Zaawansowane Systemy Baz Danych – Etap 2 „Przychodnia”

Michał Ankiersztajn 311171

1. UDF

Główne różnice między funkcją, a procedurą:

Funkcja	**Procedura**
Może tylko czytać dane	Może czytać i modyfikować dane
Autonomiczne funkcje mogą być częścią zapytania	Nie może być częścią zapytania
Kompilowana za każdym razem	Kompilowana raz i używana wielokrotnie
Brak transakcji	Wsparcie dla transakcji
Nie może wywoływać procedur	Może wywoływać funkcje
Zawsze zwraca wynik na końcu	Procedura może zwracać wynik za pomocą OUT i IN OUT

Podsumowując, funkcje lepiej wykorzystywać dla mniejszych i reużywalnych fragmentów kodu. Kod ten można potem użyć zarówno w innych funkcjach, jak i procedurach. Dodatkowo są one bardziej ograniczone, a co za tym idzie prostsze do zrozumienia! Największym ograniczeniem jest readonly dla danych.

Procedury są dużo bardziej skomplikowane i na wiele więcej pozwalają, jeśli funkcja nie jest wstanie wykonać tego co trzeba – najlepiej napisać procedurę.

Różnica pomiędzy wyzwalaczami, a procedurami jest taka, że wyzwalacze odpalane są automatycznie przy zmianach w tabeli, a procedurę musi ręcznie wywołać użytkownik. Dodatkowo wyzwalacze z powyższego powodu nie mogą zwracać żadnych wartości i są pewnego rodzaju automatyzacją naszej bazy danych.

Dodatkowo bardzo przydatna tabela z prezentacji **„KH_Wykład_ZSBD_2024_wyklad_4.pdf”:**

	UDF	SP	Trigger
Parametry wejściowe	0+	0+	0
Parametry wyjściowe	1 (skalarny albo tablica)	0+	0
Wywołanie	Zagnieżdżony lub wywołanie	Wywołanie	Automatycznie
Może wywołać	Nic	UDF, SP	UDF, SP
Zapytania	1 SELECT	Wiele DML	Wiele DML
Przeciążenie	Tak	Nie	Nie
Pętle	Nie	Tak	Tak
Obsługa wyjątków	Nie	Tak	Tak
Zależność od SZBD	Średnia	Wysoka	Bardzo wysoka

Funkcje i procedury:

1. **Manager:**

Procedura pozwalają przypisać wiele specjalizacji do doktora po nazwach specjalizacji

```
CREATE OR REPLACE PROCEDURE add_specializations(
    doctor_id int,
    specializations VARCHAR(100)[]
)
LANGUAGE plpgsql
AS $$
DECLARE
    spec_id int;
    spec_name VARCHAR(100);
BEGIN
    FOREACH spec_name IN ARRAY specializations
    LOOP
        SELECT s.id INTO spec_id FROM specialization s WHERE s.name = spec_name LIMIT 1;
        INSERT INTO doctorspecialization(DoctorID, SpecializationID)
        VALUES(doctor_id, spec_id);
    END LOOP;
END;
$$;
```

Przykład użycia:

```
MediPlaceDatabase=# SELECT * FROM doctorspecialization ds WHERE ds.doctorID = 2;
 id | doctorid | specializationid
----+----------+------------------
  2 |        2 |                2
(1 row)

MediPlaceDatabase=# CALL add_specializations(2, ARRAY['Cardiology', 'Pediatrics']);
CALL
MediPlaceDatabase=# SELECT * FROM doctorspecialization ds WHERE ds.doctorID = 2;
 id | doctorid | specializationid
----+----------+------------------
  2 |        2 |                2
 10 |        2 |                1
 11 |        2 |                3
(3 rows)
```

2. **Doktor:**

Procedura pozwalająca na dodanie leku

```
CREATE OR REPLACE PROCEDURE add_medicine(
    medicine_name VARCHAR(250),
    OUT medicine_id int
)
LANGUAGE plpgsql
AS $$
BEGIN
    SELECT ID INTO medicine_id
    FROM Medicine
    WHERE Name = medicine_name;

    IF NOT FOUND THEN
        INSERT INTO medicine(name) VALUES(medicine_name ) RETURNING ID INTO medicine_id;
    END IF;
END;
$$;
```

Przykład użycia

```
MediPlaceDatabase=# SELECT * FROM medicine;
 id |    name
----+-------------
  1 | Aspirin
  2 | Ibuprofen
  3 | Paracetamol
  4 | Apap
(4 rows)

MediPlaceDatabase=# \set medicine_id 0
MediPlaceDatabase=# CALL add_medicine('Penicylyna', :medicine_id);
 medicine_id
-------------
          10
(1 row)

MediPlaceDatabase=# SELECT * FROM medicine;
 id |    name
----+-------------
  1 | Aspirin
  2 | Ibuprofen
  3 | Paracetamol
  4 | Apap
 10 | Penicylyna
(5 rows)
```

