Principles of Database Systems (CS307)

Lecture 7: Advanced SQL and Database Design using E-R Model

Zhong-Qiu Wang

Department of Computer Science and Engineering
Southern University of Science and Technology

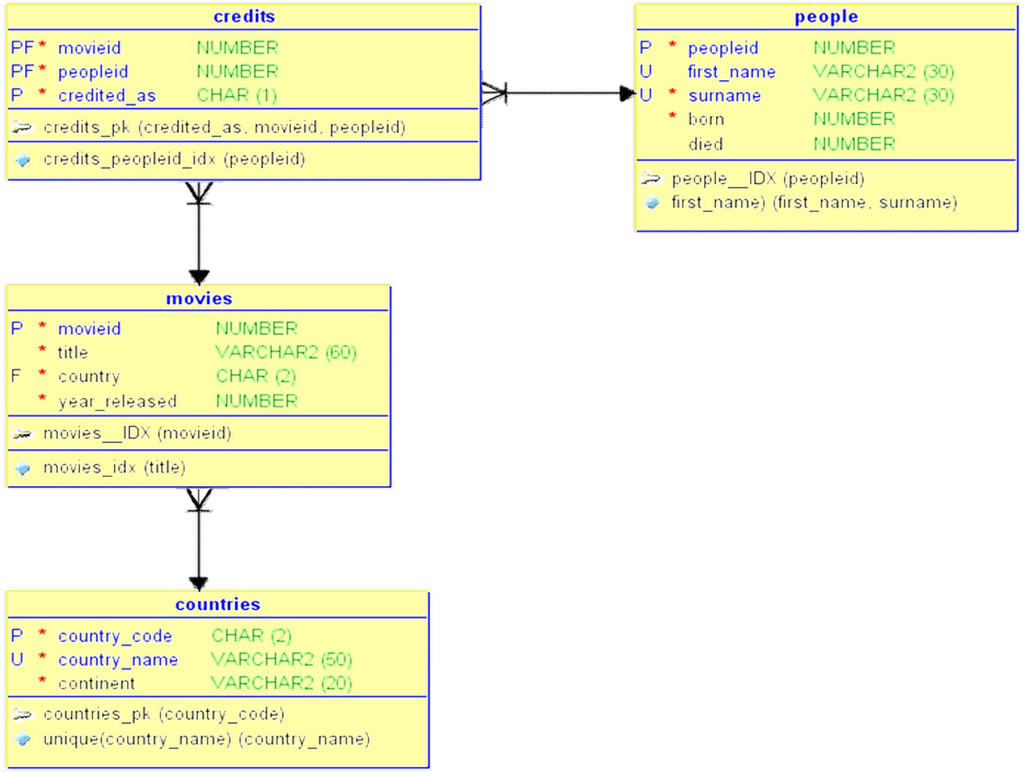
- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

Announcements

- Project I is out, due date: 23:59 on November 10th 2024, Beijing Time
  - Only accept project report in English
    - The class is designed as an English class
    - Can avoid potential unfairness when grading Chinese and English reports

Entity and Relationship

- Starring -> Actor table
- Country -> Country and Region table
  - You can also link the movies with corresponding actors, countries/regions, etc.
- Entity Relationship Diagram (E/R Diagram, ER Diagram, ERD)
  - A way of representing entity tables and their relationships (relationship tables)
  - Connect tables via foreign keys and relationship tables



Create Tables

- An SQL relation is defined using the create table command:

```
create table r (
    A1 D1, A2 D2, ..., An Dn,
    (integrity-constraint1),
    ...,
    (integrity-constraintk)
)
```

  - Relation schema: r is the name of the relation
  - Attribute: each Ai is an attribute name in the schema of relation r
  - Data type: Di is the data type of values in the domain of attribute Ai
  - Constraints: not null, unique, primary key, check function, referential integrity & foreign key

Select

- `select * from` [tablename]
  - The `select` clause lists the attributes desired in the result of a query
  - To display the full content of a table, you can use select *
    - * : all columns

```
select A1, A2, ..., An
 from r1, r2, ..., rm
       where P
```

Some Functions

- When to use functions
  - An example of showing a result that isn't stored as such is computing an age
    - You should never store an age; it changes all the time!
    - If you want to display the age of people who are alive, you must compute their age by subtracting the year when they were born from the current year.

  - In the table people:
    - Alive – died is `null`
    - Age: `<this year> - born`

```
select peopleid, surname,
       date_part('year', now()) - born as age
from people
where died is null;
```

7	7	Caroline	Aaron	1952	<null>	F
8	8	Quinton	Aaron	1984	<null>	M
9	9	Dodo	Abashidze	1924	1990	M

Aggregate Functions

`count(*)/count(col), min(col), max(col), stddev(col), avg(col)`

- These aggregate functions are supported in almost every DBMS
  - Most DBMS implement other more advanced functions
  - Some functions work with any datatype, others only work with numerical columns

  - It is strongly recommended to refer to user manual for details
    - For example, SQLite doesn't have `stddev()`

Retrieving Data from Multiple Tables

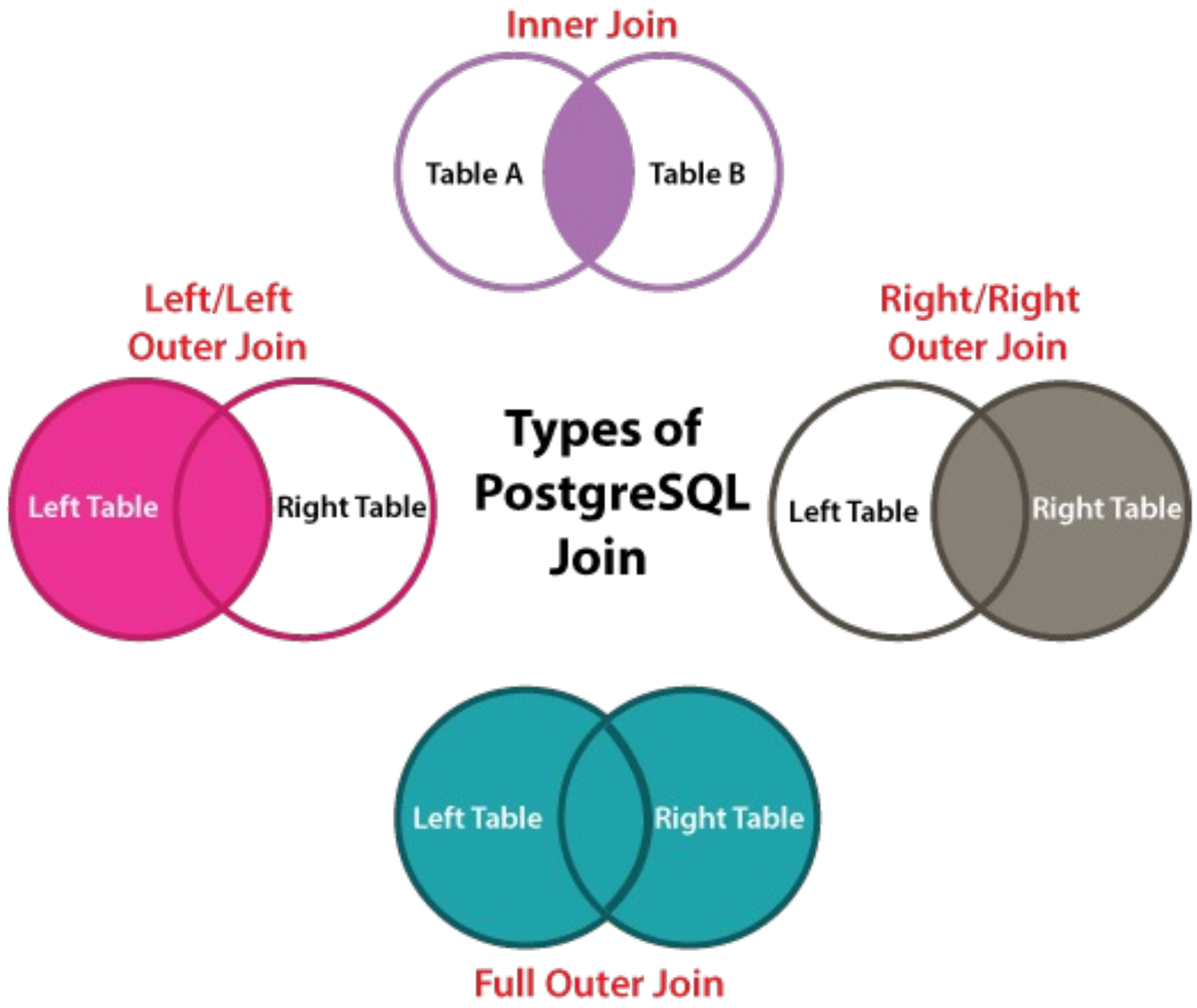
- This is done with this type of query. We retrieve, and display as a single set, pieces of data coming from two different tables.

```
select title,
    country_name,
    year_released
    from movies
    join countries
    on country_code = country;
```

title	country_name	year_released
12 stulyev	Russia	1971
Al-mummia	Egypt	1969
Ali Zaoua, prince de la rue	Morocco	2000
Apariencias	Argentina	2000
Ardh Satya	India	1983
Armaan	India	2003
Armaan	Pakistan	1966
Babettes gæstebud	Denmark	1987
Banshun	Japan	1949
Bidaya wa Nihaya	Egypt	1960
Variety	United States	2008
Bon Cop, Bad Cop	Canada	2006
Brilliantovaja ruka	Russia	1969
C'est arrivé près de chez vous	Belgium	1992
Carlota Joaquina - Princesa do Brasil	Brazil	1995
Cicak-man	Malaysia	2006
Da Nao Tian Gong	China	1965
Das indische Grabmal	Germany	1959
Das Leben der Anderen	Germany	2006
Den store gavtyv	Denmark	1956

Inner and Outer Joins

- So far, we only join the rows with matching values on the corresponding columns
- There are more things we can do with join



Subquery

- We have used subqueries after the keyword `from` before
  - … in order to build queries upon a query result

- And, we can add subqueries after `select` and `where` as well

```
select A1, A2, ..., An
 from r1, r2, ..., rm
      where P
```

Advanced Ordering

- Multiple columns
  - The result set will be order by `col1` first
  - Rows with the same value on `col1` will be ordered by `col2`

```
order by col1, col2, ...
```

- Ascending or descending order
  - Add `desc` or `asc` after the column
    - `asc` is the default option and thus always omitted

```
-- Order col1 descendingly
order by col1 desc

-- Order based on col1 first, then col2.
-- col1 will be in the descending order, col2 ascending.
order by col1 desc, col2 asc, ...
```

Window Function

- Syntax:

```
<function> over (partition by <col_p> order by <col_o1, col_o2, …>)
```

  - `<function>`
    - Ranking window functions
    - Aggregation functions
  - `partition by`
    - Specify the column for grouping
  - `order by`
    - Specify the column(s) for ordering in each group

Ranking Window Function

- Example
  - How can we rank the movies in each country separately based on the released year?
    - "order by" for subgroups

```
select country,
       title,
       year_released,
       rank() over (
           partition by country order by year_released
       ) oldest_movie_per_country
from movies;
```

Partitioned by country
- i.e., a country in a group

country	title	year_released	oldest_movie_per_country
ar	some title	1948	1
ar	some title	1959	2
ar	some title	1980	3
cn	some title	1987	1
cn	some title	2002	2
uk	some title	1985	1
uk	some title	1992	2
uk	some title	2010	3

`rank()`
- A function to say that "I want to order the rows in each partition"
- No parameters in the parentheses

`order by year_released`
- In each group (partition), the rows will be ordered by the column "`year_released`"

An order value is computed for each row in a partition.
- Only inside the partition, not across the entire result set

Update

- Make changes to the existing rows in a table
- `update` is the command that changes column values
  - You can even set a non-mandatory column to NULL
  - The change is applied to all rows selected by the `where`

```
update table_name
set column_name = new_value,
    other_col = other_new_val,
    ...
where ...
```

Delete

- As the name shows, `delete` removes rows from tables

  - If you omit the WHERE clause, then (as with UPDATE) the statement affects all rows and you **end up with an empty table!**
  - Well,
    - Many database products provide a roll-back mechanism when deleting rows
    - Transactions can also protect you (to some extent)

