

Databázové systémy

Semestrální práce

Model: informační systém pro správu veřejných událostí v ČR

Složení týmu: Ivan Shestachenko, Daniil Sofin

Jméno cvičícího: Ing. Martin Řimnáč, Ph.D.

Obsah

Popis systému.....	2
Konceptuální model.....	3
Relační model.....	5
ER model.....	6
Generace tabulek - SQL.....	7
Naplnění tabulek testovacími daty - ukázka.....	10
SQL dotazy na data.....	13
Ilustrace použití transakcí - příklad 1.....	17
Ilustrace použití transakcí - příklad 2.....	18
Vytvoření a použití triggeru.....	19
Vytvoření a použití pohledu.....	21
Vytvoření a použití indexu.....	22

Popis systému

V rámci této semestrální práce popisujeme informační systém webové platformy pro vyhledávání, organizování a hodnocení veřejných událostí v ČR. Umožňuje uživatelům objevovat nové události, přidávat recenze a spravovat vlastní organizované akce.

Uživatelé a osobní údaje

V systému existují dva typy uživatelů – **návštěvníci** a **organizátoři**. Uživatel může být zároveň organizátorem i návštěvníkem. Každý uživatel si může (ale nemusí) uložit své **osobní údaje**, jako je jméno, příjmení, datum narození a telefonní číslo. Přihlašuje se pomocí **loginu/emailu** a **hesla**.

Události a jejich správa

Každá událost (**event**) má jedinečný **název**, **kapacitu**, **cenu vstupu**, **čas začátku a konce** a **popis**. Události se konají na určitém **místě**, které je specifikováno adresou (**město, ulice, číslo popisné**). Událost musí být organizována alespoň jedním organizátorem, ale může mít i více organizátorů.

Kategorizace událostí

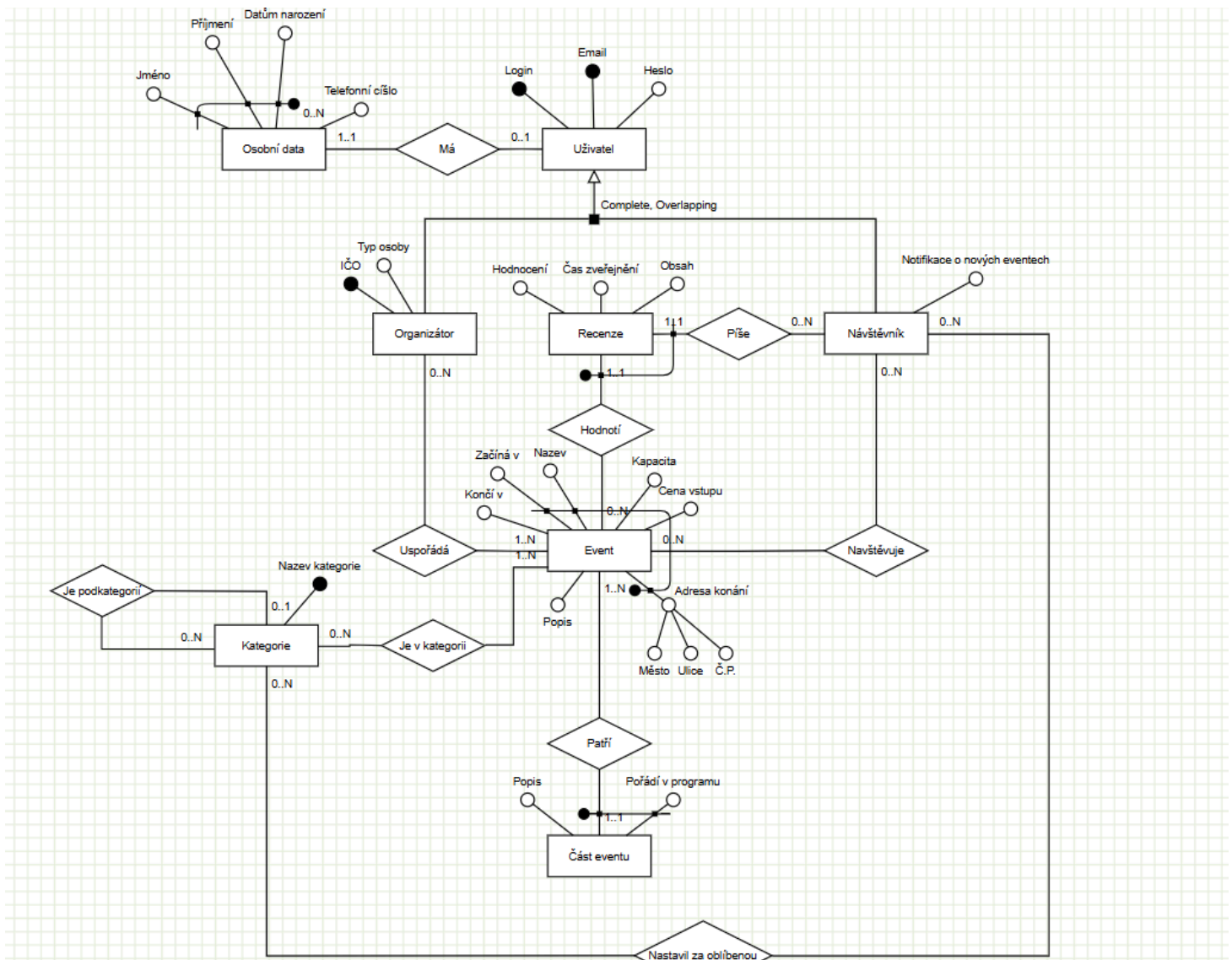
Každá událost patří do jedné nebo více **kategorií** (např. koncerty, sport, konference). Kategorie mohou mít **podkategorie**, což umožňuje podrobnější třídění akcí.