3. **Pacjent:**

Funkcja wyświetlająca wizyty pacjenta wraz z imieniem i nazwiskiem doktora, kodem recepty, listą leków i oceną pacjenta:

```sql
CREATE OR REPLACE FUNCTION get_visits(patient_id int)
returns TABLE (
date DATE,
doctor_name varchar(55),
doctor_surname varchar(55),
comment varchar(520),
satisfaction int,
prescription_code varchar(50),
medicines text
)
LANGUAGE plpgsql
AS
$$
BEGIN
    RETURN QUERY
    SELECT
        a.date,
        d.name,
        d.surname,
        a.comment,
        r.satisfaction,
        p.code AS prescription_code,
        STRING_AGG(m.name, ', ') AS medicines
    FROM appointment a
    LEFT JOIN rating r ON a.id = r.appointmentid
    LEFT JOIN prescription p ON a.id = p.appointmentid
    LEFT JOIN medicineprescription mp ON p.id = mp.prescriptionid
    LEFT JOIN medicine m ON mp.medicineid = m.id
    LEFT JOIN doctor d ON d.id = a.doctorid
    WHERE a.patientid = patient_id
    GROUP BY a.id, d.name, d.surname, a.date, a.comment, a.doctorid, a.patientid, r.satisfaction, p.code;
END
$$;
```

Przykładowe użycie:

```
MediPlaceDatabase=# SELECT * FROM get_visits(3);
    date    | doctor_name | doctor_surname |      comment       | satisfaction | prescription_code |  medicines
------------+-------------+----------------+--------------------+--------------+-------------------+-------------
 2024-10-20 | Charlie     | White          | Child consultation |              | RX77890           | Paracetamol
(1 row)
```

Funkcja wyświetlająca doktorów w zależności od podanych specjalizacji, daty oraz minimalnej ilości specjalizacji.

```sql
CREATE OR REPLACE FUNCTION get_available_doctors(
    specializations TEXT[],
    min_specializations INT,
    check_date DATE
)
RETURNS TABLE(
    name VARCHAR(55),
    surname VARCHAR(55)
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT DISTINCT d.name, d.surname
    FROM doctor d
    JOIN (
        SELECT ds.doctorID
        FROM doctorspecialization ds
        JOIN specialization s ON ds.specializationID = s.ID
        WHERE s.name = ANY(specializations)
        GROUP BY ds.doctorID
        HAVING COUNT(DISTINCT s.name) >= min_specializations
    ) AS specializedDoctors ON d.id = specializedDoctors.doctorID
    JOIN (
        SELECT a.doctorID
        FROM appointment a
        WHERE NOT a.date = check_date
    ) AS availableDoctors ON d.id = availableDoctors.doctorID;
END
$$;
```

Przykładowe użycie:

```
MediPlaceDatabase=# SELECT * FROM get_available_doctors(
    ARRAY['Cardiology', 'Dermatology'],
    2,
    '2024-10-22'
);
 name  | surname
-------+---------
 Alice | Brown
(1 row)

MediPlaceDatabase=# SELECT * FROM get_available_doctors(
    ARRAY['Cardiology', 'Dermatology'],
    1,
    '2024-10-22'
);
 name  | surname
-------+---------
 Alice | Brown
 Bob   | Green
(2 rows)
```

Procedura usuwająca wizytę o ile jest ona w przyszłości:

```sql
CREATE OR REPLACE PROCEDURE cancel_appointment(
    appointment_id INTEGER
)
LANGUAGE plpgsql
AS $$
DECLARE
    appointment_date DATE;
BEGIN
    SELECT date FROM Appointment a WHERE a.ID = appointment_id INTO appointment_date;

    IF appointment_date > CURRENT_DATE THEN
        DELETE FROM Appointment a WHERE a.ID = appointment_id;
    ELSE
        RAISE EXCEPTION 'Cannot cancel appointments from the past';
    END IF;
END;
$$;
```

Przykład użycia:

```
MediPlaceDatabase=# SELECT * FROM Appointment;
 id |    date    |      comment       | doctorid | patientid
----+------------+--------------------+----------+-----------
  1 | 2024-10-15 | Regular check-up   |        1 |         1
  2 | 2024-10-18 | Skin rash          |        2 |         2
  3 | 2024-10-20 | Child consultation |        3 |         3
  4 | 2024-10-16 | no comment         |        1 |         2
  5 | 2024-11-25 | empty              |        1 |         2
(5 rows)

MediPlaceDatabase=# CALL cancel_appointment(5);
CALL
MediPlaceDatabase=# SELECT * FROM Appointment;
 id |    date    |      comment       | doctorid | patientid
----+------------+--------------------+----------+-----------
  1 | 2024-10-15 | Regular check-up   |        1 |         1
  2 | 2024-10-18 | Skin rash          |        2 |         2
  3 | 2024-10-20 | Child consultation |        3 |         3
  4 | 2024-10-16 | no comment         |        1 |         2
(4 rows)

MediPlaceDatabase=# CALL cancel_appointment(1);
ERROR:  Cannot cancel appointments from the past
CONTEXT:  PL/pgSQL function cancel_appointment(integer) line 10 at RAISE
```

Wnioski:

Pisząc funkcje i procedury zauważyłem, że najlepiej używać snake_case'a dla zmiennych (przynajmniej przy tak przyjętym nazewnictwie dla table i kolumn jak w tym projekcie).

Funkcje są świetnym zabezpieczeniem przed modyfikacją danych.

Zarówno procedury i funkcje pozwalają na zwiększenie bezpieczeństwa systemu ograniczając to co użytkownik może zrobić.

