The Relational Data Model and SQL

COMPSCI 2DB3: Databases

Jelle Hellings Holly Koponen

Department of Computing and Software
McMaster University

Winter 2024

Recap

- Introduction.
- The Entity-Relationship Model.
- SQL: The Structured Query Language
  - Intermission: creating basic tables and basic data modifications.

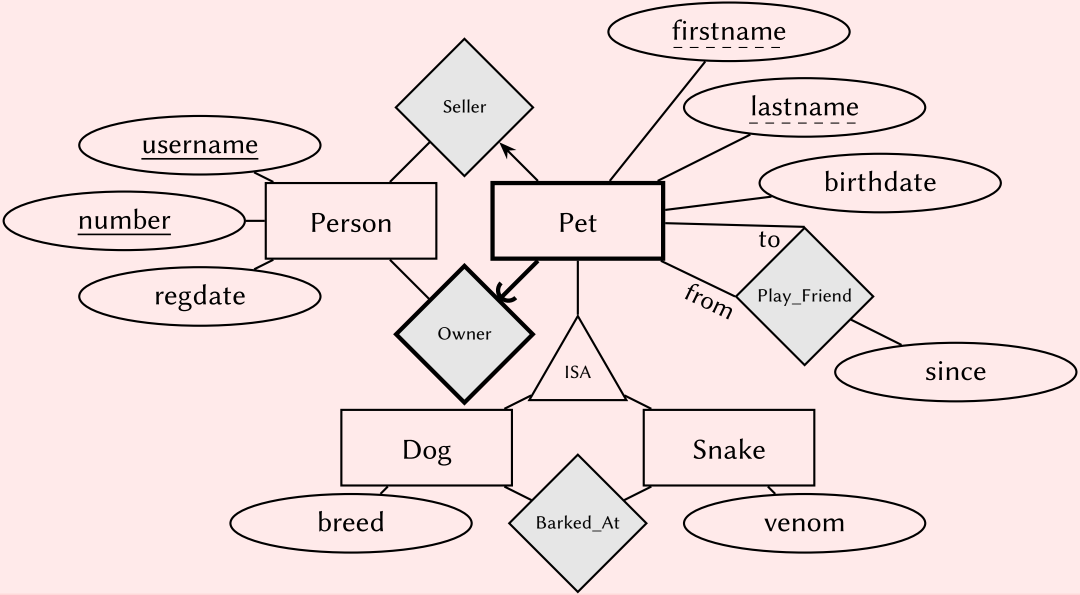
Recap

- Introduction.
- The Entity-Relationship Model.
- SQL: The Structured Query Language
  - Intermission: creating basic tables and basic data modifications.

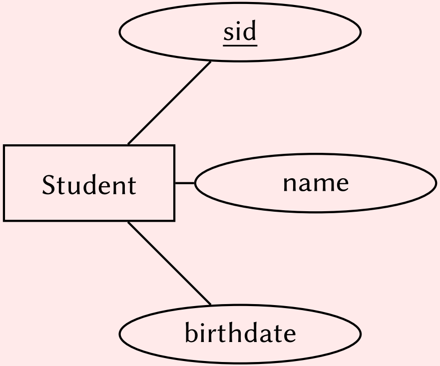
The next steps

- From ER-Diagrams to (SQL) tables.
- Expressing constraints on (SQL) tables.

Spoiler alert



Basic entities to tables



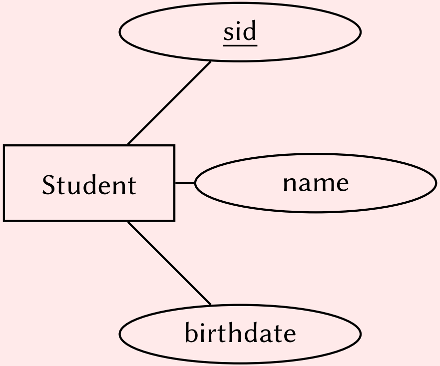
Entity name becomes the table name.

Attributes become the columns.

Keys stay the same.

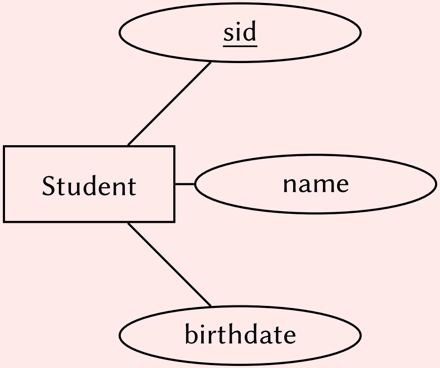
Domains (of attributes) become the column types.

Basic entities to tables



Entity	name becomes the table name.
Attributes	become the columns.
Keys	stay the same.
Domains	(of attributes) become the column types, possibly with domain-based constraints.

Basic entities to tables

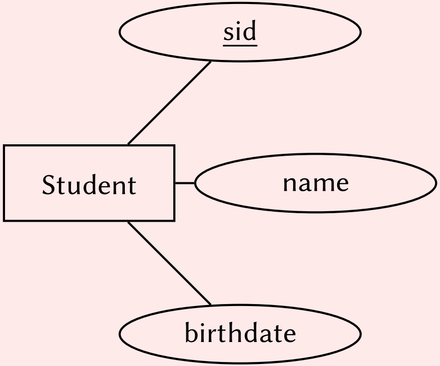


Entity	name becomes the table name.
Attributes	become the columns.
Keys	stay the same.
Domains	(of attributes) become the column types possibly with domain-based constraints.

Example: Attribute ‘score’ with values from 0-10

- The type is an integer type (e.g., **INT**).
- The constraint is: $0 \leq$ score **AND** score ≤ 10.

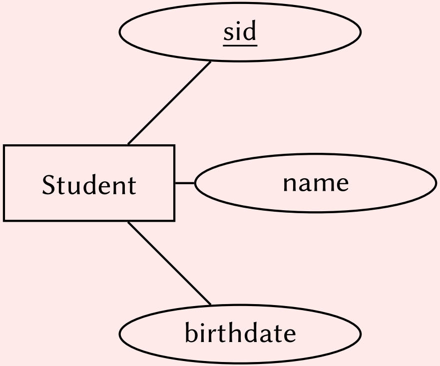
Basic entities to tables



Entity	name becomes the table name.
Attributes	become the columns.
Keys	stay the same.
Domains	(of attributes) become the column types possibly with domain-based constraints.

Student(<u>sid</u> : **INT**,
birthdate : **DATE**,
name : **CLOB**).

Basic entities to tables



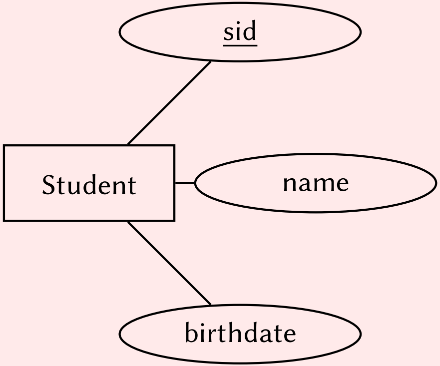
Student(<u>sid</u> : **INT**,
birthdate : **DATE**,
name : **CLOB**).

Creating the corresponding table in SQL

```
CREATE TABLE student
(
  sid INT PRIMARY KEY,
  birthdate DATE NOT NULL,
  name CLOB NOT NULL
);
```

-- **PRIMARY KEY** implies **NOT NULL** and **UNIQUE**.

Basic entities to tables

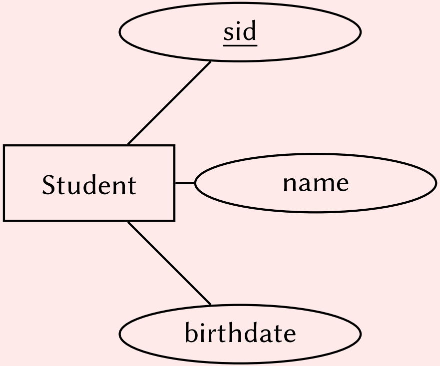


Student(<u>sid</u> : **INT**,
birthdate : **DATE**,
name : **CLOB**).

Creating the corresponding table in SQL

```
CREATE TABLE student
(
  sid INT PRIMARY KEY,
  birthdate DATE NOT NULL,
  name CLOB NOT NULL
);

-- PRIMARY KEY implies NOT NULL and UNIQUE.
-- Some DBMSs prefer or require NOT NULL, however.
```

Basic entities to tables



Student(<u>sid</u> : **INT**,
birthdate : **DATE**,
name : **CLOB**).

Creating the corresponding table in SQL

```
CREATE TABLE student
(
  sid INT GENERATED ... PRIMARY KEY,
  birthdate DATE NOT NULL,
  name CLOB NOT NULL
);
```

-- **PRIMARY KEY** implies **NOT NULL** and **UNIQUE**.

Primary key constraints

student

sid	name	birthdate
1	Alicia	July 4, 2000
2	Celeste	December 12, 1999
3	Dafni	April 17, 2001

student: `sid` **INT PRIMARY KEY**

Primary key constraints

student		
<u>sid</u>	name	birthdate
1	Alicia	July 4, 2000
2	Celeste	December 12, 1999
3	Dafni	April 17, 2001

student: `sid` **INT PRIMARY KEY**

Updating data and primary key constraints

Primary key constraints

student		
<u>sid</u>	name	birthdate
1	Alicia	July 4, 2000
2	Celeste	December 12, 1999
3	Dafni	April 17, 2001
1	Bo	June 17, 2000

student: `sid` **INT PRIMARY KEY**

Updating data and primary key constraints

INSERT into *student*.

Primary key constraints

student		
sid	name	birthdate
1	Alicia	July 4, 2000
2	Celeste	December 12, 1999
3	Dafni	April 17, 2001
~~1~~	Bo	June 17, 2000

student: `sid` **INT PRIMARY KEY**

Updating data and primary key constraints

INSERT into *student*.

The chosen primary key must be unique and not `NULL` (if not *always* automatically generated).

Primary key constraints

student		
sid	name	birthdate
1	Alicia	July 4, 2000
2	Celeste	December 12, 1999
3	Dafni	April 17, 2001

student: `sid` **INT PRIMARY KEY**

Updating data and primary key constraints

UPDATE *student*.

Primary key constraints

student		
sid	name	birthdate
1	Alicia	July 4, 2000
~~3~~	Celeste	December 12, 1999
3	Dafni	April 17, 2001

student: `sid` **INT PRIMARY KEY**

Updating data and primary key constraints

UPDATE *student*.

If the key changes ('sid'), then the new value must be unique and not `NULL` (if not *always* automatically generated).

Primary key constraints

student		
sid	name	birthdate
1	Alicia	July 4, 2000
2	Celeste	December 12, 1999
3	Dafni	April 17, 2001

student: `sid` **INT PRIMARY KEY**

Updating data and primary key constraints

DELETE from *student*.

Primary key constraints

student		
sid	name	birthdate
~~1~~	~~Alicia~~	~~July 4, 2000~~
2	Celeste	December 12, 1999
3	Dafni	April 17, 2001

student: `sid` **INT PRIMARY KEY**

Updating data and primary key constraints

DELETE from *student*.
Nothing gets invalidated.

