Informatics 1, 2009

School of Informatics, University of Edinburgh

Data and Analysis

Part I

Structured Data

Alex Simpson

Part I — Structured data

- For some application domains, data is *inherently structured*
  - For instance, all students share common information
- In such domains, it makes sense to organise the data in a way that directly maps to their *physical properties*, and to devise mechanisms to access and manipulate data
- We will deal with two main *data representation* models:
  - The *entity-relationship (ER)* model, and the *relational* model
- Finally, we will deal with data *manipulation* for the *relational model*, in particular:
  - *Relational algebra*, the *Tuple-relational calculus* and the query language *SQL*

Part I — Structured Data

Data Representation:

I.1 The entity-relationship (ER) data model

I.2 The relational model

Data Manipulation:

I.3 Relational algebra

I.4 Tuple relational calculus

I.5 The SQL query language

Required reading

You are required to read Chapter 2 of:

[DMS] R. Ramakrishnan and J. Gehrke
Database Management Systems
McGraw-Hill, Third Edition, 2003.

In particular, §§ 2.1–2.5.

Initial stages of database design

1. Requirements analysis.

   Understand what data is to be stored in the database and what operations are likely to be performed on it.

2. Conceptual design

   Develop a high-level description of data to be stored and constraints that hold over it.

   This description is often given using the ER data model.

3. Logical design

   Implement the conceptual design by mapping it to a *logical data representation*. The outcome is a *logical schema*.

   The implementation is often performed by translating the ER data model into a *relational database schema* (see I.2).

The ER data model

- What is it used for?

  The ER model is a way to describe *entities* (for example, real-world entities) and the *relationships* between them

- Why is it useful?

  Because it maps to different *logical data models*, including the relational model

- How is it used?

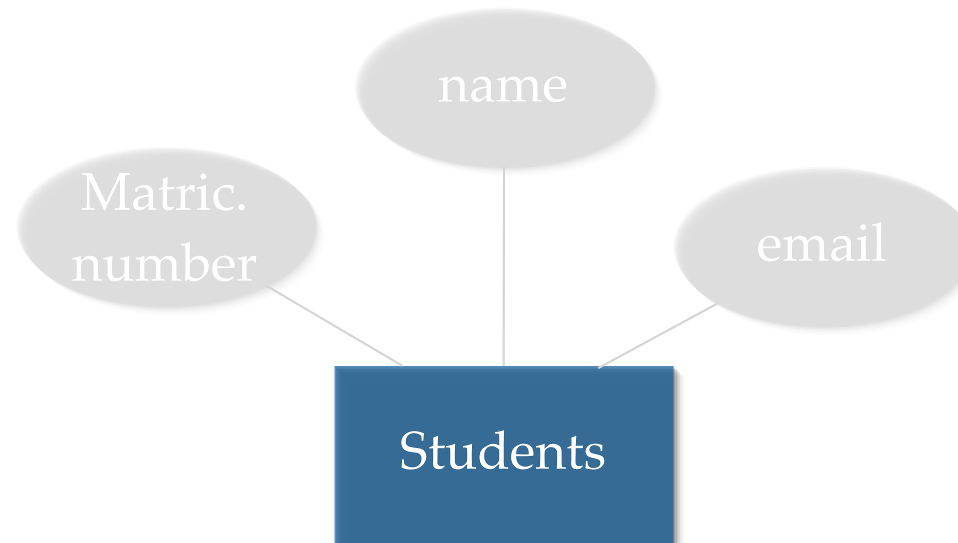
  It is essentially a way to visualise data and their dependencies

Entities and entity sets

Any distinguishable object (for example, in the real world) can be an *entity*

A collection of the same type of entities is an *entity set*

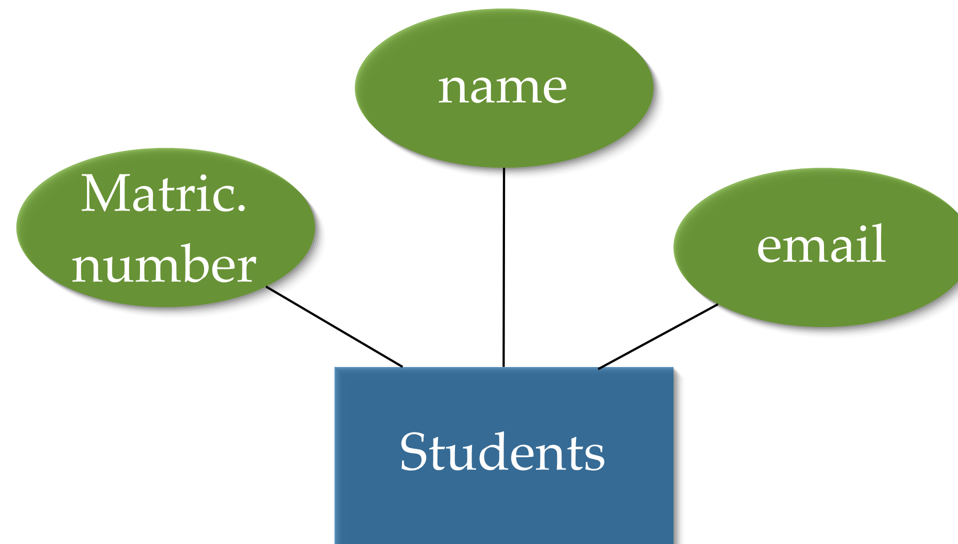
Entity sets are represented by *boxes*, labelled with the entity set's name



Attributes

Each entity of the same entity set has some characteristic *attributes*

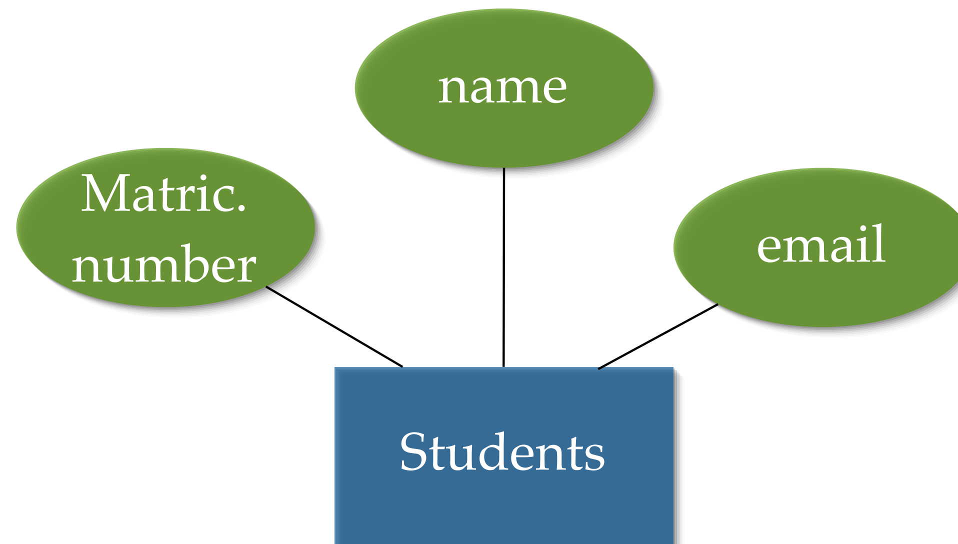
Attributes are represented by *ovals*, labelled with the attribute's name, connected to the entity set they belong to.



Domains

Each attribute has a *domain* from which allowable values are derived

E.g., Matric. number is an *integer*
name and email are *40-character strings*

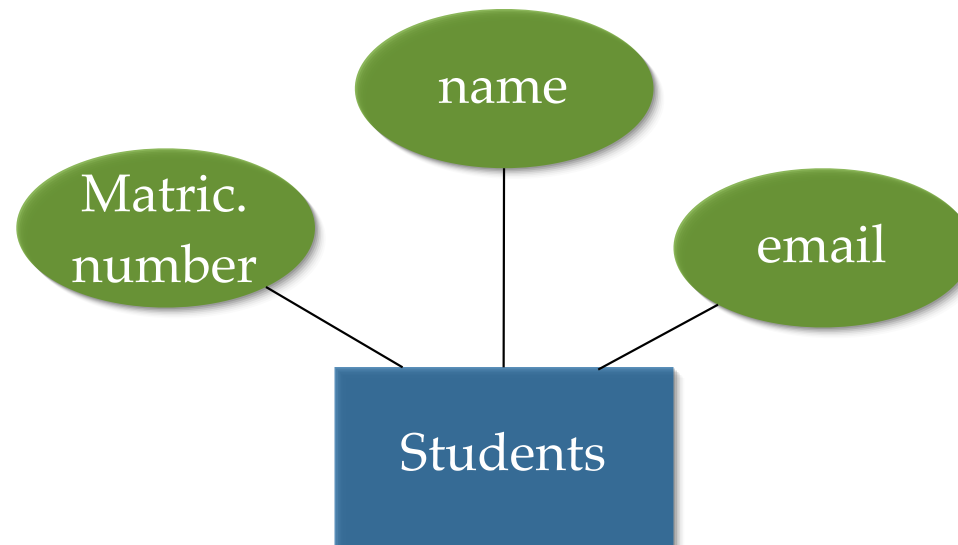


Keys

A *key* is a minimal set of attributes whose values allow us to uniquely identify an entity in an entity set

There may be more than one such minimal set, they are called *candidate keys*

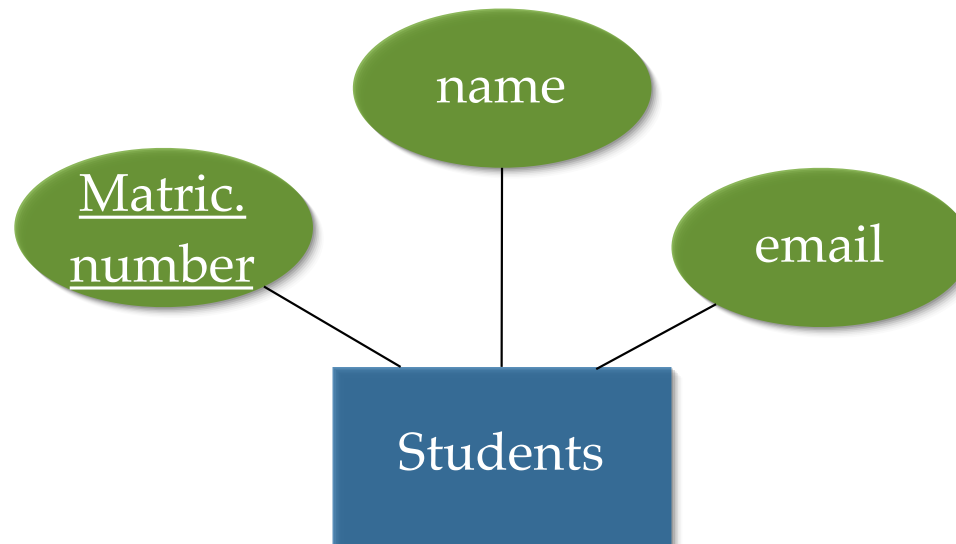
E.g., either Matric. number or email can act as keys.



Primary keys

If multiple candidate keys exist, we choose one and make it the *primary key*.

