Working with relations

Use the `spaces` table for this introduction. The needed tables were created in the *Normalization* session. If you didn't follow that lecture, you can create the neccessary tables using the hydenv CLI.

```
python -m hydenv examples spaces --normalize
```

Below, you will find the SQL queries used in the video, followed by a summary of the lessons learned.

SQL commands in the video

Relations 01

1:03:

```
-- overview
SELECT * FROM spaces LIMIT 50
```

2:40:

```
-- join another table
SELECT * FROM spaces
JOIN locations on locations.location_id=spaces.location_id
LIMIT 50
```

6:16:

```
-- aggregate over a join
SELECT
    l.country,
    count(DISTINCT location_name) as spaces_ports,
    count(*) as launches
FROM spaces s
JOIN locations l on l.location_id=s.location_id
GROUP BY l.country
ORDER BY launches DESC
```

Relations 02

2:56:

```
SELECT location_id FROM locations WHERE country='USA'
```

3:58:

```
-- filter on a subquery
SELECT * FROM
spaces WHERE location_id IN
(
    SELECT location_id FROM locations
```

```
    WHERE country='USA'
)
```

Relations 03

2:57:

```
-- get only missions started in USA from a sub-query
SELECT
    datum,
    mission_name,
    (
        SELECT
            company_name
        FROM companies
        WHERE companies.company_id=spaces.company_id
    ) as company
FROM
spacess
WHERE location_id IN
(
    SELECT location_id FROM locations
    WHERE country='USA'
)
```

3:49:

```
-- use a subquery for selecting an attribute
SELECT
    datum,
    mission_name,
    (
        SELECT
            company_name
        FROM companies
        WHERE companies.company_id=spaces.company_id
    ) as company
FROM
spacess
WHERE location_id IN
(
    SELECT location_id FROM locations
    WHERE country='USA'
)
AND mission_name LIKE '%GPS%'
```

Relations 04

4:40:

```
-- group from a subquery
SELECT
    count(*) AS missions,
    round(date_part('days', max(datum) - min(datum)) / 365) || ' years'
        AS "serving years",
    (
        SELECT
            company_name
        FROM companies
        WHERE companies.company_id=spaces.company_id
    ) AS company
FROM
spacess
WHERE location_id IN
(
    SELECT location_id FROM locations
    WHERE country='USA'
)
AND mission_detail LIKE '%GPS%'
GROUP BY company
```

Relations 05

5:22:

```
-- Create a view from previous query
DROP VIEW IF EXISTS gps_missions_from_usa;
CREATE TEMPORARY VIEW gps_missions_from_usa AS
SELECT
    company_id,
    count(*) AS missions,
    round(date_part('days', max(datum) - min(datum)) / 365) || ' years'
        AS "serving years",
    (
        SELECT
            company_name
        FROM companies
        WHERE companies.company_id=spaces.company_id
    ) AS company
FROM
spaces
WHERE location_id IN
(
```

```
    SELECT location_id FROM locations
    WHERE country='USA'
)
AND mission_detail LIKE '%GPS%'
GROUP BY company, company_id;
SELECT * FROM gps_missions_from_usa;
```

Summary

After normalizing a database, the information needed is usually split up into several tables. Thus we need a means to combine tables again. We will look at three ways: **sub-queries**, **views** and **joins**.

Joins

- Most common approach is to *join* tables.
- You *glue* two table by matching a **foreign key** of the main table `ON` the **primary key** of the other table
- `JOIN`s can be chained and multiple joins on the **same** table are possible.
- If there are common attributes (like `id`, `name`), you need to specify which one to use, eg.: `tablename.attribute_name`

Sub-query

- A sub-query `SELECT`s datasets from another `SELECT` instead of a table
- You have to alias (`AS`) the whole sub-query and can then *use* it like a table
- Sub-queries can be chained
- Sub-queries can also be used in filters

Views

Views are a very powerful concept in SQL. A `VIEW` is a structural, **persistent** element in in PostgreSQL. You can think of it like a saved `SELECT` query.

Instead of nesting multiple sub-queries, maybe including multiple `JOIN`s on each level, you can persist a **view** on the data.

`VIEW`s can be used in `SELECT` queries just like tables. But they don't store the result, only the query. That means, it is only executed at query time, potentially slowing down the whole query. On the other hand, it will always query the most recent values from tables with **dynamic** content.