Лабораторная работа № 5

NoSQL

Использование JSON в PostgreSQL

`PostgreSQL` поддерживает типы `JSON` и `JSONB`. Их применение может быть полезным в тех случаях, когда заранее неизвестна структура хранимых данных или необходимо предусмотреть возможность расширения объектов данных дополнительными параметрами.

Отличие `JSONB` от `JSON` заключается в том, что `JSONB` - это двоичная (бинарная) форма представления данных. `JSONB` ускоряет доступ к внутренним записям (ключам и значениям) `JSON`, а также для атрибутов `JSONB` можно создавать индексы `GIN` (обобщённый инвертированный индекс). `JSONB` рекомендуется использовать в тех случаях, когда необходимо осуществлять фильтрацию по данным, которые хранятся `JSON`.

Пример. Создания таблицы `person`, которая содержит атрибут `JSONB`:

```
CREATE TABLE person (
        id SERIAL PRIMARY KEY,
        first_name TEXT NOT NULL,
        last_name TEXT NOT NULL,
        data JSONB DEFAULT 'null'
);
```

Пример. Создание специального индекса `GIN` для атрибута `JSONB`:

```
CREATE INDEX idx_person_data ON person USING GIN(data);
```

Операторы JSON

Приведём краткий список операторов `JSON`. С полным списком операторов и функций можно ознакомится на странице официальной документации

Набор символов =>, используемый при описании операторов является указанием возвращаемого типа.

`{JSON|JSONB} -> {INT|TEXT} => JSON|JSONB` - для массивов (`INT`) возвращает элемент массива по индексу, индексация начинается с 0, отрицательные числа задают позиции с конца массива. Для объектов (`TEXT`) возвращает значение по ключу:

```
SELECT '[{"0": "H"}, {"1": "E"}, {"2": "L"}, {"3": "L"}, {"4": "O"}]'::json ->
-1;
```

```
SELECT '{"key_1": "foo", "key_2": "bar", "key_3": "baz"}'::json -> 'key_2';
```

`{JSON|JSONB} ->> {INT|TEXT} => JSON|JSONB` - аналогичен оператору `->`, возврат происходит в текстовом виде (тип TEXT):

```
SELECT '[{"0": "H"}, {"1": "E"}, {"2": "L"}, {"3": "L"}, {"4": "O"}]'::json ->>
-1;

SELECT '{"key_1": "foo", "key_2": "bar", "key_3": "baz"}'::json ->> 'key_2';
```

`{JSON|JSONB} #> TEXT[] => JSON|JSONB` - возвращает подобъект (внутренняя часть объекта) по заданному пути - массив ключей:

```
SELECT '{"a": {"b": {"c": {"d": "some value"}}}}'::json #> '{a,b,c}';
```

`{JSON|JSONB} #>> TEXT[] => TEXT` - аналогичен оператору `#>`, возврат значения происходит в текстовом виде (тип TEXT):

```
SELECT '{"a": {"b": {"c": {"d": "some value"}}}}'::json #>> '{a,b,c}';
```

`JSONB @> JSONB => BOOL` - проверяет вхождения в объект (слева) подобъекта (справа):

```
SELECT '{"key_1":1, "key_2":2}'::jsonb @> '{"key_2":2}'::jsonb;
```

`JSONB <@ JSONB => BOOL` - проверяет вхождения в объект (справа) подобъекта (слева):

```
SELECT '{"key_1":1, "key_2":2}'::jsonb <@ '{"key_2":2};'::jsonb;
```

`JSONB ? TEXT => BOOL` - проверяет наличие ключа в объекте `JSON`:

```
SELECT '{"key_1":1, "key_2":2}'::jsonb ? 'key_1';
```

`JSONB ?| TEXT[] => BOOL` - проверяет наличие ключей (для объектов) или значений (для массивов) (верхний уровень) из массива справа:

```
SELECT '{"key_1": "v1", "key_2": "v2", "key_3": "v3"}'::jsonb ?| array['key_2',
'key_10'];
```

```
SELECT '["key_1", "key_2", "key_3"]'::jsonb ?| array['key_2', 'key_10'];
```

