

Тема 1. Подзапросы и общие табличные выражения

Подзапросы — специальный механизм для линейного решения задач в SQL. Они помогают разбивать большой запрос на несколько более маленьких и выгружать данные из промежуточных таблиц.

```
-- подзапросы в FROM
SELECT поле_1,
       поле_2
FROM (SELECT поле_1,
             поле_2
      FROM таблица
     WHERE условие) AS псевдоним;
```

```
-- подзапросы в WHERE
SELECT поле_1,
       поле_2
FROM таблица
WHERE поле_1 IN (SELECT поле_1
                   FROM таблица
                  WHERE условие);
```

Общие табличные выражение (CTE) помогают структурировать подзапросы и выносить их за пределы основного кода. Подзапросы указывают после ключевого слова **WITH**.

```
-- общие табличные выражения
WITH
    псевдоним_1 AS (подзапрос_1),
    псевдоним_2 AS (подзапрос_2),
    псевдоним_3 AS (подзапрос_3),
    ...
    псевдоним_n AS (подзапрос_n)
SELECT /* основной запрос
внутри основного запроса работают с псевдонимами, которые назначили в WITH */
       FROM псевдоним_1 INNER JOIN псевдоним_2 ...
```

Тема 1. Подзапросы и общие табличные выражения

Рекурсивный запрос — запрос, который вызывает сам себя.

Чтобы задать базовый случай рекурсии, надо:

1. Определить первое значение.
2. Написать код для генерации последующих значений.
3. Сформулировать условие остановки.

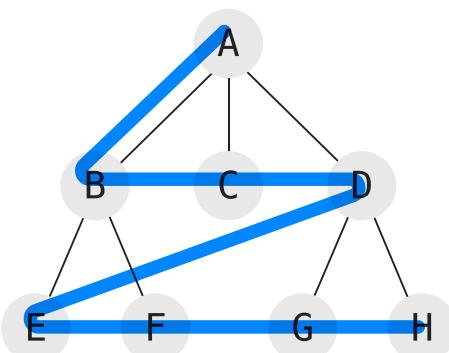
Синтаксис:

```
WITH RECURSIVE наименование_СТЕ [(список_столбцов_СТЕ)] AS (
    запрос_только_для_первого_значения
    UNION [ALL]
    SELECT список_столбцов_СТЕ FROM наименование_СТЕ
    [WHERE условие]
)
SELECT список_столбцов_СТЕ FROM наименование_СТЕ
[LIMIT условие];
```

Рекурсивные запросы используются **для иерархических структур**. Алгоритм СУБД может исследовать узлы уровней иерархии двумя путями.

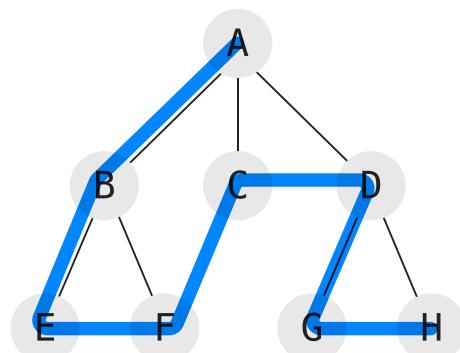
- Поиск в ширину — это такой рекурсивный обход иерархии, который начинается с корневого (или заданного условием) элемента и исследует уровни последовательно: сначала все узлы на текущем уровне, затем все узлы на уровне ниже и так далее. Этот обход используется в примере выше.
- Поиск в глубину — это такой рекурсивный обход иерархии, когда сначала мы спускаемся максимально глубоко по одной ветви, потом возвращаемся на предыдущий уровень вверх и получаем дочерние узлы этого уровня и так далее.