Interakce uživatelů s událostmi

Návštěvníci mohou v systému:

- **Navštěvovat události** (označovat akce, na kterých byli/budou);
- **Psát recenze** s hodnocením, časem zveřejnění a obsahem;
- **Označovat kategorie událostí jako oblíbené**, aby je měli snadno k dispozici;
- **Dostávat notifikace o nových událostech v oblíbených kategoriích** (pokud si uživatel uloží v nastavení, že chce takové notifikace dostávat - atribut uživatele **Notifikace o nových eventech**)

Konceptuální model



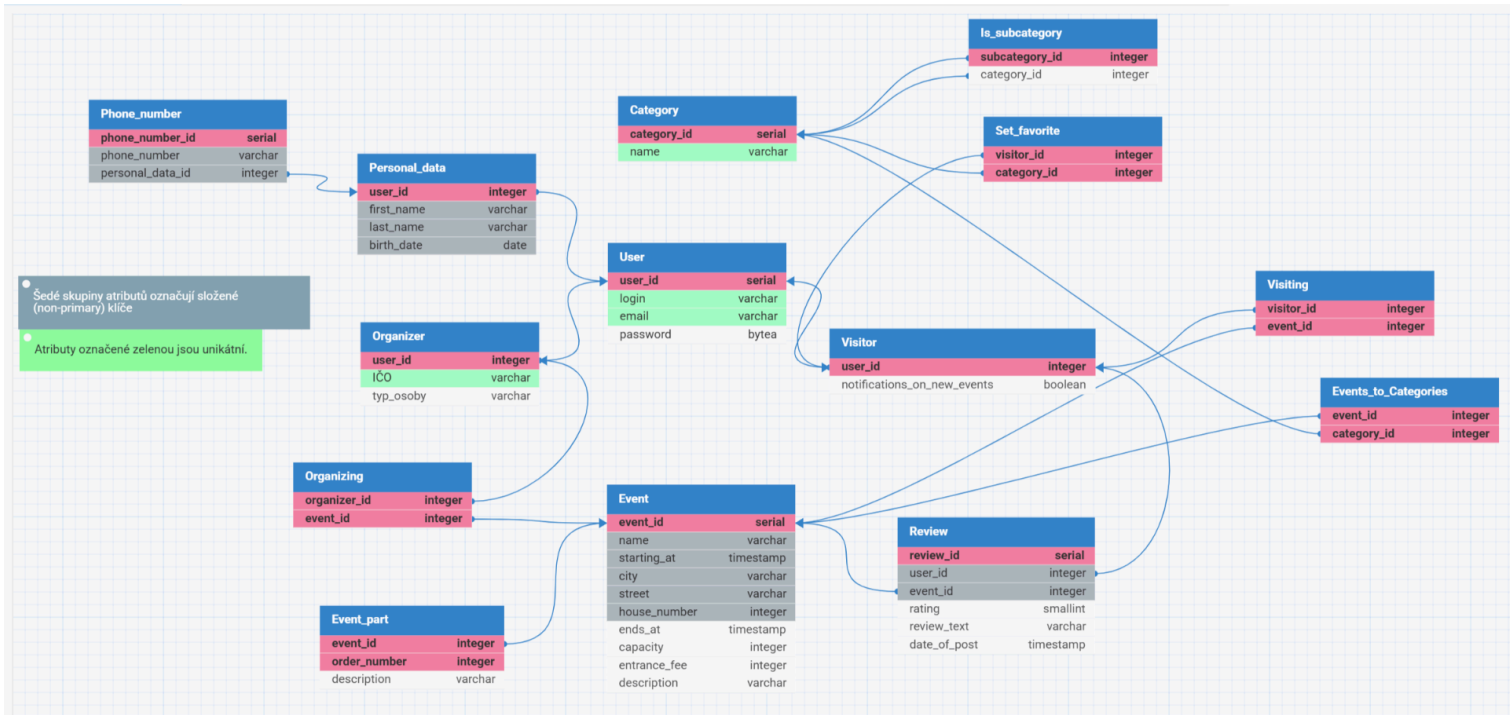
Model zahrnuje:

- Dědičnost (Uživatel ➔ Organizátor, Návštěvník);
- Mezentitní vazby se všemi možnými kardinalitami;
- Reflexivní vazbu (Kategorie je podkategorií);
- Slabý entitní typ (Část eventu).

Relační model

- Uživatel(login, email, heslo)
- Osobní_data(jméno, příjmení, datum_narození, login)
 - FK: (login) \subseteq Uživatel(login)
- Telefonní_číslo(telefonní_číslo, login)
 - FK: (login) \subseteq Osobní_data(login)
- Organizátor(IČO, login, typ_osoby)
 - FK: (login) \subseteq Uživatel(login)
- Návštěvník(login, notifikace_o_nových_eventech)
 - FK: (login) \subseteq Uživatel(login)
- Event(nazev, začíná_v, město, ulice, č.p., končí_v, kapacita, cena_vstupu, popis)
- Recenze(login, nazev, začíná_v, město, ulice, č.p., hodnocení, čas_zveřejnění, obsah)
 - FK: (login) \subseteq Návštěvník(login)
 - FK: (nazev, začíná_v, město, ulice, č.p.) \subseteq Event(nazev, začíná_v, město, ulice, č.p.)
- Uspořádá(login, začíná_v, nazev, město, ulice, č.p.)
 - FK: (login) \subseteq Organizátor(login)
 - FK: (začíná_v, nazev, město, ulice, č.p.) \subseteq Event(začíná_v, nazev, město, ulice, č.p.)
- Navštěvuje(login, začíná_v, nazev, město, ulice, č.p.)
 - FK: (login) \subseteq Návštěvník(login)
 - FK: (začíná_v, nazev, město, ulice, č.p.) \subseteq Event(začíná_v, nazev, město, ulice, č.p.)
- Kategorie(nazev_kategorie)
- Je_v_kategorii(nazev_kategorie, začíná_v, nazev, město, ulice, č.p.)
 - FK: (nazev_kategorie) \subseteq Kategorie(nazev_kategorie)
 - FK: (začíná_v, nazev, město, ulice, č.p.) \subseteq Event(začíná_v, nazev, město, ulice, č.p.)
- Nastavil_za_oblíbenou(login, nazev_kategorie)
 - FK: (login) \subseteq Návštěvník(login)
 - FK: (nazev_kategorie) \subseteq Kategorie(nazev_kategorie)
- Je_podkategorií(podkategorie, kategorie)
 - FK: (podkategorie) \subseteq Kategorie(nazev_kategorie)
 - FK: (kategorie) \subseteq Kategorie(nazev_kategorie)
- Část_eventu(pořadí_v_programu, začíná_v, nazev, město, ulice, č.p., popis)
 - FK: (začíná_v, nazev, město, ulice, č.p.) \subseteq Event(začíná_v, nazev, město, ulice, č.p.)

