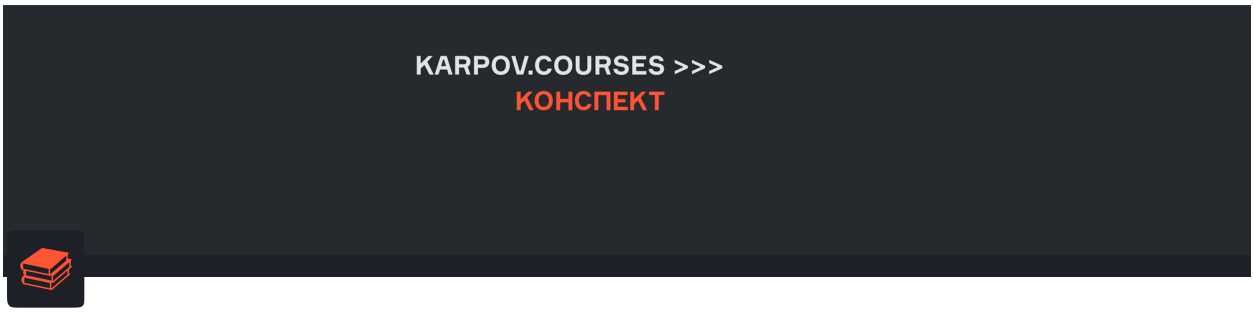


> Конспект > Урок 6 > Базы данных в Python: основы

> Оглавление

> Оглавление
> Система управления базами данных (СУБД)
> PostgreSQL
> Диаграмма отношений и транзакции
> SQL– Structured Query Language
> Основные операторы: SELECT+FROM, WHERE
> GROUP BY и ORDER BY
Агрегатные функции SQL
Сортировка в SQL
Структура sql-запроса
> Объединение таблиц: JOIN
> Виды JOIN
RIGHT JOIN
INNER JOIN
FULL OUTER JOIN
> SQL в Python
> SQL в pandas
Сохранение новой таблицы

> Система управления базами данных (СУБД)

База данных (БД) – это организованное хранение информационных ресурсов в виде интегрированной совокупности файлов, обеспечивающее удобное взаимодействие между ними и быстрый доступ к данным.

Система управления базами данных или СУБД (англ. Database Management System, сокр. DBMS) — совокупность языковых и программных средств, предназначенная для создания, ведения и совместного использования баз данных многими пользователями.

СУБД — это не один инструмент, а целый комплекс из программ, который решает несколько задач:

1. **Хранение данных.** СУБД решает, в каком формате будут храниться данные на диске и как их читать. Также СУБД может «размазывать» свои данные на несколько машин, чтобы увеличить суммарную емкость.
2. **Выгрузка данных.** Чаще всего используется единый язык для доступа к данным (он, кстати, называется SQL). Язык этот работает как конструктор, из которого вы можете собрать запрос к данным любой сложности.
3. **Обеспечение контроля доступа.** СУБД часто имеет поддержку пользователей (например, «дядя Петя» с правами «только чтение» и «админ Степан» с полным доступом на запись).

4. **Восстановление данных после сбоев,** как делать резервные копии и как следить за изменениями.

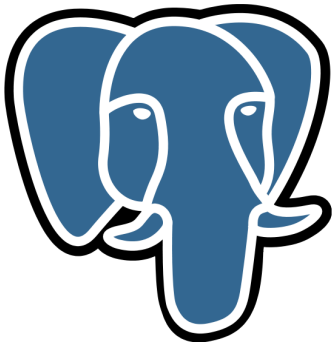
СУБД могут быть реляционными и NoSQL (not only SQL).

> PostgreSQL

PostgreSQL — это популярная свободная объектно-реляционная (от англ. relation — отношение) система управления базами данных, основанная на языке SQL.

Фундаментальная характеристика объектно-реляционной базы данных — это поддержка пользовательских объектов и их поведения, включая типы данных, функции, операции, домены и индексы.