SQL: *All changes that invalidate a primary key constraint will be rejected.*†

Intermission: Common data types in SQL

Character string and binary string types

CHAR(n) fixed-length string of *n* characters.
VARCHAR(n) variable-length string of at-most *n* characters.
CLOB large strings.

BINARY(n) fixed-length *binary* string of *n* characters.
VARBINARY(n) variable-length *binary* string of at-most *n* characters.
BLOB large *binary* strings.

Both **CLOB** and **BLOB** are intended to store large objects, not to operate on them!
E.g., cannot be primary keys, be joined on, or take part in **UNION**, ...

Intermission: Common data types in SQL

Exact and approximate numeric types

DECIMAL(p, s) exact number with *p* positions before the comma and *s* after.
Also: **DECIMAL** and **DECIMAL**(p)

SMALLINT fixed-width integer type.

INT *larger* fixed-width integer type.

BIGINT *largest* fixed-width integer type.

REAL floating point type.

DOUBLE *larger* floating point type.

BOOLEAN Boolean type.

Intermission: Common data types in SQL

Date and time types

DATE	Date (no time of day)
TIME	Time (no date).
TIMESTAMP	Date and time information.
INTERVAL	Time interval (duration). Comes in year-month and in day-time flavor.

Working with date and time types

- **CURRENT_DATE**, **CURRENT_TIME**, and **CURRENT_TIMESTAMP** yield *now*.
- Functions **YEAR**, **MONTH**, **DAY** to extract date information.
- Functions **HOUR**, **MINUTE**, **SECOND** to extract time information.

Intermission: Common data types in SQL

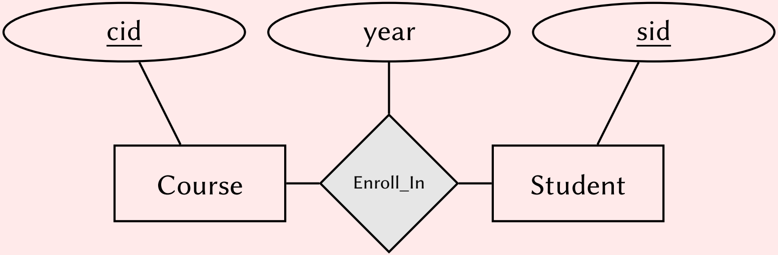
Working with date and time types

- **CURRENT_DATE**, **CURRENT_TIME**, and **CURRENT_TIMESTAMP** yield *now*.
- Functions **YEAR**, **MONTH**, **DAY** to extract date information.
- Functions **HOUR**, **MINUTE**, **SECOND** to extract time information.

Example

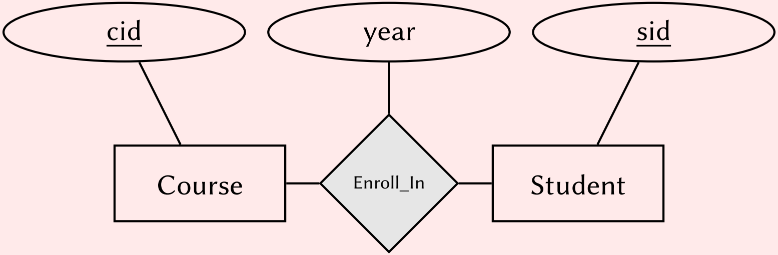
```
CREATE TABLE test
(
  value INT NOT NULL,
  stamp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

SELECT YEAR(stamp), SECOND(stamp)
FROM test;
```

The entity-relationship model: Relationships (many-to-many)



Relationship	becomes the table name.
Entity keys	become the primary key of the table.
Attributes	become extra (non-key) columns.
Domains	(of attributes) become the column types, possibly with domain-based constraints.

The entity-relationship model: Relationships (many-to-many)



Relationship	becomes the table name.
Entity keys	become the primary key of the table.
Attributes	become extra (non-key) columns.
Domains	(of attributes) become the column types, possibly with domain-based constraints.

Enroll_In(<u>sid</u> : **INT**,
<u>cid</u> : **INT**,
year : **INT**).

The entity-relationship model: Relationships (many-to-many)

enroll_in(<u>sid</u> : **INT**, <u>cid</u> : **INT**, year : **INT**).

Creating the corresponding table in SQL

```
CREATE TABLE enroll_in
(
  sid INT NOT NULL,
  cid INT NOT NULL,
  year INT NOT NULL,


);
```

The entity-relationship model: Relationships (many-to-many)

enroll_in(<u>sid</u> : **INT**, <u>cid</u> : **INT**, year : **INT**).

Creating the corresponding table in SQL

```
CREATE TABLE enroll_in
(
  sid INT NOT NULL,
  cid INT NOT NULL,
  year INT NOT NULL,
  PRIMARY KEY(sid, cid),

);
```

The entity-relationship model: Relationships (many-to-many)

enroll_in($\underline{\text{sid}}$: **INT**, $\underline{\text{cid}}$: **INT**, year : **INT**).

Creating the corresponding table in SQL

```sql
CREATE TABLE enroll_in
(
  sid INT NOT NULL,
  cid INT NOT NULL,
  year INT NOT NULL,
  PRIMARY KEY(sid, cid),
  CHECK(2020 <= year)
);
```

The entity-relationship model: Relationships (many-to-many)

enroll_in(<u>sid</u> : **INT**, <u>cid</u> : **INT**, year : **INT**).

Creating the corresponding table in SQL

```sql
CREATE TABLE enroll_in
(
  sid INT NOT NULL REFERENCES student(sid),
  cid INT NOT NULL REFERENCES course(cid),
  year INT NOT NULL,
  PRIMARY KEY(sid, cid),
  CHECK(2020 <= year)
);
```

The entity-relationship model: Relationships (many-to-many)

enroll_in(<u>sid</u> : **INT**, <u>cid</u> : **INT**, year : **INT**).

Creating the corresponding table in SQL

```sql
CREATE TABLE enroll_in
(
  sid INT NOT NULL REFERENCES student(sid),
  cid INT NOT NULL REFERENCES course(cid),
  year INT NOT NULL,
  PRIMARY KEY(sid, cid),
  CHECK(2020 <= year)
);

-- REFERENCES points to a column in another table.
```

The entity-relationship model: Relationships (many-to-many)

enroll_in(<u>sid</u> : **INT**, <u>cid</u> : **INT**, year : **INT**).

Creating the corresponding table in SQL

```sql
CREATE TABLE enroll_in
(
  sid INT NOT NULL REFERENCES student(sid),
  cid INT NOT NULL REFERENCES course(cid),
  year INT NOT NULL,
  PRIMARY KEY(sid, cid),
  CHECK(2020 <= year)
);
```

-- **REFERENCES** points to a column in another table: *foreign key* constraint.

The entity-relationship model: Relationships (many-to-many)

enroll_in(<u>sid</u> : **INT**, <u>cid</u> : **INT**, year : **INT**).

Creating the corresponding table in SQL

```
CREATE TABLE enroll_in
(
  sid INT NOT NULL REFERENCES student(sid),
  cid INT NOT NULL REFERENCES course(cid),
  year INT NOT NULL,
  PRIMARY KEY(sid, cid),
  CHECK(2020 <= year)
);
```

-- **REFERENCES** points to a column in another table: *foreign key* constraint.
-- The referenced column(s) must have a **UNIQUE** constraint.

The entity-relationship model: Relationships (many-to-many)

enroll_in(<u>sid</u> : **INT**, <u>cid</u> : **INT**, year : **INT**).

Creating the corresponding table in SQL

```sql
CREATE TABLE enroll_in
(
  sid INT NOT NULL REFERENCES student,
  cid INT NOT NULL REFERENCES course,
  year INT NOT NULL,
  PRIMARY KEY(sid, cid),
  CHECK(2020 <= year)
);
```

-- **REFERENCES** points to a column in another table: *foreign key* constraint.
-- The referenced column(s) must have a **UNIQUE** constraint.
-- **REFERENCES** points to the primary key in another table.

Foreign key constraints

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Foreign key constraints

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Foreign key constraints

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Foreign key constraints

course			enroll_in		
cid	title		cid	year	sid
1	Programming		1	2019	1
2	Discrete Mathematics		2	2020	1
3	Databases		1	2020	3
4	Advanced Databases		3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Foreign key constraints

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Foreign key constraints

course

cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in

cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

Foreign key constraints

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases
5	Database Theory

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

INSERT into *course*.

Foreign key constraints

course

cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases
5	Database Theory

enroll_in

cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid INT NOT NULL REFERENCES course(cid)`

Updating data and foreign key constraint

INSERT into *course*.
Nothing gets invalidated.

Foreign key constraints

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

UPDATE *course*.

Foreign key constraints

course	
cid	title
1	Programming
2	Discrete Mathematics
5	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	3
~~3~~	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

UPDATE *course*.

If the key changes ('cid'), then rows in enroll_in can be invalidated.

Foreign key constraints

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

DELETE from *course*.

Foreign key constraints

course	
cid	title
~~1~~	~~Programming~~
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
~~1~~	2019	1
2	2020	1
~~1~~	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

DELETE from *course*.
If a row is deleted, then rows in enroll_in can be invalidated.

Foreign key constraints

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2
6	2021	3

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

INSERT into *enroll_in*.

Foreign key constraints

course

<u>cid</u>	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in

<u>cid</u>	year	<u>sid</u>
1	2019	1
2	2020	1
1	2020	3
3	2020	2
6	2021	3

✖

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

INSERT into *enroll_in*.
Is only valid if the inserted ‘cid’ exists.

Foreign key constraints

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

UPDATE *enroll_in*.

Foreign key constraints

course

cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in

cid	year	sid
1	2019	1
5	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

UPDATE *enroll_in*.

Is only valid if the updated 'cid' exists.

Foreign key constraints

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

DELETE from *enroll_in*.

Foreign key constraints

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
~~1~~	~~2020~~	~~3~~
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

DELETE from *enroll_in*.
Nothing gets invalidated.

Foreign key constraints

course

cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in

cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

SQL: *All changes that invalidate a foreign key constraint will be rejected.*†

Foreign key constraints

course

cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in

cid	year	sid
1	2019	1
2	2020	1
1	2020	3
3	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`

Updating data and foreign key constraint

SQL: *All changes that invalidate a foreign key constraint will be rejected by default.*†

Maintaining foreign key constraints in SQL: Non-default actions

course

cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in

cid	year	sid
1	2019	1
2	2020	1
1	2020	2
2	2020	2

Maintaining foreign key constraints in SQL: Non-default actions

course

<u>cid</u>	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in

<u>cid</u>	year	<u>sid</u>
1	2019	1
2	2020	1
1	2020	2
2	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`
ON UPDATE *action* **ON DELETE** *action*

Maintaining foreign key constraints in SQL: Non-default actions

course

cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in

cid	year	sid
1	2019	1
2	2020	1
1	2020	2
2	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`
ON UPDATE *action* **ON DELETE** *action*

Actions to take when the referenced value changes

Maintaining foreign key constraints in SQL: Non-default actions

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	2
2	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`
ON UPDATE *action* **ON DELETE** *action*

Actions to take when the referenced value changes

NO ACTION reject *all invalidating* changes (default).