ER model



Generace tabulek - SQL

```
CREATE TABLE IF NOT EXISTS "User" (  
    "user_id" serial,  
    "login" varchar(255) NOT NULL UNIQUE,  
    "email" varchar(255) NOT NULL UNIQUE,  
    "password" bytea NOT NULL,  
    PRIMARY KEY ("user_id")  
);  
  
CREATE TABLE IF NOT EXISTS "Personal_data" (  
    "user_id" bigint,  
    "first_name" varchar(50) NOT NULL,  
    "last_name" varchar(50) NOT NULL,  
    "birth_date" date NOT NULL,  
    PRIMARY KEY ("user_id"),  
    UNIQUE ("first_name", "last_name", "birth_date"),  
    CONSTRAINT "Personal_data_fk4" FOREIGN KEY ("user_id") REFERENCES "User"("user_id")  
ON DELETE CASCADE  
);  
  
CREATE TABLE IF NOT EXISTS "Phone_number" (  
    "phone_number_id" serial,  
    "phone_number" varchar(20) NOT NULL,  
    "personal_data_id" bigint NOT NULL,  
    PRIMARY KEY ("phone_number_id"),  
    UNIQUE ("phone_number", "personal_data_id"),  
    CONSTRAINT "Phone_number_fk2" FOREIGN KEY ("personal_data_id") REFERENCES  
"Personal_data"("user_id") ON DELETE CASCADE  
);  
  
CREATE TABLE IF NOT EXISTS "Organizer" (  
    "user_id" bigint,  
    "IČO" varchar(255) NOT NULL UNIQUE,  
    "typ_osoby" varchar(255) NOT NULL,  
    PRIMARY KEY ("user_id"),  
    CONSTRAINT "Organizer_fk0" FOREIGN KEY ("user_id") REFERENCES "User"("user_id") ON  
DELETE CASCADE  
);  
  
CREATE TABLE IF NOT EXISTS "Visitor" (  
    "user_id" bigint,  
    "notifications_on_new_events" boolean NOT NULL,  
    PRIMARY KEY ("user_id"),  
    CONSTRAINT "Visitor_fk0" FOREIGN KEY ("user_id") REFERENCES "User"("user_id") ON  
DELETE CASCADE);
```

```

CREATE TABLE IF NOT EXISTS "Category" (
    "category_id" serial,
    "name" varchar(255) NOT NULL UNIQUE,
    PRIMARY KEY ("category_id")
);

CREATE TABLE IF NOT EXISTS "Set_favorite" (
    "visitor_id" bigint,
    "category_id" bigint,
    PRIMARY KEY ("visitor_id", "category_id"),
    CONSTRAINT "Set_favorite_fk1" FOREIGN KEY ("visitor_id") REFERENCES
"Visitor"("user_id"),
    CONSTRAINT "Set_favorite_fk2" FOREIGN KEY ("category_id") REFERENCES
"Category"("category_id") ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS "Event_part" (
    "event_id" bigint NOT NULL,
    "order_number" bigint NOT NULL,
    "description" varchar(255) NOT NULL,
    PRIMARY KEY ("event_id", "order_number"),
    CONSTRAINT "Event_part_fk1" FOREIGN KEY ("event_id") REFERENCES "Event"("event_id")
ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS "Is_subcategory" (
    "subcategory_id" bigint,
    "category_id" bigint NOT NULL,
    PRIMARY KEY ("subcategory_id"),
    CONSTRAINT "Is_subcategory_fk1" FOREIGN KEY ("subcategory_id") REFERENCES
"Category"("category_id") ON DELETE CASCADE,
    CONSTRAINT "Is_subcategory_fk2" FOREIGN KEY ("category_id") REFERENCES
"Category"("category_id") ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS "Visiting" (
    "visitor_id" bigint,
    "event_id" bigint,
    PRIMARY KEY ("visitor_id", "event_id"),
    CONSTRAINT "Visiting_fk1" FOREIGN KEY ("visitor_id") REFERENCES
"Visitor"("user_id"),
    CONSTRAINT "Visiting_fk2" FOREIGN KEY ("event_id") REFERENCES "Event"("event_id")
);

```



```

CREATE TABLE IF NOT EXISTS "Event" (
    "event_id" serial,
    "name" varchar(255) NOT NULL,
    "starting_at" timestamp with time zone NOT NULL,
    "city" varchar(255) NOT NULL,
    "street" varchar(255) NOT NULL,
    "house_number" bigint NOT NULL,
    "ends_at" timestamp with time zone NOT NULL,
    "capacity" bigint NOT NULL,
    "entrance_fee" bigint NOT NULL,
    "description" varchar(255),
    PRIMARY KEY ("event_id"),
    UNIQUE ("name", "starting_at", "city", "street", "house_number")
);

CREATE TABLE IF NOT EXISTS "Events_to_Categories" (
    "event_id" bigint,
    "category_id" bigint,
    PRIMARY KEY ("event_id", "category_id"),
    CONSTRAINT "Events_to_Categories_fk1" FOREIGN KEY ("event_id") REFERENCES
"Event"("event_id"),
    CONSTRAINT "Events_to_Categories_fk2" FOREIGN KEY ("category_id") REFERENCES
"Category"("category_id")
);

CREATE TABLE IF NOT EXISTS "Organizing" (
    "organizer_id" bigint,
    "event_id" bigint,
    PRIMARY KEY ("organizer_id", "event_id"),
    CONSTRAINT "Organizing_fk1" FOREIGN KEY ("organizer_id") REFERENCES
"Organizer"("user_id"),
    CONSTRAINT "Organizing_fk2" FOREIGN KEY ("event_id") REFERENCES "Event"("event_id")
);

CREATE TABLE IF NOT EXISTS "Review" (
    "review_id" serial,
    "user_id" bigint NOT NULL,
    "event_id" bigint NOT NULL,
    "rating" smallint NOT NULL,
    "review_text" varchar(255),
    "date_of_post" timestamp with time zone NOT NULL,
    PRIMARY KEY ("review_id"),
    UNIQUE ("user_id", "event_id"),
    CONSTRAINT "Review_fk1" FOREIGN KEY ("user_id") REFERENCES "Visitor"("user_id"),
    CONSTRAINT "Review_fk2" FOREIGN KEY ("event_id") REFERENCES "Event"("event_id")
);

