Chapter 1
Your First Database

Introduction to Relational Databases

Why use a relational database?

- Real-life entities become tables (e.g. professors, universities, and companies, and each table only contains data from one entity)
- Reduced redundancy (e.g. only one entry in companies for the bank 'Credit Suisse')
- Data integrity by relationships (e.g. a professor can work at multiple universities and companies, and a company can employ multiple professors)

```
SELECT table_schema, table_name
FROM information_schema.tables;
```

"Information_schema" is some sort of meta-database that holds information about your current database. It is not PostgreSQL specific and is also available in other database management systems like MySQL or SQL Server.

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

*** To get information on all the tables in the 'public' schema ***

```
SELECT column_name, data_type
FROM information_schema.columns
WHERE table_name = 'university_professors'
     AND table_schema = 'public'
```

*** To get columns and data types in the table 'university_professors' of the 'public' schema ***

```
SELECT COUNT (*)
FROM information_schema.columns
WHERE table_name = 'university_professors'
     AND table_schema = 'public'
```

*** To get the count of columns in the table 'university_professors' of schema 'public' ***

Tables: At the core of every database

How to create a table?

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

*** To create a table named professors with columns firstname and last name with data types text ***

```
-- Add the university_shortname column
ALTER TABLE professors
ADD COLUMN university_shortname text;
```

Update your database as the structure changes

In this chapter, we will migrate data from the university_professors table to 4 different entity tables.

INSERT DISTINCT records INTO the new tables

```
INSERT INTO organizations
SELECT DISTINCT organizations,
      organization_sector
FROM university_professors;
```

This helps us migrate the distinct organization's records from the university_professors table to a new table of organizations.

```
INSERT INTO table_name (column_a, column_b)
VALUES ("value_a", "value_b")
```

Above is the normal use case for “INSERT INTO”

Changing the name of a column in a table

```
ALTER TABLE table_name
RENAME COLUMN old_name TO new_name;
```

DROP column from a table

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

DROP a table

```
DROP TABLE table_name;
```

Chapter 2
Enforce data consistency with attribute constraints

Better data quality with constraints

Apart from storing different entity types, such as professors, in different table, the idea of a database is to push data into a certain structure - a pre-defined model, where you enforce data types, relationships, and other rules. Generally these rules are calle **integrity constraints**.

Integrity constraints

1. **Attribute constraints**, e.g. data types on columns
2. **Key constraints**, e.g. primary keys
3. **Referential integrity constraints**, enforced through foreign keys

Why constraints?

- Constraints give the data structure (e.g. everyone to type the birthdate in the same format as assigned)
- Constraints help with consistency, and thus data quality (makes it easy to pre process)
- Data quality is a business advantage / data science prerequisite
- Enforcing is difficult, but PostgreSQL helps

Attribute constraints

In the simplest form, attribute constraints are data types that can be specified for each column of a table.

There are basic data types for numbers such as “bigint”, or strings of characters, such as “character varying”.

Threr are also more high-level data types like “cidr” for IP addresses. Implementing such a type on column would disallow anything that doesn’t fir the structure of an IP.

Dealing with data types (casting)

```
CREATE TABLE weather (
   temperature integer,
   wind_speed text);
```

```
SELECT temperature * wind_speed AS wind_chill
FROM weather;
```

```
Operator does not exist: integer * text
```

As we can see above, data types also restrict possible SQL operations on the stored data. For example, it is impossible to calculate a product from an integer and a text column.

