# COMP3311 24T1 Database Systems

week 2 - 1

# Outline

- **Announcements**
- ER-Relational Mapping
- SQL Introduction

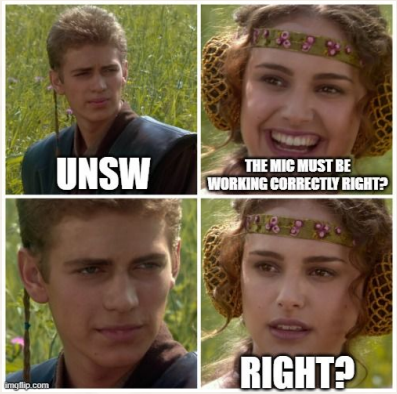
# Announcement 1

- Quiz 1
  - Activities | COMP3311 24T1 | WebCMS3 (unsw.edu.au)
  - You can try as many times as possible
  - before midnight Friday, 23rd Feb (11:59:59 pm)

# Announcement 2

- Exam Hurdle: You need to score 40% in the final exam & have a 50% overall score to pass this course.

- Lecturing Hurdle: some other courses using this LT have the same issue.
  - COMP1521, MATH1081

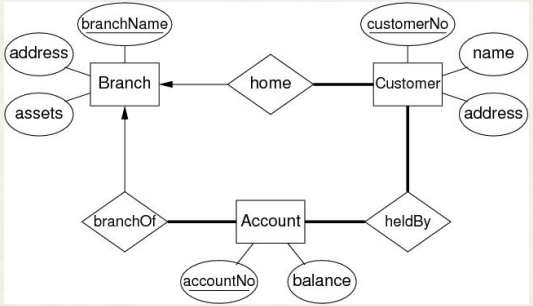


# Outline

- Announcements
- **ER-Relational Mapping**
- SQL Introduction

# ER vs Relational Models - Recap

Correspondence between ER and Relational models:



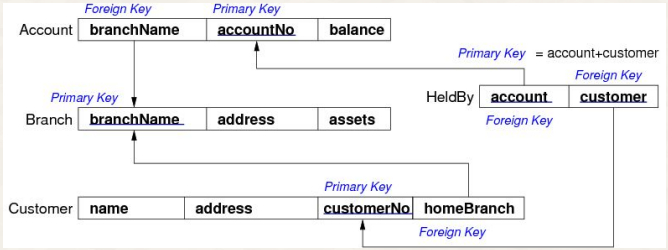

**Relational attributes** correspond to **ER attributes**

- although ER attributes generally don't have explicit domains

**Relational tuples** correspond to **ER entities**

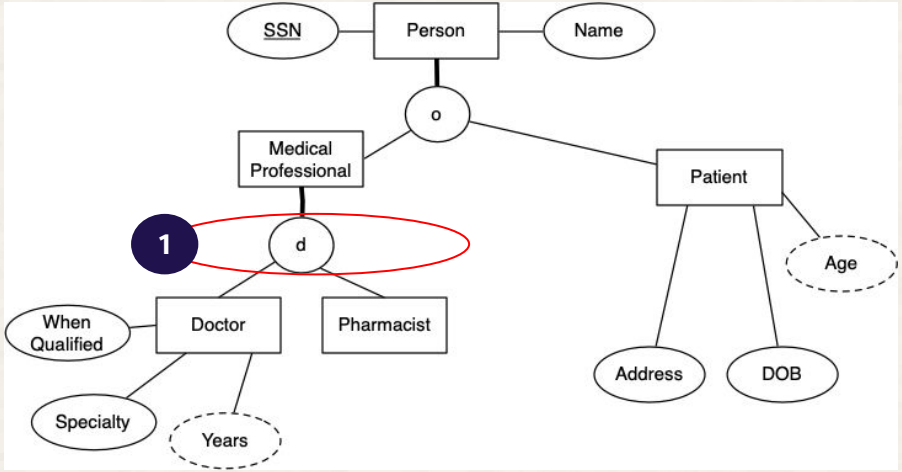
**Relations** correspond to **sets of ER entities**

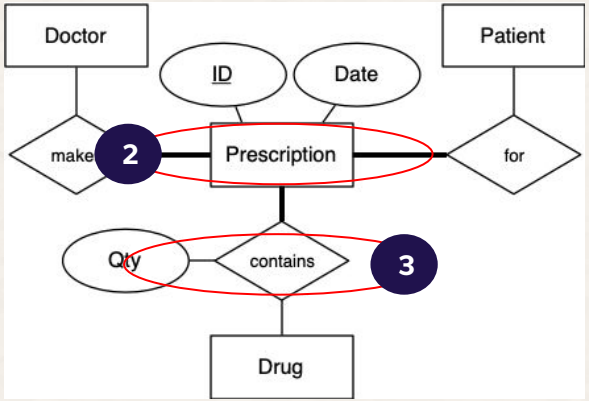
**Relations** also correspond **ER relationships**



# ER Diagram – Recap & Warm-up

Try to describe!