Поиск в ширину



Порядок в ширину:
A - B - C - D - E - F - G - H

Поиск в глубину



Порядок в глубину:
A - B - E - F - C - D - G - H

Тема 2. Оконные функции

Оконная функция — это функция, предназначенная для вычисления значений из набора записей, объединённых по конкретному признаку. Такой набор называют окном.

```
SELECT оконная_функция(поле) OVER (параметры (могут отсутствовать))
FROM таблица_1
```

Агрегирующие оконные функции

Оконная агрегирующая функция создаёт в итоговой таблице поле с агрегацией всего поля или по группам другого поля (полей) в зависимости от параметров, указанных в **OVER**.

```
SELECT SUM(поле) OVER (), -- подсчёт суммы всего поля в отдельном "столбце"
       COUNT(поле) OVER(),-- подсчёт числа записей всего поля в отдельном "столбце"
        AVG(поле) OVER (),-- подсчёт среднего всего поля в отдельном "столбце"
        MIN(поле) OVER (),-- подсчёт минимального значения всего поля в отдельном "столбце"
        MAX(поле) OVER ()-- подсчёт максимального всего поля в отдельном "столбце"
FROM таблица_1
```

Параметр **PARTITION BY** позволяет посчитать агрегацию не по всему полю, а с разделением на группы.

```
SELECT SUM(поле) OVER (PARTITION BY поле_2),
-- подсчёт суммы в отдельном "столбце" с разбиением на группы в поле_2
       COUNT(поле) OVER(PARTITION BY поле_2),
-- подсчёт числа в отдельном "столбце" с разбиением на группы в поле_2
        AVG(поле) OVER (PARTITION BY поле_2),
-- подсчёт среднего в отдельном "столбце" с разбиением на группы в поле_2
        MIN(поле) OVER (PARTITION BY поле_2),
-- подсчёт минимального в отдельном "столбце" с разбиением на группы в поле_2
        MAX(поле) OVER (PARTITION BY поле_2)
-- подсчёт максимального в отдельном "столбце" с разбиением на группы в поле_2
FROM таблица_1
```

Тема 2. Оконные функции

Кумулятивные значения — значения, подсчитывающиеся с накоплением.

Любая агрегирующая функция преобразуется в кумулятивную функцию, если к ней добавляется параметр окна **ORDER BY**:

```
SELECT SUM(revenue) OVER (ORDER BY order_data)
-- будет подсчитана кумулятивная сумма поля revenue в зависимости от
-- порядка дат в поле order_data, то есть на каждую следующую дату
-- будет рассчитана сумма поля revenue всех предыдущих дат.

    AVG(revenue) OVER (ORDER BY order_data)
-- будет подсчитано кумулятивное среднее поля revenue в зависимости от
-- порядка дат в поле order_data, то есть на каждую следующую дату
-- будет рассчитано среднее значение поля revenue всех предыдущих дат.

    MIN(revenue) OVER (ORDER BY order_data)
-- в новом поле будет рассчитано минимальное значение поля revenue,
-- в зависимости от порядка дат в поле order_data, то есть в новом поле будет
-- присутствовать минимальное значение только от предыдущих значений

    MAX(revenue) OVER (ORDER BY order_data)
-- аналогично с MIN
FROM таблица_1
```

Тема 2. Оконные функции

Оконные функции ранжирования

Эти оконные функции позволяют ранжировать записи, то есть нумеровать их по определённому правилу.

ROW_NUMBER() возвращает номер строки в зависимости от параметров. Номера строк начинаются с 1 и увеличиваются на 1 для каждой следующей строки.

RANK() возвращает ранг строки в зависимости от параметров. Если несколько строк имеют одинаковые значения, они получают один и тот же ранг, а следующий ранг будет пропущен.

DENSE_RANK() возвращает ранг строки в зависимости от параметров. Если несколько строк имеют одинаковые значения, они получают один и тот же ранг, и следующий ранг не будет пропущен.

Задать порядок назначения рангов помогает параметр оконной функции ORDER BY (оператор сортировки называется так же, но используется по-другому, будьте внимательны):

- Чтобы сформировать порядок по возрастанию, указывается ASC или ничего не указывается.
- Для порядка по убыванию указывается DESC.

