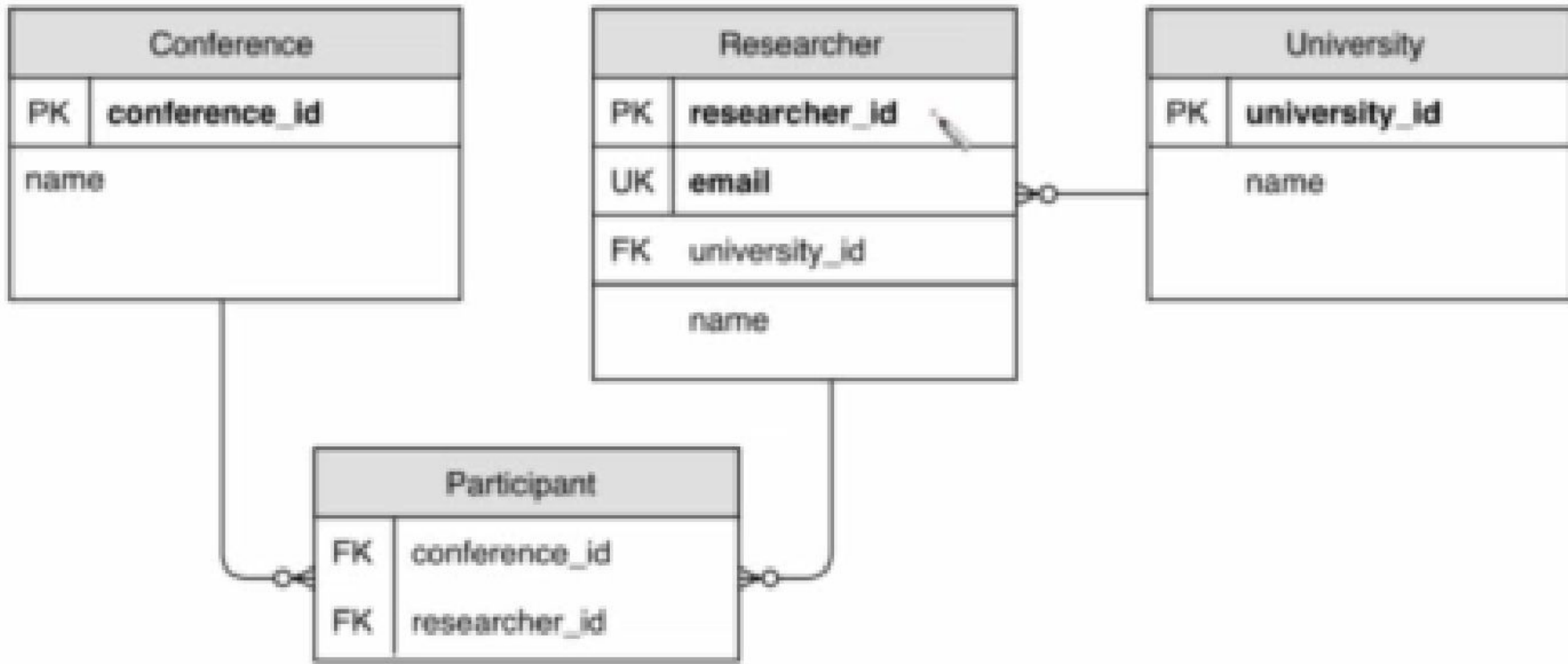


СУБД и приложение



Для каждой пары

(конференция, университет)

найти суммарное количество исследователей из данного университета, участвовавших в данной конференции.

- ▶ 200 записей в University
- ▶ 1000 записей в Conference
- ▶ 20000 записей в Researcher
- ▶ 100000 записей в Participant

```
# db.execute выполняет запрос и возвращает результат  
# в виде массива словарей  
conferences = db.execute('SELECT * FROM Conference')  
for conf in conferences:  
    selectParts = '''  
        SELECT * FROM Participant  
        WHERE conference_id = ''' + conf['conference_id']  
    for p in db.execute(selectParts):  
        researcher = db.execute('''  
            SELECT * FROM Researcher  
            WHERE researcher_id=''' + p['researcher_id'])  
        uni_id = researcher['university_id']  
        uni = db.execute('''  
            SELECT * FROM University  
            WHERE university_id=''' + uni_id)  
# Инкрементируем счетчик  
        inc(conf, uni)
```

- Сколько же времени будет выполняться продемонстрированный скрипт?

