Напредни бази-Напредна тема
Векторска база на податоци

Нашата цел е со помош на векторска база на податоци да му препорачаме настани за посетување на корисник според тоа што него му се допадлано.

Процесот при работење со векторска база на податоци е следниот. Во една или повеќе од нашите табели(во наш случај една) додаваме колона во која се чуваат векторските embeddings наменети за зачуввување на контекстот кои е извлечен од нашите податоци и уште една колона за самиот текст од кој се создава embeddingot. Тој embedding не ни е овозможен преку pgvector затоа имаме направено backend систем во python кој комуницира со embedding модел со цел да се направат тие векторски embeddings и да се ажурира базата соодветно.

Но прво од кои информации ние всушност го правиме тој embedding и во која табела се додава колоната за embedding.

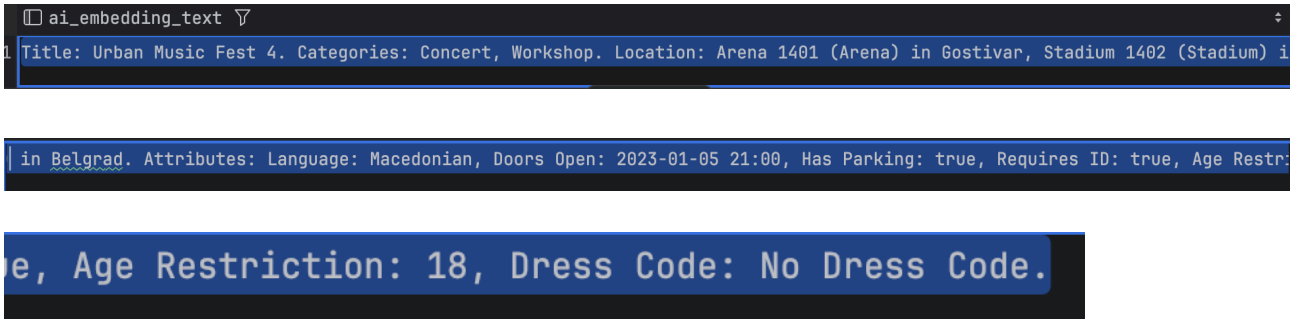
Колоната за тескстот од кои се прави embedding ние ја додаваме во табелата Event, а за тоа кои информации ги земаме за да се направи векторот тоа се:
-Насловот на настанот
-Категории во кои припаѓа настанот
-Локација каде се оджува настанот
-Некои дополнителни атрибути за настанот пример Dress code, јазик, дали има паркинг или нешто слично

Тоа го правиме преку овој view:

```sql
--Зимање на сите податоци(текстуални) кои ке се праќаат на backend и тоа преку view.
CREATE OR REPLACE VIEW public.v_event_embedding_source AS
WITH-- 1. Собирање на сите категории за настанот во една низа (пр. "Concert, Festival")
    event_cats AS (
        SELECT
            ec.event_id,
            STRING_AGG(c.name, ', ') AS category_list
        FROM public.event_category ec
                JOIN public.category c  1..n<->1:  ON ec.category_id = c.id
        GROUP BY ec.event_id
    ),

    -- 2. Собирање на локацијата (преку сесиите и секциите)
    event_locs AS (
        SELECT
            ess.event_id,
            STRING_AGG(DISTINCT l.name || ' (' || lt.type_name || ') in ' || l.city, ', ') AS location_list
        FROM public.event_schedule_session ess
                JOIN public.section s  1..n<->1:  ON ess.section_id = s.section_id
                JOIN public.location l  1..n<->1:  ON s.location_id = l.location_id
                JOIN public.location_type lt  1..n<->1:  ON l.type_id = lt.type_id
        GROUP BY ess.event_id
    ),

    -- 3. Собирање на сите атрибути (Dress Code, Language, итн.)
    event_attrs AS (
        SELECT
            v.event_id,
            STRING_AGG(
                    a.name || ': ' ||
                    COALESCE(v.value_string, v.value_int::text, TO_CHAR(v.value_datetime, 'YYYY-MM-DD HH24:MI'), v.value_bool::text),
                    ', '
            ) AS attribute_list
        FROM public.value v
                JOIN public.attribute a  1..n<->1:  ON v.attribute_id = a.attribute_id
        GROUP BY v.event_id
    )

-- 4. Главно спојување и форматирање на финалниот текст
SELECT
    e.event_id,
    e.title,
    'Title: ' || e.title || '. ' ||
    'Categories: ' || COALESCE(c.category_list, 'Uncategorized') || '. ' ||
    'Location: ' || COALESCE(l.location_list, 'Unknown') || '. ' ||
    'Attributes: ' || COALESCE(a.attribute_list, 'None') || '.' AS ai_embedding_text
FROM public.event e
        LEFT JOIN event_cats c ON e.event_id = c.event_id
        LEFT JOIN event_locs l ON e.event_id = l.event_id
```