2. Złożona procedura

Procedura lekarza, która pozwala na stworzenie recepty z losowym kodem i dodanie listy leków. Jeśli lek nie istnieje zostaje on dodany do bazy danych. Wykorzystuje ona jedną z poprzednich procedur, która dodawała leki:

```
CREATE OR REPLACE PROCEDURE create_prescription(
    appointment_id INTEGER,
    medicine_names VARCHAR[]
)
LANGUAGE plpgsql
AS $$
DECLARE
    prescription_id INTEGER;
    medicine_id INTEGER;
    prescription_code VARCHAR(50);
    medicine_name VARCHAR;
BEGIN
    BEGIN
        prescription_code := md5(random()::text);

        INSERT INTO prescription(Code, AppointmentID)
        VALUES (prescription_code, appointment_id)
        RETURNING ID INTO prescription_id;

        FOREACH medicine_name IN ARRAY medicine_names
        LOOP
            CALL add_medicine(medicine_name, medicine_id);

            INSERT INTO MedicinePrescription (PrescriptionID, MedicineID)
            VALUES (prescription_id, medicine_id);
        END LOOP;

        COMMIT;
    EXCEPTION
        WHEN OTHERS THEN
            ROLLBACK;
            RAISE EXCEPTION 'An error occurred during prescription creation. Transaction has been rolled back: %', SQLERRM;
    END;
END;
$$;
```

```
MediPlaceDatabase=# CALL create_prescription(4, ARRAY['Aspirin', 'AntyBol', 'Paracetamol']);
ERROR:  An error occurred during prescription creation. Transaction has been rolled back: cannot commit while a subtransaction is active
CONTEXT:  PL/pgSQL function create_prescription(integer,character varying[]) line 27 at RAISE
```

Początkowo tak wyglądał mój kod, okazuje się jednak, że PostgreSQL tworzy savepointy przy klauzuli EXCEPTION, więc korzystanie z explicit COMMIT powoduje subtransaction error. Oznacza to, że należy usunąć COMMIT ze skryptu, ponieważ wykorzystuje on już mechanizm transakcji pod spodem dzięki blokowi z EXCEPTION. Aktualizuję skrypt usuwając linijkę COMMIT.

Przykład użycia:

```
MediPlaceDatabase=# SELECT * FROM prescription;
 id |  code   | appointmentid
----+---------+---------------
  1 | RX12345 |             1
  2 | RX67890 |             2
  3 | RX77890 |             3
(3 rows)

MediPlaceDatabase=# SELECT * FROM medicine;
 id |    name
----+-------------
  1 | Aspirin
  2 | Ibuprofen
  3 | Paracetamol
  4 | Apap
 10 | Penicylyna
(5 rows)

MediPlaceDatabase=# SELECT * FROM medicineprescription;
 id | prescriptionid | medicineid
----+----------------+------------
  1 |              1 |          1
  2 |              2 |          2
  3 |              3 |          3
(3 rows)

MediPlaceDatabase=# CALL create_prescription(4, ARRAY['Aspirin', 'AntyBol', 'Paracetamol']);
CALL
MediPlaceDatabase=# SELECT * FROM medicine;
 id |    name
----+-------------
  1 | Aspirin
  2 | Ibuprofen
  3 | Paracetamol
  4 | Apap
 10 | Penicylyna
 18 | AntyBol
(6 rows)

MediPlaceDatabase=# SELECT * FROM medicineprescription;
 id | prescriptionid | medicineid
----+----------------+------------
  1 |              1 |          1
  2 |              2 |          2
  3 |              3 |          3
 28 |             21 |          1
 29 |             21 |         18
 30 |             21 |          3
(6 rows)

MediPlaceDatabase=# SELECT * FROM prescription;
 id |               code               | appointmentid
----+----------------------------------+---------------
  1 | RX12345                          |             1
  2 | RX67890                          |             2
  3 | RX77890                          |             3
 21 | 5b0558fc0f0930ec48b75bba5d839b84 |             4
(4 rows)
```

Jak widać dane poprawnie zostały dodane do wszystkich 3 tabel.

3. Triggery

W bazie PostgreSQL, aby użyć triggera trzeba zdefiniować funkcję, która najpierw zwraca TRIGGER.

Przy usunięciu wizyty, powinien zostać usunięty komentarz, recepta i połączenie między receptą, a listą leków.

```sql
CREATE OR REPLACE FUNCTION delete_appointment_related()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    DELETE FROM Rating r WHERE r.AppointmentID = OLD.ID;

    DELETE FROM MedicinePrescription mp
    WHERE mp.PrescriptionID IN (SELECT ID FROM Prescription p WHERE p.AppointmentID = OLD.ID);

    DELETE FROM Prescription p WHERE p.AppointmentID = OLD.ID;

    RETURN OLD;
END;
$$;

CREATE TRIGGER before_appointment_delete
BEFORE DELETE
ON Appointment
FOR EACH ROW
EXECUTE FUNCTION delete_appointment_related();
```

Korzystam z BEFORE, ponieważ chciałbym uniknąć sytuacji w której w bazie choćby przez chwilę będą tabele posiadające referencje do nieistniejącej tabeli.

