Databases for developers Task 2

Gruppemedlemmer
Josef Marc | jp-325
Sebastian Bentley | sb-287
Frederik Dinsen | fd-77
Frederik Dahl | fd-76

Github
soft_1sem_databases/Handin2 at main · dahlfrederik/soft_1sem_databases (github.com)

Introduktion

Der er blevet bedt om at lave en database, der indeholder information om hospitaler, læger, patienter og deres recepter. Denne database skal normaliseres i 3. normalform. Derudover skal der sendes en email til patienterne, når de er ved at være tom på deres recept medicin. Vi har gjort netop dette, ved at lave så meget funktionalitet som muligt ved brug af postgres funktioner, og python 3 til automatisk at sende mails.

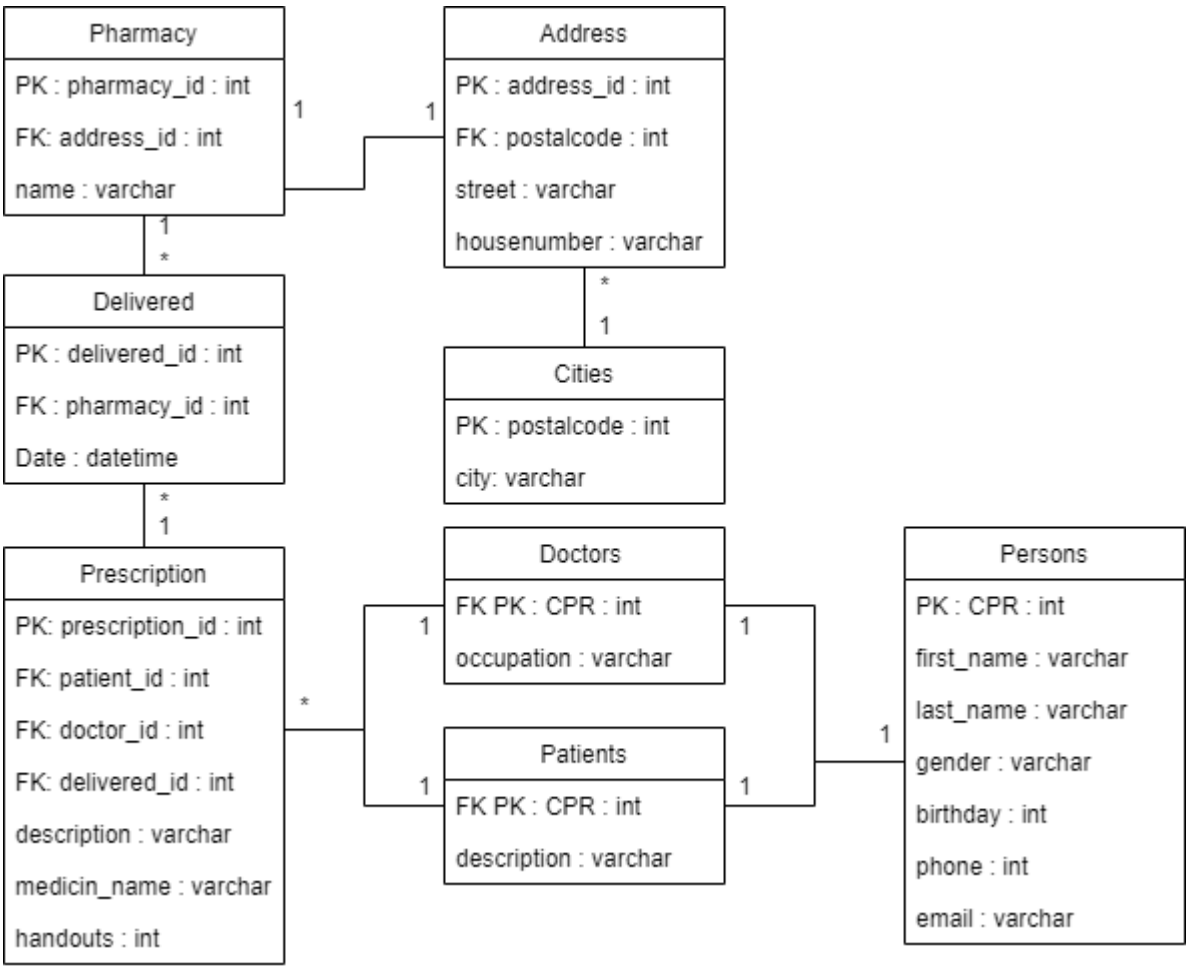
Dette projekt er udviklet i PostgreSQL og Python. Funktionerne til at oprette test data samt funktioner til at indsætte data i database strukturen er lavet i PostgreSQL. Python benyttes til at kunne køre og afvikle programmet fra commandline samt at sende automatiske emails. Se readme filen i GitHub for opsætningsguide til at benytte programmet.