Maintaining foreign key constraints in SQL: Non-default actions

course

cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in

cid	year	sid
1	2019	1
2	2020	1
1	2020	2
2	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`
ON UPDATE *action* **ON DELETE** *action*

Actions to take when the referenced value changes

NO ACTION reject *all invalidating* changes (default).

RESTRICT reject *all* changes to foreign key columns in referenced rows.

Maintaining foreign key constraints in SQL: Non-default actions

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	2
2	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`
ON UPDATE *action* **ON DELETE** *action*

Actions to take when the referenced value changes

NO ACTION reject *all invalidating* changes (default).

RESTRICT reject *all* changes to foreign key columns in referenced rows.

CASCADE apply the same changes to the foreign key.

Maintaining foreign key constraints in SQL: Non-default actions

course	
cid	title
1	Programming
2	Discrete Mathematics
3	Databases
4	Advanced Databases

enroll_in		
cid	year	sid
1	2019	1
2	2020	1
1	2020	2
2	2020	2

enroll_in: `cid` **INT NOT NULL REFERENCES** `course(cid)`
ON UPDATE *action* **ON DELETE** *action*

Actions to take when the referenced value changes

SET NULL set the foreign key to `NULL`.

SET DEFAULT set the foreign key to the default value (of that column).

†DEFERRED constraints

Consider the following two (empty) tables

```
CREATE TABLE person
(
  pid INT PRIMARY KEY,
  favdish INT NOT NULL
    REFERENCES dish(did)
);
```

```
CREATE TABLE dish
(
  did INT PRIMARY KEY,
  creator INT NOT NULL
    REFERENCES person(pid)
);
```

Every person has a favorite dish. Every dish has a creator (person).

Challenge

How to add a *first* person or a *first* dish?

†DEFERRED constraints

Consider the following two (empty) tables

```
CREATE TABLE person
(
  pid INT PRIMARY KEY,
  favdish INT NOT NULL
    REFERENCES dish(did)
);
```

```
CREATE TABLE dish
(
  did INT PRIMARY KEY,
  creator INT NOT NULL
    REFERENCES person(pid)
);
```

Challenge

How to add a *first* person or a *first* dish?

Solution

Execute a group of **INSERT** statements as a single *transaction* that defers constraint checking via **SET CONSTRAINT ALL DEFERRED** until the end of that transaction.

Transactions are a topic for later!

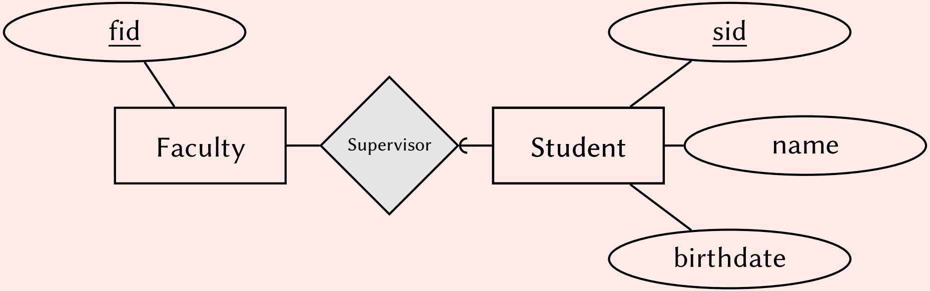
Another example of foreign keys

Consider the following

```
CREATE TABLE tree
(
  node_id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  label VARCHAR(20) NOT NULL,
  parent INT REFERENCES tree(node_id)
);
```

- This table can store tree nodes (as part of a tree structure).
- Each node can refer to a parent node (in the same table).

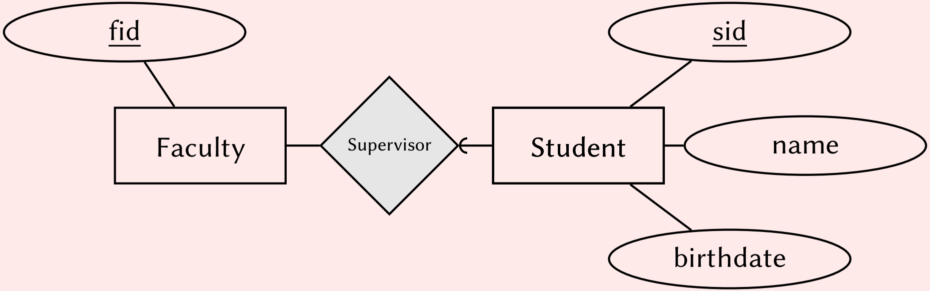
The entity-relationship model: Relationships (strict-one-to-many)



Every student has *exactly one* supervisor.

No need for separate ‘supervisor’ table: add supervisor to the student table.

The entity-relationship model: Relationships (strict-one-to-many)



Every student has *exactly one* supervisor.

No need for separate 'supervisor' table: add supervisor to the student table.

Student(<u>sid</u> : **INT**, birthdate : **DATE**,
name : **CLOB**, supervisor : **INT**).

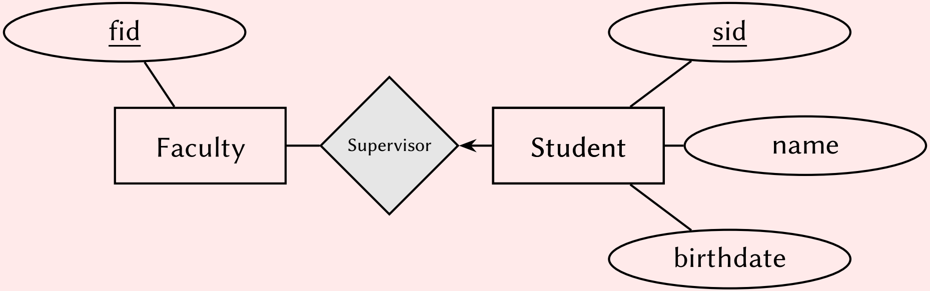
The entity-relationship model: Relationships (strict-one-to-many)

Student(<u>sid</u> : **INT**, birthdate : **DATE**,
name : **CLOB**, supervisor : **INT**).

Creating the corresponding table in SQL

```
CREATE TABLE student
(
  sid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  birthdate DATE NOT NULL,
  name CLOB NOT NULL,
  supervisor INT NOT NULL REFERENCES faculty(fid)
);
```

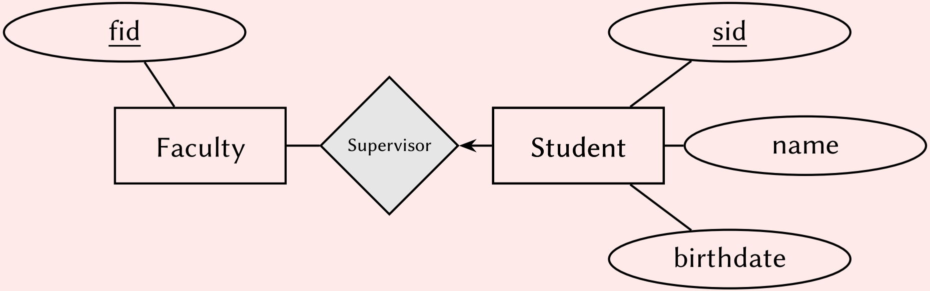
The entity-relationship model: Relationships (one-to-many)



Every student has *at-most one* supervisor.

No need for separate ‘supervisor’ table: add *optional* supervisor to the student table.

The entity-relationship model: Relationships (one-to-many)



Every student has *at-most one* supervisor.

No need for separate 'supervisor' table: add *optional* supervisor to the student table.

$\textbf{Student}(\underline{\text{sid}} : \textbf{INT}, \text{birthdate} : \textbf{DATE}, \text{name} : \textbf{CLOB}, \text{supervisor} : \textbf{INT}_{\text{optional}}).$

The entity-relationship model: Relationships (one-to-many)

Student(<u>sid</u> : **INT**, birthdate : **DATE**,
name : **CLOB**, supervisor : $\textbf{INT}_{\text{optional}}$).

Creating the corresponding table in SQL

```
CREATE TABLE student
(
  sid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  birthdate DATE NOT NULL,
  name CLOB NOT NULL,
  supervisor INT NOT NULL REFERENCES faculty(fid)
);
```

The entity-relationship model: Relationships (one-to-many)

$$\textbf{Student}(\underline{\text{sid}} : \textbf{INT}, \text{birthdate} : \textbf{DATE}, \text{name} : \textbf{CLOB}, \text{supervisor} : \textbf{INT}_{\text{optional}}).$$

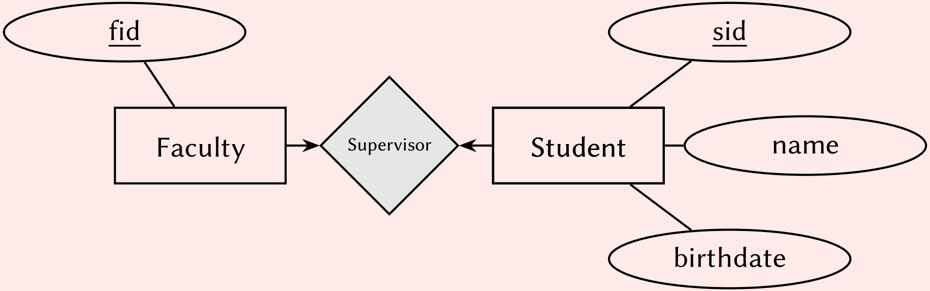
Creating the corresponding table in SQL

```
CREATE TABLE student
(
  sid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  birthdate DATE NOT NULL,
  name CLOB NOT NULL,
  supervisor INT ~~NOT NULL~~ REFERENCES faculty(fid)
);
```

An important use case for outer joins!

```
SELECT * FROM student LEFT JOIN faculty ON supervisor = fid;
```

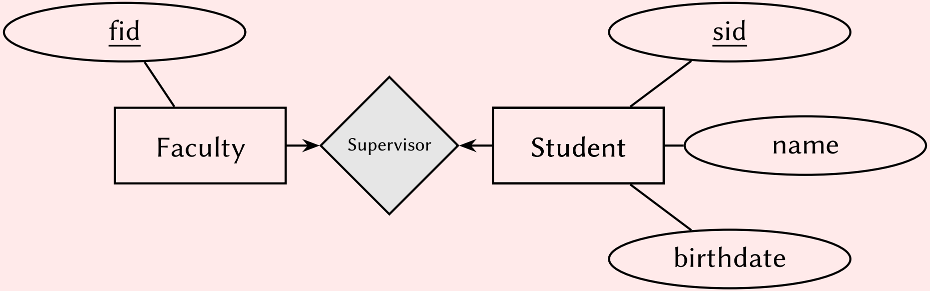
The entity-relationship model: Relationships (one-to-one)



Every student has *at-most one* supervisor, supervisors have *at-most one* student.

No need for separate ‘supervisor’ table: add to the student table or the faculty table.

The entity-relationship model: Relationships (one-to-one)



Every student has *at-most one* supervisor, supervisors have *at-most one* student.

No need for separate ‘supervisor’ table: add to the student table or the faculty table.

