simplest form, SELECT is the SQL version of projection:

- $\pi_{\text{attributes}}(R)$

**SELECT** col1 [, col2...] **FROM** table-name;

Selection  $\sigma_{\text{predicate}}(R)$  can be achieved by using a WHERE clause:

**SELECT** \* **FROM** table-name  
**WHERE** predicate;

Star (\*) means all columns of table.

# DISTINCT and ALL

- Sometimes you end up with duplicate entries
- Using **DISTINCT** removes duplicates
- Using **ALL** retains duplicates
- **ALL** is used as a default if neither is supplied
- These will work over multiple columns

```
SELECT ALL Last  
FROM Student;
```

Last
Smith
Jones
Brown
Jones
Brown

```
SELECT DISTINCT Last  
FROM Student;
```

Last
Smith
Jones
Brown

# WHERE Clauses

A **WHERE** clause restricts rows that are returned

- It takes the form of a condition
- only rows that satisfy the condition are returned

Example conditions:

- `Mark < 40`
- `First = 'John'`
- `First <> 'John'`
- `First = Last`
- `(First = 'John') AND (Last = 'Smith')`
- `(Mark < 40) OR (Mark > 70)`

`<>` NOT equal to

# WHERE: Examples

```
SELECT * FROM Grade  
WHERE Mark >= 60;
```

ID	Code	Mark
S103	DBS	72
S104	PR1	68
S104	IAI	65
S107	PR1	76
S107	PR2	60

```
SELECT DISTINCT ID  
FROM Grade  
WHERE Mark >= 60;
```

ID
S103
S104
S107

- Write an SQL query to find a list of the ID numbers and Marks for students who have passed (scored 50% or more) in IAI
- Write an SQL query to find the combined list of the student IDs for both the IAI and PR2 module.

Grade		
ID	Code	Mark
S103	DBS	72
S103	IAI	58
S104	PR1	68
S104	IAI	65
S106	PR2	43
S107	PR1	76
S107	PR2	60
S107	IAI	35

?



ID	Mark
S103	58
S104	65

?



ID
S103
S104
S106
S107
S107

Try It Yourself

# Solution

```
SELECT ID, Mark FROM Grade WHERE  
(Code = 'IAI') AND (Mark >= 50);
```

```
SELECT ID FROM Grade  
WHERE  
(Code = 'IAI' OR Code = 'PR2');
```

# SELECT and Cartesian Product

- Cartesian product of two tables can be obtained by using:

```
SELECT * FROM Table1, Table2;
```

- If the tables have columns with the same name, ambiguity will result
- This can be resolved by referencing columns with the table name:

TableName.ColumnName

# Cartesian Product: Example

**SELECT**

First, Last, Mark

**FROM**

Student, Grade

**WHERE**

(Student.ID =  
Grade.ID)

**AND** (Mark >= 40);

Student

ID	First	Last
S103	John	Smith
S104	Mary	Jones
S105	Jane	
S106	Mary	
S107	John	

ID	Code	Mark
S103	DBS	72
S103	IAI	58
S104	PR1	68
S104	IAI	65
S106	PR2	43
S107	PR1	76
S107	PR2	60
S107	IAI	35

**SELECT . . . FROM Student, Grade WHERE . . .**

ID	First	Last	ID	Code	Mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S103	John	Smith	S104	PR1	68
S103	John	Smith	S104	IAI	65
S103	John	Smith	S106	PR2	43
S103	John	Smith	S107	PR1	76
S103	John	Smith	S107	PR2	60
S103	John	Smith	S107	IAI	35
S104	Mary	Jones	S103	DBS	72
S104	Mary	Jones	S103	IAI	58
S104	Mary	Jones	S104	PR1	68
S104	Mary	Jones	S104	IAI	65

**SELECT . . . FROM** Student, Grade **WHERE**  
**(Student.ID = Grade.ID) AND . . .**

ID	First	Last	ID	Code	Mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S104	Mary	Jones	S104	PR1	68
S104	Mary	Jones	S104	IAI	65
S106	Mark	Jones	S106	PR2	43
S107	John	Brown	S107	PR1	76
S107	John	Brown	S107	PR2	60
S107	John	Brown	S107	IAI	35

```
SELECT . . . FROM Student,Grade  
WHERE (Student.ID = Grade.ID)  
AND (Mark >= 40)
```

ID	First	Last	ID	Code	Mark
S103	John	Smith	S103	DBS	72
S103	John	Smith	S103	IAI	58
S104	Mary	Jones	S104	PR1	68
S104	Mary	Jones	S104	IAI	65
S106	Mark	Jones	S106	PR2	43
S107	John	Brown	S107	PR1	76
S107	John	Brown	S107	PR2	60

```
SELECT First, Last, Mark  
FROM Student, Grade  
WHERE (Student.ID = Grade.ID)  
AND (Mark >= 40)
```

First	Last	Mark
John	Smith	72
John	Smith	58
Mary	Jones	68
Mary	Jones	65
Mark	Jones	43
John	Brown	76
John	Brown	60

# SELECT from Multiple Tables

- WHERE clause is a key feature when selecting from multiple tables.
- Unrelated combinations can be filtered out.

**SELECT \* FROM**

Student, Grade, Course

**WHERE**

Student.ID = Grade.ID **AND**

Course.Code = Grade.Code

Student			Grade			Course	
ID	First	Last	ID	Code	Mark	Code	Title
S103	John	Smith	S103	DBS	72	DBS	Database Systems
S103	John	Smith	S103	IAI	58	IAI	Introduction to AI
S104	Mary	Jones	S104	PR1	68	PR1	Programming 1
S104	Mary	Jones	S104	IAI	65	IAI	Introduction to AI
S106	Mark	Jones	S106	PR2	43	PR2	Programming 2
S107	John	Brown	S107	PR1	76	PR1	Programming 1
S107	John	Brown	S107	PR2	60	PR2	Programming 2

Diagram illustrating two foreign key constraints:

- An arrow from the **Student.ID** column points to the **Grade.ID** column, labeled **Student.ID = Grade.ID**.
- An arrow from the **Grade.Code** column points to the **Course.Code** column, labeled **Grade.Code = Course.Code**.

## Student

sID	sName	sAddress	sYear
1	Smith	5 Arnold Close	2
2	Brooks	7 Holly Avenue	2
3	Anderson	15 Main Street	3
4	Evans	Flat 1a, High Street	2
5	Harrison	Newark Hall	1
6	Jones	Southwell Hall	1

## Enrolment

sID	mCode
1	G52ADS
2	G52ADS
5	G51DBS
5	G51PRG
5	G51IAI
4	G52ADS
6	G51PRG
6	G51IAI

## Module

mCode	mCredits	mTitle
G51DBS	10	Database Systems
G51PRG	20	Programming
G51IAI	10	Artificial Intelligence
G52ADS	10	Algorithms

# Try It Yourself

Write SQL statements to do the following:

- Produce a list of all student names and all their enrolments (module codes)
- Find a list of module titles being taken by the student named “Harrison”
- Find a list of module codes and titles for all modules currently being taken by first year students

Student			
sID	sName	sAddress	sYear

Module		
mCode	mCredits	mTitle

Enrolment	
sID	mCode

```
SELECT sName, mCode  
FROM Student, Enrolment  
WHERE Student.sID = Enrolment.sID;
```

```
SELECT mTitle  
FROM Module, Student, Enrolment  
WHERE (Module.mCode = Enrolment.mCode)  
AND (Student.sID = Enrolment.sID)  
AND Student.sName = 'Harrison';
```

```
SELECT Module.mCode, mTitle  
FROM Enrolment, Module, Student  
WHERE (Module.mCode = Enrolment.mCode)  
AND (Student.sID = Enrolment.sID)  
AND sYear = 1;
```

# Aliases

- Aliases rename columns or tables
  - Can make names more meaningful
  - Can shorten names, making them easier to use
  - Can resolve ambiguous names
- Two forms:
  - Column alias  
**SELECT** column **[AS]** new-col-name
  - Table alias  
**SELECT** \* **FROM** table **[AS]** new-table-name

# Alias Example

Employee	
ID	Name
123	John
124	Mary

WorksIn	
ID	Department
123	Marketing
124	Sales
124	Marketing

**SELECT**

E.ID **AS** empID,  
E.Name, W.Department

**FROM**

Employee E,  
WorksIn W

**WHERE**

E.ID = W.ID;

Note: You cannot use a column alias in a WHERE clause

# Alias Example

emplID	Name	Department
123	John	Marketing
124	Mary	Sales
124	Mary	Marketing

**SELECT**

E.ID **AS** empID,  
E.Name, W.Department

**FROM**

Employee E,  
WorksIn W

**WHERE**

E.ID = W.ID;

# Aliases and ‘Self-Joins’

- Aliases can be used to copy a table, so that it can be combined with itself:

```
SELECT A.Name FROM  
Employee A,  
Employee B  
WHERE A.Dept = B.Dept  
AND B.Name = 'Andy' ;
```

Employee	
Name	Dept
John	Marketing
Mary	Sales
Peter	Sales
Andy	Marketing
Anne	Marketing

- Find the names of all employees who work in the same department as Andy.

# Aliases and ‘Self-Joins’

Employee A

Name	Dept
John	Marketing
Mary	Sales
Peter	Sales
Andy	Marketing
Anne	Marketing

Employee B

Name	Dept
John	Marketing
Mary	Sales
Peter	Sales
Andy	Marketing
Anne	Marketing

# Aliases and ‘Self-Joins’

```
SELECT ... FROM Employee A, Employee B ...
```

A.Name	A.Dept	B.Name	B.Dept
John	Marketing	John	Marketing
Mary	Sales	John	Marketing
Peter	Sales	John	Marketing
Andy	Marketing	John	Marketing
Anne	Marketing	John	Marketing
John	Marketing	Mary	Sales
Mary	Sales	Mary	Sales
Peter	Sales	Mary	Sales
Andy	Marketing	Mary	Sales
Anne	Marketing	Mary	Sales

# Aliases and ‘Self-Joins’

```
SELECT ... FROM Employee A, Employee B WHERE A.Dept =  
        B.Dept ...
```

A.Name	A.Dept	B.Name	B.Dept
John	Marketing	John	Marketing
Andy	Marketing	John	Marketing
Anne	Marketing	John	Marketing
Mary	Sales	Mary	Sales
Peter	Sales	Mary	Sales
Mary	Sales	Peter	Sales
Peter	Sales	Peter	Sales
John	Marketing	Andy	Marketing
Andy	Marketing	Andy	Marketing
Anne	Marketing	Andy	Marketing

# Aliases and ‘Self-Joins’

```
SELECT . . . FROM Employee A, Employee B WHERE A.Dept =  
      B.Dept AND B.Name = 'Andy' ;
```

A.Name	A.Dept	B.Name	B.Dept
John	Marketing	Andy	Marketing
Andy	Marketing	Andy	Marketing
Anne	Marketing	Andy	Marketing

# Aliases and ‘Self-Joins’

```
SELECT A.Name FROM Employee A, Employee B  
WHERE A.Dept = B.Dept AND B.Name = 'Andy' ;
```

A.Name
John
Andy
Anne

- Names of all employees who work in the same department as Andy.

# Subqueries

- A SELECT statement can be nested inside another query to form a subquery
- The results of the subquery are passed back to the containing query
- For example, retrieve a list of names of people who are in Andy's department:

```
SELECT Name FROM Employee WHERE Dept =  
  (SELECT Dept FROM Employee  
    WHERE Name = 'Andy' )
```

# Subqueries

- First the subquery is evaluated, returning ‘Marketing’

```
SELECT Name  
FROM Employee  
WHERE Dept = 'Marketing';
```

- This value is passed to the main query

```
SELECT Name FROM Employee  
WHERE Dept =  
  (SELECT Dept  
   FROM Employee  
   WHERE Name = 'Andy' )
```

Employee	
Name	Dept
John	Marketing
Mary	Sales
Peter	Sales
Andy	Marketing
Anne	Marketing

# Subqueries

- Often a subquery will return a set of values rather than a single value
- We cannot directly compare a single value to a set. Doing so will result in an error
- Options for handling sets
  - **IN** : checks to see if a value is in a set
  - **EXISTS** : checks to see if a set is empty
  - **ALL/ANY** : checks to see if a relationship holds for every/one member of a set
  - **NOT** : can be used with any of the above 4

# Handling sets: IN

- Using IN we can see if a given value is in a set of values
- NOT IN checks to see if a given value is not in the set
- The set can be given explicitly or can be produced in a subquery

```
SELECT columns  
FROM tables  
WHERE col  
IN set;
```

```
SELECT columns  
FROM tables  
WHERE col  
NOT IN set;
```

Employee

Name	Department	Manager
John	Marketing	Chris
Mary	Marketing	Chris
Chris	Marketing	Jane
Peter	Sales	Jane
Jane	Management	

```
SELECT * FROM Employee  
WHERE Department IN ('Marketing', 'Sales');
```



Employee

Name	Department	Manager
John	Marketing	Chris
Mary	Marketing	Chris
Chris	Marketing	Jane
Peter	Sales	Jane

**Employee**

Name	Department	Manager
John	Marketing	Chris
Mary	Marketing	Chris
Chris	Marketing	Jane
Peter	Sales	Jane
Jane	Management	

```
SELECT * FROM Employee  
WHERE Department = 'Marketing'  
OR Department = 'Sales';
```



**Employee**

Name	Department	Manager
John	Marketing	Chris
Mary	Marketing	Chris
Chris	Marketing	Jane
Peter	Sales	Jane

# Handling sets: NOT IN

Employee		
Name	Department	Manager
John	Marketing	Chris
Mary	Marketing	Chris
Chris	Marketing	Jane
Peter	Sales	Jane
Jane	Management	

```
SELECT *  
FROM Employee  
WHERE Name NOT IN  
(SELECT Manager  
FROM Employee);
```

# Handling sets: NOT IN

- First the subquery
  - **SELECT Manager**  
**FROM Employee**
- The original query is then the same as:  

```
SELECT * FROM
Employee
WHERE Name NOT IN
( 'Chris' , 'Jane' );
```



Manager
Chris
Chris
Jane
Jane

Name	Department	Manager
John	Marketing	Chris
Mary	Marketing	Chris
Peter	Sales	Jane

# Handling sets: EXISTS

- Using EXISTS we can see whether there is at least one element in a given set
- NOT EXISTS is true if the set is empty
- The set is always given by a subquery

```
SELECT columns  
FROM tables  
WHERE EXISTS  
set;
```

```
SELECT columns  
FROM tables  
WHERE NOT  
EXISTS set;
```

# Handling sets: EXISTS

Retrieve all the info for those employees  
who are also managers:

```
SELECT * FROM
Employee AS E1
WHERE EXISTS (
    SELECT * FROM
    Employee AS E2
    WHERE E1.Name = E2.Manager) ;
```

Employee		
Name	Department	Manager
John	Marketing	Chris
Mary	Marketing	Chris
Chris	Marketing	Jane
Peter	Sales	Jane
Jane	Management	

# Handling sets: EXISTS

```
SELECT * FROM Employee AS E1 WHERE EXISTS (SELECT * FROM Employee AS E2 WHERE E1.Name = E2.Manager);
```

Employee E1

Name	Dept	Manager
John	Marketing	Chris
Mary	Marketing	Chris
Chris	Marketing	Jane
Peter	Sales	Jane
Jane	Management	

Employee E2

Name	Dept	Manager
John	Marketing	Chris
Mary	Marketing	Chris
Chris	Marketing	Jane
Peter	Sales	Jane
Jane	Management	

**Employee**

Name	Department	Manager
John	Marketing	Chris
Mary	Marketing	Chris
Chris	Marketing	Jane
Peter	Sales	Jane
Jane	Management	

```
SELECT * FROM Employee AS E1  
WHERE EXISTS (  
    SELECT * FROM Employee AS E2  
    WHERE E1.Name = E2.Manager);
```



Name	Dept	Manager
Chris	Marketing	Jane
Jane	Management	

# Handling sets: ANY and ALL

- ANY and ALL compare a single value to a set of values
- They are used with comparison operators like = , >, <, <>, >=, <=
- `val = ANY (set)`
  - is true if there is at least one member of the set equal to value
- `val = ALL (set)`
  - is true if all members of the set are equal to the value

# Handling sets: ALL

- Find the name(s) of the employee(s) who earn the highest salary

- Employee:

Name	Salary
Mary	20,000
John	15,000
Jane	25,000
Paul	30,000

# Handling sets: ALL

Name	Salary
Mary	20,000
John	15,000
Jane	25,000
Paul	30,000

Name
Paul

Find the name(s) of the employee(s) who earn the highest salary

```
SELECT Name  
FROM Employee  
WHERE Salary >=  
ALL (  
    SELECT Salary  
    FROM Employee);
```

# Handling sets: ANY

- Find the name(s) of the employee(s) who earn more than someone else

Name	Salary
Mary	20,000
John	15,000
Jane	25,000
Paul	30,000

# Handling sets: ANY

Name	Salary
Mary	20,000
John	15,000
Jane	25,000
Paul	30,000

Name
Mary
Jane
Paul

- Find the name(s) of the employee(s) who earn more than someone else

```
SELECT Name  
FROM Employee  
WHERE Salary >  
    ANY (  
        SELECT Salary  
FROM Employee);
```

# Word Search

- Word Search
  - Commonly used for searching product catalogues etc.
  - Need to search by keywords
  - Might need to use partial keywords
- For example: Given a database of books, searching for “crypt” might return
  - “Cryptonomicon” by Neil Stephenson
  - “Applied Cryptography” by Bruce Schneier

# LIKE

- We can use the LIKE keyword to perform string comparisons in queries
- Like is not the same as '=' because it allows wildcard characters
- It is NOT normally case sensitive

```
SELECT * FROM books  
WHERE bookName LIKE "%crypt%";
```

# LIKE

- The '%' character can represent any number of characters, including none
- The '\_' character represents exactly one character

bookName **LIKE** "crypt%"

- Will return “Cryptography Engineering” and “Cryptonomicon” but not “Applied Cryptography”

bookName **LIKE** "cloud\_"

- Will return “Clouds” but not “Cloud” or “cloud computing”

# LIKE

- Sometimes you might need to search for a set of words
  - To find entries with all words you can link conditions with AND
  - To find entries with any words use OR

```
SELECT * FROM
books
WHERE
bookName
LIKE "%crypt%"
OR bookName
LIKE "%cloud%";
```

# Example

Track

cdID	Num	Track_title	Time	aID
1	1	Violent	239	1
1	2	Every Girl	410	1
1	3	Breather	217	1
1	4	Part of Me	279	1
2	1	Star	362	1
2	2	Teaboy	417	2

CD

cdID	Title	Price
1	Mix	9.99
2	Compilation	12.99

Artist

aID	Name
1	Stellar
2	Cloudboy

Write a query to find any track title containing either the string ‘boy’ or ‘girl’

# Example

```
SELECT Track_title FROM Track  
WHERE  
    Track_title LIKE "%boy%"  
OR  
    Track_title LIKE "%girl%";
```

# Date, Time, Datetime

- 3 different types for a time column

Type	Description	Example
DATE	A Day, Month and Year	'1981-12-16' or '81-12-16'
TIME	Hours, Minutes and Seconds	'15:24:39'
DATETIME	Combination of above	'1981-12-16 15:24:39'

- Timestamp: as Datetime, usually used to display current date and time ('2014-11-04 15:30:43')
- Usual conditions may be used on WHERE clauses

**SELECT \* FROM** table-name

**WHERE** date-of-event < '2012-01-01' ;

- Or

**WHERE** date-of-event **LIKE** '2014-11-%' ;

Questions?

# SQL 2.3: SELECT

Jianjun Chen

# Contents

- Joins
  - Cross, Inner, Natural, Outer
- ORDER BY to produce ordered output
- Aggregate functions
  - MIN, MAX, SUM, AVG, COUNT
- GROUP BY
- HAVING
- UNION
- Missing Information

# Joins

- JOINS can be used to combine tables in a SELECT query
- **CROSS JOIN**: Returns all pairs of rows from A and B, the same as Cartesian Product
- **INNER JOIN**: Returns pairs of rows satisfying a condition
- **NATURAL JOIN**: Returns pairs of rows with common values in identically named columns
- **OUTER JOIN**: Returns pairs of rows satisfying a condition (as INNER JOIN), BUT ALSO handles NULLS

# CROSS JOIN

- Syntax:

```
SELECT * FROM A CROSS JOIN B;
```

- Same as:

```
SELECT * FROM A, B;
```

- Usually needs **WHERE** to filter out unrelated tuples

# INNER JOIN

- **INNER JOIN** specifies a condition that pairs of rows must satisfy.

```
SELECT * FROM A INNER JOIN B  
ON condition
```

- Can also use a **USING** clause that will output rows with equal values in the specified columns

```
SELECT * FROM A INNER JOIN B USING  
(col1, col2)
```

- **col1** and **col2** must appear in both A and B.

# INNER JOIN: Example

Buyer

Name	Budget
Smith	100,000
Jones	150,000
Green	80,000

Property

Address	Price
15 High Street	85,000
12 Queen Street	125,000
87 Oak Lane	175,000

```
SELECT * FROM  
Buyer INNER JOIN  
Property  
ON Price <= Budget
```



Name	Budget	Address	Price
Smith	100,000	15 High Street	85,000
Jones	150,000	15 High Street	85,000
Jones	150,000	12 Queen Street	125,000

# INNER JOIN: Practice

Student	
ID	Name
123	John
124	Mary
125	Mark
126	Jane

Enrolment	
ID	Code
123	DBS
124	PRG
124	DBS
126	PRG

**SELECT \* FROM**

Student **INNER JOIN** Enrolment  
**USING** (ID)



?

# INNER JOIN: Practice

Student	
ID	Name
123	John
124	Mary
125	Mark
126	Jane

Enrolment	
ID	Code
123	DBS
124	PRG
124	DBS
126	PRG

```
SELECT * FROM  
Student INNER JOIN Enrolment  
USING (ID)
```



ID	Name	Code
123	John	DBS
124	Mary	PRG
124	Mary	DBS
126	Jane	PRG

A single ID row will be output representing the equal values from both **Student.ID** and **Enrolment.ID**

# NATURAL JOIN

- **SELECT \* FROM A NATURAL JOIN B;**
- A **NATURAL JOIN** is effectively a special case of an **INNER JOIN** where the **USING** clause has specified all identically named columns.
- Same as the **A  $\bowtie$  B** in relational algebra.

# NATURAL JOIN

Student (S)

ID	Name
123	John
124	Mary
125	Mark
126	Jane

Enrolment (E)

ID	Code
123	DBS
124	PRG
124	DBS
126	PRG

**SELECT \* FROM**

Student **NATURAL JOIN** Enrolment;



ID	Name	Code
123	John	DBS
124	Mary	PRG
124	Mary	DBS
126	Jane	PRG

# JOINS vs WHERE Clauses

Inner/Natural JOINs are not absolutely necessary

- You can obtain the same results by selecting from multiple tables and using appropriate WHERE clauses

Should you use them?

- Yes
  - They often lead to concise and elegant queries
  - NATURAL JOINs are extremely common
- No
  - Support for JOINs can vary between DBMSs

# OUTER JOIN

```
SELECT cols  
FROM table1 type-OUTER-JOIN table2  
ON condition;
```

- Where **type** is one of **LEFT**, **RIGHT** or **FULL** .

# Example: Left Outer Join

**SELECT \* FROM**

Student **LEFT OUTER JOIN** Enrolment  
**ON** Student.ID = Enrolment.ID;

Student		Enrolment		
ID	Name	ID	Code	Mark
123	John	123	DBS	60
124	Mary	124	PRG	70
125	Mark	125	DBS	50
126	Jane	128	DBS	80

← Dangles

Student LEFT OUTER JOIN Enrolment ON ...

ID	Name	ID	Code	Mark
123	John	123	DBS	60
124	Mary	124	PRG	70
125	Mark	125	DBS	50
126	Jane	NULL	NULL	NULL

# Example: Right Outer Join

Student

ID	Name
123	John
124	Mary
125	Mark
126	Jane

Enrolment

ID	Code	Mark
123	DBS	60
124	PRG	70
125	DBS	50
128	DBS	80

← Dangles

Student RIGHT OUTER JOIN Enrolment ON ...

ID	Name	ID	Code	Mark
123	John	123	DBS	60
124	Mary	124	PRG	70
125	Mark	125	DBS	50
NULL	NULL	128	DBS	80

# Example: Full Outer Join

Student

ID	Name
123	John
124	Mary
125	Mark
126	Jane

Enrolment

ID	Code	Mark
123	DBS	60
124	PRG	70
125	DBS	50
128	DBS	80

← Dangles

Student FULL OUTER JOIN Enrolment ON ...