Przykład użycia:

PRZED:

```
MediPlaceDatabase=# SELECT * FROM Appointment;
 id |    date    |      comment       | doctorid | patientid
----+------------+--------------------+----------+-----------
  1 | 2024-10-15 | Regular check-up   |        1 |         1
  2 | 2024-10-18 | Skin rash          |        2 |         2
  3 | 2024-10-20 | Child consultation |        3 |         3
  4 | 2024-11-30 | no comment         |        1 |         2
(4 rows)

MediPlaceDatabase=# SELECT * FROM Rating;
 id | satisfaction | appointmentid
----+--------------+---------------
  1 |            5 |             1
  2 |            4 |             2
  3 |            1 |             4
(3 rows)

MediPlaceDatabase=# SELECT * FROM Prescription;
 id |               code               | appointmentid
----+----------------------------------+---------------
  1 | RX12345                          |             1
  2 | RX67890                          |             2
  3 | RX77890                          |             3
 21 | 5b0558fc0f0930ec48b75bba5d839b84 |             4
(4 rows)

MediPlaceDatabase=# SELECT * FROM MedicinePrescription;
 id | prescriptionid | medicineid
----+----------------+------------
  1 |              1 |          1
  2 |              2 |          2
  3 |              3 |          3
 28 |             21 |          1
 29 |             21 |         18
 30 |             21 |          3
(6 rows)
```

PO:

```
MediPlaceDatabase=# CALL cancel_appointment(4);
CALL
MediPlaceDatabase=# SELECT * FROM Appointment;
 id |    date    |      comment       | doctorid | patientid
----+------------+--------------------+----------+-----------
  1 | 2024-10-15 | Regular check-up   |        1 |         1
  2 | 2024-10-18 | Skin rash          |        2 |         2
  3 | 2024-10-20 | Child consultation |        3 |         3
(3 rows)

MediPlaceDatabase=# SELECT * FROM Rating;
 id | satisfaction | appointmentid
----+--------------+---------------
  1 |            5 |             1
  2 |            4 |             2
(2 rows)

MediPlaceDatabase=# SELECT * FROM Prescription;
 id |  code   | appointmentid
----+---------+---------------
  1 | RX12345 |             1
  2 | RX67890 |             2
  3 | RX77890 |             3
(3 rows)

MediPlaceDatabase=# SELECT * FROM MedicinePrescription;
 id | prescriptionid | medicineid
----+----------------+------------
  1 |              1 |          1
  2 |              2 |          2
  3 |              3 |          3
(3 rows)
```

Wizyty powinny być tworzone tylko w przyszłości

```sql
CREATE OR REPLACE FUNCTION can_add_appointment()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF NEW.date < CURRENT_DATE THEN
        RAISE EXCEPTION 'Appointments can only be added in the future';
    END IF;
END;
$$;

CREATE TRIGGER before_appointment_insert
BEFORE INSERT
ON Appointment
FOR EACH ROW
EXECUTE FUNCTION can_add_appointment();
```

Przykład użycia:

```
MediPlaceDatabase=# INSERT INTO Appointment(date, comment, doctorid, patientid) VALUES ('2024-9-1', 'empty', 1, 1);
ERROR:  Appointments can only be added in the future
CONTEXT:  PL/pgSQL function can_add_appointment() line 4 at RAISE
```

Jedną z największych wad jest to, że przy wywaleniu się TRIGGERa możemy nawet nie wiedzieć, że to się wydażyło. Początkowo ustawiłem RETURN NEW; przy delete co zwracało NULL, jednak nie wiedziałem wtedy, że NULL oznacza abort i cofnięcie transakcji. Przez takie ukryte mechanizmy można coś przypadkiem pominąć.

Dodatkowo, edycje triggerów są skomplikowane i mogą spowodować breaking changes.

4. Agregacje

W wcześniejszych podpunktach zostały opisane potrzebne agregacje, wyzwalacze, a perspektywy w poprzednim sprawozdaniu.

Nie udało mi się zaimplementować poniższych agregacji:

- Dodanie oceny przez pacjenta.
- Edycja komentarza wizyty przez doktora
- Autouzupełnianie nazw leków przez lekarza

Nie pisałem dodatkowych procedur i funkcji dla ADMINa, ponieważ w PostgreSQL podstawowo istnieje wiele takich przydatnych procedur, część z nich wykorzystałem w skryptach jak np. CURRENT_DATE

5. Automatyzacja zadania z wykorzystaniem jobów

Przy aktualnym ustawieniu tabel ciężko jest dodać sensownego joba operującego na tabelach, dlatego dodam nową tabelę na miesięczne raporty, jest to bardzo przydatne z punktu biznesowego, aby przychodnia mogła się rozwijać wiedząc, co miało wpływ na satysfakcję, liczbę wizyt, pacjentów, doktorów i ich wykształcenie:

Nowa Tabela:

```
CREATE TABLE IF NOT EXISTS MonthlyReports (
    ID SERIAL PRIMARY KEY,
    ReportMonth INTEGER NOT NULL,
    ReportYear INTEGER NOT NULL,
    Patients INTEGER,
    Doctors INTEGER,
    AvgSatisfaction DECIMAL(3, 2),
    AppointmentsCount INTEGER,
    AvgSpecializationsPerDoctor DECIMAL(3, 2)
);
```

Procedura generująca raport (nie mieści się na screenie, jest na dole w pliku **scripts.sql**)

Do automatyzacji skorzystam z CRONa:

Najpierw instalacja przez terminal docker desktop:

Kolejno komendy:

apt-get install postgresql-17-cron

Potem w /var/lib/postgresql/data dodaję shared_preload_libraries:

```
#local_preload_libraries = ''
#session_preload_libraries = ''
shared_preload_libraries = 'pg_cron'            # (change requires restart)
cron.database_name = 'MediPlaceDatabase'
#jit_provider = 'llvmjit'               # JIT library to use
```

I restartuję serwer

```
4 UTC
zsbd-postgresql_database-1  | 2024-10-31 09:48:07.827 UTC [1] LOG:  database system is ready to accept connections
zsbd-postgresql_database-1  | 2024-10-31 09:48:07.832 UTC [32] LOG:  pg_cron scheduler started
```

Schedule:

```
CREATE EXTENSION IF NOT EXISTS pg_cron;
SELECT cron.schedule('Monthly Report Generation',
                     '59 23 $ * *',
                     'CALL GenerateMonthlyReport()');
```

Ostatniego dnia miesiąca o godz. 23:59 będzie generowany miesięczny raport.

