

Database Systems

Spring-2025 Department of

Computer Science

The Islamia University Bahawalpur

Submitted by:

Name: Syed Own Abbas Naqvi	Complete Roll number: F24BDOCS1M01263
Section: M8	Semester: 2nd
Submitted to: Muhammad Usman Ghani	Project: Relational Database

1. Introduction:

1.1 Background:

Problem

Academic data about professors, their universities, and their connections to different organizations is often stored in one large, flat table. This kind of setup causes problems like repeated data, errors, and difficulty when searching or updating information. As the amount of data grows, managing it becomes slower and more confusing.

Solution

To fix this, we built a relational database using the university_professors dataset. The data is organized into four connected tables: professors, universities, organizations, and affiliations. This design removes duplication, keeps the data accurate, and makes it easier and faster to search or update information. It's also a better solution for handling large datasets in the future

1.2 Goal:

The main objective of this project is to build a relational database system that accurately represents the relationships among university professors, their universities, and affiliated organizations. By the end of the project, the database will allow for easy insertion, retrieval, and updating of academic and organizational data, while maintaining data integrity and avoiding redundancy through normalization and relational modeling.

3 Requirements:

- The system must store professor details like first name, last name, and university.
- It must keep a list of all universities and organizations.
- Professors can be linked to many organizations (many-to-many relationship).
- See which organizations professors are affiliated with
- Users should be able to add, update, or delete records without losing data.
- The system should support join queries (e.g., get professors with their university and organizations).
- Use primary and foreign keys to connect tables and keep data correct.

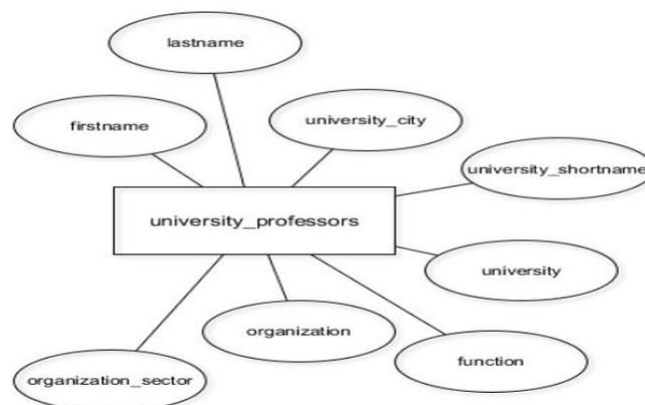
2. Functional Description:

2.1. Method of use:

This system will be used by university staff, teachers, and researchers who need to keep track of information about professors. They can use it to add, update, or view details about professors, the universities they work at, and the organizations they are connected to. This helps keep everything in one place and makes it easier to find and manage the data.

3. Entity Data Model:

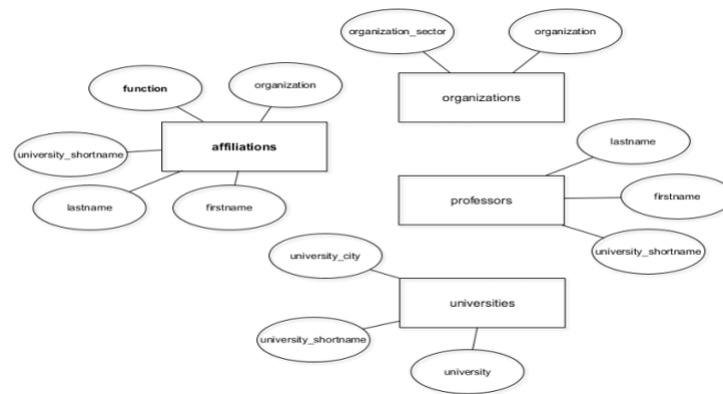
OLD



Problem – Single Table Design:

The original design used one table to store all data about professors, universities, and organizations. This caused repeated data, errors, and made searching or updating information harder

NEW



Solution – Separated Tables Design:

The data was split into four linked tables: professors, universities, organizations, and affiliations. This setup removes duplication, keeps data accurate, and makes the system easier to use and scale.

4. Table Design (Schema) Screenshots:

First, we created a table called university_professors in pgAdmin and defined its columns. Then, we imported data into this table from a CSV file. After importing, we normalized the data by creating four separate tables: professors, universities, organizations, and affiliations.

