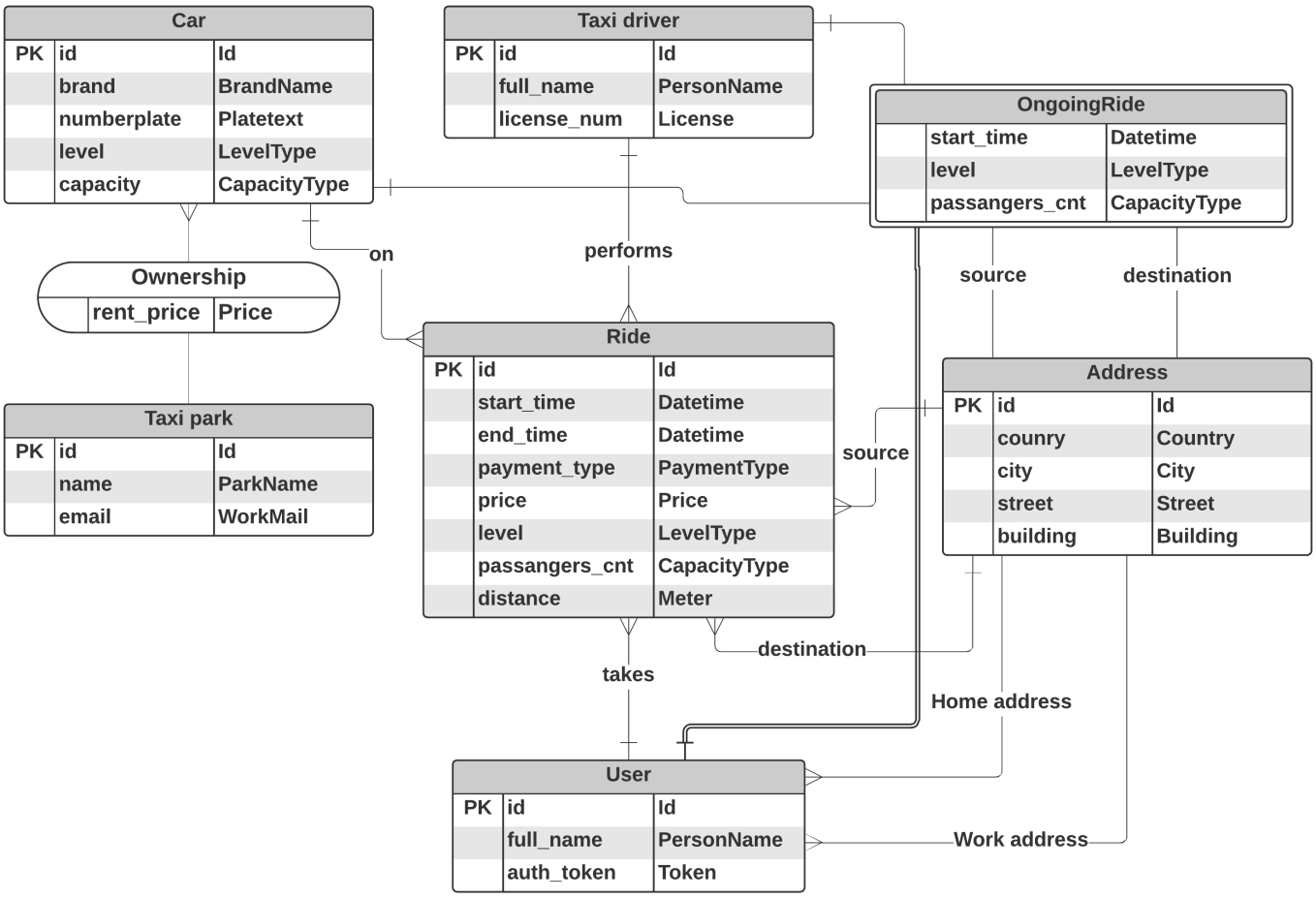
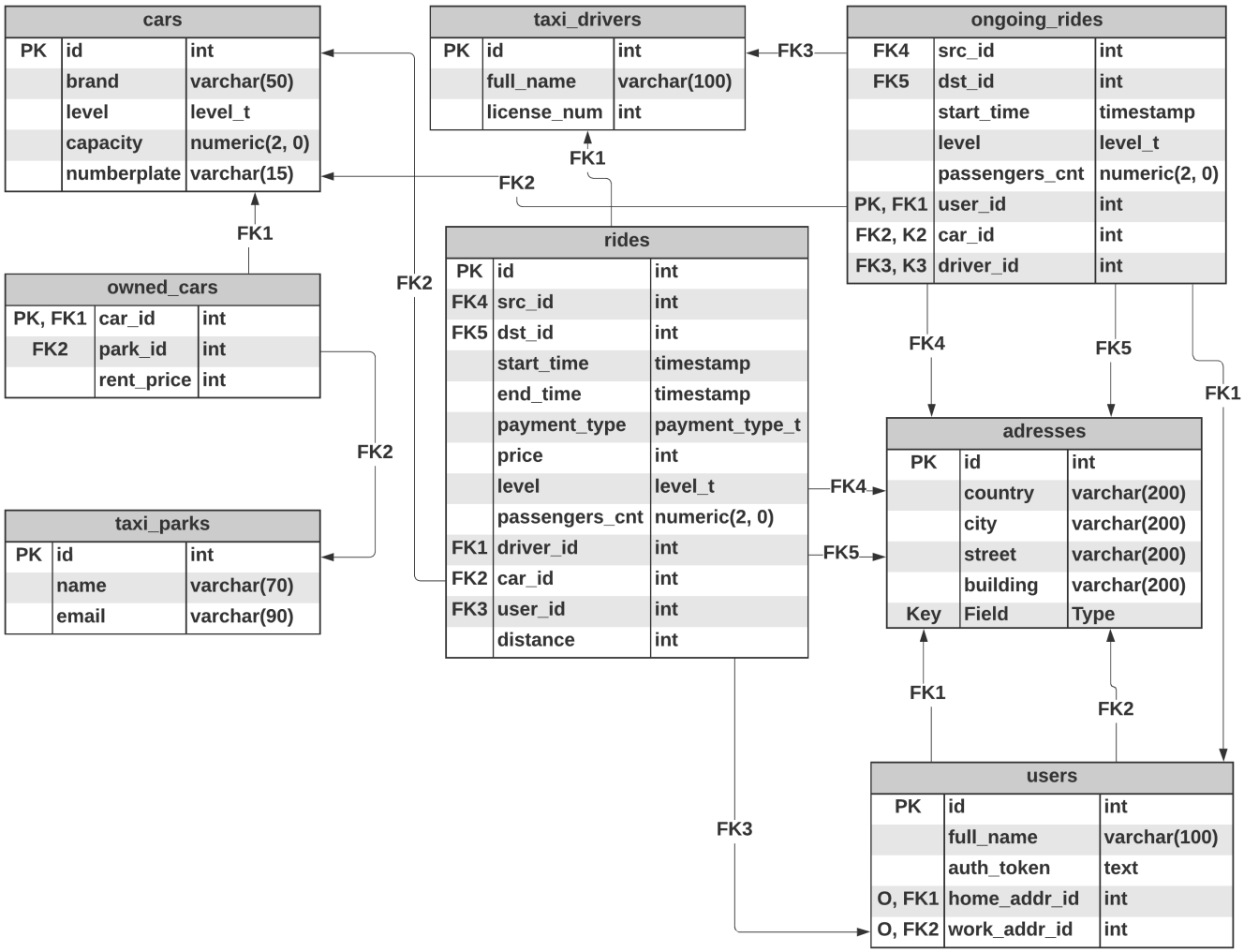
Курсовая работа