# Relational Modelling - Recap & Warm-up

Try to answer!

What does a relation have?

- Relation name + a set of attributes

What does an attribute have?

- Attribute name + domain

What is a tuple?

- a list of values

(2,3) = (3,2) ?

- (2,3) ≠ (3,2)

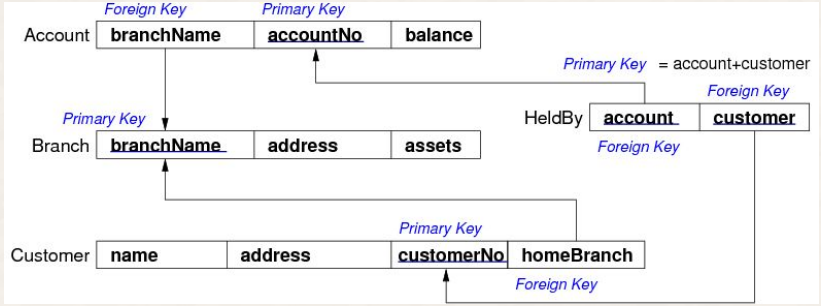
What is an instance of a relation?

- a set of tuples

{ (a,b), (c,d) } = { (c,d), (a,b) } ?

- { (a,b), (c,d) } = { (c,d), (a,b) }

How do we deal with referential integrity constraints?



# ER-Relational Mapping

**a useful strategy for database design**:

- perform initial data modelling using ER (conceptual-level modelling)
- transform conceptual design into relational model (implementation-level modelling)

A formal mapping exists for ER model → Relational model.

This maps "structures"; but additional info is needed, e.g. concrete **domains** for attributes and other **constraints**

# ER-Relational Mapping

**Correspondences** between relational and ER data models:

attribute(ER) ≅ attribute(Rel),  entity(ER) ≅ tuple(Rel)
entity set(ER) ≅ relation(Rel),  relationship(ER) ≅ relation(Rel)

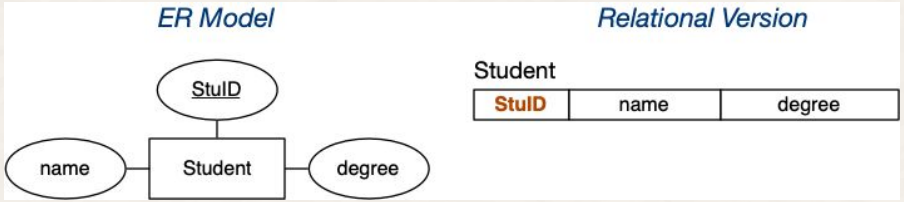
**Differences** between relational and ER models:
Rel uses **relations** to model entities and relationships
Rel has **no composite or multi-valued attributes** (only atomic)
Rel has **no object-oriented notions** (e.g. subclasses, inheritance)

# Mapping *Strong Entities*

An entity set E with atomic attributes $a_1$, $a_2$, ... $a_n$

maps to

A relation R with attributes (columns) $a_1$, $a_2$, ... $a_n$

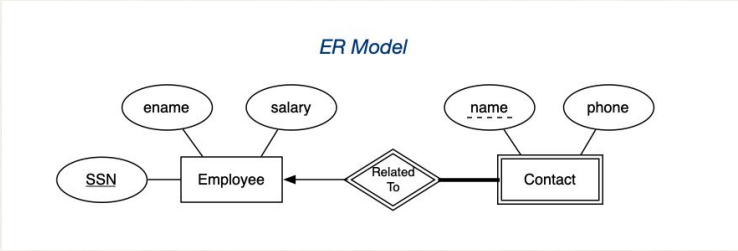


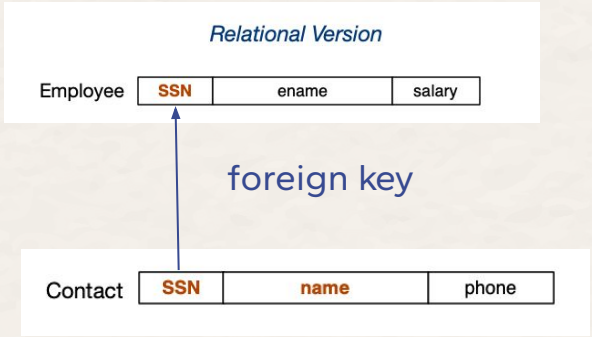

Note: the key is preserved in the mapping.

