МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Национальный исследовательский университет ИТМО»

ФАКУЛЬТЕТ ПРОГРАММНОЙ ИНЖЕНЕРИИ И КОМПЬЮТЕРНОЙ ТЕХНИКИ

**ЛАБОРАТОРНАЯ РАБОТА №4**

по дисциплине

«РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ ХРАНЕНИЯ ДАННЫХ»

**Выполнили:**

Студенты группы P3318

Рамеев Тимур

Ильгизович

Горло Евгений

Николаевич

**Преподаватель:**

Заболотная Ольга

Михайловна

Оглавление

[Оглавление 2](#_Toc209174993)

[Задание 3](#_Toc209174994)

[Требования: 3](#_Toc209174995)

[Этапы: 3](#_Toc209174996)

[Выполнение 4](#_Toc209174997)

[Этап 1. Конфигурация 4](#_Toc209174998)

[Этап 2. Симуляция и обработка сбоя. 8](#_Toc209174999)

[Этап 3. Восстановление 10](#_Toc209175000)

[Вывод 13](#_Toc209175001)

# Задание

Цель работы - ознакомиться с методами и средствами построения отказоустойчивых решений на базе СУБД Postgres; получить практические навыки восстановления работы системы после отказа.

## Требования:

* В качестве хостов использовать одинаковые виртуальные машины.
* В первую очередь необходимо обеспечить сетевую связность между ВМ.
* Для подключения к СУБД (например, через psql) использовать отдельную виртуальную или физическую машину.
* Демонстрировать наполнение базы и доступ на запись на примере **не менее, чем двух** таблиц, столбцов, строк, транзакций и клиентских сессий.

## Этапы:

**Этап 1. Конфигурация**

Настроить репликацию postgres на трёх узлах: A - основной, B и C - резервные. Для управления использовать pgpool-II. Репликация с A на B синхронная. Репликация с A на C асинхронная. Продемонстрировать, что новые данные реплицируются на B в синхронном режиме, а на C с задержкой.

**Этап 2. Симуляция и обработка сбоя**

2.1 Подготовка:

Установить несколько клиентских подключений к СУБД.

Продемонстрировать состояние данных и работу клиентов в режиме чтение/запись.

2.2 Сбой:

Симулировать неожиданное отключение основного узла - выполнить Power Off виртуальной машины.

2.3 Обработка:

Найти и продемонстрировать в логах релевантные сообщения об ошибках.

Выполнить переключение (failover) на резервный сервер.

Продемонстрировать состояние данных и работу клиентов в режиме чтение/запись.

**Этап 3. Восстановление**

Восстановить работу основного узла - откатить действие, выполненное с виртуальной машиной на этапе 2.2.

Актуализировать состояние базы на основном узле - накатить все изменения данных, выполненные на этапе 2.3.

Восстановить исправную работу узлов в исходной конфигурации (в соответствии с этапом 1).

Продемонстрировать состояние данных и работу клиентов в режиме чтение/запись.

# Выполнение

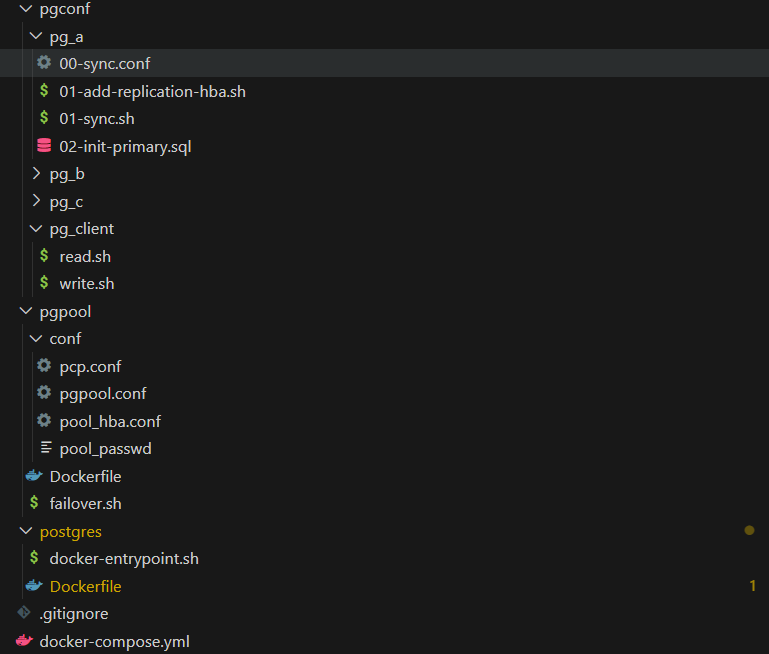
## Этап 1. Конфигурация

Репозиторий с кодом: <https://github.com/Stt1xX/ReplicaPostgreSQL>

Работа была выполнена с использованием платформы **Docker**. Было развернуто 5 виртуальных машин:

1. Pg-a контейнер primary-backend основной узел.
2. Pg-b контейнер standby-backend вспомогательный узел, с синхронной репликацией.
3. Pg-c контейнер standby-backend вспомогательный узел, с асинхронной репликацией
4. Pg-pool контейнер с запущенным pg-pool-ll, через который идет взаимодействие с кластером.
5. Pg-client контейнер который нагружает наш кластер клиентскими сессиями.

**Структура нашего проекта**

****

1. docker-compose.yml – основной docker – файл для сборки и запуска всех контейнеров.
2. Pgconf/pg-a | pg-b |pg-c pg-client – директории, файлы которых нужны для корректной работы соответствующих контейнеров. Все эти файлы будут вмонтированы в соответствующие контейнеры, а затем выполнены.
   1. 00-sync.conf – фрагмент параметров для указания узла pg-b как синхронной реплики.
   2. 01-add-replication-hba.sh – скрипт, который указывает несколько правил для pg-hba.conf, благодаря которым наши реплики смогут беспрепятственно подключиться к pg-a и сделать basebackup.
   3. 01-sync.sh – скрипт, который вставит фрагмент 00-sync.conf (п. а) в postgresql.conf
   4. 02-init-primary.sql – sql-скрипт с тестовыми данными + созданием бд и роли replicator
   5. Read.sh – sh скрипт эмулирующий постоянные запросы на чтение тестовых данных из кластера (через pgpool)
   6. Write.sh – sh скрипт эмулирующий постоянные запросы на запись новых тестовых данных в кластер (через pgpool)
   7. Restore-as-replica.sh – sh скрипт, позволяющий восстановить pg-a узел после его подения (восстановить в состояние реплики)
3. Pgpool/conf Все эти файлы будут вмонтированы в контейнер pgpool, а затем выполнены.
   1. Pgpool.conf – файл, содержащий основные настройки нашего кластера (pgpool-а). Данные обо всех узлах кластера, порты, credentials, failover command и т. д.
   2. Pool-hba.conf – файл, содержит данные о аутентификации на шаге клиент – pgpool (разрешено всем в сети docker).
   3. Pool-passwd – файл, содержит логин и пароль для подключения pgpool к узлам. (У узлов эти credentials должны быть)
   4. Dockerfile – файл-конфиг для настройки + установки компонентов в наш контейнер pgpool.
   5. Failover.sh – скрипт, который будет обрабатывать failover.
4. Postgres
   1. docker-entrypoint.sh – скрипт кастомный entrypoint, вызывается автоматически при создании контейнеров pg-a, pg-b, pg-c. Если он понимает, что он запущен в рамках создания реплики (pg-b, pg-c) выполняет basebackup основного узла.
   2. Dockerfile – файл-конфиг для настройки + установки компонентов в наши контейнеры pg-a, pg-b, pg-c

Итак, выше была кратко описана структура проекта (подробнее ее можно изучить в репозитории (ссылка тоже выше)). Далее мы переходим к работе с проектом.

Сборка проекта:

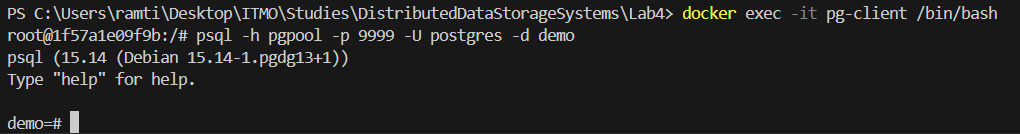


Запуск проекта:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

После того, как мы запустили наш проект, можем подключиться к основному узлу через pg-pool и посмотреть информацию о репликах:



Изображение выглядит как текст, Шрифт, снимок экрана, черный

Содержимое, созданное искусственным интеллектом, может быть неверным.

