Министерство науки и образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ ПО КУРСУ «БАЗЫ ДАННЫХ»

Лабораторная работа №3
«Создание БД для аналитики»

Авторы:
Скворцова М.А., magavrilova@bmstu.ru
Лапшин А.В.

Москва, 2022

Общие сведения

Сокращения

SQL – Structured Query Language («язык структурированных запросов»).

БД – База данных.

СУБД – Система управления базами данных.

JSON - JavaScript Object Notation.

РСУБД часто используются в аналитических целях. В этом случае объем данных постоянно увеличивается. При этом их изменение или удаление затрудненно из-за большого объема. Хранение данных в денормализованном виде с помощью массивов и json-формата упрощает их обработку. В этом случае с СУБД взаимодействует человек-аналитик, который может иметь доступ не ко всем данным. Ему может быть неудобно работать с большими SELECT-запросами. Для него сложно заранее предугадать все задачи, которые перед ним будут стоять. При этом разработчику такой базы данных необходимо уметь проектировать высокопроизводительную схему данных, ориентированную на большие объемы и регулярно появляющиеся и устаревающие данные, использовать механизмы секционирования и построения индексов, анализировать планы запросов и оптимизировать их, упрощать интерфейс базы данных с помощью процедур и представлений, ограничивать доступ с помощью ролей и прав.

Данная лабораторная работа призвана сформировать у студента понимание особенностей создания аналитических баз данных и умение их настраивать и поддерживать.

Лабораторная работа №3. Создание БД для аналитики

Данная лабораторная состоит из 2 взаимосвязанных частей и является продолжением лабораторной работы №2. В каждой разделе есть теоретическая и практическая части лабораторной работы. В конце каждого раздела есть задания для выполнения в рамках лабораторной работы.

Цель:

- Сформировать у студента понимание особенностей создания аналитических баз данных и умение их настраивать и поддерживать.

Задачи:

- Получить теоретические знания денормализации.
- Узнать о основных методах денормализации.
- Ознакомится с массивами.
- Научится (изменять\добавлять\удалять) данные в массиве с помощью встроенных операций.
- Более подробно узнать о типе данных JSON.
- Узнать о ролях и пользователях.
- Научиться пользоваться командами для того, чтобы (определять\отзывать) доступ к данным, GRANT и REVOKE.
- Научится упрощать запросы к БД с помощью представлений.

Часть 1. Проектирование схемы базы данных

1.1 Теоретическая часть и практические примеры

Денормализация

Иногда после нормализации отношений проводят их денормализацию. Обоснованием денормализации может служить требование обеспечения определённой производительности для критических запросов. В нормализованной БД одна сущность просто разбивается на несколько отношений, и для получения исходного отношения требуется выполнить операцию соединения. Эта операция занимает много времени, поэтому нормализация может привести к падению производительности БД. Денормализация бывает нескольких видов:

- Восходящая. Подразумевает перенос некоторой информации из подчинённого отношения в родительское. Например, для схемы ЗАКАЗЫ > СТРОКИ ЗАКАЗОВ обычно нужно знать общую сумму заказа, которая является вычисляемой величиной. Если эти вычисления производятся часто, то для повышения эффективности можно добавить в отношение ЗАКАЗЫ поле Общая сумма. При этом возникает проблема обеспечения актуальности значения этого поля: пересчёт общей суммы при добавлении новой строки заказа, при удалении и при модификации позиций заказа. Обычно эта проблема решается программно (в приложении) или с помощью триггеров БД.
- Нисходящая. В этом случае информация переносится из родительского отношения в подчинённое. Например, в схеме ДОЛЖНОСТИ > СОТРУДНИКИ можно в качестве внешнего ключа использовать поле Название должности, а не идентификатор. Для этого название в отношении ДОЛЖНОСТИ должно быть определено как UNIQUE. Это позволяет избежать соединения отношений при запросе должности сотрудника.
- Разбиение одного отношения на два. В одно отношение помещаются все атрибуты сущности, которые связаны с экземпляром сущности как 1:1. Но

бывает так, что запись имеет большую длину за счёт наличия атрибутов большого объёма (графические данные, текстовые описания и проч.). Если данные этих атрибутов редко используются, то можно выделить в отдельное отношение атрибуты большого объёма. Для связи с исходным отношением вводится уникальный внешний ключ. А для получения исходного отношения создаётся представление (view), которое является соединением двух полученных отношений.

1.1.1 Методы денормализации

Добавление избыточных столбцов

В этом методе в основную таблицу добавляется только избыточный столбец, который часто используется в объединениях. Другая таблица остается как есть.