Подробнее о PostgreSQL и ее преимуществах



PostgreSQL Exercises - примеры задач из лекции а также много других упражнений.

В лекции используется интегрированная среда разработки (IDE) **DataGrid**

Чтобы работать с SQL мы будем использовтаь систему redash — open source инструмент, представляющий собой SQL-консоль, которую можно бесплатно развернуть у себя на сервере и подключить в качестве датасорса множество баз данных (включая Clickhouse) или другой источник по API, например, Google Sheets.

> Диаграмма отношений и транзакции

В общем виде любая СУБД состоит из Баз данных, а те, в свою очередь, из схем (schema) — связанных по определенным ключам таблиц.

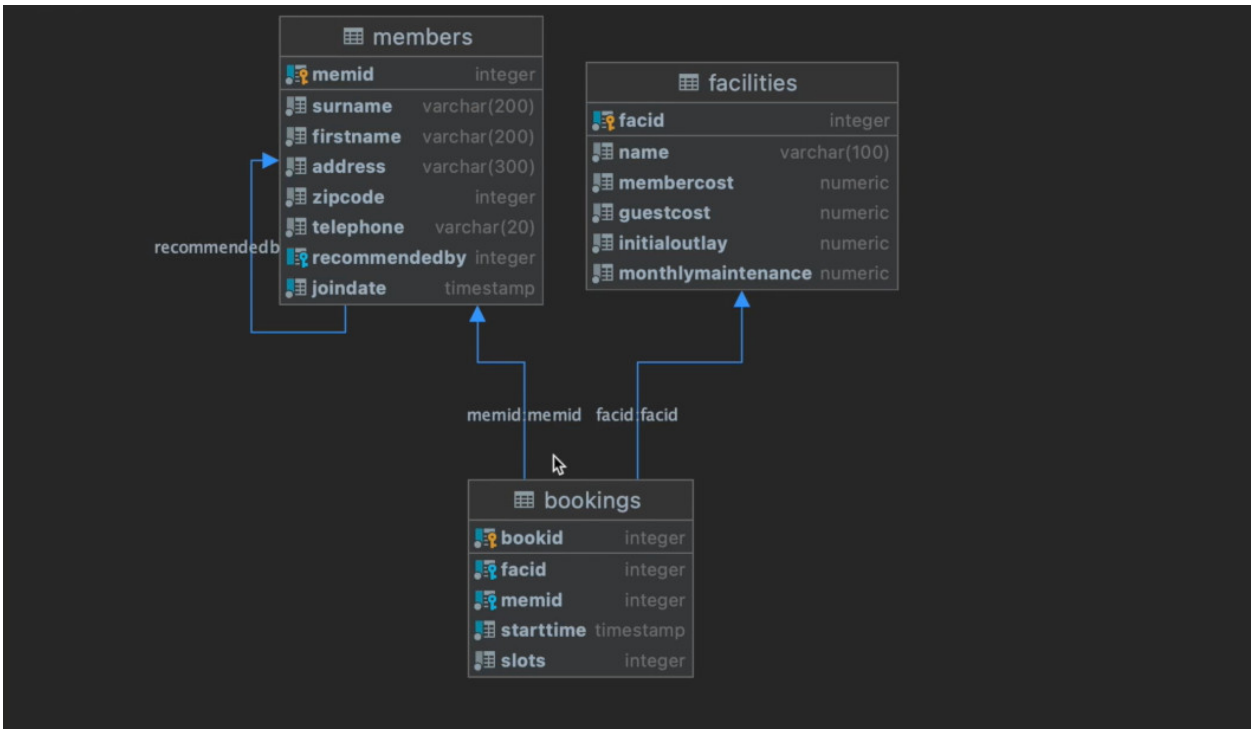
Не запутаться во множестве связей между таблицами помогает визуализация их в виде специальной диаграммы отношений или Entity Relationship Diagram (ERD) — диаграммы связи сущностей.