Также можем подключиться к контейнеру с pg-pool и через pcp утилиту посмотреть состояние наших узлов:

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Для наглядности можем провести эксперимент, подтверждающий статус синхронной\асинхронной реплики:

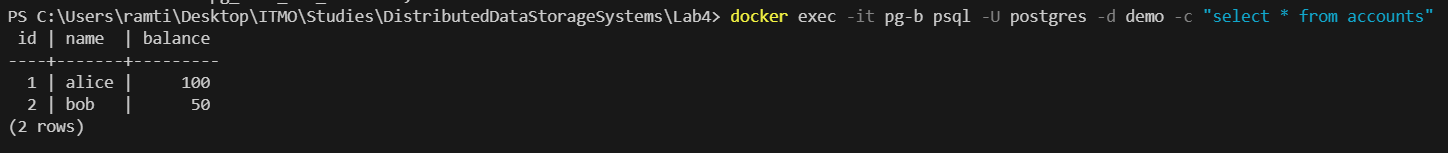
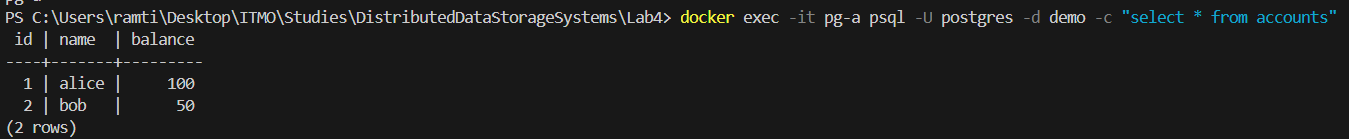
При отключении B реплики, А реплика не может подтвердить успешную запись на B, что приводит к зависанию. При включении B, работа продолжается. А при отключении C реплики, работа продолжается в штатном режиме:



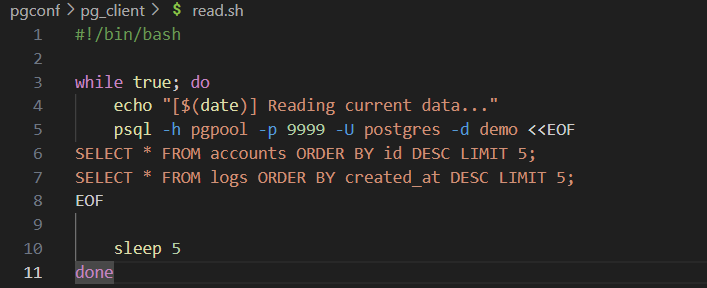


Это абсолютно нормальное поведение.

Ну и можем проверить, что наши реплики актуальны – скрипт docker-entrypoint.sh сработал, и в них есть данные, которые мы вносили в pg-a:

Для демонстрации нескольких клиентских соединений написаны два скрипта read.sh и write.sh:



Изображение выглядит как текст, снимок экрана, Мультимедийное программное обеспечение, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Скрипты запускаются с клиента и обращаются к pg-pool:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

## Этап 2. Симуляция и обработка сбоя.

Для симуляции сбоя достаточно выключить контейнер с основным узлом:



Давайте посмотрим логи pgpool-а:



Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

В логах четко видно, как pgpool теряет связь с основным узлом, происходит триггер скрипта failover, который по сети вызывает promote standby узла.

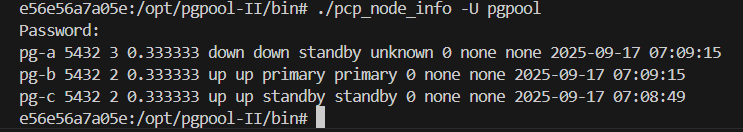
Несмотря на то, что мы отключили наш основной узел, скрипты, которые мы вызвали на стороне клиента, продолжают работать, после небольшого перебоя:

Изображение выглядит как текст, снимок экрана, Шрифт, меню

Содержимое, созданное искусственным интеллектом, может быть неверным.

Скрин не вставляем, но данные в нашем кластере не потерялись.

Состояние узлов наблюдаем через pcp:



## Этап 3. Восстановление

Итак, для начала мы восстановили узел в состояние standby, но уже с актуальными данными. Для этого мы перезапускаем контейнер pg-a с переменной окружения *- RESTORE=REPLICA* .