Constraints

- One important point with constraints (esp., foreign keys) is that they guarantee that data remains consistent
  - They not only work with `insert`, but with `update` and `delete`
  - Example: Try to delete some rows in the `country` table
    - Foreign-key constraints are especially useful in controlling `delete` operations

```
delete from countries where country_code = 'us';
```

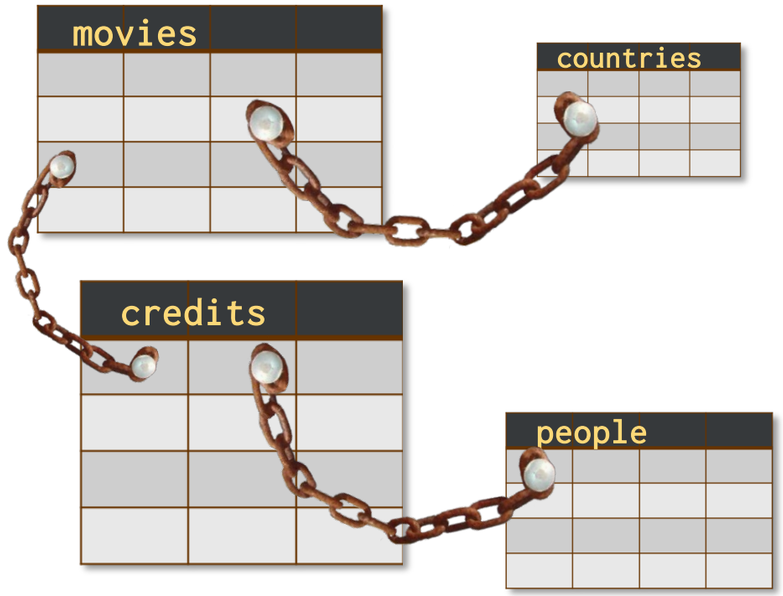
```
[23503] ERROR: update or delete on table "countries" violates foreign key constraint "movies_country_fkey" on table "movies"
Detail: Key (country_code)=(us) is still referenced from table "movies".
```

movieid	title	country	year_released
1	Casab	us	1942
2	Goodfellas	us	1990
3	Bronenosets Potyomkin	ru	1925

country_code	country_name	continent
ru	Russia	Europe
us	United States	America

Constraints

- This is why constraints are so important:
  - They ensure that whatever happens, you'll always be able to make sense of ALL pieces of data in your database
  - For any changes made to the tables, all declared constraints need to be satisfied



Self-defined Function

- Sometimes the built-in functions cannot fulfill our requirements
  - And the power of declarative language (SQL) is not strong enough

- Most DBMS implement a built-in, SQL-based programming language
  - A procedural extension to SQL

Procedural vs. Declarative

- Two different programming paradigms
  - Imperative programming （命令式编程）
    - Describe the algorithms step-by-step (i.e., how to do)
    - Procedural （过程式）: C (and many other legacy languages)
    - Object-oriented: Java
  - Declarative programming（声明式编程）
    - Describe the result without specifying the detailed steps (i.e., what to do)
    - (Pure) declarative: SQL, Regular Expressions, Markup (HTML, XML), CSS
    - Functional: Scheme, Haskell, Scala, Erlang
    - Logic programming: Prolog

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

Language Type

PostgreSQL supports 4 procedural languages: PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python

- Tcl, Perl, and Python are famous scripting languages in case you don't know

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

```
create function full_name(p_
returns varchar
as $$
begin
    return case
        when p_fname is null
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

```
create function append_test(p_code varchar)
returns varchar
as $$
    if p_code == 'cn':
        return 'China'
    else:
        return 'not China'
$$ language plpython3u;
```

Yes, we can even use Python to write functions

Language Type

PostgreSQL supports 4 procedural languages:
PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python

- Tcl, Perl, and Python are famous scripting languages in case you don't know

Functions and Procedures in (Postgre)SQL

- It follows the general definition of functions and procedures
  - Function: return a value
  - Procedure: return NO value
- However,
  - For some historical reasons, PostgreSQL actually has no implementation specifically for procedures
    - It shares the same mechanism with functions
    - Treats procedures as void functions
  - * But for some other database systems, there are separate implementations for functions and procedures

When to Use Procedures

- For business logics
  - One requirement may need a series of SQL querys and statements
  - Transactions may be used
  - Example: Insert a new movie into the databases
    - `movies` table
      - Basic information for the movie
    - `countries` table
      - Transformation between country names and codes
    - `people` table
      - new actors / directors
    - `credits` table
      - new credit information
    - Problem: Update all the tables? Input validation? Code reuse? Security?

When to Use Procedures

- To add a movie:
  - We may have a series queries to execute when inserting only one movie
  - How about one call for all the processes?
    - Benefit 1: Network overhead
      - When running multiple queries, you are going to waste time chatting over the network with the remote server
    - Benefit 2: Security
      - Prevent users from modifying data otherwise than by calling carefully written and well tested procedures
      - Ensure that users can only modify data via carefully written and well tested procedures

The Procedure

```
create function movie_registration
           (p_title        varchar,
            p_country_name varchar,
            p_year         int,
            p_director_fn  varchar,
            p_director_sn  varchar,
            p_actor1_fn    varchar,
            p_actor1_sn    varchar,
            p_actor2_fn    varchar,
            p_actor2_sn    varchar)
returns void
as $$
declare
  n_rowcount int;
  n_movieid int;
  n_people int;
begin
    insert into movies(title, country, year_released)
        select p_title, country_code, p_year
        from countries
        where country_name = p_country_name;
    get diagnostics n_rowcount = row_count;

    if n_rowcount = 0
    then
        raise exception 'country not found in table COUNTRIES';
    end if;

    n_movieid := lastval();
    select count(surname)
    into n_people
    from (select p_director_sn as surname
        union all
        select p_actor1_sn as surname
        union all
        select p_actor2_sn as surname) specified_people
    where surname is not null;

    insert into credits(movieid, peopleid, credited_as)
        select n_movieid, people.peopleid, provided.credited_as
        from (select coalesce(p_director_fn, '*') as first_name,
            p_director_sn as surname,
            'D' as credited_as
            union all
            select coalesce(p_actor1_fn, '*') as first_name,
                p_actor1_sn as surname,
                'A' as credited_as
            union all
            select coalesce(p_actor2_fn, '*') as first_name,
                p_actor2_sn as surname,
                'A' as credited_as) provided
            inner join people
        on people.surname = provided.surname
            and coalesce(people.first_name, '*') = provided.first_name
        where provided.surname is not null;

    get diagnostics n_rowcount = row_count;
    if n_rowcount != n_people
    then
        raise exception 'Some people couldn''t be found';
    end if;
end;
$$ language plpgsql;
```

Trigger（触发器）

Trigger （触发器）- Actions When Changing Tables

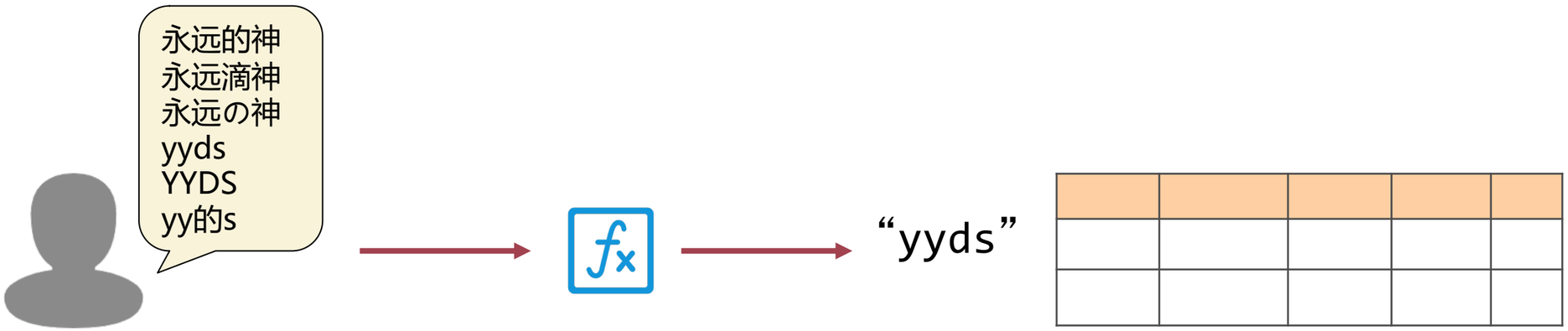
> A trigger is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed.
>
> -- Chapter 39, PostgreSQL Documentation
>
> A trigger is a statement that the system executes automatically as a side effect of a modification to the database.
>
> -- Chapter 5.3, Database System Concepts, 7th

- We can attach “actions” to a table
  - They will be executed automatically whenever the data in the table changes
- Purpose of using triggers
  - Validating data
  - Checking complex rules
  - Managing data redundancy

Purpose of Using Triggers

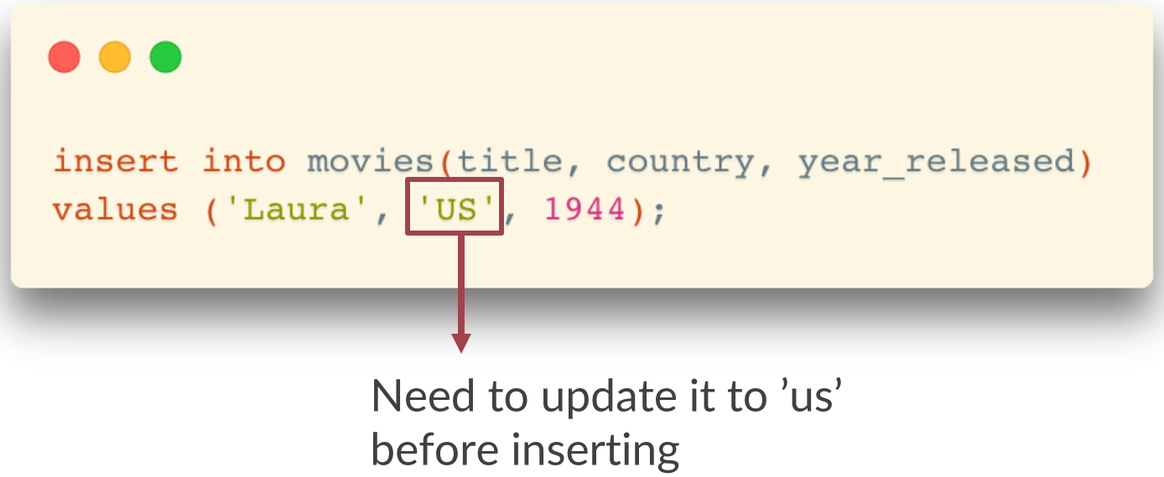
- Validating data
  - Some data are badly formatted in programs before sending to the database
  - We need to validate such data before inserting them into the database

  - “On-the-fly” modification
    - Change the input directly when the input arrives



Purpose of Using Triggers

- Validating data
  - Example: insert a row in the movies table
    - In the JDBC program, an `insert` request is written like the following:



  - Although,
    - Such validation or transformation should be better handled by the application programs

Purpose of Using Triggers

- Checking complex rules
  - Sometimes, the business rules are so complicated
    - They CANNOT be checked via declarative integrity constraints

Purpose of Using Triggers

- Managing data redundancy
  - Some data redundancy issues cannot be avoided by simply adding constraints

  - For example: inserted the same movie but in different languages

```sql
-- US
insert into movies(title, country, year_released)
values ('The Matrix', 'us', 1999);

-- China (Mainland)
insert into movies(title, country, year_released)
values ('黑客帝国', 'us', 1999);

-- Hongkong
insert into movies(title, country, year_released
values ('22世紀殺人網絡', 'us', 1999);
```

It satisfies the unique constraint on (`title, country, year_released`)

- … but they represent the same movie

Trigger Activation

- Two key points:
  - When to fire a trigger?
  - What (command) fires a trigger?

Trigger Activation

- When to fire a trigger?
  - In general: “During the change of data”
    - … but we need a detailed discussion
  - Note: “During the change” means `select` queries won’t fire a trigger.

Trigger Activation: When

- Example: Insert a set of rows with "`insert into select`"
  - One statement, multiple rows

