COPERNICUS MASTER IN DIGITAL EARTH

PARIS LODRON UNIVERSITÄT SALZBURG

Spatial Databases – Dr. Martin Sudmanns

Exercise 13: Database backend for a WebGIS

Final Project

Annabelle Kiefer (s1111172)

Winter 2024/2025

Due date: 21.03.2025

Contents

Task

Create a database for a city festival (choose any, e.g., of your hometown or create a fictional one) as a backend for a WebGIS. Search for information that you can use for data modeling. The database itself should (if no reasons are given) be in the third normal form. Import data and de-normalize – if necessary with views. Create indexes where required. Create at least three queries for the application that is defined in the second requirement to document that the database works.

Introduction

To accomplish this task, I decided to create a database for the *NatureOne Festival* in Kastellaun, Germany. The festival was founded in 1995 and today takes place on the former military base *Pydna* on the hills of the *Hunsrück*. Every year, more than 350 artists from all over the world perform on 20 different stages. These stages can be found both indoors in the form of tents or former bunkers and outdoors. The festival offers a wide range of electronic music from techno to trance and hardstyle. At the same time, the festival is known for its *CampingVillage* with hundreds of private floors and parties (source: https://www.nature-one.de/en).

To generate the data for the database, I mainly used the *Woov* app, which provides festival information in the form of a timetable and an interactive map with various polygon and point data (see figure 1). At this point, it should be noted that some of the information has been simplified. In addition, some attributes, especially in relation to the food spots, were created using pseudo-data.

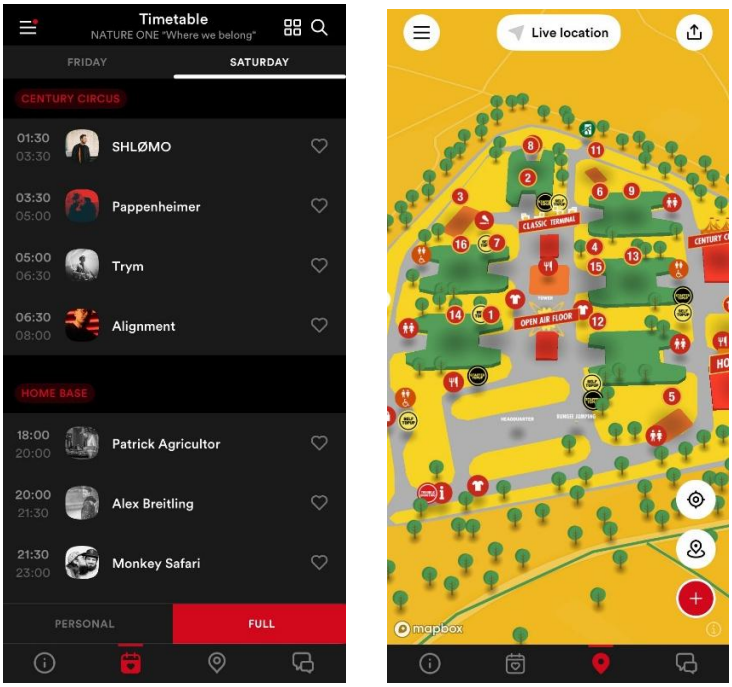

Figure 1. Timetable information (left) and an interactive map (right), retrieved from the *Woov* app.

Database modeling

Conceptual database model

The conceptual database model is an abstract model which defines the main entities and their semantic relationships (name, degree & cardinality) between them. At the same time, it ignores any specifications of the system on which it will be implemented.