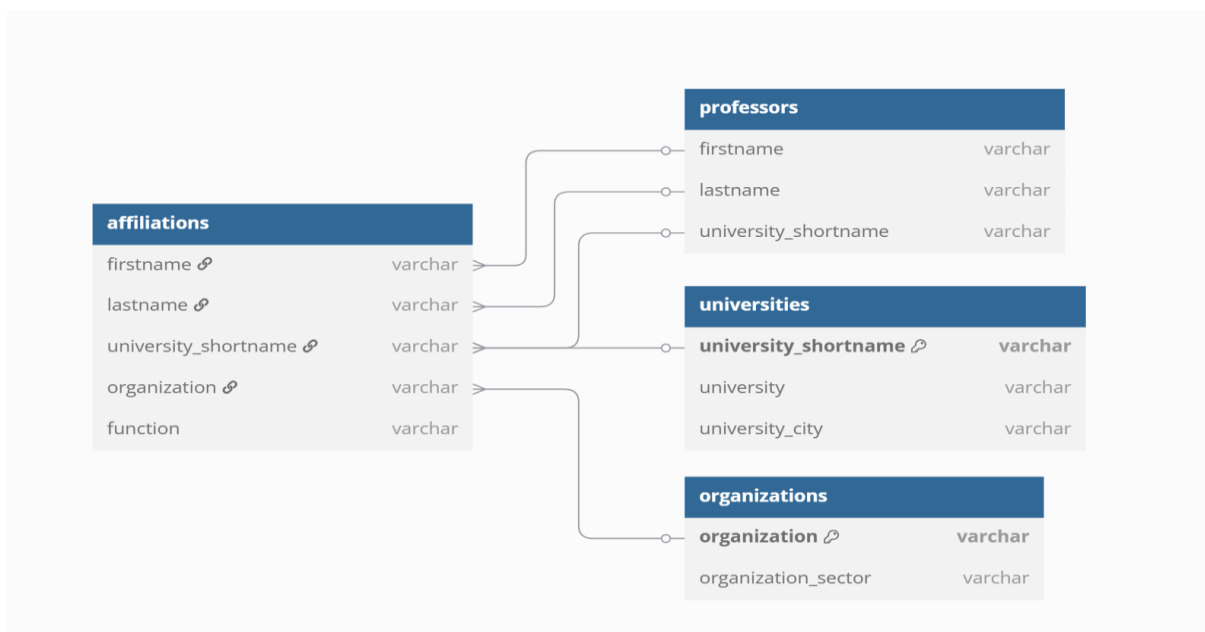


Table: university professors

university_professors

General Columns Advanced Constraints Partitions Parameters Security SQL

Inherited from table(s) Select to inherit from...

Columns							
	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
	firstname	text			<input type="checkbox"/>	<input type="checkbox"/>	
	lastname	text			<input type="checkbox"/>	<input type="checkbox"/>	
	university	text			<input type="checkbox"/>	<input type="checkbox"/>	
	university_city	text			<input type="checkbox"/>	<input type="checkbox"/>	
	function	text			<input type="checkbox"/>	<input type="checkbox"/>	
	organization	text			<input type="checkbox"/>	<input type="checkbox"/>	
	organization_ser	text			<input type="checkbox"/>	<input type="checkbox"/>	

Close Reset Save

Now lets take a look at the current table we have

Welcome First/postgres@PostgreSQL 16*

First/postgres@PostgreSQL 16

Query Query History

```
1 SELECT *
2 FROM university_professors;
```

Data Output Messages Notifications

Showing rows: 1 to 1000 Page No: 1 of 2

	firstname text	lastname text	university text	university_city text	function text
1	Karl	Aberer	ETH Lausa...	Lausanne	Chairman of L3S Advisory Board
2	Karl	Aberer	ETH Lausa...	Lausanne	Member Conseil of Zeno-Karl Schindler Foundation
3	Karl	Aberer	ETH Lausa...	Lausanne	Member of Conseil Fondation IDIAP
4	Karl	Aberer	ETH Lausa...	Lausanne	Panel Member

