COMP3311 24T1 Database Systems

week 2 - 2

Outline

- **Announcements**
- SQL Expressions
- SQL Meta-Data Definition Language
- Managing RDBMS
- Mapping ER->Rel->SQL

Announcement 1

- Quiz 1
  - Activities | COMP3311 24T1 | WebCMS3 (unsw.edu.au)
  - You can try as many times as possible
  - before midnight Saturday, 24th Feb (11:59:59 pm), TRY to do it before 8pm Friday

Power Outage (CSE building): 8pm Friday - 8pm Saturday

Announcement 2

How to learn during this stage of the course? 🧐

A long time ago, in a university far far away ...

When I was an undergraduate student ...

- Don't be afraid of the overwhelming concepts/syntax/keywords to remember
- The lectures are to "store" some **keywords** into your mind
- Practice A LOT! (lecture exercises, tutorials, assignment ...)
- Google A LOT! (with the **keywords** in your mind | stackoverflow, chatgpt, ed forum ...)

Outline

- Announcements
- **SQL Expressions**
- SQL Data Definition Language
- Managing RDBMS
- Mapping ER->Rel->SQL

Expressions in SQL

Expressions in SQL involve: **objects**, **constants**, **operators**

- objects are typically names of attributes (or PLpgSQL variables)
- operators may be symbols (e.g. +, =) or keywords (e.g. between)

SQL constants are similar to typical programming language constants

- integers: 123, -5; floats: 3.14, 1.0e-3; boolean: true, false

But strings are substantially different

- '...' rather than "...", no \n-like "escape" chars
- escape mechanisms: 'O''Brien' or E'O\'Brien' (non-standard)
- dollar quoting: $$O'Brien$$ or tagO'Brientag

SQL Operators

Comparison operators are defined on all types:

- `<` `>` `<=` `>=` `=` `<>`

In PostgreSQL, `!=` is a synonym for `<>` (but there's no ==)

Boolean operators `AND`, `OR`, `NOT` are also available

- Note `AND`, `OR` are not "short-circuit" in the same way as C's &&,||

Most data types also have type-specific operations available

- String comparison (e.g. str1 < str2) uses dictionary order

https://www.postgresql.org/docs/16/functions.html

SQL Operators (string-matching)

SQL provides pattern matching for strings via `LIKE` and `NOT LIKE`

- `%` matches anything (cf. regexp `.*`)
- `_` matches any single char (cf. regexp `.`)

Examples:

- `name LIKE 'Ja%'` name begins with 'Ja'
- `name LIKE '_i%'` name has 'i' as 2nd letter
- `name LIKE '%o%o%'` name contains two 'o's
- `name LIKE '%ith'` name ends with 'ith'
- `name LIKE 'John'` name equals 'John'

PostgreSQL also supports case-insensitive matching: `ILIKE`

SQL Operators (string-matching)

PostgreSQL provides regexp-based pattern matching via `~` and `!~`

Examples (using POSIX regular expressions):

- `name ~ '^Ja'` name begins with 'Ja'
- `name ~ '^.i'` name has 'i' as 2nd letter
- `name ~ '.*o.*o.*'` name contains two 'o's
- `name ~ 'ith$'` name ends with 'ith'
- `name ~ 'John'` name contains 'John'

Also provides case-insensitive matching via `~*` and `!~*`

SQL Operators (string)

Other operators/functions for string manipulation:

str1 || str2 ... return concatenation of str1 and str2
lower(str) ... return lower-case version of str
substring(str,start,count) ... extract substring from str
Etc. ...

Note that above operations are null-preserving (strict):
if any operand is NULL, result is NULL

SQL Operators (arith)

Arithmetic operations:
`+` `-` `*` `/` `abs` `ceil` `floor` `power` `sqrt` `sin` etc.

Aggregations "summarize" a column of numbers in a relation:

- `count(attr)` ... number of rows in attr column
- `sum(attr)` ... sum of values for attr
- `avg(attr)` ... mean of values for attr
- `min/max(attr)` ... min/max of values for attr

Note: **count** applies to columns of non-numbers as well.

The NULL value

Expressions containing NULL generally yield NULL.

