Reference

This is a DataCamp course

First database

- See sketches of tables used in this note.
- The database system used is PostgreSQL.

Attributes of relational databases

- store different real-world entities in different tables.
- allow to establish relationships between entities.
- use constraints, keys and referential integrity in order to assure data quality.

Information_schema

This is a side note about information_schema:

In MySQL

- What is the information_schema database? In MySQL version 5.02 and above there is an additional database called information_schema. This database is available inside the cPanel -> phpMyAdmin tool and provides access to the database metadata for the corresponding user.

information_schema is the database where the information about all the other databases is kept, for example names of a database or a table, the data type of columns, access privileges, etc. It is a built-in virtual database with the sole purpose of providing information about the database system itself. The MySQL server automatically populates the tables in the information_schema.

The important thing to remember about the information_schema database is that you can query it, but you cannot change its structure or modify its data.

In SQL Server

Access as this: -> database name -> security -> schema -> information_schema
An information schema view is one of several methods SQL Server provides for obtaining metadata. Information schema views provide an internal, system table-independent view of the SQL Server metadata. Information schema views enable applications to work correctly although significant changes have been made to the underlying system tables. The information schema views included in SQL Server comply with the ISO standard definition for the INFORMATION_SCHEMA.

In PostgreSQL

Access as this: -> database name -> Catalog -> ....

Extracting META information from a PostgreSQL database using the **INFORMATION_SCHEMA views** and the system catalogs (pg_class, pg_user, pg_view)....

Query information_schema with SELECT

`information_schema` is a meta-database that holds information about your current database. information_schema has multiple tables you can query with the known SELECT * FROM syntax:

tables: information about all tables in your current database
columns: information about all columns in all of the tables in your current database
... In this exercise, we only need information from the 'public' schema, which is specified as the column table_schema of the tables and columns tables. The 'public' schema holds information about user-defined tables and databases. The other types of table_schema hold system information – for this course, we're only interested in user-defined stuff.

Get information on all table names in the current database, while limiting your query to the 'public' table_schema.
Note here we skip the process how the database was created.

```
SELECT table_name
FROM information_schema.tables
WHERE table_schema = 'public';
```

```
table_name
university_professors
```

```
-- Query the right table in information_schema to get columns
SELECT column_name, data_type
FROM information_schema.columns
WHERE table_name = 'university_professors' AND table_schema = 'public';
```

```
column_name data_type
firstname text
lastname text
university text
```

CREATE your first few TABLEs

Now start implementing a better database model. To do this, create tables for the professors and universities entity types. The other tables are already created.

Create a table professors with two text columns: firstname and lastname.

```
CREATE TABLE professors (
firstname text,
lastname text
);
```

Create a table universities with three text columns: university_shortname, university, and university_city.

```
CREATE TABLE universities (
university_shortname text,
university text,
university_city text
);
```

ADD a COLUMN with ALTER TABLE

Alter professors to add the text column university_shortname. In chapter 4 of this course, we need this column for connecting the professors table with the universities table.

```
ALTER TABLE professors
ADD COLUMN university_shortname text;
```

RENAME and DROP COLUMNs in affiliations

The still empty affiliations table has some flaws. We will do some changes below.

```
ALTER TABLE affiliations
RENAME COLUMN organisation TO organization;
```

```
ALTER TABLE affiliations
DROP COLUMN university_shortname;
```

Migrate data with INSERT INTO SELECT DISTINCT

```
-- Insert unique professors into the new table
INSERT INTO professors
SELECT DISTINCT firstname, lastname, university_shortname
FROM university_professors;
```

```
-- Insert unique affiliations into the new table
INSERT INTO affiliations
SELECT DISTINCT firstname, lastname, function, organization
FROM university_professors;
```

```
 firstname    lastname    function    organization
Dimos    Poulikakos    VR-Mandat    Scrona AG
Francesco    Stellacci    Co-editor in Chief, Nanoscale    Royal Chemistry Society, UK
Alexander    Fust    Fachexperte und Coach für Designer Startups    Creative Hub
Jürgen    Brugger    Proposal reviewing HEPIA    HES Campus Biotech, Genève
Hervé    Bourlard    Director    Idiap Research Institute
Ioannis    Papadopoulos    Mandat    Schweizerischer Nationalfonds (SNF)
Olaf    Blanke    Professeur à 20%    Université de Genève
```