Overvejelser

Projektet sat op således at det er en læge som skal lave nye recepter til patienter. Når der oprettes en ny recept tilføjer lægen relevante informationer om medicin: doser, mængde, holdbarhed, antal udleverede pakker medicin etc.
Patienter med receptpligtig medicin vil modtage en email såfremt der er en eller færre pakker af medicin tilbage af deres receptpligtige medicin.

I forhold til at lave en logger så ville vi benytte os af en logger-tabel hvor der ville blive indsat et nyt “entry” hver gang der bliver foretaget en CRUD operation i de andre tabeller i SQL-databasen samtidigt ville der også skulle logges hver gang en mail afsendes.
Dette ville blive tilføjet i Python delen af projektet og hurtigt indsættes.
Denne logger database ville vi opsætte som en document database (MongoDB) idet der kan være meget forskelligt indhold og at denne ikke behøver relationer til de andre tabeller. Fordelene ved en document database er at disse har et format som minder om JSON og ikke har fastsatte kolonner fra starten af. Indhold kan derfor sættes meget specifikt ved logging af relevant input.

Database design



figur 1: database UML

Ovenstående UML diagram er en illustration af vores database design. Vi benytter en relationel database tilgang til at opbevare data på tværs af tabellerne.

Persons tabellen repræsenterer en Person og indeholder primærnøglen CPR som er unik for hver Person i databasen samt generelle informationer om Personen herunder navn, køn, fødselsdato, telefonnummer og mail. Persons har en 1-1 relation med Doctors og Patients tabellerne som indeholder CPR nøglen som både fremmede og primærnøgle. Disse to tabeller har relationen 1 - * til Prescription tabellen som holder styr på hvilken medicin der skal udleveres samt hvor mange udleveringer der er resterende.

Prescription tabellen har som primærnøgle *prescription_id* og tre fremmednøgler herunder *patient_id*, *doctor_id* og *delivered_id* den sidstnævnte fremmednøgle stammer fra Delivered tabellen som indeholder *delivered_id* som primærnøgle, en fremmednøgle med *pharmacy_id* og en date til at registrere hvornår medicinen er udleveret til et Pharmacy. Relationen mellem Prescription og Delivered er 1-* og Delivered til Pharmacy er *-1. Pharmacy tabellen indeholder fremmednøglen *address_id* som peger til Address tabellen med relationen 1-1. Address tabellen indeholder en primærnøgle *adress_id* og en fremmednøgle *postalCode* som tabellen får fra Cities tabellen med relationen *-1.

Tanker bag dette database design har været at overholde tredje normalform og kun have tabeller med relevant data for tabellerne. Tredje normalform bliver opfyldt ved nedenstående krav:

- Ingen kolonne må gentages en andens værdi (atomare værdier)
- Hvis en tabel har en sammensat nøgle skal alle felter der ikke indgår i nøglen afhænge af den samlede nøgle (ingen partiel funktionelt afhængighed)
- Ingen felter uden for primærnøglen der er indbyrdes afhængige (ingen transitive funktionelle afhængigheder mellem non-prime attributter).

Prescription

I filen *functions.sql* i GitHub repository forekommer funktionen `create_prescription` denne funktion tilføjer en recept til databasen.
Funktionen tager følgende parametre:

- *patient_id, doctor_id, description, medicin_name* som alle er af typen varchar og dermed indeholder input i form af en tekstreng
- *handouts* af typen int som er den simple datatype til heltal.
- *expire_date* af typen timestamp som benyttes til at markere datoen for hvornår recepten udløber.

`Create_prescription()` bør kaldes hver gang en læge laver en ny recept på medicin til en patient. Funktionen udfører et simpelt, men effektivt INSERT statement til at indsætte alle værdierne fra parametrene ind i INSERT QUERY og dermed tilføje en ny recept til databasen.
Som nævnt i tidligere afsnit om database strukturen er der flere fremmednøgler og dermed begrænsninger (*FK constraints*) for om lægen og patienten eksisterer. Hvis lægen eller patienten's ID ikke eksisterer kan der ikke oprettes en recept.

