Министерство науки и высшего образования РФ
ФГБОУ ВО «Ульяновский государственный университет»
Факультет математики, информационных и авиационных технологий

Кафедра телекоммуникационных технологий и сетей

Липатова Светлана Валерьевна

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ
для семинарских (практических) занятий, лабораторного практикума
и самостоятельной работы
по дисциплине

«Методы защиты базы данных»

для студентов направления
для студентов факультета математики, информационных и авиационных технологий

Ульяновск
2022

Методические рекомендации для семинарских (практических) занятий, лабораторного практикума и самостоятельной работы по дисциплине «Методы защиты баз данных» / составитель: С.В. Липатова - Ульяновск: УлГУ, 2022 – 89 с.

Настоящие методические рекомендации предназначены для студентов для студентов факультета математики, информационных и авиационных технологий. В работе приведены литература по дисциплине, темы дисциплины и вопросы в рамках каждой темы, рекомендации по изучению теоретического материала, контрольные вопросы для самоконтроля, задания для самостоятельной работы, задачи и упражнения для самостоятельной подготовки к семинарам или полностью самостоятельного освоения практических навыков, задания для лабораторного практикума и рекомендации по их выполнению.

Студентам всех форм обучения следует использовать данные методические рекомендации при подготовке к семинарам, самостоятельной подготовке, а также промежуточной аттестации по дисциплине «Базы данных».

Рекомендованы к введению в образовательный процесс

Учёным советом факультета математики, информационных и авиационных технологий УлГУ

протокол № 3/**22** от «19» апреля 2022 г.

СОДЕРЖАНИЕ

ОБЩИЕ ВОПРОСЫ

В результате изучения дисциплины «Методы защиты баз данных» студенты должны изучить:

- модели структур данных;
- способы классификации СУБД в зависимости от реализуемых моделей данных и способов их использования;
- способы хранения данных на физическом уровне, типы и способы организации файловых систем;
- способы реляционной модели данных и СУБД, реализующих эту модель, языка запросов SQL;
- проблемы и основные способы их решения при коллективном доступе к данным;
- возможности СУБД, поддерживающих различные модели организации данных, преимущества и недостатки этих СУБД при реализации различных структур данных, средствами этих СУБД;
- этапы жизненного цикла базы данных, поддержки и сопровождения;
- специализированные аппаратные и программные средства ориентированных на построение баз данных больших объёмов хранения.

Методические рекомендации для семинарских (практических) занятий, лабораторного практикума и самостоятельной работы по дисциплине «Методы защиты баз данных» направлены на повышение эффективности освоения знаний, умений, навыков и компетенций, связанных с:

- проектированием реляционных баз данных,
- программированием на языке SQL,
- способами хранения и обработки разнородных данных и т.д..

Методические рекомендации предлагают указания по всем темам дисциплины «Базы данных». Методические рекомендации разбиты по темам и содержат набор вопросов для систематизации теоретического материала, полученного на лекционных занятиях, и самостоятельного изучения теории, вопросы (тесты) для текущего контроля на практических занятиях (семинарах), задачи для усвоения практических навыков. Для лабораторного практикума приведены задания, варианты и рекомендации по выполнению лабораторных работ.

Список литературы и информационного обеспечения, приведённый в конце методических указаний, может служить основой для изучения всех рассматриваемых тем.

Дополнительная и учебно-методическая литература могут быть использованы обучающимися для закрепления изучаемого материала.

РЕКОМЕНДАЦИИ ПО ОТДЕЛЬНЫМ ТЕМАМ ДИСЦИПЛИНЫ

Тема 1. Данные и базы данных. Эволюция концепций баз данных.

Основные вопросы темы

1. Определение данных и информации.
2. Классификация наборов данных, баз данных.

Рекомендации по изучению темы

Вопрос 1 изложен в учебнике [1] на с. 14-39.

Вопрос 1 изложен в учебнике [1] на с. 39-43.

Вопросы для самоподготовки

Рекомендуется после изучения материалов лекций и специальной литературы подготовить ответы на вопросы:

1. Какие модели представления данных относят к первым (ранним)?
2. Как можно классифицировать наборы данных?
3. Какая СУБД считается первой?
4. Какой первый стандарт был принят по моделям представления данных?
5. Что подразумевает понятие Big Data?
6. Что подразумевает понятие noSQL?
7. Когда появилась реляционная модель представления данных?

Контрольные тесты

1. Первая реализация SQL была в:

Выберите один ответ:

- a. CODASYL (Conference of Data System Languages)
- b. нет правильного ответа
- c. СУБД Ingres
- d. СУБД System R

2. СУБД IBM System R является:

Выберите один ответ:

- a. иерархической
- b. многомерной

c. серевой

d. реляционной

3. Из какого проекта развивалась СУБД PostgreSQL?

Выберите один ответ:

a. System R

b. Postgres95

c. POSTGRES

d. UniSQL

4. Создали машины, работающие с перфокартами:

Выберите один или несколько ответов:

a. Кодд

b. Лум

c. Жаккар

d. Стоунбрейкер

5. К какому поколению относят реляционные базы данных?

Выберите один ответ:

a. 3

b. 4

c. 1

d. 2

e. 5

6. Автоматизированно обрабатываемые данных хранились ранее или могут хранится сейчас на:

Выберите один или несколько ответов:

a. Жесткий диск

b. Перфокарта

c. Магнитная лента

d. Картотека

☐ е. Оперативная память

Тема 2. Системы управления базами данных. СУБД PostgreSQL

Основные вопросы темы

1. Функции и структура СУБД.
2. Управление данными, управление транзакциями, журнализация изменений базы данных, восстановление после сбоев.
3. Особенности СУБД PostgreSQL.

Рекомендации по изучению темы

Вопросы 1,2 изложены в учебнике [1] на с. 39-43.

Вопросы 3 изложен в учебнике [5] в главе 2.

Вопросы для самоподготовки

Рекомендуется после изучения материалов лекций и специальной литературы подготовить ответы на вопросы:

1. Какие основные пять функций СУБД?
2. Чем отличается мягкий сбой от жесткого и какие действия по восстановлению данных после них требуется предпринять?
3. Какие языки поддерживают СУБД?
4. Как выполняется запись в журнал СУБД?
5. Какую модель представления данных поддерживает PostgreSQL?
6. По каким параметрам можно классифицировать СУБД?
7. Какие утилиты требуются для работы с PostgreSQL?

Контрольные тесты

1. **PostgreSQL является специализированной СУБД (верно или нет)**

Выберите один ответ:

○ Верно

○ Неверно

2. **После восстановления после жесткого сбоя достаточно журнала изменений базы данных.**

Выберите один ответ:

○ Верно

Неверно

3. **Одна БД может находиться под управлением нескольких СУБД одновременно.**

Выберите один ответ:

Верно

Неверно

4. **Какие СУБД обладают свойствами объектно-реляционных СУБД?**

Выберите один или несколько ответов:

a. Postgre95

b. POSTGRES

c. Oracle Database

d. UniSQL

e. Microsoft SQL Server

f. PostgreSQL

g. MySQL

h. СУБД ЛИНТЕР

i. Informix Universal Server

j. SQLite

5. **К жесткому сбою могут привести следующие проблемы:**

Выберите один ответ:

a. выход из стоя частично или полностью оперативной памяти

b. неисправность жесткого диска

c. ошибка операционной системы

6. **Система управления базами данных — это**

Выберите один ответ:

a. прикладная программа для обработки текстов и различных документов

b. оболочка операционной системы, позволяющая более комфортно работать с файлами

c. набор программ, обеспечивающий работу всех аппаратных устройств компьютера и доступ пользователя к ним

d. программная система, поддерживающая наполнение и манипулирование данными в файлах баз данных

7. Функциями СУБД являются:

Выберите один или несколько ответов:

a. поддержка различных диалектов SQL

b. управление данными во внешней памяти

c. журнализация

d. управление файлами

e. управление буферами оперативной памяти

f. распределение вычислительных ресурсов

g. управление дисками

h. поддержка языков БД

i. управление взаимодействием одновременно работающих задач

j. компиляция

k. авторизация доступа к объектам

8. Язык для работы с СУБД должен обеспечивать:

Выберите один или несколько ответов:

a. возможность описания объектов базы данных (схемы)

b. возможность управления файлами

c. возможность манипулирования данными

d. возможность построения пользовательских отчетов

9. Одна БД может находиться под управлением нескольких СУБД одновременно.

Выберите один ответ:

Верно

Неверно

Тема 3. Организация данных

Основные вопросы темы

1. Базы данных и схемы. Системный каталог.
2. Табличные пространства. Низкий уровень.

Рекомендации по изучению темы

Вопрос 1 изложен в учебнике [4] на с. 5-31.

Вопрос 2 изложен в учебнике [6] на с. 10-27.

Вопросы для самоподготовки

Рекомендуется после изучения материалов лекций и специальной литературы подготовить ответы на вопросы:

1) Что такое схема и для чего она используется?
2) Кто такое кластер?
3) Как связаны кластер, база данных и схема?
4) Какие специальные схемы существуют в PostgreSQL?
5) Что такое системный каталог и как к нему обращаться?
6) Какие объекты находятся в системном каталоге?
7) Что такое табличное пространство?

Тема 4. Задачи администрирования

Основные вопросы темы

1. Мониторинг. Сопровождение. Управление доступом.
2. Политика защиты строк. Подключение и аутентификация.

Рекомендации по изучению темы

Вопрос 1 изложен в учебнике [7] на с. 21-39.

Справочные материалы по 2 вопросу приведен в соответствующей теме раздела «Лабораторный практикум» данного пособия.

Вопросы для самоподготовки

Рекомендуется после изучения материалов лекций и специальной литературы подготовить ответы на вопросы:

1) Какая статистика собирается внутри базы данных?
2) Как организован процесс сбора статистики?

3) Какую дополнительную информацию собирают расширения?
4) Какая информация хранится в журнале сообщений?
5) Какие схемы ротации журналов можно использовать?
6) Какие есть системы внешнего мониторинга для PostgreSQL?
7) Как осуществляют мониторинг индексов?

Тема 5. Роли, привилегии и операторы для работы с ними

Основные вопросы темы

1. Понятие роли, связь роли с понятиями пользователь, группа пользователей, схема базы данных. Предопределенные роли.
2. Операторы ведения ролей. Привилегии и операторы по назначению и отмене привилегий. Виды привилегий.

Рекомендации по изучению темы

Справочные материалы по операторам языка SQL для СУБД PosgreSQL приведены в соответствующей теме раздела «Лабораторный практикум» данного пособия.

Вопросы для самоподготовки

Рекомендуется после изучения материалов лекций и специальной литературы подготовить ответы на вопросы:

1. Чем роли отличаются от пользователей?
2. Как привилегии существуют?
3. Как назначить привилегию?
4. Как назначить одну роль другой роли?
5. Как отменить привилегию?
6. Что такое предопределенная роль и приведите ее примеры?
7. Одинаковы ли привилегии для таблиц и функций, приведите примеры?

Контрольные тесты

1. **Если подключился под определенной ролью, то не можешь ее сменить до конца сеанса.**

Выберите один ответ:

- Верно
- Неверно

2. **Для того чтобы исключить роль из другой роли (группы) надо использовать оператор:**

Выберите один ответ:

- a. DROP ROLE
- b. GRANT
- c. REVOKE
- d. DELETE ROLE
- e. CREATE ROLE

3. **Какие привилегии можно назначить пользовательской таблице?**

Выберите один или несколько ответов:

- a. TEMP
- b. EXECUTE
- c. TRUNCATE
- d. SELECT
- e. TRIGGER
- f. USAGE
- g. CREATE
- h. UPDATE
- i. CONNECT
- j. TEMPORARY
- k. REFERENCES
- l. ALL PRIVILEGES
- m. DELETE
- n. INSERT

4. **Какие из утверждений верные?**

Выберите один или несколько ответов:

- a. В других СУБД (например ORACLE, MS SQL Server) это разные операторы, понятия роли и пользователя там различны.

☐ b. Операторы CREATE ROLE, CREATE USER являются синонимами в PostgreSQL, а CREATE GROUP нет.

☐ c. Операторы CREATE ROLE, CREATE GROUP являются синонимами в PostgreSQL, а CREATE USER нет.

☐ d. Все три оператора создают разные объекты базы данных.

☐ e. Операторы CREATE ROLE, CREATE USER, CREATE GROUP являются синонимами в PostgreSQL.

5. **Выберете оператор который делает активным идентификатор пользователя сеанса:**

Выберите один или несколько ответов:

☐ a. SET ROLE user1

☐ b. RESET SESSION AUTHORIZATION

☐ c. SET ROLE NONE

☐ d. RESET ROLE

☐ e. SET SESSION AUTHORIZATION user1

Тема 6. Многоверсионность

Основные вопросы темы

1. Изоляция. Страницы и версии строк. Снимки данных. HOT-обновления.
2. Очистка. Автоочистка. Заморозка.

Рекомендации по изучению темы

Справочные материалы по вопросам приведён в соответствующей теме раздела «Лабораторный практикум» данного пособия.

Вопросы для самоподготовки

Рекомендуется после изучения материалов лекций и специальной литературы подготовить ответы на вопросы:

1) Какова структура страниц?
2) Что из себя представляет снимок данных?
3) Как используется видимость версий строк при формировании снимка?
4) Чем обычная очистка отличается от полной?

5) В чем заключается заморозка версий строк?
6) Как производится настройка автоочистки для выполнения заморозки?
7) Как выполнить заморозку вручную?

Тема 7. Журналирование

Основные вопросы темы

1. Буферный кэш. Журнал предзаписи.
2. Контрольная точка. Настройка журнала.

Рекомендации по изучению темы

Справочные материалы по вопросам приведён в соответствующей теме раздела «Лабораторный практикум» данного пособия.

Вопросы для самоподготовки

Рекомендуется после изучения материалов лекций и специальной литературы подготовить ответы на вопросы:

1) Как устроен буферный кэш?
2) Как работает механизм вытеснения страниц?
3) Логическое и физическое устройство журнала?
4) Как реализован процесс упреждающей записи и восстановление?
5) Как реализован процесс контрольной точки?
6) Как реализован процесс фоновой записи?
7) Уровни журнала?

Тема 8. Блокировки

Основные вопросы темы

1. Понятие транзакции.
2. Способы организации транзакций и принципы блокировки доступа к данным.

Рекомендации по изучению темы

Вопрос 1 изложен в [3] глава 13.

Вопрос 2 изложен в [1] с. 241-245.

Вопросы для самоподготовки

Рекомендуется после изучения материалов лекций и специальной литературы подготовить ответы на вопросы:

1. Какие 4 свойства у транзакций?
2. Что такое неявная транзакция?
3. Какой оператор позволяет откатить транзакцию?
4. Какие операторы должны входить в транзакцию, чтобы она успешно завершилась?
5. Как транзакция связана с многопользовательским доступом?
6. Какие режимы блокировок бывают?
7. Что такое взаимная блокировка транзакций?

Контрольные тесты

1. Если пользователь не использует BEGIN и COMMIT, то транзакции не создаются.

Выберите один ответ:

- Верно
- Неверно

2. Если произведен откат вложенной транзакции может ли быть выполнена основная транзакция?

Выберите один ответ:

- a. Нет
- b. Частично (операторы до вложенной транзакции)
- c. Да
- d. Частично (операторы после вложенной транзакции)

3. Выберете, что может быть заблокировано (в зависимости от уровня блокировки)

Выберите один или несколько ответов:

- a. функция
- b. страница
- c. таблица
- d. схема
- e. домен

f. строка

g. база данных

4. **ROLLBACK отменяет операторы транзакции после ее фиксации оператором COMMIT.**

Выберите один ответ:

Верно

Неверно

5. **Можно отменить часть транзакции, пока она не зафиксирована.**

Выберите один ответ:

Верно

Неверно

6. **Если пользователь использует оператор SELECT, то на соответствующие данные накладывается блокировка...**

Выберите один ответ:

a. обе

b. не накладывается

c. чтения

d. записи

Тема 9. Резервное копирование

Основные вопросы темы

1. Логическое резервирование. Базовая резервная копия.
2. Архив журнала предзаписи.

Рекомендации по изучению темы

Справочные материалы по вопросам приведён в соответствующей теме раздела «Лабораторный практикум» данного пособия.

Вопросы для самоподготовки

Рекомендуется после изучения материалов лекций и специальной литературы подготовить ответы на вопросы:

1) Что такое логическая резервная копия?
2) Как можно осуществлять копирование и восстановление отдельных таблиц?
3) Чем отличается копирование и восстановление баз данных от копирования и восстановления кластера?
4) Что такое физическая резервная копия?
5) Чем холодное резервирование отличается от горячего?
6) Отличия файлового архива от потокового?
7) Как реализовать восстановление с использованием архива?

Тема 10. Репликация

Основные вопросы темы

1. Физическая репликация. Переключение на реплику.
2. Логическая репликация. Сценарии использования.

Рекомендации по изучению темы

Справочные материалы по вопросам приведён в соответствующей теме раздела «Лабораторный практикум» данного пособия.

Вопросы для самоподготовки

Рекомендуется после изучения материалов лекций и специальной литературы подготовить ответы на вопросы:

1) Какие задачи репликации?
2) Какова схема работы физической репликации?
3) Какие есть способы доставки журнальных записей?
4) Чем отличается синхронная от асинхронной репликации?
5) Как осуществить переключение на реплику?
6) Отличия логической репликации от физической?
7) Возможна ли выборочная репликация отдельных таблиц?

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Порядок выполнения лабораторных работ может быть произвольным и определяется уровнем освоения компетенций обучающегося.

Тема 3. Организация данных

Задание: просмотрите предложенный код, демонстрирующий возможности PostgreSQL и повторите на своём сервере со своей базой данных, по следующим вопросам:

- Создание БД из шаблона
- Управление базами данных
- Размер базы данных
- Схемы
- Путь поиска
- Временные таблицы и pg_temp
- Удаление объектов
- Некоторые объекты системного каталога
- Использование команд psql
- Изучение структуры системного каталога
- OID и reg-типы
- Служебные табличные пространства
- Пользовательские табличные пространства
- Управление объектами в табличных пространствах
- Размер табличного пространства
- Удаление табличного пространства
- Расположение файлов
- Размер объектов и слоев
- TOAST

Отчет по лабораторной работе должен содержать**:**

1. Фамилию и номер группы учащегося, задание
2. Краткое описание базы данных
3. Результаты выполнения операторов
4. Код

Демонстрация (пример для повторения)

Базы данных

Посмотрим список имеющихся баз:

```
postgres$ psql -l
                                  List of databases
   Name     |  Owner   | Encoding |   Collate   |    Ctype    |   Access
privileges
-----------+----------+----------+-------------+-------------+--------------
--------
 postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres
+
           |          |          |             |             |
postgres=CTc/postgres
```

```
 template1 | postgres | UTF8       | en_US.UTF-8 | en_US.UTF-8 | =c/postgres
+
           |          |            |             |             |
postgres=CTc/postgres
(3 rows)
```

Пока нас интересует только название, значение остальных полей рассмотрим позже.