Сущность (entity) — это любой объект в базе данных, который может быть идентифицирован неким способом, отличающим его от других объектов. В нашем примере «сущность» — это пользователь и площадка.

Атрибут — свойство, которое описывает некоторую характеристику объекта.

Пример: рассмотрим множество пользователей системы бронирований. Каждого из них можно описать с помощью характеристик фамилия, имя, адрес и тд. Поэтому сущность member имеет атрибуты surname, firstname, address и тд.

Таким образом, сущность фактически представляет из себя множество атрибутов, которые описывают свойства всех членов данного набора сущностей.



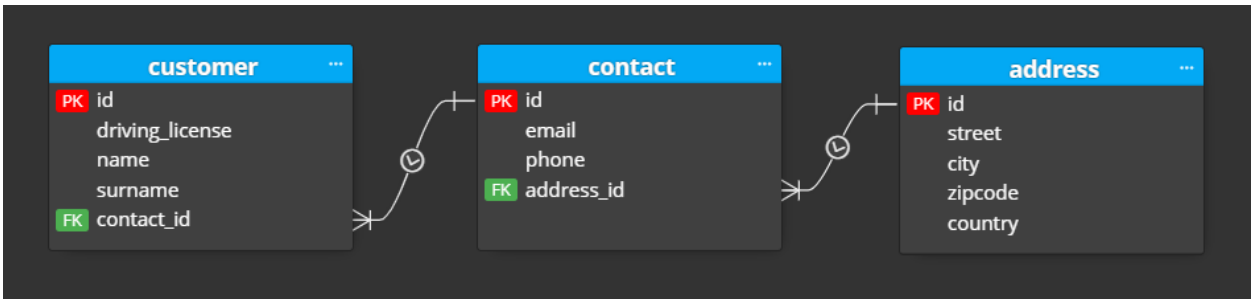
Для того, чтобы вызвать такую диаграмму, нужно перейти в среду с нашей таблицей / Diagrams / Show Visualisation

Таблицы связаны между собой ключами, благодаря которым можно перемещаться между ними, подтягивая нужные данные.

- **Первичный ключ** (PRIMARY KEY)– идентифицирует каждую запись в таблице. Как правило, это уникальный идентификатор записи. Первичный ключ столбец не может иметь значения NULL. Таблица может иметь только один первичный ключ, который может состоять из одного или нескольких полей. Когда несколько полей используются в качестве первичного ключа, их называют

  составным ключом.

- **Внешний ключ** (FOREIN KEY)– устанавливает родительско-дочерние отношения в таблице. Обычно таблица с внешним ключом – дочерняя, а таблица без внешнего – родительская.



Наследование ключей: внешний ключ (FK) для родительской таблицы является первичным (PK) для дочерней. Таблица customer - родительская для contact, а contact для address.

Транзакции – операции обработки данных, которые переводят базу из одного состояния в другое.

Транзакции подчиняются аббревиатуре ACID – атомарность, согласованность, изолированность и стойкость.

- Атомарность – гарантирует, что никакая транзакция не будет зафиксирована в системе частично
- Согласованность – каждая успешная транзакция по определению фиксирует только допустимые результаты.

- Изолированность – при параллельном выполнении транзакции никак не влияют друг на друга.
- Стойкость – если транзакция выполнена, то результат будет точно сохранен вне зависимости от проблем с оборудованием.

Статья об ACID простым языком.

> SQL– Structured Query Language

SQL (Structured Query Language — Структурированный Язык Запросов) — это декларативный язык программирования, который используется в качестве эффективного способа управления данными — их извлечения, поиска, модификации и удаления из баз данных.

Главное преимущество использования SQL - высокая скорость обработки больших данных. Когда у нас действительно большие данные, то лучше их сначала подготовить их через SQL и потом только выгружать для обработки в python/pandas.

Для выполнения различных действий внутри СУБД используются операторы SQL.

Оператор – это зарезервированное слово, или символ, который используется в SQL выражениях.

