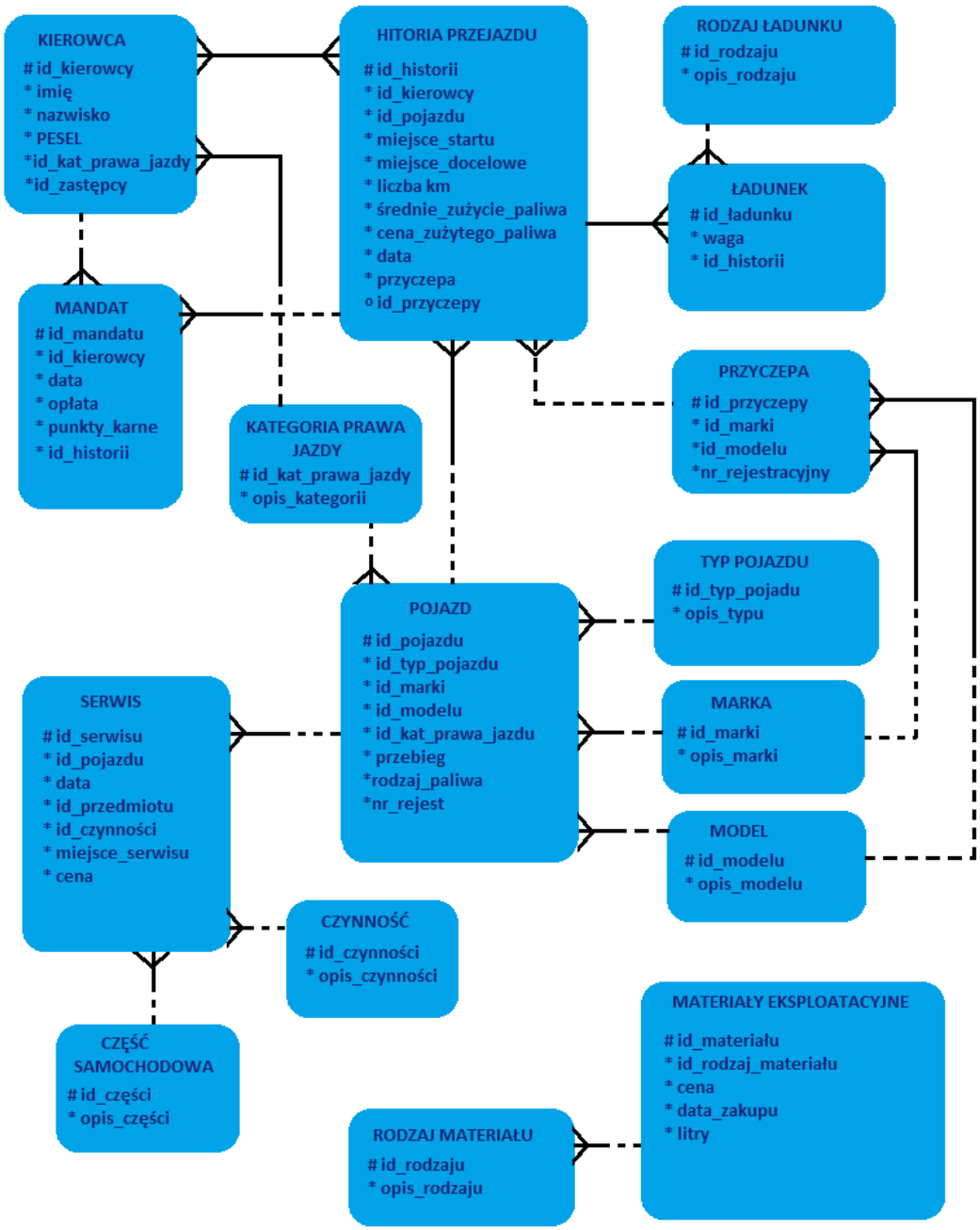
Dokumentacja projektu
System do zarządzania flotą samochodową