Лев Довжик, M3439

1 ERM



2 Физическая модель



3 Функциональные зависимости

3.1 cars

В данном отношении имеются 4 нетривиальные ФЗ: $id \to brand$, $id \to level$, $id \to capacity$, $id \to numberplate$. Остальные ФЗ получаются из данных и тривиальных по правилам вывода.

В связи с тем, что замыкание id с данным ФЗ и самоопределением даёт все атрибуты, а само множество неприводимо, то оно эквивалентно всем ФЗ данного отношения, а id является ключом.

3.2 owned_cars

В данном отношении имеются 2 нетривиальные ФЗ: $car\_id \to park\_id$, $car\_id \to rent\_price$. Остальные ФЗ получаются из данных и тривиальных по правилам вывода.

В связи с тем, что замыкание car_id с данным ФЗ и самоопределением даёт все атрибуты, а

само множество неприводимо, то оно эквивалентно всем ФЗ данного отношения, а car_id является ключом.

3.3 users

В данном отношении имеются 1 нетривиальная ФЗ: $id \to full\_name, id \to auth\_token, id \to home\_addr\_id, id \to work\_addr\_id$. Остальные ФЗ получаются из данной и тривиальных по правилам вывода.

В связи с тем, что замыкание id с данным ФЗ и самоопределением даёт все атрибуты, а само множество неприводимо, то оно эквивалентно всем ФЗ данного отношения, а id является ключом.

3.4 taxi_drivers

В данном отношении имеются 2 нетривиальные ФЗ: $id \to full\_name, id \to licens\_num$. Остальные ФЗ получаются из данных и тривиальных по правилам вывода.

