

МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Омский государственный технический университет»

О. Б. Малков, М. П. Маркова, М. В. Девятерикова

РАБОТА С СУБД

PostgreSQL

Учебное текстовое электронное издание
локального распространения

*Рекомендовано редакционно-издательским советом
Омского государственного технического университета*

Омск
Издательство ОмГТУ
2023

УДК 004.65(075)
ББК 32.973.26-018.2я73
М19

Рецензенты:

B. A. Мещеряков, д-р техн. наук, профессор кафедры цифровых технологий
Сибирского государственного автомобильно-дорожного
университета (СибАДИ);

O. H. Лучко, профессор, зав. кафедрой прикладной информатики,
математики и естественно-научных дисциплин
Омской гуманитарной академии

Малков, О. Б. Работа с СУБД PostgreSQL : учеб. пособие / О. Б. Малков, М. П. Маркова, М. В. Девятерикова ; Минобрнауки России, Ом. гос. техн. ун-т. – Омск : Изд-во ОмГТУ, 2023. – 1 CD-ROM (4,88 Мб). – Систем. требования: процессор с частотой 800 МГц и выше ; 128 Мб RAM и более ; свободное место на жестком диске 300 Мб и более ; Linux / Windows XP и выше ; MacOS X 10.4 и выше ; CD/DVD-ROM-дисковод ; ПО для просмотра pdf- и mp4-файлов. – Загл. с титул. экрана. – ISBN 978-5-8149-3707-0.

Учебное пособие посвящено основам работы с СУБД PostgreSQL, которая в настоящее время широко используется для хранения данных веб-приложений, мобильных приложений, аналитических приложений и пр. Содержит лабораторный практикум, включающий 11 лабораторных работ.

Предназначено обучающимся по направлениям 09.03.01 «Информатика и вычислительная техника», 09.03.02 «Информационные системы и технологии», 09.03.03 «Прикладная информатика», 09.03.04 «Программная инженерия». Будет полезно студентам других направлений, осваивающим дисциплины «Базы данных» и «Системы управления базами данных».

Редактор *T. A. Москвитина*
Компьютерная верстка *L. Ю. Бутаковой*

*Для дизайна этикетки использованы материалы
из открытых интернет-источников*

Сводный темплан 2023 г.
Подписано к использованию 27.09.23.
Объем 4,88 Мб.

© ОмГТУ, 2023

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	5
1. УСТАНОВКА СУБД PostgreSQL	7
2. УПРАВЛЕНИЕ СЛУЖБОЙ И ОСНОВНЫЕ ФАЙЛЫ	10
3. ПОДКЛЮЧЕНИЕ К СЕРВЕРУ БАЗЫ ДАННЫХ	13
3.1. ПРОГРАММА PSQL (ИНТЕРАКТИВНЫЙ ТЕРМИНАЛ POSTGRESQL)	14
3.2. ПРОГРАММА PGADMIN	18
4. ПРОГРАММИРОВАНИЕ НА СТОРОНЕ СЕРВЕРА	26
4.1. ПРОЦЕДУРНЫЕ ЯЗЫКИ.....	26
4.2. ПРОЦЕДУРНЫЙ ЯЗЫК PL/PGSQL	28
4.3. СТРУКТУРА БЛОКА PL/PGSQL	29
4.4. СООБЩЕНИЯ И ОШИБКИ	31
4.5. ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ И КОНСТАНТ В PL/PGSQL	33
4.6. ФУНКЦИИ И ПРОЦЕДУРЫ.....	34
4.7. АНОНИМНЫЕ БЛОКИ	42
4.8. ОСНОВНЫЕ ОПЕРАТОРЫ ЯЗЫКА PL/PGSQL.....	44
4.9. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ	55
4.10. ПУСТОЙ ОПЕРАТОР.....	58
5. ЛАБОРАТОРНЫЙ ПРАКТИКУМ.....	59
ЛАБОРАТОРНАЯ РАБОТА № 1. ПРОЕКТИРОВАНИЕ БАЗЫ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ ER-ТЕХНОЛОГИИ	59
ЛАБОРАТОРНАЯ РАБОТА № 2. Создание новой базы данных в среде PostgreSQL	64
ЛАБОРАТОРНАЯ РАБОТА № 3. Создание и связывание таблиц базы данных в среде PostgreSQL	70
ЛАБОРАТОРНАЯ РАБОТА № 4. Вставка, удаление и обновление данных.....	81

ЛАБОРАТОРНАЯ РАБОТА № 5. РАБОТА С МАССИВАМИ И ТИПАМИ JSON	90
ЛАБОРАТОРНАЯ РАБОТА № 6. Создание запросов на выборку	100
ЛАБОРАТОРНАЯ РАБОТА № 7. Представления.....	118
ЛАБОРАТОРНАЯ РАБОТА № 8. Функции и процедуры	124
ЛАБОРАТОРНАЯ РАБОТА № 9. Транзакции	130
ЛАБОРАТОРНАЯ РАБОТА № 10. Курсоры	141
ЛАБОРАТОРНАЯ РАБОТА № 11. Триггеры	151
ЗАКЛЮЧЕНИЕ	160
БИБЛИОГРАФИЧЕСКИЙ СПИСОК	162
ПРИЛОЖЕНИЕ. Типы данных в PostgreSQL	164

ВВЕДЕНИЕ

Данное учебное пособие продолжает серию учебных пособий для студентов, изучающих в техническом вузе дисциплины «Базы данных» и «Системы управления базами данных». Ранее была рассмотрена работа с СУБД MS SQL Server Express [4] и Oracle Database Express [5]. Представленный материал посвящен работе PostgreSQL, которая на сегодняшний день является самой передовой в мире базой данных с открытым исходным кодом [13].

PostgreSQL – это свободно доступная и весьма стабильная объектно-реляционная система управления базами данных, предлагающая множество функций, таких как точность, целостность, отказоустойчивость и пр. PostgreSQL широко используется для хранения данных веб-приложений, мобильных приложений, аналитических приложений и т. д.

PostgreSQL основана на СУБД POSTGRES, Version 4.2, разработанной на факультете компьютерных наук Калифорнийского университета в Беркли. СУБД POSTGRES реализовала множество новшеств, которые появились в коммерческих СУБД гораздо позднее.

PostgreSQL поддерживает большую часть стандарта SQL и предлагает множество функций: сложные запросы, внешние ключи, триггеры, изменяемые представления, поддержка транзакций, многоверсионность и пр. Пользователи PostgreSQL могут расширять ее возможности, создавая свои типы данных, функции, операторы, агрегатные функции, методы индексирования, процедурные языки и т. д.

В пособии рассматривается создание и использование реляционной базы данных средствами PostgreSQL. Теоретическая часть пособия содержит общие сведения. Практическая часть включает 11 лабораторных работ, в каждой из которых приведен теоретический материал и пример выполнения работы. В пособии использован сквозной пример – база данных службы, оформляющей заказы на товары.

Как и в предшествующих учебных пособиях, приняты ограничения:

- пособие не является учебником по языку SQL. Синтаксис базовых конструкций языка SQL можно найти в литературе, например в [2, 6] и в Интернете;
- пособие не является учебником для администратора PostgreSQL, хотя студент, выполняющий лабораторные работы на своем компьютере, становится администратором этой СУБД и вынужден решать соответствующие задачи;
- пособие является первым шагом в изучении PostgreSQL. Ответы на более продвинутые вопросы можно найти в специальной литературе, например в [1, 7, 8].

В качестве клиентского приложения в пособии используется входящая в поставку PostgreSQL и являющаяся стандартом де-факто утилита pgAdmin. Она предоставляет мощный графический интерфейс, позволяющий решать задачи администрирования, отображать объекты баз данных и выполнять запросы на языке SQL.

1. УСТАНОВКА СУБД PostgreSQL

Для установки службы PostgreSQL используем обычный дистрибутив PostgreSQL 15. Система PostgreSQL может работать под управлением любой распространенной операционной системы практически на любых компьютерах. Действия, необходимые для установки детально описаны на сайте PostgreSQL: <https://www.postgresql.org/download> (рис. 1).

The screenshot shows the PostgreSQL download page. At the top, there's a navigation bar with links to Home, About, Download, Documentation, Community, Developers, Support, Donate, and Your account. Below the navigation bar, a banner displays the release date: "9th February 2023: PostgreSQL 15.2, 14.7, 13.10, 12.14, and 11.19 Released!". On the left, there's a "Quick Links" sidebar with options like Downloads (selected), Packages, Source, Software Catalogue, File Browser, and a link to the PostgreSQL logo. The main content area is titled "Downloads" with a downward arrow icon. It features a section titled "PostgreSQL Downloads" with a sub-section "Packages and Installers". A note states: "PostgreSQL is available for download as ready-to-use packages or installers for various platforms, as well as a source code archive if you want to build it yourself." Below this, there's a "Select your operating system family:" section with five buttons: Linux (with a penguin icon), macOS (with an apple icon), Windows (with a blue square icon), BSD (with a red circle icon), and Solaris (with a sun icon). The "Windows" button is highlighted.

Рис. 1. Стартовая страница

Выбрав Windows, переходим на следующую страницу (рис. 2). Здесь предлагается загрузить интерактивный установщик, сертифицированный EDB для всех поддерживаемых версий PostgreSQL. Этот установщик включает:

- сервер PostgreSQL;
- pgAdmin – графический инструмент для управления базами данных;
- Stack Builder – менеджер пакетов, который можно использовать для загрузки и установки дополнительных инструментов и драйверов PostgreSQL.

Stack Builder включает в себя управление, интеграцию, миграцию, репликацию, геопространственные данные, соединители и другие инструменты.

Щелкнув на ссылке *Download the installer*, откроем страницу рекомендуемых загрузок, показанную на рис. 3. Для загрузки выбираем версию PostgreSQL 15.2 для платформы Windows x86-64.

The screenshot shows the PostgreSQL website's navigation bar with links for Home, About, Download, Documentation, Community, Developers, Support, Donate, and Your account. Below the navigation bar is a banner stating "9th February 2023: PostgreSQL 15.2, 14.7, 13.10, 12.14, and 11.19 Released!". The main content area is titled "Windows installers" with a blue icon. A red arrow points from the left margin to the "Interactive installer by EDB" section. This section contains a link to download the installer, a note about EDB hosting, and a description of the installer's features.

Windows installers

Interactive installer by EDB

[Download the installer](#) certified by EDB for all supported PostgreSQL versions.

Note! This installer is hosted by EDB and not on the PostgreSQL community servers. If you have issues with the website it's hosted on, please contact webmaster@enterprisedb.com.

This installer includes the PostgreSQL server, pgAdmin; a graphical tool for managing and developing your databases, and StackBuilder; a package manager that can be used to download and install additional PostgreSQL tools and drivers. Stackbuilder includes management, integration, migration, replication, geospatial, connectors and other tools.

Рис. 2. Загрузка интерактивного установщика от EDB

Download PostgreSQL

Open source PostgreSQL packages and installers from EDB

PostgreSQL Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
15.2	postgresql.org	postgresql.org			Not supported
14.7	postgresql.org	postgresql.org			Not supported

Рис. 3. Выбор версии для загрузки

Скачиваем установщик *postgresql-15.2-1-windows-x64.exe* с сайта и запускаем его. Установщик построен в традиционном стиле «мастера»: вы можете просто нажимать на кнопку «Далее», если вас устраивают предложенные варианты. Оставьте все флажки, если не уверены, какие выбрать.

Действия выполняются в следующем порядке:

- выбирается каталог для установки PostgreSQL, по умолчанию установка выполняется в папку *C:\Program Files\PostgreSQL\15*;
- выбираются компоненты устанавливаемой программы (все – *PostgreSQL Server, pgAdmin 4, Stack Builder, Command Line Tools*);
- выбирается расположение каталога для хранения баз данных, по умолчанию предлагается папка *C:\Program Files\PostgreSQL\15\data*;
- задается и подтверждается пароль администратора;
- выбирается номер порта сервера (для PostgreSQL – порт 5432);
- если вы будете хранить данные на русском языке, выберите локаль «*Russian, Russia*» (или оставьте вариант «Настройка ОС», если в Windows русская локаль).

Проверьте все настройки перед инсталляцией и запустите установку. После установки PostgreSQL запускается мастер установки дополнительного программного обеспечения *Stack Builder*. Можно ограничиться только драйверами базы данных. Проверьте выбранные пакеты и установите для них каталог загрузки.

ВНИМАНИЕ!

Предлагаемый в данном учебном пособии лабораторный практикум не требует установки дополнительного программного обеспечения.

2. УПРАВЛЕНИЕ СЛУЖБОЙ И ОСНОВНЫЕ ФАЙЛЫ

База данных – совокупность данных, доступная для приложения через сервер базы данных. Приложение выступает в роли клиента и взаимодействует непосредственно с сервером базы данных.

Чтобы выполнять действия над данными в базе, приложение должно установить соединение с сервером. Различные клиенты (приложения) могут устанавливать соединение по-разному, но обычно для этого нужна следующая информация:

- сетевой адрес сервера базы данных (имя или IP-адрес);
- номер порта, через который производится соединение;
- имя базы данных;
- имя пользователя базы данных.

Значения этих параметров могут быть записаны в конфигурационных файлах клиента и не обязательно указываются явно, но они необходимы в любом случае.

Сетевой адрес сервера и порт определяют, куда передаются сообщения, адресованные серверу базы данных. Большинство СУБД имеет фиксированный предпочтительный номер порта, который используется, если администратор не изменит его при создании сервера. Для PostgreSQL это порт 5432. Использование другого номера порта необходимо, если на одном компьютере одновременно работает несколько серверов баз данных. Сервер и клиент могут работать на одном компьютере, однако для способа установки соединения это не имеет значения, потому что и в этом случае используется сетевой протокол.

Один сервер баз данных может обрабатывать запросы к нескольким базам данных. Совокупность баз данных, обслуживаемых одним сервером, в системе PostgreSQL называется *клUSTERом*. Некоторые объекты в системе PostgreSQL (например, пользователи) определены для всего кластера, но в рамках одного соединения клиентская программа может работать только с одной базой данных (ее имя указывается при установлении соединения).

В простых конфигурациях часто достаточно одной базы данных. В одной базе можно хранить данные нескольких независимых приложений. Это упрощает совместное использование данных разных приложений в рамках одного сеанса. В то же время широко распространена практика создания отдельных баз данных для каждого приложения, даже небольшого.

Понятие *пользователь* (User) имеет много различных значений. Следует различать пользователя операционных систем на компьютерах сервера и клиента, пользователя базы данных и пользователя приложения. Например, если приложение – интернет-магазин, то его пользователем становится любой покупатель, но такой пользователь не является пользователем базы данных.

В некоторых случаях пользователи операционной системы или домена могут быть зарегистрированы как пользователи базы данных. Но в любом случае эти сущности остаются различными. Понятие пользователя базы данных важно для разграничения доступа и защиты данных на уровне базы данных. Права доступа к базе данных и к отдельным объектам определяются для пользователей базы данных.

В процессе установки создается отдельная учетная запись суперпользователя (администратора) СУБД с именем *postgres*.

При установке PostgreSQL в системе регистрируется служба *postgresql-x64-15 – PostgreSQL Server 15*. Она запускается автоматически при старте компьютера как сетевая служба (Network Service). При необходимости можно изменить параметры этой службы с помощью стандартных средств Windows.

Если при запуске службы произошла ошибка, для поиска причины следует обратиться к журналу сообщений сервера. Он находится в подкаталоге *log* каталога установки (*C:\Program Files\PostgreSQL\15\data\log*). Журнал настроен так, чтобы запись периодически переключалась в новый файл. Найти актуальный файл можно по дате последнего изменения или по имени, которое содержит дату и время переключения. Вся информация,

которая содержится в базе данных, располагается в файловой системе в каталоге *C:\Program Files\PostgreSQL\15\data*.

Есть несколько важных конфигурационных файлов, определяющих настройки сервера. Они располагаются в каталоге баз данных. Их не нужно изменять при первом знакомстве с PostgreSQL, но в реальной работе они потребуются:

- *C:\Program Files\PostgreSQL\15\data\postgresql.conf* – основной конфигурационный файл, содержащий значения параметров сервера;
- *C:\Program Files\PostgreSQL\15\data\pg_hba.conf* – файл, определяющий настройки доступа.

В целях безопасности доступ должен быть подтвержден паролем и допускается только с локального компьютера и только под пользователем, имя которого совпадает с именем пользователя в операционной системе.

Как было сказано, при установке PostgreSQL в системе автоматически создается пользователь *postgres*, от имени которого работают процессы, обслуживающие сервер, и которому принадлежат все файлы, относящиеся к СУБД.

PostgreSQL будет автоматически запускаться при перезагрузке операционной системы. С настройками по умолчанию это не проблема: если вы не работаете с сервером базы данных, он потребляет мало ресурсов системы.

3. ПОДКЛЮЧЕНИЕ К СЕРВЕРУ БАЗЫ ДАННЫХ

Сеансы, транзакции и сообщения. Когда клиентское приложение подключается к серверу базы данных, открывается *сесия* (*сессия*). Сеанс – частный канал взаимодействия приложения с СУБД. На сервере можно запустить множество экземпляров утилит доступа к СУБД, при этом каждый экземпляр создаст свой сеанс, изолированный от других сессий [3].

Транзакции дополняют концепцию сеансов. Они разбивают операции в сессии на логические единицы и позволяют выполнять их поэтапно. Транзакция – одна или несколько команд SQL, которые либо успешно завершаются, либо отменяются как единое целое. Транзакции – это фундаментальное понятие во всех СУБД. Детали этого механизма отличаются в отдельных реализациях.

В PostgreSQL чтобы начать сеанс, клиент открывает подключение к серверу и передает стартовое сообщение. Сообщение содержит имя пользователя и имя базы данных, к которой клиент хочет подключиться, а также используемую версию протокола. Сервер определяет, можно ли разрешить это подключение и какая дополнительная проверка подлинности требуется. Затем сервер отправляет сообщение с запросом аутентификации, на которое клиент должен ответить сообщением, подтверждающим его подлинность (например, по паролю). Цикл аутентификации заканчивает сервер, запрещая соединение или принимая его.

Цикл простого запроса начинает клиент, передавая серверу сообщение *Query*. Это сообщение включает команду (команды) SQL в виде текстовой строки. В ответ сервер передает одно или несколько сообщений, в зависимости от запроса, и завершает цикл, сообщая клиенту, что тот может передавать новую команду.

Ответ на запрос *SELECT* (или другие запросы, возвращающие наборы строк) обычно состоит из *RowDescription*, нуля или нескольких сообщений *DataRow*. Так как строка запроса может содержать несколько запросов (разделенных точкой с запятой), до завершения обработки всей строки сервер может передать несколько серий таких ответов. Когда сервер за-

вершает обработку всей строки и готов принять следующую строку запроса, он выдает сообщение *ReadyForQuery*. В случае ошибки выдается *ErrorResponse* с последующим *ReadyForQuery*. Сообщение *ErrorResponse* прерывает дальнейшую обработку строки запроса, даже если в ней остались другие запросы.

Клиенты PostgreSQL. Для работы с базой данных необходимы программы-клиенты. Такими программами являются:

- универсальная утилита командной строки *psql*;
- pgAdmin – графическое средство администрирования PostgreSQL.

Содержательная обработка данных, извлеченных из базы, предполагает использование программ-клиентов, написанных на языках высокого уровня для решения конкретных прикладных задач.

3.1. ПРОГРАММА PSQL (ИНТЕРАКТИВНЫЙ ТЕРМИНАЛ POSTGRESQL)

Программа *psql* входит в состав любого комплекта PostgreSQL в любой операционной системе. Предоставляет интерфейс командной строки. Есть другие клиентские программы, предоставляющие более развитый экранный интерфейс, который позволяет быстро ориентироваться в структуре базы данных, но для использования сложных конструкций SQL в любом случае необходимо текстовое представление.

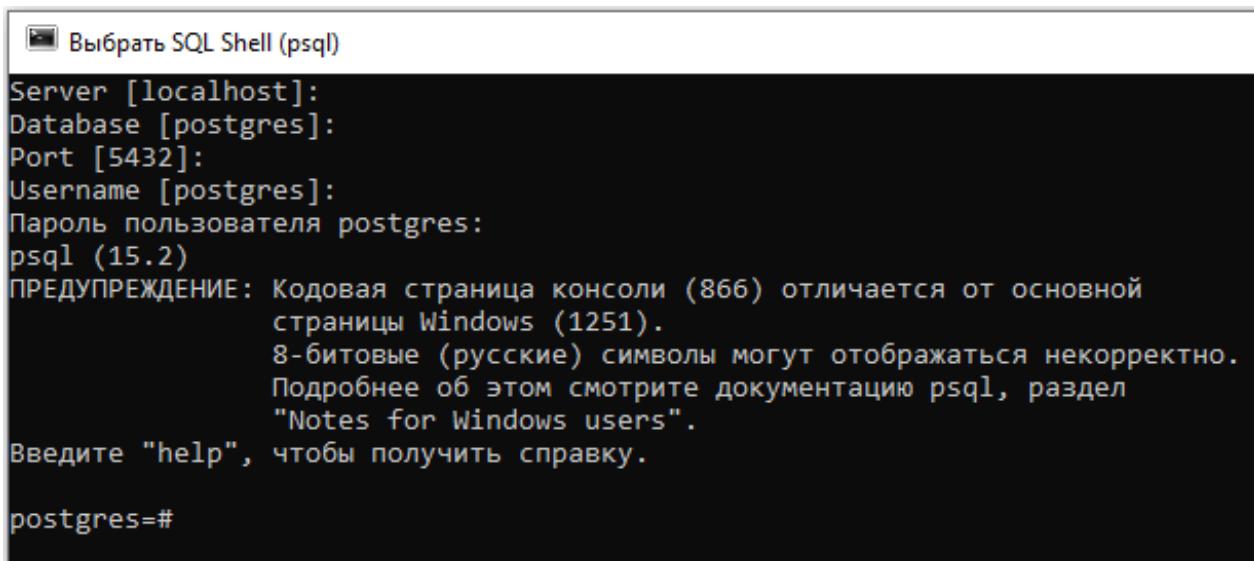
Для запуска программы *psql* нужно в командной строке ввести команду:

psql -d имя_базы_данных -U имя_пользователя -h сервер -p порт

Для соединения администратора с локальным сервером базы данных (работающим на том же компьютере, что и *psql*) можно использовать параметры, заданные по умолчанию. В Windows оболочка SQL Shell (*psql*) вызывается командой:

Пуск / PostgreSQL 15 / SQL Shell (psql)

В ответ на подсказки системы следует нажимать *<Enter>*. Требуется только ввести пароль администратора, заданный при установке (рис. 4). В некоторых комплектах эту программу можно запускать и другим способом.

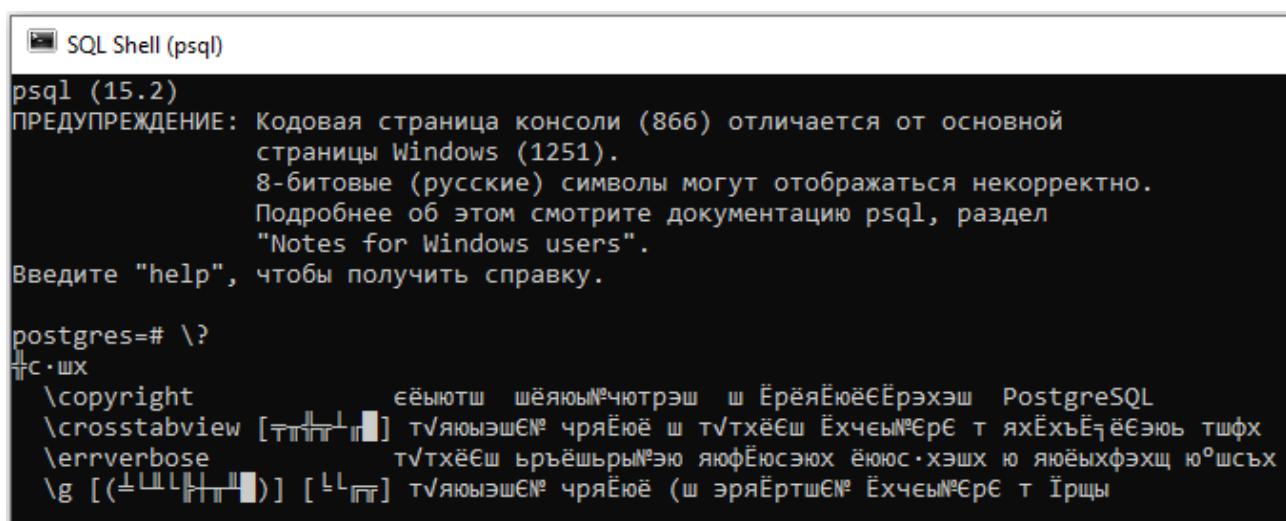


```
Выбрать SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Пароль пользователя postgres:
pgsql (15.2)
ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной страницы Windows (1251).
8-битовые (русские) символы могут отображаться некорректно.
Подробнее об этом смотрите документацию psql, раздел "Notes for Windows users".
Введите "help", чтобы получить справку.

postgres=#
```

Рис. 4. Вызов оболочки SQL Shell (psql)

При работе с psql в среде Windows пользователи сталкиваются с проблемой вывода кириллицы. Возможно некорректное отображение букв русского алфавита, например, при отображении результатов запроса к таблице, в полях которых хранятся строковые данные на русском языке (рис. 5).



```
SQL Shell (psql)
pgsql (15.2)
ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной страницы Windows (1251).
8-битовые (русские) символы могут отображаться некорректно.
Подробнее об этом смотрите документацию psql, раздел "Notes for Windows users".
Введите "help", чтобы получить справку.

postgres=# \?
\c -w
\copyright           ёыютш шёяюычютрэш ш єрёяюёеїрэхэш PostgreSQL
\crosstabview [ ] т/яюыэш€ чряїю ш т/тхёеши єхчев€р€ т яхїхъїёеэю тшфх
\errverbose          т/тхёеши ьръёшьры€ю яофїюсэю ююкос.хэшх ю яюыхфэхщ юшсъх
\g [ ] т/яюыэш€ чряїю (ш эряїртш€ єхчев€р€ т Іршы
```

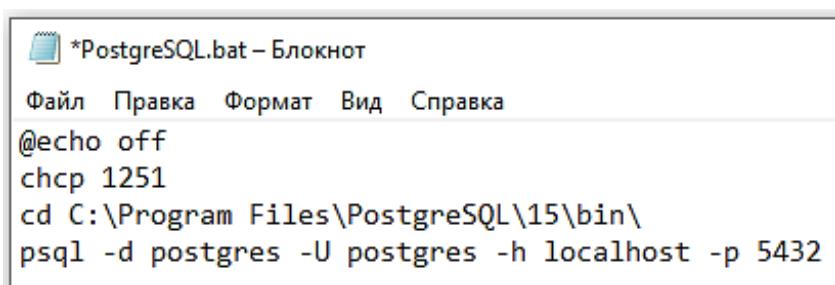
Рис. 5. Некорректное отображение кириллицы в SQL Shell (psql)

Проблема в том, что *cmd.exe* работает в кодировке CP866, а сама Windows – в WIN1251, о чём *psql* честно предупреждает при начале работы.

Для устранения этого необходимо в свойствах окна, в котором выполняется *psql*, изменить шрифт на Lucida Console, а потом с помощью команды *chcp* сменить текущую кодовую страницу на CP1251:

```
chcp 1251
```

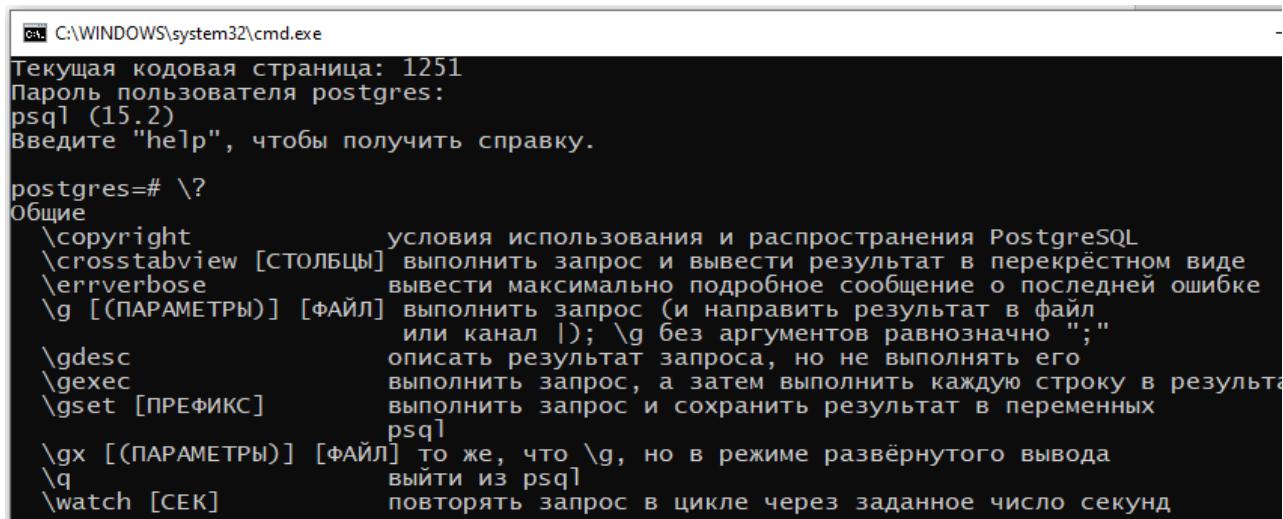
Упростить запуск клиентского приложения в этом случае помогает пакетный файл, который можно назвать, например, *PostgreSQL.bat*, содержимое которого показано на рис. 6. Он может быть создан, например, в *Блокноте*.



```
*PostgreSQL.bat - Блокнот
Файл Правка Формат Вид Справка
@echo off
chcp 1251
cd C:\Program Files\PostgreSQL\15\bin\
psql -d postgres -U postgres -h localhost -p 5432
```

Рис. 6. Пакетный файл *PostgreSQL.bat*

При запуске пакетного файла происходит изменение кодовой страницы, переход в каталог с файлом *psql.exe* и запуск этого приложения с параметрами, заданными по умолчанию (рис. 7). Пароль администратора вводится особо.



```
C:\WINDOWS\system32\cmd.exe
Текущая кодовая страница: 1251
Пароль пользователя postgres:
psql (15.2)
Введите "help", чтобы получить справку.

postgres=# \?
Общие
\copyright      условия использования и распространения PostgreSQL
\crosstabview [СТОЛБЦЫ] выполнить запрос и вывести результат в перекрёстном виде
\errverbose     вывести максимально подробное сообщение о последней ошибке
\g [(ПАРАМЕТРЫ)] [ФАЙЛ] выполнить запрос (и направить результат в файл
или канал |); \g без аргументов равнозначно ";""
\gdesc          описать результат запроса, но не выполнять его
\gexec          выполнить запрос, а затем выполнить каждую строку в результате
\gset [ПРЕФИКС] выполнить запрос и сохранить результат в переменных
            psql
\gx [(ПАРАМЕТРЫ)] [ФАЙЛ] то же, что \g, но в режиме развернутого вывода
\q              выйти из psql
\watch [СЕК]    повторять запрос в цикле через заданное число секунд
```

Рис. 7. Корректное отображение кириллицы

В командной строке *psql* можно вводить операторы SQL, которые необходимо завершать точкой с запятой, и сервисные команды самой программы *psql*, начинающиеся с символа обратной косой черты. Список команд с краткими описаниями того, что они делают, можно получить по команде `\?`

Рассмотрим некоторые команды утилиты *psql*, позволяющие ознакомиться с объектами логического уровня в реляционных базах данных, которыми являются таблицы, типы данных, домены, представления, процедуры, триггеры и др.

Программа *psql* имеет большой набор команд, позволяющих вывести информацию об объектах базы данных. В качестве параметра таких команд указывают имя или шаблон, которому должны удовлетворять имена объектов, обрабатываемые командой.

Многие такие команды начинаются с символов `\d`. Например, чтобы просмотреть список всех таблиц и представлений, созданных в той базе данных, к которой вы сейчас подключены, нужно ввести команду `\dt`. Если нужна структура какой-либо конкретной таблицы, например, *students*, нужно ввести команду `\d students`.

Эта команда выведет список столбцов (атрибутов) вместе с их типами и ограничением *NOT NULL*, а также другие ограничения целостности, определенные для этой таблицы. Для получения списка всех инструкций SQL нужно выполнить команду `\h`. Для вывода описания конкретной инструкции SQL (*CREATE TABLE*):

`\h CREATE TABLE`

Утилита *psql* позволяет сократить объем ручного ввода за счет автодополнения вводимой команды. При вводе инструкции SQL можно использовать клавишу `<Tab>` для дополнения вводимого ключевого слова команды или имени таблицы базы данных. Например, при вводе команды *CREATE TABLE* можно после символов *cr* нажать клавишу `<Tab>`, и *psql*

дополнит это слово до *create*. Аналогично можно поступить и со словом *table* – для его ввода достаточно ввести буквы *ta* и нажать клавишу *<Tab>*. Если введено слишком мало букв, утилита *psql* не сможет однозначно идентифицировать ключевое слово и дополнения не произойдет. В этом случае, нажав клавишу *<Tab>* дважды, можно получить список всех ключевых слов, начинающихся с введенной комбинации букв.

3.2. ПРОГРАММА PGADMIN

Программа pgAdmin – средство администрирования PostgreSQL, являющееся стандартом де-факто. По определению создателей – «pgAdmin – это ведущий инструмент управления с открытым исходным кодом для Postgres, самой передовой в мире базы данных с открытым исходным кодом» [13].

Программа pgAdmin предоставляет мощный графический интерфейс, который упрощает основные задачи администрирования, отображает объекты баз данных, позволяет выполнять запросы SQL. В учебном пособии приведен минимум сведений, необходимый для выполнения заданий лабораторного практикума в pgAdmin. Программа устанавливается автоматически при выборе интерактивного установщика, сертифицированного EDB. Для запуска ее в Windows нужно найти значок *pgAdmin* в меню *Пуск* и запустить его (рис. 8).

Откроется главное окно программы с окном ввода мастер-пароля программы pgAdmin (рис. 9). Вначале следует ввести мастер-пароль программы pgAdmin. Чтобы подключиться к серверу PostgreSQL, в левой части окна (навигаторе объектов) необходимо раскрыть пункт *Servers*, содержащий набор серверов PostgreSQL. Выбор *PostgreSQL 15* отобразит окно ввода пароля суперпользователя *postgres* (рис. 10). Пароль суперпользователя *postgres* был задан при установке PostgreSQL.

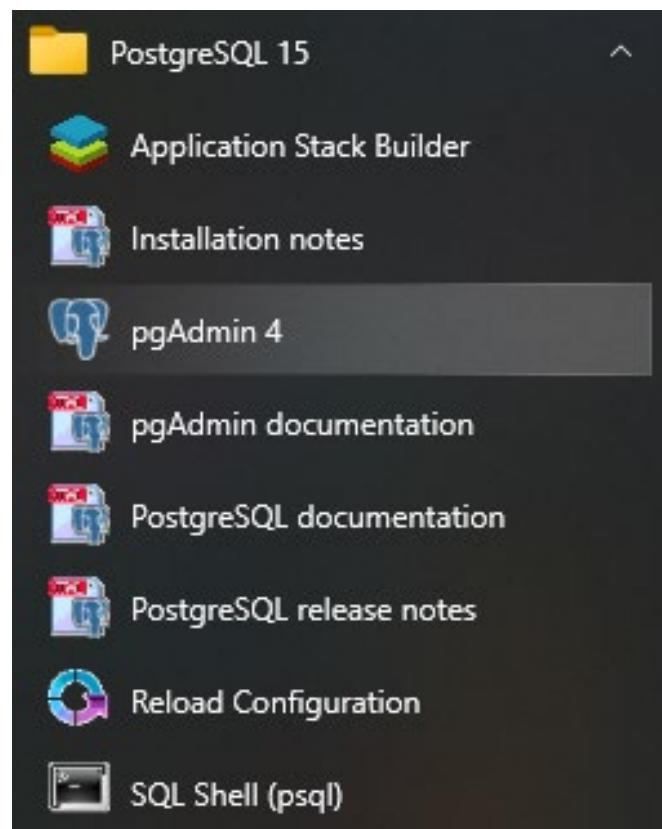


Рис. 8. Запуск программы pgAdmin

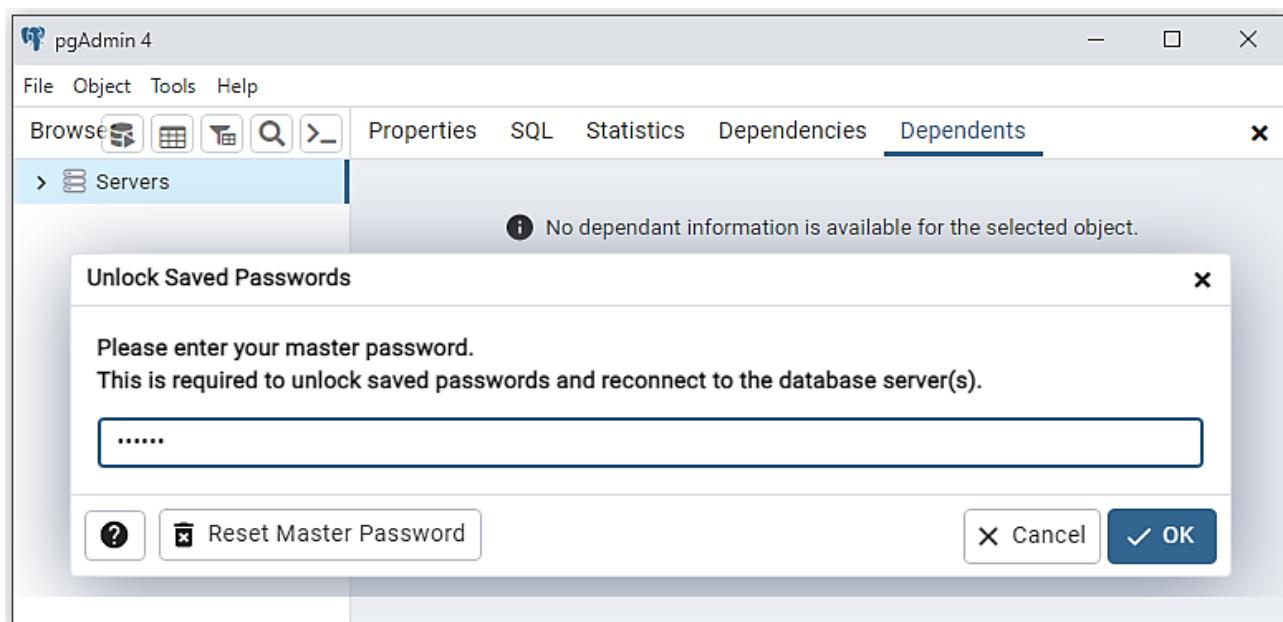


Рис. 9. Окно ввода мастер-пароля программы pgAdmin

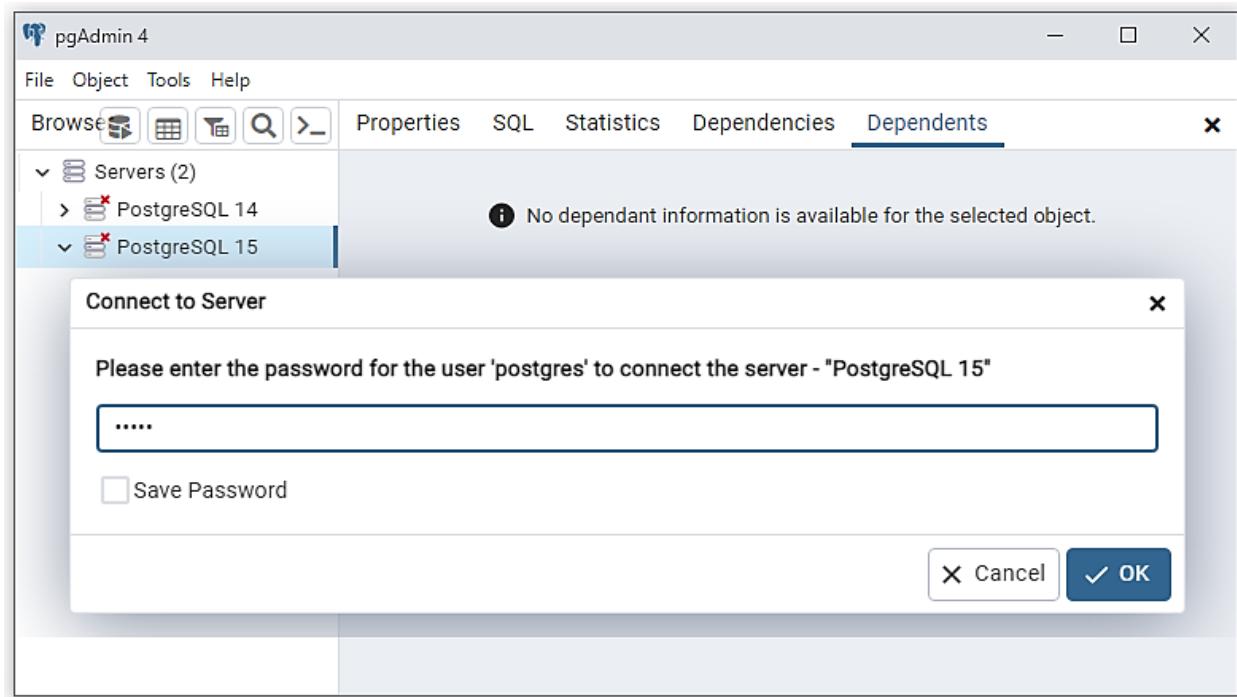


Рис. 10. Окно ввода пароля суперпользователя *postgres*

После успешного входа откроется содержимое сервера (рис. 11).