1 секунду

5 секунд

20 секунд

40 секунд

1 минуту

5 минут

Жизнь простого запроса

Приложение посылает текст серверу БД

- ▶ Сервер делает синтаксический разбор запроса

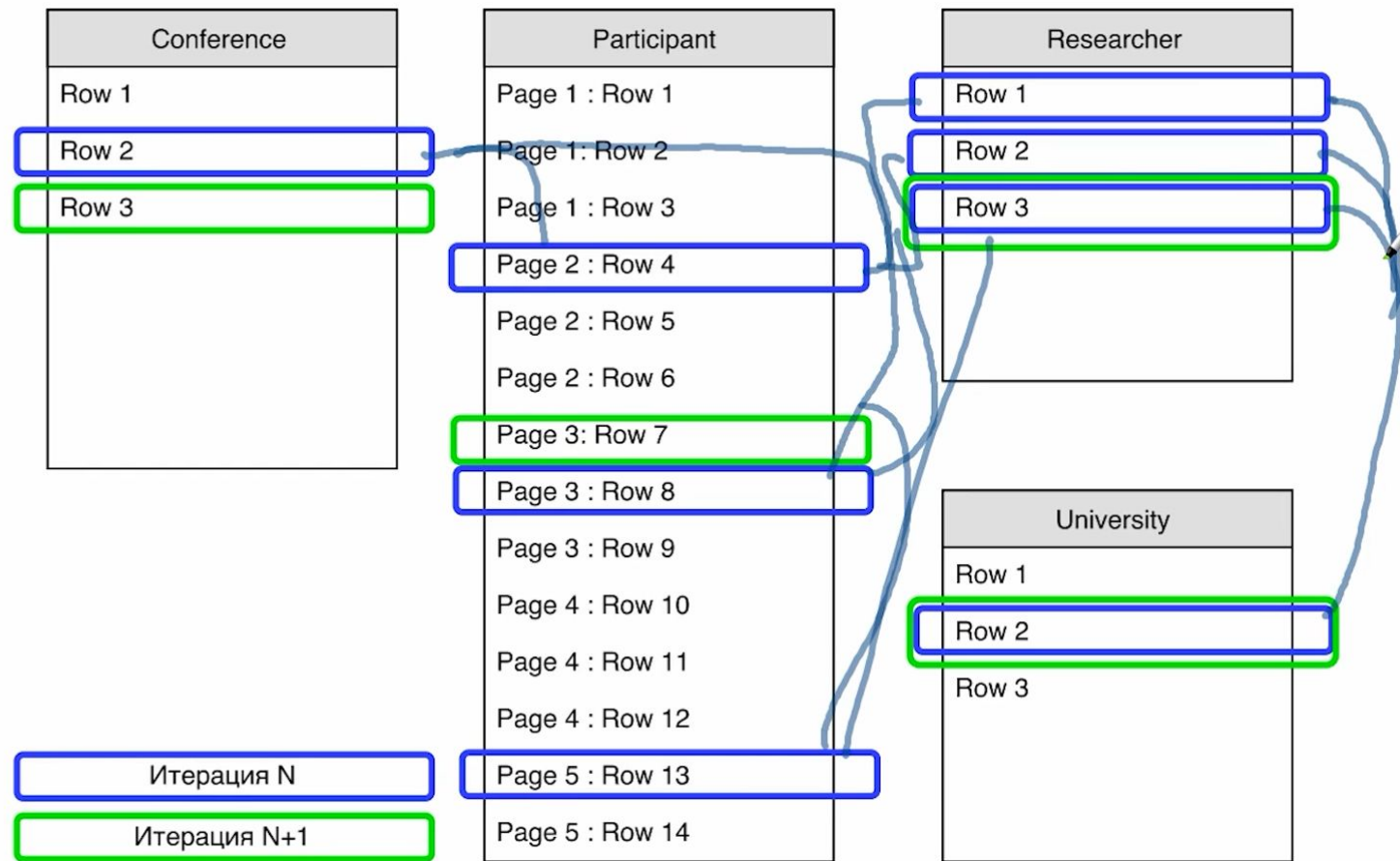
- ▶ Проверяет использованные имена

- ▶ Составляет план выполнения запроса

- ▶ Выполняет запрос

- ▶ Формирует результат и отсылает приложению

- ▶ В таблице Participant 2-3 мегабайта и примерно 200-300 страниц
- ▶ 1000 просмотров выполнялись бы за 1000 секунд
- ▶ Если бы страницы читались только с диска то один просмотр занял бы 1-2 секунды
- ▶ В самом хорошем случае за 10 секунд



Сколько строк получится в результате выполнения этого запроса, если в таблице Foo 100 строк, в таблице Bar 50 строк и в таблице Baz 30 строк?

```
-- Таблицы выглядят так:  
-- Baz(id INT PRIMARY KEY, <что-то еще>)  
-- Bar(id INT PRIMARY KEY, baz_id INT NOT NULL REFERENCES Baz, <что-то еще>)  
-- Foo(id INT PRIMARY KEY, bar_id INT NOT NULL REFERENCES Bar, <что-то еще>)  
SELECT *  
FROM Foo  
JOIN Bar ON Foo.bar_id = Bar.id  
JOIN Baz ON Bar.baz_id = Baz.id
```

30

50

100

5000

1500

3000

150000

180

Выполнение последовательности соединений:

1. Соединение Conference JOIN Participant

Conference		Participant	
id	name	conference_id	researcher_id
1	Conf1	2	100
2	Conf2	2	101
		1	98

Conference JOIN Participant

Conference.id	name	Participant.conference_id	researcher_id
1	Conf1	1	98
2	Conf2	2	100
2	Conf2	2	101

2. Соединение (Conference JOIN Participant) JOIN Researcher

Conference JOIN Participant