Student(<u>sid</u> : **INT**, birthdate : **DATE**,
name : **CLOB**, supervisor : **INT**).

The entity-relationship model: Relationships (one-to-one)

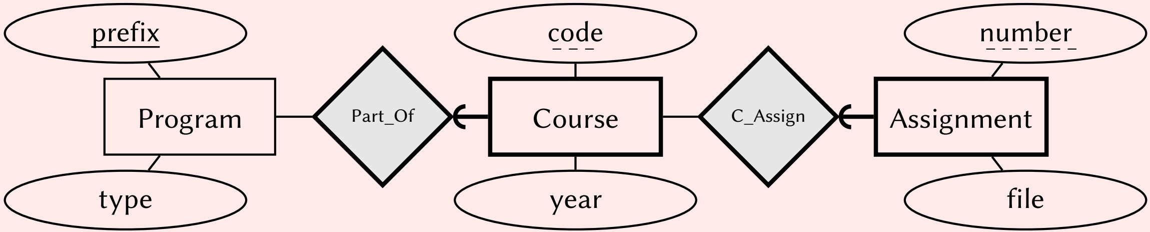
Student(<u>sid</u> : **INT**, birthdate : **DATE**,
name : **CLOB**, supervisor : **INT**).

Creating the corresponding table in SQL

```
CREATE TABLE student
(
  sid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  birthdate DATE NOT NULL,
  name CLOB NOT NULL,
  supervisor INT UNIQUE REFERENCES faculty(fid)
);
```

-- **UNIQUE** guarantees that each non-NULL value only occurs once in a column.

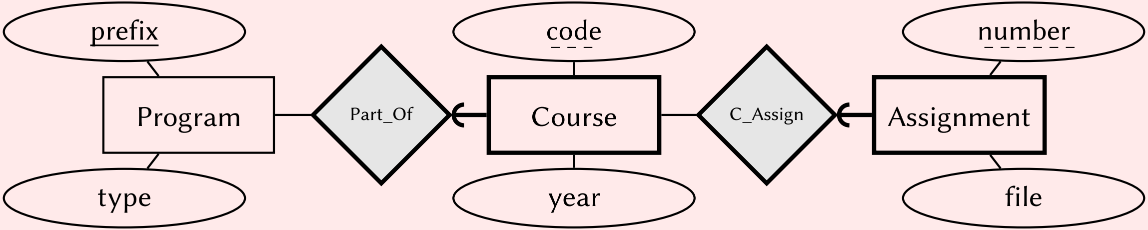
The entity-relationship model: Weak entities



General strategy

- Start with the owner (the normal entity): separate table.
- Each weak entity and corresponding identifying relation: separate table.
- Primary key of the weak entity: primary keys of the owning entities and all partial keys.
- Primary keys of the owning entities are also a foreign key!

The entity-relationship model: Weak entities

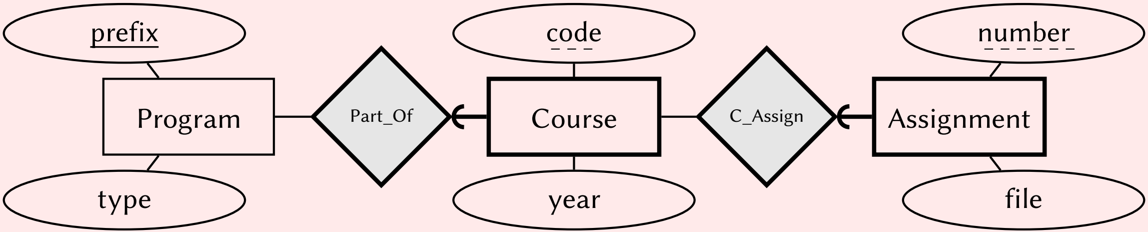


Step 1: The entity Program

Program(<u>prefix</u> : **VARCHAR**(10), type : **VARCHAR**(3)).

Type is a limited set of values? E.g., ‘BSc’, ‘MSc’, ‘PhD’.

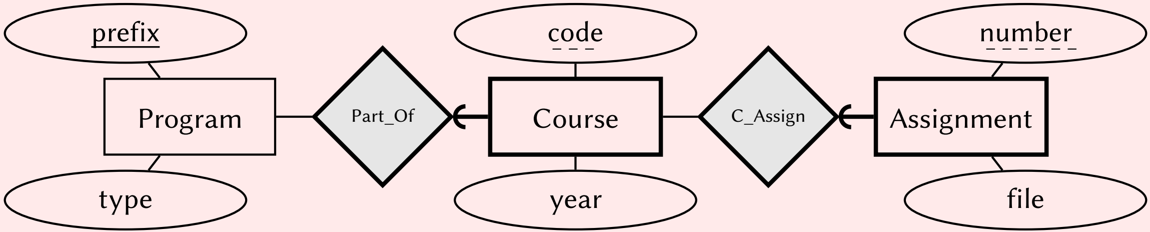
The entity-relationship model: Weak entities



Step 1: The entity Program in SQL

```
CREATE TABLE program
(
  prefix VARCHAR(10) PRIMARY KEY,
  type VARCHAR(3) NOT NULL,
  CHECK(type = 'BSc' OR type = 'MSc' OR type = 'PhD')
);
```

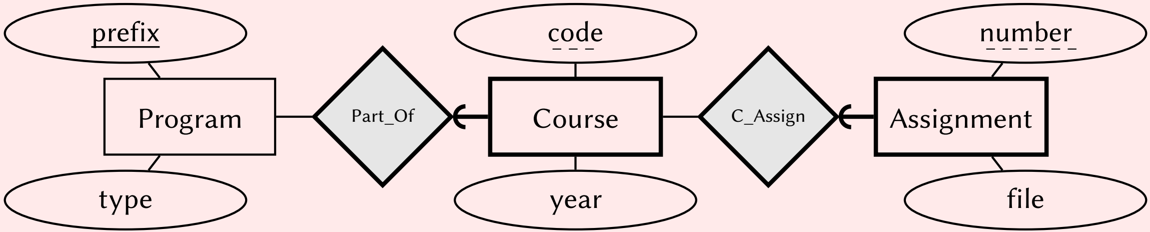
The entity-relationship model: Weak entities



Step 1: The entity Program in SQL

```
CREATE TABLE program
(
  prefix VARCHAR(10) NOT NULL PRIMARY KEY,  -- DB2.
  type VARCHAR(3) NOT NULL,
  CHECK(type = 'BSc' OR type = 'MSc' OR type = 'PhD')
);
```

The entity-relationship model: Weak entities

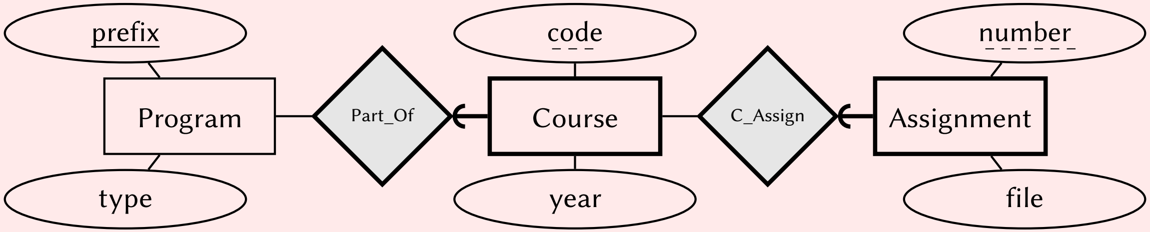


Step 2: The weak entity Course and identifying relationship Part_Of

Course(<u>prefix</u> : **VARCHAR**(10), <u>code</u> : **VARCHAR**(4), year : **INT**).

Prefix is a foreign key pointing to the primary key in **Program**.

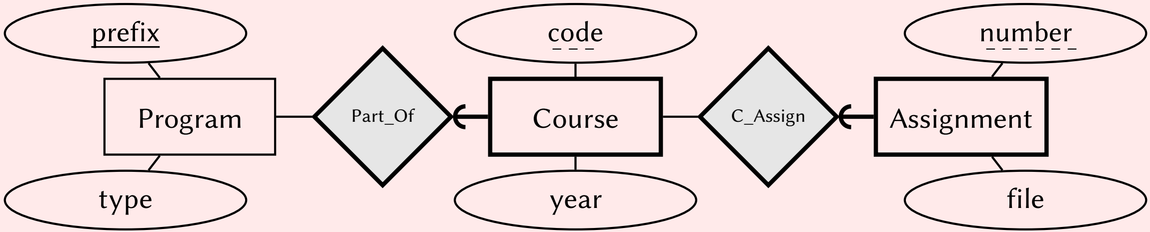
The entity-relationship model: Weak entities



Step 2: The weak entity Course and identifying relationship Part_Of in SQL

```
CREATE TABLE course
(
  prefix VARCHAR(10) NOT NULL REFERENCES program(prefix),
  code VARCHAR(4) NOT NULL,
  year INT NOT NULL,
  PRIMARY KEY(prefix, code)
);
```

The entity-relationship model: Weak entities

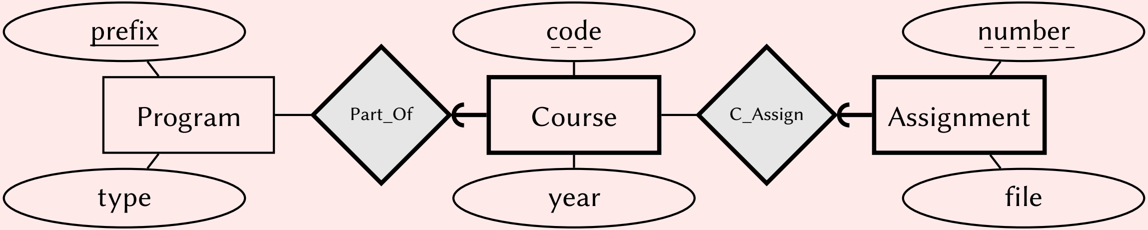


Step 3: The weak entity Assignment and identifying relationship C_Assign

Assignment(<u>prefix</u> : **VARCHAR**(10), <u>code</u> : **VARCHAR**(4), <u>number</u> : **INT**,
file : **BLOB**).

The pair (prefix, code) is a foreign key pointing to the primary key in **Course**.