```
insert into movies(title, country, year_released)
select titre, 'fr', annee
from films_francais;
```

  - Option 1: Fire a trigger only once for the statement
    - Before the first row is inserted, or after the last row is inserted
  - Option 2: Fire a trigger for each row
    - Before or after the row is inserted

Trigger Activation: When

- Different options between DBMS products

- Before statement
  - Before each row
  - After each row
- After statement

- ~~Before statement~~
  - Before each row
  - After each row
- ~~After statement~~

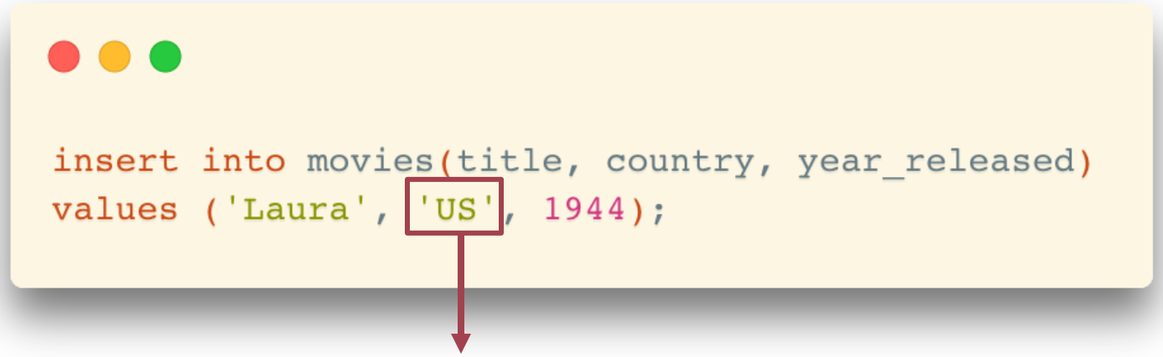
- ~~Before statement~~
  - ~~Before each row~~
  - ~~After each row~~
- After statement

Trigger Activation: What

- What (command) fires a trigger?
  - `insert`
  - `update`
  - `delete`

Before and After Triggers

- Differences between before and after triggers
  - “Before” and “after” the operation is done (`insert`, `update`, `delete`)
  - If we want to update the incoming values in an `insert` statement (e.g., “US” → “us”), “before trigger” should be used since the incoming values have not been written to the table yet



Before and After Triggers

- Typical usage scenarios for trigger settings
  - Modify input on the fly
    - before insert / update
    - for each row
  - Check complex rules
    - before insert / update / delete
    - for each row
  - Manage data redundancy
    - after insert / update / delete
    - for each row

Example of Triggers

- An Example
  - For the `people_1` table, when updating a person, count the number of movies and save the result in the `num_movies` column

```sql
-- auto-generated definition
create table people_1
(
    peopleid   integer,
    first_name varchar(30),
    surname    varchar(30),
    born       integer,
    died       integer,
    gender     bpchar,
    num_movies integer
);
```

	peopleid	first_name	surname	born	died	gender	num_movies
1	13	Hiam	Abbass	1960	<null>	F	<null>
2	559	Aleksandr	Askoldov	1932	<null>	M	<null>
3	572	John	Astin	1930	<null>	M	<null>
4	585	Essence	Atkins	1972	<null>	F	<null>
5	598	Antonella	Attili	1963	<null>	F	<null>
6	611	Stéphane	Audran	1932	<null>	F	<null>
7	624	William	Austin	1884	1975	M	<null>
8	637	Tex	Avery	1908	1980	M	<null>
9	650	Dan	Aykroyd	1952	<null>	M	<null>
10	520	Zackary	Arthur	2006	<null>	M	<null>
11	533	Oscar	Asche	1871	1936	M	<null>
12	546	Elizabeth	Ashley	1939	<null>	F	<null>

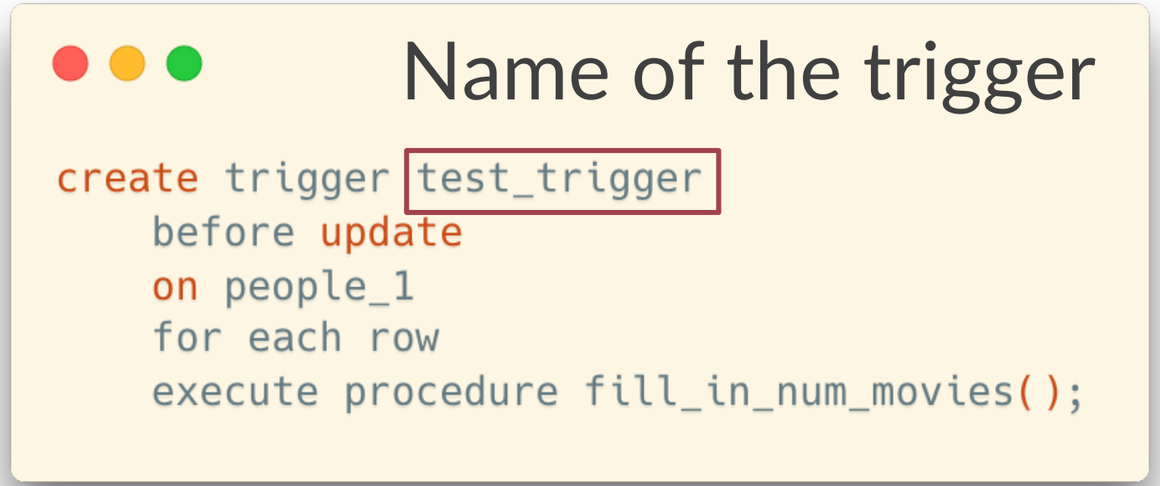
Example of Triggers

- Create a trigger

```
create trigger test_trigger
    before update
    on people_1
    for each row
    execute procedure fill_in_num_movies();
```

Example of Triggers

- Create a trigger



Example of Triggers

- Create a trigger

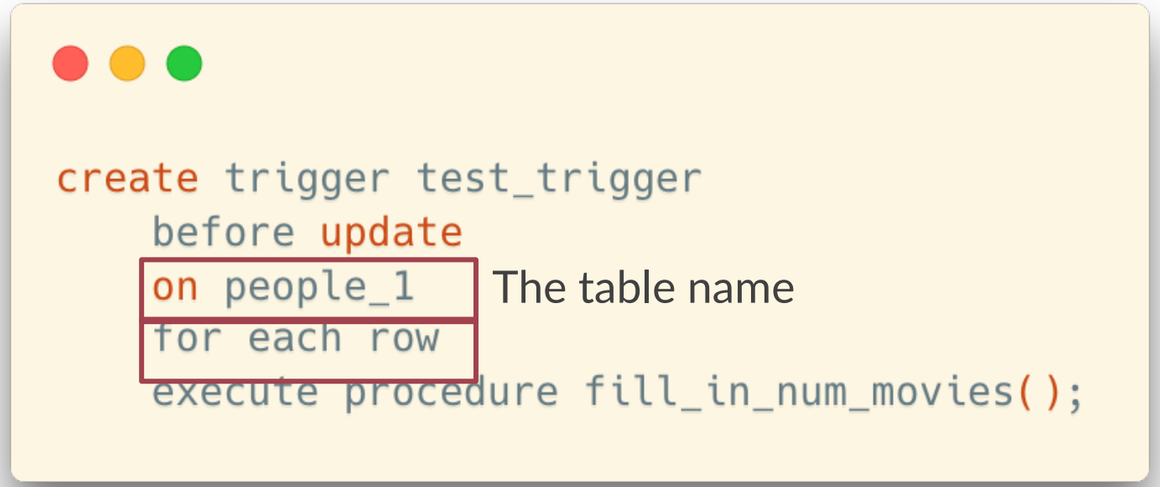
```
create trigger test_trigger
    before update
    on people_1
    for each row
    execute procedure fill_in_num_movies();
```

{ BEFORE | AFTER | INSTEAD OF } { event [OR ...] }

- Specify when the trigger will be executed
  - `before` | `after`
- … and on what operations the trigger will be executed
  - `insert [or update [or delete]]`

Example of Triggers

- Create a trigger



“for each row”
or
“for each statement” (default)

Example of Triggers

- Create a trigger

```
create trigger test_trigger
    before update
    on people_1
    for each row
    execute procedure fill_in_num_movies();
```

The actual procedure for the trigger

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well

```
create or replace function fill_in_num_movies()
    returns trigger
as
$$
begin
    select count(distinct c.movieid)
    into new.num_movies
    from credits c
    where c.peopleid = new.peopleid;
    return new;
end;
$$ language plpgsql;
```

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well

```
create or replace function fill_in_num_movies()
    returns trigger
as
$$
begin
    select count(distinct c.movieid)
    into new.num_movies
    from credits c
    where c.peopleid = new.peopleid;
    return new;
end;
$$ language plpgsql;
```

"trigger" is the return type

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well

```
create or replace function fill_in_num_movies()
    returns trigger
as
$$
begin
    select count(distinct c.movieid)
    into new.num_movies
    from credits c
    where c.peopleid = new.peopleid;
    return new;
end;
$$ language plpgsql;
```

”new” and “old” are two internal variables that represents the row before and after the changes

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well

```
create or replace function fill_in_num_movies()
    returns trigger
as
$$
begin
    select count(distinct c.movieid)
    into new.num_movies
    from credits c
    where c.peopleid = new.peopleid;
    return new;
end;
$$ language plpgsql;
```

Remember to return the result which will be used in the `update` statement

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well
    - Remember to create the procedure before creating the trigger

- Run test updates

```
-- create the procedure fill_in_num_movies() first

-- then, create the trigger

-- finally, we can run some test update statements
update people_1 set num_movies = 0 where people_1.peopleid <= 100;
```

Example: Auditing

- Another example of trigger is keeping an audit trail
  - Not really care about people who read data
    - Remember that `select` cannot fire a trigger, although with the big products you can trace all queries
  - … but it may be useful for checking people who modify data that they aren't supposed to modify

Example: Auditing

- Trace the insertions and updates to employees in a company

```
create table company(
   id int primary key     not null,
   name           text    not null,
   age            int     not null,
   address        char(50),
   salary         real
);


create table audit(
   emp_id int not null,
   change_type char(1) not null,
   change_date text not null
);
```

Example: Auditing

- Trace the insertions and updates to employees in a company

```
create trigger audit_trigger
    after insert or update
    on company
    for each row
execute procedure auditlogfunc();


create or replace function auditlogfunc() returns trigger as
$example_table$
begin
    insert into audit(emp_id, change_type, change_date)
    values (new.id,
            case
                when tg_op = 'UPDATE' then 'U'
                when tg_op = 'INSERT' then 'I'
                else 'X'
            end,
            current_timestamp);
    return new;
end ;
$example_table$ language plpgsql;
```

Example: Auditing

- Trace the insertions and updates to employees in a company

```
insert into company (id, name, age, address, salary)
values (2, 'Mike', 35, 'Arizona', 30000.00);
```

company

	id	name	age	address	salary
1	2	Mike	35	Arizona	30000

audit

	emp_id	change_type	change_date
1	2	I	2022-04-25 18:37:35.515151+00

View

Recall: Function

- Used for returning numbers, strings, dates, etc.
  - Or, return a single value

```
select full_name(first_name, surname)
from people
where surname like '%(von)';
```

… returns a string of the full name

View

- Reuse the relational operations, like how we reuse some code in a program
  - ... which returns relations (tables) instead of simple values

View

- Example: Create a view
  - … where the result of the inner `select` query looks like:

```
create view vmovies as
select m.movieid,
       m.title,
       m.year_released,
       c.country_name
from movies m join countries c
    on c.country_code = m.country;
```

	movieid	title	year_released	country_name
1	1	12 stulyev	1971	Russia
2	2	Al-mummia	1969	Egypt
3	3	Ali Zaoua, prince de la rue	2000	Morocco
4	4	Apariencias	2000	Argentina
5	5	Ardh Satya	1983	India
6	6	Armaan	2003	India
7	7	Armaan	1966	Pakistan
8	8	Babettes gæstebud	1987	Denmark
9	9	Banshun	1949	Japan
10	10	Bidaya wa Nihaya	1960	Egypt
11	11	Variety	2008	United States
12	12	Bon Cop, Bad Cop	2006	Canada
13	13	Brilliantovaja ruka	1969	Russia
14	14	C'est arrivé près de chez vous	1992	Belgium
15	15	Carlota Joaquina - Princesa d…	1995	Brazil
16	16	Cicak-man	2006	Malaysia
17	17	Da Nao Tian Gong	1965	China
18	18	Das indische Grabmal	1959	Germany
19	19	Das Leben der Anderen	2006	Germany
20	20	Den store gavtyv	1956	Denmark

View vs. Table

- In practice, there isn't much to a view
  - It’s basically a named query

