

Лабораторная работа №2. Основы DML-запросов в PostgreSQL

Цели лабораторной работы:

- освоить основные варианты DML-запросов в PostgreSQL;
- научиться создавать SQL-скрипты для заполнения таблиц данными;
- научиться работать с представлениями;
- освоить планы выполнения запросов.

Задание на лабораторную работу:

При выполнении работы используется БД, созданная в лабораторной работе №1.

Нужно выполнить следующие шаги:

1. Создать и выполнить SQL-скрипт, который будет заполнять таблицы данными. Нужно добавить не менее 3-5 строк в каждую таблицу.
2. Создать представления для нескольких таблиц, в которых собираются данные из самой таблицы и других, на которые она ссылается. Выборка из любого представления должна давать полную и осмысленную информацию по сущностям. Хотя бы одно из представлений должно быть сделано с использованием соединений (join) в запросе.
3. Для любой таблицы, содержащей столбец с числовыми данными, создать представление следующего вида, отражающее информацию по этому столбцу (в представлении ровно 3 столбца и 4 строки):

Минимальное значение	<значение>	<id записи с минимальным значением>
Максимальное значение	<значение>	<id записи с максимальным значением>
Среднее значение	<значение>	null
Сумма значений	<значение>	null

4. Создать и выполнить SQL-скрипты для выполнения различных планов запросов.

Отчет по лабораторной работе должен содержать:

- созданные скрипты с комментариями (скрипт для заполнения таблиц данными, скрипты, создающие представления и скрипты планов выполнения);
- примеры результатов выборок из созданных представлений;
- графическое представление планов выполнения запросов.

Выполнение лабораторной работы:

Ниже показан скрипт, который был создан в процессе выполнения лабораторной работы 1. Выполнение данной лабораторной работы выполним с его использованием.

```
--Создание таблицы ГРУППЫ
create table public.group(
    id bigserial primary key,
    name varchar(30) not null unique,
    number_of_students int not null check (number_of_students >= 1)
);

--Создание таблицы СТУДЕНТЫ
create table public.student(
    id bigserial primary key,
    name varchar(30) not null,
    surname varchar(30) not null,
    middle_name varchar(30),
    id_group bigint not null references public.group(id),
    address varchar(100),
    course int check (course >= 1 and course <=5)
);

--Создание таблицы ПРЕДМЕТЫ
create table public.subject(
    id bigserial primary key,
    name varchar(50) not null unique,
    number_of_hours int not null check (number_of_hours > 0)
);

--Создание таблицы УСПЕВАЕМОСТЬ
create table public.progress(
    id_student bigint not null references public.student(id),
    id_subject bigint not null references public.subject(id),
    rating int check (rating >= 2 and rating <=5)
);

--Создание типа перечисления для обозначения должности преподавателя
create type public.position_type_enum as enum (
    'docent', 'professor', 'senior lecturer', 'instructor');

--Создание таблицы ПРЕПОДОВАТЕЛИ
create table public.teacher(
    id bigserial primary key,
    name varchar(30) not null,
    surname varchar(30) not null,
    middle_name varchar(30),
    position public.position_type_enum not null default 'instructor',
    phone_number varchar(30)
);

--Создание типа перечисления для обозначения типа проводимого занятия
create type public.occupation_type_enum as enum (
    'lecture', 'laboratory work', 'practical lesson', 'seminar');

--Создание таблицы ТИП ЗАНЯТИЯ
create table public.type_of_occupation(
    id_subject bigint not null references
public.subject(id),
    id_teacher bigint not null references
public.teacher(id),
```

```

occupation_type public.occupation_type_enum not null
    default 'lecture'
);

```

- I. Создать и выполнить SQL-скрипт, который будет заполнять таблицы данными. Нужно добавить не менее 3-5 строк в каждую таблицу.

Команда insert

Команда insert используется для вставки строк в таблицу. За один запрос можно вставить ноль и более строк. Она может как добавлять строки с указанными явно значениями, так и добавлять строки, которые были получены из дополнительного запроса (см. [пункт «Подзапросы и предложение with»](#)).

Здесь будут рассмотрены базовые способы применения команды insert. *Подробнее о синтаксисе команды insert можно прочитать в документации.*

-- Пример 1

-- Вставляет новую строку в customer, id проставляется автоматически.
 -- Поле команды вставки insert into указывается имя таблицы, в скобках перечисляются заполняемые поля, затем после ключевого слова values указываются значения этих полей.

```

insert into group(name, number_of_students)
values ('ИВТ6-3302-02-00', 20);

```

-- Пример 2

-- Если указать id явно, как в примере ниже, он будет установлен в 15.
 -- При этом последовательность id никак не сдвинется!
 -- При вставке следующей строки без id случится конфликт, когда последовательность дойдет до 15.
 -- Поэтому никогда не надо ставить руками id, которые имеют тип bigserial!
 -- Если очень нужно так сделать, то после этого обязательно проверить состояние последовательности и сдвинуть её при необходимости.

```

insert into group(id, name, number_of_students)
values (15, 'ИВТ6-3302-02-00', 20);

```

-- Пример 3

-- За один раз можно указать несколько строк для вставки

```

insert into group(name, number_of_students)
values ('ИВТ6-3301-01-00', 15),
      ('ИВТ6-3302-02-00', 20),
      ('ИВТ6-4301-01-00', 12);

```

-- Пример 4

-- Два запроса ниже аналогичны, т.к. у teacher есть значение по умолчанию.
 -- Если вместо default в первом запросе указать конкретное значение, то будет вставлено оно.

```

insert into teacher(first_name, second_name, middle_name, position,
phone_number)
values ('Николай', 'Сергеевич', 'Котов', default, '89530000000');
insert into teacher(first_name, second_name, middle_name, phone_number)
values ('Николай', 'Сергеевич', 'Котов', '89530000000');

```

При создании скриптов для заполнения таблиц данными главной проблемой является поддержание ссылочной целостности. Вручную проставлять числовые значения столбцов с внешними ссылками неправильно: практически никогда нельзя с уверенностью сказать, какие первичные ключи будут сгенерированы в таблице, на которые делаются ссылки. Даже если это можно сделать, читать и модифицировать такой скрипт тяжело.

Один из вариантов решения проблемы – использование запросов insert с предложениями with (см. [пример 69](#)).

Тем не менее, при вставке большого количества данных со сложными зависимостями это может быть неудобным. Другой вариант – использовать скрипт на PL/pgSQL. Работа с PL/pgSQL подробнее рассматривается в лабораторной работе №3. В рамках текущей лабораторной работы можно воспользоваться примером 5.

```
-- Пример 5
do $$
declare

-- Здесь описываются переменные, которые будут хранить id новых записей
groupID bigint;

begin

-- Используя слово into в предложении returning, можно записывать результат
в переменную

insert into group(name, number_of_students)
values ('ИВТ6-3301-01-00', 15)
returning id into groupID;

insert into student(first_name, second_name, middle_name, id_group,
address, course)
values ('Котов', 'Сергеевич', 'Николай', groupID, 'г. Киров ул. Ломоносова
16а', 3);

insert into student(first_name, second_name, middle_name, id_group,
address, course)
values ('Олоев', 'Петрович', 'Василий', groupID, 'г. Киров ул. Свободы 4',
3);

insert into group(name, number_of_students)
values ('ИВТ6-3302-02-00', 20)
returning id into groupID;

insert into student(first_name, second_name, middle_name, id_group,
address, course)
values ('Зотов', 'Георгиевич', 'Олег', groupID, 'г. Киров ул. Воровского
122', 3);

insert into group(name, number_of_students)
values ('ИВТ6-4301-01-00', 12)
returning id into groupID;
```

```
insert into student(first_name, second_name, middle_name, id_group,
address, source) values ('Романов', 'Владимирович', 'Константин', groupID,
'г. Киров ул. Ленина 52', 4);

end;
$$ language plpgsql;
```

- II. Создать представления для нескольких таблиц, в которых собираются данные из самой таблицы и других, на которые она ссылается. Выборка из любого представления должна давать полную и осмысленную информацию по сущностям. Хотя бы одно из представлений должно быть сделано с использованием соединений (join) в запросе.

Команда select

Команда select используется для получения строк из таблиц. Она обладает очень гибким синтаксисом и огромным количеством возможностей, позволяя получать данные из некоторого множества таблиц (возможно, пустого), фильтруя, сортируя, объединяя и предварительно обрабатывая их различными способами.