ID	Name	ID	Code	Mark
123	John	123	DBS	60
124	Mary	124	PRG	70
125	Mark	125	DBS	50
126	Jane	NULL	NULL	NULL
NULL	NULL	128	DBS	80

# Full Outer Join in MySQL

Only Left and Right outer joins are supported in MySQL.  
If you really want a FULL outer join:

```
SELECT *
  FROM Student FULL OUTER JOIN Enrolment
  ON Student.ID = Enrolment.ID;
```

Can be achieved using:

```
(SELECT * FROM Student LEFT OUTER JOIN
  Enrolment ON Student.ID = Enrolment.ID)
UNION
(SELECT * FROM Student RIGHT OUTER JOIN
  Enrolment ON Student.ID = Enrolment.ID);
```

# Why Using Outer Joins?

- Sometimes an outer join is the most practical approach. We may encounter NULL values, but may still wish to see the existing information.
- The next few slides will consider this problem:
- For students graduating in absentia, find a list of all student IDs, names, addresses, phone numbers and their final degree classifications.

For students graduating in absentia, find a list of all student IDs, names, addresses, phone numbers and their final degree classifications.

Student

ID	Name	aID	pID	Grad
123	John	12	22	C
124	Mary	23	90	A
125	Mark	19	NULL	A
126	Jane	14	17	C
127	Sam	NULL	101	A

Phone

pID	pNumber	pMobile
17	1111111	07856232411
22	2222222	07843223421
90	3333333	07155338654
101	4444444	07213559864

Degree

ID	Classification
123	1
124	2:1
125	2:2
126	2:1
127	3

Address

aID	aStreet	aTown	aPostcode
12	5 Arnold Close	Nottingham	NG12 1DD
14	17 Derby Road	Nottingham	NG7 4FG
19	1 Main Street	Derby	DE1 5FS
23	7 Holly Avenue	Nottingham	NG6 7AR

# Problems with INNER JOINS

- An Inner Join with **Student** and **Address** will ignore Student 127, who doesn't have an address record
- An Inner Join with **Student** and **Phone** will ignore student 125, who doesn't have a phone record

Student				
ID	Name	aID	pID	Grad
123	John	12	22	C
124	Mary	23	90	A
125	Mark	19	NULL	A
126	Jane	14	17	C
127	Sam	NULL	101	A

Phone		
pID	pNumber	pMobile
17	1111111	07856232411
22	2222222	07843223421
90	3333333	07155338654
101	4444444	07213559864

Address			
aID	aStreet	aTown	aPostcode
12	5 Arnold Close	Nottingham	NG12 1DD
14	17 Derby Road	Nottingham	NG7 4FG
19	1 Main Street	Derby	DE1 5FS
23	7 Holly Avenue	Nottingham	NG6 7AR

For students graduating in absentia, find a list of all student IDs, names, addresses, phone numbers and their final degree classifications. What is your solution?

Student

ID	Name	aID	pID	Grad
123	John	12	22	C
124	Mary	23	90	A
125	Mark	19	NULL	A
126	Jane	14	17	C
127	Sam	NULL	101	A

Phone

pID	pNumber	pMobile
17	1111111	07856232411
22	2222222	07843223421
90	3333333	07155338654
101	4444444	07213559864

Degree

ID	Classification
123	1
124	2:1
125	2:2
126	2:1
127	3

Address

aID	aStreet	aTown	aPostcode
12	5 Arnold Close	Nottingham	NG12 1DD
14	17 Derby Road	Nottingham	NG7 4FG
19	1 Main Street	Derby	DE1 5FS
23	7 Holly Avenue	Nottingham	NG6 7AR

```
SELECT ...
```

```
FROM Student LEFT OUTER JOIN Phone  
ON Student.pID = Phone.pID
```

```
...
```

Student

Phone

ID	Name	aID	pID	Grad	pID	pNumber	pMobile
123	John	12	22	C	22	2222222	07843223421
124	Mary	23	90	A	90	3333333	07155338654
125	Mark	19	NULL	A	NULL	NULL	NULL
126	Jane	14	17	C	17	1111111	07856232411
127	Sam	NULL	101	A	101	4444444	07213559864

Address

aID	aStreet	aTown	aPostcode
12	5 Arnold Close	Nottingham	NG12 1DD
14	17 Derby Road	Nottingham	NG7 4FG
19	1 Main Street	Derby	DE1 5FS
23	7 Holly Avenue	Nottingham	NG6 7AR

Next table to combine



```

SELECT ...
    FROM Student LEFT OUTER JOIN Phone
        ON Student.pID = Phone.pID
    LEFT OUTER JOIN Address
        ON Student.aID = Address.aID
    ...

```

Student			Phone				Address		
ID	Name	aID	pID	Grad	pNumber	pMobile	aStreet	aTown	aPostcode
123	John	12	22	C	2222222	07843223421	5 Arnold...	Notts	NG12 1DD
124	Mary	23	90	A	3333333	07155338654	7 Holly...	Notts	NG6 7AR
125	Mark	19	NULL	A	NULL	NULL	1 Main...	Derby	DE1 5FS
126	Jane	14	17	C	1111111	07856232411	17 Derby...	Notts	NG7 4FG
127	Sam	NULL	101	A	4444444	07213559864	NULL	NULL	NULL

32

Next table to combine →

Degree	
ID	Classification
123	1
124	2:1
125	2:2
126	2:1
127	3

# Solution Using OUTER JOIN

```
SELECT ID, Name, aStreet, aTown, aPostcode, pNumber,  
Classification FROM  
((Student LEFT OUTER JOIN Phone  
ON Student.pID = Phone.pID)  
LEFT OUTER JOIN Address  
ON Student.aID = Address.aID)  
INNER JOIN Degree  
ON Student.ID = Degree.ID  
WHERE Grad = 'A' ;
```

Student				Phone			Address		
ID	Name	aID	pID	Grad	pNumber	pMobile	aStreet	aTown	aPostcode
123	John	12	22	C	2222222	07843223421	5 Arnold...	Notts	NG12 1DD
124	Mary	23	90	A	3333333	07155338654	7 Holly...	Notts	NG6 7AR
125	Mark	19	NULL	A	NULL	NULL	1 Main...	Derby	DE1 5FS
126	Jane	14	17	C	1111111	07856232411	17 Derby...	Notts	NG7 4FG
127	Sam	NULL	101	A	4444444	07213559864	NULL	NULL	NULL

Degree	
ID	Classification
123	1
124	2:1
125	2:2
126	2:1
127	3

# Solution Using OUTER JOIN

Student **LEFT OUTER JOIN** Phone  
ON Student.pID = Phone.pID

Address

**LEFT OUTER JOIN**

ON Student.aID = Address.aID

Student			Phone			Address			
ID	Name	aID	pID	Grad	pNumber	pMobile	aStreet	aTown	aPostcode
123	John	12	22	C	2222222	07843223421	5 Arnold...	Notts	NG12 1DD
124	Mary	23	90	A	3333333	07155338654	7 Holly...	Notts	NG6 7AR
125	Mark	19	NULL	A	NULL	NULL	1 Main...	Derby	DE1 5FS
126	Jane	14	17	C	1111111	07856232411	17 Derby...	Notts	NG7 4FG
127	Sam	NULL	101	A	4444444	07213559864	NULL	NULL	NULL

Degree

ID	Classification
123	1
124	2:1
125	2:2
126	2:1
127	3

**INNER JOIN ON** Student.ID = Degree.ID  
**WHERE** Grad = 'A' ;

# Final Result Using OUTER JOIN

ID	Name	aStreet	aTown	aPostcode	pNumber	Classification
124	Mary	7 Holly Avenue	Nottingham	NG6 7AR	3333333	2:1
125	Mark	1 Main Street	Derby	DE1 5FS	NULL	2:2
127	Sam	NULL	NULL	NULL	4444444	3

The records for students 125 and 127 have been preserved despite missing information

# SQL SELECT Overview

**SELECT**

[**DISTINCT** | **ALL**] column-list

**FROM** table-names

[**WHERE** condition]

[**GROUP BY** column-list]

[**HAVING** condition]

[**ORDER BY** column-list]

([] optional, | or)

# **ORDER BY**

- The **ORDER BY** clause sorts the results of a query
  - You can sort in ascending (default) or descending order
  - Multiple columns can be given
  - You cannot order by a column which isn't in the result  
(Really so? Check Notes under this slide)

**SELECT** columns **FROM** tables  
**WHERE** condition  
**ORDER BY** cols [**ASC** | **DESC**]

# ORDER BY: Example 1

Grades		
Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60
James	PR1	43
James	PR2	35
Jane	IAI	54

```
SELECT * FROM Grades  
ORDER BY Mark;
```



Name	Code	Mark
James	PR2	35
James	PR1	43
Jane	IAI	54
John	DBS	56
Mary	DBS	60
John	IAI	72

# ORDER BY: Example 2

Grades		
Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60
James	PR1	43
James	PR2	35
Jane	IAI	54

```
SELECT * FROM Grades  
ORDER BY  
Code ASC, Mark DESC;
```



Name	Code	Mark
Mary	DBS	60
John	DBS	56
John	IAI	72
Jane	IAI	54
James	PR1	43
James	PR2	35

# Arithmetic

- As well as columns, a SELECT statement can also be used to
  - Compute arithmetic expressions
  - Evaluate functions
- Often helpful to use an alias when dealing with expressions or functions.

```
SELECT Mark / 100 FROM Grades;
```

```
SELECT Salary + Bonus FROM Employee;
```

```
SELECT 1.20 * Price  
AS 'Price inc. VAT'  
FROM Products;
```

# Aggregate Functions

- Aggregate functions compute summaries of data in a table.
  - Most aggregate functions (except COUNT (\*)) work on a single column of numerical data
- Aggregate functions:
  - **COUNT**: The number of rows
  - **SUM**: The sum of the entries in the column
  - **AVG**: The average entry in a column
  - **MIN, MAX**: The minimum/maximum entries in a column
- Again, it's best to use an alias to name the result

# COUNT

Grades		
Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60
James	PR1	43
James	PR2	35
Jane	IAI	54

```
SELECT  
    COUNT(*) AS Count  
FROM Grades;
```

```
SELECT  
    COUNT(Code) AS Count  
FROM Grades;
```

```
SELECT  
    COUNT(DISTINCT Code)  
        AS Count  
FROM Grades;
```

# COUNT

Grades		
Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60
James	PR1	43
James	PR2	35
Jane	IAI	54

**SELECT**

```
COUNT (*) AS Count  
FROM Grades;
```



Count
6

**SELECT**

```
COUNT (Code) AS Count  
FROM Grades;
```



Count
6

**SELECT**

```
COUNT (DISTINCT Code)  
AS Count  
FROM Grades;
```



Count
4

# SUM, MIN/MAX and AVG

Grades		
Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60
James	PR1	43
James	PR2	35
Jane	IAI	54

```
SELECT  
SUM(Mark) AS Total  
FROM Grades;
```

```
SELECT  
MAX(Mark) AS Best  
FROM Grades;
```

```
SELECT  
AVG(Mark) AS Mean  
FROM Grades;
```

# SUM, MIN/MAX and AVG

**SELECT**

**SUM**(Mark) **AS** Total



Total
320

**FROM** Grades;

Grades		
Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60
James	PR1	43
James	PR2	35
Jane	IAI	54

**SELECT**

**MAX**(Mark) **AS** Best



Best
72

**FROM** Grades;

**SELECT**

**AVG**(Mark) **AS** Mean



Mean
53.33

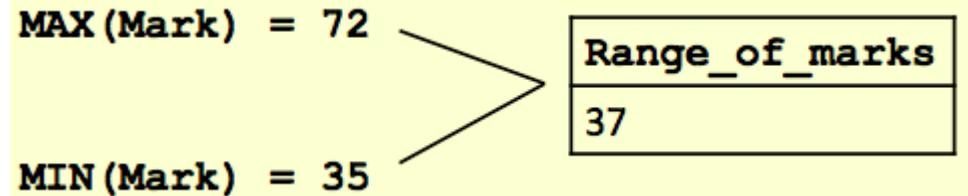
**FROM** Grades;

# Combining Aggregate Functions

- You can combine aggregate functions using arithmetic

Grades		
Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60
James	PR1	43
James	PR2	35
Jane	IAI	54

```
SELECT  
MAX(Mark) - MIN(Mark)  
AS Range_of_marks  
FROM Grades;
```



# Combining AF: Example

Modules		
Code	Title	Credits
DBS	Database Systems	10
IAI	Introduction to AI	20
PRG	Programming	10

Grades		
Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60

- Find John's average mark, weighted by the credits of each module

# Combining AF: Example

Modules		
Code	Title	Credits
DBS	Database Systems	10
IAI	Introduction to AI	20
PRG	Programming	10

Grades		
Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60

- Find John's average mark, weighted by the credits of each module

```
SELECT SUM(Mark*Credits) / SUM (Credits)  
      AS 'Final Mark'  
FROM Modules, Grades  
WHERE Modules.Code = Grades.Code  
      AND Grades.Name = 'John' ;
```

# GROUP BY

- Sometimes we want to apply aggregate functions to groups of rows
  - Example: find the average mark of each student individually
  - The GROUP BY clause achieves this.

```
SELECT column_set1 FROM tables  
WHERE predicate  
GROUP BY column_set2;
```

- Every entry in 'column\_set2' should be in 'column\_set1', be a constant, or be an aggregate function

# GROUP BY: Example

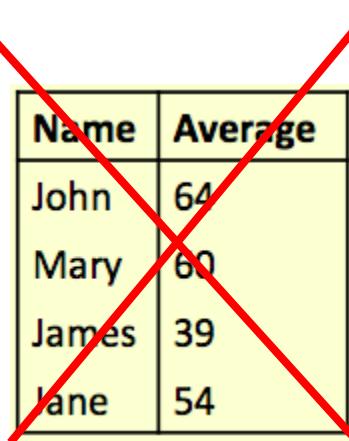
Grades		
Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60
James	PR1	43
James	PR2	35
Jane	IAI	54

```
SELECT Name,  
       AVG(Mark) AS Average  
FROM Grades  
GROUP BY Name;
```

# GROUP BY: Example

Grades		
Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60
James	PR1	43
James	PR2	35
Jane	IAI	54

```
SELECT Name,  
       AVG(Mark) AS Average  
FROM Grades  
GROUP BY Name;
```



Name	Average
John	64
Mary	60
James	39
Jane	54



Name	Average
James	39.0000
Jane	54.0000
John	64.0000
Mary	60.0000

# GROUP BY: Example

Sales		
Month	Department	Value
March	Fiction	20
March	Travel	30
March	Technical	40
April	Fiction	10
April	Fiction	30
April	Travel	25
April	Fiction	20
May	Fiction	20
May	Travel	50

- Find the total value of the sales for each department in each month
- Can group by Month then Department or Department then Month

# GROUP BY: Example

```
SELECT Month, Department,  
      SUM(Value) AS Total  
FROM Sales  
GROUP BY Month, Department;
```

Month	Department	Total
April	Fiction	60
April	Travel	25
March	Fiction	20
March	Technical	40
March	Travel	30
May	Fiction	20
May	Technical	50

```
SELECT Month, Department,  
      SUM(Value) AS Total  
FROM Sales  
GROUP BY Department, Month;
```

Month	Department	Total
April	Fiction	60
March	Fiction	20
May	Fiction	20
March	Technical	40
May	Technical	50
April	Travel	25
March	Travel	30

Same results, but produced in a different order

# HAVING

- HAVING is like a WHERE clause, except that it only applies to the results of a GROUP BY query
- It can be used to select groups which satisfy a given condition

Grades		
Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60
James	PR1	43
James	PR2	35
Jane	IAI	54

```
SELECT Name,  
       AVG(Mark) AS Average  
  FROM Grades  
 GROUP BY Name  
 HAVING  
       AVG(Mark) >= 40;
```



Name	Average
John	64
Mary	60
Jane	54

# WHERE and HAVING

- **WHERE** refers to the rows of tables, so cannot make use of aggregate functions.
- **HAVING** refers to the groups of rows, and so cannot use columns which are not in the GROUP BY or an aggregate function.
- Think of a query being processed as follows:
  1. Tables are joined
  2. WHERE clauses
  3. GROUP BY clauses and aggregates
  4. Column selection
  5. HAVING clauses
  6. ORDER BY

# SQL SELECT Overview

**SELECT**

[**DISTINCT** | **ALL**] column-list

**FROM** table-names

[**WHERE** condition]

[**GROUP BY** column-list]

[**HAVING** condition]

[**ORDER BY** column-list]

([] optional, | or)

# SET operations

- UNION, INTERSECT and EXCEPT
  - These treat the tables as sets and are the usual set operators of union, intersection and difference
  - We'll be concentrating on UNION
- They all combine the results from two select statements
- The results of the two selects should have the same columns and corresponding data types

# UNION: Example

- Find, in a single query, the average mark for each student and the average mark overall.

Grades		
Name	Code	Mark
Jane	IAI	54
John	DBS	56
John	IAI	72
James	PR1	43
James	PR2	35
Mary	DBS	60

# UNION

1. The average for each student:

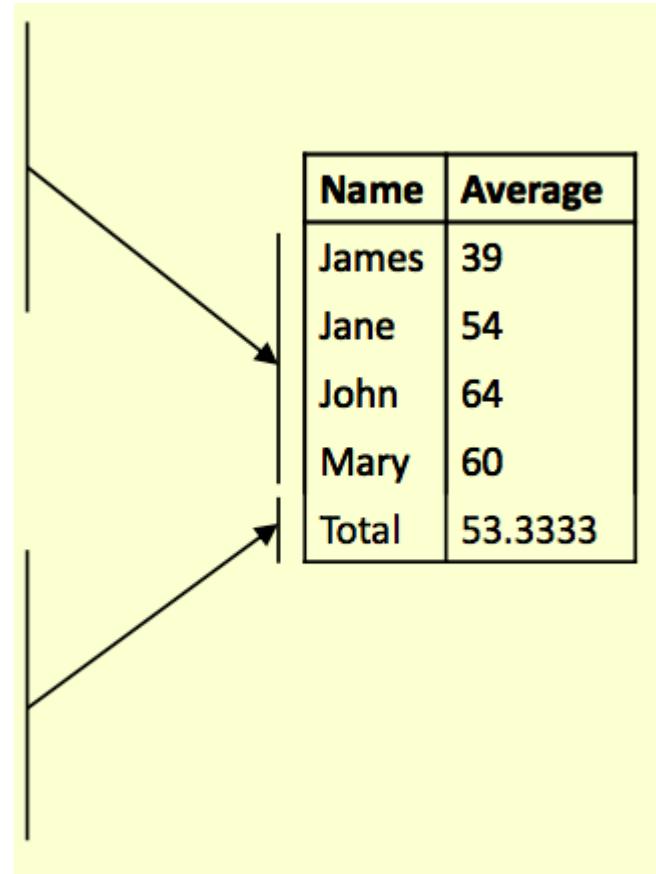
```
SELECT Name, AVG(Mark) AS Average  
FROM Grades  
GROUP BY Name;
```

2. The average overall:

```
SELECT 'Total' AS Name,  
      AVG(Mark) AS Average  
FROM Grades;
```

# UNION

```
SELECT Name,  
      AVG(Mark) AS Average  
FROM Grades  
GROUP BY Name  
UNION  
SELECT  
      'Total' AS Name,  
      AVG(Mark) AS Average  
FROM Grades;
```



The diagram illustrates the execution of a UNION query. Two arrows point from the 'UNION' keyword in the first part of the query to a table containing student names and their average marks. One arrow points to the first row (James), and the other points to the last row (Total). The table has columns 'Name' and 'Average'.

Name	Average
James	39
Jane	54
John	64
Mary	60
Total	53.3333

That's Everything for SELECT :-)

# Missing Information

# Missing Information

- Sometimes we don't know what value an entry in a relation should have
  - We know that there is a value, but don't know what it is
  - There is no value at all that makes any sense
- Two main methods have been proposed to deal with this
  - NULLs can be used as markers to show that information is missing
  - A default value can be used to represent the missing value

# Null

- Represents a state for an attribute that is currently unknown or is not applicable for this tuple.
  - Nulls are a way to deal with incomplete or exceptional data.
  - NULL is a placeholder for missing or unknown value of an attribute. **It is not itself a value.**
- E.g. A new staff is just added, but hasn't been decided which branch he belongs to.

# NULLs

Codd proposed to distinguish two types of NULLs:

- A-marks: data Applicable but not known (for example, someone's age)
- I-marks: data is Inapplicable (telephone number for someone who does not have a telephone, or spouse's name for someone who is not married)

# Problems with NULLs

- Problems extending relational algebra operations to NULLs:
  - Selection operation: if we check tuples for “Mark > 40” and for some tuple Mark is NULL, do we include it?
  - Comparing tuples in two relations: are two tuples and the same or not?
- Additional problems for SQL:
  - NULLs treated as duplicates?
  - Inclusion of NULLs in count, sum, average?
    - If yes, how?
  - Arithmetic operations behaviour with argument NULL?

# Theoretical Solutions

- Use three-valued logic instead of classical two valued logic to evaluate conditions.
- When there are no NULLs around, conditions evaluate to **true** or **false**, but if a null is involved, a condition might evaluate to the third value ('undefined', or '**unknown**')

# 3-valued logic

- If the condition involves a Boolean combination, we evaluate it as follows:

a	b	a OR b	a AND b	a == b
True	True	True	True	True
True	False	True	False	False
True	Unknown	True	Unknown	Unknown
False	True	True	False	False
False	False	False	False	True
False	Unknown	Unknown	False	Unknown
Unknown	True	True	Unknown	Unknown
Unknown	False	Unknown	False	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown

# SQL NULLs in Conditions

Employee	
Name	Salary
John	25,000
Mark	15,000
Anne	20,000
Chris	NULL

```
SELECT *  
FROM Employee  
Where  
Salary > 15,000;
```



Name	Salary
John	25,000
Anne	20,000

WHERE clause of SQL SELECT uses three-valued logic: only tuples where the condition evaluates to true are returned.

**Salary > 15,000** evaluates to ‘unknown’ on the last tuple – not included

# SQL NULLs in Conditions

Employee	
Name	Salary
John	25,000
Mark	15,000
Anne	20,000
Chris	NULL

**SELECT \* FROM Employee**

**Where**

**Salary > 15,000**

**OR**

**Name = 'Chris' ;**



Name	Salary
John	25,000
Anne	20,000
Chris	NULL

**Salary > 15,000 OR Name = 'Chris'** Is essentially **Unknown OR TRUE** on the last tuple

a	b	a OR b
Unknown	True	True

# SQL NULLs in Arithmetic

Employee	
Name	Salary
John	25,000
Mark	15,000
Anne	20,000
Chris	NULL

```
SELECT Name,  
       Salary * 0.05 AS Bonus  
FROM Employee;
```

Name	Bonus
John	1,250
Mark	750
Anne	1,000
Chris	NULL

Arithmetic operations applied to NULLs result in NULLS

# SQL NULLs in Aggregation

**SELECT**

```
AVG(Salary) AS Average,  
COUNT(Salary) AS Count,  
SUM(Salary) AS Sum  
FROM Employee;
```

Employee	
Name	Salary
John	25,000
Mark	15,000
Anne	20,000
Chris	NULL

Average = 20,000

Count = 3

Sum = 60,000

Using COUNT(\*) would give 4, even if the name of Chris is changed to NULL.