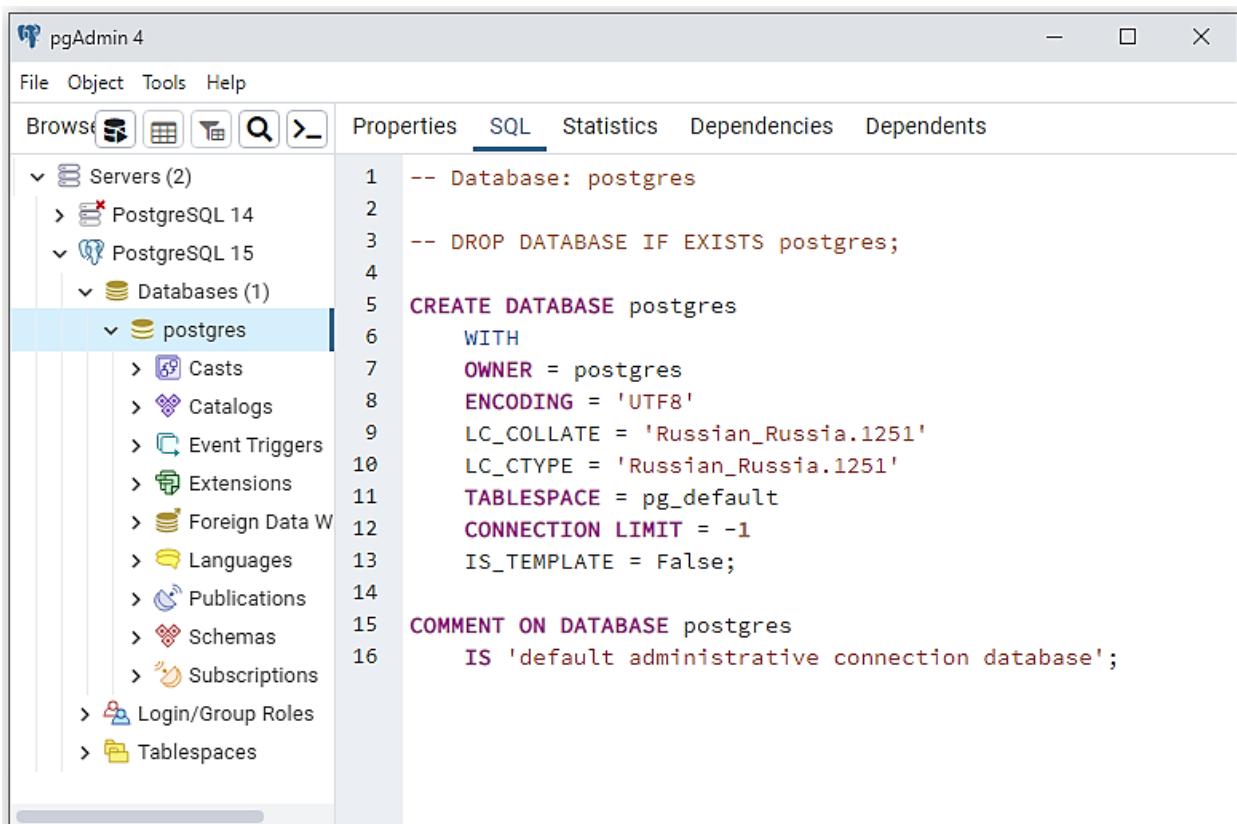


Рис. 11. Основное окно программы pgAdmin

В левой части основного окна находится *навигатор объектов*. В узле *Databases* можно увидеть все имеющиеся на сервере базы данных (по умолчанию имеется только одна база данных – *postgres*). Узел *Login/Group Roles* предназначен для управления пользователями и их ролями. Узел *Tablespaces* позволяет управлять табличными пространствами (местом хранения файлов баз данных).

В навигаторе можно выбрать какой-либо объект, и в правой части окна выводится справочная информация о выбранном объекте.

- *Свойства (Properties)* – зависят от типа объекта (например, для столбца будет показан тип его данных).
- *SQL* – инструкция SQL, с помощью которой создается выбранный в навигаторе объект (рис. 11).
- *Статистика (Statistics)* – информация, которая используется системой для построения планов выполнения запросов и может рассматриваться администратором СУБД для анализа ситуации.
- *Зависимости (Dependencies)* – отображает объекты, от которых зависит выбранный объект. Сервер следит за тем, чтобы случайно не были удалены объекты, от которых зависят другие объекты (чтобы удалить объект с зависимостью, следует использовать команду *DROP CASCADE*).
- *Зависимые (Dependents)* – отображает объекты, зависящие от объекта, выбранного в навигаторе pgAdmin. Зависимый объект можно удалить, не затрагивая объект, выбранный в данный момент в элементе управления деревом pgAdmin.

Управлять объектами (создавать, модифицировать, удалять) можно:

- в навигаторе – наведя на объект курсор мыши и щелкнув ее правой кнопкой, контекстное меню покажет допустимые действия;
- в окне запросов – вводя соответствующую инструкцию языка SQL.

В качестве примера создадим новую базу данных. Нажмем правой кнопкой мыши на узел *Databases* и в контекстном меню выберем *Create / Database...*. В открывшемся окне введем имя базы данных *testDB* и комментарий (рис. 12).

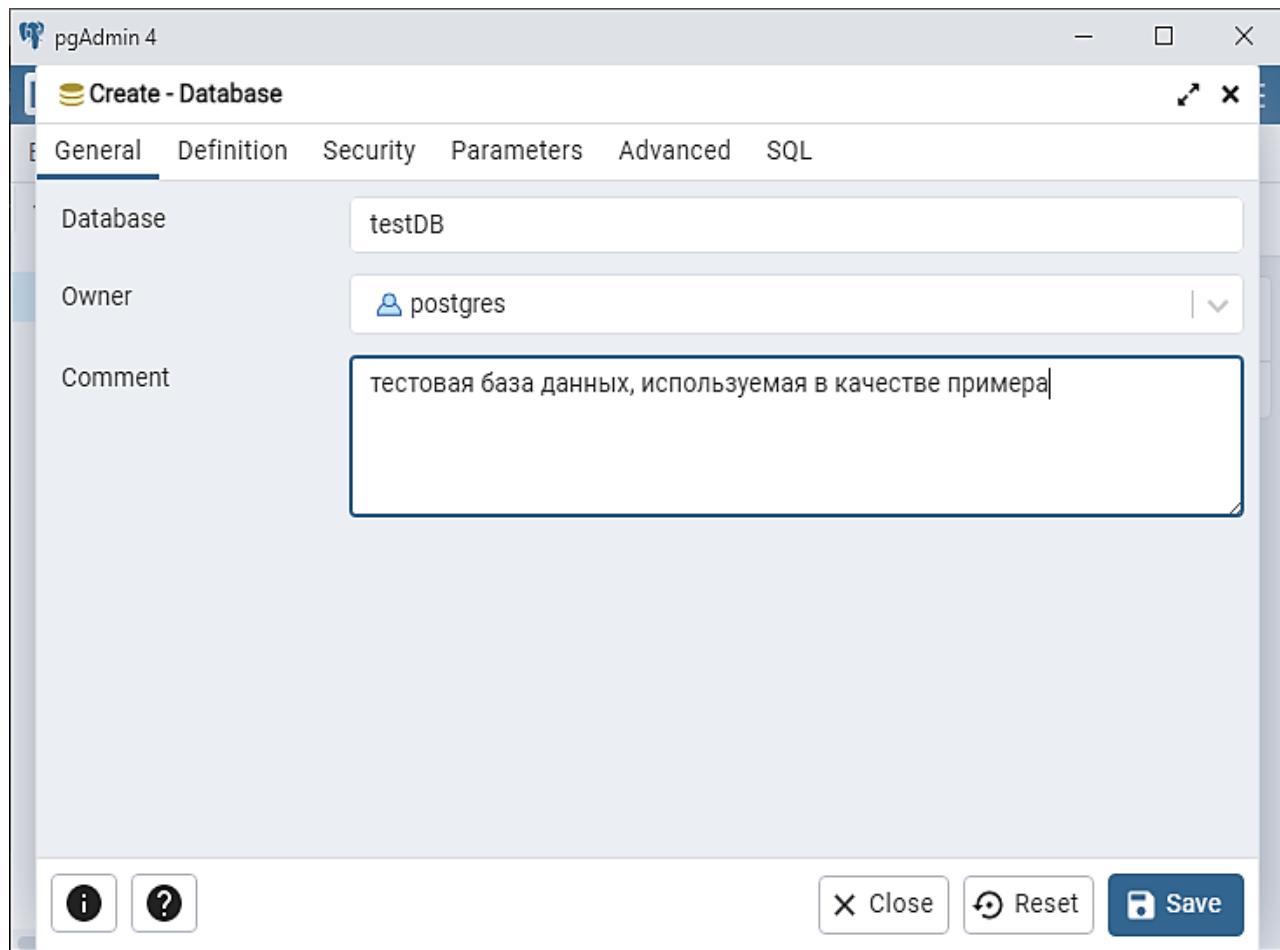


Рис. 12. Окно создания новой базы данных

Владелец (*Owner*) новой базы данных – суперпользователь *postgres*. Нажмем кнопку *Save* и база данных создана. После этого в древовидном меню навигатора отобразится содержимое созданной базы данных *testDB* (рис. 13).

Работа с базой данных осуществляется с помощью языка запросов SQL. Рассмотрим, как выполняются SQL-запросы. Добавим в созданную базу данных *testDB* таблицу и некоторые начальные данные. Для этого выделим в навигаторе правой кнопкой мыши базу данных *testDB* и в появившемся контекстном меню выберем пункт *Query Tool*. Введем запрос на создание таблицы *users* в верхней части окна на вкладке *Query* (*Редактор запросов*) и нажмем *F5*. В нижней части окна на вкладке *Data Output* (*Вывод данных*) появится результат (рис. 14).

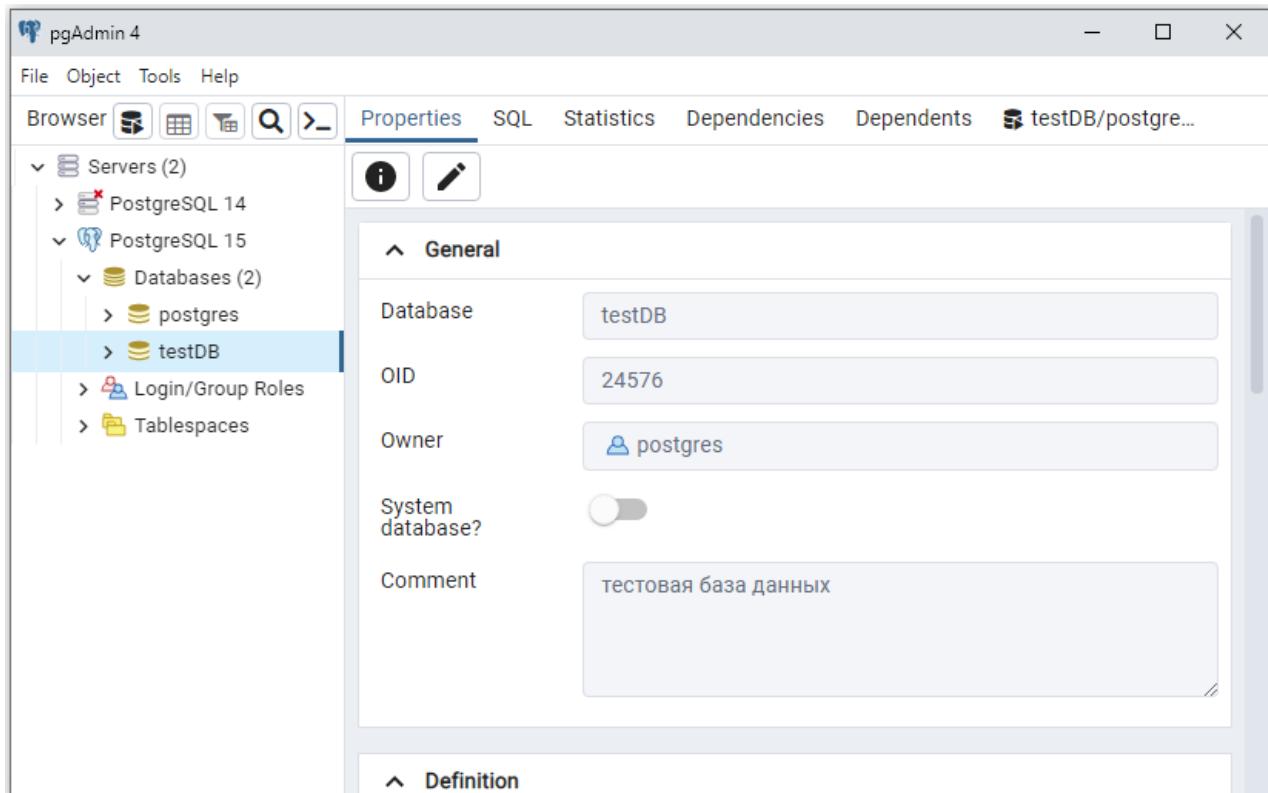


Рис. 13. Отображение созданной базы данных

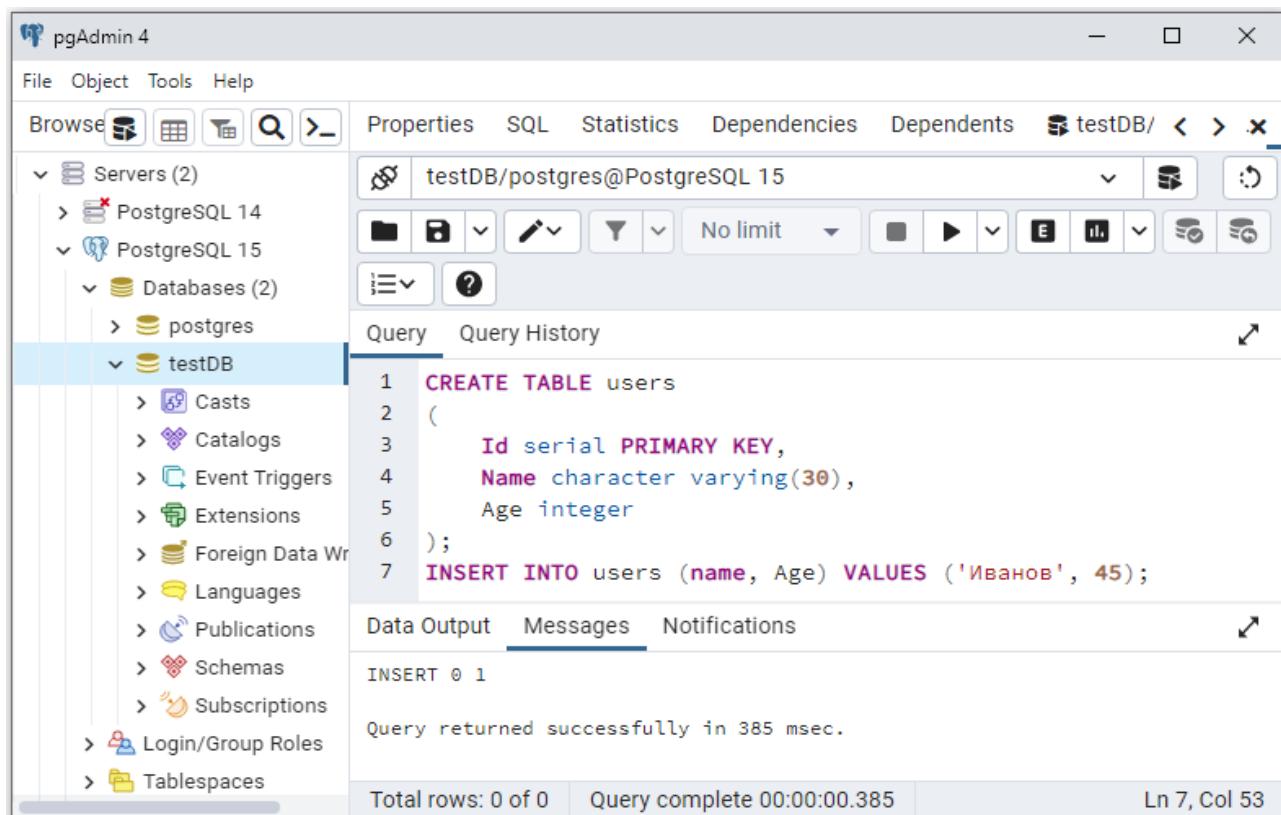


Рис. 14. Создание и заполнение таблицы *users*

Можно вводить следующий запрос на новой строке, не стирая предыдущий, просто выделите нужный фрагмент кода перед тем, как нажимать *F5*.

Первая инструкция *CREATE TABLE* создает таблицу *users* с тремя столбцами *Id*, *Name* и *Age*.

Вторая инструкция *INSERT* добавляет в таб-лицу одну строку. Аналогичным образом выполняется любой другой код SQL.

Вкладка *Data Output* (*Выход данных*) панели вывода отображает данные и статистику, сгенерированные последним выполненным запросом.

Вкладка *Messages* (*Сообщения*) панели вывода для просмотра информации о последнем выполненном запросе. Там же отображается информация об ошибках, возникших при выполнении запроса. В нашем примере команда *INSERT* завершилась успешно и вернула сообщение *INSERT 0 1* (1 – количество вставленных или обновленных строк, *oid* всегда равен 0).

Следует отметить, что для каждого объекта базы данных определяется схема. По умолчанию это схема *public*. Поэтому чтобы найти таблицу, нужно обратиться к узлу базы данных, раскрыть его, выбрать подузел *Schemas*, в нем подузел *public* (название схемы) и далее в нем подузел *Tables*, представляющий все таблицы, связанные со схемой *public*.

Чтобы просмотреть таблицу и ее данные, необходимо нажать правой кнопкой мыши на имени таблицы в навигаторе, навести курсор на *View/Edit Data* (*Просмотр/редактирование данных*) и выбрать *All Rows* (*Все строки*).

В результате откроется новая панель с запросом *SELECT*, под которой на вкладке *Data Output* (*Выход данных*) можно увидеть все данные, хранящиеся в таблице (рис. 15).

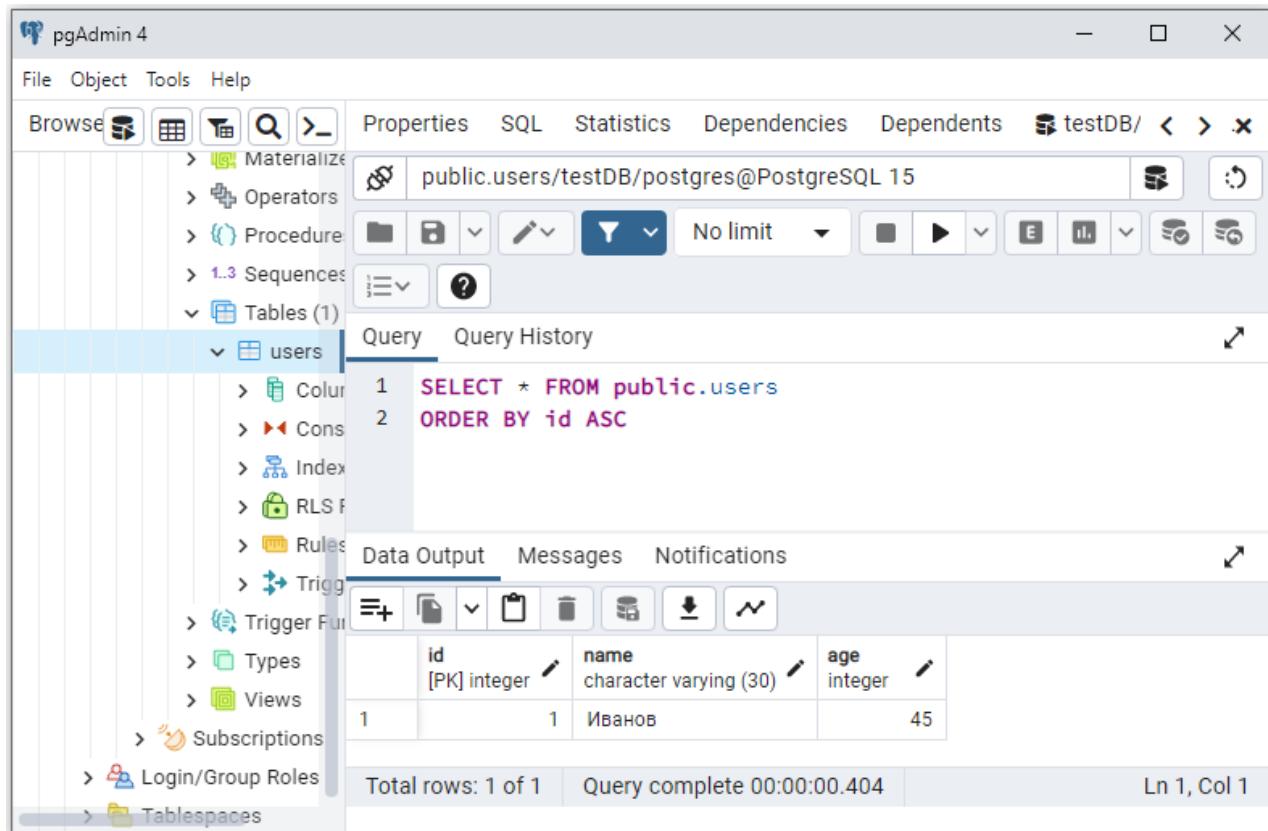


Рис. 15. Просмотр таблицы *users*

4. ПРОГРАММИРОВАНИЕ НА СТОРОНЕ СЕРВЕРА

При разработке приложений баз данных часто целесообразно переложить часть операций с данными с клиентского приложения на серверную часть. Это позволяет:

- упростить программный код приложения;
- уменьшить объем данных, передаваемых по сети;
- ускорить работу приложения.

4.1. ПРОЦЕДУРНЫЕ ЯЗЫКИ

PostgreSQL и другие реляционные СУБД используют в качестве языка запросов SQL. Однако каждый оператор SQL на сервере баз данных выполняется индивидуально. Использование процедурных языков позволяет сгруппировать блок вычислений и последовательность запросов внутри сервера. В итоге получаем силу процедурного языка и простоту использования SQL при экономии накладных расходов:

- исключаются дополнительные обращения между клиентом и сервером;
- промежуточные результаты не передаются между сервером и клиентом;
- можно избежать многочисленных разборов одного запроса.

Хранимые функции на процедурном языке предоставляют API для работы с объектами БД. Пользователи могут выполнять функции, но не иметь прямого доступа к этим объектам. Однако необходимо также учитывать и следующее:

- использование процедурных языков в тех случаях, когда достаточно чистого SQL, может существенно снизить производительность приложения;
- перенос логики приложения на сервер усиливает зависимость от СУБД.

Процедурные языки дополняют декларативный язык SQL процедурными возможностями. Это позволяет создавать пользовательские функции не только на С или чистом SQL. Процедурные языки разделяют на доверенные и недоверенные.

Для доверенных языков:

- ограничена функциональность в части взаимодействия с ОС (работа с файлами, процессами и т. д.);
- использование языка является безопасным, поэтому доступ к нему по умолчанию получают все пользователи.

Для недоверенных языков:

- нет ограничений на взаимодействие с ОС;
- доступ к языку есть только у суперпользователей PostgreSQL, поскольку функция на таком языке может выполнять любые операции в ОС;
- обычно к названию языка добавляют «и»: *plperl*;
- возможен доступ к внешним данным из хранимых функций.

Обычные пользователи также могут получить доступ к функциям на недоверенном языке. Для этого суперпользователь создает функцию с указанием *SECURITY DEFINER* и выдает право на ее исполнение нужным пользователям.

Добавление нового языка выполняется установкой соответствующего расширения. Процедурные языки должны устанавливаться в каждую базу данных, где планируется их использование. Список установленных в БД языков находится в таблице *pg_language*. Кроме языков С и SQL, считающихся внутренними языками системы, доступны следующие языки:

- в дистрибутиве: PL/pgSQL, PL/Td, PL/Perl, PL/Python;
- дополнительно: PL/Java, PL/V8, PL/R, PL/PHP, ...

Для подключения нового процедурного языка используется команда:

CREATE [TRUSTED] LANGUAGE имя

HANDLER обработчик_ вызова

[INLINE обработчик_ внедренного_ кода]

[VALIDATOR функция_ проверки];

Сервер базы данных не может интерпретировать код процедурного языка и передает код *обработчику_вызыва*. Обработчик вызова – специальная функция на С, которая понимает этот язык. При подключении языка можно указать функцию для выполнения анонимных блоков (*обработчик_внедренного_кода*) и функцию для проверки кода программы (*функция_проверки*). Необязательное ключевое слово *TRUSTED* указывает на то, что процедурный язык является доверенным.

4.2. ПРОЦЕДУРНЫЙ ЯЗЫК PL/PGSQL

PL/pgSQL – это процедурный язык для СУБД PostgreSQL. Появился в 1998 г. в версии 6.4. В версии PostgreSQL 9.0 и выше PL/pgSQL устанавливается по умолчанию. Тем не менее это по-прежнему загружаемый модуль и администраторы, особо заботящиеся о безопасности, могут при необходимости его удалить.

Целью проектирования PL/pgSQL было создание простого в использовании загружаемого процедурного языка, который:

- используется для создания функций, процедур и триггеров;
- позволяет использовать все типы данных, операторы и функции SQL;
- добавляет к языку SQL управляющие структуры и обработку ошибок;
- может выполнять сложные вычисления;
- наследует все пользовательские типы, функции, процедуры и операторы;
- может быть определен как доверенный язык.

PL/pgSQL проектировался на основе языка PL/SQL, применяемого в СУБД Oracle, и имеет похожий синтаксис.

4.3. СТРУКТУРА БЛОКА PL/PGSQL

Язык программирования PL/pgSQL является блочно-структурированным языком. Например, текст тела функции должен быть блоком. Структура блока имеет следующий вид:

```
[ «метка» ]
[ DECLARE
  объявления ]
BEGIN
  операторы
[ EXCEPTION
  обработка_ошибок ]
END [ метка ];
```

В структуре блока можно выделить:

- необязательную секцию объявления локальных переменных и курсоров;
- основную секцию исполнения, в которой располагаются операторы;
- необязательную секцию обработки ошибок.

В качестве операторов можно использовать операторы PL/pgSQL, а также большинство команд SQL. При этом SQL и PL/pgSQL интегрированы бесшовно – команды SQL используются в блоке напрямую.

Ключевые слова не чувствительны к регистру символов. Как и в обычных SQL-командах, идентификаторы неявно преобразуются к нижнему регистру, если они не взяты в двойные кавычки.

Комментарии в коде PL/pgSQL работают так же, как в обычном SQL. Двойное тире (--) начинает однострочный комментарий, который завершается в конце строки. Блочный комментарий начинается с /* и завершается */. Блочные комментарии могут быть вложенными.

Метка требуется только тогда, когда нужно идентифицировать блок в операторе *EXIT* или дополнить имена переменных, объявленных в этом блоке. Если метка указана после *END*, то она должна совпадать с меткой в начале блока.

В качестве операторов нельзя использовать команды управления транзакциями (например, *COMMIT*, *ROLLBACK*), а также команды, которые нельзя использовать внутри транзакции (например, *VACUUM*).

Любой оператор в выполняемой секции блока может быть вложенным блоком. Вложенные блоки используются для логической группировки нескольких операторов или локализации области действия переменных для группы операторов. Во время выполнения вложенного блока переменные, объявленные в нем, скрывают одноименные переменные внешних блоков. Чтобы получить доступ к внешним переменным, нужно дополнить их имена меткой блока. Например:

```
1 CREATE FUNCTION somefunc() RETURNS integer AS $$  
2 << outerblock >>  
3 DECLARE  
4     quantity integer := 30;  
5 ▼ BEGIN  
6     RAISE NOTICE 'Сейчас quantity = %', quantity; -- Выводится 30  
7     quantity := 50;  
8     --  
9     -- Вложенный блок  
10    --  
11    DECLARE  
12        quantity integer := 80;  
13    BEGIN  
14        RAISE NOTICE 'Сейчас quantity = %', quantity; -- Выводится 80  
15        RAISE NOTICE 'Во внешнем блоке quantity = %', outerblock.quantity; -- 50  
16    END;  
17    RAISE NOTICE 'Сейчас quantity = %', quantity; -- Выводится 50  
18    RETURN quantity;  
19 END;  
20 $$ LANGUAGE plpgsql;
```

```
1 SELECT somefunc();
```

Data Output Messages Notifications

ЗАМЕЧАНИЕ: Сейчас quantity = 30
ЗАМЕЧАНИЕ: Сейчас quantity = 80
ЗАМЕЧАНИЕ: Во внешнем блоке quantity = 50
ЗАМЕЧАНИЕ: Сейчас quantity = 50

The screenshot shows the pgAdmin interface. In the top-left, there is a code editor window with the following SQL query:

```
1  SELECT somefunc();
```

Below the code editor are three tabs: "Data Output", "Messages", and "Notifications". The "Data Output" tab is selected and highlighted in blue. Underneath the tabs is a toolbar with several icons.

The main area displays the results of the query in a table format:

	somefunc	integer
1		50

Каждое объявление и каждый оператор в блоке должны завершаться точкой с запятой. Вложенный блок должен иметь точку с запятой после *END*, как показано выше. Финальный *END*, завершающий тело функции, точки с запятой не требует.

В примере использована функция – основной строительный блок процедурного расширения PostgreSQL. Функции, написанные на PL/pgSQL, создаются на сервере командами *CREATE FUNCTION*, например:

```
CREATE FUNCTION somefunc (integer, text) RETURNS integer
AS тело_функции
LANGUAGE plpgsql;
```

Начало и конец блока, являющегося телом функции, выделены символами **\$\$**.

4.4. СООБЩЕНИЯ И ОШИБКИ

Для вывода сообщений и вызова ошибок предназначена команда *RAISE*:

```
RAISE [уровень] 'формат' [выражение [, ...]] [USING параметр = значение [, ...]];
```

```
RAISE [уровень] имя_условия [USING параметр = выражение [, ...]];
```

```
RAISE [уровень] SQLSTATE 'sqlstate' [USING параметр = выражение [, ...]];
```

```
RAISE [уровень] USING параметр = выражение [, ...];
```

Уровень важности ошибки: *DEBUG*, *LOG*, *INFO*, *NOTICE*, *WARNING* и *EXCEPTION* (по умолчанию). *EXCEPTION* вызывает ошибку (прерывается текущая транзакция), остальные генерируют сообщения с разными уровнями приоритета.

После уровня можно задать строку формата (строковая константа), определяющую выдаваемый текст. Затем – выражения аргументов, вставляемых в сообщение. Внутри строки формата знак *%* заменяется строковым представлением значения очередного аргумента. Число аргументов должно совпадать с числом заполнителей *%* в строке формата. В примере символ *%* будет заменен на значение *v_job_id*:

```
1 RAISE NOTICE 'Вызов функции cs_create_job(%)', v_job_id;
```

С помощью *USING* и элементов *параметр = выражение* можно добавить дополнительную информацию к отчету об ошибке. Все выражения – строковые. Возможные ключевые слова для параметра:

- *MESSAGE* – устанавливает текст сообщения об ошибке; не может использоваться, если в команде *RAISE* присутствует формат;
- *DETAIL* – предоставляет детальное сообщение об ошибке;
- *HINT* – предоставляет подсказку по вызванной ошибке;
- *ERRCODE* – устанавливает код ошибки (*SQLSTATE*); задается либо по имени, либо напрямую, как пятисимвольный код *SQLSTATE*;
- *COLUMN*, *CONSTRAINT*, *DATATYPE*, *TABLE*, *SCHEMA* – предполагает имя соответствующего объекта, связанного с ошибкой.

Пример прерывает транзакцию и выводит сообщение об ошибке с подсказкой:

```
1 RAISE EXCEPTION 'Несуществующий ID --> %', user_id
2     USING HINT = 'Проверьте ваш пользовательский ID';
```

Предложение *USING* позволяет переопределить стандартное сообщение:

```
1 RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'uniqueViolation';
2 RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

Другой синтаксис команды *RAISE* в качестве главного аргумента использует имя или код *SQLSTATE* ошибки. Например:

```
1 RAISE division_by_zero;
2 RAISE SQLSTATE '22012';
```

4.5. Объявление переменных и констант в PL/PGSQL

Все переменные в блоке должны быть определены в секции объявления. Исключение – переменная-счетчик цикла *FOR*, которая объявляется автоматически (для цикла по диапазону чисел объявляется целочисленная переменная, для цикла по результатам курсора – переменная типа *record*). Примеры объявления:

```
1 user_id integer;
2 quantity numeric(5);
3 url varchar;
4 myrow tablename%ROWTYPE;
5 myfield tablename.columnname%TYPE;
6 arow RECORD;
```

Переменные перечисляются в секции объявления:

DECLARE

имя [*CONSTANT*] тип [*NOT NULL*]
[{ *DEFAULT* | := | = } значение_по_умолчанию];

Указание *CONSTANT* запрещает изменение значения переменной после инициализации, т. е. значение остается постоянным в течение всего блока.

Предложение *DEFAULT* задает начальное значение, присваиваемое переменной при входе в блок. Если оно отсутствует, то переменная инициализируется значением *NULL*. Если указано *NOT NULL*, то попытка присвоить во время выполнения *NULL* приведет к ошибке. Переменные, объявленные как *NOT NULL*, должны иметь по умолчанию непустые значения.

Значение по умолчанию вычисляется и присваивается переменной каждый раз при входе в блок (не только при первом вызове функции). Так, если переменная типа *timestamp* имеет функцию *now()* в качестве значения по умолчанию, то она всегда будет содержать время текущего вызова функции.

Область действия переменной – текущий блок, включая вложенные блоки и секцию обработки ошибок.

В качестве типа можно указать любой тип SQL и некоторые псевдотипы. Конструкция *table%ROWTYPE* используется для объявления переменной составного типа со структурой указанной таблицы. Конструкция *table.column%TYPE* используется для переменной с таким же типом, как у столбца таблицы. В секции *DECLARE* могут также объявляться курсоры. Примеры объявления переменных:

```
1 quantity integer DEFAULT 32;
2 url varchar := 'http://mysite.com';
3 transaction_time CONSTANT timestamp with time zone := now();
```

После объявления переменной ее значение можно использовать в последующих выражениях инициализации в этом же блоке.

4.6. ФУНКЦИИ И ПРОЦЕДУРЫ

Для расширения функциональности создают хранимые подпрограммы: функции и процедуры (процедуры – начиная с версии 11). Тело подпрограммы чаще всего пишут на PL/pgSQL. В любой конфигурации

PostgreSQL можно использовать подпрограммы на С и SQL (внутренние языки). Можно писать на Perl, Python, PHP, bash и других языках, но их потребуется добавить в экземпляр PostgreSQL.

Функции:

- вызываются в контексте какого-либо выражения (в операторе SQL);
- не могут управлять транзакциями;
- возвращают результат.

Процедуры:

- вызываются оператором CALL;
- могут управлять транзакциями;
- могут возвращать результат.

Функции принимают входные данные в виде параметров и могут создавать выходные данные в виде выходных параметров или возвращаемых значений. Пользовательские функции имеют те же привилегии и полномочия, что и встроенные. Параметры функции и возвращаемые результаты могут иметь любой тип данных PostgreSQL. Параметры и возвращаемое значение для функции не обязательны. Тело любой функции оформляется как блок.

Новую функцию можно определить командой *CREATE FUNCTION*. Имя новой функции должно отличаться от имен существующих функций или процедур с такими же типами аргументов в этой схеме.

ВНИМАНИЕ!

Функции и процедуры с аргументами разных типов могут иметь одно имя (это называется перегрузкой).

Команда *CREATE OR REPLACE FUNCTION* создает новую функцию либо заменяет определение существующей. Но она не позволит изменить имя или аргументы функции (если это сделать, будет создана новая функция). Кроме того, эта команда не позволит изменить тип результата существующей функции. Для этого нужно удалить функцию и создать ее зано-

во. Если функция имеет выходные параметры (*OUT*), то изменить их типы можно, только удалив функцию.

Удалить функцию можно командой *DROP FUNCTION*. Но если удалить и затем снова создать функцию, новая функция станет другой сущностью, и потребуется удалить правила, представления, триггеры и т. п., ссылающиеся на старую функцию. Чтобы изменить определение функции, сохраняя ссылающиеся на нее объекты, следует использовать *CREATE OR REPLACE FUNCTION*. Многие дополнительные свойства существующей функции можно изменить командой *ALTER FUNCTION*.

Определение функции включает имя, параметры, тип возвращаемого значения и тело. Тело – строковая константа, содержащая код на языке, указанном в предложении *LANGUAGE*. Тело-строка сохраняется в системном каталоге и интерпретируется каждый раз при вызове функции. Так как тело функции может содержать апострофы и кавычки, удобно заключать его в символы **\$\$**.

ВНИМАНИЕ!

В определении подпрограммы (функции или процедуры) предложение *LANGUAGE* язык может указываться как до, так и после описания ее тела.

Функция с входными параметрами имеет следующий вид:

CREATE FUNCTION имя (*IN* параметр тип [, ...]) *RETURNS* тип
AS **\$\$**

[DECLARE ...]

BEGIN

...

RETURN значение;

END;

\$\$ LANGUAGE plpgsql;

Для выхода из функции и возврата значения используется оператор *RETURN*.

Функция с выходными параметрами:

```
CREATE FUNCTION имя (INOUT параметр1 тип, OUT параметр2 тип)
AS $$

[DECLARE ...]