The attributes occurring in the primary key are *underlined* in the ER diagram



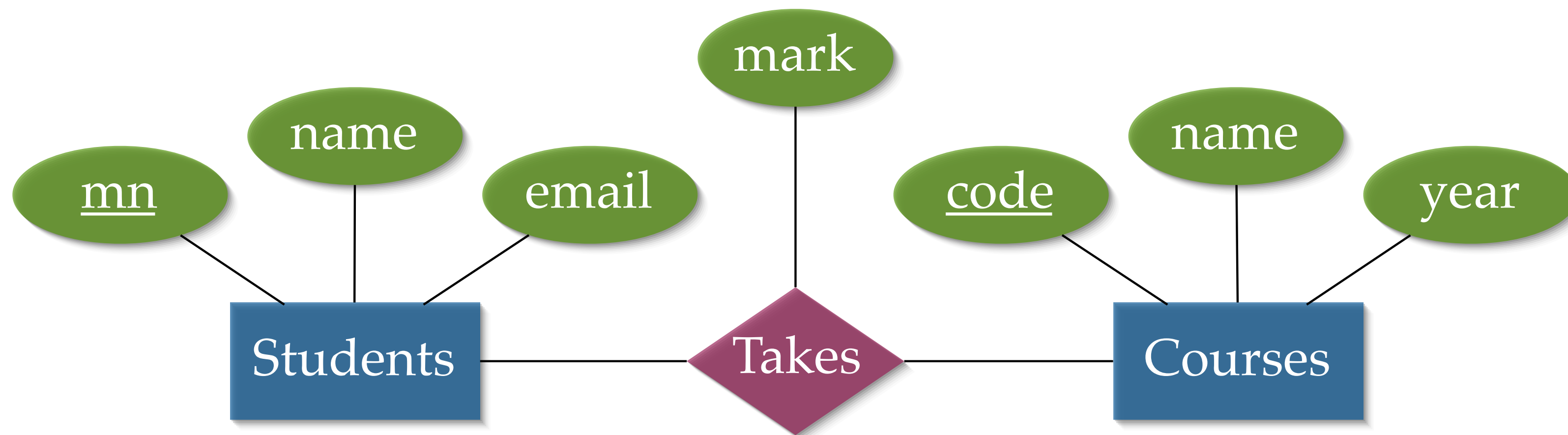
Relationships and relationship sets

Relationships model associations between entities

Relations are grouped into *relationship sets* of relationships between entities from specified entity sets.

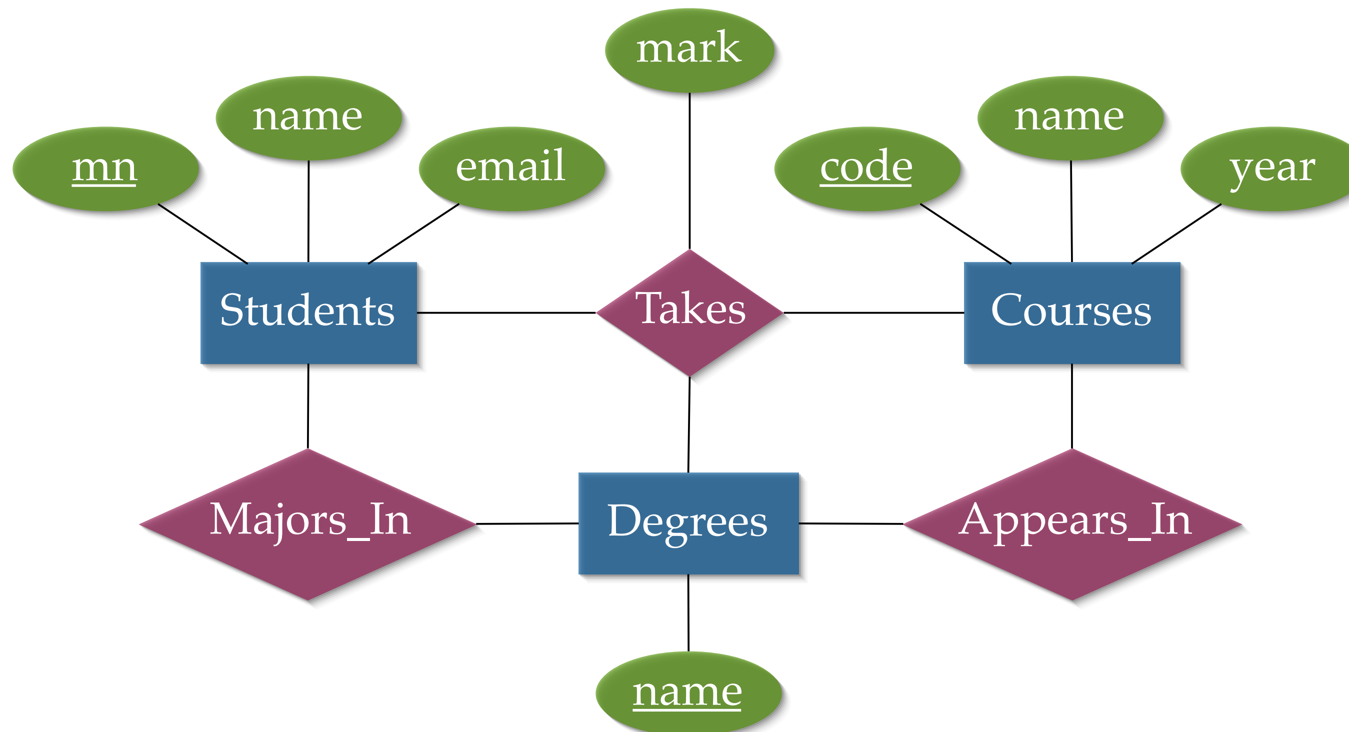
Relationship sets are represented as *diamonds* in ER diagrams

Relationship may have *attributes* of their own.



There is no bound on the number of entities participating in a relationship.

Correspondingly, there is no bound on the number of relationships an entity can participate in



Instances

Entity instances and *relationship instances* are what we obtain after instantiating the attributes of an entity or a relationship

Examples

An entity instance from the Students entity set:

$$(\mathbf{123},\ \text{Natassa},\ \text{natassa@somewhere})$$

An entity instance from the Courses entity set:

$$(\text{inf1}, \text{Informatics 1},\ \mathbf{1})$$

A relationship instance from the Takes relationship set:

$$(\mathbf{123},\ \text{Natassa},\ \text{natassa@somewhere},\ \text{inf1}, \text{Informatics 1},\ \mathbf{1},\ \mathbf{88})$$

Key constraints

A *key constraint* captures identification connections between entities participating in a relationship

Definition. Suppose R is a relationship between n entity sets, $E_1, \ldots, E_n$. There is a *key constraint* on one of the entities, E_k, if, however we instantiate the attributes of E_k, there is at most one relationship instance participated in by the attribute instantiation.

Example. Students, directors of studies (DoS), and the relationship between them (Directed-By)

- Given a Students instance, we can determine the Directed-By instance it appears in. That is, each student has a unique DoS.

One-to-many and many-to-many relationships

A *one-to-many* relationship R between entity sets E_o and E_m means that, for each instance $e_m \in E_m$, there is at most one instance $e_o \in E_o$ such that e_o and e_m appear together in some relationship instance $r \in R$.

More simply: each instance $e_o \in E_o$ may be associated (in R) with many instances $e_m \in E_m$, but each instance $e_m \in E_m$ must be associated (in R) with at most one instance $e_o \in E_o$.

If R is a binary relationship between E_o and E_m, then being one-to-many is equivalent to there being a key contraint on E_m.

A *many-to-many* relationship R between entity sets E_o and E_m means that there are no constraints on the number of times entity instances $e_o \in E_o$ and $e_m \in E_m$ may appear in relationship instances $r \in R$.

Examples

The Directed_By relationship between the Students and DoS entity sets is a many-to-one relationship.

- Each student has a single DoS, but
- each DoS may have many students

The Takes relationship between Students and Courses is a many-to-many relationship

- Each student takes many different courses;
- Each course may be taken by many different students

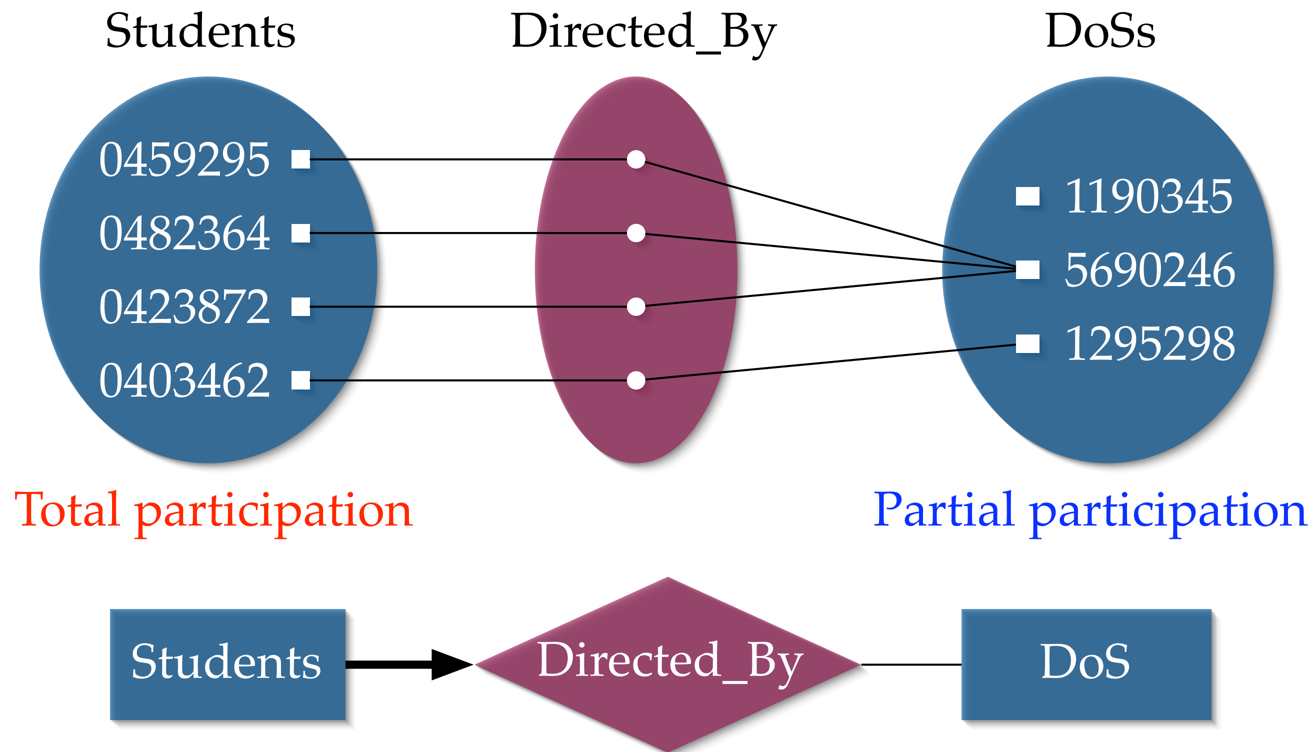
Participation constraints

Participation constraints capture the mode in which an entity participates in a relationship.

Total participation on entity set $\boldsymbol{E}$ for relationship $\boldsymbol{R}$ is declared when every entity instance $\boldsymbol{e} \in \boldsymbol{E}$ appears in at least one relationship instance of $\boldsymbol{R}$.