# SQL NULLs in GROUP BY

Employee	
Name	Salary
John	25,000
Mark	15,000
Anne	20,000
Jack	NULL
Sam	20,000
Chris	NULL

```
SELECT Salary,  
COUNT(Name) AS Count  
FROM Employee  
GROUP BY Salary;
```



Salary	Count
NULL	2
15,000	1
20,000	2
25,000	1

NULLs are treated as  
equivalents in GROUP BY  
clauses

# SQL NULLs in ORDER BY

Employee	
Name	Salary
John	25,000
Mark	15,000
Anne	20,000
Jack	NULL
Sam	20,000
Chris	NULL

```
SELECT *
FROM Employee
ORDER BY Salary;
```



Employee	
Name	Salary
Chris	NULL
Jack	NULL
Mark	15,000
Anne	20,000
Sam	20,000
John	25,000

NULLs are considered and reported in ORDER BY clauses

# Dealing with Missing Information

- Sometimes we don't know what value an entry in a relation should have
  - We know that there is a value, but don't know what it is
  - There is no value at all that makes any sense
- Two main methods have been proposed to deal with this
  - NULLs can be used as markers to show that information is missing
  - A default value can be used to represent the missing value

# Default Values

- Default values are an alternative to the use of NULLs
  - You can choose a value that makes no sense in normal circumstances:
    - age **INT DEFAULT -1,**
    - These are actual values
- Default values can have more meaning than NULLs
  - ‘none’
  - ‘unknown’
  - ‘not supplied’
  - ‘not applicable’
- Not all defaults represent missing information. It depends on the situation

# Default Values Example

- Default values are
  - “Unknown” for Name
  - -1 for Weight and Quantity
- -1 is used for Wgt and Qty as it is not sensible otherwise
- There are still problems:

**UPDATE** Parts **SET**  
Quantity = Quantity + 5

Parts			
ID	Name	Weight	Quantity
1	Nut	10	20
2	Bolt	15	-1
3	Nail	3	100
4	Pin	-1	30
5	Unknown	20	20
6	Screw	-1	-1
7	Brace	150	0

# SQL Support

- SQL allows both NULLs and defaults:
  - A table to hold data on employees
  - All employees have a name
  - All employees have a salary (default 10000)
  - Some employees have phone numbers, if not we use NULLs

```
CREATE TABLE Employee
(
    Name VARCHAR(50)
        NOT NULL,
    Salary INT
        DEFAULT 10000
        NOT NULL,
    Phone VARCHAR(15)
        NULL
);
```

# SQL Support

SQL allows you to insert  
NULLs

```
INSERT INTO Employee
VALUES ('John',
        12000, NULL);
```

```
UPDATE Employee
SET Phone = NULL
WHERE Name =
'Mark';
```

You can also check for NULLs

```
SELECT Name FROM
Employee WHERE
Phone IS NULL;
```

```
SELECT Name FROM
Employee WHERE
Phone IS NOT
NULL;
```

# SQL: The Final Example

# The Final Example

- Examiners' reports
  - We want a list of students and their average mark
  - For first and second years the average is for that year
  - For finalists (third year) it is 40% of the second year plus 60% of the final year averages
- We want the results:
  - Sorted by year (desc), then by average mark (high to low) then by last name, first name and finally ID
  - To take into account of the number of credits each module is worth
  - Produced by a single query

Student			
ID	First	Last	Year

Grade			
ID	Code	Mark	YearTaken

Module		
Code	Title	Credits

# Example Output

Student			
ID	First	Last	Year
Grade			
ID	Code	Mark	YearTaken
Module			
Code	Title	Credits	



Year	Student.ID	Last	First	AverageMark
3	11014456	Andrews	John	81
3	11013891	Smith	Mary	78
3	11014012	Brown	Amy	76
3	11013204	Jones	Steven	76
3	11014919	Robinson	Paul	74
2	11027871	Edwards	Robert	73
1	11024298	Green	Matthew	45
1	11024826	Hall	David	43
1	11027621	Wood	James	40
1	11024978	Clarke	Stewart	39
1	11026563	Wilson	Sarah	36
1	11027625	Taylor	Matthew	34
1		Williams	Paul	31

# Getting Started

- Finalists should be treated differently to other years
  - Write one SELECT for the finalists
  - Write a second SELECT for the first and second years
  - Merge the results using a UNION

**QUERY FOR FINALISTS**

**UNION**

**QUERY FOR OTHERS**

# Table Joins

- Both subqueries need information from all the tables
  - The student ID, name and year
  - The marks for each module and the year taken
  - The number of credits for each module
- This is a natural join operation
  - But because we're practicing, we're going to use a standard CROSS JOIN and WHERE clause
  - Exercise: repeat the query using natural join

# The Query so Far

```
SELECT some-information  
FROM Student, Module, Grade  
WHERE Student.ID = Grade.ID  
AND Module.Code = Grade.Code  
AND student-is-in-third-year
```

**UNION**

```
SELECT some-information  
FROM Student, Module, Grade  
WHERE Student.ID = Grade.ID  
AND Module.Code = Grade.Code  
AND student-is-in-first-or-second-  
year;
```

# Information for Finalists

- We must retrieve
  - Computed average mark, weighted 40-60 across years 2 and 3
  - First year marks must be ignored
  - The ID, Name and Year are needed as they are used for ordering
- The average is difficult
  - We don't have any statements to separate years 2 and 3 easily
  - We can exploit the fact that  $40 = 20 * 2$  and  $60 = 20 * 3$ , so YearTaken and the weighting have the same relationship

# The Query so Far

```
SELECT some-information
      FROM Student, Module, Grade
     WHERE Student.ID = Grade.ID
       AND Module.Code = Grade.Code
       AND student-is-in-third-year
UNION
    . . .
```

# Information for Finalists

```
SELECT Year, Student.ID, Last, First,  
    SUM((20*YearTaken)/100)*Mark*Credits)/120  
        AS AverageMark  
FROM  
    Student, Module, Grade  
WHERE  
    Student.ID = Grade.ID  
    AND  
    Module.Code = Grade.Code  
    AND  
    YearTaken IN (2,3)  
    AND  
    Year = 3  
GROUP BY  
    Year, Student.ID, First, Last
```

# Information for Others

- Other students are easier than finalists
  - We just need their average marks where YearTaken and Year are the same
  - As before, we need ID, Name and Year for ordering

...

**UNION**

**SELECT** some-information

**FROM** Student, Module, Grade

**WHERE** Student.ID = Grade.ID

**AND** Module.Code = Grade.Code

**AND** student-is-in-first-or-second-year;

# Information for Finalists

```
SELECT Year, Student.ID, Last, First,  
        SUM(Mark*Credits)/120 AS AverageMark  
FROM  
        Student, Module, Grade  
WHERE  
        Student.ID = Grade.ID  
        AND  
        Module.Code = Grade.Code  
        AND  
        YearTaken = Year  
        AND  
        Year IN (1,2)  
GROUP BY  
        Year, Student.ID, First, Last
```

```
SELECT Year, Student.ID, Last, First,  
       SUM((20*YearTaken)/100)*Mark*Credits)/120  
             AS AverageMark  
FROM Student, Module, Grade  
WHERE Student.ID = Grade.ID  
      AND Module.Code = Grade.Code  
      AND YearTaken IN (2,3)  
      AND Year = 3  
GROUP BY Year, Student.ID, Last, First
```

**UNION**

```
SELECT Year, Student.ID, Last, First,  
       SUM(Mark*Credits)/120 AS AverageMark  
FROM Student, Module, Grade  
WHERE Student.ID = Grade.ID  
      AND Module.Code = Grade.Code  
      AND YearTaken = Year  
      AND Year IN (1,2)  
GROUP BY Year, Student.ID, Last, First
```

**ORDER BY**

Year **desc**, AverageMark **desc**, Last, First, ID;



Questions?

# Entity-Relationship Diagrams

Jianjun Chen

# Database Design

- This lecture introduces the technique to design a database from a piece of written requirements.
- Need to consider
  - What is the database going to be used for?
  - What tables, attributes, keys are needed?
- Designing your database is important
  - Often results in a more efficient and simpler queries once the database has been created.
  - May help reduce data redundancy in the tables.

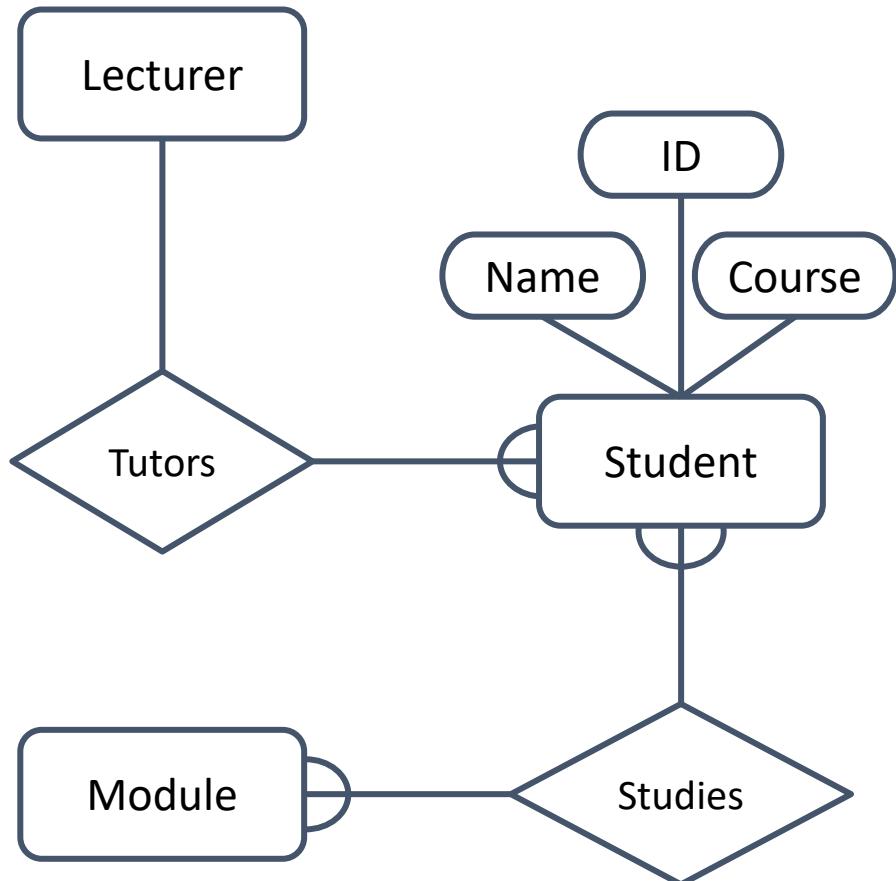
# Entity-Relationship Modelling

- E/R Modelling is used for conceptual design
  - Entities: objects or items of interest.
  - Attributes: properties of an entity.
  - Relationships: links between entities.
- For example, in a University database we might have entities for Students, Modules and Lecturers
  - Students might have attributes such as their ID, Name, and Course
  - Students could have relationships with Modules (enrolment) and Lecturers (tutor/tutee)

# Entity-Relationship Diagrams

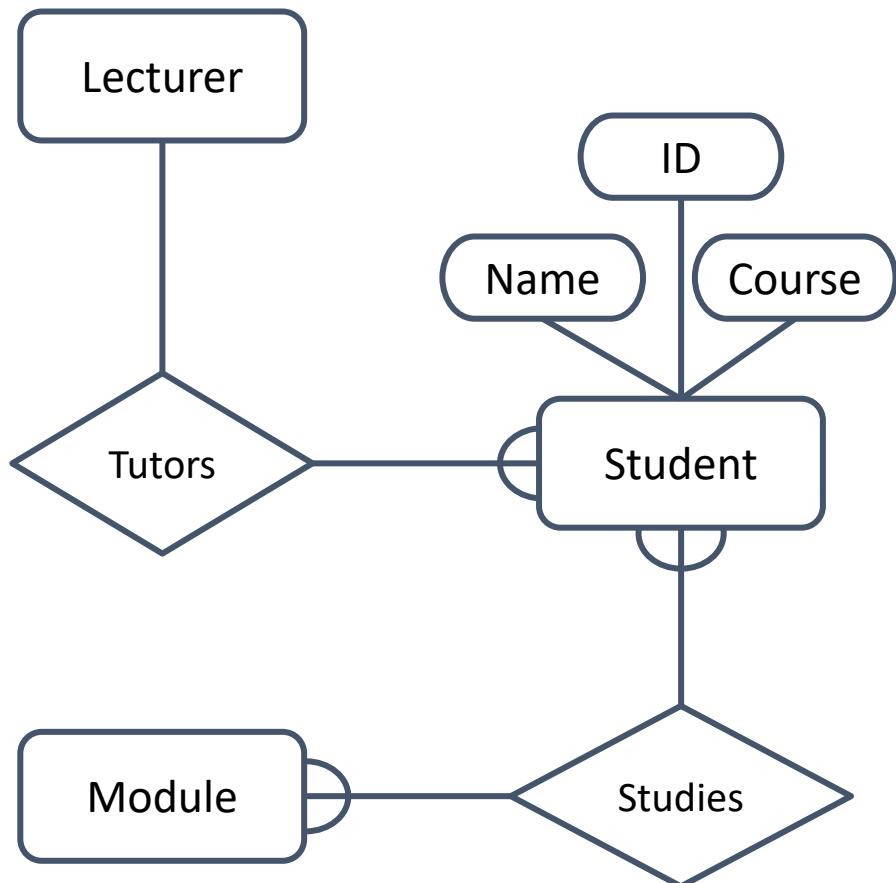
E/R Models are often represented as E/R diagrams that

- Give a conceptual view of the database
- Are independent of the choice of DBMS
- Can identify some problems in a design



# E/R: Diagram Conventions

- There are various notations for representing E/R diagrams
- These specify the shape of the various components, and the notation used to represent relationships
- For this introductory module, we will use simplified notation

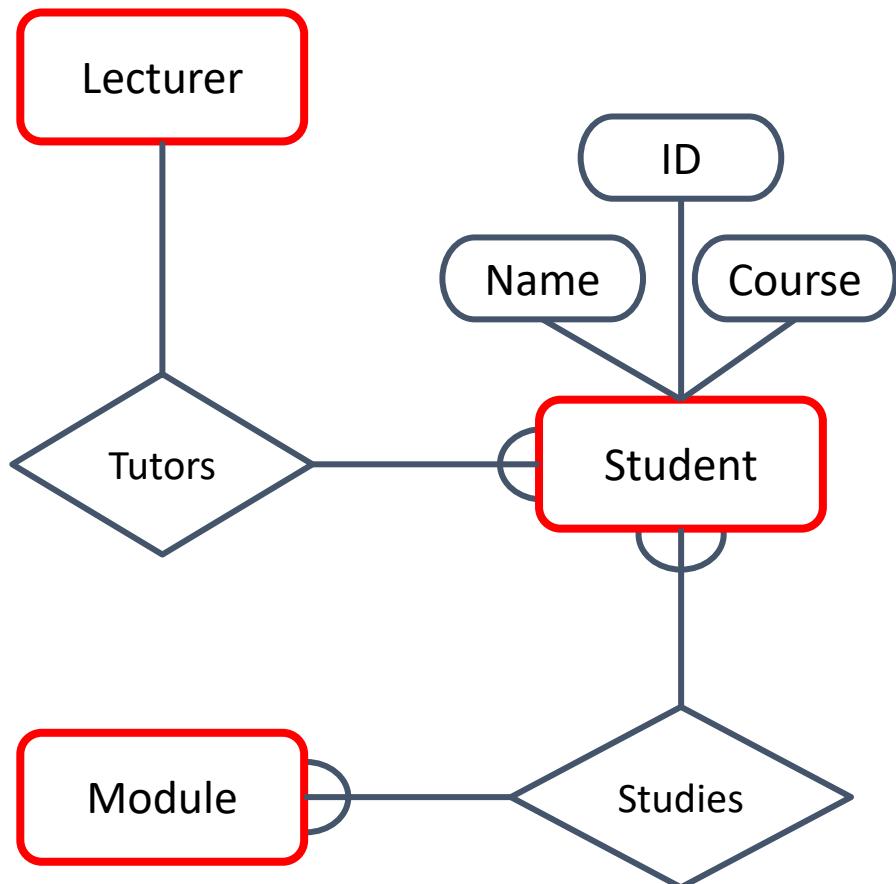


# Component 1: Entities

- Entities represent objects or things of interest
  - Physical things like students, lecturers, employees, products
  - More abstract things like modules, orders, courses, projects
- Entity:
  - Is a general type or class, such as Lecturer or Module
  - Has instances of that particular type. E.g. DBI and IAI are instances of Module
  - Has attributes (such as name, email address)

# E/R Diagram: Entities

- In E/R Diagrams, we will represent Entities as boxes with rounded corners
- The box is labelled with the name of the class of objects represented by that entity

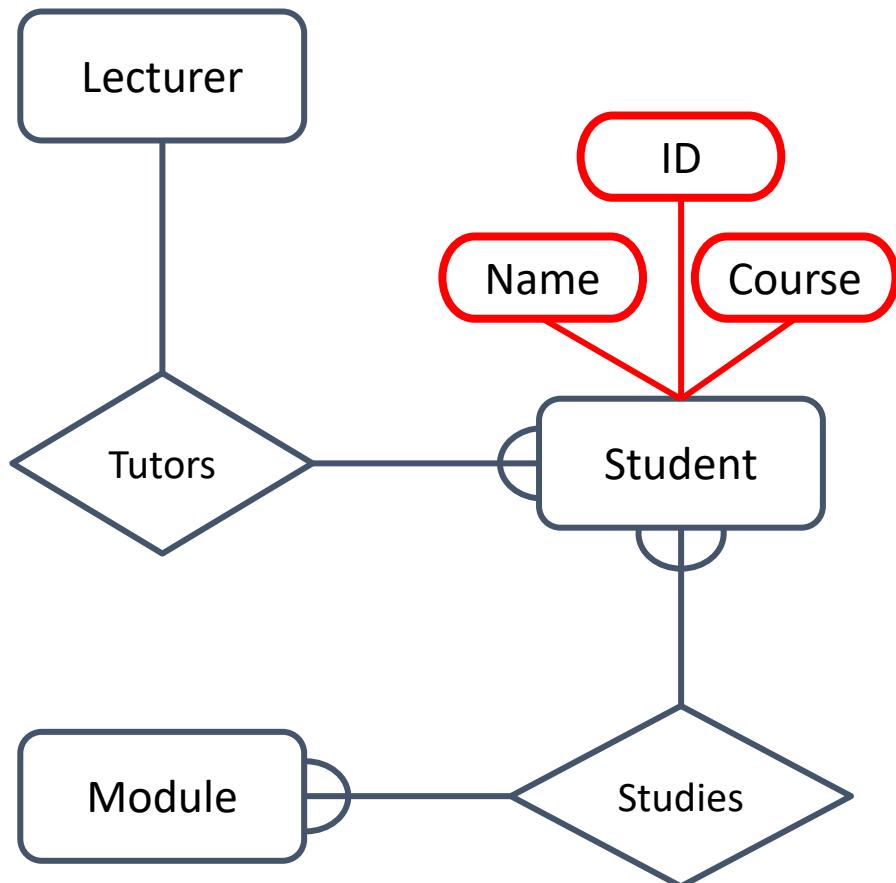


# Component 2: Attributes

- Attributes are facts, aspects, properties, or details about an entity
  - Students have IDs, names, courses, addresses, ...
  - Modules have codes, titles, credit weights, levels, ...
- Attributes have:
  - A name
  - An associated entity
  - Domains of possible values
  - For each instance of the associated entity, a value from the attributes domain

# E/R Diagram: Attributes

- In an E/R Diagram attributes are drawn as ovals
- Each attribute is linked to its entity by a line
- The name of the attribute is written in the oval



# Component 3: Relationships

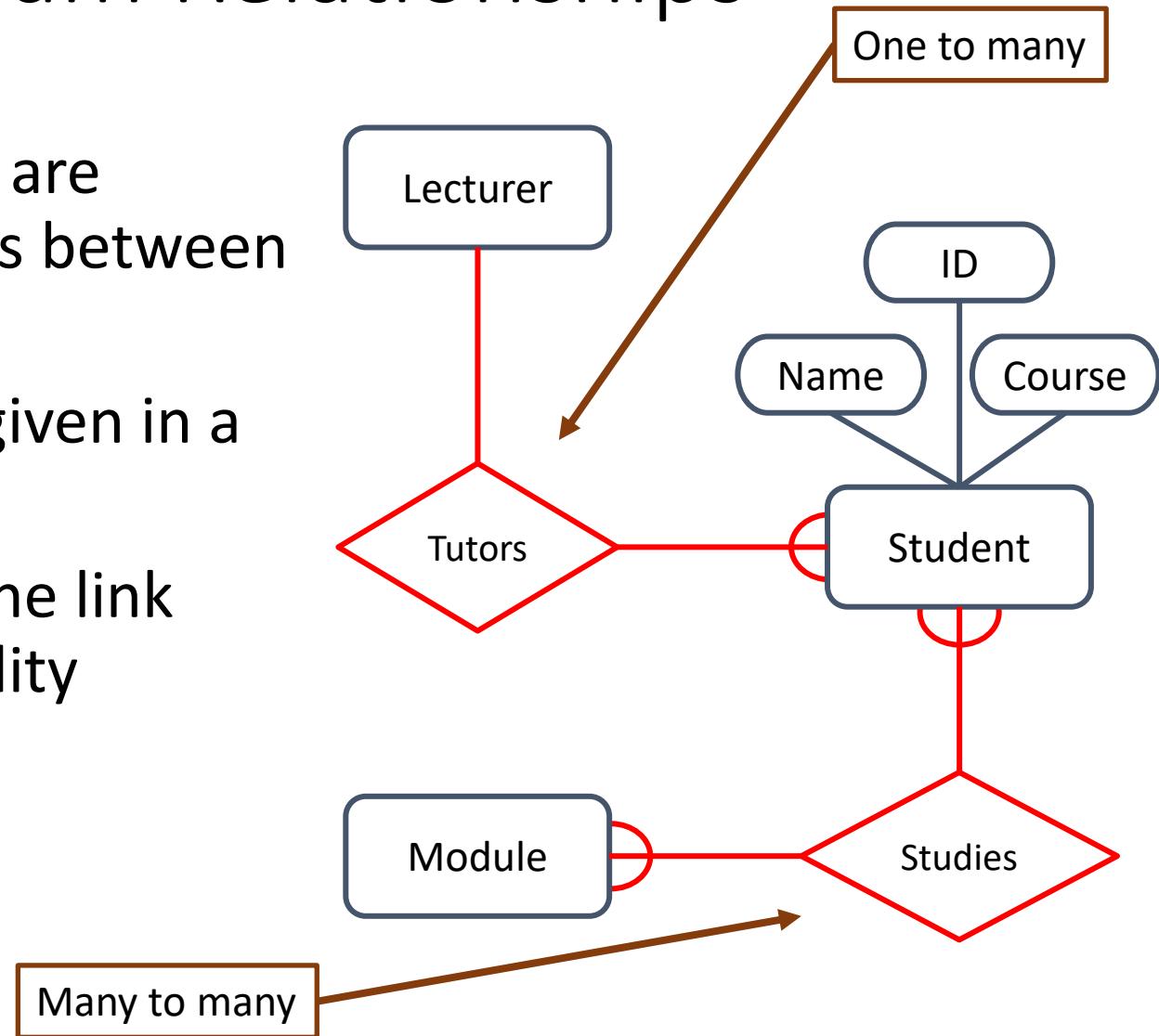
- A relationship is an association between two or more entities
  - Each Student takes several Modules.
  - Each Module is taught by a Lecturer.
  - Each Employee works for a single Department.
- Relationships have
  - A name.
  - A set of entities that participate in them .
  - A degree: the number of entities that participate (most have degree 2).
  - A cardinality ratio.

# Cardinality Ratios

- One to one (1:1)
  - Each lecturer has a unique office & offices are single occupancy
- One to many (1:M)
  - A lecturer may tutor many students, but each student has just one tutor
- Many to many (M:M)
  - Each student takes several modules, and each module is taken by several students

# E/R: Diagram Relationships

- Relationships are shown as links between two entities
- The name is given in a diamond box
- The ends of the link show cardinality



# Making E/R Models

- To make an E/R model you need to identify:
  - Entities
  - Attributes
  - Relationships
  - Cardinality ratios
- We obtain these from a problem description
- General guidelines
  - Since entities are things or objects they are often nouns in the description
  - Attributes are facts or properties, and so are often nouns also
  - Verbs often describe relationships between entities

# Example

A university consists of a number of departments. Each department offers several courses. A number of modules make up each course. Students enroll in a particular course and take modules towards the completion of that course. Each module is taught by a lecturer from the appropriate department (several lecturers work in the same department), and each lecturer tutors a group of students. A lecturer can teach more than one module but can work only in one department.

Entity, Attributes, Relationships: What shall be identified first?  
Followed by what?....

# Example - Entities

A university consists of a number of departments. Each **department** offers several **courses**. A number of **modules** make up each course. **Students** enroll in a particular course and take modules towards the completion of that course. Each module is taught by a **lecturer** from the appropriate department (several lecturers work in the same department), and each lecturer tutors a group of students. A lecturer can teach more than one module but can work only in one department.

**Entities** – Department, Course, Module, Student, Lecturer

# Example - Relationships

A university consists of a number of departments. Each **department offers** several **courses**. A number of **modules make up** each course. **Students enroll** in a particular course and **take** modules towards the completion of that course. Each module is **taught by** a **lecturer** from the appropriate department (several lecturers **work in** the same department), and each lecturer **tutors** a group of students. A lecturer can teach more than one module but can work only in one department.