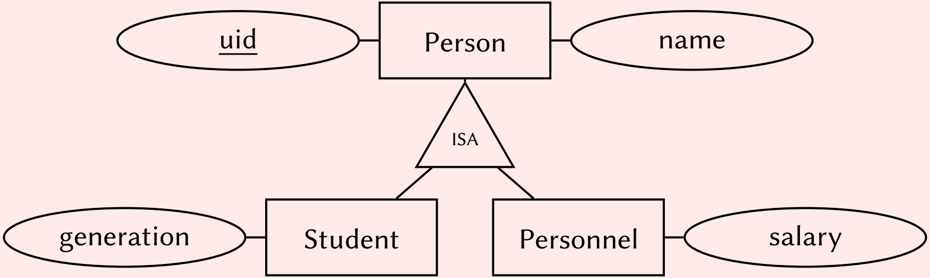
The entity-relationship model: Weak entities



Step 3: The weak entity Assignment and identifying relationship C_Assign in SQL

```
CREATE TABLE assignment (
  prefix VARCHAR(10) NOT NULL, code VARCHAR(4) NOT NULL,
  number INT NOT NULL,
  file BLOB NOT NULL,
  FOREIGN KEY(prefix, code) REFERENCES course(prefix, code),
  PRIMARY KEY(prefix, code, number)
);
```

The entity-relationship model: ISA

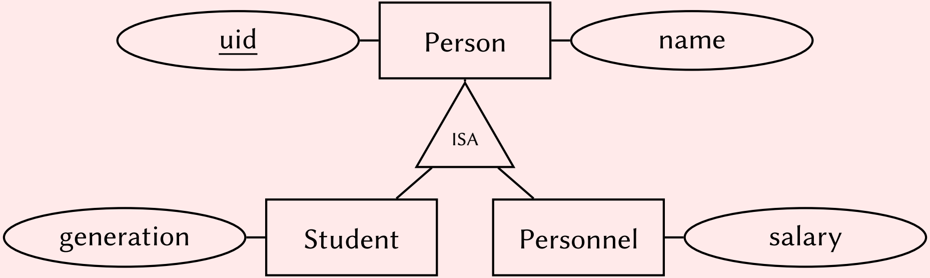


ER method Each entity in the hierarchy: Separate table.
Student personnel will be in the person, student, and personnel table.

OO method Each allowed combination of entities: A separate table.
A student personnel will be in the student_personnel table.

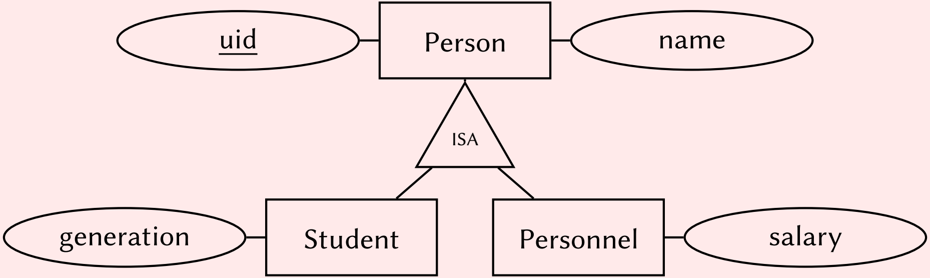
NULL method A single table person with optional attributes for specializations.
A student personnel will be in the person table.

The entity-relationship model: ISA



- Disallowing certain combinations from a ISA-hierarchy in SQL is *in general* tricky.
  E.g., allowing students, but not student personnel.
- **ASSERTION** can technically check these types of constraints.
  ASSERTIONs are often not supported or costly.
- Some DBMSs have native support for table inheritance.

The entity-relationship model: ISAs and the ER Method

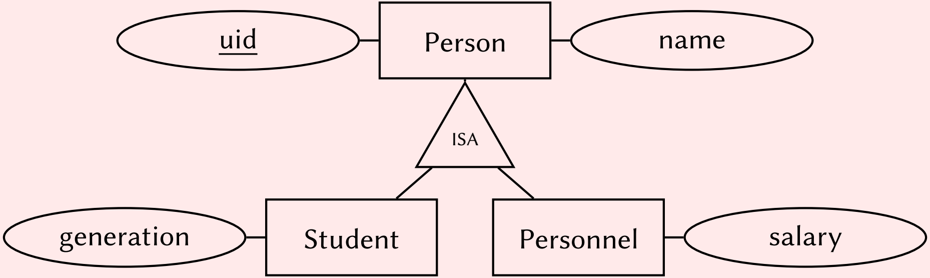


General strategy

- Start with the root of the hierarchy: make it a normal table.
- Specializations have only their own attributes and a primary key.
- The primary key of specialization are foreign keys referring to their parent.

Very flexible and efficient method: typically preferable.

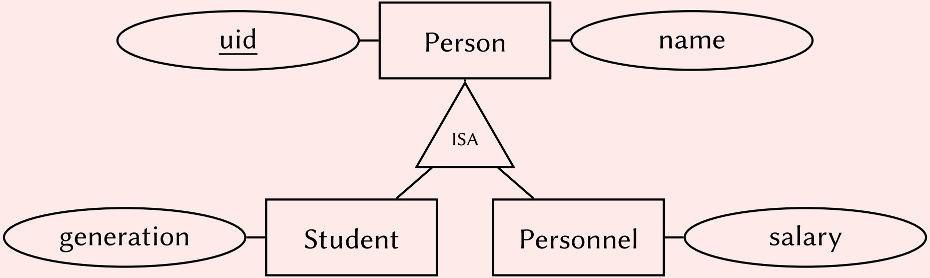
The entity-relationship model: ISAs and the ER Method



Step 1: The entity Person

Person(<u>uid</u> : **INT**, name : **CLOB**).

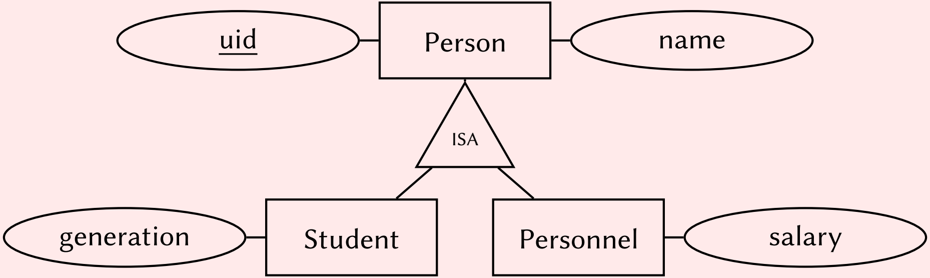
The entity-relationship model: ISAs and the ER Method



Step 1: The entity Person in SQL

```
CREATE TABLE person
(
  uid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  name CLOB NOT NULL
);
```

The entity-relationship model: ISAs and the ER Method

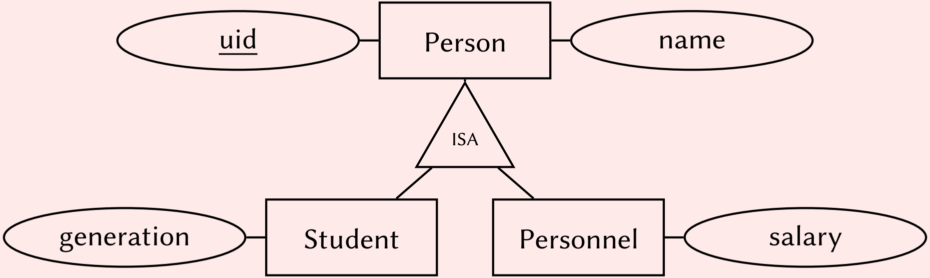


Step 2: The entity Student

Student(<u>uid</u> : **INT**, generation : **INT**).

Uid is a foreign key pointing to the primary key in **Person**.

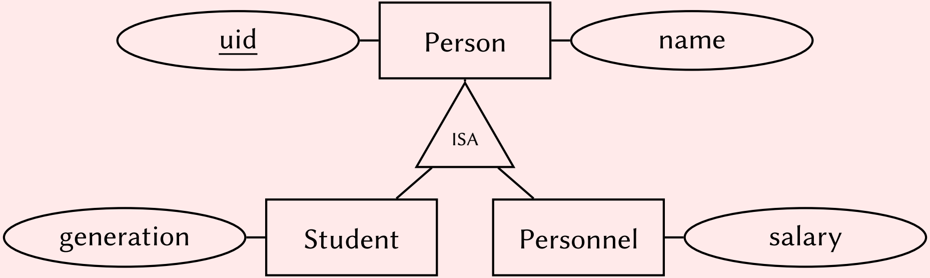
The entity-relationship model: ISAs and the ER Method



Step 2: The entity Student in SQL

```
CREATE TABLE student
(
  uid INT PRIMARY KEY REFERENCES person(uid),
  generation INT NOT NULL
);
```

The entity-relationship model: ISAs and the ER Method

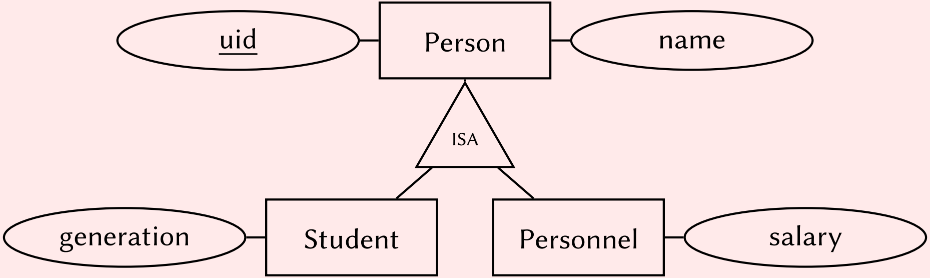


Step 3: The entity Personnel

Personnel(<u>uid</u> : **INT**, salary : **DECIMAL**).

Uid is a foreign key pointing to the primary key in **Person**.

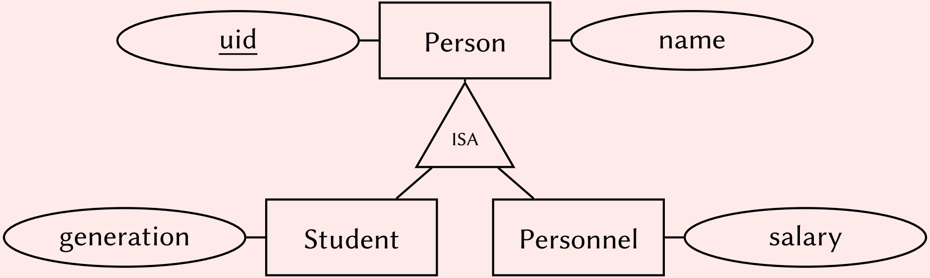
The entity-relationship model: ISAs and the ER Method



Step 3: The entity Personnel in SQL

```
CREATE TABLE personnel
(
  uid INT PRIMARY KEY REFERENCES person(uid),
  salary DECIMAL NOT NULL
);
```

The entity-relationship model: ISAs and the OO Method



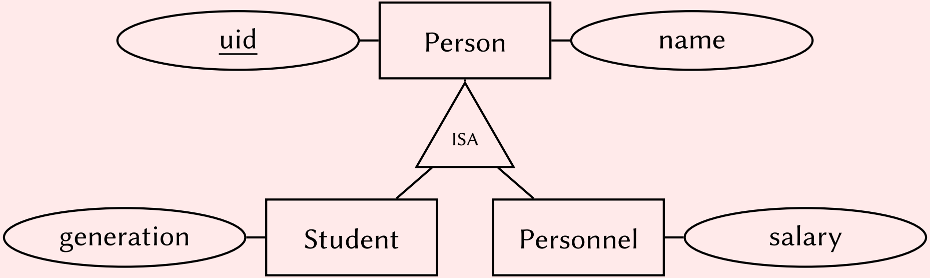
General strategy

Each allowed combination of entities: make a table with all their attributes.

Optimized to select specific combinations from the hierarchy (e.g., all student personnel).

