14-MAY-26

Tantor: DBA1-18

Администрирование PostgreSQL 18

Учебный курс
Теоретическая часть

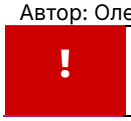
tantor

Автор: Олег Иванов

Оглавление

Авторские права

Автор: Олег Иванов



Создан: **14 мая 2026 г.**
По вопросам обучения обращайтесь: **edu@tantorlabs.ru**

I

Введение

Администрирование PostgreSQL 18

Предварительная подготовка

- Знакомство с SQL
- Навыки работы в операционной системе Linux
- Рекомендуется опыт работы с любой реляционной БД

Целевая аудитория

- Администраторы баз данных PostgreSQL
- Разработчики приложений для баз данных
- Сотрудники технической поддержки

Предварительная подготовка

Для успешного прохождения курса достаточно базовых навыков работы в операционных системах семейства Linux и базовых знаний языка SQL: понимание команд SELECT, UPDATE, INSERT и DELETE. К навыкам работы в операционной системе относится: умение запускать терминал, просматривать в терминале содержимое директорий и файлов, копировать и редактировать текстовые файлы командами ls, cp, mv, cat, mcedit; менять разрешения на файлы командами chmod, chown.

В курсе изучаются основные задачи администрирования баз данных семейства PostgreSQL. В практиках используется Tantor Postgres 18 версии. Курс универсальный: материал курса подходит для всех СУБД семейства PostgreSQL. Большая часть материала подходит не только для PostgreSQL 18 версии, но и предыдущих версий. По возможности, указывается в каких версиях появились изучаемые возможности.

Материалы курса

- Учебное пособие в форме книги в формате pdf, которое содержит теоретическую часть курса
- Практические задания
- виртуальная машина для выполнения практических заданий

Материалы курса

Материалы курса:
1) Учебное пособие в форме книги в формате pdf, которое содержит теоретическую часть курса.
2) Практические задания в формате pdf и html (на сайте dba1.ru)
3) Виртуальная машина с установленной операционной системой Astra Linux 1.8 и СУБД Tantor Postgres версии 18. Может предоставляться доступ к виртуальной машине на время курса или образ в формате ova. Образ виртуальной машины можно использовать с Oracle VirtualBox версии 6.1 и выше или любой другой программе виртуализации.
Материалы курса можно использовать в любое время и после курса.

Разделы курса

0. Введение
1. Установка и управление СУБД
2. Архитектура (5 частей)
3. Конфигурация
4. Базы данных (2 части)
5. Журналирование
6. Безопасность
7. Резервное копирование (2 части)
8. Репликация (2 части)
9. Платформа Tantor, обзор возможностей
10. Новые возможности Tantor Postgres 17.5

Разделы курса

Установка и управление СУБД
- Установка
- Управление экземпляром кластера баз данных
- Утилиты управления кластером баз данных
- Утилита psql

Архитектура
- Общие сведения и структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость
- Конфигурирование

Базы данных

Логическая и Физическая реализация

Диагностический журнал

Безопасность
- Подключение и аутентификация

Резервное копирование
- Физическое и логическое резервирование

Репликация
- Физическая и логическая

Платформа Tantor, обзор возможностей

Возможности Tantor Postgres

О курсе

- очное или дистанционное обучение с инструктором:
  - продолжительность 5 дней
  - начало в 10:00
  - перерыв на обед 13:00-14:00
  - окончание до 17:00 (последний день до 15:00)

О курсе

Курс предназначен для очного или дистанционного обучения с инструктором. Курс состоит из теоретической части, разбитой на главы, практических упражнений и перерывов. Перерывы совмещаются с практическими упражнениями, которые выполняются самостоятельно на подготовленной для курса виртуальной машине.

Примерное расписание:

1) начало в 10:00

2) перерыв на обед 13:00-14:00. Начало обеда может сдвигаться на полчаса в диапазоне от 12:30 до 13:30, так как обычно совмещается с перерывом между главами.

3) окончание теоретической части до 17:00 (в последний день курса до 15:00).

Длительность глав 30-60 минут. Точное время начала изложения глав и времени на практические задания определяет инструктор. Длительность выполнения упражнений может отличаться у разных слушателей и это не влияет на эффективность усвоения материала курса. Доделывать упражнения можно в перерывах между главами или в конце каждого дня. Порядок следования глав и упражнений на эффективность усвоения материала курса не влияет.

Проверка выполнения заданий не производится.

Для успешного усвоения материала курса достаточно:

слушать инструктора, если будут возникать вопросы - задавать их, читать текст практических заданий и самостоятельно их выполнять. При выполнении практических заданий можно набирать команды на клавиатуре, но можно и копировать в терминал из текста заданий. Ввод команд, исправление опечаток, возникающих при наборе команд, изучение ошибок выдаваемых на неправильные команды позволяет лучше запомнить команды. Ощущение понимания текста заданий обманчиво, при работе важно вспомнить основные ключевые слова и возможности команд.

О компании Тантор Лабс

- с 2016 года на международном рынке
- с 2021 года на российском рынке
- разработка СУБД Tantor Postgres
- разработка Платформы Tantor для мониторинга и управления СУБД семейства PostgreSQL, а также кластеров Patroni
- многолетний опыт эксплуатации высоконагруженных систем
- входит в Группу Астра

О компании Тантор

С 2016 года команда Тантор работала на международном рынке поддержки эксплуатации СУБД PostgreSQL и обслуживала клиентов из Европы, Северной и Южной Америки, Ближнего Востока. Команда Тантор разработала программное обеспечение Платформа Tantor и в последующем создала СУБД Tantor Postgres, основанную на программном коде свободно распространяемой СУБД PostgreSQL.

В 2021 году компания полностью переориентировались на российский рынок, где сконцентрировала свои основные направления деятельности на проектирование и разработку СУБД Tantor Postgres, а также развитие Платформы Tantor - инструмента управления и мониторинга БД, основанных на PostgreSQL.

Проектирование и разработка продуктов основывается на накопленном многолетнем опыте в эксплуатации высоконагруженных программных систем в государственном и частном секторах.

В конце 2022 года компания вошла в Группу Астра.

СУБД Tantor Postgres

Tantor BE

Новые возможности и доработки по сравнению с PostgreSQL, техническая поддержка

Tantor SE

СУБД Enterprise-уровня, подходит для наиболее нагруженных OLTP-систем или КХД размером до 100ТБ

Tantor SE 1C

СУБД для высоких нагрузок, оптимизированная и одобренная для работы с приложениями 1С

Tantor Polar

СУБД в составе Tantor xData Gen3 совместима с приложениями 1С

В составе Tantor xData

Максимальные сборки СУБД, оптимизированнаые для работы с 1С

СУБД Tantor Postgres

СУБД Tantor Postgres - реляционная база данных семейства PostgreSQL с повышенной производительностью и стабильностью. Выпускается в нескольких редакциях (сборках): BE (Basic Edition), SE (Special Edition), SE 1C, **Certified** Исполнение 1 (сертифицированные SE и SE 1C) **и Certified** Исполнение 2 (Сертифицированный BE). Редакция Special Edition высоконагруженных OLTP-систем и хранилищ данных размерами до 100ТБ. Редакция Special Edition 1C для приложений 1С.

Для всех редакций доступна техническая поддержка, помощь в построении архитектурных решений, миграции с СУБД других производителей. Программное обеспечение Тантор Лабс включено в "Единый реестр российских программ для электронных вычислительных машин и баз данных".

При приобретении СУБД Tantor Postgres **бесплатно предоставляется лицензия** на Платформу Tantor для управления приобретенными СУБД Tantor Postgres.

Tantor XData

- Программно-Аппаратный Комплекс, с высокой производительностью, отказоустойчивостью, безопасностью
- версия 2A, 2Y - процессоры AMD EPYC и x86-64
- версия 2B - процессоры Baikal-S
- Gen3 - включает Tantor Polar
- высокая производительность и масштабируемость
- снижение затрат на инфраструктуру и администрирование
- в состав входит СУБД Tantor Postgres и Платформа Tantor

Tantor XData

Программно-аппаратный комплекс (ПАК) Tantor XData обеспечивает рабочие нагрузки большого масштаба и критичности с высокой производительностью и доступностью. Консолидация различных рабочих нагрузок Tantor Postgres SE и SE 1C на машине баз данных XData в корпоративных центрах обработки данных помогает организациям повысить операционную эффективность, сократить объем администрирования и снизить затраты.

Программно-аппаратный комплекс (ПАК) Tantor XData предназначен миграции с комплексов иностранных производителей и обеспечивает аналогичную нагрузочную способность. Является заменой высоконагруженным СУБД размером до ~50Тб на один экземпляр, обслуживающие нагрузку типа OLTP, работающие на программно-аппаратных комплексах иностранных производителей. Для СУБД, обслуживающих хранилища данных размером до ~120Тб на один экземпляр.

Является заменой для тяжёлых ERP от 1С при миграции с СУБД иностранных производителей. Позволяет консолидировать несколько СУБД в одном ПАК. Может использоваться при миграции с SAP на 1С:ERP.

Предназначен для создания облачных платформ.

Преимуществом при эксплуатации xData является наличие в составе ПАК удобной графической системы мониторинга работы СУБД: Платформы Tantor.

С 2025 года выпускается **второе** поколение ПАК:

xData **2**A - на процессорах x86-64 на основе серверов Aquarius.

xData **2**Y - на процессорах x86-64 на основе серверов Yadro.

xData **2**B - на процессорах Baikal-S на основе серверов Элпитех

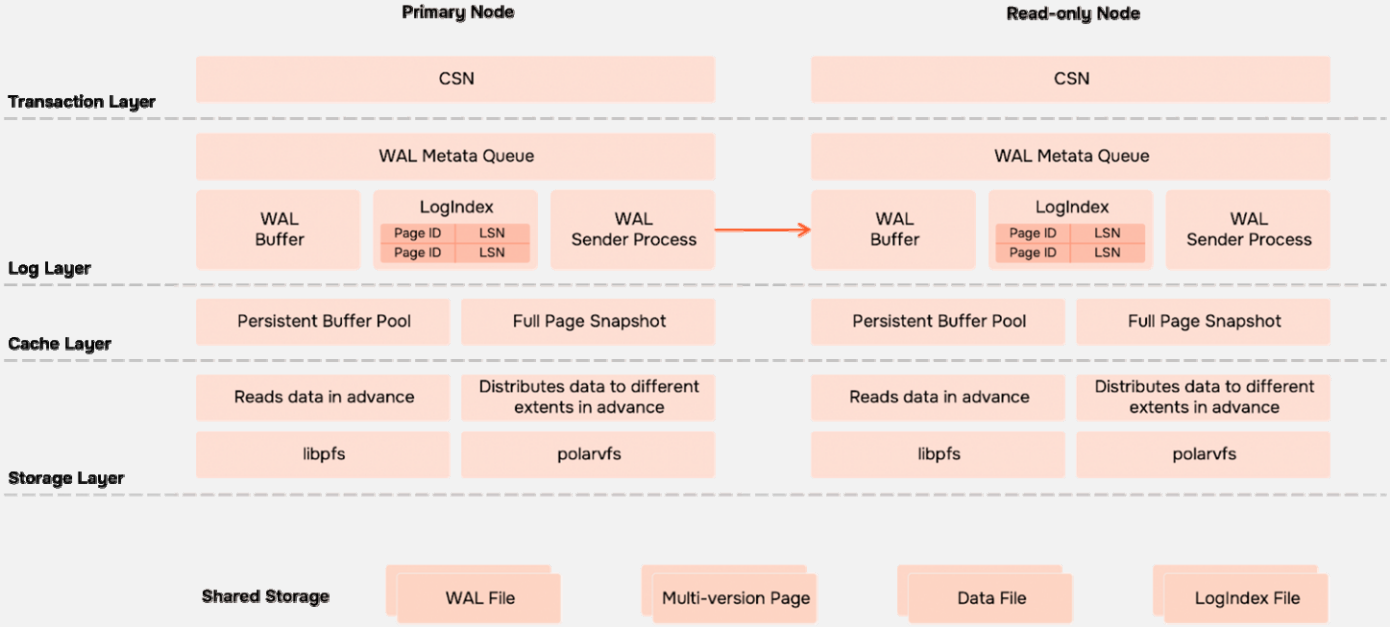
С 2026 года выпускается **третье** поколение ПАК:

xData Gen3 - использует процессоры AMD, а не Intel; первая версия, в состав которой входит СУБД **Tantor Polar**

https://tantorlabs.ru/products/xdata-gen3/tpost/zioiy9l091-postgresql-kotorii-masshtabiruetsya-kak

Tantor Polar

- несколько экземпляров обслуживают одну СУБД



Tantor Polar

Tantor Polar - набор экземпляров, работающих с одним кластером баз данных. Доступен на Tantor XData третьего поколения (Gen3). Один экземпляр - основной (primary), принимает чтение и запись. Остальные экземпляры - реплики, принимают запросы только на чтение.

Запросы могут выполняться процессами на нескольких экземплярах, что называется Elastic Parallel Query (ePQ), основанный на планировщике Greenplum Open Resource Coordinator/Optimizer (GPORCA).

Для подсоединения клиентов используется встроенный пулер (Shared Server), использующий процессы-диспетчеры и наборы (пулы) серверных процессов.

К экземплярам кластера баз данных экземпляры имеют доступ через файловую систему PolarFS, которая монтируется на нескольких узлах. PolarFS работает с блочными устройствами в режиме O_DIRECT (direct i/o, не используя страничный кэш Linux).

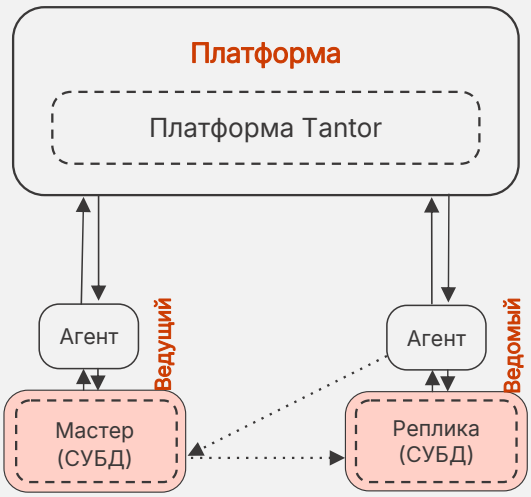
Диски подключаются к отдельным узлам (storage nodes). Для передачи данных от storage-узлов к узлам, на которых работают экземпляры (compute nodes) используется NVMe-oF, передающий данные через прямой доступ к памяти Remote Direct Memory Access (RDMA).

Экземпляры-реплики принимают от экземпляра-мастера поток изменений (WAL) и применяют их к блокам данных, которые есть в их буферном кэше. Для отслеживания изменений используется LogIndex -справочник, сопоставляющая идентификатор каждого блока данных со списком журнальных записей (LSN), которые меняли блок.

Кластера баз данных Tantor Polar могут быть территориально разнесены, отсутствие потерь транзакций при сбое экземпляров основного кластера осуществляется с помощью процесса DataMax (FarSync), который синхронно принимает поток журнальных данных от основного экземпляра первого кластера и асинхронно передаёт их по сети на второй кластер Tantor Polar.

Платформа Tantor

- программное обеспечение для управления большим количеством СУБД и кластеров Patroni
- управляет СУБД Tantor Postgres и форками PostgreSQL
- сбор показателей работы экземпляров PostgreSQL, хранение и обработка показателей, рекомендации по настройке производительности
- интеграция с почтовыми системами, службами каталогов, мессенджерами



Платформа Tantor

Платформа Tantor - программное обеспечение для управления СУБД Tantor Postgres, форками PostgreSQL, кластерами Patroni. Позволяет удобно управлять большим количеством СУБД. Относится к классу программных продуктов, в который входит Oracle Enterprise Manager Cloud Control.

Преимущества использования Платформы Tantor:

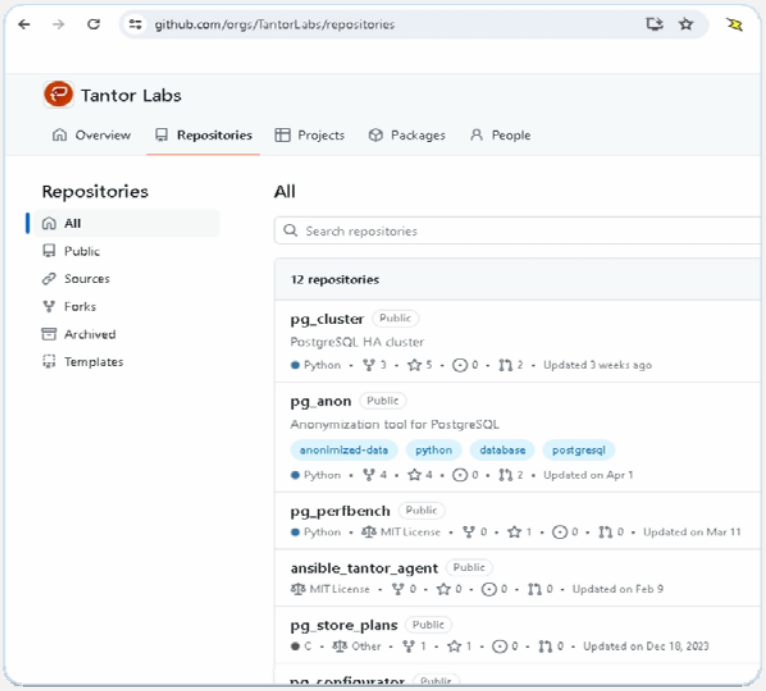
1. Сбор показателей работы экземпляров PostgreSQL, хранение и обработка показателей, рекомендации по настройке производительности
2. Интуитивно понятный и функциональный графический интерфейс позволяет сосредоточиться на показателях работы экземпляров PostgreSQL
3. Автоматизирует рутинные задачи, повышая эффективность работы и снижая вероятность ошибок
4. Управляет не только СУБД Tantor Postgres, но и другими СУБД семейства PostgreSQL
5. Интеграция с почтовыми системами, службами каталогов, мессенджерами
6. В Платформу Тантор входит программное обеспечение Тензор

Платформа Tantor DLH

Тантор Лабс выпускает Платформу Tantor DLH - программное обеспечение, позволяющее организовать процесс трансформации и загрузки данных по логике Extract Transform Load или Extract Load Transform в СУБД Tantor Postgres для организации хранилищ и витрин данных. Относится к классу программных продуктов, в который входит Oracle Data Integrator.

Доработка расширений PostgreSQL

- 1. pg_cluster
- 2. pg_anon
- 3. pg_perfbench
- 4. ansible_tantor_agent
- 5. pg_configurator
- 6. pg_store_plans
- 7. ldap2pg
- 8. citus
- 9. wal-g
- 10. odyssey
- 11. plantuner
- 12. pg_orchestrator
- 13. pgtools
- 14. pipelinedb



Доработка расширений PostgreSQL

Сотрудники Тантор Лабс дорабатывают и создают расширения для СУБД PostgreSQL.
Репозитории расширений: https://github.com/orgs/TantorLabs
Список расширений:
1. pg_cluster
2. pg_anon
3. pg_perfbench MIT License
4. ansible_tantor_agent MIT License
5. pg_configurator MIT License
6. pg_store_plans
7. ldap2pg PostgreSQL License
8. citus GNU Affero General Public License v3.0
9. wal-g Apache License, Version 2.0 (lzo - GPL 3.0+)
10. odyssey BSD 3-Clause "New" or "Revised" License
11. plantuner
12. pg_orchestrator MIT License
13. pgtools
14. pipelinedb Apache License 2.0

Конференции PGBootCamp

- Тантор Лабс принимает участие в организации конференций
- Конференция PGBootCamp проводилась:
  - Москва 19 марта 2026
  - Екатеринбург 10 апреля 2025
  - Казань 17 сентября 2024
  - Минск 16 апреля 2024
  - Москва 5 октября 2023

Конференции PGBootCamp

Тантор Лабс является активным участником организации конференций сообщества PostgreSQL в рамках глобальной инициативы PG BootCamp.

Участие в конференции **бесплатно и возможно онлайн и оффлайн**: https://pgbootcamp.ru/

Можно стать докладчиком на конференции.

Материалы докладов конференций в открытом доступе: https://github.com/PGBootCamp

Выступления https://www.youtube.com/@PGBootCampRussia и https://rutube.ru/channel/32804184/

Конференция PGBootCamp проводилась:

Москва 19 марта 2026

Екатеринбург 10 апреля 2025

Казань 17 сентября 2024

Минск 16 апреля 2024

Москва 5 октября 2023

Конференция Tantor JAM проводится осенью, участие очное и бесплатное. Tantor Jam проводилась в Москве 19 сентября 2025 года. Материалы выступлений: https://tantorlabs.ru/jam-2025 и https://tantorlabs.ru/jam-2024

1

1a

Установка

Предварительные требования

- PostgreSQL может работать под:
  - Linux, macOS, Windows, BSD, Solaris
- Tantor Postgres выпускается для Linux:
  - **RPM**: Redos 7.3, 8; AltLinux p10, p11; MSVSphere; Oracle Linux 8; Rocky 8, 9
  - **DEB**: Astra Linux Special Edition 4.7, 1.7, 1.8; Ubuntu 20, 22; Debian 10, 11, 12, 13
- с Astra Linux поставляется пакет tantor-free-server-18
- минимальные требования к оборудованию:
  - 4 ядра CPU, 4Гб RAM, 40Гб SSD

Предварительные требования

PostgreSQL может работать под Linux, macOS, Windows, BSD, Solaris (https://www.postgresql.org/download/). Для Linux PostgreSQL может устанавливаться из пакетов deb и rpm утилитами dpkg и rpm, из репозиториев утилитами apt и yum. PostgreSQL линкуется с библиотеками и может требовать установки или обновления этих библиотек. В ванильном PostgreSQL софт разделён на пакеты: postgresql-18 (СУБД), зависит от postgresql-common (утилиты-обёртки), зависит от postgresql-client-18 (клиентские утилиты и библиотеки). Пакет postgresql-common устанавливает утилиты-обёртки pg_ctlcluster (обёртка над pg_ctl), pg_createcluster (initdb), pg_backupcluster (pg_basebackup) и другие. Обёртки имеют цель упростить работу с несколькими кластерами, но при промышленном использовании усложняют работу. Расширения поставляются в большом числе отдельных пакетов (postgresql-18-pg-uuidv7, postgresql-18-repack,...).

Tantor Postgres выпускается только под Linux в пакетах deb или rpm. Только часть модулей поставляется в виде отдельных пакетов. Меньшее число пакетов упрощает установку и обновление. Tantor Postgres и большинство форков не используют утилиты-обёртки и управляются стандартными утилитами (pg_ctl, initdb).

С Astra Linux поставляется пакет **tantor-free-server-18**.

Tantor Postgres выпускается для операционных систем:

Linux с пакетным менеджером RedHat Packet Manager (rpm): Redos 7.3, 8; AltLinux p10, p11 MSVSphere; Oracle Linux 8; Rocky 8, 9

Linux с пакетным менеджером Debian (deb): Astra Linux Special Edition 4.7, 1.7, 1.8; Ubuntu 20, 22; Debian 10, 11, 12, 13.

Дистрибутивы для других операционных систем (например, ROSA) выпускаются по запросу.

Оборудование:

Число ядер центральных процессоров: от 4;

Оперативная память: от 4ГБ;

Свободное место на диске: не менее 40ГБ (плюс место под пользовательские данные которые планируется хранить). Рекомендуются твердотельные диски (SSD/NVMe).

https://docs.tantorlabs.ru/tdb/ru/18_1/be/install-binaries.html

Проверка возможности установки

- Дистрибутивы распространяются в виде файла rpm и deb
- В дистрибутивах указаны зависимости
- список нужных пакетов можно получить командами apt, dpkg, rpm

```
apt satisfy postgresql-18
postgresql-18 : Depends: postgresql-client-18 (= 18.3-1.pgdg11+1) but it is
not going to be installed
 Depends: postgresql-common (>= 275~) but 246astra6+ci1 is to be installed
 Depends: libicu67 (>= 67.1-1~) but it is not installable
 Depends: libldap-2.4-2 (>= 2.4.7) but it is not installable
 Depends: libpq5 (>= 17~~) but 15.14-astra.se3.1 is to be installed
 Depends: libssl1.1 (>= 1.1.1) but it is not installable
 Depends: liburing1 (>= 0.7) but it is not installable
```


Проверка возможности установки

Программы пользуются разделяемыми библиотеками, которые обеспечивают полезный функционал и использовались при их сборке. Если библиотеки не будут установлены в операционной системе, то в процессе работы могут возникнуть ошибки, причину которых будет сложно выяснить. В дистрибутивах перечислены те библиотеки, к функционалу которых могут обратиться утилиты и процессы. Такие пакеты называются требуемые (requires) и относятся к зависимостям. К зависимостям могут относиться не только пакеты, но и потребности командных файлов, вызываемых при установке и других инструментов.
Поскольку в разных версиях и сборках PostgreSQL список зависимостей может отличаться, то в документации список требуемых библиотек или пакетов не указан.
На практике же получение списка пакетов, которые нужно установить - актуальная задача.
Для получения полного списка зависимостей конкретного дистрибутива можно использовать команды:
Для пакетного менеджера Дебиан: **`dpkg -I tantor*.deb`**
Для пакетного менеджера RedHat: **`rpm -qp --requires tantor*.rpm`**
Ответ утилит состоит из перечисления пакетов и, возможно, версий пакетов и библиотек. Для проверки перед установкой того, что зависимости выполнены можно использовать команду:
`rpm -i --test tantor*.rpm`
или
`apt satisfy postgresql-18`
Пример:

```
The following packages have unmet dependencies:
 postgresql-18 : Depends: postgresql-client-18 (= 18.3-1.pgdg11+1) but it is not
going to be installed
 Depends: postgresql-common (>= 275~) but 246astra6+ci1 is to be installed
 Depends: libicu67 (>= 67.1-1~) but it is not installable
 Depends: libldap-2.4-2 (>= 2.4.7) but it is not installable
 Depends: libpq5 (>= 17~~) but 15.14-astra.se3.1 is to be installed
 Depends: libssl1.1 (>= 1.1.1) but it is not installable
 Depends: liburing1 (>= 0.7) but it is not installable
 Recommends: postgresql-18-jit but it is not going to be installed
```

Инсталлятор

- проверяет возможные конфликты библиотек и предлагает команду для их устранения
- добавляет адрес репозитория в список apt и yum
- скачивает подходящий дистрибутив и выполняет установку, создание кластера, создание службы
- представляет собой текстовый скрипт:
  - можно ознакомиться с тем, какие действия выполняет инсталлятор
  - легко выяснить в чем ошибки и исправить их

Инсталлятор

Для упрощения установки, Tantor Postgres может быть установлен инсталлятором. Инсталлятор скачивается командой:

```
wget https://public.tantorlabs.ru/db_installer.sh
```

После завершения загрузки поменять разрешения на файл, чтобы скрипт мог выполняться:

```
chmod +x db_installer.sh
```

Дистрибутив можно скачать из личного кабинета https://lk.astra.ru/iso-images и указать инсталлятору путь к скачанному файлу параметром **`--from-file`**:

```
./db_installer.sh --from-file=./tantor-se-server-18_17.5.0_amd64.deb
```

Инсталлятор может скачать дистрибутив из репозитория. Для этого нужно установить переменную окружения `NEXUS_URL`:

```
su -
export NEXUS_URL="nexus-public.tantorlabs.ru"
apt update
./db_installer.sh --edition=be
```

Команда `apt update` обновляет списки пакетов в репозиториев, сохраняя их в `/var/lib/apt/lists`. Скачанные пакеты будут кэшироваться в `/var/cache/apt`.

Обновлять нужно, так как инсталлятор может запросить установку дополнительных пакетов, которые нужны для установки Tantor Postgres.

Возможные ошибки:

1) Конфликты. Например, был установлен клиент (tantor-se-client-*.deb), а пакет с tantor-se-server-* включает в себя библиотеки tantor-se-client-*. В этом случае, инсталлятор выдаст ошибку и команду для ее устранения путем деинсталляции пакета с которым обнаружен конфликт:

```
E: Unmet dependencies. Try 'apt --fix-broken install' with no packages (or specify a solution).
```

После запуска **`apt --fix-broken install`** утилита запросит подтверждение на деинсталляцию пакета.

2) Инсталлятор создает файл `/etc/apt/sources.list.d/tantorlabs.list` или `/etc/yum.repos.d/tantorlabs.repo` и в последующем можно будет не устанавливать переменные окружения. Если будет ошибка аутентификации или захочется не аутентифицироваться, то нужно будет стереть указанные файлы.

3) В директории `/etc/apt/sources.list.d/` или `/etc/yum.repos.d` могут быть файлы с адресами несуществующих репозиториев или с ошибками в параметрах. Нужно удалить такие файлы.

https://docs.tantorlabs.ru/tdb/ru/18_1/be/binary-download-execute.html

Локальная установка

- Параметры инсталлятора `./db_installer.sh --help`
- при установке можно создать кластер
- дистрибутивы (пакеты rpm и deb) имеют стандартный формат, их можно разархивировать, выяснить какие изменения вносятся в операционную систему в процессе установки

```
wget https://public.tantorlabs.ru/db_installer.sh
chmod +x db_installer.sh
export NEXUS_URL="nexus-public.tantorlabs.ru"
apt update
./db_installer.sh --edition=be --major-version=18 --do-initdb
```


Локальная установка

Tantor Postgres Basic Edition (BE) доступна для оценочного использования. Чтобы установить Tantor Postgres BE, нужно установить только одну переменную окружения:

`export NEXUS_URL="nexus-public.tantorlabs.ru"`

Обновить списки пакетов из репозиториев:

`apt update`

Запустить инсталлятор, задав желаемые параметры:

`./db_installer.sh --edition=be --major-version=18 --do-initdb`

Можно указать основную версию и нужно ли создавать кластер после инсталляции. Также кластер можно создать после инсталляции утилитой `initdb`.

Инсталлятор позволяет устанавливать любые сборки СУБД Tantor Postgres из файлов пакетов. Это может быть полезно, если хост не имеет доступа в интернет.

Прежде чем приступить к установке, убедитесь, что вы загрузили правильный бинарный пакет, совместимый с вашей операционной системой и архитектурой. Файл должен иметь расширение **.deb** для систем на основе Debian, **.rpm** для систем на основе Red Hat.

Чтобы начать установку, перейдите в каталог, где находится загруженный файл. Убедитесь, что установочный скрипт `db_installer.sh` присутствует и имеет нужные права на выполнение. Локальная установка выполняется командой:

`./db_installer.sh --do-initdb --edition=se --major-version=18 --from-file=./tantor-se-server-*.deb`

Нужно указать параметром `--major-version=18` основную версию и она должна совпадать с версией (обычно присутствует в названии файла пакета), иначе инсталлятор может создать директорию с неверным номером версии.

Также можно установить пакет не пользуясь установочным скриптом, а используя пакетный менеджер операционной системы:

`rpm -i tantor*.rpm или dpkg -i tantor*.deb`

В этом случае кластер создаваться не будет и его можно создать позже утилитой initdb. Фактически инсталлятор может быть полезен при локальной установке тем, что может выполнить дополнительные действия. Недостатком может являться то, что программный код (обёрток над пакетным менеджером) может добавлять ошибки. Например, не предусматривать все возможные особенности конфигурирования операционной системы.

Процесс установки

- создается пользователь с именем postgres
- создается служба запуска кластера
- создаются директории:
  - `/opt/tantor/db/18`
  - `/var/lib/postgresql`
  - `/var/lib/postgresql/tantor-se-18/data`
  - `/var/run/postgresql`
  - `/usr/lib/systemd/system/tantor-se-server-18.service`

Процесс установки

В процессе установки:

1) создается или модифицируется пользователь postgres в операционной системе:

```
useradd -r -g postgres -c "Tantor database server" -d /var/lib/postgresql -s /bin/bash postgres
```

Менять имя пользователя postgres на другое, исходя из целей безопасности, не нужно.

2) создаётся директория `/opt/tantor/db/18`, в которой располагается софт СУБД.

3) создаётся файл описателя службы `/usr/lib/systemd/system/tantor-se-server-18.service` для того, чтобы можно было запускать экземпляр, обслуживающий кластер баз данных. Кластер баз данных - это директория в файловой системе хоста. Для того, чтобы программы-клиенты могли "работать с СУБД" (посылать команды на языке SQL, получать данные) на хосте нужно запустить набор процессов, которые будут читать и писать в директорию кластера и держать соединение ("сокет") с клиентской программой. Такой набор процессов и память, которую они используют в операционной системе хоста, называют экземпляром кластера баз данных PostgreSQL или, сокращённо, экземпляром.

Проверить статус службы можно командой:

```
systemctl status tantor-se-server-18
```

4) создается директория `/var/run/postgresql` и файл `/usr/lib/tmpfiles.d/tantor-db.conf` Файл используется службой очистки временных файлов. Директория является директорией по умолчанию для файлов Unix socket (параметр конфигурации (`unix_socket_directories`). Можно проверить, что в директории `/usr/lib/tmpfiles.d` отсутствуют другие файлы, которые могли остаться от предыдущих инсталляций postgresql, в которых указана та же самая директория, но с другими параметрами

```
systemctl status systemd-tmpfiles-* | grep Duplicate
```

5) создаётся директория для файлов кластера `/var/lib/postgresql/tantor-se-18/data`

6) в конец файла `/var/lib/postgresql/.bash_profile` добавляются строки

```
export PATH=/opt/tantor/db/18/bin:$PATH
```

Примечание:

Можно проверить что в переменной окружения `LD_PRELOAD` отсутствуют библиотеки, которые могут перекрыть библиотеки PostreSQL, так как `LD_PRELOAD` превалирует. Пути к библиотекам также указываются в файлах директории `/etc/ld.so.conf.d/`

После установки

- Установить переменную окружения PGDATA в файл профиля пользователя postgres (`/var/lib/postgresql/.bash_profile`)
- установить начальные значения параметров конфигурации
- сконфигурировать программы управления и мониторинга (Платформа Тантор)

```
cat /var/lib/postgresql/.bash_profile
export PGDATA=/var/lib/postgresql/tantor-se-18/data
#export LC_MESSAGES=ru_RU.utf8
export PATH=/opt/tantor/db/18/bin:$PATH
```


После установки

PostgreSQL не имеет ограничений по числу экземпляров, запускаемых на одном хосте. Однако, промышленные сервера баз данных обычно высоко нагружены и на одном узле не запускают несколько экземпляров. Несколько экземпляров на одном узле могут запускаться временно, например, в процессе миграции на новую версию.

В ванильном PostgreSQL имеются утилиты `pg_controlcluster, pg_createcluster`, являющиеся обёртками для стандартных утилит `pg_ctl, initdb`. Предполагается, что это упрощает работу с несколькими кластерами на одном узле. Tantor Postgres не использует эти утилиты.

После установки можно:

1) добавить в файл профиля пользователя postgres (`/var/lib/postgresql/.bash_profile`) путь к директории кластера:
`export PGDATA=/var/lib/postgresql/tantor-se-18/data`
это упростит запуск утилит управления кластером, при вызове утилит не нужно будет указывать параметр (`-D` или `--pgdata`), задающий путь к директории кластера.

2) создать кластер, если он не был ещё создан

3) запустить кластер командой: `systemctl start tantor-se-server-18`

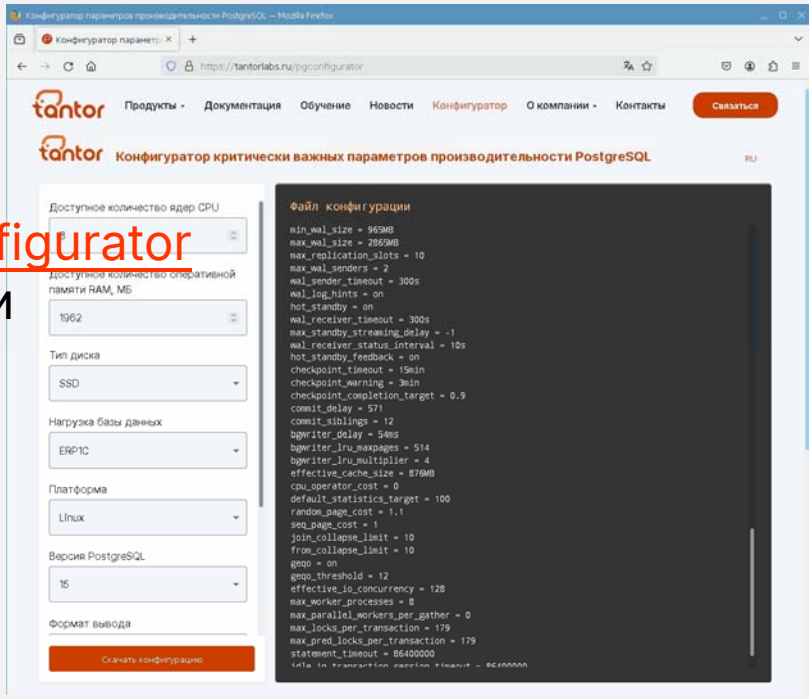
4) если автоматический запуск экземпляра был отключен (по умолчанию включён), то включить: `systemctl enable tantor-se-server-18`

5) установить начальные значения параметров конфигурации кластера, используя конфигуратор https://tantorlabs.ru/pgconfigurator

6) Tantor Postgres может управляться Платформой Тантор без дополнительных лицензий.

Конфигураторы

- утилита pg_configurator
- веб-версия: http://tantorlabs.ru/pgconfigurator
- вводятся характеристики хоста и планируемой нагрузки
- выдает параметры конфигурации



Конфигураторы

Кластер баз данных создаётся утилитой командной строки `initdb`. Утилита создает файл `postgresql.conf` со значениями по умолчанию. Эти значения рассчитаны на обслуживание не слишком нагруженного приложения так, чтобы СУБД можно было использовать на десктопе для небольших задач. Предполагается, что настройка параметров для промышленного использования будет выполнена отдельно.

Для начальной настройки можно использовать утилиту `pg_configurator`, созданную и поддерживаемую Тантор Лабс. Утилита доступна на сайте https://tantorlabs.ru/pgconfigurator/ оболочка в виде утилиты командной строки https://github.com/TantorLabs/pg_configurator

Утилита принимает 7 или ~20 параметров и дает рекомендации на их основе.

Аналоги:

1. PGconfigurator www.cybertec-postgresql.com, веб-версия pgconfigurator.cybertec.at дает рекомендации на основе 13 параметров
2. PGConfig https://github.com/pgconfig/api, веб-версия www.pgconfig.org дает рекомендации на основе 8 параметров
3. PGTune github.com/le0pard/pgtune, создан сотрудником 2ndQuadrant, веб-версия pgtune.leopard.in.ua дает рекомендации на основе 7 параметров

В процессе эксплуатации СУБД параметры конфигурации может рекомендовать конфигуратор Платформы Tantor. Конфигуратор Платформы дает рекомендации на основе ~25 параметров.

Руководство по настройке работы PostgreSQL с продуктами 1С: https://wiki.astralinux.ru/tandocs/nastrojka-postgresql-tantor-dlya-raboty-1s-294394904.html

Создание кластера утилитой initdb

- Кластер создается утилитой initdb
- Перед запуском утилиты нужно создать директорию для файлов кластера
- Выбрать настройки локализации
  - `--lc-collate` (LC_COLLATE) - правила сортировки текста
  - `--lc-ctype` (LC_CTYPE) - классификация символов (заглавные, прописные, цифры)
  - `--encoding` (из LOCALE) - схема кодирования символов

```
initdb -k -g --locale-provider=libc --encoding=UTF8 --locale=en_US.UTF8
--lc-collate=en_US.UTF8 --lc-ctype=en_US.UTF8 -c cluster_name='replica'
-c port=5433
```


Создание кластера утилитой initdb

Кластер создается утилитой командной строки `initdb`, которая запускается из под пользователя postgres.

Перед запуском утилиты нужно создать директорию, где будут находиться файлы создаваемого кластера PGDATA, установить разрешения и права владения на эту директорию и те директории, в которых она находится, для пользователя postgres.

При запуске экземпляра выполняются проверки на директорию `PGDATA` (поддиректории не проверяются):

1) владельцем должен быть пользователь postgres

2) разрешения должны быть `0700 (drwx --- ---)` или `0750 (drwx r-x ---)`
(**ноль** означает, что число восьмеричное). Параметром `-g` или `--allow-group-access` можно установить менее ограничивающие разрешения на

При создании кластера нужно выбрать настройки локализации, которые нельзя изменить после создания кластера для создаваемых при создании кластера баз данных `postgres`, `template0`, `template1`), но можно будет выбирать при создании других баз данных:

1) `--lc-collate` (если не установить, то берётся из переменной окружения `LC_COLLATE`) - порядок символов, влияют на сравнение и сортировку текста

2) `--lc-ctype` (`LC_CTYPE`) - классификация символов (заглавные буквы, прописные буквы, символы цифр и другие классы символов), влияют на функции upper(), lower(), isalpha()

3) `--encoding` (значение после точки переменной `LOCALE`) - схема кодирования символов. Стоит установить в UTF8.

Пример: **`initdb -g --locale-provider=libc --encoding=UTF8 --locale=en_US.UTF8 --lc-collate=en_US.UTF8 --lc-ctype=en_US.UTF8`**

Если не указывать параметры, то используются переменные окружения. Получить список переменных окружения можно командой **`locale`**.

Список допустимых комбинаций **`locale -a`**.

Настроить командой **`dpkg-reconfigure locales`**.

В 16 версии у утилиты появился параметр **`-c`** (или **`--set`**), которым можно добавить значения параметров конфигурации в конец создаваемого файла конфигурации `postgresql.conf`:

`initdb -c cluster_name='replica' --set port=5433 -D .`

Провайдеры локализации

- Для создания кластера и баз данных лучше выбирать кодировку UTF8
- Три провайдера: builtin, libc, icu
  - **libc** - провайдер по умолчанию, **icu** появился в 10 версии, **builtin** появился в 17 версии
- builtin некорректно сортирует буквы ё, Ё

```
 create database lab01builtin LOCALE_PROVIDER=builtin
BUILTIN_LOCALE='PG_UNICODE_FAST' TEMPLATE=template0;
 \c lab01builtin \\
 select string_agg(n, ' ') from (SELECT n FROM unnest(ARRAY['a', 'e',
'ё', 'Ж', 'я', 'Ё','E']) n ORDER BY n);
   string_agg
---------------
 Ё E Ж a e я ё
(1 row)
```


Провайдеры локализации

Провайдер (библиотека, предоставляющая функции) локализации выбирается параметром **initdb --locale-provider={builtin|libc|icu}** или LOCALE_PROVIDER команды create database.

libc - провайдер по умолчанию, **icu** появился в 10 версии, **builtin** появился в 17 версии.

Провайдер **builtin** не зависит от библиотек операционной системы и использует только код ядра PostgreSQL. Недостаток в том, что он поддерживает только три локали: C (идентична локали C в libc), C.UTF-8 (используется только, если кодировка базы данных UTF8), PG_UNICODE_FAST в которых буква Ё идёт до кириллических букв, а буква ё после.

Пример команды создания базы данных:

```
 create database lab01builtin LOCALE_PROVIDER=builtin
BUILTIN_LOCALE='PG_UNICODE_FAST' TEMPLATE=template0;
```

При использовании провайдеров libc и icu при обновлении операционной системы, меняющей версии библиотек, потребуется давать команду REFRESH COLLATION и перестраивать индексы. Индексы могут долго создаваться и это приведёт к простою. При использовании провайдера builtin этого делать не придётся, также провайдер работает немного быстрее по сравнению с libc и icu.

Репликам стоит работать на одинаковых версиях linux с одинаковыми наборами locale.

При переходе с RHEL7 на RHEL8 поменялся порядок сортировки специальных символов ($_) в библиотеке libc, чтобы соответствовать стандарту Unicode 9.

При смене версии библиотек libc или icu, возможно, придётся перестроить индексы.

Для создания кластера и баз данных лучше выбирать кодировку UTF8, а не однобайтные кодировки. Хотя кириллические символы в UTF8 и занимают два байта, но при сжатии разница нивелируется благодаря сжатию. По умолчанию, для типа text, при вытеснении в TOAST, сжатие используется (стратегия EXTENDED). При хранении текстов, даже если предполагается, что будет храниться кириллица и латиница, в текстах могут присутствовать Unicode символы и символы с диакритическими значками.

https://docs.tantorlabs.ru/tdb/ru/18_1/be/locale.html#LOCALE-PROVIDERS

1

1b

Управление

Утилита управления экземпляром pg_ctl

- Основные действия:
  - start - запуск экземпляра
  - stop -m smart | fast | immediate - остановка
  - restart - перезапуск
  - reload - перечитывает файлы конфигурации без остановки экземпляра
  - status - показывает статус экземпляра
  - promote - преобразовать реплику в мастер
- запускается под пользователем postgres

Утилита управления экземпляром pg_ctl

Чтобы созданный кластер баз данных мог обслуживать запросы, нужно запустить экземпляр. Экземпляр запускается и останавливается утилитой командной строки `pg_ctl`.

`ctl` - сокращение слова control (управление).

Преимущество утилиты - простота.

Команды, которые может выполнять утилита:

`start` - запуск экземпляра

`stop -m` `smart` | **`fast`** | **`immediate`** - остановка экземпляра

Если буферный кэш большой, то перед остановкой экземпляра стоит выполнить контрольную точку, то есть дать команду checkpoint. Это уменьшит время простоя, в который входит время на остановку экземпляра. При остановке экземпляра завершаются сессии, начинается простой, выполняется контрольная точка. Если перед этой "финальной" контрольной точкой уже была выполнена контрольная точка, то число грязных блоков в буферном кэше будет небольшое и финальная контрольная точка выполнится быстрее.

`restart` - перезапуск, эквивалентен остановке и запуску, поэтому можно указывать режим остановки параметром **`-m`** (или **`--mode=`**) и другие параметры, при запуске будут применены новые переменные окружения (`PGDATA`).

`reload` - перечитывает файлы конфигурации без остановки экземпляра

`status` - выводит статус экземпляра

`promote` - завершить восстановление реплики и преобразовать её в мастер

Также, у утилиты есть команды **`initdb`**, **`logrotate`**, **`kill`**, но они редко используются.

Относительные пути и pg_ctl

- При запуске или перезапуске экземпляра не стоит использовать относительные пути в параметрах и переменных окружения:

```
postgres@tantor:~/tantor-se-18/data$ pg_ctl start -D .
server started
postgres@tantor:~/tantor-se-18/data$ cd ..
postgres@tantor:~/tantor-se-18$ pg_ctl stop
pg_ctl: directory "." is not a database cluster directory
postgres@tantor:~/tantor-se-18$ pg_ctl status
pg_ctl: directory "." is not a database cluster directory
postgres@tantor:~/tantor-se-18$ cd data
postgres@tantor:~/tantor-se-18/data$ pg_ctl status
pg_ctl: server is running (PID: 20205)
/opt/tantor/db/18/bin/postgres "-D" "."
```


Относительные пути и pg_ctl

Для запуска экземпляра нужно указать директорию кластера - `PGDATA`. Это можно сделать, установив переменную окружения или указав в параметре `pg_ctl -D` путь к директории кластера.

Если при запуске использовались относительные пути в переменных окружения или параметрах, то они станут отсчитываться от той директории, из которой запускается команда на перезапуск, что может привести к ошибкам:

```
postgres@tantor:~/tantor-se-18/data$ export PGDATA="."
postgres@tantor:~/tantor-se-18/data$ pg_ctl start
server started
postgres@tantor:~/tantor-se-18/data$ cd ..
postgres@tantor:~/tantor-se-18$ pg_ctl restart
pg_ctl: directory "." is not a database cluster directory
postgres@tantor:~/tantor-se-18$ pg_ctl status
pg_ctl: directory "." is not a database cluster directory
postgres@tantor:~/tantor-se-18$ cd data
postgres@tantor:~/tantor-se-18/data$ pg_ctl status
pg_ctl: server is running (PID: 20290)
/opt/tantor/db/18/bin/postgres
postgres@tantor:~/tantor-se-18/data$ pg_ctl restart
waiting for server to shut down.... done
server stopped
waiting for server to start.... done
server started
```

Процесс postgres

- `pg_ctl` запускает процесс postgres, который порождает (fork) остальные процессы экземпляра
- параметром `-o` утилиты `pg_ctl` можно передавать процессу postgres параметры командной строки и параметры конфигурации кластера

```
pg_ctl start -o "--config_file=./postgresql.conf --work_mem=8MB"
pg_ctl start -o "-c config_file=./postgresql.conf -c work_mem=8MB"
```

- `postgres --single` однопользовательский и однопроцессный режим. Режим используют для исправления содержимого кластера в сложных случаях повреждений
- выхода из однопользовательского режима: `ctrl+d`

Процесс postgres

`pg_ctl` запускает процесс `postgres`, который порождает (fork) остальные процессы экземпляра, и прослушивает входящие соединения. У процесса postgres есть параметры, которые ему может передать `pg_ctl`. В старых версиях PostgreSQL процесс postgres назывался postmaster.

Для передачи параметров конфигурации от `pg_ctl` к postgres используется параметр **`-o`**. Например,

```
pg_ctl start -o "--config_file=./postgresql.conf --work_mem=8MB"
```

также можно использовать синтаксис

```
pg_ctl start -o "-c config_file=./postgresql.conf -c work_mem=8MB"
```

Посмотреть список параметров, которые можно передавать postgres:

```
postgres --help
```

Параметр --single запускает процесс postgres в режиме одного пользователя и одного процесса:

```
postgres --single
PostgreSQL stand-alone backend 18.3
backend> vacuum full
```

Для выхода из режима single mode используется комбинация клавиш `<ctrl+d>`.

Это промпт не утилиты psql, команд psql в этом режиме нет, только команды которые может принимать серверный процесс (синоним backend).

Параметр `--single` **нельзя передать через pg_ctl**, так как межпроцессного взаимодействия нет.

В этом режиме отсутствует межпроцессное взаимодействие и блокировки памяти. Благодаря этому команды выполняются быстрее. Этот режим используется в редких случаях для команд, исправляющих содержимое кластера, например `vacuum full`.

Управление экземпляром через systemctl

- управляет только экземплярами запущенными службой, экземпляр, запущенный `pg_ctl`, утилитой `systemctl` не останавливается
- `systemctl` использует `pg_ctl` для запуска экземпляра
- ожидает запуска экземпляра 300 секунд (параметр `TimeoutSec=300` в файле-описателе службы)

```
systemctl enable tantor-se-server-18
Created symlink /etc/systemd/system/multi-user.target.wants/tantor-se-server-18.service → /lib/systemd/system/tantor-se-server-18.service.
systemctl is-enabled tantor-se-server-18
enabled
systemctl start tantor-se-server-18
cat /lib/systemd/system/tantor-se-server-18.service | grep c=
TimeoutSec=300
systemctl status tantor-se-server-18
```


Управление экземпляром через systemctl

В Linux для запуска служб используется systemd. В дистрибутиве поставляется файл описания службы `/usr/lib/systemd/system/tantor-se-server-18.service` и администратору не требуется его создавать. По умолчанию используется `Type=forking`.

По умолчанию установлен таймаут 5 минут параметром **`TimeoutSec=300`** в этом файле. `systemd` **принудительно остановит экземпляр, если он не запустится в течение этого времени**. На промышленных серверах восстановление после сбоя по журналам может занять значительное время. Стоит использовать значение `infinity` , которое отключает таймаут.

Во время работы сервера его PID сохраняется в первой строке файла `PGDATA/postmaster.pid`. Файл используется для предотвращения запуска нескольких экземпляров на одной директории и может использоваться для получения PID процесса.

В случае если процессы экземпляра **погашены**, а файл **`postmaster.pid`** мешает запустить экземпляр, файл `postmaster.pid` **можно удалить**.

`systemctl` - основная команда для работы с `systemd`. По умолчанию запускается с правами пользователя `root`.

Запуск экземпляра:

`systemctl start tantor-se-server-18.service`

Суффикс "`.service`" можно не писать, так как он используется по умолчанию.

Проверить добавлен ли экземпляр в автозапуск можно командой

`systemctl` **`is-enabled`** `tantor-se-server-18`

systemctl и pg_ctl

- управляет только экземплярами запущенными ей, экземпляр, запущенный `pg_ctl` утилитой `systemctl` не управляется:

```
postgres@tantor:~$ pg_ctl start
waiting for server to start.... done
server started
postgres@tantor:~$ sudo systemctl stop tantor-se-server-18
postgres@tantor:~$ pg_ctl status
pg_ctl: server is running (PID: 33223)
```

- `systemctl` использует `pg_ctl` для запуска экземпляра:

```
cat /lib/systemd/system/tantor-se-server-18.service | grep /pg_ctl
ExecStart=/opt/tantor/db/18/bin/pg_ctl start -D ${PGDATA} -s -w -t ${PGSTARTTIMEOUT}
ExecStop=/opt/tantor/db/18/bin/pg_ctl stop -D ${PGDATA} -s -m fast
ExecReload=/opt/tantor/db/18/bin/pg_ctl reload -D ${PGDATA} -s
```


systemctl и pg_ctl

Во время работы сервера его PID сохраняется в первой строке файла `PGDATA/postmaster.pid`. Файл используется для предотвращения запуска нескольких экземпляров на одной директории и может использоваться для получения PID процесса.

В случае если процессы экземпляра **погашены**, а файл **`postmaster.pid`** мешает запустить экземпляр, файл `postmaster.pid` **можно удалить**.

Если при запуске экземпляра утилитой systemctl выдается ошибка:

```
Starting Tantor Special Edition database server 18...
pg_ctl: another server might be running; trying to start server anyway
lock file "postmaster.pid" already exists
HINT:  Is another postmaster running in data directory
"/var/lib/postgresql/tantor-se-18/data"?
pg_ctl: could not start server
```

это может означать, что экземпляр запущен не через `systemd`, а утилитой `pg_ctl` и systemd не может ни запустить, ни остановить экземпляр, так как он был запущен утилитой `pg_ctl`.

Можно проверить список процессов в операционной системе.

systemd использует для запуска/остановки и других действий утилиту pg_ctl.

Команда **`systemctl stop tantor-se-server-18`** в таком случае не может остановить экземпляр, **результат она не выдаёт и может возникнуть ложное впечатление, что экземпляр погашен.**

Параметр утилиты `pg_ctl -s` (или `--silent`) не выводит информационные сообщения, только ошибки

`-w` (или `--wait`) `-t` (или `timeout=`) не возвращает промпт, ожидая выполнения команды максимум значение, устанавливаемое переменной окружения `PGCTLTIMEOUT` или, если переменная не задана, то **60 секунд**.

Родительский процесс systemd имеет PID=1:

```
postgres@tantor:~$ ps -ef | grep init
root      1     0  0 /sbin/init splash
```

Работа в контейнере docker

- изменяемые файлы, в частности PGDATA, должны лежать на томах (volumes)
- работа в контейнере не добавляет высокой доступности
- процесс postgres **не** должен иметь **PID=1**
- при создании и запуске контейнера нужно использовать параметр `docker run -d --init`

```
root@tantor:~# docker exec имя_контейнера ps
  PID USER      TIME COMMAND
    1 postgres  0:38 postgres
   31 postgres  0:09 postgres: logger
   32 postgres  0:45 postgres: checkpointer
   33 postgres  0:38 postgres: background writer
...
root@tantor:~# docker rm -f имя_контейнера
root@tantor:~# docker run --init -d -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=postgres -e
POSTGRES_INITDB_ARGS="--data-checksums" -e POSTGRES_HOST_AUTH_METHOD=trust -p 5434:5434 -e
PGDATA=/var/lib/postgresql/data -d -v /root/data:/var/lib/postgresql/data --name имя_контейнера postgres
```


Работа в контейнере docker

Номер процесса (PID) postmaster в контейнере не должен быть равен единице (1). Процесс с PID=1 это первый пользовательский процесс, который запускается после инициализации ядра linux. Процесс с 1 порождает (запускает) все остальные процессы. Он является родительским для всех остальных, порождаемых им, процессов. У всех процессов должен быть родительский процесс. У процесса 1 есть свойство: если родительский процесс какого-либо процесса умирает, ядро автоматически назначает процесс 1 родительским для осиротевшего процесса. Процесс 1 должен усыновлять всех сирот.

Процесс postgres следит за состоянием дочерних процессов и получает статус выхода, когда какой-либо дочерний процесс останавливается. Обычное поведение postmaster, если дочерний процесс останавливается со статусом, отличным от 0 (нормальная остановка) - перезапуск экземпляра. Помимо разрыва сессий экземпляр будет недоступен на время восстановления по wal-журналу.

В контейнере Docker процесс 1 это процесс, для которого создаётся контейнер. Процесс postgres **не должен иметь PID=1**:

```
root@tantor:~# docker exec -it контейнер /usr/bin/ps -ef
PID USER      TIME COMMAND
  1 postgres  0:38 postgres
```

Чтобы использовать initd (tini) для запуска экземпляра в контейнере нужно использовать параметр `--init`.

Изменяемые файлы, в частности PGDATA, должны лежать на томах (volumes), иначе при удалении контейнера данные будут потеряны. Пример создания и запуска контейнера:

```
sudo docker pull postgres
sudo docker run -d --init -e POSTGRES_USER=postgres -e
POSTGRES_PASSWORD=postgres -e POSTGRES_INITDB_ARGS="--data-checksums" -e
POSTGRES_HOST_AUTH_METHOD=trust -p 5434:5434 -e PGDATA=/var/lib/postgresql/data -
d -v /root/data:/var/lib/postgresql/data --name postgres postgres
```

Работа экземпляра в контейнере не добавляет высокой доступности.

Работа экземпляра в контейнере даёт немного большую производительность по сравнению с работой в виртуальной машине.

Три режима остановки экземпляра

- `smart` - запрещает новые подключения и ждёт добровольного отсоединения существующих сессий. Этот режим не практичен
- `fast` - запрещаются новые подключения, всем серверным процессам отправляется сигнал прервать транзакции и завершиться. Это лучший выбор. Используется по умолчанию
- `immediate` - режим немедленного выключения

Три режима остановки экземпляра

Экземпляр можно остановить командой `pg_ctl stop`.

Синтаксис команды:

`pg_ctl stop` `[-D` ***`$PGDATA`***`][-m s[mart]|f[ast]|i[mmediate]][-W][-t` ***`секунды`***`][-s]`

На выбор есть три режима:

smart - запрещает новые подключения и ждёт добровольного отсоединения существующих сессий. Этого можно ждать часами, при этом новые подсоединения невозможны, а это простой. В Oracle Database такой режим называется "shutdown normal". В отличие от Oracle Database, после подачи сигнала на остановку в режиме smart **можно** подать сигнал на остановку в режиме fast. Если вы запустили режим smart, то имеете возможность погасить экземпляр в режиме fast.

fast - запрещаются новые подключения, всем серверным процессам отправляется сигнал прервать транзакции и завершиться (сигнал linux SIGTERM 15). Затем завершаются оставшиеся фоновые процессы экземпляра в правильном порядке. Одним из последних действий выполняется контрольная точка. В Oracle Database такой режим называется "shutdown immediate". В отличие от Oracle Database, откат транзакций в PostgreSQL выполняется моментально, поэтому задержка в остановке связана, в основном, определяется длительностью выполнения контрольной точки.

fast - режим остановки по умолчанию для остановки через `pg_ctl` и через `systmemctl`

На кластерах с большим объемом памяти, используемой экземпляром, можно минимизировать время остановки экземпляра, то есть время простоя. Для этого перед остановкой экземпляра нужно инициировать выполнение контрольной точки командой **checkpoint.** После выполнения **checkpoint** послать сигнал на остановку экземпляра. В этом случае, контрольной точке которая всё равно будет выполнена при остановке экземпляра (финальная контрольная точка) в режиме smart или fast, придется записать на диск меньше данных и финальная контрольная точка выполнится быстрее.

В режимах smart и fast все изменившиеся в памяти данные (которые нужно сохранить, то есть "защищаемые журналом предзаписи") по контрольной точке записываются в файлы, информация об успешной остановке экземпляра записывается в **управляющий файл `pg_control`**. Это называется "корректной остановкой". При последующем запуске экземпляра по управляющему файлу **`pg_control`** определяется, что экземпляр был корректно остановлен и чтения журналов WAL не требуется.

Остановка экземпляра

- Экземпляр можно остановить командой `pg_ctl stop`, независимо от того, каким способом был запущен экземпляр
- Использование `pg_ctl` - наиболее удобный и гарантированный способ погасить экземпляр
- Можно послать сигнал процессу postgres напрямую:
  `kill -INT  `head -1 $PGDATA/postmaster.pid``
- `kill -INT $(head -1 $PGDATA/postmaster.pid)`
- Не рекомендуется посылать сигнал `SIGKILL (9)` ни с одному процессу экземпляра
- Для остановки процесса используется функция `pg_terminate_backend(PID)`
- **`systemctl stop` не гарантирует остановки экземпляра**

Остановка экземпляра

immediate - режим немедленного выключения. Родительский процесс postmaster отправит сигнал немедленной остановки SIGQUIT (3) всем остальным процессам и будет ожидать их завершения. Если какой-либо процесс не завершится в течение 5 секунд, ему будет отправлен сигнал SIGKILL (9). Дальше остановится сам процесс postmaster. Это приведет к необходимости воспроизведения журнала WAL, при следующем запуске экземпляра. Режим immediate стоит использовать только в крайних случаях, например, подвисании (отсутствии дисковой активности, прогресса) при остановке в режиме fast. В Oracle Database такой режим называется "shutdown abort".

Использование `pg_ctl stop` - наиболее удобный способ погасить экземпляр, но можно послать сигнал процессу postgres напрямую:

```
kill -INT $(head -1 $PGDATA/postmaster.pid)
kill -INT `head -1 $PGDATA/postmaster.pid`
```

Кавычки обратные, а не апострофы. Для остановки в режиме immediate можно послать сигнал `QUIT`. Сигнал `SIGKILL (9)` посылать процессу postgres не стоит, так как общая память и семафоры не освободятся до перезагрузки операционной системы или до их освобождения вручную командой ipcrm. Также серверные и фоновые процессы могут остаться в памяти. Посмотреть сегменты общей памяти и семафоры можно командой операционной системы ipcs, а освободить ipcrm.

Не стоит посылать сигнал `SIGKILL (9)` и другим процессам экземпляра, в том числе серверным (как это принято при работе с Oracle Database), это может привести к немедленной остановке экземпляра.

Для отсоединения сессий и прерыванию выполняющейся команды (в чужой сессии без её прерывания) в PostgreSQL удобно использовать функции **`pg_terminate_backend`** (посылается `SIGTERM` 15 серверному процессу) и **`pg_cancel_backend`** (посылается `SIGINT` 2).

Перед выполнением процедур, требующих корректной остановки, **следует убедиться что:**
1) все процессы остановленного экземпляра были выгружены из памяти (отсутствуют в операционной системе)

2) в управляющий кластер был записан статус корректной остановки кластера:

```
pg_controldata | grep state
Database cluster state:       shut down
```

https://docs.tantorlabs.ru/tdb/ru/18_1/se/server-shutdown.html

Сообщения об остановке экземпляра

- В журнале кластера, при выполнении контрольных точек (параметр `log_checkpoints=on`), будут присутствовать сообщения типа:
  `СООБЩЕНИЕ: начата контрольная точка: shutdown immediate`
  или
  `LOG: checkpoint starting: shutdown immediate`
- Текст в сообщении "`shutdown immediate`" относится к свойствам контрольной точки, а не к режиму остановки экземпляра. При остановке экземпляра в режиме `immediate` (команда `pg_ctl stop -m immediate`) контрольная точка не выполняется
- в PostgreSQL нет команды shutdown immediate

Сообщения об остановке экземпляра

В диагностическом журнале кластера, при выполнении контрольных точек (параметр `log_checkpoints=on`), будут присутствовать сообщения типа:

```
LOG:  checkpoint starting: shutdown immediate
LOG:  checkpoint complete: wrote 0 buffers (0.0%), wrote 3 SLRU buffers; 0 WAL file(s) added, 0 removed, 0 recycled; write=0.002 s, sync=0.001 s, total=0.008 s; sync files=2, longest=0.001 s, average=0.001 s; distance=0 kB, estimate=0 kB; lsn=0/3D8DB68, redo lsn=0/3D8DB68
```

В PostgreSQL нет команды `shutdown immediate`. Текст в журнале "`shutdown immediate`" в логе относится к свойствам контрольной точки, а не к режиму остановки экземпляра. При остановке экземпляра в режиме immediate (команда `pg_ctl stop -m immediate`), финальная контрольная точка **не выполняется**.

Текст в сообщениях о контрольной точке (после **`LOG:  checkpoint starting:`**) означает:

`shutdown` - контрольная точка вызвана остановкой экземпляра

`immediate` - выполнить контрольную точку с максимальной скоростью, игнорируя значение параметра `checkpoint_completion_target`

`force`: выполнить контрольную точку даже если с прошлой контрольной точки в WAL ничего не было записано (в кластере не было активности), такое происходит если экземпляр останавливается (shutdown) или в конце восстановления (end-of-recovery)

`wait`: ждать завершения контрольной точки перед тем как вернуть управление процессу, вызвавшему контрольную точку (без `wait` процесс запустит контрольную точку и продолжит работать дальше).

`end-of-recovery`: контрольная точка по окончании наката журналов (восстановление кластера процессом startup)

`wal`: контрольная точка вызвана достижением файлами журнала половины размера, заданного параметром `max_wal_size` ("по размеру", "по требованию")

`time`: контрольная точка вызвана достижением значения параметра `checkpoint_timeout` ("по времени")

Утилиты управления (обёртки команд SQL)

- находятся в директории `/opt/tantor/db/18/bin`
- путь к директории включен в переменную окружения `PATH` пользователя postgres в Linux
- Часть утилит командной строки - обёртки для команд SQL

```
postgres@tantor:~$ createdb db01 -e
SELECT pg_catalog.set_config('search_path', '', false);
CREATE DATABASE db01;
postgres@tantor:~$ createuser alice -e
SELECT pg_catalog.set_config('search_path', '', false);
CREATE ROLE alice NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN
NOREPLICATION NOBYPASSRLS;
postgres@tantor:~$ vacuumdb --all --no-truncate --no-index-cleanup
--parallel=3 --freeze --disable-page-skipping
vacuumdb: vacuuming database "db01"
vacuumdb: vacuuming database "postgres"
vacuumdb: vacuuming database "template1"
```


Утилиты управления (обёртки команд SQL)

В директории `/opt/tantor/db/18/bin` (путь к которой добавляется для пользователя postgres в переменную окружения PATH в процессе инсталляции) находятся утилиты для работы с кластером баз данных. Утилиту `initdb` мы рассмотрели. Дальше рассмотрим основную утилиту - терминальный клиент `psql`, которая позволяет передавать на выполнение команды SQL.

Часть действий по управлению кластером, может выполняться утилитами командной строки. Утилиты-обёртки (wrappers) существуют для части команд SQL. В скриптах командной строки удобно использовать утилиты-обёртки, вместо написания вызова команды через psql:

`psql -c "команда"`

Разницы в результате между использованием утилит-оболочек и команд SQL нет.

Утилиты-обёртки:

`clusterdb` - оболочка для команды SQL CLUSTER

`createdb` - оболочка для команды CREATE DATABASE. Разницы создавать базу данных этой утилитой или командой нет

`createuser` - оболочка для команды CREATE ROLE

`dropdb` - оболочка для команды DROP DATABASE

`dropuser` - оболочка для команды SQL DROP ROLE

`reindexdb` - оболочка для SQL-команды REINDEX, параметром `-j` можно указать число параллельно выполняемых команд

`vacuum`**`db`** - оболочка для команды VACUUM.

`vacuum`**`lo`** - к вакуумированию (VACUUM) не имеет отношения. `vacuumlo` - удобная для периодического запуска утилита удаления (вычищения) осиротевших больших объектов из баз данных кластера. Автоматизировать удаление осиротевших больших объектов, можно разными способами (например, триггерами), эта утилита один из способов. Более лучший способ, использовать расширение "lo", которое содержит функцию `lo_manage()` для использования в триггерах, предотвращающих появление осиротевших больших объектов.

Параметр утилит `-e` выводит команды, которые формируют и посылают на выполнение утилиты.

Описание утилит:

https://docs.tantorlabs.ru/tdb/ru/18_1/se/reference-client.html

Утилиты резервирования

- **pg_archivecleanup** - используется на репликах (резервных кластерах)
- **pg_basebackup** - создаёт физические бэкапы
- **pg_combinebackup** - накладывает инкрементальные бэкапы на полные
- **pg_dump**, **pg_dumpall**, **pg_restore** для логического резервирования (создания дампов)
- **pg_receivewal** - для создания поточных архивов WAL
- **pg_resetwal** для изменения размера WAL-сегментов
- **pgcopydb** - утилита для логического резервирования

Утилиты резервирования

`pg_archivecleanup` используется в значении параметра `archive_cleanup_command` для удаления ненужных файлов WAL на физической реплике (резервном кластере).

`pg_basebackup` - утилита создания резервных копий кластера (бэкапов) для клонов, реплик и просто для хранения. Может вытягивать файлы по сети, используя протокол репликации.

`pg_combinebackup` - накладывает инкрементальные бэкапы на полные.

`pg_dump` - создает логическую копию объектов базы данных.

`pg_dumpall` - создает логическую копию всего кластера или общих объектов кластера в виде текстового скрпита создания баз данных и объектов в базах. Представляет интерес параметр `-g`, позволяющий выгружать общие объекты кластера.

`pgcopydb` - утилита Tantor Postgres для автоматизации переноса данных на логическом уровне между базами данных с максимальной скоростью. Утилита использует `pg_dump, pg_restore` и техники логического резервирования.

`pg_receivewal` - используется для вытягивания (pull) по протоколу репликации содержимого файлов WAL (поточного архива). Используется для организации хранения журналов WAL на хостах, хранящих бэкапы.

`pg_recvlogical` - для логической репликации, редко используется.

`pg_resetwal` очищает журнал WAL. Используется с параметром `--wal-segsize` для изменения размера WAL-сегментов, если захочется изменить их размер после создания кластера. Меняют либо по причине огромного числа файлов в директории pg_wal, либо по причине того, что максимальный размер буфера журнала в разделяемой памяти (`wal_buffers`) ограничен размером WAL-файла. Влияние размера WAL-буфера на производительность нелинейна.

`pg_restore` - утилита восстановления из логических бэкапов созданных утилитой `pg_dump` в части режимов (в других режимах для восстановления используется psql)

`pg_waldump` - показывает содержимое WAL-сегментов, используется для отладки в сложных случаях восстановления

`pg_walsummary` - показывает содержимое файла WAL summary.

`pg_createsubscriber` - быстро создаёт из физической реплики клон с бесшовно настроенной логической репликацией. Уменьшает фазу копирования данных при создании подписок. Используется при обновлении на новую версию с минимизацией простоя.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/reference-server.html

Утилиты управления (другие)

- **pg_checksums** - включение/отключение подсчета контрольных сумм блоков данных и проверка блоков данных
- **pg_rewind** - синхронизация PGDATA кластеров
- **pg_upgrade** - обновление на новую версию PostgreSQL
- **pg_test_fsync** - измеряет скорость записи в WAL сегменты в разных режимах
- **pg_test_timing** - измеряет скорость и стабильность получения меток времени
- **pg_config** - информация о параметрах сборки PostgreSQL
- **pg_controldata** - показывает содержимое управляющего файла `$PGDATA/global/pg_control`
- **pgbench** - утилита нагрузочного тестирования

Утилиты управления (другие)

`pg_amcheck` - относится к стандартному расширению (PostgreSQL extension) `amcheck`, которое имеет набор функций для проверки отсутствия повреждений в объектах в которых физически хранятся данные, называемых отношения (relations). Отношениями (relation, синоним “класс”) называются таблицы, индексы, последовательности, представления, внешние (foreign) таблицы, материализованные представления, составные типы. Если функционал amcheck сообщает о повреждениях, то они действительно есть, ложные срабатывания исключены.

`pg_checksums` - включение/отключение подсчета контрольных сумм блоков данных и проверка блоков данных кластера. В Oracle Database аналог - утилита dbv (dbverify);

`pg_rewind` - для синхронизации кластеров, обычно для восстановления бывшего мастера (основной, primary кластер) после аварийного переключения на физическую реплику (резервный, standby кластер), а также в процедурах апгрейда (перехода на новую основную версию);

`pg_upgrade` - используется при обновлении на новую основную версию PostgreSQL, а также при миграции с ванильного PostgreSQL на Tantor Postgres;

`pg_test_fsync` - используется при настройке параметров записи в WAL-журнал;

`pg_test_timing` - измеряет скорость и стабильность получения меток времени;

Полезные утилиты

`pg_config` - информация о параметрах инсталляции и сборки СУБД;

`pg_controldata` - выводит в текстовом виде содержимое управляющего файла кластера `$PGDATA/global/pg_control`;

`pgbench` - стандартная утилита PostgreSQL для нагрузочного тестирования;

https://docs.tantorlabs.ru/tdb/ru/18_1/se/reference-client.html

Утилиты управления (продолжение)

- **pg_isready** - проверка что кластер принимает соединения
- **oid2name** - удобная утилита для поиска к какому объекту относится файл или директория
- **pgcompacttable** - утилита для уменьшения размеров файлов таблиц
- **pg_repack** - относится к одноимённому расширению, реализующему аналог VACUUM FULL, только без монопольной блокировки
- **pg_ctl** - управляет экземпляром кластера, была рассмотрена ранее
- **initdb** - создает кластера, была рассмотрена ранее

Утилиты управления (продолжение)

`pg_isready` проверка, что кластер принимает соединения, аналог `psql -c "\q"`. Утилитой удобнее получать результат, но в psql можно указать дополнительные команды для проверки доступности объектов с точки зрения конкретного клиентского приложения.

`oid2name` - удобная утилита для поиска к какому объекту относится файл в директории кластера (PGDATA) и табличных пространств, а также другой информации о принадлежности файлов и директорий объектам кластера. Аналогичные действия можно выполнить и командами SQL и функциями SQL, но это гораздо сложнее.

`postgresql-check-db-dir` - скрипт поверхностной проверки структуры директории PGDATA, вызывается systemd перед вызовом `pg_ctl` для запуска экземпляра, чтобы убедиться, что в директории PGDATA лежит что-то похоже на директорию кластера.

`pgcompacttable` - утилита для уменьшения размеров файлов таблиц.

`pg_repack` - расширение, которое позволяет, не блокируя полностью объект, реорганизовать файлы в которых хранятся данные. Аналог команды `VACUUM FULL`, только без монопольной блокировки.

Рассмотренные ранее в этой главе:

`pg_ctl` - управляет экземпляром кластера

`initdb` - создаёт кластера

https://docs.tantorlabs.ru/tdb/ru/18_1/se/reference-server.html

1

1c

Утилита psql

Терминальный клиент psql

- позволяет:
  - интерактивно вводить команды SQL
  - из файла `psql -f скрипт.sql`
  - из командной строки `psql -c "select user;select now()"`
- файлы конфигурации
  - глобальный `/opt/tantor/db/18/etc/postgresql/psqlrc`
  - локальный `~/.psqlrc`
- описание параметров командной строки `psql --help`

```
postgres@tantor:~$ cat ~/.psqlrc
\set AUTOCOMMIT off
\set ON_ERROR_ROLLBACK interactive
\setenv PAGER 'less -XS'
\setenv PSQL_EDITOR '/usr/bin/mcedit'
\pset pager off
```


Терминальный клиент psql

В PostgreSQL имеется стандартный терминальный клиент (утилита командной строки) **`psql`**.

В курсе нет цели монотонно описывать все возможности psql, их много. Функционал **`psql`** шире, чем у аналогичных утилит в других СУБД. На следующих слайдах рассматриваются особенности, которые встречаются при повседневной работе. В практике к этой главе даются дополнительные примеры.

`psql` позволяет интерактивно вводить команды, отправлять их серверному процессу и просматривать результаты выполнения команд. Также **`psql`** можно передавать команды и неинтерактивно - команды могут быть взяты из файла или параметра командной строки.

`psql -f скрипт.sql`

`psql -c "CREATE SCHEMA sh; CREATE TABLE sh.t (n numeric);"`

У **`psql`** есть файлы конфигурации. Глобальный файл конфигурации находится в директории, на которую указывает результат выполнения утилиты **`pg_config --sysconfdir`**

для Tantor Postgres это файл `/opt/tantor/db/18/etc/postgresql/psqlrc`

Локальный для пользователя операционной системы лежит в его домашней директории, значение по умолчанию `~/.psqlrc` Местоположение локального файла может быть переопределено переменной окружения `PGCONFIG`.

По умолчанию файлы не созданы, но можно создать. В Oracle Database для sqlplus используется файл glogin.sql

Оба файла `~/.psqlrc` и `psqlrc` могут быть сделаны специфичными для версии psql путем добавления дефиса и идентификатора основной или минорной версии PostgreSQL к имени файла. Например, `~/.psqlrc-18` или `~/.psqlrc-17.5` Все файлы применяются, но превалирует более специфичный файл.