There are 1377 distinct combinations of professors and organisations in the dataset. We'll migrate the other two tables universities and organizations. The last thing to do in this chapter is to delete the no longer needed university_professors table. **In the section above, we basically have created some new tables from a base table already exited in the database.**

Delete tables with DROP TABLE

Obviously, the university_professors table is now no longer needed and can safely be deleted.

```
DROP TABLE university_professors;
```

Enforce data consistency with attribute constraints

Types of database constraints

The following are used to enforce a database constraint.

- Foreign keys
- The data types
- Primary keys

Conforming with data types

For demonstration purposes, I created a fictional database table that only holds three records. The columns have the data types date, integer, and text, respectively.

```
CREATE TABLE transactions (
transaction_date date,
amount integer,
fee text
);
```

The transaction_date accepts date values. According to the PostgreSQL documentation, it accepts values in the form of YYYY-MM-DD, DD/MM/YY, and so forth.

Table: transactions

```
 transaction_date    amount    fee
1999-01-08    500    20
2001-02-20    403    15
2001-03-20    3430    35
```

```
INSERT INTO transactions (transaction_date, amount, fee)
VALUES ('2018-09-24', 5454, '30');
```

```
 transaction_date    amount    fee
1999-01-08    500    20
2001-02-20    403    15
2001-03-20    3430    35
2018-09-24    5454    30
```

Data types provide certain restrictions on how data can be entered into a table. This may be tedious at the moment of insertion, but saves a lot of headache in the long run.

Type CASTs

Type casts are a possible solution for data type issues. If you know that a certain column stores numbers as text, you can cast the column to a numeric form, i.e. to integer.

```
SELECT CAST(some_column AS integer)
FROM table;
```

Now, the some_column column is **temporarily represented as integer** instead of text, meaning that you can perform numeric calculations on the column.

```
SELECT transaction_date, amount + CAST(fee AS integer) AS net_amount
FROM transactions;
```

```
 transaction_date    net_amount
1999-01-08    520
```

```
2001-02-20    418
2001-03-20    3465
1999-01-08    520
2001-02-20    418
2001-03-20    3465
2018-09-24    5484
```

Sometimes, type casts are necessary to work with data. However, it is better to store columns in the right data type from the first place.

Change types with ALTER COLUMN

```
ALTER TABLE professors
ALTER COLUMN university_shortname
TYPE char(3);
```

```
ALTER TABLE professors
ALTER COLUMN firstname
TYPE varchar(64);
```

Convert types USING a function

```
ALTER TABLE table_name
ALTER COLUMN column_name
TYPE varchar(x)
USING SUBSTRING(column_name FROM 1 FOR x)
```

You should read it like this: Because you want to reserve only x characters for column_name, you have to retain a SUBSTRING of every value, i.e. the first x characters of it, and throw away the rest. This way, the values will fit the varchar(x) requirement.

```
-- Convert the values in firstname to a max. of 16 characters
ALTER TABLE professors
ALTER COLUMN firstname
TYPE varchar(16)
USING SUBSTRING(firstname FROM 1 FOR 16);
```

However, it's best not to truncate any values in your database, so we'll revert this column to varchar(64).

Disallow NULL values with SET NOT NULL

```
ALTER TABLE professors
ALTER COLUMN firstname SET NOT NULL;
```

```
ALTER TABLE professors
ALTER COLUMN lastname SET NOT NULL;
```

What happens if you try to enter NULLs?

```
INSERT INTO professors (firstname, lastname, university_shortname)
VALUES (NULL, 'Miller', 'ETH');
```
Why does this throw an error?

Because a database constraint is violated.