Пример за една колона како би личела за даден настан:

```
SELECT ai_embedding_text
FROM v_event_embedding_source
WHERE event_id = 471401;
```



Или попрегледно во еден ред:

```
Title: Urban Music Fest 4. Categories: Concert, Workshop.
Location: Arena 1401 (Arena) in Gostivar, Stadium 1402
(Stadium) in Belgrad. Attributes: Language: Macedonian, Doors
Open: 2023-01-05 21:00, Has Parking: true, Requires ID: true,
Age Restriction: 18, Dress Code: No Dress Code.
```

Вклучување на вектор екстензијата:

```
CREATE EXTENSION IF NOT EXISTS vector;
```

Додавање на колоната за вектор embedding што ке се направи од тесктот сместен во колоната ai_embedding_text.

```
ALTER TABLE public.event ADD COLUMN event_embedding vector(1536);
```

Како што може да видиме на векторот поставувам да има 1536 димензии.

Следно ние за корисникот правиме вектор за неговиот “вкус” и овој вектор го правиме со помош на повеќе фактори.

Ги зимаме векторите на сите настани каде тој оставил оцена(review) 4 или 5. Второ ги зимаме настаните на кој тој всушност отишол т.е каде билетот бил скениран а не откажан или несекениран. Следно ги зимаме настаните на кои покажал голем интерес а ова го правиме така што проверуваме за кои настани бил во waitlist.
Од сите овие вектори ја користиме функцијата AVG() од pgvector за да направиме просек од сите овие вектори.

Ова го правиме преку овој view:

```
CREATE OR REPLACE FUNCTION fn_get_user_taste_vector(p_user_id BIGINT)
    RETURNS vector(1536) AS $$
DECLARE
    v_taste_vector vector(1536);
BEGIN
    WITH user_events AS (
        -- 1. Настани каде корисникот оставил оцена 4 или 5
        SELECT event_id
        FROM public.review
        WHERE user_id = p_user_id AND star_rating >= 4

        UNION

        -- 2. Настани каде корисникот купил билет и навистина отишол (скениран билет)
        SELECT event_id tt.event_id
        FROM public.order_cart oc
                 JOIN public.ticket t 1<->1..n: ON oc.order_id = t.order_id
                 JOIN public.ticket_type tt 1..n<->1: ON t.ticket_type_id = tt.ticket_type_id
        WHERE oc.user_id = p_user_id AND t.is_scanned = TRUE

        UNION

        -- 3. Настани за кои покажал голем интерес (Листа на чекање)
        SELECT event_id ess.event_id
        FROM public.waitlist_entry w
                 JOIN public.event_schedule_session ess 1..n<->1: ON w.event_schedule_session_id = ess.schedule_id
        WHERE w.user_id = p_user_id
    )
    -- Ја користиме вградената AVG() функција на pgvector за да ги споиме сите вектори во еден
    SELECT avg(e.event_embedding) INTO v_taste_vector
    FROM user_events ue
             JOIN public.event e ON ue.event_id = e.event_id
    WHERE e.event_embedding IS NOT NULL;

    RETURN v_taste_vector;
END;
$$ LANGUAGE plpgsql;
```

Финално ја правиме препораката на кои настани да иде корисникот. Ова го правиме со помош на прехтодната функција каде го зимаме векторот за тоа кои настани му се допаѓаат на корисникот. Што доколку е нов корисник? Ако е нов корисник тој логично дека нема вектор за тоа кои настани му се допаѓаат па само му се препорачуваат скори популани настани.