Также можно посмотреть в самой базе данных:

```
=>   SELECT   datname,   datistemplate,   datallowconn,   datconnlimit   FROM
pg_database;
  datname  | datistemplate | datallowconn | datconnlimit
-----------+---------------+--------------+--------------
 postgres  | f             | t            |           -1
 template1 | t             | t            |           -1
 template0 | t             | f            |           -1
(3 rows)
```

- datistemplate - является ли база данных шаблоном;
- datallowconn - разрешены ли соединения с базой данных,
- datconnlimit - максимальное количество соединений (-1 = без ограничений).

Создание БД из шаблона

Подключимся к шаблонной базе template1.

```
=> \c template1
You are now connected to database "template1" as user "postgres".
```

Проверим, доступна ли функция digest, вычисляющая хеш-код текстовой строки.

```
=> SELECT digest('Hello, world!', 'md5');
ERROR:  function digest(unknown, unknown) does not exist
LINE 1: SELECT digest('Hello, world!', 'md5');
               ^
HINT:  No function matches the given name and argument types. You might need
to add explicit type casts.
```

Такой функции нет.

На самом деле digest определена в пакете pgcrypto. Установим его:

```
=> CREATE EXTENSION pgcrypto;
CREATE EXTENSION
```

Если бы расширение pgcrypto не было установлено с помощью make install, мы получили бы ошибку "ERROR: could not open extension control file..."

Теперь нам доступны функции, входящие в расширение pgcrypto. Например, можно вычислить MD5-дайджест:

```
=> SELECT digest('Hello, world!', 'md5');
                digest
-----------------------------------
 \x6cd3556deb0da54bca060b4c39479839
(1 row)
```

Чтобы шаблон можно было использовать для создания базы, к нему не должно быть активных подключений, поэтому отключимся от базы template1.

```
=> \c postgres
You are now connected to database "postgres" as user "postgres".
```

Для создания новой базы данных служит команда CREATE DATABASE:

```
=> CREATE DATABASE db;
CREATE DATABASE
```

```
=> \c db
You are now connected to database "db" as user "postgres".
```
Альтернативно можно было бы воспользоваться утилитой createdb.
```
=>   SELECT   datname,   datistemplate,   datallowconn,   datconnlimit   FROM
pg_database;
  datname  | datistemplate | datallowconn | datconnlimit
-----------+---------------+--------------+--------------
 postgres  | f             | t            |           -1
 template1 | t             | t            |           -1
 template0 | t             | f            |           -1
 db        | f             | t            |           -1
(4 rows)
```

Поскольку для создания по умолчанию используется шаблон template1, в ней также будет доступны функции пакета pgcrypto:
```
=> SELECT digest('Hello, world!', 'md5');
               digest
-----------------------------------
 \x6cd3556deb0da54bca060b4c39479839
(1 row)
```

Управление базами данных

Созданную базу данных можно переименовать (к ней не должно быть подключений):
```
=> \c postgres
You are now connected to database "postgres" as user "postgres".
=> ALTER DATABASE db RENAME TO appdb;
ALTER DATABASE
=>   SELECT   datname,   datistemplate,   datallowconn,   datconnlimit   FROM
pg_database;
  datname  | datistemplate | datallowconn | datconnlimit
-----------+---------------+--------------+--------------
 postgres  | f             | t            |           -1
 template1 | t             | t            |           -1
 template0 | t             | f            |           -1
 appdb     | f             | t            |           -1
(4 rows)
```

Можно изменить и другие параметры, например:
```
=> ALTER DATABASE appdb CONNECTION LIMIT 10;
ALTER DATABASE
=>   SELECT   datname,   datistemplate,   datallowconn,   datconnlimit   FROM
pg_database;
  datname  | datistemplate | datallowconn | datconnlimit
-----------+---------------+--------------+--------------
 postgres  | f             | t            |           -1
 template1 | t             | t            |           -1
 template0 | t             | f            |           -1
 appdb     | f             | t            |           10
(4 rows)
```

Размер базы данных

Размер базы данных можно узнать с помощью функции:
```
=> SELECT pg_database_size('appdb');
 pg_database_size
```

```
------------------
          7222887
(1 row)
```

Чтобы не считать разряды, можно вывести размер в читаемом виде:

```
=> SELECT pg_size_pretty(pg_database_size('appdb'));
 pg_size_pretty
----------------
 7054 kB
(1 row)
```

В этой базе данных еще нет пользовательских объектов (кроме расширения pgcrypto); фактически, это размер "пустой" базы.

Схемы

Список схем можно узнать командой psql (dn = describe namespace):

```
=> \dn
  List of schemas
  Name  |  Owner
--------+----------
 public | postgres
(1 row)
```

Создадим новую схему:

```
=> \c appdb
You are now connected to database "appdb" as user "postgres".
=> CREATE SCHEMA app;
CREATE SCHEMA
=> \dn
  List of schemas
  Name  |  Owner
--------+----------
 app    | postgres
 public | postgres
(2 rows)
```

Если теперь создать таблицу (и не указать имя схемы), в какую схему она попадет?

Надо посмотреть на путь поиска.

```
=> SHOW search_path;
   search_path
-----------------
 "$user", public
(1 row)
```

Конструкция "$user" обозначает схему с тем же именем, что и имя текущего пользователя (в нашем случае - postgres). Поскольку такой схемы нет, она игнорируется.
Чтобы не думать над тем, какие схемы есть, каких нет, и какие не указаны явно, можно воспользоваться функцией:

```
=> SELECT current_schemas(true);
   current_schemas
---------------------
 {pg_catalog,public}
(1 row)
```

Теперь создадим таблицу:

```
=> CREATE TABLE t(s text);
```

```
CREATE TABLE
=> INSERT INTO t VALUES ('Я - таблица t');
INSERT 0 1
```
Список таблиц можно получить командой \dt:
```
=> \dt
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | t    | table | postgres
(1 row)
```

Объект можно перемещать между схемами. Поскольку речь идет о логической организации, перемещение происходит только в системном каталоге; сами данные физически остаются на месте.
```
=> ALTER TABLE t SET SCHEMA app;
ALTER TABLE
```
Теперь к таблице t можно обращаться с явным указанием схемы:
```
=> SELECT * FROM app.t;
       s
---------------
 Я - таблица t
(1 row)
```

Но если опустить имя схемы, таблица не будет найдена:
```
=> SELECT * FROM t;
ERROR:  relation "t" does not exist
LINE 1: SELECT * FROM t;
                      ^
```

Путь поиска

Установим путь поиска, например, так:
```
=> SET search_path = public, app;
SET
```
Теперь таблица будет найдена.
```
=> SELECT * FROM t;
       s
---------------
 Я - таблица t
(1 row)
```

Здесь мы установили конфигурационный параметр на уровне сеанса (при переподключении значение пропадет). Устанавливать такое значение на уровне всего кластера тоже не правильно - возможно, этот путь нужен не всегда и не всем.
Но параметр можно установить и на уровне отдельной базы данных:
```
=> ALTER DATABASE appdb SET search_path = public, app;
ALTER DATABASE
```

Теперь он будет устанавливаться для всех новых подключений к БД appdb. Проверим:
```
=> \c appdb
You are now connected to database "appdb" as user "postgres".
=> SHOW search_path;
 search_path
-------------
 public, app
(1 row)
```

```
=> SELECT current_schemas(true);
     current_schemas
-------------------------
 {pg_catalog,public,app}
(1 row)
```

Временные таблицы и pg_temp

Создадим временную таблицу:

```
=> CREATE TEMP TABLE t(s text);
CREATE TABLE
=> \dt
           List of relations
  Schema    | Name | Type  |  Owner
------------+------+-------+----------
 pg_temp_3 | t    | table | postgres
(1 row)
```

Таблица создана в специальной схеме. Каждому сеансу выделяется отдельная "временная" схема, так что он может видеть только свои собственные временные таблицы.
Но куда пропала обычная таблица t?

Ответ дает развернутый путь поиска: в него теперь подставлена временная схема, и объект в ней "перекрывает" одноименный объект схемы app.

```
=> SELECT current_schemas(true);
          current_schemas
----------------------------------
 {pg_temp_3,pg_catalog,public,app}
(1 row)

=> INSERT INTO t VALUES ('Я - временная таблица');
INSERT 0 1
```

Тем не менее, к каждой из таблиц можно обращаться с явным указанием схемы. Для временной таблицы надо использовать псевдосхему pg_temp - она автоматически отображается в нужную схему pg_temp_N:

```
=> SELECT * FROM app.t;
       s
---------------
 Я - таблица t
(1 row)

=> SELECT * FROM pg_temp.t;
           s
-----------------------
 Я - временная таблица
(1 row)
```

В принципе, во временной схеме можно создавать не только таблицы.

```
=> CREATE VIEW v AS SELECT * FROM pg_temp.t;
NOTICE:  view "v" will be a temporary view
CREATE VIEW
```

Временные таблицы и данные в них могут иметь различные сроки жизни (в зависимости от указания ON COMMIT DELETE/PRESERVE/DROP). В любом случае при переподключении все объекты во временной схеме уничтожаются:

```
=> \c appdb
```

```
You are now connected to database "appdb" as user "postgres".
=> SELECT current_schemas(true);
     current_schemas
-------------------------
 {pg_catalog,public,app}
(1 row)

=> SELECT * FROM pg_temp.v;
ERROR:  relation "pg_temp.v" does not exist
LINE 1: SELECT * FROM pg_temp.v;
                      ^
=> SELECT * FROM pg_temp.t;
ERROR:  relation "pg_temp.t" does not exist
LINE 1: SELECT * FROM pg_temp.t;
                      ^
```

Удаление объектов

Схему нельзя удалить, если в ней находятся какие-либо объекты:

```
=> DROP SCHEMA app;
ERROR:  cannot drop schema app because other objects depend on it
DETAIL:  table t depends on schema app
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

Но можно удалить схему вместе со всеми ее объектами:

```
=> DROP SCHEMA app CASCADE;
NOTICE:  drop cascades to table t
DROP SCHEMA
```

Базу данных можно удалить, если к ней нет активных подключений.

```
=> \conninfo
You are connected to database "appdb" as user "postgres" via socket in "/tmp"
at port "5432".
=> \c postgres
You are now connected to database "postgres" as user "postgres".
=> DROP DATABASE appdb;
DROP DATABASE
```

Некоторые объекты системного каталога

Создадим базу данных и тестовые объекты:

```
=> CREATE DATABASE data_catalog;
CREATE DATABASE
=> \c data_catalog
You are now connected to database "data_catalog" as user "postgres".
=> CREATE TABLE employees(id serial PRIMARY KEY, name text, manager integer);
CREATE TABLE
=> CREATE VIEW top_managers AS
  SELECT * FROM employees WHERE manager IS NULL;
CREATE VIEW
```

Некоторые таблицы системного каталога нам уже знакомы из предыдущей темы. Это базы данных:

```
=> SELECT * FROM pg_database WHERE datname = 'data_catalog' \gx
-[ RECORD 1 ]-+-------------
datname       | data_catalog
datdba        | 10
encoding      | 6
```

```
datcollate    | en_US.UTF-8
datctype      | en_US.UTF-8
datistemplate | f
datallowconn  | t
datconnlimit  | -1
datlastsysoid | 12327
datfrozenxid  | 548
datminmxid    | 1
dattablespace | 1663
datacl        |
```

И схемы:

```
=> SELECT * FROM pg_namespace WHERE nspname = 'public' \gx
-[ RECORD 1 ]--------------------------------
nspname  | public
nspowner | 10
nspacl   | {postgres=UC/postgres,=UC/postgres}
```

Важная таблица pg_class хранит описание целого ряда объектов: таблиц, представлений (включая материализованные), индексов, последовательностей. Все эти объекты называются в PostgreSQL общим словом "отношение" (relation), отсюда и префикс "rel" в названии столбцов:

```
=>    SELECT    relname,    relkind,    relnamespace,    relfilenode,    relowner,
reltablespace
FROM pg_class WHERE relname ~ '^(emp|top).*';
     relname          |  relkind  |  relnamespace  |  relfilenode  |  relowner  |
reltablespace
------------------+---------+-------------+-------------+----------+--------
-------
 employees_id_seq | S         |           2200 |         16454 |         10 |
0
 employees          | r         |           2200 |         16456 |         10 |
0
 employees_pkey    | i         |           2200 |         16463 |         10 |
0
 top_managers      | v         |           2200 |         16465 |         10 |
0
(4 rows)
```

Типы объектов различаются по столбцу relkind.

Конечно, для каждого типа объектов имеет смысл только часть столбцов; кроме того, удобнее смотреть не на многочисленные OID, а на нормальные значения. Для этого существуют различные представления, например:

```
=> SELECT schemaname, tablename, tableowner, tablespace
FROM pg_tables WhERE schemaname = 'public';
 schemaname | tablename | tableowner | tablespace
------------+-----------+------------+------------
 public     | employees | postgres   |
(1 row)
```

```
=> SELECT *
FROM pg_views WHERE schemaname = 'public';
 schemaname |   viewname   | viewowner |              definition
```

```
------------+--------------+----------+-----------------------------------
-
 public           |  top_managers   |  postgres     |     SELECT  employees.id,
+
            |                      |                 |        employees.name,
+
            |                     |                  |        employees.manager
+
            |                       |                  |       FROM  employees
+
            |              |           |   WHERE (employees.manager IS NULL);
(1 row)
```

Использование команд psql

Список всех "отношений" можно посмотреть командой \d* в psql, где * - символ, обозначающий тип объекта (как relkind). Например, таблицы:

```
=> \dt
           List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+----------
 public | employees | table | postgres
(1 row)
```

Или представления:

```
=> \dv
            List of relations
 Schema |     Name     | Type |  Owner
--------+--------------+------+----------
 public | top_managers | view | postgres
(1 row)
```

Эти команды можно снабдить модификатором "+", чтобы получить больше информации:

```
=> \dt+
                      List of relations
 Schema |   Name    | Type  |  Owner   |    Size    | Description
--------+-----------+-------+----------+------------+-------------
 public | employees | table | postgres | 8192 bytes |
(1 row)
```

Чтобы получить детальную информацию о конкретном объекте, надо воспользоваться командой \d (без дополнительной буквы):

```
=> \d top_managers
             View "public.top_managers"
 Column  |  Type   | Collation | Nullable | Default
---------+---------+-----------+----------+---------
 id      | integer |           |          |
 name    | text    |           |          |
 manager | integer |           |          |
```

Модификатор "+" остается в силе.

```
=> \d+ top_managers
```

```
                        View "public.top_managers"
 Column  |  Type   | Collation | Nullable | Default | Storage  | Description
---------+---------+-----------+----------+---------+----------+-------------
 id      | integer |           |          |         | plain    |
 name    | text    |           |          |         | extended |
 manager | integer |           |          |         | plain    |
View definition:
 SELECT employees.id,
    employees.name,
    employees.manager
   FROM employees
  WHERE employees.manager IS NULL;
```

Помимо "отношений", аналогичным образом можно смотреть и на другие объекты, такие, как схемы (\dn) или функции (\df).
Модификатор "S" позволяет вывести не только пользовательские, но и системные объекты. А с помощью шаблона можно ограничить выборку:

```
=> \dfS pg*size
                                    List of functions
   Schema   |          Name          | Result data type | Argument data types
|  Type
------------+------------------------+------------------+--------------------
-+--------
 pg_catalog  |  pg_column_size              |  integer                 |  "any"
| normal
 pg_catalog  |  pg_database_size            |  bigint                  |  name
| normal
 pg_catalog  |  pg_database_size            |  bigint                  |  oid
| normal
 pg_catalog  |  pg_indexes_size             |  bigint                  |  regclass
| normal
 pg_catalog  |  pg_relation_size            |  bigint                  |  regclass
| normal
 pg_catalog  | pg_relation_size         | bigint               | regclass, text
| normal
 pg_catalog  |  pg_table_size               |  bigint                  |  regclass
| normal
 pg_catalog  |  pg_tablespace_size          |  bigint                  |  name
| normal
 pg_catalog  |  pg_tablespace_size          |  bigint                  |  oid
| normal
 pg_catalog  |  pg_total_relation_size  |  bigint                  |  regclass
| normal
(10 rows)
```

Psql предлагает большое количество команд для просмотра системного каталога. Как правило, они имеют мнемонические имена. Например, \df - describe function, \sf - show function:

```
=> \sf pg_catalog.pg_database_size(oid)
CREATE OR REPLACE FUNCTION pg_catalog.pg_database_size(oid)
 RETURNS bigint
 LANGUAGE internal
 PARALLEL SAFE STRICT
AS $function$pg_database_size_oid$function$
```

Полный список всегда можно посмотреть в документации или командой \?.

Изучение структуры системного каталога

Все команды psql, описывающие объекты, обращаются к таблицам системного каталога. Чтобы посмотреть, какие запросы на самом деле выполняет psql, можно установить переменную ECHO_HIDDEN:

```
=> \set ECHO_HIDDEN on
```

```
=> \dt employees
********* QUERY **********
SELECT n.nspname as "Schema",
  c.relname as "Name",
  CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'm' THEN
'materialized view' WHEN 'i' THEN 'index' WHEN 'S' THEN 'sequence' WHEN 's'
THEN 'special' WHEN 'f' THEN 'foreign table' WHEN 'p' THEN 'table' END as
"Type",
  pg_catalog.pg_get_userbyid(c.relowner) as "Owner"
FROM pg_catalog.pg_class c
     LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind IN ('r','p','s','')
      AND n.nspname !~ '^pg_toast'
  AND c.relname ~ '^(employees)$'
  AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 1,2;
**************************

           List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+----------
 public | employees | table | postgres
(1 row)

=> \unset ECHO_HIDDEN
```

OID и reg-типы

Как мы видели, описания таблиц и представлений хранятся в pg_class. А столбцы располагаются в отдельной таблице pg_attribute. Чтобы получить список столбцов конкретной таблицы, надо соединить pg_class и pg_attribute:

```
=> SELECT a.attname, a.atttypid
FROM pg_attribute a
WHERE a.attrelid = (
  SELECT oid FROM pg_class WHERE relname = 'employees'
)
AND a.attnum > 0;
 attname | atttypid
---------+----------
 id      |       23
 name    |       25
 manager |       23
(3 rows)
```

Используя reg-типы, запрос можно написать проще, без явного обращения к pg_class:

```
=> SELECT a.attname, a.atttypid
FROM pg_attribute a
WHERE a.attrelid = 'employees'::regclass
AND a.attnum > 0;
 attname | atttypid
---------+----------
 id      |       23
```

```
 name    |       25
 manager |       23
(3 rows)
```

Здесь мы преобразовали строку 'employees' к типу OID. Аналогично мы можем вывести OID как текстовое значение:

```
=> SELECT a.attname, a.atttypid::regtype
FROM pg_attribute a
WHERE a.attrelid = 'employees'::regclass
AND a.attnum > 0;
 attname | atttypid
---------+----------
 id      | integer
 name    | text
 manager | integer
(3 rows)
```

Служебные табличные пространства

При создании кластера создаются два табличных пространства:

```
=> SELECT * FROM pg_tablespace;
  spcname   | spcowner | spcacl | spcoptions
------------+----------+--------+------------
 pg_default |       10 |        |
 pg_global  |       10 |        |
(2 rows)
```

- pg_global - общие объекты кластера,
- pg_default - табличное пространство по умолчанию.

Пользовательские табличные пространства

Для нового табличного пространства нужен пустой каталог, владельцем которого является пользователь postgres.