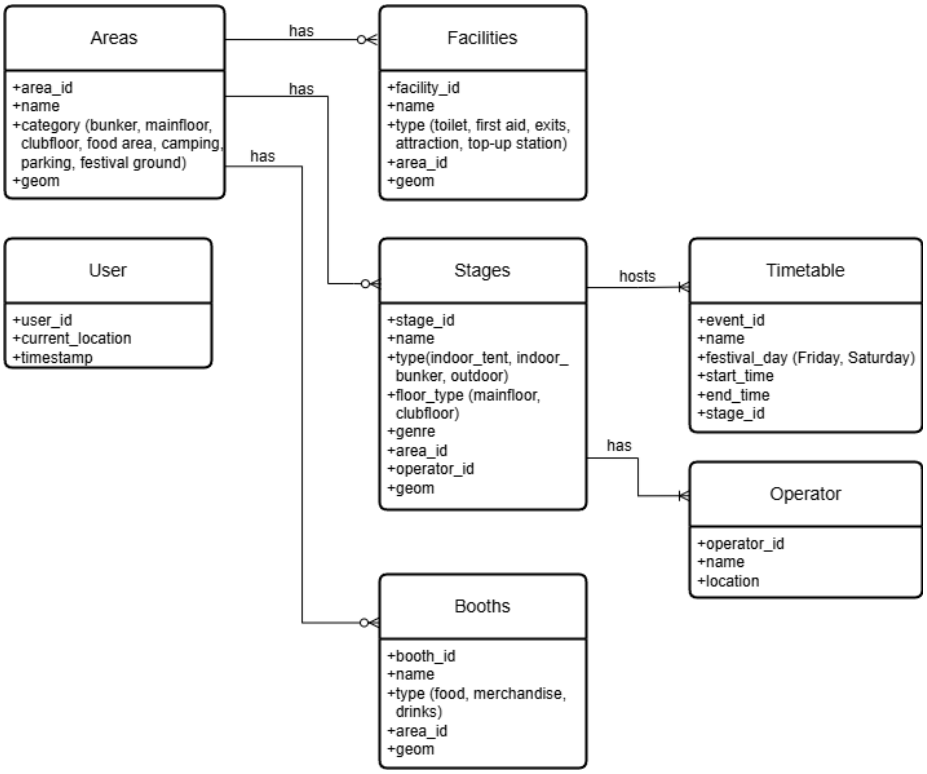

Figure 2. Conceptual database model, created using draw.io.

For this specific scenario, seven different entities, namely *areas*, *facilities*, *stages*, *operators*, *booths*, *timetable* and *user* were defined with their key attributes. At the same time, the relationships between the entities were determined. In this context, *facilities*, *stages* and *booths* are related to *areas* with a 0:N relationship, since a given area may contain zero or many of these attributes. The relationship between *stages* and *timetable* is 1:N, since at least one event takes place on a stage. The table *operator* was created to fulfill the third normal form. Otherwise, the name and location of an operator would be included in the *stages* table, although they depend on each other and not on the primary key. The relationship is 1:N, since a stage has exactly one operator, but an operator can be responsible for more than one stage.

Logical database model

The logical database model is more detailed than the conceptual database model and focuses on how the data is structured in a database, e.g. in a relational database system or in an object-oriented database system. The model is expanded to include primary and foreign keys and normalization comes into focus. Nevertheless, the logical database model is not tied to a specific software product.