Conference.id	name	Participant.conference_id	researcher_id
1	Conf1	1	98
2	Conf2	2	100
2	Conf2	2	101

```
Researcher
=====
id  name      university_id
-----
98  Jennifer  239
101 Donald    566
100 Jeffrey   239
```

Conference C JOIN Participant P JOIN Researcher R

C.id	name	P.conference_id	P.researcher_id	R.id	R.name	R.university_id
1	Conf1	1	98	98	Jennifer	239
2	Conf2	2	100	100	Jeffrey	239
2	Conf2	2	101	101	Donald	566

3. Соединение ((Conference JOIN Participant) JOIN Researcher) JOIN University

Conference C JOIN Participant P JOIN Researcher R

C.id	name	P.conference_id	P.researcher_id	R.id	R.name	R.university_id
1	Conf1	1	98	98	Jennifer	239
2	Conf2	2	100	100	Jeffrey	239
2	Conf2	2	101	101	Donald	566

University

```
=====
university_id  name
-----
239            Stanford
566            ETH
```

```
-- в таблице ниже пропущены повторяющиеся столбцы researcher_id, university_id
--
```

Conference C JOIN Participant P JOIN Researcher R JOIN University U

C.id	name	conference_id	R.id	R.name	university_id	U.name
1	Conf1	1	98	Jennifer	239	Stanford
2	Conf2	2	100	Jeffrey	239	Stanford
2	Conf2	2	101	Donald	566	ETH

EXPLAIN ANALYZE

```
SELECT Conference.name, University.name
FROM Conference JOIN Participant ON
    (Conference.conference_id = Participant.conference_id)
JOIN Researcher ON
    (Participant.researcher_id = Researcher.researcher_id)
JOIN University ON
    (Researcher.university_id = University.university_id)
```

Hash Join

Hash Cond: (p.conference_id = c.conference_id)