```
create view vmovies(v_id, v_title, v_year, v_country) as
select m.movieid,
       m.title,
       m.year_released,
       c.country_name
from movies m
         join countries c
                    on c.country_code = m.country;
```

View vs. Table

- In practice, there isn't much to a view
  - It’s basically a named query

The columns can be renamed

- Otherwise, the original names in the tables will be used

```
create view vmovies(v_id, v_title, v_year, v_country) as
select m.movieid,
       m.title,
       m.year_released,
       c.country_name
from movies m
         join countries c
                    on c.country_code = m.country;
```

View vs. Table

- In practice, there isn't much to a view
  - It’s basically a named query

The columns can be renamed

- Otherwise, the original names in the tables will be used

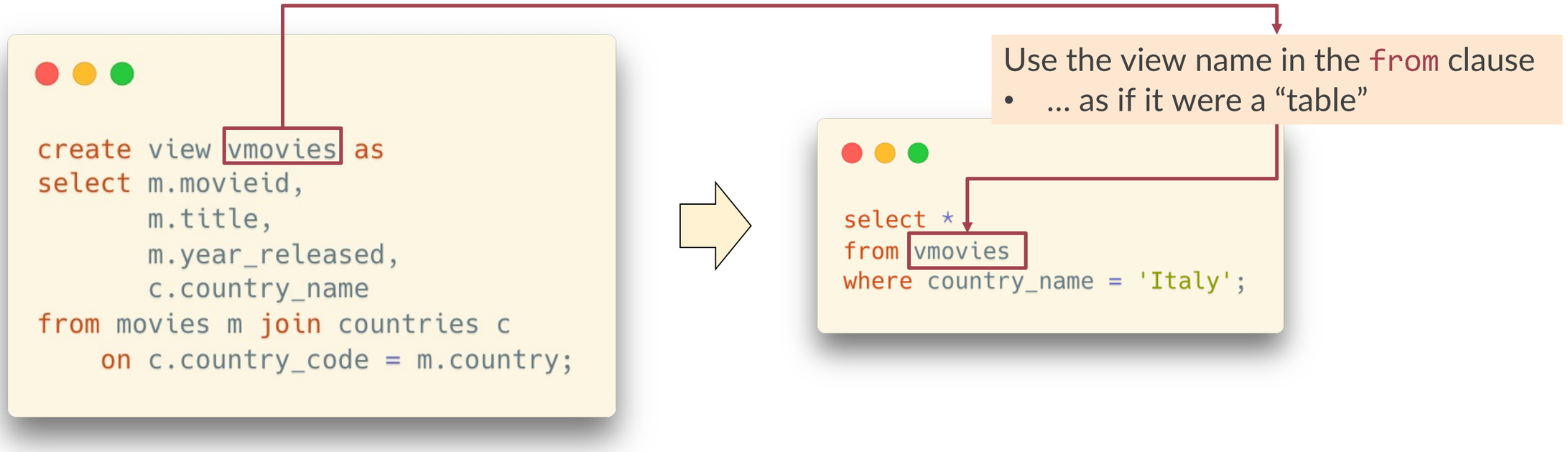
```
create view vmovies(v_id, v_title, v_year, v_country) as
select m.movieid,
       m.title,
       m.year_released,
       c.country_name
from movies m
         join countries c
                      on c.country_code = m.country;
```

Drop the existing view before creating a new one with the same name:

```
drop view vmovies;
```

View vs. Table

- Once the view is created,
  - … we can query the view exactly as if it were a table



View vs. Table

- A view can be as a complicated query as you want
  - It will usually return something that isn't as normalized as your tables, but easier to understand

- View is like a function of a `select` query
  - The view relation is not precomputed and stored
  - The DBMS stores the query expression associated with the view relation
  - The view is computed by executing the query, whenever the virtual relation is used
    - Virtual relation is created whenever needed, on demand

Content in Views is Dynamic

- When the rows change in the tables where a view relies on, the result of a view will change as well
  - Think it like a query result, or a table variable
  - * Or, think it like a relational function which returns a relation
  - A function of a `select` query
    - The view is computed by executing the query, whenever the virtual relation is used

Content in Views is Dynamic

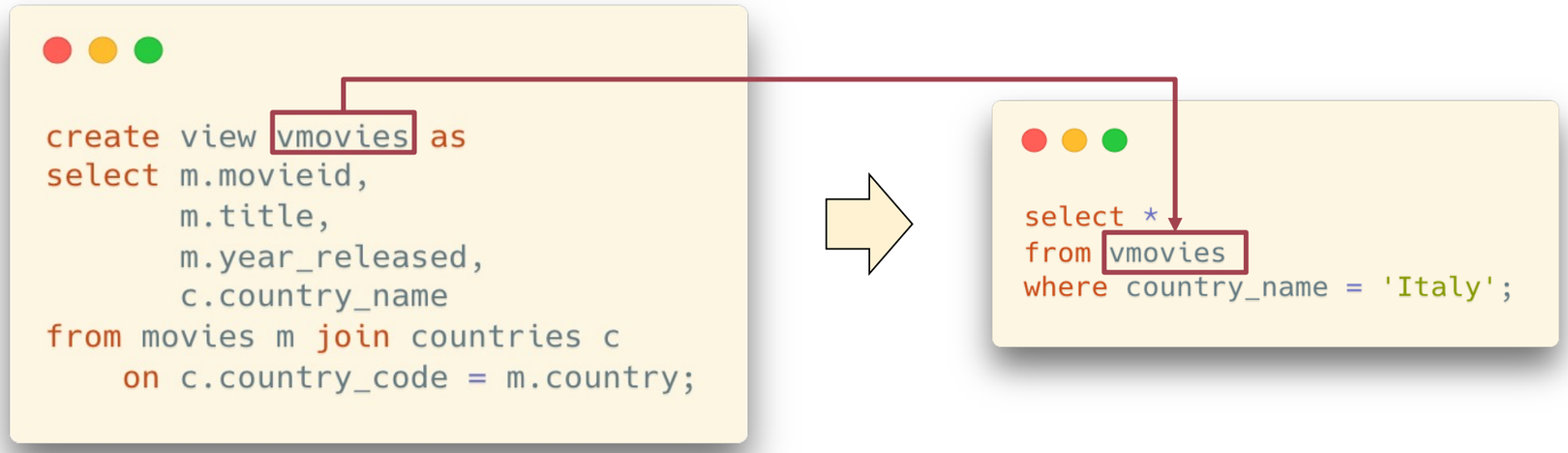
- In a view, the columns are the existing ones in tables when the view was created
  - Columns added later to tables won't be added in the view, even if the view was created with "`select *`"
    - And, "`select *`" is a bad practice in view. We will not know what exact columns are selected once the columns in the tables are changed.

```sql
-- Create the view first
create view vmovies as
select *  -- IT IS A BAD PRACTICE OF USING * HERE
from movies m join countries c
    on c.country_code = m.country;

-- Alter the table then
alter table movies add column test_col int not null default 0;

-- There won't be a column "test_col" in the query result
select * from vmovies;
```

View vs. Table (Cont.)

- View is like Table
  - Sometimes it is used as a table

  - Usage Scenario 1: Simplify Complex Queries
    - Simplify a complicated query result into a single named query
    - e.g., Many business reports are based on the same set of joins, with just variations on the columns that you aggregate or order by



View vs. Table (Cont.)

- View is like Table
  - Sometimes it is used as a table

  - Usage Scenario 2: An alternative way to implement E-R models
    - Sometimes, we may not be able to use tables to model entities and relationships
      - Access control
      - Dirty and messy original data
    - Since views look like tables, we can use views to represent entities or relationships
      - … based on some existing objects (like tables)

Drawback of Views

- View is like Table
  - But it is still not a table

  - In some cases, it may cause performance issues or unnecessary operations
    - * … since the users of the views don’t know the details of the views
  - A function of a `select` query
    - The view is computed by executing the query, whenever the virtual relation is used

Drawback of Views

- View looks like Table, tastes like Table
  - Example: A refined way to display the credit information with a view

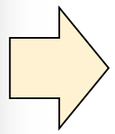
```
create view vmovie_credits
as
select m.title,
       m.year_released release,
       case c.credited_as
           when 'A' then 'Actor'
           when 'D' then 'Director'
           else '?'
           end duty,
       full_name(p.first_name, p.surname) name
from movies m
         inner join credits c
                    on c.movieid = m.movieid
         inner join people p
                    on p.peopleid = c.peopleid;
```

	title	release	duty	name
1	"Erogotoshitachi" yori Jinruigaku nyūmon	1966	Director	Shohei Imamura
2	"Erogotoshitachi" yori Jinruigaku nyūmon	1966	Actor	Shoichi Ozawa
3	"Erogotoshitachi" yori Jinruigaku nyūmon	1966	Actor	Sumiko Sakamoto
4	'76	2016	Actor	Ramsey Nouah
5	'76	2016	Actor	Ibinabo Fiberesima
6	(T)Raumschiff Surprise	2004	Actor	Michael Herbig
7	(T)Raumschiff Surprise	2004	Director	Michael Herbig
8	(T)Raumschiff Surprise	2004	Actor	Til Schweiger
9	(T)Raumschiff Surprise	2004	Actor	Anja Kling
10	(T)Raumschiff Surprise	2004	Actor	Rick Kavanian
11	(T)Raumschiff Surprise	2004	Actor	Christian Tramitz
12	(T)Raumschiff Surprise	2004	Actor	Sky du Mont
13	002 operazione luna	1965	Actor	Linda Sini
14	002 operazione luna	1965	Director	Lucio Fulci

Drawback of Views

- View looks like Table, tastes like Table
  - Example: A refined way to display the credit information with a view
    - However, if we only want to get all distinct movie titles, it will return the correct result, but there are a lot of useless works in the internal query of the view

```
select distinct title
from vmovie_credits
```



```
create view vmovie_credits
as
select m.title,
       m.year_released release,
       case c.credited_as
           when 'A' then 'Actor'
           when 'D' then 'Director'
           else '?'
           end duty,
       full_name(p.first_name, p.surname) name
from movies m
         inner join credits c
                    on c.movieid = m.movieid
         inner join people p
                    on p.peopleid = c.peopleid;
```

Do we really need the joins?

View vs. Materialized View

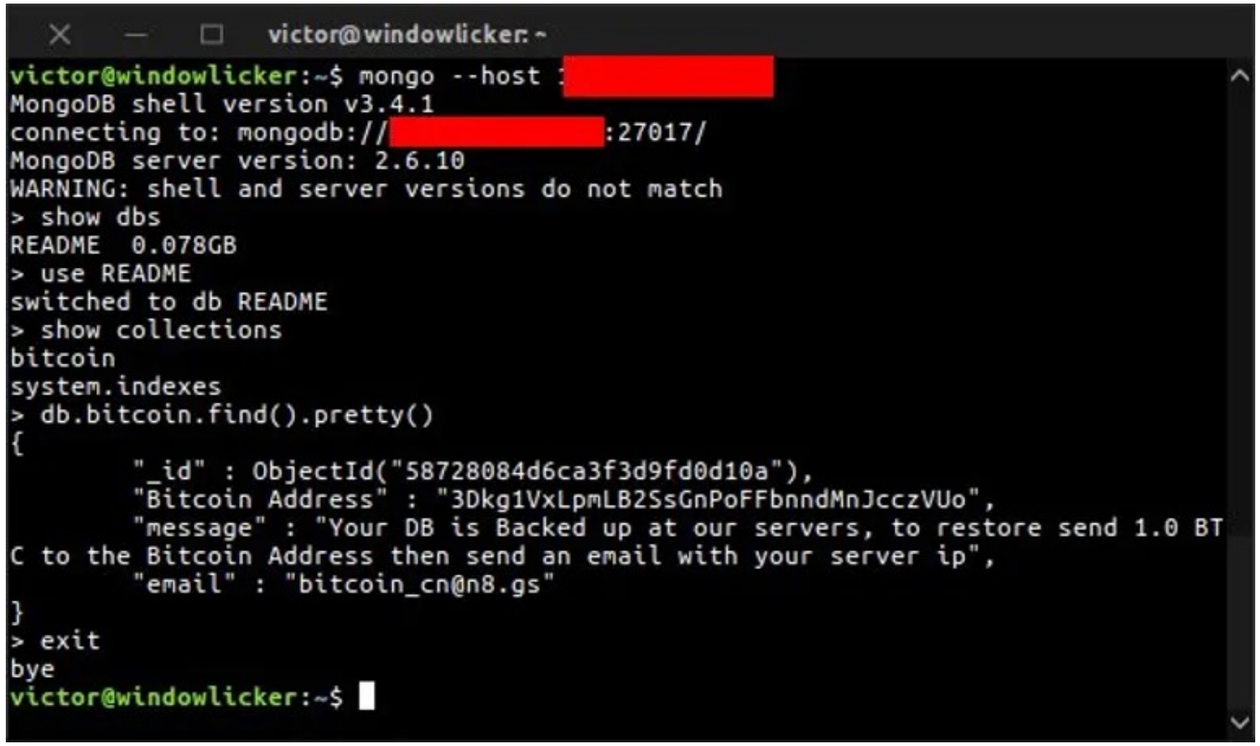
- Normal views dynamically retrieve data from the original tables
  - However, the performance won't be good

- Materialized View: A cached result of a view
  - When the corresponding tables are not changing rapidly, we can cache the view results (i.e., materialize the results)
    - Better performance than dynamic retrieval
  - The results are not updated each time the view is used
    - Manual update, or use the help of triggers

Access Control

Authentication

- To access a database, you must be authenticated
  - Entering a username and a password.
- If you don’t set a proper password, you will put your database system in danger



https://www.zdnet.com/article/mongodb-ransacked-now-27000-databases-hit-in-mass-ransom-attacks/

Privileges

- Besides, DBMS usually provide another layer of access control on objects in the system
  - Operations (select, update, insert, delete, etc.)
  - Objects (table, database, views, trigger, etc.)

Grant and Revoke Access

- `grant` and `revoke`
  - Can be used on the table or the database level