Partial participation on entity set $\boldsymbol{E}$ for relationship $\boldsymbol{R}$ is declared when there exist entities $\boldsymbol{e} \in \boldsymbol{E}$ that do not appear in instances of $\boldsymbol{R}$.

Example

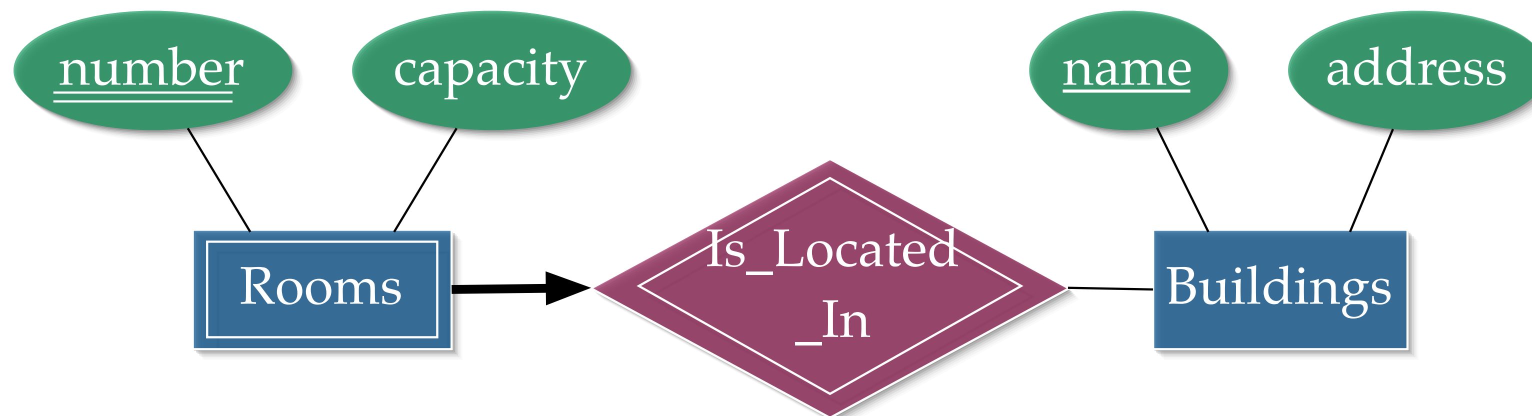


Notation. A *thick arrow* from an entity to a relationship represents that the entity both totally participates in the relationship and also satisfies a key constraint.

Weak entity sets

In certain cases, it is impossible to designate a primary key for entities of an entity set.

Instead, the only way in which set participation can be declared is by "borrowing" the key of another entity set

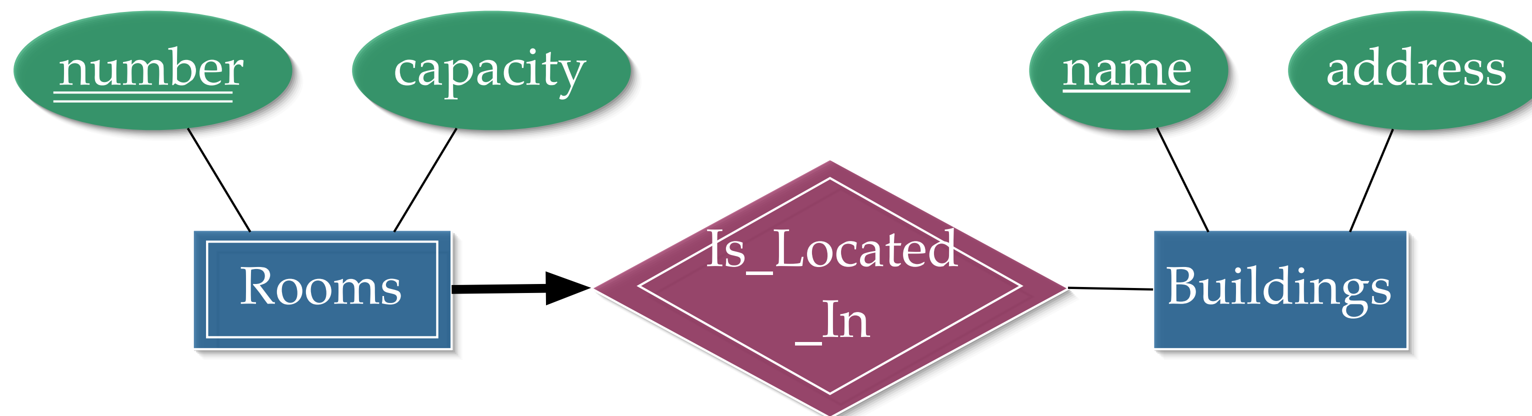


Notation

Double lines for weak entity and identifying relationship

Doubly underlined attributes of the weak entity set participating in the composite key

The *identifying relationship* is many-to-one and total.



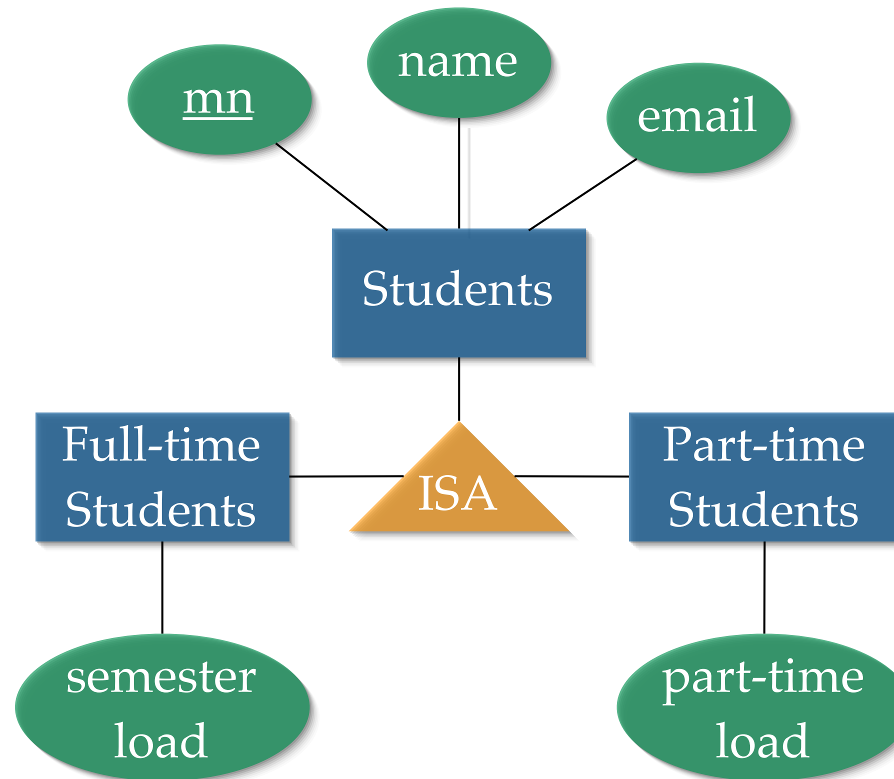
Weak entity set: Definition

- A *weak entity set* is an entity set for which a primary key consisting only of its own attributes cannot be identified
- The *key* is formed by a combination of its own attributes and the key attributes from another entity set with which it has a relationship
- The entity set from which attributes are borrowed is called the *identifying owner*
- The relationship between the weak entity set and its identifying owner is called an *identifying relationship*.
- The identifying relationship must be many-to-one and total.

Hierarchical entities and inheritance

Subclasses (Full-time Students, Part-time Students) *specialise* a *superclass* (Students) by *inheriting* attributes from the superclass.

Subclasses also have additional attributes of their own.



Part I — Structured Data

Data Representation:

I.1 The entity-relationship (ER) data model

I.2 The relational model

Data Manipulation:

I.3 Relational algebra

I.4 Tuple relational calculus

I.5 The SQL query language

Required reading: Chapter 3 of [DMS], §§ 3.1,3.2,3.4,3.5

History of relational model

- The *relational model* was introduced in 1970 by Edgar F. Codd, a British computer scientist working at IBM's Almaden Research Center in San Jose, California.
- IBM was initially slow to exploit the idea, but by the mid 1970's IBM was at the forefront of the commercial development of relational database systems with its System R project, which included the development and first implementation of SQL. (Codd was sidelined from this project!)
- Around the same time, the relational model was developed and implemented at UC Berkely (the Ingres project)
- Nowadays relational databases are a multi-billion pound industry.
- A major reason for the success of the relational model is its simplicity
- In 1981, Codd received the Turing Award for his pioneering work on relational databases

Building blocks

- The basic construct is a *relation*.
  - It consists of a *schema* and an *instance*
  - The *schema* can be thought of as the format of the relation
  - A *relation instance* is also known as a *table*
- A *schema* is a set of fields, which are (name, domain) pairs
  - *fields* may be referred to as attributes, or columns
  - *domains* are referred to as types
- The rows of a table are called *tuples* (or *records*) and they are value assignments from the specified domain for the fields of the table
- The *arity* of a relation is its number of columns (fields)
- The *cardinality* of a table is its number of rows (tuples)

Example

Fields (a.k.a. attributes, columns)

Schema →

Tuples (a.k.a. records, rows)

mn	name	age	email
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

Data definition in SQL

- SQL stands for *Structured Query Language*
- A special subset of SQL called the *Data Definition Language (DDL)* is used to declare table schemata
- Relations are called *tables* in SQL
- It is a typed language
  - For simplicity, we will assume there are only three types: (i) **integer** for integer numbers, (ii) **real** for real numbers (floating point), and (iii) **char**(n) for a string of maximum length n

General form of a DDL statement

create table *table name* **(** *attribute name* *attribute type*
[, *attribute name* *attribute type* **]***
⟨*integrity constraints*⟩ **)**

Example 1

```
create table Students (
      mn              char(8),
      name            char(20),
      age             integer,
      email           char(15),
      primary key     (mn) )
```

The example defines the **`Students`** table.

The last line implements a *primary key constraint*, it declares **`mn`** to be the chosen primary key for **`Students`**.

This constraint requires that the **`Students`** table contains at most one row with any given **`mn`** value. This is enforced by the system.