BEGIN

    параметр1 := value;
    параметр2 := value;

END;

$$ LANGUAGE plpgsql;
```

Возврат значения здесь – присвоение значений выходным параметрам (*OUT*, *INOUT*). Вместо *RETURN* нужно присвоить значения выходным параметрам.

```
1 CREATE OR REPLACE FUNCTION hello(p text) RETURNS text
2 LANGUAGE plpgsql AS $$

3 DECLARE
4     v text;
5 BEGIN
6     v := 'Hello, ';
7     RETURN v || p || '!';
8 END;
9 $$;
```

```
1 SELECT hello('world');
```

Data Output Messages Notifications

	hello	text
1	Hello, world!	

Еще один пример базовой стратегии передачи параметров:

```
1 CREATE FUNCTION mid(varchar, integer, integer) RETURNS varchar
2 AS $$

3 BEGIN
4     RETURN substring($1,$2,$3);
5 END;
6 $$

7 LANGUAGE plpgsql;
```

```

1 SELECT mid('Математика',3,4);

```

Data Output Messages Notifications

	mid
1	тема

Переданные в функцию параметры именуются идентификаторами *\$1*, *\$2* и т. д. Параметры не имеют имен и доступны в функции по относительному расположению слева направо. Доступ к параметрам по имени (псевдониму) делает результирующий код функции более читабельным:

```

1 CREATE FUNCTION mid(keyfield varchar, starting_point integer)
2 RETURNS varchar
3 AS $$ 
4 BEGIN
5     RETURN substring(keyfield,starting_point);
6 END
7 $$ 
8 LANGUAGE plpgsql;

```

```

1 SELECT mid('Иван Сергеевич Тургенев',16);

```

Data Output Messages Notifications

	mid
1	Тургенев

Пример демонстрирует перегрузку функции *mid*. Можно использовать несколько процедур с одним именем, но разными параметрами. Выше объявлена функция *mid* с тремя параметрами, а в этом примере – альтернативная форма функции *mid* с двумя параметрами. Если третий параметр опущен, результат – строка, начинающаяся с *starting_point* и продолжающаяся до конца входной строки.

С целью улучшения читаемости для параметров *\$n* объявляют псевдонимы. Создать псевдоним можно двумя способами.

Первый способ – в команде *CREATE FUNCTION*:

```
1 CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$  
2 ▼ BEGIN  
3     RETURN subtotal * 0.06;  
4 END;  
5 $$ LANGUAGE plpgsql;
```

```
1 SELECT sales_tax(100.00);
```

Data Output		Messages	Notifications
			
sales_tax	real		

1 6

Второй способ – явное объявление псевдонима:

```
1 CREATE FUNCTION sales_tax(real) RETURNS real AS $$  
2 DECLARE  
3     subtotal ALIAS FOR $1;  
4 ▼ BEGIN  
5     RETURN subtotal * 0.06;  
6 END;  
7 $$ LANGUAGE plpgsql;
```

```
1 SELECT sales_tax(100.00);
```

Data Output		Messages	Notifications
			
sales_tax	real		

1 6

Эти примеры не полностью эквивалентны. В первом случае на *subtotal* можно ссылаться как *sales_taxsubtotal*, а во втором случае такая ссылка невозможна. Если к внутреннему блоку добавить метку, то *subtotal* можно дополнить этой меткой.

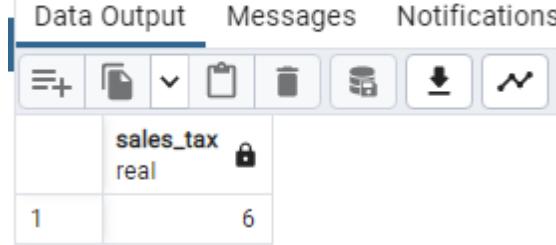
Когда функция объявляется с выходными параметрами, им также выдаются цифровые идентификаторы *\$n* и для них тоже можно создавать

псевдонимы. Выходной параметр – это переменная, стартующая с *NULL*, которой присваивается значение во время выполнения функции. Возвращается последнее присвоенное значение.

Функция *sales_tax* может быть переписана следующим образом:

```
1 CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$  
2 ▼ BEGIN  
3     tax := subtotal * 0.06;  
4 END;  
5 $$ LANGUAGE plpgsql;  
  
1 SELECT sales_tax(100.00);
```

Data Output Messages Notifications

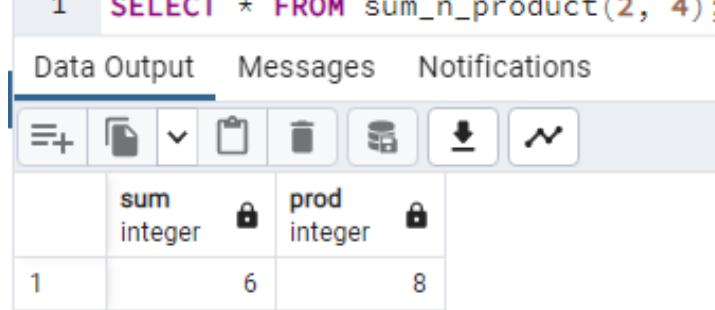


	sales_tax	real
1		6

Конструкция *RETURNS real* опущена. При вызове функции с параметрами *OUT* выходные параметры не указывают:

```
1 CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int)  
2 AS $$  
3 ▼ BEGIN  
4     sum := x + y;  
5     prod := x * y;  
6 END;  
7 $$ LANGUAGE plpgsql;  
  
1 SELECT * FROM sum_n_product(2, 4);
```

Data Output Messages Notifications



	sum	prod
	integer	integer
1	6	8

Для возвращения результата функции здесь создается анонимный тип *record*. Если используется предложение *RETURNS*, то оно должно выглядеть как *RETURNS record*. Можно объявить функцию с использованием *RETURNS TABLE*:

```

1 CREATE FUNCTION extended_sales(p_itemno int)
2 RETURNS TABLE(quantity int, total numeric) AS $$ 
3 ▼ BEGIN
4     RETURN QUERY SELECT s.quantity, s.quantity * s.price
5             FROM sales s
6             WHERE s.itemno = p_itemno;
7 END;
8 $$ LANGUAGE plpgsql;

```

Это в точности соответствует объявлению одного или нескольких параметров *OUT* и указанию *RETURNS SETOF некий тип*.

Новую процедуру создает команда *CREATE PROCEDURE*. Команда *CREATE OR REPLACE PROCEDURE* создает новую процедуру или заменяет определение существующей. Удаляется процедура командой *DROP PROCEDURE*. Чтобы изменить определение процедуры, сохраняя ссылающиеся на нее объекты, следует использовать *CREATE OR REPLACE PROCEDURE*.

Некоторые дополнительные свойства существующей процедуры можно изменить с помощью команды *ALTER PROCEDURE*.

Процедура не возвращает значения, поэтому может завершаться без *RETURN*. Если нужно досрочно завершить ее выполнение, указывают просто *RETURN*. Если у процедуры есть выходные параметры, конечные значения соответствующих им переменных будут переданы вызывающему коду.

Функция, процедура или блок *DO* может вызвать процедуру с помощью оператора *CALL*. Каждому параметру *OUT* или *INOUT* процедуры должна соответствовать переменная в операторе *CALL* (им будут присвоены возвращаемые значения):

```

1 CREATE PROCEDURE multi(INOUT x int)
2 LANGUAGE plpgsql
3 AS $$ 
4 ▼ BEGIN
5     x := x * 5;
6 END;
7 $$;

```

Осуществим вызов процедуры *multi* из анонимного блока *DO* (см. п. 4.7):

```
1 DO $$  
2 DECLARE myvar int := 3;  
3 ▼ BEGIN  
4   CALL multi(myvar);  
5   RAISE NOTICE 'myvar = %', myvar;  
6 END;  
7 $$;
```

Data Output Messages Notifications

ЗАМЕЧАНИЕ: myvar = 15

Переменная, соответствующая выходному параметру, может быть простой переменной или полем переменной составного типа. В настоящее время она не может быть элементом массива.

4.7. АНОНИМНЫЕ БЛОКИ

Для использования PL/pgSQL не обязательно создавать функции. Код можно оформить и выполнить как анонимный блок при помощи команды *DO*:

```
DO [ LANGUAGE имя_языка ] $$  
[ «метка» ]  
[ DECLARE  
  объявления ]  
BEGIN  
  операторы  
END [ метка ]  
$$;
```

Команду *DO* можно использовать с любым процедурным языком, для которого в *CREATE LANGUAGE* был определен *обработчик_*

внедренного_кода (во фразе *INLINE*). Если в команде *DO* язык не указан, то будет использоваться PL/pgSQL.

Особенности анонимных блоков:

- код анонимного блока, в отличие от функций, не сохраняется на сервере;
- нет прямой возможности передать в анонимный блок параметры или вернуть из него значение (косвенно это можно сделать, например, через таблицы).

Анонимный блок кода идеально подходит в тех случаях, когда есть некоторый сложный код, обрабатывающий данные. Для увеличения производительности требуется, чтобы код полностью выполнялся на стороне базы данных.

Все, что было сказано выше о вложенности блоков и областях видимости переменных, справедливо и в этом случае.

Однако у анонимных блоков кода есть проблема – они не приспособлены для возврата данных. Частично проблема решается выводом в окно *Messages* замечаний с помощью оператора *RAISE NOTICE*, как это показано в следующем примере.

```
1 DO $$  
2 <<first_block>>  
3 DECLARE  
4     counter integer := 0;  
5 BEGIN  
6     counter := counter + 1;  
7     RAISE NOTICE 'Текущее значение счетчика равно %', counter;  
8 END first_block $$;
```

Data Output Messages Notifications

ЗАМЕЧАНИЕ: Текущее значение счетчика равно 1
DO

Пример анонимного блока, содержащего вложенный блок (подблок):

```
1 DO $$  
2 <<outer_block>>  
3 DECLARE  
4     counter integer := 0;  
5 ▼ BEGIN  
6     counter := counter + 1;  
7     RAISE NOTICE 'Текущее значение счетчика равно %', counter;  
8  
9     DECLARE  
10        counter integer := 0;  
11     BEGIN  
12         counter := counter + 10;  
13         RAISE NOTICE 'Текущее значение счетчика в подблоке равно %', counter;  
14         RAISE NOTICE 'Текущее значение счетчика во внешнем блоке равно %',  
15             outer_block.counter;  
16     END;  
17     RAISE NOTICE 'Текущее значение счетчика во внешнем блоке равно %', counter;  
18 END outer_block $$;
```

Data Output Messages Notifications

ЗАМЕЧАНИЕ: Текущее значение счетчика равно 1
ЗАМЕЧАНИЕ: Текущее значение счетчика в подблоке равно 10
ЗАМЕЧАНИЕ: Текущее значение счетчика во внешнем блоке равно 1
ЗАМЕЧАНИЕ: Текущее значение счетчика во внешнем блоке равно 1
DO

4.8. ОСНОВНЫЕ ОПЕРАТОРЫ ЯЗЫКА PL/PGSQL

Рассмотрим типы операторов, которые понимает PL/pgSQL. Все, что не признается в качестве одного из этих операторов, считается командой SQL и отправляется для исполнения в основную машину базы данных.

Оператор присваивания

Присваивание значения переменной в PL/pgSQL записывается в виде *переменная { := | = } выражение;*

Выражение в операторе вычисляется с помощью SQL-команды *SELECT*, посылаемой в машину базы данных. Выражение должно полу-

чить одно значение (возможно, значение строки, если это переменная-кортеж или переменная типа *record*). Целевая переменная может быть:

- простой переменной (возможно, с именем блока);
- полем в целевом кортеже или записи;
- элементом или срезом целевого массива.

Для присваивания можно использовать знак равенства (=) вместо совместимого с PL/SQL :=. Пример оператора присваивания:

```
1 tax := subtotal * 0.06;
2 my_record.user_id := 20;
3 my_array[j] := 20;
4 my_array[1:3] := array[1,2,3];
5 complex_array[n].realpart = 12.3;
```

Если тип данных результата выражения не соответствует типу данных переменной, значение будет преобразовано к нужному типу с использованием приведения присваивания. Если для этой пары типов нет приведения присваивания, интерпретатор PL/pgSQL попытается преобразовать значение результата через текстовый формат, т. е. применив функцию вывода типа результата, а за ней функцию ввода типа переменной. При этом функция ввода может выдать ошибку времени выполнения, если не воспримет строковое представление значения результата.

Условные операторы

Условный оператор записывается в одной из трех форм:

<i>IF</i> условие <i>THEN</i> <i>оператор;</i> ...	<i>IF</i> условие <i>THEN</i> <i>оператор;</i> ...	<i>IF</i> условие <i>THEN</i> <i>оператор;</i> ...
<i>ELSIF</i> условие <i>THEN</i> <i>оператор;</i> ...	<i>ELSE</i> <i>оператор;</i> ...	<i>END IF;</i>
<i>[ELSIF</i> условие <i>THEN</i> <i>операторы ...]</i>	<i>END IF;</i>	
<i>[ELSE</i> <i>оператор; ...]</i>		
<i>END IF;</i>		

Простейшей формой условного оператора является *IF-THEN*. Операторы между *THEN* и *END IF* выполняются, если условие (логическое выражение) истинно. В противном случае они опускаются. Пример оператора:

```
1 ▾ IF v_user_id <> 0 THEN
2     UPDATE users SET email = v_email WHERE user_id = v_user_id;
3 END IF;
```

Форма *IF-THEN-ELSE* добавляет к *IF-THEN* возможность указать альтернативный набор операторов, которые будут выполнены, если условие не истинно (в том числе, если условие *NULL*). Пример этой формы оператора:

```
1 ▾ IF v_count > 0 THEN
2     INSERT INTO users_count (count) VALUES (v_count);
3     RETURN 't';
4 ELSE
5     RETURN 'f';
6 END IF;
```

Проверку нескольких вариантов обеспечивает форма *IF-THEN-ELSIF*. Условия в *IF* проверяются последовательно, пока не будет найдено первое истинное. Выполняются операторы, относящиеся к этому условию, и управление переходит к следующей после *END IF* команде. Последующие условия не проверяются. Если ни одно из условий *IF* не является истинным, выполняется блок *ELSE* (если есть).

Пример:

```
1 ▾ IF number = 0 THEN
2     result := 'zero';
3 ELSIF number > 0 THEN
4     result := 'positive';
5 ELSIF number < 0 THEN
6     result := 'negative';
7 ELSE
8     -- остаётся только один вариант: number имеет значение NULL
9     result := 'NULL';
10 END IF;
```

Вместо ключевого слова *ELSIF* можно использовать *ELSEIF*.

Оператор выбора альтернативных вариантов *CASE* реализует ветвление на несколько ветвей. Может записываться в одной из двух форм – с отдельными условиями на каждую альтернативу или с перечислением возможных значений выражения:

<i>CASE</i>	<i>CASE выражение</i>
<i>WHEN условие-1 THEN</i>	<i>WHEN значение-1 THEN</i>
<i>оператор; ...</i>	<i>оператор; ...</i>
<i>WHEN условие-2 THEN</i>	<i>WHEN значение-2 THEN</i>
<i>оператор; ...</i>	<i>оператор; ...</i>
<i>...</i>	<i>...</i>
<i>ELSE/</i>	<i>ELSE</i>
<i>оператор; ...</i>	<i>оператор; ...</i>
<i>END CASE;</i>	<i>END CASE;</i>

Первая форма *CASE* реализует выполнение на основании истинности логических условий. Каждое условие (логическое выражение) в предложении *WHEN* вычисляется по порядку до тех пор, пока не будет найдено истинное. Выполняются соответствующие операторы и управление переходит к следующей после *END CASE* команде.

Последующие выражения *WHEN* не проверяются. Если ни одно из условий не окажется истинным, то выполняются операторы в *ELSE*. Если *ELSE* нет, будет вызвано исключение *CASE_NOT_FOUND*. Пример оператора:

```
1 ▼ CASE
2   WHEN n BETWEEN 0 AND 10 THEN
3     msg := 'значение в диапазоне между 0 и 10';
4   WHEN n BETWEEN 11 AND 20 THEN
5     msg := 'значение в диапазоне между 11 и 20';
6 END CASE;
```

Эта форма эквивалентна *IF-THEN-ELSIF*. Отличие – при невыполнении всех условий и отсутствии *ELSE* оператор *IF* ничего не делает, а оператор *CASE* вызывает ошибку.

Вторая форма *CASE* реализует условное выполнение на основе сравнения операндов. Выражение вычисляется один раз и последовательно сравнивается с каждым значением в условиях *WHEN*. Если найдено совпадение, выполняются соответствующие операторы и управление переходит к следующей после *END CASE* команде. Последующие выражения *WHEN* не проверяются. Если совпадение не найдено, выполняются операторы в *ELSE*. Если *ELSE* отсутствует, то вызывается исключение *CASE_NOT_FOUND*. Пример:

```
1 ▼ CASE n
2     WHEN 1, 2 THEN
3         msg := 'один или два';
4     ELSE
5         msg := 'значение, отличное от один или два';
6 END CASE;
```

В операторах *IF* и *CASE* условие в скобки заключать не нужно. Составные операторы (после *THEN* и *ELSE*) также не требуют дополнительных скобок. Операторы *IF* и *CASE* являются управляющими конструкциями и не вырабатывают значения.

Оператор цикла *LOOP*

Есть различные варианты оператора цикла, среди которых есть предназначенные для обработки множеств объектов базы данных, представленных запросами или курсорами. В любом случае тело цикла выделяется ключевыми словами *LOOP* и *END LOOP*, между которыми размещается составной оператор (скобки не нужны).

[<<метка>>]

LOOP

оператор; ...

END LOOP [метка];

Оператор *LOOP* организует безусловный цикл, который повторяется до бесконечности, пока не будет прекращен операторами *EXIT* или *RETURN*. Для вложенных циклов можно использовать метку в операторах *EXIT* и *CONTINUE*, чтобы указать, к какому циклу эти операторы относятся.

Оператор EXIT

Для выхода из цикла в любом случае можно использовать оператор *EXIT*:

EXIT [метка] [WHEN условие];

```
1 ▼ LOOP
2      -- здесь производятся вычисления
3 ▼   IF count > 0 THEN
4       EXIT; -- выход из цикла
5   END IF;
6 END LOOP;
```

При наличии *WHEN* цикл прекращается, только если условие имеет значение *true*. Иначе управление переходит к оператору, следующему за *EXIT*. Пример:

```
1 ▼ LOOP
2      -- здесь производятся вычисления
3      EXIT WHEN count > 0; -- аналогично предыдущему примеру
4 END LOOP;
```

Пример использования бесконечного цикла *LOOP* с выходом *EXIT* по условию. Функция, возвращающая *n*-е число Фибоначчи (0, 1, 1, 2, 3, 5, ...):

```
1 CREATE OR REPLACE FUNCTION fib(n integer)
2 RETURNS integer
3 AS $$
4 DECLARE
5     counter integer := 0;
6     a integer := 0;
7     b integer := 1;
8 BEGIN
9    IF (n < 1) THEN
10        RETURN 0;
11    END IF;
12 LOOP
13    counter := counter + 1;
14    EXIT WHEN counter = n;
15    SELECT b,a+b INTO a,b;
16 END LOOP;
17 RETURN a;
18 END;
19 $$
20 LANGUAGE plpgsql;
```

The screenshot shows a database interface with a query window containing the command `1 SELECT fib(1);`. Below the query window is a toolbar with various icons. Underneath the toolbar is a data grid with two columns: 'fib' and 'integer'. The first row has the value '1' in the 'fib' column and '0' in the 'integer' column. There is also a lock icon next to the 'fib' column header.

1	SELECT fib(1);
1	fib integer

The screenshot shows a database interface with a query window containing the command `1 SELECT fib(6);`. Below the query window is a toolbar with various icons. Underneath the toolbar is a data grid with two columns: 'fib' and 'integer'. The first row has the value '1' in the 'fib' column and '5' in the 'integer' column. There is also a lock icon next to the 'fib' column header.

1	SELECT fib(6);
1	fib integer

В инструкции *SELECT b, a+b INTO a, b* выполняется одновременное присваивание двух переменных, что позволяет обойтись без третьей переменной.

Если метка не указана, то *EXIT* завершает самый внутренний цикл, далее выполняется оператор, следующий за *END LOOP*. Если метка указана, то она должна относиться к текущему или внешнему циклу, или это может быть метка блока. При этом в именованном цикле/блоке выполнение прекращается, а управление переходит к следующему оператору после соответствующего *END*.

Когда *EXIT* используется для выхода из блока, управление переходит к следующему оператору после окончания блока. Для выхода из блока нужно обязательно указывать метку. *EXIT* без метки не позволяет прекратить работу блока. Пример:

```
1  <<block1>>
2 ▼ BEGIN
3      -- здесь производятся вычисления
4 ▼ IF sm > 100000 THEN
5      EXIT block1; -- выход из блока BEGIN
6  END IF;
7      -- вычисления, которые не будут выполнены, если sm > 100000
8 END;
```

Оператор **CONTINUE**

Прекращает текущую итерацию цикла и выполняет переход к началу следующей итерации:

CONTINUE [метка] [WHEN условие];

Если метка не указана, то начинается следующая итерация самого внутреннего цикла. Оставшиеся в цикле операторы пропускаются и управление переходит к управляющему выражению цикла (если есть) для определения, нужна ли еще одна итерация цикла. Если метка присутствует, то она указывает на метку цикла, выполнение которого будет продолжено.

При наличии *WHEN* следующая итерация цикла начинается только тогда, когда условие имеет значение *true*. Иначе управление переходит

к оператору, следующему за *CONTINUE*. Оператор можно использовать со всеми типами циклов. Пример:

```
1 ▾ LOOP
2      -- здесь производятся вычисления
3      EXIT WHEN count > 100;
4      CONTINUE WHEN count < 50;
5      -- вычисления для count в диапазоне 50 .. 100
6  END LOOP;
```

Оператор цикла с предусловием WHILE

Цикл с предусловием. Действия в цикле выполняются до тех пор, пока условие имеет значение *true*. Условие проверяется перед выполнением тела цикла:

```
[<<метка>>]
WHILE условие LOOP
    операторы
END LOOP [ метка ];
```

Пример использования цикла с предусловием *WHILE* (модификация функции, возвращающей *n*-е число Фибоначчи):

```
1 CREATE OR REPLACE FUNCTION fib(n integer)
2 RETURNS integer
3 AS $$
4 DECLARE
5     counter integer := 1;
6     a integer := 0;
7     b integer := 1;
8 ▾ BEGIN
9 ▾   IF (n < 1) THEN
10      RETURN 0;
11   END IF;
12   WHILE counter < n
13 ▾     LOOP
14       counter := counter + 1;
15       SELECT b,a+b INTO a,b;
16   END LOOP;
17   RETURN a;
18 END;
19 $$
20 LANGUAGE plpgsql;
```

The screenshot shows two separate queries and their results. Both queries are SELECT statements on the function fib.

The first query is: `1 SELECT fib(2);` and the result is:

	fib	integer
1		1

The second query is: `1 SELECT fib(5);` and the result is:

	fib	integer
1		3

Оператор цикла с параметром FOR (целочисленный вариант)

Количество повторений цикла определяет заголовок. Переменная-параметр принимает последовательно значения отрезка целочисленной арифметической прогрессии. Этую переменную не следует определять в разделе описаний переменных.

[<<метка>>]

*FOR параметр IN [REVERSE] выражение .. выражение [BY выражение]
LOOP*

операторы

END LOOP [метка];

Параметр имеет тип *integer* и существует только внутри цикла (если есть переменная с таким именем, то внутри цикла она будет игнорироваться). Выражения для нижней и верхней границы диапазона чисел вычисляются один раз при входе в цикл. Если не указано *BY*, то шаг итерации 1, в противном случае используется значение в *BY*, которое вычисляется также один раз при входе в цикл. Если указано *REVERSE*, то после каждой итерации величина шага вычитается, а не добавляется.

```
1 ▼ FOR i IN 1..10 LOOP
2   -- внутри цикла переменная i будет иметь значения 1,2,3,4,5,6,7,8,9,10
3 END LOOP;

1 ▼ FOR i IN REVERSE 10..1 LOOP
2   -- внутри цикла переменная i будет иметь значения 10,9,8,7,6,5,4,3,2,1
3 END LOOP;

1 ▼ FOR i IN REVERSE 10..1 BY 2 LOOP
2   -- внутри цикла переменная i будет иметь значения 10,8,6,4,2
3 END LOOP;
```

Если нижняя граница цикла больше верхней границы (или меньше, в случае *REVERSE*), то тело цикла не выполняется вообще. При этом ошибка не возникает.

Пример использования цикла с параметром *FOR* (модификация функции, возвращающей *n*-е число Фибоначчи):

```

1 CREATE OR REPLACE FUNCTION fib(n integer)
2 RETURNS INTEGER
3 AS $$ 
4 DECLARE
5     a integer := 0;
6     b integer := 1;
7 BEGIN
8    IF (n < 1) THEN
9        RETURN 0;
10    END IF;
11    FOR counter IN 2..n
12    LOOP
13        SELECT b,a+b INTO a,b;
14    END LOOP;
15    RETURN a;
16 END;
17 $$ 
18 LANGUAGE plpgsql;
```

The screenshot shows a database interface with two query panes and a data output viewer.

Top Query:

```
1 SELECT fib(3);
```

Data Output:

	fib	integer
1		1

Bottom Query:

```
1 SELECT fib(4);
```

Data Output:

	fib	integer
1		2

Цикл по элементам массива **FOREACH**

Отличается от *FOR* тем, что вместо перебора строк SQL-запроса происходит перебор элементов массива. Синтаксис цикла *FOREACH*:

```
[<<метка>>]
FOREACH цель [ SLICE число ] IN ARRAY выражение LOOP
    операторы
END LOOP [ метка ];
```

Без указания *SLICE* (или если *SLICE* равен 0) цикл выполняется по всем элементам массива, полученного из *выражения*. Переменной *цель* последовательно присваивается каждый элемент массива и для него выполняется тело цикла.

Пример использования цикла *FOREACH* – функция *findmax*, находящая максимальное значение в заданном целочисленном массиве. В отличие от *FOR*, цикл *FOREACH* может использовать переменную-счетчик, которая уже объявлена.

```

1 CREATE FUNCTION findmax(int[]) RETURNS int8
2 AS $$ 
3 DECLARE
4     max int8 := 0;
5     x int;
6 BEGIN
7     FOREACH x IN ARRAY $1
8 LOOP
9         IF x > max THEN max := x;
10        END IF;
11    END LOOP;
12    RETURN max;
13 END;
14 $$ LANGUAGE plpgsql;

```

1 SELECT findmax(ARRAY[1,2,3,4,5, -1]);

Data Output Messages Notifications

	findmax	bigint
1		5

При положительном значении *SLICE* выполняются итерации по срезам массива, а не по отдельным элементам. Значение *SLICE* должно быть целым числом, не превышающим размерности массива. Переменная *цель* должна быть массивом, который получает последовательные срезы исходного массива, где размерность каждого среза задается значением *SLICE*.

Пример цикла по одномерным срезам:

```

1 CREATE FUNCTION scan_rows(int[])
2 RETURNS void AS
3 $$ 
4 DECLARE
5     x int[];
6 BEGIN
7     FOREACH x SLICE 1 IN ARRAY $1
8 LOOP
9     RAISE NOTICE 'row = %', x;
10    END LOOP;
11 END;
12 $$ LANGUAGE plpgsql;

```

```
1 SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);
```

Data Output Messages Notifications

```
ЗАМЕЧАНИЕ: row = {1,2,3}
ЗАМЕЧАНИЕ: row = {4,5,6}
ЗАМЕЧАНИЕ: row = {7,8,9}
ЗАМЕЧАНИЕ: row = {10,11,12}
```

Обход элементов проводится в порядке их хранения, независимо от размерности массива. Цель – одиночная переменная, но может быть и списком переменных, когда элементы массива имеют составной тип (записи). Тогда переменным присваиваются значения из последовательных столбцов составного элемента массива.

4.9. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Любая возникающая ошибка прерывает выполнение функции и транзакцию, в которая она выполняется. Использование в блоке секции *EXCEPTION* позволяет перехватывать ошибки и обрабатывать их. Синтаксис блока с *EXCEPTION*:

```
[<<метка>>]
[DECLARE
  объявления]
BEGIN
  операторы
EXCEPTION
  WHEN условие [ OR условие ... ] THEN операторы_обработчика
  [ WHEN условие [ OR условие ... ] THEN операторы_обработчика ... ]
END;
```

Если ошибок нет, выполняются все операторы блока и управление переходит к следующему за *END* оператору. Если происходит ошибка, управление переходит к секции *EXCEPTION*, где ищется первое исключение, условие которого соответствует ошибке. Если исключение найдено,

выполняются *операторы_обработчика* и управление переходит к следующему за *END* оператору. Если исключение не найдено, ошибка передается наружу, как будто секции *EXCEPTION* не было. Ошибку можно перехватить в секции *EXCEPTION* внешнего блока. Если ошибка так и не была перехвачена, то обработка прекращается.

В качестве условия может задаваться одно из имен, перечисленных в документации (<https://postgrespro.ru/docs/postgresql/15/errcodes-appendix>). Если задано имя категории, ему соответствуют все ошибки в данной категории. Имена условий воспринимаются без учета регистра. Условие ошибки также можно задать кодом *SQLSTATE*, например, эти два варианта равнозначны:

WHEN division_by_zero THEN ... WHEN SQLSTATE '22012' THEN ...

Если при выполнении *операторов_обработчика* возникнет новая ошибка, то она не может быть перехвачена в этой секции *EXCEPTION*. Ошибка передается наружу и ее можно перехватить в секции *EXCEPTION* внешнего блока.

При возникновении исключительной ситуации PostgreSQL выполняет откат базы данных в состояние на момент начала выполнения блока, в котором предусмотрена обработка этой исключительной ситуации и внутри которого она возникла. Если такого блока нет, выполняется откат транзакции, в которой возникла исключительная ситуация. При выполнении команд в секции *EXCEPTION* локальные переменные функции сохранят значения на момент возникновения ошибки. Но все изменения в базе данных, выполненные в блоке, будут отменены. Пример:

```
1  INSERT INTO mytab(firstname, lastname) VALUES('Иван', 'Петров');
2 ▼ BEGIN
3      UPDATE mytab SET firstname = 'Сергей' WHERE lastname = 'Петров';
4      x := x + 1;
5      y := x / 0;
6  EXCEPTION
7      WHEN division_by_zero THEN
8          RAISE NOTICE 'перехвачена ошибка division_by_zero';
9          RETURN x;
10 END;
```

В строке 5 возникнет ошибка *division_by_zero*, которая будет перехвачена в секции *EXCEPTION*. Оператор *RETURN* вернет значение *x*, увеличенное на единицу, но изменения, сделанные командой *UPDATE*, будут отменены. Изменения, выполненные предшествующей командой *INSERT*, отменены не будут. Еще один пример:

```
1 CREATE OR REPLACE FUNCTION empl_name(e_code integer)
2 RETURNS text
3 LANGUAGE plpgsql AS $$ 
4 DECLARE
5     v text;
6 BEGIN
7     SELECT emp_name
8     INTO v
9     FROM employees
10    WHERE emp_ID = e_code;
11 IF NOT FOUND THEN
12     RETURN '-- Неправильный код сотрудника --';
13 END IF;
14 RETURN v;
15 END;
16 $$;
```

```
1 SELECT empl_name(5);
```

Data Output Messages Notifications

	empl_name	
1	-- Неправильный код сотрудника --	🔒

ВНИМАНИЕ!

Наличие секции *EXCEPTION* значительно увеличивает накладные расходы на вход/выход из блока, поэтому не используйте *EXCEPTION* без надобности.

Выполняйте дополнительные проверки с помощью условных операторов.

В приведенном выше примере предотвращается возбуждение исключительных ситуаций за счет такой дополнительной проверки. Такой же результат можно получить, проверяя наличие строк в ответе на запрос вместо применения ключевого слова *STRICT*.

4.10. ПУСТОЙ ОПЕРАТОР

В некоторых случаях полезен оператор, который не делает ничего. Например, он может показывать, что одна из ветвей *IF-THEN-ELSE* сознательно оставлена пустой. Для этих целей используется оператор *NULL*.

Следующие два фрагмента кода эквивалентны (выбор – дело вкуса):

```
1 ▾ BEGIN
2     y := x / 0;
3 EXCEPTION
4     WHEN division_by_zero THEN
5         NULL; -- ошибка игнорируется
6 END;
```

