Лабораторная работа №5:
Программируемые объекты базы данных

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Данная лабораторная работа посвящена изучению дополнительных возможностей языка SQL: объединению запросов, общих табличных выражений, оконных функций и т.д. А также работе с некоторыми основными объектами баз данных: представлениями, функциями, процедурами и триггерами.

УСЛОВИЯ ДЛЯ НАЧАЛА РАБОТЫ

Для начала работы должна быть завершена работа над третьей лабораторной работой. Нежелательно начинать делать эту лабораторную до выполнения третьей.

СОСТАВ ОТЧЁТА

В отчёте необходимо указать формулировку каждого задания, формулировку каждого выполненного запроса/скрипта (то есть то, что для чего запрос/скрипт нужен), текст запроса (скрипта)/скриншот и результат выполнения запроса/скрипта.

КОЕ-ЧТО ЕЩЁ О ЛАБОРАТОРНОЙ

Все примеры написаны для тестовой базы данных Авиаперевозки, скачанной отсюда: https://postgrespro.ru/education/demodb (данные по полётам за год).

Рекомендуется придумывать более - менее адекватные по смыслу запросы.

Все скрипты нужно писать для своей базы данных, если не указано, что нужно поступать иначе. Каждый запрос должен возвращать не пустой результат. Если каких-то данных в БД не хватает, то надо их добавить, чтобы запрос вернул данные.

ВСПОМОГАТЕЛЬНЫЕ МАТЕРИАЛЫ

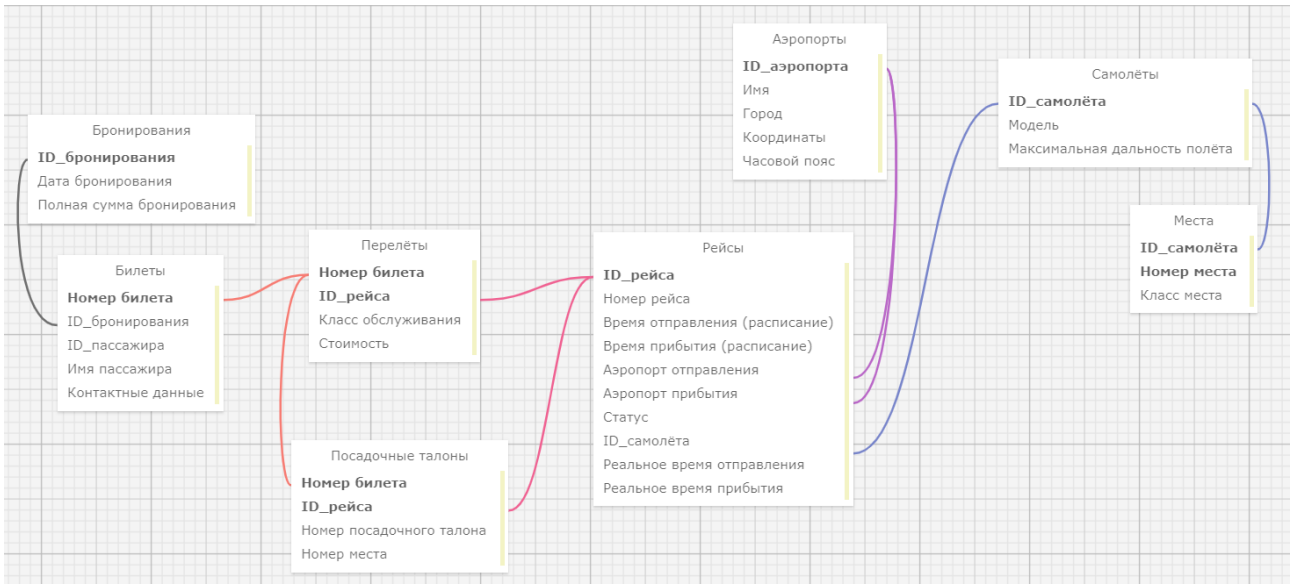
В этом разделе представлены материалы, которые использовались для создания этой лабораторной работы и которые помогут при её выполнении.

Материал	Источник
Документация PostgreSQL (eng)	https://www.postgresql.org/docs/11/index.html
Документация PostgreSQL (ru)	https://postgrespro.ru/docs/postgresql/11/index
Описание БД, используемой в примерах	https://postgrespro.ru/education/demodb
Описание команды SELECT	https://postgrespro.ru/docs/postgresql/11/sql-select
Сообщения об ошибках в PL/pgSQL	https://postgrespro.ru/docs/postgrespro/11/plpgsql-errors-and-messages
Операторы в PL/pgSQL	https://postgrespro.ru/docs/postgrespro/11/plpgsql-statements
Управляющие структуры в PL/pgSQL	https://postgrespro.ru/docs/postgrespro/11/plpgsql-control-structures
Обработка кавычек в PL/pgSQL	https://postgrespro.ru/docs/postgrespro/11/plpgsql-development-tips#PLPGSQL-QUOTE-TIPS
Подстановка переменных в PL/pgSQL	https://postgrespro.ru/docs/postgrespro/11/plpgsql-implementation#PLPGSQL-VAR-SUBST

Документация по триггерам	https://postgrespro.ru/docs/postgresql/11/sql-createtrigger

1 НАЧАЛО РАБОТЫ

Необходимо запустить выбранный ранее клиент для работы с PostgreSQL, а затем ознакомиться со схемой БД, для которой написаны запросы-примеры. По этой схеме были сделаны примеры в ЛР3.