Not optimized to get all entities of a given member in the hierarchy (e.g., all students).

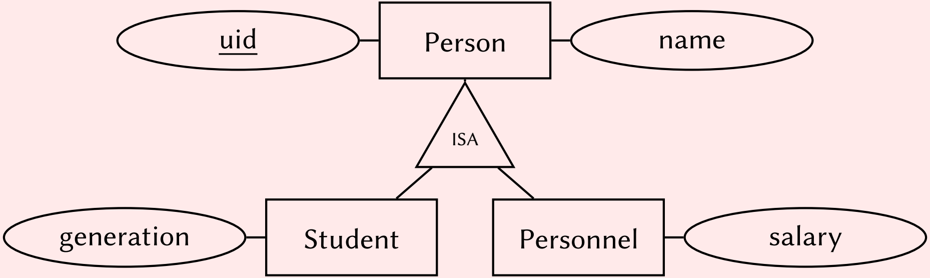
The entity-relationship model: ISAs and the OO Method



Step 1: Determine the valid combinations

- persons that are not students and personnel?
- persons that are only students?
- persons that are only personnel?
- persons that are both students and personnel?

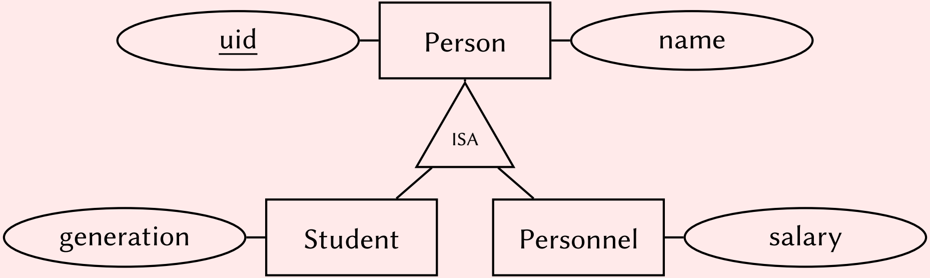
The entity-relationship model: ISAs and the OO Method



Step 1: Determine the valid combinations

- persons that are not students and personnel? ✖
- persons that are only students? ✔
- persons that are only personnel? ✔
- persons that are both students and personnel? ✔

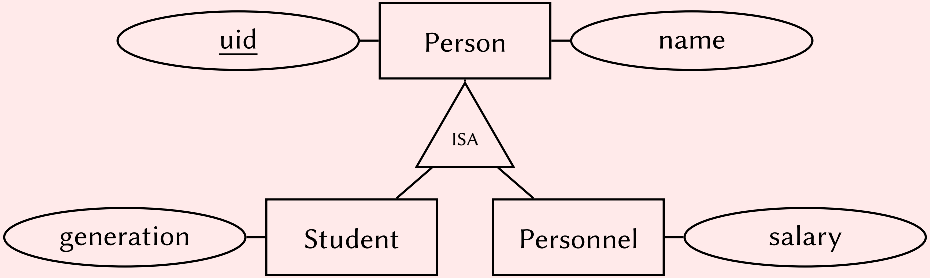
The entity-relationship model: ISAs and the OO Method



Step 2: Persons that are only students

Student(<u>uid</u> : **INT**, name : **CLOB**, generation : **INT**).

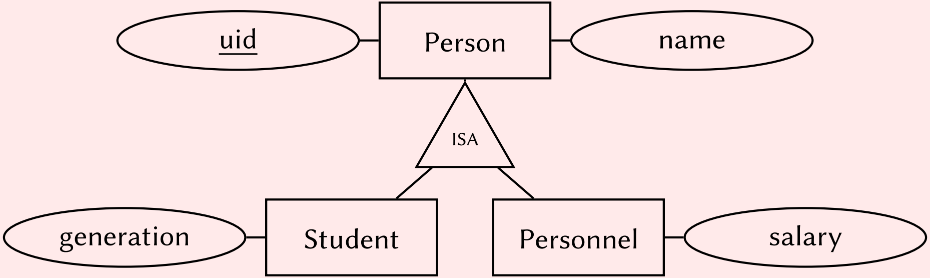
The entity-relationship model: ISAs and the OO Method



Step 2: Persons that are only students in SQL

```
CREATE TABLE student
(
  uid INT PRIMARY KEY,
  name CLOB NOT NULL,
  generation INT NOT NULL
);
```

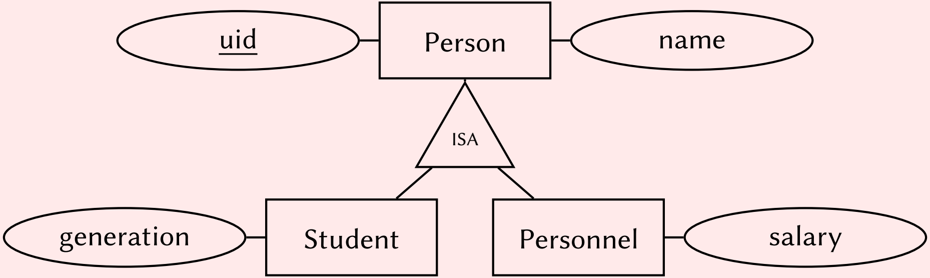
The entity-relationship model: ISAs and the OO Method



Step 3: Persons that are only personnel

Personnel(uid : **INT**, name : **CLOB**, salary : **DECIMAL**).

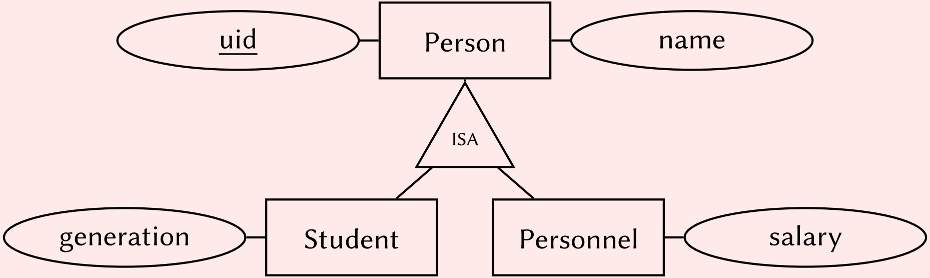
The entity-relationship model: ISAs and the OO Method



Step 3: Persons that are only personnel in SQL

```
CREATE TABLE personnel
(
  uid INT PRIMARY KEY,
  name CLOB NOT NULL,
  salary DECIMAL NOT NULL
);
```

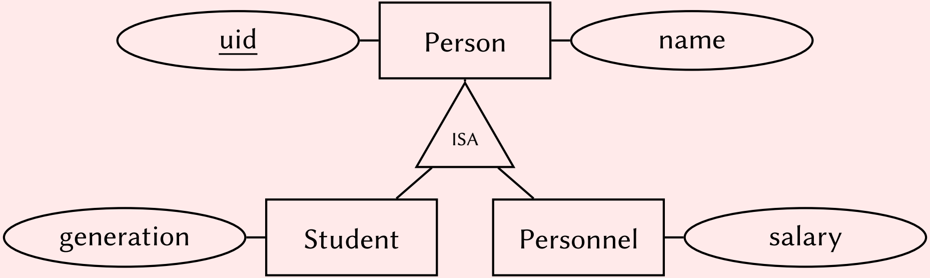
The entity-relationship model: ISAs and the OO Method



Step 4: Persons that are both students and personnel

StudentPersonnel(uid : **INT**, name : **CLOB**, generation : **INT**, salary : **DECIMAL**).

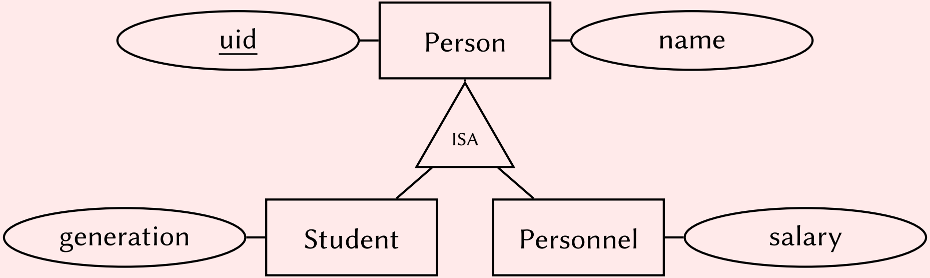
The entity-relationship model: ISAs and the OO Method



Step 4: Persons that are both students and personnel in SQL

```
CREATE TABLE student_personnel
(
  uid INT PRIMARY KEY, name CLOB NOT NULL,
  generation INT NOT NULL,
  salary DECIMAL NOT NULL
);
```

The entity-relationship model: ISAs and the OO Method

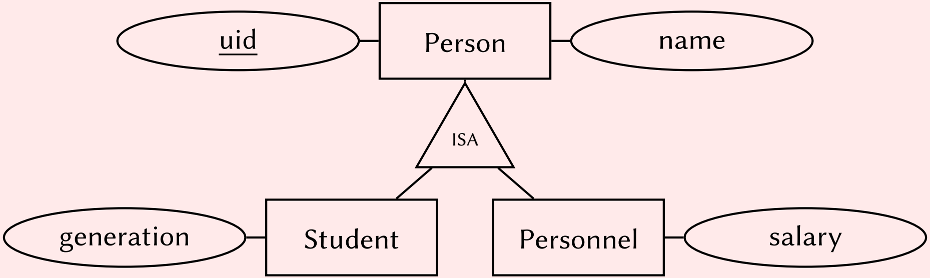


Challenge: Assure unique identifiers across tables

Assure that the tables student, personnel, and student_personnel use different identifiers.

There are no great generic solutions for this challenge (besides **ASSERTION**).

The entity-relationship model: ISAs and the NULL Method



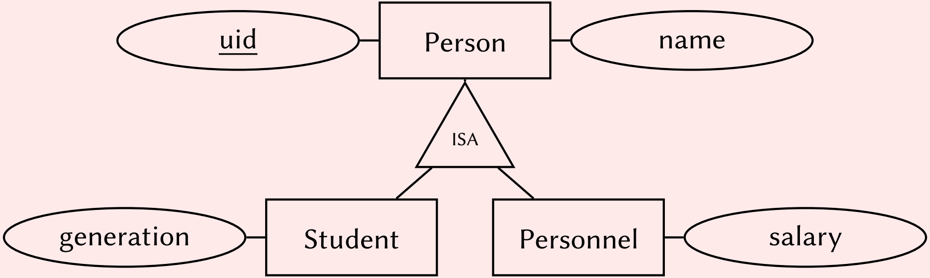
General strategy

A single table person with optional attributes for each specialization.

Optimized to select all data.

Not optimized to get all entities of a given member in the hierarchy (e.g., all students).