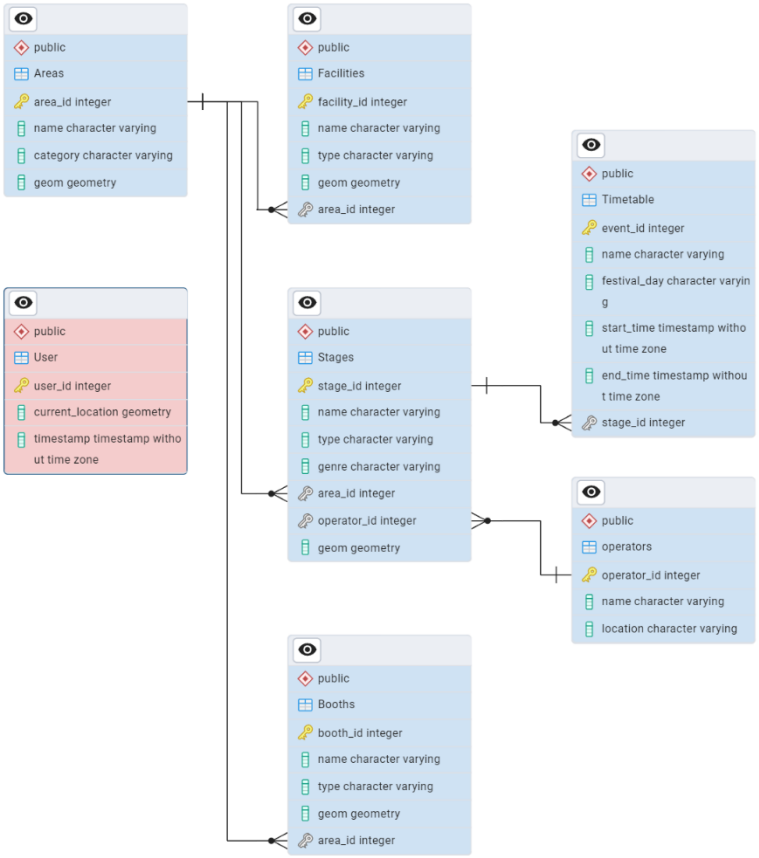

Figure 3. Logical database model created with the ERD tool in pgAdmin.

This logical database model is based on the conceptual database model that was created with draw.io. In addition to the entities with the main attributes and their relationships, the primary and foreign keys as well as the data types of the attributes are now also defined. For the tables *facilities*, *stages* and *booths,* the *area_id* was defined as a foreign key. Similarly, the *timetable* and the *operators* table are linked to the *stages* table via foreign keys. Since this database doesn't contain many-to-many relationships, no join tables were created.

Implementation

Creating tables (DDL)

The first step in implementing the database was to add the *PostGIS* extension. After that, I focused on the Data Definition Language by creating the tables for the seven entities with their rows and datatypes. Here I also added check constraints to reduce errors in certain rows, especially when categories were defined (see Code Block 1). For the tables containing geometry, the coordinate system was set to WGS 84/Pseudo-Mercator (EPSG:3857).

```
CREATE EXTENSION postgis;

CREATE TABLE areas (
    area_id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    category TEXT CHECK (category IN ('bunker', 'mainfloor', 'clubfloor', 'food_area',
'camping', 'parking', 'festival_ground')),
    geom GEOMETRY(POLYGON, 3857));
```

Code Block 1. Creating a PostGIS Extension and the tables (see the complete code under source code).

Inserting data (DML)

For the Data Manipulation Language, I had two different approaches, depending on whether a table contains geometry or not. For the tables without geometry, i.e. *timetable* & *operators*, I added the information manually using SQL. To speed up the process, I simplified the original data to a certain extent. For the *timetable*, this resulted in 277 rows with information about which DJs are playing on certain stages at a defined time.

```
INSERT INTO timetable (name, festival_day, start_time, end_time, stage_id)
VALUES
    ('Bennett', 'Friday', '2025-08-01 20:00:00', '2025-08-01 21:00:00', 1 ),
    ('LariLuke', 'Friday', '2025-08-01 21:00:00', '2025-08-01 22:30:00', 1 ),
    ('Öwnboss', 'Friday', '2025-08-01 22:30:00', '2025-08-02 00:00:00', 1 ),
    ('NERVO', 'Friday', '2025-08-02 00:00:00', '2025-08-02 01:30:00', 1 ),
    ('Neelix', 'Friday', '2025-08-02 01:30:00', '2025-08-02 03:00:00', 1 ),
    ('OBS', 'Friday', '2025-08-02 03:00:00', '2025-08-02 04:30:00', 1 ),
    ('MOGUAI', 'Friday', '2025-08-02 04:30:00', '2025-08-02 06:00:00', 1 );
```

Code Block 2. Inserting data into the tables without geometry (see the complete code under source code).

For the tables containing geometry data in the form of points or polygons, I used QGIS to create the points and polygons. Simultaneously with the creation of the point and polygon features, I also added the necessary attribute information via QGIS (see figure 4 & 5).