Если использовать функции ранжирования в сочетании с PARTITION BY, то получится упорядоченное ранжирование по группам.

```
SELECT ROW_NUMBER() OVER (),  
-- назначит номер для каждой строки таблицы  
    RANK() OVER(PARTITION BY поле_2 ORDER BY поле_3 DESC),  
-- назначит ранги в обратном порядке от поля_3 с разделением на группы в поле_2  
-- если несколько строк поля_3 имеют одинаковые значения,  
-- они получают один и тот же ранг, а следующий ранг будет пропущен.  
    DENSE_RANK() OVER (ORDER BY поле_3),  
-- назначит ранги в прямом порядке от поля_3.  
-- если несколько строк в поле_3 имеют одинаковые значения,  
-- они получают один и тот же ранг, и следующий ранг не будет пропущен.  
FROM таблица_1
```

Тема 2. Оконные функции

Оконные функции смещения

Оконная функция **LAG()** позволяет получить значение поля предыдущей строки в рамках заданного окна и порядка формирования:

```
SELECT LAG(revenue) OVER (ORDER BY order_data)
-- в новом поле итоговой таблицы сформируются предыдущие значения
-- в зависимости от поля с датами order_data
FROM таблица_1
```

Оконная функция **LEAD()** позволяет получить значение поля следующей строки в рамках заданного окна и порядка формирования:

```
SELECT LEAD(revenue) OVER (ORDER BY order_data)
-- в новом поле итоговой таблицы сформируются последующие значения
-- в зависимости от поля с датами order_data
FROM таблица_1
```

Тема 3. Представления

Представление в PostgreSQL — это псевдотаблица, сохранённый запрос **SELECT** к одной или нескольким таблицам.

Создание представлений:

```
CREATE VIEW имя_представления AS  
    SELECT ... ; -- здесь можно подставить SELECT-запрос любой сложности
```

Изменение имени представления:

```
ALTER VIEW имя_представления RENAME TO имя_нового_представления;
```

Пересоздание представления:

```
DROP VIEW имя_представления;  
CREATE VIEW имя_представления AS  
    SELECT ... ; -- здесь подставьте новый запрос
```

Материализованные представления отличаются от простых представлений тем, что результат запроса **SELECT** сохраняется на диске.

Создание материализованного представления:

```
CREATE MATERIALIZED VIEW имя_представления AS  
    SELECT ...;
```

Удаление материализованного представления:

```
DROP MATERIALIZED VIEW имя_представления;
```

Изменение материализованного представления:

```
DROP MATERIALIZED VIEW имя_представления;  
CREATE MATERIALIZED VIEW имя_представления AS  
    SELECT ...;
```

Переименование материализованного представления:

```
ALTER MATERIALIZED VIEW имя_представления RENAME TO новое_имя_представления;
```

Обновление материализованного представления:

```
REFRESH MATERIALIZED VIEW имя_представления;
```

Тема 4. Продвинутые типы данных

Продвинутые типы данных в этой теме:

- **uuid**,
- перечисляемый тип **enum**,
- составной тип,
- массивы,
- **json** и **jsonb**.

Тип **uuid**

uuid — это 128-битное значение. Если в качестве уникального идентификатора используют тип **uuid**, то первичный ключ задают с помощью функции **GEN_RANDOM_UUID()**.

```
--сгенерировать значение  
SELECT GEN_RANDOM_UUID()
```

Перечисляемый тип **enum**

```
CREATE TYPE имя_типа AS ENUM (список_значений_через_запятую);
```

Составной тип

Создание:

```
CREATE TYPE имя_типа AS (имя_поля тип_поля, имя_поля_2 тип_поля_2, ...)
```

Вставка значения через текстовую константу:

```
UPDATE cars SET specifications = '(AAA12345678901234,2015,белый,1.3,136)'  
WHERE gos_num = 'A111AA111'
```

Вставка значения с помощью выражения **ROW**:

```
INSERT INTO cars (gos_num, engine, specifications)  
VALUES ('M555MM555', 'гибридный', ROW('MMM12345678901234', 2021, 'чёрный', NULL,  
NULL))
```

Тема 4. Продвинутые типы данных

Массивы

Массив — это упорядоченный набор элементов одного типа. Массивы бывают одномерные и многомерные.