Полезные ссылки для знакомства со схемой:

Что?	Где?
Схема с реальным названием столбцов	https://postgrespro.ru/docs/postgrespro/10/apjs02.html
Общее описание схемы	https://postgrespro.ru/docs/postgrespro/10/apjs03.html
Описание таблиц	https://postgrespro.ru/docs/postgrespro/10/apjs04.html

2 ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ SELECT-ЗАПРОСОВ

2.1 Заполнение таблиц из выборки

В PostgreSQL есть возможность создания таблиц из SELECT-запросов. Такие таблицы можно использовать временно для каких-то нужд разработчиков. Например, для хранения сводных данных. Пример показан ниже. Следующий запрос создаёт отдельную таблицу для хранения информации об одном из рейсов.

```
select * into pg0216 from flights where flight_no = 'PG0216';

select * from pg0216;
```

a Output | Explain | Messages | Notifications

flight_id integer	flight_no character (6)	scheduled_departure timestamp with time zone	scheduled_arrival timestamp with time zone
2880	PG0216	2017-09-14 14:10:00+03	2017-09-14 15:15:00+03
2773	PG0216	2017-07-19 14:10:00+03	2017-07-19 15:15:00+03
2774	PG0216	2017-09-12 14:10:00+03	2017-09-12 15:15:00+03
2775	PG0216	2017-06-23 14:10:00+03	2017-06-23 15:15:00+03
2776	PG0216	2017-08-01 14:10:00+03	2017-08-01 15:15:00+03
2777	PG0216	2017-01-19 14:10:00+03	2017-01-19 15:15:00+03

Синтаксический данный запрос отличается от обычного SELECT добавлением блока INTO, после которого указывается имя создаваемой таблицы.

Задания (общее кол-во запросов(скриптов): 1):

— Создать 1 таблицу с помощью SELECT INTO на основании любого запроса

2.2 Объединение запросов

SQL позволяет комбинировать результаты двух и более запросов, благодаря механизму **объединения запросов (не путать с объединением таблиц)**. В PostgreSQL есть 3 способа объединения запросов.

Способ объединения запросов	Описание
UNION	Добавление результатов 2-го запроса к результату 1-го. В выборке отсутствуют дублирующиеся записи. Если записи необходимо сортировать, то сортировку нужно производить **во втором запросе**.
UNION ALL	То же, что и UNION, но дублирующиеся записи попадают в выборку
INTERSECT	В выборку попадают все строки, которые содержатся и в результате первого запроса, и в результате второго. В выборке отсутствуют дублирующиеся записи
INTERSECT ALL	То же, что и INTERSECT, но дублирующиеся записи попадают в выборку
EXCEPT	В выборку попадают все строки, которые содержатся в результате первого запроса, но при этом отсутствуют в результате второго

Авиакомпания решила увеличить стоимость билетов на все рейсы. Сотрудники решили посмотреть, на сколько увеличится стоимость. Для бизнес-класса средняя стоимость билетов должна увеличиться на 0,03%, а для эконом-класса – на 0,05%. Следующий скрипт объединяет 2

запроса, которые показывают старую и изменённую стоимость на рейсы.

```
(select
 fare_conditions as "Тип места",
 round(avg(amount),2) as "Старая средняя цена",
 round(avg(amount)+avg(amount)*0.03,2) as "Новая средняя цена"
from ticket_flights
where fare_conditions = 'Business'
group by fare_conditions)
union
(select
 fare_conditions as "Тип места",
 round(avg(amount),2) as "Старая средняя цена",
 round(avg(amount)+avg(amount)*0.05,2) as "Новая средняя цена"
from ticket_flights
where fare_conditions = 'Economy'
group by fare_conditions);
```

a Output | Explain | Messages | Notifications

Тип места character varying (10)	Старая средняя цена numeric	Новая средняя цена numeric
Business	51143.42	52677.72
Economy	15959.81	16757.80

Задания (общее кол-во запросов(скриптов): 4):

— Написать 1 объединение запросов с UNION;

— Написать 1 объединение запросов с UNION ALL;

— Написать 1 объединение запросов с INTERSECT или EXCEPT.

3 ПРЕДСТАВЛЕНИЯ

Представление – это тип таблиц, данные в которые попадают из других таблиц. Из представлений можно выбирать данные с помощью select-запросов, но самих данных представления не содержат.

Альтернативные названия: вью/вьюхи/вьюшки (от слова view).

Допустим, необходимо мониторить рейсы, которые прилетают в московские аэропорты. Следующий скрипт создаёт представление на основании запроса, который выводит информацию о рейсах, которые завершаются в Москве.

```
create view MoscowFL as
select fl.* from flights as fl
where fl.arrival_airport in (select airport_code from airports where city = 'Москва');
```

Из представлений можно осуществлять выборку и работать с ними, как с таблицами.

```
select MFL.flight_no, MFL.scheduled_departure, MFL.scheduled_arrival, port.airport_name
from MoscowFL as MFL
join airports as port on MFL.arrival_airport = port.airport_code;
```

ta Output | Explain | Messages | Notifications

	flight_no character (6)	scheduled_departure timestamp with time zone	scheduled_arrival timestamp with time zone	airport_name text
	PG0010	2017-09-05 12:25:00+03	2017-09-05 14:35:00+03	Внуково
2	PG0648	2017-08-31 11:35:00+03	2017-08-31 13:00:00+03	Шереметьево
3	PG0076	2017-09-05 09:15:00+03	2017-09-05 11:50:00+03	Домодедово
4	PG0483	2017-09-12 07:20:00+03	2017-09-12 11:20:00+03	Домодедово
5	PG0065	2017-09-02 12:15:00+03	2017-09-02 18:05:00+03	Внуково
6	PG0408	2017-08-07 11:55:00+03	2017-08-07 12:50:00+03	Домодедово
7	PG0406	2017-08-06 19:00:00+03	2017-08-06 19:55:00+03	Домодедово
8	PG0406	2017-08-21 19:00:00+03	2017-08-21 19:55:00+03	Домодедово
9	PG0406	2017-07-23 19:00:00+03	2017-07-23 19:55:00+03	Домодедово
10	PG0409	2017-09-07 14:55:00+03	2017-09-07 15:50:00+03	Домодедово
11	PG0408	2017-09-07 11:55:00+03	2017-09-07 12:50:00+03	Домодедово
12	PG0407	2017-09-07 12:10:00+03	2017-09-07 13:05:00+03	Домодедово

Зачем нужны представления?

1. Упрощение работы с базой данных.