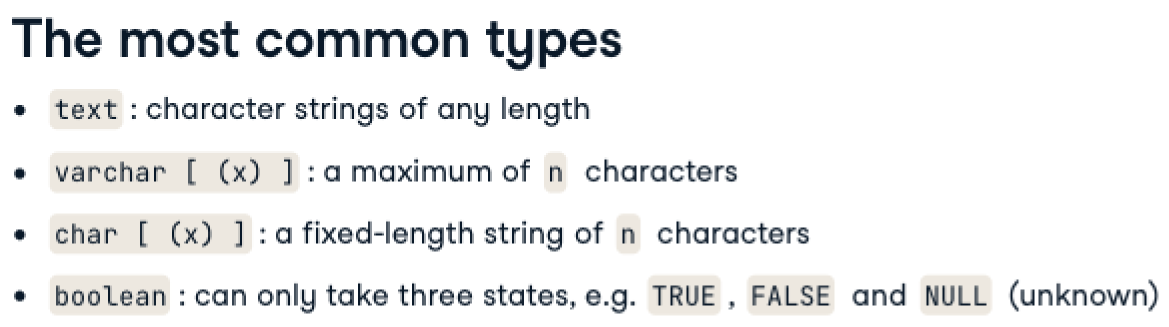
```
SELECT temperature * CAST(wind_speed AS integer) AS wind_chill
FROM weather;
```

The code use CAST function to change the data type of wind_speed on the fly and let us run the query and get the desired result.

Working with data types

- Enforced on columns (i.e. attributes)
- Define the so-called "domain" of a column (what form these values can take and what not)
- Define what operations are possible
- Enforce consistent storage of values (e.g. a postal code will have no more than 6 digits, according to your conventions)

This greatly helps with data quality.



The most common types

- `text` : character strings of any length
- `varchar [ (x) ]` : a maximum of `n` characters
- `char [ (x) ]` : a fixed-length string of `n` characters
- `boolean` : can only take three states, e.g. `TRUE` , `FALSE` and `NULL` (unknown)

The most common types (cont'd.)

- `date` , `time` and `timestamp` : various formats for date and time calculations
- `numeric` : arbitrary precision numbers, e.g. `3.1457`
- `integer` : whole numbers in the range of `-2147483648` and `+2147483647`

```
CREATE TABLE students (
ssn integer,
Name varchar(64),
dob date,
average_grade numeric (3,2), -- e.g. 5.54
tuition_paid boolean
);
```

Alter types after table creation

```
ALTER TABLE students
ALTER COLUMN name
TYPE varchar(128);
```

```
ALTER TABLE students
ALTER COLUMN average_grade
TYPE integer
-- Turns 5.54 into 6, not 5, before type conversion
USING ROUND(average_grade);
```

```
ALTER TABLE professors
ALTER COLUMN firstname
TYPE varchar(16)
USING SUBSTRING (firstname FROM 1 FOR 16)
```

The query above helped to truncate the results of before converting it which mean if the column had a first name which was greater than 16 characters,the character after 16 would be removed.

The not-null and unique constraints

The not-null constraint

- Disallow NULL values in a certain column
- Must hold true for the current state
- Must hold true for any future state

What does NULL mean?

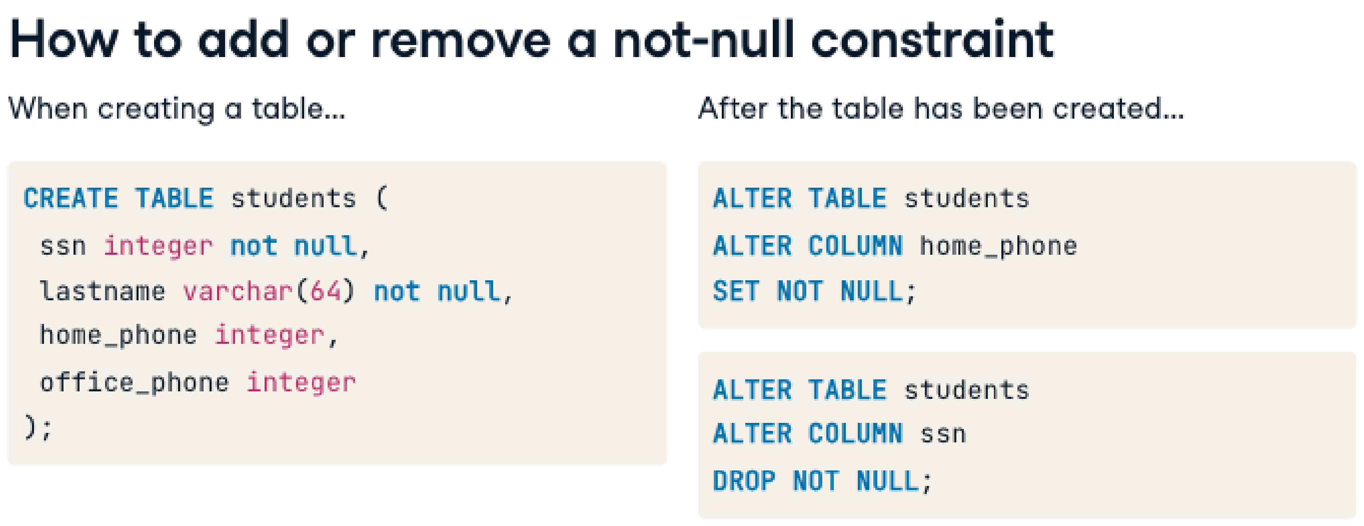
- Unknown
- Does not exist
- Does not apply