```

Naplnění tabulek testovacími daty - ukázka

```
CREATE EXTENSION IF NOT EXISTS pgcrypto; --to encrypt generated passwords with crypt()
-- Generate rows for "User" table
WITH first_names AS (
    SELECT unnest(ARRAY[
        'James','Mary','John','Patricia','Robert',
        --- 190 more unique first names ---
        'Jennifer','Michael','Linda','William','Elizabeth',
    ]) AS first_name
),
last_names AS (
    SELECT unnest(ARRAY[
        'Smith','Johnson','Williams','Brown','Jones',
        --- 190 more second names ---
        'Garcia','Miller','Davis','Rodriguez','Martinez',
    ]) AS last_name
),
domains AS (
    SELECT unnest(ARRAY['example.com', 'gmail.com', 'seznam.cz', 'hotmail.com']) AS
domain
),
name_combinations AS (
    SELECT
        row_number() OVER () AS rn,
        first_name,
        last_name
    FROM first_names, last_names
    LIMIT 32000
),
final_data AS (
    SELECT
        rn,
        initcap(first_name) || initcap(last_name) || rn AS login,
        lower(first_name) || '.' || lower(last_name) || rn || '@' ||
        (SELECT domain FROM domains ORDER BY random() LIMIT 1) AS email,
        convert_to(
            crypt(substring(md5(random()::text), 1, 16), gen_salt('bf')),
            'UTF8'
        ) AS password --generates random 16-symbol password, encrypts it, encodes to bytes
    FROM name_combinations
)
INSERT INTO "User" ("login", "email", "password")
SELECT login, email, password FROM final_data;
```

```

-- Generate rows for "Personal_data" table
WITH user_base AS (
    SELECT
        "user_id",
        substring("login" FROM '^[A-Z][a-z]+') AS first_name,
        substring("login" FROM '^[A-Z][a-z]+([A-Z][a-z]+)') AS last_name
    FROM "User"
),
birth_dates AS (
    SELECT generate_series('1960-01-01'::date, '2009-12-31'::date, interval '1 day') AS
    birth_date
),
sampled_users AS (
    SELECT
        ub.*,
        row_number() OVER () AS rn
    FROM user_base ub
    ORDER BY random()
    LIMIT 16000 -- only half of the users will have personal data
),
cycled_birth_dates AS (
    SELECT
        birth_date,
        row_number() OVER () AS rn
    FROM birth_dates
),
personal_data_ready AS (
    SELECT
        su.user_id,
        su.first_name,
        su.last_name,
        cbd.birth_date
    FROM sampled_users su
    JOIN cycled_birth_dates cbd ON cbd.rn = su.rn
)
INSERT INTO "Personal_data" ("first_name", "last_name", "birth_date", "user_id")
SELECT first_name, last_name, birth_date, user_id
FROM personal_data_ready;

```

```

-- Generate rows for "Phone_number" table
WITH phone_codes AS (
    SELECT unnest(ARRAY['+420', '+421', '+41', '+33', '+31']) AS code
),
personal_ids AS (
    SELECT user_id FROM "Personal_data"
),
number_counts AS (
    SELECT
        user_id,
        (floor(random() * 3))::int AS num_phones
    FROM personal_ids
),
replicated_ids AS (
    SELECT user_id
    FROM number_counts, generate_series(1, num_phones)
),
random_codes AS (
    SELECT
        code,
        row_number() OVER () AS rn
    FROM phone_codes
),
phone_data AS (
    SELECT
        (
            (SELECT code FROM random_codes ORDER BY random() LIMIT 1) ||
            lpad((trunc(random() * 1e9))::text, 9, '0')
        ) AS phone_number,
        user_id
    FROM replicated_ids
)
INSERT INTO "Phone_number" ("phone_number", "user_id")
SELECT phone_number, user_id
FROM phone_data;

--Generate data for "Category" table
WITH category_list AS (
    SELECT unnest(ARRAY[
        'Career&Education', 'Career Expos', 'Live Conferences', 'Open Days',
        'Culture&Art', 'Music', 'Concerts', 'Live Performances', 'Theatre',
        'Museums&Galleries', 'Cinema', 'Festivals',
        'Shopping', 'Fairs', 'Holiday Sales',
        'Sports', 'Football', 'Ice Hockey', 'Basketball'
    ]) AS name
)
INSERT INTO "Category" ("name")
SELECT name
FROM category_list;

```

SQL dotazy na data

1. Vnější spojení tabulek

```
722 SELECT pd.user_id, pd.first_name, pd.last_name, pn.phone_number
723 FROM "Personal_data" pd
724 LEFT JOIN "Phone_number" pn
725 ON pd.user_id = pn.personal_data_id;
```

Results Messages

	user_id	first_name	last_name	phone_number
6	6	James	Garcia	+33122593261
7	8	James	Davis	+33015327023
8	10	James	Martinez	+33937789596
9	10	James	Martinez	+33071835777
10	11	James	Hernandez	+33006806223
11	11	James	Hernandez	+33178429545
12	12	James	Lopez	+33319402587

Dotaz vrací všechny záznamy z tabulky "Personal_data" spolu s telefonními čísly z tabulky "Phone_number". Používá se LEFT JOIN.

2. Vnitřní spojení tabulek

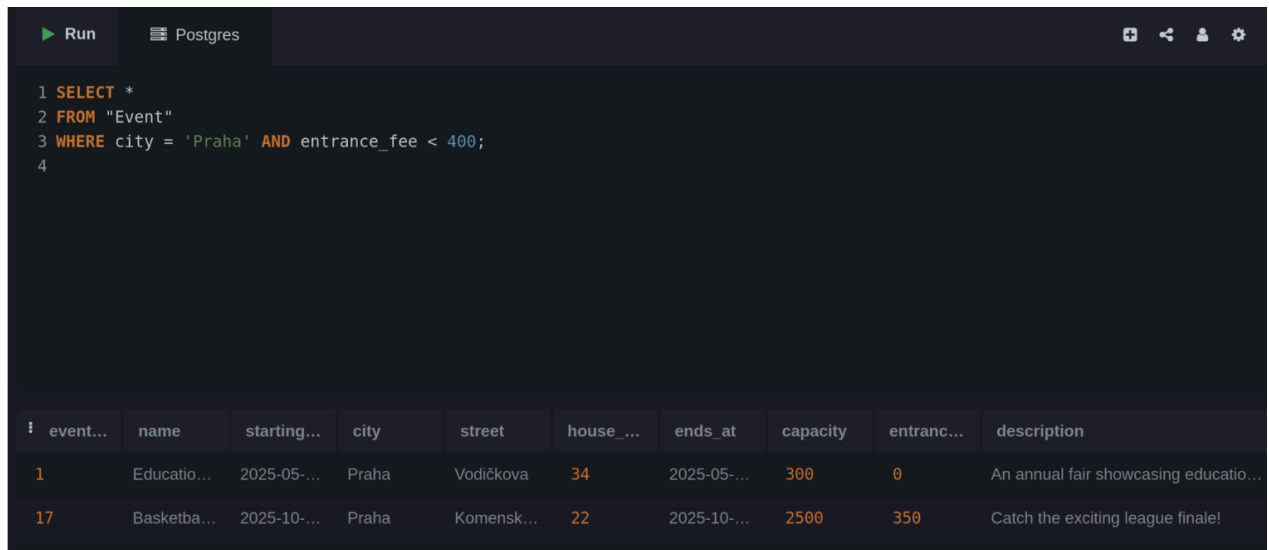
```
716 SELECT
717     u.user_id,
718     u.login,
719     e.name
720 FROM "Visiting" v
721 INNER JOIN "User" u ON v.visitor_id = u.user_id
722 INNER JOIN "Event" e ON v.event_id = e.event_id
723 ORDER BY u.user_id;
```