В связи с тем, что замыкание id с данным ФЗ и самоопределением даёт все атрибуты, а само множество неприводимо, то оно эквивалентно всем ФЗ данного отношения, а id является ключом.

3.5 addresses

В данном отношении имеются 1 нетривиальная ФЗ: $id \to name$. Остальные ФЗ получаются из данной и тривиальных по правилам вывода.

В связи с тем, что замыкание id с данным ФЗ и самоопределением даёт все атрибуты, а само множество неприводимо, то оно эквивалентно всем ФЗ данного отношения, а id является ключом.

3.6 rides

В данном отношении все нетривиальные ФЗ имеют вид $id \to ?$. Остальные ФЗ получаются из данных и тривиальных по правилам вывода.

В связи с тем, что замыкание id с данным ФЗ и самоопределением даёт все атрибуты, а само множество неприводимо, то оно эквивалентно всем ФЗ данного отношения, а id является ключом.

3.7 ongoning_rides

В данном отношении, во первых все атрибуты функционально зависят от $user\_id$, значит замыкание его с самоопределением даёт все атрибуты, а самом он является ключом. Однако так же в тройке $user\_id, car\_id$ и $driver\_id$ любой из атрибутов однозначно определяет остальные два. В таком случае каждый из них является ключём.

4 Нормализация

1 нормальная форма

Во всех наших отношениях нет повторяющихся групп, все атрибуты атомарны и присутствует ключ, значит, все они находятся в 1 н.ф.

2 нормальная форма

Все отношения в 1 н.ф., а так же выше было показано, что все неключевые атрибуты зависят от целого ключа, значит, все они в 2 н.ф.

3 нормальная форма

Все отношения во 2 н.ф., а так же было показано выше, что все неключевые атрибуты напрямую зависят от ключей, значит все они в 3 н.ф.

Нормальная форма Бойса-Кодда

Заметим, что во всех наших отношениях все нетривиальны функциональные зависимости имели вид $\langle soeme\_id \rangle \rightarrow ?$ или же выводились с их участием. Следовательно их левая часть является надмножеством ключа, т.е. надключом. Отсюда делаем вывод, что все отношения находятся в НФБК.

4 нормальная форма

В каждом из этих отношений есть простой ключ($\langle soeme\_id \rangle$) и они находятся в НФБК, следовательно, по 2 теореме Дейта-Фейгина они находятся в 4 н.ф.

5 нормальная форма

В всех отношениях все ключи имеют вид $\langle soeme\_id \rangle$ т.е является простыми. Во всех отношениях, кроме $ongoing_calls$ все нетривиальные ФЗ содержат его лишь в левой части, значит замыкание остальных атрибутов не будет содержать id. Следовательно он является частью любого ключа, а так как он сам по себе ключ, то он является единственным ключом причём простым. В $ongoing_calls$ так же логика верна, если убрать три известных простых ключа, т.е. они являются единственными ключами.

В таком случае, в силу того, что все эти отношения находятся в 3 н.ф., по первой теореме Дейта-Фейгина, они так же находятся в 5 н.ф.