Добавление произвольных столбцов

Предположим, у нас есть таблица СТУДЕНТ с данными студента, такими как его идентификатор, имя, адрес и курс. Другая таблица ОТМЕЧАЕТ его внутренние оценки по разным предметам. Необходимо создать отчет для отдельного ученика, в котором должны быть его данные, общие оценки и оценка. В этом случае мы должны запросить таблицу STUDENT, а затем присоединиться к таблице MARKS, чтобы вычислить общее количество оценок по разным предметам. Основываясь на итоговой сумме, мы также должны определить оценку в запросе выбора. Затем это должно быть напечатано в отчете.

```
SELECT std.STD_ID, std.NAME, std.ADDRESS, t.TOTAL,
CASE WHEN t.TOTAL >=80 THEN ‘A’
    WHEN t.TOTAL>= 60 AND t.TOTAL
```

Вышеупомянутый запрос будет выполняться для каждой записи об учащемся для расчета общей суммы и оценки. Представьте, сколько студентов будет существовать и сколько раз этот запрос будет извлекать данные и

производить вычисления? А что, если у нас есть общий итог и оценка, хранящиеся в самой таблице STUDENT? Это сократит время соединения и время расчета. После того, как все оценки будут вставлены в таблицу MARKS, мы можем вычислить общую сумму и GRADE для каждого студента и обновить таблицу STUDENT для этих столбцов (у нас может быть триггер на MARKS для обновления таблицы STUDENT после вставки оценок). Теперь, если нам нужно сгенерировать отчет, просто запустите запрос SELECT для таблицы STUDENT и распечатайте его в отчете.

```
SELECT std.STD_ID, std.NAME, std.ADDRESS, std.TOTAL, std.GRADE
FROM STUDENT std;
```

Сворачивание таблиц

В этом методе часто используемые таблицы объединяются в одну, чтобы уменьшить количество объединений между таблицами. Таким образом, увеличивается производительность поискового запроса. Объединение избыточного столбца в одну таблицу может вызвать избыточность в таблице. Но он игнорируется, поскольку не влияет на значение других записей в таблице.

Снимки

Это один из первых методов создания избыточности данных. В этом методе таблицы базы данных дублируются и хранятся на различных серверах баз данных. Они обновляются в определенные периоды времени, чтобы поддерживать согласованность между таблицами сервера базы данных. Используя этот метод, пользователи, находящиеся в разных местах, могли получить доступ к серверам, которые находятся ближе к ним, и, следовательно, быстро получить данные. В этом случае им не нужно обращаться к таблицам, расположенным на удаленных серверах. Это помогает ускорить доступ.

ВАРРЕИ

В этом методе таблицы создаются как таблицы VARRAY, где повторяющиеся группы столбцов хранятся в одной таблице. Этот метод VARRAY отменяет условие 1NF. Согласно 1NF, каждое значение столбца должно быть атомарное. Но этот метод позволяет хранить одни и те же данные в разных столбцах для каждой записи.

Материализованные представления

Материализованные представления аналогичны таблицам, в которых все столбцы и производные значения предварительно вычисляются и сохраняются. Следовательно, если есть какой-либо запрос с таким же запросом, который используется в материализованном представлении, то запрос будет заменен этим материализованным представлением. Поскольку в этом представлении есть все столбцы в результате объединения и предварительно рассчитанного значения, нет необходимости повторно вычислять значения. Следовательно, это сокращает время, затрачиваемое на запрос.

Массивы

PostgreSQL позволяет создавать в таблицах такие столбцы, в которых будут содержаться не скалярные значения, а массивы переменной длины. Эти массивы могут быть многомерными и могут содержать значения любого из встроенных типов, а также типов данных, определенных пользователем. Предположим, что нам необходимо сформировать и сохранить в базе данных в удобной форме графики работы пилотов авиакомпании, т. е. номера дней недели, когда они совершают полеты.

Используя полученные ранее навыки создадим таблицу, в которой эти графики будут храниться в виде единых списков, т. е. в виде одномерных массивов.

```
CREATE TABLE pilots
(
	pilot_name text,
	schedule integer[]
);
```

Для указания на то, что это массив, нужно добавить квадратные скобки к наименованию типа данных. При этом задавать число элементов не обязательно. Заполним таблицу четырьмя записями.