С помощью этих файлов можно сделать работу в psql удобнее.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/app-psql.html#psql

psql: подключение к базе данных

- `psql` подключается к одной базе данных из кластера баз данных
- Для подсоединения к базе нужно пройти аутентификацию
  - способы аутентификации рассматриваются в следующих главах
- Роль и пользователь синонимы и абсолютно одинаковы
- CREATE ROLE по умолчанию устанавливает атрибут NOLOGIN, команда CREATE USER устанавливает LOGIN
- использование пайпа (поточной передачи):

```
postgres@tantor:~$ createdb db02
postgres@tantor:~$ pg_dump -F c db01 | pg_restore -d db02
postgres@tantor:~$ pg_dump db01 | psql -q -d db02
```


psql: подключение к базе данных

psql подключается к конкретной базе данных в кластере. Для подсоединения к базе нужно пройти аутентификацию, которая обычно настраивается отдельно для локальных подсоединений через Unix-сокеты, сетевых соединений с того же хоста на адрес localhost (127.0.0.1) и соединений с других хостов. PostgreSQL поддерживает разнообразные способы аутентификации, они будут рассмотрены в следующих главах курса. Аутентификация возможна и без пароля, но сессия должна быть сопоставлена с ролью (пользователем) кластера. Подсоединение без сопоставления с ролью, заранее созданной в кластере, возможно только в однопользовательском режиме (single mode). В однопользовательском режиме подключение выполняется под пользователем который неявно наделяется правами суперпользователя.

Роль (ROLE) и пользователь (USER) - синонимы и абсолютно одинаковые понятия. Команды `CREATE ROLE` и `CREATE USER` приводят к одинаковому результату, за исключением того, что команда `CREATE ROLE` по умолчанию устанавливает атрибут NOLOGIN, а команда `CREATE USER` устанавливает по умолчанию LOGIN.

После представления имени роли, серверный процесс проверяет привилегии: может ли роль создать сессию (имеет ли она атрибут LOGIN) с конкретной базой данных. Атрибут SUPERUSER не включает в себя право создать сессию, могут существовать роли с атрибутами SUPERUSER и NOLOGIN одновременно.

Подключиться в одной сессии к нескольким базам, даже из одного кластера нельзя. Базы изолированы друг от друга с точки зрения безопасности и привилегий. Для одновременной работы с таблицами в разных базах данных можно использовать расширения `postgres_fdw` (Foreign Data Wrapper) или `dblink`. Для копирования данных между базами данных можно использовать поточную передачу данных ("пайп") и утилиту `pg_dump ... | psql ...`

Соединение с базой данных

- Параметры командной строки для соединения с базой данных:
- -U имя, по умолчанию имя пользователя операционной системы
- -d имя_базы, по умолчанию имя пользователя
- -h хост, по умолчанию /var/run/postgresql
- -p порт, по умолчанию 5432
- Переподсоединение:
  `\c имя_базы имя_пользователя хост порт`
- Параметры текущего соединения передаются символом "-"
  `\c - - localhost`
- Клиентская балансировка

```
psql --host=tantor,localhost load_balance_hosts=random --port=5432,5432
postgres=# \c - - localhost
You are now connected to database "postgres" as user "postgres" on host
"localhost" (address "127.0.0.1") at port "5432".
postgres=#
```


Соединение с базой данных

Параметры командной строки psql, которыми можно указать к какой базе и под каким пользователем подключаться:

`-U имя` или `--username=имя` - значение по умолчанию: имя пользователя операционной системы, под которым запущен psql

`-d имя_базы` или `--dbname=имя_базы` - значение по умолчанию: имя пользователя, заданное параметром `-U`

`-h хост` или `--host=хост` - значение по умолчанию: `/var/run/postgresql` (на стороне экземпляра это же значение задано при сборке и отображается в параметре **`unix_socket_directories`**).

При подсоединении можно использовать сокращенный синтаксис:

`psql имя_базы имя_пользователя`.

Например, `psql postgres postgres`

Если psql или другие утилиты выдают ошибку:

```
Is the server running locally and accepting
connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

то запускается утилита старой версии (например, из пути `/usr/bin/psql`).

Кроме передачи параметра `-h` можно указать директорию Unix-сокетов в переменной окружения `PGHOST`, например, **`export PGHOST=/var/run/postgresql`**

`-p порт` или `--port=порт` - значение по умолчанию: 5432

Для локальных соединений через Unix-сокет тоже используется порт.

Процесс postgres создаёт файл, у которого суффикс является номером порта. Например, `/run/postgresql/.s.PGSQL.5432`

Клиентская балансировка в сетевой библиотеке libpq появилась в 16 версии:

`psql --host=tantor,localhost load_balance_hosts=random --port=5432,5432`

`psql "host=tantor,localhost load_balance_hosts=random port=5432,5432"`

https://docs.tantorlabs.ru/tdb/ru/18_1/se/libpq-connect.html#LIBPQ-CONNECT-LOAD-BALANCE-HOSTS

Параметры соединения

- Команда вывода деталей текущего соединения:

```
postgres=# \conninfo
            Connection Information
      Parameter        |         Value
-----------------------+----------------------
 Database              | postgres
 Client User           | postgres
 Socket Directory      | /var/run/postgresql
 Server Port           | 5432
```

```
postgres@tantor:~$ psql -q postgres://user1:pass@localhost:5432/db01
db01=> \c - postgres /var/run/postgresql 5432
You are now connected to database "db01" as user "postgres".
db01=# \c - user1 - 5432
You are now connected to database "db01" as user "user1".
db01=> \c postgres://localhost:5432/db01?user=user1&password=pass
You are now connected to database "db01" as user "user1".
```


Параметры соединения

Полезная команда psql для вывода деталей подключения: `\conninfo`

До 18 версии:

```
You are connected to database "postgres" as user "postgres" via socket in
"/var/run/postgresql" at port "5432".
```

В 18 версии:

```
            Connection Information
      Parameter        |         Value
-----------------------+----------------------
 Database              | postgres
 Client User           | postgres
 Socket Directory      | /var/run/postgresql
 Server Port           | 5432
 Options               |
 Protocol Version      | 3.0
 Password Used         | false
 GSSAPI Authenticated  | false
 Backend PID           | 3891
 SSL Connection        | false
 Superuser             | on
 Hot Standby           | off
(12 rows)
```

Выдаётся имя пользователя, из под которой было создано соединение (пройдена аутентификация). Команды `SET ROLE` и `SET SESSION AUTHORIZATION` не меняют результат `\conninfo`

Для переподсоединения в psql используется команда

```
\c имя_базы имя_пользователя хост порт
```

Если какие-то параметры не хочется указывать, а хочется использовать значения текущего соединения, то вместо параметра в его позиции нужно использовать символ тире. Тире в конце можно не указывать. Например:

```
\c - user1
You are now connected to database "postgres" as user "user1".
\c - - localhost
You are now connected to database "postgres" as user "user1" on host "localhost"
(address "127.0.0.1") at port "5432".
```

Если новое соединение не сможет быть установлено, то сохраняется прежнее соединение.

Получение справки по командам psql

- команды psql начинаются на обратный слэш ****
- параметры командной строки **psql --help**
- справка по командам psql **\?**
- список команд SQL **\h**
- текст команды, формируемых psql:

```
postgres=# \set ECHO_HIDDEN on
postgres=# \db
/******** QUERY *********/
SELECT spcname AS "Name",
  pg_catalog.pg_get_userbyid(spcowner) AS "Owner",
  pg_catalog.pg_tablespace_location(oid) AS "Location"
FROM pg_catalog.pg_tablespace
ORDER BY 1;
/************************/
       List of tablespaces
    Name    |  Owner   | Location
------------+----------+----------
 pg_default | postgres |
 pg_global  | postgres |
```


Получение справки по командам psql

После установки, PostgreSQL можно запустить на сервере psql без параметров и тогда psql подсоединится локально (через Unix-сокет) к базе данных postgres под ролью postgres.

команды psql начинаются на обратный слэш: "\"

параметры командной строки **psql --help**

справка по командам `psql` **\?**

список команд SQL **\h**

После **\h** можно ввести начальные слова команды и получить по этой команде помощь.

Чтобы посмотреть, какие команды SQL формирует psql, чтобы исполнить команды, начинающиеся на `\d` (describe - получить описание объекта), нужно установить параметр:

```
postgres=# \set ECHO_HIDDEN on
postgres=# \db
/******** QUERY *********/
SELECT spcname AS "Name",
  pg_catalog.pg_get_userbyid(spcowner) AS "Owner",
  pg_catalog.pg_tablespace_location(oid) AS "Location"
FROM pg_catalog.pg_tablespace
ORDER BY 1;
/************************/
       List of tablespaces
    Name    |  Owner   | Location
------------+----------+----------
 pg_default | postgres |
 pg_global  | postgres |
(2 rows)
```

История команд и постраничный вывод

- если результат не помещается в окне терминала используется постраничный вывод (pageing)
  - › используются утилиты операционной системы `more` или `less`
- утилита задаётся командой **`\setenv PAGER 'less -XS'`**
- постраничный вывод отключается командой **`\pset pager off`**
- в режиме постраничного вывода для less (после двоеточия): q - выход, z - вперёд, b - назад, h - помощь
- история команд и настройки хранится в файлах:

```
postgres@tantor:~$ psql -q
postgres=# \! ls .psql*
.psql_history  .psqlrc
postgres=# \! cat .psqlrc
\setenv PAGER 'less -XS'
\setenv PSQL_EDITOR '/usr/bin/mcedit'
\pset pager off
```


История команд и постраничный вывод

Если текст не помещается на экран, то используется функционал "постраничного вывода" (pager): вы увидите двоеточие в конце вывода команды.

По нажатии клавиши `<ENTER>` высветится еще одна строка.

Если нужно высветить следующую страницу, то после двоеточия нужно нажать клавишу "z"

Если вернуться к предыдущей странице - клавишу "b" (back).

Если хочется прервать вывод можно нажать "q" (quit).

Если хочется получить помощь и узнать, какие есть еще комбинации клавиш, то можно набрать после двоеточия букву "h" (help).

Отключить постраничный вывод можно командой **`\pset pager off`**

Постраничный вывод реализуется передачей результата вывода утилите операционной системы `less` или `more`.

История команд, по умолчанию, доступна по нажатию стрелок вверх/вниз на клавиатуре.

История команд, набранных интерактивно, psql сохраняет в файле **`~/.psql_history`**.

Местоположение этого файла задаётся переменными окружения `HISTFILE` или `PSQL_HISTORY`.

Рядом с **`~/.psql_history`** лежит файл `~/.bash_history` с историей команд терминала операционной системы.

Названия файлов, начинающиеся с точки, считаются "скрытыми", то есть команда `ls` без параметра `-a` не показывает такие файлы.

`psql` работает лучше с серверами своей версии. **При подключении к более новой или сильно старой версии PostgreSQL, могут отказаться работать команды psql (которые начинаются на обратный слэш).**

Форматирование вывода в psql

- Можно настроить формат вывода параметрами `\pset` и переключателями `\a \t \x`
- посмотреть текущие настройки форматирования: `\pset`
- отображение времени выполнения команды `\timing on`

```
postgres=# select * from pg_am limit 1;
 oid | amname |      amhandler        | amtype
-----+--------+-----------------------+--------
   2 | heap   | heap_tableam_handler | t
(1 row)
postgres=# \pset linestyle unicode
Line style is unicode.
postgres=# select * from pg_am limit 1;
 oid │ amname │      amhandler        │ amtype
─────┼────────┼───────────────────────┼────────
   2 │ heap   │ heap_tableam_handler │ t
(1 row)
```


Форматирование вывода в psql

Можно посмотреть текущие настройки форматирования, набрав команду `\pset`

Если нужно повторять команду через интервалы времени, можно использовать команду: `\watch секунд count=число min_rows=строк`, выход ctrl+c или автоматически по достижении `count` или, если запрос выдаст строк меньше, чем задано в опциональном параметре `min_rows`;

`\a` включить или выключить выравнивание столбцов по вертикали;

`\t` включить/выключить отображение заголовка и итоговой строки (header and footer)

`\x` включить/выключить вывод в расширенном виде (построчно)

При выполнении долгих запросов и сравнении скорости выполнения, удобно включить отображение времени выполнения:

```
postgres@tantor:~$ psql -c "select 'abc' name" -x
Pager usage is off.
-[ RECORD 1 ]
name | abc
postgres@tantor:~$ psql -q
postgres=# \x on \\ select 'abc' name; \x off
-[ RECORD 1 ]
name | abc
```

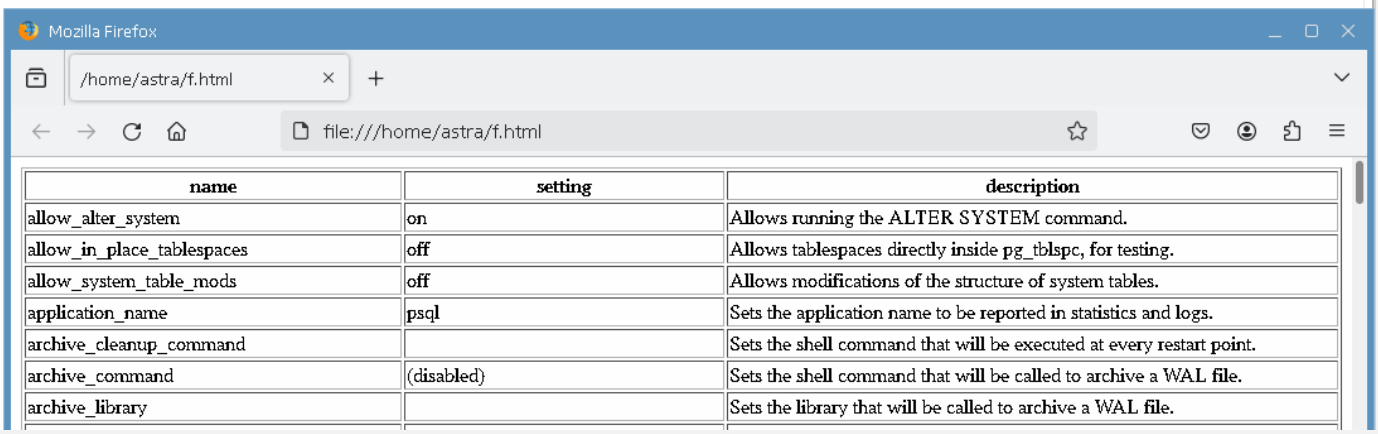
Параметр **psql -q** подавляет информационные сообщения, **psql** будет выводить только результаты запросов. Пример информационных сообщений:

```
postgres@tantor:~$ psql
Pager usage is off.
psql (18.3)
Type "help" for help.
postgres=# \x off \\
select 'abc' name;
\x on
Expanded display is off.
 name
------
 abc
Expanded display is on.
```

Вывод результата запроса в формате HTML

- можно выводить результаты запросов в формате HTML
  - **`psql -html`** или **`psql -H`**
  - или **`\pset format html`**

```
astra@tantor:~$ psql -c "show all;" -H -o f.html | xdg-open f.html
astra@tantor:~$
```



name	setting	description
allow_alter_system	on	Allows running the ALTER SYSTEM command.
allow_in_place_tablespaces	off	Allows tablespaces directly inside pg_tblspc, for testing.
allow_system_table_mods	off	Allows modifications of the structure of system tables.
application_name	psql	Sets the application name to be reported in statistics and logs.
archive_cleanup_command		Sets the shell command that will be executed at every restart point.
archive_command	(disabled)	Sets the shell command that will be called to archive a WAL file.
archive_library		Sets the library that will be called to archive a WAL file.

Вывод результата запроса в формате HTML

Если количество столбцов большое и терминальный клиент с пропорциональным шрифтом не удобен для отображения, то `psql` может сформировать результат не в текстовом формате, а в формате HTML. За это отвечает параметр **`-html`** или **`-H`** или **`\pset format html`**

Пример команды, посылающей SQL команду на выполнение и запускающую браузер с результатом в формате HTML:

`psql -c "команда;" -H -o f.html | xdg-open f.html`

Одной строкой можно получить результат больших выборок в читаемом формате.

Эта практичная команда может оказаться удобнее и быстрее в выполнении, чем использование графических утилит типа pgAdmin, а также в случаях, если графические утилиты не установлены в операционной системе.

Приглашение к вводу команд (промпт) psql

- Приглашение (промпт) можно менять командами **\set PROMPT1**
- "*" открыта и не зафиксирована транзакция
- "'" символ, парный которому нужно ввести, чтобы прекратить ввод
- "-" вторая и последующие строки (PROMPT2)
- "!" транзакция в состоянии сбоя и может только откатиться, даже если набрать **commit;**

```
postgres@postgres =# begin;
BEGIN
postgres@postgres *=# select '
postgres@postgres *'# abcd'
postgres@postgres *-# name;
 name
------
     +
 abcd
(1 row)
postgres@postgres *=# sele;
ERROR:  syntax error at or near
LINE 1: sele;
        ^
postgres@postgres !=# commit;
ROLLBACK
```


Приглашение к вводу команд (промпт) psql

Случается, что администратор дал команду "не в том окне".
Уменьшить вероятность таких случаев, помогает изменение приглашения psql (промпта).
Приглашение к вводу команд (промпт) имеет значения по умолчанию, которые различают первую набранную строку в команде и последующие.

По умолчанию, `PROMPT2` отличается от `PROMPT1` незаметными символами: = и -. Стоит обращать на них внимание.
`PROMPT1, PROMPT2` и `PROMPT3` задают внешний вид приглашения.
`PROMPT1` выдаётся, когда `psql` ожидает ввода новой команды.
`PROMPT2` если в буфере есть строка, например, потому, что команда не была завершена точкой с запятой или не были закрыты кавычки.
Типичным вопросом является: за что отвечает третий промпт?
`PROMPT3` выдаётся при выполнении команды `COPY имя FROM stdin`, когда в терминале вводятся данные для вставки в таблицу. Завершает такой режим `\.<ENTER>`
Этот режим редко используется, поэтому третий промпт не меняют и забывают, за что он отвечает.

```
postgres=# copy t from stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself, or an EOF signal.
>> \.
COPY 0
```

Удобно менять эти приглашения в файле **~\.psqlrc** чтобы видеть, к какой базе кластера подсоединены.
Пример установки цветного промпта:

```
\set PROMPT1 '%[%033[0;31m%]%n%[%033[0m%]@%[%033[0;36m%]%/%[%033[0m%]
%[%033[0;33m%]%[%033[5m%]%x%[%033[0m%]%[%033[0m%]%R%# '
\set PROMPT2 '%[%033[0;31m%]%n%[%033[0m%]@%[%033[0;36m%]%/%[%033[0m%]
%[%033[0;33m%]%[%033[5m%]%x%[%033[0m%]%[%033[0m%]%R%# '

user1@db01 =>
```

Переменные окружения, на которые реагирует psql

- Переменные окружения операционной системы, на которые реагирует psql
- Популярные переменные: `PGUSER`, `PGDATABASE`, `PGHOST`, `PGPORT`, `PSQL_EDITOR`
- Можно устанавливать интерактивно или в файлах параметров (`~/.psqlrc`)

```
postgres@tantor:~$ export PGUSER=user1
postgres@tantor:~$ export PGDATABASE=db01
postgres@tantor:~$ psql -q
db01=> select user;
 user
-------
 user1
(1 row)
```


Переменные окружения, на которые реагирует psql

В разделе документации "Окружение" https://docs.tantorlabs.ru/tdb/ru/18_1/se/app-psql.html#APP-PSQL-ENVIRONMENT указаны переменные окружения операционной системы, на которые реагирует psql.

Популярные переменные: **`PGUSER PGDATABASE PGHOST PGPORT`.** Они позволяют настроить подсоединение `psql` без указания параметров к любой базе.

Переменные окружения операционной системы можно устанавливать командой `\setenv` в в файле `~/.psqlrc` или глобальном файле `/opt/tantor/db/18/etc/postgresql/psqlrc`. Командами **`\set`, `\pset`, `\! export`** переменные **окружения** не устанавливаются.

По умолчанию, для команд редактирования **`\ef \ev \e`** используется редактор **`vi`**.

Переопределить редактор можно, установив переменную окружения. Пример:

`export PSQL_EDITOR=/usr/bin/mcedit`

Вместо имени `PLQL_EDITOR` можно использовать имена `EDITOR` или `VISUAL`.

Также, в psql можно интерактивно дать команду:

`\setenv PSQL_EDITOR /usr/bin/mcedit`

Можно вставить эту команду в файл `~/.psqlrc` или глобальный файл `/opt/tantor/db/18/etc/postgresql/psqlrc`

Переменные psql \set

- устанавливаются командой **\set имя значение**
- Срок жизни до выхода из psql или до выполнения команды **\unset имя**

```
postgres=# \set test1 'select user'
postgres=# :test1;
   user
----------
 postgres
(1 row)
```


Переменные psql \set

Переменные psql устанавливаются командой **\set имя значение**.
Командой **\set** можно посмотреть переменные psql (internal variables).
Названия **переменных чувствительны к регистру**. Часть переменных управляет работой psql и имеет значения по умолчанию. Срок жизни до выхода из psql или до выполнения команды **\unset** имя. Можно устанавливать свои переменные и пользоваться ими, как макросами.
Ссылаться на переменные можно, префиксируя их символом двоеточия, пример:

```
postgres=# \set test1 'select user'
postgres=# :test1;
   user
----------
 postgres
(1 row)
postgres=# select * from (:test1);
```

Выполнение команд в psql

- посмотреть переменные psql: команда **\set**
- очистка буфера набранных команд **\r**
- просмотр буфера или последней команды если буфер пуст **\p**
- **;** завершает команду SQL
- **\gx** завершает команду SQL, результат будет выведен в расширенном формате

```
postgres=# select 'select 123 c' abc\gset
postgres=# :abc\gx
-[ RECORD 1 ]
c | 123
postgres=# set my.abcd = 123;
postgres=# show my.abcd\gx
-[ RECORD 1 ]
my.abcd | 123
```

```
postgres=# select '
postgres'# \r
postgres'# '
postgres-# \r
postgres=# select (
postgres(# \r
postgres=# select "
postgres"# \r
postgres"# "
postgres-# \r
```


Выполнение команд в psql

Команда, начинающаяся на обратный слэш "\" обрабатывается psql. Посмотреть справку по таким командам можно командой **\?**

Стоит отличать команды **\set**, **\pset**, **set**.

\pset - устанавливает предопределённые параметры форматирования вывода утилиты psql.

Командой **\set** можно посмотреть переменные psql (internal variables). Эти переменные чувствительны к регистру. Часть переменных управляет работой psql и имеет значения по умолчанию. Можно устанавливать на время пока работает утилита psql свои переменные и пользоваться ими, как макросами.

Остальные команды посылаются как текст серверному процессу. Для отправки команды нужно ввести ";" и возврат каретки (клавиша *<ENTER>* на клавиатуре).

Команда **set** устанавливает значение параметра конфигурации в памяти серверного процесса (а не утилиты psql) на уровне сессии (**set session**) или транзакции (**set local**). Можно сохранить в памяти серверного процесса и произвольную переменную, в названии такой переменой должна быть точка. Установить значение параметра можно функцией `set_config()`. Прочесть значение можно командой `show` или функцией `current_setting()`. Сбросить в значение по умолчанию можно командой `reset`.

В psql есть команды **\g**, **\gx**, **\gexec**, **\gset**, **\g**, которые могут заменять ";", но эти команды работают только в psql.

Если не набрать ";", а просто набрать возврат каретки (клавишу <ENTER>), то psql считает, что команда многострочная и предыдущие строки накапливаются в буфере. Промпт утилиты psql при этом изменится: значок "=" в промпте заменится на "-". Для очистки буфера можно будет набрать **\r** (сокращение от **\reset**), но только, если psql не ожидает закрывающего значка "'" или "":

```
postgres=# select '
postgres'# '
postgres-# \r
postgres=# select "
postgres"# "
postgres-# \r
postgres=#
```

Посмотреть содержимое буфера или последней команды если буфер пуст **\p** (сокращение от **\print**)

Параметр ON_ERROR_ROLLBACK

- Делает интерактивныйю работу с транзакциями в psql удобной
- Ошибка в команде не переводит транзакцию в состояние сбоя, а откатывает ошибочную команду и транзакция продолжается

```
postgres=#
\set ON_ERROR_ROLLBACK off
postgres=# begin transaction;
BEGIN
postgres=*# slect 1;
ERROR:  syntax error at or near
"slect"
postgres=!# commit;
ROLLBACK
```

```
postgres=#
\set ON_ERROR_ROLLBACK interactive
postgres=# begin transaction;
BEGIN
postgres=*# slect 1;
ERROR:  syntax error at or near
"slect"
postgres=*# commit;
COMMIT
```


Параметр ON_ERROR_ROLLBACK

В psql есть параметр:
`postgres=# \set ON_ERROR_ROLLBACK interactive`
По умолчанию установлен в значение **off** (отключён). Если в транзакции команда выдаст ошибку, то транзакция откатится, зафиксировать её не удастся. Команды транзакции уже поменяли структуры хранения, но это нельзя будут зафиксировать, придётся заново повторять команды в новой транзакции. При возникновении ошибки, транзакция перходит в состояние сбоя, на что указывает значок "!" в промпте psql. На любую команду завершения транзакции (commit, end, rollback) выдаётся сообщение об откате транзакции (ROLLBACK).
При использовании значения **interactive**, при интерактивной работе в psql перед каждой командой в открытой транзакции psql будет устанавливать точку сохранения.
За счёт этого, в случае любой ошибки (например, опечатки в команде), последняя команда будет откатываться. Это делает работу в psql более удобной.
Устанавливать значение '**on**' не стоит, так как при выполнении скриптов (неинтерактивно) если в них будут открываться транзакции или будет отключен режим автофиксации, будут устанавливаться точки сохранения. Это существенно замедлит выполнение команд и будет излишне расходовать номера транзакций.

Автоматическая фиксация транзакций

- по умолчанию psql работает в режиме автоматической фиксации транзакции (AUTOCOMMIT):

```
postgres=# \set ON_ERROR_ROLLBACK interactive
postgres=# \set AUTOCOMMIT off
postgres=# create table t (c text);
CREATE TABLE
postgres=*# vacuum t;
ERROR:  VACUUM cannot run inside a transaction block
postgres=*# select * from t\gx
(0 rows)
postgres=*# rollback;
ROLLBACK
postgres=*# \d t
Did not find any relation named "t".
```


Автоматическая фиксация транзакций

По умолчанию, psql работает в режиме автоматической фиксации транзакции. Режим автофиксации, по умолчанию, используется в программах на языке Java по спецификации JDBC. Oracle Database в терминальном клиенте `sqlplus`, по умолчанию, не использует режим автофиксации.

Режим автофиксации означает, что серверный процесс самостоятельно и автоматически фиксирует выполнение команды. Клиент, в том числе утилита psql, не передаёт отдельно команду фиксации транзакции, чтобы на передачу не тратилось время, равное сетевой задержке.

Клиент может отключить режим автофиксации, тогда серверный процесс не будет самостоятельно фиксировать транзакцию. Команды будут открывать транзакции, за исключением команд, которые не могут выполняться в транзакции (создание и удаление базы данных, вакуумирование, изменение параметров конфигурации кластера).

Отключить или включить режим афтофиксации можно командой psql:

`\set AUTOCOMMIT on`

`\set AUTOCOMMIT off`

Команду можно указать в системном файле `psqlrc` или в `~/.psqlrc` или выполнить в сессии psql.

Выполнение командных файлов в psql

- `\! linux_command` - выполнить команду операционной системы
- `\o file.sql` - перенаправить вывод в файл
- `\o` - вернуть вывод на экран
- `\i file.sql` - выполнить команды из файла
- Еще примеры как выполнить команды из файла:
  ```
  psql < file.sql
  psql -f file.sql
  ```
- Выполнить каждую строку, формируемую запросом SELECT как команду в psql:
  ```
  SELECT 'checkpoint;' \gexec
  ```


Выполнение командных файлов в psql

В psql можно выполнить команду операционной системы, не выходя из psql. Для этого используется команда **\!** `команда_линукс`

Для вывода результата выполнения команд (POSIX output stream) в файл операционной системы можно использовать команду **\o** `имя_файла`. На экран при этом результаты выдаваться не будут.

Для выполнения командного файла можно использовать **\i** `имя_файла`

```
\o checkpoint.sql \\
select 'checkpoint;'\g (tuples_only=on format=unaligned)
```

\o вернуть вывод на экран

```
\i checkpoint.sql
```

Также команды из файла (скрипта) можно выполнить так:

```
psql < checkpoint.sql
psql -f checkpoint.sql
```

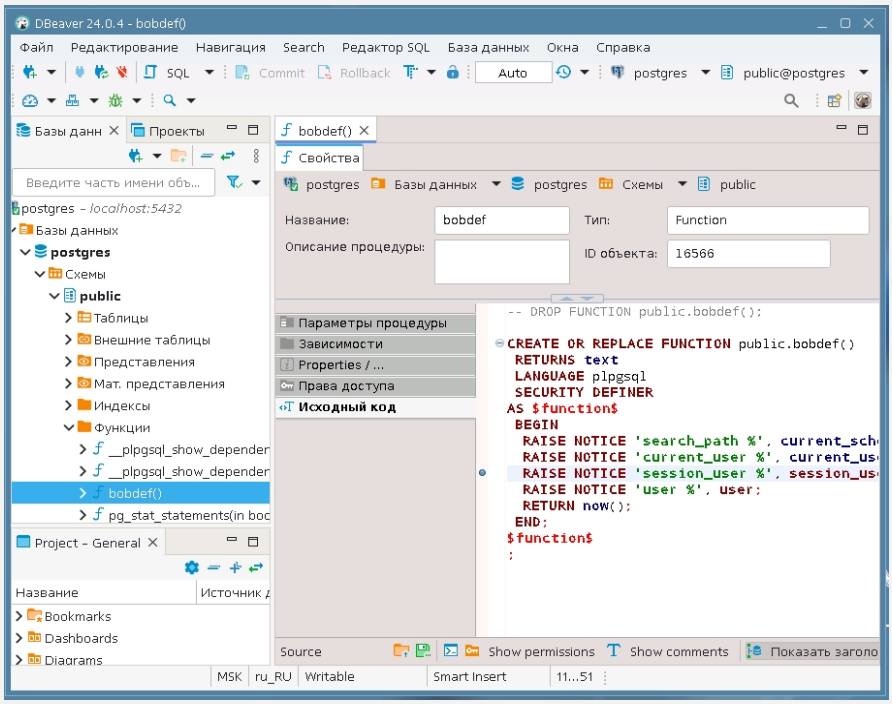
При этом ставить последней в файле команду выхода необязательно, psql сам закончит работу, дойдя до конца файла (в отличие от утилиты sqlplus в Oracle Database).

Более того, можно сформировать команды и выполнить их не создавая промежуточный файл скрипта. Для этого используется опция **\gexec**

```
postgres=# select 'checkpoint;' \gexec
CHECKPOINT
```

Графические приложения: DBeaver

- имеет свободно распространяемую версию
- написан на java



Графические приложения: DBeaver

Популярно универсальное (для разработки и для администрирования) приложение DBeaver, имеющее бесплатную версию.

Приложение можно скачать командой:

```
wget https://dbeaver.io/files/dbeaver-ce_latest_amd64.deb
```

и установить командой:

```
sudo dpkg -i dbeaver-ce_latest_amd64.deb
```

Запустить приложение можно из меню Пуск -> Development -> dbeaver-ce или командой:

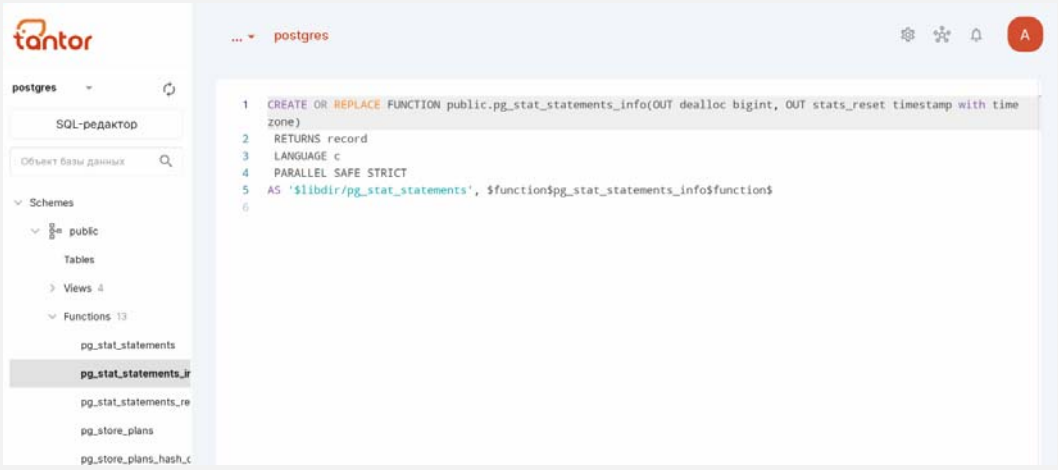
```
/usr/bin/dbeaver-ce
```

DBeaver позволяет отлаживать хранимые процедур и функции, используя интерфейс расширения pldebugger.

Для разработки приложений, также может использоваться коммерческое приложение DataGrip от JetBrains, которое интегрируется со средами разработки этой компании: IntelliJ IDEA и PyCharm. Интеграция позволяет проверять синтаксис и автозавершение команд SQL при написании программного кода.

Графические приложения: Платформа Tantor

- приложение для администрирования и мониторинга большого числа кластеров PostgreSQL и кластеров Patroni
- приложение класса Enterprise Manager



Графические приложения: Платформа Tantor

Платформа Tantor - программное обеспечение для управления любыми СУБД, основанными на PostgreSQL, а также кластерами Patroni. Позволяет удобно управлять большим числом кластеров. Относится к классу программных продуктов, в который входит Oracle Enterprise Manager Cloud Control.

Платформа Tantor активно развивается, чтобы удовлетворять потребностям администрирования PostgreSQL.

В Платформе Tantor есть SQL-редактор, в котором можно просматривать объекты, выполнять команды, создавать процедуры и функции.

https://docs.tantorlabs.ru/tp/6.1/instances/DB_browser.html

Демонстрация

- Скачивание инсталлятора
- Установка разрешения на исполнение инсталлятора
- Установка адреса расположения дистрибутивов
- Установка с созданием базы данных
- Проверка, что кластер работает
- Остановка служб
- Деинсталляция

Демонстрация

Скачивание инсталлятора
Установка разрешения на исполнение инсталлятора
Установка адреса расположения дистрибутивов
Установка с созданием базы данных
Проверка, что кластер работает
Остановка служб
Деинсталляция

Практика

1. Создание кластера
2. Создание кластера утилитой initdb
3. Режим одного пользователя
4. Передача параметров экземпляру в командной строке
5. Локализация
6. Однобайтные кодировки
7. Использование утилит управления
8. Настройка терминального клиента psql
9. Использование терминального клиента psql
10. Восстановление сохраненного кластера

Практика

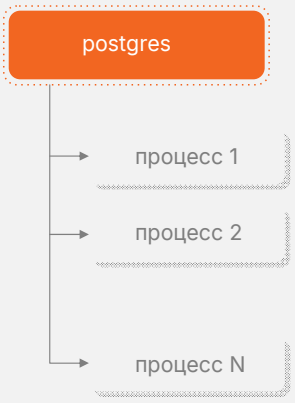
Создание кластера
Создание кластера утилитой initdb
Режим одного пользователя
Передача параметров экземпляру в командной строке
Локализация
Однобайтные кодировки
Использование утилит управления
Настройка терминального клиента psql
Использование терминального клиента psql
Восстановление сохраненного кластера

2

2a

Архитектура PostgreSQL

Экземпляр PostgreSQL



- экземпляр PostgreSQL - процесс postgres, порожденные им процессы операционной системы, память, которую используют эти процессы
- пример процессов:

```
postgres@tantor:~$ ps -eLo ppid,pid,cmd | egrep 'PPID|postgres'
   PPID     PID CMD
      1  743184 /opt/tantor/db/18/bin/postgres                  основной процесс
 743184  743185 postgres: logger                               процесс записи в журнал logging collector
 743184  743186 postgres: checkpointer                        фоновый процесс контрольной точки
 743184  743187 postgres: background writer                   процесс фоновой записи
 743184  743189 postgres: walwriter                           процесс фоновой записи в журнал
 743184  743190 postgres: autovacuum launcher                 процесс запуска автовакуума
 743184  743191 postgres: pg_stat_advisor BackgroundTaskManager процесс расширения
 743184  743192 postgres: autoprewarm leader                  процесс расширения pg_prewarm
 743184  743193 postgres: logical replication launcher        процесс запуска логической репликации
 644740  795748 psql -d demo -U alice -h /var/run/postgresql  клиент, утилита psql
 743184  795749 postgres: alice demo [local] idle             процесс, обслуживающий psql
```


Экземпляр PostgreSQL

Процесс postgres (устаревшее название postmaster) - процесс, обслуживающий PostgreSQL (database server). Это первый процесс, который запускается, прослушивает порты сетевых интерфейсов и создаёт файл Unix-сокета, через который принимает локальные соединения. Этот процесс запускает (порождает, fork) другие процессы и является их родительским процессом. Это серверные (традиционное название - backend) процессы, обслуживающие сессии клиентов и фоновые (background) процессы, которые выполняют полезные задачи по обслуживанию кластера баз данных.

Кластер баз данных PostgreSQL - набор баз данных, хранящихся в файловой системе в директории PGDATA в виде наборов файлов. Один экземпляр процесса postgres обслуживает один кластер баз данных, а кластер баз данных может обслуживаться только одним экземпляром (кроме Tantor Polar). В операционной системе может работать несколько экземпляров postgres, обслуживающие каждый свой кластер баз данных. Экземпляры postgres должны использовать разные порты, как сетевых интерфейсов, так и разные файлы Unix-сокетов.

Экземпляр PostgreSQL - процесс postgres, порожденные им процессы операционной системы, память, которую используют эти процессы. У каждого процесса есть локальная память, доступ к которой имеет только этот процесс и разделяемая память (shared memory), доступ к которой имеет несколько процессов или даже все процессы экземпляра.

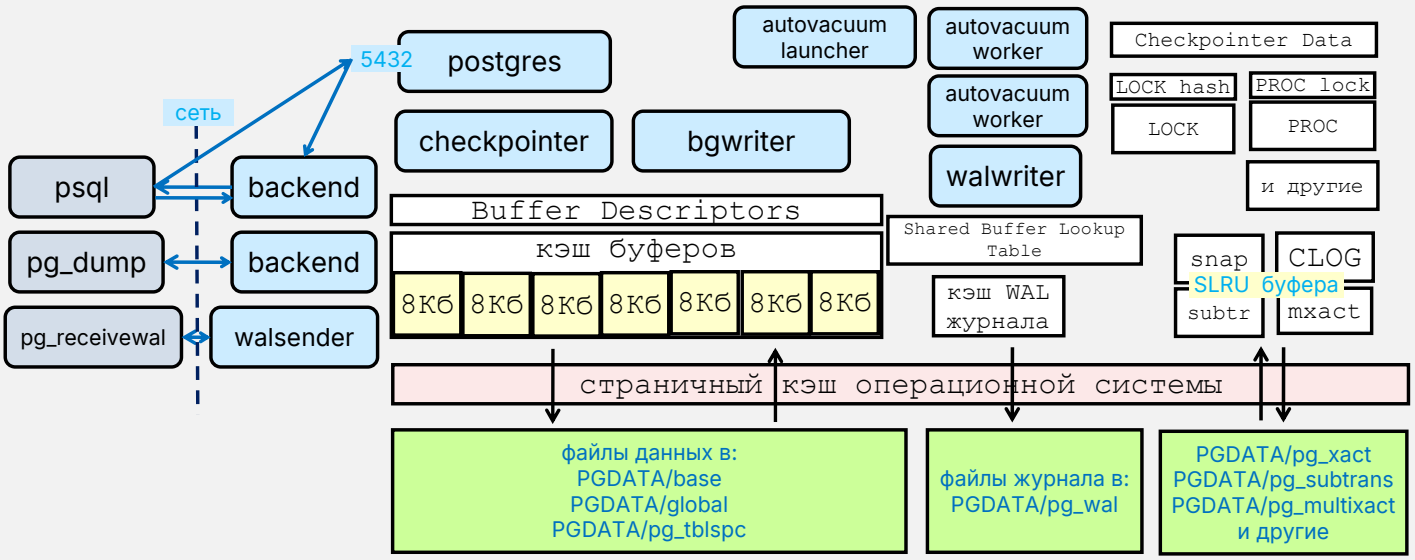
Список процессов экземпляра PostgreSQL:

```
postgres@tantor:~$ ps -eLo ppid,pid,cmd | egrep 'PPID|postgres'
   PPID     PID CMD
      1  743184 /opt/tantor/db/18/bin/postgres                  основной процесс
 743184  743185 postgres: logger                               процесс записи в журнал logging collector
 743184  743186 postgres: checkpointer                        фоновый процесс контрольной точки
 743184  743187 postgres: background writer                   процесс фоновой записи
 743184  743189 postgres: walwriter                           процесс фоновой записи в журнал
 743184  743190 postgres: autovacuum launcher                 процесс запуска автовакуума
 743184  743191 postgres: pg_stat_advisor BackgroundTaskManager процесс расширения
 743184  743192 postgres: autoprewarm leader                  процесс расширения pg_prewarm
 743184  743193 postgres: logical replication launcher        процесс запуска логической репликации
 644740  795748 psql -d demo -U alice -h /var/run/postgresql  клиент, утилита psql
 743184  795749 postgres: alice demo [local] idle             процесс, обслуживающий psql
```

Клиент подсоединился через Unix-сокет к базе данных demo под пользователем alice. Клиент обслуживается своим серверным процессом с номером процесса 795749. Остальные процессы экземпляра фоновые.

Процессы экземпляра PostgreSQL

- процессы, память, кластер



Процессы экземпляра PostgreSQL

В PostgreSQL нет строгого закрепления действий за процессами. Серверные процессы могут читать файлы с данными в память (буферный кэш), посылать в операционную систему блоки на запись, посылать на запись из журнального буфера в журнальные файлы, выполнять вакуумирование по команде VACUUM.

Основные ресурсы, которые использует экземпляр: диск, память, процессор, сеть. Наиболее нагруженный ресурс: диск. Чтобы уменьшить нагрузку, используется кэширование содержимого файлов данных в кэше буферов. Кэш буферов - структура в разделяемой памяти, обычно, имеющая самый большой размер, поэтому кэшу буферов и вспомогательным структурам памяти и обслуживающим его процессам уделяется больше внимания. Это процессы контрольной точки (checkpointer) и фоновой записи (background writer, bgwriter). Все изменения в данных выполняются через кэш буферов, прямых изменений в файлах данных нет. Для временных таблиц используется аналог буферного кэша, но только в памяти серверного процесса.

Кэш буферов - это кэш на чтение и запись (изменения удерживаются в памяти). Отказоустойчивость реализуется журналированием изменений, которые вносятся в блоки, находящиеся в кэше буферов и SLRU-буферах.

Журнал называется WAL (Write Ahead Log, журнал предварительной записи) и состоит из файлов размером по 16Мб (по умолчанию). В файлы журналов пишут серверные процессы и любые другие процессы, которые вносят изменения в данные, но есть и вспомогательный процесс walwriter.

Набор фоновых процессов автовакуума обслуживает отдельную задачу - удаления устаревших данных.

Процесс startup после того как завершит восстановление останавливается.

Процессы walsender запускаются при подсоединении клиентов (pg_basebackup, pg_receivewal, процессы реплик walreceiver) по протоколу репликации.

Запуск экземпляра, процесс postgres

- запускается процесс postgres
- считываются файлы параметров и комбинируются с параметрами, переданными в командной строке
- проверяются разрешения на директорию PGDATA
- проверяется наличие управляющего файла `pg_control`
- выделяется память, загружаются разделяемые библиотеки
- в PGDATA создается файл postmaster.pid наличие которого и правильность номера процесса проверяется раз в минуту
- процесс postgres регистрирует серверные сокеты, создается файл UNIX-сокета
- читается файл с параметрами аутентификации `pg_hba.conf`
- запускается процесс startup и фоновые процессы

Запуск экземпляра, процесс postgres

Основные шаги при запуске экземпляра:

1. Запускается процесс postgres ("postmaster");
2. Считываются файлы параметров конфигурации, параметры комбинируются с параметрами командной строки и переменными окружения;
3. Проверяются права на директорию PGDATA, они должны быть `0700` или `0750`;
4. Проверяется наличие управляющего файла pg_control, текущей директорией для процесса устанавливается PGDATA, в ней создается файл postmaster.pid, инициализируется TLS, загружаются разделяемые библиотеки, указанные в параметре `shared_preload_libraries`, регистрируется обработчик на случай исчезновения процесса для корректной остановки дочерних процессов, инициализируется (по параметрам конфигурации) менеджер памяти, регистрируется обработчик закрытия сетевых сокетов.

В postmaster.pid первая строка хранит PID запущенного postmaster. **Файл проверяется раз в минуту. Если файла нет или хранящийся в нем PID не равен PID процесса , то процесс postgres остановится по сигналу SIGQUIT**

5. регистрируются сокеты по всем адресам (параметр конфигурации `listen_addresses`). Создается файл UNIX-сокета
6. Считывается файл с настройками аутентификации `pg_hba.conf`
7. Запускается процесс startup, который определяет состояние кластера по управляющему файлу `pg_control` (если директория PGDATA не была восстановлена из бэкапа, то есть нет файла `backup.label`), выполняет восстановление кластера, если нужно. Экземпляр открывается на чтение-запись, если кластер не является физической репликой (нет файла `standby.signal`).
8. Пока процесс startup разбирается что ему делать, postgres запускает остальные фоновые процессы.

Серверные процессы запускаются, если есть запрос на создание сессии от клиентов.

Все порожденные процессы, в том числе серверные, периодически проверяются на существование.

Запуск серверного процесса

- для каждой сессии создается серверный процесс
- Инициализируются (выделяются и заполняются) три кэша в локальной памяти серверного процесса:
  - Кэш для быстрого доступа к таблицам (RelationCache)
  - Кэш таблиц системного каталога (CatalogCache)
  - Кэш планов выполнения команд (PlanCache)
- выделяется память под менеджер "порталов" TopPortalContext
- клиент аутентифицируется
- догружаются разделяемые библиотеки, указанные в параметрах `session_preload_libraries` и `local_preload_libraries`
- выделяется память MessageContext для текста команд и процесс готов к приему команд

Запуск серверного процесса

Серверный процесс запускается процессом `postgres`, если клиент хочет подсоединиться (пришел запрос на порт серверного сокета или Unix-сокет).

Основные шаги при запуске серверного процесса:

1. При запуске процесс получает структуру (часть памяти) PGPROC из списка свободных и устанавливает поля в начальные значения. Структуры PGPROC находятся в разделяемой памяти. PGPROC используют и фоновые процессы.

2. Процесс регистрирует таймауты в соответствии со значениями параметров конфигурации, которые можно посмотреть командой:

```
psql -c"\dconfig *_timeout"
```

чтобы серверный процесс мог прерваться при превышении значений этих параметров

3. Инициализируются три кэша в локальной памяти серверного процесса:

Кэш для быстрого доступа к таблицам (RelationCache)

Кэш таблиц системного каталога (CatalogCache)

Кэш планов выполнения команд (PlanCache)

4 . Выделяется память под менеджер "порталов" TopPortalContext. Портал - исполняющийся запрос, появляющийся в расширенном протоколе на этапе привязки, после парсинга. Порталы могут быть именованными (например, названием курсора) или безымянными - SELECT.

6. Обновляются значения параметров конфигурации. которые устанавливаются на этапе соединения. Выполняется задержка по параметру `post_auth_delay`

7. Обновляется структура PgBackendStatus.

8. Клиенту отправляются параметры: версия сервера, временная зона, параметры локализации, форматы типов данных, пара порядковый номер процесса (id) и токен отмены, по которым клиент может отменить выполнение запроса.

9. Серверный процесс загружает библиотеки, заданные в параметрах загружаются `session_preload_libraries` и `local_preload_libraries`. В процессе загрузки проверяется совместимость библиотек с версией PostgreSQL. Если библиотека была загружена ранее (`shared_preload_libraries`), то процесс просто получает указатель на загруженную библиотеку.

10. Выделяется память под обработку сообщений от клиента

11. Клиенту отправляется сообщение ReadyForQuery - серверный процесс готов принимать команды от клиента.

Общая память процессов экземпляра

- большее 70 структур памяти
- размеры и список - в представлении `pg_shmem_allocations`
- размер части структур задаётся параметрами конфигурации

```
select * from (select *, lead(off) over(order by off) - off as true_size from
pg_shmem_allocations) as a order by 1;
          name          |    off    |   size    | allocated_size | true_size
------------------------+-----------+-----------+----------------+----------
 <anonymous>            |           |   4946048 |        4946048 |
 Archiver Data          | 147726208 |         8 |            128 |       128
...
 XLOG Recovery Ctl      |   4377728 |       104 |            128 |       128
                        | 148145024 |   2849920 |        2849920 |
(77 rows)
```


Общая память процессов экземпляра

Примеры структур в разделяемой (общей) памяти экземпляра:
`Proc Array, PROC, PROCLOCK, Lock Hashes, LOCK, Multi-XACT Buffers, Two-Phase Structs, Subtrans Buffers, CLOG Buffers (transaction), XLOG Buffers, Shared Invalidation, Lightweight Locks, Auto Vacuum, Btree Vacuum, Buffer Descriptors, Shared Buffers, Background Writer Synchronized Scan, Semaphores, Statistics`. Всего в 18 версии 77 структур плюс структуры библиотек расширений.

К этим структурам имеют доступ процессы экземпляра. Расширения могут создавать собственные структуры. Список структур и их размеров:

```
select * from (select *, lead(off) over(order by off)-off as true from
pg_shmem_allocations) as a order by 1;
          name          |    off    |   size    | allocated_size | true_size
------------------------+-----------+-----------+----------------+----------
 <anonymous>            |           |   4946048 |        4946048 |
 Archiver Data          | 147726208 |         8 |            128 |       128
...
 XLOG Recovery Ctl      |   4377728 |       104 |            128 |       128
                        | 148145024 |   2849920 |        2849920 |
(77 rows)
```

Строка с пустым (NULL) именем отражает неиспользуемую память. Строка с именем "<anonymous>" отражает суммарный размер структур, память под которые выделялась без назначения имени.

В представлении не показываются структуры, выделяемые и освобождаемые "динамически" - в процессе работы экземпляра. Динамические структуры общей памяти используются рабочими процессами (workers). Рабочие процессы, например, используются для параллельного выполнения команд SQL.

Два вида структуры разделяемой памяти экземпляра могут использовать огромные страницы (HugePages): кэш буферов (размер задаётся параметром конфигурации `shared_buffers`) и память, выделяемая фоновыми процессами (память для них резервируется параметром конфигурации `min_dynamic_shared_memory`).

Кэш таблиц системного каталога

- выделяется в локальной памяти каждого процесса в контексте CacheMemoryContext
- при создании или удалении объекта, процесс зафиксировавший транзакцию, посылает сообщение в кольцевой буфер shmInvalBuffer разделяемой памяти
- буфер хранит до 4096 сообщений
- процессы потребляют сообщения и обновляют свои локальные кэши
- если процесс пропустит сообщения, то полностью очистит свой локальный кэш таблиц системного каталога (CatalogCache) и будет его заново заполнять

```
select * from (select *, lead(off) over(order by off) - off as true_size
from pg_shmem_allocations) as a where name='shmInvalBuffer' order by 1;
       name       |    off    | size  | allocated_size | true_size
------------------+-----------+-------+----------------+-----------
 shmInvalBuffer | 219072256 | 86816 |          86912 |     86912
(1 row)
```


Кэш таблиц системного каталога

CatalogCache выделяется в локальной памяти каждого процесса в контексте CacheMemoryContext. При обращении к таблицам системного каталога процесс ищет данные в этом кэше. Если данных не нашлось, то выбираются строки таблиц системного каталога и кэшируются. Для доступа к таблицам системного каталога используется индексный метод доступа. Если в таблице системного каталога не нашлась запись, то кэшируется признак отсутствия записи (negative entry). Например, ищется таблица, а такой таблицы нет, в локальном кэше процесса сохраняется запись, что таблицы с таким названием нет. Ограничений на размер CacheMemoryContext нет, это не круговой буфер и не стек.

При фиксации транзакции создающей, удаляющей, изменяющий объекта, приводящие к изменениям в таблицах системного каталога, процесс выполнивший изменения сохраняет сообщение о том, что объект изменён, в кольцевой буфер shmInvalBuffer в разделяемой памяти. Буфер может сохранить до 4096 сообщений.

Размер буфера:

```
select * from (select *, lead(off) over(order by off) - off as true_size from
pg_shmem_allocations) as a where name='shmInvalBuffer' order by 1;
       name       |    off    | size  | allocated_size | true_size
------------------+-----------+-------+----------------+-----------
 shmInvalBuffer | 219072256 | 86816 |          86912 |     86912
```

Если процесс не потребил половину сообщений, то ему передаётся уведомление, чтобы он потребил накопившиеся сообщения. Это позволяет уменьшить вероятность того, что процесс пропустит сообщения и ему придется очистить свой локальный кэш системного каталога. В разделяемой памяти сохраняются данные о том, какие процессы какие сообщения потребили. Если какой-либо процесс несмотря на уведомление не потребляет сообщения, а буфер заполнен, то процессу придётся полностью очистить свой кэш системного каталога.

Для того, чтобы кэши системного каталога процессов не сбрасывались слишком часто, нужно, чтобы объекты (в том числе временные таблицы) не создавались и удалялись слишком часто. Таблицы, в том числе временные не стоит часто создавать и удалять в течение сессии.

Статистика сброса кэшей и числа сообщений не собирается стандартными расширениями PostgreSQL.

Представление pg_stat_slru

- в PGDATA есть поддиректории, в которых сохраняются служебные данные кластера
- для ускорения доступа на чтение-запись в файлы этих директорий используются кэши в разделяемой памяти экземпляра
- статистика используется для установки параметров конфигурации, задающих размеры SLRU-кэшей

```
select name, blks_hit, blks_read, blks_written, blks_exists, flushes, truncates from pg_stat_slru;
       name       | blks_hit | blks_read | blks_written | blks_exists | flushes | truncates
------------------+----------+-----------+--------------+-------------+---------+-----------
 commit_timestamp |        0 |         0 |            0 |           0 |     103 |         0
 multixact_member |        0 |         0 |            0 |           0 |     103 |         0
 multixact_offset |        0 |         3 |            2 |           0 |     103 |         0
 notify           |        0 |         0 |            0 |           0 |       0 |         0
 serializable     |        0 |         0 |            0 |           0 |       0 |         0
 subtransaction   |        0 |         0 |           26 |           0 |     103 |       102
 transaction      |   349634 |         4 |           87 |           0 |     103 |         0
 other            |        0 |         0 |            0 |           0 |       0 |         0
(8 rows)
```


Представление pg_stat_slru

В PGDATA есть поддиректории, в которых сохраняются служебные данные кластера. Для ускорения доступа на чтение-запись в файлы этих директорий используются кэши в разделяемой памяти экземпляра. Файлы форматированы блоками по 8Кб. Кэши работают по простому алгоритму вытеснения давно неиспользуемых данных (simple least-recently-used, SLRU).

Статистику использования кэшей можно посмотреть в представлении:

```
select name, blks_hit, blks_read, blks_written, blks_exists, flushes, truncates
from pg_stat_slru;
```

```
       name       | blks_hit | blks_read | blks_written | blks_exists | flushes | truncates
------------------+----------+-----------+--------------+-------------+---------+-----------
 commit_timestamp |        0 |         0 |            0 |           0 |     103 |         0
 multixact_member |        0 |         0 |            0 |           0 |     103 |         0
 multixact_offset |        0 |         3 |            2 |           0 |     103 |         0
 notify           |        0 |         0 |            0 |           0 |       0 |         0
 serializable     |        0 |         0 |            0 |           0 |       0 |         0
 subtransaction   |        0 |         0 |           26 |           0 |     103 |       102
 transaction      |   349634 |         4 |           87 |           0 |     103 |         0
 other            |        0 |         0 |            0 |           0 |       0 |         0
```

В PostgreSQL начиная с 17 версии (в Tantor Postgres начиная с 15 версии) размеры кэшей SLRU настраиваются.

Статистику из представления можно использовать для установки параметров конфигурации, задающих размеры SLRU-кэшей: `\dconfig *_buffers`

```
          Parameter          | Value
-----------------------------+-------
 commit_timestamp_buffers    | 256kB
 multixact_member_buffers    | 256kB
 multixact_offset_buffers    | 128kB
 notify_buffers              | 128kB
 serializable_buffers        | 256kB
 shared_buffers              | 128MB
 subtransaction_buffers      | 256kB
 temp_buffers                | 8MB
 transaction_buffers         | 256kB
 wal_buffers                 | 4MB
```

https://docs.tantorlabs.ru/tdb/ru/18_1/se/monitoring-stats.html

Локальная память процесса

- доступна только одному процессу, поэтому блокировки для доступа к ней не нужны
- большая часть структур не занимает много памяти и интересны только для понимания алгоритмов работы процессов
- параметры, наиболее сильно влияющие на выделение локальной памяти процесса:
  - `work_mem` - выделяется для обслуживание узлов (шагов) плана выполнения (если шаги способны выполняться одновременно), в том числе каждым параллельным процессом. Вместе с параметром `hash_mem_multiplier` влияет на память, выделяемую каждым серверным и параллельным процессом.
  - `maintenance_work_mem` значение по умолчанию 64MB. Задает объем памяти, выделяемый каждым процессом (серверным, параллельным), участвующем в выполнении команд `VACUUM`, `ANALYZE`, `CREATE INDEX`, `ALTER TABLE ADD FOREIGN KEY`

Локальная память процесса

Примеры структур в локальной памяти серверного процесса: `RelationCache`, `CatalogCache`, `PlanCache`, `work_mem`, `maintenance_work_mem`, `StringBuffer`, `temp_buffers`

Локальная память доступна только одному процессу, поэтому блокировки для доступа к ней не нужны. Память выделяется под различные структуры ("контексты"). Для выделения и учета выделенной памяти используется универсальный набор функций, а не ситуативные вызовы к операционной системе. Большая часть структур не занимает много памяти и интересны только для понимания алгоритмов работы процессов. Интерес представляют те структуры, которые имеют большой размер или на размер которых можно влиять, например, параметрами конфигурации.

Параметры, наиболее сильно влияющие на выделение локальной памяти процесса:

`work_mem` - выделяется для обслуживание узлов (шагов) плана выполнения (если шаги способны выполняться одновременно), в том числе каждым параллельным процессом. Вместе с параметром `hash_mem_multiplier` влияет на память, выделяемую каждым серверным и параллельным процессом. Например, при соединении таблиц хэшированием (Hash Join) объем выделяемой памяти на обслуживание JOIN будет **`work_mem*hash_mem_multiplier*(Workers + 1)`**.

`maintenance_work_mem` значение по умолчанию 64MB. Задает объем памяти, выделяемый каждым процессом (серверным, параллельным), участвующем в выполнении команд `VACUUM`, `ANALYZE`, `CREATE INDEX`, `ALTER TABLE ADD FOREIGN KEY`. Число параллельных процессов ограничивается параметром `max_parallel_maintenance_workers`. Распараллеливается создание индексов и обычный (без FULL) вакуум. При вакуумировании только на фазе вакуумирования индексов (другие фазы не распараллеливаются) один индекс может обрабатывать один (а не несколько) параллельный процесс. Будут ли использоваться параллельные процессы зависит от размера индексов.

В Tantor Postgres есть параметры конфигурации, позволяющие настроить использование локальной памяти `enable_temp_memory_catalog` и `enable_large_allocations`.

Представление pg_backend_memory_contexts

- показывает память, выделенную серверным процессом, обслуживающим **текущую** сессию
- контекст памяти (memory contexts) - набор частей памяти (chunks), которые выделяются процессом для выполнения какой-то задачи
- для выполнения подзадачи может выделяться дочерний контекст
- Контексты образуют дерево (иерархию)
- в представлении иерархию отображают столбцы: name (название контекста памяти), parent (название родительского контекста памяти, level
- в столбце ident содержится детализация того, что хранится в контексте

```
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from pg_backend_memory_contexts;
   sum    |   sum    |  sum
----------+----------+--------
 2223144 | 1560928 | 662216
```


Представление pg_backend_memory_contexts

Представление показывает память, выделенную серверным процессом, обслуживающим **текущую** сессию. Контекст памяти (memory contexts) - набор частей памяти (chunks), которые выделяются процессом для выполнения какой-то задачи. Если памяти не хватает, она выделяется дополнительно. Для выполнения подзадачи может выделяться дочерний контекст. Контексты образуют дерево (иерархию). Корень дерева - TopMemoryContext. Цель такой организации выделения и учета памяти, чтобы при освобождении памяти не забыть освободить какую-нибудь часть, иначе возникнет "утечка" памяти. Когда освобождается контекст памяти, то освобождаются все дочерние контексты памяти.

В представлении `pg_backend_memory_contexts` иерархию отображают столбцы: name (название контекста памяти), parent (название родительского контекста памяти, level. В столбце ident содержится детализация того, что хранится в контексте. Пример иерархического запроса:

```
 WITH RECURSIVE context_tree AS (SELECT name, ident, path, level, total_bytes,
used_bytes, free_bytes, ARRAY[name] as context_path FROM
pg_backend_memory_contexts WHERE level = 1
 UNION ALL
 SELECT c.name, c.ident, c.path, c.level, c.total_bytes, c.used_bytes,
c.free_bytes, ct.context_path || c.name FROM pg_backend_memory_contexts c JOIN
context_tree ct ON c.path[1:array_length(c.path,1)-1] = ct.path)
 SELECT level, pg_size_pretty(total_bytes) as total, pg_size_pretty(used_bytes)
as used, context_path FROM context_tree ORDER BY level, context_path limit 3;
 level |   total    |   used    |             context_path
-------+------------+-----------+--------------------------------------
     1 | 97 kB      | 94 kB     | {TopMemoryContext}
     2 | 1024 kB    | 757 kB    | {TopMemoryContext,CacheMemoryContext}
     2 | 8192 bytes | 240 bytes | {TopMemoryContext,ErrorContext}
```

Память, выделенная текущему серверному процессу:

```
 select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
   sum    |   sum    |  sum
----------+----------+--------
 2223144 | 1560928 | 662216
```

Функция pg_log_backend_memory_contexts(PID)

- начиная с 17 версии у команды EXPLAIN есть опция memory (по умолчанию отключена), которая выдает сколько памяти использовал планировщик и общую память серверного процесса
- память чужих сессий можно вывести в диагностический лог кластера функцией:

```
postgres=# SELECT pg_log_backend_memory_contexts(pg_backend_pid());

LOG: logging memory contexts of PID 111
LOG: level: 1; TopMemoryContext: 99456 total in 5 blocks; 3072 free (8
chunks); 96384 used
LOG: level: 2; search_path processing cache: 8192 total in 1 blocks;
5656 free (8 chunks); 2536 used
...
LOG: Grand total: 1301048 bytes in 229 blocks; 346960 free (286
chunks); 954088 used
```


Функция pg_log_backend_memory_contexts(PID)

Память чужих сессий можно вывести в диагностический лог кластера функцией:
select pg_log_backend_memory_contexts(PID);
В журнал будут выведены сообщения:

```
LOG: logging memory contexts of PID 111
LOG: level: 1; TopMemoryContext: 99456 total in 5 blocks; 3072 free (8 chunks);
96384 used
LOG: level: 2; search_path processing cache: 8192 total in 1 blocks; 5656 free
(8 chunks); 2536 used
LOG: level: 2; RowDescriptionContext: 8192 total in 1 blocks; 6920 free (0
chunks); 1272 used
...
LOG: Grand total: 1301048 bytes in 229 blocks; 346960 free (286 chunks); 954088
used
```

Начиная с 17 версии PostgreSQL, в команду EXPLAIN добавлена опция memory (по умолчанию отключена), которая выдает сколько памяти использовал планировщик и общую память серверного процесса в виде строки в конце плана:

```
Memory: used=N bytes, allocated=N bytes
```

На этапе планирования при использовании большого (тысячи) числа секций секционированной таблицы может использоваться много памяти.

Структуры памяти, обслуживающие буферный кэш

- **Buffer Blocks** - сам буферный кэш
  - выделяется память по числу буферов*8192 байта плюс 4096 байт
  - параметр конфигурации `shared_buffers` задаёт число буферов
- **Buffer Descriptors** - описатели (заголовки) буферов
  - выделяется память по числу буферов * размер описателя (64 байта)
- в каждом описателе хранится:
  - адрес блока на диске в виде метки блока (BufferTag)
    - адрес блока содержит идентификаторы: табличное простанство, базы данных, файл, форк, номер блока от начала первого файла
  - адрес буфера в виде порядкового номера буфера в буферном кэше
- описатель связан 1:1 (один к одному) с буфером
- данных в 20 байтах которые занимает BufferTag достаточно (не нужно никуда обращаться), чтобы считать блок с диска

Структуры памяти, обслуживающие буферный кэш

Доступ к данным кластера идёт через буферный кэш. Для настройки производительности стоит познакомиться в общих чертах с моделью его работы. Это может пригодиться для предположения о том, где и в каких случаях могут возникать узкие места. Случаи: необычное или экстремальное использование функционала баз данных. Как пример: частое создание и удаление таблиц, прогрев кэша буферов.

Названия структур в разделяемой памяти экземпляра, относящиеся к буферному кэшу (как они называются в представлении `pg_shmem_allocations`):

Buffer Blocks - сам буферный кэш. Размер каждого буфера равен размеру блока. `shared_buffers` (по умолчанию 16384, максимум 1073741823=30 бит).

Buffer Descriptors - описатели (дескрипторы, заголовки) буферов. Структура описателя называется `BufferDesc`. Располагается в отдельной части памяти, **один дескриптор для каждого буфера кэша буферов**. Размер 64 байта, в которых находятся:

1) структура `BufferTag` (размер 17 байт, выравнивается до 20 байт), в которой указан прямой (самодостаточный, то есть хранящий всё, чтобы найти файл и в нём блок) адрес блока на диске. Структура состоит из:

oid (идентификатор) табличного пространства

oid базы данных

название файла, представляет собой число

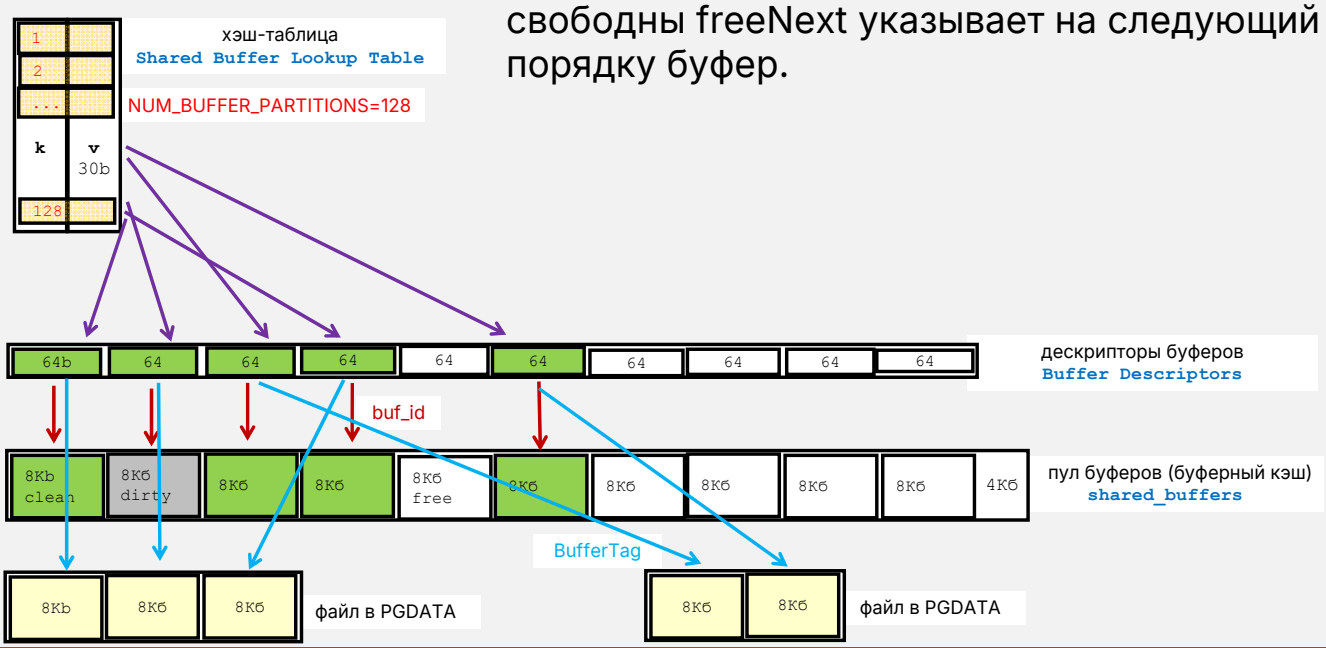
тип слоя (форка), к которому относится файл: основной (main), карта свободного места (fsm), карта видимости и заморозки (vm), инициализации для нежурналируемых таблиц (init).

номер блока относительно первого блока первого файла слоя, размер 4 байта.

2) порядковый номер буфера в кэше буферов.

Структуры памяти, обслуживающие буферный кэш

- после запуска экземпляра, пока все буфера свободны freeNext указывает на следующий по порядку буфер.



Структуры памяти, обслуживающие буферный кэш (продолжение)

3) 32 бита, которые содержат: 18 бит `refcount`, 4 бита `usage count` (всего 6 градаций), 10 бит флагов, в которых отражается:

1 - стоит блокировка на заголовок буфера
2 - блок грязный
3 - блок не поврежден
4 - блок существует в файле на диске
5 - буфер в процессе заполнения образом с диска или записи на диск
6 - предыдущая операция ввода-вывода сбойнула
7 - в процессе записи на диск загрязнился
8 - ждет снятия закреплений другими процессами чтобы заблокировать буфер для изменения
9 - **помечен процессом контрольной точки для записи на диск**
10 - относится к журналируемому объекту.

Часть этих флагов используют bgwriter и checkpointer для отслеживания не поменялся ли блок в процессе его записи на диск, так как в процессе записи (операция ввода-вывода) устанавливаются разделяемые блокировки. Это ускоряет работу СУБД.

4) идентификатор процесса, который ждет снятия закреплений буфера другими процессами (waiting for pincount 1)

Если процесс хочет работать с блоком, то он его ищет в буферном кэше. Если находит, то закрепляет. Множество процессов может закрепить буфер. Если процессу буфер не нужен, то процесс снимает закрепление.

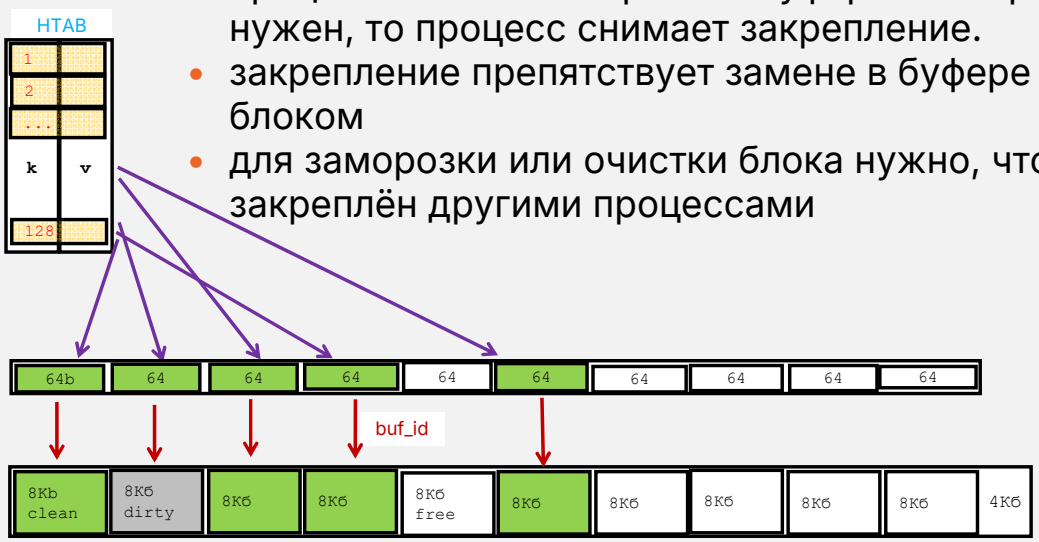
Закрепление препятствует замене в буфере блока другим блоком.

Процесс, который хочет очистить место в блоке от строк, которые вышли за горизонт базы данных, должен дождаться момента, когда блоком в буфере никакой другой процесс не интересуется кроме него самого, то есть единичка в pincount установлена им самим.

5) легковесная блокировка на содержимое буфера, устанавливаются процессами на короткое время. Два вида: Exclusive и Shared. Большинство действий используют Shared и конкурентность работы с буфером высока. Exclusive используется для заморозки строк в блоке и очистки блока от старых версий строк вакуумом и быстрой очисткой.

Закрепление блока в буфере

- если процесс хочет работать с блоком, то он его ищет в буферном кэше. Если находит, то закрепляет. Множество процессов может закрепить буфер. Если процессу буфер не нужен, то процесс снимает закрепление.
- закрепление препятствует замене в буфере блока другим блоком
- для заморозки или очистки блока нужно, чтобы блок не был закреплён другими процессами



Закрепление блока в буфере

Если процесс хочет работать с блоком, то он его ищет в буферном кэше. Если находит, то закрепляет. Несколько процессов могут одновременно закрепить ("pin") буфер. Если процессу буфер не нужен, то процесс снимает закрепление. Закрепление препятствует замене в буфере блока другим блоком.

Один блок не может находиться в двух и более буферах, он может находиться только в одном буфере.

Действия, которые требуют, чтобы буфер не был закреплён ни одним процессом:

1) вакуумирование или быстрая очистка блока от старых версий строк перераспределяет версии строк в блоке и требует, чтобы никакой процесс, кроме него не закреплял блок на время очистки

2) заморозка строк в блоке. В 18 версии PostgreSQL вероятность резкого увеличения числа конфликтов закреплений из-за необходимости автовакууму выполнить заморозку, была уменьшена. В режиме заморозки автовакуум вынужден ждать, когда pincount станет равным нулю и из-за ожиданий на большом числе блоков, может выполнять заморозку долго. Уменьшение достигается засчёт того, что до запуска в режиме заморозки, автовакуум пытается при обычной работе заморозить пятую часть блоков от планируемых к заморозке. Для настройки ранней заморозки в 18 версии введён параметр конфигурации **`vacuum_max_eager_freeze_failure_rate`**.

В 19 версии удалена структура `BufStrategyControl`, и поиск по списку свободных блоков (freelist) не производится, что устранило узкое место: https://commitfest.postgresql.org/patch/5928/

Закрепление буфера (pin) и блокировка content_lock

- закрепление (pin) используется для того, чтобы блок в буфере не был заменен на другой
- для чтения или изменения содержимого блока в буфере нужна легковесная блокировка content_lock, ссылка на которую сохраняется в описателе блока `Buffer Descriptors`
- блокировка должна удерживаться короткое время, в отличие от pin
- для удаления места занимаемого строкой (HOT cleanup или vacuum) после pin и Exclusive процесс дожидается, чтобы другие процессы незакрепляли блок в буфере
- для добавления в блок новой строки или изменения xmin, xmax существующих строк процесс должен получить блокировку content_lock типа Exclusive
- если процесс имеет pin и Shared content_lock, то он может менять некоторые биты в t_infomask, в частности статус фиксации/отката

Закрепление буфера (pin) и блокировка content_lock

Закрепление (pin) может удерживаться долго и используется для того, чтобы блок в буфере не был заменен на другой. Для чтения или изменения содержимого блока в буфере нужна легковесная блокировка `content_lock`, ссылка на которую сохраняется в описателе блока `Buffer Descriptors`. Размер описателя каждого блока 64 байта (с выравниванием). Эта блокировка должна удерживаться короткое время, в отличие от pin.

1. Для доступа к строкам и их заголовкам в блоке устанавливаются: pin и content_lock (Exclusive или Shared в зависимости от намерений процесса).
2. После нахождения нужных строк content_lock может быть снят, но pin не сниматься и в таком режиме процесс сможет читать строки блока, которые увидел пока у процесса был content_lock.
3. Для добавления в блок новой строки или изменения xmin, xmax существующих строк процесс должен получить content_lock типа Exclusive. При наличии Exclusive никто не может иметь Shared content_lock и соответственно видеть новые строки, которые в процессе изменения. Старые строки могут продолжать читать, так как они все равно не меняются: очиститься и заморозиться не могут из-за удержания горизонта событий.
4. Если процесс имеет pin и Shared content_lock, то он может менять некоторые биты в t_infomask, в частности статус фиксации/отката. Эти биты могут даже потеряться, в таком случае процесс просто перепроверит статус транзакции. Менять биты и xmin которые относятся к заморозке при этом нельзя, для этого нужнен Exclusive content_lock и такие изменения логируются. А что с контрольными суммами? При изменении любых битов контрольная сумма станет другой, но контрольная сумма меняется перед записью блока на диск.
5. Для удаления места занимаемого строкой (HOT cleanup или vacuum) после pin и Exclusive процесс **дожидается**, чтобы другие процессы не закрепляли блок. Интересно, что другие процессы при этом могут увеличивать pincount (закреплять блок, показывая намерение работать с его содержимым, ведь подгрузить блок в другой буфер не могут), так как из-за Exclusive не смогут установить Shared которая нужна чтобы заглянуть в блок.

Если pincount>1, то (авто)вакуум вписывает себя в поле описателя блока waiting for pincount 1, снимает Exclusive и ждёт (в тех режимах, когда не может пропускать блоки). HOT cleanup не ждёт. Ожидающий может быть только один, но это норма, так как таблицу может очищать только один процесс вакуума. Пока процесс вакуума в ожидании, другие процессы могут закреплять блок нескончаемым потоком и вакуум может долго ждать.

Стратегии замены буферов

- методы замены блоков в буферном кольце (buffer cache replacement strategy):
  - BULKREAD. Для последовательного чтения блоков таблиц (Seq Scan) размер которых **не меньше 1/4 кэша** буферов используется набор буферов в буферном кэше размером 256Кб
  - VACUUM. Грязные страницы не убираются из кольца, а посылаются на запись. Размер кольца задается параметром конфигурации `vacuum_buffer_usage_limit`. По умолчанию 256Кб.
  - BULKWRITE. Используется командами COPY и CREATE TABLE AS SELECT. Размер кольца 16Мб.
- в буферном кэше блок может находиться только в одном буфере
- если буфер стал грязным, он исключается из буферного кольца

Стратегии замены буферов (буферные кольца)

Чтобы команды, однократно обрабатывающие большое число блоков, не замусоривали буферный кэш, используется ограниченное число буферов (буферное кольцо). Буферные кольца не используются при работе с TOAST-таблицами.

Методы (**B**uffer**A**ccess**S**trategyType) замены блоков в буферном кольце:

1) **BAS**_BULKREAD. Для последовательного чтения блоков таблиц (Seq Scan) используется набор буферов в буферном кэше размером 256Кб. размер выбран исходя из того, чтобы эти буфера поместились в кэш второго уровня (L2) ядра процессора. Кольцо не должно быть слишком маленьким, чтобы в него поместились все закрепленные (pinned) процессом буфера. Также на случай, если другие процессы захотят сканировать те же данные, размер должен обеспечивать "зазор" чтобы процессы синхронизировались и одновременно закрепляли, сканировали, снимали закрепление тех же блоков. Этот метод могут использовать и команды, грязнящие буфера. Также другие процессы могут грязнить буфера пока они в буферном кольце читателя, так как **блок может находиться только в одном буфере**. **Если буфер стал грязным, он исключается из буферного кольца.**

Сканируемая **таблица должна быть больше четверти буферного кэша:**