```
-- grant <right> to <account>
grant select on movies to test_user;

-- revoke <right> from <account>
revoke select on movies from test_user;
```

The possible privileges are:

```
SELECT
INSERT
UPDATE
DELETE
TRUNCATE
REFERENCES
TRIGGER
CREATE
CONNECT
TEMPORARY
EXECUTE
USAGE
```

https://www.postgresql.org/docs/current/sql-grant.html

How Can Views Help

- User access control
  - By creating a view that only returns rows associated with the user name, we can control the privileges for a specific user in this table

```
create view my_stuff
as
select * from stuff
where username = user
```

```
-- E.g., restrict the users to only access
-- the movies assigned to their user names
create view user_view as
select movieid,
       title,
       country,
       year_released,
       runtime
from movies
where user_name = user;
```

`user`: an internal variable that returns the current user name

Outline

Part 1: Relational Language

Basic SQL
- Create Tables and Insert Data
- Basic Queries

Intermediate SQL
- Advanced Join
- Subquery
- Window Function
- *Transactions*

Advanced SQL
- Function
- Procedure
- Trigger
- View

Part 2: DB Design and Application

Relational DB Design
- The Entity-Relationship Model
- E-R Diagram Design

Relational Algebra

Normalization
- Functional Dependency
- Normal Forms
- FD Theory
- Decomposition w/ FD

Application Design
- Use Databases in Programs
- Complex Data Types (JSON, XML, etc.)

Part 3: Internal Mechanism Of Relational DB Systems

Storage Management
- Physical and Logical Storage Structure
- Indexing

Query Management
- Query Processing
- Query Optimization

Transaction Management
- Transaction
- Concurrency

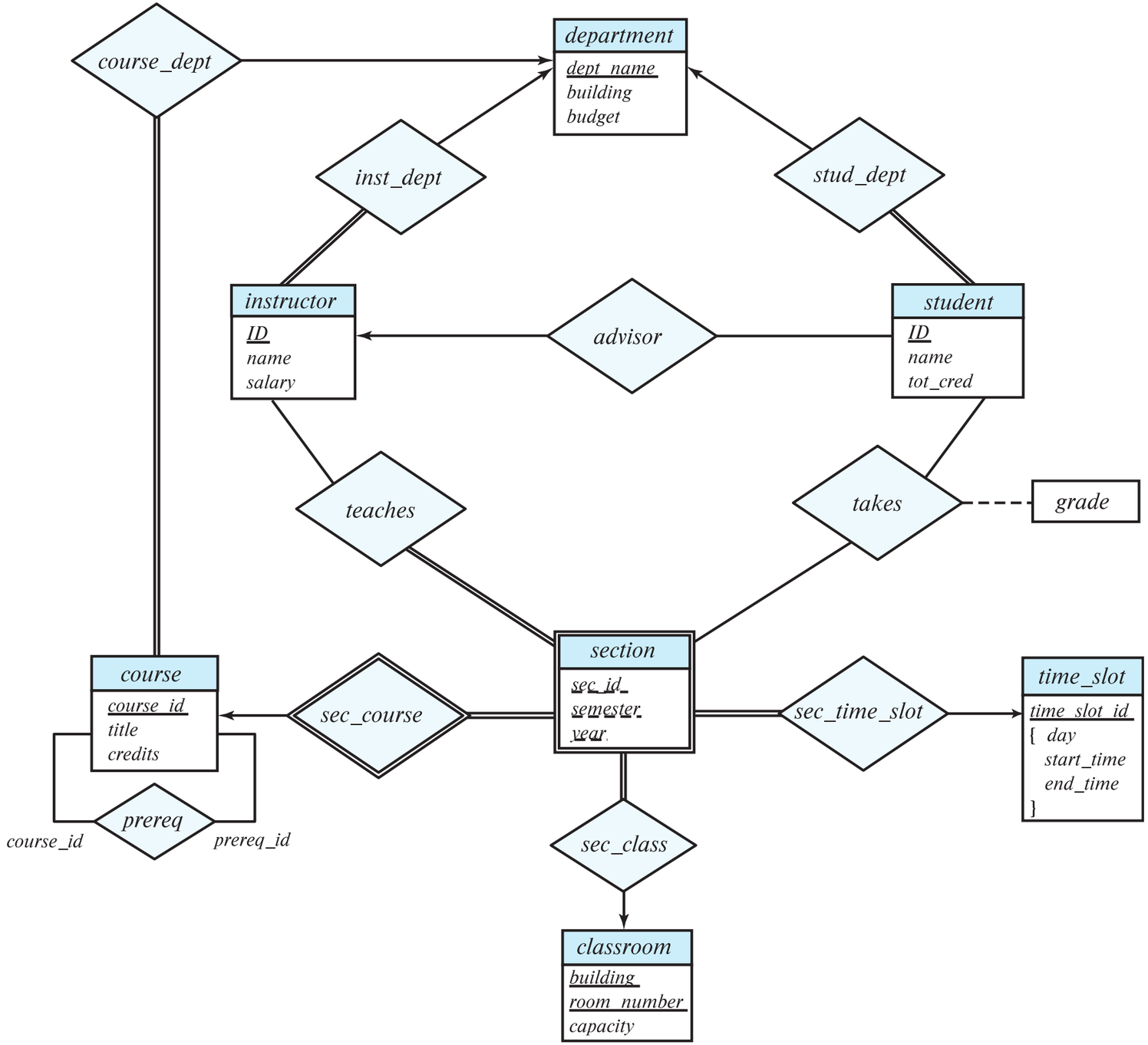
Part 4: Miscellaneous Topics

Big Data
- Big Data Storage
- Query and Process Big Data

Advanced Topics
- Beyond Relational DB
- Data Analytics

Entity-Relationship Model (E-R Model)
Entity-Relationship Diagram (E-R Diagram)

The New Running Example



Design Phases

- Initial phase: characterize fully the data needs of the prospective database users
  - Interact extensively with domain experts
  - The outcome is a specification of user requirements

Design Phases

- Second phase: choosing a data model
  - E.g., relational model, entity-relationship model, semi-structured data model, and object-oriented data model
  - Applying the concepts of the chosen data model
  - Translating these requirements into a conceptual schema of the database
    - Detailed overview of the enterprise
  - A fully developed conceptual schema indicates the functional requirements of the enterprise
    - Describes operations (e.g., update, retrieval, delete) that will be performed on the data
- Entity-relationship model is typically used
  - Outcome is an E-R diagram that provides a graphic representation of the schema
    - Entities that are represented in the database
    - Attributes of the entities
    - Relationships among entities
    - Constraints on the entities and relationship

Design Phases

- Final Phase: Moving from an abstract data model to the implementation of the database
  - Logical Design – Deciding on the database schema
    - Database design requires that we find a “good” collection of relation schemas
    - Business decision
      - What attributes should we record in the database?
    - Computer Science decision
      - What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
  - Physical Design – Deciding on the physical layout of the database
    - E.g., the form of file organization and choice of index structures
  - Physical schema is easy to change after application is built
    - But logical schema is not, because changes may affect a number of queries and updates scattered across application code

Design Alternatives

- In designing a database schema, we must ensure that we avoid two major pitfalls
  - **Redundancy**: a bad design may result in repeated information
    - E.g., store course identifier and title of a course for each course offering
      - Only store course identifier is sufficient
    - Redundant representation of information may lead to data inconsistency among the various copies of information
      - E.g., update is not performed on all the copies
  - **Incompleteness**: a bad design may make certain aspects of the enterprise difficult or impossible to model
    - E.g., only have entity for course offering, but without entity for courses
      - Impossible to model new courses that are not offered yet

Design Alternatives

- Avoiding bad designs is not enough
  - There may be many good designs from which we must choose

- For example, a customer who buys a product
  - The sale is a relationship between the customer and the product?
  - The sale is a relationship among the customer, the product, and the sale itself?
    - i.e., the sale can be considered as an entity

- Database design can be difficult
  - When #entities and #relationships are large

Design Approaches

- Entity-Relationship Model (covered in this chapter)
  - Specifies an enterprise schema representing overall logical structure of a database
  - Models an enterprise as a collection of entities and relationships
    - **Entity**: a “thing” or “object” in the enterprise that is distinguishable from other objects
      - Described by a set of attributes
    - **Relationship**: an association among several entities
  - Represented diagrammatically by an entity-relationship diagram (E-R diagram)
    - Express overall logical structure of a database graphically

- Normalization Theory (coming in the next few weeks)
  - Formalize what designs are bad, and test for them

Entity and Entity Sets

- An entity is an object that exists and is distinguishable from other objects
  - Concrete entity: specific person, company, plant, book
  - Abstract entity: flight reservation, course, course offering
  - An entity is represented by a set of attributes; i.e., descriptive properties possessed by all members of an entity set
  - Example:

```
instructor = (ID, name, salary)
course = (course_id, title, credits)
```

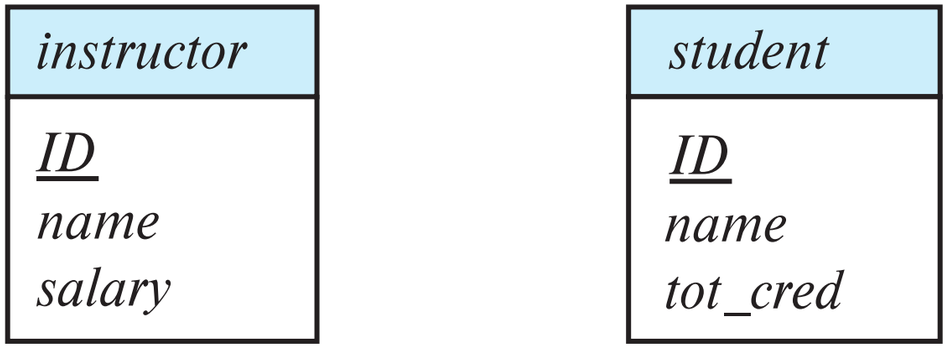
  - A subset of the attributes form a primary key of the entity set; i.e., uniquely identifying each member of the set
    - Government-issued ID number as the primary key
      - May have privacy and security issues
    - Enterprise-issued ID number

Entity and Entity Sets

- An entity set is a set of entities of the same type that share the same properties
  - Example: set of all persons, companies, trees, holidays
  - Entity sets may not be disjoint (e.g., person vs. instructor and student)

Representing Entity sets in ER Diagram

- Entity sets can be represented graphically as follows:
  - Rectangles represent entity sets.
  - Attributes listed inside entity rectangle
  - Underline indicates primary key attributes



Relationship Sets

- A relationship is an association among several entities