Массив в команде вставки представлен в виде строкового литерала с указанием типа данных и квадратных скобок, означающих массив. Обратите внимание, что все массивы имеют различное число элементов.

```
INSERT INTO pilots
VALUES
( 'Ivan', '{ 1, 3, 5, 6, 7 }'::integer[] ),
( 'Petr', '{ 1, 2, 5, 7 }'::integer[] ),
( 'Pavel', '{ 2, 5 }'::integer[] ),
( 'Boris', '{ 3, 5, 6 }'::integer[] );
```

Предположим, что руководство компании решило, что каждый пилот должен летать 4 раза в неделю. Значит, нам придется обновить значения в таблице. Пилоту по имени Boris добавим один день с помощью операции конкатенации:

```
UPDATE pilots SET schedule = schedule || 7 WHERE pilot_name = 'Boris';
```

Пилоту по имени Pavel добавим один день в конец списка (массива) с помощью функции array_append:

```
UPDATE pilots SET schedule = array_append( schedule, 6 ) WHERE pilot_name = 'Pavel';
```

Ему же добавим один день в начало списка с помощью функции array_prepend (обратите внимание, что параметры функции поменялись местами):

```
UPDATE pilots SET schedule = array_prepend( 1, schedule ) WHERE pilot_name = 'Pavel';
```

У пилота по имени Ivan имеется лишний день в графике. С помощью функции array_remove удалим из графика пятницу (второй параметр функции указывает значение элемента массива, а не индекс):

```
UPDATE pilots SET schedule = array_remove( schedule, 5 ) WHERE pilot_name = 'Ivan';
```

У пилота по имени Petr изменим дни полетов, не изменяя их общего количества. Воспользуемся индексами для работы на уровне отдельных элементов массива. По умолчанию нумерация индексов начинается с единицы, а не с нуля. При необходимости ее можно изменить. К элементам одного и того же массива можно обращаться в предложении SET по отдельности, как будто это разные столбцы.

```
UPDATE pilots SET schedule[ 1 ] = 2, schedule[ 2 ] = 3 WHERE pilot_name = 'Petr';
```

А можно было бы, используя срез (slice) массива, сделать и так:

```
UPDATE pilots SET schedule[ 1:2 ] = ARRAY[ 2, 3 ] WHERE pilot_name = 'Petr';
```

Теперь продемонстрируем основные операции, которые можно применять к массивам, выполняя выборки из таблиц. Получим список пилотов, летающих по средам:

```
SELECT * FROM pilots WHERE array_position( schedule, 3 ) IS NOT NULL;
```

Функция array_position возвращает индекс первого вхождения элемента с указанным значением в массив. Если же такого элемента нет, она возвратит NULL. Выберем пилотов, летающих по понедельникам и воскресеньям:

```
SELECT * FROM pilots WHERE schedule @> '{ 1, 7 }'::integer[];
```

Оператор @> означает проверку того факта, что в левом массиве содержатся все элементы правого массива. Конечно, при этом в левом массиве могут находиться и другие элементы, что мы и видим в графике этого пилота. Еще аналогичный вопрос: кто летает по вторникам и/или по пятницам? Для

получения ответа воспользуемся оператором &&, который проверяет наличие общих элементов у массивов, т. е. пересекаются ли их множества значений. В нашем примере число общих элементов, если они есть, может быть равно одному или двум. Здесь мы использовали нотацию с ключевым словом ARRAY, а не '{2, 5}'::integer[]. Так же можно применять ту, которая принята в рамках выполнения вашего проекта.

```
SELECT * FROM pilots WHERE schedule && ARRAY[ 2, 5 ];
```

Сформулируем вопрос в форме отрицания: кто не летает ни во вторник, ни в пятницу? Для получения ответа добавим в предыдущую SQL-команду отрицание NOT:

```
SELECT * FROM pilots WHERE NOT ( schedule && ARRAY[ 2, 5 ] );
```

Иногда требуется развернуть массив в виде столбца таблицы. В таком случае поможет функция unnest:

```
SELECT unnest( schedule ) AS days_of_week FROM pilots WHERE pilot_name =
'Ivan';
```

Тип данных JSON

В предыдущих лабораторных работах в таблицах встречался такой формат данных как JSON, собственно, настало время более подробно о нём поговорить. Типы JSON предназначены для сохранения в столбцах таблиц базы данных таких значений, которые представлены в формате JSON (JavaScript Object Notation). Существует два типа: json и jsonb. Основное различие между ними заключается в быстродействии. Если столбец имеет тип json, тогда сохранение значений происходит быстрее, потому что они записываются в том виде, в котором были введены. Но при последующем использовании этих значений в качестве операндов или параметров функций будет каждый раз выполняться их разбор, что замедляет работу. При использовании типа jsonb разбор производится однократно, при записи значения в таблицу. Это несколько замедляет операции вставки строк, в

которых содержатся значения данного типа. Но все последующие обращения к сохраненным значениям выполняются быстрее, т. к. выполнять их разбор уже не требуется.

Есть еще ряд отличий, в частности, тип json сохраняет порядок следования ключей в объектах и повторяющиеся значения ключей, а тип jsonb этого не делает. Рекомендуется в приложениях использовать тип jsonb, если только нет каких-то особых аргументов в пользу выбора типа json.