```
1 ▾ BEGIN
2     y := x / 0;
3 EXCEPTION
4     WHEN division_by_zero THEN -- ошибка игнорируется
5 END;
```

ВНИМАНИЕ!

Oracle PL/SQL не допускает пустые списки операторов, поэтому *NULL* в подобных ситуациях обязательен.
PL/pgSQL разрешает не писать ничего.

Завершая краткий обзор управляющих конструкций языка PL/pgSQL, отметим, что оператора безусловного перехода *GO TO* в языке нет.

5. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Лабораторная работа № 1

ПРОЕКТИРОВАНИЕ БАЗЫ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ ER-ТЕХНОЛОГИИ

Теоретические сведения

Для заданной предметной области: необходимо определить состав реляционных таблиц и логические связи между таблицами; для каждого атрибута следует задать тип и размер данных, ограничения целостности; для каждой таблицы – первичный ключ, потенциальные ключи и внешние ключи (при необходимости).

Практическая работа

Для выданного преподавателем варианта предметной области необходимо:

- выделить сущности, атрибуты и связи предметной области;
- построить модель данных в одной из принятых нотаций.

Пример выполнения задания

Рассмотрим упрощенный бизнес-процесс компании, связанный с оформлением заказов на товары. В этой предметной области можно выделить следующие сущности: товар, категория, производитель, заказчик, заказ, элемент заказа, сотрудник.

В предметной области действуют следующие бизнес-правила. Заказчик может делать множество заказов. Анонимные заказы не выполняются. Каждый заказ обслуживается одним сотрудником компании. Заказ может состоять из множества элементов. Элемент заказа включает код заказа, код товара и количество данного товара в заказе. Товар относится к одной категории, при этом в одной категории может быть несколько товаров. У категории может быть надкатегория. Товар содержит сведения о производителе. Остаток товара не может быть отрицательным. После заказа то-

вара остаток его на складе уменьшается соответствующим образом. При добавлении товара на склад остаток товара соответственно увеличивается.

Логическая модель данных в нотации IDEF1X представлена на рис. 16. Выделены сущности *ТОВАР*, *КАТЕГОРИЯ*, *ПРОИЗВОДИТЕЛЬ*, *ЗАКАЗЧИК*, *ЗАКАЗ*, *ЭЛЕМЕНТ ЗАКАЗА*, *СОТРУДНИК*, между которыми установлены неидентифицирующие связи мощностью «один-ко-многим». Есть рекурсивная связь, связывающая экземпляры сущности *КАТЕГОРИЯ* с другими экземплярами этой же сущности.

Физическая модель данных в нотации IDEF1X для целевой СУБД PostgreSQL представлена на рис. 17. База данных *Sales* состоит из семи таблиц: *Customers* – список заказчиков, *Employees* – список сотрудников, *Orders* – список заказов, *Items* – список элементов заказов, *Products* – список товаров, *Manufacturers* – список производителей товаров, *Categories* – список категорий товаров.

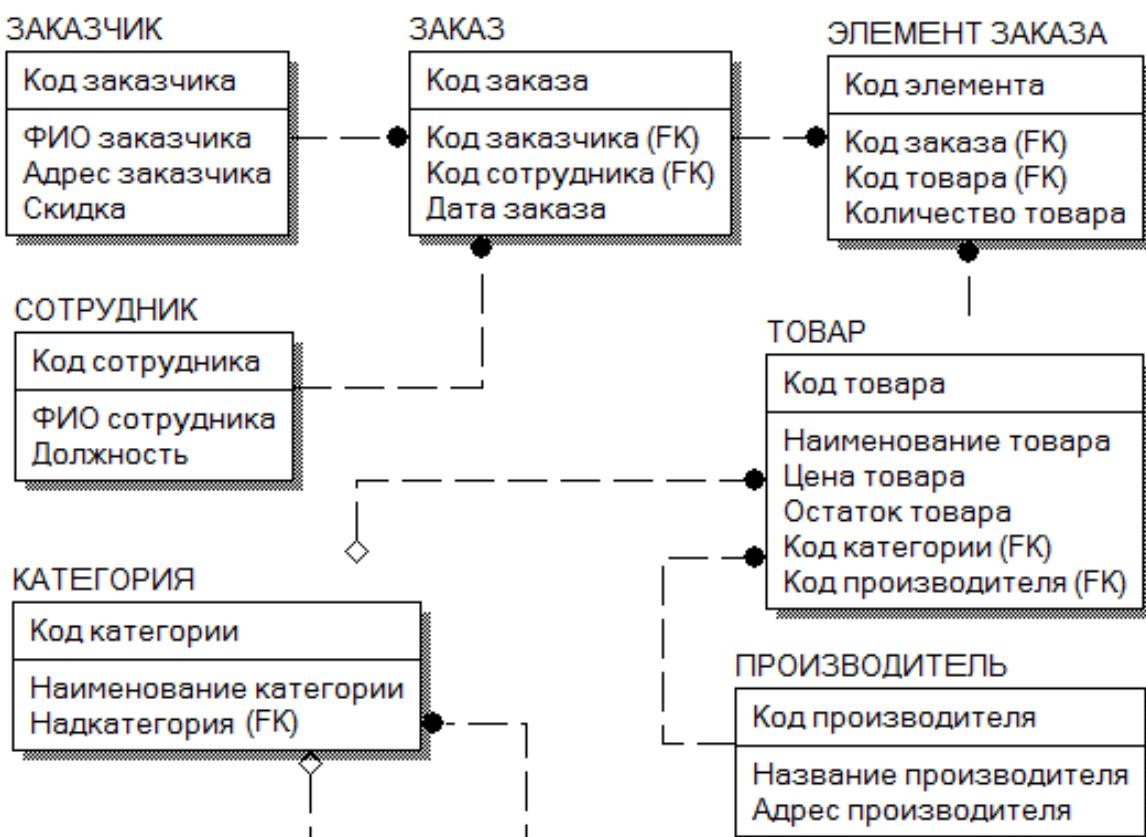


Рис. 16. Логическая модель данных предметной области

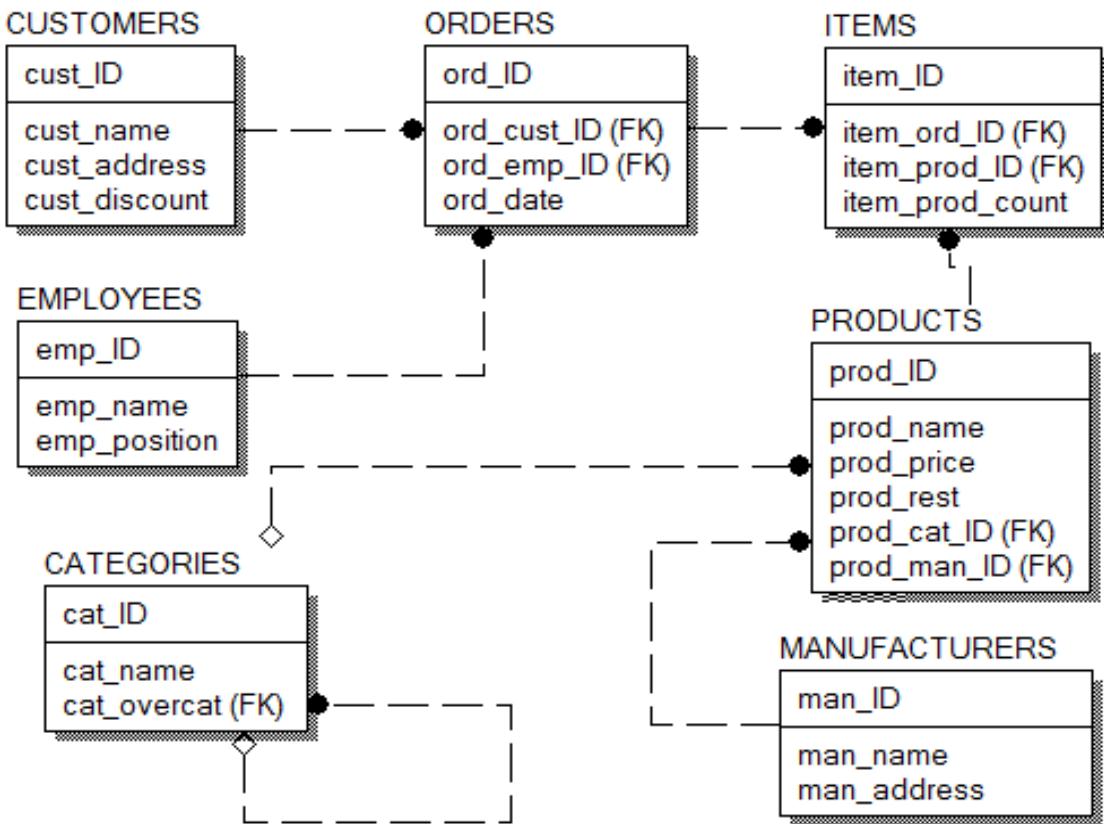


Рис. 17. Физическая модель предметной области

Таблица *Categories* состоит из трех полей:

- *cat_ID* – уникальный код категории, тип *integer*, *NOT NULL*;
- *cat_name* – наименование категории, тип *text*, *NOT NULL*;
- *cat_overcat* – код надкатегории из этой же таблицы, тип *integer*.

Поле *cat_ID* имеет ограничение первичного ключа *PRIMARY KEY*.

Поле *cat_overcat* не имеет атрибута *NOT NULL*, поскольку не все категории имеют надкатегорию. Это поле устанавливает связь между категориями, поэтому его нужно объявить как внешний ключ (FK) с правилом каскадного удаления и обновления. Удаление (обновление) надкатегории в таблице *Categories* приведет к автоматическому каскадному удалению (обновлению) всех связанных с ней категорий.

Таблица *Manufacturers* включает три поля:

- *man_ID* – уникальный код производителя, тип *integer*, *NOT NULL*;
- *man_name* – наименование производителя, тип *text*, *NOT NULL*;
- *man_address* – адрес производителя, тип *text*, *NOT NULL*.

Поле *man_ID* имеет ограничение первичного ключа *PRIMARY KEY*.

Таблица *Products* состоит из шести полей:

- *prod_ID* – уникальный код товара, тип *integer*, *NOT NULL*;
- *prod_name* – наименование товара, тип *text*, *NOT NULL*;
- *prod_price* – цена товара, тип *numeric*, *NOT NULL*;
- *prod_rest* – остаток товара, тип *integer*, *NOT NULL*;
- *prod_cat_ID* – код категории товара из таблицы *Categories*, тип *integer*, *NOT NULL*;
- *prod_man_ID* – код производителя товара из таблицы *Manufacturers*, тип *integer*, *NOT NULL*.

Поле *prod_ID* имеет ограничение первичного ключа *PRIMARY KEY*.

Таблица *Products* связана с таблицами *Categories* (за счет поля *prod_cat_ID*) и *Manufacturers* (за счет поля *prod_man_ID*). Эти поля объявлены как внешние ключи (FK) с правилом каскадного удаления и обновления. Обновление таблиц *Categories* и *Manufacturers* приведет к автоматическому обновлению таблицы *Products*. Удаление производителя в таблице *Manufacturers* приведет к автоматическому удалению всех записей в таблице *Products*, соответствующих этому производителю. Удаление категории в таблице *Categories* приведет к автоматическому удалению всех записей в таблице *Products*, ссылающихся на эту категорию.

Таблица *Customers* включает четыре поля:

- *cust_ID* – уникальный код заказчика, тип *integer*, *NOT NULL*;
- *cust_name* – ФИО заказчика, тип *text*, *NOT NULL*;
- *cust_address* – адрес заказчика, тип *text*, *NOT NULL*;
- *cust_discount* – скидка в процентах, предоставляемая заказчику, тип *integer*.

Поле *cust_ID* имеет ограничение первичного ключа *PRIMARY KEY*.

Таблица *Employees* включает три поля:

- *emp_ID* – уникальный код сотрудника, тип *integer*, *NOT NULL*;
- *emp_name* – ФИО сотрудника, тип *text*, *NOT NULL*;
- *emp_position* – должность сотрудника, тип *text*, *NOT NULL*.

Поле *emp_ID* имеет ограничение первичного ключа *PRIMARY KEY*.

Таблица *Orders* включает четыре поля:

- *ord_ID* – уникальный код заказа, тип *integer*, *NOT NULL*;
- *ord_cust_ID* – код заказчика из таблицы *Customers*, тип *integer*, *NOT NULL*;
- *ord_emp_ID* – код сотрудника из таблицы *Employees*, тип *integer*, *NOT NULL*;
- *ord_date* – дата заказа, тип *date*, *NOT NULL*.

Поле *order_ID* должно иметь ограничение первичного ключа *PRIMARY KEY*. Таблица *Orders* связана с таблицами *Customers* (за счет поля *ord_cust_ID*) и *Employees* (за счет поля *ord_emp_ID*). Эти поля объявлены как внешние ключи (FK) с правилом каскадного удаления и обновления.

Обновление таблиц *Customers* и *Employees* приведет к автоматическому обновлению таблицы *Orders*. Удаление заказчика в таблице *Customers* приведет к автоматическому удалению всех записей в таблице *Orders*, соответствующих этому заказчику. Удаление сотрудника в таблице *Employees* приведет к автоматическому удалению всех записей в таблице *Orders*, ссылающихся на этого сотрудника.

Таблица *Items* включает четыре поля:

- *item_ID* – уникальный код элемента заказа, тип *integer*, *NOT NULL*;
- *item_ord_ID* – код заказа из таблицы *Orders*, тип *integer*, *NOT NULL*;
- *item_prod_ID* – код товара из таблицы *Products*, тип *integer*, *NOT NULL*;
- *item_prod_count* – количество покупаемых товаров, тип *integer*, *NOT NULL*.

Поле *item_ID* имеет ограничение первичного ключа *PRIMARY KEY*.

Таблица *Items* связана с таблицами *Orders* (за счет поля *item_ord_ID*) и *Products* (за счет поля *item_prod_ID*). Эти поля объявлены как внешние ключи (FK) с правилом каскадного удаления и обновления.

Обновление таблиц *Orders* и *Products* приведет к автоматическому обновлению таблицы *Items*. Удаление заказа в таблице *Orders* приведет к автоматическому удалению всех записей в таблице *Items*, соответствующих этому заказу. Удаление товара в таблице *Products* приведет к автоматическому удалению всех записей в таблице *Items*, ссылающихся на этот товар.

Лабораторная работа № 2

СОЗДАНИЕ НОВОЙ БАЗЫ ДАННЫХ В СРЕДЕ PostgreSQL

Теоретические сведения

Рассматриваются следующие вопросы:

- создание, изменение, выбор и удаление баз данных;
- создание, изменение и удаление схем базы данных.

Базы данных и схемы. *База данных* – это контейнер для других объектов (таблиц, представлений, функций, индексов и пр.) На сервере PostgreSQL можно создать столько баз данных, сколько необходимо (рис. 18).



Рис. 18. Базы данных в браузере pgAdmin

Схема – это логический контейнер таблиц и других объектов внутри базы данных (логический фрагмент базы данных). Схемы являются частью стандарта ANSI-SQL. В каждой базе данных обязательно есть хотя бы одна схема. При создании базы данных в ней автоматически создается схема с именем *public*. Каждая база данных может иметь несколько схем (рис. 19).

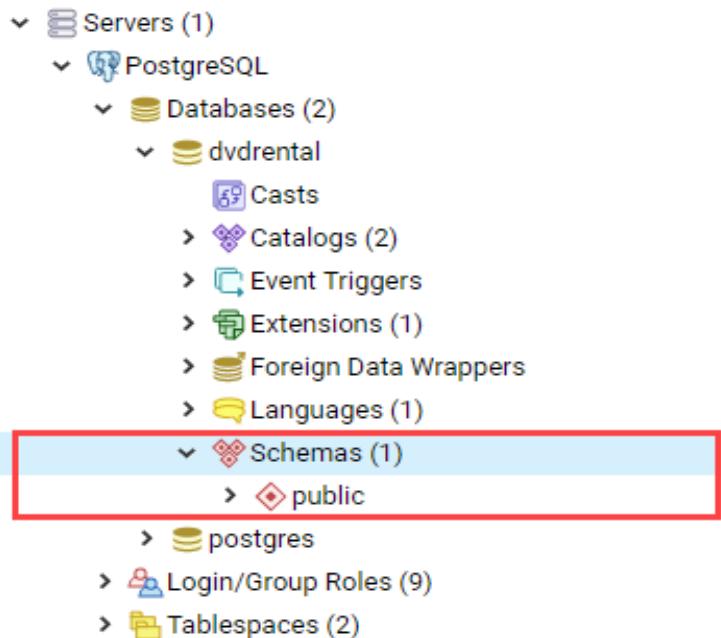


Рис. 19. Схемы в браузере pgAdmin

В каждой базе данных может содержаться более одной схемы. Имена схем должны быть уникальными в пределах этой базы данных. Имена объектов базы данных должны быть уникальными в пределах конкретной схемы, т. е. схема образует *пространство имен*. При создании объектов базы данных необходимо учитывать следующее: если имя схемы в инструкции не указано, то объект будет создан в текущей схеме. Если нужно создать объект в схеме, которая не является текущей, то следует указать ее имя перед именем создаваемого объекта, разделив их точкой.

Табличные пространства. Если базы данных и схемы определяют логическое распределение данных в кластере, то табличные пространства относятся к физическому расположению данных (определяют каталоги для файлов базы данных).

Объекты базы данных могут храниться в разных табличных пространствах. Табличное пространство PostgreSQL позволяет легко перемещать данные в разные физические места с помощью простых команд. По умолчанию PostgreSQL предоставляет два табличных пространства (рис. 20):

- *pg_default* – для хранения пользовательских данных;
- *pg_global* – для хранения системных данных.

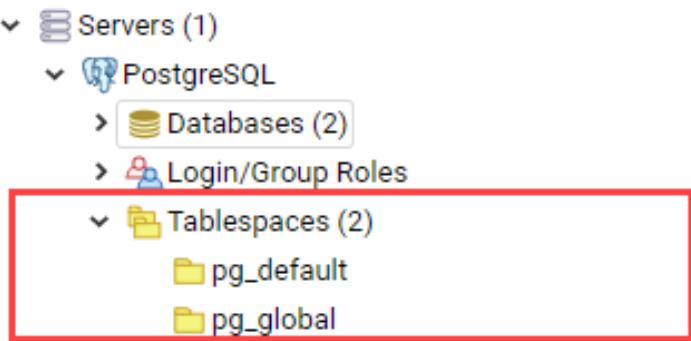


Рис. 20. Табличные пространства в навигаторе pgAdmin

PostgreSQL позволяет создать свои табличные пространства и использовать их для каких-либо объектов (баз данных, таблиц и др.).

У любой базы данных есть табличное пространство по умолчанию. В нем она располагает таблицы и другие объекты при их создании (если явно не указано другое табличное пространство). Табличное пространство по умолчанию для базы определяется шаблоном, из которого копируется новая база. Если табличное пространство по умолчанию в шаблоне изменено, то и новые базы данных, созданные из этого шаблона, получат новое табличное пространство по умолчанию.

Создание базы данных. База данных создается следующей инструкцией [12]:

CREATE DATABASE имя_базы_данных [[WITH] параметр [...]];

Имя базы данных подчиняется правилам именования идентификаторов SQL. Текущий пользователь автоматически назначается владельцем. Владелец может удалить свою базу, что также приведет к удалению всех ее объектов, в том числе имеющих других владельцев.

Изменение атрибутов базы данных. Изменить атрибуты базы данных можно с помощью инструкции [12]:

ALTER DATABASE имя_базы_данных [[WITH] параметр [...]]

Полный синтаксис инструкции можно найти в [12]. Изменения будут выполнены либо для базы данных, указанной по имени, либо для текущей базы данных.

Удаление базы данных. Реализуется следующей инструкцией [12]:

DROP DATABASE [IF EXISTS] имя [[WITH] (параметр [, ...])]

Инструкция удаляет базу данных. Она удаляет из каталогов записи, относящиеся к базе данных, а также удаляет каталог, содержащий данные. Удалить базу данных может только ее владелец. Удалить базу нельзя, пока к ней кто-либо подключен. В этом случае следует подключиться к *postgres*, любой другой базе данных или использовать параметр *FORCE* (будет сделана попытка разорвать все существующие соединения с целевой базой данных).

Параметр *IF EXISTS* не считать ошибкой, если база данных не существует. В этом случае будет выдано замечание. Действие инструкции нельзя отменить. Инструкцию *DROP DATABASE* нельзя выполнять внутри блока транзакции.

Создание схемы. Для создания схемы используется инструкция [12]:

CREATE SCHEMA schema_name [AUTHORIZATION role_specification] [schema_element [...]]

CREATE SCHEMA AUTHORIZATION role_specification [schema_element [...]]

CREATE SCHEMA IF NOT EXISTS schema_name [AUTHORIZATION role_specification]

CREATE SCHEMA IF NOT EXISTS AUTHORIZATION role_specification
где *role_specification* может быть:

user_name | CURRENT_ROLE | CURRENT_USER | SESSION_USER

Инструкция вводит новую схему в текущую базу данных. Предложение *AUTHORIZATION* задает владельца схемы. В процессе создания схемы в ней можно сформировать объекты (таблицы, представления). Подкоманды обрабатываются как отдельные команды, выдаваемые после создания схемы, но если используется *AUTHORIZATION*, то созданные объекты будут принадлежать этому владельцу.

Модификация схемы. Изменение схемы осуществляется инструкциями [12]:

ALTER SCHEMA name RENAME TO new_name

ALTER SCHEMA name OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }

Использовать инструкцию может владелец схемы. Чтобы переименовать схему, необходимо также иметь привилегию *CREATE* для базы данных. Чтобы изменить владельца, нужно быть прямым или косвенным членом новой роли – владельца и иметь привилегию *CREATE* для базы данных. Суперпользователи получают все эти привилегии автоматически.

Удаление схемы. Удаление схемы выполняется инструкцией [12]:

DROP SCHEMA [IF EXISTS] name [, ...] [CASCADE | RESTRICT]

Схема может быть удалена только ее владельцем или суперпользователем. Владелец может удалить схему (и все содержащиеся в ней объекты), даже если он не владеет некоторыми объектами в схеме. При использовании *IF EXISTS* не возникает ошибки, если схема не существует.

CASCADE – автоматически удалять объекты (таблицы, функции и пр.), содержащиеся в схеме, и, в свою очередь, все объекты, зависящие от этих объектов. Использование этой опции может привести к тому, что команда удалит объекты в других схемах, кроме названной.

RESTRICT – не удалять схему, если она содержит какие-либо объекты (значение по умолчанию).

Практическая работа

При выполнении задания необходимо для заданной предметной области средствами PostgreSQL:

- создать новую базу данных *sales*;
- составить отчет.

Пример выполнения задания

Результатом выполнения задания является база данных *sales*, созданная путем выполнения следующего SQL-запроса в программе *pgAdmin*:

```
1 -- Database: sales
2
3 -- DROP DATABASE IF EXISTS sales;
4
5 CREATE DATABASE sales
6   WITH
7     OWNER = postgres
8     ENCODING = 'UTF8'
9     LC_COLLATE = 'Russian_Russia.1251'
10    LC_CTYPE = 'Russian_Russia.1251'
11    TABLESPACE = pg_default
12    CONNECTION LIMIT = -1
13    IS_TEMPLATE = False;
14
15 COMMENT ON DATABASE sales
16   IS 'База данных для службы оформления заказов на товары';
```

Создается база данных *sales* со следующими характеристиками:

- владелец базы данных – суперпользователь *postgres*;
- кодировка *UTF-8*, в которой хранится текст;
- порядок сортировки строк определяется локалью *Russian_Russia.1251*;
- классификация символов определяется локалью *Russian_Russia.1251*;
- табличное пространство базы данных – *pg_default*;
- отсутствует ограничение на количество одновременных подключений;
- клонировать ее могут только суперпользователи и ее владелец (*false*); при *true* может клонировать любой пользователь с правами *CREATEDB*;
- комментарий – база данных для службы оформления заказов на товары.

Лабораторная работа № 3

Создание и связывание таблиц базы данных в среде PostgreSQL

Теоретические сведения

Рассматриваются следующие вопросы:

- создание таблиц с помощью инструкции *CREATE TABLE*;
- изменение структуры таблиц с помощью инструкции *ALTER TABLE*;
- удаление таблиц с помощью инструкции *DROP TABLE*.

Создание таблиц. Создавать таблицы может любой пользователь, имеющий на это право или обладающий ролью владельца базы или системного администратора.

При создании таблиц важно грамотно подойти к процедуре определения типа данных. В PostgreSQL предоставлен большой выбор встроенных типов данных (см. Приложение). Кроме того, пользователи могут создавать свои типы, используя команду *CREATE TYPE*.

Упрощенный синтаксис инструкции по созданию таблиц [12]:

CREATE TABLE имя_таблицы

```
(имя_поля тип_данных [ограничения_целостности],  
имя_поля тип_данных [ограничения_целостности],  
...  
имя_поля тип_данных [ограничения_целостности],  
[ограничение_целостности],  
[первичный_ключ],  
[внешний_ключ])
```

Инструкция создает новую пустую таблицу в текущей БД. Таблица будет принадлежать пользователю, выполнившему команду. Если указано имя схемы, то таблица создается в этой схеме (иначе – в текущей). Временные таблицы существуют в специальной схеме, поэтому имя схемы для них указать нельзя.

Имя таблицы должно отличаться от имени другой таблицы, последовательности, индекса, представления или внешней таблицы в той же схеме. После имени таблицы в круглых скобках перечисляются столбцы и ограничения таблицы. Предложения ограничений определяют набор допустимых значений в таблице.

Реализованы два способа определения ограничений: *ограничения столбца* и *ограничения таблицы*. Ограничение столбца является частью определения столбца. Ограничение таблицы не привязано к конкретному столбцу. Любое ограничение столбца может быть реализовано как ограничение таблицы.

Ограничение можно сделать именованным. В этом случае используют ключевое слово *CONSTRAINT*, за которым следует идентификатор и определение ограничения. Если не указать имя ограничения, то система выберет его сама.

В PostgreSQL реализованы следующие ограничения:

- ограничение проверки *CHECK*;
- ограничение *NOT NULL*;
- ограничение уникальности *UNIQUE*;
- ограничение первичного ключа *PRIMARY KEY*;
- ограничение внешнего ключа *FOREIGN KEY*;
- ограничение исключения *EXCLUSION*.

Ограничение *CHECK* является наиболее общим типом ограничения. Оно позволяет потребовать, чтобы значение в определенном столбце удовлетворяло некоторому условию.

Ограничение *NOT NULL* указывает, что столбец не должен принимать неопределенное значение *NULL*. Это ограничение всегда записывается как ограничение столбца. Недостаток – таким ограничениям нельзя давать явные имена.

Ограничение *UNIQUE* гарантирует, что данные, содержащиеся в столбце или группе столбцов, уникальны среди всех строк таблицы. Добавление

ограничения *UNIQUE* автоматически создает уникальный индекс В-дерева для столбца или группы столбцов, перечисленных в ограничении.

Ограничение первичного ключа *PRIMARY KEY* указывает, что столбец (группу столбцов) можно использовать как уникальный идентификатор строк в таблице (сочетание *UNIQUE* и *NOT NULL*). Добавление первичного ключа автоматически создает уникальный индекс В-дерева для столбца (группы столбцов) первичного ключа и пометит столбцы как *NOT NULL*. Таблица может иметь один первичный ключ. Возможно любое количество сочетаний *UNIQUE* и *NOT NULL*, но только одно может быть идентифицировано как первичный ключ.

Ограничение внешнего ключа *FOREIGN KEY* указывает, что значения в столбце (группе столбцов) должны совпадать со значениями, отображаемыми в некоторой строке другой таблицы. Это поддерживает *ссылочную целостность* между двумя связанными таблицами. Предложения *ON DELETE* и *ON UPDATE* определяют, что произойдет со строками дочерней таблицы при удалении связанной строки родительской таблицы или при изменении значения любого столбца в ключе родительской таблицы. Возможные варианты:

- *NO ACTION* – нет действий (по умолчанию); нарушается ссылочная целостность базы данных и необходимо каким-либо образом реагировать;
- *RESTRICT* – запрещается удалять кортеж, на который есть ссылки (сначала нужно удалить ссылающиеся кортежи или изменить значения их внешнего ключа);
- *CASCADE* – удаление всех связанных строк дочерней таблицы (изменение значений реквизитов внешнего ключа всех связанных строк дочерней таблицы);
- *SET NULL* – столбцы внешнего ключа всех связанных строк дочерней таблицы устанавливаются в *NULL* (строки дочерней таблицы остаются в базе);
- *SET DEFAULT* – столбцы внешнего ключа дочерней таблицы устанавливаются в значение по умолчанию.

Наиболее распространенные варианты – ограничение и каскадное удаление.

ВНИМАНИЕ!

Чтобы ограничения внешнего ключа соблюдались всегда, объягите столбец (столбцы) внешнего ключа как NOT NULL.

Внешний ключ должен ссылаться на столбцы, которые являются первичным ключом или имеют ограничение уникальности. Это означает, что столбцы, на которые ссылаются, всегда имеют индекс.

ВНИМАНИЕ!

Объявление ограничения внешнего ключа не создает автоматически индекс для столбцов внешнего ключа.

Добавление ограничения исключения *EXCLUSION* автоматически создаст индекс типа, указанного в объявлении ограничения.

Генерируемые столбцы. Столбцы, вычисляемые по значениям других столбцов. Есть два типа генерируемых столбцов: хранимые и виртуальные. Хранимый столбец вычисляется при его записи (вставке или обновлении) и занимает память как обычный столбец. Виртуальный столбец не занимает памяти и вычисляется при чтении. На данном шаге PostgreSQL реализует только хранимые генерируемые столбцы.

Изменение таблиц. Таблицы изменяются инструкцией *ALTER TABLE*, полный синтаксис которой можно найти в [12]. Приведем наиболее распространенные действия, связанные с модификацией таблиц. Добавление столбца:

```
ALTER TABLE имя_таблицы ADD COLUMN  
имя_поля тип_данных [ограничения_целостности];
```

Можно одновременно определить ограничения для столбца, используя все параметры, которые можно применить к описанию столбца

CREATE TABLE. Однако значение по умолчанию должно удовлетворять заданным ограничениям.

Добавить ограничение можно, используя синтаксис ограничения таблицы:

ALTER TABLE имя_таблицы ADD CHECK (ограничение_ целостности);

ALTER TABLE имя_таблицы ADD CONSTRAINT имя_ограничения UNIQUE (имя_поля);

ALTER TABLE имя_таблицы ADD FOREIGN KEY (имя_поля) REFERENCES имя_таблицы;

Ограничение будет проверено немедленно, поэтому данные таблицы должны удовлетворять ограничению, прежде чем можно будет добавить новые данные.

Чтобы удалить ограничение, нужно знать его имя. Если имя не было задано, то система присваивает сгенерированное имя, которое необходимо узнать.

ALTER TABLE имя_таблицы DROP CONSTRAINT имя_ограничения;

При работе со сгенерированным именем ограничения, например \$2, его следует заключить в двойные кавычки, чтобы сделать допустимым идентификатором.

Для удаления столбца используют инструкцию:

ALTER TABLE имя_таблицы DROP COLUMN имя_поля [CASCADE];

Данные в столбце исчезнут вместе со столбцом. Ограничения таблицы, касающиеся столбца, также удаляются (это не относится к столбцу, на который ссылается ограничение внешнего ключа другой таблицы). Можно разрешить удаление всего, что зависит от столбца, добавив *CASCADE*.

Изменить тип данных столбца можно с помощью инструкции:

ALTER TABLE имя_таблицы ALTER COLUMN имя_поля

TYPE новый_тип_данных;

Это удастся, если каждое значение в столбце может быть преобразовано к новому типу с помощью неявного приведения. Для сложных случаев можно добавить предложение *USING*, указывающее, как вычислять новые значения из старых.

Удаление ограничений *NOT NULL* (у которых нет имен):

ALTER TABLE имя_таблицы ALTER COLUMN имя_поля DROP NOT NULL;

Новое значение по умолчанию для столбца устанавливается инструкцией:

ALTER TABLE имя_таблицы ALTER COLUMN имя_поля SET DEFAULT новое_значение;

Это не повлияет на существующие строки в таблице, а просто изменит значение по умолчанию для будущих инструкций *INSERT*.

Удалить любое значение по умолчанию можно с помощью инструкции:

ALTER TABLE имя_таблицы ALTER COLUMN имя_поля DROP DEFAULT;

Фактически это то же самое, что установить по умолчанию значение *NULL*. Не будет ошибкой удалять значение по умолчанию там, где оно не было определено, поскольку значение по умолчанию неявно является значением *NULL*.

Добавить ограничение *NOT NULL*, которое нельзя записать как табличное ограничение, можно следующим образом:

ALTER TABLE имя_таблицы ALTER COLUMN имя_поля SET NOT NULL;

Переименовать столбец позволяет инструкция:

ALTER TABLE имя_таблицы RENAME COLUMN имя_поля TO новое_имя_поля;

Переименовать таблицу позволяет инструкция:

ALTER TABLE имя_таблицы RENAME TO новое_имя_таблицы;

Удаление таблиц. Осуществляется инструкцией *DROP TABLE* [12], пользоваться которой следует с осторожностью. Нельзя удалить таблицу, которая является ссылочной для другой таблицы в базе данных. Для удаления зависимых объектов необходимо использовать инструкцию каскадного удаления зависимых объектов:

DROP TABLE имя_таблицы CASCADE;

Из ссылающейся таблицы будет удален внешний ключ, ссылающийся на удаляемую таблицу. Если возможна попытка удаления несуществующей таблицы, тогда:

DROP TABLE IF EXISTS имя_таблицы CASCADE;

При этом в случае наличия таблицы выполняется ее удаление, а в случае ее отсутствия выводится замечание, а не ошибка.

Практическая работа

При выполнении задания необходимо для заданной предметной области:

- создать таблицы, определить поля таблиц и тип полей;
- определить связи между таблицами и ограничения целостности;
- составить отчет.

Пример выполнения задания

Инструкции создания таблиц в базе данных *sales* приведены ниже.

Ограничения (кроме *NOT NULL* и *PRIMARY KEY*) добавляются в отдельных запросах *ALTER TABLE*. Благодаря этому запрос *CREATE TABLE* проще понять без ограничений, а независимое определение ограничений облегчает их удаление и модификацию.

```

1 -----  

2 -- Создание таблицы categories  

3 -----  

4 CREATE TABLE categories (  

5     cat_ID integer NOT NULL,  

6     cat_name text NOT NULL,  

7     cat_overcat integer,  

8     CONSTRAINT categories_PK PRIMARY KEY (cat_ID)  

9 );  

1 -----  

2 -- Создание таблицы manufacturers  

3 -----  

4 CREATE TABLE manufacturers (  

5     man_ID integer NOT NULL,  

6     man_name text NOT NULL,  

7     man_address text NOT NULL,  

8     CONSTRAINT manufacturers_PK PRIMARY KEY (man_ID)  

9 );  

1 -----  

2 -- Создание таблицы products  

3 -----  

4 CREATE TABLE products (  

5     prod_ID integer NOT NULL,  

6     prod_name text NOT NULL,  

7     prod_price numeric(8,2) NOT NULL,  

8     prod_rest integer NOT NULL,  

9     prod_cat_ID integer NOT NULL,  

10    prod_man_ID integer NOT NULL,  

11    CONSTRAINT products_PK PRIMARY KEY (prod_ID)  

12 );  

1 -----  

2 -- Создание таблицы customers  

3 -----  

4 CREATE TABLE customers (  

5     cust_ID integer NOT NULL,  

6     cust_name text NOT NULL,  

7     cust_address text NOT NULL,  

8     cust_discount integer NOT NULL,  

9     CONSTRAINT customers_PK PRIMARY KEY (cust_ID)  

10 );

```

```
1 -----  
2 -- Создание таблицы employees  
3 -----  
4 CREATE TABLE employees (  
5     emp_ID integer NOT NULL,  
6     emp_name text NOT NULL,  
7     emp_position text NOT NULL,  
8     CONSTRAINT employees_PK PRIMARY KEY (emp_ID)  
9 );
```

```
1 -----  
2 -- Создание таблицы orders  
3 -----  
4 CREATE TABLE orders (  
5     ord_ID integer NOT NULL,  
6     ord_cust_ID integer NOT NULL,  
7     ord_emp_ID integer NOT NULL,  
8     ord_date date NOT NULL,  
9     CONSTRAINT orders_PK PRIMARY KEY (ord_ID)  
10 );
```

```
1 -----  
2 -- Создание таблицы items  
3 -----  
4 CREATE TABLE items (  
5     item_ID integer NOT NULL,  
6     item_ord_ID integer NOT NULL,  
7     item_prod_ID integer NOT NULL,  
8     item_prod_count integer NOT NULL,  
9     CONSTRAINT items_PK PRIMARY KEY (item_ID)  
10 );
```

ВНИМАНИЕ!

Чтобы созданные таблицы появились в браузере *pgAdmin*, необходимо в нем щелкнуть правой кнопкой мыши на строке *PostgreSQL 15* и выбрать в контекстном меню команду *Refresh*.

```
1 -----  
2 -- Определение внешних ключей  
3 -----  
4 ALTER TABLE categories  
5 ADD CONSTRAINT categories_FK FOREIGN KEY (cat_overcat)  
6 REFERENCES categories (cat_ID)  
7 ON DELETE CASCADE  
8 ON UPDATE CASCADE;  
9  
10 ALTER TABLE products  
11 ADD CONSTRAINT products_FK1 FOREIGN KEY (prod_cat_ID)  
12 REFERENCES categories (cat_ID)  
13 ON DELETE CASCADE  
14 ON UPDATE CASCADE;  
15  
16 ALTER TABLE products  
17 ADD CONSTRAINT products_FK2 FOREIGN KEY (prod_man_ID)  
18 REFERENCES manufacturers (man_ID)  
19 ON DELETE CASCADE  
20 ON UPDATE CASCADE;
```

```
1 ALTER TABLE orders  
2 ADD CONSTRAINT orders_FK1 FOREIGN KEY (ord_cust_ID)  
3 REFERENCES customers (cust_ID)  
4 ON DELETE CASCADE  
5 ON UPDATE CASCADE;  
6  
7 ALTER TABLE orders  
8 ADD CONSTRAINT orders_FK2 FOREIGN KEY (ord_emp_ID)  
9 REFERENCES employees (emp_ID)  
10 ON DELETE CASCADE  
11 ON UPDATE CASCADE;  
12  
13 ALTER TABLE items  
14 ADD CONSTRAINT items_FK1 FOREIGN KEY (item_ord_ID)  
15 REFERENCES orders (ord_ID)  
16 ON DELETE CASCADE  
17 ON UPDATE CASCADE;  
18  
19 ALTER TABLE items  
20 ADD CONSTRAINT items_FK2 FOREIGN KEY (item_prod_ID)  
21 REFERENCES products (prod_ID)  
22 ON DELETE CASCADE  
23 ON UPDATE CASCADE;
```

Объединив инструкции SQL, сгенерированные в процессе создания и связывания таблиц базы данных *sales*, можно получить сценарий (скрипт), позволяющий оперативно создать базу данных *sales* в среде PostgreSQL.

Построим ER-диаграмму базы данных *sales*. В главном меню выберем команду *Tools > ERD Tool*, щелкнем правой кнопкой мыши на базе данных *sales* и в контекстном меню выберем *ERD for Database*. Результат показан на рис. 21.

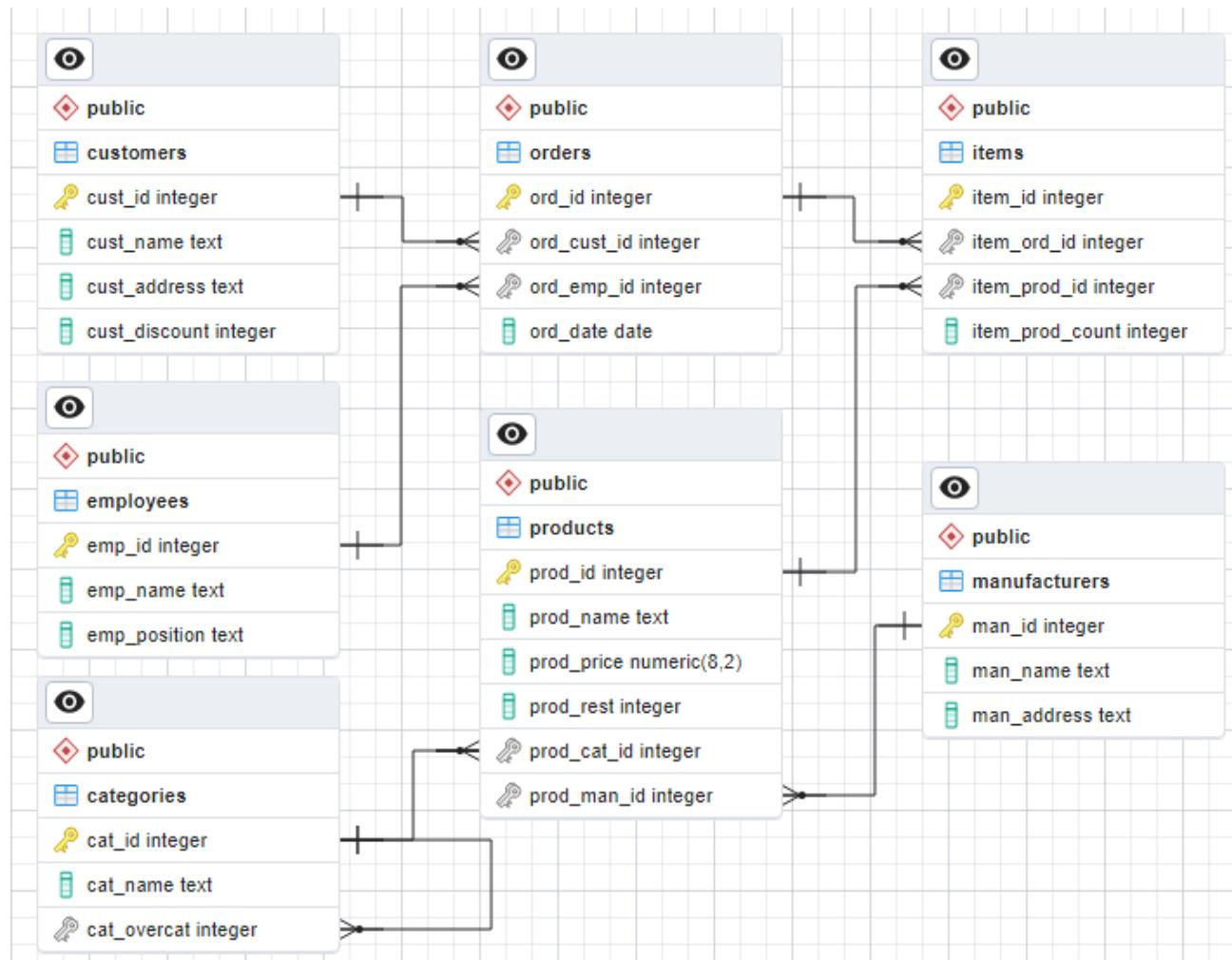


Рис. 21. Схема базы данных

Лабораторная работа № 4

ВСТАВКА, УДАЛЕНИЕ И ОБНОВЛЕНИЕ ДАННЫХ

Теоретические сведения

Рассматриваются следующие вопросы:

- вставка данных с помощью инструкции *INSERT*;
- удаление данных инструкциями *DELETE* и *TRUNCATE*;
- обновление данных с помощью инструкции *UPDATE*.

Упрощенный синтаксис инструкции вставки строк в таблицу [12]:

INSERT INTO имя_таблицы

[(имя_атрибута, имя_атрибута, ...)]

VALUES (значение_атрибута, значение_атрибута, ...);

Замечания к инструкции:

- в начале команды перечисляются атрибуты таблицы (квадратные скобки указывают на то, что список атрибутов не является обязательным);
- атрибуты можно указывать не в том порядке, в котором они перечислялись при создании таблицы;
- если список атрибутов отсутствует, то в предложении *VALUES* значения атрибутов задают с учетом того, как они следовали в определении таблицы;
- краткая форма записи команды менее универсальна (при реструктуризации таблицы придется корректировать инструкцию *INSERT* в прикладных программах).

Упрощенный синтаксис инструкции обновления строк в таблице [12]:

UPDATE имя_таблицы

SET имя_атрибута1 = значение_атрибута1,

имя_атрибута2 = значение_атрибута2, ...

WHERE условие;

ВНИМАНИЕ!

Если не задать условие в секции *WHERE*,
то будут обновлены ВСЕ строки в таблице.

Синтаксис инструкции удаления строк из таблицы [12]:

DELETE FROM имя_таблицы WHERE условие;

Удалить все строки из таблицы можно простой инструкцией:

DELETE FROM имя_таблицы;

ВНИМАНИЕ!

При вставке, обновлении и удалении данных
необходимо учитывать ограничения внешних ключей,
которые рассматривались в лабораторной работе № 3.

Очистить таблицу или набор таблиц можно инструкцией *TRUNCATE* [12]:

TRUNCATE [TABLE] имя_таблицы, имя_таблицы, ... [параметры]

Чтобы очистить таблицу этой инструкцией, нужно иметь право *TRUNCATE* для этой таблицы. Инструкция *TRUNCATE* действует как безусловная команда *DELETE* для каждой таблицы, но быстрее, так как она не сканирует таблицы. Кроме того, она немедленно высвобождает дисковое пространство. Полезна для больших таблиц.

Инструкцию *TRUNCATE* нельзя использовать с таблицей, на которую по внешнему ключу ссылаются другие таблицы, если только эти таблицы не очищаются этой же инструкцией. Если указать параметр *CASCADE*, то будут автоматически очищены все таблицы, ссылающиеся по внешнему ключу на заданные таблицы или на таблицы, затронутые в результате действия *CASCADE*.

Параметр *RESTRICT* запрещает очищение любых таблиц, на которые по внешнему ключу ссылаются другие таблицы, не перечисленные в этой команде. Это поведение по умолчанию.

При выполнении *TRUNCATE* не срабатывают триггеры *ON DELETE* для таблиц, но срабатывают триггеры *ON TRUNCATE*. Триггеры *BEFORE TRUNCATE* срабатывают до очищения, а триггеры *AFTER TRUNCATE* – после очищения последней таблицы. Триггеры срабатывают по порядку обработки таблиц (сначала для перечисленных в инструкции, затем для затрагиваемых каскадно).

Инструкция *TRUNCATE* небезопасна с точки зрения MVCC. После очищения таблицы она будет выглядеть пустой для параллельных транзакций, если они работают со снимком, полученным до очищения. *TRUNCATE* – надежная транзакционная операция в отношении данных в таблицах: очищение будет безопасно отменено, если окружающая транзакция не будет зафиксирована.

Практическая работа

При выполнении задания, используя инструкции SQL:

- заполнить согласованными данными таблицы базы данных;
- при необходимости исправить введенную информацию;
- составить отчет.

Пример выполнения задания

После заполнения данными каждой таблицы для контроля правильности ввода предусмотрим вывод всех строк таблицы инструкцией *SELECT*.

ВНИМАНИЕ!

Последовательность заполнения таблиц данными имеет значение.

Вначале заполняются таблицы, соответствующие родительским отношениям (находящимся со стороны «один» в связи «один-ко-многим»).

Заполнение таблицы *categories* с использованием одного оператора *INSERT* для добавления всех строк таблицы.