→ Hash Join

Hash Cond: (p.researcher_id = r.researcher_id)

→ Seq Scan on Participant p

→ Hash

→ Hash Join

Hash Cond: (

r.university_id = **u.university_id**

)

→ Seq Scan on Researcher r

→ **Hash**

→ **Seq Scan on University u**

→ Hash

→ Seq Scan on Conference c

Planning time: 1.033 ms

Execution time: 93.119 ms

Hash Join

Выполняет соединение $R \bowtie S$ за $B(R + S)$ операций чтения диска, где $B(R + S)$ – суммарное количество дисковых страниц в обеих таблицах, при условии что одна из таблиц помещается в памяти

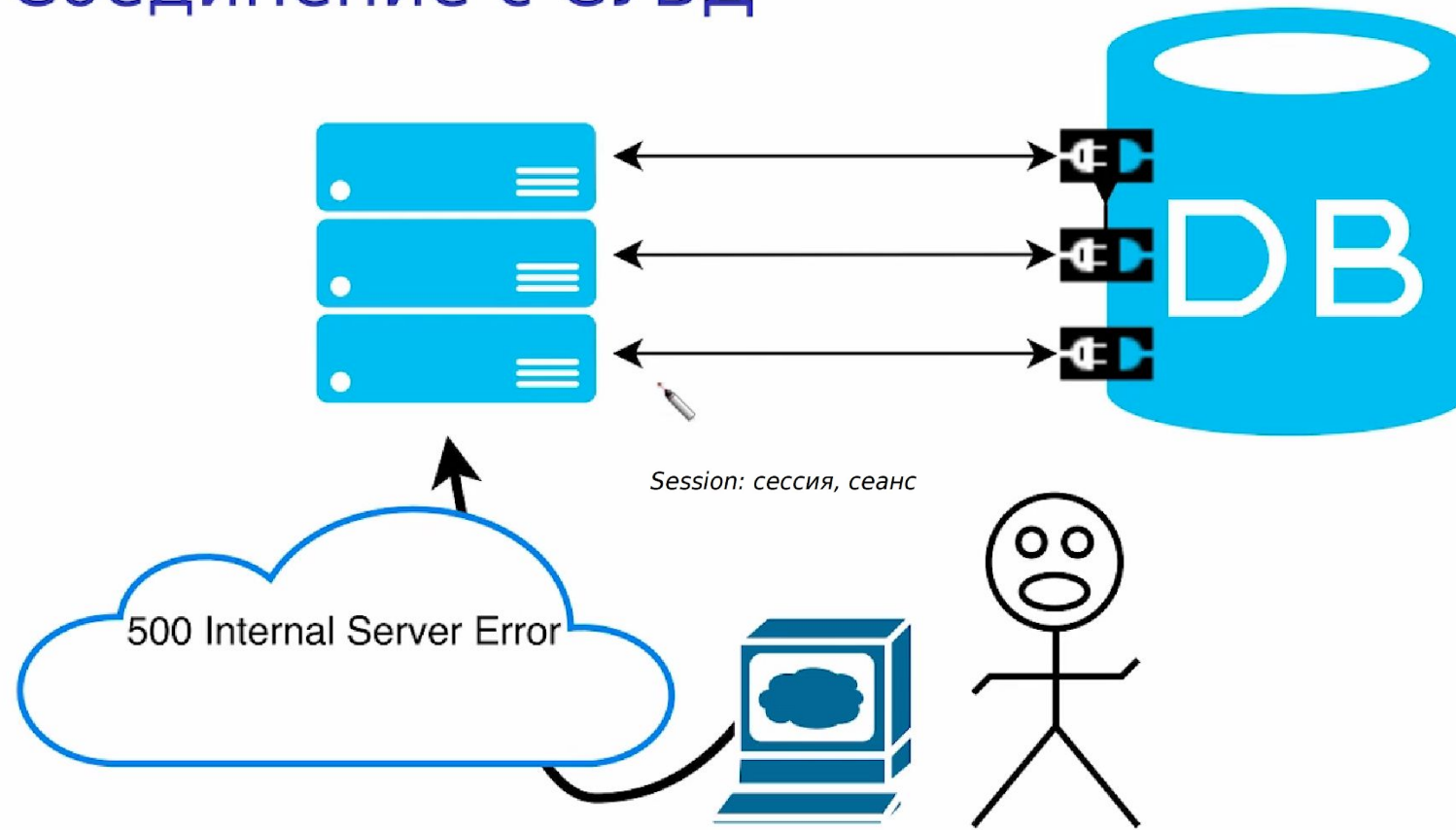
Псевдокод алгоритма HashJoin

```
# Выполняет соединение r и s по условию r.r_join_attr = s.s_join_attr
HashJoin(Relation r, Relation s, String r_join_attr, String s_join_attr):
  # проходим по всей таблице r и
  # строим словарь с дубликатами, где ключом является значение атрибута соединения из r
  # а в списке значений содержатся строки r
  hash_multimap = new HashMultimap()
  for r_page in r.pages():
    for r_row in r_page.rows():
      hash_multimap[r_row.value(r_join_attr)].add(r_row)

  # проходим по всей таблице s и для каждой строки проверяем наличие в построенном словаре
  # ключа равного значению атрибута соединения из s
  for s_page in s.pages():
    for s_row in s_page.rows():
      if s_row.value(s_join_attr) in hash_multimap:
        for r_row in hash_multimap.values(s_row.value(s_join_attr)):
          # Если таковой ключ найден то в результат отправляются все пары текущей строки из s
          # и списка значений из r
          emit(r_row, s_row)
```