Total rows: 1377 Query complete 00:00:00.458 CRLF Ln 2, Col 28

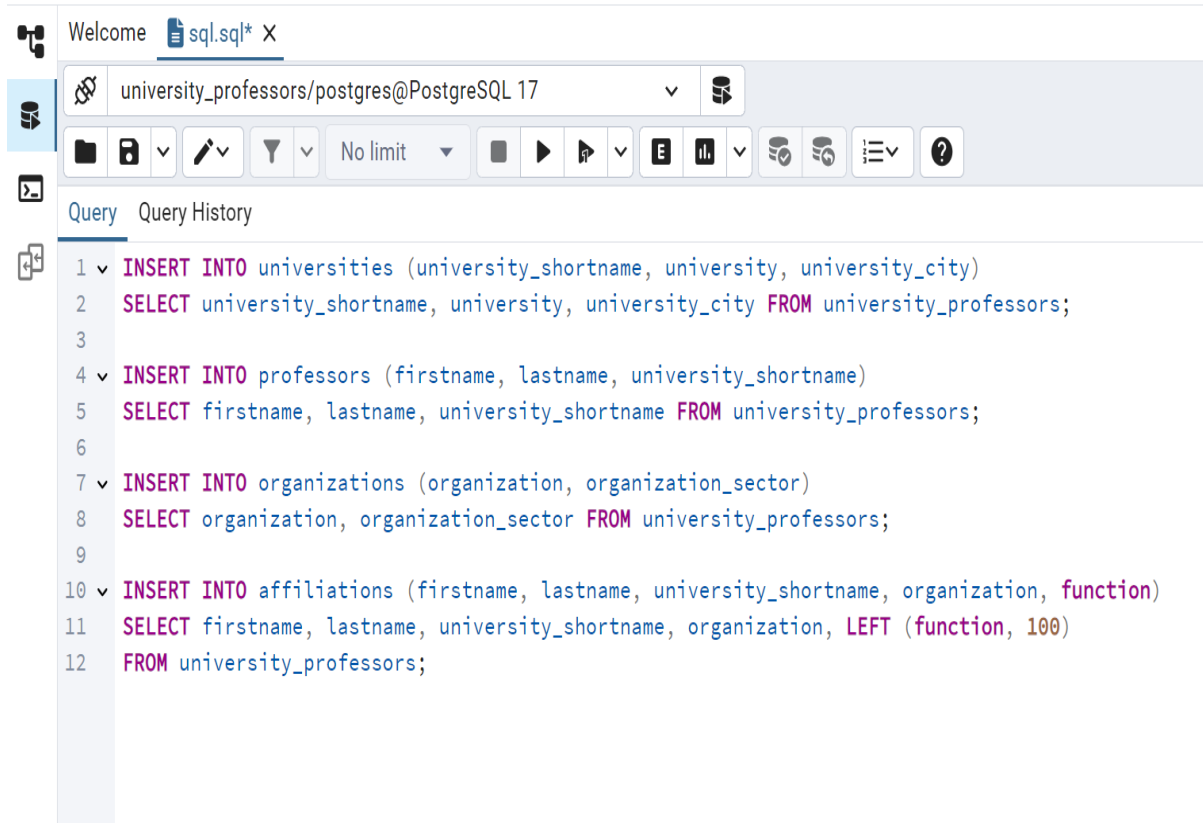
The data is organized into four connected tables: professors, universities, organizations, and affiliations.

```
Query  Query History
1  CREATE TABLE universities (
2      university_shortcode VARCHAR(50),
3      university VARCHAR(255),
4      university_city VARCHAR(100)
5  );
6  CREATE TABLE professors (
7      firstname VARCHAR(100),
8      lastname VARCHAR(100),
9      university_shortcode VARCHAR(50)
10 );
11 CREATE TABLE organizations (
12     organization VARCHAR(255),
13     organization_sector VARCHAR(100)
14 );
15 CREATE TABLE affiliations (
16     firstname VARCHAR(100),
17     lastname VARCHAR(100),
18     university_shortcode VARCHAR(50),
19     organization VARCHAR(255),
20     function VARCHAR(255)
21 );
```

And insert data into four connected tables: professors, universities, organizations, and affiliations

pgAdmin 4

File Object Tools Edit View Window Help



The screenshot shows the pgAdmin 4 interface. At the top, there's a menu bar with 'File', 'Object', 'Tools', 'Edit', 'View', 'Window', and 'Help'. Below the menu bar is a toolbar with various icons for file operations, query execution, and window management. The main window displays a SQL query in a text editor. The query consists of four INSERT statements, each followed by a SELECT statement to populate the table. The tables being populated are universities, professors, organizations, and affiliations. The affiliations table has an additional 'function' column. The query is as follows:

```
1  INSERT INTO universities (university_shortcode, university, university_city)
2  SELECT university_shortcode, university, university_city FROM university_professors;
3
4  INSERT INTO professors (firstname, lastname, university_shortcode)
5  SELECT firstname, lastname, university_shortcode FROM university_professors;
6
7  INSERT INTO organizations (organization, organization_sector)
8  SELECT organization, organization_sector FROM university_professors;
9
10 INSERT INTO affiliations (firstname, lastname, university_shortcode, organization, function)
11 SELECT firstname, lastname, university_shortcode, organization, LEFT (function, 100)
12 FROM university_professors;
```

