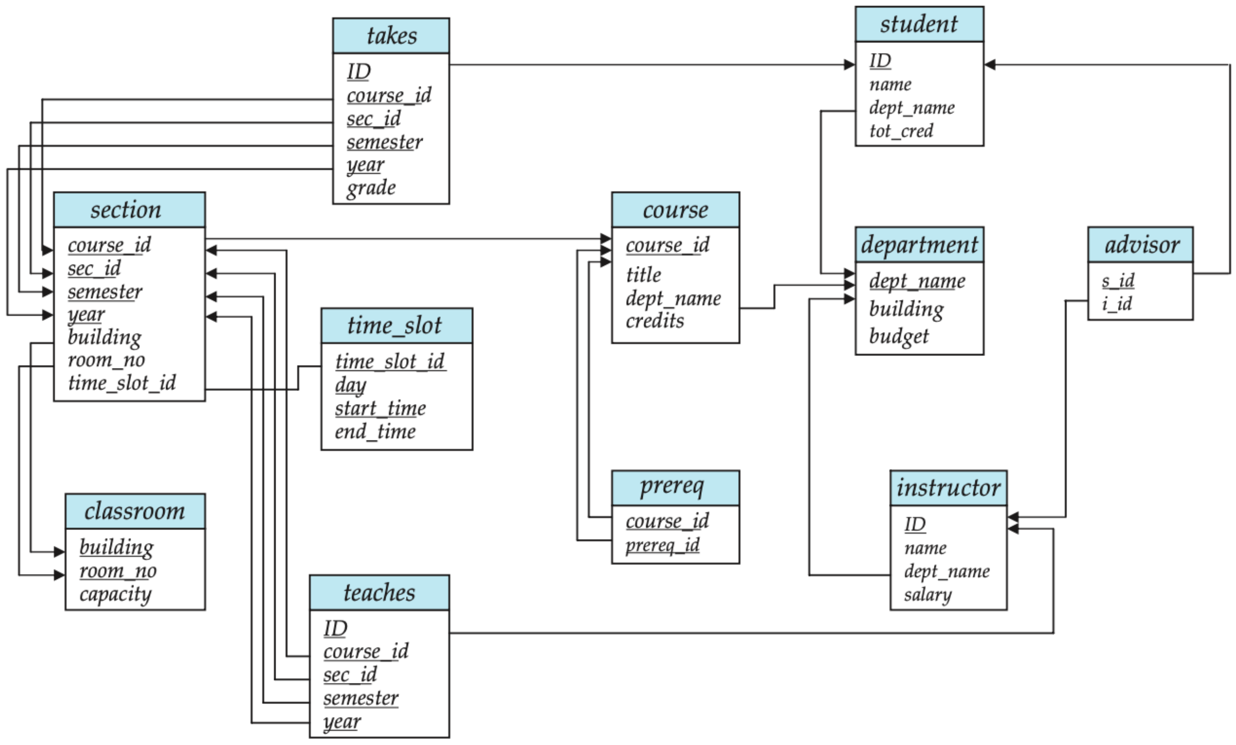
WEEK - 2 Demo

Queries on Single relations:

So far our example queries were on a single relation.

For Example: Consider a University Database



and the following query,

select *name*

from *instructor* ;

where instructure table has attributes (ID, name, dept_name, salary)

Some important cases:

- If we want to force the elimination of duplicates, we insert the keyword **distinct** after **select**. We can rewrite the preceding query as:

  select **distinct** *dept_name*

  from *instructor* ;

- The **select** clause may also contain arithmetic expressions involving the operators +, −, ∗, and / operating on constants or attributes of tuples. For example, the query,

  select *ID*, *name*, *dept_name, salary * 1.1*

  from *instructor* ;

outputs a list of all the instructors, their department name and their salaries when increased by 10%.

Queries on multiple relations:

Queries often need to access information from multiple relations. We now study how to write such queries.

1. Consider the University Database (ER Diagram) given above. Suppose we want to answer the query "Retrieve the names of all instructors, along with their department names and department building name."

Looking at the schema of the relation *instructor*, we realize that we can get the department name from the attribute *dept_name* but the department building name is present in the attribute *building* of the relation *department*.

To answer the above query, we list the relations that need to be accessed in the **from** clause, and specify the matching condition in the **where** clause.

select *name*, *instructor.dept_name*, *building*

from *instructor, department*

where *instructor.dept_name* = *department.dept_name*

(Note: The attribute *dept_name* required the prefix as the relation's name (instructor or department) and others did not. Why??)