```
postgres$ cd /home/postgres
postgres$ mkdir ts_dir
```

Теперь можно создать табличное пространство:

```
=> CREATE TABLESPACE ts LOCATION '/home/postgres/ts_dir';
CREATE TABLESPACE
```

Список табличных пространств можно получить и командой psql:

```
=> \db
              List of tablespaces
    Name    |  Owner   |       Location
------------+----------+----------------------
 pg_default | postgres |
 pg_global  | postgres |
 ts         | postgres | /home/postgres/ts_dir
(3 rows)
```

У каждой базы данных есть табличное пространство "по умолчанию".

Создадим БД и назначим ей ts в качестве такого пространства:

```
=> CREATE DATABASE appdb TABLESPACE ts;
CREATE DATABASE
```

Теперь все создаваемые таблицы и индексы будут попадать в ts, если явно не указать другое.

Подключимся к базе:

```
=> \c appdb
You are now connected to database "appdb" as user "postgres".
```

Создадим таблицу:

```
=> CREATE TABLE t1(id serial, name text);
CREATE TABLE
```

При создании объекта табличное пространство можно указать явно:

```
=> CREATE TABLE t2(n numeric) TABLESPACE pg_default;
CREATE TABLE
=> SELECT tablename, tablespace FROM pg_tables WHERE schemaname = 'public';
 tablename | tablespace
-----------+------------
 t1        |
 t2        | pg_default
(2 rows)
```

Пустое поле tablespace указывает на табличное пространство по умолчанию, а у второй таблицы поле заполнено.

Одно табличное пространство может использоваться для объектов нескольких баз данных.

```
=> CREATE DATABASE configdb;
CREATE DATABASE
```

У этой БД табличным пространством по умолчанию будет pg_default.

```
=> \c configdb
You are now connected to database "configdb" as user "postgres".
=> CREATE TABLE t(n integer) TABLESPACE ts;
CREATE TABLE
```

Управление объектами в табличных пространствах

Таблицы (и другие объекты, например, индексы), можно перемещать между табличными пространствами.

```
=> \c appdb
You are now connected to database "appdb" as user "postgres".
=> ALTER TABLE t1 SET TABLESPACE pg_default;
ALTER TABLE
=> SELECT tablename, tablespace FROM pg_tables WHERE schemaname = 'public';
 tablename | tablespace
-----------+------------
 t2        | pg_default
 t1        | pg_default
(2 rows)
```

Можно переместить и все объекты из одного табличного пространства в другое:

```
=> ALTER TABLE ALL IN TABLESPACE pg_default SET TABLESPACE ts;
ALTER TABLE
=> SELECT tablename, tablespace FROM pg_tables WHERE schemaname = 'public';
 tablename | tablespace
-----------+------------
 t2        |
 t1        |
(2 rows)
```

Важно понимать, что перенос в другое табличное пространство (в отличие от переноса в другую схему) - физическая операция, связанная с копированием файлов данных из каталога в каталог. На время ее выполнения перемещаемый объект полностью блокируется.

Размер табличного пространства

Мы уже рассматривали, как узнать объем, занимаемый базой данных. Объем можно посмотреть и в разрезе табличных пространств:

```
=> SELECT pg_size_pretty( pg_tablespace_size('ts') );
 pg_size_pretty
----------------
 7134 kB
(1 row)
```

Почему такой размер, если в табличном пространстве всего несколько пустых таблиц?

Поскольку ts является табличным пространством по умолчанию для базы appdb, в нем хранятся объекты системного каталога. Они и занимают место.

Удаление табличного пространства

Табличное пространство можно удалить, но только в том случае, если оно пусто.

```
=> DROP TABLESPACE ts;
ERROR:  tablespace "ts" is not empty
```

В отличие от удаления схемы, в команде DROP TABLESPACE нельзя использовать фразу CASCADE: объекты табличного пространства могут принадлежать разным базам данных, а подключены мы только к одной.

Но можно выяснить, в каких базах есть зависимые объекты. В этом нам поможет системный каталог.

Сначала узнаем и запомним OID табличного пространства:

```
=> SELECT OID FROM pg_tablespace WHERE spcname = 'ts';
  oid
-------
 16469
(1 row)

=> SELECT OID AS tsoid FROM pg_tablespace WHERE spcname = 'ts' \gset
```

Затем получим список баз данных, в которых есть объекты из удаляемого пространства:

```
=> SELECT datname
FROM pg_database
WHERE OID IN (SELECT pg_tablespace_databases(:tsoid));
 datname
----------
 configdb
 appdb
(2 rows)
```

Дальше подключаемся к каждой базе данных и получаем список объектов из pg_class:

```
=> \c configdb
You are now connected to database "configdb" as user "postgres".
=> SELECT relnamespace::regnamespace, relname, relkind
FROM pg_class
WHERE reltablespace = :tsoid;
```

```
 relnamespace | relname | relkind
--------------+---------+---------
 public       | t       | r
(1 row)
```

Таблица больше не нужна, удалим ее.

```
=> DROP TABLE t;
DROP TABLE
```

И вторая база данных. Но, поскольку ts является табличным пространством по умолчанию, у объектов в pg_class поле содержит ноль:

```
=> \c appdb
You are now connected to database "appdb" as user "postgres".
=> SELECT count(*) FROM pg_class WHERE reltablespace = 0;
 count
-------
   313
(1 row)
```

Это, как нам уже известно, объекты системного каталога.

Табличное пространство по умолчанию можно сменить; при этом все таблицы из старого пространства физически переносятся в новое. Предварительно надо отключиться от базы.

```
=> \c postgres
You are now connected to database "postgres" as user "postgres".
=> ALTER DATABASE appdb SET TABLESPACE pg_default;
ALTER DATABASE
```

Вот теперь табличное пространство может быть удалено.

```
=> DROP TABLESPACE ts;
DROP TABLESPACE
```

Расположение файлов

Посмотрим на файлы, принадлежащие таблице.

```
=> CREATE DATABASE data_lowlevel;
CREATE DATABASE
=> \c data_lowlevel
You are now connected to database "data_lowlevel" as user "postgres".
=> CREATE TABLE t(id serial PRIMARY KEY, n numeric);
CREATE TABLE
=> INSERT INTO t(n) SELECT g.id FROM generate_series(1,10000) AS g(id);
INSERT 0 10000
```

Базовое имя файла относительно PGDATA можно получить функцией:

```
=> SELECT pg_relation_filepath('t');
 pg_relation_filepath
----------------------
 base/16499/16502
(1 row)
```

Поскольку таблица находится в табличном пространстве pg_default, имя начинается на base. Затем идет каталог для базы данных:

```
=> SELECT OID FROM pg_database WHERE datname = 'data_lowlevel';
  oid
-------
```

```
 16499
(1 row)
```

Затем - собственно имя файла. Его можно узнать следующим образом:

```
=> SELECT relfilenode FROM pg_class WHERE relname = 't';
 relfilenode
-------------
       16502
(1 row)
```

Тем и удобна функция, что выдает готовый путь без необходимости выполнять несколько запросов к системному каталогу.

Посмотрим на файлы:

```
postgres$ ls -l /usr/local/pgsql/data/base/16499/16502*
-rw-------    1    postgres    postgres    409600    мар    31    15:25
/usr/local/pgsql/data/base/16499/16502
-rw-------    1    postgres    postgres     24576    мар    31    15:25
/usr/local/pgsql/data/base/16499/16502_fsm
```

Сейчас мы видим только два слоя: основной и карта свободного пространства. Карта видимости появляется только после выполнения очистки:

```
=> VACUUM t;
VACUUM
postgres$ ls -l /usr/local/pgsql/data/base/16499/16502*
-rw-------    1    postgres    postgres    409600    мар    31    15:25
/usr/local/pgsql/data/base/16499/16502
-rw-------    1    postgres    postgres     24576    мар    31    15:25
/usr/local/pgsql/data/base/16499/16502_fsm
-rw-------    1    postgres    postgres      8192    мар    31    15:25
/usr/local/pgsql/data/base/16499/16502_vm
```

Аналогично можно посмотреть и на файлы индекса...

```
=> \d t
                          Table "public.t"
 Column |  Type   | Collation | Nullable |            Default
--------+---------+-----------+----------+-------------------------------
 id     | integer |           | not null | nextval('t_id_seq'::regclass)
 n      | numeric |           |          |
Indexes:
    "t_pkey" PRIMARY KEY, btree (id)

=> SELECT pg_relation_filepath('t_pkey');
 pg_relation_filepath
----------------------
 base/16499/16509
(1 row)

postgres$ ls -l /usr/local/pgsql/data/base/16499/16509*
-rw-------    1    postgres    postgres    196608    мар    31    15:25
/usr/local/pgsql/data/base/16499/16509
```

...и на файлы последовательности:

```
=> SELECT pg_relation_filepath('t_id_seq');
 pg_relation_filepath
----------------------
 base/16499/16500
(1 row)

postgres$ ls -l /usr/local/pgsql/data/base/16499/16500*
```

```
-rw-------    1    postgres    postgres    8192    мар    31    15:25
/usr/local/pgsql/data/base/16499/16500
```

Существует полезное расширение oid2name, входящее в стандартную поставку, с помощью которого можно легко связать объекты БД и файлы.
Можно посмотреть все базы данных:

```
postgres$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  16499  data_lowlevel  pg_default
  12328       postgres  pg_default
  12327      template0  pg_default
      1      template1  pg_default
```

Можно посмотреть все объекты в базе:

```
postgres$ oid2name -d data_lowlevel
From database "data_lowlevel":
  Filenode  Table Name
----------------------
     16502           t
```

Или все табличные пространства в базе:

```
postgres$ oid2name -d data_lowlevel -s
All tablespaces:
   Oid  Tablespace Name
-----------------------
  1663       pg_default
  1664        pg_global
```

Можно по имени таблицы узнать имя файла:

```
postgres$ oid2name -d data_lowlevel -t t
From database "data_lowlevel":
  Filenode  Table Name
----------------------
     16502           t
```

Или наоборот, по номеру файла узнать таблицу:

```
postgres$ oid2name -d data_lowlevel -f 16502
From database "data_lowlevel":
  Filenode  Table Name
----------------------
     16502           t
```

Размер объектов и слоев

Размер каждого из файлов, входящих в слой, можно, конечно, посмотреть в файловой системе, но существуют и специальные функции.
Размер каждого слоя в отдельности:

```
=> SELECT pg_relation_size('t','main') main,
pg_relation_size('t','fsm') fsm,
pg_relation_size('t','vm') vm;
  main  |  fsm  |  vm
--------+-------+------
 409600 | 24576 | 8192
(1 row)
```

Размер таблицы без индексов:

```
=> SELECT pg_table_size('t');
 pg_table_size
---------------
```

```
        450560
(1 row)
```

Размер индексов таблицы:

```
=> SELECT pg_indexes_size('t');
 pg_indexes_size
-----------------
          196608
(1 row)
```

И размер всей таблицы, включая индексы:

```
=> SELECT pg_total_relation_size('t');
 pg_total_relation_size
------------------------
                 647168
(1 row)
```

TOAST

В таблице t есть столбец типа numeric. Этот тип может работать с очень большими числами. Например, с такими:

```
=> SELECT length( (123456789::numeric ^ 12345::numeric)::text );
 length
--------
  99907
(1 row)
```

При этом, если вставить такое значение в таблицу, размер файлов не изменится:

```
=> SELECT pg_relation_size('t','main');
 pg_relation_size
------------------
           409600
(1 row)

=> INSERT INTO t(n) SELECT 123456789::numeric ^ 12345::numeric;
INSERT 0 1
=> SELECT pg_relation_size('t','main');
 pg_relation_size
------------------
           409600
(1 row)
```

Поскольку версия строки не помещается на одну страницу, она хранится в отдельной TOAST-таблице. TOAST-таблица и индекс к ней создаются автоматически для каждой таблицы, в которой есть потенциально "длинный" тип данных, и используются по необходимости.

Имя и идентификатор такой таблицы можно найти следующим образом:

```
=> SELECT relname, relfilenode FROM pg_class WHERE OID = (
  SELECT reltoastrelid FROM pg_class WHERE relname='t'
);
    relname     | relfilenode
----------------+-------------
 pg_toast_16502 |       16506
(1 row)
```

Вот и файлы TOAST-таблицы:

```
postgres$ ls -l /usr/local/pgsql/data/base/16499/16506*
-rw-------     1     postgres     postgres     57344     мар     31     15:25
/usr/local/pgsql/data/base/16499/16506
-rw-------     1     postgres     postgres     24576     мар     31     15:25
/usr/local/pgsql/data/base/16499/16506_fsm
```

Существуют несколько стратегий работы с длинными значениями. Название стратегии указывается в поле Storage:

```
=> \d+ t
                                                  Table "public.t"
 Column |  Type   | Collation | Nullable |              Default               |
Storage | Stats target | Description
--------+---------+-----------+----------+------------------------------------+---
------+--------------+-------------
 id     | integer |           | not null | nextval('t_id_seq'::regclass)      |
plain   |              |
 n      | numeric |           |          |                                    |
main    |              |
Indexes:
    "t_pkey" PRIMARY KEY, btree (id)
```

- plain - TOAST не применяется (тип имеет фиксированную длину);
- main - приоритет сжатия;
- extended - применяется как сжатие, так и отдельное хранение;
- external - только отдельно хранение без сжатия.

Стратегию можно изменить, если это необходимо. Например, если известно, что в столбце хранятся уже сжатые данные, разумно поставить стратегию external.
Просто для примера:

```
=> ALTER TABLE t ALTER COLUMN n SET STORAGE extended;
ALTER TABLE
```

Эта операция не меняет существующие данные в таблице, но определяет стратегию работы с новыми данными.

Тема 4. Задачи администрирования

Задание: просмотрите предложенный код, демонстрирующий возможности PostgreSQL и повторите на своём сервере со своей базой данных, по следующим вопросам:

- Настройка
- Статистика
- Текущие активности
- Анализ журнала
- Оценка разрастания таблиц и индексов

Отчет по лабораторной работе должен содержать**:**

1. Фамилию и номер группы учащегося, задание
2. Краткое описание базы данных
3. Результаты выполнения операторов
4. Код

Демонстрация (пример для повторения)

Настройка

Вначале включим сбор статистики ввода-вывода и выполнения функций.

```
=> ALTER SYSTEM SET track_io_timing=on;
ALTER SYSTEM
=> ALTER SYSTEM SET track_functions='all';
ALTER SYSTEM
=> SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

Смотреть на активности сервера имеет смысл, когда какие-то активности на самом деле есть. Чтобы сымитировать нагрузку, воспользуемся pgbench - штатной утилитой для запуска эталонных тестов.

Сначала утилита создает набор таблиц и заполняет их данными.

```
=> CREATE DATABASE admin_monitoring;
CREATE DATABASE
postgres$ pgbench -i admin_monitoring
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
creating tables...
100000 of 100000 tuples (100%) done (elapsed 0.13 s, remaining 0.00 s)
vacuum...
set primary keys...
done.
```

Затем сбросим все накопленные ранее статистики.

```
=> \c admin_monitoring
You are now connected to database "admin_monitoring" as user "postgres".
=> SELECT pg_stat_reset();
 pg_stat_reset
---------------

(1 row)

=> SELECT pg_stat_reset_shared('bgwriter');
 pg_stat_reset_shared
----------------------

(1 row)
```

Статистика

Теперь запускаем тест TPC-B на несколько секунд.

```
postgres$ pgbench -T 10 admin_monitoring
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
duration: 10 s
number of transactions actually processed: 9304
latency average = 1.075 ms
tps = 930.308591 (including connections establishing)
```

```
tps = 930.517389 (excluding connections establishing)
=> VACUUM pgbench_accounts;
VACUUM
```

Теперь мы можем посмотреть статистику обращения к таблицам в терминах строк:

```
=> SELECT * FROM pg_stat_all_tables WHERE relid='pgbench_accounts'::regclass
\gx
-[ RECORD 1 ]-------+-----------------
relid               | 16519
schemaname          | public
relname             | pgbench_accounts
seq_scan            | 0
seq_tup_read        | 0
idx_scan            | 18608
idx_tup_fetch       | 18608
n_tup_ins           | 0
n_tup_upd           | 9304
n_tup_del           | 0
n_tup_hot_upd       | 7694
n_live_tup          | 0
n_dead_tup          | 3171
n_mod_since_analyze | 9304
last_vacuum         |
last_autovacuum     |
last_analyze        |
last_autoanalyze    |
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 0
```

И в терминах страниц:

```
=> SELECT * FROM pg_statio_all_tables WHERE
relid='pgbench_accounts'::regclass \gx
-[ RECORD 1 ]---+-----------------
relid           | 16519
schemaname      | public
relname         | pgbench_accounts
heap_blks_read  | 25
heap_blks_hit   | 34524
idx_blks_read   | 221
idx_blks_hit    | 40405
toast_blks_read |
toast_blks_hit  |
tidx_blks_read  |
tidx_blks_hit   |
```

Существуют аналогичные представления для индексов:

```
=> SELECT * FROM pg_stat_all_indexes WHERE relid='pgbench_accounts'::regclass
\gx
-[ RECORD 1 ]-+----------------------
relid         | 16519
indexrelid    | 16530
schemaname    | public
relname       | pgbench_accounts
indexrelname  | pgbench_accounts_pkey
idx_scan      | 18608
idx_tup_read  | 20329
idx_tup_fetch | 18608
```

```
=> SELECT * FROM pg_statio_all_indexes WHERE
relid='pgbench_accounts'::regclass \gx
-[ RECORD 1 ]-+---------------------
relid         | 16519
indexrelid    | 16530
schemaname    | public
relname       | pgbench_accounts
indexrelname  | pgbench_accounts_pkey
idx_blks_read | 221
idx_blks_hit  | 40405
```

А также варианты для пользовательских и системных объектов (all, user, sys), для статистики текущей транзакции (pg_stat_xact*) и др.

Можно посмотреть глобальную статистику по всей базе данных:

```
=> SELECT * FROM pg_stat_database WHERE datname='admin_monitoring' \gx
-[ RECORD 1 ]--+---------------------------
datid          | 16512
datname        | admin_monitoring
numbackends    | 1
xact_commit    | 9314
xact_rollback  | 0
blks_read      | 330
blks_hit       | 121986
tup_returned   | 124242
tup_fetched    | 18782
tup_inserted   | 9304
tup_updated    | 27913
tup_deleted    | 0
conflicts      | 0
temp_files     | 0
temp_bytes     | 0
deadlocks      | 0
blk_read_time  | 4.578
blk_write_time | 0
stats_reset    | 2019-03-31 15:25:42.27966+03
```

Здесь есть информация о количестве произошедших взаимоблокировок, зафиксированных и отмененных транзакций, использовании временных файлов.