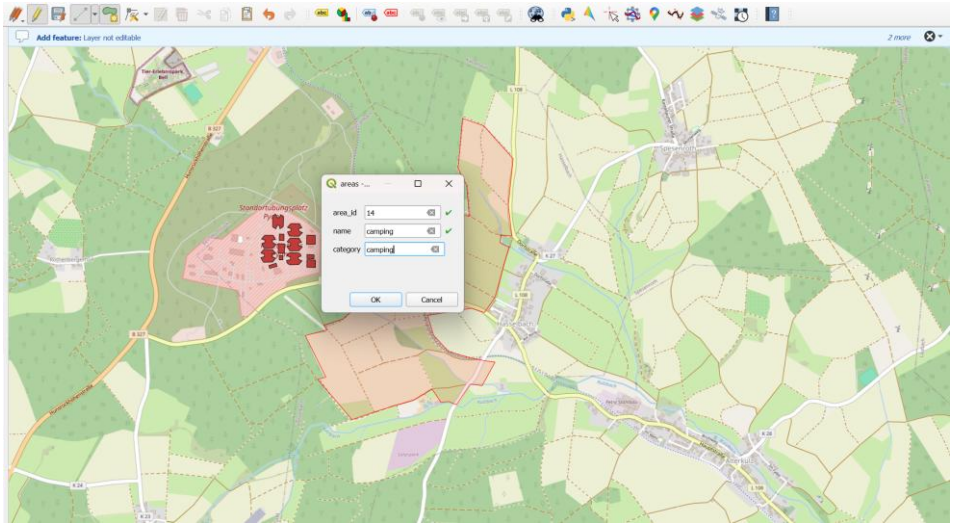

Figure 4. Creating the polygon data for the area table using Toggle Editing in QGIS.

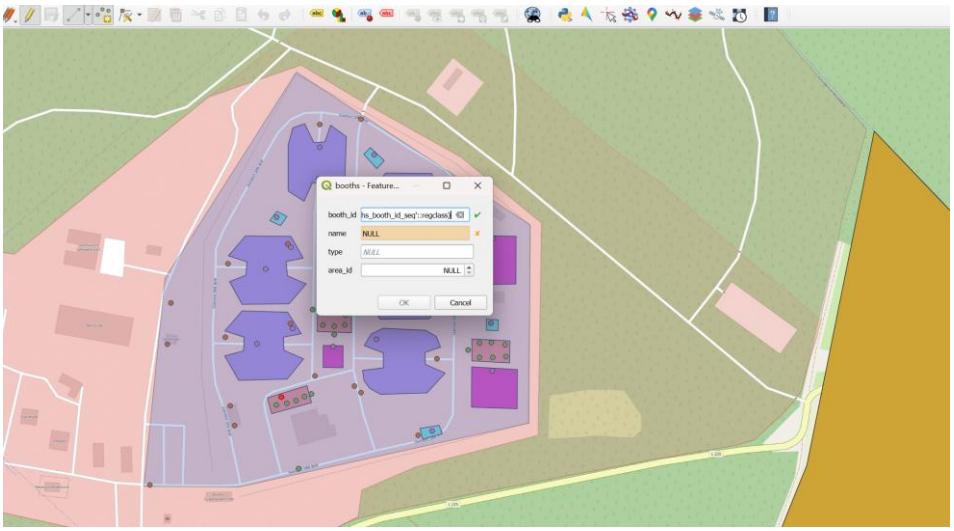

Figure 5. Creating the point data for the booths table using Toggle Editing in QGIS.