```
scan->rs_strategy = GetAccessStrategy(BAS_BULKREAD);
```

Метод используется при создании новой базы данных способом WAL_LOG для чтения таблицы pg_class исходной базы. **Для TOAST-таблиц буферные кольца не используется, так как доступ к TOAST идет всегда по TOAST-индексу.**

2) BAS_VACUUM. Грязные страницы не убираются из кольца, а посылаются на запись. Размер кольца задается параметром конфигурации `vacuum_buffer_usage_limit`. По умолчанию 256Кб.

3) BAS_BULKWRITE. Используется командами COPY, CREATE TABLE AS SELECT. Размер кольца 16Мб. При копировании (RelationCopyStorageUsingBuffer(...)) таблицы **используется два кольца**: для чтения исходной таблицы и кольцо для заполнения целевой таблицы.

Поиск блока в буферном кэше

Процесс экземпляра:

- создает в своей локальной памяти экземпляр структуры BufferTag
- вычисляет 4-байтный хэш от BufferTag
- по значению хэша определяет номер партиции в хэш-таблице **Shared Buffer Lookup Table**
- запрашивает легковесную (LWLock) блокировку типа BufMappingLock партиции таблицы в которую попало значение хэша
- в хэш-таблице находит порядковый номер блока в кэше буферов или -1 если блока нет в кэше

Поиск блока в буферном кэше

Процессу требуется работать с блоком и он
1) Вычисляет 4-байтный хэш адреса (тэга, `BufferTag`) блока
2) По значению хэша определяет номер партиции в **Shared Buffer Lookup Table**. Размер записи (hash bucket slot) в **Shared Buffer Lookup Table** 8 байт, она состоит из 4-байтного хэша и порядкового номера буфера (заголовка буфера). Число блоков в файлах кластера баз данных может быть больше числа буферов и тогда хэш от разных блоков может совпасть. В этом случае, в таблицу вставляются записи с тем же значением ключа, но со ссылками на разные буфера (cache chains).
3) Запрашивает легковесную (`LWLock`) блокировку типа `BufMappingLock` части хэш-таблицы, в которую попал хэш. Таблица разделена на `128` частей. Один процесс может получить блокировки на несколько частей, даже на все части. Блокировка удерживается недолго.
4) Получает из хэш-таблицы порядковый номер блока в кэше буферов или -1, если блока нет в кэше буферов.
5) По номеру буфера в записи без блокировок считывается заголовок буфера (**Buffer Descriptors**), атомарной операцией увеличивается refcount (он же ref_count, 18 бит) и usage_count (4 бита) которые вместе с флагами (10 бит) хранятся в 4 байтах. LWLock:BufMappingLock тут же снимается и только потом в заголовке буфера устанавливается LWLock:content_lock которая обеспечивает доступ к буферу и остальному содержимому заголовка.

Освобождение буферов при удалении файлов

- при удалении базы данных выполняется полное сканирование всех дескрипторов буферов
- полное сканирование дескрипторов выполняется, если размер удаляемого relation больше 1/32 пула буферов
- в остальных случаях поиск дескрипторов идет через хэш-таблицу
- при увеличении кэша буферов с 1Гб до 16Гб время на поиск буферов при удалении таблицы увеличивается на порядок
- поиск выполняется при удалении файлов в результате DROP, TRUNCATE и вакуума (если не отключено усечение файлов)

```
pgbench --file=CreateAndDrop.sql -j 1 -c 1 -T 10
TPS для HP 128MB 417
TPS для HP  1GB  375
TPS для HP  4GB  227
TPS для HP  8GB  127
TPS для HP 16GB   55
TPS для HP 18GB   33
```

CreateAdndDrop.sql:

```
create table x(id int);
insert into x values (1);
drop table x;
```


Освобождение буферов при удалении файлов

При удалении базы данных выполняется полное сканирование всех дескрипторов буферов (BufferDesc) для поиска буферов, относящихся к файлам базы данных. Если в заголовке указано, что буфер не относится к базе данных, он пропускается. Если относится, то на дескриптор буфера устанавливается SpinLock, дескриптор освобождается, SpinLock снимается.

Полное сканирование выполняется также, если размер удаляемого relation больше 1/32 пула буферов:

В остальных случаях (удаление, усечение файлов) поиск буферов идет по диапазону и с использованием хэш-таблицы, что **тоже не быстро**. Удаляться и усекаться файлы могут вакуумом, командой DROP, TRUNCATE над постоянными объектами. Временные объекты не хранят блоки в кэше буферов.

При большом размере пула буферов длительность выполнения этих операций может быть существенной.

Скорость создания и удаления небольшой таблицы командами:

```
begin transaction;
create table x(id int);
insert into x values (1);
drop table x;
commit;
pgbench --file=CreateAndDrop.sql -j 1 -c 1 -T 10
```

TPS для `shared_pool` без Huge Pages (HP) размером `128MB - 433`

```
1GB  367
4GB  220
8GB  123
16GB  43
18GB  32
```

Время на поиск дескрипторов буферов по хэш-таблице при удалении небольшой таблицы увеличивается в 10 раз при увеличении пула буферов с 1Гб до 16Гб. Использование Huge Pages скорость существенно не меняет, так как пул буферов не сканируется.

Процесс фоновой записи bgwriter

- процесс bgwriter записывает (writeback) грязные буфера и помечает их как чистые
- работа bgwriter снижает вероятность того, что процессы натолкнутся на грязные блоки при поиске буфера-кандидата (victim) на вытеснение (eviction) для замены другим блоком
- "поиск свободного блока" это обычно буфера-кандидата на вытеснение из буфера блока, так как все буфера обычно заняты и список свободных блоков пуст
- при вытеснении грязного блока из буфера обращения к шине ввода-вывода нет, это "writeback": копирование из памяти (буфер) в память (страничный кэш linux)

```
select name, setting, context, max_val, min_val from pg_settings where name ~ 'bgwr';
          name           | setting | context |  max_val   | min_val
-------------------------+---------+---------+------------+---------
 bgwriter_delay          | 200     | sighup  | 10000      | 10
 bgwriter_flush_after    | 64      | sighup  | 256        | 0
 bgwriter_lru_maxpages   | 100     | sighup  | 1073741823 | 0
 bgwriter_lru_multiplier | 2       | sighup  | 10         | 0
```


Процесс фоновой записи bgwriter

Грязные буфера могут записывать на диск ("очищать") процессы, работающие с буферным кэшем, в том числе checkpointer, bgwriter, серверные процессы, рабочие процессы автовакуума. Процесс `bgwriter` записывает грязные буфера и помечает их как чистые. Работа `bgwriter` снижает вероятность того, что серверные процессы натолкнутся на грязные блоки при поиске буфера-кандидата (victim) на вытеснение (eviction) для замены другим блоком. При вытеснении грязного блока из буфера обращения к шине ввода-вывода нет, это копирование из памяти (буфер) в память (страничный кэш linux). Задержки не так критичны, как может показаться. У процессов `bgwriter, walwriter, bgworker` названия созвучны, но это разные процессы. Работа процесса `bgwriter` настраивается параметрами:

```
select name, setting, context, max_val, min_val from pg_settings where name ~ 'bgwr';
          name           | setting | context |  max_val   | min_val
-------------------------+---------+---------+------------+---------
 bgwriter_delay          | 200     | sighup  | 10000      | 10
 bgwriter_flush_after    | 64      | sighup  | 256        | 0
 bgwriter_lru_maxpages   | 100     | sighup  | 1073741823 | 0
 bgwriter_lru_multiplier | 2       | sighup  | 10         | 0
```

`bgwriter_delay` на сколько миллисекунд `bgwriter` засыпает между итерациями. `bgwriter_flush_after` - число блоков после посылки на запись которых инициируется flush страничного кэша linux. Ноль отключает flush.

Число грязных буферов, записываемых в итерации, зависит от того, сколько блоков подгрузили в буферный кэш серверные процессы в предыдущих циклах. Усредненное значение умножается на `bgwriter_lru_multiplier` и указывает сколько буферов нужно очистить в текущем цикле. Процесс с максимальной скоростью пытается достичь это значение, но не больше, чем `bgwriter_lru_maxpages`. `bgwriter_lru_maxpages` - максимальное количество блоков, которые записываются в одной итерации, при нулевом значении `bgwriter` перестает работать. Исходя из этого `bgwriter_lru_maxpages` имеет смысл установить в максимальное значение.

Что если в предыдущих итерациях серверные процессы не использовали новые буфера? Чтобы не было "медленного старта", в итерации будет сканироваться не меньше, чем: `NBuffers/120000*bgwriter_delay+reusable_buffers_est` блоков. Для размера кэша буферов 128Мб и 200 миллисекундами задержки получится `27+reusable_buffers_est` блоков.

Очистка кэша буферов процессом bgwriter

- буфер не попадает в список свободных буферов, грязный буфер становится чистым
- на диск записываются только грязные, незакрепленные блоки с `usage_count=0`
- **bgwriter не меняет `usage_count`**
- при включенном подсчете контрольных сумм блок копируется из кэша буферов в локальную память процесса bgwriter и в локальной памяти подсчитывается и сохраняется в заголовке блока контрольная сумма

Очистка кэша буферов процессом bgwriter

Сначала берется спин-блокировка на дескриптор блока. Проверяются значения: **pin count (refcount)=0 (блок не нужен процессам), usage_count=0 (попадает в градацию давно не использовавшихся)**, если значения не такие как приведены, то спин-блокировка снимается и блок не сбрасывается на диск. Иначе буфер закрепляется, берется легковесная разделяемая блокировка, вызывается функция передачи буфера в страничный кэш linux, снимаются блокировка и закрепление.

В процессе сброса буфера, другие процессы могут успеть заблокировать и закрепить буфер, поменять биты-подсказки, которые разрешено менять, **имея Shared блокировку и pin**.

Считывается LSN из блока в буфере и сбрасывается содержимое WAL-буфера вплоть до этого LSN. **Этим гарантируется правило Write Ahead - журнал с изменениями в блоке должен быть записан раньше самого блока**.

Если подсчет контрольных сумм включен, то содержимое буфера копируется в локальную память процесса bgwriter. На локальной копии вычисляется контрольная сумма и эта копия размером 8Кб передается коду ядра linux, который помещает блок в виде уже двух страниц по 4Кб в страничный кэш linux.

Почему копируется в локальную память? Потому, что в блоке другие процессы могут менять биты-подсказки (infomask), пока bgwriter подсчитывает контрольную сумму и контрольная сумма окажется неверна даже при изменении одного бита. Поэтому, для подсчета контрольной суммы блок копируется в локальную память. С копированием из памяти в память и связано некоторое уменьшение производительности, при включении подсчета контрольных сумм, а не с нагрузкой на вычислительные мощности процессора.

Так как bgwriter вытесняет давно неиспользовавшиеся (usage_count=0 и pin count он же refcount=0) буфера, то невелика вероятность того, что:

1) блок понадобится другому процессу;
2) что будут ожидания получения блокировок;
3) что потребуется запись в WAL

Контрольная точка

- выполняет процесс checkpointer:
  - периодически по истечении `checkpoint_timeout`
  - по разрастанию журнала `max_wal_size`
  - в конце процедуры остановки или запуска экземпляра
  - продвижении реплики
  - командам checkpoint, create database
  - при резервировании

```
postgres=# checkpoint;
CHECKPOINT
LOG:  checkpoint starting: immediate force wait
LOG:  checkpoint complete: wrote 5 buffers (0.0%);
```


Контрольная точка

Выполняется фоновым процессом checkpointer. Контрольные точки выполняются: периодически, в конце процедуры остановки и запуска экземпляра экземпляра, продвижении реплики, резервировании, команде checkpoint, создании базы.

На реплике контрольные точки не инициируются, но выполняются restart points.

В случае падения экземпляра и последующего перезапуска, алгоритм контрольной точки должен гарантировать, что журнальные данные, начиная с LSN начала успешно завершившейся, то есть записанной в управляющий файл `pg_control` (на последней фазе выполнения контрольной точки), будут достаточны для восстановления кластера.

Контрольные точки позволяют не хранить WAL-сегменты, которые не нужны для восстановления.

Свойства контрольных точек, которые отражаются в логе кластера:

immediate - завершить уже начатую (если есть) контрольную точку с максимальной скоростью, игнорируя checkpoint_completion_target и тут же выполнить контрольную точку тоже с максимальной скоростью

force - даже если не было записи в WAL. Выполняется по команде checkpoint, продвижении реплики pg_promote(), остановке экземпляра

wait - вернуть управление только после завершения контрольной точки

Свойства могут комбинироваться друг с другом. Например, по команде checkpoint устанавливаются свойства immediate force wait.

Шаги выполнения контрольной точки

- если экземпляр останавливается, в файл `pg_control` записывается статус о начале гашения экземпляра
- вычисляется LSN следующей журнальной записи. Это будет LSN начала контрольной точки
- ждёт снятия процессами признака DELAY_CHKPT_START
- на диск сбрасываются slru буфера и другие структуры разделяемой памяти
- checkpointer в цикле пробегает все описатели буферов и для грязных блоков устанавливает флаг BM_CHECKPOINT_NEEDED, сохраняет адрес блока (5 чисел) в структуре памяти Checkpoint BufferIds для последующей сортировки
- после установки флага checkpointer не блокирует буфер и блок в буфере может быть сброшен на диск и заменен другим

Шаги выполнения контрольной точки

При выполнении контрольной точки выполняются следующие действия.

Если экземпляр останавливается, то в файл `pg_control` записывается статус о начале гашения экземпляра. Вычисляется LSN следующей журнальной записи. Это будет LSN начала контрольной точки, но отдельной журнальной записи о начале checkpointer не создает.

Другие процессы могут выставлять признак DELAY_CHKPT_START. Собирается список виртуальных идентификаторов транзакций, процессы которых установили признак. Если список не пуст, то checkpointer ждет снятия признаков в цикле, засыпая на 10 миллисекунд между проверками снятия признаков. Другие процессы могут устанавливать признаки, но они роли не играют, так как установлены после вычисленного ранее LSN. Признак устанавливается на короткое время: когда процесс выполняет логически связанное действие неатомарно: создавая разные журнальные записи. Например, обновляет статус транзакции в CLOG и создает журнальную запись о фиксации.

Дальше checkpointer начинает сбрасывать на диск SLRU-буфера (и другие структуры разделяемой памяти) в файлы которые они кэшируют и/или в WAL и выполняется синхронизация по этим файлам (fsync). Эти журнальные записи должны относиться к контрольной точке и идти после LSN ее начала.

Алгоритм выполнения действий, связанных с записью грязных блоков буферного кэша:

Контрольные точки типа IS_SHUTDOWN, END_OF_RECOVERY, FLUSH_ALL записывают все грязные буфера, в том числе, относящиеся к нежурналируемым объектам. Процесс checkpointer в цикле пробегает все описатели буферов, получает SpinLock на один блок одновременно. Дальше проверяет что блок грязный и для грязных блоков устанавливает флаг BM_CHECKPOINT_NEEDED, сохраняет адрес блока в разделяемой структуре памяти `Checkpoint BufferIds`. После чего снимает SpinLock. Адрес блока - 5 чисел структуры `BufferTag`.

Если какой-то процесс очистит буфер, то этот флаг будет снят очищающим процессом - без разницы каким процессом будет записан блок, главное чтобы все грязные буфера которые были грязными на момент начала контрольной точки были записаны на диск. Теперь checkpointer имеет список блоков, которые будет записывать на диск.

Шаги выполнения контрольной точки

- сохраненные идентификаторы блоков сортируются в порядке: tblspc, relation, fork, block
- блоки посылаются по одному в страничный кэш linux
- Если checkpoint_flush_after не равен нулю, то выполняется синхронизация по уже отсортированным диапазонам блоков по каждому файлу
- в WAL сохраняется моментальный снимок со списком активных транзакций
- формируется журнальная запись окончания контрольной точки, содержащая LSN журнальной записи, которая была сформирована на момент начала контрольной точки
- в файле `pg_control` сохраняется LSN сформированной журнальной записи
- удаляются WAL-сегменты, которые не должны удерживаться

Шаги выполнения контрольной точки (продолжение)

Дальше checkpointer сортирует идентификаторы блоков стандартным алгоритмом quick sort. Сравнение выполняется в порядке: tblspc, relation, fork, block. Первым идет tblspc и это существенно. Сортировка, в частности, нужна чтобы не было такого что блоки посылаются в табличные пространства по порядку, нагружая одномоментно одно табличное пространство. Предполагается, что табличные пространства это смонтированные отдельно файловые системы на разных устройствах.

Подсчитывается число блоков по каждому табличному пространству, определяется размер набора блоков (slice) чтобы запись во все табличные пространства финишировала примерно одинаково.

checkpointer посылает по одному блоку из своего списка с периодическими задержками (в соответствии с параметром конфигурации `checkpoint_completion_target` и вычисленной скоростью записи) в страничный кэш linux.

Если `checkpoint_flush_after` не равен нулю, то выполняется синхронизация по уже отсортированным диапазонам блоков по каждому файлу. Объединяя отсортированные диапазоны блоков (если такие были) по каждому файлу checkpointer посылает в linux системные вызовы на запись диапазонов блоков в страничный кэш linux, которые до этого были "посланы на диск" процессами.

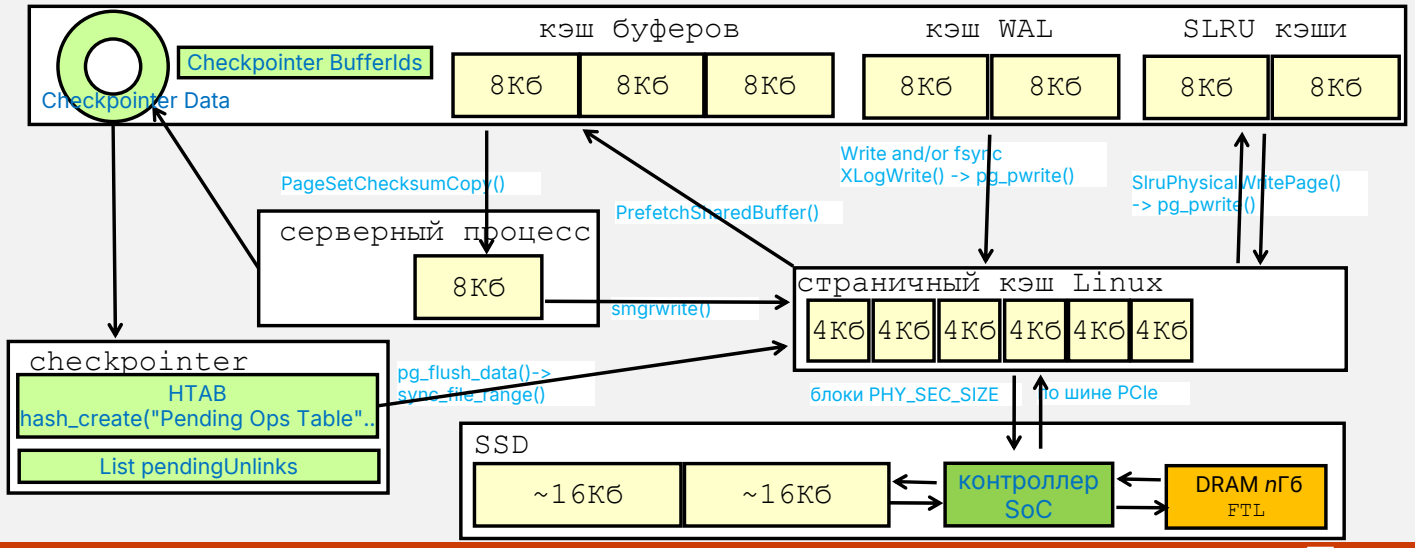
Для контрольных точек (кроме той, что выполняется по остановке экземпляра) в WAL сохраняется моментальный снимок со списком активных транзакций. Это может пригодиться репликам при восстановлении по архивным журналам.

Формируется журнальная запись, содержащая LSN журнальной записи, которая была сформирована на момент начала контрольной точки. Сформированная журнальная запись с посылается в WAL системным вызовом fdatasync (или другим методом). В pg_control сохраняется LSN сформированной записи об окончании контрольной точки. После всех этих действий контрольная точка завершена и, если экземпляр упадёт, то восстановление начнётся с начала этой контрольной точки.

Дальше checkpointer проверяет не нужно ли инвалидировать слоты репликации по причине того, что слот давно не использовался. Удаляются WAL-сегменты, которые не должны удерживаться. Для восстановления экземпляра нужны сегменты начиная с сегмента, содержащего журнальную запись с LSN начала контрольной точки. Выделяются новые или очищаются и переименовываются старые WAL сегменты в соответствии с параметрами конфигурации.

Взаимодействие процессов экземпляра с диском

- блоки по 8Кб считываются в разделяемую память через страничный кэш (блоки по 4Кб)
- блоки записываются на диск через страничный кэш
- для записи используется оптимизированный алгоритм синхронизации



Взаимодействие процессов экземпляра с диском

Ссылки на блоки для синхронизации (в будущем) записываются хэш-таблицу на 100 блоков, создаваемую **в локальной памяти** checkpointer, сортируются, чтобы упорядочить блоки для передачи диапазона блоков. **fsync() выполняется один раз по каждому файлу (где были изменения хоть в одном блоке) в конце контрольной точки.**

Для синхронизации нужно запомнить все файлы, которые были изменены с момента последней контрольной точки, чтобы выполнить синхронизацию до завершения следующей контрольной точки. В хэш-таблицах запоминаются блоки, которые нужно синхронизировать. Для команд на удаление файлов используется список pendingUnlinks, так как повторных команд (операций) удаления файлов не должно быть.

Процессы передают операции процессу checkpointer через разделяемую структуру памяти `CheckpointerShmemStruct` с названием "Checkpointer Data". Список разделяемых структур и размеры есть в представлении `pg_shmem_allocations`.

Временные таблицы не синхронизируются, так как для них отказоустойчивость не нужна.

Практика

1. Транзакция в psql
2. Список фоновых процессов
3. Буферный кэш, команда EXPLAIN
4. Журнал предварительной записи
5. Контрольная точка
6. Восстановление после сбоя

Практика

Транзакция в psql
Список фоновых процессов
Буферный кэш, команда EXPLAIN
Журнал предварительной записи
Контрольная точка
Восстановление после сбоя

2

2b Архитектура PostgreSQL

Многоверсионность

Версии строк

- В блоках таблицы хранятся версии строк (rows), которые называют кортежами (tuple)
- Запросы (SELECT) должны выдавать данные на один момент времени ("консистентные"), что называют "целостность по чтению" (read consistency)
- Все версии строк физически хранятся в файлах таблицы по возможности рядом друг с другом (в тех же блоках файлов данных)

Версии строк

В блоках таблицы хранятся версии строк (rows), которые называют кортежами (tuple). Последнее название пошло из реляционной теории, где таблицы называют отношениями, столбцы атрибутами, а типы данных столбцов доменами.

При создании таблицы в PostgreSQL автоматически создается тип данных с названием таблицы, в котором названия и типы данных полей соответствуют названиям столбцов таблицы и их типам данных. Тип данных называют составным, так как состоит из полей других типов данных.

Запросы (SELECT) должны выдавать данные на один момент времени ("консистентные"), что называют "целостность по чтению" (read consistency). Пока выполняются запросы, строки могут меняться и удаляться. Для обеспечения целостности по чтению нужно хранить старые версии строк. Если запрос не найдет версию строки на нужный ему момент времени, то он прервется с ошибкой "слишком старая моментальная копия" (snapshot too old). Все версии строк физически хранятся в файлах таблицы по возможности рядом друг с другом (в тех же блоках файлов данных).

Вторая причина, по которой сохраняются старые версии строк - для транзакций. Транзакция может обновить строку, породив новую версию строки. Транзакцию можно откатить или зафиксировать. Если транзакция фиксируется, то старая версия строки не нужна транзакции. Если транзакция откатывается, то нужна старая версия, а новая не нужна. Поэтому, нужно хранить все версии строк, порождаемые в транзакциях как минимум до их завершения. Особенностью PostgreSQL является то, что если транзакция откатилась, то версии строк которые она бы породила если бы зафиксировалась физически остаются в блоках и занимают место, а не вычищаются при откате. Поэтому откат транзакций в PostgreSQL выполняется быстро. Откаченная (ROLLBACK) транзакция называется прерванной (aborted).

Хранение версий строк называется многоверсионностью (Multi-Version Concurrency Control, MVCC).

Таблицы

- объект в котором хранятся данные
- несколько видов: обычные таблицы (heap tables, строки хранятся неупорядоченно), нежурналируемые, временные, секционированные
- расширения могут создавать новые способы хранения данных и методы доступа к ним
- число и порядок следования столбцов задаются при создании таблицы
- после создания таблицы можно добавлять и удалять столбцы. При добавлении столбца он добавляется последним - после всех существующих столбцов
- можно поменять тип столбца

Таблицы

Данные приложений хранятся в таблицах. В СУБД есть обычные таблицы (heap tables, строки хранятся неупорядоченно), нежурналируемые, временные, секционированные. Расширения могут создавать новые способы хранения данных и методы доступа к ним. В СУБД Tantor Postgres SE есть расширение `pg_columnar`.

Число и порядок следования столбцов задаются при создании таблицы. Каждый столбец имеет имя. После создания таблицы можно командой ALTER TABLE добавлять и удалять столбцы. При добавлении столбца он добавляется после всех существующих столбцов.

Поля для добавляемого столбца по умолчанию имеют значения NULL или получают значения заданные опцией DEFAULT. При добавлении столбца не будут генерироваться новые версии строк, если в DEFAULT установлено статичное значение. Если в значении используется изменчивая функция, например, `now()`, то при добавлении столбца будут обновлены все строки таблицы, что долго. В таком случае, возможно будет более оптимально сначала добавить столбец без указания DEFAULT, потом обновить строки командами UPDATE, установив значение для добавленного столбца, потом установить значение DEFAULT командой ALTER TABLE таблица ALTER COLUMN столбец SET DEFAULT значение;

Удаление столбца удаляет значения в полях каждой строки и ограничения целостности, в которые входит удаляемый столбец. Если на удаляемое ограничение целостности ссылается FOREIGN KEY, то можно его удалить заранее или использовать опцию CASCADE.

Также можно поменять тип столбца командой **`ALTER TABLE таблица ALTER COLUMN столбец TYPE тип(размерность);`**

Поменять тип можно, если все существующие (не NULL) значения в строках смогут быть неявно приведены к новому типу или размерности. Если неявного приведения нет и не хочется его создавать или устанавливать как приведение типов данных по умолчанию, можно указать опцию USING и установить как получить новые значения из существующих.

Будут преобразованы значения DEFAULT (если оно определено) и ограничения целостности, в которые входит столбец. Лучше удалить ограничения целостности перед модификацией типа столбца и потом добавить ограничения.

Для просмотра содержимого блока используются функции стандартного расширения pageinspect.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/ddl-alter.html

Служебные столбцы

- `xmin` - номер транзакции (xid), создавшей версию строки
- `xmax` - номер транзакции (xid), удаляющей или пытавшейся (транзакция не была зафиксирована по любой причине: вызван rollback, серверный процесс прерван) удалить строку или ноль
- `ctid` адрес физического расположения строки
- `tableoid` - oid таблицы, в которой физически содержится строка. Значения имеют смысл для секционированных и унаследованных таблиц
- `cmin` порядковый номер команды внутри транзакции начиная с нуля, создавшей версию строки
- `cmax` порядковый номер команды внутри транзакции начиная с нуля, удаляющей или пытавшейся удалить строку

Служебные столбцы

В командам SQL доступны псевдостолбцы (служебные, системные). Их набор зависит от вида таблицы. **Псевдостолбцы не стоит использовать в коде приложений**, только для диагностики. Для обычных (heap) таблиц доступны псевдостолбцы:

`ctid` адрес физического расположения строки. Используя `ctid` планировщик может получить доступ к странице (блоку файла основного слоя) таблицы без полного сканирования всех страниц. `ctid` изменится, если новая версия строки разместится в другом блоке.

`tableoid` - oid таблицы, в которой физически содержится строка. Значения имеют смысл для секционированных и унаследованных таблиц. Быстрый способ узнать oid таблицы, так как соответствует `pg_class.oid`.

`xmin` - номер транзакции (xid), создавшей версию строки.

`xmax` - номер транзакции (xid), удаляющей или пытавшейся (транзакция не была зафиксирована по любой причине: вызван rollback, серверный процесс прерван) удалить строку.

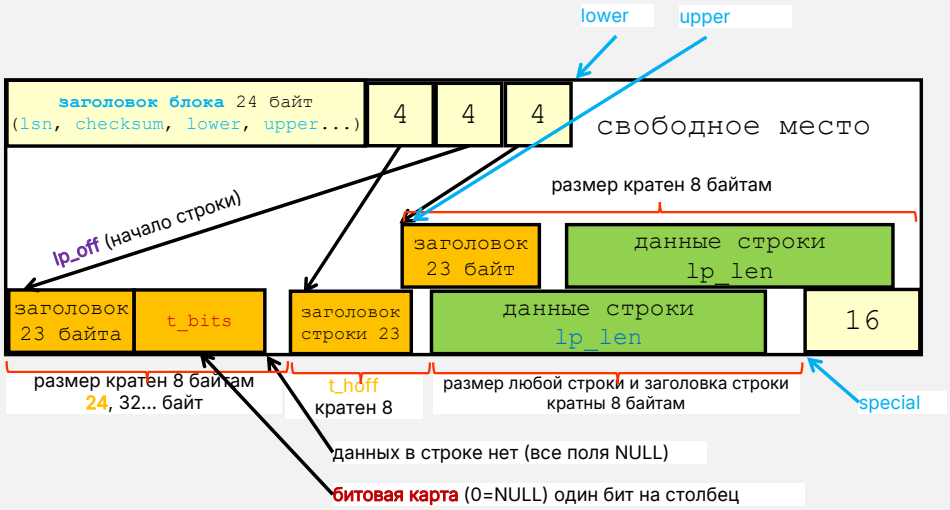
`cmin` порядковый номер команды внутри транзакции начиная с нуля, создавшей версию строки. Не имеет применения.

`cmax` порядковый номер команды внутри транзакции начиная с нуля, удаляющей или пытавшейся удалить строку. Для поддержки не очень хорошего кода, в котором, в одной транзакции несколько раз обновляется одна и та же строка.

`xmin, cmin, xmax, cmax` хранятся в трёх физических полях заголовка строки. `xmin` и `xmax` хранятся в отдельных полях. `cmin` и `cmax` интересны только в течение жизненного цикла транзакции для вставки (`cmin`) и удаления (`cmax`). `ctid` вычисляется на основе адреса строки. Физически у версии строки хранится `t_ctid` хранит адрес следующей (созданной в результате UPDATE) версии строки. Причем, это не "цепочка", связь может теряться, так как вакуум может удалить более новую версию строки раньше, чем старую (блок обработал раньше) и старая версия строки будет ссылаться на отсутствующую версию. Если версия последняя, то `t_ctid` хранит адрес этой версии. Также в процессе INSERT временно может устанавливаться "speculative insertion token" вместо адреса версии строки. **Использование псевдостолбцов в коде приложений приводит к ошибкам, которые проявляются при эксплуатации**. Попытки "улучшить" логику стандартного блокирования реляционных баз, вызваны незнанием стандартной логики и приводят к к непредсказуемым эффектам.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/ddl-system-columns.html

Структура блока данных




```
select * from page_header
(get_raw_page('t','main',0));
-[ RECORD 1 ]---------
lsn       | 0/110DCF10
checksum  | 0
flags     | 0
lower     | 928
upper     | 944
special   | 8176
pagesize  | 8192
version   | 5
prune_xid | 0
```


Структура блока данных

Приведена структура блока heap table. Размер блока 8Кб. В начале блока служебная структура фиксированного размера 24 байта. В них находится: LSN указывающий на начало журнальной записи, следующей за журнальной записью которой менялся блок. Этот LSN нужен, чтобы блок не был послан на запись, если журнальная запись не была записана на диск (реализации правила write ahead log). Также используется при восстановлении по журналу.

В Tantor Postgres SE и SE 1C используется 64-битный (8 байт) счетчик транзакций и в конце блока обычных таблиц имеется "специальное пространство" размером 16 байт, у TOAST 8 байт. В PostgreSQL специальная область для таблиц отсутствует, она есть у блоков индексов.

После фиксированной области располагаются указатели (line pointers) на начала записей (строк) в этом блоке. Для каждой строки под указатель используется 4 байта. Почему так много? Указатель содержит смещение ("**off**set") в байтах до начала строки (lp_**off** 15 бит, line pointer **off**set), 2 бита (lp_flags), 15 бит **длина** строки (lp_**len**). Два бита указывают четыре возможных статуса указателя: 1- указывает на строку, свободен и еще два статуса которыми реализуются оптимизации HOT (heap only tuple): dead и redirect.

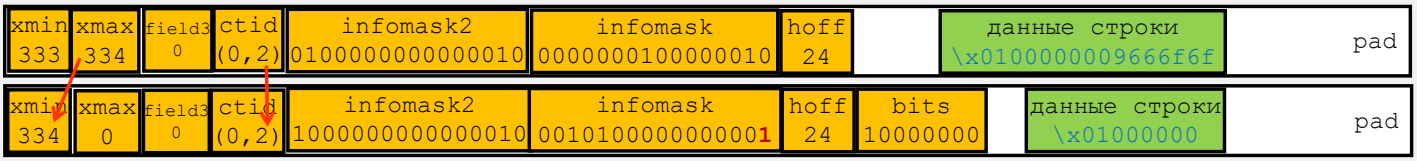
Если в таблице до 8 столбцов включительно, то размер заголовка строки 24 байта. Если в таблице 9 столбцов и больше, то размер заголовка строки **в случае, если хоть в одном поле присутствует пустое значение** (NULL) становится 32 байта, а начиная с 73 столбцов заголовок строки станет 40 байт.

Число строк в блоке, в зависимости от размера области данных в строке:

```
 rows | size
------+-----
  226 |    8
  185 |   16
  156 |   24
  135 |   32
  119 |   40
  107 |   48
   97 |   56
```

Заголовок версии строки

- xmin - xid транзакции, создавшей версию строки
- xmax - xid транзакции, удалившей версию строки
- ctid - ссылка на следующую версию этой строки
- bits - битовая карта пустых значений
- hoff - смещение до начала данных строки
- lp_off смещение до начала строки




```
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len|t_xmin|t_xmax|t_field3|t_ctid|t_infomask2|t_infomask|t_hoff| t_bits |t_oid
--+------+--------+------+------+------+--------+------+-----------+----------+------+--------+-----
 1|  8144|       1|    32|   333|   334|       0 | (0,2)|      16386|       258 |   24 |        |
 2|  8112|       1|    28|   334|     0|       0 | (0,2)|      32770|     10241 |   24 |10000000|
```


Заголовок версии строки

Заголовок строки имеет размер 24, 32, ... байт и кратен 8 байт. В нем хранится t_hoff - смещение до начала данных строки. В конце заголовка будет присутствовать битовая карта t_bits (размер кратен байту), если хоть в одном поле строки NULL. Один бит - один столбец, 1 - NULL, 0 - поле непусто. На наличие карты (присутствие NULL в любом поле) указывает один из битов t_infomask.
Пример создания второй версии строки:

```
create extension pageinspect;
create table t (n int, c text);
insert into t values (1, 'foo');
update t set c = null;
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len| xmin| xmax|  ctid| infomask2| infomask|  hoff| t_bits |
--+------+--------+------+-----+-----+------+----------+---------+------+--------+
 1|  8144|       1|    32|  333|  334| (0,2)|     16386|      258 |   24 |        |
 2|  8112|       1|    28|  334|    0| (0,2)|     32770|    10241 |   24 |10000000|
```

lp_off смещение до начала строки с точностью до байта.
lp_len длина строки
Размер заголовка строки всегда кратен 8 байт (выравнивается до 8 байт) и может занимать 24,32 ,40 байт. Размер всей строки (заголовок+данные) тоже всегда кратен 8 байтам. Для выравнивания в конец добавляются пустые (0x0000) байты.

Вставка строки

- xmin - xid транзакции, вставившей строку
- xmax -ноль
- ctid - ссылка на себя
- транзакция, вставившая строку не помечает в infomask то, что она была зафиксирована. Это будет сделано при последующем обращении к строке в другой транзакции или другим запросом


```
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len|t_xmin|t_xmax|t_field3|t_ctid|t_infomask2|t_infomask|t_hoff| t_bits |t_oid
--+------+--------+------+------+------+--------+------+-----------+----------+------+--------+-----
 1|  8144|       1|    32|   333|     0|       0 | (0,1)|         2 |     2050 |   24 |        |
```


Вставка строки

Пример вставки строки:

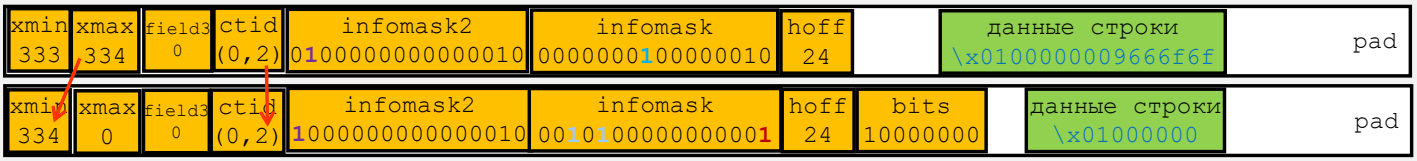
```
create extension pageinspect;
create table t (n int, c text);
insert into t values (1, 'foo');
select * from heap_page_items(get_raw_page('t','main',0));
 lp|lp_off|lp_flags|lp_len| xmin| xmax| ctid| infomask2| infomask|t_hoff| t_bits |
 --+------+--------+------+-----+-----+-----+----------+---------+------+--------+
  1|  8144|       1|    32|  333|    0|(0,1)|         2 |    2050 |   24 |        |
select * from t;
n |  c
---+-----
1 | foo
(1 row)
select * from heap_page_items(get_raw_page('t','main',0));
 lp|lp_off|lp_flags|lp_len| xmin| xmax| ctid| infomask2| infomask|t_hoff| t_bits |
 --+------+--------+------+-----+-----+-----+----------+---------+------+--------+
  1|  8144|       1|    32|  333|    0|(0,1)|         2 |    2306 |   24 |        |
```

ctid - системный столбец, который указывает на физическое местоположение строки в блоке таблицы. Этот системный столбец состоит из двух чисел: (block_number, line_pointer), где block_number - номер блока начиная с нуля, a line_pointer - номер указателя в заголовке этого блока. Индексы btree в листовых блоках хранят указатели на версии строк в виде ctid. Аналог столбца ctid в Oracle Database - служебный столбец (pseudo column) ROWID, но он уникален в пределах всей базы данных.

Транзакция, вставившая строку не помечает в infomask то, что она была зафиксирована. Это будет сделано при последующем обращении к строке в другой транзакции или другим запросом. Если девятый бит установлен, это означает, что транзакция, вставившая строку (xmin) была зафиксирована (committed).

Обновление строки

- ctid прежней версии указывает на адрес новой версии строки
- ctid актуальной версии строки указывает сам на себя
- xmax прежней версии меняется с нуля на номер транзакции, которая создала новую версию строки




```
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len|t_xmin|t_xmax|t_field3|t_ctid|t_infomask2|t_infomask|t_hoff| t_bits |t_oid
--+------+--------+------+------+------+--------+------+-----------+----------+------+--------+-----
 1|  8144|       1|    32|   333|   334|       0 | (0,2)|      16386|       258 |   24 |        |
 2|  8112|       1|    28|   334|     0|       0 | (0,2)|      32770|     10241 |   24 |10000000|
```


Вставка строки

Пример создания второй версии строки в результате обновления:

```
update t set c = null;
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len| xmin| xmax| ctid| infomask2| infomask|t_hoff| t_bits |
--+------+--------+------+-----+-----+-----+----------+---------+------+--------+
 1|  8144|       1|    32|  333|  334|(0,2)|     16386|      258 |   24 |        |
 2|  8112|       1|    28|  334|    0|(0,2)|     32770|    10241 |   24 |10000000|
```

При вставке появляется вторая версия строки. Область данных второй версии включает в себя все поля после обновления, то есть значения полей могут дублироваться.

ctid прежней версии указывает на адрес новой версии строки. ctid актуальной версии строки указывает сам на себя.

xmax прежней версии меняется с нуля на номер транзакции, которая создала новую версию строки.

Если транзакция, выполнившая UPDATE не зафиксирована, то серверные процессы других сессий видят все версии строки, но проверят что во второй версии в infomask нет битов, указывающих на то, что транзакция зафиксирована или откачена и обратятся к структуре CLOG в разделяемой памяти, чтобы проверить статус транзакции. Там они увидят, что транзакция не зафиксирована и не откачена, но процесс существует. Исходя из этого поймут, что вторую версию строки выдавать нельзя (иначе будет "грязное чтение") и вернут первую версию строки, тоже проверив статус транзакции. Статус транзакции (xmin committed), создавшей первую версию строки уже установлен в 9 бите infomask.

Биты infomask:

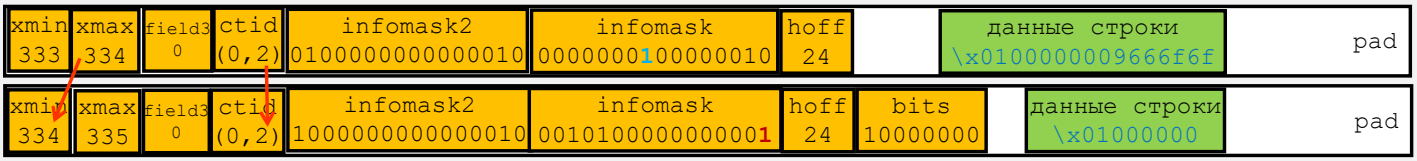
1 бит - есть пустые значения, 2 бит - есть поля переменной ширины, 3 - есть вынесенные в TOAST поля, 4 - есть поля типа OID, 5 - строка заблокирована в key-share режиме, 9 - xmin committed, 10 - xmin aborted, 11 - xmax committed, 12 - xmax aborted, 13 - в xmax мультитранзакция, 14 - актуальная версия строки.

Биты infomask2:

с 1 по 11 бит - число полей в строке, 14 - изменены ключевые поля или строка удалена, 15 - Heap Hot Updated, 16 - Heap Only Tuple.

Удаление строки

- xmax актуальной версии строки (xmax=0) устанавливается в xid удаляющей транзакции




```
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len|t_xmin|t_xmax|t_field3|t_ctid|t_infomask2|t_infomask|t_hoff| t_bits |t_oid
--+------+--------+------+------+------+--------+------+-----------+----------+------+--------+-----
 1|  8144|       1|    32|   333|   334|       0 | (0,2)|      16386|      1282 |   24 |        |
 2|  8112|       1|    28|   334|   335|       0 | (0,2)|      40962|      8449 |   24 |10000000|
```


Удаление строки

Пример удаления:

```
delete from t;
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len| xmin| xmax| ctid| infomask2| infomask|t_hoff| t_bits |
--+------+--------+------+-----+-----+-----+----------+---------+------+--------+
 1|  8144|       1|    32|  333|  334|(0,2)|     16386|     1282 |   24 |        |
 2|  8112|       1|    28|  334|  335|(0,2)|     40962|     8449 |   24 |10000000|
select * from t;
 n | c
---+---
(0 rows)
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len| xmin| xmax| ctid| infomask2| infomask|t_hoff| t_bits |
--+------+--------+------+-----+-----+-----+----------+---------+------+--------+
 1|  8144|       1|    32|  333|  334|(0,2)|     16386|     1282 |   24 |        |
 2|  8112|       1|    28|  334|  335|(0,2)|     40962|     9473 |   24 |10000000|
```

Если бы удаление строки не было зафиксировано, а откатилось, то после перечитывания строки infomask второй версии строки вместо 9473 был бы установлен в 10497 (xmax aborted). После отката и вакуумирования:

```
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len| xmin| xmax| ctid| infomask2| infomask|t_hoff| t_bits |
--+------+--------+------+-----+-----+-----+----------+---------+------+--------+
 1|     2|       2|     0|     |     |     |          |         |      |        |
 2|  8144|       1|    28|  334|  335|(0,2)|     40962|   10497 |   24 |10000000|
```

Место первой версии строки освобождено, вторая версия строки перемещена в конец блока. На строку указывает второй указатель.

Типы данных наименьшего размера: boolean, "char", char, smallint

- список типов данных и их характеристики есть в таблице `pg_type`
- если столбец будет использоваться для поиска, стоит оценить эффективность индексирования столбцов, составных индексов, эффективность сканирования индекса доступными способами (Bitmap Index Scan, Index Scan, Index Only Scan)
- Типы данных, занимающие наименьшее место:
  - `boolean` занимает 1 байт
  - `"char"` занимает 1 байт, хранит символы ASCII
  - `char` занимает 2 байта,
    - хранит символы в кодировке базы данных
  - `smallint`, занимает 2 байта
    - хранит целые числа от -32768 до 32767

Типы данных наименьшего размера: boolean, "char", char, smallint

Список типов данных и их характеристики можно найти в таблице pg_type:

```
select typname, typalign, typstorage, typcategory, typlen from pg_type where
typtype='b' and typcategory<>'A' order by typlen,typalign,typname;
```

Тип boolean занимает 1 байт. Тип "char" тоже занимает 1 байт, но хранит символы ASCII.

Можно спутать "char" с char (синоним character(1) или char(1)). char занимает 2 байта, а не 1, но хранит символы в кодировке базы данных, то есть символов больше, чем в кодировке ASCII:

```
drop table if exists t5;
create table t5( c1 "char" default '1');
insert into t5 values(default);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
 lp_off | lp_len | t_hoff | t_data
--------+--------+--------+--------
   8144 |     25 |     24 | \x31
drop table if exists t5;
create table t5( c1 char default '1');
insert into t5 values(default);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
 lp_off | lp_len | t_hoff | t_data
--------+--------+--------+--------
   8144 |     26 |     24 | \x0531
```

"char" занимает 1 байт, а char 2 байта. Почему lp_off (начало строки) одинаков? Потому, что есть выравнивание всей строки по 8 байт и о нем надо помнить. "char" предназначен для использования в таблицах системного каталога, но может использоваться в обычных таблицах. Надо учитывать как будет использоваться столбец. Если для поиска, то оценить эффективность индексирования столбцов, составных индексов, эффективность сканирования индекса доступными способами (Bitmap Index Scan, Index Scan, Index Only Scan).

Третий по компактности тип int2 (синоним smallint), значение этого типа занимает 2 байта. Стоит использовать имя smallint, так как оно определено в стандарте SQL. Диапазон -32768 ..32767.

Типы данных переменной длины

- для строк переменной длины стоит использовать тип text
- размерность для text не указывается
- занимаемое место:
  - один байт, если длина поля меньше 127 байт и строка пустая ''
  - если кодировка UTF8, то ASCII символы занимают 1 байт. Поэтому значение '1' займет 2 байта: \x0531. Значение '11' займёт 3 байта: \x073131. поле состоящее из буквы 'э' займет 3 байта: \x07d18d
  - если длина поля больше 126 символов, то заголовок поля станет 4 байта и поля будут выравниваться по 4 байта
- поля могут сжиматься и оставаться в блоке
- поля могут выноситься в TOAST, оставляя при этом в блоке 18 байт (не выравниваются)
- двоичные данные стоит хранить в типе данных bytea. Это тип данных переменной длины и его поведение такое же, как у типа text

Типы данных переменной длины

Следующими по компактности идут типы данных переменной длины.

Для строк переменной длины стоит использовать тип text. Тип отсутствует в стандарте SQL, но большинство встроенных строковых функций используют text, а не varchar. varchar описан в стандарте SQL. **Для varchar можно указать размерность varchar(1..10485760)**. Размерность для text не указывается. Размерность работает как "домен" (ограничение). На проверку ограничения тратятся ресурсы процессора. Конечно, если ограничение важно для правильности работы приложения (бизнес-правила), то не стоит от них отказываться.

Занимаемое место text и varchar одинаково:

1) Первый байт позволяет различать, что хранится в поле: байт с длиной (нечётные HEX-значения 03, 05, 07...fd, ff) и данные до 126 байт; 4 байта с длиной (первый байт чётное HEX-значение 0c, 10, 14, 18, 20...); поле вынесено в TOAST (0x01); наличие сжатия определяется по значению размера поля.

Например: если поле пустое (''), в первом байте хранится значение \x03. Если поле хранит один байт, то 0x05, если два байта - 0x07.

2) если кодировка UTF8, то ASCII символы занимают 1 байт. Поэтому значение '1' займет 1 байт: 31 (в виде HEX). Значение '11' займёт 2 байта: 3131. Кириллический символ 'э' займет 2 байта: d18d.

3) Опционально нули. **Поля длиной до 127 байт не выравниваются**. Поля от 127 байт выравниваются по pg_type.typalign (i = по 4 байта).

Пример:

```
drop table if exists t5; create table t5(c1 text default '1',c2 text default
'э', c3 text default ''); insert into t5 values(default, default, default);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
 lp_off | lp_len | t_hoff |     t_data
--------+--------+--------+----------------
   8144 |     30 |     24 | \x053107d18d03
```

Поля могут сжиматься и оставаться в блоке. В примере 05 07 03 - длина полей.

Поля могут выноситься в TOAST, оставляя при этом в блоке 18 байт (не выравниваются).

Двоичные данные стоит хранить в типе данных bytea. Это тип данных переменной длины и его поведение такое же, как у типа text. Двоичные данные могут выгружаться командой `COPY` с опцией `WITH BINARY`, иначе по умолчанию они выгружаются в текстовом виде.

Целочисленные типы данных

- целые числа можно хранить в типах `int(integer, int4), bigint (int8), smallint (int2)`
- обычно используются для столбцов PRIMARY KEY
- `bigint` выравнивается по 8 байт
- `int` для первичного или уникального ключа ограничит число строк в таблице 4млрд (2^32)
- для генерации значений для типов `smallint, int` и `bigint` используются последовательности и есть синонимы `smallserial(serial2), serial (serial4), bigserial (serial8)`
- для хранения чисел может использоваться тип переменной длины `numeric` (синоним `decimal`), **накладные расходы 4 байта на хранение длины поля**

Целочисленные типы данных

Целые числа можно хранить в типах int(integer) и bigint (помимо smallint). Эти названия определены в стандарте SQL. Они соответствуют названиям int2, int4 и int8. Эти типы обычно используются для столбцов PRIMARY KEY. bigint выравнивается по 8 байт. Использование int для первичного или уникального ключа ограничит число строк в таблице 4млрд (2^32). Число полей, вынесенных в TOAST-таблицу также ограничено 4млрд(2^32), но это ограничение может быть достигнуто и раньше.

Для генерации значений для типов smallint, int и bigint используются последовательности и есть синонимы smallserial(serial2), serial (serial4), bigserial (serial8). Это автоинкрементальные столбцы. Численные типы знаковые и если использовать только положительные числа, то serial использует диапазон от 1 до 2млрд. (2147483647), а не 4млрд.

Для хранения чисел может использоваться тип переменной длины numeric (синоним decimal), описанный в стандарте SQL. **Накладные расходы 4 байта на хранение длины поля.**

Диапазон для этого типа значительный: 131072 цифр до точки и 16383 цифр после точки. Но если при определении типа указать numeric(точность, масштаб), то максимальные значения точности и масштаба 1000. numeric можно объявить с отрицательным масштабом: значения могут округляться десятков, сотен, тысяч. Кроме чисел и null, numeric поддерживает значения Infinity, -Infinity, NaN.

Преимущество numeric в том, что обычно в столбцах хранятся небольшие числа и поля numeric используют меньше места, **чем десятичные** типы фиксированной длины.

Для обработки десятичных чисел можно использовать numeric, а не float4(real) и не float8(double precision).

Несколько рекомендаций по использованию типов данных:
https://wiki.postgresql.org/wiki/Don't_Do_This

Хранение дат, времени, их интервалов

- для хранения дат, времени, интервалов используются типы:
  - date (4 байта, с точностью до дня)
  - `timestamp, timestamptz, time` точность до микросекунды, размер одинаковый 8 байт, содержимое одинаковое
  - `timetz` - длина 12 байт, `interval` - длина 16 байт
- типы данных `timestamp, timestamptz` не хранят часовой пояс
- `timestamptz` приводит хранимое время к временной зоне клиента
- `timestamptz` физически хранит значения в UTC

```
create table t(t TIMESTAMP, ttz TIMESTAMPTZ);
insert into t values (CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
set timezone='UTC';
select t, ttz from t;
 2024-11-25 23:19:47.833968 | 2024-11-25 20:19:47.833968+00
update t set ttz=t;
select lp_off, lp_len, t_hoff, t_data from heap_page_items(get_raw_page('t','main',0)) order by lp_off;
 lp_off | lp_len | t_hoff |               t_data
--------+--------+--------+-----------------------------------
   8096 |     40 |     24 | \x70580939c1ca020070580939c1ca0200
   8136 |     40 |     24 | \x7044c4bcc3ca020070580939c1ca0200
select t, ttz from t;
 2024-11-25 20:19:47.833968 | 2024-11-25 20:19:47.833968+00
```


Хранение дат, времени, их интервалов

При хранении дат, времени, интервалов стоит учитывать размер, который будут занимать значения выбранного типа в блоках, а также есть ли функции, приведения типов, операторы для выбранного типа.

Наиболее компактный тип для хранения дат это date. Тип данных date занимает всего 4 байта и хранит данные с точностью до суток. Тип данных date не хранит время (часы, минуты).Это не является недостатком, так как не нужно задумываться об округлении до суток при сравнении дат.

Типы данных `timestamp, timestamptz` хранят время и дату с точностью до микросекунды, занимают 8 байт. Оба типа не хранят часовой пояс, значения физически хранятся в одинаковом виде.

`timestamptz` хранят данные в UTC. Тип данных `timestamp` не отображает часовой пояс, не использует временную зону, сохраняет значение как есть (без преобразований). `timestamptz` отображает и выполняет вычисления во временной зоне, задаваемой параметром `timezone`:

```
show timezone;
 Europe/Moscow
create table t(t TIMESTAMP, ttz TIMESTAMPTZ);
insert into t values (CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
SELECT t, ttz FROM t;
 2024-11-25 23:19:47.833968 | 2024-11-25 23:19:47.833968+03
set timezone='UTC';
select t, ttz from t;
 2024-11-25 23:19:47.833968 | 2024-11-25 20:19:47.833968+00
update t set ttz=t;
select lp_off, lp_len, t_hoff, t_data from heap_page_items(get_raw_page('t','main',0)) order by lp_off;
 lp_off | lp_len | t_hoff |               t_data
--------+--------+--------+-----------------------------------
8096    |     40 |     24 | \x70580939c1ca020070580939c1ca0200 -- актуальная версия строки
8136    |     40 |     24 | \x7044c4bcc3ca020070580939c1ca0200 -- старая версия строки
select t, ttz from t;
 2024-11-25 20:19:47.833968 | 2024-11-25 20:19:47.833968+00
```

Тип данных `time` хранит время с точностью до микросекунды и также занимает 8 байт, что довольно много.

Тип данных `timetz` занимает 12 байт. Тип данных `interval` занимает больше всего места, его длина 16 байт. Из-за большего размера эти два типа данных не практичны.

Типы данных для вещественных чисел

- фиксированной длины, с плавающей точкой, с округлением до 6 или 15 разрядов (значащих чисел в десятичном формате):
  - `float4 , real , float(1..24)` - в 4 байтах хранит не менее 6 разрядов
  - `float8 , float , double precision , float(25..53)` в 8 байтах хранит не менее 15 разрядов

```
select 12345678901234567890123456789.1234567890123456789::float4::numeric;
       12345700000000000000000000000
select 12345678901234567890123456789.1234567890123456789::float8::numeric;
       12345678901234600000000000000
```

- переменной длины, **без потери точности** при вычислениях:
  - `numeric , decimal`
  - точность можно задать параметрами: `numeric(presision,scale)`

```
select 1234567890123456789.123456789::numeric + 0.000000000000000000000123456789::numeric;
       1234567890123456789.123456789000000000000123456789
```

- пример экспоненциальной записи одинаковых чисел с разной мантиссой и порядком:

```
select 12345.6::float4, '12.3456e+03'::float4, '123.456e+02'::float4,'1234.56e+01'::float4;
1.235e+04 | 1.235e+04 | 1.235e+04 | 1.235e+04
```


Типы данных для вещественных чисел

Типы данных для работы с вещественными числами:
1) `float4` синоним `real` синоним `float(1..24)`
2) `float8` синоним `float` синоним `double precision` синоним `float(25..53)`
3) `numeric` синоним `decimal`.

`float4` обеспечивает точность 6 разрядов (значащих чисел в десятичной системе счисления), `float8` обеспечивает точность 15 разрядов. Последний разряд округляется:

```
select 12345678901234567890123456789.1234567890123456789 ::float4::numeric;
       12345700000000000000000000000
select 12345678901234567890123456789.1234567890123456789 ::float8::numeric;
       12345678901234600000000000000
```

Красным выделен шестой и пятнадцатый разряды, которые были округлены. Также видно, что разряды больше шестого и пятнадцатого были заменены нулями, что значит что точность не сохраняется. Недостаток этих типов данных в том, что добавление к большому числу маленького числа эквивалентно добавлению нуля:

```
select (12345678901234567890123456789.1234567890123456789::float8 +
123456789::float8)::numeric;
        12345678901234600000000000000
```

Добавление **123456789::float8** эквивалентно добавлению нуля.

Использование float может привести к плохо диагностируемым ошибкам. Например, столбец хранит дальность полёта самолёта, при тестировании на маленькие расстояния самолёт приземляется с точностью до миллиметра, а при полёте на большие расстояния с точностью до километра.

При округлении float8 учитывается шестнадцатый разряд:

```
select 1234567890123499::float8::numeric, 1234567890123499::float8::numeric;
       12345678901234500     |             12345678901234000
select 0.1234567890123499::float8::numeric, 0.1234567890123499::float8::numeric;
       0.12345678901235      |               0.12345678901234
```

При округлении float4 учитывается седьмой разряд:

```
select 123499::float4::numeric, 123449::float4::numeric;
       123500 |                 123450
select 0.123499::float4::numeric, 0.123449::float4::numeric;
       0.12345 |                  0.12345
```

Моментальный снимок

- представляет собой согласованное состояние базы данных на момент времени
- В снимок данных входит:
  - xmin - номер самой старой активной транзакции
  - xmax - значение, на единицу больше номера последней завершенной транзакции
  - xip_list - список активных транзакций
- Функция, возвращающая содержимое моментального снимка и функция экспортирующая его для другой сессии:

```
postgres=# BEGIN TRANSACTION;
postgres=*# select pg_current_snapshot();
 pg_current_snapshot
---------------------
 362:362:
postgres=*# select pg_export_snapshot();
     pg_export_snapshot
----------------------------
 00000024-000000000000000A-1
```


Моментальный снимок

В блоках данных может существовать несколько версий одной и той же строки. Каждая транзакция, несмотря на наличие нескольких версий, видит лишь одну из них. Снимок обеспечивает изоляцию транзакций, предоставляя им образ данных на определенный момент времени, даже если физически в базе могут существовать несколько версий одной строки.

Снимок представляет собой числа:

Нижняя граница снимка, xmin - номер самой старой активной транзакции. Все транзакции с меньшими номерами уже завершены (зафиксированы), и их изменения отражаются в снимке, в то время как транзакции с более высокими номерами могли быть отменены, и их изменения игнорируются.

Верхняя граница снимка, xmax - значение, на единицу больше номера последней завершенной транзакции. Это определяет момент времени, когда был создан снимок. Транзакции с номерами, большими или равными xmax, еще не завершены или не существуют, и, следовательно, изменения, связанные с такими транзакциями, не отображаются в снимке.

Список активных транзакций, xip_list (список транзакций в процессе выполнения), включает номера всех активных транзакций, за исключением виртуальных, которые не оказывают влияния на видимость данных.

Функция, возвращающая содержимое моментального снимка и функция экспортирующая его для другой сессии:

```
postgres=# BEGIN TRANSACTION;
postgres=*# select pg_current_snapshot();
 pg_current_snapshot
---------------------
 362:362:
postgres=*# select pg_export_snapshot();
     pg_export_snapshot
----------------------------
 00000024-000000000000000A-1
```

Транзакция

- Транзакция - набор команд
- Начинается явно или неявно
- Завершается одним из двух действий: фиксация (команды COMMIT, END) или откатом (команда ROLLBACK)
- Результат прерванной (aborted) транзакции такой же, как явно отменённой командой ROLLBACK
- Пример неявного начала транзакции:

```
postgres=# do $$
begin
 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
 perform 1;
end $$;
ERROR: SET TRANSACTION ISOLATION LEVEL must be called before any query
CONTEXT: SQL statement "SET TRANSACTION ISOLATION LEVEL REPEATABLE READ"
PL/pgSQL function inline_code_block line 3 at SQL statement
```

```
postgres=# do $$
begin
 ROLLBACK;
 SET TRANSACTION
ISOLATION LEVEL REPEATABLE
READ;
 perform 1;
 ROLLBACK AND CHAIN;
 perform 1;
 COMMIT AND NO CHAIN;
end $$;
DO
```


Транзакция

Транзакция - набор команд SQL. Начинается явно или неявно

Завершается одним из двух действий: фиксация (команды COMMIT, END) или откатом (команда ROLLBACK)

Результат прерванной (aborted) транзакции такой же как явно отменённой командой ROLLBACK.

Транзакция начинается явно командой BEGIN TRANSACTION или неявно - в блоке plpgsql:

```
postgres=# do $$
begin
 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
 perform 1;
end $$;
ERROR:  SET TRANSACTION ISOLATION LEVEL must be called before any query
CONTEXT:  SQL statement "SET TRANSACTION ISOLATION LEVEL REPEATABLE READ"
PL/pgSQL function inline_code_block line 3 at SQL statement
```

транзакция была начата в анонимном plpgsql блоке.

Чтобы поменять уровень изоляции в анонимном блоке plpgsql нужно откатить транзакцию или зафиксировать её:

```
postgres=# do $$
begin
 ROLLBACK;
 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
 perform 1;
end $$;
DO
```

В PostgreSQL в транзакциях можно выполнять не только select, insert, update, delete, но и почти все команды, в том числе, create, alter, drop, truncate. Нельзя выполнять команды, которые самостоятельно порождают транзакции: `vacuum`, `create/drop database`. Пример:

```
do $$
begin
 begin
  drop table if exists a;
  create table a ( id int);
 end;
 rollback and chain;
 drop table if exists a;
 commit and no chain;
 drop table if exists a;
 rollback and chain;
end $$;
```

Свойства транзакций

- Атомарность (Atomicity) - при фиксации выполнены все команды без исключений, при откате - ни одна команда не выполнена
  - после фиксации, изменения моментально становятся видны другим сессиям
- Целостность (consistency) - отсутствие нарушения декларативных ограничений целостности
- Изоляция (isolation) транзакций друг от друга
  - Реализуется одним из уровней изоляции
- Отказоустойчивость (Durability) - если клиент получил подтверждение об успешности фиксации транзакции (COMMIT COMPLETE), то может быть уверен, что результат транзакции не пропадет

Свойства транзакций

Ценность выполнения команд в транзакциях заключается в свойствах транзакций "ACID":

Атомарность (Atomicity) - при фиксации выполнены все команды без исключений, при откате - ни одна команда не выполнена. Причем, изменения с момента фиксации, одномоментно становятся видны другим сессиям.

Целостность (consistency) - отсутствие нарушения декларативных ограничений целостности (constraints).

Изоляция (isolation) транзакций друг от друга. В SQL реализуется одним из уровней изоляции и блокировками (на уровне строк и объектов).

Отказоустойчивость (Durability) - если клиент получил подтверждение об успешности фиксации транзакции (COMMIT COMPLETE), то может быть уверен, что результат транзакции не пропадет. Это гарантируется программным обеспечением PostgreSQL и администратором кластера баз данных. От администратора требуется не восстанавливать кластера на момент времени в прошлом, не менять параметры отказоустойчивости (`fsync`, `full_page_writes`, `synchronous_commit`). Для защиты от потери кластера администратору стоит обеспечивать правильное резервирование кластера. Например, иметь синхронную физическую реплику или процесс `pg_receivewal`, подтверждающие фиксации транзакций.

Если клиент отправил команду COMMIT, но не получил подтверждение о фиксации транзакций, то транзакция может оказаться зафиксированной или не зафиксированной. Такие случаи должны разрешаться приложением, стандартных возможностей определить статус транзакции нет.

В Oracle Database для таких случаев используются опции Transaction Guard и Application Continuity.

Уровни изоляции транзакций

- READ UNCOMMITTED - чтение незафиксированных данных. В PostgreSQL не используется
- READ COMMITTED - чтение зафиксированных данных. Используется по умолчанию
- REPEATABLE READ - повторяемость по чтению. Стоит использовать только для чтения, а не изменения данных (READ ONLY)
- SERIALIZABLE - упорядоченное выполнение
  - на уровнях REPEATABLE READ и REPEATABLE READ, если менялись данные, возможна ошибка сериализации (serialization failure): "не могу сериализовать доступ" (can't serialize access), транзакция переходит в состояние сбоя и не может зафиксироваться

Уровни изоляции транзакций

Уровни изоляции определяют степень видимости изменений, внесенных одной транзакцией, для других транзакций.

В стандарте SQL определены четыре уровня изоляции:

READ UNCOMMITTED - чтение незафиксированных данных: это самый низкий уровень изоляции. Он позволяет транзакциям видеть изменения, внесенные другими транзакциями, даже если эти изменения еще не были зафиксированы. В PostgreSQL не поддерживается, вместо него используется READ committed.

READ COMMITTED - чтение зафиксированных данных. Команды SELECT видят данные, зафиксированные на момент начала выполнения этого SELECT.

REPEATABLE READ - повторяемость чтения данных. Команды SELECT в одной транзакции не видят изменений, зафиксированных другими транзакциями после начала своей транзакции. Они видят изменения выполненные только в своей транзакции. Начинает транзакцию первая команда, она и формирует моментальный снимок, который используется до конца транзакции. Моментальный снимок и команды SELECT не блокируют строки.

SERIALIZABLE (упорядоченное, последовательное выполнение): при одновременном (с перекрытием по времени) выполнении транзакций этого уровня, они должны выдавать такой же результат, как если бы они фиксировались по очереди во **всех** вариантах перестановок момента фиксации. Это самый высокий уровень изоляции транзакций друг от друга. Чтобы результат не менялся, все транзакции меняющие данные, используемые в транзакциях, должны работать на этом уровне.

На уровнях REPEATABLE READ и REPEATABLE READ, если менялись данные, возможна ошибка сериализации (serialization failure): "не могу сериализовать доступ" (can't serialize access), транзакция переходит в состояние сбоя и не может зафиксироваться, она должна откатиться.

Феномены изоляции транзакций

- в стандарте ISO SQL-92 и последующих определено три феномена конкурентного доступа, которые должны отсутствовать на уровнях изоляции
- на всех уровнях не могут нарушаться ограничения целостности
- грязное чтение в PostgreSQL не допускается ни на одном из уровней изоляции, поэтому уровень Read uncommitted не отличается от Read committed

Уровень изоляции	Грязное чтение (P1)	Неповторяемое чтение (P2)	Фантомное чтение (P3)	Нарушение сериализации
Read uncommitted	Допускается, но не в PG	Возможно	Возможно	Возможно
Read committed	нет	Возможно	Возможно	Возможно
Repeatable read	нет	нет	Допускается, но не в PG	Возможно
Serializable	нет	нет	нет	нет

Феномены изоляции транзакций

В стандарте ISO SQL-92 и последующих определено три феномена (phenomena) конкурентного доступа (изоляции одновременно работающих транзакций), которые должны отсутствовать на уровнях изоляции.

Нарушение сериализации (не феномен, а следствие описания уровня Serializable) - это когда результат успешной фиксации, перекрывающихся по времени транзакций, окажется разным при всевозможных вариантах фиксации этих транзакций по очереди. Также на всех уровнях не могут нарушаться ограничения целостности. Более того, ограничения целостности не зависят от уровней изоляции.

Синоним неповторяемого чтения (P2) - fuzzy read (нечёткое чтение).

Грязное чтение в PostgreSQL не допускается ни на одном из уровней изоляции, поэтому уровень Read uncommitted не отличается от Read committed.

Неповторяемое чтение - при повторном чтении тех же данных, которые уже читались в этой транзакции раньше, обнаруживается, что данные были изменены и зафиксированы другой транзакцией.

Других феноменов в стандартах ISO SQL не описано.

На всех уровнях **изменения не должны теряться** (no updates will be lost). В ANSI SQL упоминались созвучные аномалия lost update (P4) и cursor lost update (P4C), которые допускались на уровне Read committed. **Потерянных обновлений в PostgreSQL нет** ни на одном уровне, так как команды UPDATE, DELETE, столкнувшись с заблокированной строкой, после снятия блокировки, перечитывают поля строки и видят изменения других транзакций, сделанные после начала команды UPDATE, DELETE.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/transaction-iso.html

Пример ошибки сериализации

- Пример ошибки сериализации:

```
 postgres=# drop table if exists a;
DROP TABLE
postgres=# create table a (x int);
CREATE TABLE
1 postgres=# begin transaction isolation
level serializable;
BEGIN
3 postgres=*# insert into b select count(*)
from a;
INSERT 0 1
5 postgres=*# commit;
COMMIT
```

```
 postgres=# drop table if exists b;
DROP TABLE
postgres=# create table b (x int);
CREATE TABLE


 2 postgres=# begin transaction isolation level
serializable;
BEGIN
4 postgres=*# insert into a select count(*) from b;
INSERT 0 1

 6 postgres=*# commit;
ERROR:  could not serialize access due to read/write
dependencies among transactions
DETAIL:  Reason code: Canceled on identification as
a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

- В Oracle Database на уровне SERIALIZABLE ошибки нет

Пример ошибки сериализации

Встречается описание аномалий конкурентного доступа, которым дают названия типа read skew (A5A), write skew (A5B), но такие аномалии субъективны - для кого-то результат фиксации транзакций неожиданный (аномальный), для кого-то ожидаемый.

Например, в Oracle Database уровень Serializable понимается как будто параллельных сессий нет, транзакция видит данные на момент своего начала выполнения, изменения выполненные другими транзакциями ей не видны (как будто их нет). Исходя из такого определения в Oracle Database транзакции уровня SERIALIZABLE могут одновременно выполнять **`INSERT into таблица SELECT`**, не получая ошибок. В PostgreSQL будет выдана ошибка сериализации. Можно субъективно счесть, что это "аномалия lost insert". Но Oracle Database практична и возможность вставки строк с выборкой из других таблиц считается логичным и не приводящим к нарушению бизнес логики. В PostgreSQL вторая транзакция выдает ошибку сериализации, хотя в примере результат фиксации транзакций не зависит от порядка их фиксации: при любом порядке создается две строки со значением 0 и 1. При проверке отсутствия ошибок на уровне SERIALIZABLE PostgreSQL использует "предикатные блокировки" (SIReadLock, Serializable Isolation Read Lock), которые не проверяют досконально нет ли нарушений сериализации, а выдают ошибку, если она потенциально возможна. Если вы видите блокировку с таким названием, то значит есть транзакции этого уровня:

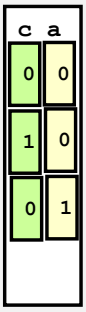
```
select locktype, relation::regclass, mode from pg_locks;
   locktype    | relation |       mode
---------------+----------+------------------
 relation      | b        | SIReadLock
```

В Oracle Database нет уровня Read uncommitted, как и в PostgreSQL, а вместо уровня Repeatable Read используется READ ONLY с повторяемостью по чтению, что уменьшает число ошибок при разработке логики использования транзакций.

В CockroachDB и YDB, по умолчанию, используется уровень изоляции Serializable, но вероятность того, что транзакция не сможет зафиксироваться очень велика и в таких СУБД предусматривают автоматические попытки фиксации на серверной стороне и стороне клиента. При большом числе параллельно работающих транзакций это может привести к уменьшению производительности. Из-за этого такие СУБД нельзя считать универсальными, они имеют свою нишу и последовательность операций в транзакциях, при которых не будет проблем с фиксацией транзакций, а значит производительностью.

Статусы транзакций (CLOG)

- Журнал представляет собой битовый массив, в котором для каждой транзакции отведено два бита
- Значения битов:
  - 00 - транзакция в процессе
  - 01 - зафиксирована
  - 10- прервана (aborted, откачена)
  - 11 - подтранзакция зафиксирована, но родительская транзакция не завершена
- К CLOG обращаются процессы, в том числе вакуум, выполняющий заморозку версий строк, чтобы узнать статус транзакций
- Максимальный размер файлов CLOG зависит от параметра конфигурации autovacuum_freeze_max_age



Статусы транзакций (CLOG)

В журнале статусов транзакций (Commit Log, CLOG) хранятся состояния прошлых транзакций, отстоящих от текущей до значения параметра конфигурации `autovacuum_freeze_max_age`. Журнал представляет собой битовый массив, в котором для каждой транзакции отведено два бита. Массив хранится в файлах директории PGDATA/pg_xact. Файлы полностью копируются в WAL в начале каждой контрольной точки. Для доступа к файлам используется буфер в разделяемой памяти transaction (прежнее название CLOG Buffers):

```
postgres=# SELECT name, allocated_size, pg_size_pretty(allocated_size) from
pg_shmem_allocations where name like '%tran%';
       name       | allocated_size | pg_size_pretty
------------------+----------------+----------------
 subtransaction   |         267520 | 261 kB
 transaction      |         529664 | 517 kB
```

Размер буфера устанавливается параметром конфигурации transaction_buffers.

Статистика использования памяти:

```
postgres=# select name, blks_zeroed, blks_hit, blks_read, blks_written  from
pg_stat_slru where name like '%tran%';
       name       | blks_zeroed | blks_hit | blks_read | blks_written
------------------+-------------+----------+-----------+--------------
 subtransaction   |        9888 |        8 |         0 |         9889
 transaction      |         308 | 24935727 |        24 |          457
```

Значения битов: 00 - транзакция в процессе, 01 - зафиксирована, 10- прервана (aborted, откачена), 11 - подтранзакция зафиксирована, но является подтранзакцией другой транзакции, которая еще не завершена. Подтранзакция создаётся, если в транзакции явно (SAVEPOINT) или неявно (блок EXCEPTION в plpgsql) создаётся точка сохранения. Подтранзакции имеют свои номера и выбирают их из общего счетчика транзакций, что быстрее исчерпывает эти номера.