Działanie:

```
MediPlaceDatabase=# SELECT * FROM MonthlyReports;
 id | reportmonth | reportyear | patients | doctors | avgsatisfaction | appointmentscount | avgspecializationsperdoctor
----+-------------+------------+----------+---------+-----------------+-------------------+-----------------------------
(0 rows)

MediPlaceDatabase=# SELECT cron.schedule('Report Generation', '15 seconds', 'CALL GenerateMonthlyReport()');
 schedule
----------
        6
(1 row)

MediPlaceDatabase=# SELECT * FROM MonthlyReports;
 id | reportmonth | reportyear | patients | doctors | avgsatisfaction | appointmentscount | avgspecializationsperdoctor
----+-------------+------------+----------+---------+-----------------+-------------------+-----------------------------
 20 |          10 |       2024 |        4 |       4 |            4.50 |                 3 |                        2.00
(1 row)
```

Dla testu działania scheduling ustawiłem 15 sekund, realnie będzie ta komenda co wyżj w sekcji ‘Schedule’.

6. Kopia zapasowa

Tworzenie backupu:

```
PS C:\Users\Michal\Desktop\ZSBD> docker exec -it zsbd-postgresql_database-1 pg_dump -U admin MediPlaceDatabase > backup
```

Weryfikacja:

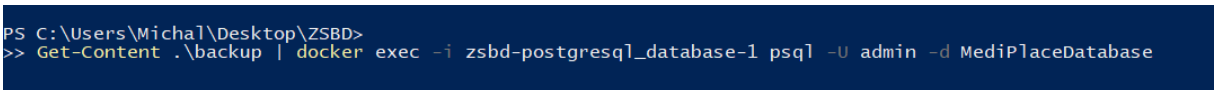
Jak to zrobię?

1. Usunę starą bazę danych

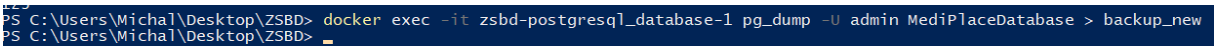
```
PS C:\Users\Michal\Desktop\ZSBD> docker compose rm
? Going to remove zsbd-postgresql_database-1 Yes
```

2. Stworzę ją od nowa bez skryptu init.sql

3. Wczytam backup



4. Eksportuję nową bazę danych



5. Porównuję, czy pliki są takie same, wklejając zawartość backup oraz backup_new

https://www.diffchecker.com/RExtzkyV/

Skorzystałem z tej strony ignorując nowe linie, ponieważ przy eksporcie nowej bazy dodawane były znaki nowej linii, które nie wpływały na odtworzenie bazy danych.

7. Konfiguracja MS SQL Server

Skorzystam z dockera o bardzo podobnej konfiguracji:

```
version: '3.8'
services:
  sql-server:
    image: mcr.microsoft.com/mssql/server
    container_name: zsbd_ms_sql
    hostname: zsbd_ms_sql

    environment:
      SA_PASSWORD: H4rdP4ssW0rD
      ACCEPT_EULA: Y
    ports:
      - "1433:1433"
    volumes:
      - ./SQL:/docker-entrypoint-initdb.d
```

Tak jak wcześniej będę to uruchamiał za pomocą:

- docker-compose pull
- docker-compose up –build

Ciekawostka – MS SQL wymusza trudniejsze hasło podczas, gdy PostgreSQL pozwala na dowolne. Początkowo chciałem ustawić je jako takie samo, jednak failowało to builda.

8. Analiza różnic PostgreSQL vs MS SQL Server

- MS SQL korzysta z T-SQL, a PostgreSQL korzysta z PL/pgSQL

- MS SQL korzysta z TRY CATCH zamiast EXCEPTION
- MS SQL nie wspiera pętli FOREACH
- MS SQL nie wymaga użycia procedur lub funkcji na triggerach
- MS SQL pozwala na pisanie triggerów w miejscu ich tworzenia, a nie przez funkcje/procedury
- MS SQL nie posiada typu ARRAY
- MS SQL posiada unikalność fk w ramach bazy danych, a nie pojedynczej tabeli.
- MS SQL Parametry funkcji i procedur zaczynają się od @
- MS SQL ma ograniczone i mniej zaawansowane indeksy względem PostgreSQL
- MS SQL ma ograniczone i mniej zaawansowane triggery względem PostgreSQL
- MS SQL nie posiada typu SERIAL
- MS SQL funkcje i procedury przyjmują parametry kolejno podawane, a postgreSQL przyjmuje je w nawiasie.
- MS SQL nie tworzy automatycznie transakcji na blokach TRY CATCH
- Różnice w składni przy dodawaniu permissionów dla użytkowników

Więcej różnic, których nie wypisałem: https://www.enterprisedb.com/blog/microsoft-sql-server-mssql-vs-postgresql-comparison-details-what-differences

Osobiście preferuję składnię w PostgreSQL, jest ona dla mnie prostsza i bardziej intuicyjna, składnia MS SQL Server jest według mnie po prostu lekko przekombinowana. Miałem też tu spore problemy z widokiem z terminala, zmusiło mnie to do instalacji DBeavera, aby sensownie operować danymi.