In order to be able to recreate the database at a later date, I then queried the insert-statements using SQL. Following this, I copied the insert-statements from *pgAdmin* and pasted them into the source code.

```
INSERT INTO stages (stage_id, name, type, floor_type, genre, area_id, operator_id, geom)
VALUES
    (1, 'OpenAirFloor', 'outdoor', 'mainfloor', 'mixed', 1, 1,
ST_SetSRID(ST_GeomFromText('POINT(826599.5616658438 6453742.386001241)'), 3857)),
    (2, 'ClassicTerminal', 'outdoor', 'mainfloor', 'mixed', 2, 1,
ST_SetSRID(ST_GeomFromText('POINT(826605.7955003252 6453952.777914982)'), 3857)),
    (3, 'CenturyCircus', 'indoor_tent', 'mainfloor', 'techno', 3, 1,
ST_SetSRID(ST_GeomFromText('POINT(826892.0324002552 6453940.050502917)'), 3857)),
    (4, 'Homebase', 'indoor_tent', 'mainfloor', 'house', 4, 1,
ST_SetSRID(ST_GeomFromText('POINT(826890.7336847382 6453698.229673668)'), 3857)),
```

```
    (5, 'DirtyWorkz', 'outdoor', 'clubfloor', 'hardstyle', 17, 2,
ST_SetSRID(ST_GeomFromText('POINT(826780.6026089027 6453587.838854727)'), 3857)),
    (6, 'Gayphoria', 'indoor_bunker', 'clubfloor', 'mixed', 8, 3,
ST_SetSRID(ST_GeomFromText('POINT(826683.9781744438 6454037.973652895)'), 3857)),
    (7, 'HardcoreGladiators', 'indoor_bunker', 'clubfloor', 'hardcore', 6, 4,
ST_SetSRID(ST_GeomFromText('POINT(826522.1100180662 6453923.433268025)'), 3857)),
    (8, 'HEAVEN&HILL', 'outdoor', 'clubfloor', 'techno', 7, 5,
ST_SetSRID(ST_GeomFromText('POINT(826575.0577964879 6454105.695812976)'), 3857)),
    (9, 'HeavensGate', 'outdoor', 'clubfloor', 'techno', 8, 6,
ST_SetSRID(ST_GeomFromText('POINT(826733.392018499 6453994.709123592)'), 3857)),
    (10, 'HEXTechnomovement', 'outdoor', 'clubfloor', 'techno', 20, 7,
ST_SetSRID(ST_GeomFromText('POINT(826888.1624477317 6453784.445349667)'), 3857)),
    (11, 'Masters_of_Hardcore', 'outdoor', 'clubfloor', 'hardcore', 19, 8,
ST_SetSRID(ST_GeomFromText('POINT(826674.8439942825 6454092.204311743)'), 3857)),
    (12, 'PLAY!', 'indoor_bunker', 'clubfloor', 'mixed', 10, 9,
ST_SetSRID(ST_GeomFromText('POINT(826678.3446000218 6453754.550620814)'), 3857)),
    (13, 'PUNX', 'outdoor', 'clubfloor', 'mixed', 9, 10,
ST_SetSRID(ST_GeomFromText('POINT(826729.066162427 6453855.776056517)'), 3857)),
    (14, 'SUNSHINE_LIVE', 'outdoor', 'clubfloor', 'mixed', 5, 11,
ST_SetSRID(ST_GeomFromText('POINT(826460.437801878 6453746.9315019995)'), 3857)),
    (15, 'Tunnel', 'indoor_bunker', 'clubfloor', 'techno', 9, 12,
ST_SetSRID(ST_GeomFromText('POINT(826680.9911401173 6453825.834958386)'), 3857)),
    (16, 'V.I.B.E.Z', 'indoor_tent', 'clubfloor', 'goa', 6, 13,
ST_SetSRID(ST_GeomFromText('POINT(826476.1114177285 6453883.857951582)'), 3857)),
    (17, 'AcidWars', 'indoor_bunker', 'clubfloor', 'techno', 5, 14,
ST_SetSRID(ST_GeomFromText('POINT(826525.0914672613 6453775.013397065)'), 3857)),
    (18, 'AirportOpenAir', 'outdoor', 'clubfloor', 'techno', 7, 15,
ST_SetSRID(ST_GeomFromText('POINT(826572.3300039219 6454047.777850686)'), 3857)),
    (19, 'BLACKLIST', 'indoor_tent', 'clubfloor', 'dubstep', 18, 16,
ST_SetSRID(ST_GeomFromText('POINT(826496.1388157597 6453978.552714013)'), 3857)),
    (20, 'DieRakete', 'indoor_bunker', 'clubfloor', 'techno', 9, 17,
ST_SetSRID(ST_GeomFromText('POINT(826683.606459567 6453890.470710499)'), 3857));
```

Code Block 3. Inserting data into the tables with geometry (see the complete code under source code).