```

1 -----
2 -- Заполнение таблицы categories
3 -----
4 INSERT INTO categories (cat_ID, cat_name, cat_overcat)
5     VALUES (1, 'Сверлильные инструменты', NULL),
6             (2, 'Пильно-отрезные инструменты', NULL),
7             (3, 'Инструменты для обработки поверхностей', NULL),
8             (4, 'Перфораторы', 1),
9             (5, 'Дрели', 1),
10            (6, 'Шуруповерты', 1),
11            (7, 'Дисковые пилы', 2),
12            (8, 'Цепные пилы', 2),
13            (9, 'Лобзики электрические', 2),
14            (10, 'Полировальные машины', 3),
15            (11, 'Электрорубанки', 3);

```

1 SELECT * FROM categories;			
Data Output	Messages	Notifications	
cat_id [PK] integer	cat_name text	cat_overcat integer	
1	Сверлильные инструменты	[null]	
2	Пильно-отрезные инструменты	[null]	
3	Инструменты для обработки поверхностей	[null]	
4	Перфораторы	1	
5	Дрели	1	
6	Шуруповерты	1	
7	Дисковые пилы	2	
8	Цепные пилы	2	
9	Лобзики электрические	2	
10	Шлифовальные машины	3	
11	Электрорубанки	3	

Таблица *categories* имеет ограничение внешнего ключа *categories_FK*, соответствующее рекурсивной связи одних экземпляров сущности *КАТЕГОРИЯ* с другими экземплярами этой же сущности (категорий с надкатегориями).

Для проверки ограничения внешнего ключа, реализуемого этой рекурсивной связью, попробуем ввести в таблицу строку со значением надкатегории 13, которое отсутствует в столбце первичного ключа *cat_ID*.

```
1 INSERT INTO categories (cat_ID, cat_name, cat_overcat)
2      VALUES (12, 'Измерительные инструменты', 13);
```

Data Output Messages Notifications

ERROR: ОШИБКА: INSERT или UPDATE в таблице "categories" нарушает ограничение внешнего ключа "categories_fk"
DETAIL: Ключ (cat_overcat)=(13) отсутствует в таблице "categories".

Заполнение таблицы *manufacturers* с использованием одного оператора *INSERT* для добавления всех строк таблицы.

```
1 -----
2 -- Заполнение таблицы manufacturers
3 -----
4 INSERT INTO manufacturers (man_ID, man_name, man_address)
5      VALUES (1, 'Мастер', 'Россия'),
6              (2, 'Техника', 'Россия'),
7              (3, 'Deco', 'Китай'),
8              (4, 'Work', 'Китай'),
9              (5, 'Kraut', 'Германия'),
10             (6, 'Yamamoto', 'Япония');
```

```
1 SELECT * FROM manufacturers;
```

Data Output Messages Notifications

The screenshot shows a database interface with a toolbar at the top containing icons for new table, open table, save, delete, export, and refresh. Below the toolbar is a table structure for the 'manufacturers' table. The table has three columns: 'man_id' (PK integer), 'man_name' (text), and 'man_address' (text). The data is as follows:

	man_id [PK] integer	man_name	man_address
1	1	Мастер	Россия
2	2	Техника	Россия
3	3	Deco	Китай
4	4	Work	Китай
5	5	Kraut	Германия
6	6	Yamamoto	Япония

Заполнение таблицы *products* с использованием одного оператора *INSERT* для добавления всех строк таблицы (контрольный вывод обрван для экономии места).

```

1 -----
2 -- Заполнение таблицы products
3 -----
4 INSERT INTO products
5     (prod_ID, prod_name, prod_price, prod_rest, prod_cat_ID, prod_man_ID)
6     VALUES (1, 'Перфоратор Basic-500', 2250, 150, 4, 6),
7             (2, 'Перфоратор Power-800', 3299, 85, 4, 5),
8             (3, 'Перфоратор Power-500', 2500, 180, 4, 5),
9             (4, 'Перфоратор Удар-650', 1500, 200, 4, 1),
10            (5, 'Перфоратор Deco-2000', 3000, 60, 4, 3),
11            (6, 'Дрель Deco-850', 2000, 160, 5, 3),
12            (7, 'Дрель Спец-700', 1800, 300, 5, 2),
13            (8, 'Шуруповерт Ветерок', 1950, 250, 6, 2),
14            (9, 'Шуруповерт Deco', 3599, 175, 6, 3),
15            (10, 'Пила дисковая Deco', 1550, 80, 7, 3),
16            (11, 'Пила дисковая Power', 2999, 100, 7, 5),
17            (12, 'Пила дисковая Kaput', 3999, 82, 7, 5),
18            (13, 'Пила цепная Deco', 1550, 175, 8, 3),
19            (14, 'Пила цепная Work', 4999, 50, 8, 4),
20            (15, 'Лобзик Basic', 1100, 250, 9, 6),
21            (16, 'Лобзик Deco', 1499, 250, 9, 3),
22            (17, 'Полировальная машина Deco-1000', 1950, 60, 10, 3),
23            (18, 'Полировальная машина Power-1200', 3000, 60, 10, 5),
24            (19, 'Электрорубанок Мастер-800', 3100, 250, 11, 1),
25            (20, 'Электрорубанок Deco-1000', 3700, 150, 11, 3);

```

```
1 SELECT * FROM products;
```

Data Output Messages Notifications

	prod_id [PK] integer	prod_name text	prod_price numeric (8,2)	prod_rest integer	prod_cat_id integer	prod_man_id integer
1	1	Перфоратор Basic-500	2250.00	150	4	6
2	2	Перфоратор Power-800	3299.00	85	4	5
3	3	Перфоратор Power-500	2500.00	180	4	5
4	4	Перфоратор Удар-650	1500.00	200	4	1
5	5	Перфоратор Deco-2000	3000.00	60	4	3
6	6	Дрель Deco-850	2000.00	160	5	3
7	7	Дрель Спец-700	1800.00	300	5	2
8	8	Шуруповерт Ветерок	1950.00	250	6	2
9	9	Шуруповерт Deco	3599.00	175	6	3
10	10	Пила дисковая Deco	1550.00	80	7	3
11	11	Пила дисковая Power	2999.00	100	7	5
12	12	Пила дисковая Kaput	3999.00	82	7	5
13	13	Пила цепная Deco	1550.00	175	8	3

Заполнение таблицы *customers* с использованием одного оператора *INSERT* для добавления всех строк таблицы.

```
1 -----  
2 -- Заполнение таблицы customers  
3 -----  
4 INSERT INTO customers (cust_ID, cust_name, cust_address, cust_discount)  
5     VALUES (1, 'Сергеев Владимир Николаевич', 'Москва', 5),  
6             (2, 'Волков Иван Владимирович', 'Красноярск', 0),  
7             (3, 'Иванова Ольга Сергеевна', 'Москва', 0),  
8             (4, 'Васильев Александр Иванович', 'Екатеринбург', 5),  
9             (5, 'Тромб Павел Гаврилович', 'Тула', 0),  
10            (6, 'Кирьянова Людмила Петровна', 'Сочи', 0),  
11            (7, 'Антипов Алексей Владимирович', 'Москва', 10),  
12            (8, 'Июньский Виктор Карлович', 'Санкт-Петербург', 5),  
13            (9, 'Простой Андрей Андреевич', 'Омск', 0),  
14            (10, 'Мотыль Ирина Александровна', 'Санкт-Петербург', 0);
```

```
1 SELECT * FROM customers;
```

Data Output Messages Notifications

	cust_id [PK] integer	cust_name text	cust_address text	cust_discount integer
1	1	Сергеев Владимир Николаевич	Москва	5
2	2	Волков Иван Владимирович	Красноярск	0
3	3	Иванова Ольга Сергеевна	Москва	0
4	4	Васильев Александр Иванович	Екатеринбург	5
5	5	Тромб Павел Гаврилович	Тула	0
6	6	Кирьянова Людмила Петровна	Сочи	0
7	7	Антипов Алексей Владимирович	Москва	10
8	8	Июньский Виктор Карлович	Санкт-Петербург	5
9	9	Простой Андрей Андреевич	Омск	0
10	10	Мотыль Ирина Александровна	Санкт-Петербург	0

Заполнение таблицы *employees* с использованием одного оператора *INSERT* для добавления всех строк таблицы.

```
1 -----  
2 -- Заполнение таблицы employees  
3 -----  
4 INSERT INTO employees (emp_ID, emp_name, emp_position)  
5      VALUES (1, 'Ребров Андрей Андреевич', 'менеджер'),  
6                  (2, 'Котов Владимир Иванович', 'старший менеджер'),  
7                  (3, 'Колосова Елена Сергеевна', 'менеджер'),  
8                  (4, 'Батова Алина Дмитриевна', 'старший менеджер');  
9  
10 SELECT * FROM employees;
```

Data Output Messages Notifications

	emp_id [PK] integer	emp_name	emp_position
1	1	Ребров Андрей Андреевич	менеджер
2	2	Котов Владимир Иванович	старший менеджер
3	3	Колосова Елена Сергеевна	менеджер
4	4	Батова Алина Дмитриевна	старший менеджер

Заполнение таблицы *orders* с использованием одного оператора *INSERT* для добавления всех строк таблицы (контрольный вывод оборван).

```
1 -----  
2 -- Заполнение таблицы orders  
3 -----  
4 INSERT INTO orders (ord_ID, ord_cust_ID, ord_emp_ID, ord_date)  
5      VALUES (1, 1, 1, '15-01-2023'),  
6                  (2, 6, 1, '17-01-2023'),  
7                  (3, 5, 4, '17-01-2023'),  
8                  (4, 9, 3, '23-01-2023'),  
9                  (5, 6, 1, '02-02-2023'),  
10                 (6, 5, 4, '10-02-2023'),  
11                 (7, 3, 2, '23-02-2023'),  
12                 (8, 5, 4, '23-02-2023'),  
13                 (9, 10, 2, '23-02-2023'),  
14                 (10, 2, 1, '04-03-2023'),  
15                 (11, 4, 4, '04-03-2023'),  
16                 (12, 7, 2, '16-03-2023'),  
17                 (13, 1, 3, '24-03-2023'),  
18                 (14, 8, 3, '25-03-2023'),  
19                 (15, 5, 2, '26-03-2023');
```

```
1 SELECT * FROM orders;
```

Data Output Messages Notifications

The screenshot shows a database interface with a toolbar at the top containing icons for edit, save, delete, and refresh. Below the toolbar is a table with 5 rows and 5 columns. The columns are labeled: ord_id [PK] integer, ord_cust_id integer, ord_emp_id integer, and ord_date date. The data in the table is as follows:

	ord_id [PK] integer	ord_cust_id integer	ord_emp_id integer	ord_date date
1	1	1	1	2023-01-15
2	2	6	1	2023-01-17
3	3	5	4	2023-01-17
4	4	9	3	2023-01-23
5	5	6	1	2023-02-02

Заполнение таблицы *items* с использованием одного оператора *INSERT* для добавления всех строк таблицы (контрольный вывод обрамлен).

```
1 -----
2 -- Заполнение таблицы items
3 -----
4 INSERT INTO items (item_ID, item_ord_ID, item_prod_ID, item_prod_count)
5     VALUES (1, 1, 2, 4),
6             (2, 1, 5, 4),
7             (3, 2, 5, 1),
8             (4, 2, 6, 1),
9             (5, 3, 10, 1),
10            (6, 4, 12, 1),
11            (7, 5, 19, 6),
12            (8, 6, 16, 5),
13            (9, 7, 1, 1),
14            (10, 8, 20, 1),
15            (11, 9, 15, 1),
16            (12, 10, 11, 1),
17            (13, 10, 14, 1),
18            (14, 11, 18, 1),
19            (15, 12, 18, 1),
20            (16, 13, 7, 5),
21            (17, 14, 4, 1),
22            (18, 14, 6, 1),
23            (19, 14, 8, 1),
24            (20, 14, 19, 1),
25            (21, 15, 18, 3);
```

1 `SELECT * FROM items;`

Data Output Messages Notifications

	item_id [PK] integer	item_ord_id integer	item_prod_id integer	item_prod_count integer
1	1	1	2	4
2	2	1	5	4
3	3	2	5	1
4	4	2	6	1
5	5	3	10	1
6	6	4	12	1
7	7	5	19	6
8	8	6	16	5
9	9	7	1	1
10	10	8	20	1

Лабораторная работа № 5

РАБОТА С МАССИВАМИ И ТИПАМИ JSON

Теоретические сведения

Рассматриваются следующие вопросы [3, 6]:

- использование таблиц, содержащих массивы переменной длины;
- использование в базе данных таблиц, содержащих значения, представленные в формате JSON (JavaScript Object Notation).

Массивы. PostgreSQL позволяет создавать в таблицах 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.

Типы JSON. Предназначены для сохранения в столбцах значений в формате JSON (JavaScript Object Notation). Существует два типа: *json*

и *jsonb*. Основное их различие в быстродействии. Если столбец имеет тип *json*, тогда сохранение значений происходит быстрее, поскольку они записываются в том виде, в котором были введены. При дальнейшем 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 этого типа. Но все последующие обращения к сохраненным значениям выполняются быстрее, так как выполнять их разбор уже не требуется.

Тип *json* сохраняет порядок следования ключей в объектах и повторяющиеся значения ключей, а тип *jsonb* этого не делает.

ВНИМАНИЕ!

Рекомендуется в приложениях использовать тип *jsonb*, если только нет каких-то особых аргументов в пользу выбора типа *json*.

Практическая работа

Необходимо для созданной базы данных:

- добавить таблицу со столбцом, содержащим массив значений;
- добавить таблицу со столбцом, содержащим значения в формате JSON;
- составить отчет.

Пример выполнения задания

1. Сформируем и сохраним в базе данных графики работы сотрудников службы оформления заказов (номера дней недели, когда они занимаются непосредственно заказами). Создадим таблицу, в которой эти графики будут храниться в виде единых списков, т. е. в виде одномерных массивов. При этом одному сотруднику может соответствовать несколько графиков, но каждый график связан с одним сотрудником.

```

1 CREATE TABLE schedules (
2     sched_ID integer NOT NULL,
3     sched_emp_ID integer NOT NULL,
4     schedule integer[],
5     CONSTRAINT schedules_PK PRIMARY KEY (sched_ID)
6 );

```

```

1 ALTER TABLE schedules
2 ADD CONSTRAINT schedules_FK FOREIGN KEY (sched_emp_ID)
3 REFERENCES employees (emp_ID)
4 ON DELETE CASCADE
5 ON UPDATE CASCADE;

```

```

1 INSERT INTO schedules VALUES
2     (1, 1, '{ 1, 3, 5, 6, 7 }'::integer[]),
3     (2, 2, '{ 1, 2, 5, 7 }'::integer[]),
4     (3, 3, '{ 2, 5 }'::integer[]),
5     (4, 4, '{ 3, 5, 6 }'::integer[]);

```

Инструкция *INSERT INTO* вводит четыре графика (по одному для каждого сотрудника). Каждый график представляет собой номер дня недели (от 1 до 7). При вводе каждый список приводится к типу целочисленного массива.

```

1 SELECT schedules.sched_ID, employees.emp_name, schedules.schedule
2 FROM schedules JOIN employees
3 ON schedules.sched_emp_ID = employees.emp_ID;

```

	sched_id	emp_name	schedule
1	1	Ребров Андрей Андреевич	{1,3,5,6,7}
2	2	Котов Владимир Иванович	{1,2,5,7}
3	3	Колосова Елена Сергеевна	{2,5}
4	4	Батова Алина Дмитриевна	{3,5,6}

Рассмотрим некоторые типичные операции над массивами. Предположим, руководство компании решило, что каждый сотрудник должен работать с заказами 4 раза в неделю.

Нужно обновить значения в таблице:

- Батовой А. Д. добавим один день с помощью операции конкatenации:

```
1 UPDATE schedules
2 SET schedule = schedule || 7
3 WHERE sched_ID = 4;
```

- Колосовой Е. С. добавим один день в конец списка (массива) с помощью функции *array_append*:

```
1 UPDATE schedules
2 SET schedule = array_append(schedule, 6)
3 WHERE sched_ID = 3;
```

- Колосовой Е. С. добавим еще один день в начало списка с помощью функции *array_prepend* (параметры функции поменялись местами):

```
1 UPDATE schedules
2 SET schedule = array_prepend(1, schedule)
3 WHERE sched_ID = 3;
```

- У сотрудника Реброва А. А. есть лишний день в графике. С помощью функции *array_remove* удалим из графика пятницу (второй параметр функции):

```
1 UPDATE schedules
2 SET schedule = array_remove(schedule, 5)
3 WHERE sched_ID = 1;
```

- У Котова В. И. изменим дни работы, не меняя их количества. Используем индексы элементов массива. По умолчанию нумерация индексов начинается с единицы, но при необходимости ее можно изменить. К элементам массива можно обращаться в предложении *SET* по отдельности, как будто это разные столбцы.

```
1 UPDATE schedules
2 SET schedule[1] = 2, schedule[2] = 3
3 WHERE sched_ID = 2;
```

В последнем случае можно было использовать срез массива (slice):

```
1 UPDATE schedules
2 SET schedule[1:2] = ARRAY[2, 3]
3 WHERE sched_ID = 2;
```

Использование ключевого слова *ARRAY* – альтернативный способ создания массива. Запись 1:2 означает индексы первого и последнего элементов диапазона массива. Присваивание производится сразу целому диапазону элементов массива.

```
1 SELECT * FROM schedules;
```

Data Output Messages Notifications

	sched_id [PK] integer	sched_emp_id integer	schedule integer[]
1	4	4	{3,5,6,7}
2	3	3	{1,2,5,6}
3	1	1	{1,3,6,7}
4	2	2	{2,3,5,7}

В примерах выводятся только идентификаторы сотрудников. Более наглядные результаты можно получить, используя соединение *JOIN* с таблицей *employees*.

Из таблиц с массивами можно осуществлять и более содержательные выборки с использованием *SELECT*. Получим список сотрудников, работающих в среду:

```
1 SELECT * FROM schedules
2 WHERE array_position(schedule, 3) IS NOT NULL;
```

Data Output Messages Notifications

	sched_id [PK] integer	sched_emp_id integer	schedule integer[]
1	4	4	{3,5,6,7}
2	1	1	{1,3,6,7}
3	2	2	{2,3,5,7}

Функция *array_position* возвращает индекс первого вхождения в массив элемента с указанным значением. Если такого элемента нет, она вернет значение *NULL*.

Выберем сотрудников, работающих по понедельникам и воскресеньям:

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

Data Output Messages Notifications

	sched_id [PK] integer	sched_emp_id integer	schedule integer[]
1	1	1	{1,3,6,7}

Оператор `@>` означает проверку того, что в левом массиве содержатся все элементы правого массива. При этом в левом массиве могут находиться и другие элементы, что мы и видим в графике сотрудника.

Чтобы найти сотрудников, работающих по вторникам и/или пятницам, используем оператор `&&`, который проверяет наличие общих элементов у массивов.

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

Data Output Messages Notifications

	sched_id [PK] integer	sched_emp_id integer	schedule integer[]
1	4	4	{3,5,6,7}
2	3	3	{1,2,5,6}
3	2	2	{2,3,5,7}

Сформулируем вопрос в форме отрицания: кто не работает во вторник и в пятницу? Добавим в предыдущую инструкцию SQL отрицание `NOT`:

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

Data Output Messages Notifications

	sched_id [PK] integer	sched_emp_id integer	schedule integer[]
1	1	1	{1,3,6,7}

Можно развернуть массив в виде столбца таблицы:

```
1 SELECT unnest(schedule) AS день_недели
2 FROM schedules
3 WHERE sched_emp_ID = 1;
```

Data Output Messages Notifications

	день_недели integer
1	1
2	3
3	6
4	7

2. Пусть руководство компании стремится улучшать здоровье, повышать уровень культуры и расширять кругозор сотрудников. Нужна специальная таблица, содержащая сведения о видах спорта, которыми занимается сотрудник, наличии домашней библиотеки и количестве туристических поездок. Новая таблица *hobbies*, как и таблица *schedules*, будет связана с таблицей *employees*, но связь будет иметь мощность «один-к-одному».

```
1 CREATE TABLE hobbies (
2     hobb_emp_ID integer NOT NULL,
3     hobbies jsonb,
4     CONSTRAINT hobbies_PK PRIMARY KEY (hobb_emp_ID)
5 );
```



```
1 ALTER TABLE hobbies
2 ADD CONSTRAINT hobbies_FK FOREIGN KEY (hobb_emp_ID)
3 REFERENCES employees (emp_ID)
4 ON DELETE CASCADE
5 ON UPDATE CASCADE;
```

Введем данные:

```
1 INSERT INTO hobbies VALUES
2     (1, '{ "sports": ["волейбол", "плавание"],
3             "home_lib": true, "trips": 3
4         }'::jsonb ),
5     (2, '{ "sports": ["теннис", "плавание"],
6             "home_lib": true, "trips": 2
7         }'::jsonb ),
8     (3, '{ "sports": ["плавание"],
9             "home_lib": false, "trips": 4
10        }'::jsonb ),
11     (4, '{ "sports": ["волейбол", "плавание", "теннис"],
12             "home_lib": true, "trips": 0
13         }'::jsonb );
```

В таблице получилось следующее:

```
1 SELECT * FROM hobbies;
```

Data Output Messages Notifications

The screenshot shows the pgAdmin interface with the 'Data Output' tab selected. It displays the 'hobbies' table with four rows of data. The columns are 'hobb_emp_id' and 'hobbies'. The data is as follows:

	hobb_emp_id	hobbies
1	1	{"trips": 3, "sports": ["волейбол", "плавание"], "home_lib": true}
2	2	{"trips": 2, "sports": ["теннис", "плавание"], "home_lib": true}
3	3	{"trips": 4, "sports": ["плавание"], "home_lib": false}
4	4	{"trips": 0, "sports": ["волейбол", "плавание", "теннис"], "home_lib": true}

При выводе строк из таблицы порядок ключей в JSON-объектах не был сохранен. Из таблиц с JSON можно осуществлять содержательные выборки с использованием *SELECT*. Можно выбрать всех волейболистов:

```
1 SELECT * FROM hobbies
2 WHERE hobbies @> '{"sports": ["волейбол"]}'::jsonb;
```

Data Output Messages Notifications

The screenshot shows the pgAdmin interface with the 'Data Output' tab selected. It displays the 'hobbies' table with two rows of data, filtering for rows where 'sports' contains 'волейбол'. The data is as follows:

	hobb_emp_id	hobbies
1	1	{"trips": 3, "sports": ["волейбол", "плавание"], "home_lib": true}
2	4	{"trips": 0, "sports": ["волейбол", "плавание", "теннис"], "home_lib": true}

Эту же задачу можно решить иначе:

```
1 SELECT hobb_emp_ID, hobbies -> 'sports' AS sports
2 FROM hobbies
3 WHERE hobbies -> 'sports' @> '[ "волейбол" ]'::jsonb;
```

Data Output Messages Notifications

≡+ ⌂ ⌄ ⌅ ⌆ ⌇ ⌈ ⌉ ⌋ ⌊ ⌊ ⌋ ⌈ ⌉ ⌁ ⌂ ⌃ ⌄ ⌅ ⌆ ⌇ ⌈ ⌉ ⌋ ⌊ ⌊ ⌋ ⌈ ⌉ ⌁

	hobb_emp_id [PK] integer	sports jsonb	🔒
1	1	["волейбол", "плавание"]	
2	4	["волейбол", "плавание", "теннис"]	

Здесь выводится информация только о спортивных предпочтениях сотрудников. Обратите внимание на использование одинарных и двойных кавычек. Операция « \rightarrow » служит для обращения к конкретному ключу JSON-объекта.

При создании столбца с типом *json* или *jsonb* в разных строках в JSON-объектах могут использоваться различные наборы ключей. В примере структуры JSON-объектов во всех строках совпадают. А если бы они не совпадали, как наличие ключа? Ключа *«sport»* в наших объектах нет. Что даст вызов функции *count*?

```
1 SELECT count( * )
2 FROM hobbies
3 WHERE hobbies ? 'sport';
```

Data Output Messages Notifications

≡+ ⌂ ⌄ ⌅ ⌆ ⌇ ⌈ ⌉ ⌋ ⌊ ⌊ ⌋ ⌈ ⌉ ⌁ ⌂ ⌃ ⌄ ⌅ ⌆ ⌇ ⌈ ⌉ ⌋ ⌊ ⌊ ⌋ ⌈ ⌉ ⌁

	count bigint	🔒
1	0	

Здесь *? – проверка наличия ключа*. А вот ключ *«sports»* присутствует. Та же проверка показывает наличие таких записей:

```

1 SELECT count( * )
2 FROM hobbies
3 WHERE hobbies ? 'sports';

```

Data Output Messages Notifications

	count	bigint
1	4	

Рассмотрим обновление JSON-объектов в строках таблицы. Предположим, что сотрудник Батова А. Д. решила заниматься только шахматами. Тогда в базе данных следует выполнить такую операцию:

```

1 UPDATE hobbies
2 SET hobbies = hobbies || '{"sports": ["шахматы"]}'
3 WHERE hobb_emp_ID = 4;

```

```

1 SELECT hobb_emp_ID, hobbies
2 FROM hobbies
3 WHERE hobb_emp_ID = 4;

```

Data Output Messages Notifications

	hobb_emp_id	hobbies
1	4	{"trips": 0, "sports": ["шахматы"], "home_lib": true}

Есть еще один способ обновления JSON-объектов. Если впоследствии сотрудник Батова А. Д. захочет возобновить занятия плаванием, то с помощью функции `jsonb_set` можно будет обновить сведения в таблице:

```

1 UPDATE hobbies
2 SET hobbies = jsonb_set(hobbies, '{ sports, 1 }', '"плавание")'
3 WHERE hobb_emp_ID = 4;

```

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 – ' "плавание" '.

В итоге в таблице получилось следующее:

```
1  SELECT hobb_emp_ID, hobbies
2  FROM hobbies
3  WHERE hobb_emp_ID = 4;
```

Data Output Messages Notifications

	hobb_emp_id [PK] integer	hobbies jsonb
1	4	{"trips": 0, "sports": ["шахматы", "плавание"], "home_lib": true}

Лабораторная работа № 6

Создание запросов на выборку

Теоретические сведения

Рассмотрим следующие вопросы:

- выборка данных из таблиц с помощью оператора *SELECT*;
- использование в запросах операторов и встроенных функций SQL Server;
- создание вложенных запросов.

Выборка строк из таблицы осуществляется SQL-командой *SELECT*, полную структуру которой можно найти в [12]. Упрощенный до предела синтаксис этой команды:

SELECT имя_атрибута, имя_атрибута, ...

FROM имя_таблицы

WHERE условие;

Условия отбора строк в предложении *WHERE* могут конструироваться с использованием операторов сравнения: $=$, \neq , $>$, \geq , $<$, \leq . Рассмотрим некоторые другие способы осуществления отбора строк.

Шаблоны LIKE и NOT LIKE. Задача: выбрать все инструменты Power:

```
1 SELECT * FROM products WHERE prod_name LIKE '%Power%';
```

Data Output Messages Notifications

prod_id [PK] integer	prod_name text	prod_price numeric (8,2)	prod_rest integer	prod_cat_id integer	prod_man_id integer
1	2 Перфоратор Power-800	3299.00	85	4	5
2	3 Перфоратор Power-500	2500.00	180	4	5
3	11 Пила дисковая Power	2999.00	100	7	5
4	18 Полировальная машина Power-1200	3000.00	60	10	5

Символ % соответствует любой последовательности символов, т. е. вместо него могут быть подставлены любые символы в любом количестве или не представлено ни одного символа. Шаблон в операторе *LIKE* покрывает всю анализируемую строку. Поэтому если нужно отыскать некоторую последовательность символов внутри строки, то шаблон должен начинаться и завершаться символом %.

Если по столбцу, к которому применяется оператор *LIKE*, создан индекс для ускорения доступа к данным, то при наличии символа % в начале шаблона этот индекс использоваться не будет. В результате может снизиться производительность.

Кроме символа % в шаблоне может использоваться и символ _, соответствующий в точности одному любому символу.

Найдем в таблице *manufacturers* компании, имеющие названия из четырех символов:

```
1 SELECT * FROM manufacturers WHERE man_name LIKE '____';
```

Data Output Messages Notifications

man_id [PK] integer	man_name text	man_address text
1	3 Deco	Китай
2	4 Work	Китай

Запрос, позволяющий узнать, производители каких стран, кроме России и Китая, представлены в нашем магазине:

```
1 SELECT * FROM manufacturers
2 WHERE man_address NOT LIKE 'Россия' AND
3       man_address NOT LIKE 'Китай';
```

Data Output Messages Notifications

≡+

	man_id [PK] integer	man_name text	man_address text
1	5	Kraut	Германия
2	6	Yamamoto	Япония

Регулярные выражения POSIX. Эти операторы имеют больше возможностей, чем оператор *LIKE*. Запрос, позволяющий выбрать компании России и Китая:

```
1 SELECT * FROM manufacturers WHERE man_address ~ '^^(Р|Ки)';
```

Data Output Messages Notifications

≡+

	man_id [PK] integer	man_name text	man_address text
1	1	Мастер	Россия
2	2	Техника	Россия
3	3	Deco	Китай
4	4	Work	Китай

Оператор *~* ищет совпадение с шаблоном с учетом регистра. Символ *^* означает, что поиск совпадения будет привязан к началу строки. Если нужно проверить наличие символа в составе строки, то перед ним ставят символ обратной косой черты: *\^*. Выражение в круглых скобках означает альтернативный выбор между значениями, разделяемыми символом *|*.

Для инвертирования смысла оператора \sim перед ним добавляют знак $!$.

Пример: найти модели инструментов, которые не оканчиваются числом 500 (вывод обрван):

```
1 SELECT * FROM products WHERE prod_name !~ '500$';
```

Data Output Messages Notifications

prod_id prod_name prod_price prod_rest prod_cat_id prod_man_id

	prod_id [PK] integer	prod_name text	prod_price numeric (8,2)	prod_rest integer	prod_cat_id integer	prod_man_id integer
1	2	Перфоратор Power-800	3299.00	85	4	5
2	4	Перфоратор Удар-650	1500.00	200	4	1
3	5	Перфоратор Deco-2000	3000.00	60	4	3

В этом регулярном выражении символ $$$ означает привязку поискового шаблона к концу строки. Если требуется проверить наличие такого символа в составе строки, то следует указать $\$\$$.

Предикаты сравнения. Могут использоваться в качестве замены традиционных операторов сравнения. Найдем товары в диапазоне от 1500 руб. до 2000 руб.:

```
1 SELECT * FROM products WHERE prod_price BETWEEN 1500 AND 2000;
```

Data Output Messages Notifications

prod_id prod_name prod_price prod_rest prod_cat_id prod_man_id

	prod_id [PK] integer	prod_name text	prod_price numeric (8,2)	prod_rest integer	prod_cat_id integer	prod_man_id integer
1	4	Перфоратор Удар-650	1500.00	200	4	1
2	6	Дрель Deco-850	2000.00	160	5	3
3	7	Дрель Спец-700	1800.00	300	5	2
4	8	Шуруповерт Ветерок	1950.00	250	6	2
5	10	Пила дисковая Deco	1550.00	80	7	3
6	13	Пила цепная Deco	1550.00	175	8	3
7	17	Полировальная машина Deco-1000	1950.00	60	10	3

Вычисляемые столбцы. Если нужно представить цену товара не только в рублях, но и в юанях, то следует вычислить это значение и при-

своить новому столбцу псевдоним с помощью ключевого слова *AS* (вывод оборван):

```
1 SELECT prod_name, prod_price, round(prod_price/11.8, 2) AS CNY
2 FROM products;
```

Data Output Messages Notifications

prod_name prod_price cny

	prod_name	prod_price	cny
1	Перфоратор Basic-500	2250.00	190.68
2	Перфоратор Power-800	3299.00	279.58

Предложение ORDER BY. СУБД не гарантирует никакого конкретного порядка строк в результирующей выборке. Можно задать возрастающий (*ASC*) или убывающий (*DESC*) порядок сортировки:

```
1 SELECT * FROM employees ORDER BY emp_name ASC;
```

Data Output Messages Notifications

emp_id emp_name emp_position

	emp_id	emp_name	emp_position
1	4	Батова Алина Дмитриевна	старший менеджер
2	3	Колосова Елена Сергеевна	менеджер
3	2	Котов Владимир Иванович	старший менеджер
4	1	Ребров Андрей Андреевич	менеджер

Ключевое слово DISTINCT. Часто при традиционной выборке получается список значений, среди которых много повторяющихся. *DISTINCT* позволяет оставить только неповторяющиеся значения. Найдем города, откуда поступили заказы:

```
1 SELECT DISTINCT cust_address FROM customers ORDER BY 1;
```

Data Output Messages Notifications

	cust_address	text
1	Екатеринбург	
2	Красноярск	
3	Москва	
4	Омск	
5	Санкт-Петербург	
6	Сочи	
7	Тула	

Столбец, по значениям которого будут упорядочены строки, указан с помощью его порядкового номера в предложении *SELECT*.

Предложение LIMIT. Пусть необходимо найти три самых дорогих товара. Следует отсортировать строки в таблице по убыванию значений столбца *prod_price* и включить в выборку только первые три строки:

```
1 SELECT prod_ID, prod_name, prod_price
2 FROM products
3 ORDER BY prod_price DESC
4 LIMIT 3;
```

Data Output Messages Notifications

	prod_id [PK] integer	prod_name	prod_price numeric (8,2)
1	14	Пила цепная Work	4999.00
2	12	Пила дисковая Карп	3999.00
3	20	Электрорубанок Deco-1000	3700.00

Предложение OFFSET. Если необходимо найти следующие три товара, занимающие места с четвертого по шестое, то предыдущий алгоритм нужно дополнить еще одним шагом: пропустить три первые строки, прежде чем начать вывод:

```

1 SELECT prod_ID, prod_name, prod_price
2 FROM products
3 ORDER BY prod_price DESC
4 LIMIT 3 OFFSET 3;

```

Data Output Messages Notifications

	prod_id [PK] integer	prod_name text	prod_price numeric (8,2)
1	9	Шуруповерт Deco	3599.00
2	2	Перфоратор Power-800	3299.00
3	19	Электрорубанок Мастер-800	3100.00

Условные выражения CASE. Позволяют вывести то или иное значение в зависимости от условий. Синтаксис выражения понятен из следующего примера:

```

1 SELECT prod_name, prod_price,
2      CASE WHEN prod_price <= 1500 THEN 'Дешевый'
3            WHEN prod_price <= 2500 THEN 'Бюджетный'
4            ELSE 'Дорогой'
5      END AS Категория
6 FROM products
7 WHERE prod_cat_ID = 4
8 ORDER BY prod_ID;

```

Data Output Messages Notifications

	prod_name text	prod_price numeric (8,2)	Категория text
1	Перфоратор Basic-500	2250.00	Бюджетный
2	Перфоратор Power-800	3299.00	Дорогой
3	Перфоратор Power-500	2500.00	Бюджетный
4	Перфоратор Удар-650	1500.00	Дешевый
5	Перфоратор Deco-2000	3000.00	Дорогой

Соединения. Соединение – операция, комбинирующая поля одной и более таблиц в запросе. Выполняется, когда после *FROM* в *SELECT* перечисляется несколько таблиц. При этом указываются поля, которые должны быть совмещены.

Эквисоединение (внутреннее соединение) – соединение, в котором одно или более полей одной таблицы (обычно внешний ключ) совмещаются с аналогичными полями другой таблицы (обычно первичный ключ) при условии равенства.

Совмещаемые поля можно указать двумя способами:

- с помощью ключевого слова *WHERE*;
- с помощью ключевого слова *JOIN*.

В условии *WHERE* сравниваются поля. Если поля обеих таблиц имеют одинаковые имена, их следует конкретизировать привязкой к соответствующей таблице.

Рассмотрим запрос, извлекающий из базы данных *sale* номера заказов с фамилиями покупателей (первые четыре строки):

```
1 SELECT orders.ord_ID, customers.cust_name
2 FROM orders, customers
3 WHERE orders.ord_cust_ID = customers.cust_ID
4 ORDER BY orders.ord_ID
5 LIMIT 4;
```

Data Output Messages Notifications

The screenshot shows a database interface with a toolbar at the top and a table below it. The table has two columns: 'ord_id' (integer) and 'cust_name' (text). The data consists of four rows, each containing an 'ord_id' value from 1 to 4 and a corresponding 'cust_name'. Row 1: ord_id 1, cust_name Сергеев Владимир Николаевич. Row 2: ord_id 2, cust_name Кирьянова Людмила Петровна. Row 3: ord_id 3, cust_name Тромб Павел Гаврилович. Row 4: ord_id 4, cust_name Простой Андрей Андреевич.

	ord_id integer	cust_name text
1	1	Сергеев Владимир Николаевич
2	2	Кирьянова Людмила Петровна
3	3	Тромб Павел Гаврилович
4	4	Простой Андрей Андреевич

Соединение нескольких таблиц аналогично соединению двух таблиц. Пусть нужно выяснить, к каким категориям относятся товары заказа с *ord_ID* = 2:

```
1 SELECT categories.cat_name
2 FROM categories, products, orders, items
3 WHERE orders.ord_ID = 2
4     AND categories.cat_ID = products.prod_cat_ID
5     AND items.item_prod_ID = products.prod_ID
6     AND orders.ord_ID = items.item_ord_ID;
```

Data Output Messages Notifications

The screenshot shows a database interface with a toolbar at the top and a table below it. The table has one column: 'cat_name' (text). The data consists of two rows, each containing a category name. Row 1: cat_name Перфораторы. Row 2: cat_name Дрели.

	cat_name text
1	Перфораторы
2	Дрели

Псевдонимы таблиц работают как псевдонимы полей, однако ключевое слово *AS* не указывается. Указывают пробел между именем таблицы и псевдонимом в списке после *FROM*.

Ключевое слово *JOIN* объединяет список таблиц после *FROM* и условия соединения. Вместо *WHERE* используется *ON*. Можно использовать *INNER JOIN*.