Соединение с СУБД



Connection: соединение, подключение, коннекция

Бывает подключение == сессия, бывает много сессий на одно подключение

Как устроены сессии ?

- Сессии - это механизм, созданный для временного хранения и передачи информации между скриптами в пределах одного сайта.
- Сессии предусматривают возможность создания собственных способов обработки информации, поэтому, в принципе, можно использовать сессии и при работе с несколькими сайтами или даже с несколькими серверами.

Что произойдет с этим скриптом, если запустить его на PostgreSQL с настройками по умолчанию?

```
import psycopg2 as pg_driver

# Инкапсулирует обращения к БД
class Researcher:
    def __init__(self, id, university_id):
        self.db = pg_driver.connect(user="postgres", host="localhost")
        self.id = id
        self.university_id = university_id

    def getUniversity(self):
        cur = self.db.cursor()
        cur.execute('''
SELECT name FROM University WHERE university_id={0}
'''.format(self.university_id))
        return cur.fetchone()[0]

researchers = []
### Шаг 1: получаем из базы всех исследователей
cur = pg_driver.connect(user="db", password="q", host="localhost", dbname="db").cursor()
cur.execute("SELECT researcher_id, university_id FROM Researcher")

### Шаг 2: создаем объекты доступа к данным
for r in cur.fetchall():
    researchers.append(Researcher(r[0], r[1]))

### Шаг 3: печатаем университет, в котором работает каждый исследователь
for r in researchers:
    print r.getUniversity()
```

В PostgreSQL существуют все необходимые объекты (пользователь, БД, таблицы), параметры соединения правильные, в таблицах есть данные и они согласованы. Всё хорошо, в общем.

Скрипт выполнится без ошибок в любом случае

Если в таблице Researcher ≥ 100 записей то скрипт аварийно завершится, ничего не напечатает

Если в таблице Researcher ≥ 100 записей то скрипт напечатает сведения о первых 99, после чего закончит свою работу, возможно, аварийно