Results Messages

	user_id	login	name
39	84	JamesWood84	Basketball Championship Final
40	84	JamesWood84	Runners United 10K
41	85	JamesJames85	Vintage Market Day
42	87	JamesGray87	Gallery Night Tour
43	87	JamesGray87	CrossFit Nationals
44	88	JamesMendoza88	National Ice Hockey Cup
45	90	JamesHughes90	Contemporary Art Exhibition
46	90	JamesHughes90	International Football Friendly
47	91	JamesPrice91	Spring Fair Extravaganza

Dotaz vrací všechny návštěvy všech návštěvníků, seřazené podle uživatelského ID.

3. Podmínka na data



The screenshot shows a PostgreSQL query editor with a dark theme. At the top, there are buttons for 'Run' and 'Postgres'. The SQL query is as follows:

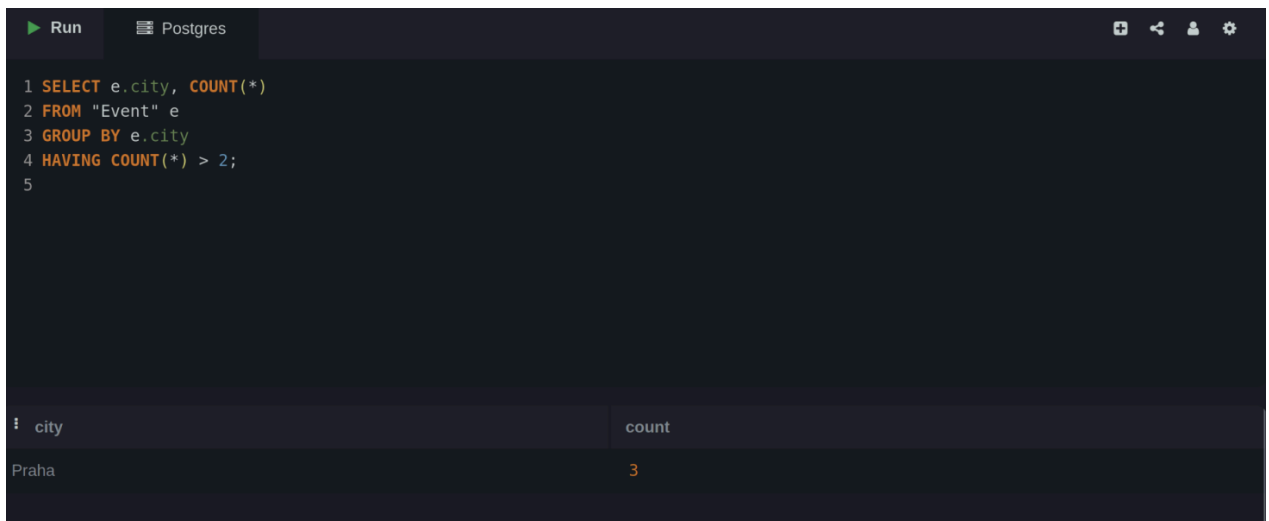
```
1 SELECT *
2 FROM "Event"
3 WHERE city = 'Praha' AND entrance_fee < 400;
4
```

Below the query, the results are displayed in a table with 10 columns: event..., name, starting..., city, street, house..., ends_at, capacity, entranc..., and description. The first two rows are visible:

event...	name	starting...	city	street	house...	ends_at	capacity	entranc...	description
1	Educatio...	2025-05-...	Praha	Vodičkova	34	2025-05-...	300	0	An annual fair showcasing educatio...
17	Basketba...	2025-10-...	Praha	Komensk...	22	2025-10-...	2500	350	Catch the exciting league finale!

Dotaz vybere všechny akce, které se konají v Praze a mají vstupné nižší než 400 Kč.

4. Agregace a podmínka na hodnotu agregační funkce



The screenshot shows a PostgreSQL query editor with a dark theme. At the top, there are buttons for 'Run' and 'Postgres'. The SQL query is as follows:

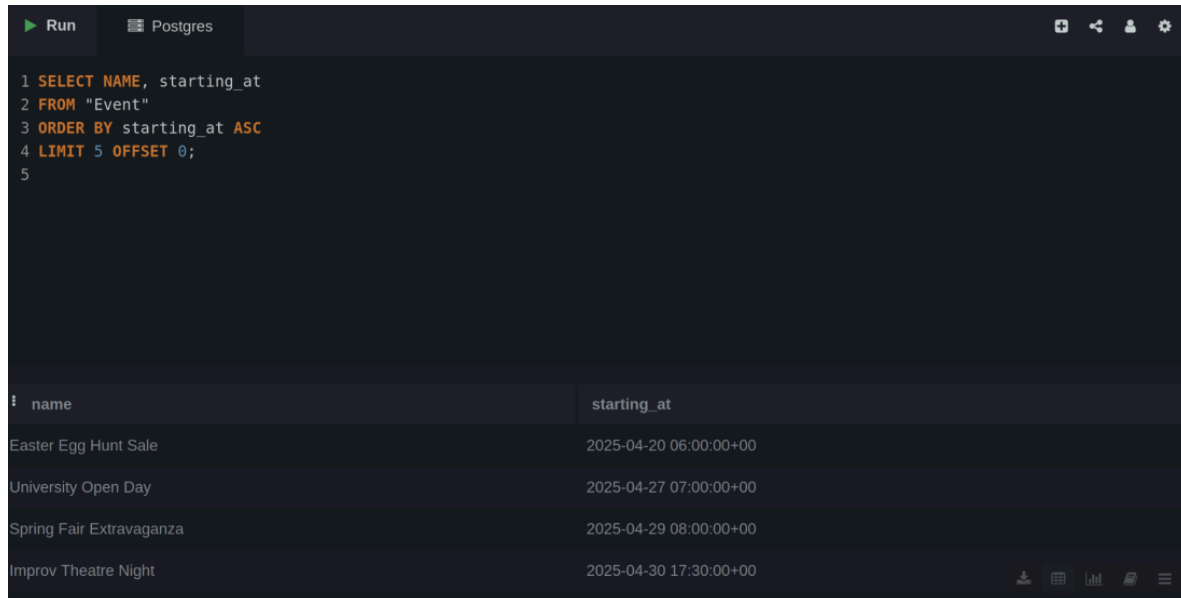
```
1 SELECT e.city, COUNT(*)
2 FROM "Event" e
3 GROUP BY e.city
4 HAVING COUNT(*) > 2;
5
```