5. Frontend Screenshots

Here a professor is uniquely identified by firstname and lastname columns, so there is no need of university_shortcode column in the affiliations table.

The screenshot shows a database query interface with a toolbar at the top. Below the toolbar, there are tabs for 'Query' and 'Query History'. The 'Query' tab is active, displaying two lines of SQL code: `1 ALTER TABLE affiliations` and `2 DROP COLUMN university_shortcode`. Below the query editor, there are tabs for 'Data Output', 'Messages', and 'Notifications'. The 'Messages' tab is active, showing the text 'ALTER TABLE' and a status message: 'Query returned successfully in 100 msec.'

We updated the professors table by setting the firstname and lastname columns to NOT NULL. Then we added an id column with serial to auto-generate unique numbers for each professor and set it as the primary key to uniquely identify each record.

The screenshot shows a database query interface with a toolbar at the top. Below the toolbar, there are tabs for 'Query' and 'Query History'. The 'Query' tab is active, displaying a multi-line SQL query: `1 ALTER TABLE professors`, `2 ALTER COLUMN firstname SET NOT NULL;`, `3`, `4 ALTER TABLE professors`, `5 ALTER COLUMN lastname SET NOT NULL;`, `6`, `7 ALTER TABLE professors`, `8 ADD COLUMN id serial;`, `9`, `10`, `11 ALTER TABLE professors`, and `12 ADD CONSTRAINT professors_pkey PRIMARY KEY (id);`. Below the query editor, there are tabs for 'Data Output', 'Messages', and 'Notifications'. The 'Messages' tab is active, showing the text 'ALTER TABLE' and a status message: 'Query returned successfully in 100 msec.'

This query first renames the column `university_shortcode` to `university_id` in the `professors` table to better reflect its purpose. Then, it adds a foreign key constraint to ensure that the `university_id` in the `professors` table must match an existing `id` in the `universities` table. This helps maintain a valid link between professors and their universities.

The screenshot shows a database query interface with a toolbar at the top. Below the toolbar, there are tabs for 'Query' and 'Query History'. The 'Query' tab is active, displaying a multi-line SQL query: `1 ALTER TABLE professors`, `2 RENAME COLUMN university_shortcode TO university_id;`, `3`, `4`, `5 ALTER TABLE professors`, and `6 ADD CONSTRAINT professors_fkey FOREIGN KEY (university_id) REFERENCES universities (id);`. Below the query editor, there are tabs for 'Data Output', 'Messages', and 'Notifications'. The 'Messages' tab is active, showing the text 'ALTER TABLE' and a status message: 'Query returned successfully in 100 msec.'

Here is professors table

Columns								+
	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default	
	firstname	character varying v	16		<input checked="" type="checkbox"/>	<input type="checkbox"/>		
	lastname	character varying v	100		<input checked="" type="checkbox"/>	<input type="checkbox"/>		
	university_id	character varying v	50		<input type="checkbox"/>	<input type="checkbox"/>		
	id	integer v			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	nextval('i	

We renamed the column to id to make the table more consistent and easier to join with other tables. Adding a PRIMARY KEY ensures that each university can be uniquely identified, which improves data integrity and allows for better relationships between tables.

Query Query History

```

1  ALTER TABLE universities
2  RENAME COLUMN university_shortcode TO id;
3
4
5  ALTER TABLE universities
6  ADD CONSTRAINT university_pk PRIMARY KEY (id);

```

We added a PRIMARY KEY constraint on the id column in the universities table to uniquely identify each row. Then, we deleted duplicate rows by keeping only one row per id using the ctid system column to remove duplicates and ensure data consistency.

Query Query History

```

1  ALTER TABLE universities
2  ADD CONSTRAINT universities_pkey PRIMARY KEY (id);
3  DELETE FROM universities
4  WHERE ctid NOT IN (
5      SELECT MIN(ctid)
6      FROM universities
7      GROUP BY id
8  );

```

This SQL command adds a foreign key constraint to the professors table. It links the university_id column in professors to the id column in the universities table. This ensures that every university_id in professors must exist in universities, helping maintain referential integrity between the two tables.