Отдельно доступна статистика по процессам фоновой записи и контрольной точки, ввиду ее важности для мониторинга экземпляра:

```
=> CHECKPOINT;
CHECKPOINT
=> SELECT * FROM pg_stat_bgwriter \gx
-[ RECORD 1 ]---------+------------------------------
checkpoints_timed     | 0
checkpoints_req       | 1
checkpoint_write_time | 35
checkpoint_sync_time  | 23
buffers_checkpoint    | 1953
buffers_clean         | 0
maxwritten_clean      | 0
buffers_backend       | 1700
buffers_backend_fsync | 0
buffers_alloc         | 391
stats_reset           | 2019-03-31 15:25:42.329588+03
```

- buffers_clean - количество страниц, записанных фоновой записью;
- buffers_checkpoint - количество страниц, записанных контрольной точкой;
- buffers_backend - количество страниц, записанных серверными процессами.

Текущие активности

Воспроизведем сценарий, в котором один процесс блокирует выполнение другого, и попробуем разобраться в ситуации с помощью системных представлений.

Создадим таблицу с одной строкой:

```
=> CREATE TABLE t(n integer);
CREATE TABLE
=> INSERT INTO t VALUES(42);
INSERT 0 1
```

Запустим два сеанса, один из которых изменяет таблицу и ничего не делает:

```
postgres$ psql -d admin_monitoring
=> BEGIN;
BEGIN
=> UPDATE t SET n = n + 1;
UPDATE 1
```

А второй пытается изменить ту же строку и блокируется:

```
postgres$ psql -d admin_monitoring
=> UPDATE t SET n = n + 2;
```

Посмотрим информацию об обслуживающих процессах:

```
=>     SELECT     pid,     query,     state,     wait_event,     wait_event_type,
pg_blocking_pids(pid)
FROM pg_stat_activity
WHERE backend_type = 'client backend' \gx
-[ RECORD 1 ]----+------------------------------------------------------------
-----------------
pid              | 30562
query            | UPDATE t SET n = n + 1;
state            | idle in transaction
wait_event       | ClientRead
wait_event_type  | Client
pg_blocking_pids | {}
-[ RECORD 2 ]----+------------------------------------------------------------
-----------------
pid              | 30022
query              | SELECT pid, query, state, wait_event, wait_event_type,
pg_blocking_pids(pid)+
                 |                    FROM                    pg_stat_activity
+
                 | WHERE backend_type = 'client backend'
state            | active
wait_event       |
wait_event_type  |
pg_blocking_pids | {}
-[ RECORD 3 ]----+------------------------------------------------------------
-----------------
pid              | 30614
query            | UPDATE t SET n = n + 2;
state            | active
wait_event       | transactionid
wait_event_type  | Lock
pg_blocking_pids | {30562}
```

Состояние idle in transaction означает, что сеанс начал транзакцию, но в настоящее время ничего не делает, а транзакция осталась незавершенной. Это может стать проблемой, если ситуация возникает систематически, например, из-за некорректной реализации

приложения или из-за ошибок в драйвере - поскольку открытый сеанс расходует оперативную память.
Начиная с версии 9.6 в арсенале администратора появился параметр:

- idle_in_transaction_session_timeout - принудительно завершает сеансы, в которых транзакция простаивает больше указанного времени.

А мы покажем, как завершить блокирующий сеанс вручную. Сначала запомним номер заблокированного процесса:

```
=> SELECT pid as blocked_pid
FROM pg_stat_activity
WHERE backend_type = 'client backend'
AND cardinality(pg_blocking_pids(pid)) > 0 \gset
```

В PostgreSQL версий ниже 9.6 нет функции pg_blocking_pids, но блокирующий процесс можно вычислить, используя запросы к таблице блокировок. Запрос покажет две строки: одна транзакция получила блокировку (granted), другая - нет и ожидает.

```
=> SELECT locktype, transactionid, pid, mode, granted
FROM pg_locks
WHERE transactionid IN (
  SELECT transactionid FROM pg_locks WHERE pid = :blocked_pid AND NOT granted
);
   locktype    | transactionid |  pid  |     mode      | granted
---------------+---------------+-------+---------------+---------
 transactionid |          9929 | 30562 | ExclusiveLock | t
 transactionid |          9929 | 30614 | ShareLock     | f
(2 rows)
```

В общем случае нужно аккуратно учитывать тип блокировки.

Выполнение запроса можно прервать функцией pg_cancel_backend. В нашем случае транзакция простаивает, так что просто прерываем сеанс, вызвав pg_terminate_backend:

```
=> SELECT pg_terminate_backend(b.pid)
FROM unnest(pg_blocking_pids(:blocked_pid)) AS b(pid);
 pg_terminate_backend
----------------------
 t
(1 row)
```

Функция unnest нужна, поскольку pg_blocking_pids возвращает массив идентификаторов процессов, блокирующих искомый серверный процесс. В нашем примере блокирующий процесс один, но в общем случае их может быть несколько.

Проверим состояние серверных процессов.

```
=> SELECT pid, query, state, wait_event, wait_event_type
FROM pg_stat_activity
WHERE backend_type = 'client backend' \gx
-[ RECORD 1 ]---+-----------------------------------------------------
pid             | 30022
query           | SELECT pid, query, state, wait_event, wait_event_type+
                | FROM pg_stat_activity                                +
                | WHERE backend_type = 'client backend'
state           | active
wait_event      |
wait_event_type |
-[ RECORD 2 ]---+-----------------------------------------------------
pid             | 30614
query           | UPDATE t SET n = n + 2;
state           | idle
wait_event      | ClientRead
wait_event_type | Client
```

Осталось только два, причем заблокированный успешно завершил транзакцию.

Начиная с версии 10 представление pg_stat_activity показывает информацию не только про обслуживающие процессы, но и про служебные фоновые процессы экземпляра:

```
=> SELECT pid, backend_type, backend_start, state
FROM pg_stat_activity;
  pid  |     backend_type     |         backend_start         | state
-------+----------------------+-------------------------------+--------
 22616 | background worker    | 2019-03-31 15:25:22.337945+03 |
 22614 | autovacuum launcher  | 2019-03-31 15:25:22.339051+03 |
 30022 | client backend       | 2019-03-31 15:25:42.252197+03 | active
 30614 | client backend       | 2019-03-31 15:25:54.465374+03 | idle
 22612 | background writer    | 2019-03-31 15:25:22.340118+03 |
 22611 | checkpointer         | 2019-03-31 15:25:22.340522+03 |
 22613 | walwriter            | 2019-03-31 15:25:22.339701+03 |
(7 rows)
```

Сравним с тем, что показывает операционная система:

```
postgres$ ps -o pid,command --ppid `head -n 1 $PGDATA/postmaster.pid`
  PID COMMAND
22611 postgres: checkpointer process
22612 postgres: writer process
22613 postgres: wal writer process
22614 postgres: autovacuum launcher process
22615 postgres: stats collector process
22616 postgres: bgworker: logical replication launcher
30022 postgres: postgres admin_monitoring [local] idle
30614 postgres: postgres admin_monitoring [local] idle
```

Можно заметить, что в pg_stat_activity не попадает процесс stats collector.

Анализ журнала

Посмотрим самый простой случай. Например, нас интересуют сообщения FATAL:

```
postgres$ grep FATAL /home/postgres/logfile | tail -n 10
2019-03-31 15:24:53.802 MSK [16551] FATAL:  terminating logical replication worker due to administrator command
2019-03-31 15:25:24.354 MSK [23709] FATAL:  lock file "postmaster.pid" already exists
2019-03-31 15:25:27.582 MSK [24944] FATAL:  lock file "postmaster.pid" already exists
2019-03-31 15:25:29.020 MSK [25547] FATAL:  lock file "postmaster.pid" already exists
2019-03-31 15:25:29.953 MSK [25838] FATAL:  lock file "postmaster.pid" already exists
2019-03-31 15:25:33.326 MSK [27301] FATAL:  lock file "postmaster.pid" already exists
2019-03-31 15:25:35.305 MSK [28143] FATAL:  lock file "postmaster.pid" already exists
2019-03-31 15:25:38.027 MSK [28969] FATAL:  lock file "postmaster.pid" already exists
2019-03-31 15:25:41.015 MSK [29832] FATAL:  lock file "postmaster.pid" already exists
2019-03-31 15:25:54.769 MSK [30562] FATAL:  terminating connection due to administrator command
```

Сообщение "terminating connection" вызвано тем, что мы завершали блокирующий процесс.

Обычное применение журнала - анализ наиболее продолжительных запросов. Добавим к строкам журнала номер процесса и включим вывод команд и времени их выполнения:

```
=> ALTER SYSTEM SET log_min_duration_statement=0;
ALTER SYSTEM
=> ALTER SYSTEM SET log_line_prefix='(pid=%p) ';
ALTER SYSTEM
=> SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

Теперь выполним какую-нибудь команду:

```
=> SELECT sum(random()) FROM generate_series(1,1000000);
       sum
------------------
 500182.961182054
(1 row)
```

И посмотрим журнал:

```
postgres$ tail -n 1 /home/postgres/logfile
(pid=30022) LOG:  duration: 364.589 ms  statement: SELECT sum(random()) FROM
generate_series(1,1000000);
```

Оценка разрастания таблиц и индексов

Оценить степень разрастания объектов, чтобы принять решение о полной очистке, можно разными способами:

- запросами к системному каталогу;
- используя расширение pgstattuple.

```
=> CREATE EXTENSION pgstattuple;
CREATE EXTENSION
```

Создадим таблицу и заполним ее данными:

```
=> CREATE TABLE bloat(id serial, s text);
CREATE TABLE
=> INSERT INTO bloat(s)
  SELECT g.id::text FROM generate_series(1,100000) AS g(id);
INSERT 0 100000
=> CREATE INDEX ON bloat(s);
CREATE INDEX
```

С помощью расширения можно проверить состояние таблицы:

```
=> SELECT * FROM pgstattuple('bloat') \gx
-[ RECORD 1 ]------+--------
table_len          | 4014080
tuple_count        | 100000
tuple_len          | 3388895
tuple_percent      | 84.43
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 4356
free_percent       | 0.11
```

- tuple_percent - доля полезной информации (не 100% из-за накладных расходов).

И индекса:

```
=> SELECT * FROM pgstatindex('bloat_s_idx') \gx
-[ RECORD 1 ]------+--------
version            | 2
tree_level         | 1
index_size         | 2252800
root_block_no      | 3
internal_pages     | 1
leaf_pages         | 273
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 89.98
leaf_fragmentation | 0
```

- avg_leaf_density - заполненность листовых страниц индекса;
- leaf_fragmentation - характеристика физической упорядоченности страниц.

Теперь обновим половину строк:

```
=> UPDATE bloat SET s = s || '!' WHERE id % 2 = 0;
UPDATE 50000
```

Посмотрим на таблицу снова:

```
=> SELECT * FROM pgstattuple('bloat') \gx
-[ RECORD 1 ]------+--------
table_len          | 6021120
tuple_count        | 100000
tuple_len          | 3438895
tuple_percent      | 57.11
dead_tuple_count   | 50000
dead_tuple_len     | 1694450
dead_tuple_percent | 28.14
free_space         | 4732
free_percent       | 0.08
```

Плотность уменьшилась.

Чтобы не читать всю таблицу целиком, можно попросить pgstattuple показать приблизительную информацию:

```
=> SELECT * FROM pgstattuple_approx('bloat') \gx
-[ RECORD 1 ]--------+-------------------
table_len            | 6021120
scanned_percent      | 100
approx_tuple_count   | 100000
approx_tuple_len     | 3438895
approx_tuple_percent | 57.113875823767
dead_tuple_count     | 50000
dead_tuple_len       | 1694450
dead_tuple_percent   | 28.1417742878401
approx_free_space    | 4732
approx_free_percent  | 0.0785900297619048
```

И посмотрим на индекс:

```
=> SELECT * FROM pgstatindex('bloat_s_idx') \gx
-[ RECORD 1 ]------+--------
version            | 2
tree_level         | 2
index_size         | 4505600
root_block_no      | 412
internal_pages     | 3
```