5 База данных в PostgeSQL

5.1 Объявление

```
-- CREATE DATABASE taxi;

CREATE TYPE payment_type_t AS ENUM ('cash', 'card', 'bitcoin');
CREATE TYPE level_t AS ENUM ('economy', 'comfort', 'lux');

CREATE TABLE addresses
(
  id       INT PRIMARY KEY,
  country  VARCHAR(200) NOT NULL,
  city     VARCHAR(200) NOT NULL,
  street   VARCHAR(200) NOT NULL,
  building VARCHAR(200) NOT NULL
);

CREATE TABLE taxi_drivers
(
  id          INT PRIMARY KEY,
  full_name   VARCHAR(100) NOT NULL,
  license_num INT          NOT NULL
);

CREATE TABLE taxi_users
(
  id           INT PRIMARY KEY,
  full_name    VARCHAR(100) NOT NULL,
  auth_token   TEXT         NOT NULL,
  home_addr_id INT REFERENCES addresses (id),
  work_addr_id INT REFERENCES addresses (id)
);

CREATE TABLE cars
(
  id          INT PRIMARY KEY,
  brand       VARCHAR(50)   NOT NULL,
```

```
  level       level_t       NOT NULL,
  capacity    NUMERIC(2, 0) NOT NULL,
  numberplate VARCHAR(15)   NOT NULL
);

CREATE TABLE taxi_parks
(
  id    INT PRIMARY KEY,
  name  VARCHAR(50) NOT NULL,
  email VARCHAR(60) NOT NULL
);

CREATE TABLE owned_cars
(
  car_id     INT PRIMARY KEY REFERENCES cars (id),
  park_id    INT NOT NULL REFERENCES taxi_parks (id),
  rent_price INT NOT NULL
);

CREATE TABLE rides
(
  id             INT PRIMARY KEY,
  src_id         INT            NOT NULL REFERENCES addresses (id),
  dst_id         INT            NOT NULL REFERENCES addresses (id),
  start_time     TIMESTAMP      NOT NULL,
  end_time       TIMESTAMP      NOT NULL,
  payment_type   payment_type_t NOT NULL,
  price          INT            NOT NULL,
  level          level_t        NOT NULL,
  passengers_cnt NUMERIC(2, 0)  NOT NULL,
  driver_id      INT            NOT NULL REFERENCES taxi_drivers (id),
  car_id         INT            NOT NULL REFERENCES cars (id),
  user_id        INT            NOT NULL REFERENCES taxi_users (id),
  distance       INT            NOT NULL
);

CREATE TABLE ongoing_rides
```

```
(
  src_id         INT           NOT NULL REFERENCES addresses (id),
  dst_id         INT           NOT NULL REFERENCES addresses (id),
  start_time     TIMESTAMP,
  level          level_t       NOT NULL,
  passengers_cnt NUMERIC(2, 0) NOT NULL,
  user_id        INT PRIMARY KEY REFERENCES taxi_users (id),
  car_id         INT UNIQUE    NOT NULL REFERENCES cars (id),
  driver_id      INT UNIQUE    NOT NULL REFERENCES taxi_drivers (id)
);
```

5.2 Ограничения

```
CREATE EXTENSION btree_gist;

-- Ride's timestamps must be correct
ALTER TABLE rides
  ADD CHECK ( start_time < end_time );

-- Can't have intersected rides on same car
ALTER TABLE rides
  ADD EXCLUDE USING gist (
    car_id WITH =,
    tsrange(start_time, end_time, '[]') WITH &&
    );

-- Can't have intersected rides with same driver
ALTER TABLE rides
  ADD EXCLUDE USING gist (
    driver_id WITH =,
    tsrange(start_time, end_time, '[]') WITH &&
    );

-- Can't have intersected rides with same user
ALTER TABLE rides
  ADD EXCLUDE USING gist (
    user_id WITH =,
```

```
    tsrange(start_time, end_time, '[]') WITH &&
    );


-- car must have enough space and satisfy class requirements
CREATE OR REPLACE FUNCTION check_passengers() RETURNS TRIGGER
AS
$check_passengers$
BEGIN
  IF NOT exists(
    SELECT *
    FROM cars
    WHERE cars.id = NEW.car_id
      AND capacity >= NEW.passengers_cnt
      AND cars.level >= NEW.level
    ) THEN
    RAISE EXCEPTION 'Non-suitable car';
  ELSE
    RETURN NEW;
  END IF;
END;
$check_passengers$ LANGUAGE plpgsql;

CREATE TRIGGER check_passengers_complete
  BEFORE INSERT OR UPDATE
  ON rides
  FOR EACH ROW
EXECUTE PROCEDURE check_passengers();

CREATE TRIGGER check_passengers_ongoing
  BEFORE INSERT OR UPDATE
  ON ongoing_rides
  FOR EACH ROW
EXECUTE PROCEDURE check_passengers();

-- there can't be ongoing ride started after current
CREATE OR REPLACE FUNCTION check_ongoing_ride() RETURNS TRIGGER
```

```
AS
$check_ongoing_ride$
BEGIN
  IF exists(
    SELECT *
    FROM ongoing_rides o_r
    WHERE (
        o_r.user_id = NEW.user_id
        OR o_r.driver_id = NEW.driver_id
        OR o_r.car_id = NEW.car_id
      )
      AND o_r.start_time <= NEW.end_time
    ) THEN
    RAISE EXCEPTION 'Ride ends after ongoing ride';
  ELSE
    RETURN NEW;
  END IF;
END;
$check_ongoing_ride$ LANGUAGE plpgsql;

CREATE TRIGGER check_ongoing_ride
  BEFORE INSERT OR UPDATE
  ON rides
  FOR EACH ROW
EXECUTE PROCEDURE check_ongoing_ride();

-- ride can't start before newest completed ride
CREATE OR REPLACE FUNCTION check_finished_ride() RETURNS TRIGGER
AS
$check_finished_ride$
BEGIN
  IF exists(
    SELECT *
    FROM rides r
    WHERE (
        r.user_id = NEW.user_id
        OR r.driver_id = NEW.driver_id
```

```
        OR r.car_id = NEW.car_id
      )
      AND r.end_time >= NEW.start_time
    ) THEN
    RAISE EXCEPTION 'Ongoing ride starts after completed ride';
  ELSE
    RETURN NEW;
  END IF;
END;
$check_finished_ride$ LANGUAGE plpgsql;

CREATE TRIGGER check_finished_ride
  BEFORE INSERT OR UPDATE
  ON ongoing_rides
  FOR EACH ROW
EXECUTE PROCEDURE check_finished_ride();
```