SQL содержит 4 основные группы операторов:

- **Операторы** описания данных (DDL - Data Definition Language): CREATE, DROP, ALTER и др.
- **Операторы** манипуляции данными (DML - Data Manipulation Language): SELECT, INSERT, DELETE, UPDATE и др.
- **Операторы** контроля баз данных (DCL – Data Control Language): GRANT / REVOKE, LOCK / UNLOCK , SET LOCK MODE).
- **Операторы** для управления транзакциями (TCL – Transaction Control Language): BEGIN TRANSACTION, COMMIT TRANSACTION и др.

Интерактивные тренажеры по SQL тут и тут.

Ссылка на справочник по SQL.

> Основные операторы: SELECT+FROM, WHERE

Любой запрос в SQL начинается с команды `SELECT.`

`SELECT` — предназначен для извлечения строк данных из одной или нескольких таблиц, используется совместно с оператором `FROM`:

```
SELECT *
FROM members;
```

Символ * (звездочка) в `SELECT *` означает "колонка с любым именем, используется когда мы хотим получить всё содержимое из таблицы.

`WHERE` — необязательный элемент запроса, который используется, когда нужно отфильтровать данные по нужному условию:

```
SELECT *FROM members
WHERE surname = 'Smith' OR firstname = 'David';
```

Как и в Python, в SQL поддерживаются логические операторы: `AND` ("и", логическое умножение), `OR` ("или", логическая сумма), `NOT` ("не", отрицание).

Можно осуществлять численные преобразования с помощью функций `SUM` (сумма), `SQRT` (извлечение корня), `ROUND` (округление), `ABS` (модуль числа) и других

Также можно делать проверки на принадлежность заданному множеству значений с помощью оператора `IN` или `NOT IN` (исключение)

```
SELECT "name" FROM facilities
WHERE membercost IN (0, 5, 10)
LIMIT 10;          -- через LIMIT можно задать количество строк для вывода
```

Поскольку язык SQL предназначен прежде всего для работы с очень большими объемами данных (тысячи и более строк), то не всегда нужно выводить таблицу целиком. Чтобы ограничить число записей для отображения, используется оператор `LIMIT` и число строк, которые хотим вывести.

> GROUP BY и ORDER BY

`GROUP BY` — позволяет группировать результаты при выборке из базы данных (аналог агрегации в python):

```
SELECT surname, COUNT(surname)
FROM members
WHERE recommendedby IS NOT NULL  -- проверка на прочерк, как в Python
GROUP BY surname;
```

Ещё одной важной особенностью языка SQL является последовательность выполнения запросов. Так, оператор `GROUP BY` может быть использован только после `WHERE`, а не наоборот. Если нам нужно применить фильтр по уже сгруппированным данным, применяется оператор `HAVING.`

Например, если нам нужно отфильтровать фамилии, которые встречаются более одного раза, то сначала необходимо сгруппировать данные по фамилии, а после применить фильтр:

```
SELECT surname, COUNT(surname)
FROM members
WHERE recommended by IS NOT NULL
GROUP BY surname
HAVING COUNT(surname) > 1; -- вот наша фильтрация на агрегат
```

Получим статистику по фамилиям среди приглашенных членов (recommended by `IS NOT NULL`).

Агрегатные функции SQL

Также как и в python, при формировании групп нужно указать, какое значение мы хотим получить:

SQL поддерживает следующие агрегатные функции:

`MIN` / `MAX` - узнать минимальное/максимальное значение по группе

`SUM` - сумировать все значения в группе

`AVG` - найти среднее значение по группе

`COUNT` - посчитать количество элементов группе

Важно отметить, что в отличии от python, в SQL агрегатные функции указываются **перед** группировкой, сразу после `SELECT.`

Агрегатные функции могут использоваться и без `GROUP BY`, но тогда запрос `SELECT` **не должен содержать простых столбцов** (кроме как столбцов, служащих аргументами агрегатной функции).

Результатом вычислений любой агрегатной функции является константное значение, отображаемое в отдельном столбце результата.