```
leaf_pages          | 546
empty_pages         | 0
deleted_pages       | 0
avg_leaf_density    | 67.6
leaf_fragmentation | 49.82
```

Плотность тоже уменьшилась.

Тема 5. Роли, привилегии и операторы для работы с ними

Задание:

1) Разработать в базе данных, созданной и заполненной на предыдущих лабораторных работах:

a. создайте две новых роли;
b. наделите первую роль привилегиями на часть таблиц;
c. назначьте второй роли первую в качестве роли;
d. отмените одну из привилегий;
e. изменить первую роль;
f. удалите вторую роль;
g. войдите под первой ролью и проверьте доступность привилегий.

Отчет по лабораторной работе должен содержать:

1. Фамилию и номер группы учащегося, задание.
2. Коды операций.
3. Принтскриты всех выполненных операторов.

Материал для выполнения

База данных содержит одну или несколько именованных *схем*, которые в свою очередь содержат таблицы. Схемы также содержат именованные объекты других видов, включая типы данных, функции и операторы. Одно и то же имя объекта можно свободно использовать в разных схемах, например и schema1, и myschema могут содержать таблицы с именем mytable. В отличие от баз данных, схемы не ограничивают доступ к данным: пользователь может обращаться к объектам в любой схеме текущей базы данных, если ему назначены соответствующие права.

Есть несколько возможных объяснений, для чего стоит применять схемы:

- Чтобы одну базу данных могли использовать несколько пользователей, независимо друг от друга.
- Чтобы объединить объекты базы данных в логические группы для облегчения управления ими.
- Чтобы в одной базе сосуществовали разные приложения, и при этом не возникало конфликтов имён.

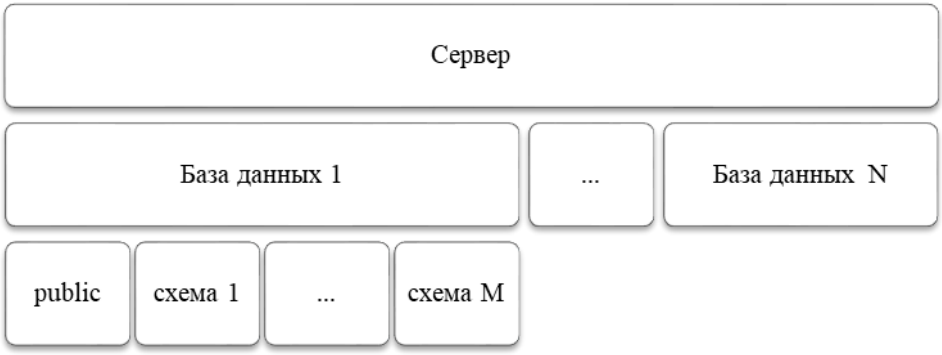
Схемы в некоторым смысле подобны каталогам в операционной системе, но они не могут быть вложенными.

В PostgreSQL схема не связана с пользователем. Роли определяется не на уровне базы данных, а на уровне сервера (кластера).

По умолчанию пользователь не может обращаться к объектам в чужих схемах. Чтобы изменить это, владелец схемы должен дать пользователю право USAGE для данной схемы. Чтобы пользователи могли использовать объекты схемы, может понадобиться назначить дополнительные права на уровне объектов.

Пользователю также можно разрешить создавать объекты в схеме, не принадлежащей ему. Для этого ему нужно дать право CREATE в требуемой схеме. Заметьте, что по умолчанию все имеют права CREATE и USAGE в схеме public.

Благодаря этому все пользователи могут подключаться к заданной базе данных и создавать объекты в её схеме public.



Роль — это сущность, которая может владеть объектами и иметь определённые права в базе; роль может представлять «пользователя», «группу» или и то, и другое, в зависимости от варианта использования.

PostgreSQL использует концепцию ролей (*roles*) для управления разрешениями на доступ к базе данных. Роль можно рассматривать как пользователя базы данных или как группу пользователей, в зависимости от того, как роль настроена. Роли могут владеть объектами базы данных (например, таблицами и функциями) и выдавать другим ролям разрешения на доступ к этим объектам, управляя тем, кто имеет доступ и к каким объектам. Кроме того, можно предоставить одной роли *членство* в другой роли, таким образом одна роль может использовать привилегии других ролей.

Концепция ролей включает в себя концепцию пользователей («users») и групп («groups»). До версии 8.1 в PostgreSQL пользователи и группы были отдельными сущностями, но теперь есть только роли. Любая роль может использоваться в качестве пользователя, группы, и того и другого.

Для добавления и удаления членов ролей, используемых в качестве групп, рекомендуется использовать GRANT и REVOKE.

Роль базы данных может иметь атрибуты, определяющие её полномочия и взаимодействие с системой аутентификации клиентов.

Для PostgreSQL: *CREATE ROLE = CREATE GROUP = CREATE USER*

Команда CREATE USER теперь является просто синонимом CREATE ROLE. Единственное отличие в том, что для команды, записанной в виде CREATE USER, по умолчанию подразумевается LOGIN, а в виде CREATE ROLE подразумевается NOLOGIN.

Оператор CREATE GROUP теперь является синонимом оператора CREATE ROLE.

Атрибуты роли

Право подключения

Только роли с атрибутом LOGIN могут использоваться для начального подключения к базе данных. Роль с атрибутом LOGIN можно рассматривать как пользователя базы данных. Для создания такой роли можно использовать любой из вариантов:

CREATE ROLE *имя* LOGIN;

CREATE USER *имя*;

Статус суперпользователя

Суперпользователь базы данных обходит все проверки прав доступа, за исключением права на вход в систему. Это опасная привилегия и она не должна использоваться небрежно. Лучше всего выполнять большую часть работы не как суперпользователь. Для создания нового суперпользователя используется CREATE

ROLE *имя* SUPERUSER. Это нужно выполнить из под роли, которая также является суперпользователем.

Создание базы данных

Роль должна явно иметь разрешение на создание базы данных (за исключением суперпользователей, которые пропускают все проверки). Для создания такой роли используется CREATE ROLE *имя* CREATEDB.

Создание роли

Роль должна явно иметь разрешение на создание других ролей (за исключением суперпользователей, которые пропускают все проверки). Для создания такой роли используется CREATE ROLE *имя* CREATEROLE. Роль с привилегией CREATEROLE может также изменять и удалять другие роли, а также выдавать и отзывать членство в ролях. Однако, для создания, изменения, удаления суперпользовательских ролей, а также изменения в них членства, требуется иметь статус суперпользователя; привилегии CREATEROLE в таких случаях недостаточно.

Запуск репликации

Роль должна иметь явное разрешение на запуск потоковой репликации (за исключением суперпользователей, которые пропускают все проверки). Роль, используемая для потоковой репликации, также должна иметь атрибут LOGIN. Для создания такой роли используется CREATE ROLE *имя* REPLICATION LOGIN.

Пароль

Пароль имеет значение, если метод аутентификации клиентов требует, чтобы пользователи предоставляли пароль при подключении к базе данных. Методы аутентификации password и md5 используют пароли. База данных и операционная система используют раздельные пароли. Пароль указывается при создании роли: CREATE ROLE *имя* PASSWORD '*строка*'.

Предопределённые роли

Роль	Разрешаемый доступ
pg_read_all_settings	Читать все конфигурационные переменные, даже те, что обычно видны только суперпользователям.
pg_read_all_stats	Читать все представления pg_stat_* и использовать различные расширения, связанные со статистикой, даже те, что обычно видны только суперпользователям.
pg_stat_scan_tables	Выполнять функции мониторинга, которые могут устанавливать блокировки ACCESS SHARE в таблицах, возможно, на длительное время.
pg_signal_backend	Передавать сигналы другим обслуживающим процессам (например, отменять запрос, завершать процесс).
pg_read_server_files	Читать файлы в любом месте файловой системы, куда имеет доступ СУБД на сервере, выполняя COPY и другие функции работы с файлами.
pg_write_server_files	Записывать файлы в любом месте файловой системы, куда имеет

	доступ СУБД на сервере, выполняя COPY и другие функции работы с файлами.
pg_execute_server_program	Выполнять программы на сервере (от имени пользователя, запускающего СУБД), так же, как это делает команда COPY и другие функции, выполняющие программы на стороне сервера.
pg_monitor	Читать/выполнять различные представления и функции для мониторинга. Эта роль включена в роли pg_read_all_settings, pg_read_all_stats и pg_stat_scan_tables.

Синтаксис

CREATE ROLE — создать роль в базе данных

CREATE ROLE *имя* [[WITH] *параметр* [...]]

Здесь *параметр*:

SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT *предел_подключений*
| [ENCRYPTED] PASSWORD '*пароль*' | PASSWORD NULL
| VALID UNTIL '*дата_время*'
| IN ROLE *имя_роли* [, ...]
| IN GROUP *имя_роли* [, ...]
| ROLE *имя_роли* [, ...]
| ADMIN *имя_роли* [, ...]
| USER *имя_роли* [, ...]
| SYSID *uid*

ALTER ROLE — изменить роль в базе данных

ALTER ROLE *указание_роли* [WITH] *параметр* [...]

Здесь *параметр*:

SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT *предел_подключений*
| [ENCRYPTED] PASSWORD '*пароль*' | PASSWORD NULL
| VALID UNTIL '*дата_время*'

ALTER ROLE *имя* RENAME TO *новое_имя*

ALTER ROLE { *указание_роли* | ALL } [IN DATABASE *имя_бд*] SET *параметр_конфигурации* { TO | = } { *значение* | DEFAULT }

ALTER ROLE { *указание_роли* | ALL } [IN DATABASE *имя_бд*] SET *параметр_конфигурации* FROM CURRENT

ALTER ROLE { *указание_роли* | ALL } [IN DATABASE *имя_бд*] RESET *параметр_конфигурации*

ALTER ROLE { *указание_роли* | ALL } [IN DATABASE *имя_бд*] RESET ALL

Здесь *указание_роли*:

имя_роли

| CURRENT_USER
| SESSION_USER

SUPERUSER
NOSUPERUSER

Эти предложения определяют, будет ли эта роль «суперпользователем», который может переопределить все ограничения доступа в базе данных. Статус суперпользователя несёт опасность и назначать его следует только в случае необходимости. Создать нового суперпользователя может только суперпользователь. В отсутствие этих предложений по умолчанию подразумевается NOSUPERUSER.

CREATEDB
NOCREATEDB

Эти предложения определяют, сможет ли роль создавать базы данных. Указание CREATEDB даёт новой роли это право, а NOCREATEDB запрещает роли создавать базы данных. По умолчанию подразумевается NOCREATEDB.

CREATEROLE
NOCREATEROLE

Эти предложения определяют, сможет ли роль создавать новые роли (т. е. выполнять CREATE ROLE). Роль с правом CREATEROLE может также изменять и удалять другие роли. По умолчанию подразумевается NOCREATEROLE.

INHERIT
NOINHERIT

Эти предложения определяют, будет ли роль «наследовать» права ролей, членом которых она является. Роль с атрибутом INHERIT может автоматически использовать в базе данных любые права, назначенные всем ролям, в которые она включена, непосредственно или опосредованно. Без INHERIT членство в другой роли позволяет только выполнить SET ROLE и переключиться на эту роль; правами, назначенными другой роли, можно будет пользоваться только после этого. По умолчанию подразумевается INHERIT.

LOGIN
NOLOGIN

Эти предложения определяют, разрешается ли новой роли вход на сервер; то есть, может ли эта роль стать начальным авторизованным именем при подключении клиента. Можно считать, что роль с атрибутом LOGIN соответствует пользователю. Роли без этого атрибута бывают полезны для управления доступом в базе данных, но это не пользователи в обычном понимании.

REPLICATION
NOREPLICATION

Эти предложения определяют, будет ли роль ролью репликации. Чтобы роль могла подключаться к серверу в режиме репликации (в режиме физической или логической репликации) и создавать/удалять слоты репликации, у неё должен быть этот атрибут (либо это должна быть роль суперпользователя). Роль, имеющая атрибут REPLICATION, обладает очень большими привилегиями и поэтому этот атрибут должны иметь только роли, фактически используемые для репликации. По умолчанию подразумевается вариант NOREPLICATION.

BYPASSRLS
NOBYPASSRLS

Эти предложения определяют, будут ли для роли игнорироваться все политики защиты на уровне строк (RLS). Значение по умолчанию — NOBYPASSRLS. Заметьте, что pg_dump по умолчанию отключает row_security (устанавливает значение OFF) для уверенности, что выгружено всё содержимое таблицы. Если пользователь, запускающий pg_dump, не будет иметь необходимых прав, он

получит ошибку. Суперпользователь и владелец выгружаемой таблицы всегда обходят защиту RLS.

CONNECTION LIMIT *предел_подключений*

Если роли разрешён вход, этот параметр определяет, сколько параллельных подключений может установить роль. Значение -1 (по умолчанию) снимает ограничение. Заметьте, что под это ограничение подпадают только обычные подключения. Ни подготовленные транзакции, ни соединения фоновых рабочих процессов в расчёт не берутся.

[ENCRYPTED] PASSWORD '*пароль*'
PASSWORD NULL

Задаёт пароль роли. (Пароль полезен только для ролей с атрибутом LOGIN, но задать его можно и для ролей без такого атрибута.) Если проверка подлинности по паролю не будет использоваться, этот параметр можно опустить. При указании пустого значения будет задан пароль NULL, что не позволит данному пользователю пройти проверку подлинности по паролю. При желании пароль NULL можно установить явно, указав PASSWORD NULL.

VALID UNTIL '*дата_время*'

Предложение VALID UNTIL устанавливает дату и время, после которого пароль роли перестаёт действовать. Если это предложение отсутствует, срок действия пароля будет неограниченным.

IN ROLE *имя_роли*

В предложении IN ROLE перечисляются одна или несколько существующих ролей, в которые будет немедленно включена новая роль. (Заметьте, что добавить новую роль с правами администратора таким образом нельзя; для этого надо отдельно выполнить команду GRANT.)

IN GROUP *имя_роли*

IN GROUP — устаревшее написание предложения IN ROLE.

ROLE *имя_роли*

В предложении ROLE перечисляются одна или несколько существующих ролей, которые автоматически становятся членами создаваемой роли. (По сути таким образом новая роль становится «группой».)

ADMIN *имя_роли*

Предложение ADMIN подобно ROLE, но перечисленные в нём роли включаются в новую роль с атрибутом WITH ADMIN OPTION, что даёт им право включать в эту роль другие роли.

USER *имя_роли*

Предложение USER является устаревшим написанием предложения ROLE.

SYSID *uid*

Предложение SYSID игнорируется, но принимается для обратной совместимости.

DROP ROLE — удалить роль в базе данных

```
DROP ROLE [ IF EXISTS ] имя [, ...]
```

SET ROLE — установить идентификатор текущего пользователя в рамках сеанса

```
SET [ SESSION | LOCAL ] ROLE имя_роли
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

SET SESSION AUTHORIZATION — установить идентификатор пользователя сеанса и идентификатор текущего пользователя в рамках сеанса

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION имя_пользователя
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

Примеры

1) Создание роли, для которой разрешён вход, но не задан пароль:

```
CREATE ROLE jonathan LOGIN;
```

2) Создание роли с паролем:

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

3) Создание роли с паролем, действующим до конца 2004 г., то есть пароль перестаёт действовать в первую же секунду 2005 г.

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

4) Создание роли, которая может создавать базы данных и управлять ролями:

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

5) Изменение пароля роли:

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

6) Удаление пароля роли:

```
ALTER ROLE davide WITH PASSWORD NULL;
```

7) Изменение срока действия пароля (в частности, определяется, что пароль должен перестать действовать в полдень 4 мая 2015 г. в часовом поясе UTC+1):

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

8) Установка бесконечного срока действия пароля:

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

9) Наделение роли правами на создание других ролей и новых баз данных:

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

10) Удаление роли:

```
DROP ROLE jonathan;
```

Привилегии (права)

Когда создаётся любой объект, ему назначается владелец. Обычно владелец — это та роль, которая запустила оператор создания данного объекта. Для большинства видов объектов, начальное состояние таково, что делать что-либо с объектом может только владелец (или суперпользователь). Чтобы разрешить другим ролям использовать его, им должны быть предоставлены привилегии.

Эти привилегии применяются к отдельному объекту, в зависимости от типа объёкта (таблица, функция и т.д.)

Объекту может быть назначен новый владелец с помощью команды ALTER для соответствующего вида объекта, например, ALTER TABLE. Суперпользователи могут делать это всегда; обычные роли могут делать это только если они одновременно и являются текущим владельцем данного объекта (или членом роли текущего владельца) и членом роли нового владельца.

Список привилегий

SELECT

Позволяет выполнять SELECT для любого столбца или перечисленных столбцов в заданной таблице, представлении или последовательности. Также позволяет выполнять COPY TO. Помимо того, это право требуется для обращения к существующим значениям столбцов в UPDATE или DELETE.

INSERT

Позволяет вставлять строки в заданную таблицу с помощью INSERT. Если право ограничивается несколькими столбцами, только их значение можно будет задать в команде INSERT (другие столбцы получат значения по умолчанию). Также позволяет выполнять COPY FROM.

UPDATE

Позволяет изменять (с помощью UPDATE) данные во всех, либо только перечисленных, столбцах в заданной таблице.

DELETE

Позволяет удалять строки из заданной таблицы с помощью DELETE.

TRUNCATE

Позволяет опустошить заданную таблицу с помощью TRUNCATE.

REFERENCES

Позволяет создавать ограничение внешнего ключа, ссылающееся на определённую таблицу либо на определённые столбцы таблицы.

TRIGGER

Позволяет создавать триггеры в заданной таблице.

CREATE

Для баз данных это право позволяет создавать схемы и публикации в заданной базе.

CONNECT

Позволяет пользователю подключаться к указанной базе данных.

TEMPORARY

TEMP

Позволяет создавать временные таблицы в заданной базе данных.

EXECUTE

Позволяет выполнять заданную функцию или процедуру и применять любые определённые поверх неё операторы. Это единственный тип прав, применимый к функциям и процедурам.

USAGE

Для процедурных языков это право позволяет создавать функции на заданном языке. Это единственный тип прав, применимый к процедурным языкам.

ALL PRIVILEGES

Даёт целевой роли все права сразу.

Синтаксис

GRANT — определить права доступа

Для таблицы

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
[, ...] | ALL [PRIVILEGES] }
ON { [TABLE] *имя_таблицы* [, ...]
| ALL TABLES IN SCHEMA *имя_схемы* [, ...] }
TO *указание_роли* [, ...] [WITH GRANT OPTION]

Для столбца

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } (*имя_столбца* [, ...])
[, ...] | ALL [PRIVILEGES] (*имя_столбца* [, ...]) }
ON [TABLE] *имя_таблицы* [, ...]
TO *указание_роли* [, ...] [WITH GRANT OPTION]

Для функции или процедуры

GRANT { EXECUTE | ALL [PRIVILEGES] }
ON { { FUNCTION | PROCEDURE | ROUTINE } *имя_подпрограммы* [([[*режим_аргумента*] [*имя_аргумента*] *тип_аргумента* [, ...]])] [, ...]
| ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA *имя_схемы* [, ...] }
TO *указание_роли* [, ...] [WITH GRANT OPTION]

Здесь *указание_роли*:

[GROUP] *имя_роли*
| PUBLIC
| CURRENT_USER
| SESSION_USER

Для назначения роли другой роли (пользователю)

GRANT *имя_роли* [, ...] TO *имя_роли* [, ...] [WITH ADMIN OPTION]

REVOKE — отозвать права доступа

Для таблицы

```
REVOKE [ GRANT OPTION FOR ]
  { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
  [, ...] | ALL [ PRIVILEGES ] }
  ON { [ TABLE ] имя_таблицы [, ...]
     | ALL TABLES IN SCHEMA имя_схемы [, ...] }
  FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]
```

Для столбца

```
REVOKE [ GRANT OPTION FOR ]
  { { SELECT | INSERT | UPDATE | REFERENCES } ( имя_столбца [, ...] )
  [, ...] | ALL [ PRIVILEGES ] ( имя_столбца [, ...] ) }
  ON [ TABLE ] имя_таблицы [, ...]
  FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]
```

Для функции или процедуры

```
REVOKE [ GRANT OPTION FOR ]
  { EXECUTE | ALL [ PRIVILEGES ] }
  ON { { FUNCTION | PROCEDURE | ROUTINE } имя_функции [ ( [ [ режим_аргумента ] [ имя_аргумента
] тип_аргумента [, ...] ] ) ] [, ...]
     | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA имя_схемы [, ...] }
  FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]
```

Для роли

```
REVOKE [ ADMIN OPTION FOR ]
  имя_роли [, ...] FROM имя_роли [, ...]
  [ CASCADE | RESTRICT ]
```

ALTER DEFAULT PRIVILEGES — определить права доступа по умолчанию

```
ALTER DEFAULT PRIVILEGES
  [ FOR { ROLE | USER } целевая_роль [, ...] ]
  [ IN SCHEMA имя_схемы [, ...] ]
  предложение_GRANT_или_REVOKE
```

Примеры

1) Следующая команда разрешает всем добавлять записи в таблицу `films`:

```
GRANT INSERT ON films TO PUBLIC;
```

2) Эта команда даёт пользователю `manuel` все права для представления `kinds`:

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

3) Включение в роль `admins` пользователя `joe`:

```
GRANT admins TO joe;
```

4) Лишение группы public права добавлять данные в таблицу `films`:

```
REVOKE INSERT ON films FROM PUBLIC;
```

5) Лишение пользователя `manuel` всех прав для представления `kinds`:

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

6) Исключение из членов роли `admins` пользователя `joe`:

```
REVOKE admins FROM joe;
```

7) Наделение всех правом SELECT для всех таблиц (и представлений), которые будут созданы в дальнейшем в схеме `myschema`, и наделение роли `webuser` правом INSERT для этих же таблиц:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT ON TABLES TO webuser;
```

8) Отмена предыдущих изменений с тем, чтобы для таблиц, создаваемых в будущем, были определены только обычные права, без дополнительных:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM webuser;
```

9) Лишение роли public права на выполнение (EXECUTE), которое обычно даётся для функций (для всех функций, которые будут созданы ролью `admin`):

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

Тема 9. Резервное копирование

Задание: просмотрите предложенный код, демонстрирующий возможности PostgreSQL и повторите на своём сервере со своей базой данных, по следующим вопросам:

- Команда COPY
- Утилита pg_dump
- Утилита pg_dump - формат custom
- Утилита pg_dump - формат directory
- Утилита pg_dumpall
- Влияние политик защиты строк

Отчет по лабораторной работе должен содержать**:**

1. Фамилию и номер группы учащегося, задание
2. Краткое описание базы данных
3. Результаты выполнения операторов
4. Код

Демонстрация (пример для повторения)

Команда COPY

Создадим базу данных и таблицу.

```
α=> CREATE DATABASE db1;
CREATE DATABASE
α=> \c db1
You are now connected to database "db1" as user "student".
α=> CREATE TABLE t(
    id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    s text
);
CREATE TABLE
α=> INSERT INTO t(s) VALUES ('Привет, мир!'), (''), (NULL);
INSERT 0 3
```

```
α=> SELECT * FROM t;
 id |      s
----+--------------
  1 | Привет, мир!
  2 |
  3 |
(3 rows)
```

Вот что показывает команда COPY (выдаем на консоль, а не в файл):

```
α=> COPY t TO stdout;
1       Привет, мир!
2
3       \N
```

Видно, как различаются в выводе пустые строки и неопределенные значения.

Формат вывода настраивается достаточно гибко. Можно изменить разделитель, представление неопределенных значений и т. п. Например:

```
α=> COPY t TO stdout WITH (NULL '<NULL>', DELIMITER ',');
1,Привет\, мир!
2,
3,<NULL>
```

Обратите внимание, что символ-разделитель внутри строки был экранирован (символ для экранирования тоже настраивается).

Вместо таблицы можно указать произвольный запрос.

```
α=> COPY (SELECT * FROM t WHERE s IS NOT NULL) TO stdout;
1       Привет, мир!
2
```

Таким образом можно сохранить результат запроса, данные представления и т. п.

Команда поддерживает вывод в формате CSV, который поддерживается множеством программ.

```
α=> COPY t TO stdout WITH (FORMAT CSV);
1,"Привет, мир!"
2,""
3,
```

Аналогично работает и ввод данных из файла или с консоли:

```
α=> TRUNCATE TABLE t;
TRUNCATE TABLE
α=> COPY t FROM stdin;
1       Привет, мир!
2
3       \N
\.
COPY 3
```

При вводе с консоли требуется маркер конца файла - обратная косая черта с точкой. В обычном файле он не нужен.

Все параметры при вводе должны совпадать с теми, что были указаны при выводе.