Чтобы не писать сложные запросы, часть их логики можно выделять в представления, а затем обращаться к ним. В представлении уже будут объединённые из нескольких таблиц данные, к которым можно обратиться в других запросах, а также в приложениях.

2. Формирование отчётов.

3. Увеличение удобства при разграничении прав пользователей. На представления можно навесить различные права, благодаря чему разные группы пользователей смогут увидеть разное кол-во данных.

Представления бывают временными и постоянными. В примере выше было создано постоянное представление. Временные представления можно создать с помощью параметра TEMP. Они могут понадобиться для того, чтобы в текущей сессии упрощать какие-то большие запросы. Временные представления удаляются после завершения сессии пользователя.

```
create TEMP view MoscowFL_temp as
select fl.* from flights as fl
where fl.arrival_airport in (select airport_code from airports where city = 'Москва');
```

Задания (общее кол-во запросов(скриптов): 6):

— Создать 1 временное представление на основании любого запроса. Выбрать данные из этого представления. Перезапустить сервер PostgreSQL. Попробовать повторно выбрать данные из этого представления и убедиться в появлении ошибки;

— Создать 1 постоянное представление на основании любого запроса, объединяющего 2 или более таблиц. Написать запрос с использованием этого представления, в котором также будет объединение 2 или более таблиц.

4 ОБЩИЕ ТАБЛИЧНЫЕ ВЫРАЖЕНИЯ

4.1 Общая информация

PostgreSQL поддерживает технологию CTE (Common Table Expressions), то есть «общих табличных выражений». Данные выражения записываются с помощью предложения WITH.

Данная тема тесно связана с понятием временных таблиц. Временные таблицы нужны для хранения в них каких-либо промежуточных данных. Они существуют до завершения сессии пользователя.

Такие промежуточные таблицы обычно используются для упрощения запросов - в таблицы мы помещаем какие-либо данные, которые используем в запросах.

Синтаксис WITH:

```
WITH имя_табл_выражения [(список_столбцов)]
as
(запрос)
основной_запрос
```

WITH позволяет оборачивать целые запросы в так называемые табличные выражения (представляющие собой временные таблицы), упрощая написание больших запросов и минимизируя количество вложенных запросов в них. Затем, к этим табличным выражениям можно обращаться как к таблицам. WITH нужен для упрощения написания запросов.

В отличие от представлений (VIEW) работа WITH распространяется только на запросы, когда как представления работают со всей БД.

Допустим, необходимо вывести информацию о пункте отправления и назначения, количестве рейсов и общей стоимости всех билетов, проданных на эти рейсы, для рейсов, общая стоимость билетов на которые превышает среднюю стоимость билетов на все рейсы в компании. Для решения данной задачи можно написать следующую конструкцию запросов.

```
create temp view flights_amount as(
select
F.flight_no,
sum(TF.amount) as flight_total_cost

from flights as F
join ticket_flights as TF on TF.flight_id = F.flight_id

group by F.flight_no

having sum(TF.amount) > (
    select round(avg(total_amount),2)
    from (select sum(TF1.amount) as total_amount
          from ticket_flights as TF1
          join flights on flights.flight_id = TF1.flight_id
          group by TF1.flight_id)
    as total_amount_select)
);
```

В данном запросе создаётся временное представление для облегчения получения нужных данных. Представление создаётся на базе запроса с двухуровневым подзапросом.

Один подзапрос (самый глубокий) считает общую стоимость всех билетов для каждого рейса, а второй (тот, что с round(…)) – среднюю общую стоимость всех билетов по всем рейсам (на основании результата другого подзапроса).

```sql
select
port1.airport_name as airport_out,
port2.airport_name as airport_in,
count(f.*) as flights_count,
v.flight_total_cost
from flights_amount as v
join flights as f on f.flight_no = v.flight_no
join airports as port1 on port1.airport_code = f.departure_airport
join airports as port2 on port2.airport_code = f.arrival_airport
group by airport_out, airport_in, v.flight_total_cost
order by flight_total_cost desc;
```

Output | Explain | Messages | Notifications

airport_out text	airport_in text	flights_count bigint	flight_total_cost numeric
Домодедово	Хабаровск-Н...	61	753478300.00
Хабаровск-Но...	Домодедово	61	733797800.00
Домодедово	Толмачёво	61	548218900.00
Толмачёво	Домодедово	61	531503700.00
Хабаровск-Но...	Пулково	61	507672400.00
Пулково	Хабаровск-Н...	61	498750700.00
Шереметьево	Толмачёво	61	458130400.00

Второй запрос выводит пункт отправления и пункт назначения; количество перелётов, выполненных по данному рейсу; сумму стоимостей всех приобретённых билетов.

Данный запрос можно переписать с использованием WITH, разбив его на несколько частей.

```sql
with flights_amount as (
    select
        f.flight_no,
        sum(tf.amount) as flight_total_amount
    from flights as f
    join ticket_flights as tf on tf.flight_id = f.flight_id
    group by f.flight_no
), top_flights as (
    select
        flight_no,
        flight_total_amount
    from flights_amount
    where flight_total_amount > (select avg(flight_total_amount) from flights_amount)
)
select
port1.airport_name as airport_out,
port2.airport_name as airport_in,
count(f.*) as flights_count,
top_flights.flight_total_amount

from top_flights
join flights as f on f.flight_no = top_flights.flight_no
join airports as port1 on port1.airport_code = f.departure_airport
join airports as port2 on port2.airport_code = f.arrival_airport
group by airport_out, airport_in, top_flights.flight_total_amount
order by top_flights.flight_total_amount desc;
```

1 2 3

Output | Explain | Messages | Notifications

airport_out text	airport_in text	flights_count bigint	flight_total_amount numeric
Домодедово	Хабаровск-Н...	61	753478300.00
Хабаровск-Но...	Домодедово	61	733797800.00
Домодедово	Толмачёво	61	548218900.00
Толмачёво	Домодедово	61	531503700.00

В пунктах «1» и «2» создаются временные таблицы из запросов.

В пункте «1» выполняется расчёт суммарной стоимости приобретённых билетов для разных рейсов.