`JSONB ?& TEXT[] => BOOL` - проверяет наличие ключей (для объектов) или значений (для массивов) из массива слева:

```
SELECT '{"key_1": "v1", "key_2": "v2", "key_3": "v3"}'::jsonb ?& array['key_2',
'key_1'];

SELECT '{"key_1": "v1", "key_2": "v2", "key_3": "v3"}'::jsonb ?& array['key_2',
'key_10'];

SELECT '["key_1", "key_2", "key_3"]'::jsonb ?& array['key_2', 'key_3'];

SELECT '["key_1", "key_2", "key_3"]'::jsonb ?& array['key_2', 'key_10'];
```

`JSONB || JSONB => JSONB` - соединяет значения `JSONB`:

```
SELECT '["a", "b", "c"]'::jsonb || '["x", "y", "z"]'::jsonb;

SELECT '{"key_1": "value_1"}'::jsonb || '{"key_1": "new_value", "key_2":
"value_2"}';
```

`JSONB - {INT|TEXT} => JSONB` - производит удаление из массива по индексу (`INT`), для объектов производит удаление записи по ключу (`TEXT`):

```
SELECT '["a", "b", "c"]'::jsonb - -1;

SELECT '{"key_1": "value_1", "key_2": "value_2"}'::jsonb - 'key_1';
```

`JSONB #- TEXT[] => JSONB` - удаляет элемент или запись по заданному пути:

```
SELECT '["key_1", {"key_2":1}]'::jsonb #- '{1,key_2}';

SELECT '{"key_1": {"key_2": {"key_3": "value"}}}'::jsonb #- '{key_1, key_2,
key_3}';
```

Redis

Redis (remote dictionary server) - NoSQL система управления базами данных, которая размещает базы в оперативной памяти. Базы данных, размещаемые в памяти называют резидентными. Они обладают малым временем доступа к данным и находят широкое применение в информационных системах с высокой нагрузкой.

Типы данных Redis

`Redis` поддерживает следующие типы данных (полный список можно посмотреть на странице официальной документации):

- `string` - строка;
- `list` - список;
- `set` - множества;
- `hash` - хеш-таблица.

Установка и настройка

Доступно несколько вариантов установки `Redis`, один из которых является установка из репозитория:

```
apt install redis-server redis
```

После установки необходимо настроить удаленный доступ к серверу `Redis`.
Для удалённого доступа будут использоваться следующий параметры:

- `client` - имя пользователя
- `1234` - пароль
- .
  Отрыть файл настроек:

```
nano /etc/redis/redis.conf
```

В файле настоек необходимо закомментировать строку:

```
#bind 127.0.0.1 ::1
```

И добавите в файла следующие записи:

```
user client allcommands allkeys on >1234
requirepass 1234
```

Использования Redis

Для `Python` существует одноименный пакет `redis`, который позволяет взаимодействовать с сервером `Redis`:

```
pip install redis
```

Доступ к `Redis`:

```
import redis
rc = redis.StrictRedis(
        host="${HOST}",
        username="client",
        password="1234",
        port=6379,
        charset="utf-8",
        decode_responses=True,
)
```

Где `${HOST}` - ip-адрес сервера.