Вот что загрузилось в таблицу (для наглядности настроим в psql вывод неопределенных значений):

```
α=> \pset null '\\N'
Null display is "\N".
α=> SELECT * FROM t;
 id |      s
----+-------------
  1 | Привет, мир!
  2 |
  3 | \N
(3 rows)
```

Утилита pg_dump

При запуске без дополнительных параметров утилита pg_dump выдает команды SQL, создающие все объекты в базе данных:

```
student$ pg_dump -d db1
--
-- PostgreSQL database dump
```

```
--

-- Dumped from database version 10.4 (Ubuntu 10.4-2.pgdg16.04+1)
-- Dumped by pg_dump version 10.4 (Ubuntu 10.4-2.pgdg16.04+1)

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET client_min_messages = warning;
SET row_security = off;

--
-- Name: plpgsql; Type: EXTENSION; Schema: -; Owner:
--

CREATE EXTENSION IF NOT EXISTS plpgsql WITH SCHEMA pg_catalog;


--
-- Name: EXTENSION plpgsql; Type: COMMENT; Schema: -; Owner:
--

COMMENT ON EXTENSION plpgsql IS 'PL/pgSQL procedural language';


SET default_tablespace = '';

SET default_with_oids = false;

--
-- Name: t; Type: TABLE; Schema: public; Owner: student
--

CREATE TABLE public.t (
    id integer NOT NULL,
    s text
);


ALTER TABLE public.t OWNER TO student;

--
-- Name: t_id_seq; Type: SEQUENCE; Schema: public; Owner: student
--

ALTER TABLE public.t ALTER COLUMN id ADD GENERATED ALWAYS AS IDENTITY (
    SEQUENCE NAME public.t_id_seq
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1
);


--
-- Data for Name: t; Type: TABLE DATA; Schema: public; Owner: student
--

COPY public.t (id, s) FROM stdin;
```

```
1       Привет, мир!
2
3       \N
\.


--
-- Name: t_id_seq; Type: SEQUENCE SET; Schema: public; Owner: student
--

SELECT pg_catalog.setval('public.t_id_seq', 3, true);


--
-- Name: t t_pkey; Type: CONSTRAINT; Schema: public; Owner: student
--

ALTER TABLE ONLY public.t
    ADD CONSTRAINT t_pkey PRIMARY KEY (id);


--
-- PostgreSQL database dump complete
--
```

Видно, что pg_dump создал таблицу t и заполнил ее с помощью уже рассмотренной нами команды COPY. Ключ --column-inserts позволяет использовать команды INSERT, но загрузка будет работать существенно дольше.

Рассмотрим некоторые полезные ключи.
Могут пригодиться при восстановлении копии на системе с другим набором ролей:

- -O, --no-owner - не генерировать команды для установки владельца объектов;
- -x, --no-acl - не генерировать команды для установки привилегий.

Полезны для выгрузки и загрузки данных частями:

- -s, --schema-only - выгрузить только определения объектов без данных;
- -a, --data-only - выгрузить только данные, без создания объектов.

Удобны, если восстанавливать копию на системе, в которой уже есть данные (и наоборот, на чистой системе):

- -c, --clean - генерировать команды DROP для создаваемых объектов;
- -C, --create - генерировать команды создания БД и подключения к ней.

Важный момент: в выгрузку попадают и изменения, сделанные в шаблонной БД template1. Поэтому восстанавливать резервную копию лучше на базе данных, созданной из template0. При использовании ключа --create это учитывается автоматически:

```
student$ pg_dump --create -d db1 | grep 'CREATE DATABASE'
CREATE DATABASE db1 WITH TEMPLATE = template0 ENCODING = 'UTF8' LC_COLLATE =
'en_US.UTF-8' LC_CTYPE = 'en_US.UTF-8';
```

Существуют ключи для выбора объектов, которые должны попасть в резервную копию:

- -n, --schema - шаблон для имен схем;
- -t, --table - шаблон для имен таблиц.

И наоборот, включить в копию все, кроме указанного:

- -N, --exclude-schema - шаблон для имен схем;
- -T, --exclude-table - шаблон для имен таблиц.

Например, восстановим таблицу t в другой базе данных на другом сервере.

```
student$ psql -p 5433
β=> CREATE DATABASE db2;
```

```
CREATE DATABASE
```

```
student$ pg_dump --table=t -d db1 | psql -p 5433 -d db2
SET
SET
SET
SET
SET
 set_config
------------

(1 row)

SET
SET
SET
SET
SET
CREATE TABLE
ALTER TABLE
ALTER TABLE
COPY 3
 setval
--------
      3
(1 row)

ALTER TABLE
```

```
β=> \c db2
You are now connected to database "db2" as user "student".
β=> SELECT * FROM t;
 id |      s
----+--------------
  1 | Привет, мир!
  2 |
  3 |
(3 rows)
```

Утилита pg_dump - формат custom

Серьезное ограничение обычного формата (plain) состоит в том, что выбирать объекты нужно в момент выгрузки. Формат custom позволяет сначала сделать полную копию, а выбирать объекты уже при загрузке.

```
student$ pg_dump --format=custom -d db1 -f /home/student/db1.custom
```

Для восстановления объектов из такой копии предназначена утилита pg_restore. Повторим восстановление таблицы t:

```
β=> DROP TABLE t;
DROP TABLE
student$ pg_restore --table=t -p 5433 -d db2 /home/student/db1.custom
```

Формат резервной копии указывать не обязательно - утилита распознает его сама.

Утилита pg_restore понимает те же ключи для фильтрации объектов, что и pg_dump, и даже больше:

- -I, --index - загрузить определенные индексы;
- -P, --function - загрузить определенные функции;

- -T, --trigger - загрузить определенные триггеры.

```
β=> SELECT * FROM t;
 id |      s
----+--------------
  1 | Привет, мир!
  2 |
  3 |
(3 rows)
```

Еще один пример: восстановим целиком исходную базу данных db1 на другом сервере.

```
student$ pg_restore --create -p 5433 -d postgres /home/student/db1.custom
pg_restore: [archiver (db)] Error while PROCESSING TOC:
pg_restore: [archiver (db)] Error from TOC entry 2878; 0 0 COMMENT EXTENSION
plpgsql
pg_restore: [archiver (db)] could not execute query: ERROR:  must be owner of
extension plpgsql
    Command  was:  COMMENT  ON  EXTENSION  plpgsql  IS  'PL/pgSQL  procedural
language';


WARNING: errors ignored on restore: 1
```

Ошибка "must be owner of extension plpgsql" возникает из-за того, что владельцем расширения является postgres. В данном случае эта ошибка не приводит к проблеме (разумеется, все сообщения надо анализировать).
Здесь мы указали БД postgres, но могли указать любую - утилита сама создаст нужную БД и тут же переключится в нее.

Проверим:

```
β=> \c db1
You are now connected to database "db1" as user "student".
β=> SELECT * FROM t;
 id |      s
----+--------------
  1 | Привет, мир!
  2 |
  3 |
(3 rows)
```

Резервную копию в обычном (plain) формате при необходимости можно изменить в текстовом редакторе. Резервная копия формата custom хранится в двоичном виде, но и для нее доступны более широкие возможности фильтрации объектов, чем рассмотренные ключи. Утилита pg_restore может сформировать список объектов - оглавление резервной копии:

```
student$ pg_restore --list /home/student/db1.custom
;
; Archive created at 2018-06-13 19:58:37 MSK
;     dbname: db1
;     TOC Entries: 13
;     Compression: -1
;     Dump Version: 1.13-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 10.4 (Ubuntu 10.4-2.pgdg16.04+1)
```

```
;     Dumped by pg_dump version: 10.4 (Ubuntu 10.4-2.pgdg16.04+1)
;
;
; Selected TOC Entries:
;
2876; 1262 16386 DATABASE - db1 student
3; 2615 2200 SCHEMA - public postgres
2877; 0 0 COMMENT - SCHEMA public postgres
1; 3079 12998 EXTENSION - plpgsql
2878; 0 0 COMMENT - EXTENSION plpgsql
197; 1259 16389 TABLE public t student
196; 1259 16387 SEQUENCE public t_id_seq student
2870; 0 16389 TABLE DATA public t student
2879; 0 0 SEQUENCE SET public t_id_seq student
2747; 2606 16396 CONSTRAINT public t t_pkey student
```

Такой список можно записать в файл, отредактировать и использовать его для восстановления с помощью ключа --use-list.

Утилита pg_dump - формат directory

Формат directory интересен тем, что позволяет выгружать данные в несколько параллельных потоков.

```
student$ pg_dump --format=directory --jobs=2 -d db1 -f /home/student/db1.directory
```

При этом гарантируется согласованность данных: все параллельные потоки будут использовать один и тот же снимок данных.

Заглянем внутрь каталога:

```
student$ ls -l /home/student/db1.directory
total 8
-rw-r--r-- 1 student student   60 июн 13 19:58 2870.dat.gz
-rw-r--r-- 1 student student 2666 июн 13 19:58 toc.dat
```

В нем находится файл оглавления и по одному файлу на каждый выгружаемый объект (у нас он всего один):

```
student$ zcat /home/student/db1.directory/2870.dat.gz
1       Привет, мир!
2
3       \N
\.
```

Восстановление из резервной копии:

```
β=> \q
student$ pg_restore --clean --create --jobs=2 -p 5433 -d postgres /home/student/db1.custom
pg_restore: [archiver (db)] Error while PROCESSING TOC:
pg_restore: [archiver (db)] Error from TOC entry 2878; 0 0 COMMENT EXTENSION plpgsql
pg_restore: [archiver (db)] could not execute query: ERROR:  must be owner of extension plpgsql
    Command was: COMMENT ON EXTENSION plpgsql IS 'PL/pgSQL procedural language';


WARNING: errors ignored on restore: 1
```

Здесь мы предварительно отключились от базы данных db1 и добавили ключ --clean, который генерирует команду удаления БД.

Утилита pg_dumpall

Утилита pg_dump годится для выгрузки одной базы данных, но никогда не выгружает общие объекты кластера БД, такие, как роли и табличные пространства. Чтобы сделать полную копию кластера, нужна утилита pg_dumpall.

```
student$ pg_dumpall --clean -U postgres -f /home/student/alpha.sql
```

Утилиты pg_dumpall, pg_dump и pg_restore не требуют каких-то отдельных привилегий, но у выполняющей их роли должны быть привилегии на чтение (создание) всех затронутых объектов. Утилитой pg_dump может, например, пользоваться владелец базы данных. Но поскольку для копирования кластера надо иметь доступ ко всем БД, мы выполняем pg_dumpall под суперпользовательской ролью.

В копию кластера дополнительно попадают такие команды, как:

```
student$ grep 'ROLE' alpha.sql
DROP ROLE postgres;
DROP ROLE student;
CREATE ROLE postgres;
ALTER ROLE postgres WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN
REPLICATION BYPASSRLS;
CREATE ROLE student;
ALTER ROLE student WITH NOSUPERUSER INHERIT CREATEROLE CREATEDB LOGIN
REPLICATION NOBYPASSRLS PASSWORD 'md550d9482e20934ce6df0bf28941f885bc';
```

Восстановление выполняется с помощью psql - никакой другой формат не поддерживается.

```
student$ psql -p 5433 -U postgres -f alpha.sql
SET
SET
SET
DROP DATABASE
DROP DATABASE
psql:alpha.sql:24: ERROR:  current user cannot be dropped
psql:alpha.sql:25: ERROR:  role "student" cannot be dropped because some
objects depend on it
DETAIL:  owner of database db2
1 object in database db2
psql:alpha.sql:32: ERROR:  role "postgres" already exists
ALTER ROLE
psql:alpha.sql:34: ERROR:  role "student" already exists
ALTER ROLE
CREATE DATABASE
CREATE DATABASE
REVOKE
GRANT
You are now connected to database "db1" as user "postgres".
SET
SET
SET
SET
SET
SET
 set_config
------------

(1 row)

SET
SET
SET
CREATE EXTENSION
```

```
COMMENT
SET
SET
CREATE TABLE
ALTER TABLE
ALTER TABLE
COPY 3
 setval
--------
      3
(1 row)

ALTER TABLE
You are now connected to database "postgres" as user "postgres".
SET
SET
SET
SET
SET
SET
 set_config
------------

(1 row)

SET
SET
SET
COMMENT
CREATE EXTENSION
COMMENT
You are now connected to database "student" as user "postgres".
SET
SET
SET
SET
SET
SET
 set_config
------------

(1 row)

SET
SET
SET
CREATE EXTENSION
COMMENT
You are now connected to database "template1" as user "postgres".
SET
SET
SET
SET
SET
SET
 set_config
------------

(1 row)

SET
SET
SET
COMMENT
```

```
CREATE EXTENSION
COMMENT
```
В процессе восстановления могут возникать ошибки из-за существующих объектов - в данном случае это нормально и не мешает процессу.

```
student$ psql -p 5433
β=> SELECT datname FROM pg_database;
  datname
-----------
 postgres
 template0
 db2
 db1
 student
 template1
(6 rows)

β=> \c db1
You are now connected to database "db1" as user "student".
β=> SELECT * FROM t;
 id |      s
----+--------------
  1 | Привет, мир!
  2 |
  3 |
(3 rows)
```

Влияние политик защиты строк

Если на таблицах определены политики защиты строк, то есть опасность выгрузить неполные данные и даже не узнать об этом. Чтобы этого не произошло, перед выполнением команды COPY можно установить параметр row_security в значение off - в этом случае применение политики приведет к явной ошибке.
Простой пример. Настроим политику так, чтобы не выводились пустые строки, и включим ее для владельца таблицы:

```
α=> CREATE POLICY t_s_not_null ON t USING (s IS NOT NULL);
CREATE POLICY
α=> ALTER TABLE t ENABLE ROW LEVEL SECURITY;
ALTER TABLE
α=> ALTER TABLE t FORCE ROW LEVEL SECURITY;
ALTER TABLE
```

Теперь запрос покажет только две строки:

```
α=> COPY t TO stdout;
1       Привет, мир!
2
```

Но с параметром, установленным в off, будет зафиксирована ошибка:

```
α=> SET row_security = off;
SET
α=> COPY t TO stdout;
ERROR:  query would be affected by row-level security policy for table "t"
HINT:  To disable the policy for the table's owner, use ALTER TABLE NO FORCE
ROW LEVEL SECURITY.
```

Утилиты pg_dump и pg_dumpall автоматически используют этот параметр, так что дополнительные действия предпринимать не нужно:

```
student$ pg_dump -d db1 > /dev/null
pg_dump: [archiver (db)] query failed: ERROR:  query would be affected by row-level security policy for table "t"
HINT:  To disable the policy for the table's owner, use ALTER TABLE NO FORCE ROW LEVEL SECURITY.
pg_dump: [archiver (db)] query was: COPY public.t (id, s) TO stdout;
```

Базовая резервная копия

Создадим базу данных и таблицу.

```
α=> CREATE DATABASE backup_base;
CREATE DATABASE
α=> \c backup_base
You are now connected to database "backup_base" as user "student".
α=> CREATE TABLE t(s text);
CREATE TABLE
α=> INSERT INTO t VALUES ('Привет, мир!');
INSERT 0 1
```

В PostgreSQL 10 значения параметров по умолчанию позволяют сразу использовать протокол репликации:

```
α=> SELECT name, setting
FROM pg_settings
WHERE name IN ('wal_level','max_wal_senders','max_replication_slots');
         name          | setting
-----------------------+---------
 max_replication_slots | 10
 max_wal_senders       | 10
 wal_level             | replica
(3 rows)
```

В более ранних версиях аналогичные значения пришлось бы установить самостоятельно.

Разрешение на локальное подключение по протоколу репликации в pg_hba.conf также прописано по умолчанию, начиная с версии 10 (хотя это и зависит от конкретной пакетной сборки):

```
postgres$ egrep '^[^#].*replication' /etc/postgresql/10/alpha/pg_hba.conf
local   replication     all                                     trust
host    replication     all             127.0.0.1/32            md5
host    replication     all             ::1/128                 md5
```

Чтобы утилита pg_basebackup могла подключиться к серверу под ролью student, эта роль должна иметь атрибут REPLICATION:

```
α=> \du student
                          List of roles
 Role name |             Attributes              | Member of
-----------+-------------------------------------+-----------
 student   | Create role, Create DB, Replication | {}
```

Выполним команду pg_basebackup от имени postgres под ролью student.

В нашем случае и сервер-источник, и резервная копия будут располагаться на одном сервере. Поскольку мы собираемся тут же развернуть новый сервер, выбираем формат plain, а в качестве каталога для сохранения используем PGDATA целевого сервера.

Если бы мы использовали табличные пространства, дополнительно пришлось бы указать для них другие пути в ключе --tablespace-map, но в данном случае этого не трубуется.

```
postgres$ rm -rf /var/lib/postgresql/10/beta/*
postgres$ pg_basebackup -U student --pgdata=/var/lib/postgresql/10/beta
```

По умолчанию в начале копирования выполняется "протяженная" контрольная точка в соответствии с обычной настройкой:

```
α=> SHOW checkpoint_completion_target;
 checkpoint_completion_target
------------------------------
 0.5
(1 row)
```

Это может занять существенное время: если контрольные точки выполняются по расписанию, то соответствующую долю от

```
α=> SHOW checkpoint_timeout;
 checkpoint_timeout
--------------------
 5min
(1 row)
```

Если требуется выполнить контрольную точку как можно быстрее, надо указать ключ --checkpoint=fast.

Сам резервный сервер уже предварительно собран и установлен.
Проверим содержимое каталога с данными, которое было записано pg_basebackup:

```
postgres$ ls -l /var/lib/postgresql/10/beta
total 80
-rw------- 1 postgres postgres  206 июн 13 19:58 backup_label
drwx------ 7 postgres postgres 4096 июн 13 19:58 base
drwx------ 2 postgres postgres 4096 июн 13 19:58 global
drwx------ 2 postgres postgres 4096 июн 13 19:58 pg_commit_ts
drwx------ 2 postgres postgres 4096 июн 13 19:58 pg_dynshmem
drwx------ 4 postgres postgres 4096 июн 13 19:58 pg_logical
drwx------ 4 postgres postgres 4096 июн 13 19:58 pg_multixact
drwx------ 2 postgres postgres 4096 июн 13 19:58 pg_notify
drwx------ 2 postgres postgres 4096 июн 13 19:58 pg_replslot
drwx------ 2 postgres postgres 4096 июн 13 19:58 pg_serial
drwx------ 2 postgres postgres 4096 июн 13 19:58 pg_snapshots
drwx------ 2 postgres postgres 4096 июн 13 19:58 pg_stat
drwx------ 2 postgres postgres 4096 июн 13 19:58 pg_stat_tmp
drwx------ 2 postgres postgres 4096 июн 13 19:58 pg_subtrans
drwx------ 2 postgres postgres 4096 июн 13 19:58 pg_tblspc
drwx------ 2 postgres postgres 4096 июн 13 19:58 pg_twophase
-rw------- 1 postgres postgres    3 июн 13 19:58 PG_VERSION
drwx------ 3 postgres postgres 4096 июн 13 19:58 pg_wal
drwx------ 2 postgres postgres 4096 июн 13 19:58 pg_xact
-rw------- 1 postgres postgres   88 июн 13 19:58 postgresql.auto.conf
```

Все необходимые файлы журнала находятся в каталоге pg_wal:

```
postgres$ ls -l /var/lib/postgresql/10/beta/pg_wal/
total 16388
-rw------- 1 postgres postgres 16777216 июн 13 19:58 000000010000000000000002
drwx------ 2 postgres postgres     4096 июн 13 19:58 archive_status
```

Восстановление из базовой резервной копии

Мы собираемся запустить сервер вместе с основным, поэтому в настройках необходимо поменять порт. Сделаем это в файле postgresql.auto.conf:

```
postgres$ echo 'port = 5433' >> /var/lib/postgresql/10/beta/postgresql.auto.conf
```

И можно запускать сервер.

```
student$ sudo pg_ctlcluster 10 beta start
```

Теперь оба сервера работают одновременно и независимо. Проверим:

```
student$ psql -p 5433 -d backup_base
β=> SELECT * FROM t;
      s
--------------
 Привет, мир!
(1 row)
```

Настройка непрерывной архивации

Архив будем хранить в каталоге /var/lib/postgresql/archive. Он должен быть доступен пользователю-владельцу PostgreSQL.

```
postgres$ mkdir /var/lib/postgresql/archive
```

В реальной практике архив может размещаться на отдельном сервере или дисковой системе с доступом по сети.