# Mapping *Weak Entities*

What should the “Employee” relation look like?

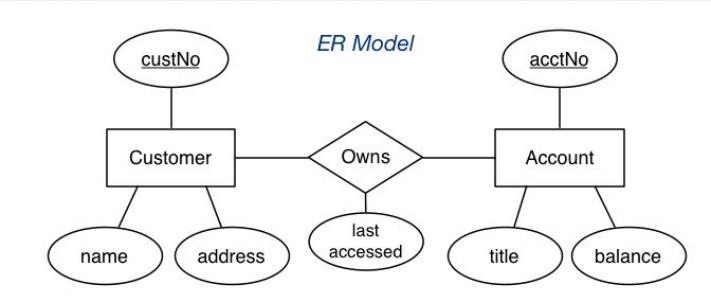
What about “Contact”?



# Mapping *N:M Relationships*

How to map N:M (many-to-many) relationships?




First, let's come up with the relations/tables for the entities.

*Relational Version*

| | | | |
|---|---|---|---|
| Customer | **custNo** | name | address |
| Account | **acctNo** | title | balance |

Then, let's come up with the relation/table for the relationship.

| | | | |
|---|---|---|---|
| Owns | **acctNo** | **custNo** | lastAccessed |

primary keys + attributes

# Mapping *1:N Relationships*

How to map 1:N (1-to-many) relationships?

Is it similar to N:M relationships?

Customer

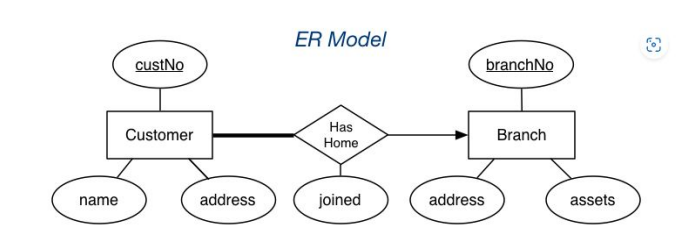
| custNo | name | address |
|---|---|---|

Branch

| branchNo | address | assets |
|---|---|---|

HasHome

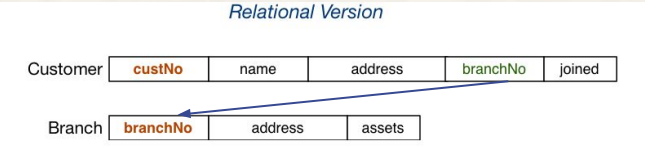
| custNo | branchNo | joined |
|---|---|---|




Is HasHome really needed as a separate table? Can we merge it into another table? If so, which one should we use?

# of HasHome depends on which entity?

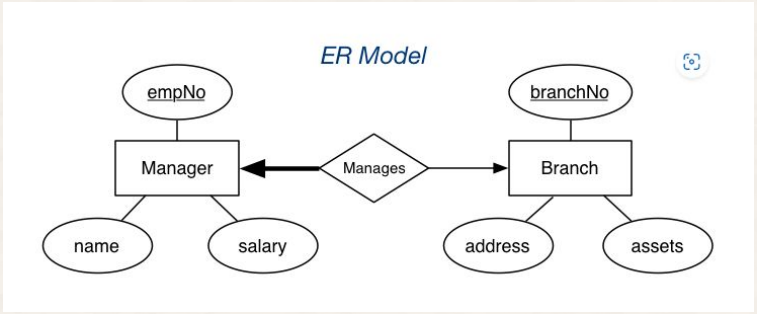
# of HasHome = # of Customer



# Mapping *1:1 Relationships*

How to map 1:1 (1-to-1) relationships?

Is it similar to N:M relationships?