Для демонстрации работы с этим типом данных создадим новую таблицу со спортивными увлечениями пилотов из предыдущего блока про массивы:

```
CREATE TABLE pilot_hobbies ( pilot_name text, hobbies jsonb );
```

И заполним её некоторыми записями.

```
INSERT INTO pilot_hobbies
VALUES
( 'Ivan', '{ "sports": [ "футбол", "плавание" ], "home_lib": true, "trips": 3 }'::jsonb
),
( 'Petr', '{ "sports": [ "теннис", "плавание" ], "home_lib": true, "trips": 2 }'::jsonb ),
( 'Pavel', '{ "sports": [ "плавание" ], "home_lib": false, "trips": 4 }'::jsonb ),
( 'Boris', '{ "sports": [ "футбол", "плавание", "теннис" ], "home_lib": true, "trips":
0 }'::jsonb );
```

Предположим, что нужно сформировать футбольную сборную команду нашей авиакомпании для участия в турнире. Мы можем выбрать всех футболистов таким способом:

```
SELECT * FROM pilot_hobbies WHERE hobbies @> '{ "sports": [ "футбол" ]
}'::jsonb;
```

Можно было эту задачу решить и таким способом, в этом решении мы выводим только информацию о спортивных предпочтениях пилотов.

Внимательно посмотрите, как используются одинарные и двойные кавычки. Операция -> служит для обращения к конкретному ключу JSON-объекта:

```
SELECT pilot_name, hobbies->'sports' AS sports FROM pilot_hobbies WHERE hobbies->'sports' @> '[ "футбол" ]'::jsonb;
```

При создании столбца с типом данных json или jsonb не требуется задавать структуру объектов, т. е. конкретные имена ключей. Поэтому в принципе возможна ситуация, когда в разных строках в JSON-объектах будут использоваться различные наборы ключей. В нашем примере структуры JSON-объектов во всех строках совпадают. А если бы они не совпадали, то как можно было бы проверить наличие ключа? Продемонстрируем это.

```
SELECT count( * ) FROM pilot_hobbies WHERE hobbies ? 'sport';
```

А вот ключ sports присутствует. Выполним ту же проверку:

```
SELECT count( * ) FROM pilot_hobbies WHERE hobbies ? 'sports';
```

А как выполнять обновление JSON-объектов в строках таблицы? Предположим, что пилот по имени Boris решил посвятить себя только хоккею. Тогда в базе данных мы выполним такую операцию:

```
UPDATE pilot_hobbies SET hobbies = hobbies || '{ "sports": [ "хоккей" ] }' WHERE pilot_name = 'Boris';
```

Если впоследствии Boris захочет возобновить занятия футболом, то с помощью функции jsonb_set можно будет обновить сведения о нем в таблице:

```
UPDATE pilot_hobbies SET hobbies = jsonb_set( hobbies, '{ sports, 1 }', '"футбол"' ) WHERE pilot_name = 'Boris';
```

Второй параметр функции указывает путь в пределах JSON-объекта, куда нужно добавить новое значение. В данном случае этот путь состоит из имени ключа (sports) и номера добавляемого элемента в массиве видов спорта (номер 1). Нумерация элементов начинается с нуля. Третий параметр имеет тип jsonb, поэтому его литерал заключается в одинарные кавычки, а само

добавляемое значение берется в двойные кавычки. В результате получается — '"футбол"':

1.2 Практическое задание для первой части ЛР

Задание №1 связано с проектированием схемы базы данных для аналитики. Будем исходить из того, что приложение, для которого была сделана база данных в лабораторной работе №2, стало очень популярным и по нему каждый день можно собирать большой объем статистической информации. Что это будет за статистика? Почему именно ее необходимо собирать, обрабатывать и анализировать? Задачей студента является ответить на эти вопросы, и, исходя из этого, изменить разработанную БД (допускается разработка новой БД, но тематика остается та же) и дозаполнить ее необходимыми атрибутами и данными. Результатом данного задания является схема базы данных, скрипты доработки базы данных и ее заполнения, обладающие следующими свойствами:

- Как минимум одна таблица должна содержать не меньше 100 млн. записей, которые со временем теряют актуальность.
- Другая таблица, связанная с первой, должна содержать не меньше 1 млн. записей.
- В одной из таблиц с количество записей больше 1 млн. должна быть колонка с текстом, по которой будет необходимо настроить полнотекстовый поиск.
- В одной из таблиц с количество записей больше 1 млн. должна быть колонка с данными в json-формате.
- В одной из таблиц с количество записей больше 1 млн. должна быть колонка с массивом.

При выполнении задания важно учитывать плюсы и минусы денормализации схемы данных и использования массивов и json-формата. При сдаче задания студент должен обосновать соответствие созданной схемы поставленной задаче. Для проектирования схемы и построения диаграммы

можно использовать любые средства, в том числе те, которые были описаны в лабораторной №1.