Пример 1. Работа со строками:

```
# Добавление строки
rc.set("key", "value")

# Доступ к строке по ключу
rc.get("key")

# Добавление подстроки в строку
rc.append("key", " of string")

# Удаление значения по ключу
rc.delete("key")
```

Пример 2. Работа со списками:

```
# Вставить элемент в список слева
rc.lpush("list", 1)

# Вставить элемент в список справа
rc.rpush("list", 2)

# Вставить ещё один элемент в список слева
rc.lpush("list", 0)

# Вставить элемент в список справа
rc.rpush("list", "[]")

# Получить все элементы списка
rc.lrange("list", 0, -1)

# Удалить из списка 1 экземпляр со значением "[]" (удаление по значению)
rc.lrem("list", 1, "[]")

# Снять значение слева
rc.lpop("test")
```

```
# Снять значение справа
rc.rpop("test")
```

Пример 3. Работа с множествами:

```
# Добавление элементов в множество
rc.sadd("set_1", "a")
rc.sadd("set_1", "b")
rc.sadd("set_1", "c")
rc.sadd("set_1", "x")

# Элемент не будет добавлен
rc.sadd("set_1", "a")

# Получение элементов множества
rc.smembers("set_1")

# Удаление значения из множества
rc.srem("set_1", "x")

# Заполнения ещё одного множества
rc.sadd("set_2", "a")
rc.sadd("set_2", "x")
rc.sadd("set_2", "y")

# Операции с множествами
rc.sunion("set_1", "set_2")
rc.sinter("set_1", "set_2")
rc.sdiff("set_1", "set_2")
```

Пример 4. Работа с хеш-таблицами:

```
# Добавление значений в хеш-таблицу
rc.hmset("hash", {"name": "user", "host": "127.0.0.1", "password": "1234"})

# Изменение значения хеш-таблицы по ключу
rc.hset("hash", "name", "user2")

# Получение значения из хеш-таблицы по ключу
rc.hget("hash", "name")

# Удаление значения из хеш-таблицы по ключу
rc.hdel("hash", "password")

# Получение всех значений из хеш-таблицы
rc.hgetall("hash")
```

Пример 5. Сервисные функции:

```
# Получение всех ключей Redis
rc.keys()

# Удаления всех данных, которые хранятся в Redis
re.flushall()
```

Практическая часть

Необходимо разработать комбинированное решение, состоящее из `PostgreSQL` и `Redis`.

База `PostgreSQL` должна включать таблицу, описывающую учётные записи пользователей:

- `email`
- `password`

Предполагается, что с учётной записью пользователя можно связать набор дополнительных данных:

- `first_name` - имя;
- `last_name` - фамилия;
- `phone_number` - телефонный номер;
- `etc`.

Разработать API - набор функции на языке `Python` (клиентское программирование):

1. `signup(email: str, password: str) -> None` - функция должна создавать учётную запись в таблице только в том случае, если учетной записи с заданным `email` не существует. Если запись существует, функция должна поднимать исключение

```
raise Exception("email is already registered")
```

2. `signin(email, password) -> token` - функция должна осуществлять процедуру аутентификации - проверять совпадение `password` для заданного `email`. Если пароль совпадает, то функция создает запись в `Redis`, в записи необходимо учитывать:

- `email` - адрес электронной почты;
- `time` - метка времени (`timestamp`), обозначающая время входа;
- `token` - случайно набор символов, можно воспользоваться `uuid` или воспользоваться функцией:

```
import random, string

def create_token(length=16):
        alphabet = string.printable
        token = "".join([*map(lambda _: random.choice(alphabet),
range(length))])
        return token
```

3. `verify_token(token) -> bool` - функция возвращать `True` в том случае, если в `Redis` существует запись с данным значением токена, а иначе - `False`.
4. `signout(token)` - функция, которая удаляет запись из `Redis`. Если токен не найден, функция должна поднимать исключение:

```
raise Exception("invalid token")
```

5. `add_account_data(token, **kwargs) -> None` - функция добавляет данные учётной записи (ключ-значения). Если токен не найден, функция должна поднять исключение:

```
raise Exception("invalid token")
```

6. `get_account_data(toke) -> str` - функция должна возвращать данные учётной записи в формате `JSON` (всё кроме пароля). Если токен не найден, функция должна поднять исключение:

```
raise Exception("invalid token")
```

Отчет должен содержать:

1. Файл `SQL`.
2. Файл реализации `Python`.
3. Файл тестов реализованных функций `Python`.