В пункте «2» на основании результатов запроса из пункта «1» осуществляется поиск рейсов, суммарная стоимость приобретённых билетов которых больше, чем средний показатель по всем рейсам.

В пункте «3» выполняется основной запрос, использующий результаты запроса из пункта «2».

4.2 Иерархические запросы

Нередко можно столкнуться с ситуацией, когда в структуре БД существует иерархическая связь либо внутри одной таблицы, либо между несколькими таблицами. Получить всю цепочку иерархических данных с помощью языка SQL можно разными способами. Но одним из самых удобных является использование WITH.

Классический пример иерархических данных: есть сотрудники, у них есть начальники, у начальников есть другие начальники. Нужно вывести Петра и всех его подчинённых.

empl_id [PK] integer	empl_name character varying (20)	manager_id integer
1	Иван	[null]
2	Виктор	[null]
3	Константин	1
4	Николай	2
5	Пётр	1
6	Фёдор	5
7	Сергей	6

Судя по исходным данным, в результате должны быть выведены записи с Петром, Фёдором и Сергеем.

Синтаксис рекурсивного CTE:

```
WITH RECURSIVE имя_табл_выражения [(список_столбцов)]
as
(
исходный_запрос
UNION [ALL]
рекурсивный запрос
)
основной_запрос
```

исходный_запрос – точка, с которой начинается рекурсия. Данный запрос выполняется только 1 раз в самом начале.
рекурсивный_запрос – запрос, который будет выполняться до тех пор, пока будет выполняться условие этого запроса.

основной_запрос – необходим для работы с данными, хранящимися в табличном выражении.

Рекурсивные запросы с WITH выполняются по следующим правилам:

1. Вычисляется нерекурсивная часть. Для UNION (но не UNION ALL) отбрасываются дублирующиеся строки. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную **рабочую таблицу**.
2. Пока рабочая таблица не пуста, повторяются следующие действия:
    a. Вычисляется рекурсивная часть так, что рекурсивная ссылка на сам запрос обращается к текущему содержимому рабочей таблицы. Для UNION (но не UNION ALL) отбрасываются дублирующиеся строки и строки, дублирующие ранее полученные. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную **промежуточную таблицу**.
    b. Содержимое рабочей таблицы заменяется содержимым промежуточной таблицы, а затем промежуточная таблица очищается.

```
with recursive example1 as(
    select
        t1.empl_id,
        t1.empl_name
    from demoWith as t1
    where empl_name = 'Пётр'

    union all

    select
        t2.empl_id,
        t2.empl_name
    from demoWith as t2
    join example1 as ex1 on ex1.empl_id = t2.manager_id
)
select * from example1;
```

Output | Explain | Messages | Notifications

empl_id integer	empl_name character varying (20)
5	Пётр
6	Фёдор
7	Сергей

Данный запрос выполняется следующим образом:

1. Выполняется первый запрос

```
select
    t1.empl_id,
    t1.empl_name
from demoWith as t1
where empl_name = 'Пётр'
```

Во временную таблицу поместилась 1 запись со значением empl_name = «Пётр». То есть временная таблица с результатами будет выглядеть так:

empl_id	empl_name	manager_id
5	Пётр	1

2. Выполняется второй запрос

```
select
    t2.empl_id,
    t2.empl_name
from demoWith as t2
join example1 as ex1 on ex1.empl_id = t2.manager_id
```

Он выполнился 2 раза, так как исходные данные позволяют всего лишь 2 раза выполниться заданному условию, а именно: **ex1.empl_id = t2.manager_id**

В первом обращении к **ex1.empl_id** производится обращение к таблице, полученной на шаге 1. То есть вместо **ex1.empl_id** подтягивается идентификатор сотрудника с именем «Пётр». И в ситуациях, когда идентификатор Петра будет указан для какого-то сотрудника в столбце «manager_id» (ID начальника), - это будет говорить о том, что был найден подчинённый Петра.

Затем во временной таблице с результатами будет уже 2 записи – будет найден прямой подчинённый Петра по имени Фёдор.

empl_id	empl_name	manager_id
5	Пётр	1
6	Фёдор	5

После этого будет найден уже косвенный подчинённый Петра – Сергей, потому что Сергей является прямым подчинённым Фёдора.

Задания (общее кол-во запросов(скриптов): 8):

— Написать запрос с использованием CTE, использующий 2 или более таблицы из схемы БД;

— Написать рекурсивный запрос с использованием CTE.

Про рекурсивный запрос:

Если в варианте есть иерархические данные (например, есть таблица «Жанр» со столбцами id_жанра, Название, id_родительского_жанра и т.п.), то необходимо написать какой-либо рекурсивный запрос **на базе имеющихся таблиц**. Например, вывести все жанры заданной книги/вывести все группы товаров (если группы товаров реализованы в виде иерархии), к которым относится товар и т.п.

Если в варианте нет иерархических данных, то необходимо создать таблицу из примера про иерархические запросы, наполнить её теми же данными и написать запрос, который выведет всех начальников сотрудника с именем «Сергей».

После выполнения задания можно удалить таблицу.

5 ВВЕДЕНИЕ В PL/PGSQL

5.1 Общая информация

В СУБД PostgreSQL как и во многих других реализовано расширение языка SQL. В данной СУБД оно называется PL/pgSQL (по сути – отдельный язык).

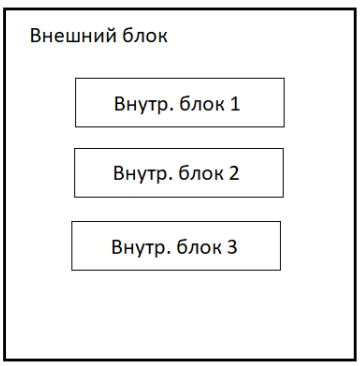
Он является блочно – структурированным процедурным языком и служит для так называемого серверного программирования на уровне СУБД.

Благодаря этому языку можно группировать множество различных операций в единые блоки кода и запускать их при необходимости самостоятельно или автоматически. Это сильно расширяет возможности СУБД.