Аргументу агрегатной функции может предшествовать одно из двух возможных **ключевых слов**:

`ALL` - вычисления выполняются над всеми значениями столбца (значение по умолчанию, указывать не обязательно).

`DISTINCT` - отбирает только уникальные значения столбца.

Например,

```
SELECT surname,
       COUNT(DISTINCT surname)        --подсчитает число уникальных фамилий в столбце
```

```
FROM members;
```

Оператор `AS` используется для переименования результирующих столбцов при выборке элементов, например:

```
SELECT COUNT('surname') AS "total_members" --выведет количество участников в колонку с названием total_members
FROM members;
```

Сортировка в SQL

`ORDER BY` — позволяет сортировать значения при выводе таблицы. По умолчанию сортировка происходит по возрастанию (`ASC`). Для того, чтобы упорядочить значения от большего к меньшему, нужно добавить `DESC` (англ. descending — убывающий):

```
SELECT *
FROM bookings
ORDER BY starttime DESC;
```

Структура sql-запроса

Резюмируя вышесказанное, рассмотрим общую структуру запроса:

```
SELECT   --столбцы или * для выбора всех столбцов; обязательный запрос
FROM     --таблица; обязательный запрос
WHERE    -- условие/фильтрация; необязательный запрос
GROUP BY --столбец, по которому хотим сгруппировать данные; необязательный запрос
HAVING   --условие/фильтрация на уровне сгруппированных данных; необязательно
ORDER BY --столбец, по которому хотим отсортировать вывод; необязательно
```

> Объединение таблиц: JOIN

`JOIN` – одна из наиболее часто используемых команд в SQL-синтаксисе, существующий только в реляционных базах данных. Предназначена для соединения двух или более таблиц из базы данных по ключу, который присутствует в обеих таблицах. Перед ключом ставится оператор `ON`.

В качестве примера рассмотрим таблицу bookings, где лежат id площадок и время бронирования. Чтобы вывести название факультета, которое хранится в таблице площадок `facilities`, необходимо найти его по соответствию `id` — (`facid` в `bookings` <--> `facid` в `facilities`):

```
SELECT f.name, b.starttime
FROM bookings b -- через пробел дали короткое имя (alias) b, чтобы в дальнейшем обращаться более коротким способом
JOIN facilities f on f.facid = b.facid;
```

Для более быстрого обращения к нужной таблице в запросе, ее можно временно переименовать, присвоив ей **алиас** (alias - псевдоним). Обычно псевдонимом выступают первые буквы названия таблицы.

Присвоить алиас можно как через оператор `AS` - `FROM` bookings `AS` b, так и через пробел - `FROM` bookings b. Далее при обращении к столбцам из нужной таблицы, указываем алиас перед точкой (b.bookings).

`JOIN` пишется по следующим правилам:

1. Он должен идти после `FROM`.
2. Он должен выглядеть следующим образом: `JOIN иная_таблица ON условие_соединения`.

При необходимости соединения более двух таблиц, `JOIN` применяется несколько раз (последовательно):

```
SELECT f.name, b.starttime, m.firstname, m.surname
FROM bookings b
```

```
JOIN facilities f on f.facid = b.facid
JOIN members m on b.memid = m.memid
WHERE firstname = 'David' AND DATE(b.starttime) > '2010-01-01';
```

Этот запрос отдаст нам имя площадки, время брони, имя и фамилию забронировшего для всех записей, где бронировавшего зовут 'David' и дата брони больше 2010-01-01.

Обратите внимание: мы воспользовались функцией в postgres. Это была функция `DATE(p.text)`, которая приняла на вход колонку `starttime` из таблицы `bookings` и вернула колонку с датой (отбросив время). Функции в postgres работают схожим с Python образом: тоже имеют аргументы и возвращаемое значение.