Manager

| empNo | name | salary |
|---|---|---|

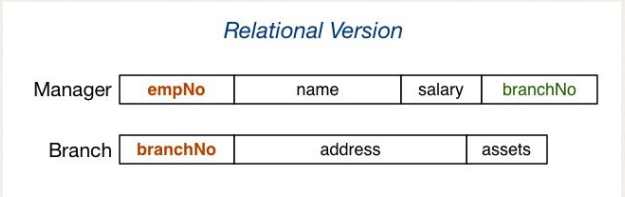
Branch

| branchNo | address | assets |
|---|---|---|

Manages

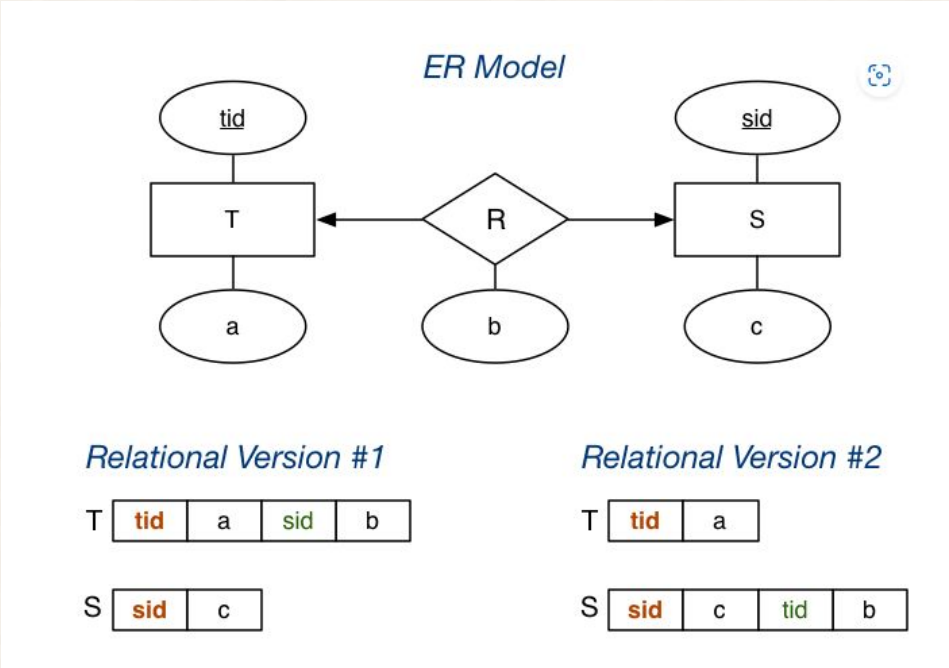
| empNo | branchNo |
|---|---|

Can we merge the relationship table?
like what we did for 1:N relationship.



# Mapping *1:1 Relationships* (cont)

If there is no reason to favour one side of the relationship ...



# Mapping *n-way Relationships*

Relationship mappings above assume binary relationship.

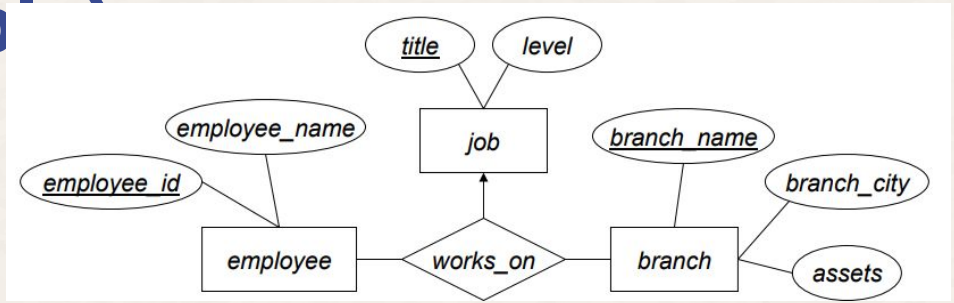
If multiple entities are involved:

- n:m generalises naturally to n:m:p:q
  - include foreign key for each participating entity
  - include any other attributes of the relationship
- other multiplicities (e.g. 1:n:m) ...
  - need to be mapped the same as n:m:p:q
  - so not quite an accurate mapping of the ER

Some people advocate converting n-way relationships into:

- a new entity, and a set of n binary relationships

# Mapping *n-way Relationships* (examp




What tables can we have for the entities?

*job*(*title*, *level*)
*employee*(*employee_id*, *employee_name*)
*branch*(*branch_name*, *branch_city*, *assets*)

What should the "works_on" table have as columns? (what attributes)

*works_on*(*employee_id*, *branch_name*, *title*)

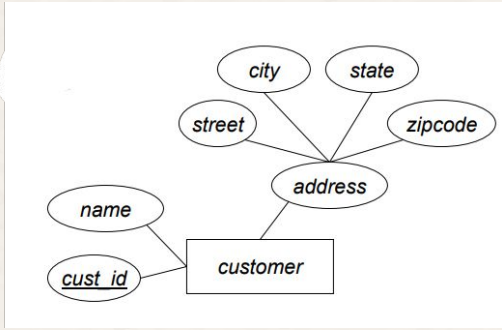
Note: title is not included in the primary key here.

# Mapping *n-way Relationships* (keys)

- If no-arrow (many-to-many):
  - primary key for the relationship is the union of all participating entity-sets' primary keys
- If one-arrow (one-to-many):
  - primary key for the relationship is the union of primary keys of entity-sets without an arrow
- Don't allow more than one arrow!
  - may think about redesigning the ER diagram

# Mapping *Composite Attributes*

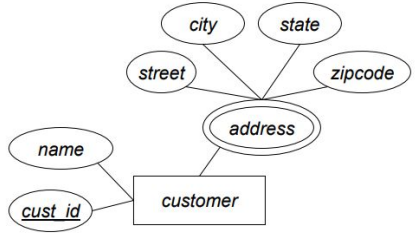
Composite attributes are mapped by concatenation or flattening.