**Entities** – Department, Course, Module, Student, Lecturer

**Relationships** – Offers, Make Up, Enroll, Take, Taught By, Work in, Tutors

# Entities in E/R Diagram

**Entities – Department, Course, Module, Student, Lecturer**

Department

Course

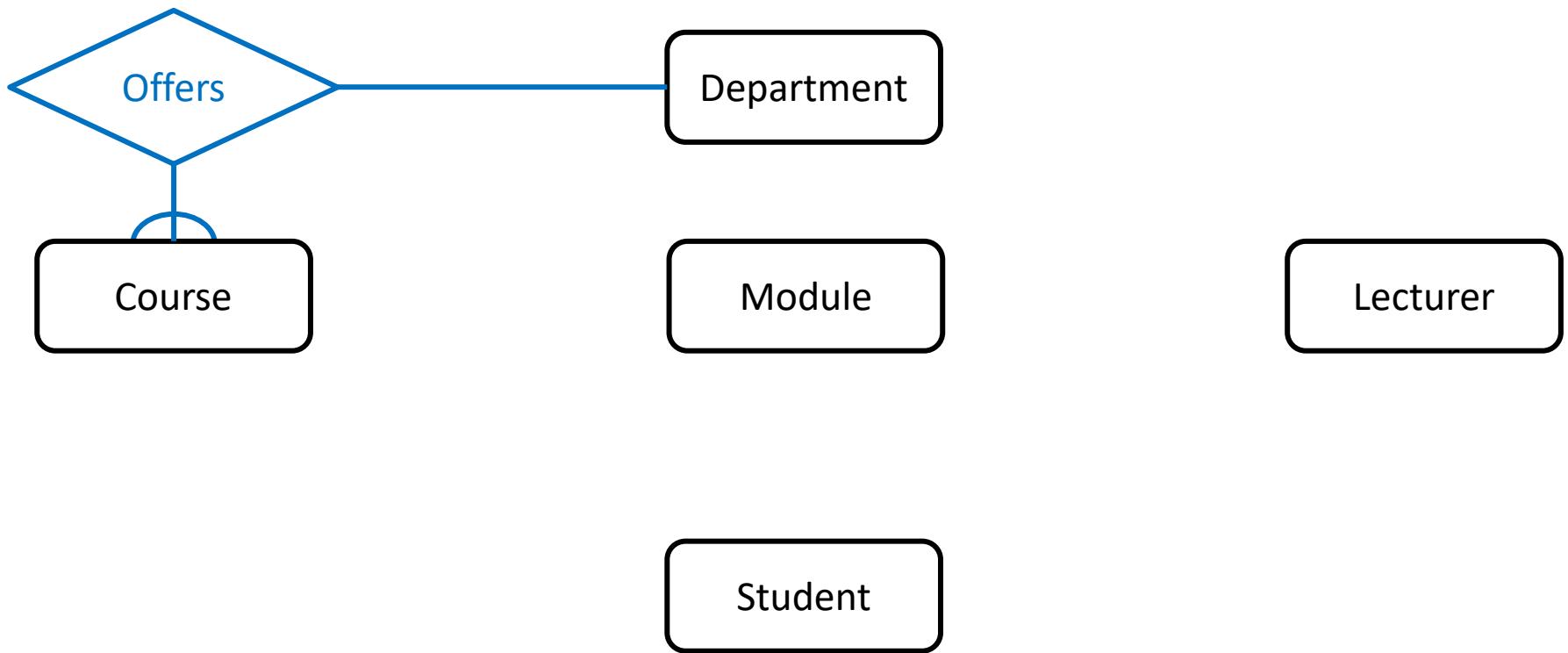
Module

Lecturer

Student

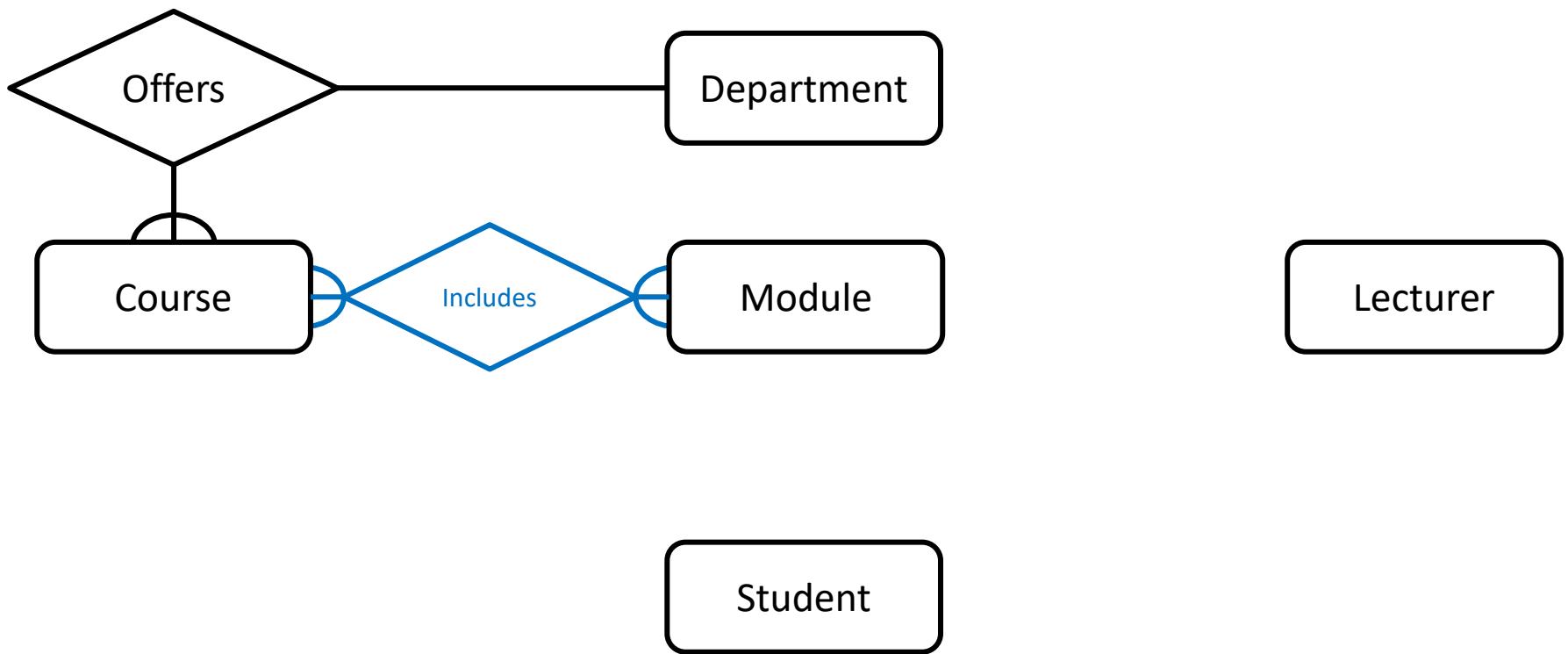
# Relationships in E/R Diagram

Each Department offers several Courses



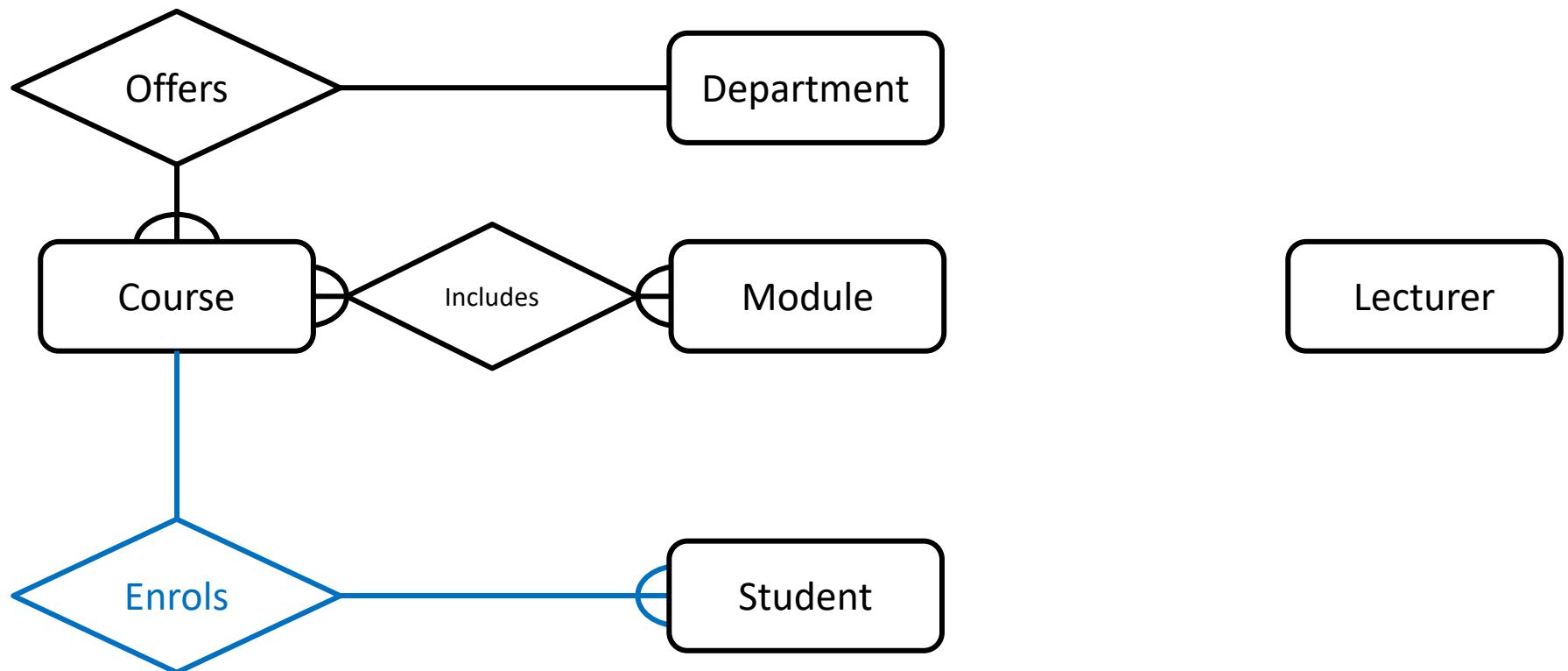
# Relationships in E/R Diagram

A number of modules make up each Course



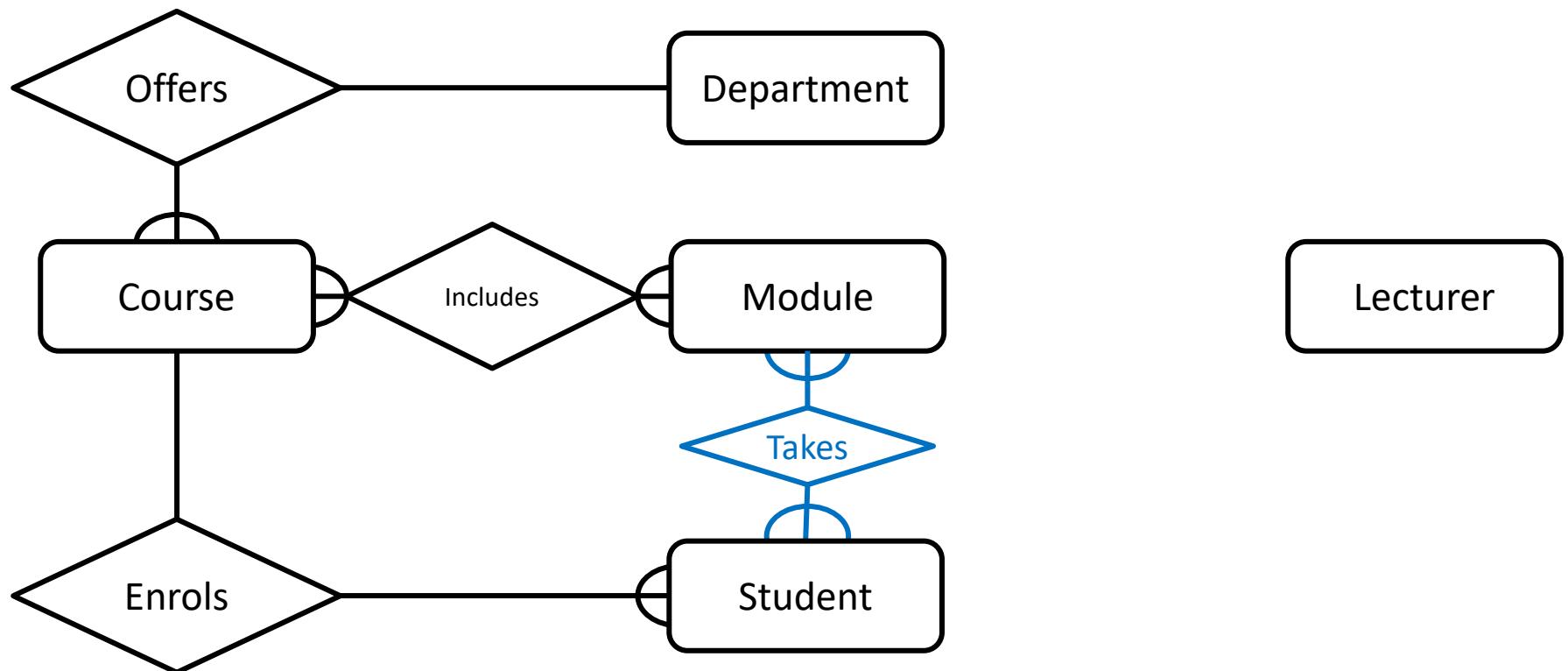
# Relationships in E/R Diagram

Students enroll in a particular course



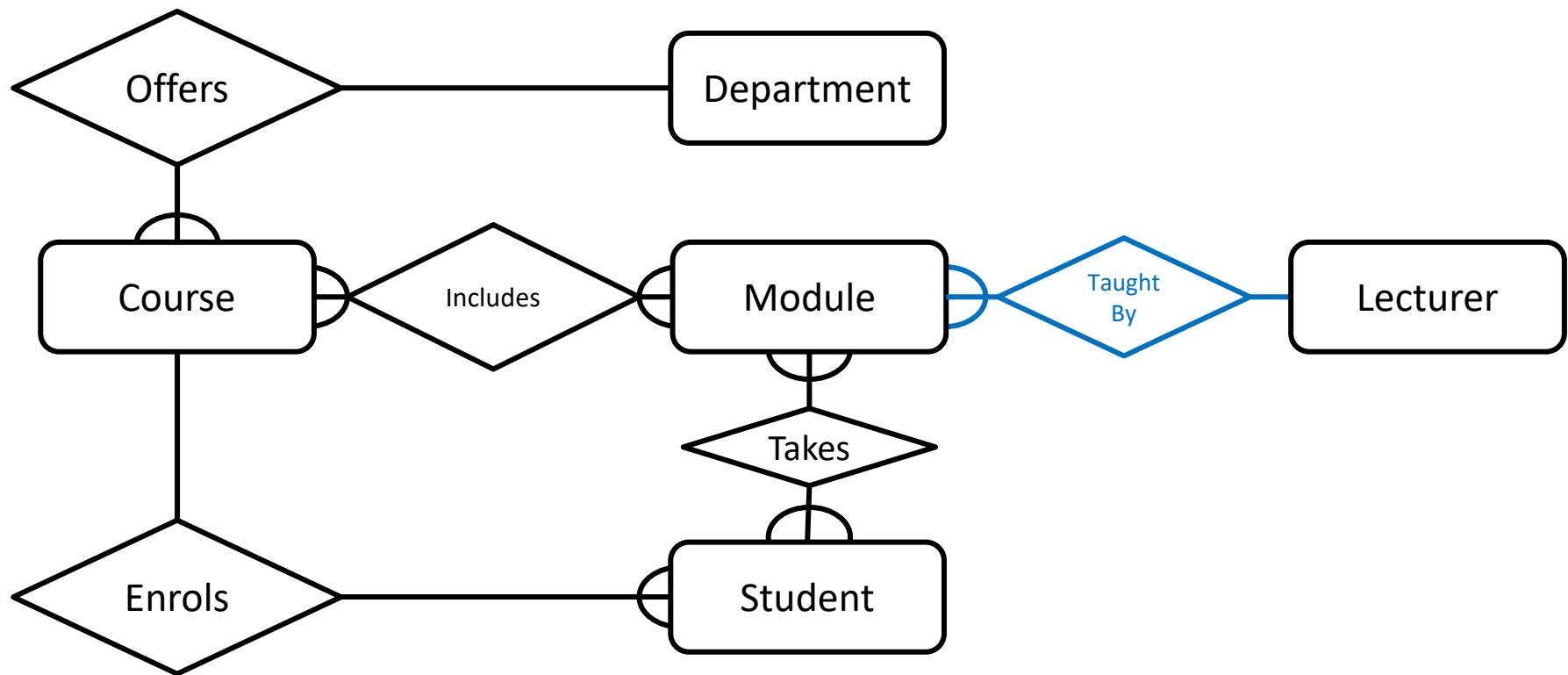
# Relationships in E/R Diagram

Students take several modules



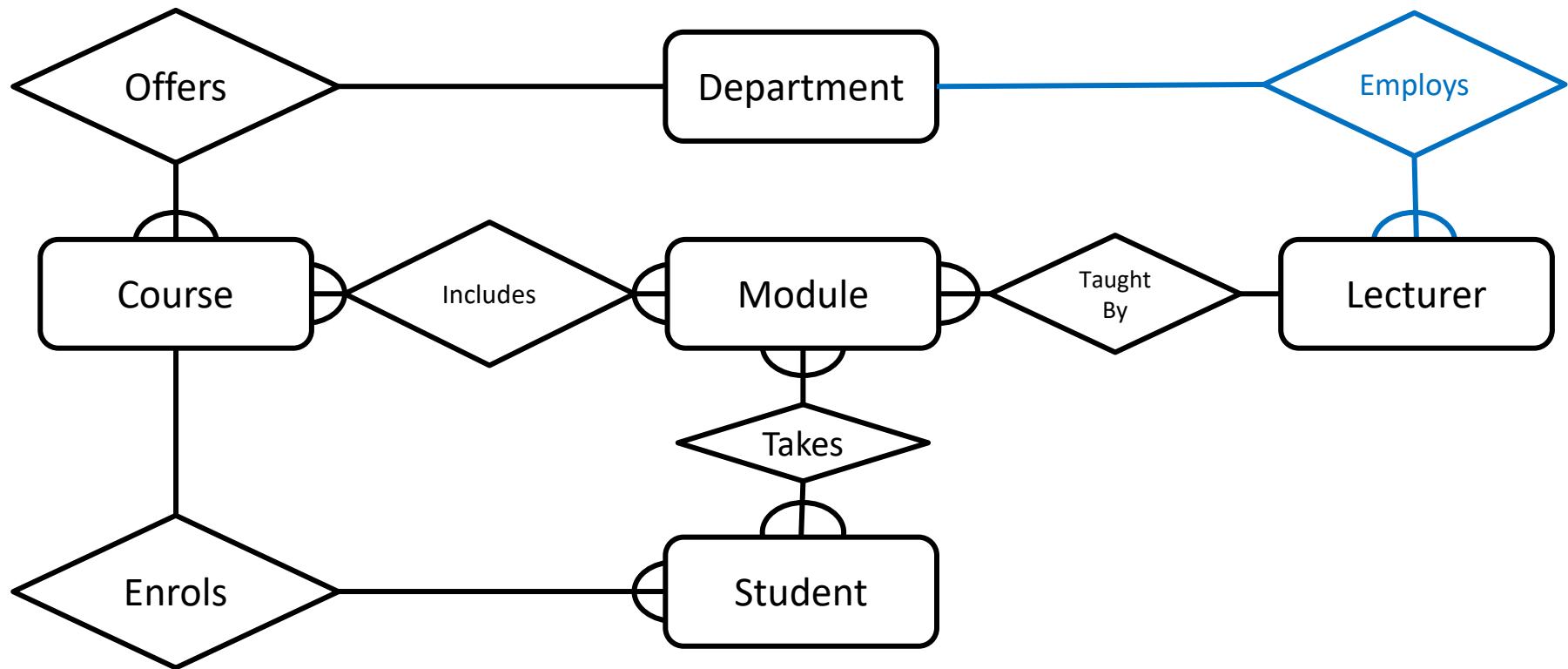
# Relationships in E/R Diagram

A lecturer can teach more than one module



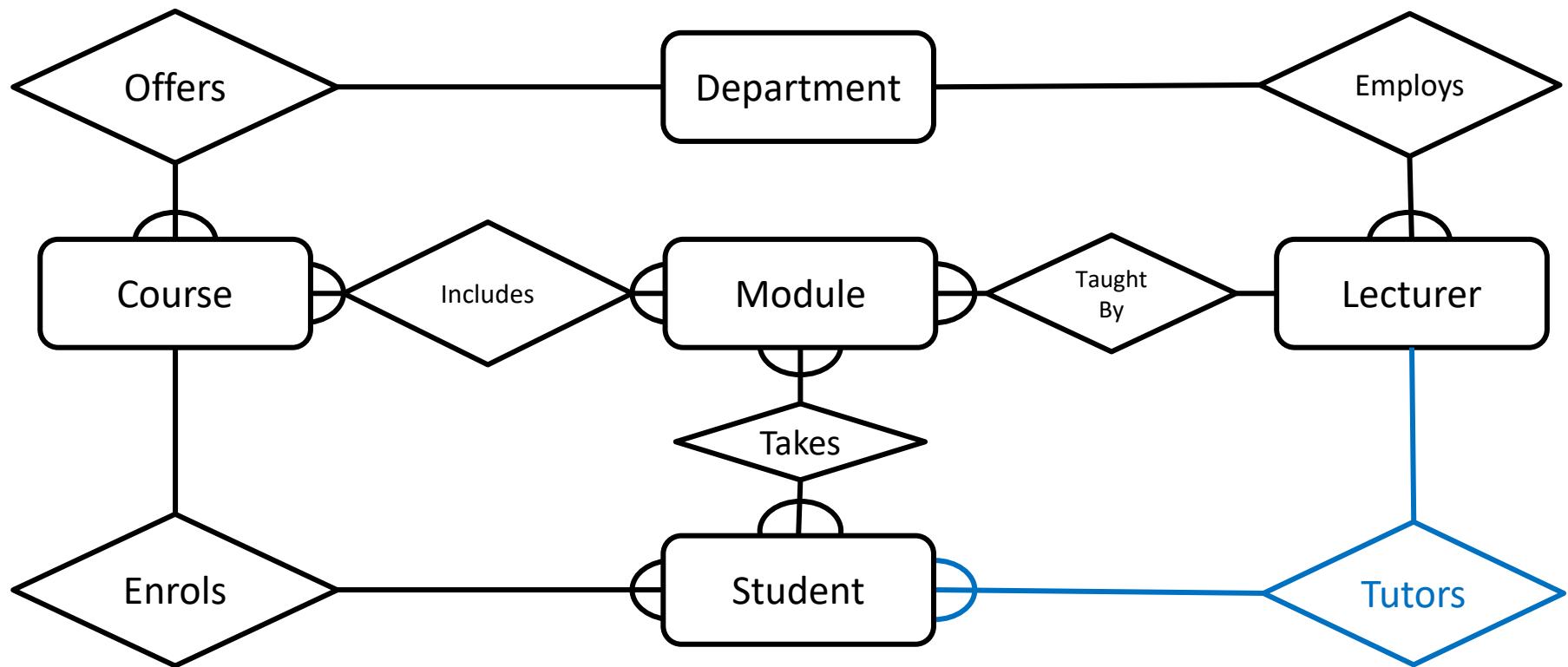
# Relationships in E/R Diagram

Each department employs a number of lecturers



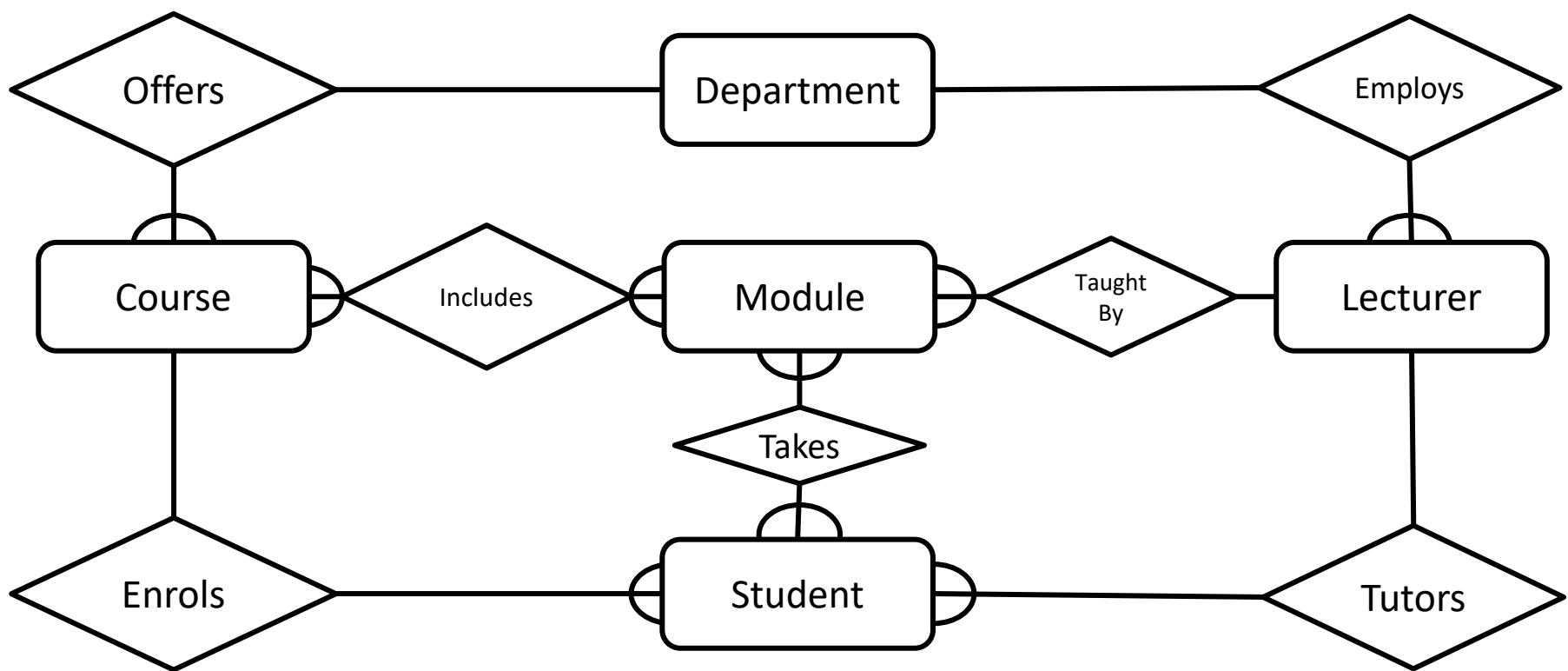
# Relationships in E/R Diagram

Each Lecturer tutors a number of Students



# The Complete E/R Diagram

- The completed diagram. All that remains is to remove M:M relationships



# Removing M:M Relationships

- Many to many relationships are difficult to represent in a database:

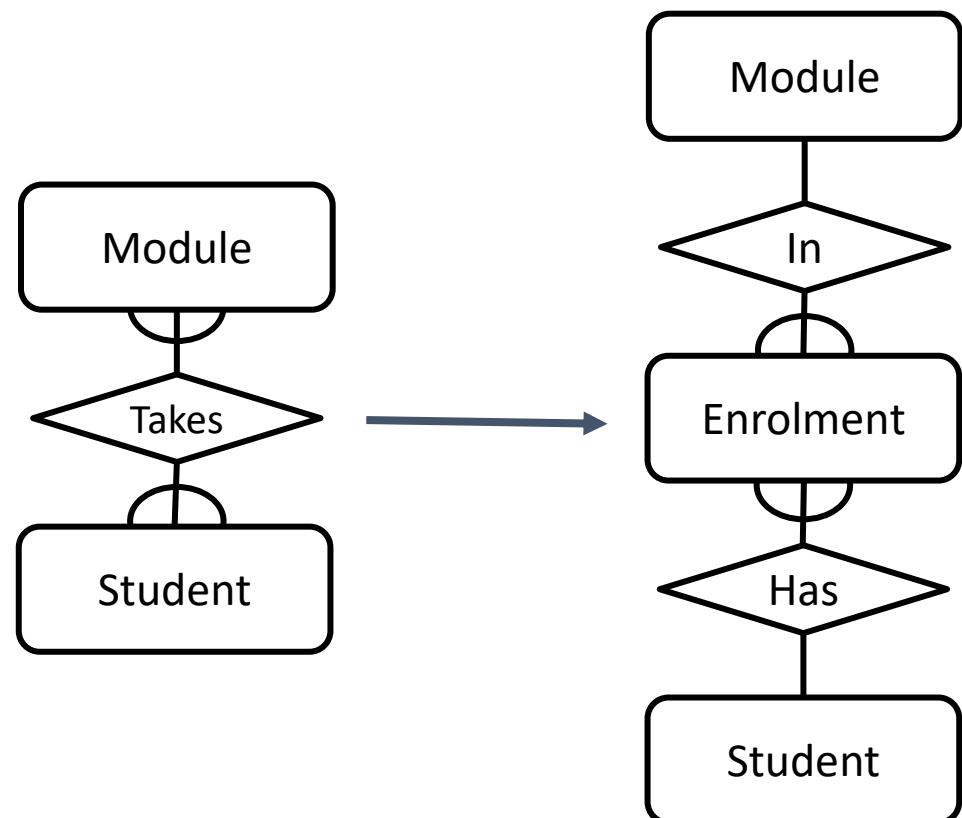
Student		
SID	sName	sMod
1001	Jack Smith	DBI
1001	Jack Smith	PRG
1001	Jack Smith	IAI
1002	Anne Jones	PRG
1002	Anne Jones	IAI
1002	Anne Jones	Vis

Module	
MID	mName
DBI	Databases and Interfaces
PRG	Programming
IAI	AI
VIS	Computer Vision

Student		
SID	sName	sMod
1001	Jack Smith	DBI, PRG, IAI
1002	Anne Jones	VIS, IAI, PRG

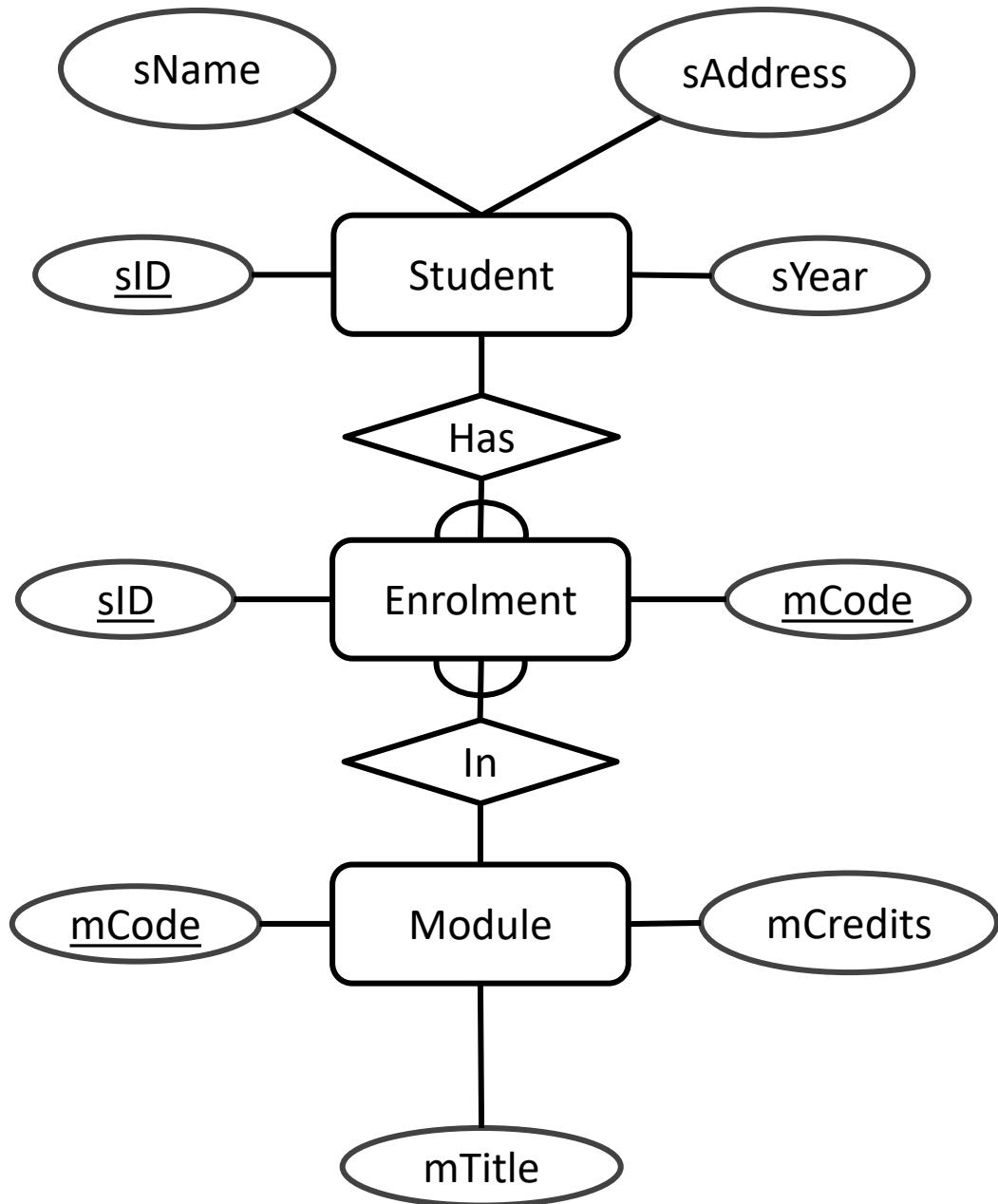
# Removing M:M Relationships

- We can split a many to many relationship into two, one to many relationships
- An additional entity is created to represent the M:M relationship



# Relationships

- The Enrolment table
  - Will have columns for the student ID and module code attributes
  - Will have a foreign key to Student for the 'has' relationship
  - Will have a foreign key to Module for the 'in' relationship



# Entities and Attributes

- Sometimes it is hard to tell if something should be an entity or an attribute
  - They both represent objects or facts about the world
  - They are both often represented by nouns in descriptions
- General guidelines
  - Entities can have attributes but attributes have no smaller parts
  - Entities can have relationships between them, but an attribute belongs to a single entity

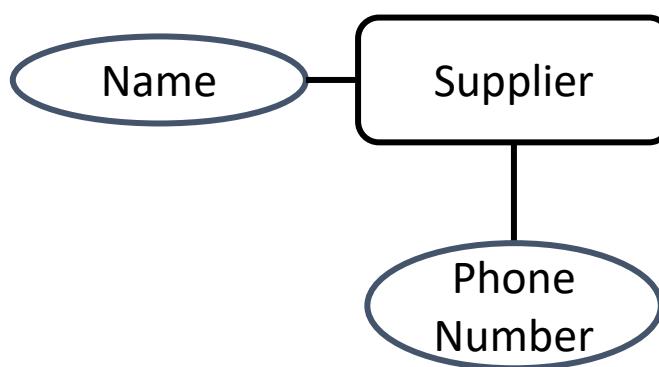
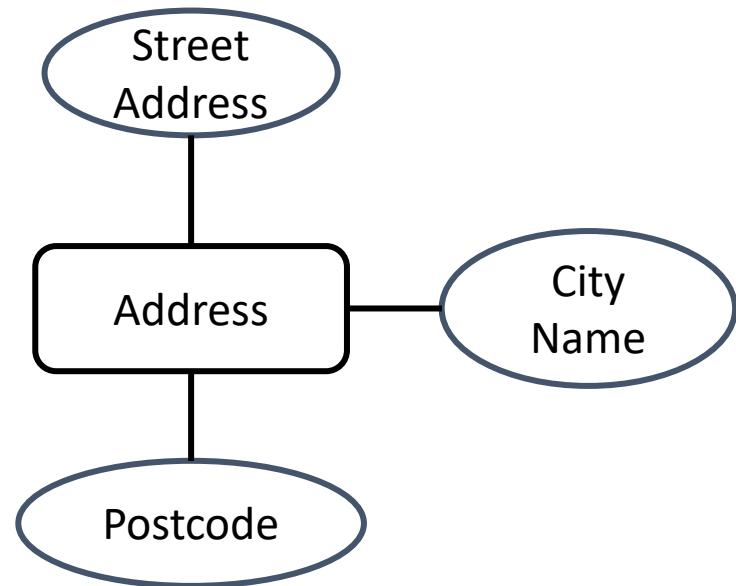
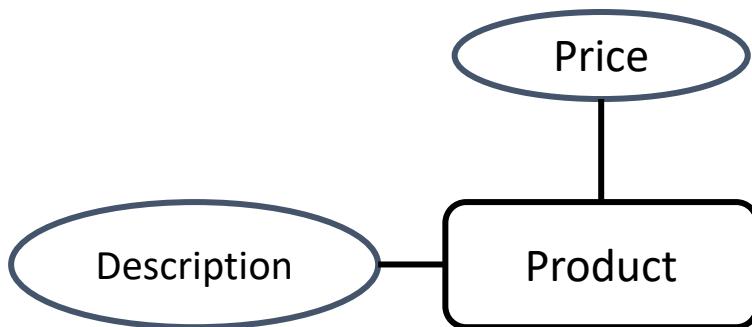
# Example

We want to represent information about products in a database. Each product has a description, a price and a supplier. Suppliers have addresses, phone numbers, and names. Each address is made up of a street address, a city name, and a postcode.

# Example - Entities/Attributes

- Entities or attributes:
  - product
  - description
  - price
  - supplier
  - address
  - phone number
  - name
  - street address
  - city name
  - postcode
- Products, suppliers, and addresses all have smaller parts so we make them entities
- The others have no smaller parts and belong to a single entity

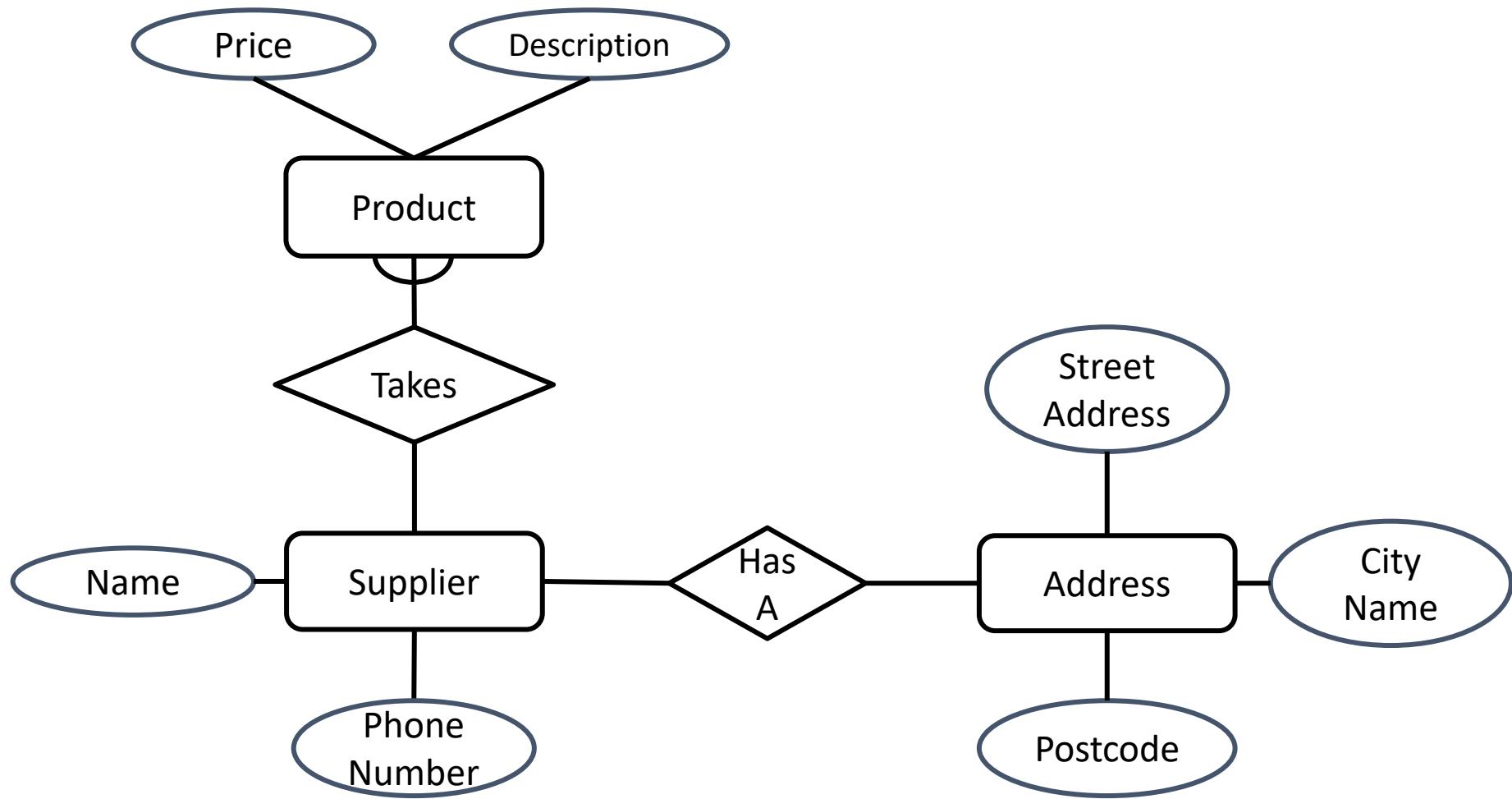
# Example - E/R Diagram



# Example - Relationships

- Each product has a supplier
  - Each product has a single supplier but there is nothing to stop a supplier supplying many products
  - A many to one relationship
- Each supplier has an address
  - A supplier has a single address
  - It does not seem sensible for two different suppliers to have the same address
  - A one to one relationship

# Example - E/R Diagram

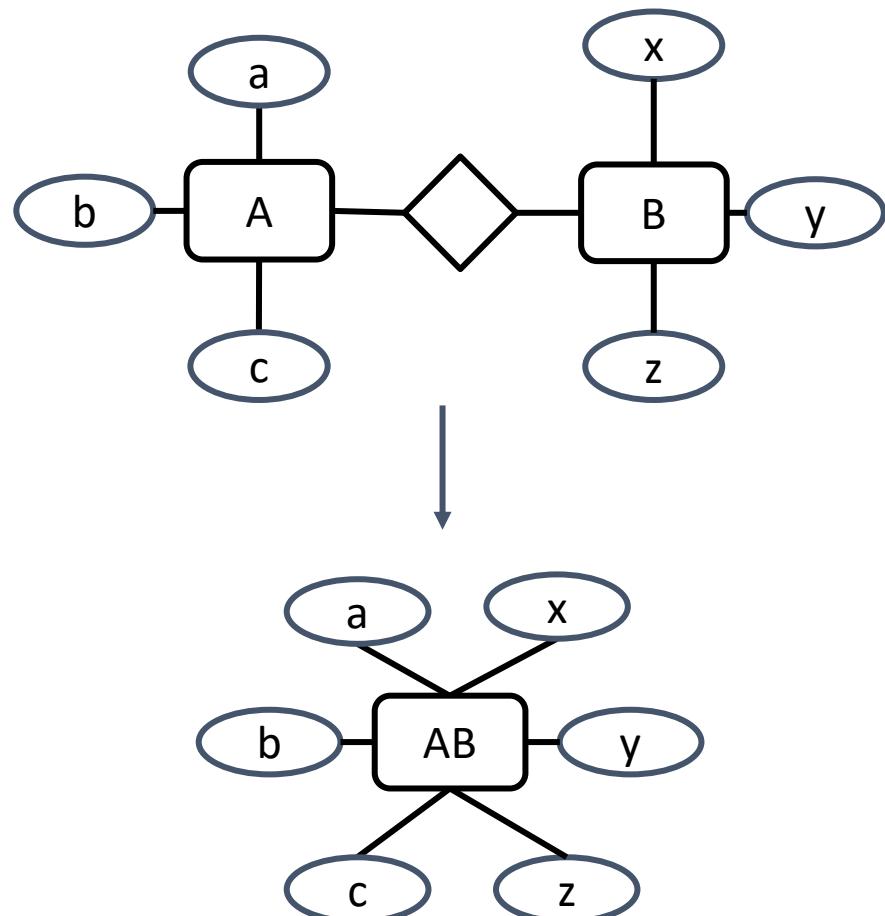


# One to One Relationships

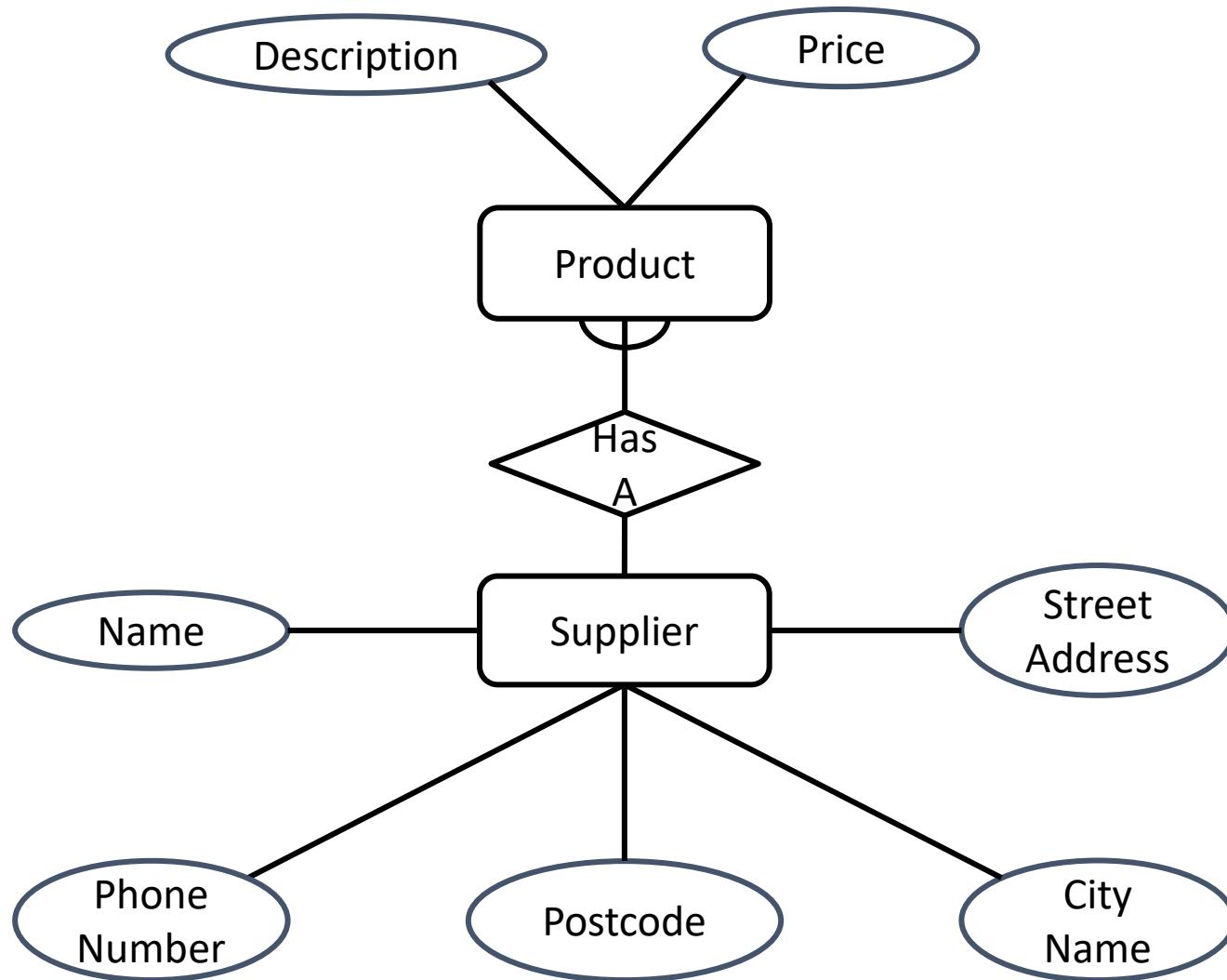
- Some relationships between entities, A and B, might be redundant if:
  - It is a 1:1 relationship between A and B
  - Every A is related to a B and every B is related to an A.
- Example
  - The supplier-address relationship - Is one to one
  - Every supplier has an address
  - We don't need addresses that are not related to a supplier

# One to One Relationships

- We can merge the two entities that take part in a redundant relationship together
  - They become a single entity
  - The new entity has all the attributes of the old ones



# Example - E/R Diagram



# E/R Diagram: Summary of Steps

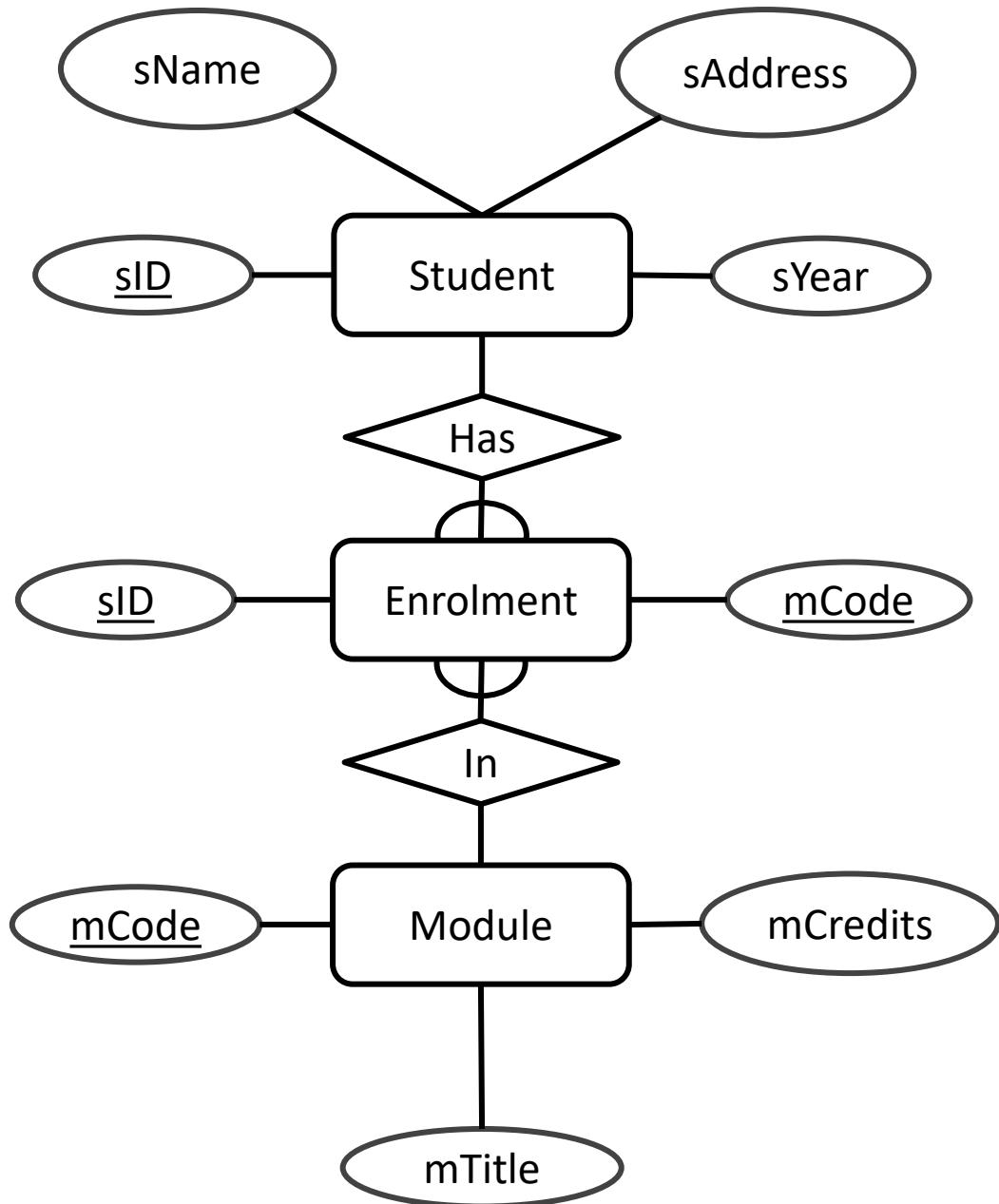
- 1) From a description of the requirements, identify:
  - Entities
  - Attributes
  - Relationships
  - Cardinality ratios of the relationships
- 2) Draw the E/R diagram and then
  - Look at one to one relationships as they might be redundant
  - Look at many to many relationships as they will often need to be split into two one to many links, using an intermediate entity

# From E/R Diagram to SQL Tables.

- Entities Become table names.
- Attributes of an entity becomes the columns.
- Relationships become foreign keys.