Query Query History

```

1  ALTER TABLE professors
2  ADD CONSTRAINT f_key FOREIGN KEY (university_id) REFERENCES universities(id);
3
4
5
6

```

Here is universities table

Columns								+
	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default	
 	id	character varying v	50		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
 	university	character varying v	255		<input type="checkbox"/>	<input type="checkbox"/>		
 	university_city	character varying v	100		<input type="checkbox"/>	<input type="checkbox"/>		

Here we add some query:

This query finds the last names of professors who work at universities located in the city of Zurich. It joins the professors and universities tables using the university's unique ID to match professors with their universities. The WHERE clause filters the results to only show professors from universities in Zurich. This helps get specific information about professors based on the location of their university.

Query Query History

```

1  SELECT professors.lastname, universities.id, universities.university_city
2  FROM universities
3  JOIN professors
4  ON professors.university_id = universities.id
5  WHERE universities.university_city = 'Zurich';

```

OUTPUT

Data Output Messages Notifications

Showing rows: 1 to 140			
	lastname character varying (100)	id character varying (50)	university_city character varying (100)
1	Abhari	ETH	Zurich
2	Axhausen	ETH	Zurich
3	Baschera	ETH	Zurich
4	Baschera	ETH	Zurich
5	Baschera	ETH	Zurich
6	Baschera	ETH	Zurich
7	Baschera	ETH	Zurich
8	Baschera	ETH	Zurich
9	Basin	ETH	Zurich
10	Bechtold	ETH	Zurich

We renamed the column to id to follow a standard naming convention and added a **PRIMARY KEY**, so each organization has a unique identifier for better data management

and relationships with other tables.





```
Query Query History
1  ALTER TABLE organizations
2  RENAME COLUMN organization TO id;
3
4  ALTER TABLE organizations
5  ADD CONSTRAINT organization_pk PRIMARY KEY (id);
```

This SQL script is used to clean up duplicate records in the organizations table and ensure data integrity going forward. It first identifies and deletes duplicate rows by keeping only one entry for each unique organization using the ctid (a unique row identifier in PostgreSQL).

After the duplicates are removed, it adds a UNIQUE constraint on the organization column to prevent any future duplicate entries from being inserted into the table. This process helps maintain consistent and accurate data in the database.

```
12
13  ALTER TABLE organizations
14  ADD CONSTRAINT organization_unq UNIQUE(organization);
15
16  SELECT organization, COUNT(*)
17  FROM organizations
18  GROUP BY organization
19  HAVING COUNT(*) > 1;
20
21  DELETE FROM organizations
22  WHERE ctid NOT IN (
23    SELECT MIN(ctid)
24    FROM organizations
25    GROUP BY organization
26  );
```

Here is organizations table

Columns								+
	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default	
 	id	character varying v	255		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
 	organization_ser	character varying v	100		<input type="checkbox"/>	<input type="checkbox"/>		

We DROP columns from affiliation table

```
Query Query History
1  ALTER TABLE affiliations
2  DROP COLUMN firstname,
3  DROP COLUMN lastname;
```

You added a new column professor_id to the affiliations table to link each affiliation to a specific professor using their unique ID. You renamed the organization column to organization_id to clearly show it stores an ID referencing the organizations table. Then, you




added foreign key constraints to both professor_id and organization_id to make sure these IDs match valid records in their respective tables. Finally, you updated the professor_id column by matching professors' first and last names so that existing affiliation records correctly point to the right professor. These changes improve data accuracy and make it easier to manage relationships between professors, organizations, and their affiliations.

```
Query Query History
1 ALTER TABLE affiliations
2 ADD COLUMN professor_id integer REFERENCES professors (id);
3
4 ALTER TABLE affiliations
5 RENAME organization TO organization_id;
6
7 ALTER TABLE affiliations
8 ADD CONSTRAINT affiliations_organization_fkey FOREIGN KEY (organization_id) REFERENCES organizations (id);
9
10
11 UPDATE affiliations
12 SET professor_id = professors.id
13 FROM professors
14 WHERE affiliations.firstname = professors.firstname AND affiliations.lastname = professors.lastname;
```