However, boolean expressions use three-valued logic:

a	b	a AND b	a OR b
T	T	T	T
T	F	F	T
T	NULL	NULL	T
F	T	F	T
F	F	F	F
F	NULL	F	NULL
NULL	NULL	NULL	NULL

The NULL value (cont)

Important consequence of NULL behaviour ...

These expressions do not work as (might be) expected:
x = NULL x <> NULL

Both return NULL regardless of the value of x

Can only test for NULL using:

⚠️ x **IS** NULL x **IS NOT** NULL

Conditional Expressions

Other ways that SQL provides for dealing with NULL:

coalesce(val_1,val_2,...val_n)

- returns first non-null value val_i (or NULL if all values are NULL).

E.g. select coalesce(mark, ...) from Marks ...

nullif(val_1,val_2)

- takes two values and returns the first value, except it returns NULL if val_1 is equal to val_2

E.g. nullif(mark,'??')

Conditional Expressions

SQL also provides a generalised conditional expression:

CASE
  WHEN $test_1$ **THEN** $result_1$
  WHEN $test_2$ **THEN** $result_2$
  ...
  ELSE $result_n$
END

E.g. case when mark>=85 then 'HD' ... else '??' end

Tests that yield NULL are treated as FALSE

If no ELSE, and all tests fail, CASE yields NULL

Outline

- Announcements
- SQL Expressions
- **SQL Meta-Data Definition Language**
- Managing RDBMS
- Mapping ER->Rel->SQL

SQL Intro – Recap

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**

Relational Data Definition

In order to give a relational data model, we need to:

- describe **tables**
- describe **attributes** that comprise tables
- describe any **constraints** on the data

A **relation schema** defines an individual table

- table name, attribute names, attribute domains, keys, etc.

A **database schema** is a collection of relation schemas that

- defines the structure the whole database
- additional constraints on the whole database

Defining a Relation Schema

Tables (relations) are described using:

```
CREATE TABLE TableName (
   attribute1   domain1   constraints1,
   attribute2   domain2   constraints2,
   ...
   table-level constraints, ...
)
```

This SQL statement ..

- defines the table schema (adds it to database meta-data)
- creates an empty instance of the table (zero tuples)

Tables are removed via `DROP TABLE TableName;`

Defining a Relation Schema

Example: defining the Students table ...

```
CREATE TABLE Students (
   zid     serial,
   family  varchar(40),
   given   varchar(40) NOT NULL,
   d_o_b   date NOT NULL,
   gender  char(1) CHECK (gender in ('M','F','N')),
   degree  integer,
   PRIMARY KEY (zid),
   FOREIGN KEY (degree) REFERENCES Degrees(did)
);
```

```
CREATE TABLE Students (
   zid     serial PRIMARY KEY, -- only if it's a single attribute key
   family  varchar(40),
   given   varchar(40) NOT NULL,
   d_o_b   date NOT NULL,
   gender  char(1) CHECK (gender in ('M','F','N')),
   degree  integer REFERENCES Degrees(did)
);
```

⚠ A primary key attribute is implicitly defined to be **UNIQUE** and **NOT NULL**

Data Integrity

Defining tables as above affects behaviour of DBMS when changing data

Constraints and types ensure that integrity of data is preserved

- no duplicate keys
- no "dangling references"
- all attributes have valid values
- etc. etc. etc.

Preserving data integrity is a **critical** function of a DBMS.

Defining Keys

Primary keys:

- if PK is one attribute, can define as attribute constraint
- if PK is multiple attributes, must define in table constraints
- PK implies **NOT NULL, UNIQUE** for all attributes in key

Foreign keys:

- if FK is one attribute, can define as attribute constraint
- can omit **FOREIGN KEY** keywords in attribute constraint
- if FK has multiple attributes, must define as a single table constraint
- should always specify corresponding PK attribute in FK constraint

Defining Keys (cont)

Defining primary keys assures **entity integrity**

- must give values for all attributes in the primary key

For example this insertion would fail ...

```
INSERT INTO Enrolments(student,course,mark,grade)
    VALUES (5123456, NULL, NULL, NULL);
```

because no course was specified; but mark and grade can be NULL

Defining primary keys assures **uniqueness**

- cannot insert a tuple which contains an existing PK value

Defining Keys (cont)

Defining foreign keys assures **referential integrity**.