Any attempt to insert a new row with an **`mn`** value that already exists in some other row of the table will fail.

```
create table Students (
     mn             char(8),
     name           char(20),
     age            integer,
     email          char(15),
     primary key    (mn) )
```

General form of a DDL statement

create table *table name* **(** *attribute name* *attribute type*
[, *attribute name* *attribute type* **]***
⟨*integrity constraints*⟩ **)**

Example 2

```
create table Takes (
      mn              char(8),
      code            char(20),
      mark            integer,
      primary key   (mn, code),
      foreign key   (mn) references Students,
      foreign key   (code) references Courses )
```

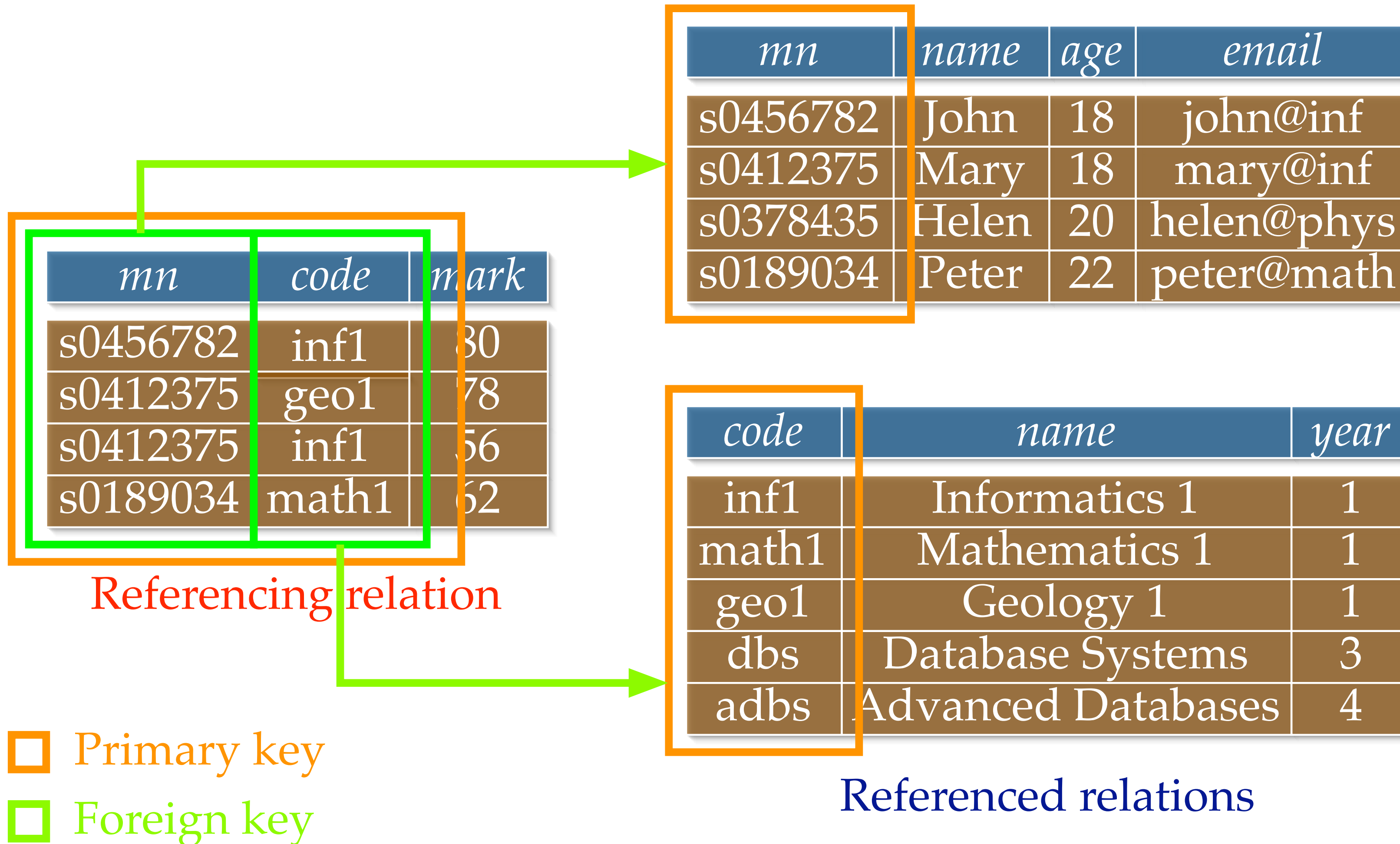
In this case, the primary key is a pair of fields.

The *foreign key constraints* enforce two further properties:

- Whenever a tuple is inserted, the value for the **mn** field must be a value that appears in the primary key column of the **Students** table
- Similarly, the value for the **code** field must be a value that appears in the primary key column of the **Courses** table

```
create table Takes (
     mn              char(8),
     code            char(20),
     mark            integer,
     primary key   (mn, code),
     foreign key   (mn) references Students,
     foreign key   (code) references Courses )
```

Key constraints example



mn	*code*	*mark*
s0456782	inf1	80
s0412375	geo1	78
s0412375	inf1	56
s0189034	math1	62

mn	*name*	*age*	*email*
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

code	*name*	*year*
inf1	Informatics 1	1
math1	Mathematics 1	1
geo1	Geology 1	1
dbs	Database Systems	3
adbs	Advanced Databases	4

Summary

We have seen two forms of constraint:

`primary key` **`(`***declaration***`)`**

`foreign key` **`(`***declaration***`)`** **`references`** *table*

- Primary key constraints declare primary keys.
- Foreign key constraints link columns of one table to the primary key columns of another table.

Both are declared by the user, but enforced by the system itself.
(Attempting to enter a tuple that violates the constraint results in failure.)

N.B. In the ER model, **`Students`** was an entity set and **`Takes`** a relationhip. In the relational model, *both* are (necessarily!) implemented as tables.

Translating an ER diagram to a relational schema

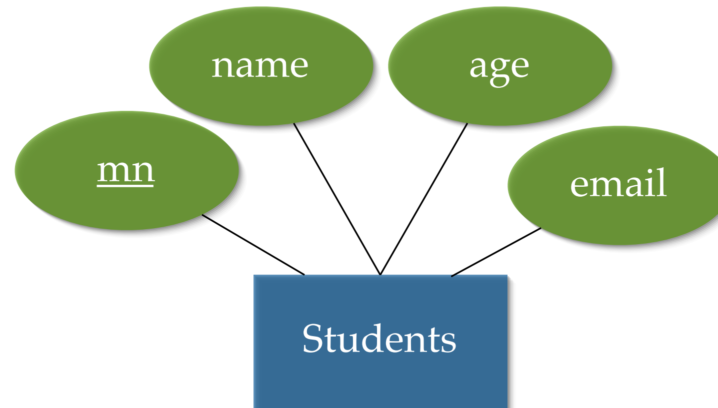
Given an ER diagram, we find a relational schema that closely aproximates the ER design.

The translation is *approximate* because it is not feasible to capture all the constraints in the ER design within the relational schema. (In SQL, certain types of constraint, for example, are inefficient to enforce, and so usually not implemented.)

There is more than one approach to translating an ER diagram to a relational schema. Different translations amount to making different implementation choices for the ER diagram.

In D&A, we just consider a few examples illustrating some of the main ideas.

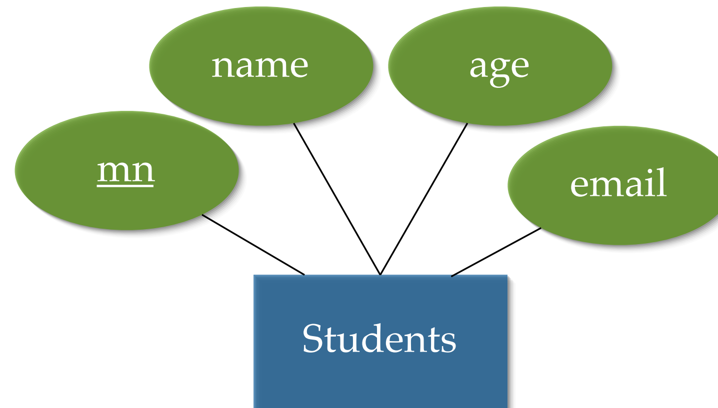
Mapping entity sets



Algorithm

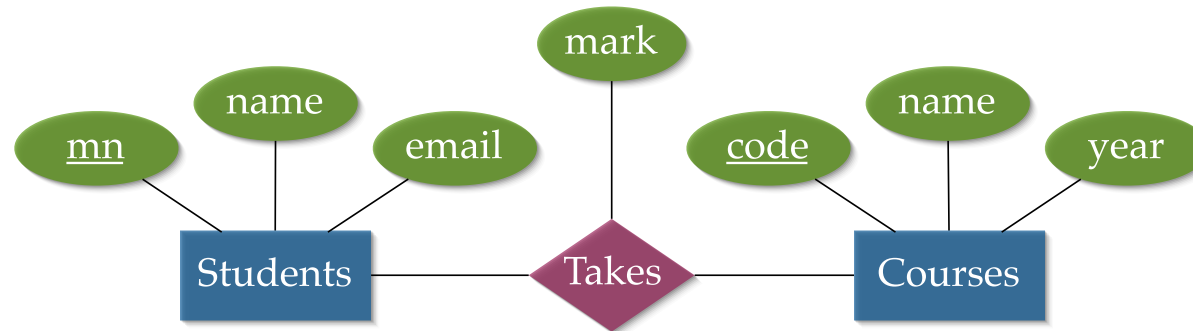
- A table is created for the entity set
- Each attribute of the entity set becomes an field of the table with an appropriate type
- A primary key is declared

Mapping entity sets




```
create table Students (
      mn             char(8),
      name           char(20),
      age            integer,
      email          char(15),
      primary key    (mn) )
```

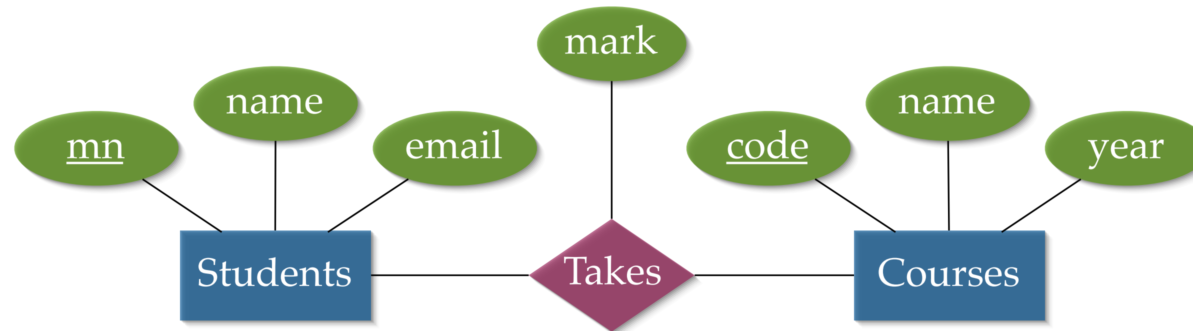
Mapping relationship sets (no key constraints)



Algorithm

- A table is created for the relationship set
- The table contains the primary keys of the participating entity sets
- Descriptive attributes of the relationship are added
- A composite primary key is declared on the table
- Foreign key constraints are declared

Mapping relationship sets (no key constraints)




```
create table Takes (
      mn              char(8),
      code            char(20),
      mark            integer,
      primary key   (mn, code),
      foreign key   (mn) references Students,
      foreign key   (code) references Courses )
```

Mapping relationship sets with key constraints

Algorithm

- A table is created for the relationship set
- The primary key of the “source” entity set is declared as the primary key of the relationship set
- Foreign key constraints are declared for both source and target entity sets

Mapping relationship sets with key constraints

```
create table Directed_By (
      mn              char(8),
      staff_id        char(8),
      primary key     (mn),
      foreign key     (mn) references Students,
      foreign key     (staff_id) references DoS )
```

N.B. The participation constraint on **`Students`** in **`Directed_By`** has not been implemented. To implement this constraint another approach is needed.

Null values

In SQL, a special value a field can have is **`null`**

A **`null`** value means that a field is undefined or missing

Null values are *not allowed* to appear in *primary key* fields,

They *are allowed* to appear in *foreign key* fields.