Make your columns UNIQUE when creating tables or ADD CONSTRAINT later

Add the UNIQUE keyword after the column_name that should be unique. This, of course, **only works for new tables:**

```
CREATE TABLE table_name (
column_name UNIQUE
);
```

If you want to add a unique constraint to an existing table (**how about the existing non-unique columns of data?**, you do it like that:

```
ALTER TABLE table_name
ADD CONSTRAINT some_name UNIQUE(column_name);
```

We have to give the constraint a name some_name. **This might be because we have to add a non-clustered index structure which needs a name.**

```
ALTER TABLE universities
```

```
ADD CONSTRAINT university_shortname_unq UNIQUE(university_shortname);

ALTER TABLE organizations
ADD CONSTRAINT organization_unq UNIQUE(organization);
```

From other notes we know: When you create a table with a UNIQUE constraint, Database Engine automatically creates a **non-clustered index**.

Uniquely identify records with key constraints

It's time to add so-called **primary and foreign keys** to the tables. They are one of the most important concepts in databases – and will be the **building blocks** of relationships between tables. **Comments: see other notes. key constraints are actually special cases of a more general concept: index. Keys are actually policed by index. For example, most sql implementations has clustered unique index built on primary key. Because primary key can have only one, we often need index (can be many) to optimize sql handling such as searching or joining.**

Below are some comments about primary and foreign keys:

- Whenever we have keys, they are usually related to hashing. For example, the key value pair, keys in dictionary, etc., in python. Sometimes 'index' should also be related to hashing. However, in SQL, primary keys or index are usually implemented with B/B+ trees, although sometimes also implemented by hashing.
- Primary key, or index, sometimes are formed by multiple columns.
- AJ's example on the use of foreign key:
  - A student table is with stdId as primary key. A course table is with cId as primary key.
  - A course selection table has its own primary key, but also has stdId, cId as its foreign keys.
  - We cannot delete a record in student table with a stdId = 5 if this id is also existing in course selection table. Something refer to this id (reference counting is not zero), and thus we cannot delete it.
  - We cannot add a record in course selection table with a stdId that is not existed in the student table due to the foreign key constraints.
- Revisit AJ's exampel:
  - Previously we only consider the case where a parent cannot delete its records when child reference it with foreign key. This is called ON DELETE NO ACTION (see example in the end of this note).
  - Another valid option is parent can delete the record as long as the system also delete the same record in the child. This is valid because it does not violate the rule of reference counting: A record is deleted when there are no references pointing to it.

Get to know SELECT COUNT DISTINCT

There's a simple way of finding out whether a certain column (or a combination) contains only unique values – and thus identifies the records in the table.

```
SELECT COUNT(*)
FROM universities;
```

```
SELECT COUNT(DISTINCT(university_city))
FROM universities;
```

Obviously, the university_city column wouldn't lend itself as a key. Because there are only 9 distinct values, but the table has 11 rows.

Identify keys with SELECT COUNT DISTINCT

There's a very basic way of finding out what qualifies for a key in an existing, populated table:

Count the distinct records for all possible combinations of columns. If the resulting number x equals the number of all rows in the table for a combination, you have discovered a **superkey**.

Then remove one column after another until you can no longer remove columns without seeing the number x decrease. If that is the case, you have discovered a (candidate) key.

The table professors has 551 rows. It has only one possible candidate key, which is a combination of two attributes.

```
SELECT COUNT(DISTINCT(firstname, lastname))
FROM professors;
```

```
count
551
```

indeed, the only combination that uniquely identifies professors is {firstname, lastname}. {firstname, lastname, university_shortname} is a superkey, and all other combinations give duplicate values. Hopefully, the concept of superkeys and keys is now a bit more clear. Let's move on to primary keys!

Identify the primary key

You have to make a wise choice as to which column should be the primary key.

```
 license_no         | serial_no |    make    |  model  | year
--------------------+-----------+------------+---------+------
 Texas ABC-739      | A69352    | Ford       | Mustang |    2
 Florida TVP-347    | B43696    | Oldsmobile | Cutlass |    5
 New York MPO-22    | X83554    | Oldsmobile | Delta   |    1
 California 432-TFY | C43742    | Mercedes   | 190-D   |   99
 California RSK-629 | Y82935    | Toyota     | Camry   |    4
 Texas RSK-629      | U028365   | Jaguar     | XJS     |    4
```

Which of the following column or column combinations could best serve as primary key?