On insertion, cannot add a tuple where FK value does not exist as a PK

For example, this insert would fail ...

```
INSERT INTO Accounts(acctNo, owner, branch, balance)
        VALUES ('A-123', 765432, 'Kensington', 5000);
```

if there is no customer with id 765432 or no branch Kensington

Attribute Value Constraints

NOT NULL and UNIQUE are special constraints on attributes.

SQL has a general mechanism for specifying attribute constraints

attrName type **CHECK** (Condition)

Condition is a boolean expression and can involve other attributes, relations and sub-queries.

```
CREATE TABLE Example
(
  gender char(1)  CHECK (gender IN ('M','F')),
  Xvalue integer  NOT NULL,
  Yvalue integer  CHECK (Yvalue > Xvalue),
  Zvalue float    CHECK (Zvalue >
                    (SELECT MAX(price)
                     FROM   Sells)
                )
);
```

(but many RDBMSs (e.g. Oracle and **PostgreSQL**) don't allow subqueries in CHECK)

reason: https://stackoverflow.com/questions/10179121/sql-sub-queries-in-check-constraint

Named Constraints

A constraint in an SQL table definition can (optionally) be named via

```
CONSTRAINT  constraintName  constraint
```

Example:

```
CREATE TABLE Example
(
  gender char(1) CONSTRAINT GenderCheck
            CHECK (gender IN ('M','F','N')),
  Xvalue integer NOT NULL,
  Yvalue integer CONSTRAINT XYOrder
            CHECK (Yvalue > Xvalue)
);
```

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

reason:
https://stackoverflow.com/questions/1397440/what-is-the-purpose-of-constraint-naming#:~:text=By%20naming%20the%20constraints%20you%20can%20differentiate%20violations,react%20differently%20depending%20on%20which%20constraint%20was%20violated.

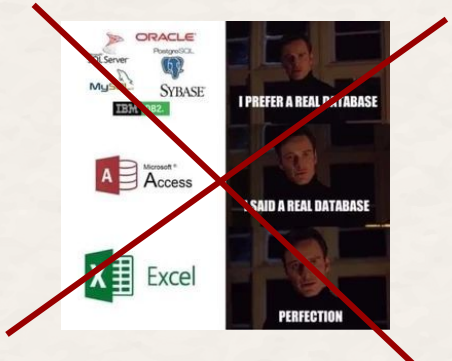
Outline

- Announcements
- SQL Expressions
- SQL Meta-Data Definition Language
- **Managing RDBMS**
- Mapping ER->Rel->SQL

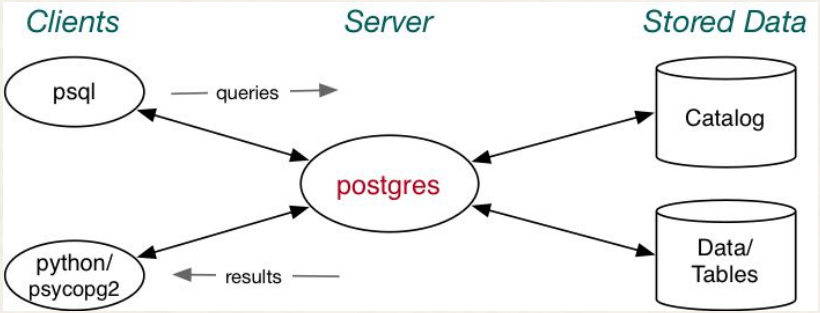
What is an RDBMS?

- A relational database management system (RDBMS) is
  - software designed to support large-scale data-intensive applications
  - allowing high-level description of data (tables, constraints)
  - with high-level access to the data (relational model, SQL)
  - providing efficient storage and retrieval (disk/memory management)
  - supporting multiple simultaneous users (privilege, protection)
  - doing multiple simultaneous operations (transactions, concurrency)
  - maintaining reliable access to the stored data (backup, recovery)

- Note: databases provide persistent storage of information (even for redis !)