Изучение данной темы отчасти выходит за рамки текущего курса и будет более подробно рассматриваться в рамках дисциплины «СУБД» (правда, на примере СУБД Microsoft SQL Server).

Структура

Так как язык является блочно-структурированным, то операторы в скриптах размещаются внутри блоков. Блоки могут быть зависимыми и независимыми друг от друга. Один блок можно вкладывать внутрь другого блока. Примерная структура изображена ниже.




```
[ <<метка>> ]
[ DECLARE
```

```
  объявления ]
BEGIN
  операторы
END [ метка ];
```

Метка – идентификатор (имя) блока. Необязательный элемент.

DECLARE – область объявления переменных. Необязательный элемент.

BEGIN – начало блока кода.

END – конец блока кода. Если END указывается для установки завершения блока кода, то к END надо дописывать точку с запятой. END также указывает на конец всего скрипта – в этом случае точка с запятой не требуется.

В следующем примере складываются 2 числа и выводится на экран результат сложения.

```
DO $$
DECLARE
    param1 int :=2;
    param2 int :=2;
    result_ int :=0;
BEGIN
    result_:=param1+param2; --каждый оператор должен заканчиваться символом ;
    RAISE INFO 'Результатом сложения param1 и param2 является число %', result_;
END $$
```

DO – это команда, которая выполняет блок кода.

После неё и в самом конце располагается 2 символа $. Они нужны для экранирования тела скрипта, потому что оно должно передаваться на выполнение в виде строки. Эти символы как раз нужны для такого преобразования.

Операторы можно записывать как большими, так и маленькими буквами – они всё равно при выполнении будут преобразовываться в нижний регистр.

Альтернативная запись:

```
DO '
DECLARE
    param1 int :=2;
    param2 int :=2;
    result_ int :=0;
BEGIN
    result_:=param1+param2;
    RAISE INFO ''Результатом сложения param1 и param2 является число %'', result_;
END '
```

В блоке объявления переменных объявляется 3 целочисленных переменных.

В теле блока операторов выполняется расчёт.

Благодаря команде **RAISE** можно вывести информационное сообщение.

INFO – это значение параметра «уровень сообщения». Подробнее о выводе сообщений можно прочитать здесь: https://postgrespro.ru/docs/postgrespro/11/plpgsql-errors-and-messages

Если говорить коротко, то данные сообщения нужны для логирования (журналирования) операций.

```
  RAISE INFO 'Результатом сложения param1 и param2 является число %', result_;
d $$
```

Вместо символа «%» подставится значение переменной «result_».

Следующий пример показывает использование двух блоков.

```
21  DO $$
22  <<main_block>>
23  DECLARE
24      result_ int := 0;
25  BEGIN
26      RAISE INFO 'Текущее значение = %', result_;
27
28      <<mini_block1>>
29      DECLARE
30          param1 int :=1;
31      BEGIN
32          RAISE INFO 'В этом блоке к переменной result_ прибавится значение param1';
33          result_:=result_+param1;
34      END mini_block1;
35
36      <<mini_block2>>
37      BEGIN
38          RAISE INFO 'В этом блоке к переменнной result_ прибавится 3';
39          result_:=result_+3;
40      END mini_block2;
41
42      RAISE INFO 'В результате получилось значение %', result_;
43  END main_block $$
44
```

Data Output | Explain | Messages | Notifications

```
ИНФОРМАЦИЯ:  Текущее значение = 0
ИНФОРМАЦИЯ:  В этом блоке к переменной result_ прибавится значение param1
ИНФОРМАЦИЯ:  В этом блоке к переменнной result_ прибавится 3
ИНФОРМАЦИЯ:  В результате получилось значение 4
DO

Query returned successfully in 132 msec.
```

Цветом выделены все блоки в данном скрипте. В примере также фигурируют метки блоков.

Переменные

Как известно, переменные объявляются после **DECLARE**

Некоторые примеры объявления переменных:

```
var1 int; --объявление целочисленной переменной
var2 varchar(10); --объявление строки, состоящей максимум из 10 символов
var3 constant int:=5; --объявление константы
var4 int default 11; --аналог записи var4 int :=11;
var5 numeric(3,1):=90.1;
```

Если переменная только объявлена, но не инициализирована, то она примет значение NULL.

```
52  BEGIN
53      RAISE INFO 'Значение переменной var1 = %',var1;
54  END $$
55
```

Data Output Explain Messages Notifications

```
ИНФОРМАЦИЯ:  Значение переменной var1 = <NULL>
DO
```

Если объявить переменную во внешнем блоке, а затем объявить другую переменную с таким же именем во внутреннем блоке, то операции во внутреннем блоке будут касаться только новой переменной. То есть область видимости – только этот блок.

```
57  DO $$
58  <<main_block>>
59  DECLARE
60      result_ int := 0;
61  BEGIN
62      RAISE INFO 'Текущее значение = %', result_;
63
64      <<mini_block1>>
65      DECLARE
66          result_ int :=10;
67          param1 int:=1;
68      BEGIN
69          RAISE INFO 'В этом блоке к переменной result_ прибавится значение param1';
70          result_:=result_+param1;
71      END mini_block1;
72
73      RAISE INFO 'В результате получилось значение %', result_;
74  END main_block $$
75
```

Data Output Explain Messages Notifications

```
ИНФОРМАЦИЯ:  Текущее значение = 0
ИНФОРМАЦИЯ:  В этом блоке к переменной result_ прибавится значение param1
ИНФОРМАЦИЯ:  В результате получилось значение 0
DO
```

Далее будет несколько примеров, которые могут помочь в освоении синтаксиса. Ещё больше материала есть в документации.