First, you check all foreign key constraints in the database to find the exact name of the constraint on the affiliations table. Then, you drop the current foreign key constraint on organization_id so you can modify it. After that, you add the foreign key constraint again but this time with ON DELETE CASCADE, which means if a row in the organizations table is deleted, all related rows in affiliations will also be automatically deleted to keep the data consistent.

```
Query Query History
1 SELECT constraint_name, table_name, constraint_type
2 FROM information_schema.table_constraints
3 WHERE constraint_type = 'FOREIGN KEY';
4
5
6 ALTER TABLE affiliations
7 DROP CONSTRAINT affiliations_organization_id_fkey;
8
9
10 ALTER TABLE affiliations
11 ADD CONSTRAINT affiliations_organization_id_fkey FOREIGN KEY (organization_id) REFERENCES organizations (id) ON DELETE CASCADE;
12
13
```

Here is affiliation table










Columns								+
	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default	
	organization_id	character varying v	255		<input type="checkbox"/>	<input type="checkbox"/>		
	function	character varying v	100		<input type="checkbox"/>	<input type="checkbox"/>		
	professor_id	integer v			<input type="checkbox"/>	<input type="checkbox"/>		

Here we add some query

Query Query History

```
1 SELECT COUNT(*), professors.university_id
2 FROM affiliations
3 JOIN professors
4 ON affiliations.professor_id = professors.id
5 GROUP BY professors.university_id
6 ORDER BY count DESC;
```

OUTPUT

Data Output Messages Notifications		
         SQL		
	count bigint	university_id character varying (50)
1	572	EPF
2	273	USG
3	162	UBE
4	128	ETH
5	75	UBA
6	40	UFR
7	36	UNE
8	35	ULA
9	33	UGE
10	7	UZH
11	4	USI

Here we add some query

```
Query Query History
1  SELECT COUNT(*), organizations.organization_sector,
2  professors.id, universities.university_city
3  FROM affiliations
4  JOIN professors
5  ON affiliations.professor_id = professors.id
6  JOIN organizations
7  ON affiliations.organization_id = organizations.id
8  JOIN universities
9  ON professors.university_id = universities.id
10 WHERE organizations.organization_sector = 'Media & communication'
11 GROUP BY organizations.organization_sector,
12 professors.id, universities.university_city
13 ORDER BY count DESC;
```

OUTPUT

	count bigint	organization_sector character varying (100)	id integer	university_city character varying (100)
1	4	Media & communication	1357	Lausanne
2	3	Media & communication	95	Lausanne
3	3	Media & communication	928	Saint Gallen
4	2	Media & communication	1059	Lausanne
5	2	Media & communication	861	Saint Gallen
6	2	Media & communication	1316	Saint Gallen
7	2	Media & communication	828	Zurich
8	2	Media & communication	106	Saint Gallen
9	2	Media & communication	1267	Lausanne
10	1	Media & communication	76	Lausanne
11	1	Media & communication	975	Lausanne
12	1	Media & communication	949	Lausanne
13	1	Media & communication	745	Basel

Here we add some query

```
Query Query History
1  SELECT COUNT(*), professors.university_id
2  FROM affiliations
3  JOIN professors
4  ON affiliations.professor_id = professors.id
5  GROUP BY professors.university_id
6  ORDER BY count DESC;
```

OUTPUT

Data Output

Messages

Notifications

SQL

Showing rows: 1 to

	count bigint	university_id character varying (50)
1	572	EPF
2	273	USG
3	162	UBE
4	128	ETH
5	75	UBA
6	40	UFR
7	36	UNE
8	35	ULA
9	33	UGE
10	7	UZH
11	4	USI

Referential Integrity:

The process of making a relationship between tables by using **FOREIGN KEY** which is the **PRIMARY KEY** of any table is known as referential integrity

Why?

It will allow us to maintain data between related tables i-e we can't **ADD** or **DELETE** rows in the table containing **FOREIGN KEY** unless we don't **DELETE** or **UPDATE** the **PRIMARY KEY** of the table

For this purpose, we use the format:

- **ALTER TABLE** b_table
- **ADD CONSTRAINT** constraint_name **FOREIGN KEY** (b_table_column) **REFERENCES** a_table (a_table_column) **ON DELETE CASCADE**

So we can only use this format during the formation of **FOREIGN KEY** so that we are required to **DROP** previous **FOREIGN CONSTRAINTS** and add new ones.

At last, the ER diagram of our database