Размерность — это число измерений. В двумерном массиве размерность равна двум, в трёхмерном — трём и так далее.

Размер — это количество элементов в массиве, или его длина. В многомерном массиве размер определяется отдельно по каждому измерению.

При создании поля с типом данных «массив» указывают тип данных элементов массива в квадратных скобках [] или ключевое слово **ARRAY**. Многомерный массив можно объявить так же, как одномерный, или использовать несколько пар квадратных скобок — столько, сколько измерений предполагается в массиве.

Чтобы **вставить значения массива в таблицу**, применяют два варианта синтаксиса:

- Фигурные скобки {}.

```
INSERT INTO clients(id, client_name, phone, add_phones)
VALUES (1, 'Лесной Анатолий Игоревич', '71112223344',
'{"78885556677", "73335556677"}')
```

Этот синтаксис подходит и для многомерных массивов: в этом случае вместо элементов указывают вложенные массивы.

```
INSERT INTO depots_economic_report(id, depot_id, date_begin, economic_data)
VALUES (1, 1, current_date, '{100, 40}, {200, 85}'))
```

- Конструктор **ARRAY**.

С помощью ключевого слова **ARRAY** можно вставлять в таблицу одномерные и многомерные массивы.

```
INSERT INTO clients(id, client_name, phone, add_phones)
VALUES (2, 'Полевой Александр Павлович', '73332221144',
ARRAY['78881112233', '77772224477'])
```

В многомерном массиве вложенные массивы заключают во вложенные квадратные скобки.

Функции и операторы для работы с массивами представлены в таблице.

Тема 4. Продвинутые типы данных

Массивы: операторы и функции

Оператор/Функция	Что делает
Оператор конкатенации <code> </code>	Обновляет данные массива, добавляя новый элемент.
Функция <code>ARRAY_APPEND</code>	Обновляет данные массива, добавляя новый элемент.
Оператор <code>ANY</code>	Находит элемент массива, если индекс элемента заранее неизвестен.
Оператор <code>&&</code>	Находит элемент массива, если индекс элемента заранее неизвестен.
Оператор <code>ALL</code>	Проверяет на соблюдение условия все элементы массива одновременно.
Операторы сравнения <code>=</code> и <code><></code>	Сравнивает массивы друг с другом.
Операторы вхождения <code>@></code> и <code><@</code>	Определяет вхождение одного массива в другой.
Функция <code>ARRAY_LENGTH</code>	Возвращает длину массива.
Функция <code>ARRAY_TO_STRING</code>	Передаёт из БД элементы массива, разделённые строкой.
Функция <code>STRING_TO_ARRAY</code>	Вставляет данные, разделённые строкой, в таблицу в виде массива.
Функция <code>UNNEST</code>	Принимает в качестве аргумента массив и разворачивает его на набор строк.

Тема 4. Продвинутые типы данных

Типы json и jsonb

json — это текстовый формат. Его широко используют для хранения неструктурированной информации, которая поступает из различных источников. В PostgreSQL, помимо json, также есть свой уникальный тип данных jsonb. Он выглядит так же, как обычный json, но хранит данные в двоичном формате. Различия этих типов — в таблице.