В рамках этого раздела будут рассмотрены базовые способы применения команды select. *Подробнее о синтаксисе команды select можно прочитать в документации.*

-- Пример 6

-- Получить все строки из таблицы student
 -- После ключевого слова select указываются столбцы, которые необходимо получить, после ключевого слова from указываются имена таблиц, из которых будут выбраны столбцы
select * from student;

id	name	surname	middle_name	id_group	address	source
1	Николай	Котов	Сергеевич	4	г. Киров ул. Ломоносова 16а	3
2	Василий	Олоев	Петрович	4	г. Киров ул. Свободы 4	3
3	Олег	Зотов	Георгиевич	5	г. Киров ул. Воровского 122	3
4	Константин	Романов	Владимирович	6	г. Киров ул. Ленина 52	4

-- Пример 7

-- Получить только значения столбца address из таблицы customer
select address from student;

address
г. Киров ул. Ломоносова 16а
г. Киров ул. Свободы 4
г. Киров ул. Воровского 122
г. Киров ул. Ленина 52

-- Пример 8

-- С помощью ключевого слова **as** можно изменять имя столбца в результирующей выборке

```
select id, address as adrs from student;
```

id	adrs
1	г. Киров ул. Ломоносова 16а
2	г. Киров ул. Свободы 4
3	г. Киров ул. Воровского 122
4	г. Киров ул. Ленина 52

Если выполнить **select** из нескольких таблиц (без каких-либо фильтров), то результатом запроса будет декартово произведение множеств их строк.

-- **Пример 9**

-- Выборка всех данных из таблицы *subject*

```
select * from subject;
```

id	name	number_of_hours
1	Технологии программирования	18
2	История	36

-- Выборка всех данных из таблицы *progress*

```
select * from progress;
```

id_student	id_subject	rating
1	1	4
2	1	5
4	2	5
2	2	4
3	2	3

-- Выборка всех данных из обеих таблиц - декартово произведение множеств их строк

-- Если одна из таблиц пуста, то результатом запроса будет пустое множество

```
select * from subject, progress;
```

id	name	number_of_hours	id_student	id_subject	rating
1	Технологии программирования	18	1	1	4
2	История	36	1	1	4
1	Технологии программирования	18	2	1	5
2	История	36	2	1	5
1	Технологии программирования	18	4	2	5
2	История	36	4	2	5
1	Технологии программирования	18	2	2	4
2	История	36	2	2	4
1	Технологии программирования	18	3	2	3
2	История	36	3	2	3

Предложение *where*

Чтобы отфильтровать результат запроса, используется предложение `where`. В нем задается любое выражение, имеющее тип `boolean`. Любая строка, не удовлетворяющая этому условию, исключается из результата. Строка удовлетворяет условию, если оно возвращает `true` при подстановке вместо ссылок на переменные фактических значений из этой строки. Ниже показаны примеры использования предложения `where`.

-- Пример 10

-- Этот запрос вернет все строки из `subject`
`select * from subject where true;`

id	name	number of hours
1	Технологии программирования	18
2	История	36

-- Пример 11

-- Этот запрос вернет 0 строк
`select * from subject where false;`

-- Пример 12

-- Этот запрос вернет только одну строку, поскольку только одна строка отвечает условию `id = 2`
`select * from subject where id = 2;`

id	name	number of hours
2	История	36

-- Пример 13

-- Этот запрос вернет строки, в которых находится информация о предмете и соответствующей оценке
`select *`
`from subject, progress`
`where subject.id = progress.id_subject;`

id	name	count of hours	id_student	id_subject	rating
1	Технологии программирования	18	1	1	4
1	Технологии программирования	18	2	1	5
2	История	36	4	2	5
2	История	36	2	2	4
2	История	36	3	2	3

-- Данный запрос работает точно так же как и предыдущий
-- В данном случае для столбца `id` в `where` не обязательно указывать имя таблицы `subject`, т.к. в таблице `progress` такого столбца нет
`select *`
`from subject, progress`
`where id = progress.id_subject;`

-- Пример 14

-- Чтобы быстрее писать запросы, удобнее использовать синонимы для таблиц
-- Запись `subject s` в разделе `from` означает, что "`s`" следует интерпретировать как "`subject`". Аналогично с `progress p`

```
select p.id_student, p.rating, s.name as subject_name
from subject s, progress p
where s.id = p.id_subject;
```

id_student	rating	name
1	4	Технологии программирования
2	5	Технологии программирования
4	5	История
2	4	История
3	3	История

Кроме предложения where в запросе могут встречаться и другие, например, order by и limit.

Предложение order by

Предложение order by используется для сортировки строк в указанном порядке. В нем указываются столбцы (в порядке их приоритета при сортировке) и рядом с каждым из них – порядок сортировки (asc – по возрастанию, desc – по убыванию; если порядок не указан, то по умолчанию – asc). Без указания предложения order by порядок вывода строк считается неопределенным.

-- Пример 15

-- Получить все строки из progress, отсортировать по rating по убыванию

```
select * from progress order by rating desc;
```

id_student	id_subject	rating
2	1	5
4	2	5
1	1	4
2	2	4
3	2	3

-- Пример 16

-- Получить все строки из progress, отсортировать по rating по убыванию; если у двух строк одинаковые значения rating, то между собой они будут сортироваться по id_subject по возрастанию.

```
select * from progress order by rating desc, id_student asc;
```

id_student	id_subject	rating
2	1	5
4	2	5
1	1	4
2	2	4
3	2	3

Предложение limit

Предложение `limit` ограничивает максимальное количество строк, которое будет возвращено одним запросом, а так же имеет вложенное предложение `offset`, показывающее, сколько строк нужно пропустить, прежде чем начать выдавать строки.

-- Пример 17

-- Получить не больше 2 строк из `progress`, сортировка по `rating` по возрастанию

```
select * from progress order by rating limit 3;
```

id_student	id_subject	rating
3	2	3
1	1	4
2	2	4

-- Пример 18

-- Получить не более 2 строк из `progress`, сортировка по `id_student` по возрастанию, пропустить одну строку перед выводом результата

-- Важно, что в списке столбцов для `select` совсем не обязательно должны быть те столбцы, по которым ведутся фильтр, сортировка и т.д.

```
select id_subject, rating from progress order by id_student limit 2 offset 1;
```

id_subject	rating
1	5
2	4

Результаты нескольких `select`-запросов, если они содержат одинаковое количество столбцов с совместимыми типами, можно различным образом комбинировать, используя предложения `union`, `intersect`, `except`.

Выполнить `union`, `except`, `intersect` можно с любыми запросами при совместимости типов. Важно помнить, что без `order by` порядок вывода не определён.

Важно отметить, что во всех запросах, кроме самого первого, нельзя указывать предложения `order by` и `limit`. Для первого они указываются после все запросов, с которыми делается `union`, `intersect` или `except`.

Предложение `union`

Предложение `union` объединяет результаты нескольких запросов в одну выборку.

-- Пример 19

-- Вернет две строки

```
select 'Первый', 1 union select 'Второй', 2;
```

?column?	?column?
----------	----------

Второй	2
Первый	1

-- Пример 20

-- Вернет одну строку, так как при union убираются дубликаты; если нужно сохранить дубликаты, нужно использовать union all

```
select 'Первый', 1 union select 'Первый', 1;
```

?column?	?column?
Первый	1

Предложение instersect

Предложение instersect создает пересечение результатов запросов.

-- Пример 21

-- Вернет пустое множество

```
select 'Первый', 1 intersect select 'Второй', 2;
```

-- Пример 22

-- Вернет одну строку

```
select 'Первый', 1 intersect select 'Первый', 1;
```

?column?	?column?
Первый	1

Предложение except

Предложение except включает в результат только те строки, которые есть в выборке из первого запроса, но отсутствуют в выборке из второго.

-- Пример 23

-- Вернет одну строку ("Первый", 1)

```
select 'Первый', 1 except select 'Второй', 2;
```

?column?	?column?
Первый	1

-- Пример 24

-- Вернет пустое множество

```
select 'Первый', 1 except select 'Первый', 1;
```

Предложение join

Одна из операций реляционной алгебры – соединение таблиц. В результате этой операции из двух таблиц получается одна, каждая строка которой содержит столбцы обеих таблиц и некоторые значения, зависящие от способа и условий соединения. Для выполнения этой операции служит предложение join в команде select.

В этом разделе рассмотрены базовые примеры применения предложения join различных видов. *Подробнее про различные виды соединений можно прочитать в документации к команде select.*

Внутреннее соединение (inner join). Соединяет строки двух различных таблиц, отвечающие заданному условию соединения. Результат получается такой же, как и при выполнении простого запроса select из двух таблиц с условием в where. В PostgreSQL это соединение существует исключительно для удобства записи. Порядок таблиц в условии соединения не важен. Пример показан ниже.

-- Пример 25

```
select s.name, s.surname, g.name as study_group
from student s, "group" g
where s.id_group = g.id;
```

-- Результат такой же, как выше. Вместо inner join можно писать просто join

```
select s.name, s.surname, g.name as study_group
from student s inner join "group" g on s.id_group = g.id;
```

name	surname	study_group
Василий	Олоев	ИВТб-3301-01-00
Николай	Котов	ИВТб-3301-01-00
Олег	Зотов	ИВТб-3302-02-00
Константин	Романов	ИВТб-4301-01-00

Перекрёстное соединение (cross join). Соединяет две таблицы, возвращая декартово произведение их строк. Не принимает условие для фильтра. Результат получается такой же, как и при выполнении запроса select из двух таблиц без условия в where, или при выполнении соединения inner join on true. В PostgreSQL это соединение существует исключительно для удобства записи. Пример показан ниже.