Postgres Structure



Managing Databases

Shell commands to create/remove databases:

- createdb dbname ... create a new totally empty database
- dropdb dbname ... remove all data associated with a DB

Shell commands to dump/restore database contents:

pg_dump dbname > dumpfile
psql dbname -f dumpfile
(Database dbname is typically created just before restore)

Managing Tables

SQL statements:

- **CREATE** TABLE table (Attributes+Constraints)
- **ALTER** TABLE table TableSchemaChanges
- **DROP** TABLE table(s) [CASCADE]
- **TRUNCATE** TABLE table(s) [CASCADE]

(All conform to SQL standard, but all also have extensions)

DROP..CASCADE also drops objects which depend on the table

- objects could be tuples or views, but not whole tables

TRUNCATE..CASCADE truncates tables which refer to the table

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

Managing Tuples

SQL statements:

- **INSERT** INTO table (Attrs) VALUES Tuple(s)
- **DELETE** FROM table WHERE condition
- **UPDATE** table SET AttrValueChanges WHERE condition

Attrs = ($attr_1$,, ... $attr_n$) Tuple = (val_1,, ... val_n)

AttrValueChanges is a comma-separated list of:

- attrname = expression
- Each list element assigns a new value to a given attribute.

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

Outline

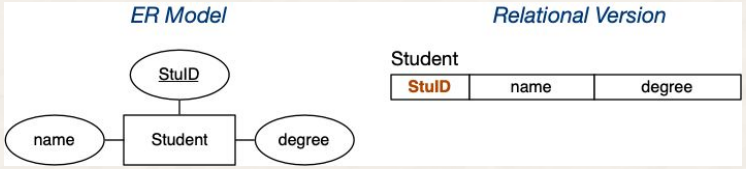
- Announcements
- SQL Expressions
- SQL Meta-Data Definition Language
- Managing RDBMS
- **Mapping ER->Rel->SQL**

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



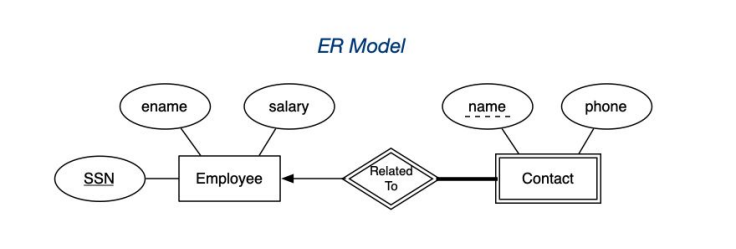
SQL Version

```
create table Students (
    stuID    integer primary key,
    name     text not null,
    degree   char(4)
);
```

Note: the key is preserved in the mapping.

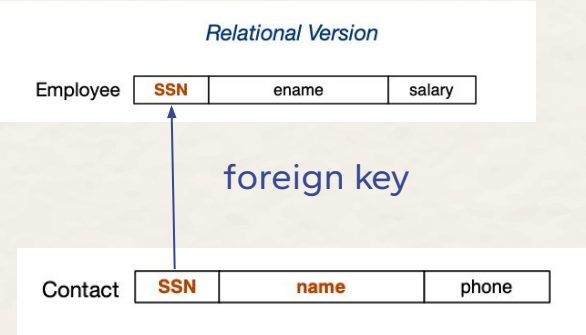
Mapping *Weak Entities*

- **foreign keys**: named after either or both of
  - **table being referenced** e.g. staff or staff_id, ...
  - **relationship being modelled** e.g. teaches, ...