Let's say we define a table "students". The first two columns for the social security number and the last name cannot be "NULL", which makes sense: this should be known and apply to every student.

The "home_phone" and "office_phone" columns though should allow for null values – which is the default, by the way.

Why? First of all, these numbers can be unknown, for any reason, or simply not exist, because a student might not have a phone. Also, some values just don't apply: Some students might not have an office, so they don't have an office phone, and so forth.

So, one important take away is that two "NULL" values must not have the same meaning. This also means that comparing "NULL" with "NULL" always results in a "FALSE" value.



How to add or remove a not-null constraint

When creating a table...

```
CREATE TABLE students (
 ssn integer not null,
 lastname varchar(64) not null,
 home_phone integer,
 office_phone integer
);
```

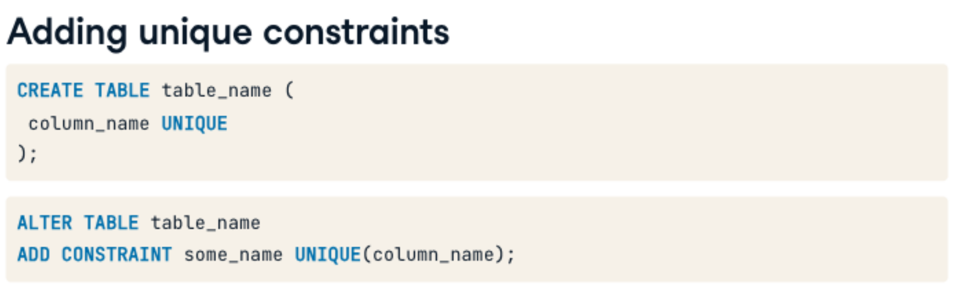
After the table has been created...

```
ALTER TABLE students
ALTER COLUMN home_phone
SET NOT NULL;
```

```
ALTER TABLE students
ALTER COLUMN ssn
DROP NOT NULL;
```

The unique constraint

- Disallow duplicate values in a column
- Must hold true for the current state
- Must hold true for the future state



Adding unique constraints

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

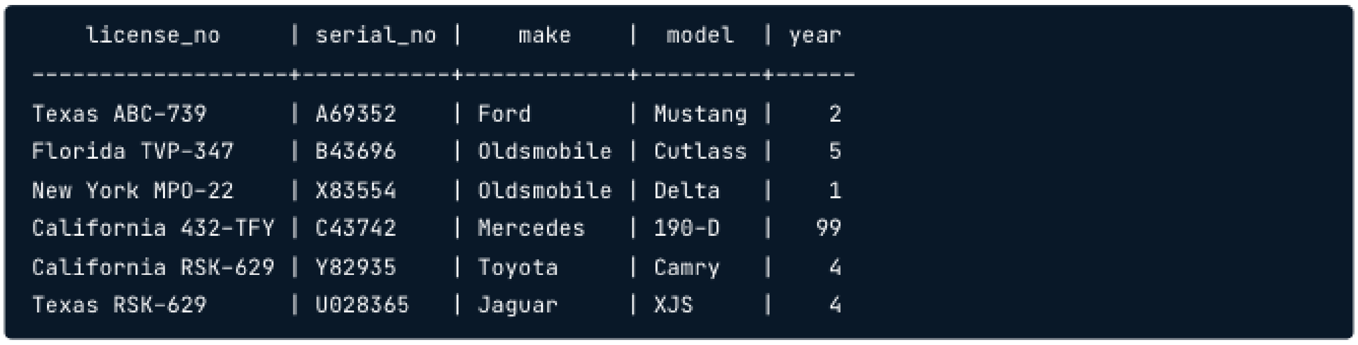
```
ALTER TABLE table_name
ADD CONSTRAINT some_name UNIQUE(column_name);
```