Below the query, the results are displayed in a table with 2 columns: city and count. The first row is visible:

city	count
Praha	3

Tento dotaz zobrazí pouze města, kde se konalo více než 2 akce.

5. Řazení a stránkování (ORDER BY, LIMIT, OFFSET)



The screenshot shows a PostgreSQL query editor with a dark theme. The query is as follows:

```
1 SELECT NAME, starting_at
2 FROM "Event"
3 ORDER BY starting_at ASC
4 LIMIT 5 OFFSET 0;
5
```

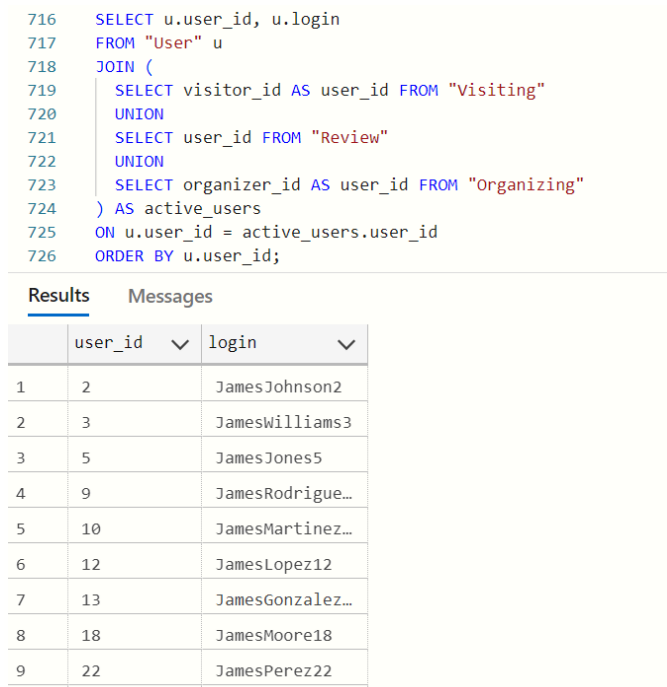
Below the query, the results are displayed in a table with two columns: 'name' and 'starting_at'.

name	starting_at
Easter Egg Hunt Sale	2025-04-20 06:00:00+00
University Open Day	2025-04-27 07:00:00+00
Spring Fair Extravaganza	2025-04-29 08:00:00+00
Improv Theatre Night	2025-04-30 17:30:00+00

Zobrazí prvních 5 ze všech nejbližších akcí (řazených podle data začátku).

OFFSET se pak v praxi často používá ke stránkování výsledků.

6. Množinové operace - UNION



The screenshot shows a PostgreSQL query editor with a dark theme. The query is as follows:

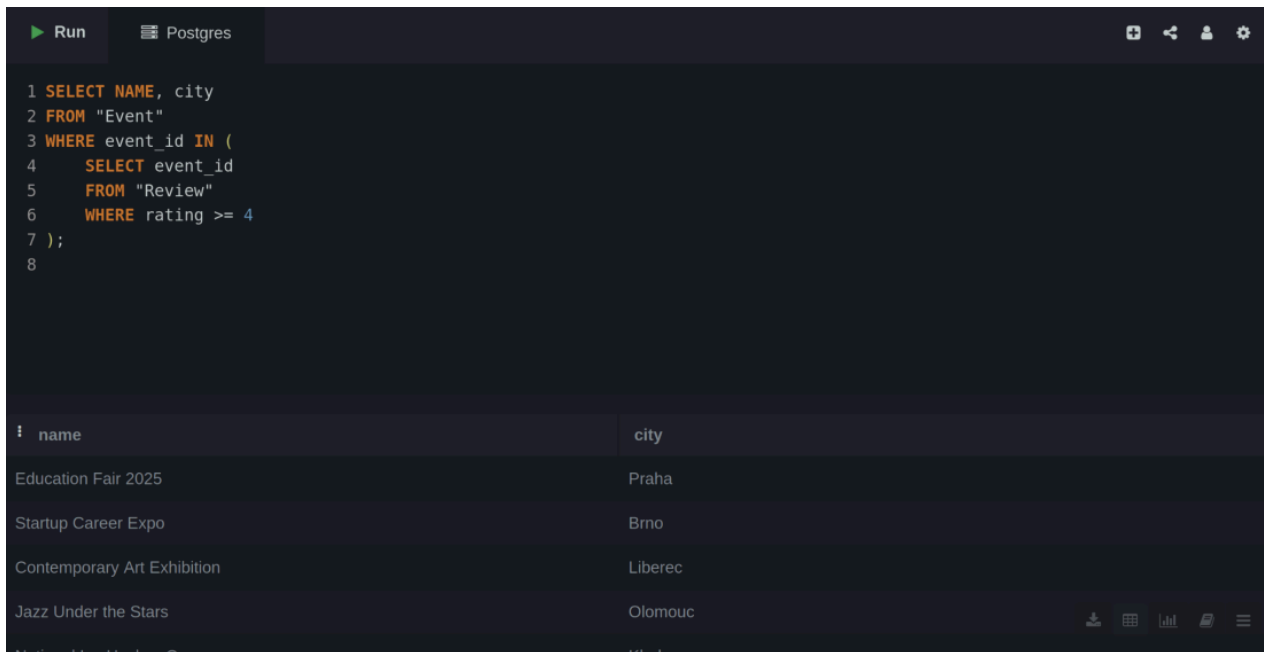
```
716 SELECT u.user_id, u.login
717 FROM "User" u
718 JOIN (
719     SELECT visitor_id AS user_id FROM "Visiting"
720     UNION
721     SELECT user_id FROM "Review"
722     UNION
723     SELECT organizer_id AS user_id FROM "Organizing"
724 ) AS active_users
725 ON u.user_id = active_users.user_id
726 ORDER BY u.user_id;
```