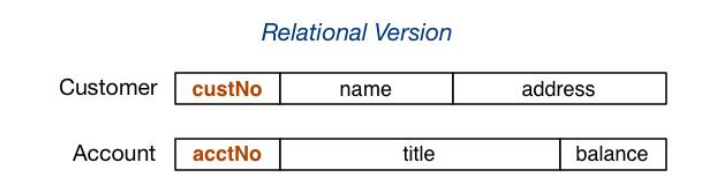


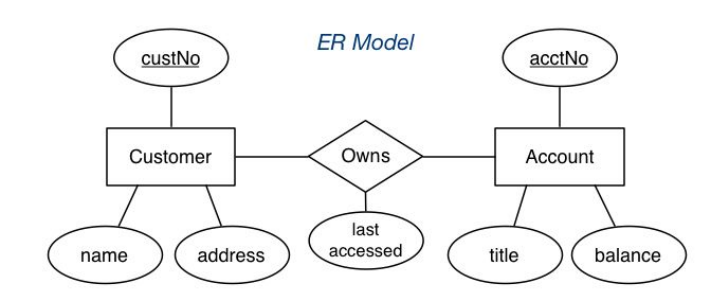

```
create table Employees (
    SSN     text primary key,
    ename   text,
    salary    currency
);
```

```
create table Contacts (
    relatedTo  text not null,  -- total participation
    name       text,           -- not null implied by PK
    phone      text not null,
    primary key (relatedTo, name),
    foreign key (relatedTo) references Employees (ssn)
);
```



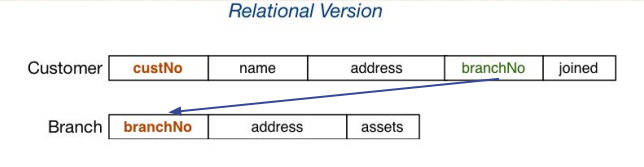
Mapping *N:M Relationships*

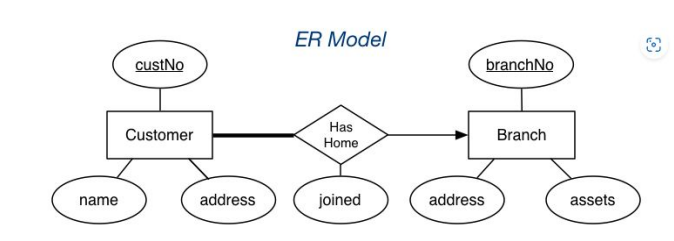





```
create table Customers (
    custNo  serial primary key,
    name    text not null,
    address text  -- don't need to know customer's address
);
create table Accounts (
    acctNo   char(5) check (acctNo ~ '[A-Z]-[0-9]{3}'),
    title    text not null,    -- acctNos are like 'A-123'
    balance  float default 0.0,
    primary key (acctNo)
);
create table Owns (
    customer_id integer references Customers(custNo),
    account_id  char(5) references Accounts(acctNo),
    last accessed timestamp,
    primary key (customer_id, account_id)
);
```

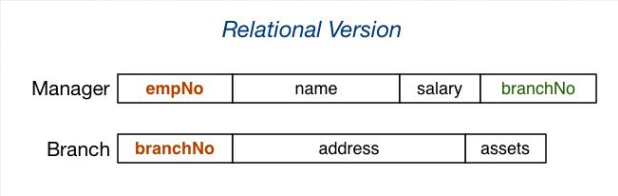
Mapping *1:N Relationships*

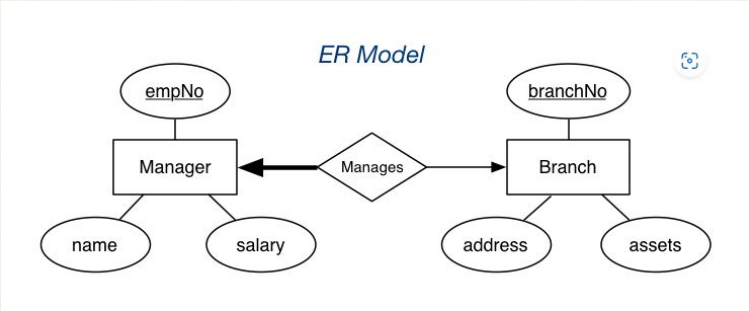





```
create table Branches (
    branchNo serial primary key,
    address  text not null,
    assets   currency
);
create table Customers (
    custNo  serial primary key,
    name    text not null,
    address text,
    hasHome integer not null, -- total participation
    joined  date not null,
    foreign key (hasHome) references Branches(branchNo)
);
```