```
44553 (Peltier)   advisor          22222 (Einstein)
student entity  relationship set   instructor entity
```

- A relationship set is a set of relationships of the same type (e.g., advising)
  - A mathematical relation among n ≥ 2 (possibly non-distinct) entities, each taken from entity sets

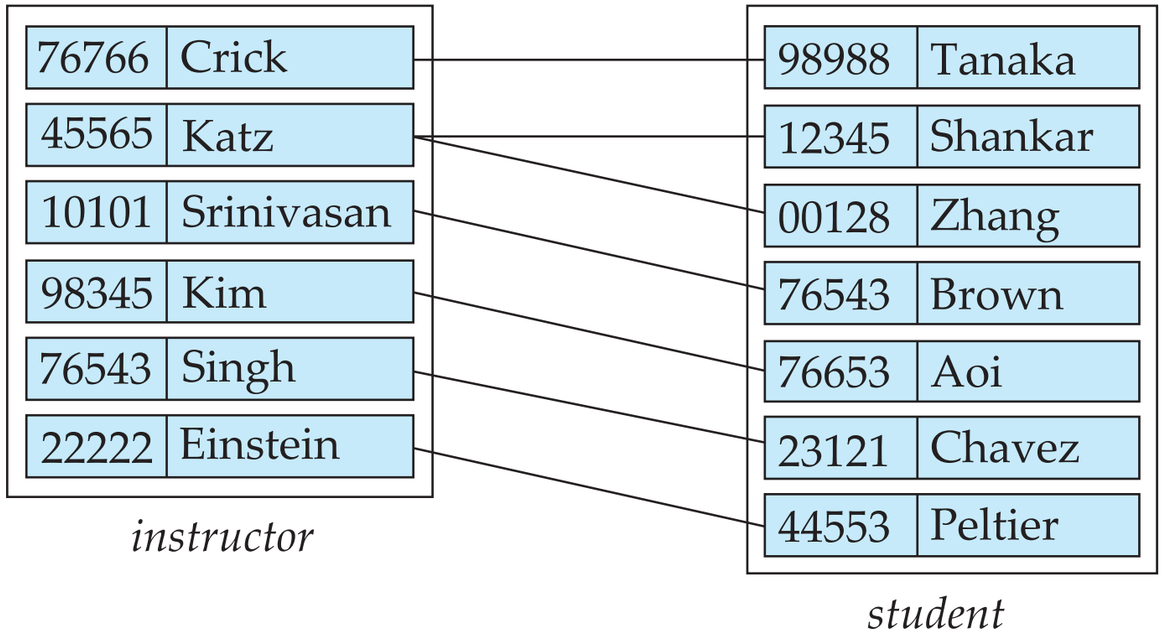
$$\{(e_1, e_2, \ldots e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

where (e_1, e_2, ..., e_n) is a relationship

  - Example: (44553,22222) ∈ advisor
  - The association between entity sets is referred to as participation
    - Entity sets E_1, E_2, … , E_n participate in relationship set R

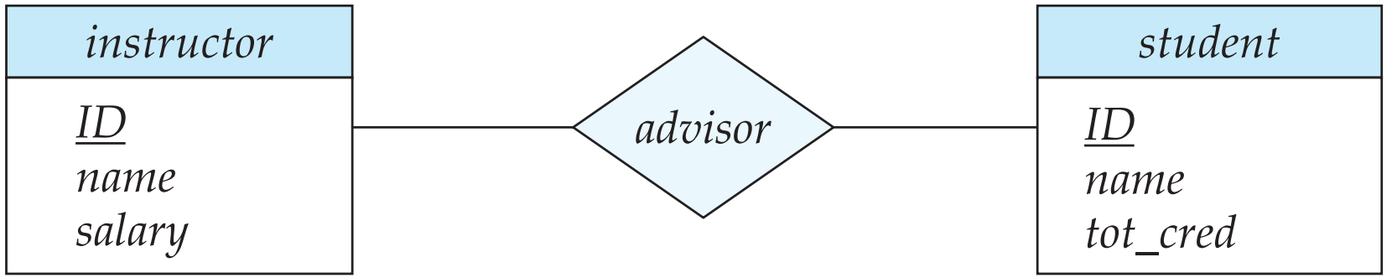
Relationship Sets

- Example: we define the relationship set `advisor` to denote the associations between students and the instructors who act as their advisors.
  - Pictorially, we draw a line between related entities



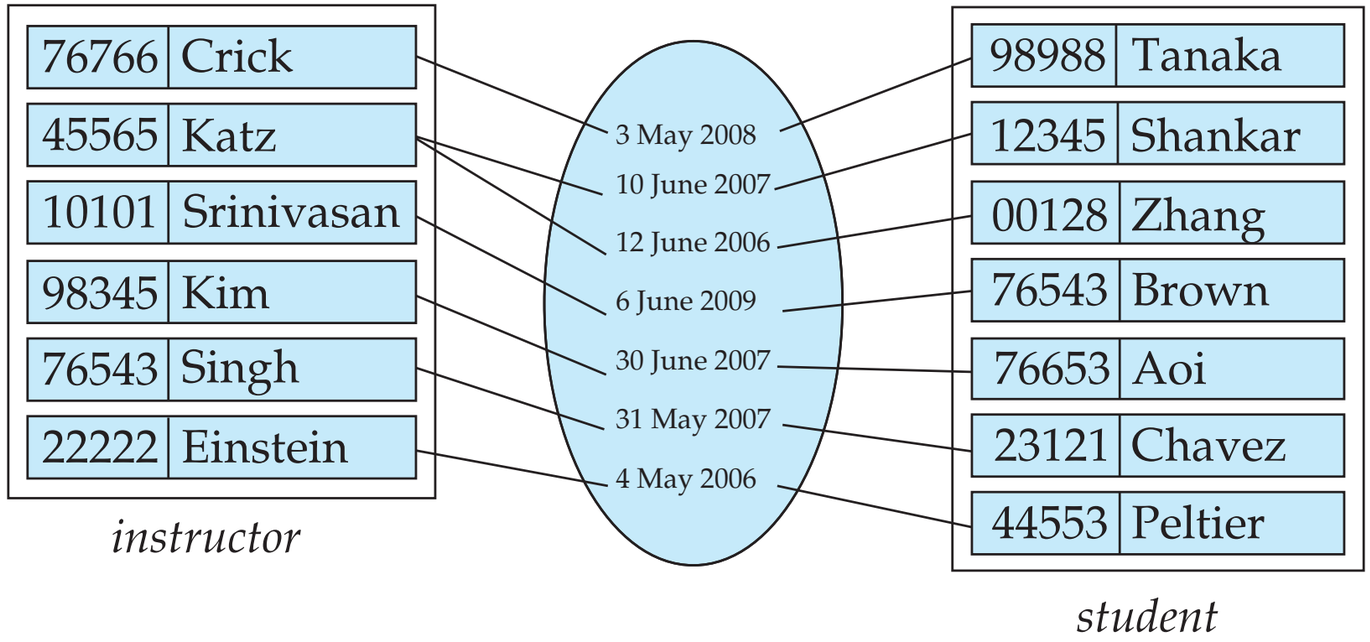
Representing Relationship Sets via E-R Diagrams

- Use diamonds to represent relationship sets



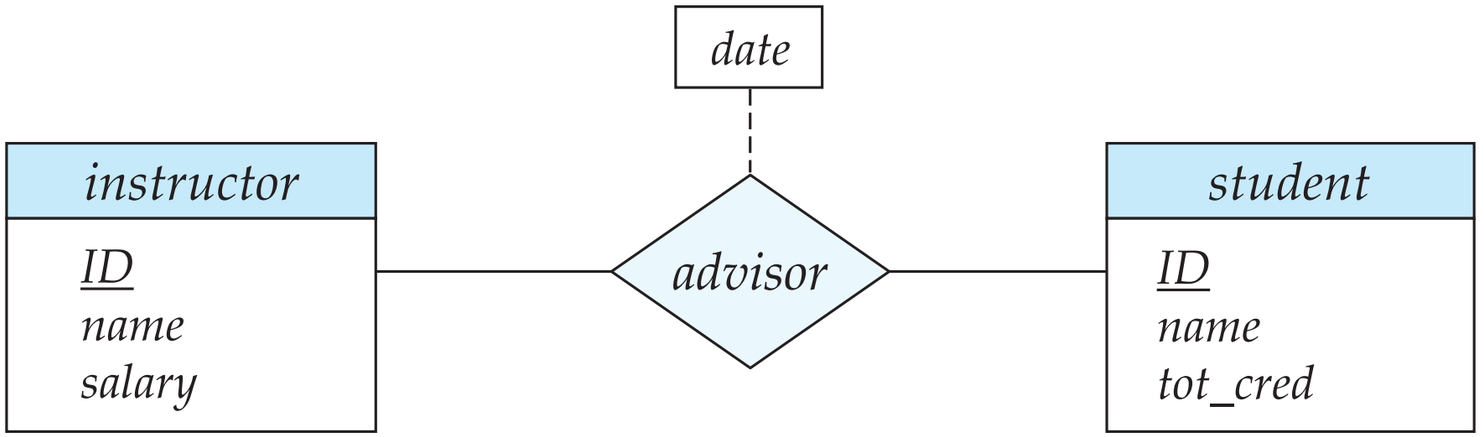
Relationship Sets (Cont.)

- A descriptive attribute can be associated with a relationship set.
  - E.g., the advisor relationship set between entity sets instructor and student may have the attribute `date`, which tracks when the student started being associated with the advisor



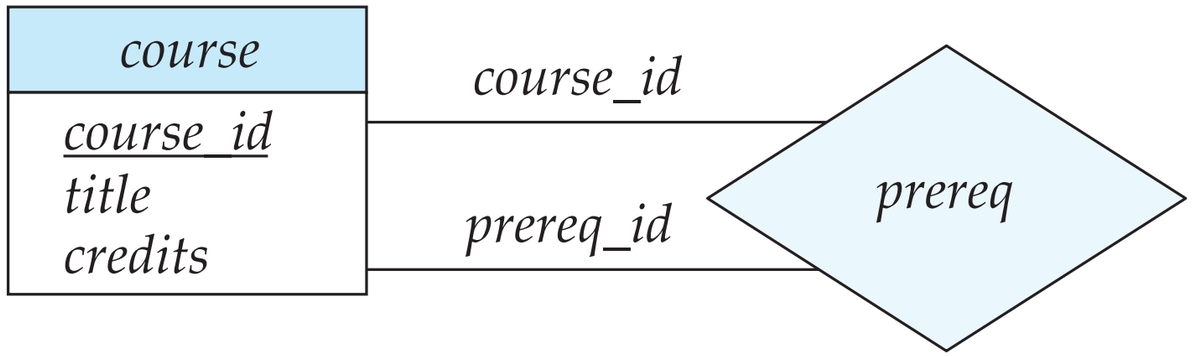
Relationship Sets with Attributes

- Represented by using an undivided rectangle
- Linked to the diamond representing the relationship set via a dashed line



Roles

- Entity sets of a relationship need not be distinct (i.e., can be the same entity set)
  - We can create self-pointing relationships for an entity set
  - Each occurrence of an entity set plays a "role" in the relationship
  - Example: A relationship set to represent the prerequisites of a course
    - E.g., Data Structure depends on Introduction to Programming
    - The labels "`course_id`" and "`prereq_id`" are called roles



- If the entity sets participating in a relationship are distinct,
  - Their roles are implicit and usually are not specified

Degree of a Relationship Set

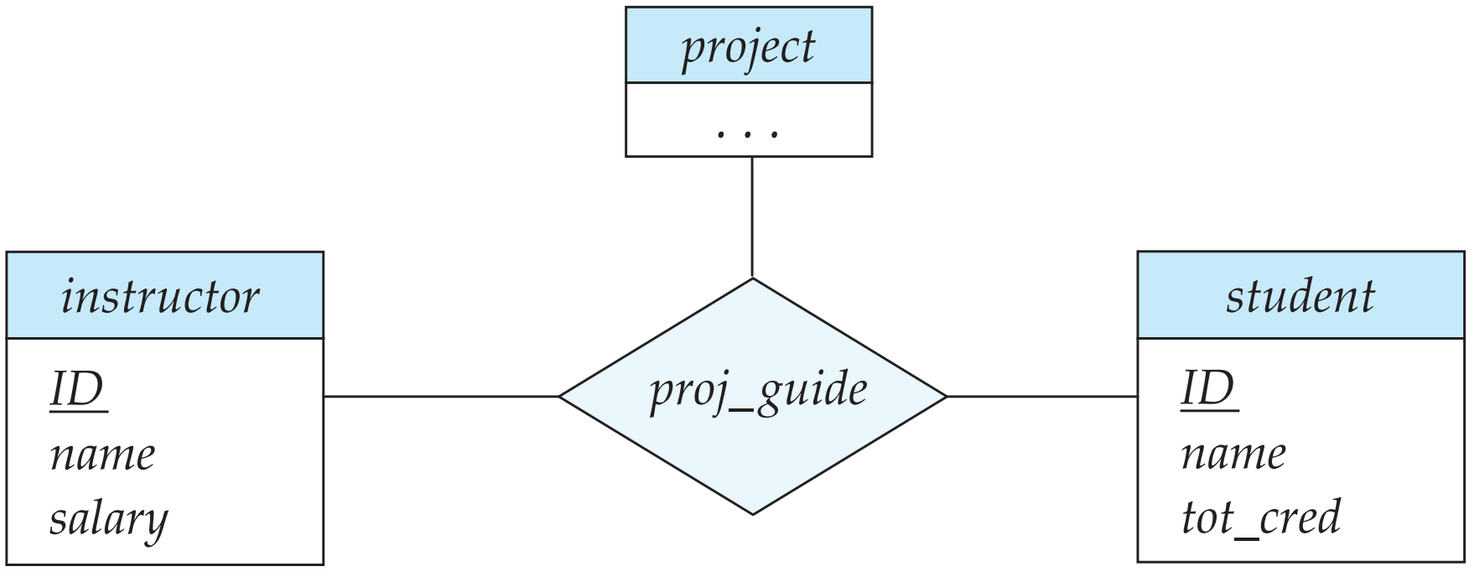
- Denoted as the number of entity sets participating in a relationship set

- Binary relationship
  - Involve **two** entity sets (or degree two)
  - Most relationship sets in a database system are binary

- Relationships between more than two entity sets are rare
  - E.g., students work on research projects under the guidance of an instructor
    - relationship `proj_guide` is a ternary relationship among instructor, student, and project
      - A particular student is guided by a particular instructor on a particular project

Non-binary Relationship Sets

- Although most relationship sets are binary,
  - There are occasions when it is more convenient to represent relationships as non-binary
- E-R Diagram with a Ternary Relationship

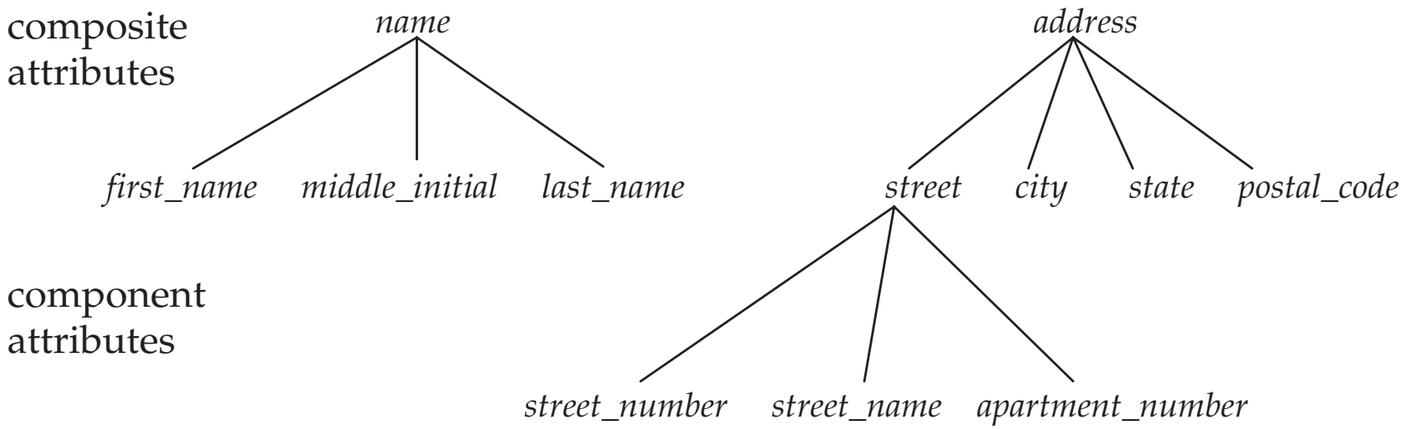


Complex Attributes

- Attribute types
  - Simple (i.e., not divided into subparts) and composite (i.e., divided into subparts) attributes

Composite Attributes

- Composite attributes allow us to divided attributes into subparts
  - Sometimes we may only use part of the attributes
  - In this case, composite attribute is a good design choice
- Allow us to group together related attributes, making the modeling cleaner
- A composite attribute may appear as a hierarchy



instructor

ID
name
- *first_name*
- *middle_initial*
- *last_name*

address
- *street*
  - *street_number*
  - *street_name*
  - *apt_number*