Creating Views & Indexes (DDL)

After inserting all the data into the different tables, I created two views, the first for the combination of *areas* with *stages*, *booths* and *facilities* and the second for the combination between the *timetable* and the different stages. Both views contain some of the most frequently used features of this database. Instead of joining the different tables together for each query, the views now provide an efficient way to retrieve the data.

```
CREATE VIEW area_details AS
SELECT
    a.area_id, a.name AS area_name, a.category AS area_category, s.stage_id, s.name AS
stage_name,
```

```
    f.facility_id,f.name AS facility_name, f.type AS facility_type,
    b.booth_id, b.name AS booth_name, b.type AS booth_type
FROM areas a
LEFT JOIN stages s ON a.area_id = s.area_id
LEFT JOIN facilities f ON a.area_id = f.area_id
LEFT JOIN booths b ON a.area_id = b.area_id;

CREATE VIEW timetable_stages AS
SELECT
    t.event_id, t.name AS event_name,t.start_time,t.end_time,t.stage_id, s.name AS
stage_name,s.type AS stage_type,
    s.floor_type,s.genre AS stage_genre,s.area_id,s.operator_id
FROM timetable t
JOIN stages s ON t.stage_id = s.stage_id;
```

Code Block 4. Creating views for stage and timetable details.

Likewise, four indexes were created in the database to increase performance. First, I created two indexes for the name rows in the *stages* and *areas* tables, since these are often part of queries. Equally important was the creation of two GIST-indexes for the *geom* data of the *areas* as well as *stages*, since these two are also frequently queried.

```
CREATE INDEX idx_stages_name ON stages(name);
CREATE INDEX idx_areas_name ON areas(name);
CREATE INDEX idx_areas_geom ON areas USING GIST(geom);
CREATE INDEX idx_stages_geom ON stages USING GIST(geom);
```

Code Block 5. Creating general & spatial indexes for the stages & areas tables.

Queries

Non-spatial queries

To test if the created views work correctly, I started with some simple non-spatial queries. First, I used the *timetable_stages* view to select all events taking place on the *OpenAirFloor* during the festival. Secondly, I wanted to see which area contained the most objects (facilities, booths & stages).

```
SELECT * FROM timetable_stages
WHERE stage_name = 'OpenAirFloor'
ORDER BY start_time;

SELECT area_name,
       COUNT(DISTINCT facility_id) AS num_facilities,
       COUNT(DISTINCT stage_id) AS num_stages,
       COUNT(DISTINCT booth_id) AS num_booths
FROM area_details
```

```
WHERE facility_id IS NOT NULL
   OR stage_id IS NOT NULL
   OR booth_id IS NOT NULL
GROUP BY area_name
ORDER BY area_name;
```

Code Block 6. Non-spatial SQL queries using the created views *timetable_stages* & *areas_details*.

	event_id integer	event_name text	start_time timestamp without time zone	end_time timestamp without time zone	stage_id integer	stage_name text	stage_type text	floor_type text	stage_genre text	area_id integer	operator_id integer
1	1	Bennett	2025-08-01 20:00:00	2025-08-01 21:00:00	1	OpenAirFloor	outdoor	mainfloor	mixed	1	1
2	2	LariLuke	2025-08-01 21:00:00	2025-08-01 22:30:00	1	OpenAirFloor	outdoor	mainfloor	mixed	1	1
3	3	Öwnboss	2025-08-01 22:30:00	2025-08-02 00:00:00	1	OpenAirFloor	outdoor	mainfloor	mixed	1	1
4	4	NERVO	2025-08-02 00:00:00	2025-08-02 01:30:00	1	OpenAirFloor	outdoor	mainfloor	mixed	1	1
5	5	Neelix	2025-08-02 01:30:00	2025-08-02 03:00:00	1	OpenAirFloor	outdoor	mainfloor	mixed	1	1
6	6	OBS	2025-08-02 03:00:00	2025-08-02 04:30:00	1	OpenAirFloor	outdoor	mainfloor	mixed	1	1
7	7	MOGUAI	2025-08-02 04:30:00	2025-08-02 06:00:00	1	OpenAirFloor	outdoor	mainfloor	mixed	1	1
8	8	BastiM	2025-08-02 18:00:00	2025-08-02 19:00:00	1	OpenAirFloor	outdoor	mainfloor	mixed	1	1

Total rows: 17 of 17 Query complete 00:00:00.094 Ln 1, Col 1

Figure 6. Result of querying the events on the *OpenAirFloor*.