2. To find instructor names and course identifiers for instructors in the Computer Science department, we could add an extra predicate to the **where** clause, as shown below.

   select *name*, *course_id*

   from *instructor, teaches*

   where *instructor.ID*= *teaches.ID* **and** *instructor.dept_name* = 'Comp. Sci.';

Aliasing or Renaming:

The basic syntax of table alias is as follows −

> **select** column1, column2....
>
> **from** *table_name* **as** *alias_name*
>
> **where [**condition**];**

The basic syntax of column alias is as follows −

> **select** *column_name* **as** *alias_name*
>
> **from** table_name
>
> **where [**condition**];**

Example: Consider the example 1 of the above section. The same query can be written as,

> **select** *name*, *dept_name* **as** *DepartmentName*, *building*
>
> **from** *instructor* **as** *ins*, *department* **as** *dept*
>
> **where** *ins*.*dept_name* = *dept*.*dept_name*

String Operations:

1. **String concatenation:** PostgreSQL allows you to directly concatenate strings, columns and int values using ‘ || ’ operator. Here is the SQL query to concatenate columns first_name and last_name using ‘ || ’ operator. You can even concatenate string with int using ‘||’ operator

   Example:

   ```
   select first_name || ' ' || last_name as customer_name
   from customer
   limit 5;
   ```

2. **Convert string to lowercase**: the LOWER function is used to convert a string, an expression, or values in a column to lowercase.

    Syntax: LOWER(string or value or expression)

    Example:

    ```
    select LOWER(title)
    from film;
    ```

3. **Convert string to uppercase:** Like the LOWER function, the UPPER function accepts a string expression or string-convertible expression and converts it to an upper case format.

    Syntax: UPPER(string_expression)

    Example:

    ```
    select UPPER(title)
    from film;
    ```

4. **Replace substring: '**Replace' function searches and replaces a substring with a new substring in a string.

    Syntax: REPLACE(source, old_text, new_text);

    where;

    - ***source*** is a string where you want to replace.
    - ***old_text*** is the text that you want to search and replace. If the *old_text* appears multiple times in the string, all of its occurrences will be replaced.
    - ***new_text*** is the new text that will replace the old text (*old_text*).

    Example:

    1.
       ```
       UPDATE table_name
       SET   column_name = REPLACE(column,old_text,new_text)
       WHERE condition
       ```

    2.
       ```
       select REPLACE('ABC AA', 'A', 'Z'); // output: ZBC ZZ
       ```

5. **TRIM Function:-** The PostgreSQL trim function is used to remove spaces (if no string is specified) or set of characters from the leading or trailing or both sides of a string.

   Syntax:

   trim([leading|trailing|both] <removing_string> from <main_string>)

   Parameters:

Parameters	Description
leading | trailing | both	The position of the main_string from where the removing_string will be removed.
removing_string	String which will be removed. It is optional.
main_string	The main string.

Eg:-

```
SELECT trim(from '   Gol D. Roger  ');
 // output:- ‘Gol D. Roger’
```

The above query will remove both trailing and leading whitespaces, we can also write the keyword BOTH as so:-

```
SELECT trim(BOTH from ‘    Shigeo Kageyama        ‘);
// output:- ‘Shigeo Kageyama’
```

Eg:- one can specify a string that needs to be removed from both the beginning or the end, as so:-

```
SELECT trim(LEADING ‘Admiral ’ from ‘Admiral Sakazuki’);
//output:- ‘Sakazuki’
//we can use trailing to remove string from the end
```

The above query will remove the string ‘Admiral ‘ from the beginning of the string.

Ordering

```
SELECT
   select_list
FROM
   table_name
ORDER BY
   sort_expression1 [ASC | DESC],
        ...
   sort_expressionN [ASC | DESC];
```

When you query data from a table, the **SELECT** statement returns rows in an unspecified order. To sort the rows of the result set, you use the ORDER BY clause.

- First, specify a sort expression, which can be a column or an expression, that you want to sort after the ORDER BY keywords. If you want to sort the result set based on multiple columns or expressions, you need to place a comma (,) between two columns or expressions to separate them.
- Second, you use the ASC option to sort rows in ascending order and the DESC option to sort rows in descending order. If you omit the ASC or DESC option, the ORDER BY uses ASC by default.

Eg:- The below query

```
select ID, name, dept_name, salary * 1.1 as new_salary

from instructor

Order by new_salary DESC, name ASC;
```

The result of the above query, the results will be first sorted by the new_salary in the descending order, for those rows where new_salary is the same, they will be sorted by the instructor name in ascending order.