*customer*(*cust_id*, *name*, *street*, *city*, *state*, *zipcode*)

# Mapping *Multi-valued Attributes* (MVAs)

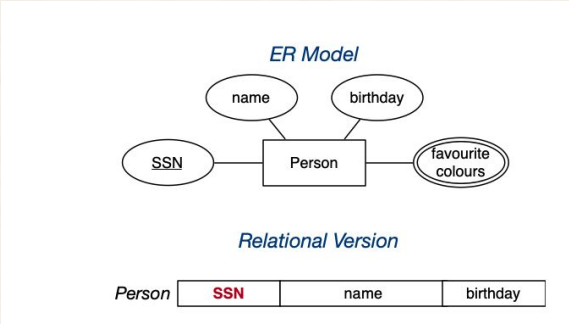
MVAs are mapped by a new table linking values to their entity.

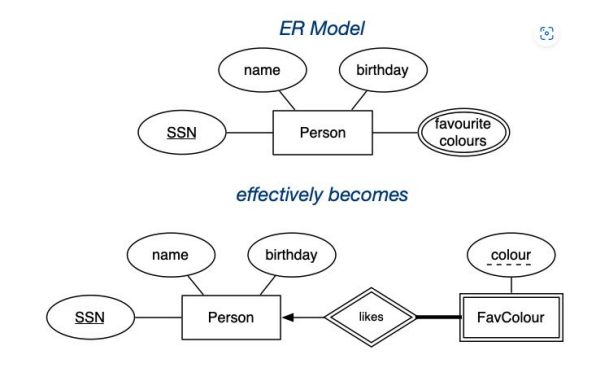



*customer*(*cust_id*, *name*)
*cust_addrs*(*cust_id*, *street*, *city*, *state*, *zipcode*)

# Mapping *Multi-valued Attributes* (MVAs)

Another example.




What should the “favColors” table have?

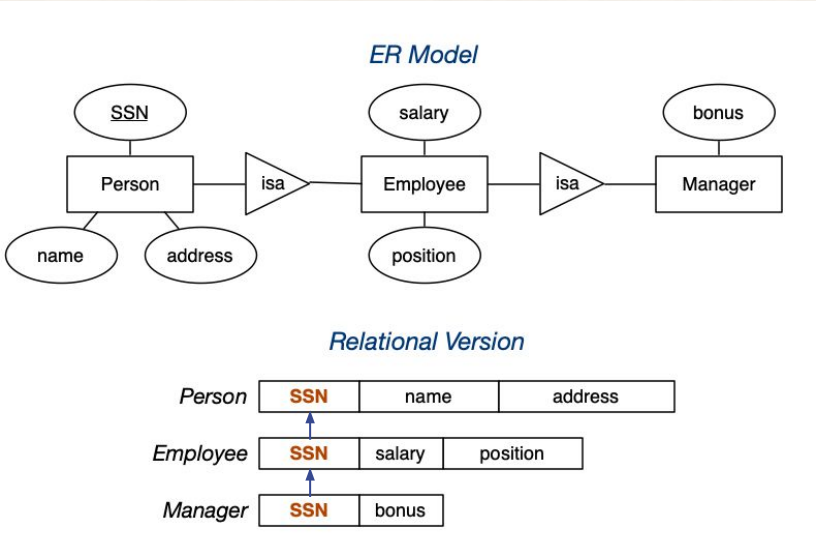
# Mapping *Subclasses*

Three different approaches to mapping subclasses to tables:

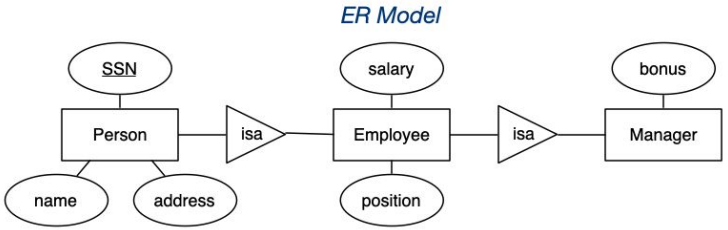
- ER style
  - each entity becomes a separate table,
  - containing attributes of subclass + FK to superclass table
- object-oriented
  - each entity becomes a separate table,
  - inheriting all attributes from all superclasses
- single table with nulls
  - whole class hierarchy becomes one table,
  - containing all attributes of all subclasses (null, if unused)

Which mapping is best depends on how data is to be used.