-- Пример 26

```
select s.name, s.surname, g.name as study_group
from student s, "group" g;
```

-- Результат такой же, как выше.

```
select s.name, s.surname, g.name as study_group
from student s cross join "group" g;
```

name	surname	study_group
Николай	Котов	ИВТб-3301-01-00
Василий	Олоев	ИВТб-3301-01-00
Олег	Зотов	ИВТб-3301-01-00
Константин	Романов	ИВТб-3301-01-00
Николай	Котов	ИВТб-3302-02-00
Василий	Олоев	ИВТб-3302-02-00
Олег	Зотов	ИВТб-3302-02-00
Константин	Романов	ИВТб-3302-02-00
Николай	Котов	ИВТб-4301-01-00

Василий	Олоев	ИВТб-4301-01-00
Олег	Зотов	ИВТб-4301-01-00
Константин	Романов	ИВТб-4301-01-00

Левое внешнее соединение (left outer join). Работает так же, как внутреннее соединение, но добавляет в результат все строки из таблицы слева, для которых не нашлись строки таблицы справа, удовлетворяющие условию. Для таких строк в столбцах из правой таблицы будет стоять null. Порядок таблиц в условии важен. Пример показан ниже.

-- Пример 27

```
select s.name, s.surname, g.name as study_group
from student s, "group" g
where s.id_group = g.id;
```

name	surname	study_group
Василий	Олоев	ИВТб-3301-01-00
Николай	Котов	ИВТб-3301-01-00
Олег	Зотов	ИВТб-3302-02-00
Константин	Романов	ИВТб-4301-01-00

-- Предположив, что в таблице student есть запись с id group = null, результат по сравнению с запросом выше будет включать и записи с id_group = null.

```
select s.name, s.surname, g.name as study_group
from student s left join "group" g on s.id_group = g.id;
```

name	surname	study_group
Василий	Олоев	ИВТб-3301-01-00
Николай	Котов	ИВТб-3301-01-00
Олег	Зотов	ИВТб-3302-02-00
Константин	Романов	ИВТб-4301-01-00
Елизавета	Никулина	null

Правое внешнее соединение (right outer join). Работает так же, как левое внешнее соединение, но вместо строк из таблицы слева, для которых не нашлось пары, добавляет строки из таблицы справа. Аналогично, для таких строк в столбцах из левой таблицы будет стоять null. Порядок таблиц в условии важен. Любой right outer join можно свести к left outer join, поменяв левую и правую таблицы в условии местами.

Полное внешнее соединение (full outer join). Работает так же, как внутреннее соединение, но добавляет в результат все строки из таблицы слева, для которых не нашлось пары, и все строки из таблицы справа, для которых не нашлось пары. Отсутствующие значения заполняются null. Порядок таблиц в условии соединения не важен.

Представления

Представление (view) — это виртуальная таблица, содержимое которой определяется запросом `select`. Запрос будет выполняться при каждом обращении к представлению. Тем не менее, представления очень удобны для работы с большими и часто повторяющимися запросами.

Как правило, при разработке приложений, использующих СУБД, делаются представления для многих таблиц, которые имеют внешние ключи.

Более того, в некоторых случаях устанавливается за правило – в клиентском приложении выполнять чтение только из представлений. Такое правило не очень обоснованно, т. к. представления иногда создают препятствия для оптимизатора и чтение из них может работать гораздо дольше, чем напрямую из таблиц. Но такие случаи редки и относятся к большим сложным запросам. Если опыт показывает, что использование представления не замедляет работу приложения, лучше всегда использовать представление, чем многократно писать запрос.

Так же, как и любые объекты БД, представления лучше размещать в схемах.

В имени представления желательно использовать суффикс «`_v`».

Подробнее о представлениях в PostgreSQL можно прочитать в документации.

Ниже показаны примеры работы с представлениями.

-- Пример 28

-- Для создания представления используется ключевое слово `view`.

`Create or replace` означает, что если представление с данным именем уже существует, то оно будет заменено.

-- В данном примере создается представление для таблицы студентов, содержащее их ФИО, курс обучения и название учебной группы.

```
create or replace view students_groups_v as
select s.name, s.middle_name, s.surname, s.course, g.name as group_name
from student s, "group" g
where s.id_group = g.id;
```

name	middle_name	surname	course	group
Василий	Петрович	Олоев	3	ИВТб-3301-01-00
Николай	Сергеевич	Котов	3	ИВТб-3301-01-00
Олег	Геогриевич	Зотов	3	ИВТб-3302-02-00
Константин	Владимирович	Романов	4	ИВТб-4301-01-00

-- Пример 29

-- В данном примере создается представление, отражающее успеваемость студентов. Для соединения строк таблиц `student`, `subject`, `progress` используется внутреннее соединение `join`.

```
create or replace view students_progress_subject_v as
select s.name, s.middle_name, s.surname, s.course, su.name as subject,
p.rating
from student s join progress p on s.id = p.id_student
join subject su on p.id_subject = su.id;
```

name	middle_name	surname	course	subject	rating
Николай	Сергеевич	Котов	3	Технологии программирования	4
Василий	Петрович	Олоев	3	Технологии программирования	5
Константин	Владимирович	Романов	4	История	5

Василий	Петрович	Олоев	3	История	4
Олег	Геогриевич	Зотов	3	История	3

-- Пример 30

-- В данном примере создается представление, отражающее занятость преподавателей.

```
create or replace view teachers_subjects_v as
select t.name, t.middle_name, t.surname, t.position, s.name as subject,
too.occupation_type
from teacher t, subject s, type_of_occupation too
where t.id = too.id_teacher and s.id = too.id_subject;
```

name	middle_name	surname	position	subject	occupation_type
Семен	Петрович	Катаев	instructor	Технологии программирования	laboratory work
Ирина	Олеговна	Коснырева	docent	Технологии программирования	lecture
Дмитрий	Алексеевич	Мусанов	professor	История	lecture

- III. Для любой таблицы, содержащей столбец с числовыми данными, создать представление следующего вида, отражающее информацию по этому столбцу (в представлении ровно 3 столбца и 4 строки):

Минимальное значение	<значение>	<id записи с минимальным значением>
Максимальное значение	<значение>	<id записи с максимальным значением>
Среднее значение	<значение>	null
Сумма значений	<значение>	null

Агрегатные функции

Часто при выполнении запросов нужно получить результат выполнения какой-либо функции над множеством значений (сумма, среднее значение, подсчет количества, объединение в массив или строку (text) и т.д.). Для этого используются агрегатные функции.

В PostgreSQL существует большое количество различных агрегатных функций. *Подробнее про них написано в документации.*

Если необходимо получить результат выполнения агрегатной функции над разными группами строк из выборки, необходимо использовать предложение group by. В нем перечисляются все столбцы, которые перечисляются в select, но не участвуют в какой-либо агрегатной функции. На основе совпадения значений в указанных столбцах строки объединяются в группы.

В таблице 1 перечислены некоторые агрегатные функции общего назначения.

Таблица 1 – Некоторые агрегатные функции общего назначения

Функция	Описание
<code>avg(выражение)</code>	арифметическое среднее для всех входных значений
<code>max(выражение)</code>	максимальное значение <i>выражения</i> среди всех входных данных
<code>min(выражение)</code>	минимальное значение <i>выражения</i> среди всех входных данных
<code>string_agg(выражение, разделитель)</code>	входные данные складываются в строку через заданный разделитель
<code>sum(выражение)</code>	сумма значений <i>выражения</i> по всем входным данным
<code>count(*)</code>	количество входных строк
<code>count(выражение)</code>	количество входных строк, для которых значение <i>выражения</i> не равно NULL

-- Пример 31

-- Два запроса ниже эквиваленты и возвращают количество записей в *student*

```
select count(*) from student;
select count(1) from student;
```

count
5

-- Пример 32

-- Возвращает строку с четырьмя значениями: сумма по столбцу *number_of_hours*, минимальное, максимальное и среднее значения в *number_of_hours*

```
select sum(number_of_hours), min(number_of_hours), max(number_of_hours),
avg(number_of_hours) from subject;
```

sum	min	max	avg
54	18	36	27

-- Пример 33

-- Этот запрос выбирает всех студентов и сданные ими предметы, отображает:

- *subjects* - список предметов (имена, разделенные запятой)
- *average_rating* - средний балл
- Записи отсортированы по возрастанию по среднему баллу
- Важно отметить, что все столбцы *st.id*, *st.course* нужно указать в *group by*, чтобы выполнить группировку, поскольку они указаны в *select*

```
select st.id, st.course,
```

```
string_agg(sb.name, ', ') as subjects, avg(p.rating) as average_rating
from student st, subject sb, progress p
where p.id_subject = sb.id and p.id_student = st.id
group by st.id, st.course order by average_rating asc;
```

id	course	subjects	average_rating
3	3	История	3
1	3	Технологии программирования	4
2	3	Технологии программирования, История	4,5
4	4	История	5

Предложение *having*

Иногда возникает потребность сделать выборку, в которой строки отвечают определённому условию, вычисляемому с помощью агрегатных функций. Такие условия следует записывать не в предложение *where*, а в предложение *having*. Оно отличается от *where*: *where* фильтрует отдельные строки до применения *group by*, а *having* фильтрует строки групп, созданных предложением *group by*.