A primary key consisting solely of "license_no" is probably the wisest choice, as license numbers are certainly unique across all registered cars in a country.

ADD key CONSTRAINTs to the tables

Two of the tables in your database already have well-suited candidate keys consisting of one column each: organizations and universities with the organization and university_shortname columns, respectively.

ALTER TABLE organizations
RENAME COLUMN organization TO id;

ALTER TABLE organizations
ADD CONSTRAINT organization_pk PRIMARY KEY (id);

ALTER TABLE universities
RENAME COLUMN university_shortname TO id;

ALTER TABLE universities
ADD CONSTRAINT university_pk PRIMARY KEY (id);

Let's tackle the last table that needs a primary key right now: professors. However, things are going to be different this time, because you'll add a so-called **surrogate key**.

Add a SERIAL surrogate key

Since there's no single column candidate key in professors (only a composite key candidate consisting of firstname, lastname), you'll add a new column id to that table.

This column has a **special data type serial**, which turns the column into an auto-incrementing number. This means that, whenever you add a new professor to the table, it will automatically get an id that does not exist yet in the table: a perfect primary key!

Is this similar to the range index in pandas DataFrame?

```
ALTER TABLE professors
ADD COLUMN id serial;
```

```
ALTER TABLE professors
ADD CONSTRAINT professors_pkey PRIMARY KEY (id);
```

```
SELECT *
FROM professors
LIMIT 10;
```

```
 firstname    lastname    university_shortname    id
Karl    Aberer    EPF    1
Reza Shokrollah    Abhari    ETH    2
Georges    Abou Jaoudé    EPF    3
```

PostgreSQL has automatically numbered the rows with the id column, which now functions as a (surrogate) primary key – it uniquely identifies professors.

CONCATenate columns to a surrogate key

Another strategy to add a surrogate key to an existing table is to concatenate existing columns with the CONCAT() function.

This might be sometimes more useful than the normal serial surrogate key, e.g. in searching.

```
SELECT COUNT(DISTINCT(make, model))
FROM cars;

ALTER TABLE cars
ADD COLUMN id varchar(128);

UPDATE cars
SET id = CONCAT(make, model);

ALTER TABLE cars
ADD CONSTRAINT id_pk PRIMARY KEY(id);

-- Have a look at the table
SELECT * FROM cars;
```

```
 make    model    mpg    id
Subaru    Forester    24    SubaruForester
Opel    Astra    45    OpelAstra
Opel    Vectra    40    OpelVectra
```

```
 table cars
make    model    mpg
Subaru    Forester    24
Opel    Astra    45
Opel    Vectra    40
```

Test your knowledge before advancing

Let's think of an entity type "student". A student has:

a last name consisting of up to 128 characters (this cannot contain a missing value),
a unique social security number of length 9, consisting only of integers,
a phone number of fixed length 12, consisting of numbers and characters (but some students don't have one).

Given the above description of a student entity, create a table students with the correct column types. Add a primary key for the social security number.

```
-- Create the table
CREATE TABLE students (
last_name varchar(128) NOT NULL,
ssn integer PRIMARY KEY,
phone_no char(12)
);
```

Glue together tables with foreign keys

- Foreign keys, just primary keys of other table. So it sets constraints to tables.
- Primary keys are usually unique clustered index, and thus using it to search is very efficient (tree or hash implementation).
- Also because foreign keys are index and fast for searching, it will facilitate filtering and table joining.

REFERENCE a table with a FOREIGN KEY

Pay attention to the naming convention employed here: Usually, a foreign key referencing another primary key with name id is named x_id, where x is the name of the referencing table in the singular form.

Add a foreign key on university_id column in professors that references the id column in universities. Name this foreign key professors_fkey.

```
ALTER TABLE professors
RENAME COLUMN university_shortname TO university_id;
```

```
ALTER TABLE professors
ADD CONSTRAINT professors_fkey FOREIGN KEY (university_id) REFERENCES universities (id);
```

Now, the professors table has a link to the universities table. Each professor belongs to exactly one university.