Set Operations

The SQL operations union, intersect, and except operate on relations and correspond to the mathematical set-theory operations ∪, ∩, and −.

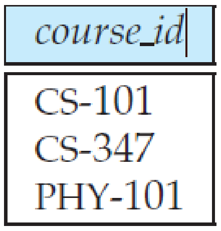
Eg:- The set of all courses taught in the Fall 2009 semester:

```
select course_id
from section
where semester = 'Fall' and year= 2009;
```

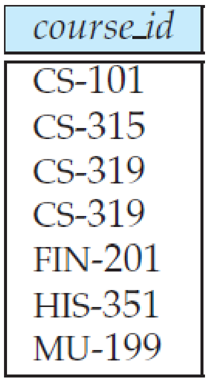
The set of all courses taught in the Spring 2010 semester:

```
select course_id
from section
where semester = 'Spring' and year= 2010;
```

In our discussion that follows,we shall refer to the relations obtained as the result of the preceding queries as c1 and c2, respectively.



course_id
CS-101
CS-347
PHY-101



course_id
CS-101
CS-315
CS-319
CS-319
FIN-201
HIS-351
MU-199

Output of c1, and c2 respectively.

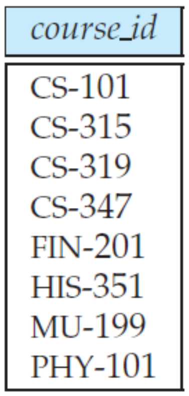
The Union Operation

To find the set of all courses taught either in Fall 2009 or in Spring 2010, or both,

we write:

```
(select course_id
from section
where semester = ’Fall’ and year= 2009)
union
(select course_id
from section
where semester = ’Spring’ and year= 2010);
```

Output of above query:-



course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

The union operation automatically eliminates duplicates. If we want to retain all duplicates, we must write union all in place of union:

```
(select course_id
from section
where semester = ’Fall’ and year= 2009)
union all
(select course_id
from section
where semester = ’Spring’ and year= 2010);
```

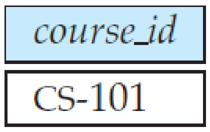
The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both c1 and c2. 3.5.2

<u>The Intersect Operation</u>

To find the set of all courses taught in the Fall 2009 as well as in Spring 2010 we write:

```
(select course_id
from section
where semester = ’Fall’ and year= 2009)
intersect
(select course_id
from section
where semester = ’Spring’ and year= 2010);
```

<u>Output of above query:-</u>



course_id
CS-101

The intersect operation automatically eliminates duplicates. If we want to retain all duplicates, we must write intersect all in place of intersect:

```
(select course_id
from section
where semester = ’Fall’ and year= 2009)
intersect all
(select course_id
from section
where semester = ’Spring’ and year= 2010);
```

The number of duplicate tuples that appear in the result is equal to the minimum number of duplicates in both c1 and c2.

The Except Operation

To find all courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we write:

```
(select course_id
from section
where semester = ’Fall’ and year= 2009)
except
(select course_id
from section
where semester = ’Spring’ and year= 2010);
```

Output of above query:-

course_id
CS-347
PHY-101

The except operation outputs all tuples from its first input that do not occur in the second input; that is, it performs set difference. The operation automatically eliminates duplicates in the inputs before performing set difference.If we want to retain duplicates, we must write except all in place of except:

```
(select course_id
from section
where semester = ’Fall’ and year= 2009)
```

except all

***(select** course_id*

***from** section*

where** semester = ’Spring’ **and** year= 2010**);

The number of duplicate copies of a tuple in the result is equal to the number of duplicate copies in c1 minus the number of duplicate copies in c2, provided that the difference is positive.