23 czerwca 2017

1 Model pojęciowy

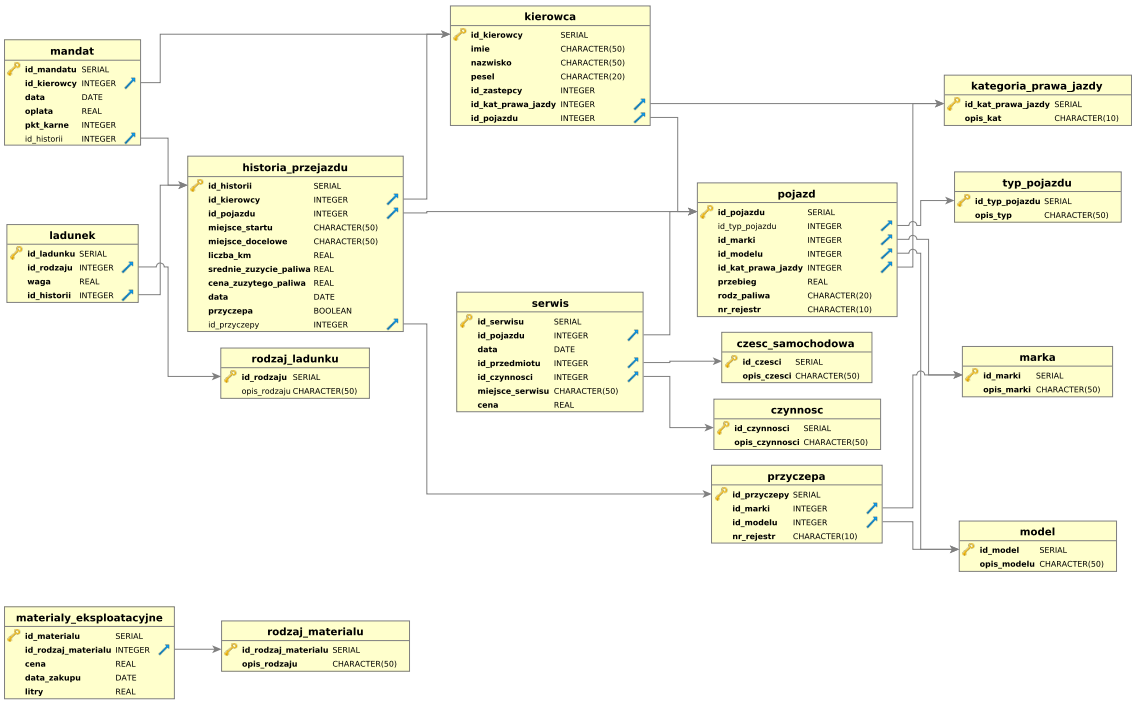
Przy tworzeniu modelu pojęciowego przeanalizowano strukturę i działanie firmy zarządzające flotą samochodową. Firma zatrudnia kierowców, którzy udają się w trasy, wioząc ze sobą ładunek. Poza głównym pojazdem kierowca może mieć doczepioną przyczepę. Każdy pojazd można opisać przez jego typ, markę i model. Każdy pojazd ma wyznaczoną kategorię prawa jazdy, która trzeba posiadać, aby móc nią kierować. Pojazd może być poddany serwisowaniu, w czasie którego części samochodu mogą być wymieniona lub naprawione albo przeglądowi technicznemu. Firma chciałaby też prowadzić rejestr mandatów, które wystawiono kierowcom, w czasie odbywania trasy. Poza tym utrzymanie floty samochodowej wymaga zakupu materiałów eksploatacyjnych, których aktualny stan oraz historia zakupu także powinny być zapisywane.

Biorąc pod uwagę powyższy opis firmy i pytania, na które klient chciałby uzyskać odpowiedź z bazy danych stworzono następujący model pojęciowy. W tworzeniu modelu pojęciowego silnie korzystano z tabel słownikowych, co pozwala na łatwą edycję utworzonych nazw (np. opisu czynności serwisowania), a także na przyszłą rozbudowę aplikacji o nowe kategorie.