5.3 Индексы

PostgreSQL создаёт древесные индексы для всех **PRIMARY KEY** и **UNIQUE**, а так же для всех **EXCLUDE** ограничений с помощью *gist*. Добавим же ещё несколько индексов для ускорения запросов:

```
-- indexes on foreign keys
CREATE INDEX ON owned_cars (car_id);
CREATE INDEX ON owned_cars (park_id);
CREATE INDEX ON rides (src_id);
CREATE INDEX ON rides (dst_id);
CREATE INDEX ON ongoing_rides (src_id);
CREATE INDEX ON ongoing_rides (dst_id);

-- boost time queries
CREATE INDEX ON rides USING btree (start_time, end_time);

-- boost level search
CREATE INDEX ON cars (level);
```

6 Заполнение БД

Так как существующие агрегаторы не делятся данным о поездках и пользователях, заполним таблицу случайными данными

```
import psycopg2
from contextlib import closing
from faker import Faker
import random
import math
import datetime

my_faker = Faker()
Faker.seed(1337)
random.seed(1337)
levels = ['economy', 'comfort', 'lux']
MAX_CAPACITY = 6


def rand_russian_letter():
    letters = "абвгежзийклмнопстуфхцчшщэюя"
    return letters[random.randint(0, len(letters) - 1)]


def rand_digit():
    return str(random.randint(0, 9))


def rand_numberplate():
    return rand_russian_letter() + \
           rand_digit() + rand_digit() + rand_digit() + \
           rand_russian_letter() + rand_russian_letter()


def rand_brand():
    brands = ['bmw', 'lada', 'mercedes', 'audi', 'toyota', 'honda', 'opel', 'ford']
    return brands[random.randint(0, len(brands) - 1)]


def gen_car(i):
    return i, rand_brand(), levels[random.randint(0, len(levels) - 1)], \
```

```
            random.randint(3, MAX_CAPACITY), rand_numberplate(),


def gen_park(i):
    name = my_faker.word()
    return i, name, name + "@gmail.com"


def gen_address(i):
    x = random.randint(1, 5000)
    y = random.randint(1, 5000)
    country = my_faker.country()
    city = my_faker.city()
    street = my_faker.street_name()
    building = my_faker.building_number()
    return (i, country, city, street, building), (x, y)


def gen_driver(i):
    return i, my_faker.name(), random.randint(1, 10000000)


def gen_user(i):
    return i, my_faker.name(), my_faker.word()


parks = list(map(gen_park, range(1, 100)))
users = list(map(gen_user, range(1, 3000)))
drivers = list(map(gen_driver, range(1, 1000)))
addresses = list(map(gen_address, range(1, 1000)))
cars = list(map(gen_car, range(1, 1000)))


def gen_owned_car(i):
    owner = random.randint(0, len(parks) - 1)
    return i, parks[owner][0], random.randint(500, 1000)


def gen_ride(i):
    payment_types = ['cash', 'card', 'bitcoin']
```

```
    src = random.randint(0, len(addresses) - 1)
    dst = random.randint(0, len(addresses) - 1)
    while src == dst:
        dst = random.randint(0, len(addresses) - 1)
    src_id = addresses[src][0][0]
    dst_id = addresses[dst][0][0]
    p1 = addresses[src][1]
    p2 = addresses[dst][1]
    dist = int(math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2))
    payment = payment_types[random.randint(0, len(payment_types) - 1)]
    car = random.randint(0, len(cars) - 1)
    level = levels[random.randint(0, levels.index(cars[car][2]))]
    passengers_cnt = random.randint(1, cars[car][3])
    price = dist * (levels.index(level) + 1)
    driver_id = drivers[random.randint(0, len(drivers) - 1)][0]
    user_id = users[random.randint(0, len(users) - 1)][0]
    length = random.randint(5, 40)
    car_id = cars[car][0]
    start_time = my_faker.date_time_this_month()
    end_time = start_time + datetime.timedelta(minutes=length)
    start_str = start_time.isoformat(sep=' ')
    end_str = end_time.isoformat(sep=' ')
    return i, src_id, dst_id, start_str, end_str, \
           payment, price, level, passengers_cnt, driver_id, car_id, user_id, dist


print('Enter username')
username = input()
print('Enter password')
password = input()
with closing(psycopg2.connect(dbname='taxi',
                              user=username,
                              password=password,
                              host='localhost')) as conn:
    conn.autocommit = True
    with conn.cursor() as cursor:
        for car in cars:
            cursor.execute(
                '''INSERT INTO cars
                (id, brand, level, capacity, numberplate)
```

```
            VALUES
            (%s, %s, %s, %s, %s)''',
            car
        )
    for park in parks:
        cursor.execute(
            '''INSERT INTO taxi_parks
            (id, name, email)
            VALUES
            (%s, %s, %s)''',
            park
        )
    for user in users:
        cursor.execute(
            '''SELECT add_user(%s, %s, %s)''',
            user
        )
    for driver in drivers:
        cursor.execute(
            '''INSERT INTO taxi_drivers
            (id, full_name, license_num)
            VALUES
            (%s, %s, %s)''',
            driver
        )
    for address in addresses:
        cursor.execute(
            '''INSERT INTO addresses
            (id, country, city, street, building)
            VALUES
            (%s, %s, %s, %s, %s)''',
            address[0]
        )
    for car_id, *_ in cars:
        owned_car = gen_owned_car(car_id)
        cursor.execute(
            '''INSERT INTO owned_cars
            (car_id, park_id, rent_price)
            VALUES
            (%s, %s, %s)''',
```

```
                owned_car
            )
        failed_inerts = 0
        for i in range(1, 6000):
            ride = gen_ride(i)
            try:
                cursor.execute(
                    '''INSERT INTO rides
                    (id, src_id, dst_id, start_time, end_time, payment_type, price,
                    level, passengers_cnt, driver_id, car_id, user_id, distance)
                    VALUES
                    (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)''',
                    ride
                )
            except:
                failed_inerts += 1
        print("Failed {} inserts".format(failed_inerts))
```