Explore foreign key constraints

See comments in the beginning of this chapter.

```
-- Try to insert a new professor
INSERT INTO professors (firstname, lastname, university_id)
VALUES ('Albert', 'Einstein', 'UZH');
```

As you can see, inserting a professor with non-existing university IDs violates the foreign key constraint you've just added. This also makes sure that all universities are spelled equally – adding to data consistency.

JOIN tables linked by a foreign key

While foreign keys and primary keys are not strictly necessary for join queries, they greatly help by telling you what to expect. For instance, you can be sure that records referenced from table A will always be present in table B – so a join from table A will always find something in table B. If not, the foreign key constraint would be violated.

```
SELECT professors.lastname, universities.id, universities.university_city
FROM professors
JOIN universities
ON professors.university_id = universities.id
WHERE universities.university_city = 'Zurich';
```

```
 lastname    id    university_city
Abhari    ETH    Zurich
Axhausen    ETH    Zurich
Baschera    ETH    Zurich
```

First, the university belonging to each professor was attached with the JOIN operation. Then, only professors having "Zurich"" as university city were retained with the WHERE clause.

Add foreign keys to the "affiliations" table

- At the moment, the affiliations table has the structure {firstname, lastname, function, organization}
- In the next three exercises, we will turn this table into the form {professor_id, organization_id, function}, with professor_id and organization_id being foreign keys that point to the respective tables.
- We're going to transform the affiliations table in-place, i.e., without creating a temporary table to cache your intermediate results.

Steps:

- Add a professor_id column with integer data type to affiliations, and declare it to be a foreign key that references the id column in professors. **Here we don't need format of constrainting foreign key in a normal way**.
- Rename the organization column in affiliations to organization_id.
- Add a foreign key constraint on organization_id so that it references the id column in organizations.

```
-- Add a professor_id column
ALTER TABLE affiliations
ADD COLUMN professor_id integer REFERENCES professors (id); --Note the REFERENCES

-- Rename the organization column to organization_id
ALTER TABLE affiliations
RENAME organization TO organization_id;

-- Add a foreign key on organization_id
ALTER TABLE affiliations
ADD CONSTRAINT affiliations_organization_fkey FOREIGN KEY (organization_id) REFERENCES organizations (id);
```

Populate the "professor_id" column

Populate professors_id of table affiliations by taking the ID directly from professors.

```
UPDATE affiliations
SET professor_id = professors.id
FROM professors
WHERE affiliations.firstname = professors.firstname AND affiliations.lastname = professors.lastname;
```

Drop "firstname" and "lastname"

The firstname and lastname columns of affiliations were used to establish a link to the professors table in the last exercise – so the appropriate professor IDs could be copied over. This **only worked because there is exactly one corresponding professor for each row in affiliations**. In other words: {firstname, lastname} is a candidate key of professors – a unique combination of columns. It isn't one in affiliations though, professors can have more than one affiliation.

Because professors are referenced by professor_id now, the firstname and lastname columns are no longer needed, so it's time to drop them. After all, one of the goals of a database is to reduce redundancy where possible.

```
ALTER TABLE affiliations
DROP COLUMN firstname;
```

```
ALTER TABLE affiliations
DROP COLUMN lastname;
```

Referential integrity violations

Given the current state of your database, what happens if you execute the following SQL statement?

```
DELETE FROM universities WHERE id = 'EPF';
```

Answer: It fails because referential integrity from professors to universities is violated. You defined a foreign key on professors.university_id that references universities.id, so referential integrity is said to hold from professors to universities.

As commented in the beginning of previous chapter, this is related to the reference counting in general. Whenever a reference still pointing to an object, we cannot delete it.