KIEROWCA
id_kierowcy
* imię
* nazwisko
* PESEL
*id_kat_prawa_jazdy
*id_zastępcy
HITORIA PRZEJAZDU
id_historii
* id_kierowcy
* id_pojazdu
* miejsce_startu
* miejsce_docelowe
* liczba km
* średnie_zużycie_paliwa
* cena_zużytego_paliwa
* data
* przyczepa
o id_przyczepy
RODZAJ ŁADUNKU
id_rodzaju
* opis_rodzaju
ŁADUNEK
id_ładunku
* waga
* id_historii
MANDAT
id_mandatu
* id_kierowcy
* data
* opłata
* punkty_karne
* id_historii
KATEGORIA PRAWA JAZDY
id_kat_prawa_jazdy
* opis_kategorii
PRZYCZEPA
id_przyczepy
* id_marki
*id_modelu
*nr_rejestracyjny
TYP POJAZDU
id_typ_pojadu
* opis_typu
POJAZD
id_pojazdu
* id_typ_pojazdu
* id_marki
* id_modelu
* id_kat_prawa_jazdu
* przebieg
*rodzaj_paliwa
*nr_rejest
SERWIS
id_serwisu
* id_pojazdu
* data
* id_przedmiotu
* id_czynności
* miejsce_serwisu
* cena
MARKA
id_marki
* opis_marki
MODEL
id_modelu
* opis_modelu
CZYNNOŚĆ
id_czynności
* opis_czynności
CZĘŚĆ SAMOCHODOWA
id_części
* opis_części
MATERIAŁY EKSPLOATACYJNE
id_materiału
* id_rodzaj_materiału
* cena
* data_zakupu
* litry
RODZAJ MATERIAŁU
id_rodzaju
* opis_rodzaju

2 Model relacyjny

Implementując model pojęciowy do modelu relacyjnego zdenormalizowano model bazy danych w celu optymalizacji zapytań. Przede wszystkim do tabeli kierowca dodano kolumnę id_pojazdu. W modelu pojęciowym tabela kierowca i pojazd były połączone przez tabelę historia_przejazdu, ponieważ kierowca i jego pojazd tworzą parę, kiedy udają się w konkretną trasę. Ponieważ jednak w analizowanej firmie każdy kierowca korzysta z tylko jednego pojazdu, który można mu przypisać, a zapytania wykorzystujące połączenie kierowca i jego pojazd będą wykonywane bardzo często, zdecydowano się zmodyfikować tabelę kierowca i wprowadzić do niej klucz obcy id_pojazdu. Dzięki w zapytaniach, w których potrzebne będą dane kierowcy i pojazdu uniknie się dodatkowego załączania tabeli historia_przejazdu.


kierowca
id_kierowcy SERIAL
imie CHARACTER(50)
nazwisko CHARACTER(50)
pesel CHARACTER(20)
id_zastepcy INTEGER
id_kat_prawa_jazdy INTEGER
id_pojazdu INTEGER
mandat
id_mandatu SERIAL
id_kierowcy INTEGER
data DATE
oplata REAL
pkt_karne INTEGER
id_historii INTEGER
kategoria_prawa_jazdy
id_kat_prawa_jazdy SERIAL
opis_kat CHARACTER(10)
historia_przejazdu
id_historii SERIAL
id_kierowcy INTEGER
id_pojazdu INTEGER
miejsce_startu CHARACTER(50)
miejsce_docelowe CHARACTER(50)
liczba_km REAL
srednie_zuzycie_paliwa REAL
cena_zuzytego_paliwa REAL
data DATE
przyczepa BOOLEAN
id_przyczepy INTEGER
pojazd
id_pojazdu SERIAL
id_typ_pojazdu INTEGER
id_marki INTEGER
id_modelu INTEGER
id_kat_prawa_jazdy INTEGER
przebieg REAL
rodz_paliwa CHARACTER(20)
nr_rejestr CHARACTER(10)
typ_pojazdu
id_typ_pojazdu SERIAL
opis_typ CHARACTER(50)
ladunek
id_ladunku SERIAL
id_rodzaju INTEGER
waga REAL
id_historii INTEGER
rodzaj_ladunku
id_rodzaju SERIAL
opis_rodzaju CHARACTER(50)
serwis
id_serwisu SERIAL
id_pojazdu INTEGER
data DATE
id_przedmiotu INTEGER
id_czynnosci INTEGER
miejsce_serwisu CHARACTER(50)
cena REAL
czesc_samochodowa
id_czesci SERIAL
opis_czesci CHARACTER(50)
marka
id_marki SERIAL
opis_marki CHARACTER(50)
czynnosc
id_czynnosci SERIAL
opis_czynnosci CHARACTER(50)
przyczepa
id_przyczepy SERIAL
id_marki INTEGER
id_modelu INTEGER
nr_rejestr CHARACTER(10)
model
id_model SERIAL
opis_modelu CHARACTER(50)
materialy_eksploatacyjne
id_materialu SERIAL
id_rodzaj_materialu INTEGER
cena REAL
data_zakupu DATE
litry REAL
rodzaj_materialu
id_rodzaj_materialu SERIAL
opis_rodzaju CHARACTER(50)