9. Migracja PostgreSQL > MS SQL Server

Aby migracja przebiegła jak najprościej się da, wezmę wcześniej stworzony backup i skorzystam z narzędzia: https://www.sqlines.com/online

Po skorzystaniu z narzędzia z init.sql musiałem poprawić fk, ponieważ wcześniej były one unikalne w ramach tabeli, a nie bazy.

Użytkowników, dane, procedury, funkcje, indeksy oraz triggery przemigrowałem za pomocą ChatGPT:

https://chatgpt.com/share/672388f0-1b74-8011-b355-20cf021c8871

Popełniłem błąd próbując najpierw stworzyć triggery i procedury, a potem zaimportować dane, co spowodowało, że musiałem się cofnąć. Główną przyczyną był trigger nie pozwalający na dodawanie Appointment w przeszłości (co przy okazji zweryfikowało, że ta funkcjonalność działa).

Dodatkowo trzeba od nowa przypisać dostęp do poszczególnych procedur i funkcji dla użytkowników. Część z rzeczy musiałem sam przepisać (jak np. permissiony do procedur

i funkcji), jednak Chat w tym wypadku wygenerował dużo sensownego i działającego kodu.

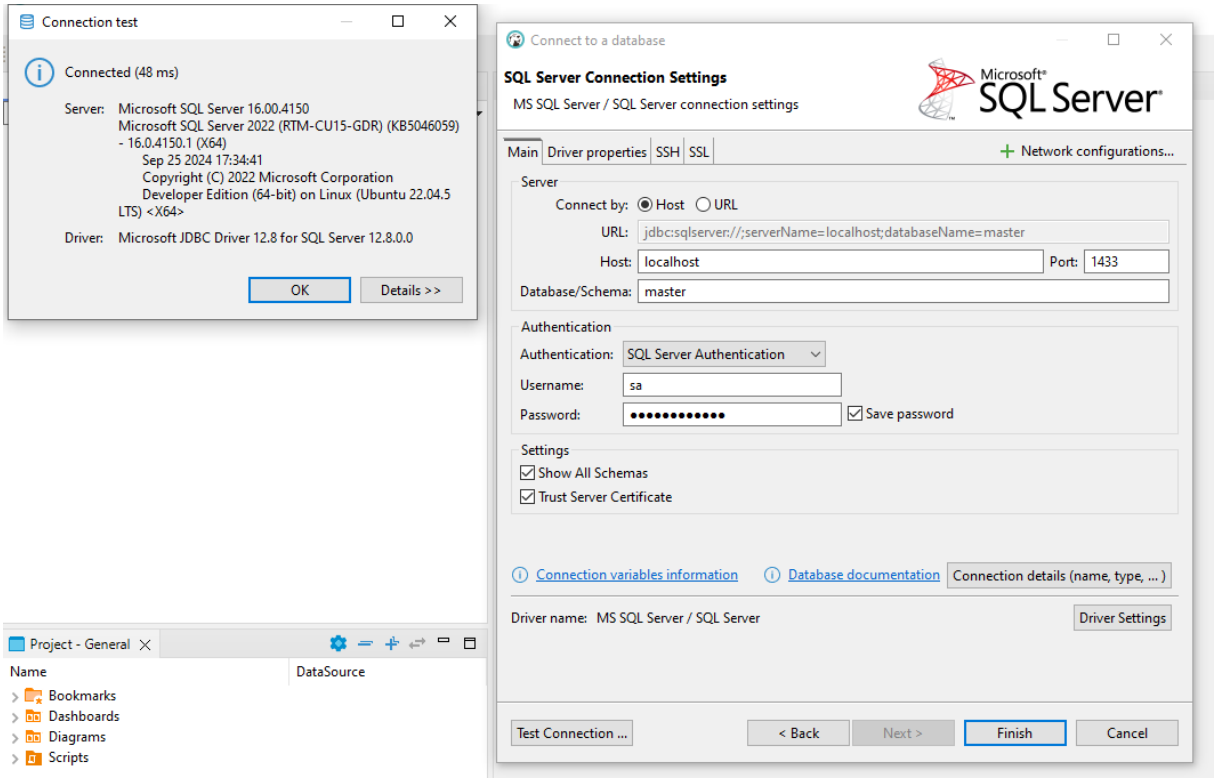
Cały skrypt z migracją znajduje się w katalogu **MS SQL Server/SQL/init.sql**

Okazuje się, że ChatGPT jest świetnym narzędziem do migracji danych z jednego systemu na drugi. Poprzednie narzędzie sqlines nie potrafiło przekonwertować procedur i funkcji nie powodując wielu błędów. Popełnia on czasem błędy, jednak są one raczej niewielkie.