Restoring Database

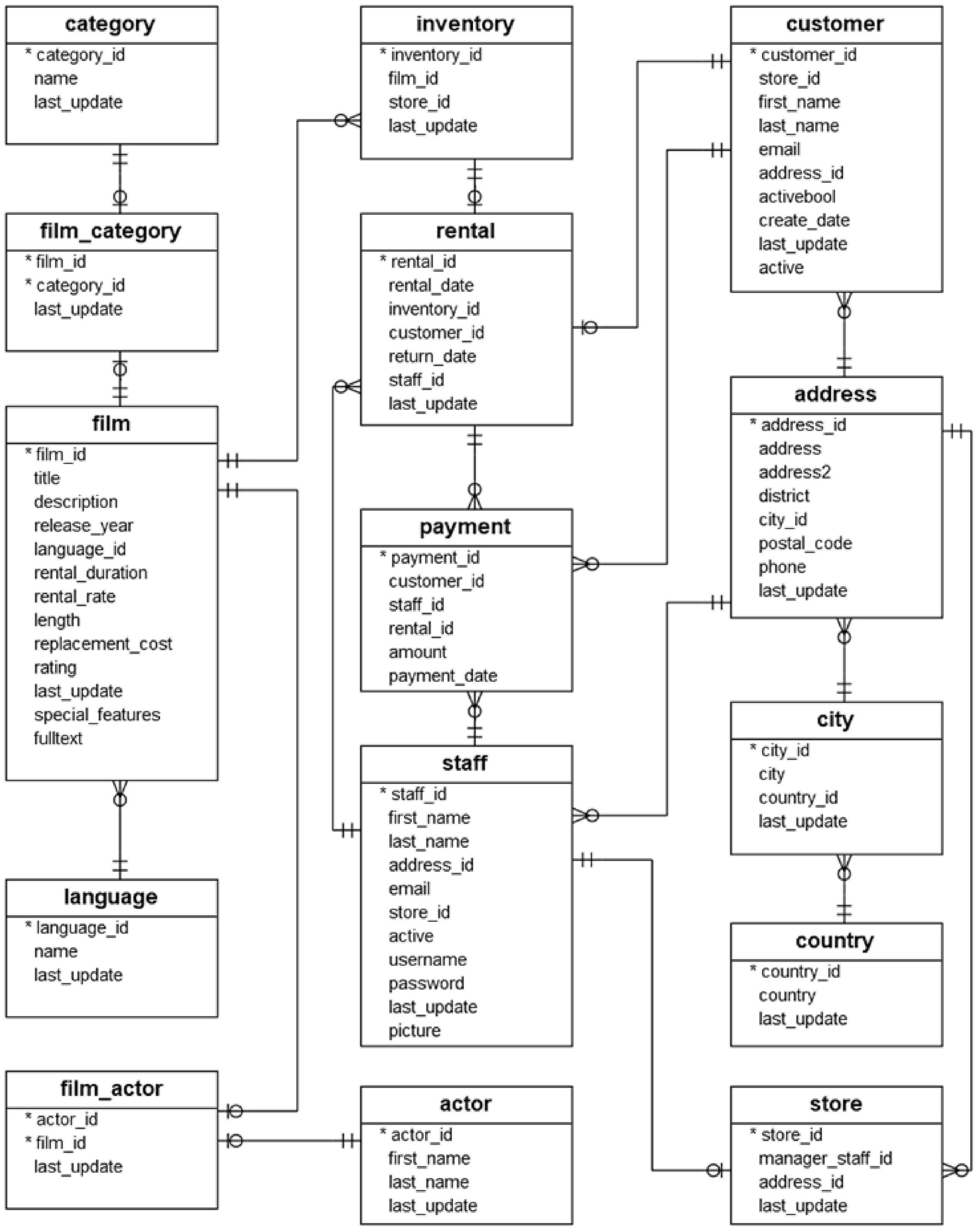
The lab from now onwards will utilize the dvdrental database. Download the database tar file from https://github.com/imkumaraju/dvdrenat-sample-databse.

CREATE DATABASE dvdrental;

Use the pg_restore tool to load data into the dvdrental database that we had just created as using the command:

pg_restore -U postgres -d dvdrental [path_to_tar_file]

The next page contains the schema of dvdrental database.


category
* category_id
name
last_update
film_category
* film_id
* category_id
last_update
film
* film_id
title
description
release_year
language_id
rental_duration
rental_rate
length
replacement_cost
rating
last_update
special_features
fulltext
language
* language_id
name
last_update
film_actor
* actor_id
* film_id
last_update
inventory
* inventory_id
film_id
store_id
last_update
rental
* rental_id
rental_date
inventory_id
customer_id
return_date
staff_id
last_update
payment
* payment_id
customer_id
staff_id
rental_id
amount
payment_date
staff
* staff_id
first_name
last_name
address_id
email
store_id
active
username
password
last_update
picture
actor
* actor_id
first_name
last_name
last_update
customer
* customer_id
store_id
first_name
last_name
email
address_id
activebool
create_date
last_update
active
address
* address_id
address
address2
district
city_id
postal_code
phone
last_update
city
* city_id
city
country_id
last_update
country
* country_id
country
last_update
store
* store_id
manager_staff_id
address_id
last_update