Скрипт выполнится без ошибок, если в таблице Researcher меньше 100 записей

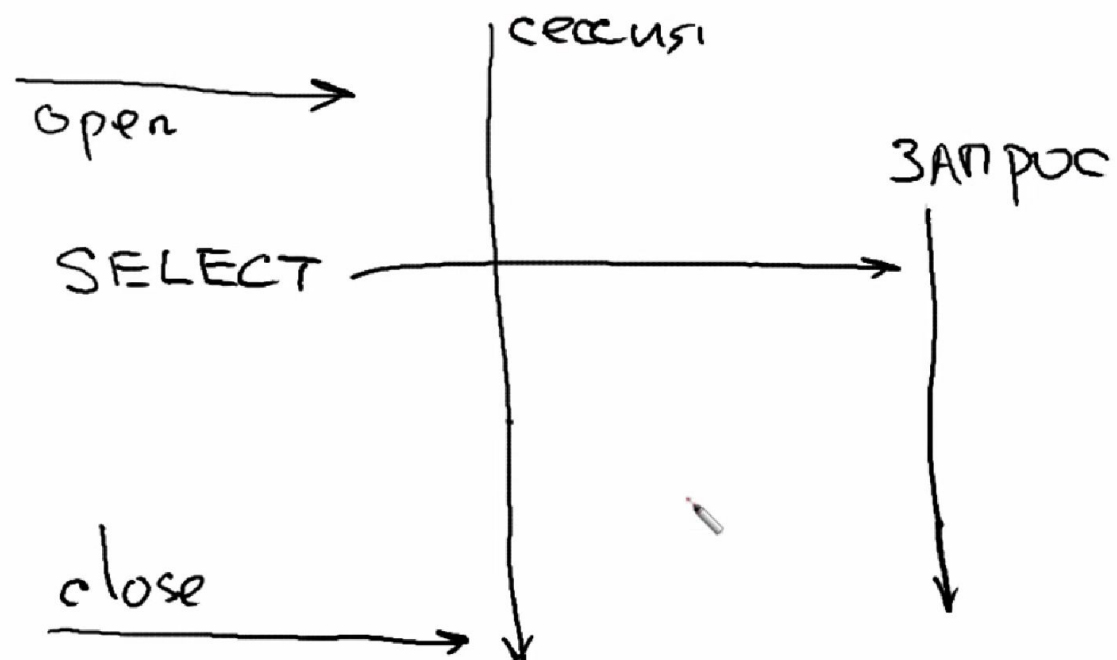
```
# vim m3_connection_demo1.py
# python m3_connection_demo1.py
Traceback (most recent call last):
  File "m3_connection_demo1.py", line 26, in <module>
    researchers.append(Researcher(r[0], r[1]))
  File "m3_connection_demo1.py", line 8, in __init__
    self.db = pg_driver.connect(user="db", password="q", host="localhost", dbname="db")
  File "/usr/lib/python2.7/dist-packages/psycopg2/__init__.py", line 179, in connect
    connection_factory=connection_factory, async=async)
psycopg2.OperationalError: FATAL: sorry, too many clients already

db=# SELECT COUNT(*) FROM Researcher;
 count
-----
      20
(1 row)

db=# ^D\q
# python m3_connection_demo1.py
Uni35
Uni111
Uni121
Uni83
Uni78
Uni96
Uni138
Uni11
Uni50
Uni4
Uni90
Uni73
Uni136
Uni52
Uni101
Uni68
Uni105
Uni87
Uni152
Uni190
# █
```

Сессии: как быть?

- ▶ Закрывать сессии



Закрывать сессии

Python `try-finally, with`

Java `try-finally, try with resources`

C++ `деструкторы, умные указатели`

```
# time python m3_connection_demo2.py
```

```
Uni60
```

```
Uni80
```

```
Uni18
```

```
real    0m1.260s
```

```
user    0m0.080s
```

```
sys    0m0.032s
```

```
Uni156
```

```
Uni60
```

```
Uni80
```

```
Uni18
```

```
real    0m0.081s
```