Chapter 3
Uniquely identify records with key constraints

Keys and superkeys

What is a key?

- Attribute(s) that identify a record uniquely
- As long as attributes can be removed: **superkey**
- If no more attributes can be removed : minimal superkey or **key**

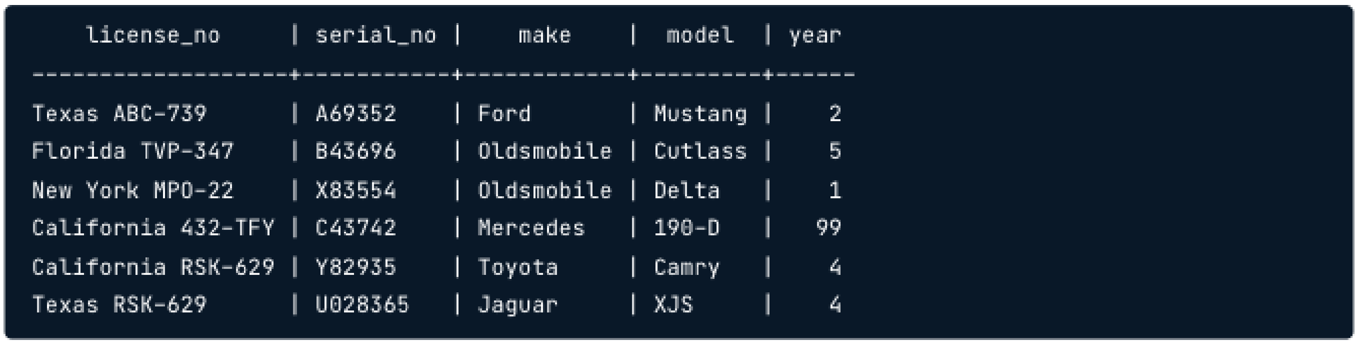


```
    license_no      | serial_no |    make    |  model  | year
--------------------+-----------+------------+---------+------
Texas ABC-739       | A69352    | Ford       | Mustang |    2
Florida TVP-347     | B43696    | Oldsmobile | Cutlass |    5
New York MPO-22     | X83554    | Oldsmobile | Delta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

SK1 = {license_no, serial_no, make, model, year}

SK2 = {license_no, serial_no, make, model}

SK3 = {make, model, year}, SK4 = {license_no, serial_no}, SKi, ..., SKn



```
    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
```

K1 = {license_no}; K2 = {serial_no}; K3 = {model}; K4 = {make, year}

These unique keys are also called candidate keys since only one key can be chosen from the candidate keys to represent our table.

Primary Keys

- One primary key per database table, chosen from candidate keys
- Uniquely identifies records, e.g. for referencing in other tables
- Unique and not-null constraints both apply
- Primary keys are time-invariant: choose columns wisely!

Specifying primary keys

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

- We can also specify two columns as primary key as shown in the right table created above. Note: that there still is just one primary key and it's just the combination of two columns.

Ideally primary key consists of as less columns as possible.