> Виды JOIN

Когда мы склеиваем две таблички, не всегда по memid в bookings может найтись соответствующая запись в members. И наоборот: не для каждого memid из members может найтись запись в bookings с соответствующим memid (к примеру, если пользователь ничего не бронировал).

В зависимости от необходимого алгоритма формирования таблицы, к оператору `JOIN` можно подставлять ключевые слова: `INNER`, `CROSS`, `FULL`, `LEFT`, `RIGHT`. По умолчанию (если нет ключевых слов) используется `INNER JOIN`.

LEFT JOIN

Берет все данные из FROM, затем берутся соответствующе данные (не все!) в таблице после JOIN. Там, где нет результатов в таблице после JOIN, выставляет NULL (символ пропуска значения).

```
SELECT f.name, b.starttime
FROM bookings b
LEFT JOIN facilities f on f.facid = b.facid; -- Берутся все площадки. Если для какого-нибудь bookings по facid не найдется площадка в facil
-- то f.name будет заполнен NULL
```

RIGHT JOIN

Аналогичен LEFT JOIN, только теперь наоборот: сначала берутся все данные из таблицы в RIGHT JOIN, затем берутся соответствующие данные из таблицы в FROM (не все). Там, где нет соответствующих значений в таблице FROM, ставятся NULL:

```
SELECT f.name, b.starttime
FROM bookings b
RIGHT JOIN facilities f on f.facid = b.facid; -- Берутся все площадки. Если для какого-нибудь facid из facilities не найдется facid в booki
-- то b.starttime будет заполнен NULL.
-- В любом случае будут возвращены все записи из facilites!
```

INNER JOIN

Выдаст в результате только пересечения обоих таблиц, то есть записи, присутствующие в обеих таблицах. По сути представляет собой комбинацию последовательных запросов LEFT JOIN и RIGHT JOIN.

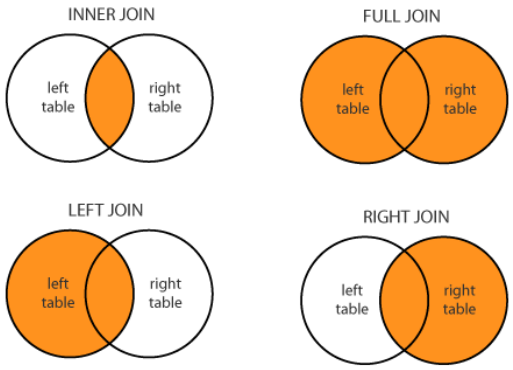
```
SELECT f.name, b.starttime
FROM bookings b
INNER JOIN facilities f on f.facid = b.facid;-- Возьмутся только те bookings, на которых найдется площадка в таблице facilities
-- и только те площадки, на которые найдется хотя бы одна запись в bookings.
-- Самый "узкий" из всех JOIN
```

FULL OUTER JOIN

Самый широкий JOIN. Берет все записи из обоих таблиц. Если нашлось соотвествие, то заполняет колонки, иначе помещает туда `NULL`.

```
SELECT f.name, b.starttime
FROM bookings b
```

```
FULL OUTER JOIN facilities f on f.facid = b.facid;-- Возьмутся все записи из bookings и все записи из facilities.
-- Там, где возможно установить соответствие, оно будет установлено.
-- Где соответствие провести нельзя, будет помещен NULL,
-- причем NULL может уйти как в таблицу слева, так и справа!
-- Самый "широкий" из всех JOIN
```



Основные типы JOIN

> SQL в Python

Чтобы работать с PostgreSQL в Python, используется специальная библиотека psycopg2. Она дает возможность подключаться к СУБД PostgreSQL и запускать SQL-запросы прямо внутри скрипта Python.