3 Model fizyczny i implementacja

W projekcie skorzystano z wolnodostępnego systemu zarządzania relacyjnymi bazami danych PosgreSQL. Na podstawie wcześniej ustalonego modelu, zgodnie z jego składnią, utworzono tabele (init.sql). Komunikację pomiędzy aplikacją a bazą danych zrealizowano z wykorzystaniem biblioteki `java.sql` (szczegółowe informacje na temat działaia aplikacji znajdują się w sekcji *Aplikacja - dokumentacja użytkownika*).

4 Testy

W celu testowania bazy danych napisano skrypt tworzący pliki `.csv` (generator), dla każdej tabeli, a następnie przekopiowano ich zawartość (init.sh).

```
\copy marka(opis_marki) FROM $DIR/csvki/marka.csv DELIMITER ',' CSV;
```

Następnie przeprowadzono testy wydajnościowe. Na ich potrzeby utworzono analogiczną bazę danych - jedyną różnicą było nałożenie indeksów na kolumny wykorzystywane do sortowania w przygotowanych zapytaniach (plik init_index.sql).

Porównano czas wykonywania poszczególnych zapytań dla bazy bez i z indeksami. Średni czas wykonania zapytania dla bazy danych bez wykorzystania indeksów wynosi 0.0242 sekundy, natomiast dla bazy z nałożonymi indeksami ten czas jest równy 0.0104 sekundy. Oznacza to, że indeksy pozwalają na przyśpieszenie odpowiedzi bazy danych o 43%, co w rzeczywistości odpowiada zwiększeniu szybkości o zaledwie 0.0138 sekundy. Z punktu widzenia użytkownika ta różnica jest niezauważalna, natomiast wykorzystanie indeksów łączy się dodatkowym nakładem pamięciowym.

5 Aplikacja - dokumentacja użytkownika

W ramach systemu zarządzania flotą samochodową stworzono aplikację okienkową działającą pod systemem Linux. Aplikacja pozwala na dodawanie i usuwanie rekordów, wyświetlanie całych tablic oraz odpowiedzi na konkretne zapytania.

5.1 Uruchamianie

Przed pierwszym uruchomieniem programu, na maszynie pełniącej rolę serwera SQL, należy stworzyć bazę danych i wszystkie jej tabele. W tym celu należy uruchomić skrypt, wpisując w konsoli

```
./init.sh
```

Podczas jego wykonania sprawdzane jest czy na komputerze zainstalowane są wymagane programy (Java i PostgreSQL) i w razie potrzeby są one instalowane. Dodatkowo importowane są dane z plików CSV.

Skrypt trzykrotnie prosi o podanie hasła: za pierwszym razem hasła administratora, następnie hasła do bazy danych. Nazwę tworzonego użytkownika, bazy danych oraz jej hasło dostępu można wybrać zmieniając początkowe linijki skryptu `init.sh`:

```
USER="bd2"
PASS="bd2"
DATABASE="bd2"
```

Domyślnie ustawione wartości to: użytkownik **bd2**, nazwa bazy danych **bd2**, hasło dostępu **bd2**.

Po pomyślnym utworzeniu bazy danych można uruchomić aplikację z odpowiednimi parametrami, wpisując w terminalu:

```
java -jar carcompany.jar [host] [uzytkownik] [nazwa bazy] [haslo]
```

gdzie *host* jest adresem IP maszyny, na której utworzona jest baza danych (jeśli aplikacja znajduje się na tym samym komputerze co baza danych wystarczy *localhost*).

5.1.1 Obsługa błędu FATAL: Peer authentication failed

W przypadku pojawienia się błędu: `FATAL: Peer authentication failed for user ...` należy w pliku `pg_hba.conf` (znajdującym się najprawdopodobniej w `/etc/postgresql/9.x/main/`) zmienić linijkę

```
# TYPE DATABASE USER ADDRESS METHOD
local   all        all            peer
```

na

```
# TYPE DATABASE USER ADDRESS METHOD
local   all        all            md5
```

Zamiana `peer` na `md5` gwarantuje, że PostgreSQL będzie zawsze pytało o hasło.