```
ALTER TABLE table_name
ADD CONSTRAINT some_name PRIMARY KEY (column_name)
```

The code above helps us to add PRIMARY KEY to the existing table.

Surrogate keys

Surrogate keys are sort of an artificial primary key. In other words, they are not based on a native column in your data, but on a column that just exists for the sake of having a primary key.

Reasons for creating surrogate keys:

- Primary keys should be built from as few columns as possible
- Primary keys should never change over time

```
    license_no      | serial_no |    make    |  model  | color
--------------------+-----------+------------+---------+------
Texas ABC-739       | A69352    | Ford       | Mustang | blue
Florida TVP-347     | B43696    | Oldsmobile | Cutlass | black
New York MPO-22     | X83554    | Oldsmobile | Delta   | silver
California 432-TFY  | C43742    | Mercedes   | 190-D   | champagne
California RSK-629  | Y82935    | Toyota     | Camry   | red
Texas RSK-629       | U028365   | Jaguar     | XJS     | blue
```

```
   make     |  model  | color
------------+---------+------
Ford        | Mustang | blue
Oldsmobile  | Cutlass | black
Oldsmobile  | Delta   | silver
Mercedes    | 190-D   | champagne
Toyota      | Camry   | red
Jaguar      | XJS     | blue
```

As you can see in the example above. In the first table it is easy to identify that license_no can be our primary key as it is highly unlikely for the license_no to change for a vehicle.

But in the table below that. Let's say we have only three columns. Make + Model seems like the ideal choice for a primary key.

Here, we can add a new surrogate key column called "id" to solve this problem.

Adding a surrogate key serial data type

Adding a surrogate key with serial data type

```
ALTER TABLE cars
ADD COLUMN id serial PRIMARY KEY;
```

The code above automatically adds serial numbers to the table and when new entries are made in the table, they are assigned a new serial number which is not already present in the id column.

There are other data types like this for the same function in other database management systems like MySQL.

```
INSERT INTO cars
VALUES ('Opel', 'Astra', 'green', 1);
```

```
duplicate key value violates unique constraint "id_pkey"
DETAIL:  Key (id)=(1) already exists.
```

If one tries to insert a value which already exists in the surrogate key column for the table. It will throw an error.

- "Id" uniquely identifies records in the table - useful for referencing!

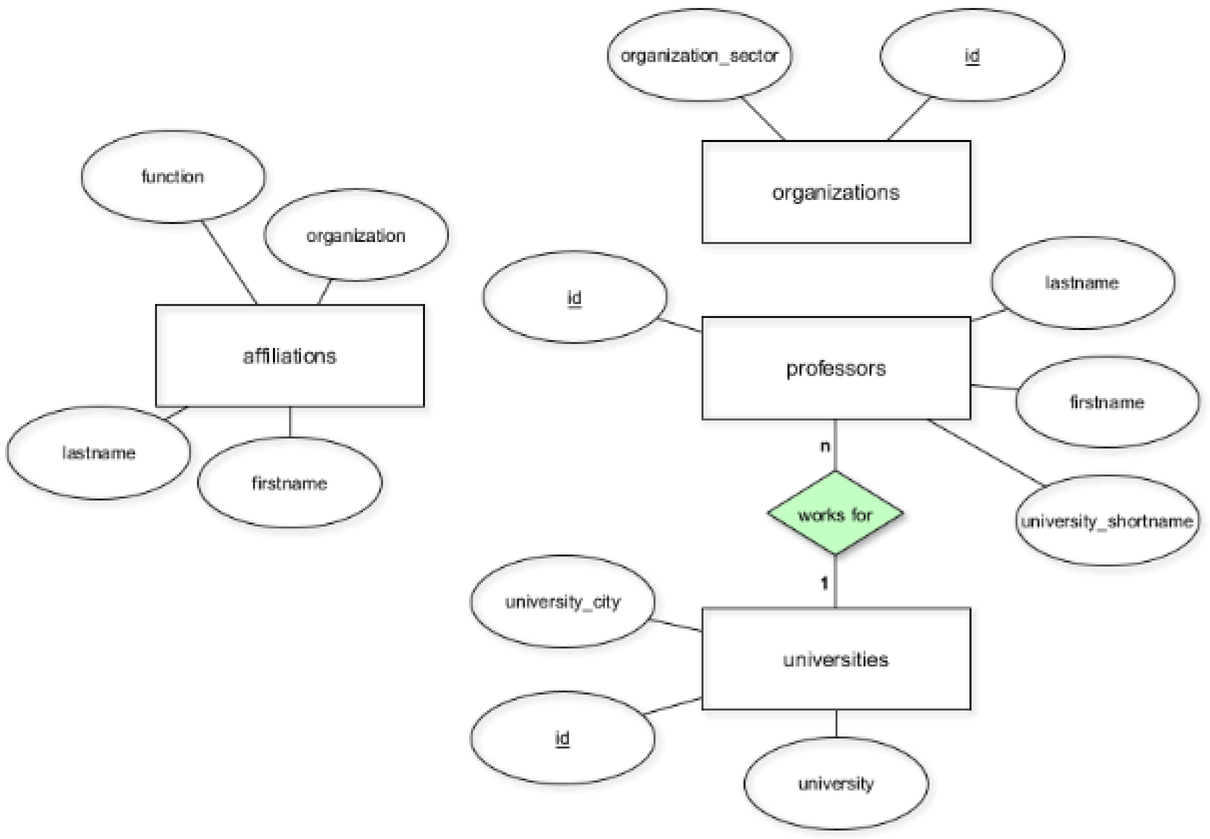
Another type of surrogate key