Mapping *1:1 Relationships*





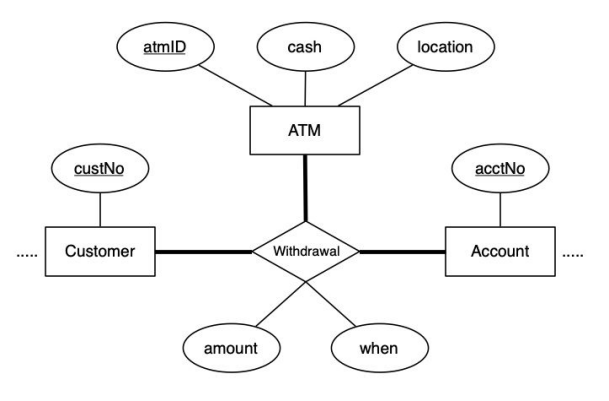

```
create table Branches (
    branchNo serial primary key,
    address  text not null,
    assets   currency          -- a new branch
);                             --    may have no accounts
create table Managers (
    empNo    serial primary key,
    name     text not null,
    salary   currency not null, -- when first employed,
                                --    must have a salary
    manages  integer not null,  -- total participation
    foreign key (manages) references Branches(branchNo)
);
```

What if there's a total participation for "Branch"?

If both entities have total participation, cannot express this in SQL except by putting a (redundant) **not null** foreign key in one table.

i.e. put a foreign key of **empNo** in the branch table.

Mapping *n-way Relationships*



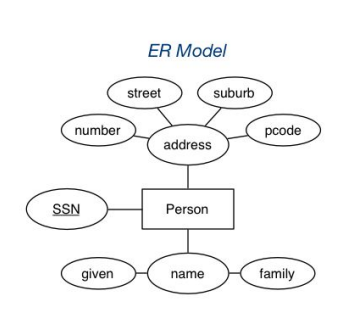

```
create table Customers (
    custNo    serial primary key, ...
);
create table Accounts (
    acctNo    char(5) ... primary key, ...
);
create table ATMs (
    atmID     serial primary key,
    cash      currency check (cash >= 0),
    location text not null
);
create table Withdrawal (
    customer_id  integer references Customers(custNo),
    account_id   char(5) references Accounts(acctNo),
    atm_id       integer references ATMs(atmID),
    amount       currency not null,
    when         timestamp default now(),
    primary key  (customer_id,account_id,atm_id)
);
```

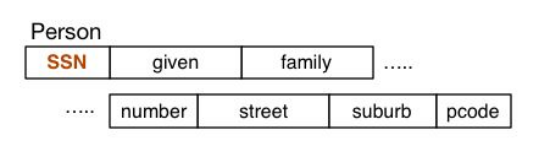
What if a customer can only have one account?
(i.e., there's an arrow pointing to account)
What's the primary key?

customer_id, atm_id

Mapping *Composite Attributes*

Composite attributes are mapped by concatenation or flattening.

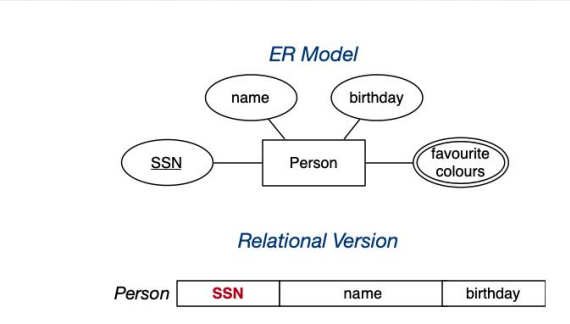





```
create table People (
    ssn     integer primary key,
    given   text not null,
    family  text,
    number  integer not null,
    street  text not null,
    suburb  text not null,
    pcode   char(4) not null check (pcode ~ '[0-9]{4}')
);
```

Mapping *Multi-valued Attributes* (MVAs)

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




```
create table People (
    ssn      integer primary key,
    name     text not null,
    birthday date
);
create table FavColour (
    person_id integer references People(ssn),
    colour    text,
    primary key (person_id,colour)
);
```