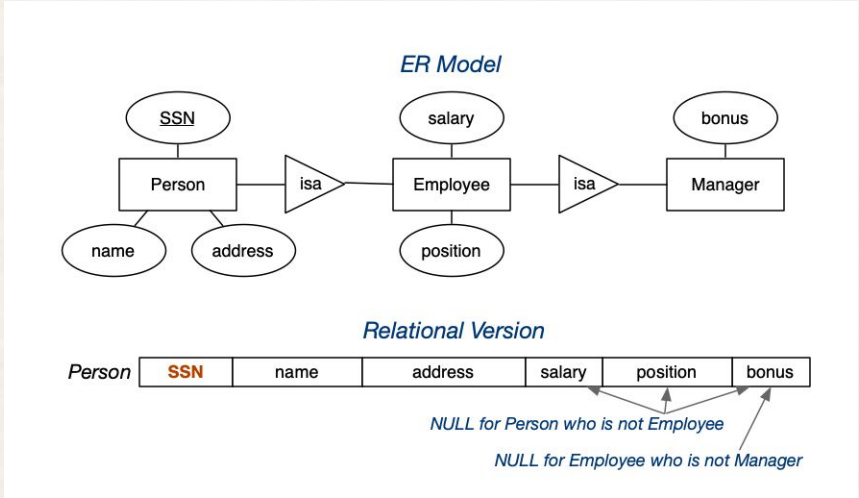
# Mapping *Subclasses – ER style*



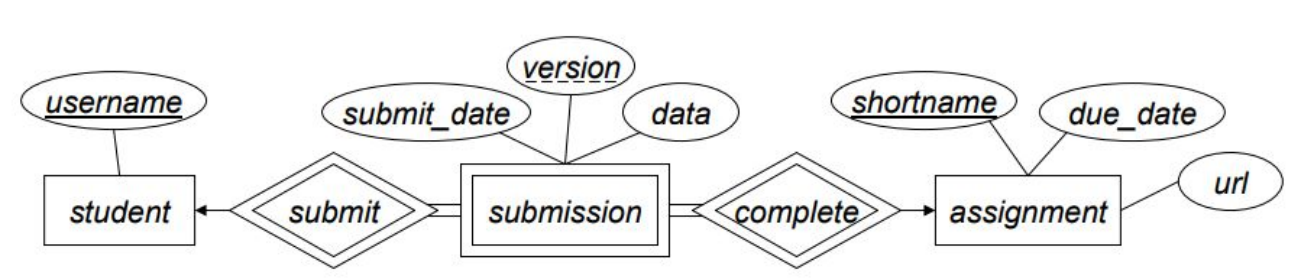
# Mapping *Subclasses – OO style*



# Mapping *Subclasses – single table style*



# Exercise: Mapping Weak Entity




How to describe the two relations?

**Every** submission is submitted by **one** student.
A student **may** submit **one or more** submissions.

**Every** submission completes **one** assignment.
An assignment **may** be completed by **one or more** submissions.

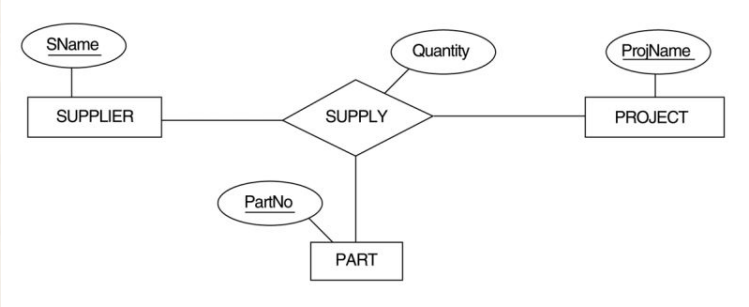
What are the tables for the strong entities?

*student*(<u>*username*</u>)
*assignment*(<u>*shortname*</u>, *due_date*, *url*)

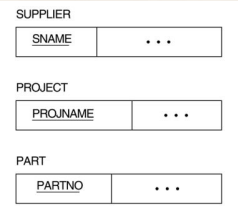
What's the table for the weak entities?

*submission*(<u>*username*, *shortname*, *version*</u>,
*submit_date*, *data*)

# Exercise: n-ary relationships




What are the tables for the entities?




What's the table for the relationship?