Change the referential integrity behavior of a key

We have implemented three foreign key constraints:

professors.university_id to universities.id
affiliations.organization_id to organizations.id
affiliations.professor_id to professors.id

These foreign keys currently have the behavior **ON DELETE NO ACTION**. Here, you're going to change that behavior for the column referencing organizations from affiliations. If an organization is deleted, all its affiliations (by any professor) should also be deleted. **The reason this is fine is because it does not violate the reference counting rule. In other words, an object can only be deleted when there are no references pointing to it.**

Altering a key constraint doesn't work with ALTER COLUMN. Instead, you have to delete the key constraint and then add a new one with a different ON DELETE behavior.

For deleting constraints, though, you need to know their name. This information is also stored in information_schema.

```
-- Identify the correct constraint name
SELECT constraint_name, table_name, constraint_type
FROM information_schema.table_constraints
WHERE constraint_type = 'FOREIGN KEY';
```

```
 constraint_name    table_name    constraint_type
affiliations_organization_id_fkey    affiliations    FOREIGN KEY
affiliations_professor_id_fkey    affiliations    FOREIGN KEY
professors_fkey    professors    FOREIGN KEY
```

Delete the affiliations_organization_id_fkey foreign key constraint in affiliations.

```
-- Drop the right foreign key constraint
ALTER TABLE affiliations
DROP CONSTRAINT affiliations_organization_id_fkey;
```

Add a new foreign key that cascades deletion if a referenced record is deleted from organizations. Name it affiliations_organization_id_fkey.

```
-- Add a new foreign key constraint from affiliations to organizations which cascades deletion
ALTER TABLE affiliations
ADD CONSTRAINT affiliations_organization_id_fkey FOREIGN KEY (organization_id) REFERENCES organizations (id) ON DELETE
CASCADE;
```

Run the DELETE and SELECT queries to double check that the deletion cascade actually works.

```
-- Delete an organization
DELETE FROM organizations
WHERE id = 'CUREM';
```

```
-- Check that no more affiliations with this organization exist
SELECT * FROM organizations
WHERE id = 'CUREM';
```

```
id organization_sector
```

As you can see, whenever an organization referenced by an affiliation is deleted, the affiliations also gets deleted. It is your job as database designer to judge whether this is a sensible configuration. Sometimes, setting values to NULL or to restrict deletion altogether might make more sense!

Count affiliations per university

- Run some exemplary SQL queries on the database with grouping by columns and joining tables.
- Find out which university has the most affiliations (through its professors). For that, we need both affiliations and professors tables, as the latter also holds the university_id.
- Count the number of total affiliations by university. Sort the result by that count, in descending order.

```
-- Count the total number of affiliations per university
SELECT COUNT(*), professors.university_id
FROM affiliations
JOIN professors
ON affiliations.professor_id = professors.id
GROUP BY professors.university_id
ORDER BY count DESC;
```

```
count university_id
579 EPF
273 USG
162 UBE
```

As you can see, the count of affiliations is completely different from university to university.

Join all the tables together

In this last exercise, you will find the university city of the professor with the most affiliations in the sector "Media & communication". For this, you need to join all the tables, group by a column, and then use selection criteria to get only the rows in the correct sector.

Due to the design of primary keys/foreign keys, the table joinings below on primary/foreign keys will be very efficient.

```
-- Join all tables
SELECT *
FROM affiliations
JOIN professors
ON affiliations.professor_id = professors.id
JOIN organizations
ON affiliations.organization_id = organizations.id
JOIN universities
ON professors.university_id = universities.id;
```

```
 function    organization_id    professor_id    id    firstname    lastname    university_id    id
organization_sector    id    university    university_city
NA    CIHA    442    442    Peter    Schneemann    UBE    CIHA    Not classifiable    UBE    Uni Bern    Bern
Panel Member     SNF Ambizione Program    1    1    Karl    Aberer    EPF    SNF Ambizione Program    Education
& research    EPF    ETH Lausanne    Lausanne
Member of Conseil Fondation IDIAP    Fondation IDIAP    1    1    Karl    Aberer    EPF    Fondation IDIAP
Education & research    EPF    ETH Lausanne    Lausanne
```