Часть 2. Создание и заполнения таблиц

2.1 Теоретическая часть и практические примеры

2.1.1 Роли и пользователи

Для того чтобы упростить процесс управления доступом, многие СУБД предоставляют возможность объединять пользователей в группы или определять роли. Роль – это совокупность привилегий, предоставляемых пользователю и/или другим ролям. Такой подход позволяет предоставить конкретному пользователю определённую роль или отнести его к определённой группе пользователей, обладающей набором прав в соответствии с задачами, которые на неё возложены. Кроме привилегий на доступ к объектам СУБД ещё может поддерживать так называемые системные привилегии: это права пользователя на создание/изменение/удаление (create/alter/drop) объектов различных типов. В некоторых системах такими привилегиями обладают только пользователи, включённые в группу АБД. Другие СУБД предоставляют возможность назначения дифференцированных системных привилегий любому пользователю в случае такой необходимости.

Первый уровень любого проекта авторизации в PostgreSQL — это роли. Роли базы данных могут представлять пользователей и/или группы. Поначалу в PostgreSQL обычно одна роль суперпользователя с именем «postgres». Для того чтобы проверить роль и подключённого к БД пользователя существуют две функции current_user и session_user.

Функция current_user (роль, или же текущий пользователь) возвращает идентификатор пользователя, по которому будут проверяться его права.

Функция session_user (пользователь сеанса БД) обычно возвращает имя пользователя, установившего текущее соединение с базой данных, но суперпользователи могут изменить это имя, выполнив команду SET SESSION AUTHORIZATION.

Ещё есть псевдороль public, она не видна, но про неё следует знать. Это групповая роль, в которую включены все остальные роли. Это означает, что

все роли по умолчанию будут иметь привилегии наследуемые от public. Поэтому иногда у public отбирают некоторые привилегии, чтобы отнять их у всех пользователей.

Собственно, и попробуем выполнить данную команду, но перед этим нужно создать новую роль «pilot»:

```
CREATE ROLE pilot LOGIN;
```

Где используется ключевое слово CREATE ROLE (противоположное значение является DROP ROLE) и LOGIN, означают соответственно дать знать транслятору что нужно создать (роль), и что нам нужно дать разрешение подключения данной роли («pilot») к нашей БД (противоположное значение является NOLOGIN).

Теперь можно и переключится на ново созданную роль:

```
SET ROLE pilot;
```

Для того чтобы проверить всё ли прошло по плану впишем команду для проверки ролей и пользователей:

```
SELECT current_user, session_user;
```

Так же можно поменять и пользователя БД:

```
SET SESSION AUTHORIZATION pilot;
```

2.1.2 Директивы GRANT и REVOKE. Привилегии.

Права доступа на работу с объектами базы данных контролируются привилегиями, управляемыми с помощью команд GRANT и REVOKE.

С точки зрения управления доступом роли можно разбить на несколько групп:

- Суперпользователи - полный доступ ко всем объектам - проверки не выполняются;

- Владельцы - владельцем становиться тот, кто создал объект. Но право владения можно передать. Владелец имеет все привилегии на принадлежащий ему объект;
- Остальные роли - доступ только в рамках выданных привилегий на определённый объект. Такие привилегии могут выдать владельцы на свои объекты. Или может выдать суперпользователь на любой другой объект.

Чтобы проверить привилегии запустим консоль (psql shell) и напишем команду «\dp». Получаем данные вида как на рисунке 1.