# Outline

- Announcements
- ER-Relational Mapping
- **SQL Introduction**

# SQL vs Relational Model

- The relational model is a formal system for
  - describing data (relations, tuples, attributes, domains, constraints)
  - manipulating data (relational algebra ... covered elsewhere)
- SQL is a "programming" language for
  - describing data (tables, rows, fields, types, constraints)
  - manipulating data (query language)

More details:
https://runestone.academy/ns/books/published/practical_db/PART3_RELATIONAL_DATABASE_THEORY/04-sql-vs-theory/sql-vs-theory.html

# SQL History

Developed at IBM in the mid-1970's (System-R)

- Standardised in 1986, and then in 1989, 1992, 1999, 2003, ... 2019

- Many database management systems (DBMSs) have been built around SQL
    - System-R, Oracle, Ingres, DB2, PostgreSQL, MySQL, SQL-server, SQLite, ...

DBMSs vs the standard

- all DBMSs implement a subset of the 1999 standard (aka SQL3)
- all DBMSs implement proprietary extensions to the standard

Conforming to standard should ensure portability of database applications

# SQL Intro

SQL has several sub-languages ...

- **meta-data** definition language (e.g. create table, etc.)
- **meta-data** update language (e.g. alter table, drop table)
- **data** update language (e.g. insert, update, delete)
- **query** language (e.g. select ... from ... where, etc.)

Meta-data languages manage the **database schema**

Data update language manages **sets of tuples**

# SQL Intro

Syntax-wise, SQL is similar to other programming languages

- has keywords, identifiers, constants, operators
- but strings are different to most PLs
  - '...' are constant strings, e.g. 'a', 'abc123', 'John''s bag'
  - "..." allow non-alpha chars in identifiers and make id's case-sensitive

In the **standard**, **all non-quoted identifiers** map to **all upper-case**
e.g. BankBranches = bankbranches are treated as BANKBRANCHES

In **PostgreSQL**, **all non-quoted identifiers** map to **all lower-case**
e.g. BankBranches = BANKBRANCHES are treated as bankbranches

In all standards-adhering DBMSs, different quoted identifiers are different
"BankBranches" ≠ "bankbranches" ≠ "BANKBRANCHES"

# SQL Syntax in a Nutshell

SQL definitions, queries and statements are composed of:

comments ... -- comments to end of line

identifiers ... similar to regular programming languages

keywords ... a large set (e.g. CREATE, DROP, TABLE)

data types ... small set of basic types (e.g. integer, date)

operators ... similar to regular programming languages

constants ... similar to regular programming languages

- Similar means "often the same, but not always" ...



Try it: PostgreSQL - OneCompiler - Write, run and share PostgreSQL code online

# SQL Syntax in a Nutshell

Comments: everything after -- is a comment

Identifiers: alphanumeric (a la C), but also "An Identifier"

Reserved words: many e.g. CREATE, SELECT, TABLE, ...

Reserved words cannot be used identifiers unless quoted e.g "table"

Strings: e.g. 'a string', 'don''t ask', but no '\n' (use e'\n')

Numbers: like C, e.g. 1, -5, 3.14159, ...

Types: integer, float, char(n), varchar(n), date, ...

Operators: =, <>, <, <=, >, >=, AND, OR, NOT, ...

Try it: PostgreSQL - OneCompiler - Write, run and share PostgreSQL code online

# Names in SQL

**Identifiers** denote:

- database objects such as tables, attributes, views, ...
- meta-objects such as types, functions, constraints, ...

**Naming conventions** that I (try to) use in this course:

- **relation names**: e.g. Branches, Students, ... (use plurals)
- **attribute names**: e.g. name, code, firstName, ...
- **foreign keys**: named after either or both of
  - table being referenced e.g. staff or staff_id, ...
  - relationship being modelled e.g. teaches, ...

We initially write SQL keywords in all upper-case in slides.

# Types/Constants in SQL - Numeric

| Name | Storage Size | Description | Range |
|---|---|---|---|
| `smallint` | 2 bytes | small-range integer | -32768 to +32767 |
| `integer` | 4 bytes | typical choice for integer | -2147483648 to +2147483647 |
| `bigint` | 8 bytes | large-range integer | -9223372036854775808 to +9223372036854775807 |
| `decimal` | variable | user-specified precision, exact | up to 131072 digits before the decimal point; up to 16383 digits after the decimal point |
| `numeric` | variable | user-specified precision, exact | up to 131072 digits before the decimal point; up to 16383 digits after the decimal point |
| `real` | 4 bytes | variable-precision, inexact | 6 decimal digits precision |
| `double precision` | 8 bytes | variable-precision, inexact | 15 decimal digits precision |
| `smallserial` | 2 bytes | small autoincrementing integer | 1 to 32767 |
| `serial` | 4 bytes | autoincrementing integer | 1 to 2147483647 |
| `bigserial` | 8 bytes | large autoincrementing integer | 1 to 9223372036854775807 |

arbitrary numeric data:

```
NUMERIC(precision, scale)
```

e.g., the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

https://www.postgresql.org/docs/current/datatype-numeric.html

# Types/Constants in SQL - Char/String

| Name | Description |
|---|---|
| `character varying(n)`, `varchar(n)` | variable-length with limit |
| `character(n)`, `char(n)`, `bpchar(n)` | fixed-length, blank-padded |
| `bpchar` | variable unlimited length, blank-trimmed |
| `text` | variable unlimited length |