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Наш кастомный entrypoint при наличии переменной окружения RESTORE запускает особый скрипт restore\_as\_replica.sh, который очищает нерелевантный узел, делает basebackup с нынешнего primary узла и запускает его в режиме standby (standby.signal).

|  |
| --- |
| **docker-compose build pg-a**  **docker-compose up pg-a** |

Немного логов:

Изображение выглядит как текст, снимок экрана, меню, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Теперь можем заметить, что узел pg-a запущен в slave режиме:

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Теперь нам осталось вернуть прежнюю конфигурацию: узел pg-a должен вновь стать primary, а узел pg-b должен стать standby.

Для этого мы перезагружаем наш узел pg-a с переменной окружения *RESTORE=PRIMARY.* Данная процедура также запускает скрипт promote\_and\_reconfigure.sh, который осуществляет promote узла a и basebackup

реплик. Логов очень много, поэтому прикреплять смысла не видим, можно будет показать на сдаче. Результатом этой операции будет такая конфигурация кластера:

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Для тестирования системы мы создали powershell скрипт *reset\_cluster.ps1*, который исполняет задачи, поставленные в лр. Также скрипт на каждом этапе создает нагрузку на сервер в виде запросов на чтение и запись.

|  |
| --- |
| $composeFile = "docker-compose.yml"  (Get-Content $composeFile) -replace '(^\s\*#?\s\*- RESTORE=.\*)', '# $1' |      Set-Content $composeFile  docker compose down -v  Start-Sleep -Seconds 3  docker compose build  Start-Sleep -Seconds 10  docker compose up -d  Start-Sleep -Seconds 10  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "INSERT INTO accounts(name, balance) VALUES ('volodya', 1000);"  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "INSERT INTO accounts(name, balance) VALUES ('nicolya', 1000);"  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "INSERT INTO accounts(name, balance) VALUES ('bodyan', 1000);"  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "SELECT \* FROM accounts;"  (Get-Content $composeFile) -replace '^\s\*#?\s\*- RESTORE=.\*', '      - RESTORE=REPLICA' |      Set-Content $composeFile  # 8. Запустить pg-a  docker compose up -d --force-recreate pg-a  Start-Sleep -Seconds 10  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "INSERT INTO accounts(name, balance) VALUES ('bob', 1000);"  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "INSERT INTO accounts(name, balance) VALUES ('pit', 1000);"  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "INSERT INTO accounts(name, balance) VALUES ('taras', 1000);"  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "INSERT INTO accounts(name, balance) VALUES ('tristan', 1000);"  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "SELECT \* FROM accounts;"  (Get-Content $composeFile) -replace '^\s\*- RESTORE=.\*', '      - RESTORE=PRIMARY' |      Set-Content $composeFile  docker compose up -d --force-recreate pg-a  Start-Sleep -Seconds 10  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "INSERT INTO accounts(name, balance) VALUES ('aaaaaa', 1000);"  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "INSERT INTO accounts(name, balance) VALUES ('bbbbbb', 1000);"  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "INSERT INTO accounts(name, balance) VALUES ('cccccc', 1000);"  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "INSERT INTO accounts(name, balance) VALUES ('dddddd', 1000);"  docker exec pg-client psql -h pgpool -p 9999 -U postgres -d demo -c "SELECT \* FROM accounts;" |

В итоге потом можно посмотреть, что все данные были записаны:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Стоит сразу оговориться, в ходе выполнения лр мы столкнулись с проблемой, связанной с устаревшей версией pgpool. На Docker более свежей версии не нашлось, чем версия 4.4.2. Да можно было заморочиться и поставить актуальную версию через исходники, но мы этим заниматься не стали. Так вот, эта версия периодически выдает segmentation fault на клиентский запрос на чтение\запись. Мы думали, что это связано с неправильной конфигурацией сервера (буферов, кешей, памяти), однако похоже это не так. Другие пользователи также сталкивались с такой проблемой на этой версии, и нейросеть подсказала нам, что это вполне для нее нормально.

Изображение выглядит как снимок экрана, текст, Шрифт, линия

Содержимое, созданное искусственным интеллектом, может быть неверным.

# Вывод

Мы организовали полноценный распределенный кластер под управлением pgpool. Настроили корректную обработку failover. И научились полностью восстанавливать конфигурацию сервера после failover (switchover) без потери данных.