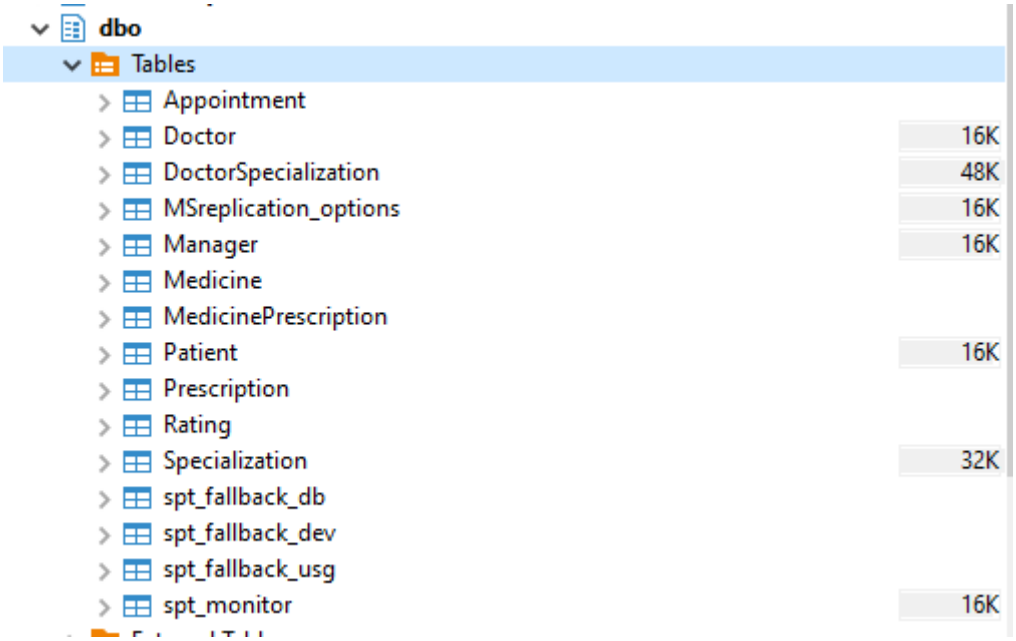
SQLines okazało się tragicznym narzędziem, jedyne w czym podołało to tworzenie tabel i insert danych. 90% procedur, funkcji i triggerów nie została poprawnie przeprowadzona.

10. Działanie bazy na MS SQL Server

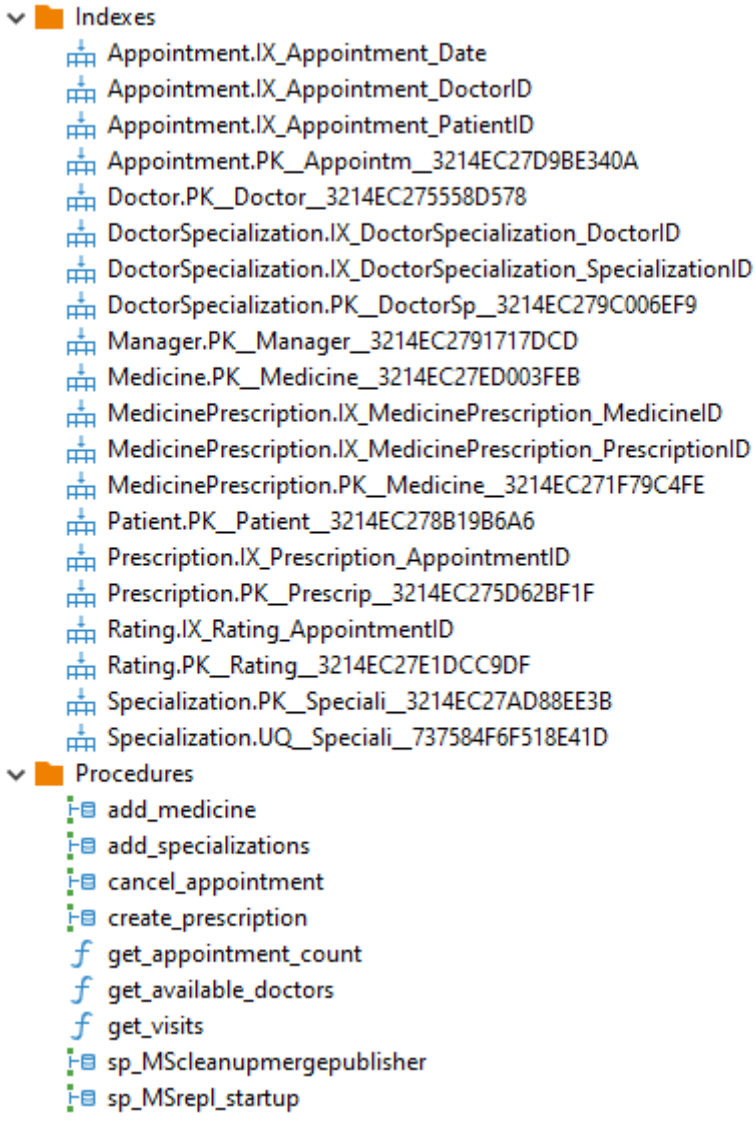
MS SQL posiada dużo gorsze narzędzie z poziomu terminala. Skorzystam z programu DBeaver:



Tabele:

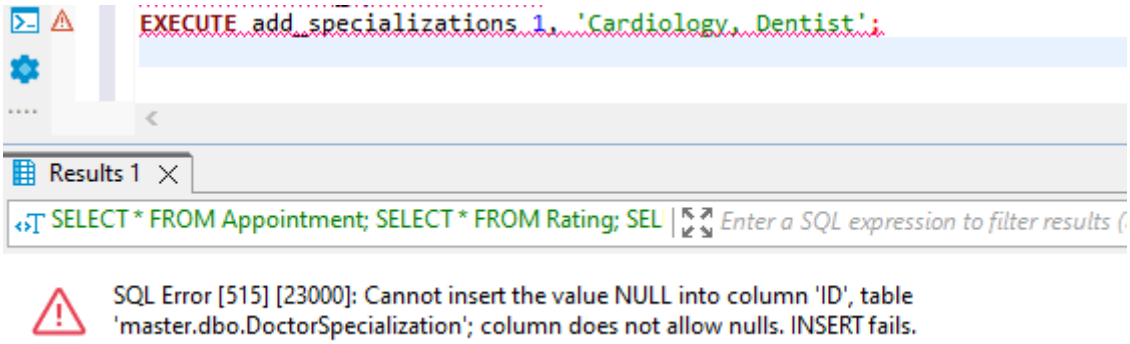


Indeksy i procedury:

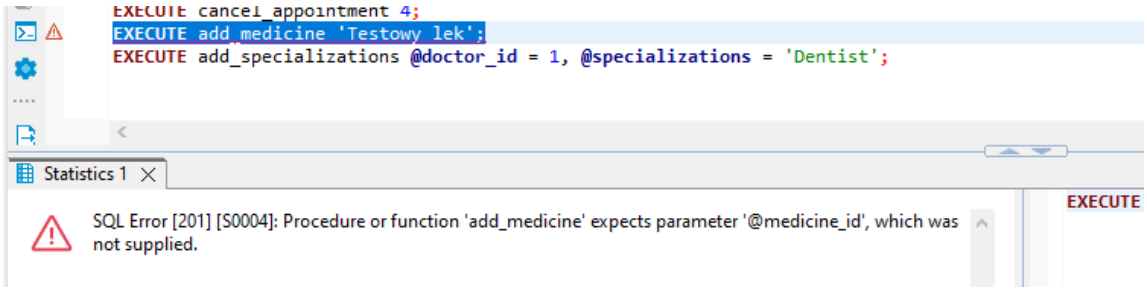


Próba usunięcia appointment(działający trigger):

Brak autoincrementu id (trzeba było dodać identity(1,1), co spowodowało potrzebę usunięcia ze skryptu id insertów:



Błąd przy próbie dodania leku:



Wygenerowana procedura miała błędy. Po poprawie:

```
EXECUTE cancel_appointment 4;
EXECUTE add_medicine 'Testowy lek';
EXECUTE add_specializations @doctor_id = 1, @specializations = 'Dent
```

Statistics 1

Name	Value
Updated Rows	1
Query	EXECUTE add_medicine 'Testowy lek';
Start time	Thu Oct 31 15:59:56 CET 2024
Finish time	Thu Oct 31 15:59:56 CET 2024

Wizyty danego pacjenta:

```
SELECT * FROM dbo.get_visits(1);
EXECUTE add_specializations @doctor_id = 1, @specializations = 'Dentist';
```

Results 1

SELECT * FROM dbo.get_visits(1); Enter a SQL expression to filter results (use Ctrl+Space)

	date	doctor_name	doctor_surname	comment	satisfaction	prescription_code	medicines
1	2024-10-15	Alice	Brown	Regular check-up	5	RX12345	Aspirin

Value: 2024-10-15

Dostępni doktorzy:

```
SELECT * FROM dbo.get_available_doctors('Cardiology, Neurology', 1, '2023-11-01');
EXECUTE add_specializations @doctor_id = 1, @specializations = 'Dentist';
```

esults 1

ELECT * FROM dbo.get_available_doctors('Cardiology, Ne Enter a SQL expression to filter results (use Ctrl+Space)

	name	surname
	Alice	Brown
	Bob	Green

Dodawanie specjalizacji doktorowi:

```
EXECUTE add_specializations @doctor_id = 1, @specializations = 'Dermatology';
```

Statistics 1

Name	Value
Updated Rows	1
Query	EXECUTE add_specializations @doctor_id = 1, @specializations = 'Dermatology';
Start time	Thu Oct 31 16:15:39 CET 2024
Finish time	Thu Oct 31 16:15:39 CET 2024

Dodanie leku:

```
DECLARE @med_id INT;
EXEC dbo.add_medicine
    @medicine_name = N'Paracetamol',
    @medicine_id = @med_id OUTPUT;
SELECT @med_id AS MedicineID;
```

Results 1

DECLARE @med_id INT; EXEC dbo.add_medicine @medici | Enter a SQL expression to filter results (use Ctr

Grid | Text

	123 MedicineID
1	3

Tworzenie recepty:

```
EXECUTE create_prescription 3, 'Aspirin, Testowy lek';
```

Statistics 1

Name	Value
Updated Rows	4
Query	EXECUTE create_prescription 3, 'Aspirin, Testowy lek';
Start time	Thu Oct 31 16:30:02 CET 2024
Finish time	Thu Oct 31 16:30:02 CET 2024

Bibliografia:

https://www.postgresql.org/docs/current/xfunc.html

https://www.geeksforgeeks.org/difference-between-trigger-and-procedure-in-dbms/

https://www.geeksforgeeks.org/difference-between-function-and-procedure/

https://neon.tech/postgresql/postgresql-triggers

https://www.postgresql.org/docs/current/backup-dump.html#BACKUP-DUMP-RESTORE

https://dev.to/mdarifulhaque/how-to-backup-a-postgresql-database-in-docker-step-by-step-guide-cp2

https://forums.docker.com/t/postgresql-backup-and-restore-with-powershell/32125

https://github.com/citusdata/pg_cron

https://medium.com/@seventechnologiescloud/local-sqlserver-database-via-docker-compose-the-ultimate-guide-f1d9f0ac1354

https://khalidabuhakmeh.com/running-sql-server-queries-in-docker

https://stackoverflow.com/questions/71688125/odbc-driver-18-for-sql-serverssl-provider-error1416f086