Null values can be disallowed from other fields using a **`not null`** declaration

In certain circumstances, by disallowing **`null`**, we can enforce a *participation constraint*

Mapping relationship sets with key+participation constraints

Algorithm

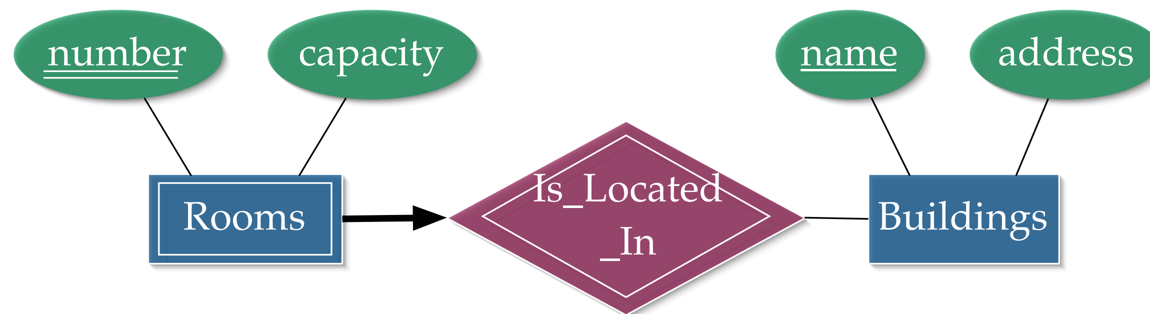
- Include a foreign key field for the "target" entity set within the table for the "source" entity set.
- Give this field a **`not null`** declaration.

N.B. By omitting the **`not null`** declaration, we obtain an alternative way of implementing the key constraint without the participation constraint.

Mapping relationship sets with key+participation constraints

```
create table Students (
      mn            char(8),
      name          char(20),
      age           integer,
      email         char(15),
      dos_id        char(8) not null,
      primary key   (mn),
      foreign key   (dos_id) references DoS )
```

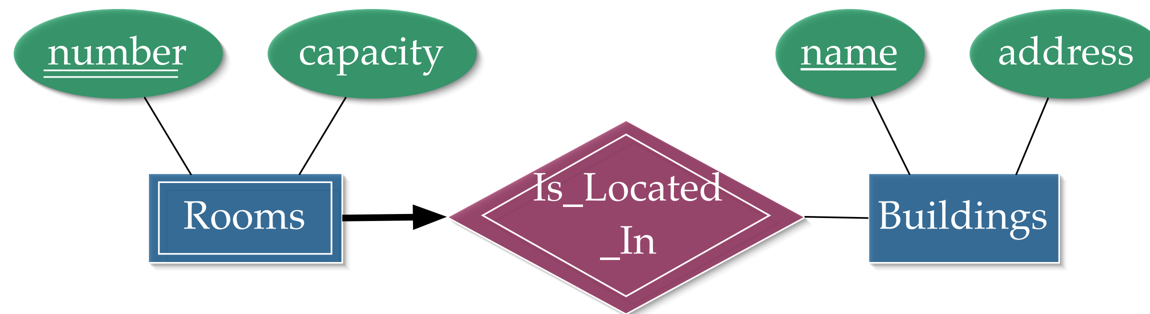
Mapping weak entity sets and identifying relationships



Algorithm

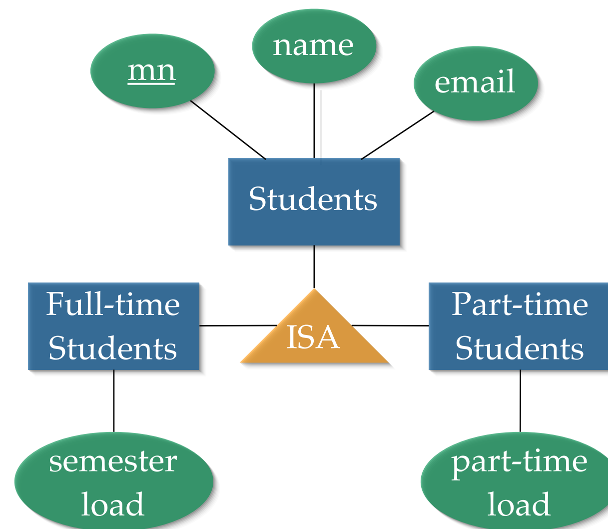
- Create a table for the weak entity set
- Add an attribute set, for the primary key of the entity set's identifying owner's
- Add a foreign key constraint on the identifying owners primary key
- Instruct the system to automatically delete any tuples in the table for which there are no owners

Mapping weak entity sets and identifying relationships



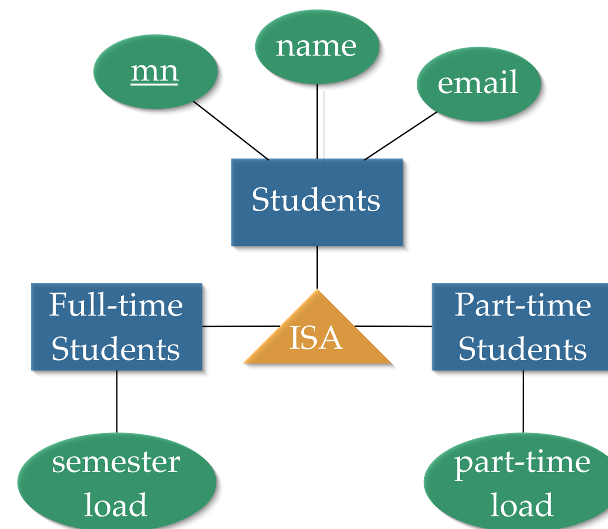

```
create table Rooms (
      number          char(8),
      capacity        integer,
      building_name   char(20),
      primary key     (number,building_name),
      foreign key     (building_name) references Buildings
                      on delete cascade )
```

Mapping hierarchical entities



- Declare a table for the superclass of the hierarchy
- For each subclass, declare another table, containing the superclass's primary key and the subclass's extra attributes
- Each subclass has the same primary key as its superclass
- Declare foreign key constraints

Mapping hierarchical entities




```
create table PT_Students (
      mn              char(8),
      pt_load         integer,
      primary key     (mn),
      foreign key     (mn) references Students )
```

Part I — Structured Data

Data Representation:

I.1 The entity-relationship (ER) data model

I.2 The relational model

Data Manipulation:

I.3 Relational algebra

I.4 Tuple relational calculus

I.5 The SQL query language

Required reading: Chapter 4 of [DMS]: §§ 4.1,4.2

Querying

Once data is organised in a relational schema, the natural next step is to *manipulate* data. For our purposes, this means querying.

Querying is the process of identifying the parts of stored data that have properties of interest

We consider three approaches.

- Relational algebra (today's topic): a *procedural* way of expressing queries over relationally represented data
- Tuple-relational calculus (see I.4): a *declarative* way of expressing queries, tightly coupled to first order predicate logic
- SQL (see I.5): a widely implemented query language influenced by relational algebra and relational calculus

Operators

The key concept in relational algebra is an *operator*

Operators accept a single relation or a pair of relations as input

Operators produce a single relation as output

Operators can be *composed* by using one operator's output as input to another operator (composition of functions)

There are five basic operators: *selection*, *projection*, *union*, *cross-product*, and *difference*

Other operators can be defined as composites of these five, but are so frequently used that they are often treated as fundamental

Selection and projection: σ and π

Recall that relational data is stored in *tables*

Selection and *projection* allow one to isolate any "rectangular subset" of a single table

- Selection identifies *rows* of interest
- Projection identifies *columns* of interest

If both are used on a single table, we extract a *rectangular subset* of the table

Selection: example

mn	name	age	email
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

Students

name	age
John	18
Mary	18
Helen	20
Peter	22

$\pi_{\text{name, age}}(\text{Students})$

mn	name	age	email
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

$\sigma_{\text{age}>18}(\text{Students})$

name	age
Helen	20
Peter	22

Combination

Selection: general form

General form: $\sigma_{\text{predicate}}$(Relation instance)

A *predicate* is a condition that is applied on each row of the table

- It should evaluate to either true or false
- If it evaluates to true, the row is propagated to the output, if it evaluates to false the row is dropped
- The output table may thus have lower cardinality than the input

Predicates are written in the Boolean form

$$\text{term}_1 \ \texttt{bop} \ \text{term}_2 \ \texttt{bop} \ \ldots \ \texttt{bop} \ \text{term}_m$$

- Where $\texttt{bop} \in \{\vee, \wedge\}$
- term_i's are of the form attribute $\texttt{rop}$ constant or attribute_1 $\texttt{rop}$ attribute_2 (where $\texttt{rop} \in \{>, <, =, \neq, \geq, \leq\}$)

Projection: example

mn	name	age	email
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

Students

name	age
John	18
Mary	18
Helen	20
Peter	22

$\pi_{\text{name, age}}(\text{Students})$

mn	name	age	email
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

$\sigma_{\text{age}>18}(\text{Students})$

name	age
Helen	20
Peter	22

Combination

Projection: general form

General form: $\pi_{\text{column list}}(\text{Relation instance})$

All rows of the input are propagated in the output

Only columns appearing in the *column list* appear in the output

Thus the *arity* of the output table may be lower than that of the input table

The resulting relation has a different schema!

Selection and projection: example

mn	name	age	email
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

Students

name	age
John	18
Mary	18
Helen	20
Peter	22

$\pi_{\text{name, age}}(\text{Students})$

mn	name	age	email
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

$\sigma_{\text{age}>18}(\text{Students})$

name	age
Helen	20
Peter	22

Combination

Note the *algebraic equivalence* between:

- $\sigma_{\text{age}>18}(\pi_{\text{name,age}}(\text{Students}))$
- $\pi_{\text{name,age}}(\sigma_{\text{age}>18}(\text{Students}))$

Set operations

There are three basic set operations in relational algebra:

- *union*
- *difference*
- *cross-product*

A fourth, *intersection*, can be expressed in terms of the others

All these set operations are binary.

Essentially, they are the well-known set operations from set theory, but extended to deal with tuples

Union

Let $\boldsymbol{R}$ and $\boldsymbol{S}$ be two relations. For union, set difference and intersection $\boldsymbol{R}$ and $\boldsymbol{S}$ are required to have compatible schemata:

- Two schemata are said to be *compatible* if they have the same number of fields and corresponding fields in a left-to-right order have the same domains. N.B., the names of the fields are not used

The *union* $\boldsymbol{R} \cup \boldsymbol{S}$ of $\boldsymbol{R}$ and $\boldsymbol{S}$ is a new relation with the same schema as $\boldsymbol{R}$. It contains exactly the tuples that appear in at least one of the relations $\boldsymbol{R}$ and $\boldsymbol{S}$

N.B. For naming purposes it is assumed that the output relation inherits the field names from the relation appearing first in the specification ($\boldsymbol{R}$ in the previous case)

Union example

mn	*name*	*age*	*email*
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

S_1

mn	*name*	*age*	*email*
s0489967	Basil	19	basil@inf
s0412375	Mary	18	mary@inf
s9989232	Ophelia	24	oph@bio
s0189034	Peter	22	peter@math
s0289125	Michael	21	mike@geo

S_2

mn	*name*	*age*	*email*
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math
s0489967	Basil	19	basil@inf
s9989232	Ophelia	24	oph@bio
s0289125	Michael	21	mike@geo

$S_1 \cup S_2$

Set difference and intersection

The *set difference* $\boldsymbol{R}-\boldsymbol{S}$ and *intersection* $\boldsymbol{R}\cap\boldsymbol{S}$ are also new relations with the same schema as $\boldsymbol{R}$ and $\boldsymbol{S}$.