При запуске скрипта не удалось всего 82 вставки поездок.

7 Функции для работы и запросы

Добавление пользователя

Наша база данных хранит пароли в зашифрованном виде, для удобства работы вся работа по их шифрованию лежит на плечах это базы, которая и предоставляет API:

```
CREATE EXTENSION pgcrypto;

CREATE OR REPLACE FUNCTION add_user(id_arg INT,
                                    name_arg VARCHAR(100),
                                    pass_arg TEXT)
  RETURNS BOOLEAN
AS
$$
DECLARE
  affected_rows INT;
BEGIN
  INSERT INTO taxi_users (id, full_name, auth_token)
```

```
  VALUES (id_arg, name_arg, crypt(pass_arg, gen_salt('bf')))
  ON CONFLICT (id) DO NOTHING;
  GET DIAGNOSTICS affected_rows = ROW_COUNT;
  RETURN affected_rows > 0;
END;
$$ LANGUAGE plpgsql;
```

Изменение пользовательских адресов

У пользователя есть возможность сохранить в быстрый доступ домашний и рабочий адрес, однако кто попало не может их изменять, для этого есть специальные функции производящие все проверки:

```
-- returns if change happened
-- null address arg means delete
CREATE OR REPLACE FUNCTION modify_home_address(user_id_arg INT,
                                               pass_arg TEXT,
                                               home_addr_arg INT)
  RETURNS BOOLEAN
AS
$$
DECLARE
  affected_rows INT;
BEGIN
  UPDATE taxi_users
  SET home_addr_id = home_addr_arg
  WHERE id = user_id_arg
    AND auth_token = crypt(pass_arg, auth_token);
  GET DIAGNOSTICS affected_rows = ROW_COUNT;
  RETURN affected_rows > 0;
END;
$$ LANGUAGE plpgsql;

-- returns if change happened
-- null address arg equals delete
CREATE OR REPLACE FUNCTION modify_work_address(user_id_arg INT,
                                               pass_arg TEXT,
                                               work_addr_arg INT)
```

```
  RETURNS BOOLEAN
AS
$$
DECLARE
  affected_rows INT;
BEGIN
  UPDATE taxi_users
  SET work_addr_id = work_addr_arg
  WHERE id = user_id_arg
    AND auth_token = crypt(pass_arg, auth_token);
  GET DIAGNOSTICS affected_rows = ROW_COUNT;
  RETURN affected_rows > 0;
END;
$$ LANGUAGE plpgsql;
```