Now group the result by organization sector, professor, and university city. Count the resulting number of rows.

```
-- Group the table by organization sector, professor and university city
SELECT COUNT(*), organizations.organization_sector,
professors.id, universities.university_city
FROM affiliations
JOIN professors
ON affiliations.professor_id = professors.id
JOIN organizations
ON affiliations.organization_id = organizations.id
JOIN universities
```

```
ON professors.university_id = universities.id
GROUP BY organizations.organization_sector,
professors.id, universities.university_city;
```

```
 count    organization_sector    id    university_city
1    Not classifiable    47    Basel
2    Media & communication    361    Saint Gallen
1    Education & research    140    Zurich
```

Only retain rows with "Media & communication" as organization sector, and sort the table by count, in descending order.

```
-- Filter the table and sort it
SELECT COUNT(*), organizations.organization_sector,
professors.id, universities.university_city
FROM affiliations
JOIN professors
ON affiliations.professor_id = professors.id
JOIN organizations
ON affiliations.organization_id = organizations.id
JOIN universities
ON professors.university_id = universities.id
WHERE organizations.organization_sector = 'Media & communication'
GROUP BY organizations.organization_sector,
professors.id, universities.university_city
ORDER BY count DESC;
```

```
count organization_sector id university_city 4 Media & communication 538 Lausanne 3 Media & communication 365 Saint Gallen
```

The professor with id 538 has the most affiliations in the "Media & communication" sector, and he or she lives in the city of Lausanne. Thanks to your database design, you can be sure that the data you've just queried is consistent. Of course, you could also put university_city and organization_sector in their own tables, making the data model even more formal. However, in database design, you have to strike a balance between modeling overhead, desired data consistency, and usability for queries like the one you've just wrote.

Sketches of tables used in this course

We create and populate the first four tables from the last one to make a better database model.

Table: affiliation

firstname	lastname	function	organization
Karl	Aberer	Chairman of L3S Advisory Board	L3S Advisory Board
Karl	Aberer	Member Conseil of Zeno-Karl Schindler Foundation	Zeno-Karl Schindler Foundation
Karl	Aberer	Member of Conseil Fondation IDIAP	

Table: Universities

id	university	university_city
EPF	ETH Lausanne	Lausanne
ETH	ETH Zürich	Zurich
UBA	Uni Basel	Basel

Table: professors

firstname	lastname	university_shortname
Karl	Aberer	EPF
Reza Shokrollah	Abhari	ETH
Georges	Abou Jaoudé	EPF

Table: organizations

id	organization_sector
2014 IARU Congress, Copenhagen, Denmark	Education & research
42matters AG	Technology
A*Star IMB Biopolis, Singapour	Education & research

Table: university_professors

firstname	lastname	university	university_shortname	university_city	function	organization	organization_sector
Karl	Aberer	ETH Lausanne	EPF	Lausanne	Chairman of L3S Advisory Board	L3S Advisory Board	Education & research
Karl	Aberer	ETH Lausanne	EPF	Lausanne	Member Conseil of Zeno-Karl Schindler Foundation	Zeno-Karl Schindler Foundation	Education & research
Karl	Aberer	ETH Lausanne	EPF	Lausanne	Member of Conseil Fondation IDIAP	Fondation IDIAP	Education & research

Karl	Aberer	ETH Lausanne	EPF	Lausanne	Panel Member	SNF Ambizione Program	Education & research
Reza Shokrollah	Abhari	ETH Zürich	ETH	Zurich	Aufsichtsratsmandat	PNE Wind AG	Energy, environment & mobility
Georges	Abou Jaoudé	ETH Lausanne	EPF	Lausanne	Professeur invité (2 interventions d'une semaine)	Kazan Federal University, Russia	Education & research
Hugues	Abriel	Uni Bern	UBE	Bern	NA	Cloetta Stiftung	Education & research
Daniel	Aebersold	Uni Bern	UBE	Bern	NA	Berner Radium-Stiftung	Pharma & health
Daniel	Aebersold	Uni Bern	UBE	Bern	NA	Janser Krebs-Stiftung	Pharma & health
Daniel	Aebersold	Uni Bern	UBE	Bern	NA	SWAN Isotopen AG	Technology