Описание основных выражений и операторов языка:
https://postgrespro.ru/docs/postgrespro/11/plpgsql-statements

https://postgrespro.ru/docs/postgrespro/11/plpgsql-control-structures
https://postgrespro.ru/docs/postgrespro/11/plpgsql-development-tips#PLPGSQL-QUOTE-TIPS
https://postgrespro.ru/docs/postgrespro/11/plpgsql-implementation#PLPGSQL-VAR-SUBST

5.2 Функции и процедуры

Хранимая процедура – объект БД, хранящийся на сервере. Может возвращать одно значение и набор значений, а также не возвращать значения вовсе. Процедуры выполняются отдельно, их нельзя вызвать в запросах и других изображениях.
Процедуры нужны для инкапсуляции повторяющихся действий, разделения бизнес-логики с клиентскими приложениями (часть логики реализована на стороне БД, часть – реализовывается в приложениях), для повышения безопасности БД, для сокращения сетевого трафика в вопросе обмена данными между клиентскими приложениями и СУБД.

Функция отличается от процедуры тем, что её можно вызывать в запросах и других выражениях. Всегда возвращает только одно значение. Функции очень полезны для упрощения написания запросов.
В отличие от процедур, функции в postgresql появились давно. Процедуры – в 11-й версии.

В следующем примере реализована функция, которая вычисляет количество оставшихся мест на указанный рейс.

```
CREATE OR REPLACE FUNCTION getFreeSeats(fl_id int) RETURNS int
LANGUAGE plpgsql -- указываем, какое расширение языка используется
AS $$
DECLARE
    ticket_count int:=0; -- кол-во приобретённых билетов
    seats_count int:=0; -- кол-во мест в самолёте
BEGIN
    --проверка входного id
    DECLARE
        isExist boolean;
    BEGIN
        -- проверка существования рейса с заданным id
        select exists(select 1 from flights where flight_id = $1) into isExist;
        IF isExist <> true THEN
            RAISE EXCEPTION 'Рейса с заданным id = % не существует',$1;
        ELSE
            -- проверка того, что заданный рейс ещё не состоялся
            select exists(select 1 from flights where status='Scheduled' and flight_id = $1) into isExist;
            IF isExist <> true THEN
            RAISE EXCEPTION 'Рейс с id = % уже состоялся',$1;
        END IF;
        END IF;
    END;
    --вычисление кол-ва купленных на рейс билетов
    BEGIN
        select count(tf.*) into ticket_count
        from ticket_flights as tf
        join flights as f on f.flight_id=tf.flight_id
        where f.flight_id = $1;
    END;
    --вычисление кол-ва билетов в самолёте
    BEGIN
        select count(s.*) into seats_count
        from seats as s
        join aircrafts as air on air.aircraft_code = s.aircraft_code
        join flights as f on f.aircraft_code = air.aircraft_code
        where f.flight_id = $1;
    END;
RETURN seats_count - ticket_count; --кол-во свободных мест: кол-во мест в самолёте - кол-во купленных билетов
END $$;
```

В функции есть 1 входной параметр **fl_id** с типом int. Для подстановки параметра используется запись типа **$№**, где № - номер параметра.

Пример вызова:

```
select getFreeSeats(22387);
```

ta Output | Explain | Messages | Notifica

getfreeseats integer
3

```
54  select getFreeSeats(14);
```

Data Output | Explain | Messages | Notifications

```
ERROR:  ОШИБКА:  Рейс с id = 14 уже состоялся
CONTEXT:  функция PL/pgSQL getfreeseats(integer), строка 18, оператор RAISE
```

```
54  select getFreeSeats(33122);
```

Data Output | Explain | Messages | Notifications

```
ERROR:  ОШИБКА:  Рейса с заданным id = 33122 не существует
CONTEXT:  функция PL/pgSQL getfreeseats(integer), строка 13, оператор RAISE
```

Как и в большинстве языков программирования, в SQL есть перегрузка функций.

В следующих примерах создаётся 2 функции. В одной функции – 1 аргумент. Во второй функции – 2 аргумента. Функция возвращает ID рейса на выбранную дату. В своём первом варианте не учитывается город, во втором – учитывается город, из которого производится отправление.

```
CREATE OR REPLACE FUNCTION getAvailableFlights(fl_date date) RETURNS SETOF integer
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY (select flight_id from flights
                  where scheduled_departure::date = $1
                  and status = 'Scheduled');

    -- Так как выполнение ещё не закончено, можно проверить, были ли возвращены строки,
    -- и если нет, выдать исключение.
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Нет рейсов на дату: %.', $1;
    END IF;

    RETURN;
 END $$
```

```
CREATE OR REPLACE FUNCTION getAvailableFlights(fl_date date, city text) RETURNS int
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY (select flight_id from flights as f
                  join airports as air on air.airport_code = f.departure_airport
                  where f.scheduled_departure::date = $1
                  and f.status = 'Scheduled'
                  and air.city = $2);

    -- Так как выполнение ещё не закончено, можно проверить, были ли возвращены строки,
    -- и если нет, выдать исключение.
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Нет рейсов на дату: %.', $1;
    END IF;

    RETURN;
 END $$

select getAvailableFlights('2017-09-10'::date,'Москва');
```

a Output | Explain | Messages | Notifications

getavailableflights integer
7784
5000
5001

Для примера используется иной способ работы с SELECT-запросами и их результатами

Пример вызова функции из запроса:

```
select flight_no, scheduled_departure, scheduled_arrival, departure_airport, arrival_airport
from flights where flight_id in (select getAvailableFlights('2017-09-10'::date,'Москва'))
order by scheduled_departure;
```

a Output | Explain | Messages | Notifications

flight_no character (6)	scheduled_departure timestamp with time zone	scheduled_arrival timestamp with time zone	departure_airport character (3)	arrival_airport character (3)
PG0120	2017-09-10 09:00:00+03	2017-09-10 14:45:00+03	SVO	MJZ
PG0591	2017-09-10 09:00:00+03	2017-09-10 12:55:00+03	SVO	TOF
PG0414	2017-09-10 09:05:00+03	2017-09-10 11:10:00+03	VKO	MMK
PG0239	2017-09-10 09:05:00+03	2017-09-10 11:40:00+03	DME	HMA

Этот запрос выводит некоторую информацию о рейсах 9 октября 2017 года, которые производятся из Москвы.