Средняя скорость

Одной из полезных характеристик водителя является скорость его вождения. Кому-то из пользователей нравятся спокойные поездки по городу, другим же интересно как можно быстрее добраться из точки А в точку Б. Заведём же **view** для этой характеристики:

```
-- km/h
CREATE VIEW average_speed AS
(
SELECT driver_id,
       full_name,
       avg(distance * 3.6 / extract(EPOCH FROM (end_time - start_time)))
         AS speed
FROM rides
     JOIN taxi_drivers on driver_id = taxi_drivers.id
GROUP BY driver_id, full_name
  );
```

Количество апгрейдов за период

На заказ с определённым уровнем комфорта может откликнуться любая машина с большим либо равным уровнем. Однако для такси это является неоптимальным расходованием ресурсов. Хочется уметь получать эту информацию, чтобы правильно оценивать количество машин каждого класса,

которые нам нужны сейчас на линии, для этого заведём **view** со всеми апгрейдами, из которого будет получать статистику за нужный период:

```
CREATE VIEW upgrades AS
(
SELECT rides.level AS requested_level,
       start_time AS time
FROM rides
     JOIN cars ON rides.car_id = cars.id
WHERE rides.level < cars.level
  );
```

```
CREATE OR REPLACE FUNCTION upgrades_in_period(start_arg TIMESTAMP,
                                              end_arg TIMESTAMP)
  RETURNS TABLE
          (
            requested_level level_t,
            cnt             BIGINT
          )
  IMMUTABLE
AS
$$
BEGIN
  RETURN QUERY
    SELECT upgrades.requested_level,
           count(1) AS cnt
    FROM upgrades
    WHERE time BETWEEN start_arg AND end_arg
    GROUP BY upgrades.requested_level;
END;
$$ LANGUAGE plpgsql;
```

Распределение способов оплаты

Полезно знать, как часто чем пользуется пользователи, что проводить акции с партнёрами или улучшать соответствующую инфраструктур:

```
WITH payment_cnt AS (
  SELECT payment_type,
```

```
        count(1) as cnt
  FROM rides
  GROUP BY payment_type
)
SELECT payment_type,
       cnt * 100.0 / (SELECT sum(cnt) FROM payment_cnt) AS percent
FROM payment_cnt;
```

Статистика за год

Многие сервисы подводят какую-то статистику по пользователям за год, а потом показывают её им. Мы не являемся исключением:

```
-- total_distance is returned distance in km
CREATE OR REPLACE FUNCTION user_statistics_for_year(year_arg INT)
  RETURNS TABLE
          (
            id             INT,
            full_name      VARCHAR(100),
            rides_cnt      BIGINT,
            total_distance FLOAT,
            total_time     INTERVAL
          )
  IMMUTABLE
AS
$$
BEGIN
  RETURN QUERY
    WITH year_rides AS (
      SELECT *
      FROM rides
      WHERE extract(YEAR FROM start_time) = year_arg
        AND extract(YEAR FROM end_time) = year_arg
    )
    SELECT taxi_users.id AS id,
           taxi_users.full_name AS full_name,
           count(year_rides.id) AS rides_cnt,
           coalesce(sum(distance) / 1000.0, 0)::FLOAT AS total_distance,
```

```
           coalesce(sum(end_time - start_time), '0 minutes'::INTERVAL) AS total_time
    FROM taxi_users
         LEFT OUTER JOIN year_rides ON taxi_users.id = year_rides.user_id
    GROUP BY taxi_users.id, taxi_users.full_name;
END;
$$ LANGUAGE plpgsql;
```

Распределение заказов по времени

Количество заказов разнится в течении дня, знание этого распределения помогут правильно выставлять машину на линию:

```
CREATE OR REPLACE FUNCTION rides_hour_distribution(from_arg TIMESTAMP,
                                                   to_arg TIMESTAMP)
  RETURNS TABLE
          (
            hour INT,
            cnt  BIGINT
          )
  IMMUTABLE
AS
$$
BEGIN
  RETURN QUERY
    SELECT extract(HOUR FROM start_time)::INT as hour,
           count(1) as cnt
    FROM rides
    WHERE start_time BETWEEN from_arg AND to_arg
    GROUP BY extract(HOUR FROM start_time)
    ORDER BY extract(HOUR FROM start_time);
END;
$$ LANGUAGE plpgsql;
```

Валидация пользователей

Запрос, который пригодится при любых запросах от пользователей для проверки их входных данных:

```
CREATE OR REPLACE FUNCTION validate_user(id_arg INT,
                                         pass_arg TEXT)
  RETURNS BOOLEAN
  IMMUTABLE
AS
$$
BEGIN
  RETURN exists(
    SELECT *
    FROM taxi_users
    WHERE id = id_arg
      AND auth_token = crypt(pass_arg, auth_token)
    );
END;
$$ LANGUAGE plpgsql;
```