```
ALTER TABLE table_name
ALTER COLUMN column_c varchar(256);

UPDATE table_name
SET column_c = CONCAT(column_a, column_b);
ALTER TABLE table_name
ADD CONSTRAINT pk PRIMARY KEY (column_c);
```

Chapter 4
Glue together tables with foreign keys

Model 1:N relationships with foreign keys

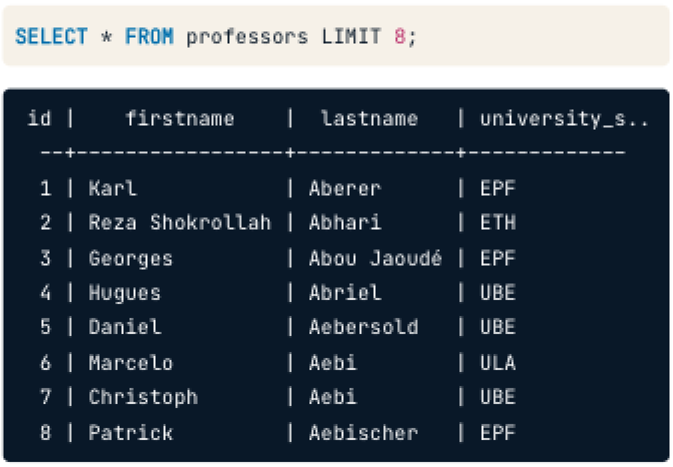


In this database, each professor works for a university. In the ER diagram, this is drawn with a rhombus. The small numbers specify the cardinality of the relationship: a professor works for at most one university, while a university can hagve any number of professors working for it - even zero.

Such relationships are implemented with foreign keys.

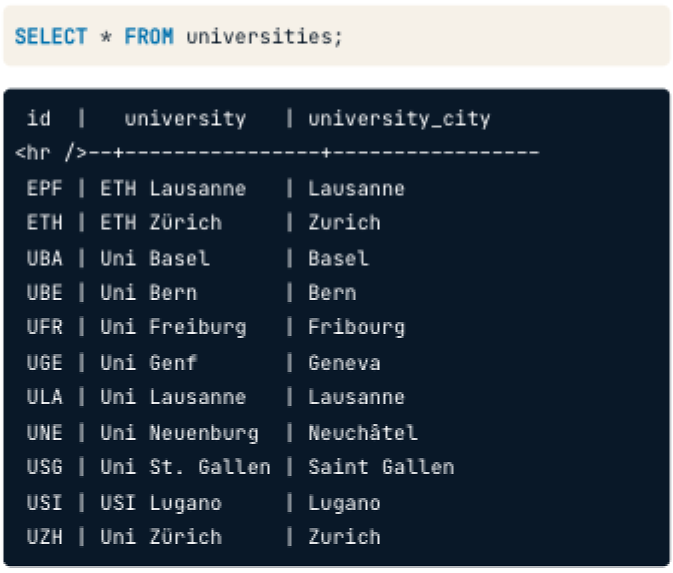
Implementing relationships with foreign keys

- A foreign key (FK) points to the primary key (PK) of another table
- Domain of FK must be equal to domain of PK
- Each value of FK must exist in PK of other table (FK constraint or “referential integrity”)
- FK’s are not actual keys (duplicates and null values are allowed)