-- Пример 34

-- Этот запрос вернет записи, у которых средний балл больше 4

```
select st.id, st.course,
string_agg(sb.name, ', ') as subjects, avg(p.rating) as average_rating
from student st, subject sb, progress p
where p.id_subject = sb.id and p.id_student = st.id
group by st.id, st.course having avg(p.rating) > 4
order by average_rating asc;
```

id	course	subjects	average_rating
2	3	Технологии программирования, История	4,5
4	4	История	5

Ниже представлен скрипт, который получился в ходе выполнения данной лабораторной работы.

--Заполнение таблиц

```
do $$
declare
    groupID bigint;
    subjectID bigint;
    teacherID bigint;
    student1 bigint;
    student2 bigint;
    student3 bigint;
    student4 bigint;
begin

insert into "group"(name, number_of_students)
values ('ИВТ6-3301-01-00', 15)
returning id into groupID;
insert into student(name, surname, middle_name, id_group, address, course)
values ('Николай', 'Котов', 'Сергеевич', groupID, 'г. Киров ул. Ломоносова
16а', 3)
```



```

returning id into student1;
insert into student(name, surname, middle_name, id_group, address, source)
values ('Василий', 'Олоев', 'Петрович', groupID, 'г. Киров ул. Свободы 4', 3)
returning id into student2;

insert into "group"(name, number_of_students)
values ('ИВТ6-3302-02-00', 20)
returning id into groupID;
insert into student(name, surname, middle_name, id_group, address, source)
values ('Олег', 'Зотов', 'Георгиевич', groupID, 'г. Киров ул. Воровского
122', 3)
returning id into student3;

insert into "group"(name, number_of_students)
values ('ИВТ6-4301-01-00', 12)
returning id into groupID;
insert into student(name, surname, middle_name, id_group, address, source)
values ('Константин', 'Романов', 'Владимирович', groupID, 'г. Киров ул.
Ленина 52', 4)
returning id into student4;

insert into subject(name, number_of_hours)
values ('Технологии программирования', 18)
returning id into subjectID;
insert into teacher(name, surname, middle_name, position, phone_number)
values ('Семен', 'Катаев', 'Петрович', 'instructor', '89530000000')
returning id into teacherID;
insert into type_of_occupation(id_subject, id_teacher, occupation_type)
values (subjectID, teacherID, 'laboratory work');
insert into progress(id_subject, id_student, rating)
values (subjectID, student1, 4);
insert into progress(id_subject, id_student, rating)
values (subjectID, student2, 5);

insert into teacher(name, surname, middle_name, position, phone_number)
values ('Ирина', 'Коснырева', 'Олеговна', 'docent', '89530000002')
returning id into teacherID;
insert into type_of_occupation(id_subject, id_teacher, occupation_type)
values (subjectID, teacherID, 'lecture');

insert into subject(name, number_of_hours)
values ('История', 36)
returning id into subjectID;
insert into teacher(name, surname, middle_name, position, phone_number)
values ('Дмитрий', 'Мусанов', 'Алексеевич', 'professor', '89530000001')
returning id into teacherID;
insert into type_of_occupation(id_subject, id_teacher, occupation_type)
values (subjectID, teacherID, 'lecture');
insert into progress(id_subject, id_student, rating)
values (subjectID, student4, 5);
insert into progress(id_subject, id_student, rating)
values (subjectID, student2, 4);
insert into progress(id_subject, id_student, rating)
values (subjectID, student3, 3);
end;
$$ language plpgsql;

--Создание представлений
create or replace view students_groups_v as
select s.name, s.middle_name, s.surname, s.source, g.name as group_name
from student s, "group" g
where s.id_group = g.id;

create or replace view students_progress_subject_v as

```

```

select s.name, s.middle_name, s.surname, s.course, su.name as subject,
p.rating
from student s join progress p on s.id = p.id_student join subject su on
p.id_subject = su.id;

create or replace view teachers_subjects_v as
select t.name, t.middle_name, t.surname, t.position, s.name as subject,
too.occupation_type
from teacher t, subject s, type_of_occupation too
where t.id = too.id_teacher and s.id = too.id_subject;

--Создание представления с числовыми данными
create or replace view subjects_v as
(select 'Минимальное значение', number_of_hours, id from subject order by
number_of_hours asc limit 1)
union all
(select 'Максимальное значение', number_of_hours, id from subject order by
number_of_hours desc limit 1)
union all
select 'Среднее значение', avg(s.number_of_hours), null from subject s
union all
select 'Сумма значений', sum(s.number_of_hours), null from subject s;

```

Далее представлено несколько дополнительных команд, которые могут быть использованы при выполнении данной лабораторной работы.

Команда update

Команда update используется для изменения существующих строк таблицы. Она позволяет поменять значения в столбцах некоторых строк, удовлетворяющих определённому условию. Столбцы, не включенные в команду, сохраняют свои предыдущие значения. Для выбора тех строк, в которых будет произведено изменение, используется предложение where, как в команде select.

Подробнее про синтаксис команды update можно прочитать в документации.

```

-- Пример 35
-- Обновляет в таблице student строку с id = 1, устанавливая столбцу id_group
значение 2.
-- Остальные столбцы не изменяют своего значения.
update student set id_group = 2 where id = 1;

```

```

-- Пример 36
-- В одной команде можно обновить несколько полей.
-- Обновляет в таблице student строку с id = 1, устанавливая столбцу id_group
значение 2, столбцу address = 'г. Киров ул. Преображенская 34'.
-- Остальные столбцы не изменяют своего значения.
update student set id_group = 2, address = 'г. Киров ул. Преображенская 34'
where id = 1;

```

```

-- Пример 37
-- Обновляет все строки в subject, увеличивая значение столбца
number_of_hours на 10.
update subject set number_of_hours = number_of_hours + 10;

```

-- Пример 38

-- Команда ниже не выполнится, возникнет ошибка, поскольку у таблицы group нет записи с id = 8
update student **set** id_group = 8 **where** id = 1;

Команда delete

Для удаления строк из таблицы используется команда delete. Она позволяет удалить некоторые строки, удовлетворяющие определенному условию. Для выбора тех строк, которые будут удалены, используется предложение where, как в команде select. *Подробнее про синтаксис команды delete можно прочитать в документации.*

-- Пример 39

-- Удалит из таблицы progress строки, в которых rating != 0
delete from progress **where** rating != 0;

-- Пример 40

-- Команда не выполнится, возникнет ошибка, т.к. на subject с id = 1 есть внешние ссылки (определено ограничением foreign key в progress и type_of_occupation)
delete from subject **where** id = 1;

-- Пример 41

-- Команда должна удалить все записи из таблицы subject, но не выполнится по той же причине, что и в предыдущем примере
delete from subject;

Операторы и встроенные функции, работа с датами

Во многих примерах ранее использовались арифметические и логические операторы, а также различные встроенные функции (например, now(), возвращающая дату начала текущей транзакции).

PostgreSQL предоставляет большое количество операторов и встроенных функций. *Подробнее можно прочитать в документации.*

Поскольку их количество очень велико, а написание совпадает во многих случаях с общепринятым в языках программирования, здесь не будет приводиться списков операторов и встроенных функций. Вместо этого ниже показаны примеры различных часто встречаемых или неочевидных ситуаций, которые могут связанных с их использованием.

-- Пример 42

-- Есть три логических оператора: and, or, not. Приоритет стандартный
select not 1 < 2 **and** 3 < 4 **or** 5 = 5; -- true

-- Пример 43

```

-- Оператор "не равно" можно записывать как в виде <>, так и !=
select 1 <> 1 or 5 != 5; -- false

-- Пример 44
-- Нельзя сравнивать значение с null с помощью = или <>. Нужно использовать
-- предикаты is null, is not null.
select 1 is null or 5 is not null; -- true

-- Пример 45
-- Для проверки вхождения в промежуток есть предикаты between, now between
select 3 between 3 and 5; -- true
select 1 not between -1 and 5; -- false

-- Пример 46
-- Для конкатенации строк используется оператор || или функция concat
select 'Hello, ' || 'Postgre!'; -- Hello, Postgre!
select concat('Hello, ', 'Postgre!'); -- Hello, Postgre!

-- Пример 47
-- Разница между ними в том, что строка || null = null, а функция concat
-- воспринимает null, как пустую строку
select 'abc' || null; -- null
select concat('abc', null, 'de'); -- abcde

-- Пример 48
-- Чтобы вставить перенос строки, можно сделать так. Функция chr возвращает
-- символ по его коду в кодировке БД
select ' first' || chr(13) || 'second'; -- first \n second

-- Пример 49
-- Но если нужно поставить одинарную кавычку в строке, можно поставить
-- одинарных кавычки подряд
select 'Don''t worry'; -- Don't worry

-- Пример 50
-- Для приведения в верхний или нижний регистр используются функции lower,
-- upper
select lower('AbC'); -- abc
select upper('AbC'); -- ABC

-- Пример 51
-- Функция trim обрезает пробельные символы по бокам строки
select trim('  a b c '); -- "a b c"

-- Пример 52
-- Конструкцию case можно использовать по аналогии со switch или для выборки
-- разных значений по условию
select case when 1 = 1 then 'AAA' else 'OOO' end; -- AAA

-- Пример 53
-- Полезна функция coalesce(a, b, c, ...), которая вернет первый аргумент,
-- отличный от null
select coalesce(null, null, 42, null, 33); -- 42

-- Пример 54
-- Строки возможно проверять по шаблону с помощью оператора like. В шаблоне %
-- означает любое количество символов, _ - один символ
select 'abcdef' like '%cdef'; -- true
select 'abcdef' like 'a_cdef'; -- true
select 'abcdef' like '%c%'; -- true
select 'abcdef' like 'ab_'; -- false

-- Пример 55