Устанавливается командой:

```
!pip install psycopg2-binary #-binary используется для того, чтобы библиотека устанавливалась уже скомпилированной
```

Вот так будет выглядеть запрос SQL в Python:

```
import psycopg2

connection = psycopg2.connect(    # connection - это объект, который отвечает за соединение с БД
    database='exercises',         # database - это база данных (именно база, не СУБД)
    host='localhost',             # это говорит, что СУБД работает на моем компьютере
    user='postgres',              # имя пользователя
    password='password'           # пароль
    # port=5432,                  # порт не указываем, по умолчанию 5432
)
cursor = connection.cursor()      # cursor - это объект, который отвечает за взаимодействие с БД
# Делаем запрос
cursor.execute("""
SELECT *
FROM cd.bookings -- cd есть схема, bookings есть таблица
-- синтаксис "схема.таблица"
LIMIT 10
""")
results = cursor.fetchall()       # Получаем результаты (fetchall() - "получить всё")
results                           # Это будет стандартный Python-объект. Не очень удобно, но работает
```

Обратите внимание, что сам запрос выделяется тройными кавычками (""") — этот знак в Python для многострочного ввода. Таким же образом выделяются многострочные комментарии в коде.

В конце работы необходимо закрывать курсор и соединение с БД:

```
cursor.close()
connection.close()
```

Как можно убедиться, код получается довольно громоздким, а вывод плохо читабельным. Тут нам на помощь приходит уже известная из прошлых модулей библиотека pandas.

> SQL в pandas

Для загрузки таблицы используется функция `pd.read_sql`.

```
import pandas as pd

# второй аргумент будет специальная строка. Для PostgreSQL имеет вид:
# postgresql://имя:пароль@хост:порт/база_данных
conn_uri = "postgresql://postgres:password@localhost/exercises"

df = pd.read_sql(
    "SELECT * FROM cd.bookings",             -- первый аргумент - SQL запрос
    conn_uri                                 -- наша строка с подключением
)
df.head()
```

```
df_1 = pd.read_sql(
    """
    SELECT
      b.starttime,
      b.slots,
      m.firstname,
      m.telephone
    FROM cd.bookings b
    INNER JOIN cd.members m
    ON b.memid = m.memid
    """,
    conn_uri,
    # можно явно указать, в каких колонках даты, их превратят в pd.datetime
    parse_dates=["starttime"]
)
df_1.sample(10, random_state=42) # выведет 10 выбранных случайным образом записей
```

Код получился намного компактнее, а выводом будет готовый датафрейм pandas с указанием всех столбцов.

При этом дополнительные манипуляции с данными (фильтрация, группировка и др) можно осуществлять как на языке SQL в рамках запроса, так и применять к уже выгруженным в формат датафрейма данным. Следующие два варианта кода выполняют одни и те же преобразования с данными. Тут все операции осуществляются в блоке SQL:

```
pd.read_sql(
    """
    SELECT b.starttime, COUNT(m.firstname)
    FROM cd.bookings b
    INNER JOIN cd.members m
    ON b.memid = m.memid
    GROUP BY b.starttime
    """,
    conn_uri,
    # можно явно указать, в каких колонках даты, их превратят в pd.datetime
    parse_dates=["starttime"]
```

А тут мы сначала выгрузили все необходимые данные из БД в формат датафрейма и после применили к ним функцию аггрегации:

```
df_1 = pd.read_sql(
    """
    SELECT
      b.starttime,
      b.slots,
      m.firstname,
      m.telephone
    FROM cd.bookings b
    INNER JOIN cd.members m
```

```
    ON b.memid = m.memid
    """,
    conn_uri,
    parse_dates=["starttime"]
)
df_1.groupby('starttime')['firstname'].count()
```

Сохранение новой таблицы

С помощью pandas можно не только читать таблицу, но и записывать данные обратно. Для этого используется функция `to_sql`:

```
df.to_sql(
"new_table",                    # имя таблицы, куда писать данные
conn_uri,                      # строка для подключения к БД
schema="cd",                   # Схема, в которой создать таблицу
if_exists='replace'            # что делать, если уже существует. Одно из трех значений:
)                               # 'fail' - выдать ошибку
# 'replace' - снести и создать с новыми данными
# 'append' - дополнить датафреймом, не трогая существующие данные
```