К CLOG обращаются процессы, в том числе вакуум, выполняющий заморозку версий строк, чтобы узнать статус транзакций. **Максимальный размер файлов CLOG зависит от параметра конфигурации `autovacuum_freeze_max_age`.**

https://eax.me/postgresql-procarray-clog/

Фиксация транзакции

- При фиксации транзакции выполняется запись в журнал транзакций (WAL) о фиксации транзакции
- Выполняется запись бита в буфер журнала CLOG
- Освобождаются ресурсы, которые использовались в процессе транзакции: блокировки, курсоры (кроме курсоров WITH HOLD), контексты (частей) локальной памяти процесса
- Файлы CLOG сохраняются в WAL в начале контрольной точки и изменения в них до следующей контрольной точки не журналируются. При восстановлении после сбоя содержимое CLOG восстанавливается по записям WAL

Фиксация транзакции

При фиксации транзакции, выполняется запись в журнал транзакций (WAL) о фиксации транзакции. Это делается для обеспечения отказоустойчивости. Выполняется запись бита в буфер журнала CLOG. В CLOG для фиксирующейся транзакции устанавливается бит, указывающий на успешную фиксацию (commit). Это позволяет определять, какие транзакции были успешно завершены.

Освобождаются ресурсы, которые использовались в процессе транзакции: блокировки, курсоры (кроме курсоров WITH HOLD), контексты (частей) локальной памяти процесса.

В случае отмены транзакции (ROLLBACK), вместо фиксации в CLOG и журнал записывается информация об отмене транзакции.

Файлы CLOG сохраняются в WAL в начале контрольной точки и изменения в них до следующей контрольной точки не журналируются. При восстановлении после сбоя содержимое CLOG восстанавливается по записям WAL.

Откат и фиксация транзакции происходит одинаково быстро.

Подтранзакции

- подтранзакции это точки сохранения
  - используются чтобы откатиться, а не переводить транзакцию в состояние сбоя
- создаются
  - командой SAVEPOINT
  - секцией EXCEPTION в блоке на языке pl/pgsql
  - в psql при открытии транзакции при установке параметра `\set ON_ERROR_ROLLBACK interactive`
- в структуре PGPROC сохраняется до 64 номеров подтранзакций
- подтранзакциям, которые только читают данные присваивается виртуальный номер
- если встречается команда изменения данных, то подтранзакциям вплоть до основной транзакции присваиваются реальные номера

Подтранзакции

В структуре PGPROC сохраняется до 64 (`PGPROC_MAX_CACHED_SUBXIDS`) подтранзакций. Подтранзакции это точки сохранения, к которым можно было бы откатиться, а не переводить транзакцию в состояние сбоя.

Подтранзакции создаются:

1) командой SAVEPOINT;

2) секцией EXCEPTION в блоке на языке pl/pgsql (точка сохранения неявно устанавливается в начале блока с секцией EXCEPTION).

Подтранзакции могут создаваться в других подтранзакциях и образуется дерево подтранзакций. Подтранзакциям, которые только читают данные присваивается виртуальный номер. Если встречается команда изменения данных, то подтранзакциям вплоть до основной транзакции присваиваются реальные номера. xid дочерней подтранзакции, всегда ниже, чем у родительской.

В структуре PGPROC каждого серверного процесса кэшируется до 64 номеров подтранзакций. Если число подтранзакций будет больше, то накладные расходы на поддержку работы с подтранзакциями существенно возрастают.

В psql есть параметр:

```
postgres=# \set ON_ERROR_ROLLBACK interactive
```

по умолчанию отключён. При использовании значения interactive, при интерактивной работе в psql перед каждой командой в открытой транзакции psql будет устанавливать точку сохранения. За счёт этого, в случае любой ошибки (например, опечатки в команде), последняя команда будет откатываться. Это делает работу в psql более удобной. Устанавливать значение 'on' не стоит, так как при выполнении скриптов (неинтерактивно) если в них будут открываться транзакции или будет отключен режим автофиксации, будут устанавливаться точки сохранения. Это существенно замедлит выполнение команд и будет излишне расходовать номера транзакций.

Типы блокировок

- spinlock (циклическая проверка)
  - Используются для очень краткосрочных действий - не дольше нескольких десятков инструкций процессора
  - Средств мониторинга нет
- легковесные (LWLocks)
  - Используются для доступа к структурам в разделяемой памяти
  - Имеют монопольный (на чтение и изменение) и разделяемый режим (чтение)
  - не больше 200 одновременно
- обычные (тяжеловесные)
  - Автоматически освобождаются по окончании транзакции
  - Есть несколько уровней блокировок
  - Обслуживают блокировки **12 типов**, в том числе advisory locks.
- предикатные блокировки (SIReadLock) используются транзакциями с уровнем изоляции SERIALIZABLE

Типы блокировок

Экземпляр использует блокировки для взаимодействия между процессами:

1) spinlock (циклическая проверка). Используются для очень краткосрочных действий - не дольше нескольких десятков инструкций процессора. Не используются, если выполняется операция ввода-вывода, так как длительность такой операции непредсказуема. Представляет собой переменную в памяти доступ к которой выполняется атомарными инструкциями процессора. Процесс, желающий получить spinlock проверяет статус переменной до тех пор пока она не окажется свободной. Если блокировка не может быть получена в течение минуты, генерируется ошибка. Средств мониторинга нет.

2) Легковесные (LWLocks). Используются для доступа к структурам в разделяемой памяти. Имеют монопольный (на чтение и изменение) и разделяемый режим (чтение). Обнаружения взаимоблокировок нет, они автоматически освобождаются в случае сбоя. Накладные расходы на получение и освобождение блокировки невелики - несколько десятков инструкций процессора, если нет конфликта за блокировку. Ожидание получения блокировки не нагружает процессор. Процессы получают блокировку в порядке очереди. Таймаутов на получение легковесных блокировок нет. При доступе к структурам LWLock используются spinlock. Количество LWLocks ограничено константой: `MAX_SIMUL_LWLOCKS=200`. Есть больше 73 поименованных LWLocks, наборы (tranches) которых защищают доступ к структурам в разделяемой памяти. Их названия присутствуют в событиях ожидания. Примеры названий: XactBuffer, CommitTsBuffer, SubtransBuffer, WALInsert, BufferContent, XidGenLock, OidGenLock.

3) Обычные (тяжеловесные). Автоматически освобождаются по окончании транзакции. Есть процедура обнаружения и разрешения взаимоблокировок. Есть несколько уровней блокировок. Обслуживают блокировки на уровне 12 типов объектов (LockTagTypeNames).

4) Предикатные блокировки (SIReadLock) - используются транзакциями с уровнем изоляции SERIALIZABLE.

Параллельные процессы объединяются в группу с их серверным процессом (лидером группы). Процессы в группе не конфликтуют, что реализуется алгоритмом их работы.

Один из типов блокировок (`pg_locks.locktype`): advisory locks (блокировки на уровне приложения, пользовательские), могут быть получены на уровне сессии и транзакции, управляются кодом приложения.

Во время ожидания получения блокировки процесс не выполняет полезную работу, поэтому чем меньше время ожидания получения блокировок, тем лучше.

Блокировки объектов

- При выполнении команд запрашивается блокировка на объекты, затрагиваемые командой
  - SELECT автоматически запрашивает блокировки ACCESS SHARE на таблицы, индексы, представления
- Пока блокировки не будут получены, команда не начнет выполняться
- Блокировки на уровне объектов используют "честную" очередь блокировок. Это означает, что блокировки будут обслуживаться в порядке их запроса независимо от уровней запрашиваемых блокировок и нет приоритетов
- параметром конфигурации lock_timeout можно установить максимальное время ожидания получения блокировки на объекты или строки

Блокировки объектов

При выполнении команд запрашивается блокировка на объекты, затрагиваемые командой. Например, SELECT для формирования плана выполнения автоматически запрашивает блокировки ACCESS SHARE на таблицы, индексы, представления, используемые в запросе. Пока блокировки не будут получены, команда не начнет выполняться.

Блокировки на уровне объектов используют "честную" очередь блокировок. Это означает, что блокировки будут обслуживаться в порядке их запроса независимо от уровней запрашиваемых блокировок и нет приоритетов.

Параметром конфигурации `lock_timeout` можно установить максимальное время ожидания получения блокировки любого объекта или строки таблицы. Если значение указать без единиц измерения, то используются миллисекунды. Таймаут действует на каждую попытку получения блокировки. При работе со строками таблицы, попыток может быть много при выполнении даже одной команды.

Совместимость блокировок

- слабые блокировки могут быть получены по быстрому пути
- сильные блокировки таблиц препятствуют установку слабых блокировок по быстрому пути
- автовакуум и автоанализ не мешают использовать быстрый путь
- автовакуум и автоанализ не блокируют команды, в случае конфликта блокировок рабочий процесс автовакуума прерывает обработку таблицы

	ACCESS SHARE	ROW SHARE	ROW EXCL.	SHARE UPDATE EXCL.	SHARE	SHARE ROW EXCL.	EXCL.	ACCESS EXCL.
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCL.				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Совместимость блокировок

Слабые блокировки могут быть получены по быстрому пути:

AccessShare - устанавливает SELECT, COPY TO, ALTER TABLE ADD FOREIGN KEY (PARENT) и любой запрос который читает таблицу. Конфликтует только с AccessExclusive.

RowShare - устанавливает SELECT FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE, FOR KEY SHARE. Конфликтует с Exclusive и AccessExclusive.

RowExclusive - устанавливают INSERT, UPDATE, DELETE, MERGE, COPY FROM. Конфликтует с Share, ShareRowExclusive, Exclusive, AccessExclusive.

Не слабая и не сильная блокировка:

ShareUpdateExclusive - **устанавливает автовакуум, автоанализ** и команды VACUUM (без FULL), ANALYZE, CREATE INDEX CONCURRENTLY, DROP INDEX CONCURRENTLY, CREATE STATISTICS, COMMENT ON, REINDEX CONCURRENTLY, ALTER INDEX (RENAME), 11 видов ALTER TABLE

Автовакуум и автоанализ не мешают использовать быстрый путь.

Сильные блокировки, если присутствуют, то не дают устанавливать слабые блокировки по быстрому пути. Их список:

Share - CREATE INDEX (без CONCURRENTLY)

ShareRowExclusive - устанавливает CREATE TRIGGER и некоторыми видами ALTER TABLE

Exclusive - устанавливает REFRESH MATERIALIZED VIEW CONCURRENTLY

AccessExclusive - устанавливает DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL и REFRESH MATERIALIZED VIEW (без CONCURRENTLY), ALTER INDEX, 21 вид ALTER TABLE.

Автовакуум не мешает выполнять команды серверным процессам. Если автовакуум или автоанализ обрабатывает таблицу и серверный процесс запрашивает блокировку, несовместимую с блокировкой которую установил автовакуум (ShareUpdateExclusive), рабочий процесс автовакуума прерывается серверным процессом через `deadlock_timeout` и в диагностический журнал записывается сообщение:

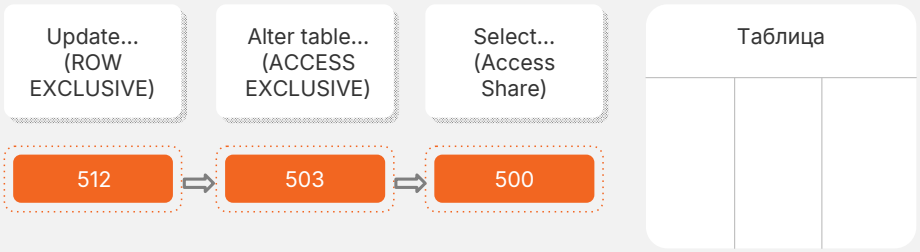
```
ERROR:  canceling autovacuum task
DETAIL: automatic vacuum of table 'имя'
```

Автовакуум в следующем цикле попробует снова обработать таблицу и ее индексы.

https://pglocks.org/

Блокировки объектов

- Блокировки на уровне объектов формируют очереди
- Уровень блокировок может быть совместим
- Очередь "честная" и транзакцию с высоким уровнем блокирования нельзя обогнать:



Блокировки объектов

Например, существует таблица. К ней происходит обращение в транзакции 500 с целью выборки данных SELECT. Накладывается блокировка Access Share. Параллельно через некоторое время с транзакции 503 приходит команда Alter table (ACCESS EXCLUSIVE). Транзакция встает в очередь Если придет еще одна транзакция, которая не совместима по уровню блокировок, например, с номером 512 Update (ROW EXCLUSIVE) она также встанет в очередь. Для произведения своих действий, транзакции будут ждать пока не отработает предыдущая.

Уровень блокировок может быть совместим. К примеру, если с обновлением строк придет обновление строк той же таблицы но других, эти транзакции могут делать свою работу параллельно.

Блокировки строк

- FOR UPDATE - устанавливает DELETE и UPDATE, который меняет значение в столбце, входящим в уникальный индекс (не по выражению и не частичный), а также SELECT FOR UPDATE
- FOR NO KEY UPDATE устанавливают все остальные UPDATE
- Если не планируете удалить строку или менять значение в ключевом столбце, всегда используйте SELECT FOR NO KEY UPDATE
- Таблица совместимости:

запрашиваемый режим	строка заблокирована в режиме			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

Блокировки строк

Блокировки на уровне строк, устанавливаются автоматически.

Транзакция может удерживать конфликтующие блокировки на одной и той же строке, но помимо этого, две транзакции никогда не могут удерживать конфликтующие блокировки на одной и той же строке. Блокировки на уровне строк не влияют на запросы данных; они блокируют только писателей и блокировщиков для одной и той же строки.

Блокировки на уровне строк освобождаются при завершении транзакции или во время отката точки сохранения, так же как и блокировки на уровне таблицы. Режимы блокирования:

FOR UPDATE: запрашивает блокировку строк для операций обновления, предотвращая их изменение или блокировку другими транзакциями до завершения текущей транзакции. Устанавливается перед выполнением команды DELETE или UPDATE, которая изменяет значение в столбце, входящим в уникальный индекс, который не содержит выражений и не является частичным индексом, а также командой SELECT FOR UPDATE.

FOR NO KEY UPDATE: устанавливается перед всеми остальными UPDATE, не влияет на команды SELECT FOR KEY SHARE.

FOR SHARE: автоматически не ставится командами, только SELECT FOR SHARE.

FOR KEY SHARE: аналогичен FOR SHARE, но блокирует SELECT FOR UPDATE и не влияет на SELECT FOR NO KEY UPDATE.

PostgreSQL не сохраняет информацию об измененных строках в памяти, и нет ограничений на число одновременно заблокированных строк.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/explicit-locking.html

Если вы используете SELECT FOR UPDATE по таблице, на которую ссылается внешний ключ, вы заблокируете INSERT строк в дочернюю таблицу, которые будут относиться к заблокированной строке родительской таблицы.

Использование SELECT FOR UPDATE плохо сказывается на конкурентном доступе, так как приводит к ненужным блокировкам. **Если вы не планируете удалить строку или изменять значение в ключевом столбце, всегда используйте SELECT FOR NO KEY UPDATE, а не SELECT FOR UPDATE.**

https://habr.com/ru/companies/tantor/articles/940066/

Мультитранзакции

- блокировка FOR NO KEY UPDATE устанавливается командой UPDATE, которая не вносит изменения в ключевые столбцы
- блокировка FOR KEY SHARE устанавливается командой DELETE и UPDATE которая обновляет значения ключевых столбцов
- также разделяемые блокировки устанавливаются командами SELECT .. FOR SHARE, FOR NO KEY UPDATE, FOR KEY SHARE
- разделяемые блокировки позволяют одновременно работать со строкой нескольким транзакциям
- одновременная работа реализуется "мультитранзакциями", которые имеют свой счетчик `xid`, свои файлы и кэши
- соответствие между `xid` транзакций и мультитранзакций хранятся в директории `PGDATA/pg_multixact`
- advisory locks не являются заменой блокировок строк, так как их количество ограничено

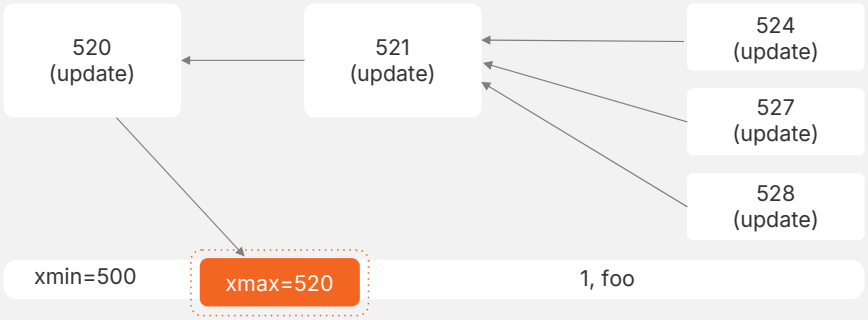
Мультитранзакции

Команды SELECT .. FOR SHARE, FOR NO KEY UPDATE, FOR KEY SHARE позволяют одновременно работать со строкой нескольким транзакциям. Блокировка FOR NO KEY UPDATE устанавливается командой UPDATE, которая не вносит изменения в ключевые столбцы. Блокировка FOR KEY SHARE устанавливается командой DELETE и UPDATE которая обновляет значения ключевых столбцов. Более детальные формулировки есть в документации. Важно то, что обычные команды DELETE и UPDATE могут устанавливать на строки разделяемые блокировки. При появлении второй транзакции, пока работает первая, второй серверный процесс создаст мультитранзакцию. Большинство приложений, которые преимущественно создают строки, не испытывают проблем, так как вставляемая строка другим сессиям не видна и они не могут ее заблокировать. Конфликт может возникнуть при вставке записи в уникальный индекс и тогда вторая транзакция будет ждать (мультитранзакций не будет). Да и это маловероятно, так как правильно спроектированные приложения используют автоинкрементальные столбцы. Обновление строк трудоемкая операция во всех реляционных СУБД, а в PostgreSQL особенно из-за того, что PostgreSQL сохраняет старые версии строк в блоках данных. Если архитектор (дизайнер) приложения активно использует UPDATE, то помимо уменьшения доли HOT cleanup, возможно, что часть транзакций "столкнется" на части строк и второй серверный процесс создаст мультитранзакцию. Последующие транзакции могут присоединяться к мультитранзакции, то есть транзакций может быть две и больше. Причем, создается новая мультитранзакция, куда включаются прежние транзакции. Это не оптимально, но вероятность того, что строку захотят менять не две, а три и более транзакции обычно не высока.

Если в приложении возникают взаимоблокировки, то это прямо указывает на ошибки в архитектуре приложения. Если вместо изменения логики работы с данными использовать разделяемые блокировки, то взаимоблокировки могут прекратиться, но производительность не улучшится.

Очередь при блокировке строки

- есть первый в очереди и остальные
- первого в очереди может обогнать транзакция, чей уровень блокирования строки совместим с уровнем транзакции, уже заблокировавшей строку
- если первый в очереди получает блокировку, то на его место выбирается транзакция случайным образом независимо от длительности ожидания



Очередь при блокировке строки

Признаком блокировки строки является заполнение поля xmax в заголовке версии строки.

Если придет транзакция, не совместимая по уровню блокировки, то она встает в очередь, пытаясь захватить номер транзакции 520.

Остальные транзакции становятся в очередь за транзакцией 521.

В случае освобождения транзакции 520, транзакция 521 захватывает новую версию строки, а следующей становится произвольная транзакция из "кучи" ожидающих транзакций. Можно сказать, что в очереди есть первый в очереди за получением блокировки и все остальные.

Первого в очереди может обогнать транзакция, чей уровень блокирования строки совместим с уровнем транзакции, уже заблокировавшей строку.

Также любые транзакции (521-528) могут обгонять транзакции, которые совместимы по блокировке с заблокировавшей строку (500) и организовывать мультитранзакции. Остальные транзакции (521-528) будут ждать завершения всех этих транзакций (организованных в мультитранзакцию.

Практика

1. Вставка, обновление и удаление строки
2. Видимость версии строки на различных уровнях изоляции
3. Состояние транзакции по CLOG
4. Блокировка таблицы
5. Блокировка строки

Практика

Вставка, обновление и удаление строки
Видимость версии строки на различных уровнях изоляции
Состояние транзакции по CLOG
Блокировка таблицы
Блокировка строки

2

2c Архитектура PostgreSQL

Регламентные работы

Автовакуум

- выполняется рабочими процессами автовакуума
- выбираются таблицы, в которых было обновлено или вставлено больше 20% от размера таблицы
- если давно не было заморозки версий строк, то выполняется заморозка
- после вакуумирования таблицы выполняет автоанализ
- Автовакуум запрашивает блокировку уровня SHARE UPDATE EXCLUSIVE и если не может получить блокировку на таблицу, то эта таблица не вакуумируется в этом цикле автовакуума
- временные таблицы автовакуум не обрабатывает

Автовакуум

Регламентная очистка (Routine Vacuuming, рутинное вакуумирование) выполняется рабочими процессами автовакуума (autovacuum workers). Автовакуум выбирает таблицы, в которых изменилось `autovacuum_vacuum_scale_factor` от размера таблицы или было вставлено `autovacuum_vacuum_insert_scale_factor` от размера таблицы. По умолчанию значения установлены в 20%. В процессе вакуумирования:

1) очищаются версии строк таблицы, вышедшие за горизонт базы данных. Блоки, в которых есть только актуальные версии строк пропускаются
2) очищаются записи в блоках индекса, которые указывают на очищаемые версии строк
3) создается или обновляется файл карты видимости (visibility map)
4) создаются или обновляются файлы карты свободного места (free space map)
5) очищаются версии строк TOAST-таблицы и TOAST-индекса

Если заморозка версий строк таблицы выполнялась больше, чем `autovacuum_freeze_max_age` транзакций назад. xid последней заморозки сохраняется для таблиц в столбцах `relfrozenxid` и `relminmxid` таблицы системного каталога `pg_class`. Значение по умолчанию: 200млн., если счетчик транзакций 32-битный (PostgreSQL и Tantor Postgres BE) и 10млрд., если 64-битный (в Tantor Postgres SE и SE 1C). Периодическая заморозка нужна для предотвращения прекращения выдачи новых номеров транзакций и остановки обслуживания пока не будет выполнена заморозка.

Формула срабатывания: `age(pg_class.relfrozenxid) > vacuum_freeze_table_age - vacuum_freeze_min_age` или `mxid_age(pg_class.relminmxid) > vacuum_multixact_freeze_table_age - vacuum_freeze_min_age`.

Автовакуум запрашивает блокировку уровня SHARE UPDATE EXCLUSIVE и, если не может получить блокировку на таблицу, то эта таблица не вакуумируется в этом цикле автовакуума.

После вакуумирования таблицы выполняет автоанализ, если поменялось больше, чем `autovacuum_analyze_scale_factor` (по умолчанию 10%) строк таблицы.

Временные таблицы автовакуум не обрабатывает. На физических репликах автовакуум не работает, так как через журнал передаются изменения, сделанные автовакуумом на мастере и их повторяет процесс startup. **Процесс startup может конфликтовать с серверными процессами (и на замораживаемых и на очищаемых блоках на мастере вакуумом и с быстрой очисткой), обслуживающими запросы на реплике из-за чего реплика начинает отставать от мастера.**

https://habr.com/ru/articles/1025254/

Процессы автовакуума

- В 18 версии появился параметр autovacuum_worker_slots (по умолчанию 16), который ограничивает число рабочих процессов автовакуума. Значение этого параметра не меняется без перезапуска экземпляра
- Начиная с 18 версии можно без перезапуска экземпляра менять значение параметра autovacuum_max_workers (по умолчанию 3)
- Если одна таблица вакуумируется дольше log_autovacuum_min_duration (по умолчанию, 10 минут), то в диагностический лог кластера выводится сообщение

Процессы автовакуума

На экземпляре работает фоновый процесс autovacuum launcher, если параметр конфигурации autovacuum не был отключён. Этот процесс для каждой базы данных, в которых была активность, каждые autovacuum_naptime (по умолчанию, 1 минута) запускает рабочий процесс автовакуума (autovacuum worker), который составляет список таблиц базы, которые планирует очищать, замораживать или собирать статистику.

Таблицы, у которых pg_claass.relfrozenxid (их и их TOAST-таблиц) отстаёт больше, чем на autovacuum_freeze_max_age транзакций, всегда вакуумируются, независимо от числа изменившихся в них строк.

Если имеется N баз данных, то рабочий процесс (если число работающих процессов не превысило autovacuum_max_workers) будет запускаться раз в autovacuum_naptime/N. Если autovacuum_max_workers достигнуто, а число баз данных, в которых была активность, превышает autovacuum_max_workers, обработка следующей базы начинается без задержки, по завершении обработки базы рабочие процессы принимаются за следующую базу. Если базы обработаны, то освободившийся процесс начнет вакуумировать базу, с которой всё ещё работают другие рабочие процессы автовакуума. Несколько процессов автовакуума могут обрабатывать разные таблицы (секции таблиц) одной и той же базы данных.

В 18 версии появился параметр autovacuum_worker_slots (по умолчанию 16), который ограничивает число рабочих процессов автовакуума. Значение этого параметра не меняется без перезапуска экземпляра.

Начиная с 18 версии можно без перезапуска экземпляра менять значение параметра autovacuum_max_workers (по умолчанию 3) вплоть до autovacuum_worker_slots.

Память, которую выделяет каждый рабочий процесс автовакуума, устанавливается параметром autovacuum_work_mem, по умолчанию равен -1, что означет, что берётся значение maintenance_work_mem (по умолчанию, 64Мб).

Если одна таблица вакуумируется дольше log_autovacuum_min_duration (по умолчанию, 10 минут), то в диагностический лог кластера выводится сообщение. На такие сообщения стоит обращать внимание. Если они появляются, то нужно настраивать работу автовакуума.

Начиная с 15 версии, есть оптимизация: если число блоков таблицы со старыми версиями строк меньше 2% от числа блоков в таблице, то индексы на таблице не вакуумируются.

Энергичная заморозка

- Если автовакуум долго не обрабатывал таблицу в режиме заморозки, цикл автовакуума он запустится в агрессивном режиме
- В агрессивном режиме автовакуум ждёт получения блокировки и не пропускает закреплённые другими процессами блоки
- Энергичная заморозка сканирует 20% блоков из карты видимости с битом all_visible и агрессивному режиму заморозки остаётся заморозить меньше блоков
- Карта заморозки была встроена в карту видимости в версии PostgreSQL 9.6

Энергичная заморозка

Если автовакуум долго не обрабатывал таблицу в режиме заморозки, цикл автовакуума он запустится в агрессивном режиме. В агрессивном режиме автовакуум ждёт получения блокировки и не пропускает закреплённые другими процессами блоки. Если таких блоков много, то автовакуум долго обрабатывает таблицу.

В 18 версии добавилось "энергичное" (eager) вакуумирование с подмораживанием части (20%) блоков. Автовакуум при обычном (не агрессивном) сканировании будет сканировать блоки с all_visible, но не all_frozen битами. Вероятность, что оба бита будут установлены, на 18 версии повышается. До 18 версии у большей части блоков был установлен бит all_visible, но не all_frozen. Алгорим "умный": чтобы не морозить всё за одно вакуумирование, а растянуть на несколько вакуумирований, алгоритм морозит только 20% от all_visible блоков за одно вакуумирование. Сканировать блоки с all_visible, но не all_frozen битами автовакуум начнёт, если pg_claass.relfrozenxid превысит vacuum_freeze_table_age транзакций. Заморозка растягивается на 5 проходов автовакуума.

Параметр `vacuum_max_eager_freeze_failure_rate` = 0.03 (3%) означает, что если на 3% от общего числа блоков в таблице в процессе сканирования 20% блоков из карты видимости с битом all_visible, не сможет быть установлен бит all_frozen, то прекратить сканировать блоки all_visible. Причина, по которой не может быть выставлен all_frozen в том, что блок закреплён каким-то другим процессом (pincount>0), то есть из блока читаются строки. Если бы строки менялись, то процесс убрал бы бит all_visible. Автовакууму нужно эксклюзивно закрепить блок, чтобы на каждой строке в этом блоке установить признак заморозки, только после этого он сможет установить бит в карте заморозки.

Карта заморозки была встроена в карту видимости в версии 9.6. До 18 версии проблема сглаживалась тем, что заморозив блок, вакуум отмечал его в карте заморозки битом all_frozen и блоки с таким битом при вакуумировании в режиме заморозки не считывались. И если вакуум периодически обрабатывал таблицу в режиме заморозки, пока она росла и накапливала строки, то после того, как таблица дорастала до терабайт, заморозка на этой таблице была не такой долгой, так как большая часть блоков уже была заморожена. Если в таблице массово менялось, вставлялось или удалялось большое число строк, то цикл заморозки мог идти долго, а на таблице с большим числом "горячих" блоков ещё дольше.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/routine-vacuuming.html

Представление pg_stat_progress_vacuum

- содержит по одной строке для каждого серверного процесса, выполняющего команду VACUUM и каждого autovacuum worker выполняющих вакуумирование в момент обращения к представлению
- в столбце phase отражается текущая фаза вакуума: `initializing` (подготовительная, проходит быстро), `scanning heap`, `vacuuming indexes`, `vacuuming heap`, `cleaning up indexes`, `truncating heap`, `performing final cleanup` (финальная)
- по столбцам `heap_blks_total` , `heap_blks_scanned` , `heap_blks_vacuumed` оценить ход выполнения очистки
- `num_dead_tuples` - число TID, которые сейчас помещены в структуру памяти. Если достигнет `max_dead_tuples` увеличится значение в **`index_vacuum_count`**
- `VACUUM` **`FULL`** отслеживается через `pg_stat_progress_`**`cluster`**
- `ANALYZE` отслеживается через `pg_stat_progress_analyze`

Представление pg_stat_progress_vacuum

Представление **pg_stat_progress_vacuum** содержит по одной строке для каждого серверного процесса, выполняющего команду VACUUM и каждого autovacuum worker, выполняющих вакуумирование в момент обращения к представлению.

Выполнения VACUUM FULL отслеживается через представление `pg_stat_progress_cluster`. VACUUM FULL является частным случаем команды CLUSTER и выполняется тем же кодом. Вместо VACUUM FULL оптимально использовать CLUSTER, так как она создает файлы данных со строками в упорядоченном виде.

Команда ANALYZE отслеживается через представление `pg_stat_progress_analyze`.

В столбце phase отражается текущая фаза вакуума: `initializing` (подготовительная, проходит быстро), `scanning heap`, `vacuuming indexes`, `vacuuming heap`, `cleaning up indexes`, `truncating heap`, `performing final cleanup` (финальная).

Столбцы `heap_blks_total` , `heap_blks_scanned` , `heap_blks_vacuumed` выдают значения в блоках. По значениям можно оценить размер таблицы и сколько блоков уже обработано (оценить ход выполнения очистки).

`max_dead_tuples` - оценка максимального числа идентификаторов строк (TID), которые поместятся в память, ограниченную параметром `autovacuum_work_mem` или `maintenance_work_mem`, действующим для процесса, к которому относится строка представления.

`num_dead_tuples` - число TID, которые сейчас помещены в структуру памяти. Если число достигнет значения, при котором память будет исчерпана (`max_dead_tuples`) начнется фаза очистки индексов и увеличится значение в поле index_vacuum_count.

Одновременно можно использовать представление pg_stat_activity, в котором также отображаются действия серверных процессов и рабочих процессов автовакуума (autovacuum workers). Это представление полезно тем, что показывает не ожидает ли чего-то процесс.

Параметры команды VACUUM

- `DISABLE_PAGE_SKIPPING` обрабатывает все блоки таблиц без исключения
- `SKIP_LOCKED false` - не дает пропускать заблокированные объекты, секции таблиц, блоки
- `INDEX_CLEANUP auto/on/off` указывает нужно ли обрабатывать индексы. OFF используется если нужно быстрее убрать мертвые строки из блоков таблиц
- `PROCESS_TOAST false` - отключает обработку таблиц TOAST
- `TRUNCATE false` - отключает пятую фазу
- `PARALLEL n`. Число *n* ограничивает число фоновых процессов
- `FULL` полная очистка, использует монопольные блокировки, последовательно устанавливаемые на каждую обрабатываемую таблицу

Параметры команды VACUUM

Вакуум можно вызвать вручную, он будет выполняться серверным процессом. Алгоритм выполнения такой же как и у автовакуума и программный код тот же самый, только команде можно передать опции выполнения. Команду VACUUM имеет смысл выполнять после создания таблиц или загрузки данных. Параметры:

`DISABLE_PAGE_SKIPPING` обрабатывает все блоки таблиц без исключения. Если блоки заблокированы, ждет получения блокировки. Включает в себя опцию FREEZE.

`SKIP_LOCKED false` - не дает пропускать заблокированные объекты, секции таблиц, блоки

`INDEX_CLEANUP auto/on/off` указывает нужно ли обрабатывать индексы. OFF используется при приближении к wrap around, когда нужно быстрее убрать мертвые строки из блоков таблиц.

`PROCESS_TOAST false` - отключает обработку таблиц TOAST

`PROCESS_MAIN false` - отключает обработку таблиц и обрабатывает TOAST

`TRUNCATE false` - отключает пятую фазу. На этой фазе устанавливается монопольная блокировка. При ожидании дольше 5 секунд по каждой таблице фаза пропускается. При постановке в очередь монопольная блокировка заставляет ждать все команды, желающие работать с таблицей. Можно установить параметр vacuum_truncate off на уровне таблицы.

В 18 версии PostgreSQL появился параметр конфигурации vacuum_truncate.

`PARALLEL n`. Число *n* ограничивает число фоновых процессов. Их также ограничивает значение параметра max_parallel_maintenance_workers. Параллельные процессы используются, если размер индекса превышает min_parallel_index_scan_size и таких индексов больше одного. На анализ не влияет, только на фазу обработки индексов.

`FULL` полная очистка, использует монопольные блокировки, последовательно устанавливаемые на каждую обрабатываемую таблицу. Требует дополнительное место на диске, так как создаются новые файлы и старые файлы не удаляются до конца транзакции. Возможно, стоит использовать команду `CLUSTER`, так как она выполняется так же, но при этом упорядочивает строки.

Параметры команды VACUUM

- SKIP_DATABASE_STATS отключает обновление pg_database.datfrozenxid
  - позволяет не выполнять полное сканирование таблицы pg_class
- VERBOSE - выводит статистику выполнения команды
- FREEZE - выполняет заморозку строк во всех блоках, кроме тех в которых все строки актуальны и заморожены
- BUFFER_USAGE_LIMIT размер буферного кольца вместо vacuum_buffer_usage_limit
  - в отличие от параметра конфигурации, BUFFER_USAGE_LIMIT можно установить в значение ноль и буферное кольцо не будет использоваться

Параметры команды VACUUM (продолжение)

SKIP_DATABASE_STATS отключает обновление числа pg_database.datfrozenxid - самый старый не замороженный XID в объектах базы данных. Для получения значения выполняется запрос к relfrozenxid, relminmxid из pg_class путем полного сканирования (нет индекса по этим столбцам и он не нужен). Если размер pg_class большой, то запрос тратит ресурсы. Можно отключать это и оставлять для любого VACUUM по любой таблице, например, раз в сутки, либо использовать:

VACUUM (ONLY_DATABASE_STATS VERBOSE) который ничего очищать не будет, а только обновит значение pg_database.datfrozenxid.

VERBOSE - выводит статистику выполнения команды. Дополнительной нагрузки не даёт, рекомендуется использовать.

ANALYZE - обновляет статистику. Обновление выполняется отдельно. Совмещение вакуумирования и анализа в одной команде не даёт преимуществ по производительности.

FREEZE - выполняет заморозку строк во всех блоках, кроме тех в которых все строки актуальны и заморожены. Называется "aggressive" режимом. Добавление указания FREEZE равносильно выполнению команды VACUUM с параметрами vacuum_freeze_min_age=0 и vacuum_freeze_table_age=0. В режиме FULL использование FREEZE избыточно, так как FULL тоже замораживает строки.

BUFFER_USAGE_LIMIT размер буферного кольца вместо параметра конфигурации vacuum_buffer_usage_limit (диапазон от 128Кб до 16Мб, по умолчанию 256Мб). В отличие от параметра конфигурации, BUFFER_USAGE_LIMIT можно установить в значение ноль. В этом случае, буферное кольцо не используется и блоки всех обрабатываемых командой объектов как при очистке, так и при анализе, могут занять все буфера. Это ускорит выполнение вакуумирования и если кэш буферов большой, загрузит обрабатываемые блоки в него. Пример команды:

```
VACUUM(ANALYZE, BUFFER_USAGE_LIMIT 0);
```

Если автовакуум запускается для защиты от переполнения счетчика транзакций, то буферное кольцо не используется и автоочистка выполняется в агрессивном режиме.

В 18 версии в команде VACUUM и ANALYZE перед именем таблицы можно поставить ONLY, тогда командой будет обработана только секционированная таблица, без секций и таблиц-наследников. Это можно использовать для сбора статистики в целом по секционированной таблице.

Параметр `default_statistics_target`

- устанавливает
  - число наиболее часто встречающихся значений в столбцах таблиц (`pg_stats.most_common_vals`)
  - количество корзин в гистрограммах распределения значений в столбцах (`pg_stats.histogram_bounds`)
  - число строк (`default_statistics_target * 300`) случайной выборки по которым собирается статистика
- по умолчанию 100
- максимальное значение 10000
- можно установить для конкретного столбца таблицы или индекса по выражению:

```
alter table test alter column id set statistics 10000;
alter index test alter column  1 set statistics 10000;
```


Параметр конфигурации `default_statistics_target`

Для сбора статистики используется случайная выборка числом `300*default_statistics_target` строк. Значение по умолчанию 100. Максимальное значение 10000. Значение по умолчанию достаточно для репрезентативной выборки и достаточной точности. Помимо этого параметр устанавливает число наиболее часто встречающихся значений в столбцах таблиц (`pg_stats.most_common_vals`) и количество корзин в гистограммах распределения значений в столбцах (`pg_stats.histogram_bounds`). Если в таблице много строк, распределение значений неравномерное, то можно увеличить значение для столбца таблицы командой:

```
alter table test alter column id set statistics 10000;
```

и планировщик будет более точно вычислять стоимость.

Чем больше значение, тем больше времени потребуется для автоанализа и объем статистики будет больше.

Значение `-1` возвращает к применению параметра `default_statistics_target`. Команда запрашивает на таблицу блокировку `SHARE UPDATE EXCLUSIVE`.

Для индексов, где индексируются выражения (индекс основанный на функции) можно установить значение командой:

```
alter index test alter column 1 set statistics 10000;
```

Так как у выражений нет уникальных имён, указывается порядковый номер столбца в индексе. Диапазон значений: 0..10000; Значение `-1` возвращает к применению параметра `default_statistics_target`.

Значение параметра в диапазоне от 100 до 10000 не влияет на длительность выполнения цикла автоанализа.

Раздувание (bloat) таблиц и индексов

- автовакуум может не обработать таблицу из-за того, что:
  - горизонт базы данных долго не сдвигался
  - в момент обращения автовакуума к таблице на ней была установлена несовместимая с автовакуумом блокировка
- после того как автовакуум отработает, размеры файлов вряд ли уменьшатся
- блоки будут использоваться в будущем под новые версии строк
- слишком часто заниматься мониторингом не нужно, более актуальна проверка свободного места на дисках
- расширение для оценки числа версий строк:

```
create extension pgstattuple;
\dx+ pgstattuple
select relname, b.* from pg_class, pgstattuple_approx(oid) b WHERE relkind='r';
select relname, b.* from pg_class, pgstatindex(oid) b WHERE relkind='i' order by 10;
```


Раздувание (bloat) таблиц и индексов

Старые версии строк хранятся в блоках таблиц. В индексах сохраняются ссылки на версии строк, в том числе старые версии. Автовакуум может не обработать таблицу из-за того, что горизонт базы данных долго не сдвигался или в момент обращения к таблице на ней была установлена несовместимая с автовакуумом блокировка. Во втором случае автовакуум пропускает обработку таблицы. Это приводит к увеличению размера файлов таблиц и индексов. После того как автовакуум отработает, размеры файлов вряд ли уменьшатся. Блоки будут использоваться в будущем под новые версии строк. Раздуванием (bloat) таблиц и индексов можно считать увеличение размера так, что свободное место не будет использовано в ближайшее время. Если размер объекта большой, то неиспользуемое место может быть ощутимо для администратора. Найти таблицы, в которых имеется неиспользуемое место и запустить задачи обслуживания можно с помощью Платформы Тантор.

Можно оценить неиспользуемое место по базовой статистике, собираемой автоанализом. Объекты вряд ли раздуваются быстро, поэтому не нужно часто заниматься мониторингом. Мониторинг свободного места на дисках более актуален. Точность оценки можно верифицировать (сопоставить с реальностью), выполнив полное вакуумирование (`CLUSTER` или `VACUUM FULL`) и сравнить результат с оценкой.

Можно использовать функции стандартного расширения `pgstattuple`:

```
create extension pgstattuple;
\dx+ pgstattuple
select relname, b.* from pg_class, pgstattuple_approx(oid) b WHERE relkind='r' order by 9 desc;
select relname, b.* from pg_class, pgstatindex(oid) b WHERE relkind='i' order by 10;
```

Оценивать можно по столбцам `dead_tuple_percent` для таблиц и `avg_leaf_density` для индексов:

```
relname             | t
table_len           | 8192
scanned_percent     | 100
approx_tuple_count  | 1
approx_tuple_len    | 32
approx_tuple_percent| 0.390625
dead_tuple_count    | 0
dead_tuple_len      | 0
dead_tuple_percent  | 0
approx_free_space   | 8112
approx_free_percent | 99.0234375
```

Внутристраничное обновление (HOT update)

- HOT update - при обновлении (`UPDATE`) строки не вносятся изменения в индексы
- условия HOT update:
  - изменяются только поля, не входящие ни в один из индексов (кроме индексов типа brin), созданных на таблице
  - новая версия строки размещается в том же блоке, что и прежняя версия
- если новая версия строки размещается в блоке, отличном от того в котором находится прежняя версия строки, то:
  - прежняя версия станет последней в цепочке HOT-версий
  - во всех индексах, созданных на таблицу будут созданы новые записи, указывающие на новую версию строки

Внутристраничное обновление (HOT update)

Heap Only Tuple update (HOT update) - оптимизация, которая позволяет вставить новую версию строки, не внося изменения в блоки индексов, созданных на таблицу. При обновлении строки (UPDATE) создается новая строка внутри блока таблицы. В индексах хранится адрес версии строки (`ctid`) , которая у новой версии будет другая и без оптимизации пришлось бы вставлять записи во все индексы на таблице. Индексные записи указывают (`ctid`) на поле в заголовке блока. Но, если **меняются только поля, не упомянутые ни в одном из индексов (кроме индексов типа brin)**, то в индексы не вносятся изменения. В этом и состоит преимущество HOT update - не нужно вносить изменения в блоки индексов.

Частичный индекс:

```
create index t5_idx on t5 (c1) where c1 is not null;
```

не дает выполнять HOT, если в команде UPDATE упоминается столбец c1 даже, если в `UPDATE` стоит условие `WHERE c1 is null`.

Аналогично частичный покрывающий индекс:

```
create index t5_idx1 on t5 (c1) include (c2) where c1 is not null;
```

не дает выполнять HOT, если в команде UPDATE упоминаются столбцы c1 и c2.

Из индекса серверный процесс попадает на старую версию строки, видит бит HEAP_HOT_UPDATED, переходит по полю t_ctid на новую версию строки (с учетом правил видимости, если ему видна эта версия, то серверный процесс на ней и останавливается), проверяет тот же бит, если он установлен, то переходит дальше на более новую версию строки. Такие версии строки называются HOT-цепочкой версий (HOT chain). С учетом правил видимости серверный процесс может дойти до самой свежей версии строки, на которой установлен бит HEAP_ONLY_TUPLE и остановиться на ней.

Если новая версия строки размещается в блоке, отличном от того в котором находится прежняя версия строки, то HOT не применяется. Поле t_ctid прежней версии будет ссылаться на более новую версию в другом блоке, но бит HEAP_HOT_UPDATED не будет установлен. Прежняя версия станет последней в цепочке HOT-версий. Во всех индексах на таблице будут созданы новые записи, указывающие на новую версию строки.

Примечание: В любом случае, если поля, вынесенные в TOAST-таблицу не изменялись, то содержимое полей, ссылающихся на записи в TOAST, будут скопированы без изменений и в TOAST-таблицах не будет изменений.

Мониторинг HOT update

- статистика HOT доступна в представлениях `pg_stat_all_tables` и `pg_stat_user_tables`
- **счетчик HOT обновлений собирается по каждой таблице и отражается в столбце: `n_tup_hot_upd`**
- все обновления отражаются в столбце `n_tup_upd`
- **случаи, если при обновлении не нашлось места для новой версии строки и цепочка HOT была оборвана, а новая версия была вставлена в другой блок показывает `n_tup_newpage_upd`**
- статистика по базе обнуляется вызовом функции `pg_stat_reset();`
  - после вызова функции рекомендуется выполнить ANALYZE по всей базе

```
 select relname, n_tup_upd, n_tup_hot_upd, n_tup_newpage_upd,
round(n_tup_hot_upd*100/n_tup_upd,2) as hot_ratio from pg_stat_all_tables where n_tup_upd<>0
order by 5;
     relname     | n_tup_upd | n_tup_hot_upd | n_tup_newpage_upd | hot_ratio
-----------------+-----------+---------------+-------------------+-----------
  pg_rewrite     |        14 |             9 |                 5 |     64.00
  pg_proc        |        33 |            23 |                10 |     69.00
  pg_class       |     71645 |         63148 |              8351 |     88.00
```


Мониторинг HOT update

Статистика HOT доступна в двух представлениях `pg_stat_all_tables` и `pg_stat_user_tables`:

```
 select relname, n_tup_upd, n_tup_hot_upd, n_tup_newpage_upd,
round(n_tup_hot_upd*100/n_tup_upd,2) as hot_ratio
 from pg_stat_all_tables where n_tup_upd<>0 order by 5;
     relname     | n_tup_upd | n_tup_hot_upd | n_tup_newpage_upd | hot_ratio
-----------------+-----------+---------------+-------------------+-----------
  pg_rewrite     |        14 |             9 |                 5 |     64.00
  pg_proc        |        33 |            23 |                10 |     69.00
  pg_class       |     71645 |         63148 |              8351 |     88.00
  pg_attribute   |       270 |           267 |                 3 |     98.00
```

Статистика накапливается с момента последнего вызова функции `pg_stat_reset()`.

`pg_stat_reset()` обнуляет счетчики накопительной статистики по текущей базе, но не обнуляет счетчики уровня кластера. **Обнуление счетчиков обнуляет счетчики по которым автовакуум решает когда нужно запустить вакуумирование и анализ. После вызова функции рекомендуется выполнить ANALYZE по всей базе.** Статистики уровня кластера, накопленные в представлениях `pg_stat_*` обнуляют ("сбрасывают") вызовы функции:

```
select pg_stat_reset_shared('recovery_prefetch');
select pg_stat_reset_shared('bgwriter');
select pg_stat_reset_shared('archiver');
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('wal');
```

Начиная с 17 версии `pg_stat_reset_shared(null)`, сбрасывает все эти кэши.

Как выполнять мониторинг? Например, создали дополнительный индекс или увеличили количество секций секционированной таблицы, стоит проверить как поменялся процент HOT обновлений. n_tup_hot_upd - счетчик обновлений HOT, n_tup_upd - все обновления.

Приблизительная оценка числа мертвых строк:

```
 select relname, n_live_tup, n_dead_tup from pg_stat_all_tables where
n_dead_tup<>0 order by 3 desc;
```

Внутристраничная очистка (HOT cleanup)

- чтобы HOT cleanup выполнился, предыдущий UPDATE строки должен превысить границу `min(90%, FILLFACTOR)`
- если размер строк в таблице такой, что в блок помещается меньше 9 строк, то восьмая строка не превысит границу 90%, а девятая строка будет больше 11% размера блока и она не поместится в блок
- наиболее эффективно при проектировании схем хранения данных делать размер строк небольшими. Разумный размер строки не больше 600 байт

Внутристраничная очистка (HOT cleanup)

Внутристраничная очистка (HOT cleanup) важна и, во многих случаях, активно работает. Если условия HOT update соблюдены, то, при обновлении строк в блоке, новая версия ищет место в блоке и создается цепочка версий в этом блоке. Если вставляемая новая версия строки поместится в блок и, при этом, процент заполнения превысит границу `min(90%, FILLFACTOR)`, то в заголовке блока будет выставлен признак, что блок можно очистить. Следующее обновление строки блока выполнит HOT cleanup - очистит блок от строк в цепочке версий, вышедших за горизонт базы и новая версия строки скорее всего поместится в блок.

Но если процент заполнения не превысил границу `min(90%, FILLFACTOR)`, а новая версия не помещается в оставшееся место в блоке, то HOT cleanup не выполняется, версия строки вставляется в другой блок, цепочка HOT прерывается, в заголовок блока вставляется флаг, что в блоке нет места. Такое будет происходить, если в блоке меньше 9 строк и `FILLFACTOR=100%` (значение по умолчанию). В таком случае, возможно, стоит установить `FILLFACTOR` в значение, при котором новая версия строки помещалась в блоке и при этом переходила границу `FILLFACTOR`. Не стоит проектировать таблицы, чтобы размер строк был настолько большим, что в блок помещалось меньше 6 строк.

```
create table t(s text storage plain) with (autovacuum_enabled=off);
insert into t values (repeat('a',2010));
update t set s=(repeat('c',2010)) where ctid::text = '(0,1)';
update t set s=(repeat('c',2010)) where ctid::text = '(0,2)';
update t set s=(repeat('c',2010)) where ctid::text = '(0,3)';
select ctid,* from heap_page('t',0);
```

```
 ctid  | lp_off | ctid  | state  | xmin  | xmax  | hhu | hot | t_ctid | multi
-------+--------+-------+--------+-------+-------+-----+-----+--------+-------
 (0,1) | 6136   | (0,1) | normal | 1001c | 1002c | t   |     | (0,2)  | f
 (0,2) | 4096   | (0,2) | normal | 1002c | 1003c | t   | t   | (0,3)  | f
 (0,3) | 2056   | (0,3) | normal | 1003c | 1004  |     | t   | (1,1)  | f
(3 rows)
```

```
select ctid from t;
 ctid
-------
 (1,1)
```

Четвертая версия строки была вставлена во второй блок.

График на слайде будет рассмотрен в практике к этой главе.

Внутристраничная очистка в таблицах

- выполняя SELECT и UPDATE, серверный процесс может удалить dead tuples (версии строк, вышедшие за горизонт видимости базы данных, xmin horizon), выполнив реорганизацию версий строк внутри блока
- внутристраничная очистка совместима с HOT и может предварительно освобождать место, которое будет использоваться новыми версиями строк, появляющимися в результате UPDATE
- выполняется, если:
  - блок заполнен более чем на 90% или FILLFACTOR (по умолчанию 100%)
  - ранее выполнявшийся UPDATE не смог разместить новую версию строки в этом блоке
- приблизительная оценка по версиям строк, которые могут быть очищены: `pg_stat_all_tables.n_dead_tup`

Внутристраничная очистка в таблицах

Серверный процесс выполняя SELECT и другие команды может удалить dead tuples (версии строк, вышедшие за горизонт видимости базы данных, xmin horizon), выполнив реорганизацию версий строк внутри блока. Это называется внутристраничной очисткой.

HOT cleanup/pruning выполняется, если выполняется одно из условий:

блок заполнен более чем на 90% или FILLFACTOR (по умолчанию 100%).

Ранее выполнявшийся UPDATE не нашел места (то есть установил в заголовке блока подсказку `PD_PAGE_FULL`).

Внутристраничная очистка работает в пределах одной табличной страницы, не очищает индексные страницы (в индексных страницах есть аналогичный алгоритм), не обновляет карту свободного пространства и карту видимости.

Внутристраничная очистка не является основным способом очистки и была создана для того, чтобы хоть как-то очищать страницы в случае, если автовакуум не справлялся или не мог работать (In fact, page pruning was designed specifically for cases where autovacuum wasn't running or couldn't keep up).

Указатели (4 байта) в заголовке блока не освобождаются, они обновляются чтобы указывать на актуальную версию строки. Указатели освободить нельзя, так как на них могут существовать ссылки из индексов, это проверить серверный процесс не может. Только вакуум сможет освободить указатели (сделать указатели unused), чтобы указатель мог снова использоваться. В области данных версии место dead tuples очищается и остальные строки сдвигаются.

Внутристраничная очистка в индексах

- выполняется при индексном сканировании
- если строка в таблице удалена, индексная запись на строку или цепочку версий может быть помечена флагом LP_DEAD
- пометка может ставиться командой SELECT
- журнальная запись не создается, но блок грязнится
- помеченная индексная запись игнорируется на мастере, но не на реплике
- помеченная индексная запись будет очищена при выполнении команд, которые меняют данные в таблице

```
create table t (id int primary key, c text) with (autovacuum_enabled = off);
insert into t SELECT i, 'simple delete ' || i from generate_series(1, 1000000) as i;
delete from t where id between 100 and 900000;
analyze t;
explain (analyze, buffers, costs off) select * from t where id between 1 and 900000;
 Index Scan using t_pkey on t (actual time=0.010..218.477 rows=99 loops=1)
   Buffers: shared hit=11489
 Execution Time: 218.600 ms
```


Внутристраничная очистка индексах

Если при индексном сканировании (Index Scan) серверный процесс обнаружит, что строка (или цепочка строк, на которую ссылается индексная запись) удалена и вышла за горизонт базы, то в индексной записи листового блока (leaf page) в lp_flags устанавливается бит-подсказка LP_DEAD (называют **known dead**, **killed tuple**). Бит можно посмотреть в столбце `dead`, **выдаваемый функцией `bt_page_items('t_idx',блок)`**. При Bitmap Index Scan и Seq Scan не устанавливается. Помеченная таким флагом строка будет удалена позже при выполнении команды, которая вносит изменения в блок индекса. Почему место в индексе не освобождается сразу? Индексное сканирование выполняется SELECT, который устанавливает разделяемые блокировки на объект и страницы. Биты-подсказки как в индексных блоках (flags), так и в блоках таблиц (infomask и infomask2) могут меняться с такими блокировками. Для остальных изменений в блоке нужна эксклюзивная блокировка на блок и другая блокировка на сам объект. Устанавливать их SELECT не будет. Из-за этого пометка записи и освобождение места разнесены во времени.

Возврат в блок и установка в нем флага добавляет накладные расходы и увеличивает время выполнения команды, но делается однократно. Зато последующие команды смогут игнорировать индексную запись и не будут обращаться к блоку таблицы.

На репликах в блок не могут вноситься никакие изменения и SELECT на репликах биты-подсказки не устанавливает. Более того на репликах игнорируется LP_DEAD ("ignore_killed_tuples"), установленный на мастере. Изменение бита LP_DEAD не журналируется, но блок грязнится и передаётся по full_page_writes. Из-за этой особенности **запросы на реплике могут выполняться на порядок медленнее, чем на мастере**. После отработки автовакуума на мастере и применении журнальных записей, сгенерированных автовакуумом на реплике, разницы в скорости не будет.

Пример SELECT с установкой битов на 899900 удаленных строк в 7308 блоках таблицы:

`Buffers: shared hit=11489` читаются блоки индекса и таблицы

`Execution Time: 218.600 ms`

Тот же SELECT повторно на еще не очищенных блоках:

`Buffers: shared hit=2463` читались блоки индекса и несколько блоков таблицы

`Execution Time: 8.607 ms`

После REINDEX или вакуумирования таблицы (результат примерно одинаков):

`Buffers: shared hit=6` читались несколько блоков индекса и таблицы

`Execution Time: 0.373 ms`

Эволюция индексов: создание, удаление, перестройка

- команды `create/drop/reindex index имя_индекса` устанавливают блокировку SHARE, не совместимую с внесением изменений в строки таблицы
- эти команды могут выполняться одновременно, они совместимы сами с собой, при этом с concurrently не совместимы
- автовакуум не совместим ни с `concurrently`, ни без
- для временных индексов на временные таблицы не надо использовать concurrently, так как блокировок на временные объекты нет
- `create/reindex concurrently` сканирует больше блоков таблицы, чем без `concurrently`
- `concurrently` позволяет выполняться командам `SELECT, WITH, INSERT, UPDATE, DELETE, MERGE` и позволяет использовать быстрый путь (fastpath) блокирования объектов (таблиц, индексов, секций)

Эволюция индексов: создание, удаление, перестройка

Создание, удаление, перестройка индекса без указания CONCURRENTLY:

```
create index название..;
drop index имя_индекса;
reindex index имя_индекса;
```

устанавливают блокировку SHARE, не совместимую с внесением изменений в строки таблицы. Блокировка SHARE позволяет работать только командам:

1) SELECT и любому запросу, который только читает таблицу (то есть устанавливает блокировку ACCESS SHARE)

2) SELECT FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE, FOR KEY SHARE (устанавливают блокировку ROW SHARE)

3) CREATE/DROP/REINDEX INDEX (**без** CONCURRENTLY). **Можно одновременно создавать, удалять, перестраивать несколько индексов на одной таблице**, так как блокировка SHARE совместима с самой собой. CONCURRENTLY не совместим с SHARE.

"Не совместим" означает, что либо команда будет ждать, либо сразу выдаст ошибку, либо выдаст ошибку после таймаута, заданного параметром `lock_timeout`.

Для временных индексов на временные таблицы не надо использовать CONCURRENTLY, так как блокировок на временные объекты нет, к ним имеет доступ только один процесс, даже параллельные процессы не имеют доступа.

`create index` **`concurrently`** `название..;` устанавливает блокировку SHARE UPDATE EXCLUSIVE, которая позволяет выполняться командам SELECT, WITH, INSERT, UPDATE, DELETE, MERGE и позволяет использовать быстрый путь блокирования объектов процессами (fastpath).

Блокировку SHARE UPDATE EXCLUSIVE устанавливают также команды DROP INDEX CONCURRENTLY, REINDEX CONCURRENTLY, а также VACUUM (без FULL), ANALYZE, CREATE STATISTICS, COMMENT ON, некоторые виды ALTER INDEX и ALTER TABLE, автовакуум и автоанализ. Эти команды не могут одновременно работать с одной таблицей. Автовакуум попускает таблицы, если не может немедленно получить блокировку. **Автовакуум несовместим с созданием, удалением, пересозданием индексов.**

CONCURRENTLY имеет существенный недостаток. Без CONCURRENTLY, таблица сканируется один раз, с CONCURRENTLY таблица сканируется два раза и используются три транзакции.

Частичные (partial) индексы

- создаются по части строк таблицы
- предикат WHERE указывается при создании индекса и определяет индексируемые строки
- полезны тем, что позволяют избежать индексирования наиболее часто встречающихся значений
- частичный индекс может быть уникальным
- размер частичного индекса обычно меньше
- пример создания частичного индекса:

```
create unique index t1_idx1 ON t1 (c2 desc nulls first, upper(c1))
include (c3,c4) WHERE c2>0;
```


Частичные (partial) индексы

Частичные (partial) индексы создаются по части строк таблицы. Часть строк определяется предикатом WHERE, который указывается при создании индекса и делает индекс частичным.

Размер индекса может быть существенно уменьшен и вакуумирование будет проходить быстрее, так как вакуумирование сканирует все блоки индекса. Можно создавать частичные (partial) индексы. Это полезно, если приложение не работает с непроиндексированными строками. При создании индекса можно указать условие WHERE. Размер индекса может быть существенно уменьшен и вакуумирование будет проходить быстрее, так как вакуумирование сканирует все блоки индекса.

Частичные индексы полезны тем, что позволяют избежать индексирования наиболее часто встречающихся значений. Наиболее часто встречающееся значение - это значение, которое содержится в значительном проценте всех строк таблицы. При поиске наиболее часто встречающихся значений индекс всё равно не будет использоваться, так как более эффективным будет сканирование всех строк таблицы. Индексировать строки с наиболее часто встречающимися значениями нет смысла. Исключив такие строки из индекса, можно уменьшить размер индекса, что ускорит вакуумирование таблицы. Также ускоряется внесение изменений в строки таблицы, если индекс не затрагивается.

Вторая причина, по которой используется частичный индекс это когда отсутствуют обращения к части строк таблицы, а если обращения и присутствуют, то используется не индексный доступ, а полное сканирование таблицы.

Частичный индекс может быть уникальным.

Создавать большое число частичных индексов, которые индексируют разные строки не стоит. Чем больше индексов на таблице, тем ниже производительность команд, изменяющих данные; автовакуума; вероятность использования быстрого пути блокировок уменьшается.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/indexes-partial.html

Команда REINDEX

- перестраивает индексы
- REINDEX блокирует практически любые запросы, кроме подготовленных, план которых был закэширован и которые не используют перестраиваемый индекс
- Для перестройки одного индекса:
- REINDEX INDEX имя_индекса;
- Если необходимо перестроить все индексы по таблице:
- REINDEX TABLE имя_таблицы;
- Также можно перестроить индексы в рамках конкретной схемы или даже всей базы данных:
- REINDEX SCHEMA имя_схемы;
- REINDEX DATABASE; перестроить можно инднексы на таблицы только текущей базы данных, кроме индексов таблиц системного каталога
- REINDEX SYSTEM; перестройка индексов на таблицы системного каталога

Команда REINDEX

Команда REINDEX перестраивает индексы. REINDEX похож на удаление и повторное создание индекса, поскольку содержимое индекса перестраивается с нуля. Однако, блокировка учитывается по-другому. REINDEX блокирует записи, но не чтение родительской таблицы индекса. Он также берет эксклюзивную блокировку ACCESS EXCLUSIVE на обрабатываемый индекс, что блокирует чтение, пытающееся использовать этот индекс. В частности, планировщик запросов пытается взять блокировку ACCESS SHARE на каждый индекс таблицы, независимо от запроса, поэтому REINDEX блокирует практически любые запросы, кроме некоторых подготовленных запросов, план которых был кэширован и которые не используют этот самый индекс.

Для перестройки одного индекса:

```
REINDEX INDEX имя_индекса;
```

Если необходимо перестроить все индексы по таблице:

```
REINDEX TABLE имя_таблицы;
```

Также можно перестроить индексы в рамках конкретной схемы или даже всей базы данных:

```
REINDEX SCHEMA имя_схемы;
```

`REINDEX DATABASE;` перестроить можно индексы на таблицы только текущей базы данных, кроме (начиная с 16 версии) индексов на таблицы системного каталога

`REINDEX SYSTEM;` перестройка индексов на таблицы системного каталога

При перестройке можно переместить индексы в другое табличное пространство, для этого достаточно указать опцию:

```
REINDEX (TABLESPACE имя) ..;
```

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-reindex.html

REINDEX CONCURRENTLY

- неблокирующее перестроение индексов
- для каждого перестраиваемого индекса выполняется первый проход, на котором строится индекс
- выполняется второй проход, на котором в индекс вносятся записи, добавленные в таблицу во время первого прохода
- ограничения целостности, использовавшие перестраиваемые индексы, переключаются на определение нового индекса и меняются имена индексов
- удаляются старые структуры индексов
- снимаются блокировки
- Перестройка может завершиться ошибкой, в этом случае REINDEX CONCURRENTLY прерывается, но оставляет после себя нерабочий новый индекс в дополнение к перестраиваемому. Этот индекс будет игнорироваться запросами, но будет обновляться при изменении данных, что повлечёт накладные расходы

REINDEX CONCURRENTLY

Перестраивает индекс, с блокировкой SHARE UPDATE EXCLUSIVE на индекс, которая совместима с командами, меняющими строки в таблице. Команда выполняется так:

1) В **pg_index** добавляется определение индекса, которое затем заменит перестраиваемый индекс. Для предотвращения каких-либо изменений в схеме во время операции обрабатываемые индексы, а также связанные с ними таблицы защищаются блокировкой SHARE UPDATE EXCLUSIVE на уровне сессии.

2) Для каждого перестраиваемого индекса выполняется первый проход, на котором строится индекс. Когда индекс построен, его флаг `pg_index.indisready` переходит в состояние `true`, чтобы этот индекс был готов к добавлениям, и таким образом он становится видимым для других транзакций, начавшихся после перестройки индексов. Это действие выполняется в отдельной транзакции для каждого индекса. Транзакции, начавшиеся до завершения перестройки индексов, новые индексы не видят и не используют.

3) Выполняется второй проход, на котором в индекс вносятся записи, добавленные в таблицу во время первого прохода. Это действие также выполняется в отдельной транзакции для каждого индекса.

4) Ограничения целостности, использовавшие перестраиваемые индексы, переключаются на определение нового индекса и меняются имена индексов. В этот момент флаг `pg_index.indisvalid` нового индекса принимает значение true, а старого - false и производится сброс кешей системного каталога, и все сессии, обращавшиеся к старому индексу, станут работать с новой структурой индекса. Флаг `pg_index.indisread` старого индекса сбрасывается в false во избежание добавления в него новых записей, как только завершатся текущие запросы, которые могли обращаться к этому индексу.

5) Удаляются старые структуры индексов. Блокировки SHARE UPDATE EXCLUSIVE уровня сеанса, установленные для индексов и таблиц, снимаются.

Перестройка может завершиться ошибкой, в этом случае REINDEX CONCURRENTLY прерывается, но оставляет после себя нерабочий новый индекс в дополнение к перестраиваемому. Этот индекс будет игнорироваться запросами, но будет обновляться при изменении данных, что увеличит накладные расходы. Команда psql `\d` обозначает такие индексы как INVALID (нерабочие).

Гипотетические индексы (расширение HypoPG)

- устанавливается командой `CREATE EXTENSION hypopg;`
- используется при настройке выполнения команд SQL
- позволяет создать определение индексов, существующих только в текущей сессии и не влияющие на работу других сессий
- позволяет выяснить будет ли планировщик использовать индекс при выполнении конкретных команд без создания индекса
- расширение позволяет скрыть любые существующие индексы в текущей сессии чтобы они не влияли на планировщик
- гипотетические индексы создаются функцией `hypopg_create_index('CREATE INDEX...')`, которой передаётся текст команды создания индекса

Гипотетические индексы (расширение HypoPG)

В процессе настройки выполнения запросов, может возникнуть вопрос: если создать индекс с желаемыми параметрами, будет ли этот индекс использоваться планировщиком для выполнения запросов, которые оптимизируются? Создавать реальный индекс не хочется, потому что он может повлиять на работу сессий приложения - замедлить команды, изменяющие данные; создание индекса занимает много времени. Расширение позволяет создать определение индексов, существующих только в текущей сессии и не влияющих на работу других сессий. Это определение (гипотетический индекс) принимается во внимание при создании плана выполнения в той сессии, где он создан как существующий. При выполнении команды и при EXPLAIN (analyze) такой индекс не используется. Также в текущей сессии можно скрыть от планировщика любые индексы, в том числе существующие, и посмотреть, как это повлияет на формируемые планы выполнения команд.

Расширение имеет два представления, в которых можно посмотреть, какие индексы скрыты в текущей сессии и какие гипотетические индексы есть: `hypopg_hidden_indexes`, `hypopg_list_indexes`.

Работа с индексами осуществляется с помощью одиннадцати функций, входящих в расширение. Гипотетические индексы создаются функцией `hypopg_create_index('CREATE INDEX...')`, которой передаётся текст команды создания индекса. Скрытие от планировщика в текущей сессии любого, в том числе обычного индекса, выполняется вызовом функции: `hypopg_hide_index('имя_индекса'::regclass);`

План выполнения просматривается командой EXPLAIN.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/hypopg.html

Счетчик транзакций

- Для 32-битного счетчика транзакций (xid) максимальное значение равно 4млрд.
- когда этот предел достигается, счетчик транзакций переходит "через ноль" и нумерация транзакций начинается со значения 3
- Чтобы не происходило переполнение счетчика версии строк "замораживаются" (freeze), что означает, что версия строки единственная, актуальная, видна во всех моментальных снимках

Счетчик транзакций

Счетчики транзакций (xid) и мультитранзакций (mxid) используются для отслеживания порядка транзакций и определения того, какие версии строк могут быть видны для каждой транзакции. В PostgreSQL счетчик транзакций реализован как 32-битное значение. Чтобы не происходило переполнение счетчика версии строк "замораживаются" (freeze), что означает, что версия строки единственная, актуальная, видна во всех моментальных снимках.

Для 32-битного счетчика транзакций (XID) в PostgreSQL максимальное значение равно 4млрд. Когда этот предел достигается, счетчик транзакций переходит "через ноль" и нумерация транзакций начинается со значения 3. Значения 0,1,2 для обычных транзакций не используются. xid=2 является признаком замороженной строки. xid=0 в поле xmax означает, что версия строки не удалялась.

Номера самых старых, незамороженных транзакций сохраняются в pg_database в столбцах datfrozenxid и datminmxid. Если текущий номер транзакций будет отстоять от значений чуть меньше, чем на 2млрд., новые номера транзакций серверным процессам перестанут выдаваться. Обновить значения можно будет вакуумированием с заморозкой таблиц. Значения определяются одной из таблиц, по которой дольше всего не было заморозки. Вакуумировав эту таблицу, значения установятся на следующую самую старую таблицу, по которой давно не было заморозки.

В Tantor Postgres SE и SE 1C используется 64-битный счетчик транзакций. На 64-битных счетчиках транзакций проблем с переполнением счетчика нет. Единственно, если запрос или транзакция работают долго, а за это время пройдёт 2млрд. транзакций, то такие транзакции и запросы нужно прервать.

https://habr.com/ru/companies/tantor/articles/937992/

Практика

1. Обычная очистка таблицы
2. Анализ таблицы
3. Перестройка индекса
4. Полная очистка
5. Расширение HypoPG

Практика

Обычная очистка таблицы
Анализ таблицы
Перестройка индекса
Полная очистка
Расширение HypoPG

2

2d Архитектура PostgreSQL

Выполнение запросов

SQL - декларативный язык

- SQL - декларативный язык программирования для работы с базами данных, где пользователь описывает желаемые результаты запроса, не указывая конкретные шаги выполнения
- Когда пользователь отправляет запрос SQL системе управления базами данных (СУБД) PostgreSQL, происходит следующее:
  - Парсинг (синтаксический и семантический разбор) запроса
  - Переписывание
  - Планирование
  - Выполнение

SQL - декларативный язык

SQL (Structured Query Language) - это декларативный язык программирования, вам нужно описать, чего вы хотите достичь, а не указывать, как это сделать шаг за шагом. В императивных языках программирования код предоставляет последовательность команд, которые выполняются по алгоритму. Код декларативных языков задаёт что должно быть достигнуто, а не указывает способ достижения результата. Оптимизация и реализация деталей исполнения выполняется исполняемой машиной, в которой работает программа.

На языке SQL пишутся запросы, указывая, какие данные вы хотите получить или какие операции выполнить, но не указываете, каким образом система должна это сделать. Язык SQL удобен для работы с данными, позволяя коду серверного процесса оптимизировать выполнение запросов и скрывать детали хранения данных.

Серверный процесс принимает от клиента запрос СУБД и выполняет:

Парсинг (parse, разбор): анализирует запрос пользователя, проверяет его синтаксис и осуществляет семантический анализ для понимания значения запроса. Состоит из синтаксического и семантического разбора.

Трансформация (rewrite, переписывание): структура запроса преобразуется в эквивалентную, более удобную для следующих шагов

Планирование: Оптимизатор создает оптимальный план выполнения запроса, решая, какие индексы использовать, как объединять таблицы и в каком порядке выполнять операции.

Выполнение: запрос выполняется, по выбранному плану. Этот шаг включает в себя чтение строк из блоков данных, обработку строк и возврат результата.

Примечание: "запрос" - это команда (statement) типа `SELECT, INSERT, UPDATE, DELETE, MERGE, VALUES, EXECUTE, DECLARE, CREATE TABLE AS, CREATE MATERIALIZED VIEW AS`. Запрос не в смысле "запросить данные" (select, выбрать данные), а "запросить выполнение действий по обработке данных". Команды типа `create, alter, drop` не называют запросами, так как они не планируются (выполняются запрограммированным способом), меняют определения объектов (метаданные), а не данные приложений.

Синтаксический разбор

- Синтаксический разбор включает в себя следующие шаги:
  - лексический анализ
  - построение синтаксического дерева
  - проверка на соответствие грамматике

Синтаксический разбор

Синтаксический разбор - разбор входящей последовательности символов (токенов) для определения структуры слов по правилам грамматики языка. В контексте языков программирования или запросов SQL, синтаксический разбор используется для проверки того, соответствует ли введенный текст правильному синтаксису языка.

Шаги:

1) Лексический анализ (токенизация): Входящая строка разбивается на набор токенов, представляющих минимальные синтаксические единицы, такие как ключевые слова, операторы, идентификаторы и числа.

Постфиксные операторы в PostgreSQL отсутствуют с 14 версии, что упрощает анализ.

2) Построение синтаксического дерева: Токены объединяются в структуру данных, называемую синтаксическим деревом, которая отражает иерархию и структуру языка. Это дерево представляет собой абстрактное синтаксическое представление введенного выражения.

3) Проверка на соответствие грамматике: Синтаксический анализатор проверяет, соответствует ли построенное синтаксическое дерево правилам грамматики языка. Если нет - генерируется ошибка, указывающая на некорректность синтаксиса.

В случае SQL-запроса проверяет, что запрос соответствует синтаксическим правилам SQL, что позволяет представить (интерпретировать) и выполнить запрос.

Семантический разбор

- определение смысла (семантики), проверка существования таблиц, столбцов, согласованность типов данных
- проверка прав доступа: имеет ли пользователь право на выполнение команды, права доступа к объектам, указанным в запросе: схемам, таблицам, функциям, представлениям и т.п.
  - идёт обращение к **таблицам системного каталога**, в которых хранятся определения объектов. Например, `pg_class, pg_attribute, pg_type, pg_depend, pg_constraint, pg_namespace, pg_inherits, pg_attrdef, pg_sequence`
  - Выбранные данные кэшируются в локальной памяти процесса, обслуживающего сессию пользователя

Семантический разбор

Определение смысла (семантики): Этот этап парсинга SQL включает анализ смысла запроса, проверку существования таблиц, столбцов, согласованность типов данных.

Проверка прав доступа: имеет ли пользователь право на выполнение команды, права доступа к объектам, указанным в запросе: схемам, таблицам, функциям, представлениям и т.п.

На этом шаге идёт обращение к таблицам системного каталога, в которых хранятся определения объектов. Например, `pg_class, pg_attribute, pg_type, pg_depend, pg_constraint, pg_namespace, pg_inherits, pg_attrdef, pg_sequence`. Выбранные данные кэшируются в локальной памяти процесса в структуре памяти (называются "контекстами") `CacheMemoryContext`, обслуживающего сессию пользователя. В будущем, если в строки таблиц системного каталога будут вноситься изменения, процесс, который вносит изменения передает изменения в кольцевой (новые сообщения по кругу затирают старые) буфер (`shmInvalBuffer`) в разделяемой памяти размером в 4096 сообщений:

```
postgres=# select name, size from (select name, lead(off) over(order by off) -
off as size from pg_shmem_allocations) as a where name='shmInvalBuffer';
       name       |  size
------------------+--------
 shmInvalBuffer | 291072
(1 row)
```

Если процесс не потребил половину сообщений, то ему передаётся уведомление, чтобы он потребил накопившиеся сообщения. Это позволяет уменьшить вероятность того, что процесс пропустит сообщения и ему придется очистить свой локальный кэш системного каталога. В разделяемой памяти сохраняются данные о том, какие процессы какие сообщения потребили. Если какой-нибудь процесс, несмотря на уведомление, не потребляет сообщения (например, выполняет операцию и не может прерваться), а буфер заполнен, то процессу придётся полностью очистить свой кэш системного каталога.

Устанавливаются блокировки на все объекты, которые используются в запросе и могут использоваться для создания плана: таблицы, индексы, секции таблиц. Блокировки нужны, чтобы пока запрос планируется или выполняется используемые объекты не удалялись и не менялась их структура, что привело бы к ошибке при создании плана или выполнения запроса.

Трансформация (переписывание) запроса

- преобразование исходной структуры запроса в аналогичную с точки зрения получения результата в целях лучшей оптимизации на этапах планирования и выполнения в целях лучшей оптимизации на этапах планирования и выполнения
  - имена представлений, если такие были в запросе, заменяются запросами, на основе которых созданы представления

```
LOG:  rewritten parse tree:
DETAIL:  (
            {QUERY
            :commandType 1
            :querySource 0
            :canSetTag true
            :utilityStmt <>
            :resultRelation 0
            :hasAggs false
            :hasWindowFuncs false
...
```


Трансформация (переписывание) запроса

Трансформация (переписывание, rewrite) запроса - это преобразование исходной структуры запроса в аналогичную с точки зрения получения результата в целях лучшей оптимизации на этапах планирования и выполнения.

Например, имена представлений, если такие были в запросе, заменяются запросами, на основе которых созданы представления.