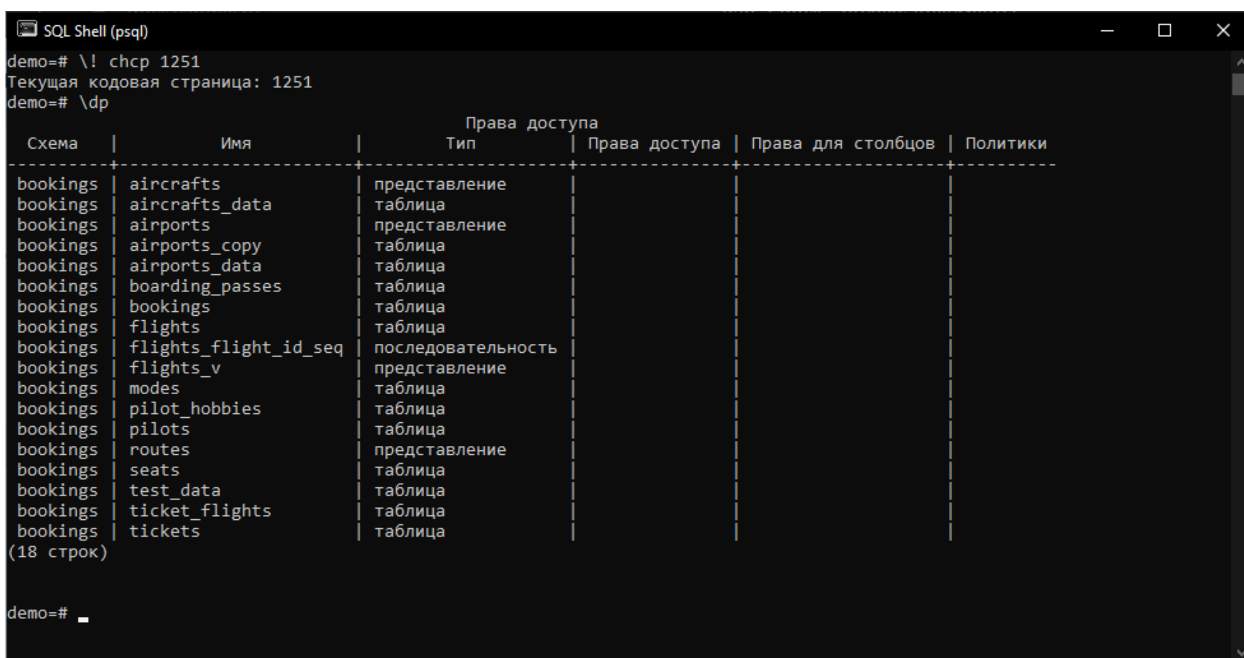

Рисунок 1 – Права доступа

В столбцах привилегий ничего нет, поэтому над этими объектами базы данных (таблицами и последовательностями) не допускаются никакие действия, если они не выполняются суперпользователем или владельцем объекта базы данных.

Перечисляя таблицы с помощью «\dt», мы видим, что «postgres» является владельцем всех таблиц, рисунок 2.

```
SQL Shell (psql)
demo=# \dt
              Список отношений
  Схема   |      Имя       |   Тип   | Владелец
----------+----------------+---------+----------
 bookings | aircrafts_data  | таблица | postgres
 bookings | airports_copy   | таблица | postgres
 bookings | airports_data   | таблица | postgres
 bookings | boarding_passes | таблица | postgres
 bookings | bookings        | таблица | postgres
 bookings | flights         | таблица | postgres
 bookings | modes           | таблица | postgres
 bookings | pilot_hobbies   | таблица | postgres
 bookings | pilots          | таблица | postgres
 bookings | seats           | таблица | postgres
 bookings | test_data       | таблица | postgres
 bookings | ticket_flights  | таблица | postgres
 bookings | tickets         | таблица | postgres
(13 строк)


demo=#
```

Рисунок 2 – Владелицы таблиц

Давайте разрешим роли «pilot» делать выборку из таблицы с пилотами, а затем снова проверим привилегии, где «GRANT» это ключевое слово для добавление привилегий «SELECT» это то, что стоит разрешить для пользователя. Далее поле «ON» пишем название таблицы и после «TO» имя пользователя (Всё тоже самое, но с использованием «REVOKE» и место «TO» – «FROM» приводит к удалению привилегии):

```
GRANT SELECT ON pilots TO pilot;
-- (REVOKE SELECT ON pilots FROM pilot) удаление
```

Результат работы данного кода, можно наблюдать в обновленной таблице на рисунке 3.

```
SQL Shell (psql)
demo=# GRANT SELECT ON pilots TO pilot;
GRANT
demo=# \dp
                                              Права доступа
  Схема   |         Имя          |       Тип        |        Права доступа         | Права для столбцов | Политики
----------+----------------------+------------------+------------------------------+--------------------+----------
 bookings | aircrafts            | представление    |                              |                    |
 bookings | aircrafts_data       | таблица          |                              |                    |
 bookings | airports             | представление    |                              |                    |
 bookings | airports_copy        | таблица          |                              |                    |
 bookings | airports_data        | таблица          |                              |                    |
 bookings | boarding_passes      | таблица          |                              |                    |
 bookings | bookings             | таблица          |                              |                    |
 bookings | flights              | таблица          |                              |                    |
 bookings | flights_flight_id_seq | последовательность |                            |                    |
 bookings | flights_v            | представление    |                              |                    |
 bookings | modes                | таблица          |                              |                    |
 bookings | pilot_hobbies        | таблица          |                              |                    |
 bookings | pilots               | таблица          | postgres=arwdDxt/postgres+   |                    |
          |                      |                  | pilot=r/postgres             |                    |
 bookings | routes               | представление    |                              |                    |
 bookings | seats                | таблица          |                              |                    |
 bookings | test_data            | таблица          |                              |                    |
 bookings | ticket_flights       | таблица          |                              |                    |
 bookings | tickets              | таблица          |                              |                    |
(18 строк)
```

Рисунок 3 – Обновленные привилегии

Теперь мы видим некоторые привилегии. Привилегия select (read), которую мы предоставили роли «pilot» из роли «postgres», появляется вместе с полным набором возможных привилегий, которые имплицитно предоставляются владельцу таблицы (postgres). Что означают все эти буквы? В документации PostgreSQL на рисунке 4.