```

```
-- Часто преобразование типов происходит неявно
select 42||'...'; -- "42..."
```

-- Пример 56

```
-- Но можно делать его вручную с помощью оператора ::
select 42::text; -- "42"
```

-- Пример 57

```
-- Возвращает timestamp на начало текущей транзакции
select now();
```

В двух последних примерах было рассмотрено преобразование типов. Часто оно используется при написании дат в запросах (преобразовать строку в дату). В этом случае принимается практически любой стандартный формат даты. Тем не менее, он во многом определяется местоположением клиента, и, чтобы избежать неожиданных эффектов, лучше использовать функции `to_date`, `to_timestamp`, принимающих вторым аргументом формат даты. Примеры показаны ниже.

-- Пример 58

```
-- Два запроса ниже аналогичны, если формат даты на клиенте совпадает с указанным
```

```
select '12-05-2019'::date;
select to_date('12-05-2019', 'DD-MM-YYYY');
```

-- Пример 59

```
-- Два запроса ниже аналогичны, если формат даты на клиенте совпадает с указанным
```

```
select '12-05-2019 12:08'::timestamp;
select to_timestamp('12-05-2019 12:08', 'DD-MM-YYYY HH:MI');
```

Если суммировать `date` и число, к дате прибавится указанное количество дней. Суммировать `timestamp` и число нельзя. Для более очевидного прибавления временных промежутков лучше суммирование не с числами, а с временными интервалами. Важно помнить, что при сложении `date` и временного интервала получается `timestamp`. Примеры показаны ниже.

-- Пример 60

```
-- Прибавляет один день к дате
```

```
select to_date('12-05-2019', 'DD-MM-YYYY') + 1; -- 13-05-2019
```

-- Пример 61

```
-- Складывать timestamp и число нельзя
```

```
select to_timestamp('12-05-2019 12:08', 'DD-MM-YYYY HH:MI') + 1; -- Ошибка
```

-- Пример 62

```
-- К датам лучше прибавлять временные интервалы
```

```
select to_date('12-05-2019', 'DD-MM-YYYY') + interval '1 day'; -- 13-05-2019 00:00
```

```
select to_date('12-05-2019', 'DD-MM-YYYY') - interval '5 year'; -- 12-05-2014 00:00
```

```
select to_timestamp('12-05-2019 12:08', 'DD-MM-YYYY HH:MI') + interval '1 year 2 month 11 second'; -- 12-07-2020 00:08:11
```

-- Пример 63

```
-- Даты также можно вычитать, результат - количество дней
select to_date('12-05-2019', 'DD-MM-YYYY') - to_date('10-05-2019', 'DD-MM-YYYY'); -- 2
```

-- Пример 64

-- Если вычесть date из timestamp, timestamp из date или timestamp из timestamp, получится временной интервал

```
select to_timestamp('12-05-2019 12:08', 'DD-MM-YYYY HH:MI') -
       to_date('10-05-2019', 'DD-MM-YYYY');
```

```
-- 0 years 0 mons 2 days 0 hours 8 mins 0.00 secs
```

Подзапросы и предложение with

Если запрос что-либо возвращает, к результатам его работы можно обратиться в другом запросе. Например, в разделе from помимо таблиц можно указывать подзапросы (при этом обязательно давать им имена).

Примеры использования подзапросов показаны ниже.

-- Пример 65

-- Бесполезный пример

```
select * from (select * from student) s;
```

id	name	surname	middle_name	id_group	address	course
1	Николай	Котов	Сергеевич	4	г. Киров ул. Ломоносова 16а	3
2	Василий	Олоев	Петрович	4	г. Киров ул. Свободы 4	3
3	Олег	Зотов	Георгиевич	5	г. Киров ул. Воровского 122	3
4	Константин	Романов	Владимирович	6	г. Киров ул. Ленина 52	4

-- Пример 66

-- Запрос может обращаться к полям вложенного запроса не глубже, чем на один уровень

-- Вложенные запросы одного уровня не могут обращаться друг к другу

```
select s.* from ( -- select s.* здесь сделать нельзя
select ss.* from (select * from student) ss -- Дописать на этом же уровне еще
один вложенный запрос, который обращается к ss, нельзя
) s;
```

-- Пример 67

-- Если возвращается одна строка, то подзапросы можно использовать так (с оператором =)

```
delete from student where id = (select id_student from progress where rating
= '5');
```

-- Пример 68

-- Если значений несколько, можно использовать оператор in

```
select name, surname from student where id in (select id_student from
progress where rating = '4');
```

name	surname
Николай	Котов
Василий	Олоев

Использование подзапросов очень полезно и многие задачи нельзя решить без их применения. Тем не менее, с этим связаны некоторые проблем:

- область видимости результатов подзапроса ограничена;
- если для выполнения всего запроса необходимо СУБД обратиться много раз к результатам подзапроса, он может выполняться каждый раз.

Указанные выше проблемы решаются с помощью предложения `with`. Оно может размещаться перед командами `select`, `insert`, `update` и `delete`. В нем можно расположить несколько именованных подзапросов и далее использовать их результаты. Такие подзапросы называют *общими табличными выражениями* (Common Table Expressions, CTE).

Каждое CTE может обращаться к результатам работы вышестоящих CTE в предложении `with`. Создаваемая в этом случае выборка вычисляется только один раз и сохраняется во временную таблицу, существующую на время выполнения всего запроса. Кроме того, запросы с предложениями `with` гораздо проще читать и понимать.

Подробнее о предложении `with` и CTE можно прочитать в документации.

Ниже показаны примеры использования предложения `with`.

-- Пример 69

--Создание таблицы `good_students`

```
create table public.good_students(  
    id bigserial primary key,  
    name varchar(30) not null,  
    surname varchar(30) not null,  
    middle_name varchar(30),  
    id_group bigint not null references public.group(id),  
    address varchar(100),  
    course int check (course >= 1 and course <=5)  
);
```

-- Этот запрос перенесет записи из таблицы `student` в таблицу `good_students` данные студентов с оценками > 4

```
with items_to_insert as (select id_student from progress where rating < 4)  
insert into good_students (select id, name, surname, middle_name, id_group,  
address, course from student where id in (select * from items_to_insert));
```

Предложение `returning`

Полезным часто бывает то, что команды `insert`, `update` и `delete` могут возвращать строки. Для этого используется предложение `returning`, которое ставится в конце запроса и возвращает строки, которые были затронуты

выполнением команды. Для update возвращаются уже измененные значения строк.

Результат, получаемый с помощью returning, нельзя использовать как подзапрос, но можно использовать в предложении with. С помощью этого, например, просто сохранять ссылочную целостность при заполнении таблиц данными: в предложении with можно вставлять строки, на которые будут ссылаться вставляемые в основной части запроса.

Ниже показаны примеры использования предложения returning.

-- Пример 70

-- Этот запрос обновит таблицу subject и вернет все обновленные строки
update subject **set** number_of_hours = number_of_hours - 2 **where**
number_of_hours >= 30 **returning** *;

id	name	count_of_hours
2	История	34

IV. Создать и выполнить SQL-скрипты для выполнения различных планов запросов.

Часто необходимо проверить производительность только что написанного запроса в PostgreSQL в поисках способа улучшить его производительность. Для этого вам нужно получить отчет о выполнении запроса, который называется планом выполнения. План выполнения запроса дает суммарную информацию о выполнении запроса с подробным отчетом о времени, потраченном на каждом шаге, и затратах на его выполнение.

Структура плана выполнения запроса

Структура плана запроса представляет собой дерево, состоящее из так называемых узлов плана (plan nodes).

Узлы на нижних уровнях дерева отвечают за просмотр и выдачу строк таблиц, которые осуществляются с помощью методов доступа, описанных выше. Если конкретный запрос требует выполнения операций агрегирования, соединения таблиц, сортировки, то над узлами выборки строк будут располагаться дополнительные узлы дерева плана.