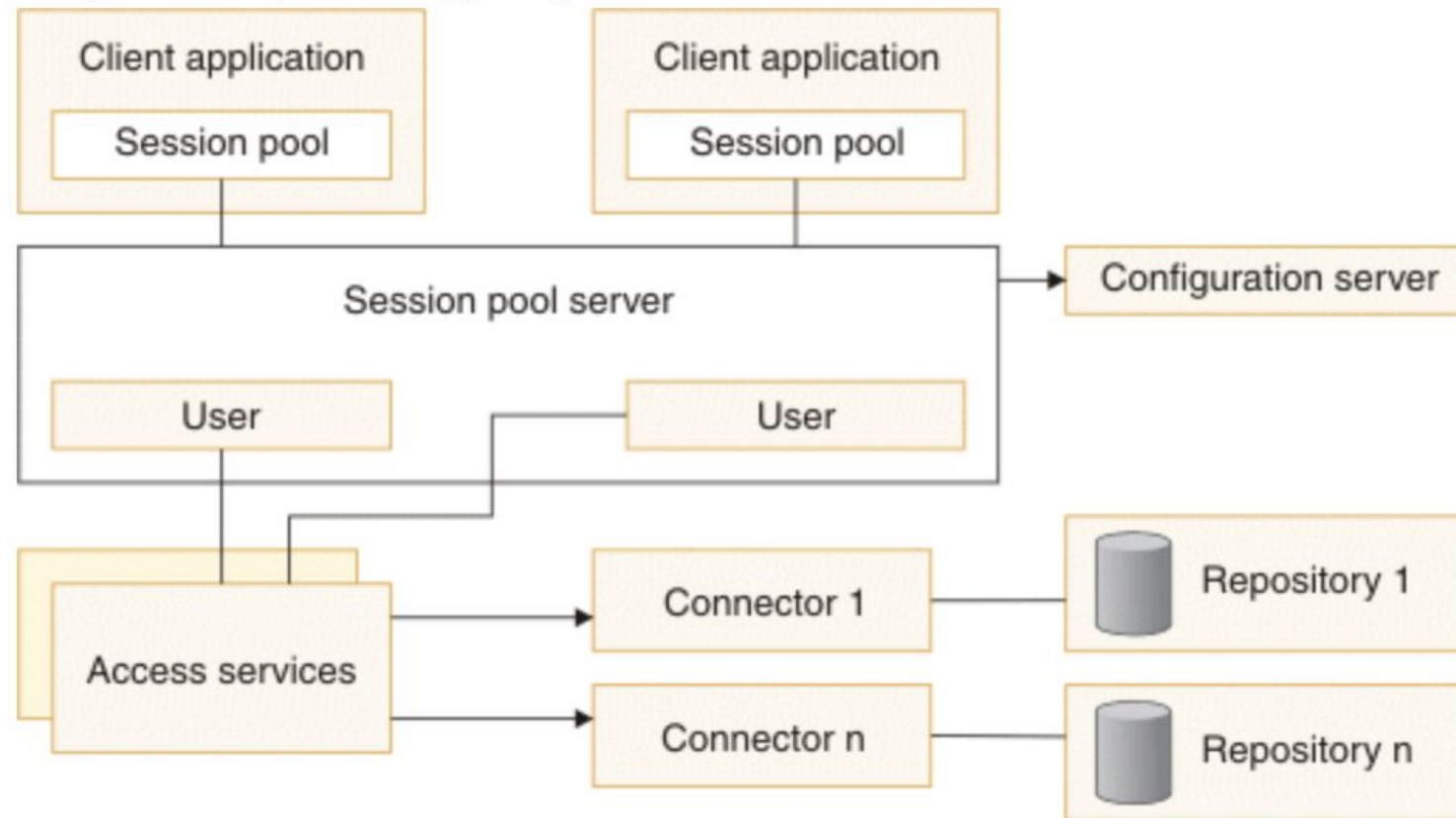
```
user    0m0.024s
```

```
sys    0m0.008s
```


- Пулы сеансов позволяют приложениям повторно использовать подключения к репозиторию внутри приложения или между различными приложениями. Пулы сеансов также можно использовать для ограничения количества подключений к репозиторию, выполняемых приложением во время пиковой активности. Пулы сеансов также сводят к минимуму количество входов и выходов из репозитория, необходимых для выполнения действий с репозиторием, и предотвращают накладные расходы, вызванные подключениями для входа в репозиторий.