Below the query, the results are displayed in a table with two columns: 'user_id' and 'login'.

	user_id	login
1	2	JamesJohnson2
2	3	JamesWilliams3
3	5	JamesJones5
4	9	JamesRodrigue...
5	10	JamesMartinez...
6	12	JamesLopez12
7	13	JamesGonzalez...
8	18	JamesMoore18
9	22	JamesPerez22

Dotaz vybere všechny uživatele, kteří jakýmkoliv způsobem interagovali s platformou - organizovali eventy, zaznamenávali své návštěvy eventů, psali recenze na eventy. Union odstraní duplicitní záznamy ze sjednoceného výsledku dotazů na tabulky návštěv, recenzí a uspořádání eventů.

7. Vnořený SELECT



```
1 SELECT name, city
2 FROM "Event"
3 WHERE event_id IN (
4     SELECT event_id
5     FROM "Review"
6     WHERE rating >= 4
7 );
8
```

name	city
Education Fair 2025	Praha
Startup Career Expo	Brno
Contemporary Art Exhibition	Liberec
Jazz Under the Stars	Olomouc
National Ice Hockey Cup	Kladno

Dotaz zobrazí všechny eventy, které získaly alespoň jednu recenzi s hodnocením 4 nebo více.

8. Definování oprávnění (GRANT)

```
GRANT SELECT, INSERT ON TABLE "Review" TO shestiva;
```

Umožnit kolegovi (Ivanu Shestachenko) číst (SELECT) a vkládat (INSERT) nové záznamy do tabulky Review.

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO shestiva;
```

Zajistit oprávnění na všechny operace na úrovni schématu nad všemi *aktuálně existujícími* tabulkami tohoto schématu pro uživatele.

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT ALL PRIVILEGES ON TABLES TO shestiva;
```

Zajistit oprávnění na všechny operace na úrovni schématu nad všemi tabulkami tohoto schématu, *včetně těch dosud nedefinovaných*, pro uživatele.

```
GRANT ALL PRIVILEGES ON DATABASE db_events TO shestiva;
```

Zajistit oprávnění na všechny operace na úrovni databáze nad celou databází pro uživatele.

```
ALTER ROLE shestiva WITH SUPERUSER;
```

Zajistit kompletní administrátorské oprávnění (root-přístup na úrovni PostgreSQL serveru) - nejrozsáhlejší oprávnění ze všech možných, pouze pro ověřené uživatele.

Ilustrace použití transakcí - příklad 1.

Uvažujeme-li transakci úrovně izolace Repeatable Read:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

WITH prague_events_entry_under_400 AS (
    SELECT * FROM "Event" WHERE city = 'Praha' AND entrance_fee < 400
)
SELECT name, capacity
FROM "Event"
WHERE event_id IN (SELECT event_id FROM prague_events_entry_under_400);

COMMIT;
```

A paralelní transakci:

```
BEGIN;

WITH prague_events_entry_under_400 AS (
    SELECT event_id, name, capacity
    FROM "Event"
    WHERE city = 'Praha' AND entrance_fee < 400
    ORDER BY event_id
    LIMIT 1
)
UPDATE "Event"
SET entrance_fee = entrance_fee + 500
WHERE event_id IN (SELECT event_id FROM prague_events_entry_under_400);

COMMIT;
```

Tímto příkladem ilustrujeme ochranu proti konfliktu: Non-repeatable read.

V rámci první transakce provádíme dva selecty, které dotazují na stejné řádky v tabulce Event. Předpokládá se, že mezi provedením těchto dvou selectů v rámci první transakce, druhá paralelní transakce mění určitou hodnotu na jednom z řádků, dotazovaných v první transakci. V takovém případě dostáváme různá data z identických selectů v rámci první transakce, což by prošlo při defaultní úrovni izolace první transakce (Read Committed) a zřejmě by způsobilo nekonzistentní výsledky.

Transakce úrovně izolace Repeatable Read zajišťuje kontrolu, zdali byly uložené (commitnuté) nějaké změny (updaty) hodnot na dotazovaných řádcích paralelními transakcemi v její průběhu - takže v tomto případě by první transakce spadla s příslušným varováním a museli bychom ji spustit znovu. Transakce jako taková taky zajišťuje to, že příkazy v její složení se spouští jedním jednodlým blokem a projdou buď všechny a databáze přejde do dalšího validního stavu, nebo blok příkazu spadne jako celek, a databáze se zůstane v aktuálním validním stavu.

Ilustrace použití transakcí - příklad 2.

Uvažujeme-li transakci úrovně izolace Serializable:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
SELECT COUNT(*) AS current_visits_count
  FROM Visiting
 WHERE event_id = 10;
```

```
SELECT capacity AS event_capacity
  FROM "Event"
 WHERE event_id = 10;
```

-- zde logika v aplikaci: pokud event_capacity > current_visits_count, povolit zápis další návštěvy.

```
INSERT INTO Visiting (visitor_id, event_id)
VALUES (42, 10);
```

```
COMMIT;
```

A paralelní transakci:

```
BEGIN;
```

```
INSERT INTO Visiting (visitor_id, event_id)
VALUES (99, 10);
```

```
COMMIT;
```

Tímto příkladem ilustrujeme ochranu proti konfliktu: Phantom read.

V rámci první transakce provádíme select na aktuální počet uložených návštěv na konkrétní event a select na kapacitu tohoto eventu. V aplikaci probíhá kontrola, zda je počet uložených návštěv menší než kapacita eventu, a na základě výsledku kontroly se povoluje nebo nepovoluje zápis další návštěvy. Předpokládá se, že před vložení další návštěvy v rámci první transakce proběhne paralelní transakce, která také uloží do databáze návštěvu tohoto eventu.