Следующий запрос по циклу выводит на экран некоторую информацию о рейсе. Входные параметры: пункт отправления и пункт назначения.

```sql
CREATE OR REPLACE PROCEDURE for_example(dep_air char(3), arr_air char(3))
LANGUAGE plpgsql
AS $$
DECLARE
    query_result record;
BEGIN
--вывод информационных сообщений
RAISE NOTICE 'Пункт отправления - %', $1;
RAISE NOTICE 'Пункт назначения - %', $2;
RAISE NOTICE 'Список рейсов с данными пунктами назначения и отправления:';
FOR query_result IN -- переменной query_result присваиваются результаты запроса
    select *
    from flights
    where departure_airport = $1 and arrival_airport = $2
LOOP
    RAISE NOTICE '№ рейса: %', query_result.flight_no; --из переменной, хранящей результат запроса, берётся только № рейса.
END LOOP;
END $$;

call for_example('VKO','HMA');

```

Data Output | Explain | Messages | Notifications

```
ЗАМЕЧАНИЕ:  Пункт отправления - VKO
ЗАМЕЧАНИЕ:  Пункт назначения - HMA
ЗАМЕЧАНИЕ:  Список рейсов с данными пунктами назначения и отправления:
ЗАМЕЧАНИЕ:  № рейса: PG0052
ЗАМЕЧАНИЕ:  № рейса: PG0052
ЗАМЕЧАНИЕ:  № рейса: PG0052
ЗАМЕЧАНИЕ:  № рейса: PG0052
ЗАМЕЧАНИЕ:  № рейса: PG0052
```

Задания (общее кол-во запросов(скриптов): 12):

— Реализовать функцию, возвращающую результат каких-либо вычислений. Вызвать функцию в запросе, из которого в эту функцию будут подставляться значения.

Если в БД есть информация о стоимости чего-либо/закупках/продаже товара, то необходимо задействовать таблицы с этой информацией. Если в БД нет, то можно придумать любую другую функцию, вычисляющую чего – либо.

— Реализовать функцию с использованием условных операторов и цикла FOR (используя таблицы из своей БД) с 1 или большим числом входных параметров. Вызвать функцию в запросе, из которого в неё будут подставляться значения.

Задания на процедуры можно выполнить только при наличии версии PostgreSQL >=11. Если нет возможности установить такие версии, то необходимо реализовать ещё 2 любые функции для расчёта чего-либо/преобразования каких-либо данных вашей БД.

— Реализовать процедуру, которая создаёт тестовую таблицу с тремя любыми столбцами.

— Реализовать процедуру с входными параметрами, которая с помощью функции RAISE выводит на экран результат каких-то операций со столбцами существующей в БД таблицы и этими входными параметрами.

5.3 Триггеры

Триггер – это хранимая процедура, целью которой является выполнение чего – либо при наступлении некоторых событий. Триггеры в некотором смысле можно назвать обработчиком событий.

Механизм триггеров позволяет выполнять определенные действия «в ответ» на определенные события.

Триггер состоит из двух частей: собственно триггера (который определяет события) и триггерной функции (которая определяет действия). И триггер, и функция являются самостоятельными объектами БД.

Когда возникает событие, на которое «подписан» триггер, вызывается триггерная функция. Ей передается контекст вызова, чтобы можно было определить, какой именно триггер и в каких условиях вызвал функцию.

Триггерная функция — это обычная функция, которая написана с учетом некоторых соглашений:

— она пишется на любом языке, кроме чистого SQL (в нашем случае на PL/pgSQL);

— она не имеет параметров;

— она возвращает значение типа trigger (на самом деле это псевдотип, по факту возвращается запись, соответствующая строке таблицы).

Триггеры могут срабатывать на вставку (insert), обновление (update) или удаление (delete) строк в таблице или представлении, а также на опустошение (truncate) таблиц.

Триггер может срабатывать до выполнения действия (before), после него (after), или вместо него (instead of).

Триггер может срабатывать один раз для всей операции (for each statement), или каждый раз для каждой затронутой строки (for each row).

В следующем примере создаётся таблица для «аудита изменения времени отправления и прибытия у рейсов» и триггер, который добавляет прежнее и новое значение времени в таблицу для аудита + записывает время изменения записи.

```
-- Функция, которая будет вызвана триггером
CREATE OR REPLACE FUNCTION flights_audit_function() RETURNS trigger
LANGUAGE plpgsql
AS $$
BEGIN
    IF (NEW.scheduled_departure IS NOT NULL AND NEW.scheduled_departure <> OLD.scheduled_departure) THEN
        insert into flights_audit(flight_id,old_scheduled_date,new_scheduled_date,change_date)
        values(OLD.flight_id,OLD.scheduled_departure,NEW.scheduled_departure,now());
    END IF;
    IF (NEW.scheduled_arrival IS NOT NULL AND NEW.scheduled_arrival <> OLD.scheduled_arrival) THEN
        insert into flights_audit(flight_id,old_arrival_date,new_arrival_date,change_date)
            values(OLD.flight_id,OLD.scheduled_arrival,NEW.scheduled_arrival,now());
    END IF;
    RETURN NEW;
END $$;

CREATE TRIGGER flight_audit
AFTER INSERT OR UPDATE OR DELETE ON flights FOR EACH ROW EXECUTE FUNCTION flights_audit_function();

update flights set scheduled_departure = '2017-09-10 10:50:00+03', scheduled_arrival = '2017-09-10 15:55:00+03'
where flight_id = 1185;

select * from flights_audit;
```

Output | Explain | Messages | Notifications

id [PK] integer	flight_id integer	old_scheduled_date timestamp with time zone	new_scheduled_date timestamp with time zone	old_arrival_date timestamp with time zone	new_arrival_date timestamp with time zone	change_date timestamp with time zone
2	1185	2017-09-10 09:50:00+03	2017-09-10 10:50:00+03	[null]	[null]	2020-05-31 12:16:21.363961+03
3	1185	[null]	[null]	2017-09-10 14:55:00+03	2017-09-10 15:55:00+03	2020-05-31 12:16:21.363961+03

Первый скрипт – создание триггерной функции.