```
SELECT * FROM professors LIMIT 8;
```

```
 id |     firstname     |  lastname   | university_s..
 --+-----------------+------------+-------------
  1 | Karl              | Aberer      | EPF
  2 | Reza Shokrollah | Abhari      | ETH
  3 | Georges           | Abou Jaoudé | EPF
  4 | Hugues            | Abriel      | UBE
  5 | Daniel            | Aebersold   | UBE
  6 | Marcelo           | Aebi        | ULA
  7 | Christoph         | Aebi        | UBE
  8 | Patrick           | Aebischer   | EPF
```



```
SELECT * FROM universities;
```

```
 id  |    university    | university_city
<hr />--+-----------------+-----------------
 EPF | ETH Lausanne     | Lausanne
 ETH | ETH Zürich       | Zurich
 UBA | Uni Basel        | Basel
 UBE | Uni Bern         | Bern
 UFR | Uni Freiburg     | Fribourg
 UGE | Uni Genf         | Geneva
 ULA | Uni Lausanne     | Lausanne
 UNE | Uni Neuenburg    | Neuchâtel
 USG | Uni St. Gallen   | Saint Gallen
 USI | USI Lugano       | Lugano
 UZH | Uni Zürich       | Zurich
```

As you can see, the column "university_shortname" of "professors" has the same domain as the "id" column of the "universities" table.

If you go through each record of "professors", you can always find the respective "id" in the "universities" table.

So both criteria for a foreign key in the table "professors" referencing "universities" are fulfilled. Also, you see that "university_shortname" is not really a key because there are duplicates.

For example, the id "EPF" and "UBE" occur three times each.

Specifying foreign keys

```
CREATE TABLE manufactures (
 Name varchar(255) PRIMARY KEY);

INSERT INTO manufacturers
VALUES ('Ford'), ('VW'), ('GM');
```

```
CREATE TABLE cars (
 Model varchar(255) PRIMARY KEY,
 Manufacturer_name varchar(255) REFERENCES manufacturers (name));

INSERT INTO cars
VALUES ('Ranger','Ford'), ('Beetle','VW');
```

```
--Throws an error!
INSERT INTO cars
VALUES ('Tundra','Toyota')
```

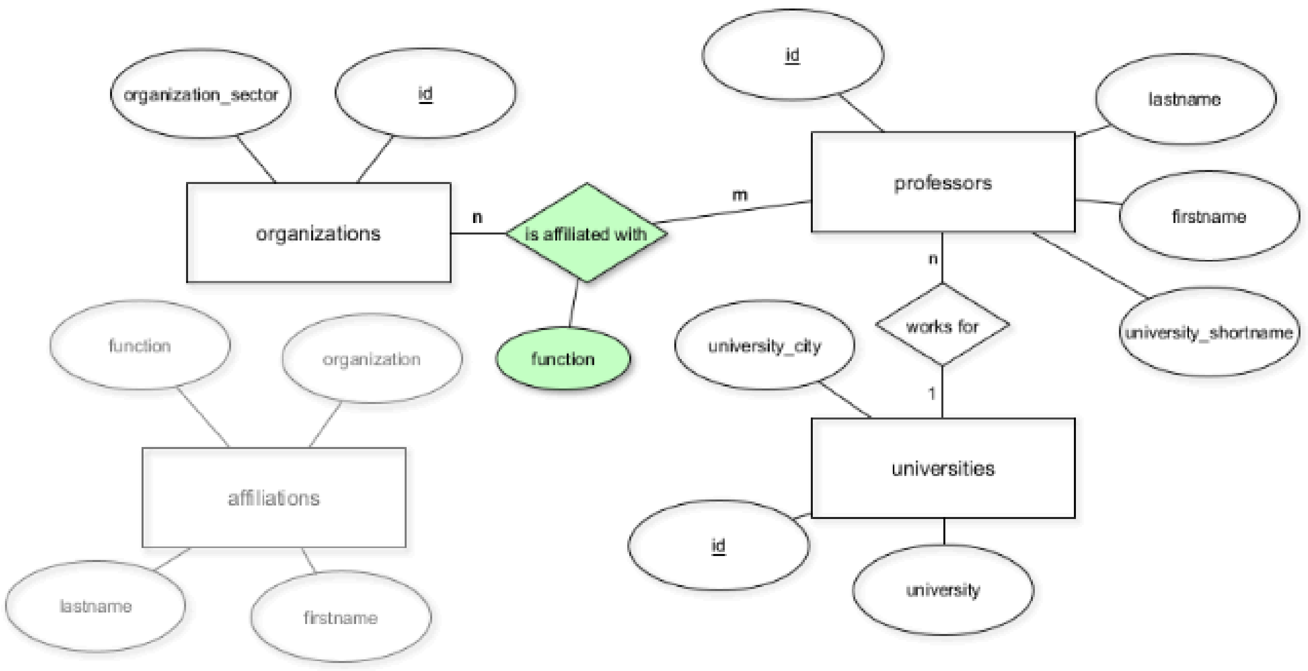
We use REFERENCES keyword to assign foreign key. In the example above once the foreign key is assigned it is not possible to enter the model of the manufacturer not present in the manufacturers table because of the foreign key constraint.

Specifying foreign keys to existing tables