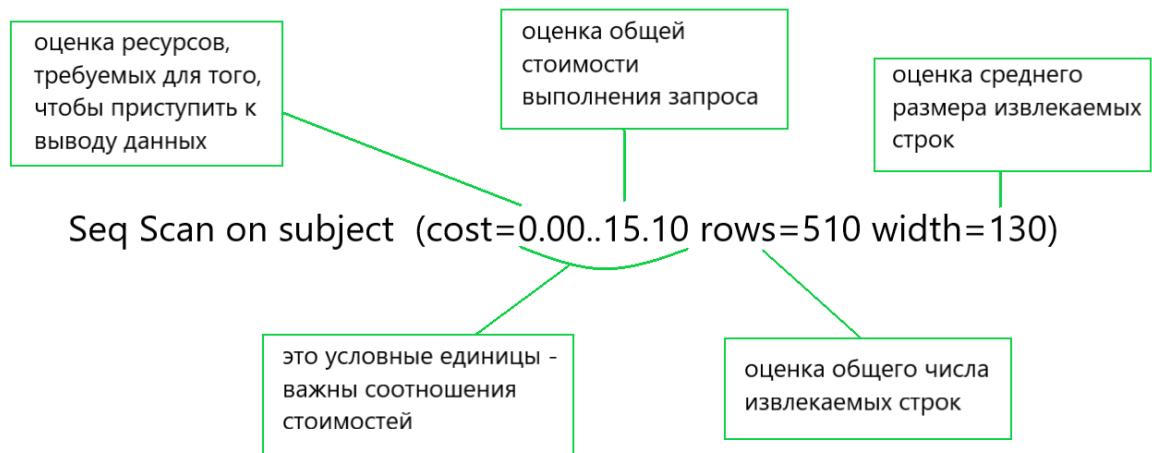
Например, для соединения наборов строк будут использоваться способы, которые мы только что рассмотрели.

Для каждого узла дерева плана команда EXPLAIN выводит по одной строке, при этом выводятся также оценки стоимости выполнения операций на каждом узле, которые делает планировщик. В случае необходимости для конкретных

узлов могут выводиться дополнительные строки. Самая первая строка плана содержит общую оценку стоимости выполнения данного запроса.

--Пример 71

```
EXPLAIN select * from subject;
```



Поскольку в этом запросе нет предложения WHERE, он должен просмотреть все строки таблицы, поэтому планировщик выбирает последовательный просмотр (sequential scan).

В скобках приведены важные параметры плана.

Первая оценка равна нулю, поскольку никакие дополнительные операции с выбранными строками не предполагаются, и PostgreSQL может сразу же выводить прочитанные строки. Формируя эту оценку, планировщик исходит из предположения, что данный узел плана запроса выполняется до конца, т. е. извлекаются все имеющиеся строки таблицы. Исключения: в запросе SELECT предложение LIMIT. Обе оценки вычисляются на основе ряда параметров сервера баз данных.

Для каждого запроса планировщик формирует несколько планов.

При сравнении различных вариантов плана, как правило, для выполнения выбирается тот, который имеет наименьшую общую стоимость выполнения запроса.

Оценки числа строк и их размера планировщик получает на основе статистики, накапливаемой в специальных системных таблицах.

EXPLAIN

Сгенерировать план выполнения запроса позволяет ключевое слово EXPLAIN в PostgreSQL. Синтаксис создания плана в PostgreSQL имеет вид:

```
EXPLAIN [ ( параметр [, ...] ) ] оператор  
EXPLAIN [ ANALYZE ] [ VERBOSE ] оператор
```

Здесь допускается параметр:

- ANALYZE [boolean]
- VERBOSE [boolean]
- COSTS [boolean]
- BUFFERS [boolean]
- TIMING [boolean]
- FORMAT { TEXT | XML | JSON | YAML }

Параметры:

ANALYZE

Выполнить команду и вывести фактическое время выполнения и другую статистику. По умолчанию этот параметр равен FALSE.

--Пример 72

```
EXPLAIN (analyze) select * from student;
```

QUERY PLAN	
text	
1	Seq Scan on student (cost=0.00..11.60 rows=160 width=472) (actual time=0.032..0.034 rows=6 loops=1)
2	Planning Time: 0.251 ms
3	Execution Time: 0.067 ms

VERBOSE

Вывести дополнительную информацию о плане запроса. В частности, включить список столбцов результата для каждого узла в дереве плана, дополнить схемой имена таблиц и функций, всегда указывать для переменных в выражениях псевдоним их таблицы, а также выводить имена всех триггеров, для которых выдаётся статистика. По умолчанию этот параметр равен FALSE.

--Пример 73

```
EXPLAIN (verbose) select * from student;
```

QUERY PLAN	
text	
1	Seq Scan on public.student (cost=0.00..11.60 rows=160 width=472)
2	[...] Output: id, first_name, second_name, middle_name, id_group, address, course

COSTS

Вывести рассчитанную стоимость запуска и общую стоимость каждого узла плана, а также рассчитанное число строк и ширину каждой строки. Этот параметр по умолчанию равен TRUE.

--Пример 74

```
EXPLAIN (costs) select * from student;
```

QUERY PLAN	
	text
1	Seq Scan on student (cost=0.00..11.60 rows=160 width=472)

BUFFERS

Включить информацию об использовании буфера. В частности, вывести число попаданий, блоков прочитанных, загрязненных и записанных в разделяемом и локальном буфере, а также число прочитанных и записанных временных блоков. **Попаданием** (hit) считается ситуация, когда требуемый блок уже находится в кеше и чтения с диска удаётся избежать. Блоки в общем буфере содержат данные обычных таблиц и индексов, в локальном — данные временных таблиц и индексов, а временные блоки предназначены для краткосрочного использования при выполнении сортировки, хеширования, материализации и подобных узлов плана. Число **загрязнённых** блоков (dirty) показывает, сколько ранее не модифицированных блоков изменила данная операция; тогда как число **записанных** блоков (written) показывает, сколько ранее загрязнённых блоков данный серверный процесс вынес из кеша при обработке запроса. Значения, указываемые для узла верхнего уровня, включают значения всех его дочерних узлов. В текстовом формате выводятся только ненулевые значения. Этот параметр действует только в режиме ANALYZE. По умолчанию его значение равно FALSE.

--Пример 75

```
EXPLAIN (analyze,buffers) select * from student;
```

QUERY PLAN	
	text
1	Seq Scan on student (cost=0.00..11.60 rows=160 width=472) (actual time=2.137..2.140 rows=6 loops=1)
2	[...] Buffers: shared hit=1
3	Planning Time: 0.686 ms
4	Execution Time: 2.217 ms

TIMING

Включить в вывод фактическое время запуска и время, затраченное на каждый узел. Постоянное чтение системных часов может значительно замедлить запрос, так что если достаточно знать фактическое число строк, имеет смысл сделать этот параметр равным FALSE. Время выполнения всего оператора замеряется всегда, даже когда этот параметр выключен и на уровне узлов время не подсчитывается. Этот параметр действует только в режиме ANALYZE. По умолчанию его значение равно TRUE.

--Пример 76

```
EXPLAIN (analyze,timing) select * from student;
```

QUERY PLAN	
	text
1	Seq Scan on student (cost=0.00..11.60 rows=160 width=472) (actual time=0.013..0.015 rows=6 loops=1)
2	Planning Time: 0.066 ms
3	Execution Time: 0.026 ms

FORMAT

Установить один из следующих форматов вывода: TEXT, XML, JSON или YAML. Последние три формата содержат ту же информацию, что и текстовый, но больше подходят для программного разбора. По умолчанию выбирается формат TEXT(выше приведены примеры использования формата TEXT). В дальнейшем все примеры будут рассмотрены в JSON формате представления запросов.

--Пример 77

--Присоединение таблицы с id учителя к id предмета

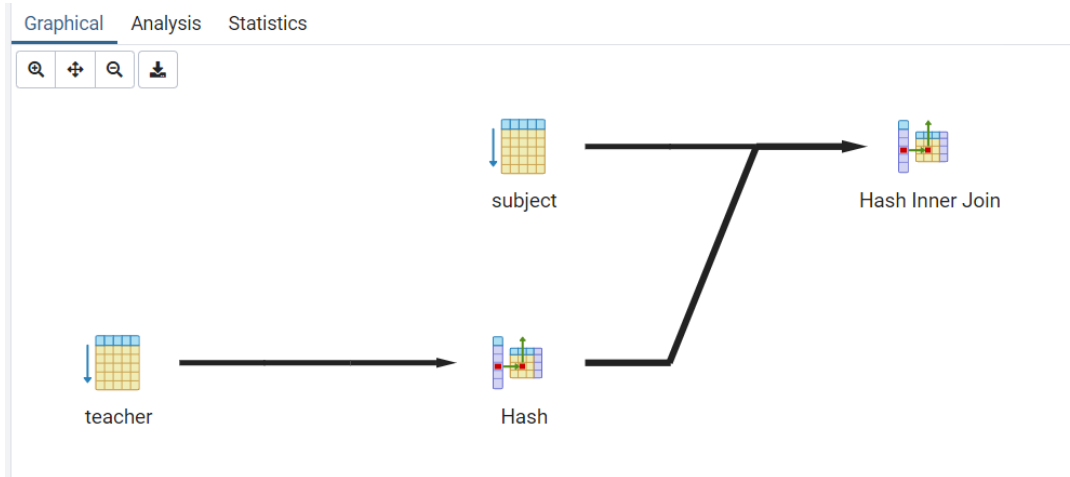
```
EXPLAIN (FORMAT XML) select * from subject  
join teacher on subject.id = teacher.id;
```

```
<explain xmlns="http://www.postgresql.org/2009/explain">
  <Query>
    <Plan>
      <Node-Type>Hash Join</Node-Type>
      <Parallel-Aware>false</Parallel-Aware>
      <Async-Capable>false</Async-Capable>
      <Join-Type>Inner</Join-Type>
      <Startup-Cost>15.18</Startup-Cost>
      <Total-Cost>31.62</Total-Cost>
      <Plan-Rows>230</Plan-Rows>
      <Plan-Width>454</Plan-Width>
      <Inner-Unique>true</Inner-Unique>
      <Hash-Cond>(subject.id = teacher.id)</Hash-Cond>
    <Plans>
      <Plan>
        <Node-Type>Seq Scan</Node-Type>
```