The second query clearly shows that the festival ground contains the most objects with 18 facilities and 7 booths. This is followed by the food areas with 12, 6 and 5 booths respectively.

	area_name text	num_facilities bigint	num_stages bigint	num_booths bigint
1	BLACKLIST	0	1	0
2	bunker_1	1	2	0
3	bunker_2	1	2	0
4	bunker_3	0	2	0
5	bunker_4	0	2	0
6	bunker_5	0	3	0
7	bunker_6	0	1	0
8	CenturyCircus	0	1	0
9	ClassicTerminal	0	1	0
10	DirtyWorkz	0	1	0
11	festival_ground	18	0	7
12	food_area_1	0	0	12
13	food_area_2	0	0	6
14	food_area_3	0	0	5
15	HEXTechnomovement	0	1	0
16	Homebase	0	1	0
17	Masters_of_Hardcore	0	1	0
18	OpenAirFloor	0	1	0

Figure 7. Result showing the number of facilities, stages and booths per area.

Spatial queries

The database allows a variety of spatial queries. I first wanted to query an object that is closest to a certain location. I entered the location of to the *CenturyCircus* stage and then queried for the nearest facilities with toilets.

```
WITH stage_location AS (
    SELECT geom FROM stages WHERE name = 'CenturyCircus')
SELECT f.name, f.type, f.geom, ST_Distance(f.geom, s.geom) AS distance
FROM facilities f, stage_location s
```

```
WHERE f.type = 'toilet'
ORDER BY distance;
```

Code Block 7. Spatial query used to identify toilets near the CenturyCircus stage.

This query can also be performed in QGIS in order to visualize the toilets closest to the stage. In the following example, you can see the *CenturyCircus* stage in black and the toilets closest to it in light colors. The dark colors show the toilets that are furthest away from the selected stage.

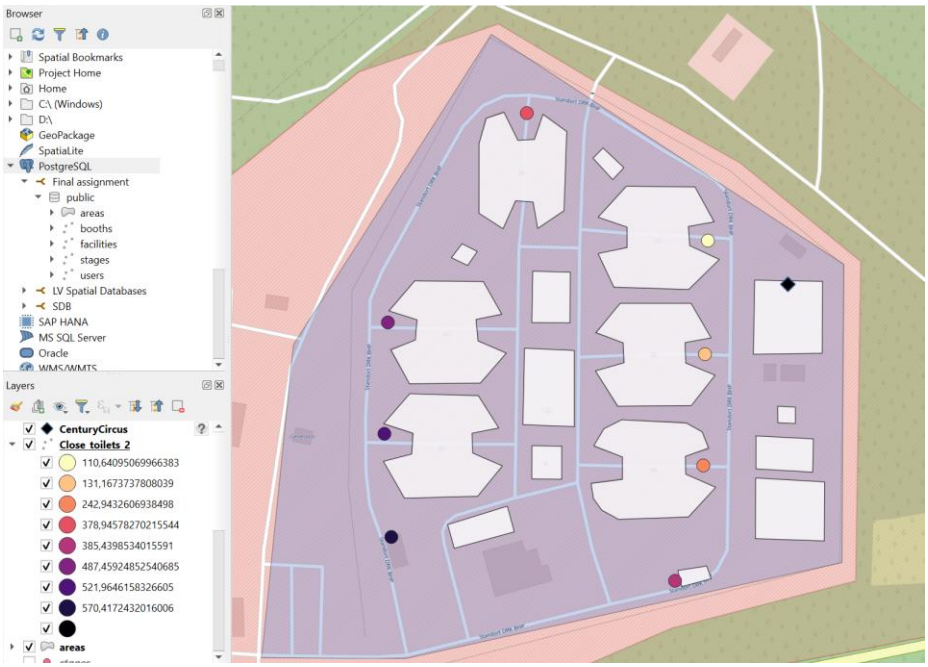

Figure 8. Result showing the toilets (facilities) near the *CenturyCircus* stage in QGIS.