Table 5.1. ACL Privilege Abbreviations

Privilege	Abbreviation	Applicable Object Types
SELECT	r (“read”)	LARGE OBJECT, SEQUENCE, TABLE (and table-like objects), table column
INSERT	a (“append”)	TABLE, table column
UPDATE	w (“write”)	LARGE OBJECT, SEQUENCE, TABLE, table column
DELETE	d	TABLE
TRUNCATE	D	TABLE
REFERENCES	x	TABLE, table column
TRIGGER	t	TABLE
CREATE	C	DATABASE, SCHEMA, TABLESPACE
CONNECT	c	DATABASE
TEMPORARY	T	DATABASE
EXECUTE	X	FUNCTION, PROCEDURE
USAGE	U	DOMAIN, FOREIGN DATA WRAPPER, FOREIGN SERVER, LANGUAGE, SCHEMA, SEQUENCE, TYPE

Рисунок 4 – Таблица привилегий

2.1.3 Представления

При работе с базами данных зачастую приходится многократно выполнять одни и те же запросы, которые могут быть весьма сложными и требовать обращения к нескольким таблицам. Чтобы избежать необходимости многократного формирования таких запросов, можно использовать так называемые представления (views). Если речь идет о выборке данных, то представления практически неотличимы от таблиц с точки зрения обращения к ним в командах SELECT.

Упрощенный синтаксис команды CREATE VIEW, предназначенной для создания представлений, таков:

```
CREATE VIEW имя-представления [ ( имя-столбца [, ...] ) ] AS запрос;
```

Давайте создадим простое представление. За основу возьмем запрос для подсчета количества мест в салонах для всех моделей самолетов с учетом класса обслуживания:

```
SELECT aircraft_code,
       fare_conditions,
       count( * )
FROM seats
GROUP BY aircraft_code, fare_conditions
ORDER BY aircraft_code, fare_conditions;
```

По данному запросу создадим представление и дадим ему имя, отражающее суть этого представления:

```
CREATE VIEW seats_by_fare_cond AS
SELECT aircraft_code,
       fare_conditions,
       count( * )
FROM seats
GROUP BY aircraft_code, fare_conditions
ORDER BY aircraft_code, fare_conditions;
```

Теперь мы можем вместо написания сложного первоначального запроса обращаться непосредственно к представлению, как будто это обычная таблица:

```
SELECT * FROM seats_by_fare_cond;
```

СУБД PostgreSQL предлагает свое расширение команды CREATE VIEW, а именно – фразу OR REPLACE. Если представление уже существует, то можно его не удалять, а просто заменить новой версией. Однако нужно помнить о том, что при создании новой версии представления (без явного удаления старой с помощью команды DROP VIEW) должны оставаться неизменными имена столбцов представления. Если же вы хотите изменить имя хотя бы одного столбца, то сначала нужно удалить представление с помощью команды DROP VIEW, а уже затем создать его заново.

Имена столбцов можно явно указать в команде, но если они не указаны, то СУБД сама «вычислит» эти имена. В только что созданном нами представлении третий столбец получит имя count. Если мы захотим изменить это имя, то возможны два способа: первый заключается в том, чтобы создать псевдоним для этого столбца с помощью ключевого слова AS, а второй – в указании списка имен столбцов в начале команды CREATE VIEW.

Если захотим в представления изменить имя столбца, а не просто изменить логику работы в представлении, нам придется сначала удалить это представление, а затем создать его заново:

```
DROP VIEW seats_by_fare_cond;
 CREATE OR REPLACE VIEW seats_by_fare_cond AS
SELECT aircraft_code,
        fare_conditions,
       count( * ) AS num_seats
FROM seats
GROUP BY aircraft_code, fare_conditions
ORDER BY aircraft_code, fare_conditions;
```

Есть второй способ задания имен столбцов в представлении — с помощью списка их имен, заключенного в скобки:

```
DROP VIEW seats_by_fare_cond;
 CREATE OR REPLACE VIEW seats_by_fare_cond
( code, fare_cond, num_seats ) AS
SELECT aircraft_code,
        fare_conditions,
       count( * ) AS num_seats
FROM seats
GROUP BY aircraft_code, fare_conditions
ORDER BY aircraft_code, fare_conditions;
```

Бывают ситуации, когда заранее известно, что возможна попытка удаления несуществующего представления. В таких случаях обычно стараются избежать ненужных сообщений об ошибке отсутствия представления. Для этого в команду DROP VIEW добавляют фразу IF EXISTS.