OK

--Пример 78

```
EXPLAIN (FORMAT JSON) select * from subject
join teacher on subject.id = teacher.id;
```



--Пример 79

```
EXPLAIN (FORMAT YAML) select * from subject
join teacher on subject.id = teacher.id;
```

```
- Plan:
  Node Type: "Hash Join"
  Parallel Aware: false
  Async Capable: false
  Join Type: "Inner"
  Startup Cost: 15.18
  Total Cost: 31.62
  Plan Rows: 230
  Plan Width: 454
  Inner Unique: true
  Hash Cond: "(subject.id = teacher.id)"
  Plans:
    - Node Type: "Seq Scan"
      Parent Relationship: "Outer"
      Parallel Aware: false
      Async Capable: false
      Relation Name: "subject"
      Alias: "subject"
      Startup Cost: 0.00
      Total Cost: 15.10
      Plan Rows: 510
      Plan Width: 120
```

OK

boolean

Включает или отключает заданный параметр. Для включения параметра можно написать TRUE, ON или 1, а для отключения — FALSE, OFF или 0. Значение **boolean** можно опустить, в этом случае подразумевается TRUE.

--Пример 80

```
EXPLAIN (timing off) select * from student;
```

	QUERY PLAN
	text
1	Seq Scan on student (cost=0.00..11.60 rows=160 width=472)


Оператор

Любой оператор SELECT, INSERT, UPDATE, DELETE, VALUES, EXECUTE, DECLARE, CREATE TABLE AS и CREATE MATERIALIZED VIEW AS, план выполнения которого вас интересует.

--Пример 81

--Добавление группы

```
EXPLAIN insert into public.group(name, number_of_students)
values ('ИКТБ-2301-01-00', 23);
```

	QUERY PLAN	
	text	
1	Insert on "group" (cost=0.00..0.01 rows=0 width=0)	
2	[...] -> Result (cost=0.00..0.01 rows=1 width=90)	

Методы доступа

Метод доступа характеризует тот способ, который используется для просмотра таблиц и извлечения только тех строк, которые соответствуют критерию отбора. Существуют различные методы доступа: те, при которых индекс не используется, и группа методов, основанных на использовании индекса. Поскольку и таблицы, и индексы хранятся на диске, то для работы с ними эти объекты считываются в память, в которой они представлены разбитыми на отдельные фрагменты, называемые страницами. Эти страницы имеют специальную структуру. Размер страниц по умолчанию составляет 8 килобайт.

Sequential scan(последовательный просмотр)

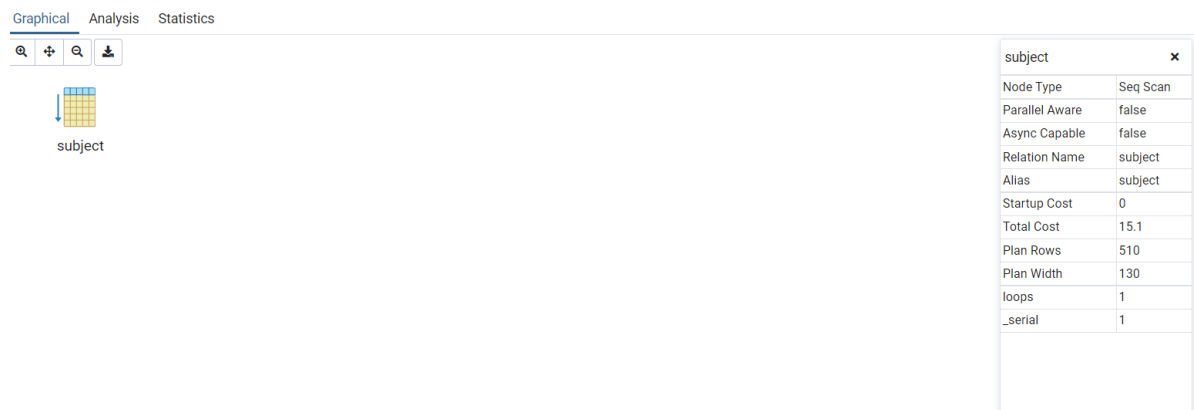
При выполнении последовательного просмотра (sequential scan) обращения к индексам не происходит, а строки извлекаются из табличных страниц в соответствии с критерием отбора.

В том случае, когда в запросе нет предложения WHERE, тогда извлекаются все строки таблицы.

Данный метод применяется, когда требуется выбрать все строки таблицы или значительную их часть, т. е. когда так называемая селективность выборки низка. В таком случае обращение к индексу не ускорит процесс просмотра, а возможно даже и замедлит.

--Пример 82

```
EXPLAIN (FORMAT JSON) select * from subject;
```



Index scan(просмотр на основе индекса)

Просмотр на основе индекса (index scan) предполагает обращение к индексу, созданному для данной таблицы.

Поскольку в индексе для каждого ключевого значения содержатся уникальные идентификаторы строк в таблицах, то после отыскания в индексе нужного ключа производится обращение к соответствующей странице таблицы и извлечение искомой строки по ее идентификатору.

При этом нужно учитывать, что хотя записи в индексе упорядочены, но обращения к страницам таблицы происходят хаотически, поскольку строки в таблицах не упорядочены.

В таком случае при низкой селективности выборки, т. е. когда из таблицы отбирается значительное число строк, использование индексного поиска может не только не давать ускорения работы, но даже и снижать производительность.

--Пример 83

```
EXPLAIN (FORMAT JSON) select * from subject where subject.id =1;
```

Graphical Analysis Statistics



subject_pkey

subject_pkey x	
Node Type	Index Scan
Parallel Aware	false
Async Capable	false
Scan Direction	Forward
Index Name	subject_pkey
Relation Name	subject
Alias	subject
Startup Cost	0.15
Total Cost	8.17
Plan Rows	1
Plan Width	130
Index Cond	(id = 1)
loops	1
_serial	1

Index only scan(просмотр исключительно на основе индекса)

Просмотр исключительно на основе индекса (*index only scan*), как следует из названия метода, не должен, казалось бы, требовать обращения к строкам таблицы, поскольку все данные, которые нужно получить с помощью запроса, в этом случае присутствуют в индексе. Однако в индексе нет информации о видимости строк транзакциям — нельзя быть уверенным, что данные, полученные из индекса, видны текущей транзакции. Поэтому сначала выполняется обращение к карте видимости (*visibility map*), которая существует для каждой таблицы.

В ней одним битом отмечены страницы, на которых содержатся только те версии строк, которые видны всем без исключения транзакциям.

Если полученная из индекса версия строки находится на такой странице, значит, эта строка видна текущей транзакции и обращаться к самой таблице не требуется.

Поскольку размер карты видимости очень мал, то в результате сокращается объем операций ввода/вывода. Если же строка находится на странице, не отмеченной в карте видимости, тогда происходит обращение и к таблице; в результате никакого выигрыша по быстродействию в сравнении с обычным индексным поиском не достигается.

Просмотр исключительно на основе индекса особенно эффективен, когда выбираемые данные изменяются редко. Он может применяться, когда в предложении `SELECT` указаны только имена столбцов, по которым создан индекс.

--Пример 84

```
EXPLAIN (FORMAT JSON)
```

```
select subject.id from subject where subject.id =1;
```


Graphical	Analysis	Statistics
Q	+	Q
subject_pkey		
subject_pkey		
Node Type	Index Only Scan	
Parallel Aware	false	
Async Capable	false	
Scan Direction	Forward	
Index Name	subject_pkey	
Relation Name	subject	
Alias	subject	
Startup Cost	0.15	
Total Cost	8.17	
Plan Rows	1	
Plan Width	8	
Index Cond	(id = 1)	
loops	1	
_serial	1	

Bitmap scan(Просмотр на основе битовой карты)

Просмотр на основе битовой карты (bitmap scan) является модификацией просмотра на основе индекса.