- *city*
- *state*
- *zip*

{ phone_number }
date_of_birth
age ( )

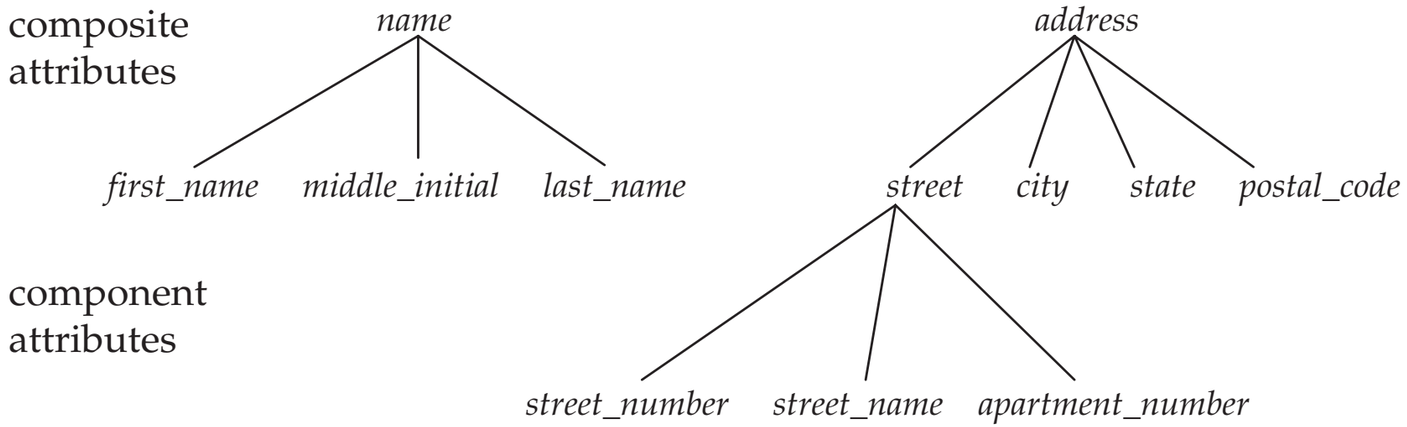
E-R notation

Complex Attributes

- Attribute types
  - Simple (i.e., not divided into subparts) and composite (i.e., divided into subparts) attributes
  - Single-valued and multivalued attributes
    - Single-valued attribute: e.g., for a student, only one *student_id*
    - Multivalued attribute
      - *phone_numbers*: a person can have 0, 1, or multiple phone numbers at the same time
      - *grades*: each student may have multiples grades in a course
      - *dependent_name*: an instructor may be hired by 0, 1, or multiple departments
  - Derived attributes
    - Can be computed from other attributes
    - Example: *age* computed based on *date_of_birth*, #students advised by an instructor
- **Domain**: The set of permitted values for each attribute
  - *course_id* might be the set of all text strings of a certain length
  - *semester* might be strings from the set {Fall, Winter, Spring, Summer}

E-R notation of Composite Attributes

- E-R notations for
  - Composite attributes
  - Multivalued attributes
  - Derived attributes

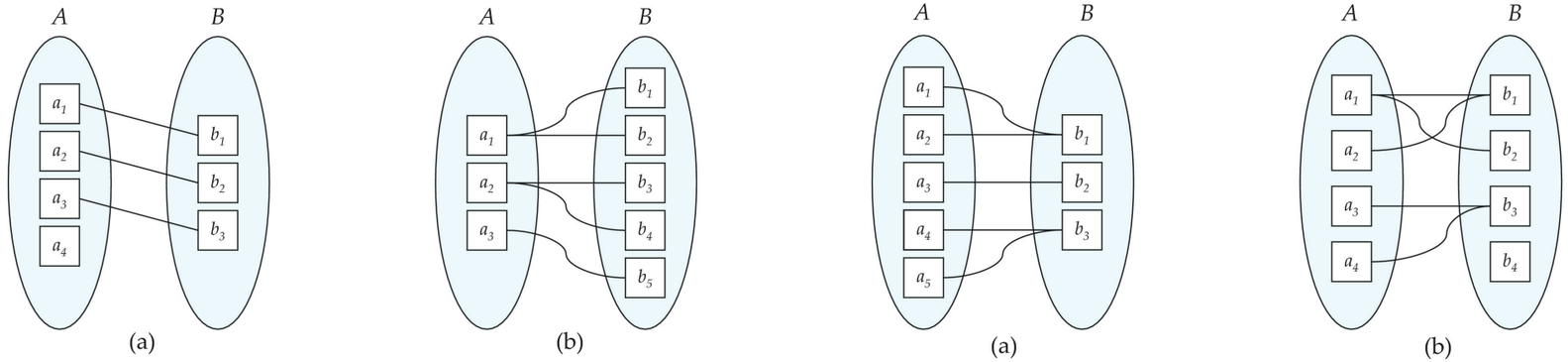


instructor

ID
name
first_name
middle_initial
last_name
address
street
street_number
street_name
apt_number
city
state
zip
{ phone_number }
date_of_birth
age ( )

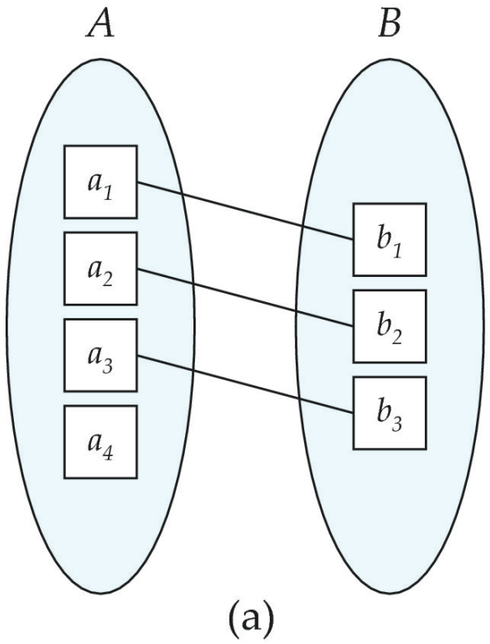
Mapping Cardinality Constraints

- Mapping Cardinality（映射基数）
  - Express the number of entities to which another entity can be associated via a relationship set
    - Most useful in describing binary relationship sets
    - Can contribute to the description of non-binary relationship sets

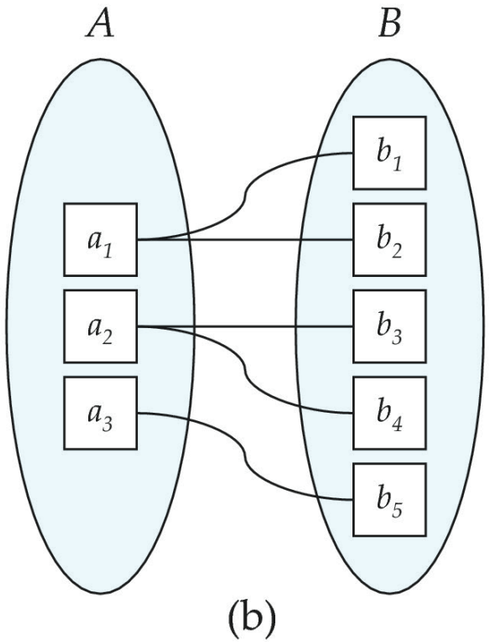


- For a binary relationship set, the mapping cardinality must be one of the following:
  - One to one, one to many, many to one, many to many

Mapping Cardinalities



(a)



(b)

One to one

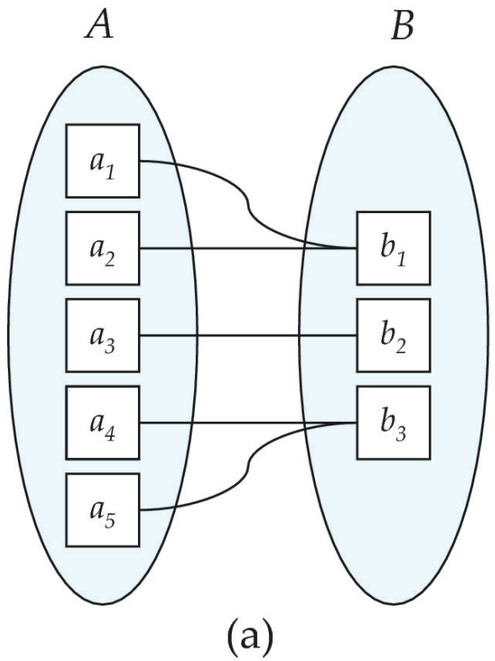
- Every entity in A is associated with at most one entity in B
- Same for entity set B

One to many

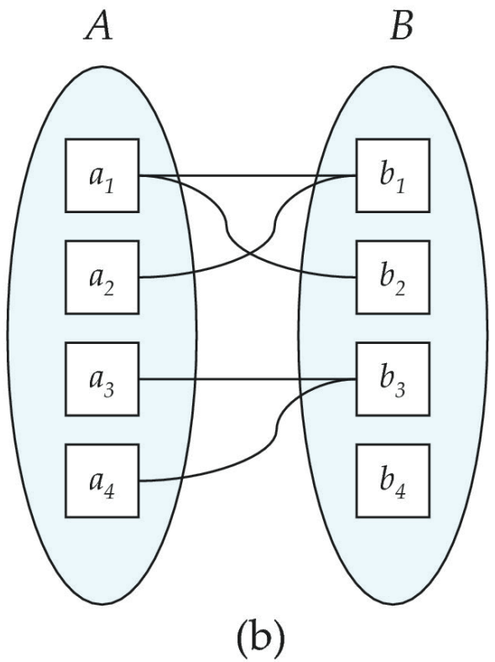
- Every entity in A is associated with 0, 1, or more entities in B
- Every entity in B is associated with at most one entity in A

Note: Some entities in *A* and *B* may not be mapped to any entities in the other set

Mapping Cardinalities



(a)



(b)

Many to one

- Every entity in A is associated with at most one entity in B
- Every entity in B is associated with 0, 1 or more entities in A

Many to many

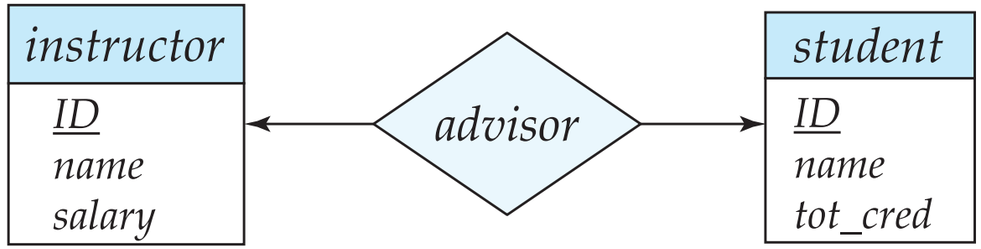
- Every entity in A is associated with 0, 1, or more entities in B
- Same for B

Note: Some entities in *A* and *B* may not be mapped to any entities in the other set

Representing Cardinality Constraints in ER Diagram

- We express cardinality constraints by:
  - drawing either a directed line (→), signifying "one"
  - or an undirected line (—), signifying "many"
- … between the relationship set and the entity set

- One-to-one relationship between an instructor and a student :
  - A student is associated with at most one instructor via the relationship advisor



Representing Cardinality Constraints in ER Diagram

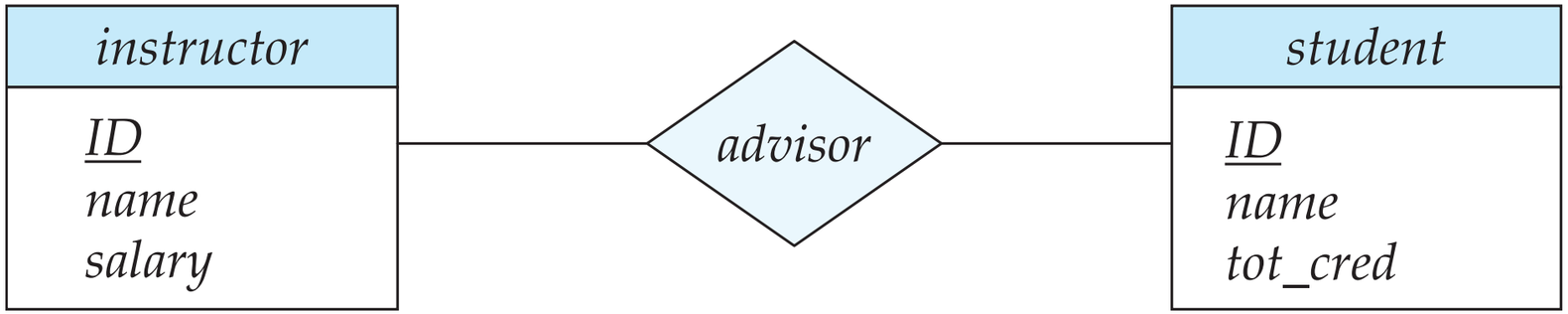
- One-to-many relationship between an instructor and a student
  - an instructor is associated with several (including 0) students via advisor
  - a student is associated with at most one instructor via advisor

Representing Cardinality Constraints in ER Diagram

- In a many-to-one relationship between an instructor and a student,
  - an instructor is associated with at most one student via advisor