$\boldsymbol{R}-\boldsymbol{S}$ contains exactly those tuples that appear in $\boldsymbol{R}$ but which do not appear in $\boldsymbol{S}$

$\boldsymbol{R}\cap\boldsymbol{S}$ contains exactly those tuples that appear in both $\boldsymbol{R}$ and $\boldsymbol{S}$

For both operations, the same naming conventions apply as for union

Note that intersection can be defined from set difference by
$\boldsymbol{R}\cap\boldsymbol{S}=\boldsymbol{R}-(\boldsymbol{R}-\boldsymbol{S})$

Set difference example

mn	*name*	*age*	*email*
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

S_1

mn	*name*	*age*	*email*
s0489967	Basil	19	basil@inf
s0412375	Mary	18	mary@inf
s9989232	Ophelia	24	oph@bio
s0189034	Peter	22	peter@math
s0289125	Michael	21	mike@geo

S_2

mn	*name*	*age*	*email*
s0456782	John	18	john@inf
s0378435	Helen	20	helen@phys

$S_1 - S_2$

Intersection example

mn	*name*	*age*	*email*
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

S_1

mn	*name*	*age*	*email*
s0489967	Basil	19	basil@inf
s0412375	Mary	18	mary@inf
s9989232	Ophelia	24	oph@bio
s0189034	Peter	22	peter@math
s0289125	Michael	21	mike@geo

S_2

mn	*name*	*age*	*email*
s0412375	Mary	18	mary@inf
s0189034	Peter	22	peter@math

$S_1 \cap S_2$

Cross product

The *cross-product* (also known as the *Cartesian product*) $\boldsymbol{R} \times \boldsymbol{S}$ of two relations $\boldsymbol{R}$ and $\boldsymbol{S}$ is a new relation where

- The schema of the relation is obtained by first listing all the fields of $\boldsymbol{R}$ (in order) followed by all the fields of $\boldsymbol{S}$ (in order).
- The resulting relation contains one tuple $\langle \boldsymbol{r}, \boldsymbol{s} \rangle$ for each pair of tuples $\boldsymbol{r} \in \boldsymbol{R}$ and $\boldsymbol{s} \in \boldsymbol{S}$. (Here $\langle \boldsymbol{r}, \boldsymbol{s} \rangle$ denotes the tuple obtained by appending $\boldsymbol{r}$ and $\boldsymbol{s}$ together, with $\boldsymbol{r}$ first and $\boldsymbol{s}$ second.)

Note that if there is a field name common to $\boldsymbol{R}$ and $\boldsymbol{S}$ then two separate columns with this name appear in the cross-product schema, as defined above, causing a *naming conflict*.

N.B. The two relations need not have the same schema to begin with.

Cross-product example

mn	*name*	*age*	*email*
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

S_1

code	*name*	*year*
inf1	Informatics 1	1
math1	Mathematics 1	1

R

mn	*name*	*age*	*email*	*code*	*name*	*year*
s0456782	John	18	john@inf	inf1	Informatics 1	1
s0456782	John	18	john@inf	math1	Mathematics 1	1
s0412375	Mary	18	mary@inf	inf1	Informatics 1	1
s0412375	Mary	18	mary@inf	math1	Mathematics 1	1
s0378435	Helen	20	helen@phys	inf1	Informatics 1	1
s0378435	Helen	20	helen@phys	math1	Mathematics 1	1
s0189034	Peter	22	peter@math	inf1	Informatics 1	1
s0189034	Peter	22	peter@math	math1	Mathematics 1	1

$S_1 \times R$

Renaming

The renaming operator changes the names of tables and columns.

This can be used to avoid *naming conflicts* when the application of an operator results in a schema with duplicate column names

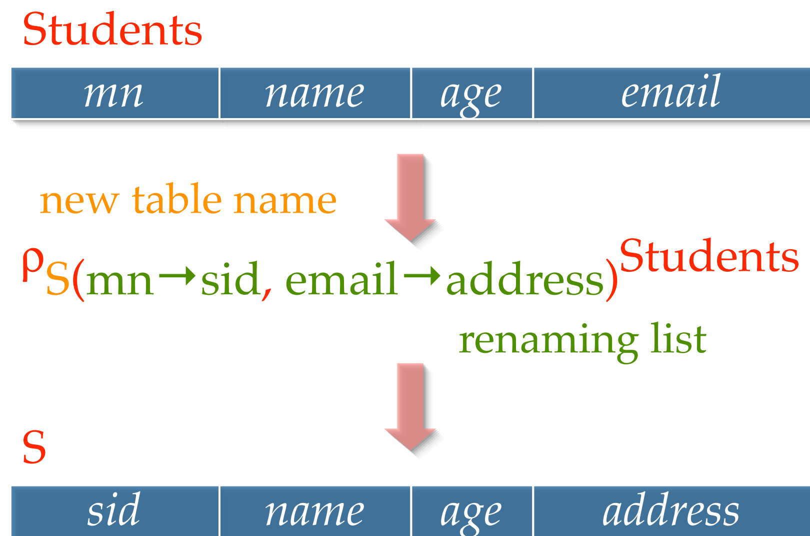
General form

$$\rho_{\text{New-relation-name}(\text{renaming-list})}(\text{Original-relation-name})$$

Semantics:

- The relation is assigned the new relation name
- The renaming list consists of terms of the form oldname $\longrightarrow$ newname which rename a field named oldname to newname
- For ρ to be well-defined there should be no naming conflicts in the output

Renaming example



N.B.

- The types of the columns do not change
- Either the renaming list, or the new table name may be empty

Join

The *relational join* $R \bowtie_p S$ is the most frequently used relational operator.

It is a *derived operator*, it can be defined in terms of cross-product and selection.

The format for a join is $R \bowtie_p S$ where R and S are relations and the *join predicate* p is a predicate (as defined on slide 3.54) that applies to the schema of $R \times S$.

For example, p may have the form $\mathrm{col}_1 \, \texttt{rop} \, \mathrm{col}_2$ where col_1, col_2 are columns of R, S and $\texttt{rop} \in \{>, <, =, \neq, \geq, \leq\}$

Formally, the relational join is *defined* by:

$$R \bowtie_p S = \sigma_p(R \times S)$$

Join example

mn	*name*	*age*	*email*
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

Students

mn	*code*	*mark*
s0412375	inf1	80
s0378435	math1	70

Takes

mn	*name*	*age*	*email*	*mn*	*code*	*mark*
s0456782	John	18	john@inf	s0412375	inf1	80
s0456782	John	18	john@inf	s0378435	math1	70
s0412375	Mary	18	mary@inf	s0412375	inf1	80
s0412375	Mary	18	mary@inf	s0378435	math1	70
s0378435	Helen	20	helen@phys	s0412375	inf1	80
s0378435	Helen	20	helen@phys	s0378435	math1	70
s0189034	Peter	22	peter@math	s0412375	inf1	80
s0189034	Peter	22	peter@math	s0378435	math1	70

$Students \bowtie_{Students.mn = Takes.mn} Takes$

Equijoin

An *equijoin* is a commonly occurring join operation in which the predicate is a conjunction of equalities of the form $\boldsymbol{R}.\texttt{name}_{\mathbf{1}} = \boldsymbol{S}.\texttt{name}_{\mathbf{2}}$.
(A *conjunction* is a list of conditions connected by $\wedge$.)

The schema of the equijoin consists of the fields of $\boldsymbol{R}$, followed by just those fields of $\boldsymbol{S}$ that are not mentioned in the join equalities. The equijoin is computed by *projecting* the join onto the fields that remain (all those of $\boldsymbol{R}$, and those from $\boldsymbol{S}$ that have not been removed). Put more simply: remove from the join those columns labelled with $\boldsymbol{S}$-fields that appear in the equalities.

Note that the example on the previous slide,
$\texttt{Students} \bowtie_{\texttt{Students.mn} = \texttt{Takes.mn}} \texttt{Takes}$, is naturally treated as an equijoin . The resulting relation is then as before, but with the second column labelled `mn` removed.

Natural join

The *natural join* is a special equijoin in which the equalities are between *all* fields that have the same name in $\boldsymbol{R}$ and $\boldsymbol{S}$.

We simply write $\boldsymbol{R} \bowtie \boldsymbol{S}$ for such an equijoin.

Note that the equijoin version of the example on slide 3.69 is in fact the natural join `Students` $\bowtie$ `Takes`. (The common field name is `mn`.)

This is a very natural way of joining two relations, hence the name. It frequently occurs when joining two tables in which one has a foreign key constraint referencing the other.

Part I — Structured Data

Data Representation:

I.1 The entity-relationship (ER) data model

I.2 The relational model

Data Manipulation:

I.3 Relational algebra

I.4 Tuple-relational calculus

I.5 The SQL query language

Required reading: Chapter 4 of [DMS], §§ 4.3

Motivation

Tuple-relational calculus is another way of writing queries for relational data.

Its power lies in the fact that it is entirely *declarative*

That is, we specify the properties of the data we are interested in retrieving, but (in contrast to relational algebra) we do not describe a method by which the data can be retrieved

Basic format

Queries are based on *tuple variables*.

Each tuple variable has an associated schema. The variable ranges over all possible tuples of values matching the schema declaration.

A query has the form

$$\{T \mid p(T)\}$$

where T is a tuple variable and $p(T)$ is a (first-order predicate logic) formula (in which the tuple variable T occurs free).

The result of this query is the set of all possible tuples t (consistent with the schema of T) for which the formula $p(T)$ evaluates to true with $T = t$

Simple example

Find all students at least 19 years old

$$\{S \mid S \in \texttt{Students} \wedge S.\texttt{age} > 18\}$$

In detail:

- Tuple variable S is introduced
- S instantiated over all tuples in the Students table
- Predicate $S.\texttt{age} > 18$ is evaluated on each individual tuple
- If and only if the predicate evaluates to true, the tuple is propagated to the output

Formal syntax of atomic formulae

An *atomic formula* is one of the following:

- $\boldsymbol{R} \in \mathit{Rel}$
- $\boldsymbol{R.a}\ \ \mathit{op}\ \ \boldsymbol{S.b}$
- $\boldsymbol{R.a}\ \ \mathit{op}\ \ \mathit{constant}$
- $\mathit{constant}\ \ \mathit{op}\ \ \boldsymbol{S.b}$

where: $\boldsymbol{R}, \boldsymbol{S}$ are tuple variables, *Rel* is a relation name, $\boldsymbol{a}, \boldsymbol{b}$ are attributes of $\boldsymbol{R}, \boldsymbol{S}$ respectively, and *op* is any operator in the set $\{>, <, =, \neq, \geq, \leq\}$

Formal syntax of (composite) formulae

A *formula* is (recursively defined) to be one of the following:

- any atomic formula
- $\neg p, \quad p \wedge q, \quad p \vee q, \quad p \Rightarrow q$
- $\exists R.\, p(R), \quad \forall R.\, p(R)$

where $p(R)$ denotes a formula in which the variable R appears free.