Deliver funktioner

I filen *deliver.sql* fundet i vedlagte GitHub repository, er der en funktion `deliverPrescription()` som tager tre parameter; *pat_id*, *med_name* og *phar_name.*
Denne funktion har til formål at opdatere handouts i prescription tabellen, og derefter indsætte en ny række i delivered tabellen. I denne funktion bliver der altså benyttet UPDATE metoden til at tælle antal(handouts) ned, for hver udlevering af en prescription. Det er denne værdi, som senere afgører om der sendes en mail ud til patienten, om at de mangler medicin.
I INSERT metoden for delivered, bliver der benyttet `NOW()` metoden, for at angive tidspunktet, hvorpå en prescription blev afhentet. For yderligere at optimere sql for databasen, er der lavet et Index på person(cpr), for at optimere databasekald, der henter information om personer. Var der lavet funktion til at tilgå specifik information, havde vi benyttet CREATE ROLE og GRANT funktionerne, til at give specifik adgang til doktor og patienter, så kun en specifik rolle kunne opdatere og indsætte i databasen.

Random Data funktioner

For at facilitere test af performance og funktionalitet af systemet har vi udviklet en samling af PostgreSQL funktioner til at generere tilfældig test-data. Den data vi skal bruge inkluderer et fornavn, efternavn, køn, telefonnummer, email, og cpr-nummer. Derudover skal hver entry tildeles en rolle som doctor, patient eller begge.

Som base til den tilfældige data bruger vi 200 entries fra Random Data Generator (randat.com), omformet til et SQLScript via SQLizer (denne base findes i names.sql scriptet.) Vi bruger følgende funktioner til at generere test-data:

- `generate_name` - Tager et fornavn og et efternavn fra to tilfældige entries i base-tabellen.
- `generate_random_gender` - returnerer enten 'Male' eller 'Female'.
- `generate_birthdate` - som tager en alder, udregner fødselsåret og tilføjer en tilfældig dato fra det år.
- `generate_cpr` - kalder følgende funktioner og returnerer et cpr-nummer
    - `format_birthdate` - formaterer fødselsdagen i cpr-formatet
    - `generate_last_digits` - kalder følgende funktioner og returnerer de fire sidste cifre i CPR-nummeret
        - `generate_three_digits` - returnerer et tal mellem 1000 og 9990, hvis sidste ciffer er 0.
        - `generate_last_digit` - returnerer et tilfældigt mellem 0 og 10. Et lige tal hvis personen er en kvinde, et tilfældigt ulige tal hvis personen er en mand, eller et vilkårligt tal hvis de er ingen af delene.
- `generate_person` - bruger ovenstående funktioner til at generere én person
- `generate_people` - bruger `generate_person` til at generere et vilkårligt antal personer.
- `set_doctor_or_patient` - går igennem alle entries i persons tabellen og tildeler dem enten doctor, patient eller begge roller.

For at køre funktionerne, og oprette test-dataen køres scriptet *inserting-people-into-table.sql*. Et problem vi stødte på er dog at randat.com kun kan generere aldre indenfor et begrænset interval, og vi støder derfor overraskende hurtigt på cpr-nummer kollisioner. Dette sætter en grænse for hvor mange testbrugere vi kan lave. I fremtiden vil vi lade være med at bruge alderen fra basen, og bare generere et vilkårligt år, så vi kan udvide antallet af entries.

Email Reminder

Vi har valgt at benytte python 3, til at hente information fra vores database, og udsende mails. Koden for dette, findes i filen *sendEmail.py*. Til dette har vi blandt andet benyttet python iblioteket *psycopg2* til at tilgå vores database, for at få information om antal prescription, og hvilken bruger samt doktor der er relevant for emailen. Hvis en prescription har to eller under handouts tilbage, bliver der sendt en mail til patienten om hvilket medicin der er mangel på, samt hvilken doktor der skal kontaktes. I sql'en der henter denne information, er der benyttet JOINS frem for subselects, for at optimere databasekaldet, på vores normaliserede database.

Da vi har postgres kørende på et docker miljø, installerede *python3-pip, libpq* og *psycopg2* på docker miljøet. *sendEmail.py* kan nu køres igennem docker. Python scriptet henter alle patienter, hvis prescription snart er tom, og sender en mail til den specifikke patients behov. Til at teste dette, har vi benyttet en stmp service *wpoven*[1]. For automatisk at lave dette tjek, kan der blive lavet et *Cron*[2] job, der automatisk sender en tjekker og sender en mail ud, én gang om dagen. Sådan en funktion har postgres, med *cron.schedule()* metoden.

[1] https://www.wpoven.com/tools/free-smtp-server-for-testing
[2] What is Cron Job? - Cron Jobs and Scheduled Tasks - Hivelocity Hosting