  - and a student is associated with several (including 0) instructors via advisor

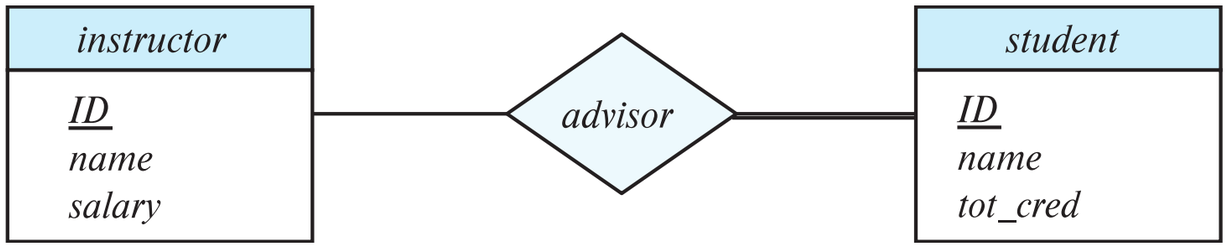
Representing Cardinality Constraints in ER Diagram

- Many-to-many relationship:
  - An instructor is associated with several (possibly 0) students via advisor
  - A student is associated with several (possibly 0) instructors via advisor



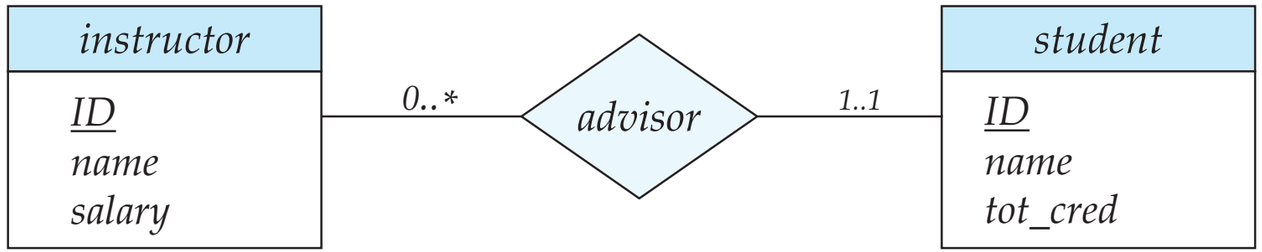
Total and Partial Participation

- Total participation (indicated by *undirected double line, i.e., =*)
  - Every entity in the entity set participates in at least one relationship in the relationship set
  - Example: Participation of student in advisor relation is total
    - i.e., every student must have an associated instructor
- Partial participation (undirected line, i.e., —)
  - Some entities may not participate in any relationship in the relationship set
  - Example: participation of instructor in advisor is partial



Notation for Expressing More Complex Constraints

- A line may have an associated minimum and maximum cardinality, shown in the form *l..h*, where l is the minimum and h the maximum cardinality
  - A minimum value of 1 indicates total participation
  - A maximum value of 1 indicates that the entity participates in at most one relationship
  - A maximum value of * indicates no limit



  - Example
    - Instructor can advise 0 or more students
    - A student must have 1 advisor; cannot have multiple advisors

Primary Key

- Primary keys provide a way to specify how entities and relations are distinguished

Primary Key for Entity Sets

- By definition, individual entities are distinct
  - From database perspective, the differences among them must be expressed in terms of their attributes

- The values of the attribute values of an entity must be such that they can uniquely identify the entity
  - No two entities in an entity set are allowed to have exactly the same value for all attributes

- A key for an entity is a set of attributes that suffice to distinguish entities from each other

Primary Key for Relationship Sets

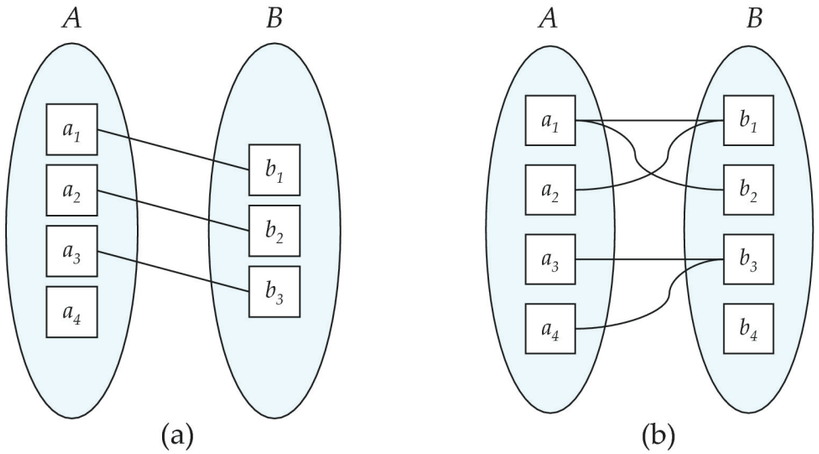
- To distinguish among the various relationships of a relationship set, we use the **individual primary keys** of the entities in the relationship set.
  - Let R be a relationship set involving entity sets E1, E2, .. En
  - The primary key for R consists of the union of the primary keys of entity sets E1, E2, ..En
  - If the relationship set R has attributes $a_1, a_2, .., a_m$ associated with it, the primary key of R also includes the attributes $a_1, a_2, .., a_m$

- Example: relationship set “advisor”.
  - The primary key consists of instructor.ID and student.ID

- The choice of the primary key for a relationship set depends on the mapping cardinality of the relationship set

Choice of Primary key for Binary Relationship

- Many-to-Many relationships
  - The union of the primary keys is a minimal superkey and is chosen as the primary key
- One-to-one relationships
  - The primary key of either one of the participating entity sets forms a minimal superkey, and either one can be chosen as the primary key.



* *K* is a **superkey** of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r(R)*

- Example: {*ID*} and {ID,name} are both superkeys of *instructor*.

Choice of Primary key for Binary Relationship

- One-to-Many relationships
  - The primary key of the “Many” side is a minimal superkey and is used as the primary key
- Many-to-one relationships
  - The primary key of the “Many” side is a minimal superkey and is used as the primary key