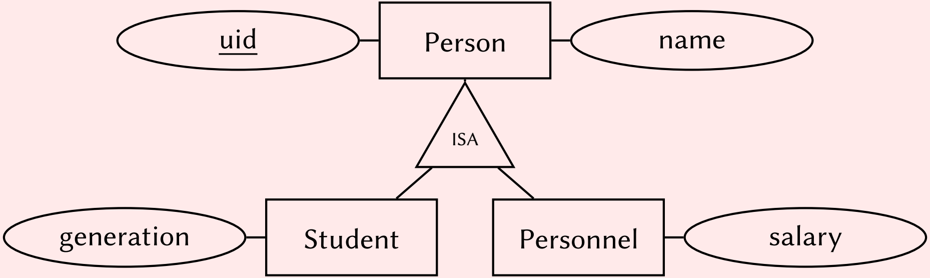
The entity-relationship model: ISAs and the NULL Method



Step 1: The entity person

$$\textbf{person}(\underline{\text{uid}} : \textbf{INT}, \text{name} : \textbf{CLOB}, \text{generation} : \textbf{INT}_{\text{optional}}, \text{salary} : \textbf{DECIMAL}_{\text{optional}}).$$

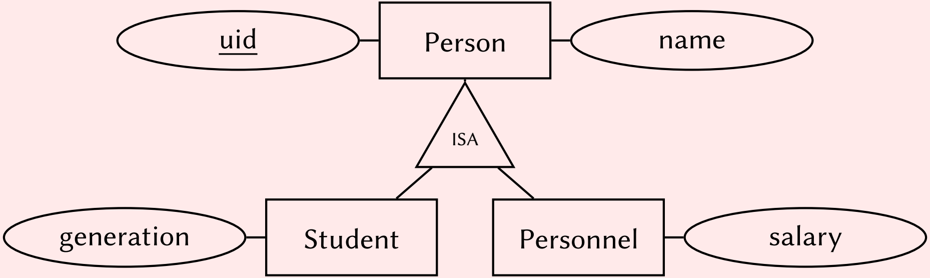
The entity-relationship model: ISAs and the NULL Method



Step 1: The entity person in SQL

```
CREATE TABLE person
(
  uid INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  name CLOB NOT NULL,
  generation INT, salary DECIMAL
);
```

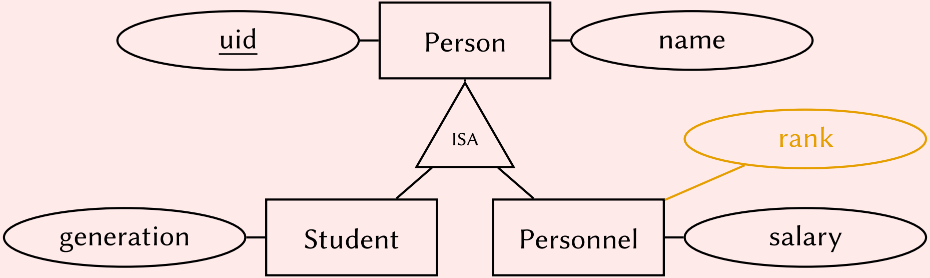
The entity-relationship model: ISAs and the NULL Method



Challenge: Specializations with several attributes

All or no attributes of each specialization must have a value.

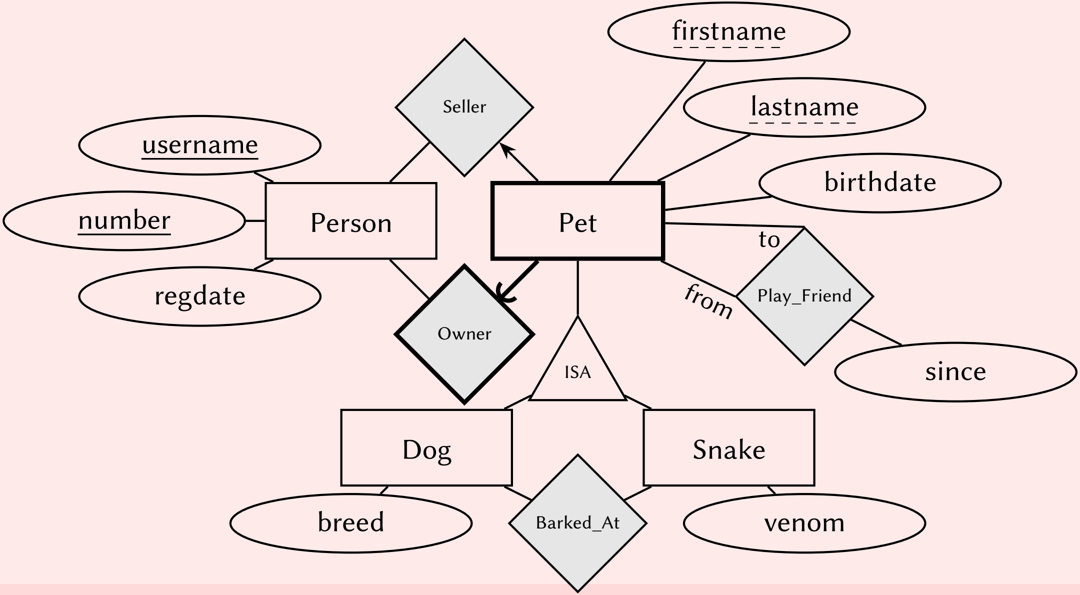
The entity-relationship model: ISAs and the NULL Method



Challenge: Specializations with several attributes

```
CREATE TABLE person
(
  ..., salary DECIMAL, rank INT,
  CHECK((salery IS NULL AND rank IS NULL) OR
        (salery IS NOT NULL AND rank IS NOT NULL))
);
```

A more complicated example



A more complicated example

Step 1: The entity Person

Person(<u>username</u> : **VARCHAR**(20), <u>number</u> : **SMALLINT**, regdate : **TIMESTAMP**).

The column ‘regdate’ will have **CURRENT_TIMESTAMP** as the default.

A more complicated example

Step 1: The entity Person in SQL

```
CREATE TABLE person
(
  username VARCHAR(20) NOT NULL,
  number SMALLINT NOT NULL,
  regdate TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY(username, number)
);
```

A more complicated example

Step 2: The weak entity and ISA hierarchy Pet

We are going to use the ER-method for the Pet ISA.

Just to be clear: dogs cannot be snakes.

We can have pets that are neither (e.g., cats).

A more complicated example

Step 2: The weak entity and ISA hierarchy Pet

$\textbf{Pet}(\underline{\text{oname}} : \textbf{VARCHAR}(20), \underline{\text{onumber}} : \textbf{SMALLINT},$
$\underline{\text{firstname}} : \textbf{VARCHAR}(100), \underline{\text{lastname}} : \textbf{VARCHAR}(100),$
$\text{birthdate} : \textbf{DATE}, \text{sname} : \textbf{VARCHAR}(20)_{\text{optional}}, \text{snumber} : \textbf{SMALLINT}_{\text{optional}})$

The pair (oname, onumber) is a foreign key pointing to an owner (primary key in **Person**).

A more complicated example

Step 2: The weak entity and ISA hierarchy Pet

$\textbf{Pet}(\underline{\text{oname}} : \textbf{VARCHAR}(20), \underline{\text{onumber}} : \textbf{SMALLINT},$
$\underline{\text{firstname}} : \textbf{VARCHAR}(100), \underline{\text{lastname}} : \textbf{VARCHAR}(100),$
$\text{birthdate} : \textbf{DATE}, \text{sname} : \textbf{VARCHAR}(20)_{\text{optional}}, \text{snumber} : \textbf{SMALLINT}_{\text{optional}})$

The pair (oname, onumber) is a foreign key pointing to an owner (primary key in **Person**).

Pets have *at-most* one seller, integrate this relationship.
The pair (sname, snumber) is a foreign key pointing to a seller (primary key in **Person**).

A more complicated example

Step 2: The weak entity and ISA hierarchy Pet in SQL

```
CREATE TABLE pet
(
  oname VARCHAR(20) NOT NULL,
  onumber SMALLINT NOT NULL,
  firstname VARCHAR(100) NOT NULL,
  lastname VARCHAR(100) NOT NULL,
  birthdate DATE NOT NULL,
  sname VARCHAR(20),
  snumber SMALLINT,
  FOREIGN KEY(oname, onumber) REFERENCES person(username, number),
  FOREIGN KEY(sname, snumber) REFERENCES person(username, number),
  PRIMARY KEY(oname, onumber, firstname, lastname)
);
```

A more complicated example

Step 3: The relationship Play_Friend

Play_Friend(<u>foname</u> : **VARCHAR**(20), <u>fonumber</u> : **SMALLINT**,
<u>ffname</u> : **VARCHAR**(100), <u>flname</u> : **VARCHAR**(100),
<u>toname</u> : **VARCHAR**(20), <u>tonumber</u> : **SMALLINT**,
<u>tfname</u> : **VARCHAR**(100), <u>tlname</u> : **VARCHAR**(100),
since : **DATE**).

The tuple (foname, fonumber, ffname, flname) is a foreign key pointing to the *from*-friend (primary key in **Pet**).

The tuple (toname, tonumber, tfname, tlname) is a foreign key pointing to the *to*-friend (primary key in **Pet**).

A more complicated example

Step 3: The relationship Play_Friend in SQL

```
CREATE TABLE play_friend
(
  foname VARCHAR(20) NOT NULL, fonumber SMALLINT NOT NULL,
  ffname VARCHAR(100) NOT NULL, flname VARCHAR(100) NOT NULL,
  toname VARCHAR(20) NOT NULL, tonumber SMALLINT NOT NULL,
  tfname VARCHAR(100) NOT NULL, tlname VARCHAR(100) NOT NULL,
  since DATE NOT NULL,
  FOREIGN KEY(foname, fonumber, ffname, flname)
              REFERENCES pet(oname, onumber, firstname, lastname),
  FOREIGN KEY(toname, tonumber, tfname, tlname)
              REFERENCES pet(oname, onumber, firstname, lastname),
  PRIMARY KEY(foname, fonumber, ffname, flname,
               toname, tonumber, tfname, tlname)
);
```

A more complicated example

Step 4: The entity Dog

Dog(<u>oname</u> : **VARCHAR**(20), <u>onumber</u> : **SMALLINT**,
<u>firstname</u> : **VARCHAR**(100), <u>lastname</u> : **VARCHAR**(100),
breed : **VARCHAR**(100)).

The tuple (oname, onumber, firstname, lastname) is a foreign key pointing to the pet entity this dog belongs to (primary key in **Pet**).

A more complicated example

Step 4: The entity Dog in SQL

```
CREATE TABLE dog
(
  oname VARCHAR(20) NOT NULL,
  onumber SMALLINT NOT NULL,
  firstname VARCHAR(100) NOT NULL,
  lastname VARCHAR(100) NOT NULL,
  breed VARCHAR(100) NOT NULL,
  FOREIGN KEY(oname, onumber, firstname, lastname)
              REFERENCES pet(oname, onumber, firstname, lastname),
  PRIMARY KEY(oname, onumber, firstname, lastname)
);
```

A more complicated example

Step 5: The entity Snake

Snake(<u>oname</u> : **VARCHAR**(20), <u>onumber</u> : **SMALLINT**,
<u>firstname</u> : **VARCHAR**(100), <u>lastname</u> : **VARCHAR**(100),
venom : **BOOLEAN**).

The tuple (oname, onumber, firstname, lastname) is a foreign key pointing to the pet entity this dog belongs to (primary key in **Pet**).