# Relationships

- Relationships as Foreign Keys:
  - 1:1 are usually not used, or can be treated as a special case of M:1
  - M:1 are represented as a foreign key from the M-side to the 1.
  - M:M are split into two M:1 relationships.



Questions?

# Database: Normalisation

Jianjun Chen

# What is Normalisation?

“Like E/R diagram. Normalisation is another way of creating accurate representations of the data, relationships between the data, and constraints on the data that is pertinent to the enterprise.”

“It is a design technique for producing a set of suitable relations that support the data requirements of an enterprise.”

# “Suitable Relations”

Characteristics of a suitable set of relations include:

- The minimal number of attributes necessary to support the data requirements of the enterprise;
- Attributes with a close logical relationship are found in the same relation;
- Minimal redundancy with each attribute represented only once with the important exception of attributes that form all or part of foreign keys.

From textbook

# Minimal Redundancy?

- Given the example from E/R diagram, department and lecturer can be designed as:

Department

dName	lecName
CS	Matt
CS	Andrew
EEE	Thomas
EEE	John

Lecturer

lecName
Matt
Andrew
Thomas
John

Department

dName
CS
EEE

Lecturer

lecName	dName
Matt	CS
Andrew	CS
Thomas	EEE
John	EEE



# Minimal Redundancy?

- However, the department table in the left version has duplications of dName in the department table.
- Also, the number of rows is the same as Lecturer's
  - Why not combining the table with Lecturer table then?
  - If we combine them, we will have the same problems as the staffbranch table (see next slide).

Department

dName	lecName
CS	Matt
CS	Andrew
EEE	Thomas
EEE	John

Lecturer

lecName
Matt
Andrew
Thomas
John

Department

dName
CS
EEE

Lecturer

lecName	dName
Matt	CS
Andrew	CS
Thomas	EEE
John	EEE

# Data Redundancy

Staff

staffNo	sName	position	salary	branchNo
SL21	John White	Manager	30000	B005
SG37	Ann Beech	Assistant	12000	B003
SG14	David Ford	Supervisor	18000	B003
SA9	Mary Howe	Assistant	9000	B007
SG5	Susan Brand	Manager	24000	B003
SL41	Julie Lee	Assistant	9000	B005

Branch

branchNo	bAddress
B005	22 Deer Rd, London
B007	16 Argyll St, Aberdeen
B003	163 Main St, Glasgow

Staff Branch

staffNo	sName	position	salary	branchNo	bAddress
SL21	John White	Manager	30000	B005	22 Deer Rd, London
SG37	Ann Beech	Assistant	12000	B003	163 Main St, Glasgow
SG14	David Ford	Supervisor	18000	B003	163 Main St, Glasgow
SA9	Mary Howe	Assistant	9000	B007	16 Argyll St, Aberdeen
SG5	Susan Brand	Manager	24000	B003	163 Main St, Glasgow
SL41	Julie Lee	Assistant	9000	B005	22 Deer Rd, London

# Data Redundancy

- StaffBranch relation has redundant data; the details of a branch are repeated for every member of staff.
- In contrast, the branch information appears only once for each branch in the Branch relation and only the branch number (branchNo) is repeated in the Staff relation, to represent where each member of staff is located.

Staff Branch

staffNo	sName	position	salary	branchNo	bAddress
SL21	John White	Manager	30000	B005	22 Deer Rd, London
SG37	Ann Beech	Assistant	12000	B003	163 Main St, Glasgow
SG14	David Ford	Supervisor	18000	B003	163 Main St, Glasgow
SA9	Mary Howe	Assistant	9000	B007	16 Argyll St, Aberdeen
SG5	Susan Brand	Manager	24000	B003	163 Main St, Glasgow
SL41	Julie Lee	Assistant	9000	B005	22 Deer Rd, London

Branch

branchNo	bAddress
B005	22 Deer Rd, London
B007	16 Argyll St, Aberdeen
B003	163 Main St, Glasgow

# Data Redundancy and Update Anomalies

These examples illustrate the problems of data redundancy. Let's take StaffBranch for example:

- Insert anomalies: Must input the correct branch information every time a new staff is added.
- Delete anomalies: Deleting the last staff of a branch also deletes the information of that branch.
- Modification anomalies: To change one branch information, all staffs (rows) in that branch must also be updated.

# How to Split Tables?

- As a result, StaffBranch is better split into staff and branch.
  - And we know the answer beforehand.
- What information can help us to redesign tables when we don't know the answer?
- How about working out the E/R model in the reversed way?
  - An entity is associated with **its** attributes.
  - **M:1** relationships will become foreign keys.

We will be talking about this until slide 26

# How to Identify Entities?

“An entity is associated with **its** attributes.”

- That means all attributes of an entity must fit into one context.
- For example, a lecturer table can have:
  - LecID
  - LName
  - LEmail
- Attributes like these won’t fit:
  - StudentID
  - SName

# How to Identify Entities?

- Apart from the attribute names, what else is critical when we tries to classify these attributes?
- A lecturer ID is associated with a lecturer name and his email address.
- Does lecturer ID has the same relationship with student name? No!
- There's a term for this: **functional dependency**

LecturerStudent{**LecID**, LName, LEmail, **StudentID**, SName}

# Functional Dependency

A method for finding relationships between attributes within a table

# Functional Dependencies

- We want to find relationship between attributes within a table so that we can regroup attributes based on their context and split the big table.
- Introducing **functional dependency (FD)**:

“If A and B are attribute sets of relation R, B is functionally dependent on A (**denoted  $A \rightarrow B$** ), if each value of A in R is associated with exactly one value of B in R.”

- A is called **determinant**.

# Functional Dependencies

For the attributes below:

LecturerStudent{**LecID**, LName, LEmail, **StudentID**, SName}

- $\text{LecID} \rightarrow \text{LName}, \text{LEmail}$ 
  - Each LecID of a lecturer is associated with exactly one lecturer name and his email address.
- $\text{StudentID} \rightarrow \text{SName}$ 
  - Each studentID of a student is associated with exactly one student name.

# Functional Dependencies

- Why not LName  $\rightarrow$  LecID ?
  - Two staffs may have the same name. Thus a name will probably be associated with two IDs.
- Why not LecID  $\rightarrow$  SName?
  - They don't seem to have connections at all.
- How about LEmail  $\rightarrow$  LecID, LName ?
  - This is a valid FD. One staff only has one assigned email address.
  - It looks strange though.

LecturerStudent{LecID, LName, LEmail, StudentID, SName}

# Functional Dependencies

Observe these functional dependencies carefully, you will realise that:

- If these attributes are put together, they can form a relation, with the determinant being the unique key or primary key of the relation
- For example:
  - LecID → LName, LEmail
  - StudentID → SName

LecturerStudent{**LecID**, LName, LEmail, **StudentID**, SName}

# Functional Dependencies

- However, we have a big problem here, the FD below is also true:
  - $\underline{\text{LecID}, \text{LName}, \text{LEmail}} \rightarrow \text{LName, LEmail}$
- Is  $(\underline{\text{LecID}, \text{LName}, \text{LEmail}})$  a good primary key?
- Recall the definition of super key, candidate key and primary key.

# Full Functional Dependency

- **Full functional dependency** indicates that if A and B are two sets of attributes of a relation, B is fully functionally dependent on A, if B is functionally dependent on A, but not on any proper subset of A.
- In other words, determinants should have the minimal number of attributes necessary to maintain the functional dependency with the attribute(s) on the right hand-side.

# Partial Functional Dependency

## Partial FDs:

- A FD,  $A \rightarrow B$ , is a partial FD, if some attribute of A can be removed and the FD still holds
- Formally, there is some proper subset of A,  $C \subset A$ , such that  $C \rightarrow B$

# Examples

- Full functional dependency in the Staff relation:
  - $\text{staffNo} \rightarrow \text{branchNo}$
- How about:
  - $\text{staffNo}, \text{sName} \rightarrow \text{branchNo}$ ?
  - This is a partial dependency since  $\text{branchNo}$  is also functionally dependent on a subset of  $(\text{staffNo}, \text{sName})$ , namely  $\text{staffNo}$ .

Staff				
staffNo	sName	position	salary	branchNo
SL21	John White	Manager	30000	B005
SG37	Ann Beech	Assistant	12000	B003
SG14	David Ford	Supervisor	18000	B003
SA9	Mary Howe	Assistant	9000	B007
SG5	Susan Brand	Manager	24000	B003
SL41	Julie Lee	Assistant	9000	B005

# Determinant in Full/Partial FDs

Determinants in full functional dependencies:

- Can become candidate keys if we split the table

Determinants in partial functional dependencies:

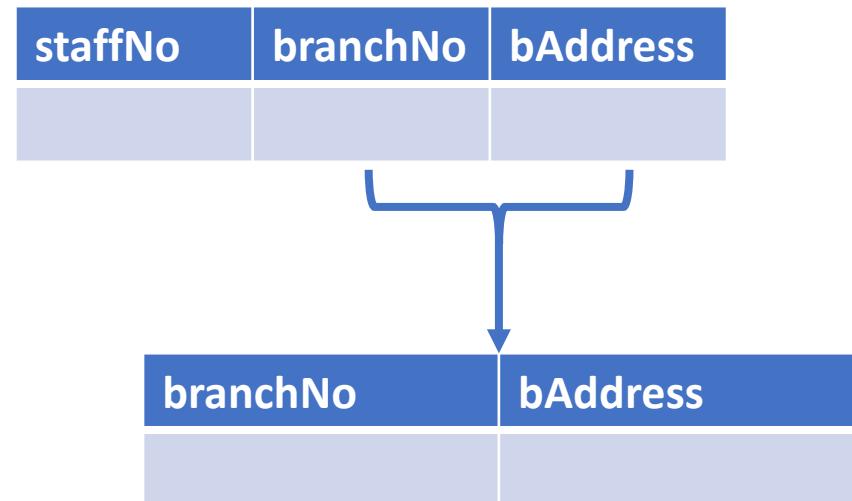
- Can only become super keys.

# FDs in Normalisation

We only care about full FDs in normalisation. Why?

- Partial FDs results in super keys, if you use a foreign key to refer to a super key in another table, you will need extra columns in the referencing table.
  - branchNo VS (branchNo, bAddress) inside Staff table.
  - Unnecessary!

staffNo	branchNo
branchNo	bAddress



# More about Determinants

- If you observe the tuples in the Staff and Branch table, you can find out that **the determinant has a M:1 or 1:1 relationship with other attributes in FDs.**
- Two staffs may have the same name.
  - Thus, one staff name can be associated with more than one staff number. (M:1)
- Two staffs may have the same position.
  - Thus, one position may be associated with more than one staff number (M:1)

Staff				
staffNo	sName	position	salary	branchNo
SL21	John White	Manager	30000	B005
SG37	Ann Beech	Assistant	12000	B003
SG14	David Ford	Supervisor	18000	B003

# More about Determinants

“The determinant has a M:1 or 1:1 relationship with other attributes in FDs”

- Two staffs may work in the same branch.
  - Thus, one branchNo may be associated with more than one staffNo. (M:1)
- A branch address is associated with exactly one branchNo
  - 1:1 relationship

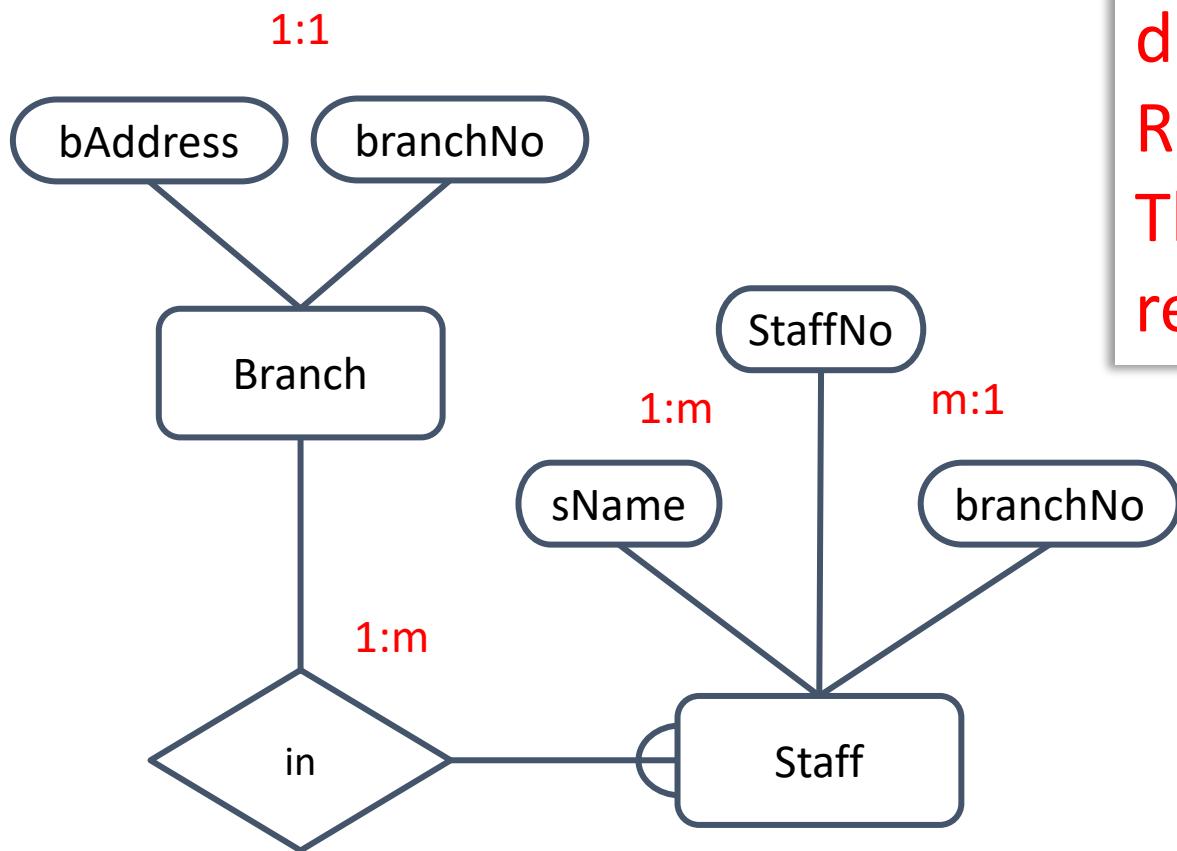
Branch	
branchNo	bAddress
B005	22 Deer Rd, London
B007	16 Argyll St, Aberdeen
B003	163 Main St, Glasgow

# More about Determinants

“The determinant has a M:1 or 1:1 relationship with other attributes in FDs”

- Remember in ER Modelling, M:1 relationships between tables will become foreign keys.
- We can infer that, For attributes that have a M:1 relationships, if they belong to the same context, they will be grouped into one relation, otherwise, they will be split into two tables and linked with a foreign key.

“If they belong to the same context, they will be grouped into one relation, otherwise, they will be split into two tables and linked with a foreign key”



That's why these databases are called Relational databases:  
They are really about relationships! ☺

# Full FDs in StaffBranch

Staff Branch					
staffNo	sName	position	salary	branchNo	bAddress
SL21	John White	Manager	30000	B005	22 Deer Rd, London
SG37	Ann Beech	Assistant	12000	B003	163 Main St, Glasgow
SG14	David Ford	Supervisor	18000	B003	163 Main St, Glasgow
SA9	Mary Howe	Assistant	9000	B007	16 Argyll St, Aberdeen
SG5	Susan Brand	Manager	24000	B003	163 Main St, Glasgow
SL41	Julie Lee	Assistant	9000	B005	22 Deer Rd, London

1. staffNo → sName, position, salary, branchNo, bAddress
2. branchNo → bAddress

Observe this dependency chain carefully:

- staffNo → branchNo → bAddress
  - This looks like a relationship between staff and branch.

# Transitive Dependency

The previous example is a **transitive dependency**.

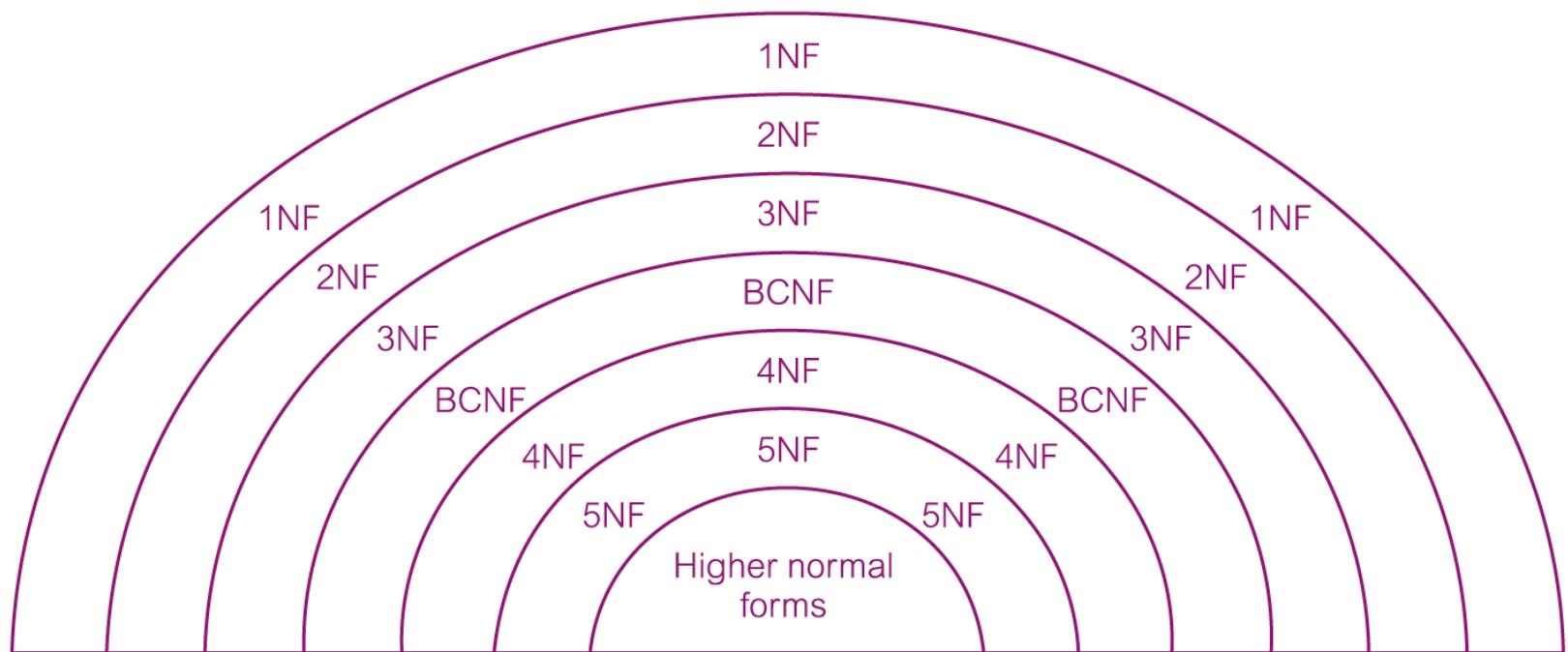
“Transitive dependency describes a condition where A, B, and C are attributes of a relation such that if  $A \rightarrow B$  and  $B \rightarrow C$ , then C is transitively dependent on A via B (provided that A is not functionally dependent on B or C).”

- Why is the underlined constraint needed?  
(Explained later)

# Normal Forms

How can we use FDs and TDs to redesign tables

# The Process of Normalisation



Steps of redesigning tables to make them “suitable”

# Normalisation: Some Notes

- In relational database, we always try to assign tables with primary keys (or at least candidate keys).
- Why? Because relational databases are about relations.
  - Attributes within one context forms an entity (table).
  - Primary keys make referencing possible.
  - Attributes connecting entities become foreign keys.
  - Foreign keys form these connections.
- Can I insist that no unique keys are used?
  - Then its beyond the topic of this module :-)
  - Remember, we are learning **one of the possible design decisions** of managing data, don't limit your imagination.

# First Normal Form

- In most definitions of the relational model
  - All data values should be atomic
  - This means that table entries should be single values, not sets or composite objects
- A relation is said to be in **first normal form (1NF)**:
  - if all data values are atomic

# Unnormalised Form (UNF)

Unnormalised relation

Module	Dept	Lecturer	Texts
M1	D1	L1	T1, T2
M2	D1	L1	T1, T3
M3	D1	L2	T4
M4	D2	L3	T1, T5
M5	D2	L4	T6

Primary key is Module

# Problems With UNF

Look at “Texts” attribute:

- To **update** the textbook name “T1” to something else, you need to manually update all “T1”s.
- To **delete** T1 from the pool of textbooks, you need to manually do so, too.
- You cannot **add** a textbook without giving the information of Module (because it’s a Primary Key).

## Method 1:

1. Remove the repeating group by entering appropriate data into the empty columns of rows containing the repeating data (also called ‘flattening’ the table).
2. Assign a new primary/unique key to the new table.

Unnormalised relation

Module	Dept	Lecturer	Texts
M1	D1	L1	T1, T2
M2	D1	L1	T1, T3
M3	D1	L2	T4
M4	D2	L3	T1, T5
M5	D2	L4	T6

1NF

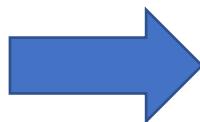
Module	Dept	Lecturer	Text
M1	D1	L1	T1
			T2
M2	D1	L1	T1
M2	D1	L1	T3
M3	D1	L2	T4
M4	D2	L3	T1
M4	D2	L3	T5
M5	D2	L4	T6

## Method 2:

1. Identify primary/unique key.
2. Place the repeating data along with a copy of the original (unique) key attribute(s) into a separate relation.

Unnormalised relation

Module	Dept	Lecturer	Texts
M1	D1	L1	T1, T2
M2	D1	L1	T1, T3
M3	D1	L2	T4
M4	D2	L3	T1, T5
M5	D2	L4	T6



1NFb

Module	Dept	Lecturer
M1	D1	L1
M2	D1	L1
M3	D1	L2
...	...	...

1NFa

Module	Texts
M1	T1
M1	T2
M2	T1
M2	T2

1NF

Module	Dept	Lecturer	Text
M1	D1	L1	T1
M1	D1	L1	T2
M2	D1	L1	T1
M2	D1	L1	T3
M3	D1	L2	T4
M4	D2	L3	T1
M4	D2	L3	T5
M5	D2	L4	T6

# Problems in 1NF

Look at the Primary key (Module, Text)

- Changing module code from “M1” to something else requires you to check the whole table.
- Adding a new lecturer with no modules and text is impossible
- If a (department, lecturer) pair is modified, the change must be made to all (Module, Text) pairs.
  - For example, to change (D1, L1) to some other value, you must manually change the first four rows.
- If (M3, T4) is deleted, L2 will be permanently lost!