Включим режим архивирования и установим команду копирования заполненных сегментов журнала.

Выполняем команды от имени суперпользовательской роли:

```
student$ psql -U postgres -c "ALTER SYSTEM SET archive_mode = on"
ALTER SYSTEM
student$ psql -U postgres -c "ALTER SYSTEM SET archive_command = 'test ! -f /var/lib/postgresql/archive/%f && cp %p /var/lib/postgresql/archive/%f'"
ALTER SYSTEM
```

В archive_command мы сначала проверяем наличие файла с указанным именем в архиве, и копируем его только в случае отсутствия.

В archive_command можно указать произвольную команду, лишь бы она завершалась со статусом 0 только в случае успеха. Например, можно организовать сжатие архивируемых сегментов:

```
α=> -- SET archive_command = 'test ! -f /var/lib/postgresql/archive/%f && gzip <%p >/var/lib/postgresql/archive/%f'
```

В идеале команда архивирования должна выполнять sync, чтобы файл гарантированно попал в энергонезависимую память. Иначе при сбое он может пропасть из архива, а сервер может успеть стереть его из каталога pg_wal.

В общем случае удобно поместить всю необходимую логику архивирования в отдельный скрипт и вызываеть его:

```
α=> -- SET archive_command = 'archive.sh "%f" "%p"'
```

Для того, чтобы настройки вступили в силу, потребуется рестарт сервера.

```
α=> \q
student$ sudo pg_ctlcluster 10 alpha restart
```

Проверим работу настройки. Создадим базу и таблицу.

```
student$ psql
α=> CREATE DATABASE backup_archive;
```

```
CREATE DATABASE
α=> \c backup_archive
You are now connected to database "backup_archive" as user "student".
α=> CREATE TABLE t(s text);
CREATE TABLE
α=> INSERT INTO t VALUES ('Привет, мир!');
INSERT 0 1
```

Вот какой сегмент WAL используется сейчас:

```
α=> SELECT pg_walfile_name(pg_current_wal_lsn());
     pg_walfile_name
--------------------------
 000000010000000000000003
(1 row)
```

Обратите внимание: первые восемь цифр в имени файла - номер текущей ветви времени.

Чтобы заполнить файл, пришлось бы выполнить большое количество операций, но в тестовых целях проще принудительно переключить сегмент:

```
student$ psql -U postgres -c "SELECT pg_switch_wal()"
 pg_switch_wal
---------------
 0/3026220
(1 row)
α=> INSERT INTO t VALUES ('Доброе утро, страна!');
INSERT 0 1
```

Сегмент сменился:

```
α=> SELECT pg_walfile_name(pg_current_wal_lsn());
     pg_walfile_name
--------------------------
 000000010000000000000004
(1 row)
```

А предыдущий должен был попасть в архив. Проверим:

```
postgres$ ls -l /var/lib/postgresql/archive
total 16384
-rw------- 1 postgres postgres 16777216 июн 13 19:58 000000010000000000000003
```

Таким образом, архивация насторена и работает.

Базовая резервная копия

```
postgres$ rm -rf /var/lib/postgresql/10/beta/*
```

Поскольку мы используем архив, попросим pg_basebackup не добавлять файлы журнала к резервной копии:

```
postgres$ pg_basebackup -U student --wal-method=none --pgdata=/var/lib/postgresql/10/beta
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
```

Каталог pg_wal резервной копии пуст:

```
postgres$ ls -l /var/lib/postgresql/10/beta/pg_wal/
total 4
drwx------ 2 postgres postgres 4096 июн 13 19:59 archive_status
```

А в архиве прибавилось файлов:

```
postgres$ ls -l /var/lib/postgresql/archive
total 49156
```

```
-rw------- 1 postgres postgres 16777216 июн 13 19:58 000000010000000000000003
-rw------- 1 postgres postgres 16777216 июн 13 19:58 000000010000000000000004
-rw------- 1 postgres postgres 16777216 июн 13 19:59 000000010000000000000005
-rw-------   1   postgres   postgres                  302   июн   13   19:59
000000010000000000000005.00000024.backup
```

Заглянем в сгенерированный файл метки:

```
postgres$ cat /var/lib/postgresql/10/beta/backup_label
START WAL LOCATION: 0/5000024 (file 000000010000000000000005)
CHECKPOINT LOCATION: 0/5000058
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2018-06-13 19:58:59 MSK
LABEL: pg_basebackup base backup
```

Главная информация в этом файле - указание начальной точки для восстановления (строка START WAL LOCATION). Теоретически восстановление можно начать и с более ранней (но не более поздней) позиции, но это потребует больше времени.

Эту же строку надо принимать во внимание, определяя, какие файлы из архива еще нужны, а какие можно удалить.

Восстановление из базовой резервной копии

Изменим порт:

```
postgres$ echo 'port = 5433' >>
/var/lib/postgresql/10/beta/postgresql.auto.conf
```

Перенаправим непрерывное архивирование в другой каталог, чтобы не возникало коллизий с работающим первым сервером:

```
postgres$ mkdir /var/lib/postgresql/archive_beta
postgres$ sed -i 's/\/archive/\/archive_beta/g'
/var/lib/postgresql/10/beta/postgresql.auto.conf
```

Вот что получилось:

```
postgres$ cat /var/lib/postgresql/10/beta/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
archive_mode = 'on'
archive_command = 'test ! -f /var/lib/postgresql/archive_beta/%f && cp %p
/var/lib/postgresql/archive_beta/%f'
port = 5433
```

Теперь надо создать файл recovery.conf. В простейшем случае в него достаточно поместить команду восстановления, которая будет копировать указанный сегмент WAL обратно из архива по указанному пути:

```
postgres$ echo "restore_command = 'cp /var/lib/postgresql/archive/%f %p'"
>/var/lib/postgresql/10/beta/recovery.conf
```

Если в recovery.conf не указана целевая точка восстановления (один из параметров recovery_target*), то к базовой резервной копии будут применены записи WAL из всех файлов в архиве.

Следует иметь в виду, что самые последние изменения не попадут в архив, пока очередной сегмент WAL не будет заполнен (или пока не пройдет время, указанное в параметре archive_timeout).

```
α=> INSERT INTO t VALUES ('Еще не в архиве');
INSERT 0 1
```

Запускаем сервер.

```
student$ sudo pg_ctlcluster 10 beta start
```

Файл recovery.conf переименовался в recovery.done:

```
postgres$ ls -l /var/lib/postgresql/10/beta/recovery.*
-rw-r--r--  1  postgres  postgres  57  июн  13  19:59
/var/lib/postgresql/10/beta/recovery.done
```

Проверим, что восстановлено в таблице:

```
student$ psql -p 5433 -d backup_archive
β=> SELECT * FROM t;
          s
----------------------
 Привет, мир!
 Доброе утро, страна!
(2 rows)
```

Как и ожидалось, последняя строка не попала в архив и не восстановилась.

Что стало с ветвью времени после восстановления?

```
β=> SELECT pg_walfile_name(pg_current_wal_lsn());
     pg_walfile_name
--------------------------
 000000020000000000000006
(1 row)
```

Номер увеличился на единицу.

В каталоге pg_wal появился файл истории, соответствующий этой ветви:

```
postgres$ ls -l /var/lib/postgresql/10/beta/pg_wal/00000002.history
-rw-------  1  postgres  postgres  41  июн  13  19:59
/var/lib/postgresql/10/beta/pg_wal/00000002.history
```

В нем есть информация о "точках ветвления", через которые мы пришли в данную ветвь времени:

```
postgres$ cat /var/lib/postgresql/10/beta/pg_wal/00000002.history
1       0/6000000       no recovery target specified
```

Эти файлы PostgreSQL использует, когда мы указываем ветвь времени в recovery.conf. Поэтому файл истории подлежит архивации вместе с сегментами WAL, и удалять из архива их не надо:

```
postgres$ ls -l /var/lib/postgresql/archive_beta
total 4
-rw------- 1 postgres postgres 41 июн 13 19:59 00000002.history
```

Тема 10. Репликация

Задание: просмотрите предложенный код, демонстрирующий возможности PostgreSQL и повторите на своём сервере со своей базой данных, по следующим вопросам:

- Настройка репликации без архива
- Проверка репликации
- Мониторинг репликации
- Настройка для pg_rewind
- Настройка репликации без архива
- Проверка репликации
- Переход на реплику
- Возвращение в строй бывшего мастера

– Предварительная настройка
– Логическая репликация
– Конфликты
– Триггеры на подписчике
– Удаление подписки

Отчет по лабораторной работе должен содержать**:**

1. Фамилию и номер группы учащегося, задание
2. Краткое описание базы данных
3. Результаты выполнения операторов
4. Код

Демонстрация (пример для повторения)

Настройка репликации без архива

Поскольку в нашей конфигурации не будет архива журнала предзаписи, важно на всех этапах использовать слот репликации - иначе при определенной задержке мастер может успеть удалить необходимые сегменты и весь процесс придется повторять с самого начала.

Создаем слот:

```
α=> SELECT pg_create_physical_replication_slot('replica');
 pg_create_physical_replication_slot
-------------------------------------
 (replica,)
(1 row)
```

Вначале слот не инициализирован (restart_lsn пустой):

```
α=> SELECT * FROM pg_replication_slots \gx
-[ RECORD 1 ]-------+---------
slot_name           | replica
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | f
active_pid          |
xmin                |
catalog_xmin        |
restart_lsn         |
confirmed_flush_lsn |
```

Как мы помним из модуля "Резервное копирование", в версии 10+ все необходимые настройки есть по умолчанию:

- wal_level = replica;
- max_wal_senders
- разрешение на подключение в pg_hba.conf.

Создадим автономную резервную копию, используя созданный слот. Копию расположим в каталоге PGDATA будущей реплики.

```
postgres$ rm -rf /var/lib/postgresql/10/beta/*
postgres$ pg_basebackup -U student --pgdata=/var/lib/postgresql/10/beta -R --slot=replica
```

Возможность использовать слот появилась у pg_basebackup, начиная с версии 9.6. В более ранних версиях следовало бы заранее инициализировать слот, подключившись к нему утилитой pg_receive_xlog.

После выполнения резервной копии слот инициализировался, и мастер хранит все файлы журнала с начала копирования (restart_lsn).

```
α=> SELECT * FROM pg_replication_slots \gx
-[ RECORD 1 ]-------+----------
slot_name           | replica
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | f
active_pid          |
xmin                |
catalog_xmin        |
restart_lsn         | 0/8000000
confirmed_flush_lsn |
```

Мы собираемся запустить реплику одновременно с основным сервером на одной машине, поэтому в настройках необходимо поменять порт:

```
postgres$ echo 'port = 5433' >> /var/lib/postgresql/10/beta/postgresql.auto.conf
```

Чтобы реплика выполняла запросы (была "горячей"), необходимо выставить параметр hot_standby. Если этого не сделать, реплика будет "теплой" и не будет принимать подключения.

```
postgres$ echo 'hot_standby = on' >> /var/lib/postgresql/10/beta/postgresql.auto.conf
```

Файл recovery.conf был подготовлен утилитой pg_basebackup, и уже включает указание режима непрерывного восстановления (standby_mode) и слота репликации (primary_slot_name):

```
postgres$ cat /var/lib/postgresql/10/beta/recovery.conf
standby_mode = 'on'
primary_conninfo = 'user=student passfile=''/var/lib/postgresql/.pgpass'' host=''/var/run/postgresql'' port=5432 sslmode=prefer sslcompression=1 krbsrvname=postgres target_session_attrs=any'
primary_slot_name = 'replica'
```

Можно запускать реплику.

Журнальные записи, необходимые для восстановления согласованности, реплика получит от мастера по протоколу репликации. Далее она войдет в режим непрерывного восстановления, как указано в recovery.conf, и будет продолжать получать и проигрывать поток записей.

```
student$ sudo pg_ctlcluster 10 beta start
```

Посмотрим на процессы реплики.

```
postgres$ ps -o pid,command --ppid `head -n 1 /var/lib/postgresql/10/beta/postmaster.pid`
  PID COMMAND
 1427 postgres: 10/beta: startup process waiting for 000000010000000000000009
 1428 postgres: 10/beta: checkpointer process
 1429 postgres: 10/beta: writer process
 1430 postgres: 10/beta: stats collector process
 1431 postgres: 10/beta: wal receiver process
```

Процесс wal receiver принимает поток журнальных записей, процесс startup применяет изменения.

Процессы wal writer и autovacuum launcher отсутствуют.

И сравним с процессами мастера.

```
postgres$ ps -o pid,command --ppid `head -n 1 /var/lib/postgresql/10/alpha/postmaster.pid`
  PID COMMAND
 1035 postgres: 10/alpha: checkpointer process
 1036 postgres: 10/alpha: writer process
 1037 postgres: 10/alpha: wal writer process
 1038 postgres: 10/alpha: autovacuum launcher process
 1039 postgres: 10/alpha: stats collector process
 1040 postgres: 10/alpha: bgworker: logical replication launcher
 1090 postgres: 10/alpha: student student [local] idle
 1432 postgres: 10/alpha: wal sender process student [local] idle
```

Здесь добавился процесс wal sender, обслуживающий подключение по протоколу репликации.

Проверка репликации

Выполним несколько команд на мастере:

```
α=> CREATE DATABASE replica_physical;
CREATE DATABASE
α=> \c replica_physical;
You are now connected to database "replica_physical" as user "student".
α=> CREATE TABLE test(s text);
CREATE TABLE
α=> INSERT INTO test VALUES ('Привет, мир!');
INSERT 0 1
```

Проверим реплику:

```
student$ psql -p 5433 -d replica_physical
β=> SELECT * FROM test;
       s
--------------
 Привет, мир!
(1 row)
```

При этом изменения на реплике не допускаются:

```
β=> INSERT INTO test VALUES ('Replica');
ERROR:  cannot execute INSERT in a read-only transaction
```

Вообще реплику от мастера можно отличить с помощью функции:

```
β=> SELECT pg_is_in_recovery();
 pg_is_in_recovery
-------------------
 t
(1 row)
```

Мониторинг репликации

Состояние репликации можно смотреть в специальном представлении на мастере. Чтобы пользователь получил доступ к этой информации, ему должна быть выдана роль pg_read_all_stats (или он должен быть суперпользователем).

```
α=> \c - postgres
You are now connected to database "replica_physical" as user "postgres".
α=> GRANT pg_read_all_stats TO student;
GRANT ROLE
α=> \c - student
```

```
You are now connected to database "replica_physical" as user "student".
```

```
α=> SELECT * FROM pg_stat_replication \gx
-[ RECORD 1 ]----+------------------------------
pid              | 1432
usesysid         | 16384
usename          | student
application_name | walreceiver
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2018-06-13 19:59:13.044052+03
backend_xmin     |
state            | streaming
sent_lsn         | 0/9025620
write_lsn        | 0/9025620
flush_lsn        | 0/9025620
replay_lsn       | 0/9025620
write_lag        | 00:00:00.000199
flush_lag        | 00:00:00.001131
replay_lag       | 00:00:00.001184
sync_priority    | 0
sync_state       | async
```

Обратите внимание на поля *_lsn (и *_lag) - они показывают отставание реплики на разных этапах. Сейчас все позиции совпадают, отставание нулевое.

Теперь вставим в таблицу большое количество строк, чтобы увидеть репликацию в процессе работы.

```
α=> INSERT INTO test SELECT 'Just a line' FROM generate_series(1,1000000);
INSERT 0 1000000
α=> SELECT *, pg_current_wal_lsn() from pg_stat_replication \gx
-[ RECORD 1 ]------+------------------------------
pid                | 1432
usesysid           | 16384
usename            | student
application_name   | walreceiver
client_addr        |
client_hostname    |
client_port        | -1
backend_start      | 2018-06-13 19:59:13.044052+03
backend_xmin       |
state              | streaming
sent_lsn           | 0/D1299E0
write_lsn          | 0/D1299E0
flush_lsn          | 0/D1299E0
replay_lsn         | 0/C47A7FC
write_lag          | 00:00:00.135193
flush_lag          | 00:00:00.143305
replay_lag         | 00:00:00.743007
sync_priority      | 0
sync_state         | async
pg_current_wal_lsn | 0/D1299E0
```

Видно, что возникла небольшая задержка. Однако не следует рассчитывать на точность выше чем полсекунды, так как статистика обновляется примерно с такой частотой.

Проверим реплику:

```
β=> SELECT count(*) FROM test;
  count
---------
```

```
 1000001
(1 row)
```

Все строки успешно доехали.

И еще раз проверим состояние репликации:

```
α=> SELECT *, pg_current_wal_lsn() from pg_stat_replication \gx
-[ RECORD 1 ]------+------------------------------
pid                | 1432
usesysid           | 16384
usename            | student
application_name   | walreceiver
client_addr        |
client_hostname    |
client_port        | -1
backend_start      | 2018-06-13 19:59:13.044052+03
backend_xmin       |
state              | streaming
sent_lsn           | 0/D1299E0
write_lsn          | 0/D1299E0
flush_lsn          | 0/D1299E0
replay_lsn         | 0/D1299E0
write_lag          | 00:00:00.135193
flush_lag          | 00:00:00.143305
replay_lag         | 00:00:00.54367
sync_priority      | 0
sync_state         | async
pg_current_wal_lsn | 0/D1299E0
```

Все позиции выровнялись.

Настройка для pg_rewind

Мы планируем использовать утилиту pg_rewind, поэтому убедимся, что включены контрольные суммы на страницах данных:

```
student$ psql
α=> SHOW data_checksums;
 data_checksums
----------------
 on
(1 row)
```

Этот параметр служит только для информации; изменить его нельзя - подсчет контрольных сумм включается при инициализации кластера (включение "на лету" планируется, но появится не раньше версии 12).

И проверим, что параметр full_page_writes включен:

```
α=> SHOW full_page_writes;
 full_page_writes
------------------
 on
(1 row)
```

Настройка репликации без архива

Процесс настройки реплики аналогичен рассмотренному в предыдущей теме.