Ако не е нов корисник се извршува хибридно пребарување. Прво проверува тоа колку се слични просечниот вектор на корисникот со векторите на настаните. Ова го прави користејки cosine distance што во pgvector се означува вака <=>. Со помош на cosine distance го зема аголот помеѓу двата вектори тоест поконкретно косинус вредноста на аголот помеѓу векторите. Колку е помал аголот толку векторите се послични. Оваа проверка враќа вредност помеѓу 0 и 2(0-векторите покажуваат во иста насока и аголот е ист и се многу слични, 1-аголот е 90 степени разлика и нема корелација помеѓу аглите, 2-векторите покажуваат на сосема спротивни страни и аголот е 180 степени и векторите не се слични) но ние правиме да има поклопување од 0 до 100% за да е полесно за читање. Потоа се филтрираат за да се покажат настаните кои се само во иднината а исто така и се филтрираат настаните за кои веќе има купено или резервирано билет.

Финално настаните се сортираат според векторите враќа број на настани колку ке спесифицираме преку p_limit.

Ова го правиме со овој view:

```sql
CREATE OR REPLACE FUNCTION fn_get_event_recommendations(p_user_id BIGINT, p_limit INT DEFAULT 5)
    RETURNS TABLE (
                      recommended_event_id INT,
                      event_title VARCHAR,
                      match_score NUMERIC
                  ) AS $$
DECLARE
    v_taste_vector vector(1536);
BEGIN
    -- 1. Го земаме "Векторот на Вкусот" за корисникот (претходната функција)
    v_taste_vector := fn_get_user_taste_vector( p_user_id p_user_id);

    -- 2. Справување со "Cold Start" проблем (Ако корисникот е нов и нема историја)
    IF v_taste_vector IS NULL THEN
        -- Враќаме најскорашни популарни настани бидејќи немаме податоци за неговиот вкус
        RETURN QUERY
            SELECT event_id, title, 0.00::NUMERIC
            FROM public.event
            WHERE start_datetime > CURRENT_TIMESTAMP
            ORDER BY start_datetime ASC
            LIMIT p_limit;

        RETURN;
    END IF;

    -- 3. ХИБРИДНОТО ПРЕБАРУВАЊЕ
    RETURN QUERY
        SELECT
            e.event_id,
            e.title,
            -- Претвораме растојание (0 до 2) во процент на поклопување (0% до 100%) за полесно читање
            ( (1 - (e.event_embedding <=> v_taste_vector)) * 100 )::NUMERIC(5,2) AS match_score
        FROM
            public.event e
        WHERE
            e.event_embedding IS NOT NULL

          -- SQL ФИЛТЕР 1: Само настани што се во иднината
          AND e.start_datetime > CURRENT_TIMESTAMP

          -- SQL ФИЛТЕР 2: Не му препорачувај настан за кој ВЕЌЕ има купено валиден билет
          AND e.event_id NOT IN (
            SELECT tt.event_id
            FROM public.order_cart oc
                    JOIN public.ticket t 1<->1..n: ON oc.order_id = t.order_id
                    JOIN public.ticket_type tt 1..n<->1: ON t.ticket_type_id = tt.ticket_type_id
            WHERE oc.user_id = p_user_id AND t.status IN ('VALID', 'RESERVED')
        )
        ORDER BY
            -- ВЕКТОРСКО СОРТИРАЊЕ: Ги редиме од најслични кон најразлични
            e.event_embedding <=> v_taste_vector ASC
        LIMIT p_limit;
END;
$$ LANGUAGE plpgsql;
```