Параметр конфигурации `debug_print_rewritten` позволяет увидеть результат переписывания в диагностическом журнале. Пример:

```
postgres@tantor:~$ cat $PGDATA/log/postgresql-*
STATEMENT:  select * from t limit 1;
LOG:  rewritten parse tree:
DETAIL:  (
         {QUERY
         :commandType 1
         :querySource 0
         :canSetTag true
         :utilityStmt <>
         :resultRelation 0
         :hasAggs false
         :hasWindowFuncs false
         :hasTargetSRFs false
         :hasSubLinks false
         :hasDistinctOn false
         :hasRecursive false
         :hasModifyingCTE false
         :hasForUpdate false
         :hasRowSecurity false
         :isReturn false
         :cteList <>
         :rtable (
            {RANGETBLENTRY
            :alias <>
            :eref
               {ALIAS
               :aliasname now
               :colnames ("now")
               }
            :rtekind 3
...
```

Планирование выполнения (оптимизация)

- Цель этого шага - получить наилучший способ (план) выполнения запроса
- Генерируются возможные способы выполнения запроса, оценивается трудоемкость выполнения и выбирается способ (план) выполнения с наименьшей стоимостью
- Для оценки стоимости используется
- статистика, описывающая объекты
- весовые коэффициенты из параметров конфигурации
- параметры конфигурации, разрешающие использовать способы выборки и обработки строк
- В расчет стоимости включаются две части: вычислительная сложность (процессор) и ввод-вывод

Планирование выполнения (оптимизация) запроса

Это процесс получения наилучшего способа выполнения запроса.

Планировщик (оптимизатор) - это код (написанный на языке С) серверного процесса, который выполняет запрос. Логика кода алгоритмическая. Генерируются возможные способы выполнения запроса, оценивается трудоемкость выполнения и выбирается способ (план) выполнения с наименьшей стоимостью. Для оценки стоимости используется статистика, описывающая объекты. Например, число строк и блоков в таблицах, индексах, число уникальных значений в столбцах, число нескольких наиболее часто встречающихся значений и т.п. В коде оптимизатора есть весовые коэффициенты для расчета стоимости. Часть коэффициентов указано в параметрах конфигурации, чтобы их можно было настроить. Например, `seq_page_cost`, `random_page_cost`, `parallel_setup_cost`, `parallel_tuple_cost`, `cpu_tuple_cost`, `cpu_index_tuple_cost`, `cpu_operator_cost`. Также принимаются во внимание параметры конфигурации, которыми можно склонить оптимизатор к выбору способов выборки и обработки данных. Названия большинства этих параметров начинаются на `enable`. В PostgreSQL 18 версии таких параметров 24, в Tantor Postges SE и SE 1C их 37.

Примееры параметров: `enable_seqscan` (возможность сканирования всех блоков таблиц для выборки строк из них); `enable_nestloop` (возможность соединять наборы строк методом вложенных циклов).

В расчет стоимости включаются две части: вычислительная сложность (процессор) и ввод-вывод.

План выполнения запроса можно посмотреть командой **`explain`**:

```
postgres=# explain select 1;
                QUERY PLAN
-----------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=4)
(1 row)
```

Блокировки на объекты, которые не использовались в созданном плане, могут сниматься.

Выполнение запроса

- Чтение данных: читаются строки из блоков таблиц, индексов, функций
- Обработка данных: фильтрации, сортировки, группировки, вычисления
- Соединение наборов строк: если запрос включает в себя соединение таблиц или других источников данных
- Группировка строк: например, если используются групповые функции типа COUNT, SUM, AVG
- Возврат результата: возврат строк клиенту или в код, пославший запрос на выполнение
- Освобождение ресурсов: снимаются блокировки с объектов и освобождается память, использованная при выполнении запроса

Выполнение запроса

Выполнение - последний шаг в обработке запроса. На этомшаге выполняются действия, в соответствии с планом выполнения. Типичные этапы выполнения:

Чтение данных: читаются строки из блоков таблиц, индексов, функций.

Обработка данных: фильтрации, сортировки, группировки, вычисления.

Соединение наборов строк: если запрос включает в себя соединение таблиц или других источников данных.

Группировка строк: например, если используются групповые функции типа COUNT, SUM, AVG, выражение GROUP BY.

Возврат результата: процесс возврата строк клиенту или в код, пославший запрос на выполнение.

Освобождение ресурсов: процесс, выполнявший запрос, освобождает использованные ресурсы: снимает блокировки с объектов и освобождает (номинально для повторного использования или с возвратом в операционную систему) память, использованную при выполнении запроса, удаляет временные файлы, если они использовались.

Команда EXPLAIN

- выдаёт план выполнения запроса
  - по умолчанию запрос не выполняется.
- Если указать опцию analyze, то запрос выполнится, после выполнения будет показан план с деталями выполнения.
  - без опции timing off "Execution Time" может выдать время выполнения, превышающее реальное в несколько раз
  - опция buffers показывает число буферов, которые считывались при планировании и выполнении

```
postgres=# explain (analyze, buffers) select * from t limit 1;
           QUERY PLAN
----------------------------------------
Limit  (cost=0.03..0.04 rows=1 width=8) (actual time=0.048..0.067 rows=1 loops=1)
 Buffers: shared hit=2
 -> Seq Scan on t  (cost=0.00..14425.00 rows=1000000 width=8) (actual time=0.015..0.020 rows=1 loops=1)
    Buffers: shared hit=2
Planning Time: 0.040 ms
Execution Time: 0.198 ms
(6 rows)
```


Команда EXPLAIN

Команда EXPLAIN показывает план выполнения запроса, который выбран как оптимальный. По умолчанию запрос не выполняется.

Если указать опцию analyze, то запрос выполнится, хоть строки и не будут выданы, и только после выполнения запроса будет выдан план с дополнительными деталями. При использовании analyze в строках плана появятся реальные данные после "(actual ". Если время выполнения строки плана "actual time" не нужно, то можно указать опцию "timing off", это позволит получить реальные данные в строке "Execution Time", так как обращения к счетчику могут быть частыми, а на обращения тоже тратится время.

Полезно использование опции **buffers** - будет показано число буферов, которые считывались. Показатель buffers долгое время недооценивался с точки зрения оптимизации, но его важность осознали к 18 версии PostgreSQL, в которой параметр buffers включён по умолчанию.

Пример использования команды EXPLAIN:

```
postgres=# explain (analyze, buffers) select * from t limit 1;
           QUERY PLAN
----------------------------------------
 Limit  (cost=0.03..0.04 rows=1 width=8) (actual time=0.048..0.067 rows=1
loops=1)
  Buffers: shared hit=2
  -> Seq Scan on t (cost=0.00..14425.00 rows=1000000 width=8)(actual
time=0.015..0.020 rows=1 loops=1)
     Buffers: shared hit=2
 Planning Time: 0.040 ms
 Execution Time: 0.198 ms
```

План запроса позволяет оценить, какими способами обрабатываются данные, не было ли ошибок в предсказании числа строк (отличия планируемого числа `rows` от `actual rows` - реально считанных). Это называют ошибками в расчете "кардинальности" (синоним "мощность" или даже число строк, но эти термины менее распространены) и "селективности" (доля строк) - эти термины пришли в SQL из реляционной теории.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-explain.html

Параметры команды EXPLAIN

- 12 параметров, указываются в круглых скобках

```
postgres=# EXPLAIN (analyze, verbose, buffers, serialize text, settings, memory, wal) SELECT *
FROM t WHERE i = 100 AND j = 10;
                        QUERY PLAN
-----------------------------------------------------
Gather  (cost=1000.00..11676.00 rows=10 width=8) (actual time=0.421..65.852 rows=10 loops=1)
 Output: i, j
 Workers Planned: 2
 Workers Launched: 2
 Buffers: shared hit=4425
 ->Parallel Seq Scan on public.t(cost=0.00..10675.00 rows=4 width=8)(actual
time=37.521..57.814 rows=3 loops=3)
         Output: i, j
         Filter: ((t.i = 100) AND (t.j = 10))
         Rows Removed by Filter: 333330
         Buffers: shared hit=4425
         Worker 0:  actual time=55.948..55.955 rows=0 loops=1
           Buffers: shared hit=1396
         Worker 1:  actual time=56.416..56.422 rows=0 loops=1
           Buffers: shared hit=1399
Query Identifier: 687797574221341570
Planning:
  Memory: used=11kB  allocated=16kB
Planning Time: 0.054 ms
Serialization: time=0.042 ms  output=1kB  format=text
Execution Time: 66.150 ms
```


Параметры команды EXPLAIN

ANALYZE (по умолчанию false) выполнить запрос, но не передавать клиенту результат. Позволяет оценить реальное число строк, время выполнения и использовать расширения и оптимизации, как при выполнении запроса. ANALYZE для команд INSERT, UPDATE, DELETE выполняет изменения данных.

VERBOSE (false) выводит дополнительные данные в плане. Например, имена схем, псевдонимы таблиц, имена переменных привязки, идентификатор запроса (Query Identifier), чтобы можно было найти его статистику выполнения, собираемую расширениями (`pg_stat_statements`).

COSTS (true) отображает расчетную стоимость каждого узла плана, rows и width.

SETTINGS (по умолчанию false) выдает параметры конфигурации, влияющие на планировщик, значения которых отличны от значений по умолчанию

GENERIC_PLAN (false) позволяет показать план для запроса, в котором используются переменные привязки вида $1, $2.. Показывает общий (generic) план, который будет использоваться вместо частных, если он не хуже их. Нельзя одновременно использовать с ANALYZE.

Для получения данных о выполняющихся запросах используются представления `pg_stat_activity` и расширение Tantor Postgres **`pg_trace`**. Для части команд, выполняемых вручную или автоматически, есть представления: `pg_stat_progress_analyze`, `pg_stat_progress_cluster`, `pg_stat_progress_create_index`, `pg_stat_progress_basebackup`, `pg_stat_progress_copy`, `pg_stat_progress_vacuum`

Параметры команды EXPLAIN

- 12 параметров, указываются в круглых скобках

```
postgres=# EXPLAIN (analyze, verbose, buffers, serialize text, settings, memory, wal) SELECT *
FROM t WHERE i = 100 AND j = 10;
                        QUERY PLAN
-----------------------------------------------------
Gather  (cost=1000.00..11676.00 rows=10 width=8) (actual time=0.421..65.852 rows=10 loops=1)
 Output: i, j
 Workers Planned: 2
 Workers Launched: 2
 Buffers: shared hit=4425
 ->Parallel Seq Scan on public.t(cost=0.00..10675.00 rows=4 width=8)(actual
time=37.521..57.814 rows=3 loops=3)
         Output: i, j
         Filter: ((t.i = 100) AND (t.j = 10))
         Rows Removed by Filter: 333330
         Buffers: shared hit=4425
         Worker 0:  actual time=55.948..55.955 rows=0 loops=1
           Buffers: shared hit=1396
         Worker 1:  actual time=56.416..56.422 rows=0 loops=1
           Buffers: shared hit=1399
Query Identifier: 687797574221341570
Planning:
  Memory: used=11kB  allocated=16kB
Planning Time: 0.054 ms
Serialization: time=0.042 ms  output=1kB  format=text
Execution Time: 66.150 ms
```


Параметры команды EXPLAIN (продолжение)

BUFFERS (начиная с 18 версии по умолчанию true) выдает информацию о прочитанных из кэша (hit) плюс из операционной системы (read) буферах из разделяемого кэша буферов (shared) или локального кэша для временных таблиц (local). Может выдавать dirtied - число буферов (уже входят в read или hit), которые не были грязными и их содержимое поменялось запросом. При загрузке в буфер блок является "чистым", то есть соответствующим своему образу в файле. Блок может также стать чистым, уже находясь в буфере. Процесс контрольной точки может сделать блок "чистым", записав блок на диск.

written - число грязных (в том числе загрязненных другими запросами) буферов, которые были посланы на запись (evicted) из-за того, что серверному процессу понадобилось освободить буфер для загрузки в буфер другого блока.

SERIALIZE (NONE) включает информацию о стоимости сериализации (с выделением памяти под строковый буфер) выходных данных запроса (после SELECT или RETURNING), преобразовывая данные в текстовый или двоичный формат для отправки клиенту. Актуально, если поля выбираются из TOAST, так как по умолчанию из TOAST данные командой EXPLAIN не выбираются. Команда EXPLAIN никогда не отправляет полученные данные клиенту, поэтому трудоемкость передачи по сети не считается. Работать только с ANALYZE. Значения: NONE, SERIALIZE [TEXT], SERIALIZE BINARY.

WAL (false) выводит число журнальных записей, полных образов страниц (**fpi, full page images**), объём сгенерированных записей в байтах.

TIMING (true) выводить время, затраченное на каждый узел. Может существенно увеличить общее время выполнения запроса. Используется с ANALYZE.

MEMORY (false) использованная на этапе планирования память

SUMMARY (true если используется ANALYZE) выводит Planning Time после плана запроса

FORMAT (TEXT) кроме TEXT может быть XML, JSON, YAML.

Начиная с 18 версии под узлами плана CTE Scan, Materialize, Recursive Union, Table Function Scan, WindowAgg выдаётся строка "Storage:" с типом потреблённой памяти (Memory или Disk) и пиковым значением потреблённой памяти (после "Maximum Storage:").

Индексы для ограничений целостности

- для ограничений целостности PRIMARY KEY и UNIQUE необходим **уникальный** индекс **типа btree** со столбцами, входящими в ограничение целостности.
- остальные индексы могут использоваться для ускорения запросов ("аналитические индексы"), полнотекстового поиска
- индексы ускоряют поиск строк и замедляют добавление, изменение, удаление строк
- индексы используют место на диске, размер сопоставим с размером таблицы
- пример замены индекса другим индексом:

```
create table t3 (n int4 primary key, m int4);
Indexes:
    "t3_pkey" PRIMARY KEY, btree (n)
create unique index concurrently t3_pkey1 on t3 (m,n);
ALTER TABLE t3 DROP CONSTRAINT t3_pkey, ADD CONSTRAINT t3_pkey PRIMARY KEY USING INDEX t3_pkey1;
NOTICE:  ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index "t3_pkey1" to "t3_pkey"
Indexes:
    "t3_pkey" PRIMARY KEY, btree (m, n)
```


Индексы для ограничений целостности

Если в команде CREATE INDEX не указать тип индекса, то создаётся индекс типа btree. btree наиболее распространенный тип индекса в реляционных базах данных, работающий со многими типами данных.

Для ограничений целостности PRIMARY KEY (PK) и UNIQUE (UK) необходимы индексы типа btree. Для других ограничений целостности необязательны и создаются если: ускоряют запросы, существенно не замедляют изменение данных, используемое индексами место не критично.

При создании ограничений целостности PRIMARY KEY (PK) и UNIQUE (UK) создаются уникальные индексы типа btree. Правила использования индексов с ограничениями целостности отличаются от Oracle Database.

Например, в PostgreSQL без уникального индекса ограничения PK и UK не могут существовать:

```
ERROR:  PRIMARY KEY constraints cannot be marked NOT VALID
```

и не могут использовать неуникальные индексы:

```
alter table t3 drop constraint t3_pkey, add constraint t3_pkey primary key using index t3_pkey1;
ERROR:  "t3_pkey1" is not a unique index;
```

В Oracle Database есть включённое и отключённое состояние ограничений целостности, индекс создаётся при включении ограничения целостности, можно использовать неуникальные индексы. Такие отличия не дают преимуществ или недостатков, об отличиях полезно знать при эксплуатации и поддержке таблиц, если вы имеете опыт работы с СУБД, отличных от PostgreSQL.

В PostgreSQL только индекс типа btree поддерживает свойство UNIQUE (может быть уникальным):

```
select amname, pg_indexam_has_property(a.oid, 'can_unique') as p from pg_am a where amtype = 'i' and pg_indexam_has_property(a.oid, 'can_unique') = true order by 1;
  amname | p
--------+---
  btree  | t
```

Способы доступа к данным в плане запроса

- способы, которыми процесс получает данные (строки, записи) из источников данных
- источниками могут быть таблицы, внешние таблицы, функции и т.п.
- расширения могут создавать свои методы доступа
- для способа Bitmap строится битовая карта по индексу
  - построение карты отмечается строкой Bitmap Index Scan
  - по битовой карте сканируются строки или блоки таблицы, что отмечается строкой Bitmap Heap Scan в плане:

```
Bitmap Heap Scan on tab (cost=10..1000.51 rows=998 width=11)
  Recheck Cond: (col1 < '1000'::numeric)
  -> Bitmap Index Scan on t_col1_idx (cost=0.00..9.60 rows=998 width=0)
       Index Cond: (col1 < '1000'::numeric)
```


Способы доступа к данным в плане запроса

Способов (алгоритмов) доступа к данным много: Sequential Scan, Index Scan. Index Only Scan, Bitmap Heap Scan, Bitmap Index Scan, CTE Scan, Custom Scan, Foreign Scan, Function Scan, Subquery Scan, Table Function Scan, Table Sample Scan, Tid Scan, Values Scan, Work Table Scan и другие. При распараллеливании перед названием способа в строке плана добавляется слово Parallel. Источниками данных могут быть таблицы, внешние таблицы, табличные функции и т.п. Расширения могут добавлять свои "методы" (реализация алгоритма) доступа, например, для табличного - в способ Custom Scan.

Для обычных таблиц способы делятся на табличные - Sequential и с помощью индексов - Index, Index Only, Bitmap Heap, Bitmap Index. Для способа Bitmap - строится битовая карта. Построение карты отмечается строкой Bitmap Index Scan. Затем с использованием битовой карты сканируются строки или блоки таблицы, что отмечается строкой Bitmap Heap Scan в плане:

```
Bitmap Heap Scan on tab (cost=10..1000.51 rows=998 width=11)
  Recheck Cond: (col1 < '1000'::numeric)
  -> Bitmap Index Scan on t_col1_idx (cost=0.00..9.60 rows=998 width=0)
       Index Cond: (col1 < '1000'::numeric)
```

Пример доступа к таблице с колоночным хранением:

```
Custom Scan (ColumnarScan) on public.perf_columnar (cost=0.00..138.24 rows=1
width=8)
```

Возможные типы узлов (строк) плана перечислены в файле `src/include/nodes/plannodes.h` исходного кода PostgreSQL

К временным таблицам имеет доступ только один серверный процесс, поэтому распараллеливания при сканировании временной таблицы нет.

Методы доступа к строкам

```
\dA
List of access methods
   Name    | Type
-----------+-------
 bloom     | Index
 brin      | Index
 btree     | Index
 columnar  | Table
 gin       | Index
 gist      | Index
 hash      | Index
 heap      | Table
 spgist    | Index
(9 rows)
```

- два типа "методов" (способов) доступа к строкам таблиц: табличный и индексный
- Методы доступа можно добавлять расширениями:
  - `create extension pg_columnar;`
  - `create extension bloom;`
- Табличные методы доступа определяют способ хранения данных в таблицах и обычно считывают все строки
- Индексные методы обычно считывают часть блоков таблицы
- Для индексных методов (способов) нужно создать вспомогательный объект, называемый индексом
- Индексы создаются на один или несколько столбцов таблицы:

```
create table t (id int8, d date, s text);
create index t_idx on  t using btree (id int8_ops, d date_ops);
create index t_idx1 on t using btree (s text_ops);
create index if not exists t_idx2 on t (id, d);
```


Методы доступа к строкам

Есть два типа "методов" доступа к строкам таблиц: табличный и индексный.

Список доступных методов доступа: **\dA** или запрос:

```
SELECT * FROM pg_am;
```

```
   oid   |  amname  |          amhandler          | amtype
---------+----------+-----------------------------+--------
       2 | heap     | heap_tableam_handler        | t
     403 | btree    | bthandler                   | i
     405 | hash     | hashhandler                 | i
     783 | gist     | gisthandler                 | i
    2742 | gin      | ginhandler                  | i
    4000 | spgist   | spghandler                  | i
    3580 | brin     | brinhandler                 | i
```

Методы доступа можно добавлять расширениями:

```
create extension pg_columnar;
create extension bloom;
```

Расширения добавят в таблицу `pg_am` методы доступа:

```
 2425358 | columnar | columnar.columnar_handler | t
 2425512 | bloom    | blhandler                 | i
```

Табличные методы доступа определяют способ хранения данных в таблицах. Чтобы планировщик использовал индексный метод доступа, нужно создать вспомогательный объект, называемый индексом. **"Тип индекса" и "индексный метод доступа" синонимы.**

Индексы создаются на один или несколько столбцов таблицы:

```
create table t (id int8, s text);
create index t_idx on t using btree (id int8_ops) include (s) with (fillfactor = 90, deduplicate_items = off);
```

При создании индекса указывается название таблицы и столбец или столбцы ("составной индекс"), значения которых будут индексироваться. Опция INCLUDE позволяет сохранить в структуре индекса значения столбцов, выражения нельзя использовать. Для типов данных таких столбцов не нужны классы операторов. Смысл включения столбцов: чтобы планировщик использовал Index Only Scan.

Можно создать несколько одинаковых индексов, но с разными названиями.

Названия класса операторов обычно не указывается, так как есть класс по умолчанию для типа столбца. По умолчанию используется тип индекса btree.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-createindex.html

Способы соединения наборов строк

- Способы:
  - Соединение вложенным циклом (Nested Loop Join) с мемоизацией и без
  - Соединение хешированием (Hash Join) с использованием временных файлов и без использования
  - Соединение слиянием (Merge Join)
- Наборы строк всегда соединяются попарно

Способы соединения наборов строк

Наборы строк всегда соединяются попарно, то есть соединяются два набора (выборки). В PostgreSQL существуют три способа соединения наборов строк (выборок):

Соединение вложенным циклом (Nested Loop Join): один набор строк последовательно просматривается для каждой строки из второго набора Способ оптимален для соединения наборов с небольшим числом строк. Его вычислительная сложность равна **произведению числа строк** в выборках. Занижение числа строк при оценке (ошибки в оценке кардинальности) приводит к существенному увеличению времени выполнения для этого способа соединения. Порядок таблиц при соединении этим способом не играет роли. Первая строка отдаётся без задержки. Может использоваться с условием соединения, отличным от равенства.

Есть вариация этого способа с мемоизацией - кэшированием набора, который сканируется много раз. При использовании мемоизации этот набор должен иметь меньший размер. Узел Memoize встраивается в плане между узлом поставляющим данные и Nested Loop.

Соединение хешированием (Hash Join) - возможен только для соединения по условию равенства. Сначала выбирается выборка с наименьшим размером, который определяется числом строк и размером строки выборки, которая состоит из тех столбцов, которые упоминаются в запросе (в таблице же может быть больше столбцов). По этому набору за один проход в памяти процесса строится хэш-структура (называется хэш-таблица), которая существует до завершения запроса. Потом идет просмотр второй выборки и из хэш-структуры выбираются строки, если имеется соответствие. Вычислительная сложность пропорциональна сумме строк в обеих выборках. Первая строка отдаётся только после построения хэш-таблицы, то есть после считывания первого набора строк. Если памяти для хэш-таблицы не хватает, то используются временные файлы и время на выполнение соединения увеличивается из-за добавления файловых операций.

Соединение слиянием (Merge Join): при этом способе нужно, чтобы обе выборки были отсортированы по столбцам, по которым идёт соединение. В планах запросов отсортированные строки является побочным эффектом нижестоящих узлов плана выполнения. Например, при сканировании по индексу btree строки поступают на вышестоящий узел в отсортированном (по ключевым столбцам индекса) виде. Вычислительная сложность пропорциональна сумме строк из двух выборок. Первая строка отдаётся без задержки, так как хэш-таблица не строится.

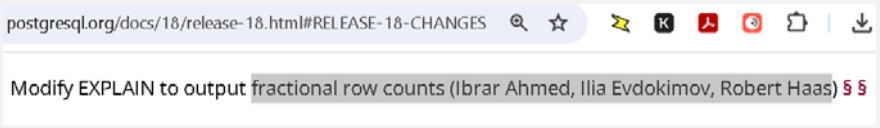
Все способы соединения могут выполняться параллельными процессами.

Кардинальность и селективность

- кардинальным числом отношения (сокращенно кардинальностью) или мощностью отношения называется число строк (они же tuples, кортежи)
  - в плане rows и actual rows
  - с 18 версии PostgreSQL rows - десятичное число

```
Gather (actual rows=2.00 loops=1)
-> Parallel Seq Scan on bookings (actual rows=0.67 loops=3)
```

- Селективность - доля (от нуля до 1) строк из выборки



Кардинальность и селективность

В реляционной теории используются термины, усложняющие понимание. Число атрибутов (столбцов) называют арностью или степенью отношения. Типы данных называют доменами или множествами допустимых значений. Соединение таблиц определяют как декартово произведение, к которому применена операция выборки (ограничения) с предикатом (условием соединения). Само декартово произведение не имеет практического смысла, но оно похоже на умножение, поэтому и было определено. Однако, когда появилась реляционная теория, были популярны сетевые базы данных, которые ещё более запутаннее. Сейчас реляционная теория и алгебра Кодда представляет исторический интерес. Некоторые менее вычурные термины до сих пор используются, такие как кардинальность и селективность. В конце концов появился язык SQL, который слабо основан на реляционной алгебре и таблицы в SQL это не совсем отношения. Например, можно создать таблицу с одинаковыми строками.

В реляционной модели данных кардинальным числом отношения (сокращенно кардинальностью) или мощностью отношения называется число строк (они же tuples, кортежи). Ближе к практике - это показатель rows в узлах плана выполнения. До 18 версии PostgreSQL это целое число. Начиная с 18 версии PostgreSQL, значение rows выдаётся в десятичном виде (два знака после точки). Авторы патча: Ибрар Ахмед, Илья Евдокимов (Тантор Лабс) и Роберт Хаас. Пример:

```
Gather (actual rows=2.00 loops=1)
-> Parallel Seq Scan on bookings (actual rows=0.67 loops=3)
```

Причина введения десятичных значений: `0.67*3=2.00`, а в предыдущих версиях `1*3=2` выглядела как расхождение.

```
Gather (actual rows=2 loops=1)
   ->  Parallel Seq Scan on bookings (actual rows=1 loops=3)
```

Селективность - доля (от нуля до 1) строк из выборки. Например, если через условие WHERE (называют предикатом - термин из реляционной теории) которое фильтрует строки, проходит 10% строк, то "селективность предиката" 0,1. Если фильтрации нет, то селективность выборки 1. Если вернется ноль строк, то селективность ноль.

Наиболее частые ошибки планировщика - неверная оценка селективности, на что в плане указывает расхождение планируемых rows и actual rows более, чем на порядок.

https://www.postgresql.org/docs/18/release-18.html#RELEASE-18-CHANGES

Стоимость плана запроса

- Стоимость (cost) - числовая оценка трудоемкости выполнения узла плана или всего запроса
- Состоит из двух чисел, между которыми две точки
  - `cost=0.00..14425.00`
- Первое число (startup cost) - стоимость отдачи первой строки в выборке
- Второе число (total cost) - стоимость отдачи всех строк
- Значение стоимости имеет смысл только для сравнения планов одного и того же запроса
- Для всех запросов, кроме курсоров, выбирается план с наименьшим вторым числом
- Стоимость одного и того же запроса коррелирует с временем выполнения этого запроса, но нелинейно

Стоимость плана запроса

Стоимость (cost) - числовая оценка трудоемкости выполнения узла плана или всего запроса. Состоит из двух чисел, между которыми две точки. Первое число (startup cost) - стоимость отдачи первой строки в выборке. Второе число (total cost) - стоимость отдачи всех строк. Для всех запросов, кроме курсоров, выбирается план с наименьшим вторым числом.

Первое число учитывается при выборе оптимального плана только для курсоров, для них выбирается план с наименьшим значением: **первое число + cursor_tuple_fraction * (второе число - первое число)**. По умолчанию, значение параметра конфигурации:

```
show cursor_tuple_fraction;
 cursor_tuple_fraction
-----------------------
 0.1
```

Значение стоимости имеет смысл только для сравнения планов одного и того же запроса. Значения для разных запросов слабо сравнимы. Стоимость одного и того же запроса коррелирует с временем выполнения этого запроса, но нелинейно. При нагрузке на ядра центральных процессоров или ввод-вывод стоимость не меняется, а время выполнения запроса увеличивается.

Пример расчета стоимости:

```
postgres=# EXPLAIN (analyze, buffers) SELECT * FROM t;
                                 QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..14425.00 rows=1000000 width=8) (actual time=0.016..3924.918
rows=1000000 loops=1)
   Buffers: shared hit=4425
 Planning Time: 0.033 ms
 Execution Time: 7797.977 ms
postgres=# select relpages, reltuples::numeric, current_setting('seq_page_cost') seq_page_cost,
current_setting('cpu_tuple_cost') cpu_tuple_cost, current_setting('seq_page_cost')::float * relpages
CPU, current_setting('cpu_tuple_cost')::float * reltuples IO,
current_setting('seq_page_cost')::float * relpages + current_setting('cpu_tuple_cost')::float *
reltuples total_cost  from pg_class c where relname = 't';
 relpages | reltuples | seq_page_cost | cpu_tuple_cost | cpu  |  io   | total_cost
----------+-----------+---------------+----------------+------+-------+------------
     4425 |   1000000 | 1             | 0.01           | 4425 | 10000 |      14425
```

В примере вклад в стоимость ввода-вывода в 10000/144,25=70%.

Примечание: JDBC-драйвер не использует команду DECLARE для создания курсора, поэтому **cursor_tuple_fraction** и первое число стоимости не используется.

https://habr.com/ru/articles/942938/

Статистики

- используется планировщиком при выборе плана
- **базовая** и **расширенная** статистика хранится в таблицах системного каталога
- сбор расширенной статистики устанавливается вручную командой `CREATE STATISTICS` или автоматически расширением pg_stat_advisor
  - после установки расширенная статистика собирается вместе с базовой автовакуумом (автоанализом) или при выполнении команд VACUUM (ANALYZE)
- **накопительная** статистика хранится в разделяемой памяти, доступна через представления pg_stat_all_* и функциями pg_stat_get*
  - при остановке экземпляра сохраняется в файл в директории PGDATA/pg_stat
  - используется автовакуумом

Статистики

Планировщик использует статистику. Статистика собирается и хранится в таблицах системного каталога для таблиц и индексов.

Базовая статистика включает в себя информацию о распределении данных, числе уникальных значений, размере таблиц и индексов, а также другие метрики.

Расширенная статистика также собирается автоматически, но нужно определить параметры командой `CREATE STATISTICS...`

Статистика не обновляется, она пересобирается автовакуумом (на фазе автоанализа) или командой `ANALYZE`.

Статистика сохраняется в таблицах системного каталога:

pg_class и pg_index: содержат информацию о размерах таблиц и индексов, а также о числе строк в таблицах.

pg_statistic: содержит статистические данные о значениях столбцов, такие как минимальные и максимальные значения, среднее, стандартное отклонение и др.

Расширенная статистика хранится в `pg_statistic_ext` и `pg_statistic_ext_data`.

Накопительная статистика доступна в представлениях pg_stat_all_*, pg_statio_*, которые выдают данные функциями pg_stat_get* из памяти экземпляра. При остановке экземпляра (кроме немедленной), накопительная статистика сохраняется в директории `PGDATA/pg_stat`.

Таблица pg_statistic

- Таблица pg_statistic хранит базовую статистику
- Собирается автоанализом и командой ANALYZE , используется для оптимизации запросов планировщиком
- По умолчанию пересобирается по default_statistics_target * 300 = 30000 строк
- содержатся данные по каждому столбцу таблиц
- Пример: доля строк с NULL в третьем столбце таблицы test:

```
select stanullfrac from pg_statistic
 where starelid = 'test'::regclass and staattnum = 3;
 stanullfrac
-------------
   0.9988884
```


Таблица pg_statistic

Таблица pg_statistic хранит базовую статистику. Собирается автоанализом и командой ANALYZE , используется для оптимизации запросов планировщиком. Статистика представляет приблизительные значения, даже если она актуальна. По умолчанию, собирается по default_statistics_target * 300 = 30000 строк.

В таблице `pg_statistic` содержатся данные по каждому столбцу таблиц.

Например, доля строк с NULL в третьем столбце таблицы test:

```
select stanullfrac from pg_statistic where starelid = 'test'::regclass and staattnum = 3;
 stanullfrac
-------------
   0.9988884
```

Статистика о доле пустых значений используется планировщиком.

Более подробно в документации https://docs.tantorlabs.ru/tdb/ru/18_1/se/catalog-pg-statistic.html

Накопительная статистика

- по таблицам:

```
 select schemaname||'.'||relname name, seq_scan, idx_scan, idx_tup_fetch, autovacuum_count, autoanalyze_count from
pg_stat_all_tables where idx_scan is not null order by 3 desc limit 1;
          name           | seq_scan | idx_scan | idx_tup_fetch | autovacuum_count | autoanalyze_count
-------------------------+----------+----------+---------------+------------------+-------------------
  public.pgbench_accounts |        0 | 11183162 |      11183162 |             1512 |               266
  select relname name, n_tup_ins ins, n_tup_upd upd, n_tup_del del, n_tup_hot_upd hot_upd, n_tup_newpage_upd
newblock, n_live_tup live, n_dead_tup dead, n_ins_since_vacuum sv, n_mod_since_analyze sa from pg_stat_all_tables
where idx_scan is not null order by 3 desc limit 1;
       name       | ins |   upd   | del | hot_upd | newblock |  live  |  dead   | sv |  sa
------------------+-----+---------+-----+---------+----------+--------+---------+----+------
  pgbench_tellers |   0 | 5598056 |   0 | 5497197 |   100859 |     10 | 1456051 |  0 |  165
```

- В представлении `pg_statio_all_tables` статистика по всем индексам на таблицу. Статистику (чтение с подгрузкой с диска и из кэша буферов) по конкретному индексу можно посмотреть в представлении `pg_statio_all_indexes`.

Накопительная статистика

В `pg_stat_all_tables` - статистика чтения блоков таблиц, всех индексов на эту таблицу, TOAST таблицы и её индекса (к TOAST доступ идёт всегда через TOAST-индекс и поэтому данные по TOAST-индексу пропорциональны данным по TOAST-таблице) с подгрузкой с диска (столбцы `*_blks_read`) и из кэша буферов (столбцы `*_blks_hit`).

В представлении `pg_statio_all_tables` статистика по всем индексам на таблицу. Статистику (чтение с подгрузкой с диска и из кэша буферов) по конкретному индексу можно посмотреть в представлении `pg_statio_all_indexes`.

Статистика по таблицам:

```
 select schemaname||'.'||relname name, seq_scan, idx_scan, idx_tup_fetch, autovacuum_count, autoanalyze_count
from pg_stat_all_tables where idx_scan is not null order by 3 desc limit 3;
          name           | seq_scan | idx_scan | idx_tup_fetch | autovacuum_count | autoanalyze_count
-------------------------+----------+----------+---------------+------------------+-------------------
  public.pgbench_accounts |        0 | 11183162 |      11183162 |             1512 |               266
  public.pgbench_tellers  |   906731 |  4684850 |       4684850 |             1524 |              1536
  public.pgbench_branches |   907256 |  4684327 |       4684327 |             1527 |              1536
 select relname name, n_tup_ins ins, n_tup_upd upd, n_tup_del del, n_tup_hot_upd hot_upd, n_tup_newpage_upd
newblock, n_live_tup live, n_dead_tup dead, n_ins_since_vacuum sv, n_mod_since_analyze sa from pg_stat_all_tables
where idx_scan is not null order by 3 desc limit 3;
       name       | ins |   upd   | del | hot_upd | newblock |  live  |  dead   | sv |  sa
------------------+-----+---------+-----+---------+----------+--------+---------+----+------
  pgbench_tellers |   0 | 5598056 |   0 | 5497197 |   100859 |     10 | 1456051 |  0 |  165
  pgbench_branches |   0 | 5598056 |   0 | 5589787 |     8269 |      1 | 1456044 |  0 |  175
  pgbench_accounts |   0 | 5598056 |   0 | 3923068 |  1674988 | 100001 | 1456032 |  0 | 7619
```

Статистика **`n_tup_hot_upd`** вакуумом не обновляется.

Представление pg_stat_xact_all_tables имеет те же столбцы, что и pg_stat_all_tables, но показывает только действия, выполненные в текущей транзакции к настоящему моменту и ещё не попавшие в `pg_stat_all_*`. Столбцы для **`n_live_tup, n_dead_tup`** и относящиеся к вакуумированию и анализу в этих представлениях отсутствуют:

```
 select schemaname||'.'||relname name, seq_scan, idx_scan, idx_tup_fetch, n_tup_ins ins, n_tup_upd upd, n_tup_del
del, n_tup_hot_upd hot_upd, n_tup_newpage_upd newblock from pg_stat_xact_all_tables where idx_scan is not null
order by 3 desc limit 3;
          name           | seq_scan | idx_scan | idx_tup_fetch | ins | upd | del | hot_upd | newblock
-------------------------+----------+----------+---------------+-----+-----+-----+---------+----------
  pg_catalog.pg_namespace |        1 |        2 |             0 |   0 |   0 |   0 |       0 |        0
```

Расширение pg_stat_statements

- детальная статистика работы экземпляра с точностью до команд SQL
- для установки нужно загрузить библиотеку и установить расширение:

```
alter system set shared_preload_libraries = pg_stat_statements;
create extension pg_stat_statements;
```

- в расширение входят 3 функции и 2 представления:

```
\dx+ pg_stat_statements
 function pg_stat_statements(boolean)
 function pg_stat_statements_info()
 function pg_stat_statements_reset(oid,oid,bigint,boolean)
 view pg_stat_statements
 view pg_stat_statements_info
```

- команды объединяются в одну строку в `pg_stat_statements`, когда они выполняются одним и тем же пользователем и имеют идентичные структуры запросов, то есть семантически равнозначны, за исключением литералов и переменных подстановки (literal constants)
- Например: `select * from t where id = 'a'` и `select * from t where id = 'b'` объединятся в `select * from t where id = $1`

Расширение pg_stat_statements

Стандартное расширение. Выдает детальную статистику работы экземпляра с точностью до команд SQL. Для установки нужно загрузить библиотеку и установить расширение:

```
alter system set shared_preload_libraries = pg_stat_statements;
create extension pg_stat_statements;
```

В расширение входят 3 функции и 2 представления:

```
\dx+ pg_stat_statements
 function pg_stat_statements(boolean)
 function pg_stat_statements_info()
 function pg_stat_statements_reset(oid,oid,bigint,boolean)
 view pg_stat_statements
 view pg_stat_statements_info
```

Расширение собирает статистики выполнения команд, сгруппированные по командам.

Для группировки команд используется функционал, устанавливаемый параметром конфигурации `compute_query_id`. Значение параметра должно быть в auto (значение по умолчанию) или on.

Команды объединяются в одну команду в `pg_stat_statements`, когда они выполняются одним и тем же пользователем и имеют идентичную структуру, то есть семантически равнозначны, за исключением литералов и переменных подстановки (literal constants). Например, запросы: `select * from t where id = 'a'` и `select * from t where id = 'b'` объединятся в запрос: `select * from t where id = $1`. Запросы с визуально различными текстами могут быть объединены, если они семантически равнозначны. Из-за коллизии хеша могут объединиться разные команды, но вероятность этого невелика. И обратно: команды с одинаковым текстом могут считаться разными, если они получили разное дерево разбора, например, из-за разного `search_path`.

Статистика сбрасывается вызовом функции `pg_stat_statements_reset()`.

В Tantor Postgres 17.5 добавлен параметр конфигурации `pg_stat_statements.sample_rate`, который устраняет проблему деградации производительности при использовании расширения на нагруженных кластерах.

В Tantor Postgres 17.5 также были добавлены параметры конфигурации `pg_stat_statements.mask_const_arrays, pg_stat_statements.mask_temp_tables`, которые маскируют имена массивов и имена временных таблиц, что позволяет более точно группировать статистику по однотипным запросам.

Параметры расширения pg_stat_statements

- pg_stat_statements.max задаёт максимальное число команд, отслеживаемых расширением, то есть, максимальное число строк в представлении pg_stat_statements
- pg_stat_statements.save определяет, должна ли статистика сохраняться после перезагрузки сервера
- pg_stat_statements.track определяет, какие команды будут отслеживаться: top отслеживаются только команды верхнего уровня
- pg_stat_statements.track_planning устанавливает, будут ли отслеживаться операции планирования и длительность фазы планирования
- pg_stat_statements.track_utility будут ли отслеживаться команды, отличные от SELECT, INSERT, UPDATE, DELETE, MERGE

```
select name, setting, context, min_val, max_val from pg_settings where name like
'pg_stat_statements%';
                name                 | setting |  context   | min_val |  max_val
-------------------------------------+---------+------------+---------+------------
 pg_stat_statements.max              | 5000    | postmaster | 100     | 1073741823
 pg_stat_statements.save             | on      | sighup     |         |
 pg_stat_statements.track            | top     | superuser  |         |
 pg_stat_statements.track_planning   | on      | superuser  |         |
 pg_stat_statements.track_utility    | on      | superuser  |         |
```


Параметры расширения pg_stat_statements

```
select name, setting, context, min_val, max_val from pg_settings where name like 'pg_stat_statements%';
                name                 | setting |  context   | min_val |  max_val
-------------------------------------+---------+------------+---------+------------
 pg_stat_statements.max              | 5000    | postmaster | 100     | 1073741823
 pg_stat_statements.save             | on      | sighup     |         |
 pg_stat_statements.track            | top     | superuser  |         |
 pg_stat_statements.track_planning   | on      | superuser  |         |
 pg_stat_statements.track_utility    | on      | superuser  |         |
```

Параметры конфигурации расширения:

pg_stat_statements.max задаёт максимальное число команд, отслеживаемых расширением, то есть, максимальное число строк в представлении pg_stat_statements. Статистика о редко выполняющихся командах обычно не нужна и увеличивать значение не нужно, так как это увеличивает объем разделяемой памяти, выделяемой расширением. Значение по умолчанию 5000.

pg_stat_statements.save определяет, должна ли статистика сохраняться после перезагрузки сервера. Если значение off, то статистика при остановке экземпляра не сохраняется. Значение по умолчанию - on, что означает сохранение статистики при остановке или перезапуске экземпляра.

pg_stat_statements.track определяет, какие команды будут отслеживаться. Принимает значения:

1) top (значение по умолчанию) отслеживаются только команды верхнего уровня (передаваемые клиентами в сессии)

2) all - в дополнение к командам верхнего уровня отслеживаются команды внутри вызываемых функций

3) none - сбор статистики отключён.

pg_stat_statements.track_planning устанавливает, будут ли отслеживаться операции планирования и длительность фазы планирования. Значение on может привести к заметному снижению производительности, особенно когда в нескольких сессиях в одно время выполняются команды с одинаковой структурой запросов, в результате чего эти сессии пытаются одновременно изменить одни и те же строки в pg_stat_statements. Значение по умолчанию off.

pg_stat_statements.track_utility определяет, будет ли расширение отслеживать служебные команды. Служебными командами считаются команды, отличные от SELECT, INSERT, UPDATE, DELETE, MERGE. Значение по умолчанию on.

Практика

1. Создание объектов для запросов
2. Извлечение данных последовательно
3. Возвращение данных по индексу
4. Низкая селективность
5. Использование статистики
6. Представление pg_stat_statements

Практика

Создание объектов для запросов
Извлечение данных последовательно
Возвращение данных по индексу
Низкая селективность
Использование статистики
Представление pg_stat_statements

2

2e Архитектура PostgreSQL

Расширения PostgreSQL

Расширяемость PostgreSQL

- добавление функционала без внесения изменений в файлы исходного кода и перекомпиляции
- можно создавать типы данных, операторы, групповые функции, приведения типов
- создание хранимых подпрограмм на большом числе языков программирования
- создание и использование расширений (extensions)
  - набор любых объектов базы данных, который можно установить или удалить как единое целое
- расширение функционала разделяемыми библиотеками
- обёртки внешних данных (FDW)

Расширяемость PostgreSQL

Расширяемость (extensibility) PostgreSQL - способность быть легко адаптируемым под потребности приложений, администраторов и пользователей. Исторически, PostgreSQL разрабатывался с упором на расширяемость. В ранних версиях PostgreSQL, еще когда она называлась Postgres, создатель системы Майкл Стоунбрейкер уделял внимание расширяемости - добавлению функционала без необходимости вносить изменения в файлы исходного кода, написанного на языке C. Нерасширяемые и закрытые продукты обычно исчезают, остаются только продукты, функционал которых можно легко наращивать сторонним разработчикам.

Можно создавать типы данных, операторы, групповые функции, приведения типов.

Устанавливать языки программирования для написания хранимых подпрограмм.

Расширения (extensions) - набор любых объектов базы данных, который можно установить или удалить как единое целое.

Есть возможность расширять функционал разделяемыми библиотеками (.so файлы)

С помощью расширений (командой CREATE EXTENSION) можно установить (Foreign Data Wrapper, FDW, обёртку внешних данных). FDW обеспечивает возможность работать с данными, расположенными во внешних системах (базах данных, службах, файлах и т.п.) с помощью внешних таблиц (foreign table) с которыми можно работать, как с обычными таблицами. FDW описан в стандарте SQL, как способ работы с внешними данными.

Директории файлов расширений и библиотек

- Файлы библиотек находятся в директории:
- /opt/tantor/db/18/lib/postgresql
- Файлы расширений (*.control и *.sql) находятся в директории:
- /opt/tantor/db/18/share/postgresql/extension
- Узнать местоположение можно командами:

```
postgres@tantor:/opt/tantor/db/18/lib$ pg_config --libdir
/opt/tantor/db/18/lib
postgres@tantor:/opt/tantor/db/18/lib$ pg_config --sharedir
/opt/tantor/db/18/share/postgresql
```

- Установка способом "PGXS":

```
root@tantor:~# export PATH=/opt/tantor/db/18/bin:$PATH
root@tantor:~# export USE_PGXS=1
root@tantor:~# cd директория_расширения
root@tantor:~# make
root@tantor:~# make install
```

- обычно расширения, библиотеки, утилиты, приложения устанавливаются пакетами deb и rpm

Директории файлов расширений и библиотек

Файлы библиотек находятся в директории:
/opt/tantor/db/18/lib/postgresql
Файлы расширений (*.control и *.sql) находятся в директории:
/opt/tantor/db/18/share/postgresql/extension
Узнать местоположение можно командами:

```
postgres@tantor:~$ pg_config --libdir
/opt/tantor/db/18/lib
postgres@tantor:~$ pg_config --sharedir
/opt/tantor/db/18/share/postgresql
```

или запросом:

```
postgres=# SELECT * FROM pg_config where name in ('LIBDIR','SHAREDIR');
   name    |                setting
-----------+-----------------------------------
 LIBDIR    | /opt/tantor/db/18/lib
 SHAREDIR  | /opt/tantor/db/18/share/postgresql
```

Однако, к этим конфигурационным параметрам нужно добавлять поддиректории postgresql и extension. Это неудобно запоминать. Установка расширений возможна путем копирования файлов в эти директории или появившимся позже способом "PGXS", который появился в PostgreSQL относительно недавно, но такой же не удобный. Для этого способа нужно добавить в PATH директорию с утилитой pg_config и переменную окружения, указывающую утилите make использовать логику установки расширений PGXS:

```
root@tantor:~# export PATH=/opt/tantor/db/18/bin:$PATH
root@tantor:~# export USE_PGXS=1
```

после чего перейти в директорию расширения и выполнить команды **make** и **make install**.
Способ довольно запутанный. Поэтому, распространена практика установки расширений, библиотек, утилит, приложений с помощью пакетов deb и rpm.

Установка расширений

- Расширения могут включать в себя разделяемую библиотеку и/или текстовые файлы: управляющий файл расширения и один или более файлов скриптов
- библиотеки загружаются параметрами или командой:

```
postgres=# \dconfig *librar*
 archive_library           |
 dynamic_library_path      | $libdir
 local_preload_libraries   |
 session_preload_libraries |
 shared_preload_libraries  | pg_stat_statements
postgres=# load 'библиотека';
LOAD
postgres=# \dx
```

- объекты расширения устанавливаются командой:

```
postgres=# create extension имя;
CREATE EXTENSION
```

- доступные к установке расширения перечислены в представлении `pg_available_extentions`

Установка расширений

Расширения могут включать в себя разделяемую библиотеку и/или текстовые файлы: управляющий файл расширения и один или более файлов скриптов. Если расширение состоит из одной лишь библиотеки, то библиотека может загружаться несколькими способами, которые должны быть указаны в описании библиотеки. Библиотека должна быть указана в одном из параметров:

```
postgres=# \dconfig *librar*
 archive_library           |
 dynamic_library_path      | $libdir
 local_preload_libraries   |
 session_preload_libraries |
 shared_preload_libraries  | pg_stat_statements
```

или командой LOAD:

```
postgres=# load 'библиотека';
LOAD
```

Если у расширения есть объекты внутри базы данных, например, функции, процедуры, представления, таблицы и т.п., то команды их создания указаны в файле скрипта `.sql`, а характеристики расширения в файле `.control`. Посмотреть список таких расширений можно в представлениях:

```
postgres=# \dv *exten*
                          List of relations
   Schema   |               Name                | Type |  Owner
------------+-----------------------------------+------+----------
 pg_catalog | pg_available_extension_versions | view | postgres
 pg_catalog | pg_available_extensions         | view | postgres
```

Список установленных расширений можно посмотреть командой **\dx**

Устанавливается расширение командой: `CREATE EXTENTION имя`, удаляется командой `DROP EXTENTION`. Командой `ALTER EXTENSION` можно заменить расширение на другую версию или поменять свойства расширения.

Если расширение не стоит использовать, в его название вставляют тире и чтобы его установить нужно имя заключить в двойные кавычки:

```
postgres=# create extension "uuid-ossp";
CREATE EXTENSION
```

У особенно неудачных расширений в названия параметров конфигурации вставляют тире.

Файлы расширений

- пример управляющего файла расширения:

```
postgres@tantor:~$ cat
/opt/tantor/db/18/share/postgresql/extension/pg_stat_kcache.control
# pg_stat_kcache extension
comment = 'Kernel statistics gathering'
default_version = '2.3.1'
requires = 'pg_stat_statements'
module_pathname = '$libdir/pg_stat_kcache'
relocatable = true
```

- пример файла скрипта pg_stat_kcache--2.3.0.sql

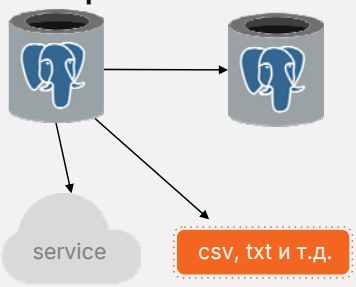
```
\echo Use "CREATE EXTENSION pg_stat_kcache" to load this file. \quit
SET client_encoding = 'UTF8';
CREATE FUNCTION pg_stat_kcache_reset()
    RETURNS void
    LANGUAGE c COST 1000
    AS '$libdir/pg_stat_kcache', 'pg_stat_kcache_reset';
REVOKE ALL ON FUNCTION pg_stat_kcache_reset() FROM public;
...
```


Файлы расширений

Файлы расширений можно просматривать, чтобы изучить как создаются объекты расширения.
Управляющий файл имеет формат `имя.control`
Также должен быть хотя бы один файл скрипта SQL, который следует шаблону именования **`имя--версия.sql`**
который располагается там же, где управляющий файл - в директории SHAREDIR/extension, если только в управляющем файле не указан параметр directory. Если не указан абсолютный путь, то путь относительно директории `SHAREDIR`, что эквивалентно указанию `directory = 'extension'`.
Параметры в управляющем файле:
encoding - кодировка для файлов скриптов. По умолчанию - кодировка базы данных.
requires - названия расширений через запятую и пробел, от которых зависит данное расширение, без них не установится.
relocatable - можно ли объекты расширения переместить в другую схему. По умолчанию нельзя, значение false.
schema - только для неперемещаемых расширений. Схема, в которой создаются объекты расширения командой `CREATE EXTENSION`. При обновлении расширения игнорируется - объекты не перемещаются.
Для отдельных версий расширения там же где управляющий файл могут существовать управляющие файлы с именами вида:
`имя--версия.control`. Параметры, указанные в них, перекрывают параметры основного управляющего файла.
Формат названия файлов скриптов: **`имя--версия.sql`**. Для скриптов смены версий:
`имя-версия--версия.sql`. Содержимое этих файлов выполняется в транзакции, поэтому в них не могут присутствовать команды begin, commit и команды, которые не могут выполняться внутри транзакции.
https://docs.tantorlabs.ru/tdb/ru/18_1/se/extend-extensions.html#EXTEND-EXTENSIONS-FILES

Foreign Data Wrapper

- Обертка внешних данных - функционал для обращения к данным, находящимся вне базы данных из сессии с базой данных PostgreSQL стандартизованным и относительно простым способом
- В PostgreSQL включены две обертки (драйвера): **postgres_fdw** для работы с таблицами в базах данных PostgreSQL и **file_fdw** к содержимому текстовых файлов
- FDW устанавливается как расширение
- Существуют расширения: mysql_fdw, oracle_fdw, sqlite_fdw, mongo_fdw, redis_fdw
- команды psql для просмотра объектов FDW: **`\dew, \des, \deu, \det`**



Foreign Data Wrapper

Foreign Data Wrapper (FDW, Обертка внешних данных) - функционал для обращения к данным, находящимся вне базы данных из сессии с базой данных PostgreSQL стандартизованным и относительно простым способом. Похож на функционал прозрачных шлюзов и dblink в Oracle Database. В PostgreSQL включены две обертки (драйвера): **postgres_fdw** для работы с таблицами в базах данных PostgreSQL и **file_fdw** к содержимому текстовых файлов.

Списки FDW объектов можно посмотреть командами psql: **`\dew, \des, \deu, \det`**

Существуют расширения: mysql_fdw, oracle_fdw, sqlite_fdw, mongo_fdw, redis_fdw и другие.

По умолчанию, соединения, устанавливаемые postgres_fdw со сторонними службами, остаются открытыми для повторного использования в той же сессии, в которой было обращение к внешней таблице.

Расширение postgres_fdw

```
create extension postgres_fdw;
CREATE EXTENSION
create database test_db;
CREATE DATABASE
\! pgbench -i -d test_db 2> /dev/null
create server test foreign data wrapper postgres_fdw options (host
'localhost', port '5432', dbname 'test_db', use_scram_passthrough
'true');
CREATE SERVER
create user mapping for postgres server test options (user 'postgres');
CREATE USER MAPPING
import foreign schema public limit to (pgbench_tellers) from server test
into public;
IMPORT FOREIGN SCHEMA
select * from pgbench_tellers limit 1;
 tid | bid | tbalance | filler
-----+-----+----------+--------
   1 |   1 |        0 |
```


Расширение postgres_fdw

Расширение `postgres_fdw` позволяет создавать или импортировать определения внешних таблиц (FOREIGN TABLE) в других базах данных того же или другого кластера PostgreSQL, находящимся на том же или другом хосте. С внешними таблицами можно работать так же, как с обычными таблицами. Внешние таблицы могут участвовать в любых командах вместе с обычными таблицами и представлениями.

FDW устанавливается как расширение, может включать в себя библиотеку. `postgres_fdw` не требует загрузки библиотеки, так как функционал встроен в ядро PostgreSQL, как и функционал `file_fdw`. Расширение реализует логику драйвера (адаптера) доступа к внешней программной системе по ее протоколу.

После установки расширения создаются объекты:

FOREIGN SERVER - указываются детали подсоединения к внешней системе. Например, название базы, сетевой адрес, порт. Пример:

```
CREATE SERVER conn1 FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'localhost',
port '5432', dbname 'postgres', use_scram_passthrough 'true');
```

В 18 версии появилась возможность не хранить пароли пользователей в USER MAPPING, указав параметр на уровне SERVER или FOREIGN TABLE.

USER MAPPING - если у внешней системы есть аккаунты (пользователи, группы, роли), то можно сопоставить роли кластера с аккаунтами внешней системы.

FOREIGN TABLE - всегда создается или импортируется. Для внешнего источника данных в базе данных PostgreSQL создается локальный объект, выглядящий как таблица или представление. При использовании FDW работа с внешними данными идёт как с таблицами.

Эти таблицы можно использовать в запросах, соединениях с обычными, временными и прочими таблицами. Команды insert, update, delete могут быть реализованы, но это зависит от внешнего источника данных.

Расширение file_fdw

- file_fdw позволяет создавать виртуальные таблицы на основе данных, хранящихся в текстовых файлах
- вносить изменения в файлы нельзя, только читать
- индексов нет, при каждом SELECT просматривается весь файл
- пример создания объектов FDW:

```
CREATE EXTENSION file_fdw;
CREATE SERVER csv_server FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE t1 (
  column1 text,
  column2 numeric,
  ...
)
SERVER csv_server OPTIONS (filename '/path/to/file.csv', format 'csv');
```


Расширение file_fdw

file_fdw позволяет создавать виртуальные таблицы на основе данных, хранящихся в файлах различных форматов, таких как CSV. Используется для чтения строк текстовых файлов и представления их как обычных таблиц. Пример:

```
CREATE EXTENSION file_fdw;
CREATE SERVER csv_server FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE t1 (
  column1 text,
  column2 numeric,
  ...
)
SERVER csv_server OPTIONS (filename '/path/to/file.csv', format 'csv',
reject_limit 1000);
```

Для `file_fdw` удаление, изменение, вставка строк в текстовые файлы не реализована. Можно только читать содержимое файлов (команды SELECT и WITH).

В 18 версии появился параметр **`reject_limit`**, задающий максимальное число ошибок при приведении значения поля к типу данных его столбца.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/file-fdw.html

Расширение dblink

- позволяет посылать команды на выполнение и получать результат

```
SELECT * FROM dblink('dbname=postgres user=postgres', $$ select 7; $$ ) as (col1 int);
 7
SELECT * FROM dblink_connect('connection1','host=/var/run/postgresql port=5432');
 OK
SELECT * FROM dblink_send_query('connection1', $$ select 8 from pg_sleep(1); $$ );
 1
SELECT dblink_is_busy('connection1');
 1
SELECT * FROM dblink_get_result('connection1') as t(col1 int);
 8
SELECT dblink_is_busy('connection1');
 0
SELECT * FROM dblink_exec('connection1', $$ CHECKPOINT; $$);
 CHECKPOINT
SELECT * FROM dblink_disconnect('connection1');
 OK
```


Расширение file_fdw

Для доступа к базам данных PostgreSQL, может использоваться стандартное расширение "dblink". Функциями этого расширения можно посылать любые команды и получать результат. Его способ работы не похож на dblink Oracle Database. Расширение появилось раньше, чем спецификация FDW.

dblink позволяет посылать на выполнение любые команды и получать результат. Пример:

```
SELECT * FROM dblink('dbname=postgres user=postgres', $$ select 7; $$ ) as (col1 int);
  7
SELECT * FROM dblink_connect('connection1', 'host=/var/run/postgresql port=5432');
  OK
SELECT * FROM dblink_send_query('connection1', $$ select 8 from pg_sleep(1); $$ );
  1
SELECT dblink_is_busy('connection1');
  1
SELECT * FROM dblink_get_result('connection1') as t(col1 int);
  8
SELECT dblink_is_busy('connection1');
  0
SELECT * FROM dblink_exec('connection1', $$ CHECKPOINT; $$);
  CHECKPOINT
SELECT * FROM dblink_disconnect('connection1');
  OK
```

Примечание: в 17 и 18 версии без вызова, помеченного цветом, последующая команда выдаст ошибку.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/dblink.html

Практика

1. Определение директории с файлами расширения
2. Просмотр установленных расширений
3. Просмотр доступных расширений
4. Установка и удаление произвольного обновления
5. Просмотр доступных версий расширений
6. Обновление до актуальной версии
7. Обертки внешних данных

Практика

Определение директории с файлами расширения
Просмотр установленных расширений
Просмотр доступных расширений
Установка и удаление произвольного обновления
Просмотр доступных версий расширений
Обновление до актуальной версии
Обертки внешних данных

3

Конфигурирование PostgreSQL

Обзор

- имеется около 370 параметров
- влияют на работу экземпляра
- название нечувствительно к регистру
- расширения и приложения могут использовать свои параметры конфигурации, у таких параметров в названии присутствует точка
- настройка экземпляра - установка значений параметров конфигурации так, чтобы работа экземпляра была оптимальной при текущей нагрузке

Обзор

В 18 версии PostgreSQL 398 параметров конфигурации без учёта параметров расширений. В Tantor Postgres SE 18 - 429 параметра.

Настройка экземпляра, по большей части, заключается в установке значений параметров конфигурации на различных уровнях так, чтобы работа экземпляра была оптимальной при текущей нагрузке.

Параметры имеют название, которое нечувствительно к регистру и значение.

Типы значений параметров:
логические (значение "`bool`" в столбце vartype представления pg_settings)
строковыми ("`string`")
целые числа ("`integer`", "`int64`")
вещественные числа ("`real`")
числа ("`integer`", "`int64`", "`real`") с единицей измерения (`unit`) в байтах или времени значениями из списка ("`enum`").

Названия типов параметров не связаны с типами данных SQL. Максимальные и минимальные значения численных типов для каждого параметра указаны в столбцах `min_val` и `max_val` представления `pg_settings`.

Значения строковых параметров лучше заключать в апострофы. Если в самом значении присутствует апостроф, то задублировать апостроф (два апострофа).

Для численных параметров с единицами измерений допустимыми обозначениями единиц измерения являются (регистр важен): `B` (байты), `kB` (килобайты), `MB` (мегабайты), `GB` (гигабайты) и `TB` (терабайты); `us` (микросекунды), `ms` (миллисекунды), `s` (секунды), `min` (минуты), `h` (часы) и `d` (дни). Сами значения лучше заключать в апострофы.

Для "`enum`" список допустимых значений можно посмотреть в столбце `enumvals` представления `pg_settings`.

Расширения и приложения могут определять и использовать свои параметры конфигурации, у таких параметров в названии присутствует точка.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-custom.html
https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config.html

Параметры конфигурации

- Первым просматривается основной файл конфигурации `postgresql.conf`
  Параметры могут дублироваться применяется последний
- Дальше применяются параметры `$PGDATA/postgresql.auto.conf`
- Параметры командной строки передаваемые процессу postgres параметром `-c` превалируют над значениями, установленными в файлах параметров:

```
postgres@tantor:~/tantor-se-18/data$
pg_ctl start -o "-c config_file=./postgresql.conf"
waiting for server to start.... done
server started
```


Параметры конфигурации

При создании кластера создаются два файла с параметрами конфигурации:
1) основной файл с параметрами конфигурации кластера `postgresql.conf`
Если кластер запущен расположение можно посмотреть в значении параметра `config_file`
Установить значение параметра `config_file` можно **только в командной строке при запуске кластера.**
Можно посмотреть параметры основного процесса: `postgres --help`

```
 postgres is the PostgreSQL server.
 Usage:
   postgres [OPTION]...
 Options:
   -B NBUFFERS         number of shared buffers
   -c NAME=VALUE       set run-time parameter
   -C NAME             print value of run-time parameter, then exit
   -d 1-5              debugging level
   -D DATADIR          database directory
```

Ключом `-c` можно передавать любые параметры конфигурации. Пример:

```
pg_ctl start -o "-c config_file=/opt/postgresql.conf"
```

2) файл `postgresql.auto.conf`, который всегда находится в директории `PGDATA`. Если кластер запущен, расположение `PGDATA` можно посмотреть в значении параметра `data_directory`
Выносить файл `postgresql.conf` вне директории `PGDATA` может быть удобно при резервировании и восстановлении. Утилита резервирования `pg_basebackup` копирует только содержимое `PGDATA` (и табличных пространств) и параметры, специфичные для резервного хоста, можно поместить в файл `postgresql.conf` вне `PGDATA`. То, что находится вне `PGDATA` (и директорий табличных пространств) не затирается утилитами. Утилита `pg_rewind` также синхронизирует только директорию `PGDATA` (и директории табличных пространств) и всё, что в них находится, копирует с мастера.

Просмотр параметров

- Команда `\dconfig`
  - можно искать используя символы * ?
  - можно использовать клавишу табуляции
  - без параметров выдаёт все параметры, которые отличаются от значений по умолчанию
- Команда `SHOW имя_параметра;`
  - символы * ? нельзя использовать
  - можно использовать клавишу табуляции
- функция `current_setting('имя параметра')`

```
postgres=# \dconfig d?ta_d*
              List of configuration parameters
      Parameter       |                  Value
----------------------+----------------------------------------
 data_directory       | /var/lib/postgresql/tantor-se-18/data
 data_directory_mode  | 0750
```


Просмотр параметров

Текущие значения параметров кластера удобно просматривать командой:

```
postgres=# \dconfig d?ata_d*
              List of configuration parameters
      Parameter       |                  Value
----------------------+----------------------------------------
 data_directory       | /var/lib/postgresql/tantor-se-18/data
 data_directory_mode  | 0750
```

Команда показывет значения параметров, в которых встречается строка `d?ta_d`

Команда `SHOW` покажет текущие значения параметра. Клавиша табуляции в `psql` покажет допустимые значения. `SHOW` показывает один параметр.

Неудобство команды `SHOW` в том, что ее надо завершать "**;**" или "**\gx**".

```
postgres=# show data_directory;
            data_directory
---------------------------------------
 /var/lib/postgresql/tantor-se-18/data
(1 row)
```

Очистить буфер psql можно командой **\reset**, посмотреть командой **\print**:

```
postgres=# show data_directory
postgres-# select 1
postgres-# \p
show data_directory
select 1
postgres-# \r
Query buffer reset (cleared).
postgres=#
```

Функция `current_setting(параметр)` - аналог команды `SHOW`

Просмотр параметров

- Команда `\dconfig`
- В представлениях `pg_settings pg_file_settings`
- Командой `SHOW ALL;`
  - нельзя отфильтровать параметры
  - аналог `select * from pg_settings;`
- В файлах `postgresql.conf postgresql.auto.conf`
- просмотр значения одного параметра на запущенном или остановленном экземпляре `postgres -C имя_параметра`

```
postgres@tantor:~$ postgres -C autovacuum_freeze_max_age
10000000000
postgres@tantor:~$ ls -CF $PGDATA
base/           pg_logical/    pg_stat/      pg_wal/
global/         pg_multixact/  pg_stat_tmp/  pg_xact/
pg_commit_ts/   pg_notify/     pg_subtrans/  postgresql.auto.conf
pg_dynshmem/    pg_replslot/   pg_tblspc/    postgresql.conf
pg_hba.conf     pg_serial/     pg_twophase/  postmaster.opts
pg_ident.conf   pg_snapshots/  PG_VERSION    postmaster.pid
```


Просмотр параметров (продолжение)

Текущие значения параметров кластера удобно просматривать командой psql **`\dconfig маска_параметра`**

Функция `current_setting(имя параметра)` - аналог команды `SHOW`.

Просмотр одного параметра на запущенном или остановленном экземпляре:

```
postgres -C имя_параметра
```

Расчётное значение, сколько экземпляру потребуется разделяемой памяти и памяти в Huge Pages (если сможет выделить) можно посмотреть до запуска экземпляра командами:

```
postgres@tantor:~$ postgres -C shared_memory_size
218
postgres@tantor:~$ postgres -C shared_memory_size_in_huge_pages
109
```

Параметры появились в 15 версии. Память - в мегабайтах, страницы в виде числа страниц.

Примечательно, что на запущенном экземпляре эти же команды выдают ошибку:

```
postgres@tantor:~$ postgres -C shared_memory_size
FATAL:  lock file "postmaster.pid" already exists
HINT:  Is another postmaster (PID 163496) running in data directory
"/var/lib/postgresql/tantor-se-18/data"?
```

При этом, можно подключиться и посмотреть значения командой **`\dconfig`**:

```
postgres=# \dconfig *shared_mem*|*huge*
      List of configuration parameters
            Parameter              | Value
-----------------------------------+-------
 dynamic_shared_memory_type        | posix
 huge_pages                        | try
 huge_page_size                    | 0
 huge_pages_status                 | off
 min_dynamic_shared_memory         | 0
 shared_memory_size                | 218MB
 shared_memory_size_in_huge_pages  | 109
 shared_memory_type                | mmap
(8 rows)
```

Представления для просмотра параметров

- Представление `pg_settings` показывает текущие действующие значения параметров
- Представление `pg_file_settings`
  - параметры, которые явно указаны в файлах параметров
  - не показывает текущие значения, которые использует экземпляр
  - столбец `applied` имеет значение "f", если значение параметра: отличается от текущего и для применения значения из файла требуется перезапуск кластера
- Представление `pg_hba_file_rules` показывает содержимое файла `pg_hba.conf`

```
postgres=# select pg_read_file('./postgresql.auto.conf') \g
   (tuples_only=on format=unaligned)
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
log_filename = 'postgresql-%F.log'
cluster_name = 'master'
logging_collector = 'on'
```


Представления для просмотра параметров

Представление `pg_file_settings` параметры, которые явно указаны в файлах параметров. Это представление может быть полезным для предварительного тестирования изменений в конфигурационных файлах - не допущена ли ошибка при редактировании файлов. Представление `pg_file_settings` не показывает текущие значения, которые использует экземпляр. Столбец `applied` имеет значение **"f"**, если значение параметра: отличается от текущего и для применения значения из файла требуется перезапуск кластера. В остальных случаях (значение не менялось или достаточно перечитать файлы) значение в столбце `applied` будет "`t`".

Представление `pg_settings` показывает текущие действующие значения параметров. Команда `SHOW ALL;` аналог запроса к представлению `pg_settings`, но нельзя вывести только часть параметров, поэтому `SHOW ALL;` неудобна.

Содержимое **любого файла** можно посмотреть с помощью функции

```
select pg_read_file('./postgresql.auto.conf') \g (tuples_only=on
format=unaligned)
```

Представление `pg_hba_file_rules` показывает содержимое файла `pg_hba.conf`. Из столбца error этого представления можно узнать описание ошибки, если она допущена при редактировании файла. Файл `pg_hba.conf` и `pg_ident.conf` содержат настройки безопасности.

Файл `postgresql.conf` редактируется вручную.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/config-setting.html

Основной файл параметров postgresql.conf

- основной файл, в котором хранятся параметры конфигурации кластера
- может содержать ~400 параметров плюс параметры расширений и разделяемых библиотек (*.so)
- Создаётся на основе файла postgresql.conf.sample утилитой initdb
- Можно редактировать
- Можно включать содержимое других файлов параметрами конфигурации include и include_dir заданными внутри postgresql.conf

Основной файл параметров postgresql.conf

Файл `postgresql.conf` - основной файл, в котором хранятся параметры конфигурации кластера. В 18 версии PostgreSQL имеется 398 параметров конфигурации плюс параметры расширений и разделяемых библиотек (`*.so`), загружаемых с помощью параметров конфигурации `*_preload_libraries` и команды `LOAD`. В Tantor Postgres SE 18 имеется 429 параметра.

Файл создаётся утилитой `initdb` из файла-шаблона:

```
/opt/tantor/db/18/share/postgresql$ ls -w 1 *.sample
pg_hba.conf.sample
pg_ident.conf.sample
pg_service.conf.sample
postgresql.conf.sample
psqlrc.sample
```

Закомментированные строки начинаются символом `#`.

Список параметров, на которые реагирует postgres (а не параметры расширений и произвольные параметры приложений) можно вывести в файл

```
postgres --describe-config > file.txt
```

Столбцы в файле разделены табуляцией.

Использование `include` и `include_dir` может быть полезным для компаний, предоставляющих облачные решения в виде большого числа кластеров с почти одинаковой конфигурацией для разных клиентов. Но нужно помнить о том, что параметр указанный "ниже" перекрывает параметр заданный "выше" (ближе к началу файла конфигурации).

Файл параметров postgresql.auto.conf

- Расположен в директории `PGDATA`
- Параметры добавляются командой
  › `ALTER SYSTEM SET параметр = значение;`
- Параметры убираются командой
  `ALTER SYSTEM RESET параметр;`
  `ALTER SYSTEM RESET ALL;`
- Чтобы новые значения вступили в силу после изменения командой `ALTER SYSTEM` нужно перечитать конфигурацию или перегрузить кластер

```
postgres=# select pg_reload_conf();
 pg_reload_conf
----------------
 t
postgres=# \! pg_ctl reload
server signaled
```


Файл параметров postgresql.auto.conf

Файл `postgresql.auto.conf` это текстовый файл, расположенный в директории `PGDATA`. Его можно редактировать напрямую, но нежелательно, так как можно опечататься. Цель его создания - возможность вносить изменения в параметры конфигурации кластера командой `ALTER SYSTEM`, в том числе при подсоединении по сети, без необходимости редактировать файлы в файловой системе сервера.

Синтаксис

```
ALTER SYSTEM SET параметр { TO | = } { значение [, ...] | DEFAULT };
ALTER SYSTEM RESET параметр;
ALTER SYSTEM RESET ALL;
```

Изменения после этой команды, а также после редактирования любых файлов конфигурации не применяются, нужно перечитать конфигурацию или перегрузить кластер. Перегрузка кластера нужна только для применения параметров, которые не могут быть изменены динамически (без перегрузки кластера). Такие параметры можно назвать "статическими".

Только пользователи с атрибутом `SUPERUSER` и пользователи, которым предоставлена привилегия `ALTER SYSTEM`, могут изменять параметры кластера с помощью команды `ALTER SYSTEM`.

Команда `ALTER SYSTEM` не может выполняться в открытой транзакции.

```
postgres=# begin;
BEGIN
postgres=*# alter system set work_mem = '4MB';
ERROR:  ALTER SYSTEM cannot run inside a transaction block
postgres@tantor:~$ psql -c "alter system set work_mem = '4MB';select
pg_reload_conf()"
ERROR:  ALTER SYSTEM cannot run inside a transaction block
postgres@tantor:~$ psql -c "alter system set work_mem = '4MB'" && psql -c
"select pg_reload_conf()"
ALTER SYSTEM
 pg_reload_conf
----------------
 t
```

Применение изменений параметров

- Для применения изменений перечитываются все файлы конфигурации и применяются изменения которые можно провести без перезагрузки кластера
- Способы:
  - `SELECT pg_reload_conf();`
  - `pg_ctl reload`
  - `sudo killall -HUP postgres`
- Параметры в postgresql.auto.conf перекрывают значения параметров postgresql.conf
- Если в postgresql.conf параметр указан несколько раз, применяется указанный ближе к концу файла

Применение изменений параметров конфигурации

Для применения изменений (перечитывания) в текстовых файлах параметров конфигурации, удобно использовать функцию

```
SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

Можно использовать `pg_ctl`:

```
pg_ctl reload -D /var/lib/postgresql/tantor-se-18/data
server signaled
```

Можно послать сигнал SIGHUP (номер 1) основному процессу.

Например, послать сигнал процессам с названием postgres (синоним "postmaster") всех запущенных экземпляров PostgreSQL:

```
killall -1 postgres
```

Параметры, заданные в postgresql.auto.conf перекрывают значения параметров postgresql.conf

Если в файле конфигурации параметр указан несколько раз, то применяется то значение, которое находится ближе к концу файла.

Привилегии на изменение параметров

- Тип параметра указан в столбце `context` представления `pg_settings`
- Команду ALTER SYSTEM может выполнять роль с атрибутом SUPERUSER и те роли, которым была дана привилегия `GRANT ALTER SYSTEM on PARAMETER имя_параметра to роль;`
- Посмотреть список привилегий можно командой psql:

```
\dconfig+ имя_привилегии
      Parameter       | Value | Type |  Context   |  Access privileges
----------------------+-------+------+------------+---------------------
 update_process_title | off   | bool | superuser  | postgres=sA/postgres+
                      |       |      |            | user1=s/postgres
```

- или запросом select * from pg_parameter_acl;
- A - право на ALTER SYSTEM, s - право на SET
- параметром `allow_alter_system` можно отключить возможность использования команды ALTER SYSTEM

Привилегии на изменение параметров конфигурации

Часть параметров конфигурации может быть изменена только ролью с атрибутом `SUPERUSER`. Пример установки атрибута у пользователя `user1`:

```
alter user user1 superuser;
```

Начиная с 16 версии, появилась возможность давать привилегию на изменение параметров, которые может менять только роль с атрибутом `SUPERUSER`.

Выдача привилегии на изменение параметра конфигурации:

```
create role user1 login;
grant alter system on parameter update_process_title to user1;
```

Также, есть привилегия на установку параметров на уровне сессии:

```
grant set on parameter update_process_title to user1;
```

Отозвать выданную привилегию можно командой

```
revoke alter system, set on parameter update_process_title from user1;
```

Посмотреть список привилегий можно командой psql:

```
\dconfig+ *
```

Привилегии будут указаны в столбце `Access privileges`

```
postgres=# \dconfig+ update_process_title
                    List of configuration parameters
      Parameter       | Value | Type |  Context   |  Access privileges
----------------------+-------+------+------------+---------------------
 update_process_title | off   | bool | superuser  | postgres=sA/postgres+
                      |       |      |            | user1=s/postgres
```

Где `A` - право на `ALER SYSTEM`, `s` - право на `SET`.

Недостаток - нельзя отфильтровать по наличию привилегии. Более удобно использовать запрос `select * from pg_parameter_acl;` который выдаёт только те параметры, на которые были назначены привилегии

В 17 версии появился параметр `allow_alter_system`. Если его отключить, то команда alter system не будет работать (даже у суперпользователей).

https://docs.tantorlabs.ru/tdb/ru/18_1/se/ddl-priv.html
https://docs.tantorlabs.ru/tdb/ru/18_1/se/view-pg-settings.html

Классификация параметров: Контекст

- internal - параметры только для чтения
- postmaster - нужен перезапуск экземпляра кластера
- sighup - достаточно перечитать файлы конфигурации
- superuser - могут устанавливаться на уровне сессии, но у пользователя должен быть атрибут SUPERUSER или привилегия на изменение этого параметра
- superuser-backend - не могут быть изменены после создания сессии, но могут быть установлены для конкретной сессии в момент подсоединения, если есть привилегии
- backend - как и предыдущий, но привилегий не нужно
- user - можно менять в течение сессии или на уровне кластера

Классификация параметров: Контекст

Параметров конфигурации много - около 400. Рассмотрим, как классифицируются параметры. Первый разрез классификации - по способу (контексту) применения параметра.

В столбце `context` представления `pg_context` есть 7 вариантов значений.