V takovém případě, při úrovni izolace první transakce slabším než Serializable, pokud by tř. skutečně na tento event zbývalo jediné volné místo, uložily by se na něj dvě návštěvy, jelikož porovnání počtu návštěv a kapacity eventu v první transakci by probíhalo na základě zastaralých dat.

Transakce úrovně Serializable ale zajišťuje ochranu proti konfliktům tohoto typu tím, že při její spuštění se dělá snapshot databáze, který se porovnává při commitu s aktuálním stavem databáze, a při tomto porovnání jsou odhalitelné i změny počtu řádků, které vyhovují podmínkám v selectech této transakce, čímž je definován konflikt Phantom read.

Vytvoření a použití triggeru

Za základ pro trigger bylo zvoleno automatické přepočítání průměrného hodnocení na každý event podle nových recenzí návštěvníků.

1) Vytvoření příslušné tabulky:

```
CREATE TABLE IF NOT EXISTS "Event_rating" (  
    "event_id" bigint PRIMARY KEY,  
    "average_rating" NUMERIC(3,2) CHECK ("average_rating" >= 1 AND "average_rating" <= 5) NOT NULL  
    "review_count" integer NOT NULL DEFAULT 1,  
    CONSTRAINT "Event_rating_fk1" FOREIGN KEY ("event_id") REFERENCES  
    "Event"("event_id") ON DELETE CASCADE);
```

numeric(3,2) označuje formát povolených hodnot - trojčíslné, 2 desetinná místa.

check (...) označuje omezení intervalu možných hodnot v tomto sloupci - hodnocení návštěvníků je skutečně myšleno jako počet hvězdiček (1 až 5).

2) Vytvoření trigger-funkce na přidání nové recenze na jednotlivý event:

```
CREATE FUNCTION update_event_rating()  
RETURNS TRIGGER AS $$  
BEGIN  
    IF EXISTS (SELECT 1 FROM "Event_rating" WHERE "event_id" = NEW."event_id") THEN  
        UPDATE "Event_rating"  
        SET  
            average_rating = ROUND(((average_rating * review_count) +  
NEW.rating)::numeric / (review_count + 1), 2),  
            review_count = review_count + 1  
        WHERE "event_id" = NEW."event_id";  
    ELSE  
        INSERT INTO "Event_rating" ("event_id", average_rating)  
        VALUES (NEW."event_id", NEW.rating);  
    END IF;  
  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

- Konstrukce \$\$ se používá na odstínění těla funkce v jazyce PL/pgSQL od vnějšího skriptu.
- V rámci funkce kontrolujeme, je-li příslušný event v naší tabulce: pokud ano, přepočítáme jeho hodnotu average_rating s hodnocením aktuálně vkládané recenze a inkrementujeme hodnotu review_count; pokud ne - vkládáme příslušný event a hodnocení z aktuální recenze jako průměrné, hodnota review_count se nastavuje na 1 automaticky tím, jak je definován její sloupec v naší tabulce.

- “return new;” je řádnou syntaktickou konstrukcí pro trigger-funkce, její výsledek v daném případě ale žádným způsobem nepoužíváme, je to after-trigger.

3) Vytvoření samotného triggeru a jeho propojení s tabulkou recenzí a trigger-funkcí:

```
CREATE TRIGGER trg_update_event_rating  
AFTER INSERT ON "Review"  
FOR EACH ROW  
EXECUTE FUNCTION update_event_rating();
```

Zavolání trigger-funkce proběhne po vložení každého řádku (nikoli jen po každé operaci vložení, ty mohou v sobě zahrnovat vložení několika řádků najednou). Data, která byla v tabulce “Review” před přidáním triggeru, nebudou jím zpracované - trigger se spustí pouze na nových vloženích řádků do “Reviews”.

Vytvoření a použití pohledu

1. Pohled se sloučením recenzí a jejich autorů do jedné tabulky pro jednodušší přístup:

```
CREATE VIEW review_with_visitor AS
SELECT
    r.review_id,
    pd.first_name,
    pd.last_name,
    e.name AS event_name,
    r.rating,
    r.review_text,
    r.date_of_post
FROM "Review" r
JOIN "Visitor" v ON r.user_id = v.user_id
JOIN "Personal_data" pd ON v.user_id = pd.user_id
JOIN "Event" e ON r.event_id = e.event_id;
```

```
SELECT * FROM review_with_user WHERE rating >= 4 ORDER BY date_of_post DESC;
```

Vybrat vybrat všechny recenze včetně jejich autorů, s hodnocením výš nebo rovná se 4, s řazením podle data vložení sestupně.

2. Pohled se sloučením eventů a jejich organizátorů do jedné tabulky pro jednodušší přístup:

```
CREATE VIEW organizers_events AS
SELECT
    o.user_id AS organizer_id,
    pd.first_name,
    pd.last_name,
    o."IČO",
    e.event_id,
    e.name AS event_name,
    e.starting_at
FROM "Organizer" o
JOIN "User" u ON o.user_id = u.user_id
JOIN "Personal_data" pd ON u.user_id = pd.user_id
JOIN "Organizing" orgz ON o.user_id = orgz.organizer_id
JOIN "Event" e ON orgz.event_id = e.event_id;

SELECT * FROM organizers_events WHERE starting_at > NOW();
```

Vybrat všechny eventy včetně jejich organizátorů, s datem a časem začátku pozdějším než datum a čas provedení dotazu (té, které dosud nezačaly).

Vytvoření a použití indexu

```
CREATE INDEX index_event_name  
ON Event (name)
```

Tento příkaz vytvoří index na sloupci name v tabulce Event. Indexování tohoto sloupce výrazně zvyšuje výkon dotazů, které filtrují data podle názvu události. Místo toho, aby databáze prohledávala celý obsah tabulky, využije vytvořený index a najde požadované řádky efektivněji.

Například následující dotaz:

```
SELECT * FROM "Event" WHERE name = "some_event_name";
```

bude díky indexu vykonán mnohem rychleji. Hledání hodnoty ve sloupci “name” totiž proběhne pomocí binárního vyhledávání v indexové struktuře (obvykle B-strom), což má logaritmickou časovou složitost, oproti lineárnímu procházení celé tabulky.

Použití indexu je zde velmi vhodné i z hlediska charakteru tabulky Event. Tato tabulka nebude často upravována (nebude vznikat velké množství zápisů), takže overhead aktualizaci indexu bude minimální. Naopak dotazy typu *“vyhledej událost podle názvu”* budou velmi časté.