# Second Normal Form

Definition of **second normal form 2NF**:

- A relation is in second normal form (2NF) if it is in 1NF and
- no non-key attribute is partially dependent **on the primary key**
- In other words, no  $C \rightarrow B$  where C is a strict subset of a primary key and B is a non-key attribute.

# Second Normal Form: Example

R

Module	Dept	Lecturer	Text
M1	D1	L1	T1
M1	D1	L1	T2
M2	D1	L1	T1
M2	D1	L1	T3
M3	D1	L2	T4
M4	D2	L3	T1
M4	D2	L3	T5
M5	D2	L4	T6

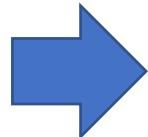
- R is in 1NF but not in 2NF
- We have the FD:  
 $\{\text{Module}, \text{Text}\} \rightarrow \{\text{Lecturer}, \text{Dept}\}$
- But also  
 $\{\text{Module}\} \rightarrow \{\text{Lecturer}, \text{Dept}\}$
- And so Lecturer and Dept are partially dependent on the primary key

# 1NF to 2NF

- Identify the primary key(s) for the 1NF relation.
- Identify the functional dependencies in the relation.
- If partial dependencies exist on the primary key, remove them by placing attributes of the corresponding full FD in a new relation and leave the its determinant in the original table.

# 1NF to 2NF

1NF			
Module	Dept	Lecturer	Text
M1	D1	L1	T1
M1	D1	L1	T2
M2	D1	L1	T1
M2	D1	L1	T3
M3	D1	L2	T4
M4	D2	L3	T1
M4	D2	L3	T5
M5	D2	L4	T6



2NFa		
Module	Dept	Lecturer
M1	D1	L1
M2	D1	L1
M3	D1	L2
M4	D2	L3
M5	D2	L4

2NFb	
Module	Text
M1	T1
M1	T2
M2	T1
M2	T3
M3	T4
M4	T1
M4	T5
M1	T6

$\{ \text{Module}, \text{Text} \} \rightarrow \{ \text{Lecturer}, \text{Dept} \}$

But also

$\{ \text{Module} \} \rightarrow \{ \text{Lecturer}, \text{Dept} \}$

2NFa

Module	Dept	Lecturer
M1	D1	L1
M2	D1	L1
M3	D1	L2
M4	D2	L3
M5	D2	L4

# Problems in 2NF

Look at the table 2NFa

- We cannot **add** a lecturer who is not assigned with any module. Because module is the primary key.
  - This is a theoretical discussion. Similar issues can be a real problem in some other tables.
- By **deleting** M3, we lose L2 forever.
- To **change** the department for L1, we need to change multiple rows manually

# Third Normal Form (3NF)

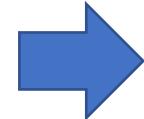
- Based on the concept of transitive dependency.
- A relation that is in 1NF and 2NF and in which no non-key attribute is transitively dependent on **the primary key**.

# 2NF to 3NF

- Identify the primary key in the 2NF relation.
- Identify functional dependencies in the relation.
- If transitive dependencies exist on the primary key, remove them by placing them in a new relation along with a copy of their determinant.

# 2NF to 3NF – Example

Module	Dept	Lecturer
M1	D1	L1
M2	D1	L1
M3	D1	L2
M4	D2	L3
M5	D2	L4



Lecturer	Dept
L1	D1
L2	D1
L3	D2
L4	D2

Module	Lecturer
M1	L1
M2	L1
M3	L2
M4	L3
M5	L4

$\{Module\} \rightarrow \{Lecturer\}$

$\{Lecturer\} \rightarrow \{Dept\}$

So there is a transitive FD from the primary key  $\{Module\}$  to  $\{Dept\}$

3NFa

Lecturer	Dept
L1	D1
L2	D1
L3	D2
L4	D2

3NFb

Module	Lecturer
M1	L1
M2	L1
M3	L2
M4	L3
M5	L4

## Problems Resolved in 3NF

- Problems in 2NF
  - INSERT: Can't add lecturers who teach no modules
  - UPDATE: To change the department for L1 we must alter two rows
  - DELETE: If we delete M3 we delete L2 as well
- In 3NF all of these are resolved (for this relation – but 3NF can still have anomalies!)

# Transitive Dependency and FK

Let's go back and check the definition of the transitive dependency:

“Transitive dependency describes a condition where A, B, and C are attributes of a relation such that if  $A \rightarrow B$  and  $B \rightarrow C$ , then C is transitively dependent on A via B. **Provided that A is not functionally dependent on B or C”**

- Translation:

“Provided that  $B \rightarrow A$  or  $C \rightarrow A$  is not true”

# Transitive Dependency

Example:

- staffNo (A) → branchNo (B) → bAddress (C)
  - This is transitive dependency.
- LecID (A) → LEmail (B) → LName (C)
  - Not a transitive dependency. Because LEmail is functionally dependent on LecID.
- Why the second example should not be considered as a transitive dependency (should not be split into two tables)?

# Transitive Dependency and FK

- LEmail (B) → LeclD (A)
  - For each Email address, you can find one LeclD.
  - Below are valid values of email and id:
    - Oyo@abc.com → 001
    - Yoy@abc.com → 001
- But LeclD is a candidate key, it is not allowed to have duplicates.
  - Thus, for each Email address, you can find a **unique** LeclD.
- Thus LEmail and LeclD has a 1:1 relationship.

# Transitive Dependency and FK

- After splitting them:

Table1: LecID, LEmail

Table2: LEmail, LName

- Table1 and Table2 has 1:1 relationship, with LEmail being the foreign key.
- You have learned E/R modelling, this is not worth splitting.

# Transitive Dependency and FK

LecID (A) → LEmail (B) → LName (C)

- The confusion comes from the name “transitive”.
- The above dependency chain is indeed “transitive”.
- But it is not the type of “transitive” we are looking for.
- The type of transitive dependencies we are looking for should support **relationships, or foreign keys**.

# Normalisation: Practice

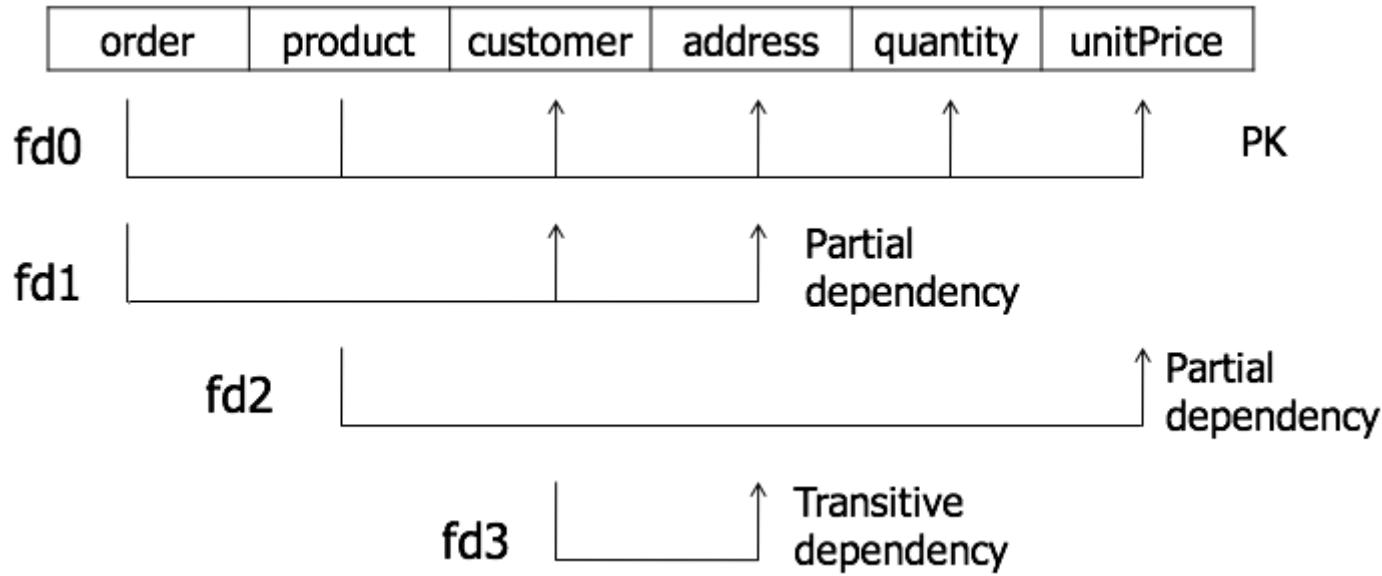
# Normalisation Example

- We have a table representing orders in an online store, Each entry in the table represents an item on a particular order.  
{order, product, customer, address, quantity, unitPrice}
- For example: (001, Laptop, Jianjun, SEB435 UNNC, 1, \$500)
- Primary key is {order, product}
- No other candidate key
- Task: Normalise it to 3NF.

# Functional Dependencies

- Each order is for a single customer and each customer has a single address.
  - FD1:  $\{\text{order}\} \rightarrow \{\text{customer}, \text{address}\}$
  - FD2: see FD 4 below
- Each product has a single price
  - FD3:  $\{\text{product}\} \rightarrow \{\text{unitPrice}\}$
- Each order transitively determines address via customer.
  - FD4:  $\{\text{order}\} \rightarrow \{\text{customer}\} \rightarrow \{\text{address}\}$

# Functional Dependencies



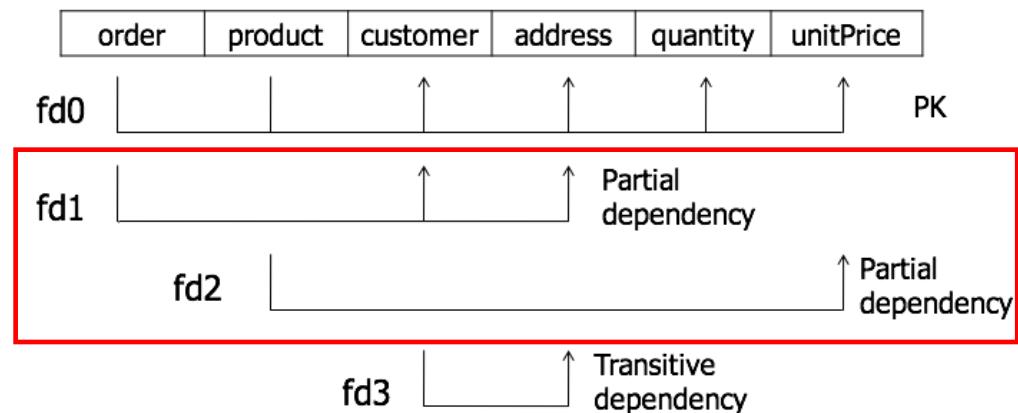
# Normalisation to 2NF

2NF: no partial dependencies on candidate keys

$\{\text{order}\} \rightarrow \{\text{customer}, \text{address}\}$

$\{\text{product}\} \rightarrow \{\text{unitPrice}\}$

- To remove the first FD we move  $\{\text{customer}, \text{address}\}$  to another relation, along with a copy of its determinant, *order*.
  - R1:  $\{\underline{\text{order}}, \text{customer}, \text{address}\}$
- With remaining relation
  - R2:  $\{\underline{\text{order}}, \underline{\text{product}}, \text{quantity}, \text{unitPrice}\}$



# Normalisation to 2NF

## Current Tables

R1: {order, customer, address} 2NF

R2: {order, product, quantity, unitPrice}

Next:



- There is a partial FD in R2: {Product}  $\rightarrow$  {UnitPrice}
- To remove this we move it to another relation R3
  - R3: {Product, unitPrice}
- and with remaining relation
  - R4: {order, product, quantity}

# Normalisation to 3NF

Current Tables:

R1: {order, customer, address}

R3: {Product, unitPrice} 3NF

R4: {order, product, quantity} 3NF

Next:

- R1 has a transitive FD on its key
- To remove  $\{order\} \rightarrow \{customer\} \rightarrow \{Address\}$
- we decompose R1 over
  - R5: {order, customer}
  - R6: {customer, address}

# Normalisation

- 1NF:
  - $\{\underline{\text{order}}, \underline{\text{product}}, \text{customer}, \text{address}, \text{quantity}, \text{unitPrice}\}$
- 2NF:
  - R1:  $\{\underline{\text{order}}, \text{customer}, \text{address}\}$  2NF
  - R3:  $\{\underline{\text{product}}, \text{unitPrice}\}$  3NF
  - R4:  $\{\underline{\text{order}}, \underline{\text{product}}, \text{quantity}\}$  3NF
- 3NF:
  - R3:  $\{\underline{\text{product}}, \text{unitPrice}\}$  3NF
  - R4:  $\{\underline{\text{order}}, \underline{\text{product}}, \text{quantity}\}$ , 3NF
  - R5:  $\{\underline{\text{order}}, \text{customer}\}$  3NF
  - R6:  $\{\underline{\text{customer}}, \text{address}\}$  3NF

# Boyce-Codd Normal Form

Jianjun Chen

# Previous Lecture

- In the previous lecture we demonstrated how 2NF and 3NF disallow partial and transitive dependencies **on the primary key** of a relation, respectively.
- Relations that have these types of dependencies may suffer from the update anomalies

# In This Lecture

- More normalisation
  - Lossless decomposition
  - Boyce-Codd normal form (BCNF)
  - Higher normal forms
  - Denormalisation
- For more information
  - Connolly and Begg Chapter 15

# Lossless Decomposition

- To normalise a relation, we used projections
- If  $R(A,B,C)$  satisfies  $A \rightarrow B$  then we can project it on  $A,B$  and  $A,C$  without losing information

- Lossless decomposition:

$$R = \pi_{AB}(R) \bowtie \pi_{AC}(R)$$

where  $\pi_{AB}(R)$  is projection of  $R$  on  $AB$  and  $\bowtie$  is natural join.

$R$	$\pi_{AB}(R)$										
<table border="1"><thead><tr><th>A</th><th>B</th><th>C</th></tr></thead><tbody><tr><td></td><td></td><td></td></tr></tbody></table>	A	B	C				<table border="1"><thead><tr><th>A</th><th>B</th></tr></thead><tbody><tr><td></td><td></td></tr></tbody></table>	A	B		
A	B	C									
A	B										

# Example of Lossless Decomposition

Assume that a module is taught by one lecturer

**R**

Module	Lecturer	Text
DBS	rzb	CB
DBS	rzb	UW
RDB	nza	UW
APS	rcb	B

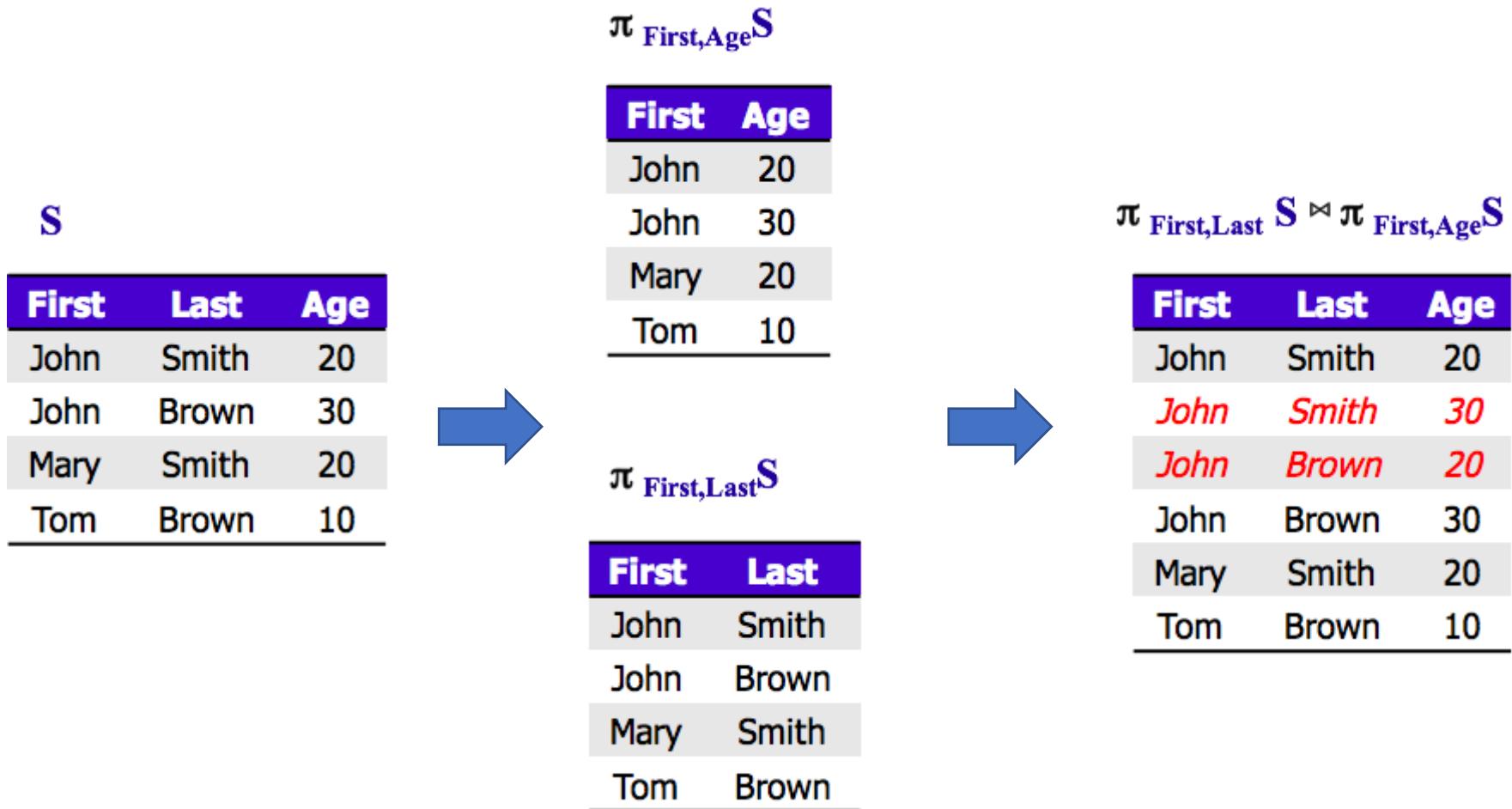
$\pi_{\text{Module}, \text{Lecturer}}(R)$

Module	Lecturer
DBS	rzb
RDB	nza
APS	rcb

$\pi_{\text{Module}, \text{Text}}(R)$

Module	Text
DBS	CB
DBS	UW
RDB	UW
APS	B

# When is Decomposition not Lossless: No FD



# Question:

- Is the normalisation process from 1NF to 3NF lossless or lossy?
- If we break a FD  $A \rightarrow B$  by splitting them into two separate tables {A} and {B}, will it be lossy or lossless?

# Boyce-Codd Normal Form

# The Stream Relation

Student	Course	Time
John	Databases	12:00
Mary	Databases	12:00
Richard	Databases	15:00
Richard	Programming	10:00
Mary	Programming	10:00
Rebecca	Programming	13:00

- Consider a relation, **Stream**, which stores information about times for various streams of courses (Online teaching)
- Assume:
  - Each course has several streams
  - Only one stream (of any course at all) takes place at any given time
  - Each (student, course) pair is assigned to a single stream for it. That is, (student, course) is the primary key.

# The Stream Relation

Student	Course	Time
John	Databases	12:00
Mary	Databases	12:00
Richard	Databases	15:00
Richard	Programming	10:00
Mary	Programming	10:00
Rebecca	Programming	13:00

Candidate keys: {Student, Course} and {Student, Time}

What are the functional dependencies?

# FDs in the Stream Relation

- Stream has the following non-trivial FDs

$\{\text{Student}, \text{Course}\} \rightarrow \{\text{Time}\}$

$\{\text{Student}, \text{Time}\} \rightarrow \{\text{Course}\}$

$\{\text{Time}\} \rightarrow \{\text{Course}\}$

- Trivial FD:  $(A, B) \rightarrow B$

- $B$  is a subset of  $(A, B)$ .

- non-trivial FD:  $A \rightarrow B$

- $B$  is not a subset of  $A$

- Is Stream table in 3NF?

# Anomalies in Stream

- INSERT anomalies:
  - You can't add an empty stream (a stream with no students)
- Modification anomalies:
  - Moving the 12:00 class to 9:00 means changing two rows
- DELETE anomalies
  - Deleting Rebecca removes a stream

Student	Course	Time
John	Databases	12:00
Mary	Databases	12:00
Richard	Databases	15:00
Richard	Programming	10:00
Mary	Programming	10:00
Rebecca	Programming	13:00

# Boyce-Codd Normal Form

“A relation is in BCNF, if and only if, every determinant is a candidate key.”

- The difference between 3NF and BCNF is that for a functional dependency  $A \rightarrow B$ , The process of normalising to 3NF does not care whether the determinant A is a candidate key or not.
- whereas BCNF insists that for this dependency to remain in a relation, A must be a candidate key.

# Stream and BCNF

- Stream is not in BCNF as the FD  $\{Time\} \rightarrow \{Course\}$  is non-trivial and  $\{Time\}$  is not a candidate key

Student	Course	Time
John	Databases	12:00
Mary	Databases	12:00
Richard	Databases	15:00
Richard	Programming	10:00
Mary	Programming	10:00
Rebecca	Programming	13:00

# Conversion to BCNF

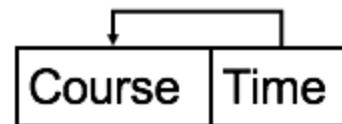
$\{\text{Student, Course}\} \rightarrow \{\text{Time}\}$

$\{\text{Student, Time}\} \rightarrow \{\text{Course}\}$

$\{\text{Time}\} \rightarrow \{\text{Course}\}$

Student	Course	Time
---------	--------	------

Student	Course
---------	--------



Stream has been put into BCNF but we have lost the FD  
 $\{\text{Student, Course}\} \rightarrow \{\text{Time}\}$

# BCNF & Decomposition Properties

- In relation {A, B, C} with primary key (A, B), we have functional dependency  $(A, B) \rightarrow C$ .
- If we also have FD  $C \rightarrow B$ , and C cannot be considered as a candidate key in the relation, then this relation is in 3NF but not in BCNF. Stream table shows that such a relation has update anomalies.
  - We have FD  $(A, B) \rightarrow C \rightarrow B$ , which looks like a transitive dependency, but  $(A, B) \rightarrow B$  is a trivial FD.
  - That's the reason why this table is in 3NF.

# BCNF & Decomposition Properties

- To convert  $\{A, B, C\}$  into BCNF, split it into  $\{A, B\}$  and  $\{C, B\}$ 
  - But we lose FD  $(A, B) \rightarrow C$ . We can no longer reproduce such relationship.
- As a result, BCNF in this case is **lossy**.

# Decomposition Properties

1. Lossless: Data should not be lost or created when splitting relations up
  2. Dependency preservation: It is desirable that FDs are preserved when splitting relations up
- Normalisation to BCNF may not preserve all dependencies.
    - As a result, BCNF is NOT always necessary.
    - Depends on whether anomalies are acceptable or not.

# Another Example

Student	Course	Time
John	Databases	12:00
Mary	Databases	12:00
Richard	Databases	15:00
Richard	Programming	10:00
Mary	Programming	10:00
Rebecca	Programming	13:00

- What if (student, time) is the primary key?

FD1: {Student, Course} → {Time}

FD2: {Student, Time} → {Course}

FD3: {Time} → {Course}

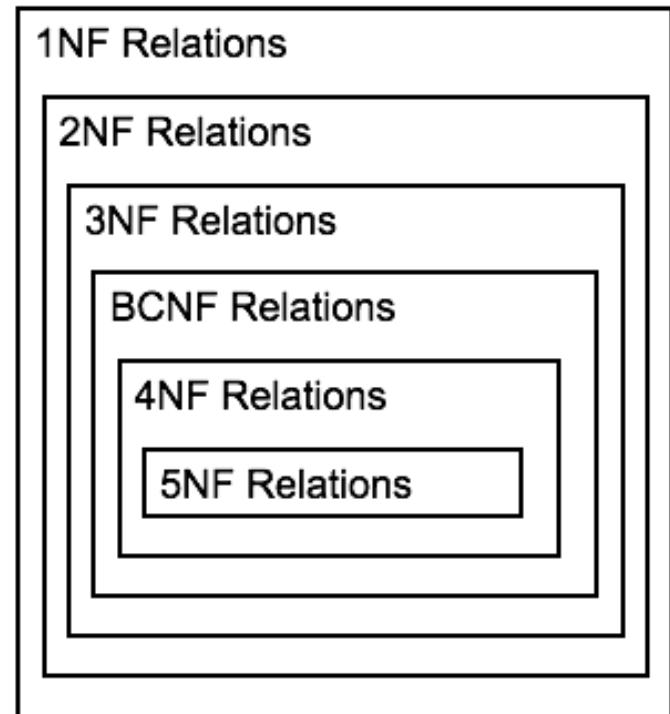
- In FD2, Course is partially dependent on the primary key. The table is not in 2NF.
- Splitting the table to {Time, Course} and {Student, Time} results in a lossy decomposition.
  - FD {Student, Course} → {Time} is broken
  - You can check the result by yourself.
- Normalisation to 3NF is **NOT** always lossless and dependency preserving

# Higher Normal Forms

- Violation of BCNF is quite rare.
- The potential to violate BCNF may occur in a relation that:
  - contains two (or more) composite candidate keys;
  - the candidate keys overlap, that is have at least one attribute in common.