```
select context, count(name) from pg_settings where name not like '%.%' group by context order by 1;
```

```
       context       | count
---------------------+-------
 backend             |     2
 internal            |    20
 postmaster          |    75
 sighup              |   106
 superuser           |    50
 superuser-backend   |     4
 user                |   172
(7 rows)
```

internal - не устанавливаются в файлах конфигурации и доступны только для чтения

postmaster - для применения требуют перезапуск экземпляра кластера

sighup - для применения достаточно перечитать файлы, например, выполнить функцию **`pg_reload_conf()`** или команду **`pg_ctl reload`**

superuser - могут устанавливаться на уровне сессии, но у пользователя должен быть атрибут `SUPERUSER` или привилегия на изменение этого параметра

superuser-backend - не могут быть изменены после создания сессии, но могут быть установлены для конкретной сессии в момент подсоединения, если есть привилегии

backend - не могут быть изменены после создания сессии, но могут быть установлены для конкретной сессии в момент подсоединения любой ролью

user - можно менять в течение сессии или на уровне кластера в файлах параметров, в последнем случае перечитав файлы

https://docs.tantorlabs.ru/tdb/ru/18_1/se/view-pg-settings.html

Параметры контекста internal

- 20 параметров в 18 версии
- только для чтения

```
postgres=# select name, setting from pg_settings where context='internal' order by 1;
              name               | setting
---------------------------------+----------
 block_size                      | 8192
 data_checksums                  | on
 data_directory_mode             | 0700
 debug_assertions                | off
 huge_pages_status               | off
 in_hot_standby                  | off
 integer_datetimes               | on
 max_function_args               | 100
 max_identifier_length           | 63
 max_index_keys                  | 32
 num_os_semaphores               | 174
 segment_size                    | 131072
 server_encoding                 | UTF8
 server_version                  | 17.2
 server_version_num              | 170002
 shared_memory_size              | 189
 shared_memory_size_in_huge_pages | 95
 ssl_library                     | OpenSSL
 wal_block_size                  | 8192
 wal_segment_size                | 16777216
(20 rows)
```


Параметры контекста internal

В 18 версии PostgreSQL есть 20 параметров, значения которых нельзя поменять. Они не устанавливаются в файлах конфигурации и доступны только для чтения.

В 17 версии было 19 параметров. В 18 версии появился параметр `num_os_semaphores`, который выдаёт сколько семафоров понадобится экземпляру, исходя из установленных значений параметров: `max_connections`, `autovacuum_worker_slots`, `max_wal_senders`, `max_worker_processes`.

Параметр `autovacuum_worker_slots` появился в 18 версии и устанавливает максимальное ограничение на значение на число запущенных рабочих процессов автовакуума `autovacuum_max_workers`.

Часть параметров задана при сборке и устанавливает ограничения (лимиты) PostgreSQL. Часть параметров описательная - отражение текущего режима работы экземпляра или кластера и при смене режима по документированной процедуре изменит значение.

Список параметров этого типа (**`internal`**) можно посмотреть запросом:

```
select * from pg_settings where context='internal' order by 1;
```

Параметры, значения которых могут меняться:

`in_hot_standby` - описательный параметр для реплики

`data_directory_mode` - описательный, показывает разрешения, которые были установлены на `data_directory (PGDATA)` на момент запуска экземпляра

`server_encoding` - задаётся при создании кластера

`server_version` и `server_version_num` - процедурой обновления версии

`wal_segment_size` - меняется утилитой `pg_resetwal`

`shared_memory_size*` - описательные параметры, зависят от `huge_page_size`

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-preset.html

Классификация параметров: Уровни

- Кластера командой
  - `ALTER SYSTEM SET параметр = значение;`
- Базы данных
  - `ALTER DATABASE имя SET параметр = значение;`
- Роли
  - `ALTER ROLE роль SET параметр = значение;`
- Сессии создаваемой ролью, подсоединяющейся к конкретной базе данных
  - `ALTER ROLE роль IN DATABASE имя SET параметр = значение;`

Классификация параметров: Уровни

Если у параметра в столбце `context` представления `pg_settings` значение `не internal`, то этот параметр можно поменять командой `ALTER SYSTEM` или отредактировав файлы параметров конфигурации.

Если у параметра в столбце `context` представления `pg_settings` значения `user, backend, superuser`, то значение параметра можно поменять на других уровнях:

На уровне базы данных можно установить значение параметра командами:

```
ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT };
ALTER DATABASE name SET parameter FROM CURRENT;
ALTER DATABASE name RESET parameter;
ALTER DATABASE name RESET ALL;
```

На уровне роли или роли, подсоединенной к базе данных:

```
ALTER ROLE .. [ IN DATABASE name ] SET parameter { TO | = } { value | DEFAULT }
ALTER ROLE .. [ IN DATABASE name ] SET parameter FROM CURRENT;
ALTER ROLE .. [ IN DATABASE name ] RESET parameter;
ALTER ROLE .. [ IN DATABASE name ] RESET ALL;
```

Примечание: столбец `category` представления `pg_settings` отражает название подсистемы, на которую влияет параметр, а не уровень установки. Этот столбец используется для классификации параметров.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-alterdatabase.html
https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-alterrole.html

Классификация параметров: Уровни

- Внутри сессии
  - `SET work_mem to '16MB';`
  - `SELECT set_config('work_mem', '16MB', false);`
- Транзакции
  - `SET LOCAL work_mem to '16MB';`
  - `SELECT set_config('work_mem', '16MB', true);`
- На время выполнения функции или процедуры
  - `CREATE {FUNCTION|PROCEDURE} .. SET параметр {TO значение | = значение | FROM CURRENT}`
  - `ALTER {PROCEDURE | FUNCTION}.. SET параметр {TO | = }{значение | DEFAULT};`

Классификация параметров: Уровни (продолжение)

На уровне транзакции значение меняется командой `SET LOCAL`.
Пример:
`SET work_mem to '16MB'; или SELECT set_config('work_mem', '16MB', false);` если `false`, то установить на уровне сессии
`SET work_mem to DEFAULT;` сбрасывает на значение, которое имел бы параметр, если бы в текущей сессии не выполнялись команды SET
`RESET work_mem;` то же самое, что и предыдущая команда
`SET LOCAL work_mem to '16MB';` или `SELECT set_config('work_mem', '16MB', true);`
`ALTER {PROCEDURE | FUNCTION}` и дальше одно из перечисленного:

```
SET параметр { TO | = } { значение | DEFAULT };
   SET параметр FROM CURRENT
   RESET параметр
   RESET ALL
```

Последние **две опции** убирают из свойств подпрограммы значения параметров, которые были ранее установлены (при ее создании или изменении).
https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-set.html
https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-alterprocedure.html

Параметры хранения на уровне таблиц

- Устанавливаются командой
  - `ALTER TABLE имя SET(параметр_хранения = значение);`
- Можно установить в команде CREATE TABLE
- Перекрывают аналогичные параметры конфигурации при работе автовакуума с конкретной таблицей и/или её TOAST таблицей
- Команда перекрывающая значение параметра конфигурации default_statistics_target для столбца таблицы:
  - `ALTER TABLE имя ALTER COLUMN имя SET STATISTICS число;`

Параметры хранения на уровне таблиц и индексов

На уровне таблиц **и индексов** есть возможность установить параметры хранения. Параметрами хранения на уровне таблиц можно переопределить параметры для автовакуума при работе с таблицей и/или её TOAST таблицей.

`ALTER TABLE имя SET ( параметр_хранения = значение);`

`ALTER TABLE имя ALTER COLUMN имя SET STATISTICS число;` перекрывает значение параметра конфигурации `default_statistics_target` для столбца таблицы. Диапазон значений от 0 до 10000. -1 возвращает к использованию `default_statistics_target`.

`ALTER` **`INDEX`** `имя ALTER COLUMN` **`номер_столбца_индекса`** `SET STATISTICS число;` перекрывает значение параметра конфигурации `default_statistics_target` для столбца индекса.

Параметры с префиксом "toast." влияют на работу с TOAST таблицей. Если они не установлены, на TOAST действует параметры таблицы.

```
postgres=# alter table имя set (toast.<нажать клавишу табуляции два раза>
toast.autovacuum_enabled
toast.autovacuum_freeze_max_age
toast.autovacuum_freeze_min_age
toast.autovacuum_freeze_table_age
toast.autovacuum_multixact_freeze_max_age
toast.autovacuum_multixact_freeze_min_age
toast.autovacuum_multixact_freeze_table_age
toast.autovacuum_vacuum_cost_delay
toast.autovacuum_vacuum_cost_limit
toast.autovacuum_vacuum_insert_scale_factor
toast.autovacuum_vacuum_insert_threshold
toast.autovacuum_vacuum_scale_factor
toast.autovacuum_vacuum_threshold
toast.log_autovacuum_min_duration
toast.vacuum_index_cleanup
toast.vacuum_truncate
postgres=# alter table t set (toast.<нажать клавишу табуляции два раза>
```

В PostgreSQL есть довольно много типов индексов. Параметры хранения индексов зависят от их типа. Например, для индексов типа btree, hash, GiST, SP-GIST можно установить параметр fillfactor. Для btree - deduplicate_items. Для GiST - buffering. Для GIN - fastupdate. Для BRIN - pages_per_range и autosummarize. В PostgreSQL можно добавлять и индексы и методы хранения данных в таблицах с помощью расширений.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-createtable.html
https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-altertable.html
https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-createindex.html

Классификация параметров: Категории

- Категории описывают для чего предназначены параметры

```
select category, count(name) from pg_settings group by category
  order by 2 desc;
                    category                          | count
------------------------------------------------------+-------
 Query Tuning / Planner Method Configuration          |    28
 Resource Usage / Memory                              |    28
 Client Connection Defaults / Statement Behavior      |    27
 Developer Options                                    |    27
 Reporting and Logging / What to Log                  |    22
 Preset Options                                       |    20
 Query Tuning / Planner Cost Constants                |    17
 Query Tuning / Other Planner Options                 |    17
 Vacuuming / Automatic Vacuuming                      |    15
...
(47 rows)
```


Классификация параметров: Категории

Параметры логически разбиты на категории. Категории описывают для чего предназначены параметры. Названия категорий можно посмотреть запросом:

```
select category, count(name) from pg_settings group by category order by 2 desc;
                    category                          | count
------------------------------------------------------+-------
 Query Tuning / Planner Method Configuration          |    28
 Resource Usage / Memory                              |    28
 Client Connection Defaults / Statement Behavior      |    27
 Developer Options                                    |    27
 Reporting and Logging / What to Log                  |    22
 Preset Options                                       |    20
 Query Tuning / Planner Cost Constants                |    17
 Query Tuning / Other Planner Options                 |    17
 Vacuuming / Automatic Vacuuming                      |    15
 Connections and Authentication / SSL                 |    15
 Write-Ahead Log / Settings                           |    15
 Replication / Standby Servers                        |    14
 Reporting and Logging / Where to Log                 |    13
 Client Connection Defaults / Locale and Formatting   |    12
...
(47 rows)
```

Много параметров относятся к настройке производительности работы: настройке выполнения запросов, автовакуума.

Категория: "Для разработчиков"

- Категория параметров `Developer Options`
- Параметры этой категории не должны использоваться при промышленной эксплуатации
- Параметры из этой категории могут помочь получить данные в сложных случаях повреждения страниц
- `ignore_system_indexes` - игнорировать системные индексы при чтении системных таблиц (но все же обновлять индексы при изменении таблиц). Может пригодиться, если повреждения в системных индексах не позволяют создать сессию для устранения повреждений
- `zero_damaged_pages` - повреждения в служебной области блока не даёт прочесть данные из этого блока. Параметр позволяет считать блок пустым, как будто в нем нет строк

Категория: "Для разработчиков"

Для примера рассмотрим параметры категории `Developer Options`.

Категория `Developer Options` включает в себя параметры, которые не должны использоваться в производственной базе данных. Однако некоторые из этих параметров могут быть использованы для восстановления содержимого таблиц, если в них повреждён блок и восстановление другими способами не привело к успеху (блок повреждён в физических репликах и бэкапах). Пример таких параметров:

`ignore_system_indexes`

Игнорировать системные индексы при чтении системных таблиц (но все же обновлять индексы при изменении таблиц). Может пригодиться, если повреждения в системных индексах не позволяют создать сессию для устранения повреждений.

`zero_damaged_pages`

Повреждения в служебной области блока (страницы) не дают прочесть данные на этой странице и выборка строк (командой SELECT) прервётся. Параметр позволяет пропустить содержимое страницы, считая, что в ней нет строк и продолжить работу с другими страницами. Это позволяет извлечь строки из неповрежденных страниц. Однако, на логическом уровне целостность данных может пострадать. Параметр не меняет содержимое страниц: они остаются поврежденными, а не заполняются нулями.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-developer.html

Категория: "Пользовательские настройки"

- Категория параметров `Customized Options`
- В категорию входят параметры расширений, библиотек или просто произвольные параметры, в названии которых присутствует точка
- разработчики библиотек и расширений дают возможность стандартным образом настраивать их функционал

```
postgres=# set myapp.par1 = '20';
SET
postgres=# show myapp.par1;
 myapp.par1
------------
 20
(1 row)
```


Категория: "Пользовательские настройки"

Расширения и библиотеки, подгружаемые параметрами **`local_preload_libraries, session_preload_libraries, shared_preload_libraries`** или командой **`LOAD`**, могут иметь свои параметры конфигурации. Эти параметры обрабатываются по логике обычных параметров. Однако, эти параметры неизвестны СУБД до тех пор, пока библиотека не подгрузится. В частности, СУБД не может проверить допустимость значений параметров, при их изменении командой `ALTER SYSTEM`. Поэтому, до загрузки библиотеки, эта команда не может устанавливать неизвестные СУБД параметры, даже если в названии параметра есть точка. Однако, на уровне сессии такие параметры можно устанавливать, они будут считаться параметрами приложения. По умолчанию, если в названии параметра присутствует точка, СУБД считает такие параметры customized options (можно перевести как "пользовательские настройки", "внесистемные параметры"). Разработчики расширений и библиотек в качестве префикса указывают название своего расширения и придумывают названия параметров. Также можно сохранять и произвольные названия параметров, если в названии присутствует точка в `postgresql.conf`. Как только библиотека подгружается (например, команда `LOAD`) и "регистрирует" программным вызовом свои параметры, СУБД проверяет значения параметров и если они недопустимы, то устанавливает их в значение по умолчанию которое указывает библиотека. Те параметры, которые библиотека не регистрировала при загрузке программным вызовом, удаляются из памяти, как будто их не устанавливали в файле конфигурации `postgresql.conf` и на других уровнях. Предупреждение об этом может быть записано в журнал кластера.

Имена параметров **без точки** в названии должны существовать в СУБД, использование несуществующего имени параметра (например, опечатка) в файле `postgresql.conf` не даст запустить кластер.

```
 waiting for server to start....
LOG:  unrecognized configuration parameter "myappparam1" in file
"/var/lib/postgresql/tantor-se-18/data/postgresql.conf" line 834
 FATAL:  configuration file "/var/lib/postgresql/tantor-se-18/data/postgresql.conf"
contains errors
```

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-custom.html

Названия и значения параметров конфигурации

- Названия параметров нечувствительны к регистру
- Значения параметров - один из пяти типов:
- boolean: значения можно указывать как `on, off, true, false, yes, no, 1, 0`
- строка: значения лучше задать в апострофах
- целое или десятичное число: в единицах измерения из столбца `unit` представления `pg_settings` или просто число
- `enum`: значение из списка в столбце `enumvals` представления pg_settings

Названия и значения параметров конфигурации

В начале главы мы рассматривали, что значения параметров могут быть нескольких типов. Рассмотрим подробнее. Типы параметров:

Логическое значение: значения можно задавать как `on, off, true, false, yes, no, 1, 0`

Строка: лучше использовать апострофы. Если внутри строки встречается символ апострофа, то поставить два апострофа вместо одного. Если строка содержит целое число, кавычки не обязательны

Целое или десятичное число: если целое число записывается в шестнадцатеричном виде (начинаются с 0x) нужно обрамить кавычками. Если вначале идёт ноль, то это целое значение в восьмеричном виде.

Число с единицей измерения: некоторые числовые параметры имеют неявную единицу измерения, так как они описывают объем памяти или времени. Если указать число без единицы измерения, то число может трактоваться как байт, килобайт, блок, миллисекунды, секунды, минуты. Единицу измерения можно узнать из столбца `unit` представления `pg_settings`. Удобно использовать в качестве суффикса единицу измерения. Её можно указывать сразу после числа или через один пробел. В любом случае, обязательно обрамлять значения апострофами.
Допустимые единицы измерения памяти (регистр важен):
`B` (байты), **`kB`** (килобайты), **`MB`** (мегабайты), **`GB`** (гигабайты) и **`TB`** (терабайты).
Допустимые значки для времени:
`us` (микросекунды), **`ms`** (миллисекунды), **`s`** (секунды), **`min`** (минуты), **`h`** (часы) и **`d`** (дни).

Enum: записываются так же, как и строковые параметры, но ограничены набором допустимых значений, нечувствительных к регистру. Список значений указан в столбце `enumvals` представления `pg_settings`.

Параметр конфигурации `transaction_timeout`

- позволяет отменить любую транзакцию, длительность которой превышает указанный период времени
- действие параметра распространяется как на явные транзакции (начатые с помощью команды BEGIN), так и на неявно начатые транзакции, соответствующие отдельной команде
- Значение ноль (по умолчанию) отключает таймаут
- Позволяет устанавливать ограничение на длительность транзакций
- `statement_timeout` позволяет установить максимальное время выполнения отдельной команды

Параметр конфигурации `transaction_timeout`

Рассмотрим примеры нескольких параметров конфигурации. Это поможет понять, как изменение значений параметров влияет на работу экземпляра.

`transaction_timeout` позволяет отменить любую транзакцию или одиночную команду, длительность которой превышает указанный период времени, а не только простаивающую. Действие параметра распространяется как на явные транзакции (начатые с помощью команды BEGIN), так и на неявно начатые транзакции, соответствующие отдельному оператору. Значение ноль (по умолчанию) отключает таймаут.

`statement_timeout` позволяет установить максимальное время выполнения отдельной команды. При превышении времени, команда прерывается. Время отсчитывается с момента получения серверным процессом команды и до завершения ее выполнения.

Транзакции и одиночные запросы (используют моментальный снимок) удерживают горизонт событий базы данных. Это не даёт вычищать старые версии строк. Параметры **`transaction_timeout`** и **`statement_timeout`** позволяют защититься удержания горизонта транзакциями и запросами.

Для защиты от простаивающих транзакций можно использовать **`idle_in_transaction_session_timeout`**. При превышении сессия разрывается:

```
postgres=*# commit;
ВАЖНО:  закрытие подключения из-за тайм-аута простоя в транзакции
сервер неожиданно закрыл соединение
        Скорее всего сервер прекратил работу из-за сбоя
        до или в процессе выполнения запроса.
Подключение к серверу потеряно. Попытка восстановления удачна.
```

Параметр `transaction_timeout` может быть установлен на уровне сессии, что позволяет использовать его для реализации логики, по которой по истечению времени результаты выполнения транзакций не актуальны и не нужны.

Параметр `transaction_timeout` появился PostgreSQL в 17 версии, в Tantor Postgres 15.4.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-resource.html

Автономные транзакции

- В Tantor Postgres SE и SE 1C есть автономные транзакции
- используются в подпрограммах на plpgsql
- позволяют выполнить команды и сохранить результат их выполнения независимо от того зафиксируется, откатится или прервётся транзакция, в которой была вызвана автономная транзакция

```
postgres=# create table t (a int);
CREATE TABLE
postgres=# create or replace
procedure p() LANGUAGE plpgsql
AS $$
DECLARE
 PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
 insert into t values (1);
END;
$$;
CREATE PROCEDURE
postgres=# begin transaction;
BEGIN
postgres=*# call p();
CALL
postgres=*# rollback;
ROLLBACK
postgres=# select * from t;
 a
---
 1
(1 row)
```


Автономные транзакции

Автономные транзакции можно реализовать через dblink к своей же базе, но проблема в производительности. Автономные транзакции в Tantor Postgres SE и SE 1C обеспечивают высокоскоростную реализацию автономных транзакций. Создается пул автономных сессий, обслуживающимися фоновыми рабочими процессами (background workers). Пул порождается при создании первой автономной транзакции. Серверные процессы выхватывают из пула сессию, передают на выполнение операторы автономной транзакции и возвращают соединение в пул. Ресурсы на порождение-остановку процессов, обслуживающих автономные транзакции, не расходуются. Серверный и фоновый процессы обмениваются данными синхронно через разделяемую память. Допускаются вложенные автономные транзакции. Для обслуживания вложенных автономных транзакций запускаются дополнительные (до сотни) фоновые процессы.

Пример как работает автономная транзакция:

```
create table t (a int);
create or replace function func() returns void
 LANGUAGE plpgsql
AS $$
DECLARE
 PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
 insert into t values (1);
END;
$$;
begin;
select func();
rollback;
select * from t;
```

Реализация автономных транзакций предложена Tantor Labs сообществу разработчиков PostgreSQL:
https://www.postgresql.org/message-id/f7470d5a-3cf1-4919-8404-5c4d91341a9f@tantorlabs.com

Параметр конфигурации transaction_buffers

- задаёт размер общей памяти, используемый для кэширования содержимого `PGDATA/pg_xact` - подкаталога со статусом фиксации транзакций
- Значение по умолчанию - 0, что равно размеру разделяемого пула буферов, деленному на 512 (shared_buffers/512)

Параметр конфигурации transaction_buffers

В PostgreSQL есть буфера в общей памяти экземпляра, которые называют "SLRU-буфера", так как используют Simple Least Recently Used алгоритм вытеснения из буфера. Начиная с версии 17 PostgreSQL (в Tantor Postgres, начиная с версии 15.4) размеры SLRU-кэшей можно задать параметрами конфигурации `commit_timestamp_buffers`, `multixact_member_buffers`, `multixact_offset_buffers`, `notify_buffers`, `serializable_buffers`, `subtransaction_buffers`, `transaction_buffers`.

Значения по умолчанию параметров `commit_timestamp_buffers`, `transaction_buffers`, `subtransaction_buffers` устанавливаются в зависимости от размера кэша буферов (значения параметра shared_buffers).

Параметр `transaction_buffers` задаёт размер общей памяти, используемой для кэширования содержимого `PGDATA/pg_xact` подкаталога о статусе фиксации транзакций. Значение по умолчанию - 0, что равно размеру размер разделяемого пула буферов, деленному на 512 (`shared_buffers/512`), но не менее 4 блоков. Изменение значения требует перезапуск экземпляра.

Кэширование помогает быстро определить статус транзакции. Потребность в определении статуса последних транзакций и вплоть до горизонта баз данных кластера, возникает у серверных процессов очень часто. Когда процессы видят версии изменившихся строк в блоках им часто нужно определить статус транзакции каждой обрабатываемой версии строки. Статус фиксации транзакции использует вакуум для поиска статуса транзакции при очистке старых версий строк. Статус фиксации использует два бита на транзакцию (зафиксирована `COMMIT` или явно отменена `ROLLBACK` или неявно - прервана). Если `autovacuum_freeze_max_age` установлено на максимально допустимое значение для 32-битных счетчиков транзакций в 2млрд., **размер `pg_xact` ожидается около полугигабайта**, а `pg_commit_ts` - около 20ГБ.

Недостатком увеличения значения `autovacuum_freeze_max_age` (а также `vacuum_freeze_table_age`) является то, что подкаталоги `pg_xact` и `pg_commit_ts` кластера баз данных будут занимать больше места. Значение по умолчанию в сборках, использующих 32-битный счетчик транзакций: 200 миллионов транзакций, соответствует примерно 50 МБ для хранения `pg_xact` и около 2 ГБ для хранения `pg_commit_ts`. Для 64-разрядных счетчиков значение по умочанию `autovacuum_freeze_max_age` 10млрд.

Сохраняются и статусы подтранзакций. При фиксации или откате транзакции верхнего уровня, статусы подтранзакций (два бита на каждую) также записываются в директорию `PGDATA/pg_xact`. При прерывании транзакции верхнего уровня, все её подтранзакции также прерываются.

Параметры multixact_members_buffers и multixact_offsets_buffers

- задают размер общей памяти, используемой для кеширования содержимого содержимого двух поддиректорий `PGDATA/pg_multixact`
- Убедиться, что в кластере используются 64-битные идентификаторы транзакций можно по значениям параметров:

```
\dconfig autovacuum_*age
        List of configuration parameters
             Parameter              |    Value
------------------------------------+-------------
 autovacuum_freeze_max_age          | 10000000000
 autovacuum_multixact_freeze_max_age | 20000000000
```

- если значения больше 4 миллиардов, то используются 64-битные счетчики транзакций и мультитранзакций

Параметры multixact_members_buffers и multixact_offsets_buffers

Экземпляры могут сталкиваться с деградацией производительности, если имеется большое число параллельно работающих транзакций, подтранзакций, было выполнено много мультитранзакций, транзакций на уровне SERIALIZABLE. Увеличение размеров буферов ("SLRU кэшей") помогают улучшить производительность.

Параметры PostgreSQL `multixact_offsets_buffers` и `multixact_members_buffers` задают размер разделяемой памяти, используемой для кэширования содержимого двух поддиректорий `PGDATA/pg_multixact`, которые хранят историю выполнившихся и выполняющихся мультитранзакций. История нужна для проверки статуса (не завершились, зафиксированы, прерваны) транзакций. Изменение значения параметров требуют перезапуск экземпляра.

Вакуумирование позволяет удалять старые файлы из подкаталогов `pg_multixact/members` и `pg_multixact/offsets`.

Поскольку в заголовке строки может храниться только один номер транзакции (поле "xmax"), в PostgreSQL используются мультитранзакции для поддержки блокировки строки одновременно несколькими транзакциями. Список транзакций, включенных в идентификатор мультитранзакции, хранится в директории `PGDATA/pg_multixact`.

В Tantor Postgres SE и SE 1C используются 64-битные идентификаторы транзакций, которые вряд ли дойдут до максимума своего значения. На уровне страницы проблема wrap around возможна, если какая-то сессия удерживает моментальный снимок, в котором накопилось более 4 миллиардов транзакций.

Проверить то, что в кластере используются 64-битные идентификаторы транзакций можно по значениям параметров:

```
\dconfig autovacuum_*age
        List of configuration parameters
             Parameter              |    Value
------------------------------------+-------------
 autovacuum_freeze_max_age          | 10000000000
 autovacuum_multixact_freeze_max_age | 20000000000
```

Показаны значения 10 миллиардов и 20 миллиардов, что больше чем 4 миллиарда, являющегося максимумом для 32-битных чисел.

Параметр конфигурации subtransaction_buffers

- задаёт размер общей памяти, используемой для кеширования содержимого `PGDATA/pg_subtrans`
- подтранзакция создается:
  - командой SAVEPOINT
  - если в блоке на языке plpgsql присутствует секция EXCEPTION
  - в psql неявно перед каждой командой в транзакции, если включён (interactive или on) параметр командой:
    `\set ON_ERROR_ROLLBACK  interactive`
- наличие секции EXCEPTION неявно устанавливает точку сохранения перед началом блока (перед BEGIN)

Параметр конфигурации subtransaction_buffers

`subtransaction_buffers` задаёт размер общей памяти, используемой для кэширования содержимого `PGDATA/pg_subtrans`.

Размер буфера можно посмотреть:

```
SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM
pg_shmem_allocations where name like '%btrans%';
       name       | allocated_size | pg_size_pretty
------------------+----------------+----------------
 subtransaction |         267520 | 261 kB
```

Подтранзакции могут явным образом запускаться как при помощи команды `SAVEPOINT`, так и другими способами, например посредством предложения `EXCEPTION` языка PL/pgSQL. То есть подтранзакции используются достаточно активно.

Идентификатор непосредственной родительской транзакции каждой подтранзакции записывается в каталог `pg_subtrans`. Идентификаторы транзакций верхнего уровня не записываются, поскольку у них нет родительской транзакции. Также не записываются и идентификаторы подтранзакций в режиме только для чтения.

Чем больше подтранзакций остаётся открытыми в каждой транзакции (в отношении которых не выполнен откат или освобождение), тем выше будут издержки. В общей памяти для каждого серверного процесса (backend) кэшируется по умолчанию до 64 открытых `subxid`. После превышения этого значения издержки на дисковый ввод-вывод существенно возрастают, так как приходится искать данные о `subxid` в `pg_subtrans`. Параметр `subtrans_buffers` позволяет этого избежать.

Команды `VACUUM`, `CREATE/DROP DATABASE`, `CREATE/DROP TABLESPACE` не могут выполняться в транзакции, так как неявно порождают транзакции:

```
postgres=# begin;
BEGIN
postgres=*# vacuum;
ERROR:  VACUUM cannot run inside a transaction block
```

Параметр конфигурации notify_buffers

- задаёт размер общей памяти, используемой для кеширования содержимого `PGDATA/pg_notify`
- Значение по умолчанию 8 блоков (64Кб)
- Используются в архитектуре NOTIFY/LISTEN обмена данными между процессами:

```
postgres=# listen abc;
LISTEN
postgres=# notify abc;
NOTIFY
Asynchronous notification "abc" received from
server process with PID 1234.
```


Параметр конфигурации notify_buffers

Параметр конфигурации `notify_buffers`, который задаёт размер общей памяти, используемой для кэширования содержимого `PGDATA/pg_notify`.

Используются в архитектуре `NOTIFY/LISTEN` обмена данными между процессами:

```
postgres=# listen abc;
LISTEN
postgres=# notify abc;
NOTIFY
Asynchronous notification "abc" received from server process with PID 1284.
```

Задание параметров при создании кластера

- переменными окружения и параметрами утилиты initdb
- параметры **initdb --lc-collate, --lc-ctype, --encoding** нельзя изменить после создания кластера
- значения параметров локализации можно посмотреть командой psql `\l`
- включить и отключить подсчет контрольных сумм можно после создания кластера
- проверить статус в psql **`\dconfig data_checksums`**
- проверка контрольных сумм блоков данных выполняется утилитами pg_checksums и pg_verifybackup

Задание параметров при создании кластера

Утилита создания кластера `initdb` имеет параметры (ключи), которые задают свойства создаваемого кластера. На `initdb` влияют и переменные окружения, установленные перед запуском утилиты. Параметры `initdb` перекрывают значения, заданные переменными окружения. Часть параметров нельзя изменить после создания кластера.

Часть параметров, заданных при создании кластера может меняться после его создания.

Параметр утилиты `initdb` **`-k`** или **`--data-checksums`** задает подсчет контрольных сумм в блоках файлов данных, находящихся в табличных пространствах. Начиная с 18 версии, по умолчанию, подсчёт контрольных сумм включён при создании кластера.

Включить, выключить или проверить контрольные суммы файлов можно утилитой `pg_checksums`, появившейся в 12 версии. Для проверки бэкапов используется `pg_verifybackup`. Узнать включены ли контрольные суммы на кластере можно утилитой `pg_controldata` или посмотреть значение параметра конфигурации (только для чтения) `data_checksum`.

```
pg_controldata -D . | grep checksum
Data page checksum version:           1
```

Ноль означает отключено. Отличное нуля значение - включено.

Не рекомендуется отключать проверку контрольных сумм. Если возникнет повреждение блока данных на диске при доступе к этому блоку процессы, в том числе процессы очистки не смогут продолжить работу. Это может привести к невозможности очистки и заморозки страниц.

Если в процессе перехода на новые мажорные версии СУБД требуется создавать кластер, то он должен быть создан с такими же параметрами, что и тот, который обновляют.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/locale.html

Разрешения на директорию PGDATA

- Параметр `initdb -g` или `--allow-group-access` устанавливает разрешения `0750 (rwx r-x ---)`
- если не указать, то пересоздавать кластер не нужно, можно поменять разрешения на PGDATA (и поддиректории) на допустимые значения
- Допустимые значения 0750 и 0700:
- **pg_ctl start**

```
waiting for server to start....
FATAL:  data directory
  "/var/lib/postgresql/tantor/se16/data" has invalid
  permissions
DETAIL:  Permissions should be u=rwx (0700) or u=rwx,g=rx
  (0750).
stopped waiting
```


Разрешения на директорию PGDATA

Разрешения на директорию `PGDATA` устанавливаются при создании кластера. Параметр `initdb -g` или `--allow-group-access` устанавливает разрешения 0750 (rwx r-x ---) на директорию и ее содержимое, позволяющее членам группы читать содержимое `PGDATA`, что может быть удобно для резервирования. После создания кластера можно вручную поменять разрешения на уровне файловой системы (`chmod -R 750 $PGDATA`) установив для `PGDATA` и ее поддиректорий маску `0750` или `0700`.

При запуске кластера проверяется, что на `PGDATA` установлены разрешения либо `0750`, либо `0700`. Если разрешения другие, то кластер не запустится:

```
pg_ctl start -D .
waiting for server to start....
FATAL:  data directory "/var/lib/postgresql/tantor-se-18/data" has invalid
permissions
DETAIL:  Permissions should be u=rwx (0700) or u=rwx,g=rx (0750).
 stopped waiting
```

Параметр `data_directory_mode` контекста `internal` показывает значение, с которым был запущен кластер:

```
select name, setting, min_val, max_val from pg_settings where context
='internal' and name like 'data_di%';
         name          | setting | min_val | max_val
-----------------------+---------+---------+---------
 data_directory_mode   | 0750    | 0       | 511
(1 row)
```

В PostgreSQL 10 версии было одно допустимое значение 0700. До 10 версии ограничений не было. В 11 версии добавили 0750.

Размер блока данных PostgreSQL

- размер 8192 байт = 8Кб задан при компиляции
- синоним “страница”
- при нахождении в памяти занимает буфер в буферном кэше

Размер блока данных PostgreSQL

По умолчанию, размер страницы (блока данных) составляет 8 килобайт (или 8192 байт). Размер блока данных задан при компиляции и не может быть изменён без перекомпиляции программного обеспечения.

Узнать размер блока можно командой:

```
pg_controldata | grep 'block size'
Database block size:                  8192
WAL block size:                       8192
```

или параметром конфигурации `block_size`

Размер блока данных определяет ограничения (лимиты) для многих характеристик кластера PostgreSQL.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/limits.html

Ограничения PostgreSQL

размер базы данных	не ограничен
количество баз данных в кластере	4,294,950,911
отношений в одной базе	1,431,650,303
размер отношения	32ТБ (блок 8КБ)
блоков в таблице	4,294,950,911
столбцов в таблице	1600
столбцов в выборке (SELECT)	1664
размер поля (в том числе text, bytea)	1ГБ
размер больших объектов (lo)	4ТБ
общий объем lo в базе	32ТБ (блок 8КБ)
длина идентификатора	63 байта
индексов на таблицу	не ограничено
размер строкового буфера	1Гб-1
столбцов в составном индексе	32

Ограничения PostgreSQL

Размер блока данных PostgreSQL может быть 16Кб, 32Кб. В настоящее время эмпирически (опытным путём) выбран 8Кб. Он определяется текущим развитием hardware (например, размерами кэшей). Внутренние алгоритмы работы, константы, параметры выбирались исходя из размера блока 8Кб. При изменении размера блока возможно появление узких мест под большой нагрузкой. Отношениями (синоним "класс") называются: **таблицы, индексы, последовательности, представления, внешние (foreign) таблицы, материализованные представления, составные типы**. Если объем хранимых данных в блоках таблицы будет превышать 32Тб, стоит использовать секционированные (partitioned) таблицы.

На таблицы TOAST ограничение тоже 32Тб, что **может ограничить** число строк в основной таблице. Более того, число полей, которые могут быть вынесены из версий строк в TOAST - не больше 4млрд (2 в степени 32). Это **может ограничить** число строк в таблице.

Размер блока влияет на максимальный размер отношения. Большие значения полей до 1Гб можно хранить в столбцах типа `text, varchar, bytea`. Это ограничение следует из того, что максимальный размер поля в TOAST таблице 1Гб.

Можно использовать устаревший тип данных `lo`. Все значения этого типа в одной базе данных хранятся в одной таблице системного каталога. Так как максимальный размер несекционированной таблицы 32Тб, то и максимальный объем lo в одной базе тоже 32Тб. Например, в одной базе можно хранить не больше 8 полей размером по 4Тб.

Количество столбцов, по которым можно создать индекс ограничено макросом `INDEX_MAX_KEYS`. Значение константы показывает параметр `max_index_keys`.

Также есть ограничение на количество параметров функций равное 100, но оно может быть увеличено до 600 (при размере блока 8Кб) перекомиляцией (макрос `FUNC_MAX_ARGS` в `pg_config_manual.h`).

Максимальный размер строкового буфера (`MaxAllocSize` в `stringinfo.c`) `0x3fffffff` = 1 Гигабайт минус 1 байт. При обработке строк (команды `SELECT *` и `COPY`) выделяется память под строковый буфер. Если размер обрабатываемых данных больше и буфер при очередном увеличении своего размера выходит за этот лимит, то выдаётся ошибка: "`Cannot enlarge string buffer`".

Параметром конфигурации Tantor Postgres `enable_large_allocations` и аналогичным параметром утилиты `pg_dump` можно увеличить размер строкового буфера до 2Гб.

Ограничения на длину идентификаторов

- Максимальная длина идентификаторов (например, имен таблиц, столбцов, индексов и т. д.) составляет 63 символа
- Например, вы можете создать таблицу с именем, содержащим до 63 символов:
- `CREATE TABLE my_really_long_table_name_with_63_characters (...);`
- Или столбец с именем, также содержащим до 63 символов:
- `ALTER TABLE my_table_name ADD COLUMN my_really_long_column_name_with_63_characters INTEGER;`
- идентификаторы превышающие этот размер усекаются, о чем выдается предупреждение

Ограничения на длину идентификаторов

Максимальная длина идентификаторов (имён таблиц, столбцов, индексов и т. д.) составляет 63 символа. Это означает, что идентификатор может содержать до 63 символов. Это стандартное ограничение и оно применяется ко всем идентификаторам в базе данных.
Например, вы можете создать таблицу с именем, содержащим до 63 символов:

```
CREATE TABLE my_really_long_table_name_with_63_characters (...);
```

Или столбец с именем, также содержащим до 63 символов:

```
ALTER TABLE my_table_name ADD COLUMN
my_really_long_column_name_with_63_characters INTEGER;
```

Это ограничение установлено для обеспечения совместимости с различными системами и для упрощения работы с базами данных.
Идентификаторы, превышающие 63 символа, усекаются, о чем выдается предупреждение

```
create table шестьдесяттрисимвола456789 (n numeric);
NOTICE:  identifier "шестьдесяттрисимвола456789" will be truncated to
"шестьдесяттрисимвола"
CREATE TABLE
\d ш*
Table "public.шестьдесяттрисимвола"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 n      | numeric |           |          |
```

К идентификаторам относятся названия отношений, столбцов. Идентификаторы можно заключать в кавычки. Если длинна идентификатора превышает 63 байта, то он усекается. Идентификаторы без кавычек должны начинаться с буквы.
Максимальная длинна идентификатора определяется макросом `NAMEDATALEN-1`, установленной при компиляции. Значение константы показывает параметр

```
postgres=# show max_identifier_length;
 max_identifier_length
-----------------------
 63
```

Есть и другие ограничения, например, максимальное количество аргументов функции 100, параметров в запросе 65535.

Конфигурационные параметры

- установлены при сборке и не меняются
- можно посмотреть:
  - утилитой командной строки pg_config
  - в представлении pg_config
- Параметр SHAREDIR определяет директорию с файлами расширений
- PKGLIBDIR указывает на директорию разделяемых библиотек
- BINDIR определяет директорию с исполняемыми файлами

```
postgres=# select * from pg_config where name in
('BINDIR','LIBDIR','PKGLIBDIR','SHAREDIR','SYSCONFDIR');
    name    |              setting
------------+-----------------------------------
 BINDIR     | /opt/tantor/db/18/bin
 LIBDIR     | /opt/tantor/db/18/lib
 PKGLIBDIR  | /opt/tantor/db/18/lib/postgresql
 SHAREDIR   | /opt/tantor/db/18/share/postgresql
 SYSCONFDIR | /opt/tantor/db/18/etc/postgresql
```


Конфигурационные параметры

"Конфигурационные параметры" (config) и "параметры конфигурации" (settings) созвучны, но это разные понятия.

Конфигурационные параметры устанавливаются при сборке (компиляции, линковке).

Посмотреть Конфигурационные параметры можно:

1) утилитой командной строки `pg_config`

2) в представлении **`select * from pg_config;`**

`SHAREDIR` определяет директорию с файлами расширений.

Управляющие файлы расширений лежат в поддиректории `extension` директории `SHAREDIR`.

Список управляющих файлов расширений:

`ls $(pg_config --sharedir)/extension/*.control`

`PKGLIBDIR` указывает на директорию разделяемых библиотек по умолчанию (файлы с расширением **.so**). Библиотеки могут загружаться командой `LOAD` в сессии или параметрами `shared_preload_libraries`, `session_preload_libraries` , `local_preload_libraries`.

Разработчик библиотеки определяет способ её загрузки.

`pg_config --pkglibdir`

`/opt/tantor/db/18/lib/postgresql`

`BINDIR` определяет директорию с исполняемыми файлами, которую добавляют в профиль:

`cat ~/.bash_profile`

`export PATH=/opt/tantor/db/18/bin:$PATH`

`PGSYSCONFDIR` указывает директорию, где находится файл служб подключений `pg_service.conf`

Если в файле служб создать описание службы, то можно будет им пользоваться:

`psql "service=описание_службы"`

В Oracle Database файл служб имеет аналог в виде файла tnsnames.ora. Файл pg_service.conf не востребован, он не используется JDBC-драйверами, только библиотекой libpq.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/libpq-pgservice.html

Демонстрация

- Просмотр параметров конфигурации

Демонстрация

Просмотр параметров конфигурации

Практика

1. Обзор параметров конфигурации
2. Параметры конфигурации с единицей измерения
3. Параметры конфигурации логического типа
4. Конфигурационные параметры
5. Файл служб

Практика

Обзор параметров конфигурации
Параметры конфигурации с единицей измерения
Параметры конфигурации логического типа
Конфигурационные параметры
Файл служб

4

4a

Логическая структура кластера

Кластер баз данных

- Кластер баз данных PostgreSQL - это набор баз данных
  - хранят свои файлы в директории PGDATA
  - обслуживаются экземпляром
- Приложение подсоединяется к одной базе данных и имеет доступ к её объектам
- Кластер PostgreSQL создаётся создаётся утилитой initdb
- Базы данных в кластере PostgreSQL можно создавать и удалять
- Кластер Patroni - это несколько кластеров PostgreSQL

Кластер баз данных

База данных - логическое место хранения объектов приложений, с которыми можно работать одновременно (например, соединять таблицы в одной выборке).

Кластер (объединение) баз данных PostgreSQL или сокращённо "кластер" создаётся утилитой командной строки initdb. Эта утилита создаст набор файлов и директорий в физическом месте хранения - директории путь к которой указывается в параметрах initdb. Эту директорию называют PGDATA. В PGDATA хранится кластер баз данных.

Чтобы приложения могли подсоединиться к какой-нибудь базе данных кластера нужно, чтобы на хосте был запущен "экземпляр" - набор серверных, фоновых (вспомогательных) процессов и основной процесс postgres (он же postmaster). Изначально создаётся три базы данных, позже, запустив кластер, можно командой CREATE DATABASE создать базу данных в кластере. Подсоединяться при этом можно к любой базе данных кластера, база данных будет создана и будет равной среди всех остальных баз данных кластера.

Если вы работали с Oracle Database, то аналог базы данных PostgreSQL это Pluggable Database (PDB). Аналог кластера - multitenant container database. Корневой базы (CDB Root) в PostgreSQL нет, для управления кластером PostgreSQL можно подключиться к любой из баз данных. Аналог Oracle Seed PDB - базы данных template0 и template1. В Tantor Polar несколько экземпляров обслуживают один кластер баз данных - аналог Oracle Real Application Cluster (RAC). В Tantor Polar один экземпляр читает-пишет, остальные экземпляры только для чтения.

Кластер Patroni - это набор кластеров PostgreSQL. Один из них - primary (основной, мастер, читает-пишет), остальные - standby (реплики, только читают). Patroni может работать с экземплярами Tantor Polar.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/glossary.html

Экземпляр

- Экземпляр это набор процессов и используемой ими памяти, обслуживающие один кластер баз данных
- Кластер баз данных обслуживается только одним экземпляром
- Первый и основной процесс экземпляра называется postgres или postmaster
- Необходимый параметр экземпляра - порт (port), по которому postmaster просушивает входящие соединения по протоколам TCP и локально
- По умолчанию параметр конфигурации port имеет значение 5432

Экземпляр

Экземпляр (instance) кластера баз данных - набор процессов и используемой ими памяти (общей для них и локальной для каждого из этих процессов) посредством которых приложения подсоединяются (создают сессии) к базам данных. Базам данных кластера, так как один экземпляр обслуживает ровно один кластер. В кластере же можно создавать и удалять базы данных. Экземпляр то же самое, что экземпляр single-instance Oracle Database.

Экземпляр состоит из процесса postgres (postmaster), серверных (обслуживающих, backend, foreground), вспомогательных (фоновых) процессов, которые используют разделяемую память чтобы обмениваться данными между собой. На одном хосте могут работать несколько экземпляров СУБД, если не будет конфликта по номеру порта, в том числе в имени файла Unix-сокета.

Порт - это число, которое устанавливается в параметре конфигурации port. По умолчанию 5432. Порт используется в имени Unix-сокета (файл) и как номер TCP портов сетевых интерфейсов (IP-адресов), которые перечислены в параметре local_addresses. Значение по умолчанию localhost. * - все адреса IPV4 и IPV6, '0.0.0.0' - все адреса IPv4, '::' - все адреса IPv6. Но можно задать список имён и/или числовых IP-адресов узлов, разделённых запятыми. Пустая строка означает, что подключиться к экземпляру можно будет только через Unix-сокет.

Процесс экземпляра postgres прослушивает этот порт. В Oracle Database этим занимаются процессы прослушиватели (listeners), не принадлежащие экземплярам.

Экземпляр через свои процессы реализует все функциональные возможности СУБД: читает и записывает файлы, работает с общей памятью, обеспечивает свойства ACID транзакций, принимает подключения клиентских процессов, проверяет права доступа, выполняет восстановление после сбоя, осуществляет репликацию и прочие задачи.

Приложение подсоединяется через сокет к своему серверному процессу. Фоновые процессы соединениями с приложениями не связаны и выполняют общую полезную работу.

Примечание: Название postmaster используется для обозначения основного процесса экземпляра, так как слово PostgreSQL может обозначать много понятий, например семейство СУБД, к которому относится Tantor Postgres. Tantor Postgres - ответвление (fork) свободно распространяемого PostgreSQL.

База данных

- База данных - логическое место хранения объектов SQL
- База данных - часть кластера
- Содержимое базы данных может быть выгружено на логическом уровне (в виде команд SQL) и загружено в другую базу данных этого или другого кластера
- Приложение создаёт сессию (session) с одной базой данных
- В сессии есть доступ к объектам одной базы данных
- Доступа к объектам других баз данных кластера в одной сессии нет

База данных

Приложение хранит данные в СУБД и получает к ним доступ через соединение (connection) с серверным процессом экземпляра. В рамках соединения (локального через Unix-сокет или сетевой TCP-сокет) создается сеанс. Сеанс, соединение, сессия, подключение часто (в документации somtimes) используются как синонимы, потому что для приложения важно давать команды SQL и получать результат. Отличия соединений от сессий играют роль при настройке балансировщиков нагрузки (например, приложение pgBouncer) и сетевых настроек. Соединение - физическое понятие, сессия - логическое.

Создав соединение, приложение должно иметь доступ ко всем своим объектам. Например, иметь возможность соединять выборки из нескольких таблиц и использовать свои хранимые функции. Поэтому все (за небольшим исключением) объекты хранения, которые использует приложение локальны для базы данных, хранятся в ней.

Соединение устанавливается только с одной базой данных кластера. Данные, хранящиеся в разных базах данных кластера,/ изолированы друг от друга исходя из того, что предназначены обычно для использования разными приложениями, а приложения не должны пересекаться друг с другом в том числе с точки зрения разграничения доступа.

Идеи изоляции приложений с помощью баз данных и объединения объектов приложения в одной базе данных технически можно обойти, так как потребности у приложений разные. Например, используя расширения (видов fdw, dblink) приложение в своей сессии может работать с данными в нескольких базах данных. А несколько приложений используя схемы и роли могут хранить таблицы с одинаковыми именами в одной базе данных не мешая друг другу.

Список баз данных

- Получение списка это команда psql:
  `\l` или `\l+`

```
postgres=# \l
                                 List of databases
   Name    |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    |
-----------+----------+----------+-----------------+-------------+-------------+
 postgres  | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
 template0 | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
 template1 | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
 testdb    | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |
(4 rows)

postgres=# SELECT datname FROM pg_database;
  datname
-----------
 postgres
 template1
 template0
 testdb
```


Список баз данных

Изначально после создания кластера есть три базы данных с именами `postgres, template0, template1`. К базе `template0` нельзя подсоединиться, она не предназначена для внесения в неё изменений. Список баз можно получить:

командами psql **`\l`** или **`\l+`**

командой **`SELECT datname FROM pg_database;`**

Создание базы данных

- команда CREATE DATABASE или утилита createdb
- два режима создания базы: WAL_LOG и FILE_COPY
- создание базы это клонирование другой базы со всем содержимым
- у базы есть роль-владелец (OWNER)
- владельца и имя базы можно сменить после её создания
- можно создать базу со своими параметрами локализации (кодировкой)
- кодировку после создания базы изменить нельзя
- свойство базы "шаблонная" (IS_TEMPLATE) влияет на возможность ее клонирования непривилегированной ролью и удаление
- Функция pg_import_system_collations('pg_catalog') добавляет параметры локализации, появившиеся в linux,после создания кластера

Создание базы данных

Базу данных может создать роль с атрибутом `SUPERUSER` или `CREATEDB`:

```
CREATE DATABASE имя_базы параметр=значение параметр=значение;
```

У команды есть утилита-обёртка createdb, она удобна если нужно создавать базы данных из командной строки.

У команды имеется 17 параметров в 18 версии. Основные параметры:

`OWNER` - имя пользователя, который будет обладать привилегиями, схожими с суперпользователем в сессиях к этой базе. По умолчанию создатель становится владельцем базы.

`TEMPLATE` - имя базы, которую склонируете. Это любая база, не обязательно имеющая свойство `IS_TEMPLATE`. По умолчанию, используется база `template1`. Но если вы захотите создать базу с параметрами локализации, отличными от тех, что у `template1`, нужно использовать `template0` (unmodifiable empty database).

`IS_TEMPATE` - можно менять после создания базы Если `IS_TEMPATE=true`, эту базу сможет клонировать любой пользователь с атрибутом `CREATEDB`; в противном случае (по умолчанию), клонировать эту базу смогут только суперпользователи и её владельцы. Также база со свойством шаблона не может быть удалена. Для удаления нужно убрать свойство шаблона.

Кодировка и классификация символов связаны с типом сортировки. Для создания базы данных с кодировкой, отличной от кодировки, с которой был создан кластер может потребоваться указать четыре параметра:

```
create database имя_базы LC_COLLATE='ru_RU.iso88595' LC_CTYPE='ru_RU.iso88595'
ENCODING='ISO_8859_5' TEMPLATE=template0;
```

Доступные к использованию типы сортировок можно посмотреть в таблице `pg_collation`.

Типы сортировок `"C"` и `"POSIX"` совместимы со всеми кодировками. Не нужно их использовать, так как порядок сортировки кириллических символов не соответствует лингвистическим правилам.

Доступные к использованию параметры локализации, определяются в момент создания кластера, сохраняются в этой таблице, и после создания кластера, могут дополняться вызовом функции `select pg_import_system_collations('pg_catalog');`.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-createdatabase.html

Изменение свойств базы данных

- Командой ALTER DATABASE имя можно поменять некоторые свойства существующей базы данных
- Часть свойств можно поменять только, если нет ни одной сессии к базе данных
- Командой ALTER DATBASE имя SET можно поменять конфигурационные параметры сессий, создаваемых с этой базой
- на уровне базы данных можно установить около 190 параметров конфигурации
- база данных - общий объект кластера и менять её свойства можно, подсоединившись к любой базе данных в кластере

Изменение свойств базы данных

`STRATEGY` - обратите внимание на этот параметр, если вы создаете базу на промышленном кластере или база, которую вы используете в качестве шаблона (которую клонируете), имеет большой размер. Параметр появился в 14 версии PostgreSQL и сразу, по умолчанию, стал использовать новую стратегию `WAL_LOG`, при которой составляется список объектов и весь объем клонируемой базы проходит через WAL. Причина появления новой стратегии в том, что прежняя стратегия (`FILE_COPY`) **выполняла контрольную точку**, потом копировала директории (журналируя только команды копирования), потом **вторую контрольную точку**. Если шаблон маленького размера, то первая контрольная точка на промышленном кластере даёт повышенную нагрузку (вторая несущественную). Причина не только в одномоментном и косвенном возрастании объема записи (грязный блок записывается по контрольной точке но меняется и реже будет записан второй, а мог бы один раз если бы не было контрольной точки) нагрузке на ввод-вывод (систему хранения, диски), а в том, что после каждой контрольной точки, каждый изменяющийся блок записывается в журнал полностью (8Кб), так как, по умолчанию, параметр `full_page_writes=on` (а отключать его небезопасно). Если размер шаблонной базы больше значения параметра `max_wal_size`, то контрольная точка тоже будет выполнена (может быть даже неоднократно, если размер базы больше значения параметра `max_wal_size` в несколько раз).

Если размер шаблона небольшой, например, меньше 16Мб (размер WAL-сегмента), или в несколько раз больше, то можно создавать базу данных со стратегией по умолчанию. Если же больше, то стоит выбрать время когда кластер наименее нагружен. Если есть реплики, то оценить пропускную способность сети и, возможно, указать стратегию `FILE_COPY`. Если размер шаблона больше половины от `max_wal_size`, то `FILE_COPY` предпочтительнее.

Можно дать описание (comment) базе данных. Описания, к почти любым объектам, можно давать командой:

```
comment on database db1 is 'Database for my purpose';
```

Описание можно посмотреть командой **\l+**

Описания на функционал не влияют.

Параметры конфигурации на уровне базы данных (`ALTER DATABASE`) и разрешения на уровне базы данных (`GRANT`) из шаблонной базы данных в клон не переносятся.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-alterdatabase.html

Команда ALTER DATABASE

- Синтаксис команды по которому видно какие свойства можно поменять:
- ```
  ALTER DATABASE имя_базы
    ALLOW_CONNECTIONS true|false
    CONNECTION LIMIT -1/число
    IS_TEMPLATE true|false
    RENAME TO новое
    OWNER TO новый
    SET TABLESPACE новое
    REFRESH COLLATION VERSION
    SET параметр=значение
  ```


Команда ALTER DATABASE

Изменить свойства базы данных можно командой `ALTER DATABASE`. Пример:
`alter database имя is_template=true;`
`alter database имя SET имя=значение;` меняет один из около 190 параметров сессий, которые можно установить на уровне базы данных.

Переименовать базу данных может владелец с атрибутом `CREATEDB` или `SUPERUSER`. Переименовать базу, к которой есть соединения, нельзя, нужно сначала подключиться к любой другой базе и переименовывать.

Можно поменять табличное пространство по умолчанию, но к базе никто не должен быть подсоединён и на уровне файловой системы будут перемещаться все файлы (кроме тех которые находятся в других табличных пространствах) и файлы объектов системного каталога.

Можно поменять владельца базы данных.

Можно установить параметры конфигурации, чтобы настроить поведение процессов (и фоновых и обслуживающих сессии), работающих с объектами этой базы данных.

Параметры локализации можно выбрать при создании базы данных, после создания базы данных изменить нельзя. Основное это кодировка (`encoding`) и значения `collation` (правила сортировки), `ctype` (классификация символов), которые связаны со значением `encoding`, провайдер локализации (`libc, icu, builtin`). Часть параметров локализации относятся к сессии и их можно поменять командой `ALTER DATABASE SET`.

Создание базы данных и изменение табличного пространства нетранзакционны (нельзя выполнять в рамках транзакции). Нетранзакционные команды не могут выполняться при установке расширения командой `create extension`.
  ```

Удаление базы данных

- Выполняется командой DROP DATABASE или утилитой-оберткой dropdb
- Для выполнения команды нужно подсоединиться к любой базе кластера, отличной от удаляемой
- База данных удаляется со всем содержимым
- Команда нетранзакционна - откатить её нельзя
- Физически удаляются файлы в которых хранятся данные объектов базы

Удаление базы данных

Если содержимое базы данных не нужно, то базу данных можно удалить.

При удалении локальные объекты в других базах данных не затрагиваются. Команда удаления:

```
DROP DATABASE [IF EXISTS] имя;
```

В квадратных скобках опциональные ключевые слова.

`IF EXISTS` (если существует) есть во многих командах и позволяет не генерировать ошибку (уровень важности `ERROR`), если объекта нет, но обычно сообщает (уровень важности `NOTICE`), что такого объекта нет. Уровни важности влияют на то, как будет обработано сообщение: выдано клиенту, передано в журнал сообщений кластера.

Следующая команда:

```
DROP DATABASE имя (FORCE);
```

позволяет отсоединить сессии, которые соединены с этой базой, прервать их транзакции и удалить базу.

Базу данных со свойством `IS_TEMPLATE` (шаблона) можно удалить, если убрать свойство шаблона.

Не стоит удалять базу `template0`.

Схемы в базе данных

- синоним схемы - пространство имён (namespace)
- используются для упорядочивания хранения объектов базы данных
- Схема - локальный объект базы данных
- Схемы позволяют иметь несколько таблиц и других типов объектов с одинаковыми именами в одной и той же базе данных
- Объект не может находиться в нескольких схемах
- Схемы могут использоваться для логики объединения (пакетирования) подпрограмм
- Объектов "пакеты" в PostgreSQL нет

Схемы в базе данных

Синоним схемы - пространство имён (namespace).

Схемы используются для упорядочивания хранения объектов базы данных. Аналогия: файлы в файловой системе могут располагаться в разных директориях. Так же и таблицы, представления, подпрограммы могут располагаться в разных схемах одной и той же базы данных.

Схема - **локальный** объект базы данных, то есть в каждой базе данных кластера свой набор схем. В разных базах данных могут существовать схемы с одинаковыми именами (идентификаторами).

Схемы позволяют иметь несколько таблиц и других типов объектов с одинаковыми именами в одной и той же базе данных.

Схемы позволяют объединять подпрограммы (процедуры и функции), логически связанные друг с другом.

Большинство объектов, с которыми работают приложения, должны принадлежать какой-либо одной схеме. Такие объекты не могут существовать без схемы. Перед удалением схемы нужно будет переназначить объекты в другую схему. Объект не может находиться в нескольких схемах одновременно. символических и жестких ссылок, как в файловой системе, нет.

При обращении к таким объектам можно указывать перед именем объекта схему и символ точки. Например:

```
SELECT схема.функция();
```

или

```
SELECT * FROM схема.таблица;
```

В Oracle Database есть объекты пакет и тело пакета. В PostgreSQL таких объектов нет. Схемы могут использоваться для обеспечения основного функционала пакетов - возможности объединять схожие по логике подпрограммы в модули (пакеты). Использование расширений, реализующих пакеты путем добавления команд `create package` приводит к созданию непереносимого (на другие СУБД семейства PostgreSQL) кода.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/ddl-schemas.html

Создание и изменение схем

- Схемы не связаны с пользователями, но имеют владельца
- При создании схемы можно назначить пользователя-владельца:
  `CREATE SCHEMA имя_схемы AUTHORIZATION владелец;`
- Можно сменить владельца:
  `ALTER SCHEMA имя_схемы OWNER TO пользователь;`
- Можно переименовать схему:
  `ALTER SCHEMA имя_схемы RENAME TO имя;`
- На схемы можно выдавать привилегии CREATE, USAGE ролям

Создание и изменение схем

Схемы не связаны с ролями. Имена владельца объекта и имя схемы (в которой находится объект) могут быть разными и их можно менять после создания объекта.

Схемы имеют владельца. При создании схемы его можно установить:

```
CREATE SCHEMA имя_схемы AUTHORIZATION владелец;
```

а позже поменять:

```
ALTER SCHEMA имя_схемы OWNER TO пользователь;
```

Можно переименовать схему, но стоит помнить о пути поиска, в значении которого вероятно нужно будет отразить новое имя схемы.

В Oracle Database схемы и пользователи связаны друг с другом, что ограничивает гибкость. В Oracle Database по этой причине есть объекты "синонимы", в PostgreSQL аналогов "синонимов" нет, так как они не нужны.

На схемы можно выдавать пользователям привилегии `CREATE` и/или `USAGE`. Тем самым управлять "видимостью" объектов в схеме как единым целым. Аналогия: в файловой системе может быть привилегия доступа к файлу, но если нет привилегии на директорию, в которой расположен файл, то доступа к файлу не будет.

Схемы можно удалять:

```
DROP SCHEMA [ IF EXISTS ] имя [CASCADE] ;
```

Если в схеме расположены объекты, по умолчанию схема не удалится. Если объекты нужны их стоит перенести в другую схему. Если объекты не нужны, их можно удалить вместе со схемой, используя опцию `CASCADE`.

Путь поиска объектов в схемах

- задаёт список схем, в которых ищется объект
- устанавливается параметром `search_path`, который может быть изменён на любом уровне
- значение по умолчанию `"$user", public`

```
postgres=# show search_path;
   search_path
-----------------
 "$user", public
postgres=# select current_schema , current_schema(),
current_schemas(true), current_schemas(false);
 current_schema | current_schema |   current_schemas    | current_schemas
----------------+----------------+----------------------+----------------
 public         | public         | {pg_catalog,public} | {public}
(1 row)
```


Путь поиска объектов в схемах

С объектами схемы связано понятие путь поиска и соответствующий параметр конфигурации `search_path`. Этот параметр установлен на уровне кластера и имеет значение по умолчанию `"$user", public`

`$user` - подставляется имя роли, в которой текущий момент работает сессия.

Этот параметр может быть установлен на любых уровнях и меняться любой ролью.

В файловых системах есть аналог - переменная окружения `PATH`.

В пути поиска можно указать несколько схем, в которых будет искаться объект, если перед именем объекта явно не указано имя схемы. Объект ищется по порядку схем, чьи имена перечислены в пути поиска. Если схема не существует или на неё нет прав, то объект ищется в схемах, перечисленных дальше, и никаких ошибок не выдаётся. Алгоритм поиска аналогичен поиску файлов в файловой системе.

В шаблонных базах есть схема с именем `public`, поэтому при создании любых баз данных, схема с именем public будет существовать. Схема `public` указана в пути поиска: `"$user", public`.

Логика использования пути поиска, обычно, выбирается заранее и впоследствии значение параметра `search_path` на уровне кластера или базы данных не меняется потому, что изменение пути поиска может привести к тому, что объекты в подпрограммах перестанут находиться.

Значение по умолчанию позволяет создавать схемы с таким же именем, как и имена ролей и это удобно. При этом нужно не забывать, что схема - локальный объект базы данных, а роль - общий для всего кластера. Если роль имеет право соединяться с несколькими базами в кластере, то в каждой из них можно будет создать одноимённую схему.

Специальные схемы

- **pg_catalog** - в этой схеме расположены объекты "системного каталога"
- **information_schema** - схема, описанная в стандарте SQL
- **pg_toast** - схема для особых таблиц TOAST
- **pg_temp** (ссылка на pg_tempN, где N число) - схема для временных таблиц
- **pg_toast_temp** (ссылка на pg_toast_tempN, где N число) - схема для временных TOAST таблиц на временные таблицы

```
postgres=# \dnS
            List of schemas
        Name         |       Owner
---------------------+-------------------
 information_schema | postgres
 pg_catalog         | postgres
 pg_toast           | postgres
 public             | pg_database_owner
```


Специальные схемы

В PostgreSQL существуют служебные схемы:

`pg_catalog` - в этой схеме расположены объекты "системного каталога" - служебные таблицы, представления, функции и другие объекты

`information_schema` - схема, описанная в стандарте SQL. Содержит таблицы со стандартизованными именами и названиями столбцов. Разработчики стандарта полагали что производители СУБД создадут эту схему и таблицы, что позволит разработчикам одной и той же командой `SELECT` получать данные работая с СУБД разных производителей. Распространения эта идея не получила, так как информация из стандартизованных таблиц не сильно востребована при разработке, а также потому, что спецификации интерфейсов доступа JDBC содержат методы, позволяющие стандартным образом независимо от используемой СУБД получить гораздо более полезную информацию о СУБД и объектах в ней.

Есть схемы для специфических типов таблиц, которые определены исходя из принципа, что у таблиц должна быть схема (таблицы должны располагаться в какой-то схеме):

`pg_toast` - схема для особых таблиц TOAST, которые используются для хранения полей большого размера. Эти таблицы стараются скрывать, чтобы они не создавали "информационный шум". В этих целях TOAST-таблицы (и их индексов) создаются в этой служебной схеме. Об этой схеме можно знать на случай если её имя где-то встретится. Работа с TOAST полностью автоматизирована и отдельных команд для работы с TOAST-объектами и схемой нет. Для изменения свойств, касающихся TOAST используются опции команд `CREATE TABLE` и `ALTER TABLE` для обычных таблиц.

`pg_toast_temp` (ссылка на `pg_toast_tempN`, где `N` число) - схема для временных TOAST таблиц (и индексов) к временным таблицам. Существуют не дольше жизни сессии.

`pg_temp` (ссылка на `pg_tempN`, где `N` число) - схема для временных таблиц. Временные таблицы, индексы, представления (их определения и данные) существуют либо до конца транзакции, либо до конца сессии. Неявно присутствует в начале пути поиска.

Практический смысл имеет знание о схеме `pg_catalog`. Имя этой схемы можно использовать в командах psql для поиска служебных таблиц, представлений и функций.

Знания о временных объектах нужны для разработчиков и администраторов, если они сталкиваются с наличием большого количества файлов временных объектов. Использование Tantor Postgres SE и SE 1C позволяет уменьшить проблемы при работе большим числом временных объектов.

Определение текущего пути поиска

- командой psql `SHOW search_path;`
  - выдает установленный для этого места путь поиска
- функцией `current_schemas(false)`
  - выдает действующий в этом месте путь поиска в виде массива
- функция `current_schemas(true)`
  - добавляет служебные схемы, а именно pg_catalog и pg_temp_N
- функция `current_schema` или `current_schema()`
  - выдает одно имя первой по порядку схемы в пути поиска или NULL, если путь поиска пустой

Определение текущего пути поиска

Текущий путь поиска можно получить:

командой psql `SHOW search_path;` выдает установленный для этого места путь поиска в виде строки. Разделители - запятые.

функцией `current_schemas(false)` - выдает действующий в этом месте путь поиска в виде массива. При этом, в отличие от `search_path` не выдаёт несуществующие схемы, только конкретные имена существующих схем. Функцию удобно использовать в хранимых подпрограммах.

`current_schemas(true)` - добавляет служебные схемы, а именно `pg_catalog` и `pg_temp_N` (если она была автоматически создана в сессии) если он неявно присутствует в пути поиска. Схемы для TOAST не выдаёт, так сделано. Этот вариант функции используется для определения того, будет ли искаться имя объекта сначала в схеме системного каталога. Например, ищется функция или таблица имя которой начинается с "`pg_`" (так начинаются названия всех объектов системного каталога), пользовательские объекты по соглашению (code conventions) которого обычно придерживаются разработчики приложений не должны иметь имена начинающиеся на "`pg_`". Можно поменять путь поиска так, чтобы `pg_catalog` шел в списке не первым, но это не имеет смысла и не практикуется.

функцией `current_schema` или `current_schema()`.

Примечание. В PostgreSQL после имени функции без аргументов обязательно ставить круглые скобки "`()`". Однако для части функций, описанных в стандарте SQL, к которым относится эта функция `current_schema` , их ставить не обязательно, потому что в стандарте SQL круглые скобки не указаны. Эта функция выдает одно имя первой по порядку схемы в пути поиска (`search_path`) или `NULL`, если путь поиска пустой. В этой схеме будут создаваться пользовательские объекты, если в команде создания не указать явно имя схемы. Если функция выдает `NULL`, то объект не будет создан без указания схемы.

В какой схеме будет создан объект

- имя этой схемы для обычных объектов выдает функция `current_schema()`
- временные объекты создаются в служебных схемах
- служебные схемы для временных объектов создаются автоматически

В какой схеме будет создан объект

Для определения схемы, в которой будет создан объект используется путь поиска, действующий в данном месте исполнения команды. Имя этой схемы для обычных объектов выдает функция `current_schema()`. Однако, если объект "необычный" (временный), то используются схемы, в которых могут располагаться объекты этого специфического типа. Это относится к временным таблицам, индексам на временные таблицы, временным представлениям, TOAST таблицам к временным таблицам. В этом случае если схема отсутствует, то она будет создана (или назначена из созданных ранее и неиспользуемая другими сессиями) - ей будет назначен номер и он будет использоваться как суффикс в имени схемы. В этом случае имя такой служебной схемы будет неявно существовать в пути поиска. Соответственно такие объекты неявно будут искаться и префиксировать их имена именем служебной схемы не нужно.

Таким образом, создание временной таблицы приводит к добавлению строк в таблицы системного каталога. При массированном порождении временных объектов таблицы и индексы системного каталога, а также файловая система могут стать узким местом. После окончания сессии объекты временных схем удаляются, а сама схема остается для повторного использования другими сессиями, чтобы не порождать частое удаление строк в таблице системного каталога `pg_namespace`.

Если нужно явно задать место в пути поиска для служебных схем, то можно указывать названия `pg_catalog` и `pg_temp` в нужном порядке среди обычных схем. Этот порядок и будет использоваться. Однако, лучше не допускать перекрытия имён объектов и делать имена уникальными, чтобы не приходилось прибегать к изменению пути поиска.

Путь поиска в подпрограммах SECURITY DEFINER

- при использовании $user в пути поиска в теле подпрограммы будет подставлено имя владельца подпрограммы
- перед вызовом такой подпрограммы вызывающий ее может поменять значение в пути поиска и убрать $user, в этом случае в теле подпрограммы будет использоваться значение установленное вызывающим и подпрограмма может начать работать с другими объектами
- На уровне любой (INVOKER и DEFINER) подпрограммы можно установить значение для параметров конфигурации, которые могут меняться на уровне сессии

Путь поиска в подпрограммах SECURITY DEFINER

В подпрограммах со свойством `SECURITY DEFINER` есть особенность с путем поиска. Например, с переменной подстановки `$user`. В теле подпрограмм (процедур и функций) используются права владельца `(DEFINER)`. Функция user в таких подпрограммах выдаёт имя владельца. В пути поиска с переменной подстановки будет имя владельца. Так как `$user` присутствует в значении по умолчанию, обычно создатель такой подпрограммы тестирует подпрограмму с этим значением `search_path`.

Если вызывающий подпрограмму поменяет перед вызовом подпрограммы `search_path` в своей сессии или транзакции, то именно такое значения и будет действовать в теле подпрограммы. Видимость объектов может поменяться.

Чтобы не зависеть от такого изменения параметра `search_path` его можно установить принудительно в свойствах подпрограммы:

```
CREATE FUNCTION имя(параметры)
 RETURNS тип
 LANGUAGE язык
 SET  search_path TO 'значение'
 SECURITY {DEFINER | INVOKER}
AS
BEGIN
END;
```

Если ставить `SET` внутри блока `BEGIN` и `END` ошибки не будет, но поведение будет другим - установленное значение останется после выхода из подпрограммы, а если произошел откат транзакции (даже неявно при наличии в подпрограмме секции `EXCEPTION`) изменение значения параметра отменится. Это создаёт неоднозначность и порождает трудно выявляемые ошибки.

На уровне любой (`INVOKER` и `DEFINER`) подпрограммы можно установить значение для параметра конфигурации которые допускают изменение значения на уровне сессии (контекст **user**, **superuser**).

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-createfunction.html

Маскировка объектов схем

- Если в командах перед именем объектов не используется имя схемы, добавление в путь поиска имея схемы, где создан объект с тем же именем маскирует исходный объект
- По умолчанию, временная схема и схема системного каталога неявно присутствуют в начале пути поиска и создание временного объекта маскирует любую таблицу, представление, последовательность

```
postgres=# \dt pg_authid
              List of tables
   Schema   |   Name    | Type  |  Owner
------------+-----------+-------+---------
 pg_catalog | pg_authid | table | postgres
(1 row)

postgres=# create temporary table
pg_authid(t text);
CREATE TABLE
postgres=# select * from pg_authid;
 t
---
(0 rows)

postgres=# select current_schemas(true);
        current_schemas
-------------------------------
 {pg_temp_2,pg_catalog,public}
(1 row)
```


Маскировка объектов схем

В документации написано: "Для обеспечения безопасности `search_path` должен быть настроен таким образом, чтобы исключить любые схемы, доступные для записи ненадежными пользователями. Это предотвращает возможность создания злонамеренными пользователями объектов (например, таблиц, функций и операторов), которые могут **замаскировать** объекты, предназначенные для использования функцией. Особенно важной в этом отношении является временная схема, которая по умолчанию ищется первой и обычно доступна для записи всем. Безопасное расположение может быть достигнуто путем принудительного поиска временной схемы в конце. Для этого напишите `pg_temp` как последний элемент в `search_path`."

Другими словами, чтобы подпрограмма с тегом `DEFINER` была безопасна, `search_path` должен:

1) быть установлен на уровне определения подпрограммы

2) исключать любые схемы, доступные для создания или изменения пользователям с меньшим уровнем привилегий, чем у владельца такой подпрограммы

3) схема `pg_temp` должна быть указана явно в конце пути поиска.

По умолчанию, после создания подпрограммы роль PUBLIC получает право выполнять подпрограмму. Это поведение можно изменить, используя привилегии по умолчанию (`default privileges`).

Объекты системного каталога, в том числе функции, можно замаскировать явно указав схему `pg_catalog` в пути поиска после схемы с маскирующим объектом. Например:

```
set search_path = public, pg_catalog;
```

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-createfunction.html

Системный каталог

- Системный каталог это таблицы, представления, функции, индексы и другие объекты которые используются для хранения метаданных
- Объекты системного каталога располагаются в схеме `pg_catalog`
- Объекты системного каталога (кроме глобальных) всегда располагаются в табличном пространстве по умолчанию для базы данных
- названия объектов приводятся к нижнему регистру
- Таблицы системного каталога неявно используют процессы кластера на этапе выполнения команд SQL

Системный каталог

Системный каталог - таблицы, представления, функции, индексы (на столбец с именем `oid` который есть в каждой таблице системного каталога) и другие объекты которые используются для хранения метаданных (данных о данных) и в служебных целях. Когда создаётся таблица или другой объект, то выполняется вставка строк в таблицы системного каталога и создаются файлы в файловой системе для хранения строк таблицы. Таблицы системного каталога неявно используют процессы кластера на этапе выполнения команд SQL, например, чтобы проверить существование таблиц, наличие привилегий, названия файлов в которых предстоит искать строки.

Например, команда создания базы данных помимо большого количества действий вставляет строчку в таблицу `pg_database`.

Объекты системного каталога располагаются в схеме `pg_catalog`.

Аналог системного каталога в Oracle Database называется "словарём данных" (data dictionary).

Названия объектов приводятся к нижнему регистру и хранятся в нижнем регистре (если не использовались двойные кавычки при задании имён).

Объекты системного каталога (кроме глобальных) всегда располагаются в табличном пространстве по умолчанию для базы данных.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/catalogs.html

Общие объекты кластера

- это объекты (таблицы и вспомогательные объекты) системного каталога, хранящие данные об общих объектах кластера
- располагаются в табличном пространстве `pg_global`
- Общие объекты кластера:
- базы данных
- роли
- табличные пространства
- подписки логической репликации
- источники логической репликации

Общие объекты кластера

В кластере есть и глобальные объекты, информация о них хранится в нескольких таблицах (глобальные объекты кластера), которые располагаются в табличном пространстве `pg_global`. Эти таблицы видны одинаковым образом в сессиях, соединённых с любой базой данных кластера. К глобальным объектам относятся 11 таблиц и 21 индекс на их столбцы, 9 TOAST таблиц и 9 индексов на TOAST таблицы. Всего 50 объектов.

Общие объекты кластера:
1)базы данных - таблица `pg_database`
2)роли - `pg_authid, pg_auth_members`
3)табличные пространства `pg_tablespace`
4)подписки логической репликации `pg_subscription`
5)источники логической репликации - `pg_replication_origin, pg_shseclabel, pg_shdepend, pg_shdescription`

Также в глобальном каталоге хранятся привилегии на право менять значения параметров `pg_parameter_acl` и параметры конфигурации ролей в сессиях с конкретными базами данных `pg_db_role_setting`.

Роли, табличные пространства, источники репликации, подписки логической репликации, а также сами базы данных не являются локальными SQL-объектами, так как они существуют вне какой-либо отдельной базы; они называются глобальными объектами. Имена таких объектов должны быть уникальными во всём кластере баз данных.