N.B. First-order logic was introduced in more detail in Inf1A Computation & Logic. Here, we use different notation for the connectives: $\neg$ for *not*; $\wedge$ for *and*; $\vee$ for *or*; and $\Rightarrow$ for $\longrightarrow$. Our notation agrees with Ramakrishnan & Gehrke "Database Management Systems". The main difference from standard first-order logic is the use of variables ranging over tuples (rather than individuals), and the correspondingly specialised forms of atomic formula.

A subtle point

In ordinary first-order logic, one can, in theory, form quantifications $\exists \boldsymbol{R}.\ \boldsymbol{p}$ and $\forall \boldsymbol{R}.\ \boldsymbol{p}$ even when $\boldsymbol{R}$ does not occur in $\boldsymbol{p}$. (In practice, such quantifications are normally useless since they are vacuous.)

In tuple-relational calculus we only allow $\exists \boldsymbol{R}.\ \boldsymbol{p}$ and $\forall \boldsymbol{R}.\ \boldsymbol{p}$ when $\boldsymbol{R}$ occurs free in $\boldsymbol{p}$ for the following reason.

- Under this rule, every tuple variable $\boldsymbol{R}$ that appears in a formula is forced to appear in at least one atomic subformula. The atomic formulae in which $\boldsymbol{R}$ appears then determine the schema of $\boldsymbol{R}$. The schema is taken to be the smallest one containing all the fields that are declared as attributes of $\boldsymbol{R}$ within the formula itself.

Illustrative example

An example illustrating the previous point.

$$\{P \mid \exists S \in \texttt{Students}\,(S.\texttt{age} > 20 \;\wedge\; P.\texttt{name} = S.\texttt{name} \;\wedge\; P.\texttt{age} = S.\texttt{age})\}$$

- The schema of S is that of the **`Students`** table. This is declared by the atomic formula $S \in \texttt{Students}$.
- The schema of P has just two fields **`name`** and **`age`**, with the same types as the corresponding fields in **`Students`**.
- The query returns a table with two fields **`name`** and **`age`** containing the names and ages of all students aged 21 or over.

Note the use of $\exists S \in \texttt{Students}\,(p)$ for $\exists S\,(S \in \texttt{Students} \wedge p)$.
We make free use of such (standard) abbreviations.

Further examples (1)

Query: Find the names of students who are taking Informatics 1

Relational algebra:

$$\pi_{\texttt{Students.name}}(\texttt{Students} \bowtie_{\texttt{Students.mn=Takes.mn}} (\texttt{Takes} \bowtie_{\texttt{Takes.code=Courses.code}} (\sigma_{\texttt{name=`Informatics 1'}}(\texttt{Courses}))))$$

Tuple-relational calculus:

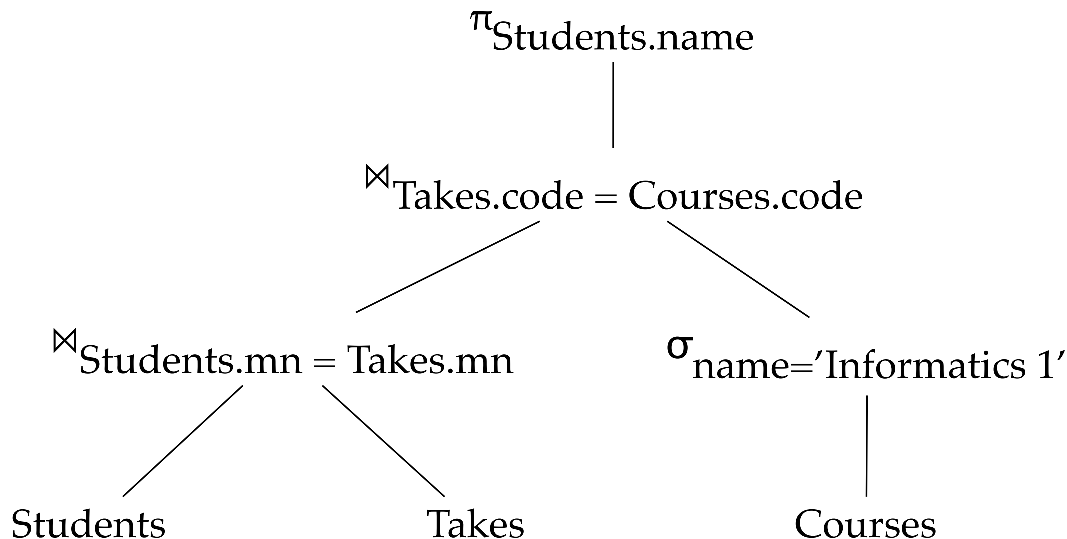
$$\{P \mid \exists S \in \texttt{Students}\ \exists T \in \texttt{Takes}\ \exists C \in \texttt{Courses}\ (C.\texttt{name} = \text{`Informatics 1'} \ \wedge\ C.\texttt{code} = T.\texttt{code} \ \wedge\ S.\texttt{mn} = T.\texttt{mn} \ \wedge\ P.\texttt{name} = S.\texttt{name})\}$$

Tree representation of algebraic expression (abstract syntax)

For the previous query, changing the bracketing does not change the query.

$$\pi_{\texttt{Students.name}}((\texttt{Students} \bowtie_{\texttt{Students.mn}=\texttt{Takes.mn}} \texttt{Takes}) \bowtie_{\texttt{Takes.code}=\texttt{Courses.code}} (\sigma_{\texttt{name}=\text{'Informatics 1'}}(\texttt{Courses})))$$

A tree representation can help one visualise a relational algebra query.



Further examples (2)

Query: Find the names of all courses taken by (everyone called) Joe

Relational algebra:

$$\pi_{\texttt{Courses.name}}((\sigma_{\texttt{name}=\text{'Joe'}}(\texttt{Students})) \bowtie_{\texttt{Students.mn}=\texttt{Takes.mn}}$$
$$(\texttt{Takes} \bowtie_{\texttt{Takes.code}=\texttt{Courses.code}} \texttt{Courses}))$$

Tuple-relational calculus:

$$\{P \mid \exists S \in \texttt{Students}\ \exists T \in \texttt{Takes}\ \exists C \in \texttt{Courses}$$
$$(S.\texttt{name} = \text{'Joe'} \ \wedge \ S.\texttt{mn} = T.\texttt{mn} \ \wedge$$
$$C.\texttt{code} = T.\texttt{code} \ \wedge \ P.\texttt{name} = C.\texttt{name})\}$$

Further examples (3)

Query: Find the names of all students who are taking Informatics 1 or Geology 1

Relational algebra:

$$\pi_{\texttt{Students.name}}(\texttt{Students} \bowtie_{\texttt{Students.mn=Takes.mn}}$$
$$(\texttt{Takes} \bowtie_{\texttt{Takes.code=Courses.code}}$$
$$(\sigma_{\texttt{name='Informatics 1'} \vee \texttt{name=}\text{'Geology 1'}}(\texttt{Courses}))))$$

Tuple-relational calculus:

$$\{P \mid \exists S \in \texttt{Students}\ \exists T \in \texttt{Takes}\ \exists C \in \texttt{Courses}$$
$$((C.\texttt{name} = \text{'Informatics 1'} \ \vee \ C.\texttt{name} = \text{'Geology 1'}) \ \wedge$$
$$C.\texttt{code} = T.\texttt{code} \ \wedge S.\texttt{mn} = T.\texttt{mn} \ \wedge \ P.\texttt{name} = S.\texttt{name})\}$$

Further examples (4)

Query: Find the names of students who are taking both Informatics 1 and Geology 1

Relational algebra:

$$
\begin{array}{l}
\pi_{\text{Students.name}}(\\
\quad (\texttt{Students} \bowtie_{\texttt{Students.mn}=\texttt{Takes.mn}} \\
\qquad (\texttt{Takes} \bowtie_{\texttt{Takes.code}=\texttt{Courses.code}} \\
\qquad\quad (\sigma_{\texttt{name}=\text{`Informatics 1'}}(\text{Courses})))) \\
\quad \cap \\
\quad (\texttt{Students} \bowtie_{\texttt{Students.mn}=\texttt{Takes.mn}} \\
\qquad (\texttt{Takes} \bowtie_{\texttt{Takes.code}=\texttt{Courses.code}} \\
\qquad\quad (\sigma_{\texttt{name}=\text{`Geology 1'}}(\text{Courses})))) \;)
\end{array}
$$

Further examples (4 continued)

Query: Find the names of students who are taking both Informatics 1 and Geology 1

Tuple-relational calculus:

$$\{P \mid \exists S \in \texttt{Students}\,(P.\texttt{name} = S.\texttt{name} \;\wedge$$
$$\forall C \in \texttt{Courses}$$
$$((C.\texttt{name} = \text{`Informatics 1'} \;\vee\; C.\texttt{name} = \text{`Geology 1'}) \Rightarrow$$
$$(\exists T \in \texttt{Takes}\,(T.\texttt{mn} = S.\texttt{mn} \;\wedge\; T.\texttt{code} = C.\texttt{code})))) \,\}$$

Exercise. What does this query return in the case that there is no course in **Courses** called ‘Geology 1’? Find a way of rewriting the query so that it only returns an answer if both ‘Informatics 1’ and ‘Geology 1’ courses exist.

Further examples (5)

Query: Find the names of all students who are taking all courses

Tuple-relational calculus:

$$\{P \mid \exists S \in \texttt{Students}\,(P.\texttt{name} = S.\texttt{name} \wedge \forall C \in \texttt{Courses}\,(\exists T \in \texttt{Takes}\,(T.\texttt{mn} = S.\texttt{mn} \;\wedge\; T.\texttt{code} = C.\texttt{code})))\}$$

Exercise. Try to write this query in relational algebra.

Relational algebra and tuple-relational calculus compared

Relational algebra (RA) and tuple-relational calculus (TRC) have the *same* expressive power

That is, if a query can be expressed in RA, then it can be expressed in TRC, and vice-versa

Why is it useful to have both approaches?

Declarative versus procedural

Recall that TRC is *declarative* and RA is *procedural*.

This suggests the following methodology.

- *Specify* the data that needs to be retrieved using TRC.
- Translate this to an *equivalent query* in RA that gives an *efficient method* of retrieving the data.

This methodology underpins practical approaches to *query optimisation* in relational databases.

In practice, queries are written in a real-world query language such as SQL, rather than TRC.

Nevertheless, query optimisation is of enormous importance in applications.

Part I — Structured Data

Data Representation:

I.1 The entity-relationship (ER) data model

I.2 The relational model

Data Manipulation:

I.3 Relational algebra

I.4 Tuple-relational calculus

I.5 The SQL query language

Required reading: Chapter 5 of [DMS]: §§ 5.1,5.2,5.3,5.5,5.6

A brief history

SQL stands for *Structured Query Language*

Originally developed at IBM in SEQUEL-XRM and System-R projects (1974–77)

Caught on very rapidly

Currently, most widely used commercial relational database language

Continues to evolve in response to changing needs. (Adopted as a standard by ANSI in 1986, ratified by ISO 1987, revised: 1989, 1992, 1999, 2003, 2006, 2008!)

Pronounced S. Q. L.

Data Manipulation Language

In note 2 we met the SQL *Data Definition Language (DDL)*, which is used to define relational schemata.