PostgreSQL provides extended strings containing \ escapes, e.g:
`E'\n'` `E'O\'Brien'` `E'[A-Z]{4}\\d{4}'` `E'John'`

Type-casting via `Expr::Type` (e.g. `'10'::integer`)

Try it: PostgreSQL - OneCompiler - Write, run and share PostgreSQL code online

https://www.postgresql.org/docs/current/datatype-character.html

# Types/Constants in SQL - Logical

Logical type: BOOLEAN, TRUE and FALSE (or true and false)

PostgreSQL also allows 't', 'true', 'yes', 'f', 'false', 'no'

```
-- create
CREATE TABLE EMPLOYEE (
  empId INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  dept TEXT NOT NULL,
  retired BOOLEAN default 'no'
);

-- insert
INSERT INTO EMPLOYEE VALUES (0001, 'Clark', 'Sales');
INSERT INTO EMPLOYEE VALUES (0002, 'Dave', 'Accounting');
INSERT INTO EMPLOYEE VALUES (0003, 'Ava', 'Sales', true);

-- fetch
SELECT * FROM EMPLOYEE WHERE retired = false;
```

Try it: PostgreSQL - OneCompiler - Write, run and share PostgreSQL code online

# Types/Constants in SQL - Time

| Name | Storage Size | Description | Low Value | High Value | Resolution |
|---|---|---|---|---|---|
| `timestamp [ (p) ] [ without time zone ]` | 8 bytes | both date and time (no time zone) | 4713 BC | 294276 AD | 1 microsecond |
| `timestamp [ (p) ] with time zone` | 8 bytes | both date and time, with time zone | 4713 BC | 294276 AD | 1 microsecond |
| `date` | 4 bytes | date (no time of day) | 4713 BC | 5874897 AD | 1 day |
| `time [ (p) ] [ without time zone ]` | 8 bytes | time of day (no date) | 00:00:00 | 24:00:00 | 1 microsecond |
| `time [ (p) ] with time zone` | 12 bytes | time of day (no date), with time zone | 00:00:00+1559 | 24:00:00-1559 | 1 microsecond |
| `interval [ fields ] [ (p) ]` | 16 bytes | time interval | -178000000 years | 178000000 years | 1 microsecond |

Subtraction of timestamps yields an interval

https://www.postgresql.org/docs/current/datatype-datetime.html


# Types/Constants in SQL - Others

PostgreSQL also has a range of non-standard types, e.g.

- geometric (point/line/...), currency, IP addresses, JSON, XML, objectIDs, ...
- non-standard types typically use string literals ('...') which need to be interpreted

https://www.postgresql.org/docs/current/datatype.html

# Types/Constants in SQL - User-defined

-- domains: constrained version of existing type

```
CREATE DOMAIN Name AS Type CHECK ( Constraint )
```

-- tuple types: defined for each table

```
CREATE TYPE Name AS ( AttrName AttrType, ... )
```

-- enumerated type: specify elements and ordering

```
CREATE TYPE Name AS ENUM ( 'Label', ... )
```

# Types/Constants in SQL - User-defined (Examples)

```
-- positive integers
CREATE DOMAIN PosInt AS integer CHECK (value > 0);

-- a UNSW course code
CREATE DOMAIN CourseCode AS char(8)
   CHECK (value ~ '[A-Z]{4}[0-9]{4}');

-- a UNSW student/staff ID
CREATE DOMAIN ZID AS integer
   CHECK (value betweem 1000000 and 9999999);

-- standard UNSW grades (FL,PS,CR,DN,HD)
CREATE DOMAIN Grade AS char(2)
   CHECK (value in ('FL','PS','CR','DN','HD'));
-- or
CREATE TYPE Grade AS ENUM ('FL','PS','CR','DN','HD');
```

# Tuple and Set Literals

Tuple and set constants are both written as:
( val1, val2, val3, ... )

The correct interpretation is worked out from the context.

```
INSERT INTO Student(studeID, name, degree)
   VALUES (2177364, 'Jack Smith', 'BSc')
  -- tuple literal

CONSTRAINT CHECK gender IN ('male','female','unspecified')

   -- set literal
```

# Thank you!