Завершение поездки

В такси водитель заканчивает поездку, сделаем же удобную функцию которая позволяет это сделать, добавив необходимую информацию в соответствующую таблицу:

```
CREATE OR REPLACE FUNCTION finish_ride(driver_id_arg INT,
                                       price_arg INT,
                                       distance_arg INT,
                                       ride_id_arg INT,
                                       end_time_arg TIMESTAMP,
                                       payment_type_arg payment_type_t)
  RETURNS BOOLEAN
AS
$$
DECLARE
  cur_ride ongoing_rides;
BEGIN
  DELETE
  FROM ongoing_rides
  WHERE driver_id = driver_id_arg
  RETURNING *
```

```
    INTO cur_ride;
  IF cur_ride IS NULL THEN
    RETURN FALSE;
  ELSE
    INSERT INTO rides (id, src_id, dst_id, start_time,
                       end_time, payment_type, price,
                       level, passengers_cnt, driver_id,
                       car_id, user_id, distance)
    VALUES (ride_id_arg, cur_ride.src_id, cur_ride.dst_id, cur_ride.start_time,
            end_time_arg, payment_type_arg, price_arg,
            cur_ride.level, cur_ride.passengers_cnt, driver_id_arg,
            cur_ride.car_id, cur_ride.user_id, distance_arg);
    RETURN TRUE;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

8 Уровни изоляции запросов

Добавление пользователя

В данном запросы мы читаем всего один раз не более чем одну строку, а затем пишем. В таком случае нам достаточно **read commited** уровня изоляции. От фантомных же записей нас сохраняют ограничение $PRIMARY\ KEY$ на $taxi\_users.id$.

Изменение пользовательских адресов

В данных запросах мы так же читаем 1 раз 1 строку, в потом в неё пишем. Фантомные записи на не страшны так как мы не производим никаких вставок, и не ориентируемся на другие строки таблицы. Исходя из этого **read commited** будет достаточно.

Средняя скорость

Это большой но read-only запрос, однако результаты его вряд ли сильно меняются за время его исполнения. К тому же в таблице $taxi_drivers$ вряд ли происходит много изменений, а в таблице $rides$ подавляющие число изменений append-only, что так же исходя из вышесказанного не сильно искажает результат.

Исходя из всего этого можно заключить, что **read uncommited** будет вполне достаточно.

Количество апгрейдов за период

Тяжесть этого запроса зависит от размера периода и проходится он по таблице в преимущественно append-only изменениями. В таком случае главной нашей проблемой могут быть фантомные записи, если конец периода близок к текущему времени. Однако за большой такой период такие записи вряд ли внесут серьёзные изменения, а за маленький запроса отработает достаточно быстро, что так же уменьшит погрешность. От фантомных записей ничего кроме **serializable** не спасает, а тормозит работу мы не хотим. В таком случае можно выбрать самый дешёвый из оставшихся уровней: **read uncommited**.

Распределение способов оплаты

Тяжесть этого запроса зависит от периода, который мы запрашиваем. Сам запрос read-only, и пробегается по таблице, в которой подавляющие число изменений append-only. Однако вряд ли данные за очень короткий период в настоящем будут в любом случае сколько-то репрезентативны Исходя из этого **read uncommited** уровня изоляции ему должно хватить.

Статистика за год

Данные для этого запроса почти не меняются так как собираются за целый год, но в связи с этим объём их велик и не хочется чтобы он блокировал работу сервиса под Новый Год (скорее всего в это время его будут запускать). В таком случае снова **read uncommited**.

Распределение заказов по времени

Логика ограничений этого запроса полностью аналогична запросу распределения способов оплаты, так что **read uncommited**.

Валидация пользователей

Несмотря на то что этот запрос read-only и читает всего одну строку из таблицы, логика его использования подразумевает, чтобы данные были корректны в какой-то момент времени для линеризации транзакций. Исходя из этого выбираем **read commited**.

Завершение поездки

В силу того, что мы читаем из таблицы *ongoing_rides* один раз, фантомные вставки в неё нам не страшны, подобные же вставки в *rides* будет отловлены ограничениями $PRIMARY\ KEY$. В таком случае нам достаточно **read commited**.

9 Пример работы

В данном примере мы начинаем и заканчиваем поездку, а затем смотрим статистику по пользователям в 2020 году(в будущем эти даты стоит поменять).

```
INSERT INTO ongoing_rides (src_id, dst_id, start_time, level,
                           passengers_cnt, user_id, car_id, driver_id)
VALUES (1, 2, '2020-02-01', 'lux', 2, 1, 3, 1);

SELECT finish_ride(1, 700, 1000, 6003,
                   '2020-02-02', 'cash');

SELECT *
FROM user_statistics_for_year(2020);
```