This lecture introduces the *Data Manipulation Language (DML)*

The DML allows users to:

- insert, delete and modify rows
- query the database

Note. SQL is a large and complex language. The purpose of this lecture is to introduce some of the basic and most important query forms, sufficient for expressing the kinds of query already considered in relational algebra and tuple-relational calculus. (SQL is currently covered in more detail in the third-year "Database Systems" course.)

Inserting data

Assume a table **Students** with schema:

```
Students   (mn:char(8), name:char(20),
            age:integer, email:char(15))
```

Insert data using:

```
INSERT
INTO Students (mn, name, age, email)
VALUES ('s0765432', 'Bob', 19, 'bob@sms')
```

Although SQL allows the list of column names to be omitted from the **INTO** clause (SQL merely requires the tuple of values to be presented in the correct order), it is considered good style to write this list explicitly.

One reason for this is that it means the **INSERT** command can be understood without separate reference to the schema declaration.

Deleting data

Delete *all* students called Bob from **Students**.

```
DELETE
FROM Students S
WHERE S.name = 'Bob'
```

Updating data

Rename student ‘s0765432’ Bobby.

```
UPDATE Students S
SET S.name = 'Bobby'
WHERE S.mn = 's0765432'
```

Form of a basic SQL query

SELECT [DISTINCT] *select-list*

FROM *from-list*

WHERE *qualifications*

- The **SELECT** clause specifies columns to be retained in the result. (N.B., it performs a *projection* rather than a *selection*.)
- The **FROM** clause specifies a cross-product of tables.
- The **WHERE** clause specifies selection conditions on the rows of the table obtained via the **FROM** clause
- The **SELECT** and **FROM** clauses are required, the **WHERE** clause is optional.

A simple example

Query: Find all students at least 19 years old

```
SELECT *
FROM Students S
WHERE S.age > 18
```

This returns all rows in the **Students** table satisfying the condition.

Alternatively, one can be explicit about the fields.

```
SELECT S.mn, S.name, S.age, S.email
FROM Students S
WHERE S.age > 18
```

The first approach is useful for interactive querying. The second is preferable for queries that are to be reused and maintained since the schema of the result is made explicit in the query itself.

A simple example continued

Query: Find the names and ages of all students at least 19 years old

```
SELECT S.name, S.age
FROM Students S
WHERE S.age > 18
```

This query returns a table with one row (with the specified fields) for each student in the **Students** table whose age is 19 years or over.

```
SELECT DISTINCT S.name, S.age
FROM Students S
WHERE S.age > 18
```

This differs from the previous query in that only distinct rows are returned. If more than one student have the same name and age (> 18 years) then the corresponding name-age pair will be included only once in the output table.

Query syntax in detail

- The *from-list* in the **`FROM`** clause is a list of tables. A table name can be followed by a *range variable*; e.g., **`S`** in the queries above.
- The *select-list* in the **`SELECT`** clause is a list of (expressions involving) column names from the tables named in the *from-list*. Column names can be prefixed by range variables.
- The *qualification* in the **`WHERE`** clause is a boolean combination (built using **`AND`**, **`OR`**, and **`NOT`**) of conditions of the form *exp* op *exp*, where op $\in$ {**`<`**, **`=`**, **`>`**, **`<=`**, **`<>`**, **`>=`**, } (the last three stand for $\leq$, $\neq$, $\geq$ respectively), and *exp* is a column name, a constant, or an arithmetic/string expression.
- The **`DISTINCT`** keyword is optional. It indicates that the table computed as an answer to the query should not contain duplicate rows. The default is that duplicate rows are not eliminated.

The meaning of a query

A query computes a table whose contents can be understood via the following *conceptual evaluation strategy* for computing the table.

1. Compute the cross-product of the tables in the *from-list*.
2. Delete rows in the cross-product that fail the *qualification* condition.
3. Delete all columns that do not appear in the *select-list*.
4. If **`DISTINCT`** is specified, eliminate duplicate rows.

This is a *conceptual* evaluation strategy in the sense that it determines the answer to the query, but would be inefficient to follow in practice. Real-world database management systems use *query optimisation* techniques (based on relational algebra!) to find more efficient strategies for evaluating queries.

Diversion: multisets

The sensitivity of SQL to duplicate rows in tables means that SQL models a table as a *multiset* of rows, rather than as a *set* of rows. (In contrast, in the relational model, a table is simply a *relation*, which is just a *set* of tuples.)

A *multiset* (sometimes called a *bag*) is like a set except that it is sensitive to *multiplicities*, i.e., to the number of times a value appears inside it.

For example, the following define the same set, but are *different* multisets:

$$\{2,3,5\} \qquad \{2,3,3,5\} \qquad \{2,3,3,5,5,5\} \qquad \{2,2,2,3,3,5\}$$

Although multisets are sensitive to multiplicities, they are not sensitive to the order in which values are given.

For example, the following define the same multiset.

$$\{2,3,3,5\} \qquad \{3,2,5,3\} \qquad \{5,3,3,2\}$$

Example tables

mn	name	age	email
s0456782	John	18	john@inf
s0412375	Mary	18	mary@inf
s0378435	Helen	20	helen@phys
s0189034	Peter	22	peter@math

Students

code	name	year
inf1	Informatics 1	1
math1	Mathematics 1	1

Courses

mn	code	mark
s0412375	inf1	80
s0378435	math1	70

Takes

Example query (1)

Query: Find the names of all students who are taking Informatics 1

```
SELECT S.name
FROM Students S, Takes T, Courses C
WHERE S.mn = T.mn AND T.code = C.code
     AND C.name = 'Informatics 1'
```

Example query (1 continued)

Query: Find the names of all students who are taking Informatics 1

```
SELECT S.name
FROM Students S, Takes T, Courses C
WHERE S.mn = T.mn AND T.code = C.code
     AND C.name = 'Informatics 1'
```

Step 1 of conceptual evaluation constructs the cross-product of **Students**, **Takes** and **Courses**.

For the example tables, this has 16 rows and 10 columns. (The columns are: **S.mn**, **S.name**, **S.age**, **S.email**, **T.mn**, **T.code**, **T.mark**, **C.code**, **C.name**, **C.year**.)

Example query (1 continued)

Query: Find the names of all students who are taking Informatics 1

```
SELECT S.name
FROM Students S, Takes T, Courses C
WHERE S.mn = T.mn AND T.code = C.code
     AND C.name = 'Informatics 1'
```

Step 2 of conceptual evaluation selects the rows satisfying the condition:

```
S.mn = T.mn AND T.code = C.code
   AND C.name = 'Informatics 1'
```

For the example tables, this has just 1 row (and still 10 columns).

Example query (1 continued)

Query: Find the names of all students who are taking Informatics 1

```
SELECT S.name
FROM Students S, Takes T, Courses C
WHERE S.mn = T.mn AND T.code = C.code
     AND C.name = 'Informatics 1'
```

Step 3 of conceptual evaluation eliminates all columns except **S.name**.

For the example tables, this produces the table

Mary

Step 4 of conceptual evaluation does not apply since **DISTINCT** is not specified. (If **DISTINCT** were specified it would not change the result for our example tables, but it would for other choices of data.)

Example query (2)

Query: Find the names of all courses taken by (everyone called) Mary.

```
SELECT C.name
FROM Students S, Takes T, Courses C
WHERE S.mn = T.mn AND T.code = C.code
     AND S.name = 'Mary'
```

Example query (3)

Query: Find the names of all students who are taking Informatics 1 or Mathematics 1.

```
SELECT S.name
FROM Students S, Takes T, Courses C
WHERE S.mn = T.mn AND T.code = C.code AND
  (C.name='Informatics 1' OR C.name='Mathematics 1')
```

Example query (3 continued)

Query: Find the names of all students who are taking Informatics 1 or Mathematics 1.

```
SELECT S1.name
FROM Students S1, Takes T1, Courses C1
WHERE S1.mn = T1.mn AND T1.code = C1.code
     AND C1.name = 'Informatics 1'
UNION
SELECT S2.name
FROM Students S2, Takes T2, Courses C2
WHERE S2.mn = T2.mn AND T2.code = C2.code
     AND C2.name = 'Mathematics 1'
```

Example query (4)

Query: Find the names of all students who are taking both Informatics 1 and Mathematics 1.

```
SELECT S.name
FROM Students S, Takes T1, Courses C1,
  Takes T2, Courses C2,
WHERE S.mn = T1.mn AND T1.code = C1.code
  AND S.mn = T2.mn AND T2.code = C2.code
  AND C1.name = 'Informatics 1'
  AND C2.name = 'Mathematics 1'
```

This is complicated, somewhat counterintuitive (and also inefficient!)

Example query (5)

Query: Find the matriculation numbers and names of all students who are taking both Informatics 1 and Mathematics 1.

```
SELECT S1.mn, S1.name
FROM Students S1, Takes T1, Courses C1
WHERE S1.mn = T1.mn AND T1.code = C1.code
     AND C1.name = 'Informatics 1'
INTERSECT
SELECT S2.mn, S2.name
FROM Students S2, Takes T2, Courses C2
WHERE S2.mn = T2.mn AND T2.code = C2.code
     AND C2.name = 'Mathematics 1'
```

Example query (6)

Query: Find the matriculation numbers and names of of all students who are taking Informatics 1 but not Mathematics 1.

```
SELECT S1.mn, S1.name
FROM Students S1, Takes T1, Courses C1
WHERE S1.mn = T1.mn AND T1.code = C1.code
     AND C1.name = 'Informatics 1'
EXCEPT
SELECT S2.mn, S2.name
FROM Students S2, Takes T2, Courses C2
WHERE S2.mn = T2.mn AND T2.code = C2.code
     AND C2.name = 'Mathematics 1'
```

Example query (7)

Query: Find all pairs of matriculation numbers such that the first student in the pair obtained a higher mark than the second student in Informatics 1.

```
SELECT S1.mn, S2.mn
FROM Students S1, Takes T1, Courses C,
  Students S2, Takes T2
WHERE S1.mn = T1.mn AND T1.code = C.code
  AND S2.mn = T2.mn AND T2.code = C.code
  AND C.name = 'Informatics 1'
  AND T1.mark > T2.mark
```

Aggregate operators

In addition to retrieving data, we often want to perform computation over data.

SQL includes five useful *aggregate operations*, which can be applied on any column, say **A**, of a table.

1. **COUNT ([DISTINCT] A)**: The number of [distinct] values in the **A** column.
2. **SUM ([DISTINCT] A)**: The sum of all [distinct] values in the **A** column.
3. **AVG ([DISTINCT] A)**: The average of all [distinct] values in the **A** column.
4. **MAX (A)**: The maximum value in the **A** column.
5. **MIN (A)**: The minimum value in the **A** column.

Example query (8)

Query: Find the number of students taking Informatics 1.

```
SELECT COUNT(T.mn)
FROM Takes T, Courses C
WHERE T.code = C.code AND C.name = 'Informatics 1'
```

Example query (9)

Query: Find the average mark in Informatics 1.

```
SELECT AVG(T.mark)
FROM Takes T, Courses C
WHERE T.code = C.code AND C.name = 'Informatics 1'
```

Beyond this lecture

There are many topics we haven't covered:

- Nested queries
- The **GROUP BY** and **HAVING** clauses
- Treatment of **NULL** values
- Complex integrity constraints
- Triggers

These are treated in some detail in Chapter 5 of Ramakrishnan & Gehrke's "Database Management Systems".

Knowledge of these topics is *not required* for Informatics 1.