json vs jsonb

json	jsonb
Сохраняет точную копию введённого текста.	Введённый текст сохраняется в разобранном двоичном виде.
Сохраняет незначимые пробелы между элементами.	Удаляет незначимые пробелы между элементами.
Сохраняет повторяющиеся ключи — при обработке будет выдаваться только последний.	Если есть повторяющиеся ключи, сохраняет только последний.
Сохраняет порядок ключей.	Не сохраняет порядок ключей.
Данные вводятся быстрее.	Данные вводятся медленнее из-за необходимости двоичного разбора.
Данные обрабатываются медленнее.	Данные обрабатываются быстрее, потому что не нужен многократный разбор текста.
Не поддерживает индексацию.	Поддерживает индексацию.
Не поддерживает индексацию.	Поддерживает индексацию.
Не поддерживает операторы конкатенации, вхождения, присутствия и удаления.	Поддерживает операторы конкатенации, вхождения, присутствия и удаления.
Не поддерживает встроенные функции для замены значения ключа.	Можно заменить значение ключа с помощью встроенной функции.

Тема 4. Продвинутые типы данных

Функции и операторы для работы с `json` и `jsonb`:

`json` и `jsonb`: операторы и функции

Оператор/Функция	Что делает
Оператор чтения и поиска <code>->></code>	Возвращает результат типа <code>text</code> .
Оператор чтения и поиска <code>-></code>	Возвращает результат типа <code>json</code> или <code>jsonb</code> . Ищет элементы массива по индексу элемента.
Операторы поиска <code>#></code> и <code>#>></code>	Выдают значение по пути — последовательному перечню всех вложенных друг в друга ключей, которые ведут к искомому значению.
Оператор конкатенации <code> </code> для <code>jsonb</code>	Соединяет два объекта <code>jsonb</code> .
Операторы вхождения <code>@></code> и <code><@</code>	Проверяет вхождение одного объекта в другой.
Оператор присутствия <code>? </code>	Позволяет узнать, присутствует ли любой элемент из массива.
Оператор присутствия <code>?&</code>	Позволяет узнать, встречаются ли все элементы массива в качестве ключей.
Оператор удаления <code>-</code>	Удаляет пару <code>ключ-значение</code> из объекта или объект из массива.
Функции <code>JSON_ARRAY_LENGTH</code> и <code>JSONB_ARRAY_LENGTH</code>	Находят количество элементов (длину) массива <code>json</code> или <code>jsonb</code> .
Функции <code>JSON_EACH</code> и <code>JSONB_EACH</code>	Разворачивают <code>json</code> или <code>jsonb</code> в набор пар <code>ключ-значение</code> .
Функции <code>JSON_EACH_TEXT</code> и <code>JSONB_EACH_TEXT</code>	Разворачивают <code>json</code> или <code>jsonb</code> в набор пар <code>ключ-значение</code> и возвращают все значения в формате <code>text</code> .
Функции <code>JSON_ARRAY_ELEMENTS</code> и <code>JSONB_ARRAY_ELEMENTS</code>	Разворачивают массив в набор значений.
Функция <code>JSONB_SET</code> для <code>jsonb</code>	Заменяет в объекте <code>jsonb</code> любое значение ключа на новое.

Тема 5. Геоданные

Геоданные — это информация о местоположении и атрибутах объектов на поверхности Земли.

Использовать геоданные в PostgreSQL помогает расширение PostGIS. Оно превращает сервер баз данных в геоинформационную систему.

geometry и **geography** — самые распространённые типы данных в PostGIS.

```
-- задаём тип данных при создании таблицы
CREATE TABLE my_geodata (
    id SERIAL PRIMARY KEY,
    place TEXT,
    gm GEOMETRY,
    gg GEOGRAPHY
);
```

Основные двумерные объекты PostGIS: Точка (Point), Линия (LineString), Полигон (Polygon), Мультиполигон, Коллекция точек (MultiPoint), Коллекция линий (MultiLineString).

Форматы данных

Объекты в PostGIS можно представить в разных форматах.