Пул сеансов — это механизм распределенного объединения, что означает, что пул сеансов может отслеживать ресурсы для нескольких приложений, работающих на любом количестве серверов. На рисунке 1 показана одна из возможных конфигураций.

Рисунок 1. Архитектура пула сеансов



```

import psycopg2 as pg_driver
import psycopg2.pool as pool

pg_pool = pool.SimpleConnectionPool(1, 2, user="db", password="q", host="localhost", dbname="db")
# Инкапсулирует обращения к БД
class Researcher:
    def __init__(self, id, university_id):
        self.id = id
        self.university_id = university_id

    def getUniversity(self):
        db = pg_pool.getconn()
        try:
            cur = db.cursor()
            cur.execute('''
                SELECT name FROM University WHERE university_id={0}
            '''.format(self.university_id))
            return cur.fetchone()[0]
        finally:
            pg_pool.putconn(db)

researchers = []
### Шаг 1: получаем из базы всех исследователей
cur = pg_driver.connect(user="db", password="q", host="localhost", dbname="db").cursor()
cur.execute("SELECT researcher_id, university_id FROM Researcher")

### Шаг 2: создаем объекты доступа к данным
for r in cur.fetchall():
    researchers.append(Researcher(r[0], r[1]))

### Шаг 3: печатаем университет, в котором работает каждый исследователь
for r in researchers:
    print r.getUniversity()

```

Uni25
Uni63
Uni157

real 0m0.134s
user 0m0.032s
sys 0m0.016s

"m3_connection_demo4.py" 35L, 1294C

Не перекладывай работу СУБД на приложение

Закрывай сессии

Пользуйся пулом сессий

Параметризуй контекстом объекты доступа к данным

- Представьте себе, что вы программист, занимающийся производительностью вашего приложения. Вы увидели приведённый ниже код на языке Python и поняли, что он занимает слишком много времени. Попробуйте переписать его одним запросом.
- Решением является текст SQL запроса. Обратите внимание на очередность столбцов в ответе. Прочими различиями в формате текстового вывода питона и SQL пренебрегите.

- Схема БД

- CREATE TABLE Spacecraft(
 - id SERIAL PRIMARY KEY,
 - name TEXT UNIQUE,
 - service_life INT DEFAULT 1000,
 - birth_year INT CHECK(birth_year > 0));
- CREATE TABLE Planet(
 - id SERIAL PRIMARY KEY,
 - name TEXT UNIQUE,
 - distance NUMERIC(5,2)););

```
CREATE TABLE Commander(  
  id SERIAL PRIMARY KEY,  
  name TEXT  
);
```

```
CREATE TABLE Flight(  
  id INT PRIMARY KEY,  
  spacecraft_id INT REFERENCES Spacecraft,  
  planet_id INT REFERENCES Planet,  
  commander_id INT REFERENCES Commander,  
  start_date DATE,  
  UNIQUE(spacecraft_id, start_date),  
  UNIQUE(commander_id, start_date)
```

Код на Python

```
# encoding: utf-8
import psycopg2 as pg_driver

# Postgres, запущенный в докере, не требует пароля, если клиент находится на localhost
db = pg_driver.connect(user="postgres", host="localhost")
cur = db.cursor()

def fun1(commander_id):
    result = []
    cur.execute("SELECT commander_id, spacecraft_id, start_date FROM Flight")
    for flight in cur.fetchall():
        if flight[0] != commander_id:
            continue
        cur.execute("SELECT id, name FROM Spacecraft")
        for spacecraft in cur.fetchall():
            if spacecraft[0] == flight[1]:
                result.append((spacecraft, flight)) # добавляем в результат пару
    return result
```

```
def fun2(commander_id):
    result = []
    for spacecraft, flight in fun1(commander_id):
        cur.execute("SELECT id, name FROM Commander")
        for cmndr in cur.fetchall():
            if cmndr[0] == flight[0]:
                result.append((cmndr[1], spacecraft[1], flight[2])) # добавляем в результат тройку
    return result

for row in fun2(6):
    print row
```