```
1 SELECT orders.ord_ID, customers.cust_name
2 FROM orders JOIN customers
3 ON orders.ord_cust_ID = customers.cust_ID
4 LIMIT 4;
```

Data Output Messages Notifications

	ord_id	cust_name
1	1	Сергеев Владимир Николаевич
2	2	Кирьянова Людмила Петровна
3	3	Тромб Павел Гаврилович
4	4	Простой Андрей Андреевич

Пример запроса, выдающего число товарных позиций в категориях:

```
1 SELECT products.prod_cat_ID AS "Код категории",
2 COUNT(products.prod_ID) AS "Число позиций"
3 FROM categories JOIN products
4 ON categories.cat_ID = products.prod_cat_ID
5 GROUP BY products.prod_cat_ID
6 ORDER BY products.prod_cat_ID;|
```

Data Output Messages Notifications

	Код категории	Число позиций
1	4	5
2	5	2
3	6	2
4	7	3
5	8	2
6	9	2
7	10	2
8	11	2

Внешние соединения. Рассмотренные выше соединения являются внутренними (список результатов – строки, для которых найдено совпадение в объединяемых таблицах). Иногда требуется включить в результат не совпавшие строки.

Внешнее соединение включает в результаты запроса не совпавшие строки как минимум одной из таблиц. Все значения, выбранные из другой таблицы, где соответствующая строка не найдена, являются неопределенными. Существуют:

- *левое внешнее соединение* – возвращает все строки таблицы слева от *JOIN* со строками правой таблицы, которые имеют совпадения;
- *правое внешнее соединение* – возвращает все строки таблицы справа от *JOIN* со строками левой таблицы, которые имеют совпадения;
- *полное внешнее соединение* – возвращает все строки из обеих таблиц. Объединяет каждую строку одной таблицы с *NULL* или с несколькими совпадающими строками другой таблицы.

Чтобы проиллюстрировать использование внешних соединений, добавим в таблицу *customers* еще двух заказчиков без заказов (потом их можно будет удалить):

```
1 INSERT INTO customers (cust_ID, cust_name, cust_address, cust_discount)
2      VALUES (11, 'Сидоров Виктор Владимирович', 'Краснодар', 0),
3              (12, 'Жуков Павел Денисович', 'Новосибирск', 0);
```

Запрос, выдающий список заказчиков и сделанных ими заказов:

```
1 SELECT c.cust_ID, c.cust_name, o.ord_ID
2 FROM customers c JOIN orders o
3 ON c.cust_ID = o.ord_cust_ID
4 ORDER BY c.cust_ID, o.ord_ID;
```

Однако этот запрос не показывает заказчиков, не делавших заказы. Можно использовать левое внешнее соединение (таблица *customers* долж-

на быть левой). Строки со значениями *NULL* в списке результатов – клиенты, у которых нет заказов.

```
1 SELECT c.cust_ID, c.cust_name, o.ord_ID
2 FROM customers c LEFT JOIN orders o
3 ON c.cust_ID = o.ord_cust_ID
4 ORDER BY c.cust_ID, o.ord_ID;
```

Data Output Messages Notifications

cust_id cust_name ord_id

	cust_id integer	cust_name text	ord_id integer
1	1	Сергеев Владимир Николаевич	1
2	1	Сергеев Владимир Николаевич	13
3	2	Волков Иван Владимирович	10
14	9	Простой Андрей Андреевич	4
15	10	Мотыль Ирина Александровна	9
16	11	Сидоров Виктор Владимирович	[null]
17	12	Жуков Павел Денисович	[null]

Аналогичный запрос можно реализовать с помощью правого внешнего соединения. Тогда таблица *customers* должна быть правой:

```
1 SELECT c.cust_ID, c.cust_name, o.ord_ID
2 FROM orders o RIGHT JOIN customers c
3 ON c.cust_ID = o.ord_cust_ID
4 ORDER BY c.cust_ID, o.ord_ID;
```

Этот же запрос с использованием полного внешнего соединения *FULL JOIN* даст тот же результат, что и правое внешнее соединение (каждый заказ должен иметь своего заказчика). Если бы существовали абстрактные заказы, полное внешнее соединение позволило бы увидеть заказчиков, которые не делали заказы, и заказы, не принадлежащие заказчикам.

Список клиентов, которые не делали заказов. В запросе используется тот факт, что несовпадающие строки возвращаются как значения *NULL*:

```

1 SELECT c.cust_ID, c.cust_name
2 FROM orders o RIGHT OUTER JOIN customers c
3 ON c.cust_ID = o.ord_cust_ID
4 WHERE o.ord_ID IS NULL;

```

Data Output Messages Notifications

The screenshot shows a table with two columns: 'cust_id' and 'cust_name'. The first row contains '11' and 'Сидоров Виктор Владимирович'. The second row contains '12' and 'Жуков Павел Денисович'.

	cust_id [PK] integer	cust_name
1	11	Сидоров Виктор Владимирович
2	12	Жуков Павел Денисович

Рефлексивные соединения (самообъединение таблиц). Соединение таблицы с самой собой за счет использования псевдонимов. Используется для отношений с рекурсивными связями. Найдем количество категорий в каждой надкатегории:

```

1 SELECT parents.cat_name AS надкатегория,
2 COUNT(children.cat_name) AS "количество категорий"
3 FROM categories AS parents INNER JOIN categories AS children
4 ON parents.cat_id = children.cat_overcat
5 GROUP BY надкатегория
6 ORDER BY "количество категорий" DESC;

```

Data Output Messages Notifications

The screenshot shows a table with two columns: 'надкатегория' and 'количество категорий'. The first row contains 'Пильно-отрезные инструменты' and '3'. The second row contains 'Сверлильные инструменты' and '3'. The third row contains 'Инструменты для обработки поверхностей' and '2'.

	надкатегория text	количество категорий bigint
1	Пильно-отрезные инструменты	3
2	Сверлильные инструменты	3
3	Инструменты для обработки поверхностей	2

В приведенном примере рефлексивное соединение используется для отношения *categories*, имеющего рекурсивную связь (первичный и внешний ключи находятся в одной таблице).

Определим, какие товары есть в категории, содержащей товар с названием '*Шуруповерт Deco*'. В запросе для таблицы *products* определены два разных псевдонима. После этого они объединяются, как любые другие таблицы:

```

1 SELECT p2.prod_name
2 FROM products p1 JOIN products p2
3 ON p1.prod_name = 'Шуруповерт Deco'
4 AND p1.prod_cat_ID = p2.prod_cat_ID;

```

Data Output Messages Notifications

The screenshot shows a database interface with a toolbar at the top containing icons for new, open, save, delete, export, and refresh. Below the toolbar is a table with two rows of data. The table has two columns: 'prod_name' (text) and an empty column with a lock icon. Row 1 contains 'Шуруповерт Ветерок'. Row 2 contains 'Шуруповерт Deco'.

	prod_name text	lock
1	Шуруповерт Ветерок	
2	Шуруповерт Deco	

Вложенные запросы. Позволяют использовать результат одного запроса в другом запросе. Обычно помещаются после *WHERE*. В качестве вложенного запроса всегда выступает *SELECT*-запрос. Внешним запросом может быть запрос с участием любого SQL-оператора. Вложенные запросы всегда должны быть заключены в скобки.

Некоррелированный вложенный запрос – запрос, в котором внутренний запрос не обращается к содержащему его внешнему. Сначала можно выполнить внутренний запрос, а затем использовать полученный результат во внешнем запросе.

Пусть требуется вывести названия и цены товаров из таблицы *products* для категории «Дрели» таблицы *categories*:

```

1 SELECT prod_name, prod_price
2 FROM products
3 WHERE prod_cat_ID = (SELECT cat_ID
4                      FROM categories
5                      WHERE cat_name = 'Дрели')
6 ORDER BY prod_price;

```

Data Output Messages Notifications

The screenshot shows a database interface with a toolbar at the top containing icons for new, open, save, delete, export, and refresh. Below the toolbar is a table with two rows of data. The table has two columns: 'prod_name' (text) and 'prod_price' (numeric (8,2)). Both columns have lock icons. Row 1 contains 'Дрель Спец-700' and '1800.00'. Row 2 contains 'Дрель Deco-850' and '2000.00'.

	prod_name text	lock	prod_price numeric (8,2)	lock
1	Дрель Спец-700		1800.00	
2	Дрель Deco-850		2000.00	

Аналогичный результат можно получить при помощи многотабличного запроса, но есть ряд задач, которые решаются только при помощи вложенных запросов. Вложенный запрос может применяться не только с условием *WHERE*, но и в конструкциях *DISTINCT*, *GROUP BY*, *ORDER BY* и т. д. Различают:

- вложенные запросы, возвращающие одно значение;
- вложенные запросы, возвращающие несколько строк.

В первом случае вложенный запрос возвращает скалярное значение или литерал, которые используются во внешнем запросе. Например, необходимо определить категорию, к которой относится самый дорогой товар:

```
1 SELECT cat_name AS Категория
2 FROM categories
3 WHERE cat_ID = (SELECT prod_cat_ID
4                   FROM products
5                   WHERE prod_price = (SELECT MAX(prod_price)
6                                         FROM products));
```

The screenshot shows the MySQL Workbench interface. At the top, there are tabs for 'Data Output' (which is selected), 'Messages', and 'Notifications'. Below the tabs is a toolbar with various icons. The main area displays a table with one row of data. The table has two columns: 'Категория' (Category) and 'prod_cat_ID'. The first row contains the value 'Цепные пилы' (Chain saws) under 'Категория' and '1' under 'prod_cat_ID'. The table has a header row with column names and a footer row with the number '1'.

	Категория	prod_cat_ID
1	Цепные пилы	1

Вложенные запросы чаще всего используются в операциях сравнения в условиях, которые задаются ключевыми словами *WHERE*, *HAVING* или *ON*.

Ключевое слово *ANY* может применяться с любым оператором сравнения. Используется логика *ИЛИ* (достаточно, чтобы срабатывало хотя бы одно из условий).

Рассмотрим запрос, возвращающий имена и фамилии покупателей, совершивших хотя бы одну покупку:

```

1 SELECT cust_name
2 FROM customers
3 WHERE cust_ID = ANY(SELECT ord_cust_ID FROM orders);

```

Data Output Messages Notifications

	cust_name	
	text	
1	Сергеев Владимир Николаевич	
2	Волков Иван Владимирович	
3	Иванова Ольга Сергеевна	
4	Васильев Александр Иванович	
5	Тромб Павел Гаврилович	

6	Кирьянова Людмила Петровна
7	Антипов Алексей Владимирович
8	Июньский Виктор Карлович
9	Простой Андрей Андреевич
10	Мотыль Ирина Александровна

Ключевое слово *ALL* также может применяться с любым оператором сравнения, но при этом используется логика *И* (должны срабатывать все условия). Рассмотрим запрос, возвращающий все товары, цена которых превышает среднюю цену в каждой из категорий:

```

1 SELECT prod_name, prod_price
2 FROM products
3 WHERE prod_price > ALL(SELECT AVG(prod_price) FROM products
4 GROUP BY prod_cat_ID)
5 ORDER BY prod_price;

```

Data Output Messages Notifications

	prod_name	prod_price
	text	
		numeric (8,2)
1	Шуруповерт Deco	3599.00
2	Электрорубанок Deco-1000	3700.00
3	Пила дисковая Kaput	3999.00
4	Пила цепная Work	4999.00

Результирующая таблица, возвращаемая вложенным запросом, может не содержать ни одной строки. Для проверки этого факта могут использоваться ключевые слова *EXISTS* и *NOT EXISTS*. Запрос, формирующий список покупателей, совершивших хотя бы одну покупку, можно переписать следующим образом:

```
1 SELECT cust_name
2 FROM customers
3 WHERE EXISTS(SELECT * FROM orders
4               WHERE orders.ord_cust_ID = customers.cust_ID);
```

Data Output Messages Notifications

The screenshot shows the Oracle SQL Developer interface. At the top, there is a code editor window containing the SQL query. Below it is a toolbar with various icons. The main area displays two tables of data. The first table, titled 'Data Output', contains four rows of customer names. The second table, also titled 'Data Output', contains six more rows of customer names. Both tables have columns for row numbers (1-4 and 5-10 respectively) and the customer name itself.

	cust_name
1	Сергеев Владимир Николаевич
2	Волков Иван Владимирович
3	Иванова Ольга Сергеевна
4	Васильев Александр Иванович

	cust_name
5	Тромб Павел Гаврилович
6	Кирьянова Людмила Петровна
7	Антипов Алексей Владимирович
8	Июньский Виктор Карлович
9	Простой Андрей Андреевич
10	Мотыль Ирина Александровна

Коррелированный вложенный запрос – запрос, в котором внутренний запрос обращается к значениям, полученным с помощью внешнего. Эти запросы менее эффективны, чем некоррелированные, так как внутренний запрос должен выполняться для каждой строки, возвращенной внешним запросом (для некоррелированного вложенного запроса внутренний запрос выполняется один раз).

Компания решила предоставить скидку всем заказчикам, сделавшим больше одного заказа. Необходимо найти таких заказчиков:

```
1  SELECT cust_ID, cust_name FROM customers c
2  WHERE 1 < (SELECT COUNT(o.ord_cust_ID)
3            FROM orders o
4            WHERE c.cust_ID = o.ord_cust_ID);
```

Data Output Messages Notifications

cust_id [PK] integer cust_name

	cust_id	cust_name
1	1	Сергеев Владимир Николаевич
2	5	Тромб Павел Гаврилович
3	6	Кирьянова Людмила Петровна

Отличительными признаками коррелированного вложенного запроса являются псевдонимы таблиц во внешнем и внутреннем запросах и их использование после ключевого слова *WHERE* во внутреннем запросе.

Внешний запрос возвращает список, содержащий *cust_ID* из таблицы *customers*. Каждое возвращенное значение передается внутреннему запросу, который находит число заказов клиента. Если число заказов больше 1, условие *WHERE* вернет значение *TRUE*, и значение *cust_ID* добавляется в список результатов запроса.

Практическая работа

Необходимо для созданной базы данных:

- построить запрос на выборку с использованием какого-либо соединения;
- построить запрос на выборку, содержащий вложенный запрос;
- составить отчет.

Пример выполнения задания

1. Создадим многотабличный запрос на выборку, который выводит коды покупателей, совершивших две покупки и более:

```
1 SELECT customers.cust_ID, COUNT(orders.ord_ID) AS total
2 FROM customers LEFT JOIN orders
3 ON customers.cust_ID = orders.ord_cust_ID
4 GROUP BY customers.cust_ID
5 HAVING COUNT(orders.ord_ID) >= 2
6 ORDER BY total DESC, customers.cust_ID ASC;
```

Data Output Messages Notifications

The screenshot shows a database interface with a toolbar at the top and a table below it. The table has two columns: 'cust_id' and 'total'. The data is as follows:

	cust_id [PK] integer	total bigint
1	5	4
2	1	2
3	6	2

2. Создадим запрос на выборку с вложенным запросом, выводящим перечень товаров, которые никогда не заказывались покупателями:

```
1 SELECT prod_ID, prod_name, prod_price
2 FROM products
3 WHERE NOT EXISTS
4     (SELECT * FROM items
5      WHERE items.item_prod_ID = products.prod_ID);
```

Data Output Messages Notifications

The screenshot shows a database interface with a toolbar at the top and a table below it. The table has three columns: 'prod_id', 'prod_name', and 'prod_price'. The data is as follows:

	prod_id [PK] integer	prod_name text	prod_price numeric (8,2)
1	3	Перфоратор Power-500	2500.00
2	9	Шуруповерт Deco	3599.00
3	13	Пила цепная Deco	1550.00
4	17	Полировальная машина Deco-1000	1950.00

Лабораторная работа № 7

ПРЕДСТАВЛЕНИЯ

Теоретические сведения

Рассмотрим следующие вопросы:

- создание представлений с помощью оператора *CREATE VIEW*;
- модификация представлений с помощью оператора *ALTER VIEW*;
- удаление представлений с помощью оператора *DROP VIEW*.

Часто приходится многократно выполнять сложные запросы, требующие обращения к нескольким таблицам. Избежать необходимости многократного формирования таких запросов позволяют представления (*Views*). Если речь идет о выборке данных, то представления практически неотличимы от таблиц с точки зрения обращения к ним в командах *SELECT*.

ВНИМАНИЕ!

В отличие от таблиц, представления не содержат данных.

Данные выбираются из таблиц, на основе которых создано представление при каждом обращении к нему в инструкции *SELECT*.

Создание представлений. Упрощенный синтаксис инструкции [12]:

CREATE [OR REPLACE] VIEW имя_представления [(имя_столбца [, ...])]

AS query;

Если список имен столбцов не приведен, то их имена формируются на основании текста запроса *query*. Создаваемое представление лишено физической материализации, поэтому указанный запрос будет выполняться при каждом обращении к представлению.

Если задано имя схемы, то представление создается в указанной схеме, в противном случае – в текущей. Имя представления должно отличаться от имен других таблиц, последовательностей, индексов, представ-

лений, материализованных представлений или сторонних таблиц в этой схеме.

Предложение *OR REPLACE* – расширение инструкции. Если представление с этим именем существует, оно заменяется. Однако новый запрос должен выдавать те же столбцы, что выдавал запрос, ранее определенный для этого представления (столбцы с такими же именами должны иметь те же типы данных и следовать в том же порядке). При этом можно добавить несколько новых столбцов в конце списка. Вычисления, формирующие столбцы представления, могут быть другими.

Изменение представлений. Изменить различные дополнительные свойства представления можно с помощью инструкции *ALTER VIEW*. Для изменения запроса, определяющего представление, следует использовать *CREATE OR REPLACE VIEW*.

Удаление представлений. Осуществляется с помощью инструкции [12]:

DROP VIEW имя_представления;

Если заранее известно, что возможна попытка удаления несуществующего представления, можно избежать ненужных сообщений об ошибке путем добавления в инструкцию фразы *IF EXISTS*:

DROP VIEW IF EXISTS имя_представления;

Материализованные представления. Материализованное представление похоже на обычную таблицу. Однако оно отличается от таблицы тем, что не только сохраняет данные, но также запоминает запрос, с помощью которого эти данные были собраны. Упрощенный синтаксис инструкции для их создания [12]:

*CREATE MATERIALIZED VIEW [IF NOT EXISTS] имя_таблицы
[(имя_столбца [, ...])]
AS query [WITH [NO] DATA];*

Материализованное представление заполняется данными в момент выполнения инструкции его создания, если в инструкции не было фразы *WITH NO DATA*. Если же она была включена, то в момент создания пред-

ставление данными не заполняется, а для заполнения его данными нужно использовать инструкцию

REFRESH MATERIALIZED VIEW имя_таблицы;

Материализованные представления необходимо своевременно обновлять с помощью инструкции *REFRESH*, чтобы они содержали актуальные данные.

Материализованное представление удаляется с помощью инструкции:

DROP MATERIALIZED VIEW имя_таблицы;

Использование представлений дает следующие преимущества [6].

1. Упрощается разграничение полномочий пользователей на доступ к хранимым данным. Разным типам пользователей нужны различные данные, хранящиеся в одних и тех же таблицах. Это касается как строк, так и столбцов таблиц. Создание различных представлений для разных пользователей избавляет от необходимости создавать дополнительные таблицы, дублируя данные.

2. Упрощаются запросы к базе данных. Представления позволяют скрыть сложные запросы от прикладного программиста и сделать запросы более простыми и наглядными.

3. Снижается зависимость прикладных программ от изменений структуры таблиц базы данных. Интерфейсом к запросу, реализуемому представлением, являются столбцы представления (их имена, типы данных и порядок следования). Если этот интерфейс не изменяется, то SQL-запросы, использующие представление, не потребуется корректировать. При изменении структуры базовых таблиц представления нужно лишь перестроить запрос, выполняемый представлением.

4. Снижается время выполнения сложных запросов за счет использования материализованных представлений. Если сводный отчет формируется длительное время, а запросы к отчету будут неоднократными, то стоит сформировать его заранее и сохранить в материализованном представлении.

Практическая работа

Необходимо для созданной базы данных:

- создать несколько простых и материализованных представлений и включить их в базу данных;
- сформировать запросы к этим представлениям;
- составить отчет.

Пример выполнения задания

1. Создадим на базе таблицы *orders* представление *number_of_orders*, отображающее общее количество заказов, сделанных каждым заказчиком:

```
1 CREATE VIEW number_of_orders AS
2 SELECT ord_cust_ID, count( * )
3 FROM orders
4 GROUP BY ord_cust_ID
5 ORDER BY ord_cust_ID;
```

К этому представлению можно обратиться, как к обычной таблице, используя запрос *SELECT*:

The screenshot shows a database interface with a query editor and a results viewer. The query editor contains the SQL code for creating a view:

```
1 SELECT * FROM number_of_orders;
```

The results viewer shows the data from the 'number_of_orders' view:

	ord_cust_id	count
1	1	2
2	2	1
3	3	1
4	4	1
5	5	4
6	6	2
7	7	1
8	8	1
9	9	1
10	10	1

Попытаемся изменить представление *number_of_orders*, добавив ключевое слово *AS* после вызова функции *count* и используя предложение *OR REPLACE*:

```

1 CREATE OR REPLACE VIEW number_of_orders AS
2 SELECT ord_cust_ID, count( * ) AS num_ord
3 FROM orders
4 GROUP BY ord_cust_ID
5 ORDER BY ord_cust_ID;

```

Data Output Messages Notifications

ERROR: ОШИБКА: изменить имя столбца "count" на "num_ord" в представлении нельзя
HINT: Чтобы изменить имя столбца представления, выполните ALTER VIEW ... RENAME COLUMN ...

Ошибка связана с тем, что при первоначальном создании этого представления СУБД уже назначила столбцу имя *count*. В этом случае нужно удалить это представление инструкцией *DROP VIEW*, а затем создать его заново.

2. Создадим на базе таблицы *products* материализованное представление *price_view*, отображающее общую стоимость товаров по каждой категории:

```

1 CREATE MATERIALIZED VIEW price_view
2 AS
3 SELECT prod_cat_ID, SUM(prod_price * prod_rest) AS price
4 FROM products
5 GROUP BY prod_cat_ID;

```

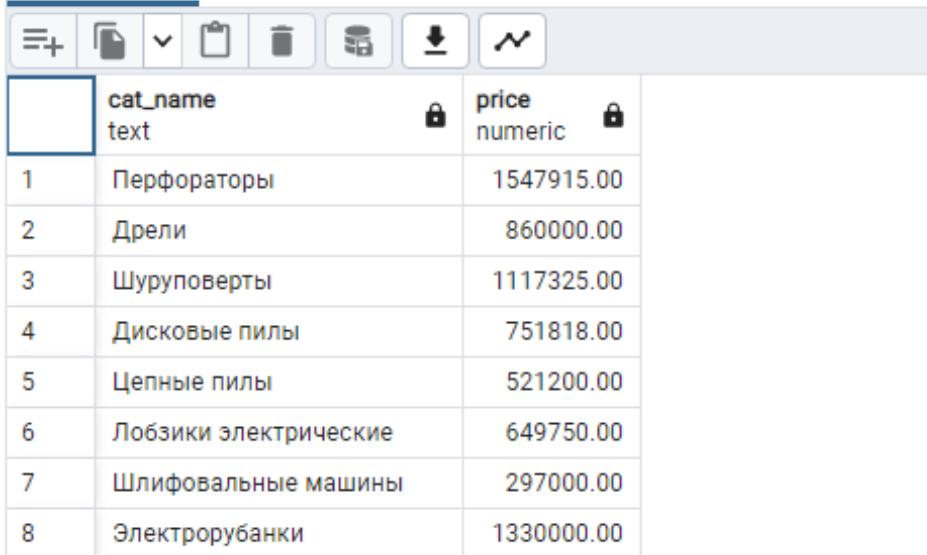
Сформируем запрос к таблице *categories* и представлению *price_view*:

```

1 SELECT categories.cat_name, price_view.price
2 FROM price_view, categories
3 WHERE price_view.prod_cat_ID = categories.cat_ID
4 ORDER BY categories.cat_ID;

```

Data Output Messages Notifications



	cat_name	price
	text	numeric
1	Перфораторы	1547915.00
2	Дрели	860000.00
3	Шуруповерты	1117325.00
4	Дисковые пилы	751818.00
5	Цепные пилы	521200.00
6	Лобзики электрические	649750.00
7	Шлифовальные машины	297000.00
8	Электрорубанки	1330000.00

3. Создадим еще одно материализованное представление *cust_view*, которое выберет из таблицы *customers* всех заказчиков, имеющих скидку. Выбранные записи будут отсортированы по возрастанию в столбце *cust_discount*. Предложение *WITH NO DATA* в конце запроса указывает, что запрос не будет сохранять какие-либо данные в материализованном представлении.

```
1 CREATE MATERIALIZED VIEW cust_view
2 AS
3 SELECT cust_ID, cust_name, cust_address, cust_discount
4 FROM customers
5 WHERE cust_discount > 0
6 ORDER BY cust_discount
7 WITH NO DATA;
```

Если используется предложение *WITH NO DATA*, создается ненаполненное представление. Запрос *SELECT* не выдает записи вновь созданного представления.

```
1 SELECT * FROM cust_view;
```

Data Output Messages Notifications

ERROR: ОШИБКА: материализованное представление "cust_view" не было наполнено
HINT: Примените команду REFRESH MATERIALIZED VIEW.

Заполнить материализованное представление позволяет следующий запрос:

```
1 REFRESH MATERIALIZED VIEW cust_view;
```

Снова выберем записи материализованного представления *cust_view* с помощью инструкции *SELECT*. Теперь запрос *SELECT* работает правильно, поскольку инструкция *REFRESH* загрузила в материализованное представление данные.

1	SELECT * FROM cust_view;			
Data Output	Messages Notifications			
	cust_id integer	cust_name text	cust_address text	cust_discount integer
1	1	Сергеев Владимир Николаевич	Москва	5
2	4	Васильев Александр Иванович	Екатеринбург	5
3	8	Июньский Виктор Карлович	Санкт-Петербург	5
4	7	Антипов Алексей Владимирович	Москва	10

Лабораторная работа № 8

ФУНКЦИИ И ПРОЦЕДУРЫ

Теоретические сведения

Рассмотрим следующие вопросы:

- создание функций и процедур на языке PL/pgSQL;
- создание функций и процедур на языке SQL.

Основные теоретические сведения о хранимых функциях и процедурах приведены в гл. 4 «Программирование на стороне сервера», в которой содержится большое количество примеров функций и процедур.

Также разберем некоторые важные вопросы, не рассмотренные в этой главе.

Если функция не возвращает значения, то возвращаемым типом будет *void*. Если оператор SQL возвращает одну строку, то значения, содержащиеся в этой строке, можно записать в переменные с помощью *INTO*. Для *SELECT* это предложение размещается после списка выбираемых значений, для остальных операторов DML (*INSERT*, *UPDATE*, *DELETE* с предложением *RETURNING*) – последним предложением, т. е. тоже после списка возвращаемых значений.

```

1 CREATE OR REPLACE FUNCTION cust_city(c_code text) RETURNS text
2 LANGUAGE plpgsql AS $$ 
3 DECLARE
4     v text;
5 BEGIN
6     SELECT cust_address
7     INTO v
8     FROM customers
9     WHERE cust_ID = c_code;
10    RETURN v;
11 END;
12 $$;

```

The screenshot shows a database interface with a query window containing the command: `1 SELECT cust_city(9);`. Below the query window is a table labeled 'Data Output' with one row. The table has two columns: 'cust_city' (text type) and its value 'ОМСК'.

	cust_city
1	ОМСК

Код иллюстрирует применение *INTO*, но не является рекомендуемым стилем программирования. Функцию можно записать более компактно:

```

1 CREATE OR REPLACE FUNCTION cust_city(c_code integer) RETURNS text
2 LANGUAGE plpgsql AS $$ 
3 BEGIN
4     RETURN (SELECT cust_address
5             FROM customers
6             WHERE cust_ID = c_code);
7 END;
8 $$;

```

The screenshot shows a database interface with a query window containing the command: `1 SELECT cust_city(6);`. Below the query window is a table labeled 'Data Output' with one row. The table has two columns: 'cust_city' (text type) and its value 'Сочи'.

	cust_city
1	Сочи

Предложение *INTO* полезно, когда получаемые значения используются далее в той же функции. Если после *INTO* указано слово *STRICT*, то выполняемый оператор SQL должен возвращать ровно одну строку. Если оператор не возвращает ни одной строки или возвращает больше одной, возникает исключительная ситуация. Если слово *STRICT* не указано, то в переменные записываются значения из первой строки или значения *NULL*, если не возвращено ни одной строки.

Результатом выполнения функции может быть множество строк. Это множество может использоваться в предложении *FROM* как таблицы и представления. Для задания имен столбцов такого множества можно использовать определение типа.

Обработать результат из нескольких строк позволяет специальная форма цикла *FOR*. Рассмотрим применение такого цикла в функции для

возвращения в качестве результата множества строк предварительно определенного составного типа:

```
1 CREATE TYPE instrum AS (code integer, model text);

1 CREATE OR REPLACE FUNCTION instrum_set() RETURNS SETOF instrum
2 LANGUAGE plpgsql AS $$
3 DECLARE
4     v record;
5 BEGIN
6     FOR v IN SELECT * FROM products
7     LOOP
8         RETURN NEXT ROW (v.prod_ID, v.prod_name)::instrum;
9     END LOOP;
10 END;
11 $$;
```

```
1 SELECT * FROM instrum_set()
2 LIMIT 5;
```

Data Output Messages Notifications

	code integer	model text
1	1	Перфоратор Basic-500
2	2	Перфоратор Power-800
3	3	Перфоратор Power-500
4	4	Перфоратор Удар-650
5	5	Перфоратор Deco-2000

Цикл для обработки запроса можно применять и в том случае, если функция возвращает только одно значение или вовсе не возвращает значений.

Другой способ обработки результатов запросов, возвращающих несколько строк, использует понятие курсора (см. лабораторную работу № 10).

Функции и процедуры на языке SQL. Тело функции или процедуры на языке SQL – последовательность команд SQL, разделенных точкой с запятой. Возвращаемым значением будет первая строка результата вы-

полнения последней SQL-команды. Если результат функции специфицирован не как таблица (*TABLE* или *SETOF*), то возвращается первая строка результата последнего оператора. Для процедур результат не требуется.

Для простых подпрограмм запись тела на языке SQL обычно получается более компактной, чем на любых других языках, доступных в PostgreSQL, включая язык PL/pgSQL. Приведем несколько примеров записи функций на языке SQL.

Функция, возвращающая одно скалярное значение:

```
1 CREATE OR REPLACE FUNCTION hello(p text)
2 RETURNS text
3 LANGUAGE sql
4 AS $$
5 SELECT 'Hello, ' || p || '!';
6 $$;
```

The screenshot shows the pgAdmin interface. On the left, the SQL editor contains the code for creating a scalar function named 'hello' that takes a parameter 'p' of type text and returns a text value by concatenating 'Hello, ' and 'p' with a double exclamation mark. On the right, the 'Data Output' tab is selected, showing the result of executing the function with the argument 'world'. The output table has one row with the value 'Hello, world!'. Below the table are various export and refresh icons.

	hello	
1	Hello, world!	

Функция, возвращающая множество строк составного типа:

```
1 CREATE OR REPLACE FUNCTION inst_set() RETURNS SETOF instrum
2 LANGUAGE sql AS $$
3     SELECT ROW(prod_ID, prod_name)::instrum
4     FROM products;
5 $$;
```



```
1 SELECT * FROM inst_set()
2 LIMIT 4;
```

The screenshot shows the pgAdmin interface. On the left, the SQL editor contains the code for creating a set-returning function named 'inst_set' that returns rows from the 'products' table as a set of 'instrum' type. On the right, the 'Data Output' tab is selected, showing the results of executing the function with a limit of 4. The output table has four rows, each containing a 'code' (integer) and a 'model' (text). The models listed are 'Перфоратор Basic-500', 'Перфоратор Power-800', 'Перфоратор Power-500', and 'Перфоратор Удар-650'. Below the table are various export and refresh icons.

	code	model
1	1	Перфоратор Basic-500
2	2	Перфоратор Power-800
3	3	Перфоратор Power-500
4	4	Перфоратор Удар-650

Если в спецификации функции на языке SQL нет указания *SETOF* (или *TABLE*), то функция возвращает первую строку результата, но это не может считаться рекомендуемой практикой программирования.