- **WKT (Well-Known Text)** — текстовый формат, представляет геометрический объект в виде текстовой строки, которая содержит тип объекта и его координаты — могут быть целыми или дробными.

```
-- Ключевое слово POINT и координаты в скобках, разделённые пробелом.
POINT(2 4)
```

- **WKB (Well-Known Binary)** — формат бинарных данных, используется для внутреннего представления геометрических данных в бинарном формате. Если в запросе нужно получить геообъект, то в зависимости от того, в каком виде находится WKB, необходимо применять разные способы конвертации.

```
-- Здесь зашифрована точка 'POINT(1 1)'.
SELECT '01010000000000000000F03F000000000000F03F'::geometry;
```

- **GeoJSON (Geographic JSON)** — специальный формат данных в геоинформационных системах, представляет данные на поверхности Земли. Объект GeoJSON содержит два обязательных атрибута: type — тип объекта, coordinates — координаты объекта. Координаты задаются перечислением точек, каждая точка — это массив из двух чисел (долгота и широта).

```
-- Московский Кремль в формате GeoJSON выглядит так
{
  "type": "Point",
  "coordinates": [37.61733, 55.750997]
}
```

Тема 5. Геоданные

SRID

SRID — это числовой идентификатор, который определяет систему координат и проекцию пространственных данных в геоинформационных системах. Для типа **geometry** значение SRID равняется нулю, для типа **geography** SRID по умолчанию — это 4326.

```
CREATE TABLE geometry_4326 (
    id SERIAL PRIMARY KEY,
    place TEXT,
    -- Указываем тип данных в столбце и его SRID вторым аргументом через запяту
    -- ю.
    gm geometry(POINT, 4326)
);
```

Функции для работы с геоданными

Название функции	Для чего нужна
ST_GeomFromText	создать геометрический объект из его текстового представления
ST_GeomFromWKB	для конвертации из WKB в геометрический объект
ST_AsText	сконвертировать данные в WKT
ST_SRID	узнать SRID данных
ST_SetSRID	назначить SRID
ST_Transform	изменить координаты соответственно указанной SRID
ST_AsGeoJSON	преобразовать данные в GeoJSON
ST_GeomFromGeoJSON	получить геометрический объект из GeoJSON
ST_Equals	роверяет тождество двух геометрических объектов, возвращая булево значение true или false
ST_Within(A, B)	возвращает true , если все точки объекта A находятся внутри объекта B
ST_Contains(A, B)	возвращает true , если нет точек объекта B, которые лежат снаружи объекта A, и хотя бы одна точка объекта B лежит внутри A
ST_Covers(A, B)	возвращает true , если ни одна точка объекта B не находится снаружи объекта A
ST_Intersects	определяет, пересекаются ли объекты. Она возвращает true или false
ST_Intersection	возвращает непосредственно геометрические объекты — пересечения исходного объекта и объектов из таблицы
ST_Distance	вычисляет расстояния между геометрическими объектами
ST_Length	вычисляет длину линии или мультилиний
ST_Perimeter	считает периметр полигона и мультиполигона
ST_Area	вычисляет площади

Тема 5. Геоданные

Примеры использования некоторых функций:

Трансформация координат:

```
SELECT ST_AsText(ST_Transform(gm, 3857)) FROM geometry_4326;
```

Назначение SRID:

```
SELECT ST_SetSRID(gm, 3857) FROM geometry_4326;
```

Выяснение SRID:

```
SELECT ST_SRID(gm) FROM geometry_4326;
```

Конвертация данных в WKT:

```
SELECT
    id,
    place,
    ST_AsText(gm) geometry_text,
    ST_AsText(gg) geography_text
FROM my_geodata;
```

Конвертация в объект из WKB:

```
SELECT ST_GeomFromWKB('x01010000000000000000f03f000000000000f03f');
```

Создание текстового представления из объекта:

```
SELECT ST_GeomFromText('LINESTRING(-5 0, -4 4, -2 -4, 1 2, 12 0)');
```