A more complicated example

Step 5: The entity Snake in SQL

```
CREATE TABLE snake
(
  oname VARCHAR(20) NOT NULL,
  onumber SMALLINT NOT NULL,
  firstname VARCHAR(100) NOT NULL,
  lastname VARCHAR(100) NOT NULL,
  venom BOOLEAN NOT NULL,
  FOREIGN KEY(oname, onumber, firstname, lastname)
              REFERENCES pet(oname, onumber, firstname, lastname),
  PRIMARY KEY(oname, onumber, firstname, lastname)
);
```

A more complicated example

Step 6: The relationship Barked_At

Barked_At(<u>doname</u> : **VARCHAR**(20), <u>donumber</u> : **SMALLINT**,
<u>dfname</u> : **VARCHAR**(100), <u>dlname</u> : **VARCHAR**(100),
<u>soname</u> : **VARCHAR**(20), <u>sonumber</u> : **SMALLINT**,
<u>sfname</u> : **VARCHAR**(100), <u>slname</u> : **VARCHAR**(100)).

The tuple (doname, donumber, dfname, dlname) is a foreign key
pointing to the dog that barked (primary key in **Dog**).

The tuple (soname, sonumber, sfname, slname) is a foreign key
pointing to the snake that got barked at (primary key in **Snake**).

A more complicated example

Step 6: The relationship Barked_At in SQL

```
CREATE TABLE barked_at
(
  doname VARCHAR(20) NOT NULL, donumber SMALLINT NOT NULL,
  dfname VARCHAR(100) NOT NULL, dlname VARCHAR(100) NOT NULL,
  soname VARCHAR(20) NOT NULL, sonumber SMALLINT NOT NULL,
  sfname VARCHAR(100) NOT NULL, slname VARCHAR(100) NOT NULL,
  FOREIGN KEY(doname, donumber, dfname, dlname)
              REFERENCES dog(oname, onumber, firstname, lastname),
  FOREIGN KEY(soname, sonumber, sfname, slname)
              REFERENCES snake(oname, onumber, firstname, lastname),
  PRIMARY KEY(doname, donumber, dfname, dlname,
               soname, sonumber, sfname, slname)
);
```

Summary: Column-level constraints in SQL

CREATE TABLE *table name*
(...
  column name **TYPE**
      NOT NULL$_{\texttt{optional}}$

      single-column constraint(s)
...);

Summary: Column-level constraints in SQL

CREATE TABLE *table name*
(...
  column name **TYPE**
    NOT NULL$_{\texttt{optional}}$

    single-column constraint(s)
...);

The *type* of a column restricts the domain and the operations on a column.

NOT NULL further constraints the column by disallowing `NULL` values.

Summary: Column-level constraints in SQL

CREATE TABLE *table name*
(...
  column name **TYPE**
  NOT NULL$_{\texttt{optional}}$

  single-column constraint(s)
...);

PRIMARY KEY each row has a unique not-NULL value in this column.
Typically used to *identify* the row.

UNIQUE each row with a not-NULL value has a unique this column.
If also **NOT NULL**: a *candidate key*.

REFERENCES each not-NULL value in this column points to a unique value in a column.
These *foreign keys* are important when representing relationships.

CHECK additional conditions on the value in the column.
Often used to express *domain constraints*.

Summary: Column-level constraints in SQL

CREATE TABLE *table name*
(. . .
    column name **TYPE**
        NOT NULL$_{\texttt{optional}}$
        CONSTRAINT *name*$_{\texttt{optional}}$
        single-column constraint(s)
. . .);

Summary: Table-level constraints in SQL

Constraints over one-or-more columns are done at the table level:

CREATE TABLE *table name*

(

...

column definitions & *table-level constraint definitions*

...

);

Summary: Table-level constraints in SQL

Constraints over one-or-more columns are done at the table level:

CREATE TABLE *table name*
(
 ...
 column definitions & *table-level constraint definitions*
 ...
);

Table-level constraints can also be named:

...
CONSTRAINT *name table-level constraint*
...

Summary: Table-level constraints in SQL

Constraints over one-or-more columns are done at the table level:

CREATE TABLE *table name*
(
   ...
   column definitions & *table-level constraint definitions*
   ...
);

Table-level constraints can also be named:

   ...
   CONSTRAINT *name table-level constraint*
   ...

Constraints can be **PRIMARY KEY**, **UNIQUE**, **FOREIGN KEY**, or **CHECK**.

A quick overview of other SQL features

- Changing and removing tables and constraints.
- Using views for query shorthands.
- Multi-table constraints.
- Triggers.

A quick overview of other SQL features

- Changing and removing tables and constraints.
- Using views for query shorthands.
- Multi-table constraints.
- Triggers.

…thereby completing our look at the tip of the SQL-Iceberg.

ALTERing and **DROP**ing tables

Tables can be modified via **ALTER TABLE**. E.g.,

- columns can be added, changed, or removed;
- table constraints can be added, changed, or removed; and
- other changes can be made.

What about constraints?

You can **DROP** and re**CREATE** them.

Views: Shorthands for **SELECT** queries

Views can be used to define *shorthands*:
Views act like tables, but are defined by a **SELECT** statement:

> **CREATE VIEW** *view name*
> **AS SELECT** *query* ;

'Simple' views are typically updatable (**INSERT**, **UPDATE**, **DELETE** work properly).

DBMSs have typically options to *materialize* a view.

Multi-table constraints via **CHECK**

A pet dog is not a pet snake.

Multi-table constraints via **CHECK**

A pet dog is not a pet snake.

```
CREATE TABLE dog
(
  dogid INT ... KEY,
  ...


);
```

```
CREATE TABLE snake
(
  snakeid INT ... KEY,
  ...


);
```

Multi-table constraints via **CHECK**

A pet dog is not a pet snake.

```
CREATE TABLE dog
(
  dogid INT ... KEY,
  ...,
  CHECK (dogid NOT IN(
    SELECT snakeid FROM snake))
);
```

```
CREATE TABLE snake
(
  snakeid INT ... KEY,
  ...
);
```

Multi-table constraints via **CHECK**

A pet dog is not a pet snake.

```
CREATE TABLE dog
(
  dogid INT ... KEY,
  ...,
  CHECK (dogid NOT IN(
    SELECT snakeid FROM snake))
);
```

```
CREATE TABLE snake
(
  snakeid INT ... KEY,
  ...
);
```

Issues with this ‘solution’

- **CHECK**s in table *x* are only evaluated when values are inserted or updated in *x*.

Multi-table constraints via **CHECK**

A pet dog is not a pet snake.

```
CREATE TABLE dog
(
  dogid INT ... KEY,
  ...,
  CHECK (dogid NOT IN(
    SELECT snakeid FROM snake))
);
```

```
CREATE TABLE snake
(
  snakeid INT ... KEY,
  ...,
  CHECK (snakeid NOT IN(
    SELECT dogid FROM dog))
);
```

Issues with this 'solution'

- **CHECK**s in table *x* are only evaluated when values are inserted or updated in *x*.

Multi-table constraints via **CHECK**

A pet dog is not a pet snake.

```
CREATE TABLE dog
(
  dogid INT ... KEY,
  ...,
  CHECK (dogid NOT IN(
    SELECT snakeid FROM snake))
);
```

```
CREATE TABLE snake
(
  snakeid INT ... KEY,
  ...,
  CHECK (snakeid NOT IN(
    SELECT dogid FROM dog))
);
```

Issues with this 'solution'

- **CHECK**s in table *x* are only evaluated when values are inserted or updated in *x*.
- **CHECK**s are not evaluated when values are deleted (not a problem in this example).

Multi-table constraints via **CHECK**

A pet dog is not a pet snake.

```
CREATE TABLE dog
(
  dogid INT ... KEY,
  ...,
  CHECK (dogid NOT IN(
    SELECT snakeid FROM snake))
);
```

```
CREATE TABLE snake
(
  snakeid INT ... KEY,
  ...,
  CHECK (snakeid NOT IN(
    SELECT dogid FROM dog))
);
```

Issues with this 'solution'

- **CHECK**s in table x are only evaluated when values are inserted or updated in x.
- **CHECK**s are not evaluated when values are deleted (not a problem in this example).
- Not supported by most DBMSs.

Multi-table constraints via **ASSERTION**s

A pet dog is not a pet snake.

```
CREATE ASSERTION dogsnake
CHECK NOT EXISTS (
    SELECT dogid FROM dog
  INTERSECT
    SELECT snakeid FROM snake);
```

Multi-table constraints via **ASSERTION**s

A pet dog is not a pet snake.

```
CREATE ASSERTION dogsnake
CHECK NOT EXISTS (
    SELECT dogid FROM dog
  INTERSECT
    SELECT snakeid FROM snake);
```

ASSERTIONs are conditions that must always hold $\longrightarrow$
powerful tool for expressing multi-table constraints.

Multi-table constraints via ASSERTIONs

A pet dog is not a pet snake.

```
CREATE ASSERTION dogsnake
CHECK NOT EXISTS (
    SELECT dogid FROM dog
  INTERSECT
    SELECT snakeid FROM snake);
```

ASSERTIONs are conditions that must always hold ⟶
powerful tool for expressing multi-table constraints.

Issues with this ‘solution’

- Not supported by most DBMSs.

Triggers: The manual way to maintain constraints

Triggers allow for event-driven active databases.

- Triggers are activated by *database events*.
  E.g., changes due to **INSERT**, **UPDATE**, or **DELETE**.
- Triggers can specify an optional *condition* under which they are executed.
- Triggers can perform an *action* that performs arbitrary operations on the database.

Triggers: The manual way to maintain constraints

Triggers allow for event-driven active databases.

- Triggers are activated by *database events*.
  E.g., changes due to **INSERT**, **UPDATE**, or **DELETE**.
- Triggers can specify an optional *condition* under which they are executed.
- Triggers can perform an *action* that performs arbitrary operations on the database.

Triggers are typically very low-level and can be written in DBMS-specific languages.

Triggers: The manual way to maintain constraints

Triggers allow for event-driven active databases.

- Triggers are activated by *database events*.
  E.g., changes due to **INSERT**, **UPDATE**, or **DELETE**.
- Triggers can specify an optional *condition* under which they are executed.
- Triggers can perform an *action* that performs arbitrary operations on the database.

Triggers are typically very low-level and can be written in DBMS-specific languages.

Can be used to *implement* constraints, but are hard to decipher and debug.

Triggers: The manual way to maintain constraints

Triggers allow for event-driven active databases.

- Triggers are activated by *database events*.
  E.g., changes due to **INSERT**, **UPDATE**, or **DELETE**.
- Triggers can specify an optional *condition* under which they are executed.
- Triggers can perform an *action* that performs arbitrary operations on the database.

Triggers are typically very low-level and can be written in DBMS-specific languages.

Can be used to *implement* constraints, but are hard to decipher and debug.

Trigger development is tricky: Triggers can activate other triggers (*recursive triggers*).

A final note

- *Query* at `https://xkcd.com/1409/`.
- *Exploits of a Mom* at `https://xkcd.com/327/`.
  It is not funny that this comic is a real-world issue.
- *Eventual Consistency* at `https://xkcd.com/2315/` is for later!