```
1 CREATE OR REPLACE FUNCTION inst_row() RETURNS instrum
2 LANGUAGE sql AS $$ 
3   SELECT ROW(prod_ID, prod_name)::instrum
4   FROM products;
5 $$;
```

The screenshot shows a database interface with the following details:

- SQL Editor pane:

```
1 SELECT * FROM inst_row();
```
- Toolbars: Data Output, Messages, Notifications.
- Table View pane:

	code integer	model text
1	1	Перфоратор Basic-500

Применение подпрограмм на языке SQL позволяет вынести запросы из кода приложения и может быть полезно как инструмент ограничения доступа. Но некоторые виды функций, такие как функции триггеров, на SQL писать нельзя.

Практическая работа

Необходимо для созданной базы данных:

- написать несколько хранимых процедур и включить их в базу данных;
- составить отчет.

Пример выполнения задания

Для базы данных необходим набор хранимых процедур, обеспечивающий выборку, вставку, модификацию и удаление данных для всех таблиц. Это ограничит возможности обычного пользователя в плане нанесения ущерба базе данных и обеспечит эффективный код приложения.

Создадим эти процедуры для работы с таблицей *orders*.

1. Вставка данных (процедура *order_insert*):

```
1 CREATE OR REPLACE PROCEDURE order_insert
2     (order_ID integer, customer_ID integer,
3      employee_ID integer, order_date date)
4 LANGUAGE sql
5 AS $$
6     INSERT INTO orders (
7         ord_ID, ord_cust_ID, ord_emp_ID, ord_date)
8     VALUES (
9         order_ID, customer_ID, employee_ID,
10        order_date);
11 $$;
```

Вызов процедуры *order_insert* с указанием значений параметров *order_ID*, *customer_ID*, *employee_ID*, *order_date* позволяет ввести новую строку в таблицу *orders*:

```
1 CALL order_insert(16, 1, 2, '03-04-2023');
```

2. Обновление данных (процедура *order_update*):

```
1 CREATE OR REPLACE PROCEDURE order_update
2     (order_ID integer, customer_ID integer,
3      employee_ID integer, order_date date)
4 LANGUAGE sql
5 AS $$
6     UPDATE orders SET
7         ord_ID = order_ID, ord_cust_ID = customer_ID,
8         ord_emp_ID = employee_ID, ord_date = order_date
9     WHERE ord_ID = order_ID;
10 $$;
```

Вызов процедуры *order_update* с указанием значений параметров *order_ID*, *customer_ID*, *employee_ID*, *order_date* изменяет строку в таблицу *orders*:

```
1 CALL order_update(16, 3, 2, '04-04-2023');
```

3. Удаление данных (процедура *order_delete*):

```
1 CREATE OR REPLACE PROCEDURE order_delete
2                     (order_ID integer)
3 LANGUAGE sql
4 AS $$
5     DELETE FROM orders
6     WHERE ord_ID = order_ID;
7 $$
```

Вызов процедуры *order_delete* с указанием значения параметра *order_ID* позволяет удалить требуемую строку в таблице *orders*:

```
1 CALL order_delete(16);
```

Для остальных таблиц базы данных *sales* хранимые процедуры пишутся аналогично. Что касается аналогичной хранимой процедуры для выборки данных, то для ее организации целесообразно использовать курсор. Создание этой процедуры рассматривается в лабораторной работе № 10, посвященной курсорам.

Лабораторная работа № 9 ТРАНЗАКЦИИ

Теоретические сведения

Рассмотрим следующие вопросы:

- создание транзакции в сеансах и сообщениях;
- создание транзакций с точкой сохранения в режиме явных транзакций;
- создание транзакций с точкой сохранения в режиме неявных транзакций;
- проведение отката к точке сохранения и фиксации транзакций.

Транзакция – одна или несколько операций над базой данных, образующих логически целостную процедуру, которые либо успешно завершаются как единое целое, либо отменяются как единое целое. Фиксация

транзакции (*COMMIT*) обеспечивает запись на диск сделанных изменений. Откат транзакции (*ROLLBACK*) аннулирует все изменения, сделанные операциями транзакции.

Транзакции обеспечивают согласованность (непротиворечивость) базы данных наряду с накладываемыми на таблицы ограничениями целостности. Есть различие между понятиями целостности (*integrity*) и согласованности (*consistency*). Целостность определяют ограничения целостности, и СУБД предотвращает любые попытки нарушения их. Согласованность может временно нарушаться и восстанавливается только в конце транзакции [7].

Разработчик (пользователь), исходя из смысла обработки данных, самостоятельно определяет, какая последовательность операций составляет транзакцию.

Транзакции обладают следующими свойствами (*ACID*):

- *атомарность* (*atomicity*) – транзакция должна быть выполнена целиком или не выполнена вообще (все или ничего);
- *согласованность* (*consistency*) – транзакция переводит данные из одного согласованного состояния в другое согласованное состояние;
- *изолированность* (*isolation*) – при выполнении транзакции другие транзакции должны оказывать по возможности минимальное влияние на нее;
- *долговечность* (*durability*) – после фиксации транзакции данные надежно сохранены в базе и не зависят от последующих возможных сбоев в работе системы.

Реализация транзакций в PostgreSQL базируется на *многоверсионной модели MVCC* (Multiversion Concurrency Control). Каждый запрос SQL видит *снимок данных* (*snapshot*) – согласованное состояние (версию) базы данных, которое она имела в определенный момент времени. Снимок – это не физическая копия базы данных, это несколько чисел, идентифици-

рующих текущую транзакцию и транзакции, которые уже выполнялись к моменту начала текущей. Параллельно выполняемые транзакции, даже изменяющие базу данных, не нарушают согласованности данных этого снимка (когда параллельные транзакции изменяют одни и те же строки, создаются версии этих строк, доступные соответствующим транзакциям). Благодаря MVCC операции чтения никогда не блокируются операциями записи, а операции записи никогда не блокируются операциями чтения.

Режимы выполнения транзакций в современных СУБД:

- *автоматическая фиксация* – каждый оператор SQL представляет собой отдельную транзакцию и фиксируется по завершении;
- *неявные транзакции* – транзакции начинаются без объявления начала транзакции. В разных СУБД этот режим устанавливается разными способами;
- *явные транзакции* – несколько операторов SQL объединяются в одну транзакцию с использованием явной установки границ транзакций.

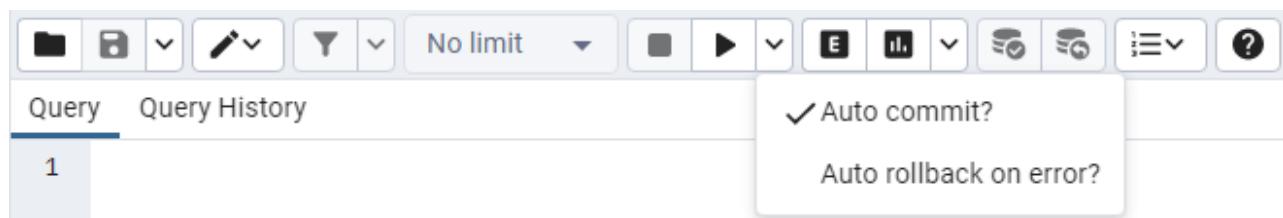
PostgreSQL поддерживает все три режима [11]. По умолчанию работа происходит в режиме автофиксации. Отключить этот режим в psql можно командой

```
\set AUTOCOMMIT off.
```

Восстановить режим автофиксации в psql позволяет команда

```
\set AUTOCOMMIT on.
```

В pgAdmin режим автофиксации устанавливается и отменяется в настройках самой программы:



Для имитации параллельной работы нескольких пользователей лучше использовать несколько открытых окон *Query Tool* в программе pgAdmin.

По умолчанию каждый простой запрос из одной инструкции SQL в PostgreSQL является отдельной транзакцией и фиксируется по завершении (автофиксация):

```
1 INSERT INTO users VALUES(2, 'Петров', 20);
```

Data Output Messages Notifications

INSERT 0 1

В таблицу внесены изменения, которые сохранены в базе данных.

```
1 SELECT * FROM users;
```

Data Output Messages Notifications

	id [PK] integer	name character varying (30)	age integer
1	1	Иванов	45
2	2	Петров	20

Однако если сообщение простого запроса содержит несколько операторов SQL, разделенных точкой с запятой, и если среди них нет явных команд управления транзакциями, то эти операторы выполняются в одной транзакции (неявный режим).

```
1 INSERT INTO users VALUES(3, 'Сидорова', 20);
2 SELECT 1/0;
3 INSERT INTO users VALUES(4, 'Николаев', 30);
```

Data Output Messages Notifications

ERROR: ОШИБКА: деление на ноль

В этом случае ошибка деления на ноль в *SELECT* приведет к откату результата первого *INSERT*. Более того, вследствие прерывания обработки сообщения на первой ошибке второй *INSERT* не будет выполняться совсем.

Операторы в составном сообщении запроса выполняются в неявном блоке транзакции. Неявный блок транзакции автоматически закрывается в конце сообщения Query – либо неявно фиксируется при отсутствии ошибок, либо неявно откатывается в противном случае (как для одного оператора).

Можно явно установить границы транзакций, сообщив, какие операции составляют транзакцию (явный режим). Используются следующие инструкции SQL.

- *START TRANSACTION* – сообщает системе, что приложение начинает новую транзакцию. То же самое делает инструкция *BEGIN*. PostgreSQL выдаст сообщение об ошибке, если приложение уже выполняет транзакцию. Такое поведение соответствует стандарту SQL (другие СУБД начинают вложенную транзакцию).
- *COMMIT* – сообщает, что все операции транзакции завершены и ее нужно зафиксировать. После фиксации можно начать новую транзакцию командой *BEGIN*.
- *ROLLBACK* – вызывает откат текущей транзакции по инициативе приложения. СУБД отменяет все изменения, выполненные этой транзакцией, и завершает ее. После этого приложение может начать новую транзакцию.

В инструкциях *BEGIN*, *COMMIT* и *ROLLBACK* можно указывать необязательное слово *TRANSACTION*, которое не оказывает никакого влияния на их работу.

Пример использования операторов управления транзакциями:

```
1 BEGIN;
2 INSERT INTO users VALUES(3, 'Сидорова', 20);
3 COMMIT;
4 INSERT INTO users VALUES(4, 'Николаев', 30);
5 SELECT 1/0;
```

Data Output Messages Notifications

ERROR: ОШИБКА: деление на ноль

Результат первого *INSERT* фиксируется командой *COMMIT*. Второй *INSERT* и последующий *SELECT* будут обрабатываться в другой транзакции, поэтому ошибка деления на ноль приведет к откату второго *INSERT* и не затронет первый. В этом можно убедиться, просмотрев содержимое таблицы *users*:

1	<code>SELECT * FROM users;</code>																
Data Output Messages Notifications																	
	<table border="1"> <thead> <tr> <th></th> <th><code>id</code> [PK] integer</th> <th><code>name</code> character varying (30)</th> <th><code>age</code> integer</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>1</td> <td>Иванов</td> <td>45</td> </tr> <tr> <td>2</td> <td>2</td> <td>Петров</td> <td>20</td> </tr> <tr> <td>3</td> <td>3</td> <td>Сидорова</td> <td>20</td> </tr> </tbody> </table>		<code>id</code> [PK] integer	<code>name</code> character varying (30)	<code>age</code> integer	1	1	Иванов	45	2	2	Петров	20	3	3	Сидорова	20
	<code>id</code> [PK] integer	<code>name</code> character varying (30)	<code>age</code> integer														
1	1	Иванов	45														
2	2	Петров	20														
3	3	Сидорова	20														

Транзакции в сеансах и сообщениях. В PostgreSQL необходимо учитывать следующие особенности использования транзакций в сеансах и сообщениях.

1. Если в рамках сеанса уже начат блок транзакции (выполнен *BEGIN*), сообщение Query просто продолжает этот блок независимо от того, один в нем оператор или несколько. Но если сообщение Query содержит команду *COMMIT* или *ROLLBACK*, закрывающую существующий блок транзакций, то все последующие команды в нем выполняются в неявном блоке транзакции.

2. Напротив, если составное сообщение Query содержит команду *BEGIN*, она начинает обычный блок транзакции, который будет закончен только явными командами *COMMIT* или *ROLLBACK*, в каком бы сообщении Query, текущем или последующих, они ни содержались. Если *BEGIN* следует за операторами, которые выполнялись в неявном блоке транзакции, эти операторы не фиксируются немедленно – они задним числом включаются в новый обычный блок транзакции.

3. Операторы *COMMIT* и *ROLLBACK* в неявном блоке транзакции закрывают неявный блок, но будет выдано предупреждение, так как *COMMIT* или *ROLLBACK* без предшествующего *BEGIN* могут выполняться по ошибке. Если за этими операторами следуют другие, для них будет начат новый неявный блок транзакции.

4. Точки сохранения в неявных блоках транзакций не допускаются, они будут конфликтовать с правилом автоматического закрытия блока при любой ошибке.

ВНИМАНИЕ!

Вне зависимости от наличия любых команд управления транзакциями, выполнение сообщения Query останавливается при первой же ошибке.

Выполним следующие команды в одном сообщении Query:

```
1 BEGIN;
2 SELECT 1/0;
3 ROLLBACK;
```

Data Output Messages Notifications

ERROR: ОШИБКА: деление на ноль

Сеанс останется внутри прерванного обычного блока транзакции, так как команда *ROLLBACK* не достигается после ошибки деления на ноль. Для приведения сеанса в порядок потребуется выполнить еще одну команду *ROLLBACK*.

Лексический и синтаксический анализ производится для всей строки запроса до выполнения какой-либо ее части. Простые ошибки (опечатка в ключевом слове) в последующих операторах могут привести к тому, что не будет выполнен ни один из предшествующих операторов. Это обычно незаметно для пользователей, так как эти операторы все равно откатились бы при выполнении в неявном блоке транзакции. Но эта особенность может проявиться при попытке выполнить несколько транзакций в одном составном запросе. Например, допущена опечатка:

```
1 BEGIN;
2 INSERT INTO users VALUES(4, 'Николаев', 30);
3 COMMIT;
4 INSERT INTO users VALUES(5, 'Дмитриева', 35);
5 SELCT 1/0;
```

Data Output Messages Notifications

ERROR: ОШИБКА: ошибка синтаксиса (примерное положение: "SELCT")
LINE 5: SELCT 1/0;
^

1	<code>SELECT * FROM users;</code>
Data Output Messages Notifications	
id [PK] integer	name character varying (30)
1	Иванов
2	Петров
3	Сидорова

Ни один из операторов не выполнен, и даже первый *INSERT* не зафиксирован. Ошибки, выявленные на стадии семантического анализа или позже (опечатки в имени таблиц или столбца), такого влияния не оказывают.

Точки сохранения. Для длинных транзакций целесообразно использовать точки сохранения. Точка сохранения – специальная отметка внутри транзакции, позволяющая откатить все команды, выполненные после нее, и восстановить состояние на момент установки этой точки. Команда создания точки сохранения запоминает состояние всех данных, измененных транзакцией к моменту выполнения команды. Этот механизм дает возможность организовать сложное поведение в рамках одной транзакции, но для других транзакций ее поведение остается атомарным.

Определить новую точку сохранения в текущей транзакции можно командой

SAVEPOINT имя_точки_сохранения;

Для отката к установленной точке сохранения предназначена команда

ROLLBACK TO имя_точки_сохранения;

Уничтожить точку сохранения, сохранив изменения, произведенные после того, как она была установлена, можно командой

RELEASE SAVEPOINT имя_точки_сохранения.

Точки сохранения могут быть установлены только внутри блока транзакции. В одной транзакции можно определить несколько точек сохранения.

Пример установления точки сохранения, а затем отмены действия всех команд, выполненных после установленной точки:

```
1 BEGIN;
2     INSERT INTO users VALUES(4, 'Николаев', 30);
3     SAVEPOINT my_savepoint;
4     INSERT INTO users VALUES(5, 'Дмитриева', 35);
5     ROLLBACK TO SAVEPOINT my_savepoint;
6     INSERT INTO users VALUES(6, 'Михеева', 17);
7 COMMIT;
```

Data Output Messages Notifications

COMMIT

```
1 SELECT * FROM public.users
2 ORDER BY id ASC
```

Data Output Messages Notifications

	id [PK] integer	name character varying (30)	age integer
1	1	Иванов	45
2	2	Петров	20
3	3	Сидорова	20
4	4	Николаев	30
5	6	Михеева	17

Пример, показывающий, как установить и уничтожить точку сохранения:

```
1 BEGIN;
2     INSERT INTO users VALUES (7, 'Танина', 21);
3     SAVEPOINT my_savepoint;
4     INSERT INTO users VALUES (8, 'Димов', 32);
5     RELEASE SAVEPOINT my_savepoint;
6 COMMIT;
```

В следующем примере иллюстрируется работа операторов создания точек сохранения. Очистим таблицу *users* и выполним следующий запрос:

```

1 BEGIN;
2   INSERT INTO users VALUES (5, 'Берг', 45);
3   SAVEPOINT my_savepoint;
4   INSERT INTO users VALUES (9, 'Антонов', 37);
5   SAVEPOINT my_savepoint;
6   INSERT INTO users VALUES (10, 'Шплинт', 54);
7   -- откат ко второй точке сохранения
8   ROLLBACK TO SAVEPOINT my_savepoint;
9   -- освобождение второй точки сохранения
10  RELEASE SAVEPOINT my_savepoint;
11  -- откат к первой точке сохранения
12  ROLLBACK TO SAVEPOINT my_savepoint;
13 COMMIT;
14 SELECT * FROM users; -- вывод только первой строки

```

Data Output Messages Notifications

The screenshot shows a PostgreSQL client interface. At the top, there's a toolbar with various icons. Below it is a tab bar with 'Data Output' selected, followed by 'Messages' and 'Notifications'. The main area displays a table with three columns: 'id', 'name', and 'age'. The data shows a single row with id 5, name 'Берг', and age 45. There are edit icons next to each column header and the first data cell.

	id [PK] integer	name character varying (30)	age integer
1	5	Берг	45

В блоке транзакции выполняется ввод строки и создается точка сохранения, вводится вторая строка и создается точка сохранения с тем же именем. Затем вводится третья строка и выполняется откат ко второй точке сохранения. Освобождается вторая точка сохранения и выполняется откат к первой точке сохранения. После фиксации транзакции в таблице остается только первая строка.

Стандарт SQL требует, чтобы точка сохранения уничтожалась автоматически при создании другой точки сохранения с тем же именем.

В PostgreSQL старая точка сохранения остается, хотя при откате или уничтожении будет выбираться только самая последняя (после уничтожения последней точки сохранения, доступной для команд *ROLLBACK TO SAVEPOINT* и *RELEASE SAVEPOINT*, становится следующая). В остальном оператор *SAVEPOINT* полностью соответствует стандарту.

Практическая работа

Необходимо для созданной базы данных:

- создать транзакцию с точкой сохранения в режиме явных транзакций, произвести откат к точке сохранения и фиксацию;
- составить отчет.

Пример выполнения задания

Объединим в явную транзакцию несколько операций по добавлению в таблицу *categories* новых записей, создадим точку сохранения, добавим еще записи, произведем откат транзакции к точке сохранения.

```
1 BEGIN;
2     INSERT INTO categories (cat_ID, cat_name, cat_overcat)
3         VALUES (12, 'Измерительные инструменты', NULL);
4     INSERT INTO categories (cat_ID, cat_name, cat_overcat)
5         VALUES (13, 'Клининговое оборудование', NULL);
6     SAVEPOINT point_one;
7     INSERT INTO categories (cat_ID, cat_name, cat_overcat)
8         VALUES (14, 'Оснастка', NULL);
9     INSERT INTO categories (cat_ID, cat_name, cat_overcat)
10        VALUES (15, 'Наборы электроинструментов', NULL);
11    ROLLBACK TO SAVEPOINT point_one;
12 COMMIT;
```

Data Output Messages Notifications

COMMIT

Проверим полученный результат:

```
1 SELECT * FROM categories;
```

Data Output Messages Notifications

The screenshot shows a table with three columns: cat_id, cat_name, and cat_overcat. The data is as follows:

	cat_id [PK] integer	cat_name	cat_overcat integer
1	1	Сверлильные инструменты	[null]
2	2	Пильно-отрезные инструменты	[null]
3	3	Инструменты для обработки поверхностей	[null]
4	4	Перфораторы	1
5	5	Дрели	1
6	6	Шуруповерты	1
7	7	Дисковые пилы	2
8	8	Цепные пилы	2
9	9	Лобзики электрические	2
10	10	Шлифовальные машины	3
11	11	Электрорубанки	3
12	12	Измерительные инструменты	[null]
13	13	Клининговое оборудование	[null]

Лабораторная работа № 10

КУРСОРЫ

Теоретические сведения

Рассмотрим следующие вопросы:

- извлечение данных с помощью курсора;
- возврат курсора из функции.

Декларативный язык SQL обеспечивает работу с множествами строк. Процедурный язык PL/pgSQL работает с каждой строкой отдельно, используя явные циклы. Курсоры как раз и подразумевают итеративную обработку.

Причины использования курсоров:

- полная выборка слишком велика, и результаты обрабатываются по частям;
- нужна выборка неизвестного заранее размера;
- необходимо предоставить управление выборкой клиенту;
- требуется построчная обработка (обычно ее можно заменить чистым SQL, код в итоге окажется проще и будет быстрее работать).

Для манипулирования курсорами используются переменные, параметры и результаты функций типа *refcursor*. При этом различают:

1) несвязанные с запросом курсорные переменные:

- объявляется переменная типа *refcursor*;
- конкретный запрос указывается при открытии;

2) связанные с запросом курсорные переменные:

- при объявлении указывается запрос (возможно, с параметрами);
- при открытии указываются фактические значения параметров.

Значением курсорной переменной является имя курсора, которое задают явно или оно будет сгенерировано PL/pgSQL автоматически. Переменные PL/pgSQL в запросе становятся неявными параметрами (значения подставляются при открытии курсора). Запрос, открытый с помощью курсора, предварительно подготавливается.

Операции с курсором. Курсор объявляется командой *DECLARE*. Для работы с ним необходимо открыть его командой *OPEN* для выполнения запроса.

Для получения данных из курсора используется команда *FETCH*. Выборка строк из курсора возможна только построчно. Команда *FETCH INTO* читает строку в переменную составного типа или в набор скалярных переменных (как *SELECT INTO*). Для изменения текущей позиции в множестве строк, составляющих результат запроса, применяется команда *MOVE*. Курсор можно явно закрыть командой *CLOSE* и использовать его для выполнения другого запроса. При завершении транзакции все открытые в ней курсоры закрываются автоматически.

Кроме последовательного считывания данных в прямом направлении возможны движение в обратном направлении, установка на любую строку по ее номеру или перемещение на заданное количество строк от текущей. Цикл *FOR* для чтения результатов запроса также создает неявный курсор. Такой курсор считывается всегда в прямом направлении, другие операции с ним невозможны.

Процедурная обработка подразумевает применение циклов для перебора и обработки строк, возвращаемых курсором. Возможный синтаксис цикла по курсору:

OPEN курсорная_переменная FOR запрос;

LOOP

FETCH курсорная_переменная INTO цель;

EXIT WHEN NOT FOUND;

тело-цикла

END LOOP;

CLOSE курсорная_переменная;

Для выхода из цикла используется переменная *FOUND*, которая показывает, была ли выбрана командой *FETCH* очередная строка.

Наиболее простой пример использования курсора в транзакции:

```
1 BEGIN WORK;
2 -- Создание курсора
3 DECLARE cur SCROLL CURSOR FOR SELECT * FROM employees;
4 -- Получение первых 2 строк через курсор cur
5 FETCH FORWARD 2 FROM cur;
6 -- Получение предыдущей строки
7 FETCH PRIOR FROM cur;
8 -- Закрытие курсора и завершение транзакции
9 CLOSE cur;
10 COMMIT WORK;
```

В утилите *pgAdmin* будет выведено только сообщение о фиксации транзакции. Убедитесь в том, что курсор работает, можно только приостановив выполнение транзакции:

```
1 BEGIN WORK;
2 -- Создание курсора
3 DECLARE cur SCROLL CURSOR FOR SELECT * FROM employees;
4 -- Получение первых 2 строк через курсор cur
5 FETCH FORWARD 2 FROM cur;
```

Data Output Messages Notifications

The screenshot shows a pgAdmin interface with a table titled 'Data Output'. The table has three columns: 'emp_id [PK] integer', 'emp_name text', and 'emp_position text'. There are two rows of data: Row 1 contains emp_id 1, emp_name 'Ребров Андрей Андреевич', and emp_position 'менеджер'; Row 2 contains emp_id 2, emp_name 'Котов Владимир Иванович', and emp_position 'старший менеджер'.

	emp_id [PK] integer	emp_name	emp_position
1	1	Ребров Андрей Андреевич	менеджер
2	2	Котов Владимир Иванович	старший менеджер

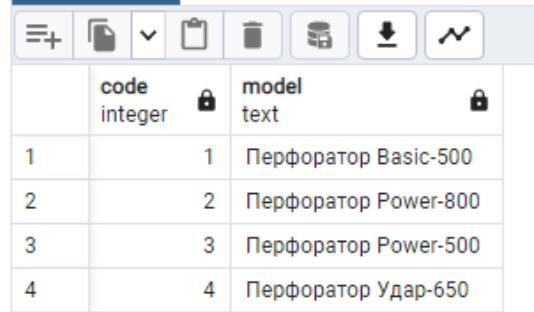
Еще один пример использования курсора для обработки результатов запроса, возвращающего несколько строк. Создаем функцию с курсором:

```
1 CREATE OR REPLACE FUNCTION instr_set_cur()
2 RETURNS SETOF instrum
3 LANGUAGE plpgsql AS $$
4 DECLARE
5     cur refcursor;
6     v record;
7 BEGIN
8     OPEN cur FOR SELECT * FROM products;
9     LOOP
10        FETCH NEXT FROM cur INTO v;
11        EXIT WHEN NOT FOUND;
12        RETURN NEXT ROW(v.prod_ID, v.prod_name)::instrum;
13    END LOOP;
14    CLOSE cur;
15 END;
16 $$;
```

Затем обычным образом, с помощью *SELECT*, извлекаем строки из курсора:

```
1 SELECT * FROM instr_set_cur()
2 LIMIT 4;
```

Data Output Messages Notifications

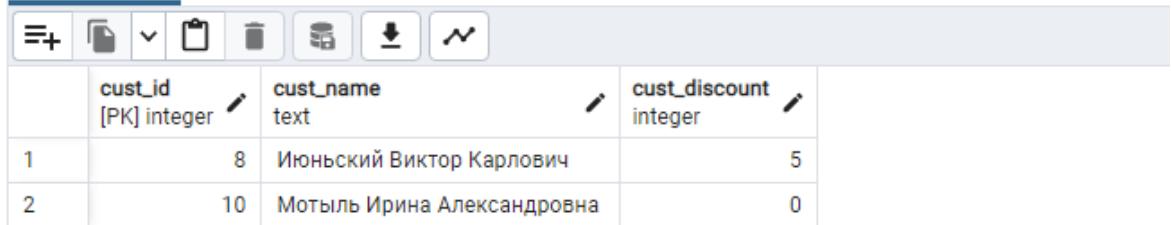


	code integer	model text
1	1	Перфоратор Basic-500
2	2	Перфоратор Power-800
3	3	Перфоратор Power-500
4	4	Перфоратор Удар-650

Рассмотрим еще один пример. Получим список заказчиков из г. Санкт-Петербурга. Задача решается путем использования простого оператора *SELECT*, но никакие дальнейшие действия с полями результирующего набора невозможны.

```
1 SELECT customers.cust_ID, customers.cust_name, customers.cust_discount
2 FROM customers
3 WHERE customers.cust_address = 'Санкт-Петербург';
```

Data Output Messages Notifications



	cust_id [PK] integer	cust_name text	cust_discount integer
1	8	Июньский Виктор Карлович	5
2	10	Мотыль Ирина Александровна	0

Если использовать курсор, то можно проверять содержимое записей, а также выполнять различные операции с использованием содержимого полей:

```
1 DO $$
2 DECLARE
3     c_ID customers.cust_ID%type;
4     c_name customers.cust_name%type;
5     c_discount customers.cust_discount%type;
6     my_cur CURSOR FOR
```

```

7   SELECT customers.cust_ID, customers.cust_name, customers.cust_discount
8   FROM customers
9   WHERE customers.cust_address = 'Санкт-Петербург';
10  BEGIN
11    OPEN my_cur;
12  LOOP
13    FETCH my_cur INTO c_ID, c_name, c_discount;
14    EXIT WHEN not found;
15    RAISE NOTICE 'ID: %', c_ID;
16    RAISE NOTICE 'ФИО: %', c_name;
17    RAISE NOTICE 'Скидка: %', c_discount;
18  END LOOP;
19  CLOSE my_cur;
20 END;
21 $$;
22 LANGUAGE plpgsql;

```

Data Output Messages Notifications

ЗАМЕЧАНИЕ:	ID: 8
ЗАМЕЧАНИЕ:	ФИО: Июньский Виктор Карлович
ЗАМЕЧАНИЕ:	Скидка: 5
ЗАМЕЧАНИЕ:	ID: 10
ЗАМЕЧАНИЕ:	ФИО: Мотыль Ирина Александровна
ЗАМЕЧАНИЕ:	Скидка: 0

Обработка курсора в цикле. Еще один из вариантов цикла *FOR* позволяет перебирать строки, возвращенные курсором. Его синтаксис:

FOR запись IN курсорная_переменная LOOP

тело-цикла

END LOOP;

Этот вариант не предусматривает указания запроса, так что курсорная переменная должна быть заранее связана с запросом при объявлении. Курсор не может быть открытым. Команда *FOR* автоматически открывает курсор и автоматически закрывает его при завершении цикла. Переменная цикла автоматически определяется как переменная типа *record* и существует только внутри цикла (другие объявленные переменные с таким именем игнорируются в цикле). Каждая возвращаемая курсором строка последовательно присваивается этой переменной, и выполняется тело цикла.

Возврат курсора из функции. Открытый курсор можно возвращать как результат функции и принимать его в качестве параметра. Для этого в функции открывается курсор и его имя возвращается вызывающему. Вызывающий может извлекать строки из курсора. Курсор может быть закрыт вызывающим или он автоматически закрывается при завершении транзакции.

Имя портала для курсора указывает разработчик или оно генерируется автоматически. Указать имя портала можно, присвоив строку переменной *refcursor* перед его открытием. Значение этой строки команда *OPEN* использует как имя портала. Если переменная *refcursor* имеет значение *NULL*, то *OPEN* автоматически сгенерирует имя и присвоит его переменной *refcursor*.

Следующий пример показывает один из способов передачи имени курсора вызывающему (должна быть запущена транзакция):

```
1 CREATE OR REPLACE FUNCTION ref_func(refcursor)
2 RETURNS refcursor
3 LANGUAGE 'plpgsql'
4 AS $$
5 ▼ BEGIN
6     OPEN $1 FOR SELECT * FROM manufacturers;
7     RETURN $1;
8 END;
9 $$;
10 -- транзакция
11 BEGIN;
12 SELECT ref_func('funcursor');
13 FETCH ALL IN funcursor;
14 COMMIT;
```

Чтобы увидеть результат, приостановим выполнение транзакции:

```
1 BEGIN;
2 SELECT ref_func('funcursor');
3 FETCH ALL IN funcursor;
```

Data Output Messages Notifications

	man_id [PK] integer	man_name	man_address
1	1	Мастер	Россия
2	2	Техника	Россия
3	3	Deco	Китай
4	4	Work	Китай
5	5	Kraut	Германия
6	6	Yamamoto	Япония

Практическая работа

Необходимо для созданной базы данных:

- решить несколько задач с использованием курсоров, включив курсор в хранимую процедуру или функцию;
- составить отчет.

Пример выполнения задания

1. Используем курсор при написании хранимой процедуры *order_select*, обеспечивающей выборку данных для таблицы *orders*.

```

1 CREATE OR REPLACE PROCEDURE order_select(
2     IN order_ID integer,
3     IN customer_ID integer,
4     IN employee_ID integer)
5 LANGUAGE 'plpgsql'
6 AS $$
7 DECLARE
8     c_ord_ID integer; c_cust_ID integer; c_cust_name text;
9     c_emp_ID integer; c_emp_name text;    c_ord_date date;
10    cur CURSOR FOR SELECT
11        o.ord_ID, o.ord_cust_ID, c.cust_name,
12        o.ord_emp_ID, e.emp_name, o.ord_date
13    FROM customers c JOIN orders o ON c.cust_ID = o.ord_cust_ID
14          JOIN employees e ON e.emp_ID = o.ord_emp_ID
15    WHERE (order_ID IS NULL OR o.ord_ID = order_ID) AND
16        (customer_ID IS NULL OR o.ord_cust_ID = customer_ID) AND
17        (employee_ID IS NULL OR o.ord_emp_ID = employee_ID);
18    BEGIN
19        OPEN cur;
20    LOOP
21        FETCH cur INTO c_ord_ID, c_cust_ID, c_cust_name, c_emp_ID,
22        c_emp_name, c_ord_date;
23        EXIT WHEN not found;
24        RAISE NOTICE 'Заказ: %, %, %, %, %', c_ord_ID, c_cust_ID,
25                           c_cust_name, c_emp_ID, c_emp_name, c_ord_date;
26    END LOOP;
27    CLOSE cur;
28 END;
29 $$;
```

Процедура *order_select* использует базовую таблицу *orders* и связанные таблицы *customers* и *employees*. Параметры процедуры – *order_ID*, *customer_ID* и *employee_ID* обеспечивают достаточно широкие возможности по выборке данных.

Вызов процедуры *order_select* с использованием для всех трех параметров значений *NULL* позволяет получить список всех заказов.

```
1 CALL order_select(NULL, NULL, NULL);
```

Data Output Messages Notifications

ЗАМЕЧАНИЕ: Заказ: 1, 1, Сергеев Владимир Николаевич, 1, Ребров Андрей Андреевич, 2023-01-15
ЗАМЕЧАНИЕ: Заказ: 2, 6, Кирьянова Людмила Петровна, 1, Ребров Андрей Андреевич, 2023-01-17
ЗАМЕЧАНИЕ: Заказ: 3, 5, Тромб Павел Гаврилович, 4, Батова Алина Дмитриевна, 2023-01-17
ЗАМЕЧАНИЕ: Заказ: 4, 9, Простой Андрей Андреевич, 3, Колосова Елена Сергеевна, 2023-01-23
ЗАМЕЧАНИЕ: Заказ: 5, 6, Кирьянова Людмила Петровна, 1, Ребров Андрей Андреевич, 2023-02-02
ЗАМЕЧАНИЕ: Заказ: 6, 5, Тромб Павел Гаврилович, 4, Батова Алина Дмитриевна, 2023-02-10
ЗАМЕЧАНИЕ: Заказ: 7, 3, Иванова Ольга Сергеевна, 2, Котов Владимир Иванович, 2023-02-23
ЗАМЕЧАНИЕ: Заказ: 8, 5, Тромб Павел Гаврилович, 4, Батова Алина Дмитриевна, 2023-02-23
ЗАМЕЧАНИЕ: Заказ: 9, 10, Мотыль Ирина Александровна, 2, Котов Владимир Иванович, 2023-02-23
ЗАМЕЧАНИЕ: Заказ: 10, 2, Волков Иван Владимирович, 1, Ребров Андрей Андреевич, 2023-03-04
ЗАМЕЧАНИЕ: Заказ: 11, 4, Васильев Александр Иванович, 4, Батова Алина Дмитриевна, 2023-03-04
ЗАМЕЧАНИЕ: Заказ: 12, 7, Антипов Алексей Владимирович, 2, Котов Владимир Иванович, 2023-03-16
ЗАМЕЧАНИЕ: Заказ: 13, 1, Сергеев Владимир Николаевич, 3, Колосова Елена Сергеевна, 2023-03-24
ЗАМЕЧАНИЕ: Заказ: 14, 8, Июньский Виктор Карлович, 3, Колосова Елена Сергеевна, 2023-03-25
ЗАМЕЧАНИЕ: Заказ: 15, 5, Тромб Павел Гаврилович, 2, Котов Владимир Иванович, 2023-03-26

Вызов процедуры *order_select* с указанием *order_ID* = 1 (значения остальных параметров – *NULL*) позволяет получить информацию о заказе с кодом 1:

```
1 CALL order_select(1, NULL, NULL);
```

Data Output Messages Notifications

ЗАМЕЧАНИЕ: Заказ: 1, 1, Сергеев Владимир Николаевич, 1, Ребров Андрей Андреевич, 2023-01-15

Вызов процедуры с *customer_ID* = 3 (значения остальных параметров – *NULL*) позволяет получить заказ пользователя с кодом 3:

```
1 CALL order_select(NULL, 3, NULL);
```

Data Output Messages Notifications

ЗАМЕЧАНИЕ: Заказ: 7, 3, Иванова Ольга Сергеевна, 2, Котов Владимир Иванович, 2023-02-23

Вызов процедуры с *employee_ID* = 1 (значения остальных параметров – *NULL*) позволяет получить список всех заказов, оформленных сотрудником с кодом 1:

```
1 CALL order_select(NULL, NULL, 1);
```

Data Output Messages Notifications

ЗАМЕЧАНИЕ: Заказ: 1, 1, Сергеев Владимир Николаевич, 1, Ребров Андрей Андреевич, 2023-01-15

ЗАМЕЧАНИЕ: Заказ: 2, 6, Кирьянова Людмила Петровна, 1, Ребров Андрей Андреевич, 2023-01-17

ЗАМЕЧАНИЕ: Заказ: 5, 6, Кирьянова Людмила Петровна, 1, Ребров Андрей Андреевич, 2023-02-02

ЗАМЕЧАНИЕ: Заказ: 10, 2, Волков Иван Владимирович, 1, Ребров Андрей Андреевич, 2023-03-04

Вызов процедуры с *customer_ID* = 6 и *employee_ID* = 1 (*order_ID* = *NULL*) даст список заказов пользователя с кодом 6, оформленных сотрудником с кодом 1:

```
1 CALL order_select(NULL, 6, 1);
```

Data Output Messages Notifications

ЗАМЕЧАНИЕ: Заказ: 2, 6, Кирьянова Людмила Петровна, 1, Ребров Андрей Андреевич, 2023-01-17

ЗАМЕЧАНИЕ: Заказ: 5, 6, Кирьянова Людмила Петровна, 1, Ребров Андрей Андреевич, 2023-02-02

2. Создадим хранимую процедуру, записывающую в новую таблицу *february* все заказы, сделанные в феврале 2023 г. Нужна пустая таблица *february*:

```
1 CREATE TABLE february (
2     f_ord_ID integer NOT NULL,
3     f_ord_cust_ID integer NOT NULL,
4     f_ord_emp_ID integer NOT NULL,
5     f_ord_date date NOT NULL,
6     CONSTRAINT f_orders_PK PRIMARY KEY (f_ord_ID)
7 );
```

Хранимая процедура *ord_febr ()* использует курсор *cur*, который в цикле читает данные из таблицы *orders* и добавляет их в таблицу *february*:

```

1 CREATE OR REPLACE PROCEDURE ord_febr()
2 LANGUAGE 'plpgsql'
3 AS $$ 
4 DECLARE
5     order_ID integer; customer_ID integer; employee_ID integer;
6     order_date date;
7     cur CURSOR FOR SELECT * FROM orders
8         WHERE ord_date BETWEEN '01-02-2023' AND '28-02-2023';
9 ▼ BEGIN
10    OPEN cur;
11 ▼ LOOP
12        FETCH cur INTO order_ID, customer_ID, employee_ID, order_date;
13        EXIT WHEN not found;
14        INSERT INTO february VALUES
15            (order_ID, customer_ID, employee_ID, order_date);
16    END LOOP;
17    CLOSE cur;
18 END;
19 $$
```

Вызываем процедуру. Для просмотра результата выполнения процедуры используем полную выборку из таблицы *February*:

```

1 CALL ord_febr();
2 SELECT * FROM february
3 ORDER BY f_ord_id ASC;
```

Data Output Messages Notifications

	f_ord_id [PK] integer	f_ord_cust_id integer	f_ord_emp_id integer	f_ord_date date
1	5	6	1	2023-02-02
2	6	5	4	2023-02-10
3	7	3	2	2023-02-23
4	8	5	4	2023-02-23
5	9	10	2	2023-02-23

Лабораторная работа № 11

ТРИГГЕРЫ

Теоретические сведения

Рассмотрим следующие вопросы:

- создание триггеров, срабатывающих при модификации данных;
- создание триггеров событий.

PostgreSQL может не только выполнять действия, явно указанные приложением, но также реагировать на события в БД. Такая активность реализуется с помощью аппарата триггеров. *Триггер* – функция на процедурном языке, которая автоматически вызывается системой при срабатывании связанных с ней предписаний.

В PostgreSQL различают [11, 12]:

- триггеры, срабатывающие при модификации данных (*INSERT*, *UPDATE*, *DELETE* и *TRUNCATE*);
- триггеры событий, срабатывающие при выполнении операторов языка описания данных (*ALTER*, *CREATE*, *DROP*, *GRANT*, *REVOKE*).

Процедурный код в триггере может дополнить или изменить семантику стандартных операторов SQL. Так, триггеры можно использовать для проверки условий целостности, которые невозможно описать стандартными средствами языка SQL, или для регистрации изменений, выполняемых приложением, в другой таблице.

Действия в триггере выполняются всегда, когда возникает специфицированная ситуация, в рамках той же транзакции. Приложение не может обойти или отменить действие триггера. Это делает триггеры полезными, но и потенциально опасными, так как ошибки в коде триггеров могут разрушить функциональность СУБД.

Триггер в PostgreSQL задается функцией, определяемой в БД до его создания. Триггерная функция не должна иметь аргументов. Тип возвращаемого значения – *trigger* (для триггеров, срабатывающих при изменениях данных) или *event trigger* (для триггеров событий).

Такие функции могут быть написаны на любом процедурном языке, доступном в PostgreSQL, при этом способ доступа к контексту, в котором возбужден триггер, зависит от языка программирования. В функциях на PL/pgSQL для триггеров автоматически определяются специальные локальные переменные с именами вида *TG_имя*, описывающие условие, по-влекшее вызов триггера [10].

Триггеры, срабатывающие при модификации данных. Для определения триггера модификации используется оператор *CREATE TRIGGER*, в котором указывается следующая информация.

- Уровень триггера.

Триггеры могут быть определены на уровне операторов SQL (*FOR EACH STATEMENT*) или на уровне строк (*FOR EACH ROW*). На уровне оператора триггер выполняется один раз при исполнении оператора. На уровне строк триггер вызывается для каждой строки таблицы, которая обновляется оператором SQL.

- Операторы, выполнение которых возбуждает триггер (*INSERT, UPDATE, DELETE, TRUNCATE*).
- Объект базы данных, при модификации которого запускается триггер (таблица или представление).
- Время срабатывания триггера (*BEFORE, AFTER, INSTEAD OF*).

Триггеры *BEFORE* срабатывают непосредственно до, а триггеры *AFTER* – сразу после возбуждающего оператора. Триггеры *INSTEAD OF* используются для представлений и позволяют определить, какие действия должны выполняться вместо операций модификации данных.

- Дополнительные условия, ограничивающие запуск триггера (*WHEN*).
- Функция триггера, выполняющая необходимые действия.
- Возможно, дополнительные параметры функции триггера.

При этом одна и та же функция триггера может использоваться для определения разных триггеров.

Когда функция на PL/pgSQL срабатывает как триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных:

- *NEW* (тип данных *record*). Переменная содержит новую строку БД для команд *INSERT/UPDATE* в триггерах уровня строки. В триггерах уровня оператора и для команды *DELETE* эта переменная имеет значение *NULL*.
- *OLD* (тип данных *record*). Переменная содержит старую строку БД для команд *UPDATE/DELETE* в триггерах уровня строки. В триггерах уровня оператора и для команды *INSERT* эта переменная имеет значение *NULL*.
- *TG_NAME* (тип *name*). Переменная содержит имя сработавшего триггера.
- *TG_WHEN* (тип данных *text*). Стока, содержащая *BEFORE*, *AFTER* или *INSTEAD OF*, в зависимости от определения триггера.
- *TG_LEVEL* (тип данных *text*). Стока, содержащая *ROW* или *STATEMENT*, в зависимости от определения триггера.
- *TG_OP* (тип данных *text*). Стока, содержащая *INSERT*, *UPDATE*, *DELETE* или *TRUNCATE*, в зависимости от того, для какой операции сработал триггер.
- *TG_RELID* (тип данных *oid*). *OID* таблицы, для которой сработал триггер.
- *TG_RELNAME* (тип данных *name*). Имя таблицы, для которой сработал триггер. Переменная устарела, рекомендуется использовать *TG_TABLE_NAME*.
- *TG_TABLE_NAME* (тип *name*). Имя таблицы, для которой сработал триггер.
- *TG_TABLE_SCHEMA* (тип данных *name*). Имя схемы, содержащей таблицу, для которой сработал триггер.
- *TG_NARGS* (тип данных *integer*). Число аргументов в команде *CREATE TRIGGER*, которые передаются в триггерную функцию.

- *TG_ARGV[]* (тип данных массив *text*). Аргументы от оператора *CREATE TRIGGER*. Индекс массива начинается с 0. Для недопустимых значений индекса (меньше 0, или больше, или равно *tg_nargs*) возвращается *NULL*.

В PL/pgSQL для триггеров уровня строк определены переменные *OLD* и *NEW*, содержащие старое и новое значения строки. При этом для оператора *INSERT* не существует старого значения, а для *DELETE* – нового.

Триггеры *BEFORE* могут изменять значения атрибутов в переменной *NEW*. Чтобы выполнение оператора, возбудившего триггер, было продолжено, функция триггера должна вернуть непустое значение. В триггерах для *INSERT* и *UPDATE* это значение будет использовано как новое значение обновляемого кортежа, поэтому функция должна вернуть исходное или измененное значение переменной *NEW*.

Если функция триггера возвращает значение *NULL*, то для триггеров уровня строк прекращается обработка соответствующей строки, а для триггеров уровня оператора прекращается выполнение всего оператора. Однако откат транзакции ни в том, ни в другом случае не производится.

Рассмотрим функцию триггера, которая выводит значения переменных, определяющих контекст вызова триггера, и не выполняет никаких других действий:

```

1 CREATE OR REPLACE FUNCTION show_trigger_parameters()
2 RETURNS trigger
3 AS $$
4 ▼ BEGIN
5   RAISE NOTICE '%: % %.% % %',
6   TG_NAME, TG_OP, TG_TABLE_SCHEMA, TG_TABLE_NAME, TG_WHEN, TG_LEVEL;
7 ▼ IF TG_OP = 'DELETE' THEN
8   RETURN OLD;
9 ELSE
10  RETURN NEW;
11 END IF;
12 END;
13 $$ LANGUAGE plpgsql;
```

Используем эту функцию для создания нескольких триггеров:

```
1 CREATE TRIGGER row_before
2 BEFORE INSERT OR DELETE OR UPDATE
3 ON employees
4 FOR EACH ROW
5 EXECUTE PROCEDURE show_trigger_parameters();
```

```
1 CREATE TRIGGER row_after
2 AFTER INSERT OR DELETE OR UPDATE
3 ON employees
4 FOR EACH ROW
5 EXECUTE PROCEDURE show_trigger_parameters();
```

```
1 CREATE TRIGGER stmt_before
2 BEFORE INSERT OR DELETE OR UPDATE
3 ON employees
4 FOR EACH STATEMENT
5 EXECUTE PROCEDURE show_trigger_parameters();
```

```
1 CREATE TRIGGER stmt_after
2 AFTER INSERT OR DELETE OR UPDATE
3 ON employees
4 FOR EACH STATEMENT
5 EXECUTE PROCEDURE show_trigger_parameters();
```

При выполнении операции обновления таблицы *employees* будут возбуждены все описанные триггеры:

```
1 BEGIN TRANSACTION;
2 UPDATE employees
3 SET emp_position = 'Уволен'
4 WHERE emp_id = 1;
5 ROLLBACK;
```

Data Output Messages Notifications

ЗАМЕЧАНИЕ: stmt_before: UPDATE public.employees BEFORE STATEMENT
ЗАМЕЧАНИЕ: row_before: UPDATE public.employees BEFORE ROW
ЗАМЕЧАНИЕ: row_after: UPDATE public.employees AFTER ROW
ЗАМЕЧАНИЕ: stmt_after: UPDATE public.employees AFTER STATEMENT
ROLLBACK

Откат транзакции необходим, поскольку сохранять сделанные изменения в демонстрационной базе данных не нужно.

Триггеры событий. Для спецификации триггеров событий используется команда *CREATE EVENT TRIGGER*. Написание функций для этих триггеров и определение самих триггеров в целом аналогичны определениям функций и триггеров модификации данных.

PostgreSQL требует, чтобы функция, которая вызывается как событийный триггер, объявлялась без аргументов и типом возвращаемого значения был *event_trigger*. Когда функция на PL/pgSQL вызывается как событийный триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных:

- *TG_EVENT* (тип данных *text*). Стока, содержащая событие, для которого сработал триггер.
- *TG_TAG* (тип данных *text*). Переменная, содержащая тег команды, для которой сработал триггер.

Следующий пример демонстрирует реализацию функции событийного триггера на PL/pgSQL. Триггер в примере просто выдает сообщение *NOTICE* каждый раз, когда выполняется команда DDL.

```
1 CREATE OR REPLACE FUNCTION snitch()
2 RETURNS event_trigger
3 AS $$
4 BEGIN
5     RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
6 END;
7 $$ LANGUAGE plpgsql;
```

```
1 CREATE EVENT TRIGGER snitch
2 ON ddl_command_start
3 EXECUTE FUNCTION snitch();
```

```
1 CREATE TABLE table1(a integer);
2 DROP TABLE table1;
```

Data Output Messages Notifications

ЗАМЕЧАНИЕ: snitch: ddl_command_start CREATE TABLE
ЗАМЕЧАНИЕ: snitch: ddl_command_start DROP TABLE
DROP TABLE

Внутри функции триггера можно выполнять любые операторы SQL, допустимые в функциях. Это может вызвать каскадный запуск другого или того же самого триггера, в том числе может вызвать бесконечную рекурсию, за которую отвечает программист. Например, триггер на таблицу T1, регистрирующий изменения в таблице T2, может стать некорректным после того, как на таблице T2 определяется триггер, модифицирующий T1. Обновление любой из этих таблиц приведет к запуску триггера, модифицирующего другую таблицу (бесконечная рекурсия).

Ошибки, связанные с применением триггеров, трудно обнаруживать, потому что триггеры вызываются неявно. Перенос логики на уровень триггеров базы данных несет в себе существенные риски.

Практическая работа

Необходимо для созданной базы данных:

- написать два триггера (можно ограничиться триггерами, срабатывающими при модификации данных);
- составить отчет.

Пример выполнения задания

1. Создадим для базы данных *sales* триггер аудита, гарантирующий, что любое добавление, изменение или удаление строки в таблице *employees* будет зафиксировано в таблице аудита *emp_audit*. Кроме того, триггер фиксирует текущее время, имя пользователя и тип выполняемой операции. Проверим срабатывание этого триггера.

```
1 CREATE TABLE emp_audit(  
2     operation    char(1)    NOT NULL,  
3     stamp        timestamp NOT NULL,  
4     userid       text      NOT NULL,  
5     emp_ID       integer   NOT NULL,  
6     emp_name     text      NOT NULL,  
7     emp_position text      NOT NULL  
8 );
```

```

1 CREATE OR REPLACE FUNCTION process_emp_audit()
2 RETURNS TRIGGER
3 AS $$ 
4 BEGIN
5    IF (TG_OP = 'DELETE') THEN
6        INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
7    ELSIF (TG_OP = 'UPDATE') THEN
8        INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
9    ELSIF (TG_OP = 'INSERT') THEN
10       INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
11   END IF;
12   RETURN NULL; -- возвращаемое значение для триггера AFTER
13 END;
14 $$ LANGUAGE plpgsql;

```

```

1 CREATE TRIGGER emp_audit
2 AFTER INSERT OR UPDATE OR DELETE ON employees
3      FOR EACH ROW EXECUTE FUNCTION process_emp_audit();

```

```

1 INSERT INTO employees VALUES (5, 'Штольц Иван Карлович', 'стажер');
2 SELECT * FROM employees
3 ORDER BY emp_ID ASC;

```

Data Output Messages Notifications

The screenshot shows a table with three columns: emp_id, emp_name, and emp_position. The data is as follows:

	emp_id [PK] integer	emp_name	emp_position
1	1	Ребров Андрей Андреевич	менеджер
2	2	Котов Владимир Иванович	старший менеджер
3	3	Колосова Елена Сергеевна	менеджер
4	4	Батова Алина Дмитриевна	старший менеджер
5	5	Штольц Иван Карлович	стажер

1 SELECT * FROM emp_audit;

Data Output Messages Notifications

The screenshot shows a table with six columns: operation, stamp, userid, emp_id, emp_name, and emp_position. The data is as follows:

	operation character (1)	stamp timestamp without time zone	userid text	emp_id integer	emp_name	emp_position
1	I	2023-07-21 16:58:54.681	postgres	5	Штольц Иван Карлович	стажер

```
1 DELETE FROM employees WHERE emp_ID = 5;
2 SELECT * FROM emp_audit;
```

Data Output Messages Notifications

	operation character (1)	stamp timestamp without time zone	userid text	emp_id integer	emp_name text	emp_position text
1	I	2023-07-21 16:58:54.681	postgres	5	Штольц Иван Карлович	стажер
2	D	2023-07-21 17:07:58.427572	postgres	5	Штольц Иван Карлович	стажер

2. Создадим для базы данных *sales* триггер, запрещающий добавление в таблицу *customers* заказчика с фамилией Брудастый.

```
1 CREATE OR REPLACE FUNCTION delete_brud()
2 RETURNS trigger
3 AS $$
4 BEGIN
5 IF NEW.cust_name LIKE 'Брудастый%' THEN
6     RAISE EXCEPTION 'Заказчик не обслуживается';
7 END IF;
8 RETURN NEW;
9 END;
10 $$ LANGUAGE plpgsql;
```

```
1 CREATE TRIGGER brud
2 BEFORE INSERT ON customers
3 FOR EACH ROW EXECUTE PROCEDURE delete_brud();
```

```
1 INSERT INTO customers VALUES (13, 'Брудастый Орест Сергеевич', 'Вильно', 0);
```

Data Output Messages Notifications

ERROR: ОШИБКА: Заказчик не обслуживается
CONTEXT: функция PL/pgSQL delete_brud(), строка 4, оператор RAISE

После этого удалим триггер:

```
1 DROP TRIGGER brud ON customers;
```

Data Output Messages Notifications

DROP TRIGGER

ЗАКЛЮЧЕНИЕ

Предлагаемый в учебном пособии лабораторный практикум может стать первым шагом в освоении PostgreSQL – одной из наиболее популярных в настоящее время систем управления реляционными базами данных. Сегодня PostgreSQL выросла в полноценную СУБД уровня предприятия и составляет реальную альтернативу коммерческим базам данных.

Во многом это достигнуто за счет соответствия PostgreSQL стандартам языка SQL. PostgreSQL обеспечивает поддержку развивающегося стандарта ANSI SQL. Причем это относится ко всем версиям стандарта – от SQL-92 до последней SQL:2016, стандартизировавшей поддержку работы с форматом JSON.

В целом PostgreSQL обеспечивает высокий уровень соответствия стандарту и поддерживает большинство обязательных возможностей, а также большое количество необязательных.

Рамки учебного пособия ограничены рассмотрением функций системы оперативной обработки транзакций OLTP (On-Line Transactional Processing), основная задача которых – ввод, обновление и удаление данных в реляционную базу данных.

Опираясь на полученные базовые знания, можно перейти к самостоятельному изучению тем, связанных со следующими вопросами [1, 7, 9, 10, 11]:

- использование преимуществ расширяемости системы (добавление пользовательских типов данных, функций и операторов для работы с новыми типами, языков серверного программирования, подключение к внешним источникам данных, применение загружаемых расширений);
- практическое использование различных способов индексирования, реализованных в PostgreSQL;
- оптимизация пользовательских запросов и использование планировщика запросов, позволяющего оптимизировать самые сложные запросы;
- многопользовательская работа и сериализация транзакций;

- обеспечение надежности и устойчивости при работе с критически важными данными (настройка горячего резервирования, восстановление на заданный момент времени, репликация данных);
- безопасность (подключение по защищенному SSL-соединению, парольная аутентификация, использование клиентских сертификатов, аутентификация с помощью внешних сервисов);
- управление пользователями и доступом к объектам базы данных (создание и управление пользователями и групповыми ролями, детальное управление доступом на уровне отдельных столбцов и строк, мандатное управление доступом).

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Домбровская, Г. Оптимизация запросов в PostgreSQL: полное руководство по созданию эффективных запросов / Г. Домбровская, Б. Новиков, А. Бейликова. – Москва : ДМК Пресс, 2022. – 277 с. – ISBN: 978-5-97060-963-7.
2. Кригель, А. SQL. Библия пользователя / А. Кригель, Б. Трухнов. – 2-е изд. – Москва : Вильямс, 2010. – 744 с. – ISBN 978-5-8459-1546-7.
3. Лузанов, П. В. Postgres. Первое знакомство / П. В. Лузанов, Е. В. Рогов, И. В. Лёвшин. – 6-е изд., перераб. и доп. – Москва : Постгрес Профессиональный, 2020. – 178 с. – ISBN 978-5-6045970-1-9.
4. Малков, О. Б. Работа с Transact-SQL : учеб. пособие / О. Б. Малков, М. В. Девятерикова ; Ом. гос. техн. ун-т. – Омск : Изд-во ОмГТУ, 2015. – 1 CD-ROM (2,4 Мб). – Загл. с этикетки диска. – ISBN 978-5-8149-1923-6.
5. Малков, О. Б. Oracle SQL. Базовая часть : учеб. пособие / О. Б. Малков ; Ом. гос. техн. ун-т. – Омск : Изд-во ОмГТУ, 2019. – 1 CD-ROM (1,02 Мб). – Загл. с этикетки диска. – ISBN 978-5-8149-2849-8.
6. Моргунов, Е. П. PostgreSQL. Основы языка SQL : учеб. пособие / Е. П. Моргунов. – Санкт-Петербург : БХВ-Петербург, 2018. – 335 с. – ISBN 978-5-9775-4022-3.
7. Новиков, Б. А. Основы технологий баз данных : учеб. пособие / Б. А. Новиков, Е. А. Горшкова, Н. Г. Графеева. – 2-е изд. – Москва : ДМК Пресс, 2020. – 582 с. – ISBN 978-5-97060-841-8.
8. Редмонд, Э. Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQ / Э. Редмонд, Д. Р. Уилсон ; под ред. Ж. Картер. – Москва : ДМК Пресс, 2013. – 384 с. – ISBN 978-5-94074-866-3.
9. Рогов, Е. В. PostgreSQL 15 изнутри / Е. В. Рогов. – Москва : ДМК Пресс, 2023. – 662 с. – ISBN 978-5-93700-178-8.
10. PostgreSQL Server Programming / U. Dar, H. Krosing, J. Mlodgenski, K. Roybal. – 2nd Edition Extend. – Birmingham : Packt Publishing, 2015. – 508 p.

11. PostgreSQL. Documentation. 9th February 2023: PostgreSQL 15.2, 14.7, 13.10, 12.14, and 11.19 Released. – URL: <https://www.postgresql.org/docs/> (дата обращения: 30.03.2023).
12. PostgresPro. Документация к PostgreSQL 15.2. – URL: <https://postgrespro.ru/docs/postgresql/15/> (дата обращения: 30.03.2023).
13. pgAdmin 4. Documentation. Release 7.1. The pgAdmin Development Team May 03, 2023. – URL: <https://www.pgadmin.org/docs/pgadmin4/latest/index.html> (дата обращения: 10.05.2023).

Типы данных в PostgreSQL

PostgreSQL предоставляет пользователям богатый ассортимент встроенных типов данных. Кроме того, пользователи могут создавать свои типы, используя команду *CREATE TYPE*. PostgreSQL поддерживает следующие типы данных:

- числовые;
- символьные;
- дата и время;
- двоичные строки;
- пространственные;
- другие (не входящие в перечисленные категории).

Числовые типы используются для хранения чисел. Различаются:

- точные числовые типы;
- приближенные числовые типы.

К точным числовым типам (табл. П.1) относят логический тип *BIT*, целый тип *INT* и его вариации, а также денежный тип *MONEY* и его вариации.

Таблица П.1

Тип данных	Размер (байт)	Диапазон значений
<i>SMALLINT</i>	2	От -32 768 до 32 767 (от -2^{15} до $2^{15}-1$)
<i>INTEGER</i>	4	От -2 147 683 648 до 2 147 683 647 (от -2^{31} до $2^{31}-1$)
<i>BIGINT</i>	8	От -2^{63} до $2^{63}-1$
<i>NUMERIC [(p[, s])]</i> <i>DECIMAL [(p[, s])]</i>	перемен-ный	p – точность (полное количество десятичных цифр до и после запятой), s – масштаб (количество цифр после запятой). Диапазон от $-10^{38}+1$ до $10^{38}-1$

Типы *DECIMAL* и *NUMERIC* равнозначны. *NUMERIC* рекомендуется для хранения денежных сумм и других величин, где важна точность. Вычисления с типом *NUMERIC* дают точные результаты, но выполняются медленнее, чем с целыми числами или с типами с плавающей запятой. Если точность и масштаб не указаны, то будет создан столбец «неограниченное число», в котором можно сохранять числовые значения любой длины до предела, обусловленного реализацией. В столбце этого типа входные значения не будут приводиться к какому-либо масштабу.

Числовые значения физически хранятся без дополняющих нулей слева или справа. Таким образом, объявляемые точность и масштаб столбца определяют максимальный, а не фиксированный размер хранения.

Тип *NUMERIC* включает специальные значения: *Infinity*, *-Infinity*, *NAN*. Это особые значения – «бесконечность», «минус бесконечность» и «не число». Записывая эти значения в виде констант в команде SQL, их нужно заключать в апострофы: *UPDATE table SET x = '-Infinity'*. Регистр символов в этих строках неважен. Значения бесконечности могут быть записаны как *inf* и *-inf*. Бесконечность может быть сохранена только в столбце типа «неограниченный *NUMERIC*».

Значение *NAN* (не число) используют для представления неопределенных результатов вычислений. В большинстве реализаций *NAN* считается не равным любому другому значению (и самому *NAN*). Чтобы значения *NUMERIC* можно было сортировать и использовать в древовидных индексах, PostgreSQL считает, что значения *NAN* равны друг другу и при этом больше любых числовых значений.

К приближенным числовым типам (табл. П.2) относятся:

- *DOUBLE PRECISION* – представление чисел с плавающей запятой с двойной точностью;
- *REAL* – представление чисел с плавающей запятой с обычной точностью.

Тип данных	Размер (байт)	Диапазон значений
<i>DOUBLE PRECISION</i>	8	От 1E-307 до 1E+308 с точностью не меньше 15 десятичных цифр
<i>REAL</i>	4	От 1E-37 до 1E+37 с точностью не меньше 6 десятичных цифр

Типы с плавающей точкой *REAL* и *DOUBLE PRECISION* хранят приближенные числовые значения с переменной точностью (полученное значение может несколько отличаться от записанного). Отметим следующее:

- если нужна точность при хранении и вычислениях (например, для денежных сумм), используйте вместо них тип *NUMERIC*;
- если нужно выполнять с этими типами сложные вычисления, имеющие большую важность, тщательно изучите реализацию операций в вашей среде и поведение в крайних случаях (бесконечность, антипереполнение);
- проверка равенства двух чисел с плавающей точкой может не всегда давать ожидаемый результат.

Тип *REAL* – значения от 1E-37 до 1E+37 с точностью не меньше 6 десятичных цифр. Тип *DOUBLE PRECISION* – значения в диапазоне от 1E-307 до 1E+308 с точностью не меньше 15 десятичных цифр. Попытка сохранить слишком большие или слишком маленькие значения приведет к ошибке. Если точность вводимого числа слишком велика, оно будет округлено. При попытке сохранить число, близкое к 0, но непредставимое как отличное от 0, возникнет ошибка антипереполнения.

В дополнение к обычным числовым значениям типы с плавающей точкой принимают специальные значения: *Infinity*, *-Infinity*, *Nan*, представляющие особые значения – «бесконечность», «минус бесконечность» и «не число».

К последовательным числовым типам (табл. П.3) относятся:

- *SMALLSERIAL* – небольшое целое с автоувеличением;
- *SERIAL* – целое с автоувеличением;
- *BIGSERIAL* – большое целое с автоувеличением.

Таблица П.3

Тип данных	Размер (байт)	Диапазон значений
<i>SMALLSERIAL</i>	2	1 ... 32767
<i>SERIAL</i>	4	1 ... 2147483647
<i>BIGSERIAL</i>	8	1 ... 9223372036854775807

Типы данных *SMALLSERIAL*, *SERIAL* и *BIGSERIAL* не настоящие типы, а удобное средство создания столбцов с уникальными идентификаторами (как *AUTO_INCREMENT* в некоторых СУБД). При определении такого типа создается целочисленный столбец со значением по умолчанию, извлекаемым из генератора последовательности. Так как эти типы реализованы через последовательности, в числовом ряду значений столбца могут образовываться пропуски («дыры»), даже если строки не удалялись. Значение, выделенное из последовательности, считается задействованным, даже если строку с этим значением не удалось вставить в таблицу. Это может произойти при откате транзакции, добавляющей данные.

Чтобы вставить в столбец *SERIAL* следующее значение последовательности, ему нужно присвоить значение по умолчанию. Это можно сделать с помощью ключевого слова *DEFAULT*.

Последовательность, созданная для столбца *SERIAL*, автоматически удаляется при удалении связанного столбца. Последовательность можно удалить и отдельно от столбца, но при этом также будет удалено определение значения по умолчанию.

К денежным типам (табл. П.4) относится:

- *MONEY* – хранение денежной суммы с фиксированной дробной частью.

Таблица П.4

Тип данных	Размер (байт)	Диапазон значений
<i>MONEY</i>	8	-92233720368547758.08 ... +92233720368547758.07

Предполагается, что число содержит два знака после запятой. Входные данные могут быть записаны по-разному, в том числе в виде целых и дробных чисел, а также в виде строки в денежном формате '\$1,000.00'. Выводятся эти значения в денежном формате, зависящем от региональных стандартов.

Символьные типы общего назначения, доступные в PostgreSQL, перечислены в табл. П.5.

Таблица П.5

Тип данных	Описание
<i>CHARACTER(n)</i> <i>CHAR(N)</i>	Строка фиксированной длины до n символов (не байт). Если n задано, оно должно быть больше нуля и меньше или равно 10485760. Записи <i>character</i> без указания длины соответствует <i>character(1)</i> . Если помещаемая строка короче n, она дополняется справа пробелами.
<i>CHARACTER VARYING(n),</i> <i>VARCHAR(n)</i>	Строка ограниченной переменной длины до n символов (не байт). Если n задано, оно должно быть больше нуля и меньше или равно 10485760. Если помещаемая строка короче n, то она не дополняется пробелами. Если же длина не указана, этот тип будет принимать строки любого размера.
<i>TEXT</i>	Строка неограниченной переменной длины. Максимально возможный размер строки – около 1 ГБ.

Если необходимо сохранять строки без определенного предела длины, используйте типы *TEXT* или *CHARACTER VARYING* без указания длины, а не задавайте какое-либо большое максимальное значение.

По быстродействию эти три типа почти не отличаются, но на практике *CHARACTER(N)* обычно оказывается медленнее других. Поэтому в большинстве случаев вместо него лучше применять *TEXT* или *CHARACTER VARYING*.

К типам даты и времени (табл. П.6) относятся:

- *TIMESTAMP* – дата и время (без часового пояса);
- *TIMESTAMP WITH TIME ZONE* – дата и время (с часовым поясом);
- *DATE* – дата (без времени суток);
- *TIME* – время суток (без даты);
- *TIME WITH TIME ZONE* – время дня (без даты) с часовым поясом;
- *INTERVAL* – временной интервал.

Таблица П.6

Тип данных	Размер (байт)	Диапазон значений	Точность
<i>TIMESTAMP [(p)] [WITHOUT TIME ZONE]</i>	8	Даты от 4713 до н. э. до 294276 н. э.	1 микросекунда
<i>TIMESTAMP [(p)] WITH TIME ZONE</i>	8	Даты от 4713 до н. э. до 294276 н. э.	1 микросекунда
<i>DATE</i>	4	4713 до н. э. 5874897 н. э.	1 день
<i>TIME [(p)] [WITHOUT TIME ZONE]</i>	8	Время от 00:00:00 до 24:00:00.	1 микросекунда
<i>TIME [(p)] WITH TIME ZONE</i>	12	Время от 00:00:00+1559 до 24:00:00-1559	1 микросекунда
<i>INTERVAL [поля] [(p)]</i>	16	От -178000000 лет до 178000000 лет	1 микросекунда

Даты считаются по Григорианскому календарю.

Типы *TIME*, *TIMESTAMP* и *INTERVAL* принимают необязательное значение точности *p*, определяющее, сколько знаков после запятой должно сохраняться в секундах. По умолчанию точность не ограничивается. Допустимые значения *p* лежат в интервале от 0 до 6.

Тип *INTERVAL* ограничивает набор сохраняемых полей фразами: *YEAR*, *MONTH*, *DAY*, *HOUR*, *MINUTE*, *SECOND*, *YEAR TO MONTH*, *DAY TO HOUR*, *DAY TO MINUTE*, *DAY TO SECOND*, *HOUR TO MINUTE*, *HOUR TO SECOND*, *MINUTE TO SECOND*. Если указаны и поля, и точность *p*, указание поля должно включать *SECOND*, так как точность применима только к секундам.

Тип *TIME WITH TIME ZONE* определен стандартом SQL, но ценность его сомнительна. В большинстве случаев сочетание типов *DATE*, *TIME*, *TIMESTAMP WITHOUT TIME ZONE* и *TIMESTAMP WITH TIME ZONE* удовлетворяет все потребности в функционале дат/времени.

Значения даты и времени принимаются в любом разумном формате, включая ISO 8601, SQL-совместимый, традиционный формат POSTGRES и т. д.

В некоторых форматах порядок даты, месяца и года во вводимой дате неоднозначен. Для явного определения формата предназначен параметр *DateStyle*. Когда он имеет значение *MDY*, выбирается интерпретация месяц-день-год, значению *DMY* соответствует день-месяц-год, а *YMD* – год-месяц-день.

Вводимые значения даты и времени нужно заключать в апострофы, как текстовые строки. В табл. П.7 приведены допустимые значения типа *DATE*.

Таблица П.7

Пример	Описание
1999-01-08	ISO 8601; 8 января в любом режиме (рекомендуемый формат)
January 8, 1999	Воспринимается однозначно в любом режиме datestyle
1/8/1999	8 января в режиме MDY и 1 августа в режиме DMY
1/18/1999	18 января в режиме MDY; недопустимая дата в других режимах
01/02/03	2 января 2003 г. в режиме MDY; 1 февраля 2003 г. в режиме DMY и 3 февраля 2001 г. в режиме YMD
1999-Jan-08	8 января в любом режиме

Окончание табл. П.7

Пример	Описание
Jan-08-1999	8 января в любом режиме
08-Jan-1999	8 января в любом режиме
99-Jan-08	8 января в режиме YMD; ошибка в других режимах
08-Jan-99	8 января; ошибка в режиме YMD
Jan-08-99	8 января; ошибка в режиме YMD
19990108	ISO 8601; 8 января 1999 в любом режиме
990108	ISO 8601; 8 января 1999 в любом режиме
1999.008	год и день года
J2451187	юлианский день
January 8, 99 BC	99 до н. э.

Допустимые значения времени суток без даты состоят из записи времени суток и необязательного указания часового пояса (табл. П.8).

Таблица П.8

Пример	Описание
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	то же, что и 04:05; AM не меняет значение времени
04:05 PM	то же, что и 16:05; часы должны быть <= 12

Допустимые значения типов *TIMESTAMP* состоят из записи даты и времени, после которого может указываться часовой пояс и необязательное уточнение AD или BC (до н. э. или н. э. соответственно).

Таким образом: '*1999-01-08 04:05:06*' и '*1999-01-08 04:05:06 -8:00*' – допустимые варианты, соответствующие стандарту ISO 8601. В дополнение к этому поддерживается распространенный формат:

'January 8 04:05:06 1999 PST'

PostgreSQL поддерживает также несколько специальных значений даты/времени (табл. П.9).

Таблица П.9

Вводимая строка	Допустимые типы	Описание
<i>EPOCH</i>	<i>DATE, TIMESTAMP</i>	1970-01-01 00:00:00+00 (точка отсчета времени в Unix)
<i>INFINITY</i>	<i>DATE, TIMESTAMP</i>	время после максимальной допустимой даты
<i>-INFINITY</i>	<i>DATE, TIMESTAMP</i>	время до минимальной допустимой даты
<i>NOW</i>	<i>DATE, TIME, TIMESTAMP</i>	время начала текущей транзакции
<i>TODAY</i>	<i>DATE, TIMESTAMP</i>	время начала текущих суток (00:00)
<i>TOMORROW</i>	<i>DATE, TIMESTAMP</i>	время начала следующих суток (00:00)
<i>YESTERDAY</i>	<i>DATE, TIMESTAMP</i>	время начала предыдущих суток (00:00)
<i>ALLBALLS</i>	<i>TIME</i>	00:00:00.00 UTC

Значения *INFINITY* и *-INFINITY* отображаются в том же виде, тогда как другие варианты при чтении преобразуются в значения даты/времени. В частности, *NOW* и подобные строки преобразуются в актуальные значения времени в момент чтения. Чтобы использовать эти значения в качестве констант в командах SQL, их нужно заключать в апострофы.

Для получения текущей даты/времени соответствующего типа можно использовать следующие SQL-совместимые функции: *CURRENT_DATE*, *CURRENT_TIME*, *CURRENT_TIMESTAMP*, *LOCALTIME* и *LOCALTIMESTAMP*. Во входных строках эти SQL-функции не распознаются.

Входные значения *NOW*, *TODAY*, *TOMORROW* и *YESTERDAY* корректно работают в интерактивных SQL-командах, но когда команды сохраняются для последующего выполнения (в подготовленных операторах, представлениях или определениях функций), их поведение может быть неожиданным. Такая строка может преобразоваться в конкретное значение времени, которое затем будет использоваться гораздо позже момента, когда оно было получено. В таких случаях следует использовать одну из SQL-функций. Например, *CURRENT_DATE + 1* будет работать надежнее, чем '*tomorrow*'::*date*.

В качестве выходного формата типов даты/времени можно использовать один из четырех стилей: ISO 8601, SQL (Ingres), традиционный формат POSTGRES (формат *date* в Unix) или German (табл. П.10). По умолчанию – формат ISO 8601.

Таблица П.10

Стиль	Описание	Пример
<i>ISO</i>	ISO 8601, стандарт SQL	<i>1997-12-17 07:37:16-08</i>
<i>SQL</i>	традиционный стиль	<i>12/17/1997 07:37:16.00 PST</i>
<i>Postgres</i>	изначальный стиль	<i>Wed Dec 17 07:37:16 1997 PST</i>
<i>German</i>	региональный стиль	<i>17.12.1997 07:37:16.00 PST</i>

ISO 8601 указывает, что дата должна отделяться от времени буквой T в верхнем регистре. PostgreSQL принимает этот формат при вводе, но при выводе вставляет вместо T пробел. Это сделано для улучшения читаемости и совместимости.

В стилях *SQL* и *POSTGRES* день выводится перед месяцем, если установлен порядок DMY, а в противном случае месяц выводится перед днем.

В табл. П.11 показано несколько примеров допустимых вводимых значений типа *INTERVAL*.

Таблица П.11

Пример	Описание
1-2	Стандартный формат SQL: 1 год и 2 месяца
3 4:05:06	Стандартный формат SQL: 3 дня 4 часа 5 минут 6 секунд
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	Традиционный формат Postgres: 1 год 2 месяца 3 дня 4 часа 5 минут 6 секунд
P1Y2M3DT4H5M6S	«Формат с кодами» ISO 8601: то же значение, что и выше
P0001-02-03T04:05:06	«Альтернативный формат» ISO 8601: то же значение, что и выше

В табл. П.12 показано несколько стилей вывода интервалов. Выбрать нужный стиль позволяет команда *SET intervalstyle* (по умолчанию выбран *postgres*).

Таблица П.12

Стиль	Интервал год-месяц	Интервал день-время	Смешанный интервал
<i>sql_standard</i>	1-2	3 4:05:06	-1-2 +3 -4:05:06
<i>postgres</i>	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days - 04:05:06
<i>postgres_verbose</i>	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
<i>iso_8601</i>	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

В PostgreSQL есть стандартный логический тип *BOOLEAN* (табл. П.13). Тип *BOOLEAN* может иметь следующие состояния: *true*, *false* и *unknown*, которое представляется SQL-значением *NULL*.

Таблица П.13

Имя	Размер	Описание
<i>BOOLEAN</i>	1 байт	состояние: истина или ложь

Логические константы могут представляться в запросах ключевыми словами *TRUE*, *FALSE* и *NULL*. Функция ввода данных типа *BOOLEAN* воспринимает строковые представления «*true*»: *true*, *yes*, *on*, *1* и представления «*false*»: *false*, *no*, *off*, *0*.

Также воспринимаются уникальные префиксы этих строк (*t* или *n*). Регистр символов не имеет значения, пробелы в начале и в конце строки игнорируются.

Ключевые слова *TRUE* и *FALSE* являются предпочтительными (соответствуют стандарту SQL) для записи логических констант в SQL-запросах. Но можно использовать строковые представления, которые допускает синтаксис строковых констант, например, '*yes*::*boolean*'.

При анализе запроса *TRUE* и *FALSE* автоматически считаются значениями типа *BOOLEAN*, но для *NULL* это не так (ему может соответствовать любой тип). Поэтому может потребоваться привести *NULL* к типу *BOOLEAN* явно: *NULL*::*boolean*.

Кроме указанных, в PostgreSQL имеются: двоичные типы данных, типы перечислений, геометрические типы, типы, описывающие сетевые адреса, битовые строки, типы, предназначенные для текстового поиска, тип *UUID*, тип *XML*, типы *JSON*, массивы, составные типы, диапазонные типы, типы доменов, идентификаторы объектов, тип *pg_lsn*, псевдотипы. Информацию о них можно получить в [11].