Вычисление площади:

```
SELECT ST_Area(
    ST_GeomFromGeoJSON('{
        "type": "Polygon",
        "coordinates": [
            -- Здесь 65 точек полигона, полностью приводить не будем.
        ]
    }'))::geography
)
```

Тема 5. Геоданные

Вычисление расстояния:

```
SELECT ST_Distance(
    ST_Transform(ST_GeomFromText('POINT(30.316 59.939)', 4326), 3857),
    ST_Transform(ST_GeomFromText('POINT(37.6205 55.7541)', 4326), 3857)
)
```

Проверка вхождения:

```
SELECT hs.id, hs.name, hs.category, c.city
FROM household_services hs
JOIN cities c ON ST_Within(hs.dislocation, c.shape);
```

Тема 6. Транзакции и блокировки

Транзакции

Транзакция в БД — это одна или несколько операций, которые выполняются как единое целое.

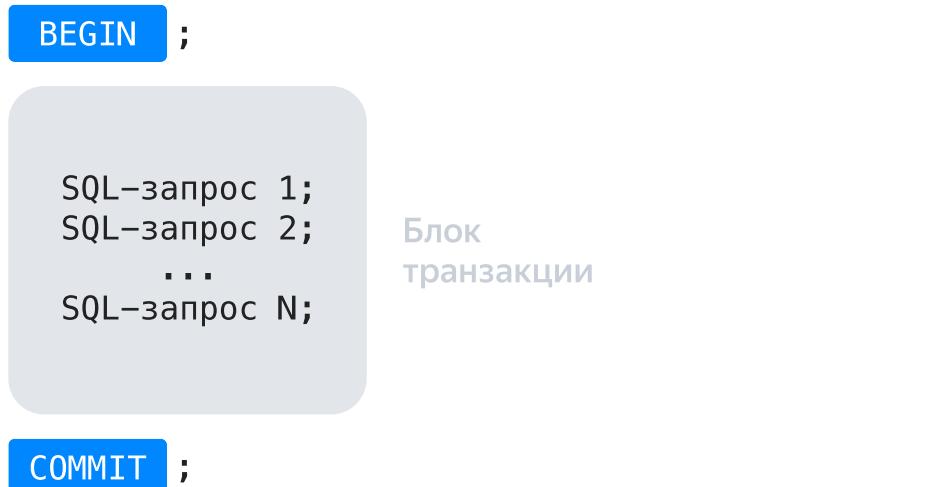
Требования к транзакциям (ACID):

- **Атомарность (A — Atomicity)** — указывает на неделимость и цельность транзакции. Это значит, что все операции в транзакции либо успешно выполняются вместе, либо отменяются.
- **Согласованность (C — Consistency)** — транзакция не разрушает взаимной согласованности данных. Это значит, что каждая успешная транзакция фиксирует только допустимые для этой БД результаты.
- **Изолированность (I — Isolation)** — обеспечивает независимость каждой транзакции от других. Каждая транзакция выполняется так, как если бы она была единственной операцией, которая происходит в системе. Это даёт надёжность обработки данных и помогает избежать проблем, которые могут возникнуть при одновременном доступе к общим данным.
- **Долговечность или надёжность (D — Durability)** — гарантирует, что если пришло подтверждение о выполнении транзакции, то изменения, вызванные этой транзакцией, уже не могут быть потеряны из-за каких-либо сбоев (например, отключения электропитания). Изменения гарантированно внесены в БД, а данные согласованы.

Команды в транзакциях:

- **BEGIN ;** — начинает транзакцию,
- **COMMIT ;** — заканчивает, подтверждает транзакцию.

Блок транзакции — код SQL между **BEGIN ;** и **COMMIT ;**



Тема 6. Транзакции и блокировки

Использование откатов транзакций и точек сохранения

`ROLLBACK` — откатит состояние базы данных к тому, что было до начала транзакции.