Вака изгледа изведувањето и резултатот од функцијата:

```sql
SELECT * FROM fn_get_event_recommendations( p_user_id 1,  p_limit 2);
```

	recommended_event_id	event_title	match_score
1	472636	Traditional Food Festival 1239	99.91
2	472637	Gaming Weekend 1240	99.91

Сега во однос на правењето на embeddings за настаните. Првата идеа беше ова да биде како trigger при insert и update на настан но ова не е добар начин. Доколку ја натераме

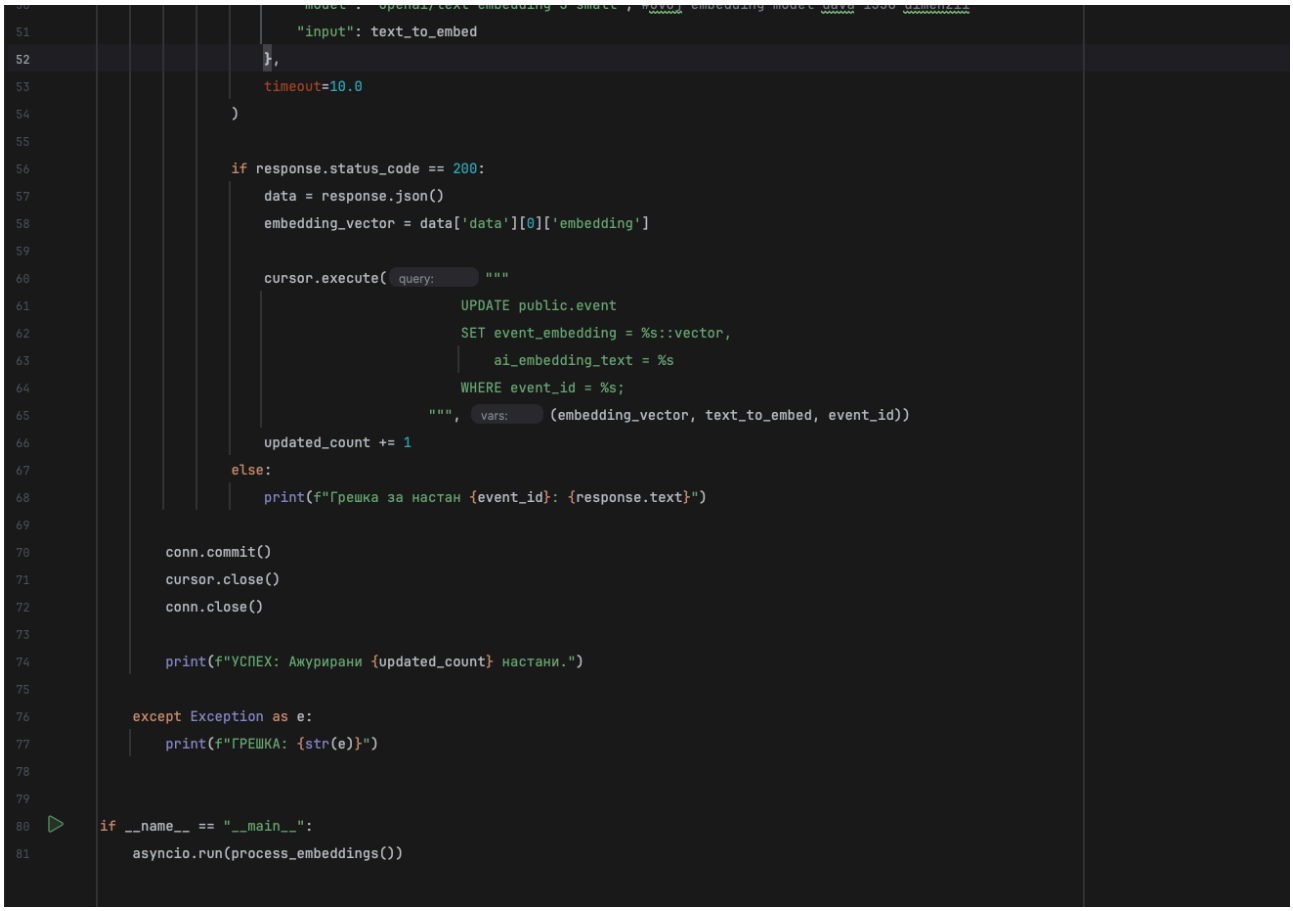
базата да комуницира со некој надворешен извор додека таа комуникација не заврши базата е блокирана што е неприфатливо бидејки базата секогаш треба да е достапна. Затоа решението што го направивме е во форма на job т.е поконкретно системски cron job. Секој ден на полноќ со помош на оперативниост систем ке се изведува скриптата која ја дискутиравме претходно. Ова значи дека поставено е на оперативнисот систем на патеката каде се наоѓаа оваа скрипта cron тајмер кој секојдневно во 12 часот на полноќ ке ја активира сркиптата.
Командите за поставување на за овој системски cron job се:

crontab -e
0 0 * * * /Users/GorjanStefanovski/EventFlow/venv/bin/python /Users/GorjanStefanovski/EventFlow/vector_job.py >> /Users/GorjanStefanovski/EventFlow/cron_log.txt 2>&1

Задачата на скриптата е да ги земе сите настни кои немаат embedding т.е каде колоните event_embedding и ai_embedding_text се null или пак каде редот ai_embedding_text е сменет(ова би значело дека се извршило update на настанот) и да ги пополни или ажурира овие редови преку комуникација со embedding моделот што е дискутирана и напишана на почетокот.

Скриптата:

```python
import psycopg2
import httpx
import asyncio
from datetime import datetime

from main import OPENROUTER_API_KEY, DB_CONFIG

OPENROUTER_API_KEY = {OPENROUTER_API_KEY}
DB_CONFIG = {
    "dbname": "postgres",
    "user": "postgres",
    "password": {DB_CONFIG["password"]},
    "host": "localhost",
    "port": "5432"
}


async def process_embeddings():  1 usage
    print(f"\n[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}] ЗАПОЧНУВА CRON JOB...")
    try:
        conn = psycopg2.connect(**DB_CONFIG)
        cursor = conn.cursor()

        cursor.execute("""
                    SELECT v.event_id, v.ai_embedding_text
                    FROM public.v_event_embedding_source v
                    JOIN public.event e ON v.event_id = e.event_id
                    WHERE e.event_embedding IS NULL
                       OR e.ai_embedding_text IS DISTINCT FROM v.ai_embedding_text
                    LIMIT 10;
                """)
        events = cursor.fetchall()

        if not events:
            print("Нема нови настани за векторизирање. Завршувам.")
            cursor.close()
            conn.close()
            return

        updated_count = 0
        async with httpx.AsyncClient() as client:
            for event_id, text_to_embed in events:
                response = await client.post(
                    url: "https://openrouter.ai/api/v1/embeddings",
                    headers={
                        "Authorization": f"Bearer {OPENROUTER_API_KEY}",
                        "HTTP-Referer": "http://localhost:8000"
                    },
                    json={
                        "model": "openai/text-embedding-3-small", #ovoj embedding model dava 1536 dimenzii
```



```
                    "input": text_to_embed
                },
                timeout=10.0
            )

            if response.status_code == 200:
                data = response.json()
                embedding_vector = data['data'][0]['embedding']

                cursor.execute( query: """
                                UPDATE public.event
                                SET event_embedding = %s::vector,
                                    ai_embedding_text = %s
                                WHERE event_id = %s;
                            """, vars: (embedding_vector, text_to_embed, event_id))
                updated_count += 1
            else:
                print(f"Грешка за настан {event_id}: {response.text}")

        conn.commit()
        cursor.close()
        conn.close()

        print(f"УСПЕХ: Ажурирани {updated_count} настани.")

    except Exception as e:
        print(f"ГРЕШКА: {str(e)}")


if __name__ == "__main__":
    asyncio.run(process_embeddings())
```