```
ALTER TABLE a
ADD CONSTRAINT a_fkey FOREIGN KEY (b_id) REFERENCES b (id);
```

Model more complex relationships

N:M relationship



Now in case of professors and organizations, we can see that professors can be associated with multiple orgranizations and organizations can be associated with multiple professors and this relationship has its own attribute, the function. Each affiliation comes up with a functions, for instance, “chairman”.

One important thing here is that now this relationship can be used to query anything that affiliations entity had to offer and affiliations as an entity can be removed altogether. It is still shown in the diagram above but its not needed.

How to implement N:M-relationships

- Create a table
- Add foreign keys for every connected table
- Add additional attributes

```
CREATE TABLE affiliations (
   professor_id integer REFERENCES professors (id),
   organization_id varchar(256) REFERENCES organizations (id),
   Function varchar(256)
);
```

- No primary key!
- Possible PK = {professor_id, organization_id, function} (one could possibly define the combination of 3 as PK to have some kind of constraint but that would be a bit over the top)

Referential integrity

- A record referencing another table must refer to an existing record in that table
- Specified between two table
- Enforced through foreign keys

Referential integrity violations

Referential integrity from table A to table B is violated

- ...if a record in table B that is referenved from a record in table A is deleted.
- ...if a record in table A referencing a non-existing record from table B is inserted.
- Foreign keys prevent these violations

Dealing with violations

If we specify a foreign key on a column, we can actually tell the database system what should happen if an entry in the referenced table is deleted.

```
CREATE TABLE a (
  Id integer PRIMARY KEY,
  column_a varchar(64),
  ...,
  b_id integer REFERENCES b (id) ON DELETE NO ACTION
);
```

The above code "ON DELETE NO ACTION" keyword is automatically appended to a foreign key definition. This means that if we try to delete a record in table B which is referenced from table A, the system will throw an error.

However, there is another way.

```
CREATE TABLE a (
  Id integer PRIMARY KEY,
  column_a varchar(64),
  ...,
  b_id integer REFERENCES b (id) ON DELETE CASCADE
);
```

The CASCADE option above will first allow the deletion of the record in table B, and then will automatically delete all referencing records in table A. So that deletion is cascaded.

Other ways:

- RESTRICT: throws an error (almost similar to NO ACTION, small technical differences beyond the scope of the course right now)
- SET NULL: set the referencing column to NULL
- SET DEFAULT: set the referencing column to its default value (needs to be specified what the default value is in this case)

```
SELECT constraint_name, table_name, constraint_type
FROM information_schema.table_constraints
WHERE constraint_type = 'FOREIGN KEY';
```

The code above checks for foreign key constraints.

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

The above code changes the constraint from NO ACTION to CASCADE. It is not possible to do it directly so we have to remove the constraint first and then add constraint again.