`ROLLBACK TO [SAVEPOINT] имя_точки_сохранения` — откатит транзакцию к точке сохранения, то есть отменит запросы, которые были выполнены после неё.

`RELEASE [SAVEPOINT] имя_точки_сохранения` — удалит точку сохранения.

Тема 6. Транзакции и блокировки

Уровни изоляции

Уровни от самого свободного до самого строгого:

Уровень изоляции	Описание	«Грязное» чтение	Неповторяющееся чтение	Фантомное чтение	Аномалия сериализации
READ UNCOMMITTED	Чтение незафиксированных данных	В PostgreSQL невозможно	Возможно	Возможно	Возможно
READ COMMITTED	Чтение зафиксированных данных	Невозможно	Возможно	Возможно	Возможно
REPEATABLE READ	Повторяющееся чтение	Невозможно	Невозможно	В PostgreSQL невозможно	Возможно
SERIALIZABLE	Сериализуемость	Невозможно	Невозможно	Невозможно	Невозможно

Варианты установки уровня изоляции:

- указать при старте

```
BEGIN TRANSACTION ISOLATION LEVEL <*нужный_уровень_изоляции*>;
```

- указать внутри транзакции

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL <*нужный_уровень_изоляции*>;
```

Тема 6. Транзакции и блокировки

Блокировки

Виды блокировок по строгости действия:

- **Разделяемая** (shared lock) — позволяет другим транзакциям читать данные, но не позволяет изменять их, пока блокировка активна. Блокировки на чтение обычно разделяемые — это обеспечивает целостность данных при чтении.
- **Исключительная** (exclusive lock) — позволяет только одной операции в определённый момент времени читать или изменять данные. Запрещает любой доступ к данным для других транзакций до тех пор, пока блокировка не снимется. Это означает, что пока эта блокировка активна, ни одна другая транзакция не может ни читать, ни изменять данные.

Виды блокировок по области действия:

- **На уровне таблицы.** Блокирует всю таблицу целиком. Другие процессы не могут читать или изменять данные в этой таблице, пока текущая транзакция не завершится.

```
LOCK TABLE имя_таблицы IN уровень_блокировки MODE;
```

- **На уровне строки.** Ограничивает доступ только к конкретным строкам. Это удобно, когда нужно точечно изменить или посмотреть что-то конкретное, не мешая другим работать с остальными данными таблицы.

```
BEGIN;  
  
SELECT ...  
FOR вариант_блокировки  
[OF имя_таблицы]  
[NOWAIT]
```

Тема 6. Транзакции и блокировки

Режимы блокировки строк

Режим	Как работает	Вид блокировки	Одновременная активность
<code>FOR SHARE</code>	Разрешает другим транзакциям читать строку. Вносить изменения нельзя.	Разделяемая	Несколько транзакций
<code>FOR KEY SHARE</code>	Разрешает другим транзакциям вносить изменения в строку, но только для не ключевых атрибутов — явно заданные первичные и внешние ключи менять нельзя. В PostgreSQL часто применяется автоматически при контроле соответствия внешних ключей.	Разделяемая	Несколько транзакций
<code>FOR UPDATE</code>	Разрешает полностью менять или удалять строки.	Исключительная	Одна транзакция
<code>FOR NO KEY UPDATE</code>	Разрешает изменять только те поля, которые не входят в явно заданные первичные и внешние ключи — при таком изменении внешние ключи не меняются.	Исключительная	Одна транзакция

Совместимость режимов

Транзакция (1)	<code>FOR UPDATE</code>	<code>FOR NO KEY UPDATE</code>	<code>FOR SHARE</code>	<code>FOR KEY SHARE</code>
Транзакция (2)				
<code>FOR UPDATE</code>	Нет	Нет	Нет	Нет
<code>FOR NO KEY UPDATE</code>	Нет	Нет	Нет	Да
<code>FOR SHARE</code>	Нет	Нет	Да	Да
<code>FOR KEY SHARE</code>	Нет	Да	Да	Да