Использование системного каталога

- из таблиц и представлений системного каталога можно получать информацию командами SELECT, WITH, командами psql (начинаются на символ \), графическими приложениями и Платформой Tantor
- **\?** справка по командам psql
- Вносить изменения в таблицы системного каталога если, это не описано в документации нежелательно
- Зная архитектуру PostgreSQL, понятия, термины вы сможете легко получать информацию командами psql

Использование системного каталога

Изменения в таблицы системного каталога вносятся в процессе выполнения команд DDL. Таблицы системного каталога не заблокированы от внесения изменения. Вносить в таблицы системного каталога изменения, если это не документировано, напрямую командами SQL нежелательно. Выбирать данные из таблиц системного каталога напрямую командами `SELECT` и `WITH` возможно и используется в коде приложений и при администрировании кластера. Однако структура таблиц системного каталога не очень удобна для чтения человеком. Структура создавалась много лет назад, когда размеры систем хранения были небольшими, как и память на компьютерах. Для удобства работы с системным каталогом есть представления, которыми удобно пользоваться.

Получить список представлений системного каталога можно командой psql `\dvs`

Суффикс **s** в конце команд psql позволяет выдавать содержимое системного каталога, которые обычно по умолчанию не выдаются.

Более практична работа с системным каталогом с помощью команд psql. Команда `\?` выдает список всех команд psql. Выдастся в том числе и справка по самой команде `\?`:

```
Справка
\? [commands] справка по командам psql (то есть, начинающихся с символа \)
\? options    справка по параметрам командной строки утилиты psql
\? variables  справка по переменным, которыми меняется поведение psql
\h [ИМЯ]     справка по SQL-команде; * - по всем командам
```

Зная архитектуру PostgreSQL, понятия, термины вы сможете легко получать информацию командами psql.

Обращение к системному каталогу

- Выполняется командами SELECT и WITH
- можно соединять таблицы и представления
- столбец с именем oid - первичный ключ
- для связей между таблицами используется столбец oid
- максимальное число строк в таблицах системного каталога 4 миллиарда
- первые три символа в названии столбцов - сокращенное название таблицы

Обращение к системному каталогу

Можно обращаться к таблицам и представлениям системного каталога командой SELECT. Имена таблиц и представлений можно получить командой `psql \dtvS pg_*термин*`

По названию таблицы или представлению понять какая таблица или представление содержит нужную информацию. Дальше командой **\d** имя получить названия столбцов. Первые три символа в названии столбцов таблиц системного каталога по традиции содержат буквосочетание, похожее на название таблицы, где этот столбец создан. Например, в `pg_namespace` префикс "**nsp**". Начиная с четвертого символа обычно присутствует английское слово или его понятное сокращение.

Если на таблицу или столбцы созданы комментарии, то посмотреть их можно добавив "+" к команде **\d+** `имя_объекта`. К сожалению, описания к таблицам системного каталога не заданы. Описания можно посмотреть в документации.

В таблицах системного каталога первый столбец называется oid и тип у него oid. Посмотрим описание типа командой `\dT oid`

```
                      Список типов данных
   Схема     | Имя |                  Описание
-------------+-----+--------------------------------------------
 pg_catalog | oid | object identifier(oid), maximum 4 billion
(1 строка)
```

К этому типу дано описание, в котором написано, что максимальное количество значений 4 миллиарда. Отсюда следует, что в таблице системного каталога может быть не больше 4 миллиардов строк. Это означает, что если есть таблица для хранения типов (`pg_class`), то может быть не больше 4 миллиардов типов в одной базе данных. Также не больше 4 миллиардов отношений в одной базе данных. На столбец oid таблиц системного каталога создан индекс, а сам столбец является первичным ключом. Если количество строк в таблице системного каталога достигнет 4 миллиардов, то экземпляр и его процессы продолжат работать. В столбец **oid** значения вносятся автоинкрементом. По достижении 4 миллиардов серверные процессы, обслуживающие команды, которым нужно вставить новую строку в какую-либо таблицу системного каталога, будут выполнять поиск неиспользуемого значения (такие могут накопиться, значения в **oid** освобождаются после удаления объекта) в столбце **oid**, что приведет к замедлению выполнения команд. Не стоит миллиардами создавать объекты и затем миллиардами удалять их. Также надо помнить, что вакуумирование и заморозка работает и для таблиц системного каталога.

reg-типы

- созданы для 11 таблиц системного каталога
- используются для приведения текстового имени объекта к значению типа oid и обратно
- позволяют проще писать запросы к таблицам системного каталога уменьшая количество соединений таблиц
- Пример: найти название TOAST таблицы созданной для таблицы pg_tablespace:

```
select reltoastrelid, reltoastrelid::regclass from pg_class
where relname='pg_tablespace';
 reltoastrelid |     reltoastrelid
---------------+------------------------
          4185 | pg_toast.pg_toast_1213
```


reg-типы

Чтобы получить данные из таблиц системного каталога может потребоваться соединить несколько таблиц. Строки таблиц системного каталога связаны через столбец **oid**, то есть - число. В PostgreSQL можно создавать типы данных (`CREATE TYPE`), приведения типов (`CREATE CAST`) . Этим пользуются и разработчики PostgreSQL. Были созданы 11 типов данных и привидения типов, которые позволяют легко преобразовывать **oid** (число) в столбце одной из 11 таблиц системного каталога к имени объекта в этой таблице и обратно. Эти типы называются reg-типами. Использование reg-типов и привидений типов позволяет писать запросы к таблицам системного каталога не используя соединений (JOIN) и тем самым упрощая команду выборки. psql при обслуживании своих команд начинающихся на "\" формирует команду SELECT к таблицам системного каталога и иногда использует приведения типов. Такие SELECT можно посмотреть, установив переменную:

```
\pset ECHO_HIDDEN on
```

Список reg-типов можно посмотреть командой **\dT reg***

```
Список типов данных
   Схема    |      Имя      |                Описание
------------+---------------+---------------------------------------
 pg_catalog | regclass      | registered class
 pg_catalog | regcollation  | registered collation
 pg_catalog | regconfig     | registered text search configuration
 pg_catalog | regdictionary | registered text search dictionary
 pg_catalog | regnamespace  | registered namespace
 pg_catalog | regoper       | registered operator
 pg_catalog | regoperator   | registered operator (with args)
 pg_catalog | regproc       | registered procedure
 pg_catalog | regprocedure  | registered procedure (with args)
 pg_catalog | regrole       | registered role
 pg_catalog | regtype       | registered type
(11 строк)
```

Пример: **`SELECT relname, reltoastrelid::regclass FROM pg_class WHERE reltoastrelid>0 AND relnamespace='pg_catalog'::text::regnamespace order by 1;`** выдаст названия TOAST таблиц ~35 таблиц системного каталога, у которых они есть.

Часто используемые команды psql

- \l - список баз данных
- \du \dg - список ролей кластера
- \dn - список схем базы
- \db - список табличных пространств
- \dconfig *имя* - список параметров конфигурации
- \dfS pg* - список системных функций и процедур, полезных для администрирования. Часть информации о работе экземпляра и кластера можно получить только с помощью функций. Некоторые служебные представления используют функции
- \dvS pg* - полезные служебные представления
- символ + в конце команды показывает больше информации пример: \db+ покажет размер и привилегии

Часто используемые команды psql

\l - список (list) баз данных
\du или \dg - список ролей (user, group) кластера, \drg - назначения ролей ролям
\dn - список схем базы (namespace)
\db - список табличных пространств
\dconfig *имя* - список параметров конфигурации (config) кластера
\ddp - список привилегий по умолчанию (default privileges). Это особый тип привилегий или отзывающих привилегий, специфичный для PostgreSQL.
\dfS pg* - список системных функций (function) и процедур, полезных для администрирования. Часть информации о работе экземпляра и кластера можно получить только с помощью функций. Некоторые служебные представления используют функции. Процедуры появились в PostgreSQL позже функций, поэтому "f" используется и для процедур.
\dvS pg* - полезные системные (System) представления (view)
\dx - список установленных расширений (extention)
\dy - список триггеров на события, обычно, событийные триггера создают расширения или администраторы командой:

```
create event trigger имя on {LOGIN, DDL_COMMAND_START, DDL_COMMAND_END,
SQL_DROP, TABLE_REWRITE} execute function имя.
```

При вводе команды в psql помните о том, что можно нажать два раза клавишу табуляции на клавиатуре и psql высветит список возможных значений, которые можно ввести дальше:

```
postgres=# \
Display all 108 possibilities? (y or n)
```

Список полезных администратору функций:
https://docs.tantorlabs.ru/tdb/ru/18_1/se/functions-admin.html

Демонстрация

- Просмотр списка баз данных кластера
- Создание базы данных
- Переименование базы данных
- Ограничение на соединение с базой
- форматирование вывода psql

Демонстрация

Просмотр списка баз данных кластера
Создание базы данных
Переименование базы данных
Ограничение на соединение с базой
форматирование вывода psql

Практика

1. Установка параметров конфигурации на различных уровнях
2. Установка пути поиска в функциях и процедурах

Практика

Установка параметров конфигурации на различных уровнях
Установка пути поиска в функциях и процедурах

4

4b

Физическая структура кластера

Директория файлов кластера PGDATA

- Кластер хранит свои файлы в файловой системе
- Директория кластера называется PGDATA
- Директория или её поддиректории могут быть точками монтирования, жесткими, символическими ссылками
- Блочные устройства и неформатированные разделы жесткого диска не используются
- По умолчанию при работе с файлами кластера используется кэш операционной системы

Директория файлов кластера PGDATA

Файлы кластера баз данных хранятся в директории, которую называют `PGDATA`, по имени переменной окружения операционной системы, которую устанавливают для того, чтобы не указывать утилитам управления кластера директорию при каждом вызове утилит. Параметра (ключ) у утилит называется "**-D** директория" или "**--pgdata** директория". Если утилите указать параметр, он перекроет значение переменной окружения.

Кластер может хранить файлы данных вне директории `PGDATA` с помощью "табличных пространств", которые мы рассмотрим дальше в этой главе.

По умолчанию, инсталлятор Tantor Postgres создает директорию `/var/lib/postgresql/tantor-se-18/data` для хранения файлов кластера и файл службы `/usr/lib/systemd/system/tantor-se-server-18.service`, где указывает путь к этой директории. Параметрами **--edition** и **--major-version** инсталлятору можно задать другие значения. У каждого кластера своя директория `PGDATA`. Каждый кластер обслуживается одним экземпляром. На одном хосте может быть несколько кластеров, для их нужно создать директории PGDATA и файлы служб.

По умолчанию, при работе с файлами кластера, используется кэш операционной системы. Параметром конфигурации для разработчиков **`debug_io_direct`** возможно установить работу с файлами данных и журнальных (WAL) файлов в режиме прямого чтения-записи (direct i/o). Практических преимуществ в производительности и отказоустойчивости для PostgreSQL этот режим не даёт. Этот режим в PostgreSQL не стоит использовать.

Direct i/o используется в файловой системе PolarFS.

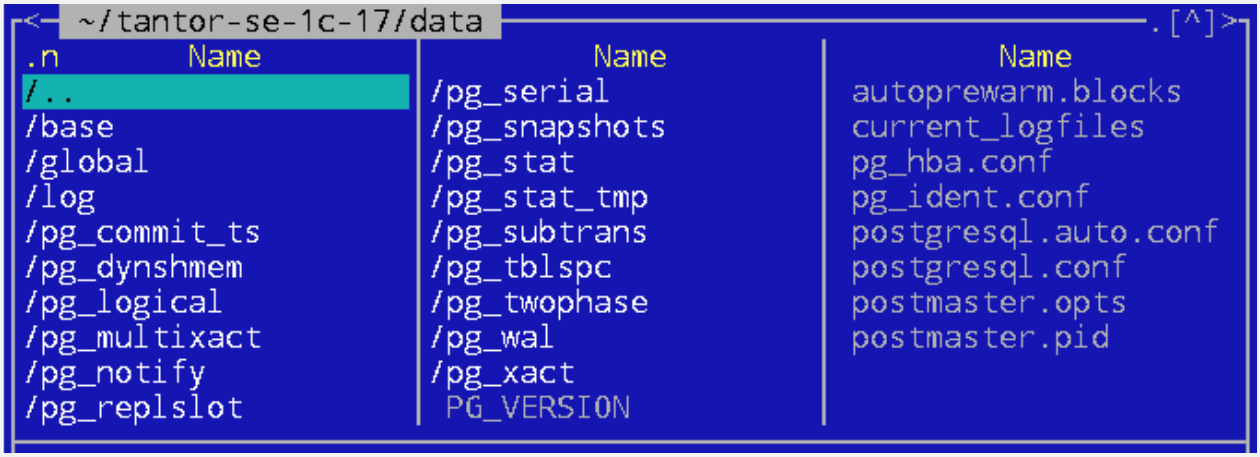
PostgreSQL не дублирует (не мультиплексирует) файлы кластера. Отказоустойчивость работы с файлами должна обеспечиваться на более низких уровнях - файловой системы, оборудования.

PostgreSQL пользуется функционалом файловых систем: символическими и жёсткими ссылками. Символические ссылки используются с директориями `PGDATA/pg_wal` и в директори `PGDATA/pg_tblspc`, в других директориях их нес тоит использовать. Жесткие ссылки используются утилитой pg_upgrade.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/storage-file-layout.html

Директории и файлы в PGDATA

- содержит файлы параметров
- поддиректории



Директория и файлы в PGDATA

В директории `PGDATA` находятся поддиректории с предопределёнными названиями.

По умолчанию, в корне директории `PGDATA` располагаются текстовые файлы параметров кластера: `postgresql.conf, pg_hba.conf` и `pg_ident.conf`, хотя они могут размещаться в других директориях. Файл параметров `postgresql.auto.conf` располагается только в корне `PGDATA`.

`current_logfiles` - текстовый файл с названием текущего файла, в который сборщик сообщений записывает журнал сообщений сервера. Сборщик сообщений включается параметром конфигурации `logging_collector` (**`ALTER SYSTEM SET logging_collector = on;`**) Измерение параметра требует перезапуска экземпляра. Использование сборщика сообщений рекомендуется при промышленной эксплуатации или при большом объеме записываемых в журнал сообщений данных.

`postmaster.opts` - содержит параметры командной строки, с которыми был запущен экземпляр

`PG_VERSION` - содержит номер основной версии (major release)

`postmaster.pid` - файл "блокировки", традиционно используемый в linux. Содержит номер (PID) основного процесса экземпляра; путь к `PGDATA`, метку времени запуска экземпляра, номер порта экземпляра, путь к каталогу Unix-сокет, IP-адрес по которому доступен экземпляр, идентификатор разделяемого сегмента памяти (SHM). Размер сегмента небольшой (56 байт). Разделяемая память по умолчанию использует тип mmap. Тип можно поменять параметром `shared_memory_type`, но этого делать не нужно.

Основные поддиректории:

`base` и `global` - директории двух табличных пространств, в них хранятся данные объектов кластера

`pg_stat` и `pg_stat_tmp` директории собираемой статистики. В директорию `pg_stat_tmp` идёт активная запись, располагать её на SSD не стоит (большой объем записи), возможно ее стоит расположить в памяти (in-memory file system).

`pg_tblspc` - содержит символические ссылки на директории табличных пространства. Удобно видеть какие директории кластера располагаются вне `PGDATA`.

`pg_wal` - содержащий файлы ("сегменты") журнала предзаписи (WAL - write ahead log). Потеря WAL файлов приводит к невозможности запуска кластера

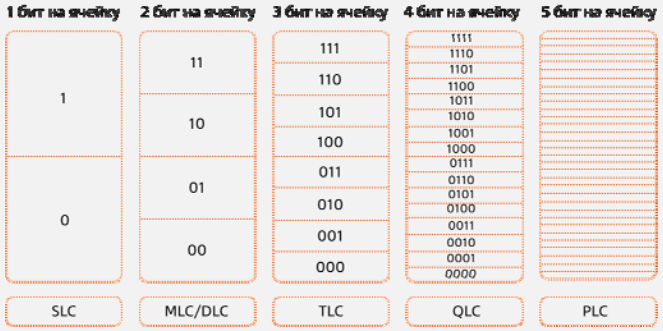
Директория `log` создана вручную для журнала сообщений.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/kernel-resources.html

Файлы журнала предварительной записи (WAL)

- Параметры `wal_recycle` и `wal_init_zero` определяют работу с файлами WAL журнала
- По умолчанию, файлы повторно используются, вновь создаваемые заполняются нулями
- **PGDATA/pg_wal** хранит файлы журнала
- При использовании SSD большая часть раздела файловой системы должна быть свободна

```
wal_block_size             | 8192
wal_buffers                | 4MB
wal_compression            | off
wal_consistency_checking   |
wal_init_zero              | on
wal_level                  | replica
wal_recycle                | on
wal_segment_size           | 16MB
wal_sync_method            | fdatasync
```



Файлы журнала предварительной записи (WAL)

Файлы журнала (WAL) создаются в директории `PGDATA/pg_wal`. В журнал записываются все изменения в блоках данных файлов кластера, за исключением нежурналируемых и временных объектов. Это значительный объем.

По умолчанию, значение параметра конфигурации `wal_recycle=on`. Это означает, что файлы не удаляются, а переименовываются и их тело повторно перезаписывается. Запись в тело файлов идёт потоком от начала файла до конца (если только не переключить на следующий файл функцией `pg_switch_wal()`).

Второй параметр `wal_init_zero`, значение по умолчанию ноль, что означает: при создании файлов заполнить нулями. При использовании `wal_recycle=on` файлы повторно используются и нечасто создаются, поэтому дополнительного объема записи немного.

При значении `wal_init_zero = off` при создании файла даётся команда записи последнего байта, чтобы зарезервировать место в файловой системе. Запись байта, а не блока оптимальна, так как операционная система будет использовать блок подходящего ей размера.

Если `PGDATA/pg_wal` смонтирован на SSD, то стоит следить за тем, чтобы объем хранящихся данных не превышал объем "SLC-кэша" который определяется технологией и алгоритмом контроллера. Для TLC (triple level cell, 3 бита на ячейку) объем "SLC-кэша" (логический термин, означающий, что контроллер записывает в высокоскоростной первый слой выдерживающий ~100тыс. циклов записи и не успевает переносить данные в другие слои, потому что блоки SSD занятые WAL файлами перезаписываются или очищаются discard*ом*) не может быть больше 1/3. Если превысить, то возникает деградация производительности (зависит от алгоритма работы контроллера) и долговечности. Другими словами, при использовании систем хранения на основе SSD общий объем файлов на точке монтирования `PGDATA/pg_wal` не должен быть больше примерно 20% от размера. Большой объем свободного места пригодится в случае, если реплика будет испытывать затруднения в приеме журнальных данные и мастер будет их удерживать.

https://wiki.archlinux.org/title/Solid_state_drive_(Русский)
https://en.wikipedia.org/wiki/Multi-level_cell

Директория с файлами журнала

- PGDATA/pg_wal может быть символической ссылкой, указывающей на смонтированный раздел диска
- утилите initdb можно указать в параметре --waldir=путь к директории, где будут храниться файлы журнала
- после создания кластера можно создать символическую ссылку вручную
- Можно включить сжатие образов страниц параметром wal_compression, алгоритмы: pglz, lz4, zstd

```
postgres@tantor:~/data$ initdb -D . --waldir=/var/lib/postgresql/wal
...
Success. You can now start the database server using:
    pg_ctl -D . -l logfile start
postgres@tantor:~/data$ ls -l pg_wal
lrwxrwxrwx 1 postgres postgres 44 pg_wal -> /var/lib/postgresql/wal
```


Директория с файлами журнала

PGDATA/pg_wal **может быть символической ссылкой**, указывающей на смонтированный раздел диска.

При создании кластера утилите initdb можно указать в параметре -X или --waldir=путь к директории, где будут храниться файлы журнала. Это означает, что будет создана символическая ссылка PGDATA/pg_wal. После создания кластера, можно остановить экземпляр, переместить содержимое директории и создать символическую ссылку.

У утилиты pg_basebackup также есть параметр --waldir=, который работает так же, как у утилиты initdb.

Пример ошибки, связанной с нехваткой места. Серверный процесс, который не смог записать в журнал, прерывается:

```
LOG:  server process (PID 6353) was terminated by signal 6: Aborted
```

Экземпляр падает:

```
LOG:  all server processes terminated; reinitializing
```

После перезапуска экземпляра, если места всё ещё нет:

```
LOG:  database system was not properly shut down; automatic recovery in progress
FATAL:  could not write to file "pg_wal/xlogtemp.6479": No space left on device
```

На SSD стоит монтировать файловую систему директории WAL с опцией discard (непрерывный TRIM) вместо службы fstrim, которая оптимальна для файловых систем, хранящих данные которые нечасто меняются. Проверить включён ли DISCARD можно командой линукс: lsblk --discard

Выбор оставить wal_recycle включённым зависит от алгоритма работы контроллера памяти SSD и файловой системы. Параметр wal_init_zero стоит отключить.

Параметр wal_compression по умолчанию отключён, он позволяет указать алгоритм сжатия, которым будут сжиматься полные образы страниц (full page writes), которые периодически записываются в журнал. Возможные значения pglz, lz4, zstd, on, off. По умолчанию off. Даст ли преимущество включение сжатия, нужно проверять.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-wal.html

Табличные пространства

- предназначены для того, чтобы кластер мог располагаться на нескольких устройствах хранения
- физически это директория в файловой системе
- являются общими объектами кластера
- могут хранить файлы нескольких баз данных кластера

Табличные пространства

Табличные пространства предназначены для того, чтобы кластер мог располагаться на нескольких устройствах хранения. Устройства хранения монтируются в разные директории. Табличное пространство это общий объект кластера, представляющий собой ссылку на директорию.

В командах создания объектов можно использовать имя табличного пространства и файлы объектов будут автоматически создаваться в поддиректориях этой директории. На табличные пространства можно давать привилегию USAGE пользователям. У табличного пространства есть владелец. Табличное пространство не относится ни к базе данных, ни к схеме, оно относится к кластеру.

Причины создания табличных пространств следующие. В операционной системе могут иметься точки монтирования файловых систем с разными характеристиками: объем места, автоматическое добавление места, производительность, отказоустойчивость. Администратор может распределять объекты баз данных по этим точкам монтирования (директориям).

Можно перемещать объекты между табличными пространствами, при этом будут даваться команды в операционную систему на создание, удаление и поблочное копирование содержимого файлов.

Табличное пространство, в котором нет объектов ни одной базы данных кластера можно удалить.

Табличные пространства: характеристики

- Поддиректории создаются автоматически
- При создании кластера создаются табличные пространства **pg_global** и **pg_default**
- Список табличных пространств: команда **\db** и таблица **pg_tablespace**
- содержимое списка одинаково во всех базах данных кластера
- **pg_tablespace** - глобальная таблица системного каталога

Табличные пространства: характеристики

После создания в кластере имеется два табличных пространства соответствующие поддиректориям `base` и `global` директории `PGDATA`:

```
postgres=# \db
      Список табличных пространств
    Имя       | Владелец | Расположение
------------+----------+--------------
 pg_default | postgres |
 pg_global  | postgres |
```

Табличное пространство `pg_default` используется по умолчанию для баз данных `template1`, `template0`, `postgres`.

Табличное пространство `pg_global` используется для хранения глобальных таблиц системного каталога и не должно использоваться для хранения пользовательских объектов. В этом табличном пространстве хранятся файлы таблицы `pg_tablespace`.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/manage-ag-tablespaces.html

Табличные пространства: характеристики

- Без кластера не существуют
- Не могут перемещаться между кластерами
- Не могут резервироваться отдельно от кластера
- Не являются объектами схем
- Имеют роль-владельца
- Директория табличного пространства:
- создаётся вручную в операционной системе
- должна располагаться в файловых системах, устойчивых к потере питания
- рекомендуется создавать вне PGDATA
- в директории не стоит удалять вручную файлы, они удаляются автоматически командами SQL

Табличные пространства: характеристики (продолжение)

Табличные пространства являются частью кластера баз данных. Даже если они находятся не в `PGDATA`, табличные пространства не могут рассматриваться как автономный набор файлов данных. Данные о том, какие объекты в каких файлах находятся хранится в системном каталоге, а не в табличном пространстве.

Табличные пространства не могут быть "отсоединены" и "присоединены" к другому кластеру базы данных. Они не могут резервироваться по отдельности.

Если табличное пространство будет повреждено (удалён файл, сбой диска) и экземпляр некорректно остановится, то экземпляр не запустится, так как потребуется восстановление по WAL-журналу блоков отсутствующих файлов. Кластер станет полностью недоступным. Поэтому размещать табличные пространства с объектами постоянного хранения на файловой системе неустойчивой к сбоям (в оперативной памяти) нельзя.

Размещать табличные пространства только с временными объектами (временные таблицы) если полностью уверены, что объектов постоянного хранения в них нет на файловой системе в памяти можно. При этом нужно учитывать, достаточно ли места под временные таблицы. Если место закончится, то команда вставки строки во временную таблицу выдаст ошибку, файл временной таблицы не удалится. Только команда удаления таблицы сможет удалить файл и освободить место. Команда усечения таблицы может выдать ошибку, так как сначала она создаёт новый файл, а места под него может не быть.

Экземпляр работает с директорией табличного пространства и её содержимым с правами того пользователя, из под которого запускаются процессы экземпляра. При создании табличного пространства, на уровне файловой системы на директорию должны быть даны привилегии на чтение-запись пользователю postgres операционной системы.

Команды управления табличными пространствами

- Команда создания:
  ```
  CREATE TABLESPACE имя [ OWNER роль ] LOCATION
  'директория'
   [ WITH ( параметр = значение [, ...] ) ]
  ```
- Смена владельца:
  ```
  ALTER TABLESPACE имя OWNER TO роль;
  ```
- Переименование:
  ```
  ALTER TABLESPACE имя RENAME TO имя;
  ```
- База данных имеет табличное пространство "по умолчанию", в нем находятся файлы объектов системного каталога этой базы данных
- Команда изменения табличного пространства по умолчанию для базы данных:
  ```
  ALTER DATABASE база SET TABLESPACE имя;
  ```


Команды управления табличными пространствами

У базы данных есть свойство: табличное пространство по умолчанию. В нём физически находятся файлы объектов системного каталога. Можно изменить табличное пространство по умолчанию, при этом содержимое файлов системного каталога будет перемещаться в новые файлы.

Команда создания табличного пространства:

```
CREATE TABLESPACE имя [ OWNER роль ] LOCATION 'директория'
[ WITH ( параметр = значение [, ...] ) ]
```

Директорию табличного пространства располагайте вне `PGDATA`.

Команда изменения табличного пространства по умолчанию для конкретной базы данных:

```
ALTER DATABASE база SET TABLESPACE имя;
```

Переименование табличного пространства:

```
ALTER TABLESPACE имя RENAME TO имя;
```

Смена владельца:

```
ALTER TABLESPACE имя OWNER TO роль;
```

Удаление табличного пространства (директория на диске не удаляется):

```
DROP TABLESPACE [ IF EXISTS ] имя;
```

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-createtablespace.html

Изменение директории табличного пространства

- Кластер определяет директорию табличного пространства по символической ссылке в директории **PGDATA/pg_tblspc**
- Процедура замены:
  - Найдите название символической ссылки в директории **PGDATA/pg_tblspc**, указывающую на директорию табличного пространства
  - Остановите экземпляр
  - Переместите директорию табличного пространства
  - Обновите символическую ссылку
  - Запустите экземпляр

Изменение директории табличного пространства

Команды для изменения директории (свойство `LOCATION`) табличного пространства нет, так как в табличном пространстве могут располагаться файлы локальных объектов нескольких баз данных кластера, а сессия, в которой даётся команда, не должна видеть локальные объекты других баз данных. Однако можно изменить директорию по следующей процедуре:

1)В директории `PGDATA/pg_tblspc` имеется символическая ссылка у которой название это **oid** (число) табличного пространства. Ссылка указывает на директорию табличного пространства:

```
ls -al | grep число
число -> /u01/postgres/my_tblspc
```

2)Убедитесь, что значение **oid** соответствует имени табличного пространства, которое вы хотите переместить:

```
SELECT oid, spcname FROM pg_tablespace;
```

3)Остановите экземпляр:

```
pg_ctl stop
```

4)Убедитесь, что экземпляр остановлен:

```
pg_controldata | grep down
Database cluster state: shut down
```

5)Переместите командой операционной системы или системы хранения директорию табличного пространства в нужное местоположение. При этом можно перемещать директорию внутри той же файловой системы (точки монтирования), либо в любую другую:

```
mv /u01/postgres/my_tblspc /u02/postgres
```

6)Убедитесь, что пользователю из под которого запускается экземпляр (postgres) даны разрешения на уровне файловой системы на чтение и запись в директорию и ее содержимое

7)Обновите символическую ссылку `PGDATA/pg_tblspc/число` , которая указывает на директорию табличного пространства:

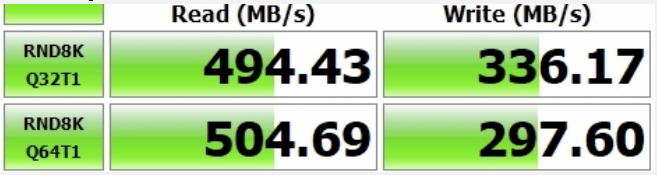
```
ln -fs /u02/postgres/my_tblspc $PGDATA/pg_tblspc/ЧИСЛО
```

8)Запустите экземпляр: `systemctl start tantor-se-server-18`

9)Проверьте, что местоположение поменялось. Например, командой psql **\db**

Параметры табличных пространств

- Четыре параметра: **seq_page_cost, random_page_cost, effective_io_concurrency, maintenance_io_concurrency**
- Перекрывают значения параметров, установленных на уровне кластера
- Влияют на стоимость плана выполнения команд SQL
- Можно задать при создании табличного пространства:
  CREATE TABLESPACE имя WITH (параметр = значение [, ...]);
- Можно изменить позже:
  ALTER TABLESPACE имя SET (параметр = значение [, ...]);
- Можно сбросить установленное значение:
  ALTER TABLESPACE имя RESET (параметр [, ...]);



Параметры табличных пространств

Для табличных пространств доступны четыре параметра: `seq_page_cost`, `random_page_cost`, `effective_io_concurrency`, `maintenance_io_concurrency`, которые можно установить на уровне табличного пространства. Установка этих значений влияет на создание планов выполнения запросов. Параметры представляют собой весовые коэффициенты, которые используются планировщиком в вычислениях стоимости. Параметры влияют на оценку планировщика какой ресурс "дороже" - дисковая подсистема или процессор.

По умолчанию, параметры установлены на уровне кластера.

`seq_page_cost (float)` - стоимость чтения блока с диска при последовательном чтении блоков. Файлы, в которых хранятся данные объектов делятся на блоки. Последовательным чтением считается, если блок логически идёт следующим - по смещению от начала файла. Экземпляр не знает о физическом расположении блоков в секторах жестких дисков. **По умолчанию 1.0**

`random_page_cost (float)` - стоимость чтения блока с диска при произвольном доступе к блокам файлов. **По умолчанию 4.0** Для SSD последовательный и произвольный доступ не отличаются по скорости, то есть `random_page_cost` можно сделать равным `seq_page_cost`. Уменьшение `random_page_cost` относительно `seq_page_cost` склоняет планировщик к методу доступа "Index Scan" вместо метода доступа "Seq Scan".

`effective_io_concurrency (integer)` - **По умолчанию 1**. Диапазон от 1 до 1000. Значение 0 отключает асинхронный ввод-вывод (не стоит устанавливать ноль).
Задаёт ограничение на число блоков, которые каждый серверный процесс будет асинхронно читать-писать. Для систем хранения на основе HDD отправной точкой может быть число жестких дисков. Для SSD можно увеличить до значения, после которого ускорение чтения-записи 8-килобайтными блоками перестаёт существенно расти (например, 64). Также этот параметр учитывается планировщиком при оценке стоимости Bitmap Index Scan.

`maintenance_io_concurrency (integer)` - **По умолчанию 10**. тот же смысл что и `effective_io_concurrency`, но используется фоновыми процессами и серверными при выполнении команд поддержки данных. Например, создание индексов, вакуумирование. Его значение должно быть не меньше `effective_io_concurrency`.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-resource.html

Временные файлы

- создаются в директориях табличных пространств
- табличные пространства для временных файлов задаются параметром `temp_tablespaces`
- рекомендуется создать отдельное табличное пространство для временных файлов
- временные файлы создаются если обрабатываемые данные не помещаются в память процесса
- временные файлы создаются для временных таблиц и их индексов, при выполнении команд SQL обрабатывающих большие объемы данных (например, сортировка, создание индекса)

Временные файлы

Ограничение на размер временных файлов используемых одним процессом можно установить параметром **`temp_file_limit`**. По умолчанию ограничение не установлено. Ограничивается объём временных файлов, которые используются каждым процессом экземпляра. Этот параметр также ограничивает суммарный файлов временных таблиц. При превышении ограничения команда, выполняемая процессом прервётся.

Временные таблицы могут активно использоваться приложениями.

При создании временного объекта создаются строки в таблицах системного каталога, а это объекты постоянного хранения. Также в файловой системе создаются обычные файлы. С временной таблицей работает одна транзакция и один процесс. Параллельные процессы работать с временной таблицей не могут, с ней работает только серверный процесс.

Если временная таблица часто очищается командой `TRUNCATE`, то эта команда (если не использовать расширения и сборки улучшающие работу с временными таблицами) создаёт новый файл в файловой системе с новым именем и обновляет поле `relfilenode` в таблице `pg_class`. Файл таблицы системного каталога может вырасти в размерах и автовакуум может отрабатывать чаще. Статистика на временные таблицы также сохраняется в объектах постоянного хранения. При частом создании временных таблиц с большим количеством столбцов порождается много строк в таблицах системного каталога. Таблицы системного каталога могут разрастись до десятков гигабайт.

В Tantor Postgres имеются оптимизации для работы с временными таблицами. Оптимизации включаются параметрами конфигурации enable_delayed_temp_file и enable_temp_memory_catalog

При включении параметра **enable_temp_memory_catalog** при создании, удалении и изменении временных объектов в таблицы системного каталога изменения не вносятся.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-client.html#GUC-TEMP-TABLESPACES

Основной слой хранения данных

- состоит из файлов размером до 1Гб
- новые файлы создаются при достижении предыдущего файла слоя 1Гб
- Максимальный размер файлов слоя 32Тб
- Работа с файлами всех слоёв объектов постоянного хранения происходит через буферный кэш
- Работа с файлами всех слоёв временных объектов происходит через буфер в локальной памяти серверного процесса
- Файлы всех слоёв располагаются в одной директории одного табличного пространства

Основной слой хранения данных

Файлы объектов в табличном пространстве делятся на типы, которые в PostgreSQL называют forks (ответвления, слои). Все файлы делятся на блоки размером 8Кб. Минимальный размер файла 8Кб.

Данные объекта хранятся в файлах основного слоя (main fork). Сначала создаётся первый файл основного слоя и увеличивается до 1Гб. Потом создаётся следующий файл и растет до 1Гб и дальше следующие. Максимальный размер таблицы (и любого relation) 32 терабайт (для размера блока 8Кб). Доступа к блокам всех слоёв объектов постоянного хранения происходит через буферный кэш общий для всех процессов кластера. Размер буферного кэша определяется параметром shared_buffers.

Доступ к блокам временных объектов (временных таблиц и индексов на них, последовательностей) происходит через буфер в локальной памяти серверного процесса. Размер буфера определяется параметром `temp_buffers`. Значение можно поменять в сессии, но только до первого обращения к временному объекту. Файлы временных объектов имеют такой же формат, как у объектов постоянного хранения.

Файлы всех слоёв располагаются в одном табличном пространстве в одной директории и не могут находиться в нескольких табличных пространствах.

Для обычных объектов, префикс имени файла представляет собой число и хранится в столбце `relfilenode` таблицы `pg_class`.

Если файл слоя (`main`, `fsm`) дорастает до 1Гб, создается новый файл с суффиксом "`.1`". Следующие файлы будут иметь суффикс "`.2`" и так далее.

Дополнительные слои

Карта свободного пространства (fsm)
- не создается для хэш-индексов
- создается вакуумированием

Карта видимости (vm)
- не создается для индексов
- создается вакуумированием
- два бита на блок основного слоя
- установленный первый бит означает что все строки блока актуальны и нет старых версий
- установленный второй бит означает что все строки блока заморожены

Слой инициализации (init)
- создается для нежурналируемых таблиц и индексов
- файл размером один блок без данных

Дополнительные слои

Для объектов (кроме хэш-индексов) создаётся слой "fsm" (карта свободного пространства, free space map). В файлах этого слоя хранится структура, отражающая наличие свободного места в блоках основного слоя. Структура организована не в виде списка, а в виде сбалансированного дерева, чтобы процессы могли быстро найти блок для вставки новой записи в блок основного слоя.

Для отношений (кроме индексов) создается слой "vm" (карта видимости и заморозки, visibility map). В файле этого слоя хранится по два бита на блок основного слоя таблицы. Единичка первого бита указывает, что в блоке основного слоя все строки самой последней версии (нет строк которые можно очистить). Этот бит используется при вакуумировании и при методе доступа индексного сканирования (index only scan), к блокам с таким битом они не обращаются. Если во втором бите единичка (бит взведён), это означает, что все строки на этой странице заморожены. Этот бит используется при вакуумировании в режиме заморозки, чтобы пропускать блоки, обработанные в прошлый раз и не менявшиеся с того времени. Файл создается и обновляется процессом, выполняющим вакуумирование. Если файл отсутствует (потерян), он создаётся заново, при этом обрабатываются все блоки основного слоя.

У нежурналируемых таблиц и индексов на них имеется слой "init", состоящий из файла размером один блок (8Кб), который после некорректной остановки экземпляра копируется на место первого файла основного слоя (если есть другие файлы они удаляются) нежурналируемого объекта и объект становится пустым.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/storage.html

Расположение файлов объектов

- Если объект расположен в `pg_default` то файлы располагаются в директории:
  `PGDATA/base/{oid базы данных}`
- Если в других табличных пространствах (значение столбца reltablespace в pg_class не равно нулю), то:
  `PGDATA/pg_tblspc/{reltablespace}/{oid базы данных}`
- Имена файлов объектов начинаются на relfilenode
- Для получения местоположения (относительно PGDATA) первого файла данных объекта используется функция `pg_relation_filepath(oid)`

Расположение файлов объектов

Если объект расположен в табличном пространстве по умолчанию, то его файлы располагаются в директории:
`PGDATA/base/{oid` базы данных из `pg_database}`
Если объект расположен других табличных пространствах (значение столбца reltablespace в pg_class не равно нулю), то файлы объекта располагаются в директории:
`PGDATA/pg_tblspc/{reltablespace из pg_class}/{oid базы данных}`
Имена файлов объектов начинаются на `relfilenode` из `pg_class`.
Для временных объектов имя файла имеет форму `tB_FFF`, где `B` - число, которое соответствует значению в названии временной схемы, в которой создан временный объект, `FFF` - значение `relfilenode` таблицы `pg_class`. Значения столбцов `relfilenode` и `oid` могут не совпадать, так как команды `TRUNCATE, REINDEX, CLUSTER` и другие создают файл с новым именем, а `oid` объекта не меняют. Более того, для некоторых объектов в `relfilenode` значение ноль.
Для получения местоположения (относительно `PGDATA`) первого файла основного слоя (main) используется функция `pg_relation_filepath(oid)`
Для получения префикса имени файлов используется функция `pg_relation_filenode(oid)`

Размеры табличных пространств и баз данных

- размер табличных пространств кластера `\db+` или функция `pg_tablespace_size()`

```
postgres=# \db+
                                  Список табличных пространств
    Имя     | Владелец | Расположение | Права доступа | Параметры | Размер
------------+----------+--------------+---------------+-----------+--------
 pg_default | postgres |              |               |           | 30 MB
 pg_global  | postgres |              |               |           | 565 kB
postgres=# SELECT spcname, pg_size_pretty(pg_tablespace_size(oid)) FROM pg_tablespace;
  spcname   | pg_size_pretty
------------+----------------
 pg_default | 30 MB
 pg_global  | 565 kB
```

- размер баз данных: `\l+` или функция `pg_database_size()`

```
postgres=# SELECT datname, pg_size_pretty(pg_database_size(datname)) FROM pg_database;
    datname    | pg_size_pretty
---------------+----------------
 postgres      | 7737 kB
 template1     | 7609 kB
 template0     | 7377 kB
 lab01iso88595 | 7537 kB
```


Размеры табличных пространств и баз данных

Размеры табличных пространств всего кластера можно посмотреть командой psql `\db+`

```
postgres=# \db+
                                  Список табличных пространств
    Имя     | Владелец | Расположение | Права доступа | Параметры | Размер
------------+----------+--------------+---------------+-----------+--------
 pg_default | postgres |              |               |           | 30 MB
 pg_global  | postgres |              |               |           | 565 kB
```

Также можно посмотреть функцией `pg_tablespace_size(oid)`:

```
postgres=# SELECT spcname, pg_size_pretty(pg_tablespace_size(oid)) FROM
pg_tablespace;
  spcname   | pg_size_pretty
------------+----------------
 pg_default | 30 MB
 pg_global  | 565 kB
```

Размер баз данных команда `l+` или функция `pg_database_size(имя)`:

```
postgres=# SELECT datname, pg_size_pretty(pg_database_size(datname)) FROM
pg_database;
    datname    | pg_size_pretty
---------------+----------------
 postgres      | 7737 kB
 template1     | 7609 kB
 template0     | 7377 kB
 lab01iso88595 | 7537 kB
```

Функция `pg_size_pretty()` выводит число в удобном для чтения виде, добавляя символы **kB MB GB TB**.

Функции определения размера

- Для получения размера файлов объектов есть набор функций
- **pg_relation_size(regclass, 'main' | 'vm' | 'fsm' | 'init')** выдаёт размеры слоёв
- **pg_indexes_size()** размер всех индексов созданных на таблицу
- **pg_table_size()** размер таблицы (TOAST и всех слоёв) без индексов
- **pg_total_relation_size()** размер таблицы включая TOAST, все индексы и слои

Функции определения размера

Определение размеров объекта может быть полезно, чтобы выяснить какие объекты занимают больше всего места и требуют внимания.

Список функций, выдающих размер объектов можно получить командой:
\dfS *size или запросом

```
SELECT proname, pg_get_function_arguments(oid) FROM pg_proc WHERE proname LIKE
'%size' ORDER BY 1;
```

```
  proname                | pg_get_function_arguments
-------------------------+---------------------------
 pg_column_size          | "any"
 pg_database_size        | name
 pg_database_size        | oid
 pg_indexes_size         | regclass
 pg_relation_size        | regclass
 pg_relation_size        | regclass, text
 pg_table_size           | regclass
 pg_tablespace_size      | name
 pg_tablespace_size      | oid
 pg_total_relation_size  | regclass
(10 строк)
```

Функции могут выдать размеры отдельных слоёв, общий размер таблицы с TOAST таблицей и индексами или без. Описание какая функция что выдаёт, можно посмотреть в разделе документации по функциям для администрирования:

https://docs.tantorlabs.ru/tdb/ru/18_1/se/functions-admin.html

Начиная с 14 версии, статистику записи в WAL можно посмотреть в представлении:

```
select * from pg_stat_wal;
```

```
 wal_records | wal_fpi | wal_bytes | wal_buffers_full | stats_reset
-------------+---------+-----------+------------------+----------------
     1115828 |    5572 | 102405739 |             6854 | 2026-05-07 11:03
```

Перемещение объектов

- можно переместить файлы таблиц, индексов, материализованных представлений
- в новые файлы копируется содержимое файлов всех слоёв
- устанавливает монопольную блокировку, несовместимую даже с командой `SELECT`
- Команда `ALTER тип ALL IN TABLESPACE` перемещает все объекты одного типа в текущей базе данных

Перемещение объектов

Можно переместить файлы таблиц, индексов, материализованных представлений из одного табличного пространства в другое.

При перемещении поблочно читаются файлы и их содержимое копируются в новые файлы. После перемещения файлы в исходном табличном пространстве удаляются. Весь объем перемещаемых данных проходит через WAL.

Второе ,что важно учитывать это то, что блокировки, устанавливаемые на перемещаемые объекты не дадут возможности работать с объектами даже командам `SELECT`, так как почти все (кроме запускаемых с опцией `CONCURRENTLY`) требуют блокировки уровня `ACCESS EXCLUSIVE` (монопольный режим работы с объектом). Сначала команда перемещения ставится в очередь на получение блокировки и ждёт, пока все транзакции и любые одиночные команды закончат работать с объектом, который нужно переместить. При этом, команда перемещения приводит к ожиданию любых команд, желающих работать с перемещаемым объектом, пока она не получит блокировку и не закончит перемещение.

Команды на перемещение файлов объектов в другое табличное пространство:

```
ALTER {TABLE | INDEX | MATERIALIZED VIEW } [ IF EXISTS ] имя SET TABLESPACE куда;
ALTER {TABLE | INDEX | MATERIALIZED VIEW } ALL IN TABLESPACE имя [ OWNED BY роль [, ... ] ] SET TABLESPACE куда [ NOWAIT ];
REINDEX [ TABLESPACE куда ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } [ CONCURRENTLY ] имя;
```

При использовании опции NOWAIT выдаётся ошибка, если команда не может немедленно получить все блокировки на все затрагиваемые объекты. **В 18 версии** появился параметр **`log_lock_failures`**, позволяющий отслеживать такие ошибки с NOWAIT.

Существует параметр `lock_timeout`, которым можно задать максимальное время ожидания получения любой командой блокировки. Если с объектом постоянно работают сессии, использование этого параметра может позволить получить блокировку, установив приемлемое время ожидания.

Существует параметр `statement_timeout` который должен быть больше `lock_timeout`, так как учитывается время ожидания получения блокировок. Этот параметр задаёт максимальное время выполнения команды, по достижению которого команда отменяется. Для команд перемещения `statement_timeout` вряд ли полезен.

Смена схемы и владельца

- можно менять владельца объекта и схему у тех объектов, у которых они должны быть
- Для смены владельца используется команда:
  `ALTER тип_объекта имя OWNER TO роль;`
- Для смены схемы используется команда:
  `ALTER тип_объекта имя SET TO схема;`
- Для массового переназначения всех объектов ролей в одной базе на другую роль используется команда:
  `REASSIGN OWNED BY роль TO роль;`
- удаление объектов принадлежащих роли в базе:
  `DROP OWNED BY имя [CASCADE];`

Смена схемы и владельца

Кроме перемещения файлов в другое табличное пространство можно менять владельца объекта и схему у тех объектов, у которых они должны быть.

При смене схемы или владельца изменения распространяются на зависимые объекты. Например, вместе с таблицей в другую схему перемещаются созданные на неё индексы, ограничения целостности, последовательности связанные со столбцами.

Владельцем и схемой индекса всегда является и становится владелец и схема таблицы.

Для смены владельца используется команда:

```
ALTER тип_объекта имя OWNER TO роль;
```

Для смены схемы используется команда:

```
ALTER тип_объекта имя SET SCHEMA схема;
```

Эти команды не комбинируются друг с другом, выполняются по отдельности и требуют `ACESS EXCLUSIVE` блокировки, но на короткое время.

Для массового переназначения всех объектов ролей в одной базе на другую роль используется команда:

```
REASSIGN OWNED BY роль TO роль;
```

Также есть команда удаления объектов принадлежащих роли в базе:

`DROP OWNED BY имя [CASCADE];` Опцию `CASCADE` можно использовать, чтобы удалить зависимые объекты, принадлежащие другим ролям.

Реорганизация и перемещение таблиц утилитой pg_repack

- утилита командной строки pg_repack
- имеется в Tantor Postgres
- нужно установить расширение в базах данных, где нужно реорганизовать таблицы
- реорганизация файлов таблиц с перемещением в другое табличное пространство или без перемещения
- в процессе реорганизации устанавливается блокировка ACCESS SHARE, позволяющая выполнять команды VACUUM, ANALYZE, SELECT, DML
- в конце реорганизации на короткое время устанавливается эксклюзивная блокировка

Реорганизация и перемещение таблиц утилитой pg_repack

В Tantor Postgres есть расширение `pg_repack`, которым можно переместить объекты в другое табличное пространство, не устанавливая на время работы монопольную блокировку на объекты, устанавливается только `ACCESS SHARE`.

монопольная блокировка устанавливается на короткое время в конце перемещения. Можно установить таймаут на получение этой блокировки параметром `--wait-timeout`. По истечении таймаута `pg_repack` может отменить свою операцию, если установить параметр `--no-kill-backend`. По умолчанию же `pg_repack` отменяет команды, мешающие ему получить блокировку. Если по истечении еще такого же периода времени он всё равно не может получить блокировку, pg_repack прервёт работу сессий функцией `pg_terminate_backend()`.

Перемещение объектов в другое табличное пространство не основная задача `pg_repack`, эта утилита реорганизует файлы объектов, делая файлы более компактными.

Можно задать количество параллельных сессий параметром `--jobs`, чтобы одновременно перестраивать несколько индексов на одной таблице в режиме полной реорганизации таблицы.

Реорганизация объектов запускается утилитой командной строки `pg_repack`, но для её работы **должно** быть установлено расширение в базах данных. Для этого достаточно выполнить команду `CREATE EXTENSION pg_repack;` в базах данных объекты которых хочется реорганизовать. **Базы, в которых не установлено расширение, утилитой игнорируются**.

Реорганизацию можно проводить в разных режимах: аналог `VACUUM FULL`, `CLUSTER`, `REINDEX`. **На время работы требуется дополнительное свободное место**: размер реорганизуемых объектов плюс изменения строк, которые накопятся за время переноса. **Также весь объем переносимых данных проходит через WAL**.

Перемещение организуется с помощью создания триггера, захватывающего изменения и сохраняющего их в таблицу логирования изменений. Создается новая таблица, в нее переносятся данные исходной таблицы, это самая долгая часть. После завершения переноса создаются индексы на новой таблице. Затем из таблицы с логом изменений переносятся накопленные изменения, пока в ней не останется пара десятков строк, тогда устанавливается монопольная блокировка на исходную таблицу, эти строки переносятся и исходная таблица заменяется на новую. При использовании на таблице ограничений целостности с отложенной проверкой возможны ошибки в работе расширения во время переноса строк из таблицы лога.

Скорость работы сравнима со скоростью работы команды CLUSTER.

https://docs.tantorlabs.ru/tdb/ru/18_1/be/pg_repack.html

Уменьшение размера файлов таблиц утилитой pgcompacttable

- Утилита написанная на языке Perl
- использует стандартное расширение pgstattuple
- Отличия от pg_repack:
  - свободное место равно размеру самого большого индекса, а не двойной размер таблицы и индексов
  - таблицы обрабатываются с адаптивной задержкой, а не с полной нагрузкой
  - не может перемещать файлы в другое табличное пространство

Уменьшение размера файлов таблиц утилитой pgcompacttable

С Tantor Postgres доступна утилита **pgcompacttable** , путь к ней: `/opt/tantor/db/18/tools/pgcompacttable`.

Утилита уменьшает размер файлов таблиц и индексов без тяжелых блокировок и без резкой нагрузки, влияющей на производительность. Файлы могут увеличиться ("bloat", раздуться) в размере из-за большого количества удалённых строк или частых обновлений строк, если автовакуум не мог очищать старые версии строк.

Отличия от pg_repack:

1) Требуемое для работы свободное место равно размеру самого большого индекса. pg_repack требует двойной размер таблицы и индексов. **pgcompacttable** обрабатывает содержимое файлов таблиц, индексы перестраиваются по очереди сначала меньший, потом больший по размеру файлов

2) таблицы обрабатываются с задержкой, чтобы предотвратить резкие скачки ввода-вывода и задержки в репликации (если она имеется). pg_repack работает с максимальной скоростью и нагрузкой на файловую систему

3) не может перемещать файлы в другое табличное пространство.

Утилита pgcompacttable сама не уменьшает размер файлов, за неё это делает вакуум. **При отключении параметром `vacuum_truncate` фазы усечения таблиц** (до 17 версии фазу усечения отключал параметр конфигурации `old_snapshot_threshold`) **pgcompacttable не уменьшит размеры файлов.**

Установка:

1)в базах данных нужно установить стандартное расширение `pgstattuple`:

```
create extension pgstattuple;
```

2)установить Perl: `apt-get install libdbi-perl libdbd-pg-perl`

или `yum install perl-Time-HiRes perl-DBI perl-DBD-Pg`

3) Дать права на выполнение утилиты:

```
sudo chmod 755 -R /opt/tantor/db/18/tools
```

https://docs.tantorlabs.ru/tdb/ru/18_1/se/pgcompacttable.html

TOAST (The Oversized-Attribute Storage Technique)

- Обычные таблицы (heap tables) хранят данные "построчно"
- Если строка не помещается в блоке данных, то используется технология TOAST
- Четыре режима хранения на уровне каждого столбца: `PLAIN`, `EXTERNAL`, `EXTENDED`, `MAIN`
- Для типов данных небольшого размера не предусмотрено хранение в TOAST и режим единственный: `PLAIN`
- для большинства остальных типов данных по умолчанию установлен режим `EXTENDED`
- используется сжатие на уровне полей, для полей небольшого размера сжатие неэффективно

```
create table t(n numeric storage main, t text storage plain);
alter table t alter column n set storage plain;
```


TOAST (The Oversized-Attribute Storage Technique)

Обычные таблицы (heap tables) хранят данные "построчно" - все поля одной строки физически рядом, потом все поля другой строки, если эти поля "влезают" в один блок данных размером 8Кб. Если строка не "влезает" в блок данных, то используется технология TOAST (**T**he **O**versized-**A**ttribute **S**torage **T**echnique): часть полей переносится в отдельную служебную TOAST-таблицу. Название этой таблицы не используется в командах SQL и ее использование полностью прозрачно. Можно на каждом столбце таблицы командой `ALTER TABLE имя ALTER COLUMN имя SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT }` установить режим хранения полей этих столбцов. Начиная с 16 версии, стратегию можно указать в команде создания таблицы:

```
create table t(n numeric storage main, t text storage plain);
```

Например, для режима `EXTENDED` установленного на столбцах, сначала поля таких столбцов будут сжиматься и, если строка со сжатыми полями поместится в блоке, то строка будет сохранена в блоке таблицы. Если же строка не влезет в блок, то часть полей строки будет перенесена в TOAST-таблицу. Для каждого типа данных, который потенциально может не поместиться в блоке (тип данных, "поддерживающий" хранение в TOAST) режим хранения определён по умолчанию (называется "стратегия" хранения полей этого типа) и для большинства типов данных установлена стратегия `EXTENDED`. Этот режим оптимален, если команды SQL будут обрабатывать поле целиком и значения хорошо сжимаются. Если значения плохо сжимаются или планируется обрабатывать значения полей (например, текстовые поля функциями `substr`, `upper`), то возможно более эффективным будет использование режима `EXTERNAL`. Для типов данных, размер которых невелик и для которых не предусмотрено хранение в TOAST (например, тип `DATE`) установлена "стратегия" (режим по умолчанию) хранения `PLAIN` и поменять режим командой `ALTER TABLE` на другой нельзя, будет выдана ошибка "`ERROR: column data type тип can only have storage PLAIN`".

Способ хранения для heap tables допускает сжатие значений отдельных полей. На данных небольшого размера алгоритмы сжатия менее эффективны. Доступ к отдельным столбцам не очень эффективен из-за того, что серверному процессу нужно найти блок в котором хранится часть строки влезающая в блок, затем по каждой строке отдельно выяснить нужно ли обращаться к строкам TOAST-таблицы, читать её блоки и "склеивать" части полей (`chunk`), которые в ней хранятся в виде строк этой таблицы.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/storage-toast.html

TOAST (The Oversized-Attribute Storage Technique)

- TOAST это техника хранения "атрибутов" (полей) большого размера
- поля большого размера - это поля не помещающиеся в блок
- позволяет хранить поля размером до 1Гб
- если в таблице есть заведомо большое поле или вставляется строка с большим полем, то создается toast-таблица и индекс на toast-таблицу
- в TOAST выносятся отдельные поля строк
- вынесенные поля делятся на части (chunk) по 1996 байт:

```
postgres@tantor:~$ pg_controldata | grep TOAST
Maximum size of a TOAST chunk:        1996
postgres@vanilla-x32:~$ pg_controldata | grep TOAST
Maximum size of a TOAST chunk:        2000
```


TOAST (The Oversized-Attribute Storage Technique)

TOAST (The Oversized-Attribute Storage Technique, техника хранения атрибутов большого размера) используется не только для хранения отдельных полей в TOAST-таблице. Код ядра PostgreSQL используется при обработке длинных значений в памяти. Не все встроенные типы данных поддерживают технику TOAST. Не поддерживают типы данных фиксированной длины, так как их длина небольшая и для любых значений одинакова ("фиксирована"), например, 1,2,4,8,12,16 байт.

Вынесенные в TOAST поля делятся на части - "чанки" (после сжатия, если сжатие применялось) размером 1996 байт:

```
postgres@tantor:~$ pg_controldata | grep chunk
Maximum size of a TOAST chunk:        1996
Size of a large-object chunk:         2048
```

которые располагаются в строках TOAST-таблицы размером 2032 байта. **Значения выбраны так, чтобы в блок таблицы TOAST поместилось четыре строки.**

В таблице TOAST есть три столбца:

chunk_id (тип OID, уникальный для поля вынесенного в TOAST размер 4 байта),

chunk_seq (порядковый номер чанка, размер 4 байта),

chunk_data (данные поля, тип bytea, размер сырых данных плюс 1 или 4 байта на хранение размера). Для быстрого доступа к чанкам на TOAST-таблицу создается составной уникальный индекс по столбцам chunk_id и chunk_seq.

В блоке таблицы остаётся указатель на первый чанк поля и другие данные.

Общий размер остающейся в таблице части поля всегда 18 байт.

В 32-разрядном PostgreSQL размер чанка на 4 байта больше: 2000 байт.

https://docs.tantorlabs.ru/tdb/ru/15_6/se/storage-toast.html

Поля переменной длины

- строка должна поместиться в один блок
- поля varlena, которые не помещаются в блок выносятся в таблицу TOAST
- типы данных фиксированной длины не сжимаются и не выносятся в TOAST
- стратегию хранения можно установить командой `ALTER TABLE имя ALTER COLUMN имя SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT }`.
- сжатие поддерживается для MAIN и EXTENDED
- 1 байт в начале поля переменной длины хранит длину поля длиной до 127 байт
- 4 байта в начале поля переменной длины хранит длину поля для полей длиннее 126 байт

Поля переменной длины

Строка (запись) таблицы должна поместиться в один блок размером 8Кб и не может находиться в нескольких блоках файлов таблицы. Однако, строки могут иметь размер больше 8Кб. Для их хранения применяется TOAST.

Запись в индексе btree не может превышать примерно треть блока (после сжатия проиндексированных столбцов, если оно применялось в таблице).

TOAST поддерживают типы данных `varlena (pg_type.typlen=-1)`. **Поля фиксированной длины не могут храниться вне блока таблицы**, так как для этих типов данных не написан код, реализующий хранение вне блока таблицы (в TOAST-таблице). При этом строка должна поместиться в один блок и фактическое число столбцов в таблице будет меньше, чем лимит в 1600 столбцов (`MaxHeapAttributeNumber` в `htup_details.h`).

Чтобы поддерживать TOAST, **в поле типа varlena первый байт или первые 4 байта всегда (даже если размер поля небольшой и не вытеснен в TOAST) содержат общую длину поля** в байтах (включая эти 4 байта). Причем, эти байты могут (но не всегда) быть сжаты вместе с данными, то есть храниться в сжатом виде. Один байт используется, если длина поля не превышает 126 байт. Поэтому, при хранении данных поля размером до 127 байт "экономится" три байта на каждой версии строки, а также отсутствует выравнивание, на чем можно сэкономить до 3 (typealign='i') или до 7 байт (typealign='d').

Другими словами, проектировщику схем хранения **лучше задать char(126) и меньше, чем char(127) и больше**.

Поля `varlena` с одним байтом длины **не выравниваются**, а **поля с 4 байтами длины выравниваются до pg_type.typealign.** Для большинства типов переменной длины выравнивание до 4 байт (`pg_type.typalign=i`). Отсутствие выравнивания даёт выигрыш в объёме хранения, что ощутимо для коротких значений. Но всегда нужно помнить о выравнивании **всей строки до 8 байт, которое выполняется всегда**.

Сжатие поддерживается только для типов данных переменной длины. Сжатие производится только, если режим хранения столбца установлен в MAIN или EXTENDED. Если поле хранится в TOAST и команда UPDATE не затрагивает это поле, то поле не будет специально сжиматься-разжиматься.

Для большинства типов переменной длины по умолчанию используется режим EXTENDED, кроме типов:

```
select distinct typname, typalign, typstorage, typcategory, typlen from pg_type where typtype='b' and typlen<0 and typstorage<>'x' order by typname;
```

```
   typname   | typalign | typstorage | typcategory | typlen
-------------+----------+------------+-------------+--------
 cidr        | i        | m          | I           |     -1
 gtsvector   | i        | p          | U           |     -1
 inet        | i        | m          | I           |     -1
 int2vector  | i        | p          | A           |     -1
 numeric     | i        | m          | N           |     -1
 oidvector   | i        | p          | A           |     -1
 tsquery     | i        | p          | U           |     -1
(7 rows)
```

Режим можно поменять командой:

```
alter type inet set (storage = external);
```

но этого не стоит делать на стандартных типах, это влияет на все таблицы.

Для каждого столбца помимо режима можно еще установить алгоритм сжатия (командами CREATE или ALTER TABLE). Если не устанавливать, то используется алгоритм из параметра `default_toast_compression`, который по умолчанию установлен в `pglz`.

Режим (стратегию) хранения можно установить командой **`ALTER TABLE имя ALTER COLUMN имя SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT }`**.

`EXTERNAL` похож на **`EXTENDED`**, только без сжатия и по умолчанию не установлено на стандартных типах. Если алгоритм `pglz` не может сжать первый килобайт данных, он прекращает попытку сжатия.

Вытеснение полей в TOAST

- выносятся поля строк длиной больше 2032 байт
  - при этом поля режутся на части по 1996 байт
- алгоритм (очередность) вытеснения зависит от порядка следования столбцов
- при доступе к каждому вытесненному полю дополнительно читается 2-3 блока TOAST-индекса
- функция проверки, вытеснено ли поле в TOAST: `pg_column_toast_chunk_id()`

```
select reltoastrelid, reltoastrelid::regclass
from pg_class where relname='t';
 reltoastrelid |      reltoastrelid
---------------+-------------------------
         74295 | pg_toast.pg_toast_74292
\d+ pg_toast.pg_toast_74292
TOAST table "pg_toast.pg_toast_74292"
   Column    |  Type   | Storage
-------------+---------+---------
 chunk_id    | oid     | plain
 chunk_seq   | integer | plain
 chunk_data  | bytea   | plain
Owning table: "public.t"
Indexes:
    "pg_toast_74292_index" PRIMARY KEY,
            btree (chunk_id, chunk_seq)
Access method: heap
select chunk_id, chunk_seq, length(chunk_data)
from pg_toast.pg_toast_74292;
 chunk_id | chunk_seq | length
----------+-----------+--------
    74297 |         0 |   1996
    74297 |         1 |      9
```


Вытеснение полей в TOAST

Способ хранения для обычных таблиц (heap tables) допускает сжатие значений отдельных полей. На данных небольшого размера алгоритмы сжатия менее эффективны. Доступ к отдельным столбцам не очень эффективен из-за того, что серверному процессу нужно найти блок в котором хранится часть строки, помещающаяся в блок, затем по каждой строке отдельно выяснить нужно ли обращаться к строкам TOAST-таблицы, читать её блоки и склеивать части полей (chunk), которые в ней хранятся в виде строк этой таблицы.

У одной таблицы может быть только одна связанная с ней таблица TOAST и один TOAST-индекс (уникальный btree индекс по столбцам `chunk_id` и `chunk_seq`). oid TOAST-таблицы хранится в поле `pg_class.reltoastrelid`.

При доступе к каждому вытесненному полю дополнительно читается 2-3 блока TOAST-индекса, что снижает производительность даже, если блоки в буферном кэше. Основное замедление на получение блокировки для чтения каждого лишнего блока. Любые разделяемые ресурсы (то, что не в локальной памяти процесса) требуют получения блокировки даже для чтения ресурса.

Поля после сжатия (если оно есть) делятся на части (chunk) по 1996 байт:

```
postgres@tantor:~$ pg_controldata | grep TOAST
Maximum size of a TOAST chunk:        1996
```

В PostgreSQL строка рассматривается на предмет помещения части ее полей в TOAST, если размер строки больше 2032 байт. Поля будут сжиматься и рассматриваться на предмет хранения в TOAST пока строка не поместится в 2032 байта или `toast_tuple_target` байт, если значение было установлено командой:

```
alter table t set (toast_tuple_target = 2032);
```

Оставшаяся часть строки в любом случае должна поместиться в один блок (8Кб).

В 17 версии появилась функция, которой можно узнать вынесено ли поле в TOAST:

```
create table t(n numeric);
insert into t values (1),(123456789::numeric^12345);
select length(n::text), pg_column_toast_chunk_id(n) chunk from t;
 length | chunk
--------+------
      1 |
  99890 | 41123
```

Алгоритм вытеснения полей в TOAST

- при вставке строки в таблицу она полностью размещается в памяти серверного процесса в строковом буфере размера 1Гб (или 2Гб
- при обновлении строки обработка выполняется по затрагиваемым командой полям в пределах строкового буфера. Поля, не затрагиваемые командой, представлены в буфере заголовком длиной 18 байт.
- после обработки (сжатия или выноса) каждого поля проверяется размер строки. Если размер не превышает `toast_tuple_target` (по умолчанию 2032 байт) строка сохраняется в буфер и обработка строки заканчивается
- обработка начинается с поля EXTENDED и EXTERNAL от наибольшего размера к меньшему
- сжатие и вынос полей MAIN выполняется только после выноса всех полей EXTENDED и EXTERNAL
- По умолчанию, используется сжатие pglz

Алгоритм вытеснения полей в TOAST

При вставке строки в таблицу, она полностью размещается в памяти серверного процесса в строковом буфере размера 1Гб (или 2Гб для сессий у которых установлен параметр конфигурации `enable_large_allocations=on`).

Алгоритм вытеснения в четыре прохода:

1) По очереди выбираются поля EXTENDED и EXTERNAL в порядке от наибольшего размера до меньшего. После обработки каждого поля проверяется размер строки и если размер меньше или равен `toast_tuple_target` (по умолчанию, 2032 байт), то вытеснение останавливается и строка сохраняется в блоке таблицы.

Значение `toast_tuple_target` можно переопределить только на уровне таблицы:

```
ALTER TABLE t SET (toast_tuple_target = 2032);
```

Берётся поле EXTENDED или EXTERNAL. EXTENDED сжимается. Если размер строки с полем в сжатом виде превышает 2032, поле вытесняется в TOAST. Поле EXTERNAL вытесняется не сжимаясь.

Параметр, устанавливающий алгоритм сжатия pglz или lz4: `default_toast_compression`

По умолчанию, используется сжатие pglz.

2) Если размер строки всё ещё превышает 2032, во второй проход вытесняются оставшиеся уже сжатые EXTENDED и EXTERNAL по очереди, пока размер строки не станет меньше 2032.

3) Если размер строки не стал меньше 2032, по очереди в порядке размера сжимаются поля MAIN. После сжатия каждого поля проверяется размер строки.

4) Если размер строки не стал меньше 2032, по очереди вытесняются сжатые на 3 проходе MAIN.

5) Если размер строки не помещается в блок, выдаётся ошибка:

```
row is too big: size ..., maximum size ...
```

При обновлении строки обработка выполняется по затрагиваемым командой полям в пределах строкового буфера. Поля, не затрагиваемые командой, представлены в буфере заголовком 18 байт.

TOAST chunk

- при использовании EXTERNAL поля размером от 1997 байт создают второй чанк небольшого размера из-за которого в блок TOAST помещается только 3 чанка большого размера
- для полей EXTERNAL с размером от 1997 до ~2300 байт есть вероятность менее плотного хранения

```
select reltoastrelid, reltoastrelid::regclass
from pg_class where relname='t';
 reltoastrelid |      reltoastrelid
---------------+-------------------------
         74295 | pg_toast.pg_toast_74292
\d+ pg_toast.pg_toast_74292
TOAST table "pg_toast.pg_toast_74292"
   Column    |  Type   | Storage
-------------+---------+---------
 chunk_id    | oid     | plain
 chunk_seq   | integer | plain
 chunk_data  | bytea   | plain
Owning table: "public.t"
Indexes:
    "pg_toast_74292_index" PRIMARY KEY,
            btree (chunk_id, chunk_seq)
Access method: heap
select chunk_id, chunk_seq, length(chunk_data)
from pg_toast.pg_toast_74292;
 chunk_id | chunk_seq | length
----------+-----------+--------
    74297 |         0 |   1996
    74297 |         1 |      9
```


TOAST chunk

Поле вытесняется в TOAST, если размер строки больше, чем 2032 байта, а резаться поле будет на части по 1996 байт. Из-за этого для поля больше 1996 байт появится чанк небольшого размера, который будет вставлен серверным процессом в блок с чанком большого размера. Например, в таблицу вставить 4 строки:

```
drop table if exists t;
create table t (c text);
alter table t alter column c set storage external;
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
```

в блок TOAST поместится 3 длинных чанка:

```
SELECT lp,lp_off,lp_len,t_ctid,t_hoff FROM heap_page_items(get_raw_page( (SELECT
reltoastrelid::regclass::text FROM pg_class WHERE relname='t'),'main',0));
 lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+-------
  1 |   6152 |   2032 | (0,1)  |     24
  2 |   6104 |     45 | (0,2)  |     24
  3 |   4072 |   2032 | (0,3)  |     24
  4 |   4024 |     45 | (0,4)  |     24
  5 |   1992 |   2032 | (0,5)  |     24
  6 |   1944 |     45 | (0,6)  |     24
```

Полный размер строки с длинным чанком **2032** байт (6104-4072).

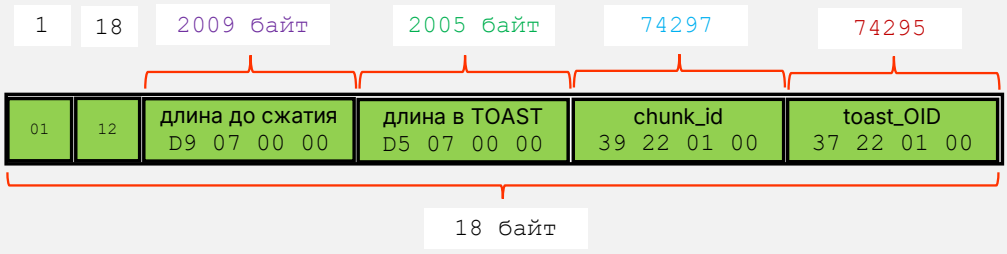
```
select lower, upper, special, pagesize from page_header(get_raw_page( (SELECT
reltoastrelid::regclass::text FROM pg_class WHERE relname='t'),'main',0));
 lower| upper | special | pagesize
------+-------+---------+---------
   48 |  1944 |    8184 |     8192
```

Пример как рассчитывать место в блоке для 4 строк размера 2032 байт (с 4 чанками):
24 (заголовок)+ 4*4 (заголовок) + 2032*4 + 8 (pagesize-special)=8176. Не используется 16 байт, но они бы и не могли использоваться, так строки выравниваются по 8 байт, а их 4.

Ограничения TOAST

- в таблицу TOAST может быть вынесено не больше 2^32 полей (~4млрд.)
- для вынесенного в TOAST поля в блоке таблицы хранится 18 байт, они не выравниваются, но вся строка выравнивается
- значение, остающееся в блоке таблицы, после выноса поля в TOAST:

```
select lp_len, t_data from heap_page_items(get_raw_page('t', 0));
 lp_len |                  t_data
--------+-----------------------------------------
     42 | \x0112d9070000d5070000392201003722 0100
```



Ограничения TOAST

В PostgreSQL служебной области special в конце блоков таблиц нет:

```
  48 |  1952 |    8192 |      8192
```

В 32-разрядном PostgreSQL:

```
Maximum size of a TOAST chunk: 2000
```

При использовании EXTENDED скорее всего поле будет сжато и маленького чанка не будет.

https://eax.me/postgresql-toast/

Каждое поле хранится в TOAST таблице в виде набора строк (chunk) хранится в виде одной строки в TOAST-таблице.

В поле основной таблицы хранится указатель на первый chunk **размером 18 байт** (независимо от размера поля). В этих 18 байтах хранится структура `varatt_external`, описанная в `varatt.h`:

первый байт имеет значение 0x01, это признак того, что поле вынесено в TOAST;

второй байт - длина этой записи (значение 0x12 = 18 байт);

4 байта длина поля с заголовком поля до сжатия;

4 байта длина того, что вынесено в TOAST;

4 байта - указатель на первый чанк в TOAST (столбец chunk_id таблицы TOAST);

4 байта - oid toast-таблицы (`pg_class.reltoastrelid`)

В столбце chunk_id (тип oid 4 байта) может быть 4млрд. (2 в степени 32) значений. Это значит, что в одной таблице в TOAST может быть вытеснено только 4млрд. **полей** (даже не строк). Это существенно ограничивает количество строк в исходной таблице и, вероятно, желателен мониторинг. Обойти ограничение можно секционированием.

Режим MAIN применяется для хранения внутри блока в сжатом виде, EXTERNAL - для хранения в TOAST в несжатом виде, EXTENDED для хранения в TOAST в сжатом виде. Если значения плохо сжимаются или планируется обрабатывать значения полей (например, текстовые поля функциями substr, upper), то эффективным будет использование режима EXTERNAL. Для типов фиксированной длины установлен режим PLAIN, который поменять командой ALTER TABLE нельзя, будет выдана ошибка "`ERROR: column data type тип can only have storage PLAIN`".

Колоночное хранение: общая информация

- снизить трудоёмкость доступа к данным в столбцах путём совместного хранения значений столбцов
- возможно эффективно сжимать данные по сравнению со сжатием полей в heap tables
- Для использования колоночного способа хранения достаточно указать при создании таблицы:
  `CREATE TABLE имя (...) USING columnar;`
- расширение создаёт табличный метод доступа **`columnar`**
- список методов доступа есть в таблице `pg_am`

Колоночное хранение: общая информация

Идея колоночного способа хранения (реализация Hydra) в том, чтобы снизить трудоёмкость доступа к данным в столбцах путём совместного хранения значений столбцов. При таком способе хранения физически рядом хранятся данные одного столбца целиком или по большому количеству строк. Благодаря тому, что в каждом столбце данные схожи возможно эффективно сжимать данные большими "наборами" строк (chunk). Размер "набора" можно установить на уровне таблицы параметром `columnar.chunk_group_row_limit`.

Для использования колоночного способа хранения достаточно указать при создании таблицы способ хранения:

```
CREATE TABLE имя (...) USING columnar;
```

Смена формата хранения командой `ALTER TABLE .. SET ACCESS METHOD` не реализована. Если реализовать функцию для изменения способа хранения и назвать её, к примеру, `alter_table_set_access_method`, то этой функции придётся перегружать все данные в новые файлы с блокировкой таблицы. Неблокирующая перегрузка данных является более универсальной и сложной задачей, которая должна реализовываться отдельным расширением и назвать его, к примеру, pg_reorg.

Поскольку хранение данных отличается от обычного, то табличный метод доступа heap не может использоваться и расширение создаёт свой табличный (`amtype = 't'`) метод доступа. Список методов доступа хранится в таблице системного каталога `pg_am`:

```
SELECT * FROM pg_am WHERE amtype = 't';
  oid  |  amname  |             amhandler              | amtype
-------+----------+------------------------------------+--------
     2 | heap     | heap_tableam_handler               | t
 18276 | columnar | columnar.columnar_handler          | t
```

Расширение **pg_columnar** создаёт схему `columnar`, которую использует для хранения своих объектов.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/hydra.html

Колоночное хранение: особенности использования

- поддерживаются UPDATE и DELETE
- поддерживаются команды: TRUNCATE, INSERT (в том числе одной строки), COPY
- параллельное сканирование реализовано
- поддерживаются типы индексов, используемые в ограничениях целостности: btree, hash
- остальные типы индексов не поддерживаются: gist, gin, spgist, brin
- совместимость с секционированием таблиц: секции могут иметь разные форматы хранения
- нет служебных столбцов xmin, xmax
- есть служебный столбец ctid

Колоночное хранение: особенности использования

Заменяет ли формат `columnar` формат `heap`? Нет. Формат heap эффективнее работает с запросами одиночных строк. В базах данных обслуживающих типичные бизнес-задачи (OLTP - online transaction processing) типа ведение продаж, складской и кадровый учет запросы к одиночным строкам встречаются чаще, чем выборка большого количества строк.

Хранение в формате `columnar` более эффективно в случаях: периодической загрузки большого набора строк в таблицу, чтения только части столбцов, отсутствия обновления и удаления одиночных строк. Формат `columnar` удобен для хранилищ данных, где данные накапливаются и по ним выполняются аналитические (обрабатывающие большое количество строк в целях создания отчета или анализа накопленных данных) запросы.