# Higher Normal Forms

- BCNF is as far as we can go with FDs
  - Higher normal forms are based on other sorts of dependency
  - Fourth normal form removes multi-valued dependencies
  - Fifth normal form removes join dependencies



# Denormalisation

# Denormalisation

- Normalisation
  - Removes data redundancy
  - Solves INSERT, UPDATE, and DELETE anomalies
  - This makes it easier to maintain the information in the database in a consistent state
- However
  - It leads to more tables in the database
  - Often these need to be joined back together, which is expensive to do
  - So sometimes (not often) it is worth ‘denormalising’

# Denormalisation

- You might want to denormalise if
  - Database speeds are unacceptable (not just a bit slow)
  - There are going to be very few INSERTs, UPDATEs, or DELETEs
  - There are going to be lots of SELECTs that involve the joining of tables

Address			
Number	Street	City	Postcode

Not normalised since  
 $\{Postcode\} \rightarrow \{City\}$

Address1		
Number	Street	Postcode

Address2	
Postcode	City

# Transactions and Recovery

Jianjun Chen

# Contents

- Transactions
- Recovery
  - System and Media Failures
- Concurrency
  - Concurrency problems
- For more information:
  - Connolly and Begg chapter 22

# Transactions: Definition

- A transaction is an action, or a series of actions, carried out by a single user or an application program, which reads or updates the contents of a database.

# Transaction Support In MySQL

```
1 START TRANSACTION  
2     [transaction_characteristic [, transaction_characteristic] ...]  
3  
4 transaction_characteristic: {  
5     WITH CONSISTENT SNAPSHOT  
6     | READ WRITE  
7     | READ ONLY  
8 }  
9  
10 BEGIN [WORK]  
11 COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]  
12 ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]  
13 SET autocommit = {0 | 1}
```

# Transaction Support In MySQL

```
1 START TRANSACTION;
2 SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
3 UPDATE table2 SET summary=@A WHERE type=1;
4 COMMIT;
```

- More information can be found at :

<https://dev.mysql.com/doc/refman/8.0/en/commit.html>

# Transactions

- A transaction is a ‘logical unit of work’ on a database
  - Each transaction does something in the database
  - No part of it alone achieves anything of use or interest
- Transactions are the unit of recovery, consistency, and integrity as well
- ACID properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability

# Example of transaction

Transfer £50 from account A to account B:

1. Read(A)
2.  $A = A - 50$
3. Write(A)
4. Read(B)
5.  $B = B + 50$
6. Write(B)

- **Atomicity** - shouldn't take money from A without giving it to B
- **Consistency** - money isn't lost or gained
- **Isolation** - other queries shouldn't see A or B change until completion
- **Durability** - the money does not go back to A

# Atomicity and Consistency

- Atomicity:
  - Transactions are atomic: they don't have parts (conceptually)
  - So a transaction can't be executed partially.
  - "One transaction should be treated as one single action"
- Consistency
  - Transactions take the database from one **consistent state** into another.
  - In the middle of a transaction the database might not be consistent

# Isolation and Durability

- Isolation
  - The effects of a transaction are not visible to other transactions until it has completed
  - From outside the transaction has either happened or not
  - To me this actually sounds like a consequence of atomicity...
- Durability
  - Once a transaction has completed, its changes are made permanent
  - Even if the system crashes, the effects of a transaction must remain in place

# The Transaction Manager

The transaction manager enforces the ACID properties

- It schedules the operations of transactions
- COMMIT and ROLLBACK are used to ensure atomicity
- Locks are used to ensure consistency and isolation for concurrent transactions
- A log is kept to ensure durability in the event of system failure

# COMMIT and ROLLBACK

- COMMIT signals the successful end of a transaction
  - Any changes made by the transaction should be saved
  - These changes are now visible to other transactions
  - Writes the data change in the memory to the DB file.
- ROLLBACK signals the unsuccessful end of a transaction
  - Any changes made by the transaction should be undone
  - It is now as if the transaction never existed

# Recovery



- Transactions should be durable, but we cannot prevent all sorts of failures:
  - System crashes
  - Power failures
  - Disk crashes
  - User mistakes
  - Sabotage
  - Natural disasters
- Prevention is better than cure
  - Reliable OS
  - Security
  - UPS and surge protectors
  - RAID arrays
- However, we cannot protect against everything

# Media Failures

- System failures are not too severe
  - Only information since the last checkpoint is affected
  - This can be recovered from the transaction log
- Media failures (disk crashes etc) are more serious
  - The data stored to disk is damaged
  - The transaction log itself may be damaged

# Backups

- Backups are needed to recover from media failure
  - The transaction log and entire contents of the database is written to secondary storage (often tape)
  - Time consuming, and often requires down time
- Backups frequency
  - Frequent enough that little information is lost
  - Not so frequent as to cause problems
  - Every day (night) is common
- Backup storage

# Recovery from Media Failure

1. Restore the database from the last backup
2. Use the transaction log to redo any changes made since the last backup

If the transaction log is damaged, you can't do step 2

- Store the log on a separate physical device to the database
- The risk of losing both is then reduced

# Concurrency

- Large databases are used by many people
  - Many transactions to be run on the database
  - It is desirable to let them run at the same time as each other
  - Need to preserve isolation
- If we don't allow for concurrency then transactions are run sequentially
  - Have a queue of transactions
  - Long transactions (e.g. backups) will make others wait for long periods

# Concurrency Problems

- In order to run transactions concurrently we interleave their operations
- Each transaction gets a share of the computing time
- This leads to several sorts of problems
  - Lost updates
  - Uncommitted updates
  - Incorrect analysis
- All arise because isolation is broken

# Lost Update

T1	T2
<b>Read(X)</b>	
<b>X = X - 5</b>	
<b>Write(X)</b>	
<b>COMMIT</b>	
	<b>Read(X)</b>
	<b>X = X + 5</b>
	<b>Write(X)</b>
	<b>COMMIT</b>

- T1 and T2 read X, both modify it, then both write it out
  - The net effect of T1 and T2 should be no change on X
  - Only T2's change is seen, however, so the final value of X has increased by 5

# Uncommitted Update

T1	T2
<b>Read(X)</b>	
<b>X = X - 5</b>	
<b>Write(X)</b>	
	<b>Read(X)</b>
	<b>X = X + 5</b>
	<b>Write(X)</b>
<b>ROLLBACK</b>	<b>COMMIT</b>

- T2 sees the change to X made by T1, but T1 is rolled back
  - The change made by T1 is undone on rollback
  - It should be as if that change never happened

# Inconsistent analysis

T1	T2
<b>Read (X)</b> $X = X - 5$ <b>Write (X)</b>	<b>Read (X)</b> <b>Read (Y)</b> $Sum = X+Y$
<b>Read (Y)</b> $Y = Y + 5$ <b>Write (Y)</b>	

- T1 doesn't change the sum of X and Y, but T2 sees a change
  - T1 consists of two parts
    - take 5 from X and then add 5 to Y
  - T2 sees the effect of the first, but not the second

# Concurrency

Concurrency control, Serialisability, Locks, Deadlocks

# Schedules

- A **schedule** is a sequence of the operations by a set of concurrent transactions that preserves the order of operations in each of the individual transactions
  - A **serial schedule** is a schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions (each transaction commits before the next one is allowed to begin)
  - Non-serial schedule: operations from transactions are interleaved

# Serialisability

- A non-serial schedule is **serialisable** if it produces the same results as some serial execution.
- The objective of serializability is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another

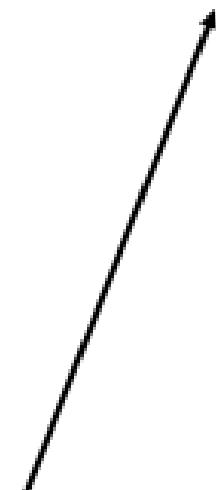
# Serial and Serializable

## Interleaved Schedule

T1 Read(X)  
T2 Read(X)  
T2 Read(Y)  
T1 Read(Z)  
T1 Read(Y)  
T2 Read(Z)

## Serial Schedule

T2 Read(X)  
T2 Read(Y)  
T2 Read(Z)  
  
T1 Read(X)  
T1 Read(Z)  
T1 Read(Y)



This schedule is serialisable:

# Order of Read and Write

- If two transactions only read a data item, they do not conflict and order is not important.
- If two transactions either read or write completely separate data items, they do not conflict and order is not important.
- If one transaction writes a data item and another either reads or writes the same data item, **the order of execution** is important (for preventing interference). They have **conflict**

# Conflict Serialisability

A schedule is **conflict serialisable** if transactions in the schedule have a conflict but the schedule is still serializable (equivalent to some serial schedule).

Previous definitions:

- Schedule: sequence of operations from multiple transactions.
- Serial schedule: One transaction after another.
- Serializable: interleaved but still has same effect as serial schedule.

# Conflict Serialisable Schedule

Interleaved Schedule

T1 Read(X)  
T1 Write(X)  
T2 Read(X)  
T2 Write(X)  
T1 Read(Y)  
T1 Write(Y)  
T2 Read(Y)  
T2 Write(Y)

Serial Schedule

T1 Read(X)  
T1 Write(X)  
T1 Read(Y)  
T1 Write(Y)  
  
T2 Read(X)  
T2 Write(X)  
T2 Read(Y)  
T2 Write(Y)

This schedule is serialisable,  
even though T1 and T2 read  
and write the same resources  
X and Y: they have a conflict

# Conflict Serialisability

- Conflict serialisable schedules are the main focus of concurrency control
- They allow for interleaving and at the same time they are guaranteed to behave as a serial schedule
- Important questions:
  - Given a schedule, how to determine whether a schedule is conflict serialisable?
  - How to construct conflict serialisable schedules?

# Precedence Graphs

To determine if a schedule is conflict serializable, we use a precedence graph.

Steps:

1. Create a node for each transaction.
2. Create a directed edge  $T_i \rightarrow T_j$ 
  - If  $T_j$  reads the value of an item written by  $T_i$ .
  - If  $T_j$  writes a value into an item after it has been read by  $T_i$ .
  - If  $T_j$  writes a value into an item after it has been written by  $T_i$ .

# Precedence Graphs

- If an edge  $T_i \rightarrow T_j$  exists in the precedence graph for  $S$ , then in any serial schedule  $S'$  equivalent to  $S$ ,  $T_i$  must appear before  $T_j$ .
- If the precedence graph contains a cycle the schedule is not conflict serializable.
- The precedence graph can help us identify whether lock is needed for a certain resource. Lock prevents other transaction from accessing the same resource.

# Precedence Graph Example

- The lost update schedule has the precedence graph:

T1 Write(X) followed by T2 Write(X)



T2 Read(X) followed by T1 Write(X)

T1	T2
<b>Read(X)</b>	
<b>X = X - 5</b>	
	<b>Read(X)</b>
	<b>X = X + 5</b>
<b>Write(X)</b>	
<b>Commit</b>	
	<b>Write(X)</b>
	<b>Commit</b>

# Precedence Graph Example

- No cycles: conflict  
serialisable schedule

T1 reads X before T2 writes X and  
T1 writes X before T2 reads X and  
T1 writes X before T2 writes X



T1	T2
<b>Read(X)</b> <b>Write(X)</b>	<b>Read(X)</b> <b>Write(X)</b>

# Precedence Graph Explanation

- If we have  $T_1 \rightarrow T_2$ , the schedule requires that  $T_2$  should be done after  $T_1$ .
- If we have  $T_2 \rightarrow T_1$ , the schedule requires that  $T_1$  should be done after  $T_2$ .
- If both edges exists, that's a conflict on interleaving that we cannot solve.
  - If  $T_1$  is requested before  $T_2$ , then the precedence of  $T_2$

# Locking

# Locking

- Locking is a procedure used to control concurrent access to data (to ensure serialisability of concurrent transactions)
- In order to use a ‘resource’ (table, row, etc) a transaction must first acquire a lock on that resource
- This may deny access to other transactions to prevent incorrect results

# Two types of locks

- Two types of lock
  - Shared lock (S-lock or read-lock)
  - Exclusive lock (X-lock or write-lock)
- Read lock allows several transactions simultaneously to read a resource
  - but no transactions can change it at the same time
- Write lock allows one transaction exclusive access to write to a resource.
  - No other transaction can read this resource at the same time.
- The lock manager in the DBMS assigns locks and records them in the data dictionary

# Locking

- Before reading from a resource a transaction must acquire a **read-lock**
- Before writing to a resource a transaction must acquire a **write-lock**
- Locks are released on commit/rollback

# Locking

- A transaction may not acquire a lock on any resource that is write-locked by another transaction
- A transaction may not acquire a write-lock on a resource that is locked by another transaction
- If the requested lock is not available, transaction waits

# Two-Phase Locking

- A transaction follows the ***two-phase locking protocol (2PL)*** if all locking operations precede the first unlock operation in the transaction:
- **Growing phase** where locks are acquired on resources
- **Shrinking phase** where locks are released

# Example

- T1 follows 2PL protocol
  - All of its locks are acquired before it releases any of them
- T2 does not
  - It releases its lock on X and then goes on to later acquire a lock on Y

T1	T2
read-lock(X)	read-lock(X)
Read(X)	Read(X)
write-lock(Y)	unlock(X)
unlock(X)	write-lock(Y)
Read(Y)	Read(Y)
$Y = Y + X$	$Y = Y + X$
Write(Y)	Write(Y)
unlock(Y)	unlock(Y)

# Serialisability Theorem

Any schedule of two-phased transactions is conflict serialisable

# Lost Update can't happen with 2PL

T1	T2
read-lock(X)  cannot acquire write-lock(X): T2 has read- lock(X)	<b>Read (X)</b> $X = X - 5$  <b>Write (X)</b>  <b>COMMIT</b>

read-lock(X)	<b>Read (X)</b> $X = X + 5$  <b>Write (X)</b>  <b>COMMIT</b>
--------------	---

# Uncommitted Update cannot happen with 2PL

	T1	T2	
read-lock(X)	<b>Read (X)</b>		
write-lock(X)	<b>X = X - 5</b> <b>Write (X)</b>	<b>Read (X)</b> <b>X = X + 5</b> <b>Write (X)</b>	Waits till T1 releases write-lock(X)
Locks released	<b>ROLLBACK</b>	<b>COMMIT</b>	

# Inconsistent analysis cannot happen with 2PL

	T1	T2
read-lock(X)	<b>Read (X)</b>	
	<b>X = X - 5</b>	
write-lock(X)	<b>Write (X)</b>	<b>Read (X)</b>
		<b>Read (Y)</b>
read-lock(Y)	<b>Read (Y)</b>	<b>Sum = X+Y</b>
	<b>Y = Y + 5</b>	
write-lock(Y)	<b>Write (Y)</b>	

Waits till T1 releases write-locks on X and Y

Question: What will happen if Read(Y) from T2 is before Read(x)?

# Locks in MySQL

- <https://www.oreilly.com/library/view/mysql-reference-manual/0596002653/ch06s07.html>

# Deadlocks

And Prevention Methods.

# Deadlocks

- A deadlock is an impasse that may result when two or more transactions are waiting for locks to be released which are held by each other.
  - For example:
    - T1 has a lock on X and is waiting for a lock on Y, and
    - T2 has a lock on Y and is waiting for a lock on X.
- Given a schedule, we can detect deadlocks which will happen in this schedule using a **wait-for graph (WFG)**.

# Precedence/Wait-For Graphs

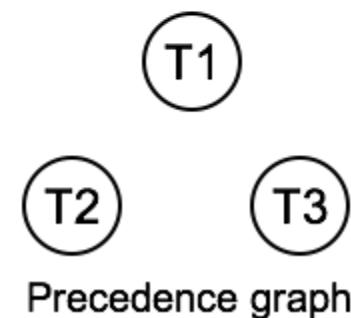
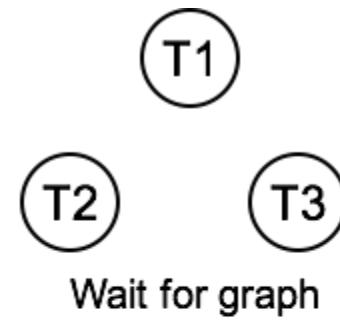
## Creating Wait-for Graph

- Each transaction is a vertex
- Arcs from  $T_1$  to  $T_2$  if  $T_1$  is waiting to lock an item that is currently locked by  $T_2$ .

Deadlock exists if and only if the WFG contains a circle.

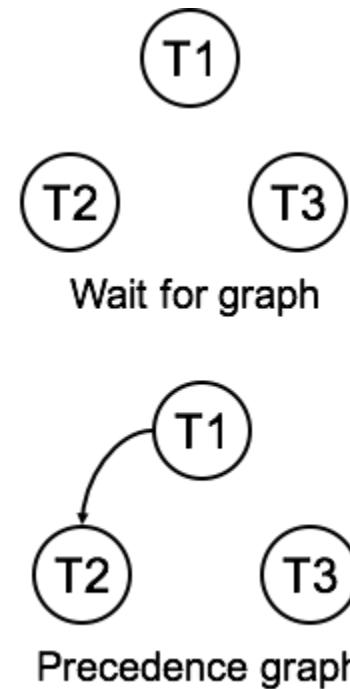
# Example: Precedence Graph

- T1 Read(X)
- T2 Read(Y)
- T1 Write(X)
- T2 Read(X)
- T3 Read(Z)
- T3 Write(Z)
- T1 Read(Y)
- T3 Read(X)
- T1 Write(Y)



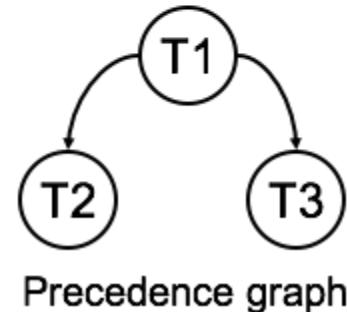
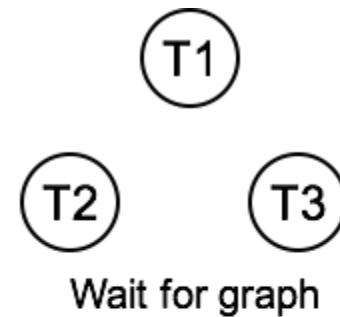
# Example: Precedence Graph

- T1 Read(X)
- T2 Read(Y)
- T1 Write(X)
- T2 Read(X)
- T3 Read(Z)
- T3 Write(Z)
- T1 Read(Y)
- T3 Read(X)
- T1 Write(Y)



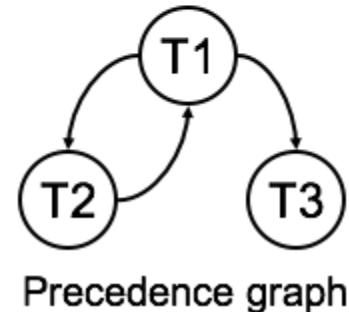
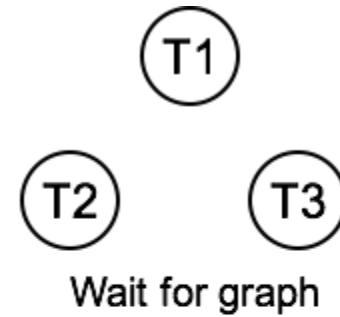
# Example: Precedence Graph

- T1 Read(X)
- T2 Read(Y)
- **T1 Write(X)**
- T2 Read(X)
- T3 Read(Z)
- T3 Write(Z)
- T1 Read(Y)
- **T3 Read(X)**
- T1 Write(Y)



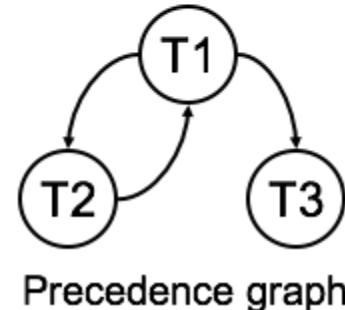
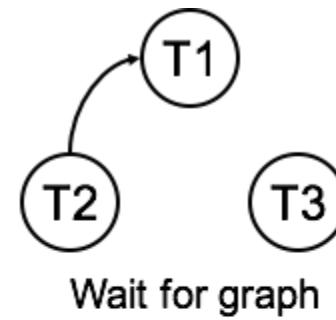
# Example: Precedence Graph

- T1 Read(X)
- T2 Read(Y)
- T1 Write(X)
- T2 Read(X)
- T3 Read(Z)
- T3 Write(Z)
- T1 Read(Y)
- T3 Read(X)
- T1 Write(Y)



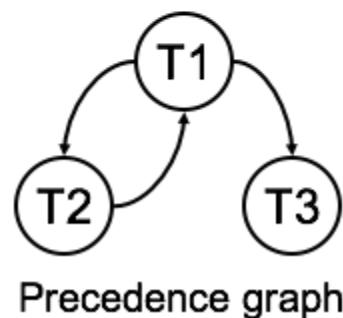
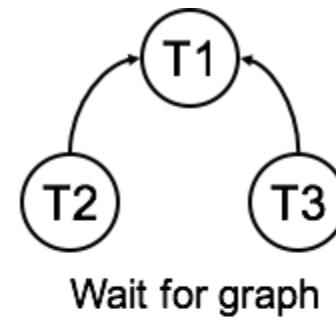
# Example: Wait-for Graph

- T1 Read(X) read-locks(X)
- T2 Read(Y) read-locks(Y)
- T1 Write(X) write-lock(X)
- T2 Read(X) tries read-lock(X)
- T3 Read(Z)
- T3 Write(Z)
- T1 Read(Y)
- T3 Read(X)
- T1 Write(Y)



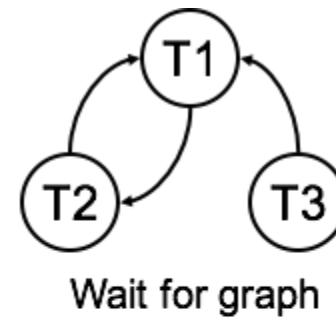
# Example: Wait-for Graph

- T1 Read(X) read-locks(X)
- T2 Read(Y) read-locks(Y)
- T1 Write(X) write-lock(X)
- T2 Read(X) tries read-lock(X)
- T3 Read(Z) read-lock(Z)
- T3 Write(Z) write-lock(Z)
- T1 Read(Y) read-lock(Y)
- T3 Read(X) tries read-lock(X)
- T1 Write(Y)

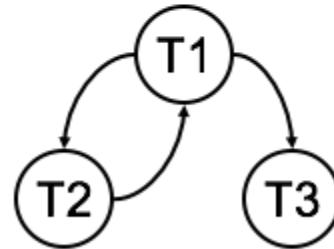


# Example: Wait-for Graph

- T1 Read(X) read-locks(X)
- T2 Read(Y) read-locks(Y)
- T1 Write(X) write-lock(X)
- T2 Read(X) tries read-lock(X)
- T3 Read(Z) read-lock(Z)
- T3 Write(Z) write-lock(Z)
- T1 Read(Y) read-lock(Y)
- T3 Read(X) tries read-lock(X)
- T1 Write(Y) tries write-lock(Y)



Wait for graph



Precedence graph

# Deadlock Prevention

- Deadlocks can arise with 2PL
  - Deadlock is less of a problem than an inconsistent DB
  - We can detect and recover from deadlock
  - It would be nice to avoid it altogether
- Conservative 2PL
  - All locks must be acquired before the transaction starts
  - Hard to predict what locks are needed
  - Low ‘lock utilisation’ - transactions can hold on to locks for a long time, but not use them much

# Deadlock Prevention: Resource Reordering

- We impose an ordering on the resources
  - Transactions must acquire locks in this order
  - Transactions can be ordered on the last resource they locked
- This prevents deadlock
  - If T1 is waiting for a resource from T2 then that resource must come after all of T1's current locks
  - All the arcs in the wait-for graph point 'forwards' - no cycles

# Example of resource ordering

- Suppose resource order is:
  - $X < Y$
- This means, if you need locks on X and Y, you first acquire a lock on X and only after that a lock on Y
  - even if you want to write to Y before doing anything to X
- It is impossible to end up in a situation when T1 is waiting for a lock on X held by T2, and T2 is waiting for a lock on Y held by T1.