В ней значения **NEW** – это изменённые значения в столбцах, а значения **OLD** – прежние значения, которые были до изменения.

Для получения момента времени изменения данных используется функция now().

Второй скрипт – создание триггера, который будет выполняться после вставки/обновления/удаления записи в таблице для каждой строки, которая была добавлена/изменена/удалена.

В третьем запросе на 1 час меняется время вылета и прибытия.

Последний запрос выводит записи таблицы аудита.

Задания (общее кол-во запросов(скриптов): 14):

— Реализовать триггер для аудита изменения каких-либо данных в любой таблице, за которой в теории можно было бы следить (например, сохранение истории перевода студента из одной группы в другую (это всего лишь UPDATE-операция по изменению группы студента), изменение ФИО человека и т.д.).

— Если в БД есть вычисляемые столбцы, то реализовать триггер для вычисления значений таких столбцов.

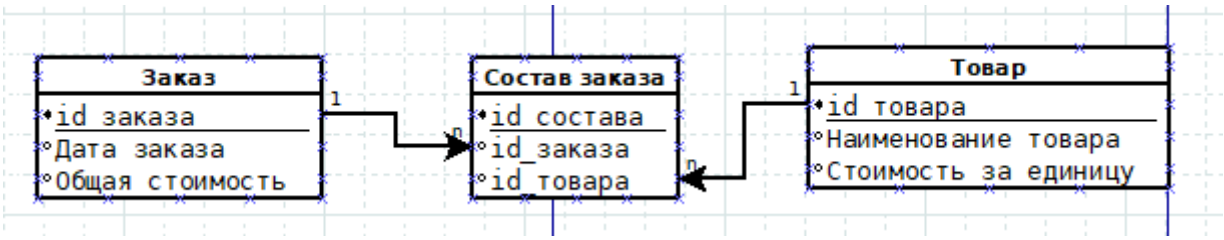
Если таких столбцов нет, то необходимо реализовать ещё один триггер для аудита чего-либо.

Пример вычисляемого столбца показан ниже:

Например, в таблице с заказами есть столбец «Общая сумма заказа»), то реализовать триггер, который будет обновлять суммовой столбец. (Данное задание нужно только для ознакомления с триггерами).

Пример:

Есть следующая структура таблиц



Со следующими данными:

Data Output | Explain | Messages | Notifications

	order_id [PK] integer	order_date date	total_cost integer
1	1	2020-06-06	[null]

Общую стоимость не записываем пока. Посчитаем её триггером.

```sql
select orders.order_id, orders.order_date, prod.product_name, prod.product_cost, op.product_count
from orders
join order_prod as op on op.order_id = orders.order_id
join products as prod on prod.product_id = op.product_id;
```

ta Output | Explain | Messages | Notifications

order_id integer	order_date date	product_name character varying (30)	product_cost integer	product_count integer
1	2020-06-06	Товар 1	500	3
1	2020-06-06	Товар 2	1000	2

Создана триггерная функция и триггер, срабатывающий после записи значений в таблицу/изменения или удаления значений.

Обратите внимание, что если в триггерной функции, которая вызывается после INSERT/UPDATE/DELETE-запроса, возникает ошибка, то INSERT/UPDATE/DELETE-запросы откатываются.

```sql
CREATE OR REPLACE FUNCTION calc_total_cost() RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    total_sum int; -- переменная для общей суммы заказа
    ord_id int; --идентификатор заказа
BEGIN

    --NEW.order_id появляется тогда, когда в таблицу order_prod (состав заказа)
    --появляется новая запись (например, при добавлении нового товара в заказ)
    --OLD.order_id используется тогда, когда новый товар в заказ не добавлялся, а, например, обновлялось кол-во товара
    IF NEW.order_id IS NULL THEN
        ord_id := OLD.order_id;
    ELSE ord_id := NEW.order_id;
    END IF;

    --в следующем update запросе происходит обновление общей суммы заказа
    --общая сумма вычисляется во вложенном select-запросе
    update orders set total_cost = (
        select sum(pr.product_cost*op.product_count)
        from orders
        join order_prod as op on op.order_id = orders.order_id
        join products as pr on pr.product_id = op.product_id
        )
        where order_id = ord_id; --ord_id здесь будет равен либо OLD.order_id, либо NEW.order_id
RETURN NEW;
END $$;

CREATE TRIGGER orders_sum_audit
AFTER INSERT OR UPDATE OR DELETE on order_prod FOR EACH ROW EXECUTE FUNCTION calc_total_cost();
```

Выше было показано, что общая сумма заказа составляет 500*3 + 1000*2 = 3500

```
select orders.order_id, orders.order_date, prod.product_name, prod.product_cost, op.product_count
from orders
join order_prod as op on op.order_id = orders.order_id
join products as prod on prod.product_id = op.product_id;
```

ta Output | Explain | Messages | Notifications

order_id integer	order_date date	product_name character varying (30)	product_cost integer	product_count integer
1	2020-06-06	Товар 1	500	3
1	2020-06-06	Товар 2	1000	2

Изменим количество какого-нибудь товара в заказе.

```
update order_prod set product_count = 4 where order_id = 1 and product_id = 2;
```

```
select orders.order_id, orders.order_date, prod.product_name, prod.product_cost, op.product_count
from orders
join order_prod as op on op.order_id = orders.order_id
join products as prod on prod.product_id = op.product_id;

CREATE OR REPLACE FUNCTION calc_total_cost() RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
```

ta Output | Explain | Messages | Notifications

order_id integer	order_date date	product_name character varying (30)	product_cost integer	product_count integer
1	2020-06-07	Товар 1	500	3
1	2020-06-07	Товар 2	1000	4

Теперь сумма заказа должна составлять 500*3 + 1000*4 = 5500

На следующем скриншоте видно, что общая сумма заказа обновилась.

```
select * from orders;

update order_prod set product_count =
```

a Output | Explain | Messages | Notifications

order_id [PK] integer	order_date date	total_cost integer
1	2020-06-07	5500