Furthermore, I wanted to test the interaction with dynamic user requests. To do this, I created two users who are at a specific location at a certain time. Let's assume we have two friends who eat something at the *DeepHouse_Döner* before going to see *OGUZ* at the *HexTechnomovement* stage. They both want to know if they will make it to the gig in time. Person 1 has finished his food before person 2, so he has more time to get to the stage. To determine the time the user needs to get to their destination, we assume they are walking 1.4 m/s.

```
INSERT INTO users (user_id, current_location, timestamp)
VALUES ('1', ST_SetSRID(ST_MakePoint(826620.8964438796, 6453781.233965318), 3857),
TIMESTAMP '2025-08-02 01:24:00');

INSERT INTO users (user_id, current_location, timestamp)
VALUES ('2', ST_SetSRID(ST_MakePoint(826620.8964438796, 6453781.233965318), 3857),
TIMESTAMP '2025-08-02 01:27:00');
```

```
--Query for user 1
WITH user_location AS (
    SELECT current_location, timestamp FROM users WHERE user_id = 1
),
event_details AS (
    SELECT
        e.event_name,
        e.stage_name,
        s.geom AS stage_geom,
        ST_Distance(u.current_location, s.geom) AS distance_meters,
        e.start_time AS event_start_time,
        u.timestamp AS user_timestamp
    FROM timetable_stages e
    JOIN stages s ON e.stage_id = s.stage_id
    CROSS JOIN user_location u
    WHERE e.event_name = 'OGUZ'
)
SELECT *,
    (distance_meters / 1.4) AS estimated_travel_seconds,
    event_start_time - user_timestamp AS time_until_event,
    CASE
        WHEN (distance_meters / 1.4) < EXTRACT(EPOCH FROM (event_start_time -
user_timestamp))
        THEN '☑ User will make it'
        ELSE '❌ User will not make it'
    END AS arrival_status
FROM event_details;
```

Code Block 7. Dynamic SQL query showing if a user will arrive on time for a performance

After running this query, it becomes clear that user 1, who has finished their food before user 2, will be able to reach the performance on time. User 2, on the other hand, only has 3 minutes until the start of the performance. He must therefore be faster than 1.4 m/s to make it in time.

User 1:

	distance_meters double precision	event_start_time timestamp without time zone	user_timestamp timestamp without time zone	estimated_travel_seconds double precision	time_until_event interval	arrival_status text
1	267.28529664859224	2025-08-02 01:30:00	2025-08-02 01:24:00	190.91806903470876	00:06:00	☑ User will make it

User 2:

	distance_meters double precision	event_start_time timestamp without time zone	user_timestamp timestamp without time zone	estimated_travel_seconds double precision	time_until_event interval	arrival_status text
1	267.28529664859224	2025-08-02 01:30:00	2025-08-02 01:27:00	190.91806903470876	00:03:00	❌ User will not make it

Figure 9. Result showing if users 1 & 2 will make it to the selected performance on time.

Conclusion

In conclusion, the created database fulfills the necessary criteria to serve as a backend for a WebGIS. It organizes booths, facilities and stages in different areas. In addition, it allows for user-specific dynamic queries that display facilities near a visitor's current position, as well as the use of a user's last timestamp to link them to current events. To ensure that the database works efficiently, it was built in the third normal form. At the same time, views were created for easy access to important data, as well as indexes where required.

To improve the database even further, more data could be added to the database, especially in relation to the *CampingVillage*, since it is an important part of the festival and is currently only shown as an area without facilities, booths or stages. At the same time, more indexes and views could be created as needed to improve the performance of the database. Finally, the dynamic user information could be improved, since it currently only contains two examples, but is an important part of the database, especially with regard to the creation of the web app.