Слот:

```
α=> SELECT pg_create_physical_replication_slot('replica');
 pg_create_physical_replication_slot
```

```
-----------------------------------
 (replica,)
(1 row)
```

Создаем автономную резервную копию, используя созданный слот, в каталоге PGDATA будущей реплики.

```
postgres$ rm -rf /var/lib/postgresql/10/beta/*
postgres$ pg_basebackup -U student --pgdata=/var/lib/postgresql/10/beta -R --
slot=replica
```

Меняем порт и включаем режим горячего резерва:

```
postgres$ echo 'port = 5433' >>
/var/lib/postgresql/10/beta/postgresql.auto.conf
postgres$ echo 'hot_standby = on' >>
/var/lib/postgresql/10/beta/postgresql.auto.conf
```

Файл recovery.conf подготовлен утилитой pg_basebackup:

```
postgres$ cat /var/lib/postgresql/10/beta/recovery.conf
standby_mode = 'on'
primary_conninfo = 'user=student passfile=''/var/lib/postgresql/.pgpass''
host=''/var/run/postgresql'' port=5432 sslmode=prefer sslcompression=1
krbsrvname=postgres target_session_attrs=any'
primary_slot_name = 'replica'
```

Запускаем реплику.

```
student$ sudo pg_ctlcluster 10 beta start
```

Проверка репликации

Выполним несколько команд на мастере:

```
α=> CREATE DATABASE replica_switchover;
CREATE DATABASE
α=> \c replica_switchover;
You are now connected to database "replica_switchover" as user "student".
α=> CREATE TABLE test(s text);
CREATE TABLE
α=> INSERT INTO test VALUES ('Привет, мир!');
INSERT 0 1
```

Проверим реплику:

```
student$ psql -p 5433 -d replica_switchover
β=> SELECT * FROM test;
      s
--------------
 Привет, мир!
(1 row)
```

Переход на реплику

Сейчас второй сервер является репликой (находится в режиме восстановления):

```
β=> SELECT pg_is_in_recovery();
 pg_is_in_recovery
-------------------
```

```
 t
(1 row)
```

Повышаем реплику:

```
student$ sudo pg_ctlcluster 10 beta promote
```

Теперь бывшая реплика стала полноценным экземпляром.

```
β=> SELECT pg_is_in_recovery();
 pg_is_in_recovery
-------------------
 f
(1 row)
```

Мы можем изменять данные:

```
β=> INSERT INTO test VALUES ('Я - бывшая реплика (новый мастер).');
INSERT 0 1
```

Между тем первый сервер еще не выключен и тоже может изменять данные:

```
α=> INSERT INTO test VALUES ('Die hard');
INSERT 0 1
```

В реальности такой ситуации необходимо всячески избегать, поскольку теперь непонятно, какому серверу верить. Придется либо полностью потерять изменения на одном из серверов, либо придумывать, как объединить данные.

Наш выбор - потерять изменения, сделанные на первом сервере. Остановим его (аккуратно, с выполнением контрольной точки).

```
α=> \q
student$ sudo pg_ctlcluster 10 alpha stop
```

Возвращение в строй бывшего мастера

Создадим на втором сервере слот для будущей реплики.

```
β=> SELECT pg_create_physical_replication_slot('replica');
 pg_create_physical_replication_slot
-------------------------------------
 (replica,)
(1 row)
```

Для того, чтобы ввести в строй бывший мастер в качестве новой реплики, мы можем создать новую резервную копию, а можем воспользоваться утилитой pg_rewind, которая решает задачу быстрее.

В ключах утилиты надо указать каталог PGDATA целевого сервера и способ обращения к серверу-источнику: либо подключение от имени суперпользователя (если сервер работает), либо местоположение его каталога PGDATA (если он выключен).

```
postgres$                /usr/lib/postgresql/10/bin/pg_rewind                -D
/var/lib/postgresql/10/alpha --source-server='user=postgres port=5433' -P
connected to server
servers diverged at WAL location 0/F025744 on timeline 1
rewinding from last common checkpoint at 0/F001580 on timeline 1
reading source file list
reading target file list
reading WAL in target
need to copy 54 MB (total source directory size is 83 MB)
    0/55990 kB (0%) copied
```

```
55990/55990 kB (100%) copied

creating backup label and updating control file
syncing target data directory
Done!
```

В результате работы pg_rewind "откатывает" файлы данных на ближайшую контрольную точку до того момента, как пути серверов разошлись, а также создает файл backup_label, который обеспечивает применение нужных журналов для завершения восстановления. Заглянем в backup_label:

```
postgres$ cat /var/lib/postgresql/10/alpha/backup_label
START WAL LOCATION: 0/F001548 (file 000000010000000000000F)
CHECKPOINT LOCATION: 0/F001580
BACKUP METHOD: pg_rewind
BACKUP FROM: standby
START TIME: 2018-06-13 19:59:36 MSK
```

Файл postgresql.auto.conf был скопирован с нового мастера, его нужно исправить.

```
postgres$ echo 'hot_standby = on' > /var/lib/postgresql/10/alpha/postgresql.auto.conf
```

Чтобы бывший мастер после восстановления стал новой репликой, потребуется файл recovery.conf.

```
postgres$ echo $'standby_mode = \'on\'' > /var/lib/postgresql/10/alpha/recovery.conf
postgres$ echo $'primary_conninfo = \'user=postgres port=5433\'' >> /var/lib/postgresql/10/alpha/recovery.conf
postgres$ echo $'primary_slot_name = \'replica\'' >> /var/lib/postgresql/10/alpha/recovery.conf
```

Вот что вышло:

```
postgres$ cat /var/lib/postgresql/10/alpha/recovery.conf
standby_mode = 'on'
primary_conninfo = 'user=postgres port=5433'
primary_slot_name = 'replica'
```

Можно стартовать новую реплику.

```
student$ sudo pg_ctlcluster 10 alpha start
```

Слот репликации инициализировался и используется:

```
β=> SELECT * FROM pg_replication_slots \gx
-[ RECORD 1 ]-------+----------
slot_name           | replica
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | t
active_pid          | 3143
xmin                |
catalog_xmin        |
restart_lsn         | 0/F04A1CC
confirmed_flush_lsn |
```

Данные, измененные на новом мастере, получены:

```
student$ psql -p 5432 -d replica_switchover
α=> SELECT * FROM test;
                  s
-----------------------------------
 Привет, мир!
 Я - бывшая реплика (новый мастер).
(2 rows)
```

Проверим еще:

```
β=> INSERT INTO test VALUES ('Еще строка с нового мастера.');
INSERT 0 1
α=> SELECT * FROM test;
                  s
-----------------------------------
 Привет, мир!
 Я - бывшая реплика (новый мастер).
 Еще строка с нового мастера.
(3 rows)
```

Таким образом, два сервера поменялись ролями.

Предварительная настройка

Создадим базу данных.

```
α=> CREATE DATABASE replica_logical;
CREATE DATABASE
α=> \c replica_logical
You are now connected to database "replica_logical" as user "student".
```

Сначала создадим второй сервер как копию первого. Для этого выполним резервное копирование в каталог PGDATA второго сервера.

```
postgres$ rm -rf /var/lib/postgresql/10/beta/*
postgres$ pg_basebackup -U student --pgdata=/var/lib/postgresql/10/beta
```

Меняем порт и запускаем сервер:

```
postgres$ echo 'port = 5433' >> /var/lib/postgresql/10/beta/postgresql.auto.conf
student$ sudo pg_ctlcluster 10 beta start
```

Теперь на первом сервере создадим таблицу и заполним ее данными.

```
α=> CREATE TABLE test(id serial PRIMARY KEY, descr text);
CREATE TABLE
α=> INSERT INTO test(descr) VALUES ('Раз'), ('Два'), ('Три');
INSERT 0 3
```

Логическая репликация

Мы хотим настроить между серверами логическую репликацию таблицы test. Для этого нам понадобится изменить уровень журнала.

```
student$ psql -U postgres -c "ALTER SYSTEM SET wal_level = logical"
ALTER SYSTEM
student$ sudo pg_ctlcluster 10 alpha restart
```

На втором сервере таблицы test нет. Поскольку команды DDL не реплицируются, таблицу необходимо создать вручную. При этом таблица подписчика может содержать и дополнительные столбцы, если это необъодимо.

```
student$ psql -p 5433 -d replica_logical
β=> CREATE TABLE test(id serial PRIMARY KEY, descr text, additional text);
CREATE TABLE
```

На первом сервере создаем публикацию для таблицы test. Публикация относится к конкретной базе данных; в нее можно включить и несколько таблиц, а можно даже все таблицы сразу (FOR ALL TABLES).

```
student$ psql -d replica_logical
α=> CREATE PUBLICATION test_pub FOR TABLE test;
CREATE PUBLICATION
α=> \dRp+
                Publication test_pub
  Owner  | All tables | Inserts | Updates | Deletes
---------+------------+---------+---------+---------
 student | f          | t       | t       | t
Tables:
    "public.test"
```

На втором сервере подписываемся на публикацию. При этом на публикующем сервере будет создан слот логической репликации.

Подписку может создать только суперпользователь. А роль для подключения к публикующему серверу долна иметь атрибуты REPLICAION и LOGIN, и должна иметь право чтения публикуемых таблиц - роль student подходит под эти требования.

```
β=> \c - postgres
You are now connected to database "replica_logical" as user "postgres".
β=> CREATE SUBSCRIPTION test_sub
CONNECTION 'port=5432 user=student dbname=replica_logical'
PUBLICATION test_pub;

NOTICE:  created replication slot "test_sub" on publisher
CREATE SUBSCRIPTION
```

```
β=> \c - student
You are now connected to database "replica_logical" as user "student".
β=> \dRs
            List of subscriptions
   Name    |  Owner   | Enabled | Publication
-----------+----------+---------+-------------
 test_sub | postgres | t       | {test_pub}
(1 row)
```

По умолчанию данные сначала синхронизируются между серверами, и только после этого запускается процесс репликации. Это выполняется "бесшовно" с гарантией того, что никакие изменения не будут потеряны.

```
β=> SELECT * FROM test;
 id | descr | additional
----+-------+------------
  1 | Раз   |
  2 | Два   |
  3 | Три   |
(3 rows)
```

Проверим, как работает репликация изменений.

```
α=> INSERT INTO test(descr) VALUES ('Четыре');
INSERT 0 1
```

```
β=> SELECT * FROM test;
 id | descr  | additional
----+--------+------------
  1 | Раз    |
  2 | Два    |
  3 | Три    |
  4 | Четыре |
(4 rows)
```

Состояние подписки можно посмотреть в представлении:

```
β=> SELECT * FROM pg_stat_subscription \gx
-[ RECORD 1 ]---------+------------------------------
subid                 | 41010
subname               | test_sub
pid                   | 4004
relid                 |
received_lsn          | 0/11030C6C
last_msg_send_time    | 2018-06-13 19:59:55.353334+03
last_msg_receipt_time | 2018-06-13 19:59:55.353694+03
latest_end_lsn        | 0/11030C6C
latest_end_time       | 2018-06-13 19:59:55.353334+03
```

- received_lsn - позиция в журнале, до которой получены изменения;
- latest_end_lsn - позиция в журнале, подтвержденная процессу wal sender.

К процессам сервера добавился logical replication worker (его номер указан в pg_stat_subscription.pid):

```
postgres$ ps -o pid,command --ppid `head -n 1 /var/lib/postgresql/10/beta/postmaster.pid`
  PID COMMAND
 3726 postgres: 10/beta: checkpointer process
 3727 postgres: 10/beta: writer process
 3728 postgres: 10/beta: wal writer process
 3729 postgres: 10/beta: autovacuum launcher process
 3730 postgres: 10/beta: stats collector process
 3731 postgres: 10/beta: bgworker: logical replication launcher
 4004 postgres: 10/beta: bgworker: logical replication worker for subscription 41010
 4023 postgres: 10/beta: student replica_logical [local] idle
```

Заметим, что последовательности не реплицируются. На втором сервере создалась своя собственная последовательность:

```
β=> INSERT INTO test(descr) VALUES ('Пять - локально');
ERROR:  duplicate key value violates unique constraint "test_pkey"
DETAIL:  Key (id)=(1) already exists.
```

А вот так получится:

```
β=> INSERT INTO test VALUES (5, 'Пять - локально');
INSERT 0 1
```

Конфликты

Что произойдет, если значение с таким же ключом (5) появится на публикующем сервере?

```
α=> INSERT INTO test(descr) VALUES ('Пять');
INSERT 0 1
α=> INSERT INTO test(descr) VALUES ('Шесть');
INSERT 0 1
```

При репликации возникнет конфликт, и она будет приостановлена.

```
β=> SELECT * FROM test;
 id |      descr      | additional
----+-----------------+------------
  1 | Раз             |
  2 | Два             |
  3 | Три             |
  4 | Четыре          |
  5 | Пять - локально |
(5 rows)
```

Фактически, процесс logical replication worker будет периодически перезапускаться, проверяя, не устранен ли конфликт. Поэтому информация в pg_stat_subscription пропадает:

```
β=> SELECT * FROM pg_stat_subscription \gx
-[ RECORD 1 ]---------+---------
subid                 | 41010
subname               | test_sub
pid                   |
relid                 |
received_lsn          |
last_msg_send_time    |
last_msg_receipt_time |
latest_end_lsn        |
latest_end_time       |
```

В журнал сообщений будут попадать записи о нарушении ограничений целостности:

```
postgres$ tail -n 3 /var/log/postgresql/postgresql-10-beta.log
2018-06-13 19:59:56.666 MSK [4004] ERROR:  duplicate key value violates unique constraint "test_pkey"
2018-06-13 19:59:56.666 MSK [4004] DETAIL:  Key (id)=(5) already exists.
2018-06-13 19:59:56.668 MSK [3724] LOG:  worker process: logical replication worker for subscription 41010 (PID 4004) exited with exit code 1
```

Чтобы разрешить этот конфликт, надо удалить конфликтующую строку из таблицы:

```
β=> DELETE FROM test WHERE id = 5;
DELETE 1
```

...и немного подождем.

Проверим:

```
β=> SELECT * FROM test;
 id | descr  | additional
----+--------+------------
  1 | Раз    |
  2 | Два    |
  3 | Три    |
  4 | Четыре |
  5 | Пять   |
  6 | Шесть  |
(6 rows)
```

Данные появились, репликация восстановлена.

Триггеры на подписчике

На подписчике могут выполняться триггеры, но если просто создать триггер, то он не отработает. Это удобно, если на обоих серверах созданы одинаковые таблицы с одинаковым набором триггеров: в таком случае триггер уже отработал на публикующем сервере, его не надо выполнять на подписчике.

Попробуем.

```
β=> CREATE FUNCTION change_descr() RETURNS trigger AS $$
BEGIN
  NEW.additional := 'получено из публикации';
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION
```

```
β=> CREATE TRIGGER test_before_row
BEFORE INSERT OR UPDATE ON test
FOR EACH ROW
EXECUTE PROCEDURE change_descr();
CREATE TRIGGER
```

```
α=> INSERT INTO test(descr) VALUES ('Семь');
INSERT 0 1
β=> SELECT * FROM test;
 id | descr  | additional
----+--------+------------
  1 | Раз    |
  2 | Два    |
  3 | Три    |
  4 | Четыре |
  5 | Пять   |
  6 | Шесть  |
  7 | Семь   |
(7 rows)
```

Чтобы триггер работал, надо изменить таблицу:

```
β=> ALTER TABLE test ENABLE REPLICA TRIGGER test_before_row;
ALTER TABLE
```

```
α=> INSERT INTO test(descr) VALUES ('Восемь');
INSERT 0 1
β=> SELECT * FROM test;
 id | descr  |       additional
----+--------+------------------------
  1 | Раз    |
  2 | Два    |
  3 | Три    |
  4 | Четыре |
  5 | Пять   |
  6 | Шесть  |
  7 | Семь   |
  8 | Восемь | получено из публикации
(8 rows)
```

Можно сделать и так, чтобы триггеры срабатывали не только для репликации, но и локально (ENABLE ALWAYS TRIGGER). Различить эти ситуации можно с помощью функции pg_replication_origin_session_is_setup, но она требует прав суперпользователя.

Удаление подписки

Если репликация больше не нужна, надо удалить подписку - иначе на публикующем сервере останется открытым репликационный слот.

```
β=> \c - postgres
You are now connected to database "replica_logical" as user "postgres".
β=> DROP SUBSCRIPTION test_sub;
NOTICE:  dropped replication slot "test_sub" on publisher
DROP SUBSCRIPTION
```

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА И ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ

Список рекомендуемой литературы

основная

1) Советов, Б. Я. Базы данных : учебник для прикладного бакалавриата / Б. Я. Советов, В. В. Цехановский, В. Д. Чертовской. — 3-е изд., перераб. и доп. — Москва : Издательство Юрайт, 2019. — 420 с. — (Бакалавр. Прикладной курс). — ISBN 978-5-534-07217-4. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://www.biblio-online.ru/bcode/431947
2) Латыпова Р.Р., Базы данных. Курс лекций: учебное пособие [Электронный ресурс] / Латыпова Р.Р. - М. : Проспект, 2016. - 96 с. - ISBN 978-5-392-19240-3 - Режим доступа: http://www.studentlibrary.ru/book/ISBN9785392192403.html
3) SQL / Фиайли К. - М. : ДМК Пресс, 2008. - ISBN 5-94074-233-5 - Текст : электронный // ЭБС "Консультант студента" : [сайт]. - URL : https://www.studentlibrary.ru/book/ISBN5940742335.html (дата обращения: 11.11.2020). - Режим доступа : по подписке.

дополнительная

4) Кусмарцева, Н. Н. Разработка и эксплуатация удаленных баз данных : учебное пособие / Н. Н. Кусмарцева. — Волгоград : Волгоградский институт бизнеса, 2009. — 141 с. — ISBN 978-5-9061-7236-5. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/11343.html (дата обращения: 18.12.2020). — Режим доступа: для авторизир. пользователей
5) Редмонд Э., Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL / Эрик Редмонд, Джим. Р. Уилсон ; Пер. с англ. Слинкин А.А. - М. : ДМК Пресс, 2013. - 384 с. - ISBN 978-5-94074-866-3 - Текст : электронный // ЭБС "Консультант студента" : [сайт]. - URL : https://www.studentlibrary.ru/book/ISBN9785940748663.html (дата обращения: 11.11.2020). - Режим доступа : по подписке.
6) Гутман, Г. Н. Объектно-реляционная СУБД PostgreSQL : учебное пособие / Г. Н. Гутман. — Самара : Самарский государственный технический университет, ЭБС АСВ, 2016. — 125 с. — ISBN 2227-8397. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL:

http://www.iprbookshop.ru/90660.html (дата обращения: 18.12.2020). — Режим доступа: для авторизир. Пользователей

7) Разработка и защита баз данных в Microsoft SQL Server 2005 / . — 2-е изд. — Москва : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. — 147 c. — ISBN 2227-8397. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/73719.html (дата обращения: 18.12.2020). — Режим доступа: для авторизир. пользователей

учебно-методическая

8) SQL-запросы: учеб.-метод. пособие / Кондратьев Алексей Евгеньевич, О. А. Фатьянова; Ульяновск. гос. ун-т, ФМиИТ. - Ульяновск : УлГУ, 2009. URL: ftp://10.2.96.134/Text/Kondratiev_SQL.pdf

Программное обеспечение

1. Open System Architect (open source),
2. СУБД PostgreSQL (open source),
3. pgAdmin4 (open source).