Данный метод позволяет оптимизировать индексный поиск за счет того, что сначала производится поиск в индексе для всех искомых строк и формирование так называемой битовой карты, в которой указывается, в каких страницах таблицы эти строки содержатся.

После того как битовая карта сформирована, выполняется извлечение строк из страниц таблицы, но при этом обращение к каждой странице производится только один раз.

--Пример 85

```
EXPLAIN (FORMAT JSON)
```

```
select * from student where student.course > 2;
```

Graphical	Analysis	Statistics
Q	+	Q
student		
student		
Node Type	Seq Scan	
Parallel Aware	false	
Async Capable	false	
Relation Name	student	
Alias	student	
Startup Cost	0	
Total Cost	12	
Plan Rows	53	
Plan Width	472	
Filter	(course > 2)	
loops	1	
_serial	1	

Способы соединения наборов строк

Другим важным понятием является способ соединения наборов строк (join). Набор строк может быть получен из таблицы с помощью одного из методов доступа, описанных выше.

Набор строк может быть получен не только из одной таблицы, а может быть результатом соединения других наборов.

Важно различать способ соединения таблиц (JOIN) и способ соединения наборов строк. Первое понятие относится к языку SQL и является высокоуровневым, логическим, оно не касается вопросов реализации. А второе относится именно к реализации, это — механизм непосредственного выполнения соединения наборов строк.

Принципиально важным является то, что за один раз соединяются только два набора строк.

Существует три способа соединения: вложенный цикл (nested loop), хеширование (hash join) и слияние (merge join). Они имеют свои особенности, которые PostgreSQL учитывает при выполнении конкретных запросов.

Nested loop(вложенный цикл)

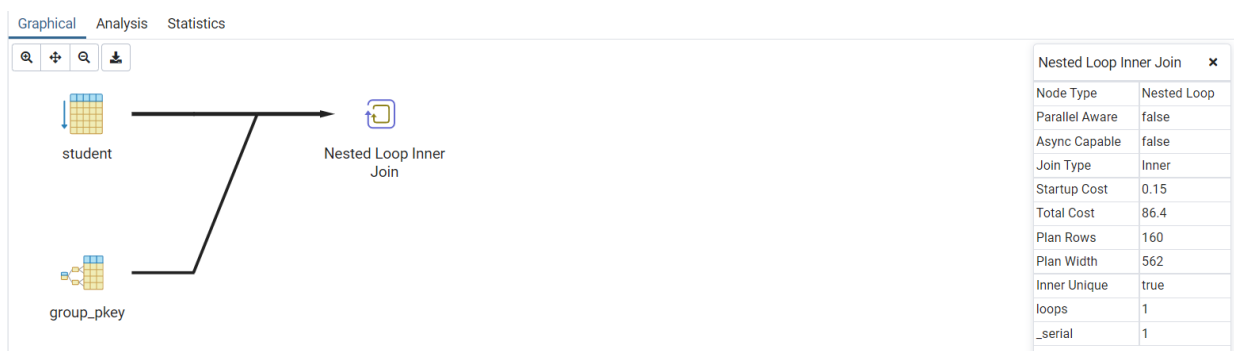
Суть способа «вложенный цикл» в том, что перебираются строки из «внешнего» набора и для каждой из них выполняется поиск соответствующих строк во «внутреннем» наборе. Если соответствующие строки найдены, то выполняется их соединение со строкой из «внешнего» набора. При этом способы выбора строк из обоих наборов могут быть различными.

Метод поддерживает соединения как на основе равенства значений атрибутов (эквисоединения), так и любые другие виды условий.

Поскольку он не требует подготовительных действий, то способен быстро приступить к непосредственной выдаче результата.

Метод эффективен для небольших выборок.

```
--Пример 86
--Соединение таблиц по Id
set enable_mergejoin = off;
set enable_hashjoin = off;
EXPLAIN (FORMAT JSON)
select * from student join public.group on student.id_group
= public.group.id
```



Hash join(Соединение хешированием)

При соединении хешированием строки одного набора помещаются в хеш-таблицу, содержащуюся в памяти, а строки из второго набора перебираются, и для каждой из них проверяется наличие соответствующих строк в хеш-таблице.

Ключом хеш-таблицы является тот столбец, по которому выполняется соединение наборов строк.

Как правило, число строк в том наборе, на основе которого строится хеш-таблица, меньше, чем во втором наборе. Это позволяет уменьшить ее размер и ускорить процесс обращения к ней.

Данный метод работает только при выполнении эквисоединений, поскольку для хеш-таблицы имеет смысл только проверка на равенство проверяемого значения одному из ее ключей.

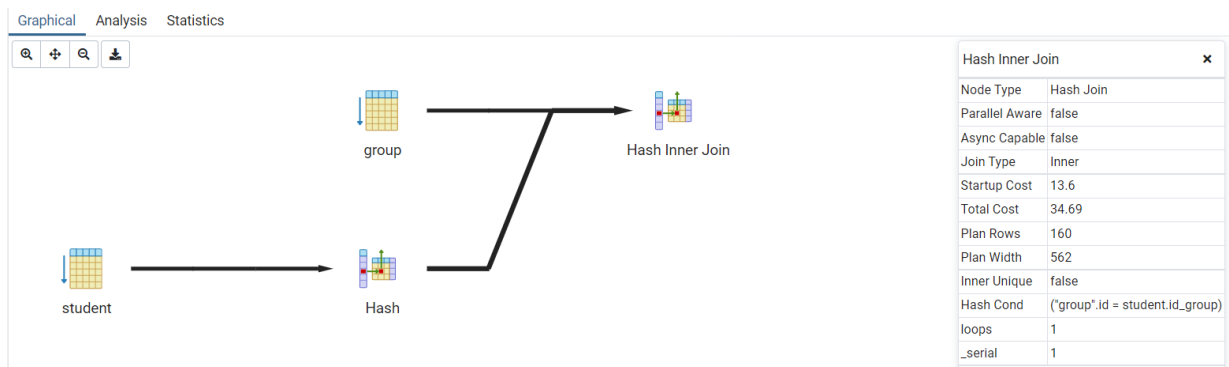
Метод эффективен для больших выборок.

--Пример 87

```
set enable_hashjoin = on;
```

```
EXPLAIN (FORMAT JSON)
```

```
select * from student join public.group on student.id_group  
= public.group.id
```



Merge join(Соединение методом слияния)

Соединение методом слияния производится аналогично сортировке слиянием. В этом случае оба набора строк должны быть предварительно отсортированы по тем столбцам, по которым производится соединение. Затем параллельно читаются строки из обоих наборов и сравниваются значения столбцов, по которым производится соединение. При совпадении значений формируется результирующая строка. Этот процесс продолжается до исчерпания строк в обоих наборах.

Этот метод, как и метод соединения хешированием, работает только при выполнении эквисоединений.

Он пригоден для работы с большими наборами строк.

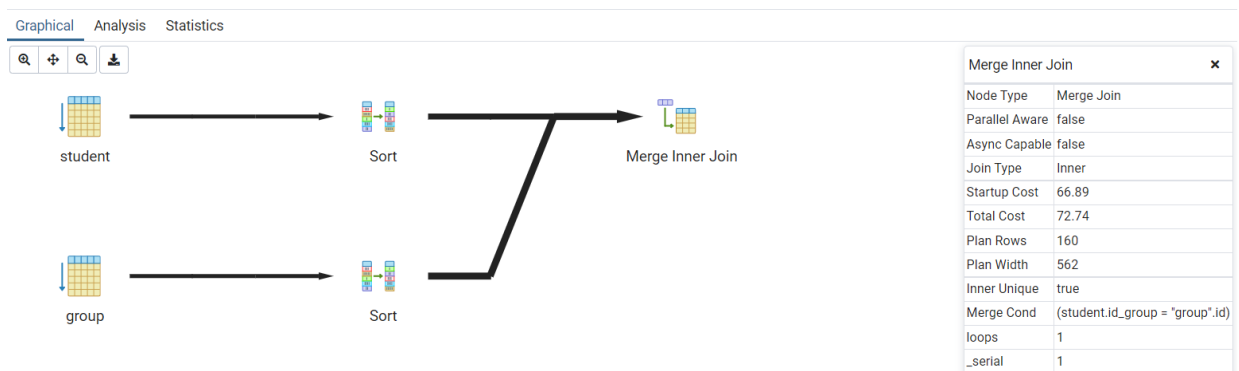
--Пример 88

```
set enable_mergejoin = on;
```

```
set enable_hashjoin = off;
```

```
EXPLAIN (FORMAT JSON)
```

```
select * from student join public.group on student.id_group  
= public.group.id
```



Управление планировщиком

Чтобы запретить планировщику использовать метод соединения на основе хеширования, нужно сделать так:

```
SET enable_hashjoin = off;
```

Чтобы запретить планировщику использовать метод соединения слиянием, нужно сделать так:

```
SET enable_mergejoin = off;
```

А для того чтобы запретить планировщику использовать соединение методом вложенного цикла, нужно сделать так:

```
SET enable_nestloop = off;
```

По умолчанию все эти параметры имеют значение «on» (включено).