В базе данных «demo» создано представление «Рейсы» (flights_v), сконструированное на основе таблицы «Рейсы» (flights), на нём и воспользуемся IF EXISTS:

```
DROP VIEW IF EXISTS flights_v;
```

2.2 Практическое задание для второй части ЛР

Целью седьмого практического задания является освоение работы с представлениями и другими способами управления доступом. При выполнении задания необходимо:

- Создать пользователя test и выдать ему доступ к базе данных.
- Составить и выполнить скрипты присвоения новому пользователю прав доступа к таблицам, созданным в практическом задании 1. При этом права доступа к различным таблицам должны быть различными, а именно:
    - По крайней мере, для одной таблицы новому пользователю присваиваются права SELECT, INSERT, UPDATE в полном объеме.

    - По крайней мере, для одной таблицы новому пользователю присваиваются права SELECT и UPDATE только избранных столбцов.
    - По крайней мере, для одной таблицы новому пользователю присваивается только право SELECT.

- Создать стандартную роль уровня базы данных, присвоить ей право доступа (UPDATE на некоторые столбцы) к представлению, созданному в практическом задании №3.3, назначить новому пользователю созданную роль.
- Выполнить от имени нового пользователя некоторые выборки из таблиц и представления. Убедиться в правильности контроля прав доступа.
- Выполнить от имени нового пользователя операторы изменения таблиц с ограниченными правами доступа. Убедиться в правильности контроля прав доступа.
- Составить SQL-скрипты для создания нескольких представлений, которые позволяли бы упростить манипуляции с данными или позволяли бы ограничить доступ к данным, предоставляя только необходимую информацию.

3 Темы для самостоятельной проработки

- Денормализация

https://habr.com/ru/company/latera/blog/281262/

https://habr.com/ru/post/64524/

- Массивы

https://postgrespro.ru/docs/postgrespro/11/arrays

https://postgrespro.ru/docs/postgrespro/11/functions-array

- Json

https://postgrespro.ru/docs/postgrespro/11/datatype-json

https://postgrespro.ru/docs/postgrespro/11/functions-json

- Наполнение базы данных

https://postgrespro.ru/docs/postgrespro/11/populate

- Роли и пользователи.

https://postgrespro.ru/docs/postgresql/11/user-manag

https://postgrespro.ru/docs/postgrespro/11/app-createuser

- Директивы GRANT и REVOKE.

https://postgrespro.ru/docs/postgrespro/11/ddl-priv

https://postgrespro.ru/docs/postgrespro/11/ddl-schemas#DDL-SCHEMAS-PRIV

- Представления.

https://postgrespro.ru/docs/postgrespro/11/sql-createview

https://postgrespro.ru/docs/postgrespro/11/rules-views

https://postgrespro.ru/docs/postgrespro/11/rules-materializedviews

- Полное описание синтаксиса встретившихся команд

https://postgrespro.ru/docs/postgrespro/11/sql-commands

3 Вопросы для самостоятельного изучения

- Для чего нужна денормализация?
- Виды денормализации.
- Какими методами можно реализовать денормализацию?
- В чем преимущество использования массивов?
- Разница между json и jsonb.
- Для чего нужны роли?
- Что такое схема?
- Рассказать про директивы GRANT и REVOKE.
- Для чего нужна роль PUBLIC?
- Как добавить нового пользователя в текущую базу данных?
- Как позволить пользователю заходить на сервер?
- Какие существуют права?
- Исправить ошибки в обязательной части.
- Сменить владельца базы данных.
- Сменить пароль для пользователя.
- Определить роль с заданными правами.
- Объяснить, как работают написанные запросы.
- Рассказать о CHECK OPTION.
- Рассказать о модификации данных через представления.
- Рассказать о вставке данных через представления.
- Примеры вопросов по оператору SELECT см. в задании №1.
- Исправить неверно работающий запрос (запросы).
- Упростить один или несколько запросов.
- Продемонстрировать изменение и вставку данных через представления.
- Написать или модифицировать запрос по сформулированному заданию.
- Продемонстрировать полезность материализованного представления.

5 СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Карпова И.П. Базы данных. Учебное пособие. – Московский государственный институт электроники и математики (Технический университет): учебное пособие / И.П. Карпова; – М., 2009. – 140-141 с, 102 с.

2. PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова. — СПб.: БХВ-Петербург, 2018. — 65-68 с, 68-72 с.

3. Tutorialcup [Электронный ресурс] – Режим доступа: URL: https://ru.tutorialcup.com/dbms/denormalization.htm#Methods_of_De-normalization

4. Habr [Электронный ресурс] – Режим доступа: URL: https://habr.com/ru/company/timeweb/blog/661771/