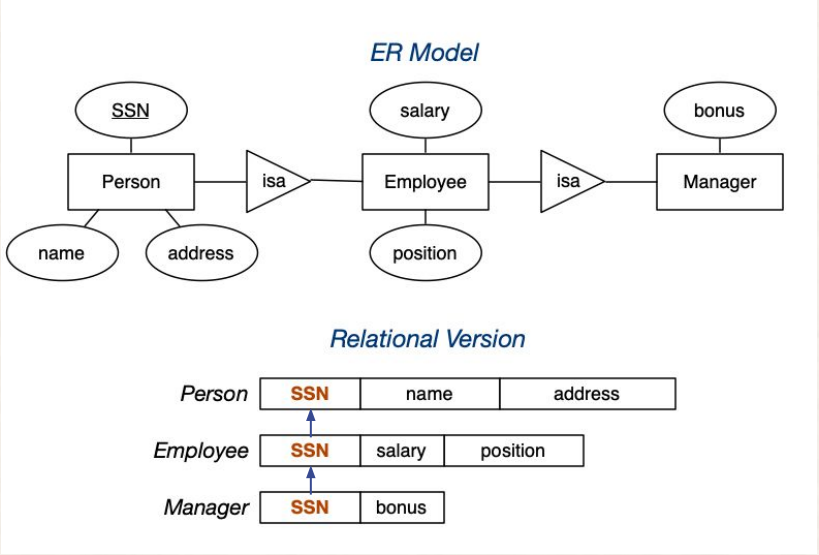
Mapping *Subclasses*

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)

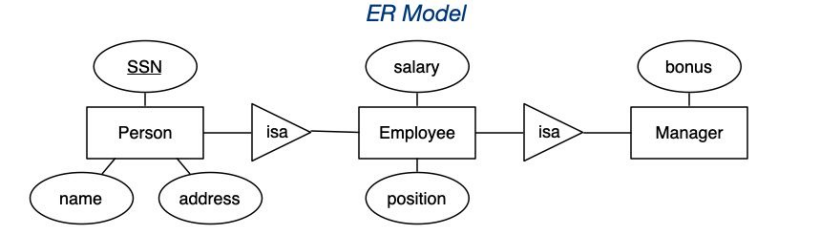
Which mapping is best depends on how data is to be used.

Mapping *Subclasses – ER style*




```
create table People (
    ssn     integer primary key,
    name    text not null,
    address text
);
create table Employees (
    person_id integer primary key,
    salary    currency not null,
    position  text not null,
    foreign key (person_id) references People(ssn)
);
create table Managers (
    employee_id integer primary key,
    bonus       currency,
    foreign key (employee_id)
                references Employees(person_id)
);
```

Mapping *Subclasses – OO style*

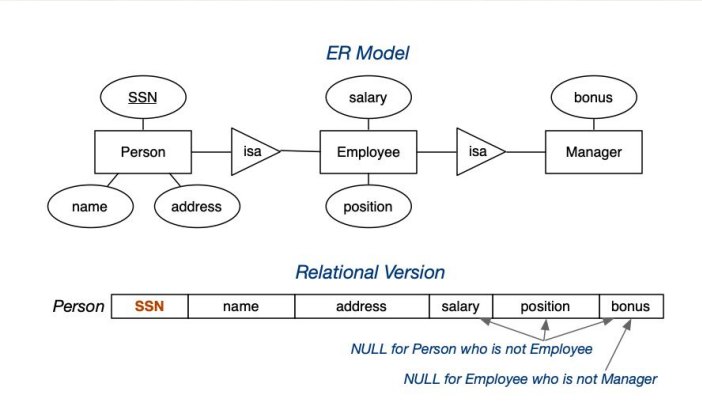



```
create table People (
    ssn     integer primary key,
    name    text not null,
    address text
);
create table Employees (
    ssn       integer primary key,
    name      text not null,
    address   text
    salary    currency not null,
    position  text not null

);
create table Managers (
    ssn       integer primary key,
    name      text not null,
    address   text
    salary    currency not null,
    position  text not null.
    bonus     currency

);
```

Mapping *Subclasses – single table style*




```
create table People (
    ssn       integer primary key,
    ptype     char(1)  not null
                       check (ptype in ('P','E','M')),
    name      text not null,
    address   text
    salary    currency,
    position  text,
    bonus     currency,
    constraint subclasses check
               ((ptype = 'P' and salary is null
                and position is null and bonus is null)
               or
                (ptype = 'E' and salary is not null
                 and position is not null and bonus is null)
               or
                (ptype = 'M' and salary is not null
                 and position is not null and bonus is not null))
);
```

Thank you!