Hydra columnar поддерживает `UPDATE` и `DELETE`.

Поддерживаются `TRUNCATE`, `INSERT` (в том числе одной строки), `COPY`. Это основное, что ограничивает применение этого способа хранения.

Служебный столбец `ctid` в таблицах формата `columnar` есть, а xmin, xmax отсутствуют:

```
postgres=# select xmin, xmax, * from perf_columnar where id=3;
ERROR:  MIN / MAX TransactionID or CommandID not supported for ColumnarScan
```

TOAST с форматом `columnar` не используется, так как большие значения хранятся внутренне. Параллельное сканирование реализовано. Поддерживаются индексы типа `btree, hash` для быстрой проверки ограничений целостности (`PRIMARY KEY`, `UNIQUE` которые поддерживаются), а также в опции секционирования. Не поддерживаются типы индексов `gist, gin, spgist, brin`, поскольку индексный доступ неэффективен. Расширение совместимо с секционированием таблиц: секционированная таблица может иметь секции использующие и `heap` и `columnar` форматы хранения.

`relnamespace='pg_catalog'::text::regnamespace order by 1;` выдаст названия TOAST таблиц 36 таблиц системного каталога, у которых они есть.

Реализация колоночного хранения Citus columnar не поддерживает: команды `UPDATE` и `DELETE`; распараллеливание, служебный столбец `ctid`. В Tantor Postgres используется реализация Hydra, а не Citus.

Колоночное хранение: параметры

- у расширения есть параметры с префиксом "**columnar.**"
- последовательная запись в таблицу упорядоченных данных может улучшить сжатие, значительно сократить размер индексов и объем сканируемых блоков данных
- данные часто упорядочиваются по времени и называются "Time Series"
- наиболее эффективный алгоритм сжатия zstd, он установлен по умолчанию
- при чтении небольших объемов данных использование индексов может оказаться более эффективным

Колоночное хранение: параметры

Последовательная запись в таблицу упорядоченных данных может значительно сократить размер индексов (если они будут созданы) и объем распакованных наборов строк (`chunk`). Объем распаковываемых данных уменьшается за счет того, что типичным является задавать условие фильтрации по тому столбцу, по которому данные упорядочиваются, а большая часть запрашиваемых данных в случае последовательной вставки хранится совместно. Например, выбираются данные за последний час или сотня последних заказов (номер заказа генерируется последовательностью). Поэтому рекомендуется упорядочивать строки перед их вставкой в таблицу.

В практике часто встречается упорядочивание по времени. Такие данные называют "Time Series", строки вставляются последовательно во времени. Например, последовательная вставка в таблицу показателей измерений какого-нибудь параметра (цены акций, координат транспортного средства) во времени. Сжатие в таких таблицах обычно более эффективно, так как значения соседних полей похоже или даже не меняется (цена акций в последовательных сделках была одинаковой).

Самым эффективным методом сжатия данных является zstd.

При чтении небольших объемов данных использование индексов может оказаться более эффективным.

Расширение имеет параметры конфигурации:

```
postgres=# \dconfig columnar.*
            Parameter             | Value
----------------------------------+--------
 columnar.chunk_group_row_limit   | 10000
 columnar.column_cache_size       | 200MB
 columnar.compression             | zstd
 columnar.compression_level       | 3
 columnar.enable_column_cache     | off
 columnar.min_parallel_processes  | 8
 columnar.planner_debug_level     | debug3
 columnar.stripe_row_limit        | 150000
```

Можно устанавливать параметры на уровне таблиц. Параметры можно посмотреть в представлении `options` схемы `columnar`.

```
SELECT * FROM columnar.options;
 regclass |chunk_group_row_limit|stripe_row_limit|compression_level|compression
----------+---------------------+----------------+-----------------+-----------
 tab_name |               10000 |         150000 |               3 | lz4
```

Демонстрация

- Директория для временных файлов
- Перемещение директории табличного пространства

Демонстрация

Директория для временных файлов
Перемещение директории табличного пространства

Практика

1. Создание соединения с базой данных
2. Содержимое табличного пространства
3. Файл последовательности
4. Перемещение таблицы в другое табличное пространство
5. Перемещение таблицы в другое табличное пространство утилитой pg_repack
6. Использование pgcompacttable
7. Колоночное хранение pg_columnar

Практика

Создание соединения с базой данных
Содержимое табличного пространства
Файл последовательности
Перемещение таблицы в другое табличное пространство
Перемещение таблицы в другое табличное пространство утилитой pg_repack
Использование pgcompacttable
Колоночное хранение pg_columnar

5

Диагностический журнал

Диагностический журнал

- накапливает диагностические сообщения от процессов экземпляра
- используется для:
  - диагностики проблем
    - мониторинга и настройки производительности
  - аудита безопасности
  - исторического анализа того, что происходило при работе экземпляра
  - анализа выполнения запросов

```
postgres@tantor:~$ cat $PGDATA/log/postgresql-*.log
23:17:09.415 [784] LOG:  database system is ready to accept connections
23:17:09.732 [791] LOG:  autoprewarm successfully prewarmed 13763 of 13763 previously-loaded blocks
23:27:09.762 [786] LOG:  checkpoint starting: time
23:27:19.982 [800] STATEMENT:  select * from tickets1 where ticket_no='0005432020304';
23:27:21.200 [800] ERROR:  index "tickets1_ticket_no_idx" contains unexpected zero page at block 3
23:27:21.200 [800] HINT:  Please REINDEX it.
```


Диагностический журнал

Журнал сообщений (лог) PostgreSQL используется для отслеживания и анализа работы экземпляра. Процессы экземпляра могут создавать сообщения о том, что они делают. Эти сообщения полезны для:

1) диагностики проблем - встречались ли процессы с ошибками или неожиданными ситуациями

2) настройки и мониторинга производительности. Например, сообщения о долго выполнявшихся запросах или долгого вакуумирования таблицы

3) аудита безопасности. Например, логирования создания сессий, выдачи привилегий.

4) исторического анализа того, что происходило при работе экземпляра. Например, в какое время экземпляр запустился и начал принимать соединения

5) анализа выполнения запросов. Например, логирование планов запросов и статистик выполнения команд.

Сообщения от всех процессов направляются в единый журнал. В Tantor Postgres есть расширения `pgaudit` и `pgauditтofile`, которые могут использоваться для логирования событий безопасности в отдельный файл, чтобы не замусоривать сообщениями аудита безопасности диагностический журнал.

Уровни важности сообщений

- В коде ядра PostgreSQL, коде библиотек расширений, коде plpgsql сообщения помечаются уровнями важности (severity level)
- **log_min_messages** устанавливает сообщения каких уровней важности будут передаваться в диагностический журнал
  - допустимые значения и порядок важности для этого параметра: DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL, PANIC
- **client_min_messages** устанавливает сообщения каких уровней важности будут передаваться клиенту, создавшему сессию
  - допустимые значения и порядок важности для этого параметра: DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, LOG, NOTICE, WARNING, ERROR

```
postgres=# \dconfig *_min_messages
List of configuration parameters
      Parameter       |  Value
----------------------+---------
 client_min_messages | notice
 log_min_messages    | warning
(2 rows)
```


Уровни важности сообщений

В коде ядра PostgreSQL, коде библиотек расширений, коде plpgsql сообщения помечаются уровнями важности (severity level).

Параметр конфигурации **log_min_messages** устанавливает сообщения каких уровней важности будут передаваться в диагностический журнал ("лог"). Значение по умолчанию WARNING. Это означает, что в журнал будут попадать сообщения уровней "важнее" WARNING: WARNING, ERROR, LOG, FATAL, PANIC. Допустимые значения и **порядок важности** для этого параметра: DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL, PANIC.

Параметр конфигурации **client_min_messages** устанавливает сообщения каких уровней важности будут передаваться клиенту, создавшему сессию. Значение по умолчанию NOTICE. Это означает, что в журнал будут попадать сообщения уровней "важнее" NOTICE: NOTICE, WARNING, ERROR. Допустимые значения и порядок важности для этого параметра: DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, LOG, NOTICE, WARNING, ERROR.

Порядок важности и набор значений у двух перечисленных параметров **различаются**.

Менять значения по умолчанию нет смысла.

В plpgsql есть команда RAISE {DEBUG, LOG, INFO, NOTICE, WARNING, EXCEPTION} 'формат', выражения USING параметр = значение; для порождения сообщений. Уровень EXCEPTION аналогичен ERROR, откатывает транзакцию к неявной точке сохранения перед BEGIN и передает управление в секцию EXCEPTION, если такая секция есть. Пример:

```
postgres=# DO $$ BEGIN
 RAISE INFO 'info: %!', 'variable1' USING
  DETAIL = 'info detail', HINT = 'info hint';
 RAISE EXCEPTION 'text: %!', 'variable' USING ERRCODE = 'P0001',
  DETAIL = 'error detail', HINT = 'error hint';
END; $$;
INFO:  info: variable1!
DETAIL:  info detail
HINT:  info hint
ERROR:  text: variable!
DETAIL:  error detail
HINT:  error hint
CONTEXT:  PL/pgSQL function inline_code_block line 4 at RAISE
```

Расположение журнала

- `log_destination` определяет местоположение, куда будут выводиться диагностические сообщения
  - допустимые значения: `stderr, csvlog, jsonlog, syslog`
- если местоположений несколько, то они будут выводиться одновременно во все местоположения
- `logging_collector=on` запускает фоновый процесс **`logger`**, который перехватывает stderr и направляет сообщения в директорию `log_directory`
- в корне `PGDATA` создаётся текстовый файл с названием `current_logfiles`, в который записывается местоположение и текущие (куда сейчас идет запись) названия файлов диагностического журнала
- `log_filename` задаёт название файла (в `log_destination` одно значение `stderr`) или файлов лога
  - удобно значение `log_filename = postgresql-%F.log`

Расположение журнала

В параметре `log_destination` можно через запятую указать местоположение, куда будут выводиться диагностические сообщения. Допустимые значения: **stderr, csvlog, jsonlog, syslog**. Если местоположений несколько, то они будут выводиться одновременно во все местоположения. Значение по умолчанию stderr, что означает, что сообщения выводятся в текстовом виде в стандартный поток ошибок. Если экземпляр запущен через systemd, то по умолчанию stderr направлен в общий журнал linux. Если экземпляр запущен утилитой `pg_ctl start`, то stderr выводится в терминал. Если экземпляр запущен утилитой `pg_ctl start -l путь_к_файлу`, то есть с параметром `-l` или `--log=путь_к_файлу`, то журнал направляется в файл.

При промышленной эксплуатации экземпляр запускается systemd. Использовать общий журнал linux не удобно, так как в нём сохраняются сообщения от процессов экземпляра вперемешку с сообщениями других процессов операционной системы и удобно использовать параметр `logging_collector=on`.

Параметр `logging_collector=on` запускает фоновый процесс **`logger`**, который перехватывает stderr и направляет сообщения в директорию `log_directory`, в которой создается файл или файлы с названием `log_filename`. Для того, чтобы `logging_collector` мог логировать сообщения нужно, чтобы в `log_destination` был указан stderr и/или csvlog и/или jsonlog. Эти значения задают формат сообщений журнала. Форматы csvlog и jsonlog без **`logger`** не создаются. При указании в `log_destination` stderr и/или csvlog и/или jsonlog, в корне `PGDATA` создаётся текстовый файл с названием `current_logfiles`, в который записывается местоположение и текущие (куда в данный момент идёт запись) названия файлов диагностического журнала. Пример содержимого этого файла:

```
stderr log/postgresql-2025-12-25.log
csvlog log/postgresql-2025-12-25.csv
jsonlog log/postgresql-2025-12-25.json
```

Параметр `log_filename` задаёт название файла или файлов лога. Значение по умолчанию `postgresql-%Y-%m-%d_%H%M%S.log` Расширение файла действует для текстового формат stderr, для csv и json расширение файла (log) заменяется на csv и json. Маска в значении по умолчанию (`%H%M%S`) приводит к тому, что при каждом запуске экземпляра создается файл с новым именем. Более удобно значение `postgresql-%F.log` (`%F` эквивалент `%Y-%m-%d`).

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-logging.html
https://pubs.opengroup.org/onlinepubs/009695399/functions/strftime.html

Передача сообщений syslog

- при `log_destination = syslog` сообщения передаются службе операционной системы syslog
- в вывод syslog попадают не все сообщения, в **`logger`** все
  - **`logger`** предпочтительнее - он не теряет сообщения
- параметры конфигурации для взаимодействия с syslog:

```
postgres=# \dconfig syslog*
  List of configuration parameters
       Parameter        |  Value
------------------------+----------
 syslog_facility         | local0
 syslog_ident            | postgres
 syslog_sequence_numbers | on
 syslog_split_messages   | on
(4 rows)
```


Передача сообщений syslog

Сообщения могут передаваться службе операционной системы `syslog`.

В параметре `log_destination` можно указать значение `syslog`. Параметры конфигурации для `syslog`:

```
postgres=# \dconfig syslog*
  List of configuration parameters
       Parameter        |  Value
------------------------+----------
 syslog_facility         | local0
 syslog_ident            | postgres
 syslog_sequence_numbers | on
 syslog_split_messages   | on
(4 rows)
```

Некоторые типы сообщений могут не появляться в выводе syslog. Например, сообщения об ошибке динамической линковки библиотек, сообщения об ошибках при выполнении скриптов, указанных в параметрах конфигурации типа `archive_command`. Поэтому рекомендуется использовать **`logger`**:

```
logging_collector=on;
log_filename=postgresql-%F.log
```

`syslog` работает не надёжно, он может обрезать или терять сообщения, особенно когда они нужны. По умолчанию `syslog` сбрасывает каждое сообщение на диск, от чего уменьшается производительность. Для отключения этой синхронной записи можно добавить "-" перед именем файла в файле конфигурации `syslog`.

Ротация файлов диагностического журнала

- при использовании **logger** файлы журнала могут ротироваться
  - старые файлы не удаляются
- Параметр log_truncate_on_rotation позволяет при ротации по времени (но не при ротации по размеру или при запуске экземпляра) перезаписывать существующие файлы лога, а не дописывать в их конец
- Параметр log_file_mode задаёт разрешения на файлы диагностического журнала (по умолчанию 0600)
  - значение 0640 разрешит читать файлы членам группы postgres

```
postgres=# \dconfig *rotation*
 List of configuration parameters
        Parameter          | Value
---------------------------+-------
 log_rotation_age          | 1d
 log_rotation_size         | 10MB
 log_truncate_on_rotation | off
(3 rows)
```


Ротация файлов диагностического журнала

Чтобы файлы логов не разрастались, при использовании **logger** предусмотрена их ротация. При использовании syslog ротация настраивается в syslog. Параметры для настройки ротации:

```
postgres=# \dconfig *rotation*
 List of configuration parameters
        Parameter          | Value
---------------------------+-------
 log_rotation_age          | 1d
 log_rotation_size         | 10MB
 log_truncate_on_rotation | off
(3 rows)
```

Параметр log_truncate_on_rotation позволяет при ротации по времени (но не при ротации по размеру или при запуске экземпляра) перезаписывать существующие файлы лога, а не дописывать в их конец. Например, если log_filename=postgresql-%a.log и log_rotation_age=1d, то на каждый день недели будет создан свой файл и если log_truncate_on_rotation=on, то файлы будут перезаписываться раз в сутки.

Параметр log_file_mode задаёт разрешения на файлы диагностического журнала. Значение 0640 разрешит читать файлы членам группы. Разрешения на директорию, в которых лежат файлы, этот параметр не меняет.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/logfile-maintenance.html

Диагностический журнал

- `logging_collector` (по умолчанию `off`) рекомендуется установить значение в `on`
- запустится фоновый процесс **`logger`**, который собирает (collect) сообщения, отправленные в stderr и записывает их в файлы лога
- `log_min_messages=WARNING` по умолчанию, что означает логирование сообщений с уровнями `WARNING, ERROR, LOG, FATAL, PANIC`
- `log_min_error_statement=ERROR` по умолчанию. Минимальный уровень важности для команд SQL, которые завершились с ошибкой
- `log_directory=log` (PGDATA/log) по умолчанию. Задаёт путь к директории файлов лога, можно поменять на точку монтирования, скорость и объем которых достаточны для приема логов. Есть параметры, настраивающие ротацию файлов лога

```
psql -c "alter system set logging_collector = on;"
sudo systemctl restart tantor-se-server-18
ps -ef | grep logger
postgres   21861   21860  0 09:37  00:00:00 postgres: logger
```


Диагностический журнал

В коде PostgreSQL вставлены вызовы функции вида:
`ereport(WARNING, (errcode(КОД_СООБЩЕНИЯ), errmsg("текст сообщения")));`
Первый параметр - уровень важности (error level codes). В elog.h определено 15 уровней.
Включение процесса-сборщика (коллектора) логов:
`logging_collector=on` (по умолчанию off). Рекомендуется установить значение в on. По умолчанию сообщения передаются syslog и записываются в его формате, что неудобно для анализа. При большом количестве сообщений, с которыми нельзя справиться (скорость записи в файл ниже, чем скорость генерации) syslog не записывает часть сообщений (и правильно делает), logger же не очищает буфер errlog и процессы экземпляра, генерирующие сообщения блокируются до тех пор пока logger не запишет всё что накопилось (что тоже правильно). Другими словами logger не теряет сообщения, что может быть важным для диагностики. Такая ситуация может возникнуть из-за сбоя записи в файлы лога или включения высокого уровня логирования.
Если `logging_collector=on`, запускается фоновый процесс logger, который собирает (collect) сообщения, отправленные в stderr и записывает их в файлы лога.
Уровень сообщений, записываемых в журнал кластера задаётся параметрами:
`log_min_messages`, по умолчанию `WARNING`, что означает логирование сообщений с уровнями `WARNING, ERROR, LOG, FATAL, PANIC`.
`log_min_error_statement`, по умолчанию `ERROR`. Задает минимальный уровень важности для команд SQL, которые завершились с ошибкой
`log_destination=stderr` менять не нужно
`log_directory=log` (PGDATA/log) по умолчанию. Задаёт путь к директории файлов лога. Можно задать абсолютный путь (/u01/log) или относительно PGDATA (../log).
Название текущего файла (или файлов) лога указано в текстовом файле `PGDATA/current_logfiles`
Уровни важности от большей детальности к меньшей для лога:
`DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1` для отладки
`INFO` сообщения, обычно запрашиваемые опцией команды (VERBOSE)
`NOTICE` Полезные для клиентов сообщения
`WARNING` Предупреждения о возможных проблемах
`ERROR` ошибка, из-за которой прервана текущая команда
`LOG` сообщения, полезные для администраторов
`FATAL` ошибка из-за которой серверный процесс был остановлен (завершена сессия)
`PANIC` остановка серверных процессов основным процессом

Параметры диагностики

- параметров конфигурации логирования больше 35
- также есть больше 8 параметров для отладки команд SQL
- расширения могут иметь параметры для логирования
- основные параметры:

```
alter system set logging_collector=on;
alter system set log_min_duration_statement='8s';
alter system set log_statement=ddl;
alter system set log_min_error_statement=ERROR;
alter system set log_temp_files='1MB';
alter system set cluster_name='main';
alter system set log_autovacuum_min_duration='10s';
alter system set log_disconnections=on;
alter system set log_connections=on;
alter system set log_lock_waits=true;
alter system set deadlock_timeout='60s';
alter system set log_recovery_conflict_waits=on;
select pg_reload_conf();
```

```
\dconfig *debug*
debug_assertions                        | off
debug_discard_caches                    | 0
debug_io_direct                         |
debug_logical_replication_streaming     | buffered
debug_parallel_query                    | off
debug_pretty_print                      | on
debug_raw_parse                         | off
debug_print_plan                        | off
debug_print_rewritten                   | off
debug_print_upper_expr                  | off
(10 rows)
\dconfig log*
log_autovacuum_min_duration             | 10min
log_checkpoints                         | on
log_connections                         | off
log_disconnections                      | off
log_duration                            | off
log_error_verbosity                     | default
log_executor_stats                      | off
logging_collector                       | off
log_lock_waits                          | off
log_file_mode                           | 0600
log_filename                            | postgresql-
%Y-%m-%d_%H%M%S.log
logging_collector                       | on
log_hostname                            | off
logical_decoding_work_mem               | 64MB
log_line_prefix                         | %m [%p]
log_lock_waits                          | off
log_min_duration_sample                 | -1
log_min_duration_statement              | -1
```


Параметры диагностики

Какие параметры можно использовать для мониторинга возможных проблем с производительностью?

`log_min_duration_statement='8s'` в лог будут записаны все команды, выполняющиеся заданное значение и дольше. При нулевом значении записывается продолжительность выполнения всех команд. По умолчанию -1 ничего не записывается. Рекомендуется установить, чтобы выявлять длительно выполняющиеся команды (удерживают горизонт базы данных); случаи снижения производительности из-за которого длительность выполнения команд возрастает; возникновение проблем с командами: например, перестал использоваться индекс и время выполнения команд резко возросло. Пример:

```
LOG:  duration: 21585.110 ms
STATEMENT:  CREATE INDEX ON test(id);
```

Выдана длительность и команда.

`log_duration=off` регистрирует продолжительность всех команд после их выполнения. Недостаток: логируются все команды (без текста), одна строка на команду. Включать на уровне кластера не стоит. Преимущество - не логируется текст команд. Параметр можно использовать для сбора статистики по всем командам, но для этого нужно будет какой-то программой обработать файл журнала для анализа собранных данных. Необязательно включать на всём кластере, параметр можно включить на любом уровне. Пример:

```
LOG:  duration: 21585.110 ms
```

`log_statement=ddl` какие типы команд SQL будут логироваться. Значения: `none` (отключено), `ddl, mod` (то что ddl плюс команды dml), `all` (все команды).По умолчанию none. Рекомендуется установить в значение ddl. Команды ddl обычно устанавливают более высокий уровень блокирования, что увеличивает конкуренцию (contention). С помощью параметра можно выявить или исключить выполнение команды ddl как причину снижения производительности. Команды с синтаксическими ошибками по умолчанию не записываются. Если нужно логировать команды с ошибками синтаксиса, то нужно установить **`log_min_error_statement=ERROR`** (или детальнее). Нужно ли логировать команды с синтаксическими ошибками? Команды не нагружают серверный процесс, но могут сильно увеличить сетевой трафик. Причина ошибок может быть в коде приложения, которое безостановочно в цикле повторяет команду. Можно **периодически** включать логирование ошибочных команд. Пример записи при установленном `log_statement=ddl`:

```
LOG:  statement: drop table test;
```

Отслеживание использования временных файлов

- `cluster_name = 'main'` По умолчанию пусто. Рекомендуется установить. Значение добавляется к названию процессов экземпляра, что упрощает их идентификацию. На реплике по умолчанию используется для идентификации `wal_receiver`
- `log_temp_files='1MB'` (по умолчанию отключено) логирует имена и размеры создаваемых временных файлов **в момент их удаления**
- при нулевом значении логируются файлы любого размера
- временные файлы создаются в директории табличных пространств, указанных в параметре `temp_tablespaces`
  - можно ограничить параметром `temp_file_limit`
  - рекомендуется установить `log_temp_files` и `temp_file_limit`
- пример сообщения о том, что временный файл дорос до 97Мб:

```
STATEMENT:  CREATE INDEX ON test(id);
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp137894.0.fileset/0.0",
size 101810176
```


Отслеживание использования временных файлов

Рассмотрим примеры использования диагностического журнала и параметров логирования.

При большом количестве команд и замусоривании журнала можно использовать параметры `log_min_duration_sample` и `log_statement_sample_rate`. Параметр `log_transaction_sample_rate` имеет большие накладные расходы, так как обрабатываются все транзакции.

`cluster_name = 'main'` По умолчанию пусто. Рекомендуется установить. Значение добавляется к названию процессов экземпляра, что упрощает их идентификацию. На реплике по умолчанию используется для идентификации `wal_receiver`.

`log_temp_files='1MB'` логирует имена и размеры создаваемых временных файлов в момент их удаления. Почему в момент удаления? Потому что файлы растут в размерах и размер до которого доросли известен только в момент удаления файла. Как предотвратить рост файлов? Размер временных файлов (**в том числе файлов временных таблиц**) можно ограничить параметром `temp_file_limit`. При превышении размера команды будут выдавать ошибку. Пример:

```
insert into temp1 select * from generate_series(1, 1000000);
ERROR:  temporary file size exceeds temp_file_limit (1024kB)
```

Установка `temp_file_limit` поможет выявить ошибки, из-за которых план выполнения становится неоптимальным. Например, невозможность использования индекса и вместо него выполнение сортировки огромных объемов строк.

При нулевом значении логируются файлы любого размера, а при положительном - файлы, размер которых не меньше заданного значения. Значение по умолчанию -1, логирование отключено. Рекомендуется установить `log_temp_files` в относительно большое значение, чтобы обнаружить появление команд, которые нагружают дисковую систему. Дисковая система является наиболее нагруженным ресурсом в СУБД.

```
LOG: temporary file: path "base/pgsql_tmp/pgsql_tmp36951.0", size 71835648
STATEMENT:  explain (analyze) select p1.*, p2.* from pg_class p1, pg_class p2
order by random();
```

Временные файлы создаются в директории табличных пространств, указанных в параметре `temp_tablespaces`.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-logging.html

Отслеживание работы автовакуума и автоанализа

- log_autovacuum_min_duration, по умолчанию установлен в 10 минут. Если автовакуум превысит это время при обработке таблицы, то в лог кластера запишется сообщение. При возникновении таких сообщений стоит выяснять причину долгого вакуумирования таблицы

```
LOG:  automatic vacuum of table "postgres.public.test": index scans: 37
pages: 0 removed, 88496 remain, 88496 scanned (100.00% of total)
tuples: 10000000 removed, 10000000 remain, 0 are dead but not yet removable
removable cutoff: 799, which was 0 XIDs old when operation ended
new relfrozenxid: 798, which is 2 XIDs ahead of previous value
frozen: 1 pages from table (0.00% of total) had 82 tuples frozen
index scan needed: 44249 pages from table (50.00% of total) had 10000000 dead item identifiers removed
index "test_id_idx": pages: 54840 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 518.021 MB/s, avg write rate: 47.473 MB/s
buffer usage: 385747 hits, 1864788 misses, 170895 dirtied
WAL usage: 231678 records, 83224 full page images, 248448578 bytes
system usage: CPU: user: 25.72 s, system: 0.38 s, elapsed: 28.12 s

LOG:  automatic analyze of table "postgres.public.test"
avg read rate: 498.808 MB/s, avg write rate: 0.018 MB/s
buffer usage: 2906 hits, 27199 misses, 1 dirtied
system usage: CPU: user: 0.42 s, system: 0.00 s, elapsed: 0.42 s
```


Отслеживание работы автовакуума и автоанализа

Логирование полезно для наблюдения за автовакуумом.

log_autovacuum_min_duration, начиная с 15 версии, установлен в 10 минут. Если автовакуум превысит это время при обработке таблицы и индексов, то в лог кластера запишется сообщение. При возникновении таких сообщений стоит выяснять причину долгого вакуумирования таблицы.

Сообщение записывается в лог после завершения обработки таблицы и ее индексов.

Сообщение записывается, если "**elapsed:**" > log_autovacuum_min_duration

Полная длительность обработки таблицы и ее индексов указывается в "**elapsed:**". Разница **elapsed**-(user+system), это длительнность выполнения операций ввода-вывода.

В первую очередь стоит смотреть на "**elapsed:**" - это длительность транзакции автовакуума. Для TOAST будет отдельная запись в логе со своими показателями, как у обычной таблицы. Записи об автоанализе для TOAST не будет, так как **TOAST не анализируются**:

```
analyze pg_toast.pg_toast_25267;
WARNING:  skipping "pg_toast_25267" --- cannot analyze non-tables or special system tables
```

Во вторую очередь стоит обратить внимание на число проходов по индексам "index scans:". Значение больше 1 указывает на то, что памяти для построения списка TID не хватило. В этом случае стоит увеличить значение параметра **autovacuum_work_mem**.

В третью очередь, показатели эффективности цикла автовакуума "tuples:" и "frozen:". "scanned" будет меньше 100%, если блоки были очищены в предыдущем цикле вакуума, это нормально.

Значение "full page images" (и "bytes" пропорциональное ему), к эффективности вакуума не относятся и определяются случайностью: как давно была контрольная точка, либо нужно увеличить checkpoint_timeout. Даже наоборот, если значение "full page images" большие, то это может объяснять долгий цикл (значение в "**elapsed:**"). Большие значения "full page images" и "bytes" вместе с "tuples: число removed" означают эффективность цикла работы автовакуума или то, что он давно не обрабатывал таблицу (например, не мог заблокировать).

"avg read rate" и "avg write rate" ввод-вывод нельзя оценивать, так как он может не быть узким местом.

Наблюдение за контрольными точками

- первая запись появляется в момент начала контрольной точки
- total = 09:31:35.070 - 09:27:05.095
  - 270 секунд = checkpoint_completion_target * checkpoint_timeout = 0.9*300
- total=write+sync время записи в WAL-файлы
- sync= время, затраченное на вызовы fdatasync по WAL-файлам
- sync files=15 (синхронизировано файлов) - число файлов данных, в которые велась запись, по ним в конце контрольной точке посылается вызов fsync
- longest=0.003 s (самая_долгая синхр.) - наибольшая длительность обработки одного файла

```
09:27:05.095 LOG:  checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%), wrote 1 SLRU
buffers; 0 WAL file(s) added, 0 removed, 6 recycled; write=269.938 s, sync=0.009 s,
total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s; distance=109699 kB,
estimate=109699 kB; lsn=8/1164B2E8, redo lsn=8/BC98978
```


Наблюдение за контрольными точками

log_checkpoints по умолчанию, включён начиная с 15 версии и вдиагностическом журнале создаются записи о начале и завершении контрольной точки. Статистика выводится в записи о завершении.

Если вызывается контрольная точка (команда checkpoint или финальная), то контрольная точка по времени продолжает выполняться, но уже без задержки, создаёт запись об окончании и, только после ее завершения, начинается немедленная контрольная точка.

log_checkpoints создает записи в логе такого вида:

```
09:27:05.095 LOG:  checkpoint starting: time
09:31:35.070 LOG:  checkpoint complete: wrote 4315 buffers (26.3%), wrote 1 SLRU
buffers; 0 WAL file(s) added, 0 removed, 6 recycled; write=269.938 s, sync=0.009
s, total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s;
distance=109699 kB, estimate=109699 kB; lsn=8/1164B2E8, redo lsn=8/BC98978
```

Как читать записи:

1) Запись о начале передаётся в лог, когда начинается контрольная точка. Между этой записью и записью об окончании контрольной точки в логе может быть много записей не связанных с контрольной точкой. Значение total = 09:31:35.070 - 09:27:05.095 (270 секунд) получаются перемножением checkpoint_completion_target * checkpoint_timeout (0.9*300=270). Число блоков, которые checkpointer должен послать на запись, рассчитывается часто, но ближе к концу интервала, может внезапно увеличиться нагрузка на ввод-вывод и checkpointer может не успеть за заданный интервал. Чтобы минимизировать вероятность не вписаться в интервал между контрольными точками (checkpoint_timeout) для checkpoint_completion_target, по умолчанию, выбрано значение 0.9, которое оставляет 10% на случай если ввод-вывод не будет справляться.

2) total=write+sync. sync - это время затраченное на вызовы fsync. Большое время sync указывает на повышенную нагрузку на ввод-вывод. Эти показатели относятся к файлам данных.

3) sync files=15 (синхронизировано файлов) - число обработанных файлов, чьи блоки располагаются в буферном кэше (relations). Контрольная точка в начале записывает блоки буферов slru кэшей, но их размеры невелики. longest=0.003 s (самая_долгая синхр.) - наибольшая длительность обработки одного файла. average=0.001 s - среднее время обработки одного файла. Эти показатели относятся к файлам табличных пространств.

Описание записей log_checkpoints

- wrote 4315 buffers число грязных буферов, которые записаны по контрольной точке. Одновременно с checkpointer грязные блоки могут записывать другие процессы экземпляра
- (26.3%) процент от общего количества буферов буферного кэша, задаваемых параметром shared_buffers
- file(s) added, 0 removed, 6 recycled число созданных, удалённых, повторно использованных WAL файлов
- distance=109699 kB (расстояние) - объем записей WAL между **началом предыдущей** контрольной точки и началом текущей

```
09:22:05.087 LOG: checkpoint starting: time
09:26:35.066 LOG: checkpoint complete: wrote 3019 buffers (18.4%), wrote 1 SLRU buffers; 0
WAL file(s) added, 0 removed, 6 recycled; write=269.951 s, sync=0.009 s, total=269.980 s;
sync files=14, longest=0.004 s, average=0.001 s; distance=99467 kB, estimate=108859 kB;
lsn=8/AA004C8, redo lsn=8/5177990
09:27:05.095 LOG: checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%); 0 WAL file(s) added, 0
removed, 6 recycled;
write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s,
average=0.001 s; distance=109699 kB, estimate=109699 kB; lsn=8/1164B2E8, redo lsn=8/BC98978
```


Описание записей log_checkpoints

Как читать записи (продолжение):

4) wrote 4315 buffers число грязных блоков, которые были записаны по контрольной точке. Одновременно с checkpointer, грязные блоки могут записывать серверные процессы и bgwriter.

(26.3%) процент от общего числа буферов в буферном кэше (shared_buffers=128MB=16384).

В примере 4315/16384*100%=26.3366699%

5) file(s) added, 0 removed, 6 recycled число созданных удалённых, повторно использованных WAL файлов (по 16Мб).

6) distance=109699 kB (расстояние) - объем записей WAL между началом предыдущей контрольной точки и началом завершенной контрольной точки

select '8/BC98978'::pg_lsn-'8/5177990'::pg_lsn; = 112332776 = 109699kB

Описание записей log_checkpoints

- estimate=109699 kB (расстояние которое ожидалось) рассчитывается, чтобы оценить сколько WAL сегментов будет использовано в следующей контрольной точке
- если нули в "0 WAL file(s) added, 0 removed", то оценка estimate верная. Сколько файлов удалить, определяется параметрами min_wal_size, max_wal_size, wal_keep_size, max_slot_wal_keep_size, wal_init_zero=on, wal_recycle=on

```
09:22:05.087 LOG: checkpoint starting: time
09:26:35.066 LOG: checkpoint complete: wrote 3019 buffers (18.4%), wrote 1 SLRU
buffers; 0 WAL file(s) added, 0 removed, 6 recycled; write=269.951 s, sync=0.009 s,
total=269.980 s; sync files=14, longest=0.004 s, average=0.001 s; distance=99467 kB,
estimate=108859 kB; lsn=8/AA004C8, redo lsn=8/5177990
09:27:05.095 LOG: checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%); 0 WAL file(s)
added, 0 removed, 6 recycled;
write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s,
average=0.001 s;
distance=109699 kB, estimate=109699 kB; lsn=8/1164B2E8, redo lsn=8/BC98978
```


Описание записей log_checkpoints (продолжение)

7) После checkpoint starting: указываются свойства контрольной точки. time означает, что контрольная точка была вызвана "по времени" по истечении checkpoint_timeout .

Если размер WAL превышает max_wal_size будет сообщение:

LOG: checkpoint starting: wal

Если контрольная точка по wal начнется раньше, чем checkpoint_warning, то выдастся сообщение:

```
LOG:  checkpoints are occurring too frequently (23 seconds apart)
HINT:  Consider increasing the configuration parameter "max_wal_size".
```

23 секунды меньше, чем установлено checkpoint_warning= '30s'

Для контрольных точек после рестарта экземпляра:

LOG: checkpoint starting: end-of-recovery immediate wait

8) estimate=109699 kB (расстояние которое ожидалось) - обновляется по формуле:

```
if (estimate < distance) estimate = distance
 else estimate=0.90*estimate+0.10*distance;
```

(числа фиксированы в коде PostgreSQL)

Показатель estimate рассчитывается, чтобы оценить сколько WAL сегментов будет использовано в следующей контрольной точке. Исходя из estimate в конце контрольной точки определяется сколько файлов переименовать с целью повторного использования, а оставшиеся удалить. Сколько файлов удалить определяется параметрами min_wal_size, max_wal_size, wal_keep_size, max_slot_wal_keep_size, wal_init_zero=on, **wal_recycle=on**.

Повторное использование файлов не стоит отключать, оно оптимально для файловой системы ext4. Другие файловые системы (zfs, xfs, btrfs) не стоит использовать.

Если нули в "0 WAL file(s) added, 0 removed", то оценка estimate верная. Такие значения должны быть большую часть контрольных точек. Цель отображения значения estimate в этом. Объем журнальных записей между контрольными точками это **distance**.

9) Между контрольными точками прошло 09:27:05.095 - 09:22:05.087 = 300.008 секунд, что с высокой точностью равно checkpoint_timeout=300s

Про другие файловые системы: "btrfs assumes that pages do not change while being written out withdirect-io, and corrupts itself if they do" https://www.postgresql.org/message-id/fvfmkr5kk4nyex56ejgxj3uzi63isfxovp2biecb4bspbjrze7%40az2pljabhnff

Как выглядят ошибки на примере xfs: https://habr.com/ru/companies/postgrespro/articles/980218/

Утилита pg_waldump и записи log_checkpoints

- для просмотра записей в WAL-файлах используется утилита pg_waldump. По умолчанию утилита ищет WAL-файлы в "." (текущей директории откуда она запущена), дальше в ./pg_wal, $PGDATA/pg_wal
- в логе и выводе pg_controldata в LSN ведущие нули после "/" не печатаются
- в выводе pg_waldump в lsn и prev ноль печатается, а в redo не печатается

```
pg_controldata | grep check | head -n 3
Latest checkpoint location:           8/1164B2E8
Latest checkpoint's REDO location:    8/0BC98978
Latest checkpoint's REDO WAL file:    000000010000000080000000B
```

```
pg_waldump -s 8/0B000000 | grep CHECKPOINT
rmgr: XLOG len (rec/tot): 148/148, tx: 0,
lsn: 8/1164B2E8, prev 8/1164B298, desc: CHECKPOINT_ONLINE redo 8/0BC98978;
tli 1; prev tli 1; fpw true; xid 8064948; oid 33402; multi 1; offset 0; oldest xid 723 in DB 1;
oldest multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0; oldest running xid 8064947; online
pg_waldump: error: error in WAL record at 8/1361C488: invalid record length at 8/1361C4B0:
expected at least 26, got 0
```

```
09:27:05.095 LOG:  checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%), wrote 1 SLRU buffers; 0 WAL file(s) added,
0 removed, 6 recycled; write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s,
average=0.001 s; distance=109699 kB, estimate=109699 kB; lsn=8/1164B2E8, redo lsn=8/0BC98978
```


Утилита pg_waldump и записи log_checkpoints

Данные о последней контрольной точке записываются в управляющий файл. Для просмотра содержимого управляющего файла используется утилита **pg_controldata**:

```
pg_controldata | grep check | head -n 3
Latest checkpoint location:           8/1164B2E8
Latest checkpoint's REDO location:    8/0BC98978
Latest checkpoint's REDO WAL file:    000000010000000080000000B
```

Ноль после слэша ("/") не печатается, в примерах на слайде и под слайдом нули добавлены вручную.

Данные соответствуют записи о последней контрольной точке в логе.

Для просмотра записей в WAL-файлах используется утилита pg_waldump. По умолчанию утилита ищет WAL-файлы в текущей директории откуда она запущена, потом в директориях ./pg_wal, $PGDATA/pg_wal. Пример просмотра записи в журнале об окончании контрольной точки:

```
pg_waldump -s 8/0B000000 | grep CHECKPOINT
```

или

```
pg_waldump -s 8/BC98978 | grep CHECKPOINT
rmgr: XLOG len (rec/tot): 148/148, tx: 0,
lsn: 8/1164B2E8, prev 8/1164B298, desc: CHECKPOINT_ONLINE redo 8/0BC98978;
tli 1; prev tli 1; fpw true; xid 8064948; oid 33402; multi 1; offset 0; oldest xid 723 in DB 1; oldest multi 1
in DB 5; oldest/newest commit timestamp xid: 0/0; oldest running xid 8064947; online
```

Утилите не указан LSN до которого сканировать журнал (параметр -e), поэтому дойдя до самой последней журнальной записи, которая была записана в журнал, утилита выводит сообщение, что следующая запись пустая:

```
pg_waldump: error: error in WAL record at 8/1361C488: invalid record length at
8/1361C4B0: expected at least 26, got 0
```

В логе, выводе утилиты pg_controldata в LSN ведущие нули после "/" не печатаются. В выводе pg_waldump в lsn и prev ноль печатается, а в redo не печатается. Перед числом 8 нули тоже незримо присутствуют, но их отсутствие не создаёт путаницу. Можно запомнить, что после слэша должно идти восемь HEX-символов.

Утилита pg_waldump и записи log_checkpoints

- lsn 8/1164B2E8 запись о конце контрольной точки
- redo 8/0BC98978 запись о начале контрольной точки, с которой начнется восстановление в случае сбоя экземпляра
- prev 8/1164B298 адрес начала предыдущей записи в журнале
- distance объем журнала от начала предыдущей до начала завершенной контрольной точки '8/0BC98978'::pg_lsn-'8/05177990'::pg_lsn

```
pg_controldata | grep check | head -n 3
Latest checkpoint location:           8/1164B2E8
Latest checkpoint's REDO location:    8/BC98978
Latest checkpoint's REDO WAL file:    000000010000000080000000B
```

```
pg_waldump -s 8/0B000000 | grep CHECKPOINT
rmgr: XLOG len (rec/tot): 148/148, tx: 0,
lsn: 8/1164B2E8, prev 8/1164B298, desc: CHECKPOINT_ONLINE redo 8/BC98978;...
```

```
09:26:35.066 LOG: checkpoint complete: wrote 3019 buffers (18.4%), wrote 1 SLRU buffers; 0 WAL file(s) added,
0 removed, 6 recycled; write=269.951 s, sync=0.009 s, total=269.980 s; sync files=14, longest=0.004 s,
average=0.001 s; distance=99467 kB, estimate=108859 kB; lsn=8/AA004C8, redo lsn=8/5177990
09:27:05.095 LOG:  checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%); 0 WAL file(s) added, 0 removed,
6 recycled; write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s;
distance=109699 kB, estimate=109699 kB; lsn=8/1164B2E8, redo lsn=8/BC98978
```


Утилита pg_waldump и записи log_checkpoints (продолжение)

lsn 8/1164B2E8, запись о конце контрольной точки.

redo 8/0BC98978 запись о начале контрольной точки, с которой начнется восстановление в случае сбоя экземпляра. Из записи выбирается адрес записи, которая формировалась на момент начала контрольной точки (redo), читается эта запись. Все записи от redo до lsn должны быть прочитаны и наложены на файлы кластера. После наложения lsn файлы кластера считаются согласованными.

prev 8/1164B298 адрес начала предыдущей записи в журнале. По журналу можно скользить "назад". При этом в журнальных записях отсутствуют LSN следующей журнальной записи. Почему? Адрес следующей журнальной записи можно рассчитать по полю len (rec/tot): 148/148 которое хранит длину журнальной записи. Минимальная длина журнальной записи 26 байт (expected at least 26). При этом реальная длина журнальной записи дополняется (padding) до 8 байт. Реальная длина записи в примере будет **152** байта, а не 148. Пример:

```
pg_waldump -s 8/1164B298 -e 8/1164B3E8
 rmgr: Standby     len (rec/tot):     76/    76, tx:        0, lsn: 8/1164B298, prev 8/1164B240, desc:
RUNNING_XACTS nextXid 8232887 latestCompletedXid 8232885 oldestRunningXid 8232886; 1 xacts: 8232886
 rmgr: XLOG        len (rec/tot):    148/   148, tx:        0, lsn: 8/1164B2E8, prev 8/1164B298, desc:
CHECKPOINT_ONLINE redo 8/BC98978; ...
 rmgr: Heap        len (rec/tot):     86/    86, tx: 8232886, lsn: 8/1164B380, prev 8/1164B2E8, desc: HOT_UPDATE
...
```

lsn+len + padding до 8 байт = LSN начала следующей записи

По записи в логе или управляющем файле можно узнать размер журнальных записей. От него зависит длительность восстановления.

Объем записанных WAL в контрольной точке вычисляется по этим полям:

`select pg_wal_lsn_diff('8/1164B2E8','8/BC98978');` = 94054768 = 91850kB.

Объем WAL от начала до конца контрольной точки 91850kB.

Объем от начала предыдущей контрольной точки до начала завершенной, то есть расстояние (distance) между контрольными точками:

`select '8/BC98978'::pg_lsn - '8/5177990'::pg_lsn;` = 112332776 = 109699kB

Для вычислений можно использовать функцию pg_wal_lsn_diff или оператор "-", результаты одинаковы. Для использования оператора нужно привести строку к типу pg_lsn.

Логирование соединений

- выявляет нет ли слишком частых соединений и коротких сессий
- порождение сессии для выполнения одного запроса не оптимально
- Для логирования соединений используются параметры:

```
log_connections={all, authentification, receipt,
  authorization, setup_duration, on, off}
log_disconnections=on
pgaudit.log_connections=on
pgaudit.log_disconnections=on
```

```
LOG: connection received: host=[local]
LOG: connection authorized: user=postgres database=db2 application_name=psql
FATAL: database "db2" does not exist
LOG: connection received: host=[local]
LOG: connection authorized: user=alice database=alice application_name=psql
FATAL: role "alice" does not exist
```


Логирование соединений

Логирование соединений с экземпляром полезно для выявления нет ли слишком частых соединений и коротких сессий. Могут встречаться приложения, работающие в режиме подсоединение-запрос-отсоединение. Причина существования таких приложений: использование скриптовых языков, которые использовались для создания html-страниц. Каждая страница создавалась одним скриптом. В базах данных, использовавшихся такими приложениями, создание сессии было недорогой с точки зрения использования ресурсов операцией так как функционал баз данных был достаточно прост и рассчитан на простые запросы к одиночным таблицам без аутентификации и контроля доступа. В СУБД PostgreSQL при создании сессии порождается процесс в операционной системе, выполняются подготовительные операции (аутентификация, проверка прав доступа, регистрация сигналов, выделение структур памяти), что относительно трудоёмко. Порождение сессии для выполнения одного запроса не оптимально и приводит к бесполезному использованию вычислительных ресурсов и памяти. Oracle Database использует такую же архитектуру. Промышленные приложения используют языки и архитектуры, которые используются пулы соединений на уровне серверов приложений. Приложения мониторинга экземпляра PostgreSQL могут подсоединяется к базе данных с частотой раз в несколько секунд или десятков секунд, выполнять несколько запросов и отсоединяться.

Логирование соединений позволяет выявить такие приложения и системы мониторинга. Для этого достаточно логировать каждое соединение и его длительность.

Вторая причина по которой используется логирование соединений - регуляционные требования безопасности "аудит соединений". Аудит используется для того, чтобы в случае взлома программной системы определить какие данные и когда были похищены для устранения последствий. Например, замену номеров платежных карт или кодов доступа, которые были похищены. Поэтому аудит соединений может быть включён постоянно.

Для логирования соединений используются параметры:

`log_connections={all, authentification, receipt, authorization, setup_duration}`

(До 18 версии параметр был булевым `on/off`)

`log_disconnections=on`

`pgaudit.log_connections=on`

`pgaudit.log_disconnections=on`

Параметр log_connections

- `log_connections` записывает в диагностический журнал кластера попытки подключения к экземпляру, попытки аутентификации и успешную аутентификацию
- изменить значение можно только на уровне кластера
- ни на уровне роли, ни на уровне базы данных параметра не может устанавливаться:

```
postgres=# alter user alice set log_connections = 'all';
ERROR:  parameter "log_connections" cannot be set after connection start
```

- пример сообщений:

```
LOG:  connection received: host="10.0.2.15"
LOG:  connection authorized: user=fff database=fff application_name=psql
FATAL:  role "fff" does not exist
```


Параметр log_connections

В 18 версии принимает комбинацию значений:`{all, authentification, receipt, authorization, setup_duration}`. До 18 версии параметр был булевым `on/off`, что эквивалентно `authentification, receipt, authorization` и может использоваться для совместимости с предыдущими версиями.

Параметр `log_connections=on` записывает в диагностический журнал кластера попытки подключения к экземпляру, попытки аутентификации и успешную аутентификацию. Параметр может породить несколько диагностических записей в журнале, связанных с одним соединением. По умолчанию параметр отключён. Изменить значение можно только на уровне кластера, хотя в документации декларируется возможность поменять значение до установления соединения, но это неверно.

Ни на уровне роли, ни на уровне базы данных параметра не может устанавливаться:

```
postgres=# alter user alice set log_connections = 'all';
ERROR:  parameter "log_connections" cannot be set after connection start
```

Для применения значения достаточно перечитать конфигурацию:

```
alter system set log_connections = 'all';
select pg_reload_conf();
```

При попытке подсоединения к несуществующей роли по умолчанию в журнале будет строка:

```
FATAL:  role "fff" does not exist
```

При включении параметра добавятся дополнительно строки:

```
LOG:  connection received: host="10.0.2.15"
LOG:  connection authorized: user=fff database=fff application_name=psql
FATAL:  role "fff" does not exist
```

К сообщению можно добавлять атрибуты параметром `log_line_prefix`, который может быть установлен только на уровне кластера. Для изменения параметра достаточно перечитать конфигурационные файлы. По умолчанию значение параметра `'%m [%p] '` и к сообщению добавляется дата, время и номер процесса в квадратных скобках:

```
2027-01-01 11:01:01.924 MSK [1773081]
```

Добавив значение `%r` или `%h` в параметр `log_line_prefix = '%h '` можно добавить логирование IP-адреса или имени хоста клиента. IP адрес будет присутствовать в каждом сообщении:

```
10.0.2.15 FATAL:  role "fff" does not exist
```

Параметр log_disconnections

- `log_disconnections=on` записывает в диагностический журнал одно сообщение при остановке серверного процесса, обслуживавшего сессию
- изменить значение можно только на уровне кластера, а также на клиенте перед установкой соединения:

```
export PGOPTIONS="-c log_disconnections=on"
```

- ни на уровне роли, ни на уровне базы данных параметра не может устанавливаться:

```
postgres=# alter database postgres set log_disconnections = 'all';
ERROR:  parameter "log_disconnections" cannot be set after connection start
```

- пример сообщений:

```
 LOG:  disconnection: session time: 0:00:00.007 user=postgres database=postgres
host=127.0.0.1 port=34298
```


Параметр log_disconnections

Параметр `log_disconnections=on` записывает в диагностический журнал **одно** сообщение при остановке серверного процесса, обслуживавшего сессию. В сообщении присутствует длительность сессии. По умолчанию параметр отключён. Изменить значение можно на уровне кластера, а также, в отличие от параметра `log_connections`, параметр `log_disconnections` **можно** изменить перед созданием сессии на уровне сеанса:

```
export PGOPTIONS="-c log_disconnections=on -c work_mem=5MB"
psql -h 127.0.0.1 -c "show work_mem;"
 work_mem
----------
 5MB
```

Также параметр можно изменить установив свойство соединения в JDBC-драйвере.

Пример сообщения в журнале:

```
tail -n 1 $PGDATA/log/postgresql-*
LOG:  disconnection: session time: 0:00:00.007 user=postgres database=postgres
host=127.0.0.1 port=34298
```

Изменить значение может роль с атрибутом SUPERUSER или роль, которой даны привилегии для изменения параметра.

Ни на уровне роли, ни на уровне базы данных параметра не может устанавливаться:

```
postgres=# alter user alice set log_disconnections = 'all';
ERROR:  parameter "log_disconnections" cannot be set after connection start
```

Для применения значения достаточно перечитать конфигурацию:

```
alter system set log_disconnections = 'all';
select pg_reload_conf();
```

Преимущество параметра в том, что если какая-то утилита или клиент часто соединяется с базой данных, перед её запуском можно установить переменную окружения на клиентском узле и отключить логирование их сессий. Это уменьшает ненужные сообщения в диагностическом журнале кластера. Параметр `log_connections` таким образом не меняется, так как используется для логирования в целях безопасности, и отключение логирования попыток соединения на стороне клиента было бы нежелательно.

Расширения pgaudit и pgaudittofile

- расширениями `pgaudit` и `pgauditlogtofile` можно направить сообщения о создании сессий и их длительности в отдельный файл или файлы аудита
- для работы расширений нужно загрузить их библиотеки:

```
alter system set shared_preload_libraries = pgaudit, pgauditlogtofile;
```

- Расширения работают независимо и параллельно с журналом кластера и управляются собственными параметрами, которые имеют префикс "`pgaudit.`"
- параметры `pgaudit.log_connections` и `pgaudit.log_disconnections` аналогичны одноимённым параметрам PostgreSQL и могут создавать аналогичные записи **в отдельном файле аудита**

Расширения pgaudit и pgaudittofile

При использовании параметров `log_connections` и `log_disconnections` сообщения записываются в журнал кластера. При промышленной эксплуатации в этот журнал записывается множество других сообщений. Логирование соединений не нужно для повседневного анализа и замусоривает общий журнал, затрудняя чтение более важных сообщений и желательно, чтобы логирование соединений, команд ddl и других команд выполнялось в не в журнал кластера, а в отдельный файл или файлы.

В Tantor Postgres есть расширения `pgaudit` и `pgauditlogtofile`, которыми можно направить сообщения о создании сессий и их длительности в отдельный файл или файлы аудита. Расширение `pgauditlogtofile` перенаправляет записи, создаваемые расширением `pgaudit` в отдельный файл или файлы. Без него записи попадают в журнал кластера. Расширение `pgauditlogtofile` зависит от расширения `pgaudit` и не работает без него. Для использования расширений достаточно загрузить две библиотеки:

```
alter system set shared_preload_libraries = pgaudit, pgauditlogtofile;
```

Библиотеки расширений регистрируют в экземпляре параметры конфигурации, которыми можно настроить что и куда будет логироваться. Расширения работают независимо и параллельно с журналом кластера и управляются собственными параметрами, которые имеют префикс "`pgaudit.`"

В 16 версии было 18 параметров, в 18 версии уже 26 параметров. 8 параметров относятся к библиотеке `pgauditlogtofile`, в том числе параметры `pgaudit.log_connections` и `pgaudit.log_disconnections`. Эти параметры аналогичны одноимённым параметрам PostgreSQL и могут создавать аналогичные записи, но только **в отдельном файле аудита, а не в журнале кластера, в чём большое преимущество этих параметров**. Преимущество перевешивает недостатки в виде необходимости загрузки двух библиотек и неудобств их использования. Параметры библиотеки устанавливаются только на уровне кластера, указание этих параметров в переменной окружения приводит к ошибке и невозможности соединиться, в отличие от стандартных параметров:

```
export PGOPTIONS="-c pgaudit.log_connections=off"
psql
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL:  parameter "pgaudit.log_connections" cannot be changed now
```

Параметр `pgaudit.log_disconnections` , в отличие от параметра `log_disconnections` не может быть установлен при создании сессии.

Конфигурирование расширений pgaudit и pgaudittofile

- восемь параметров относятся к библиотеке `pgauditlogtofile`
- чтобы создавался журнал аудита нужно установить параметр `pgaudit.log` как минимум в значение `'misc'`
  - `'none'` - файл журнала аудита не создаётся
  - `'role'` и `'ddl'` `pgaudit.log_connections` и `pgaudit.log_disconnections` не действуют

```
postgres=# \dconfig pgaudi*
            Parameter              |    Value
-----------------------------------+----------
 pgaudit.ddl_notemp                | off
 pgaudit.dml_notemp                | off
 pgaudit.log                       | none
 pgaudit.log_autoclose_minutes     | 0
 pgaudit.log_catalog               | on
 pgaudit.log_client                | off
 pgaudit.log_connections           | off
 pgaudit.log_directory             | log
 pgaudit.log_disconnections        | off
 pgaudit.log_filename              | audit-%F.log
 pgaudit.log_format                | csv
 pgaudit.log_level                 | log
 pgaudit.log_parameter             | off
 pgaudit.log_parameter_max_size    | 0
 pgaudit.log_relation              | off
 pgaudit.log_rotation_age          | 1d
 pgaudit.log_rotation_size         | 0
 pgaudit.log_rows                  | off
 pgaudit.log_statement             | on
 pgaudit.log_statement_once        | off
 pgaudit.marking_log_directory     |
 pgaudit.marking_log_filename      | pgaudit_mark-
%Y-%m-%d.csv
 pgaudit.marking_rules_enabled     | off
 pgaudit.marking_rules_max         | 1000
 pgaudit.role                      |
 pgaudit.whitelist_role            |
(26 rows)
```


Конфигурирование расширений pgaudit и pgaudittofile

Неудобство использования параметров расширений в том, что нужно установить параметр `pgaudit.log` как минимум в значение `'misc'`, чтобы создавался журнал аудита. Но значение `'misc'` заставляет логировать команды DISCARD, FETCH, CHECKPOINT, VACUUM, SET и раздувает журнал аудита. При значении по умолчанию `'none'` файл журнала не создаётся. При установке в значения `'role'` и `'ddl'` параметры `pgaudit.log_connections` и `pgaudit.log_disconnections` не действуют.

Установка расширения pgauditlogtoile командой бесполезна, так как в расширении нет обьектов:

```
create extension pgauditlogtofile;
\dx+ pgauditlogtofile
Objects in extension "pgauditlogtofile"
--------------------
(0 rows)
```

В расширение `pgaudit` входит два триггера и две триггерные функции:

```
event trigger pgaudit_ddl_command_end
event trigger pgaudit_sql_drop
function pgaudit_ddl_command_end()
function pgaudit_sql_drop()
```

Переменная подмены `'%F'` (или её эквивалент `%Y-%m-%d`) в названии журнала аудита и журнала кластера удобнее значения по умолчанию (`%Y%m%d_%H%M`) тем, что не создаёт отдельный файл при рестарте экземпляра. Новый файл создаётся раз в сутки. Пример установки значений:

```
alter system set pgaudit.log_filename = 'audit-%F.log';
alter system set log_filename = 'postgresql-%F.log';
```

Синим цветом на слайде отмечены параметры, котоыре появились в 18 версии.

Диагностика частоты соединений с базой данных

- `log_disconnections=on` записывает в лог событие завершения сессии. Записывается та же информация, что `log_connections` плюс длительность сессии.
  - преимущество в том, что выводится одна строка, что не замусоривает лог
  - позволяет идентифицировать короткие по времени сессии
  - пример сообщения о длительности сессии сессии 4 секунды:

```
LOG: disconnection: session time: 0:00:04.056 user=oleg database=db1 host=[vm1]
```

- `log_connections=on` записывает в лог попытки установить сессию
  - недостаток в том, что неудачные попытки отличаются только дополнительной строкой:

```
LOG: connection received: host=[local]
LOG: connection authorized: user=postgres database=db2 application_name=psql
FATAL: database "db2" does not exist
LOG: connection received: host=[local]
LOG: connection authorized: user=alice database=alice application_name=psql
FATAL: role "alice" does not exist
```


Диагностика частоты соединений с базой данных

`log_disconnections=on` записывает в лог событие завершения сессии. Записывается та же информация, что `log_connections` плюс **длительность сессии**. Преимущество в том, что выводится одна строка, что не замусоривает лог. Позволяет идентифицировать короткие по времени сессии. Короткие сессии приводят к частому порождению серверных процессов, что увеличивает нагрузку и снижает производительность:

```
LOG: disconnection: session time: 0:00:04.056 user=oleg database=db1 host=[vm1]
```

В примере длительность сессии 4 секунды.

`log_connections=on` записывает в лог попытки установить сессию. Недостаток в том, что для многих типов клиентов в журнал выводится две строки: первая строка об определении способа аутентификации (без пароля, с паролем), вторая строка - аутентификация. Если не используется балансировщик соединений (pgbouncer), то до аутентификации порождается серверный процесс, это трудоемкая операция. Параметр полезен для идентификации проблем, когда клиент безостановочно пытается подсоединиться с неверным паролем или к несуществующей базе или с несуществующей ролью. Недостаток в том, что неудачные попытки отличаются только дополнительной строкой:

```
LOG: connection received: host=[local]
LOG: connection authorized: user=postgres database=db2 application_name=psql
FATAL: database "db2" does not exist
LOG: connection received: host=[local]
LOG: connection authorized: user=alice database=alice application_name=psql
FATAL: role "alice" does not exist
```

`log_hostname=off`. Включать не стоит, так как вносит существенные задержки при логировании создания сессии.

Диагностика блокирующих ситуаций

- `log_lock_waits=true`. По умолчанию отключен. Рекомендуется включить, чтобы получать сообщения в диагностический журнал что какой-либо процесс ждет дольше, чем: `deadlock_timeout`
- `deadlock_timeout='60s'`. По умолчанию 1 секунда, что слишком мало и на нагруженных экземплярах создает значительные издержки
- `log_startup_progress_interval='10s'` не стоит отключать
- `log_recovery_conflict_waits=on`. По умолчанию `off`. Процесс `startup` запишет сообщение в лог реплики, если не сможет применить WAL к реплике дольше, чем `deadlock_timeout`

```
LOG: recovery still waiting after 60.555 ms: recovery conflict on lock
DETAIL: Conflicting process: 5555.
```

- Наличие конфликтов можно увидеть в представлении (мало деталей):

```
select * from pg_stat_database_conflicts where datname='postgres';
datid|datname |tblspc|confl_lock|confl_snapshot|confl_bufferpin|deadlock
-----+--------+------+----------+--------------+---------------+--------
13842|postgres|    0 |        0 |            1 |             1 |      0
```


Диагностика блокирующих ситуаций

`log_lock_waits=true`. Включен, по умолчанию, в 19 версии. Рекомендуется включить, чтобы получать сообщения в диагностический журнал что какой-либо процесс ждет дольше, чем: `deadlock_timeout='60s'`. По умолчанию 1 секунда, что слишком мало и на нагруженных экземплярах создает значительные издержки. Рекомендуется настраивать значение `deadlock_timeout` так, чтобы сообщения об ожиданиях получения блокировки возникали редко. Как первое приближение можно ориентироваться на длительность типичной транзакции (для реплики - самый долгий запрос).

В 15 версии появился параметр `log_startup_progress_interval='10s'` который **не стоит отключать** (устанавливать ноль). Если процесс `startup` (выполняет восстановление), столкнется с длительной операцией, то в лог будет записано сообщение об этой операции. Сообщения позволят выявить либо проблемы с файловой системой, либо высокую нагрузку на дисковую систему. Пример сообщений процесса startup при восстановлении:

```
LOG: syncing data directory (fsync), elapsed time: 10.07 s, current path: ./base/4/2658
LOG: syncing data directory (fsync), elapsed time: 20.16 s, current path: ./base/4/2680
LOG: syncing data directory (fsync), elapsed time: 30.01 s, current path: ./base/4/PG_VERSION
```

`log_recovery_conflict_waits=on`. По умолчанию `off`. **Параметр появился в 14 версии.** Процесс `startup` запишет сообщение в лог реплики, если не сможет применить WAL к реплике дольше, чем `deadlock_timeout`. Задержка может произойти из-за того, что серверный процесс на реплике выполняет команду или транзакцию (для повторяемости по чтению) и блокирует применение WAL из-за параметра `max_standby_streaming_delay` (по умолчанию 30s). Позволяет идентифицировать случаи отставания реплики. Действует на реплике, на мастере можно установить заранее. Рекомендуется установить в значение `on`.

```
LOG: recovery still waiting after 60.555 ms: recovery conflict on lock
DETAIL: Conflicting process: 5555.
CONTEXT: WAL redo at 0/3044D08 for Heap2/PRUNE: latestRemovedXid 744 nredirected
0 ndead 1; blkref #0: rel 1663/13842/16385, blk 0
```

Наличие конфликтов можно увидеть в представлении, но в нем мало деталей:

```
select * from pg_stat_database_conflicts where datname='postgres';
datid|datname |tblspc|confl_lock|confl_snapshot| confl_bufferpin|deadlock
-----+--------+------+----------+--------------+----------------+--------
13842|postgres|    0 |        0 |            1 |              1 |      0
```

Практика

1. Какая информация попадает в журнал
2. Расположение журналов сервера
3. Как информация попадает в журнал
4. Добавление формата csv
5. Включение коллектора сообщений

Практика

Какая информация попадает в журнал
Расположение журналов сервера
Как информация попадает в журнал
Добавление формата csv
Включение коллектора сообщений

6

Безопасность

Пользователи (роли) в кластере баз данных

- роль синоним пользователя
- общий объект кластера
- у объектов может быть только одна роль-владелец
- у ролей нет роли-владельца
- ролям можно предоставлять привилегии на объекты
- у ролей есть атрибуты
- разница между командами CREATE USER и CREATE ROLE:

```
postgres=# create user alice;
CREATE ROLE
postgres=# create role bob;
CREATE ROLE
postgres=# \du
                                 List of roles
 Role name |                         Attributes
-----------+------------------------------------------------------------
 alice     |
 bob       | Cannot login
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS
```


Пользователи (роли) в кластере баз данных

В PostgreSQL role ("роль"), то же самое что пользователь (user). Роль - общий объект кластера. Это означает, что после создания роль видна в любой базе данных в этом кластере. Роль аналогична группе в других системах безопасности.

У большинства объектов (таблицы, процедуры, функции, базы данных, схемы и т.п.) должна иметься одна роль-владелец этого объекта. Пока у роли есть объекты во владении, нельзя удалить роль. Владельца объекта можно поменять.

Роли могут иметь привилегии (права) на объекты. Например, привилегия на создание объектов в схеме, привилегия вставлять строки в таблицу, выполнять процедуру. Привилегии в PostgreSQL - аналоги объектных привилегий в Oracle Database.

У ролей есть девять атрибутов (свойств). Атрибуты можно поменять после создания роли. Роль можно переименовать. Атрибуты можно уподобить системным или административным привилегиям (привилегии на выполнение действий без привязки к объекту) в Oracle Database. Например, атрибут SUPERUSER схож с административной привилегией SYSDBA в Oracle Database, а атрибут BYPASSRLS схож с системной привилегией EXEMPT ACCESS POLICY.

Роли и схемы - разные объекты. Схемы - локальные объекты базы данных, роли - общие объекты кластера.

Роли создаются командой CREATE ROLE или CREATE USER, удаляются DROP ROLE, меняются ALTER ROLE.

Отличие CREATE USER от CREATE ROLE в том, что первая команда **по умолчанию** устанавливает атрибут LOGIN, а вторая NOLOGIN:

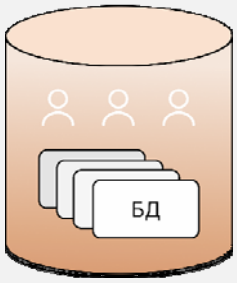
```
postgres=# create user alice;
CREATE ROLE
postgres=# create role bob;
CREATE ROLE
postgres=# \du
                                 List of roles
 Role name |                         Attributes
-----------+------------------------------------------------------------
 alice     |
 bob       | Cannot login
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS
```

https://docs.tantorlabs.ru/tdb/ru/18_1/se/database-roles.html

Пользователи (роли)

- Список ролей можно посмотреть командами **\duS \dgS**
- Список ролей хранится в глобальной таблице **pg_authid**
- Есть псевдороль public, в которую входят все роли
  - › ей можно выдавать привилегии
- Атрибуты роли postgres:

```
postgres=# select * from pg_authid where rolname='postgres'\gx
-[ RECORD 1 ]--+------------------------------
oid            | 10
rolname        | postgres
rolsuper       | t
rolinherit     | t
rolcreaterole  | t
rolcreatedb    | t
rolcanlogin    | t
rolreplication | t
rolbypassrls   | t
rolconnlimit   | -1
rolpassword    | SCRAM-SHA-256$4096:oejDqb5w...
rolvaliduntil  |
```



Пользователи (роли)

Список ролей кластера можно посмотреть командой \duS или \dgS (u - user, g - group) или в таблице pg_authid или представлении pg_roles:

```
postgres=# \duS
                                   List of roles
          Role name          |   Attributes
-----------------------------+------------------------------
 pg_checkpoint               | Cannot login
 pg_create_subscription      | Cannot login
 pg_database_owner           | Cannot login
 pg_execute_server_program   | Cannot login
 pg_maintain                 | Cannot login
 pg_monitor                  | Cannot login
 pg_read_all_data            | Cannot login
 pg_read_all_settings        | Cannot login
 pg_read_all_stats           | Cannot login
 pg_read_server_files        | Cannot login
 pg_signal_backend           | Cannot login
 pg_stat_scan_tables         | Cannot login
 pg_use_reserved_connections | Cannot login
 pg_write_all_data           | Cannot login
 pg_write_server_files       | Cannot login
 postgres                    | Superuser, Create role, Create DB, Replication, Bypass RLS
postgres=# select * from pg_authid where rolname='postgres'\gx
-[ RECORD 1 ]--+------------------------------
oid            | 10
rolname        | postgres
rolsuper       | t
rolinherit     | t
rolcreaterole  | t
rolcreatedb    | t
rolcanlogin    | t
rolreplication | t
rolbypassrls   | t
rolconnlimit   | -1
rolpassword    | SCRAM-SHA-256$4096:oejDqb5wqdHc...
rolvaliduntil  |
```

Также имеется псевдороль public, в которую входят все роли кластера:

```
postgres=# drop role public;
ERROR:  cannot use special role specifier in DROP ROLE
postgres=# create role public;
ERROR:  role name "public" is reserved
```

Атрибуты (параметры, свойства) пользователей

- есть 7 атрибутов типа включён/выключен
- LOGIN - право создания начального соединения с базами данных
- SUPERUSER - обходит проверки прав доступа, кроме начального соединения
- REPLICATION LOGIN - (без атрибута LOGIN атрибут REPLICATION бесполезен) имеет право резервировать весь кластер
- CREATEROLE - может создавать новые роли
  - у ролей нет владельца
  - созданная роль грантуется создающей роли с опцией ADMIN OPTION, но без SET и без INHERIT, поэтому ни грант WITH ADMIN, ни атрибут CREATEROLE не позволяют поднять своей роли привилегии путем создания и переключения в созданную роль
- Грант с ADMIN OPTION не даёт право менять атрибуты CREATEROLE, BYPASSRLS, REPLICATION, CREATEDB, SUPERUSER
  - эти атрибуты роль сможет менять у ролей, на которые есть ADMIN OPTION, только, если у неё имеется такой же атрибут

Атрибуты (параметры, свойства) пользователей

LOGIN - право создания начального соединения с базами данных. Соединившись, находясь в сессии с базой данных, можно переключиться в этой сессий в грантованную роль командой SET ROLE (вернуться в начальную роль команда RESET ROLE). У роли, в которую переключаются в сессии может не быть атрибута LOGIN. Сменить базу данных путем переключения в другую роль нельзя, это возможно только создав новое соединение (в psql команда `\connect`).

SUPERUSER - обходит проверки прав доступа, кроме начального соединения. Без атрибута LOGIN роль с атрибутом SUPERUSER не может соединиться ни с одной базой данных.

CREATEDB - роль может создавать базы данных. Создав базу данных, роль станет владельцем созданной базы и сможет удалить эту базу. Удалить базу может только владелец или роль с атрибутом SUPERUSER.

REPLICATION LOGIN - роль с этими атрибутами (без атрибута LOGIN атрибут REPLICATION бесполезен) имеет право подсоединяться по протоколу репликации и резервировать весь кластер.

CREATEROLE - роль может создавать роли. У ролей нет владельца. Созданная роль грантуется создающей с опцией ADMIN OPTION. Эта опция позволяет изменять атрибуты (пароль, INHERIT, CONNECTION LIMIT, VALID UNTIL), переименовывать, удалять грантованную с этой опцией роль, грантовать и отзывать эту роль у других, менять параметры конфигурации, которые устанавливаются на уровне роли (команда ALTER ROLE имя SET work_mem = '16MB'), менять описание роли командой COMMENT, менять SECURITY LABEL этой роли. Поэтому по умолчанию роль с атрибутом CREATEROLE может изменять и удалять созданные ей роли. Если у роли появится атрибут SUPERUSER, её смогут удалять или менять свойства только роли с атрибутом SUPERUSER. Грант с ADMIN OPTION не даёт право менять атрибуты CREATEROLE, BYPASSRLS, REPLICATION, CREATEDB, SUPERUSER. Эти атрибуты роль сможет менять у ролей, на которые у неё есть ADMIN OPTION, только, если у неё имеется такой же атрибут. Запомнить эти правила сложно. Можно считать, что ни грант WITH ADMIN, ни атрибут CREATEROLE не позволяют поднять своей роли привилегии путем создания и переключения в созданную роль.

BYPASSRLS - на роль с этим атрибутом не действуют политики Row Level Security.

CONNECTION LIMIT - число сессий (начальных соединений). По умолчанию, число сессий не ограничено (значение -1).

VALID UNTIL '2027-11-01' - сорок действия пароля типа timestamp with time zone.

Параметр конфигурации createrole_self_grant

- По умолчанию, создатель не получает SET и INHERIT, только ADMIN

```
postgres@postgres=# create user alice createrole;
CREATE ROLE
postgres@postgres=# grant all on schema public to alice with grant option;
GRANT
postgres@postgres=# \c postgres alice
You are now connected to database "postgres" as user "alice".
alice@postgres=> create user bob createrole;
CREATE ROLE
alice@postgres=> set role bob;
ERROR:  permission denied to set role "bob"
alice@postgres=> grant all on schema public to bob;
GRANT
alice@postgres=> drop user bob;
ERROR:  role "bob" cannot be dropped because some objects depend on it
DETAIL:  privileges for schema public
alice@postgres=> revoke all on schema public from bob;
REVOKE
alice@postgres=> drop user bob;
DROP ROLE
```


Параметр конфигурации createrole_self_grant

Начиная с 16 версии, когда пользователь с атрибутом CREATEROLE создаёт новую роль, она ему выдаётся с опцией WITH ADMIN, что позволяет удалять созданную роль. У роли нет владельца. Атрибут CREATEROLE не позволяет удалять другие роли, а атрибута DROPROLE нет.

Однако, на созданную роль не выдается права SET и INHERIT и создатель не может ни переключиться в созданную роль, ни пользоваться привилегиями созданной роли.

В примере на слайде **alice** создала пользователя **bob**, но **не смогла в него переключиться** и не может пользоваться его привилегиями.

Это не защищает от случаев, когда администратор дал бы созданной не им роли привилегию или атрибут и создатель этой роли смог бы ими воспользоваться, так как параметр `createrole_self_grant` (появился в 16 версии https://github.com/postgres/postgres/commit/e5b8a4c0) не требует для установки прав суперпользователя:

```
alice@postgres=> set createrole_self_grant = 'SET, INHERIT';
SET
alice@postgres=> create user bob createrole;
CREATE ROLE
alice@postgres=> set role bob;
SET
```

Посмотреть свойства, с которыми выданы роли можно запросом:

```
bob@postgres=> select roleid::regrole, member::regrole, grantor::regrole,
admin_option, inherit_option, set_option from pg_auth_members where roleid =
'bob'::regrole;
 roleid | member | grantor  | admin_option | inherit_option | set_option
--------+--------+----------+--------------+----------------+------------
 bob    | alice  | postgres | t            | f              | f
 bob    | alice  | alice    | f            | t              | t
(2 rows)
```

Начиная с 18 версии выданные привилегии препятствуют удалению пользователя, который их имеет. Для удаления придётся отозвать привилегии. Зависимости хранятся в таблице `pg_shdepend`.

Выданные пользователям привилегии

- Зависимости объектов хранятся в `pg_shdepend` и `pg_depend`
- Функция `pg_get_acl()` выдаёт список прав доступа

```
\pset xheader_width 20
Expanded header width is 20.
select pg_describe_object(classid, objid, objsubid) obj,
pg_describe_object(refclassid, refobjid, 0) ref, pg_get_acl(classid,
objid, objsubid) acl from pg_shdepend\gx
-[ RECORD 1 ]-------
obj | schema public
ref | role alice
acl | {pg_database_owner=UC/pg_database_owner,=U/pg_database_owner,
alice=U*C*/pg_database_owner, bob=UC/pg_database_owner}
-[ RECORD 2 ]-------
obj | schema public
ref | role bob
acl | {pg_database_owner=UC/pg_database_owner,=U/pg_database_owner,
alice=U*C*/pg_database_owner, bob=UC/pg_database_owner}
```


Выданные пользователям привилегии

Начиная с 18 версии выданные привилегии препятствуют удалению пользователя, который их имеет. Для удаления придётся отозвать привилегии.

Зависимости хранятся в таблице `pg_shdepend`. Таблица хранит зависимости, если в них участвуют общие объекты кластера, к которым относятся роли. Зависимости локальных объектов хранятся в таблице `pg_depend`.

В таблицах хранятся OID и для получения названий удобно использовать функции `pg_describe_object()` и `pg_identify_object()`.

Функция `pg_get_acl()` выдаёт список прав доступа к объекту (появилась в 18 версии).

Запрос, приведённый в документации для получения списка прав на объекты в текущей базе данных:

```
SELECT distinct (pg_identify_object(s.classid,s.objid,s.objsubid)).*,
pg_catalog.pg_get_acl(s.classid,s.objid,s.objsubid) AS acl
FROM pg_catalog.pg_shdepend AS s
JOIN pg_catalog.pg_database AS d ON d.datname = current_database() AND d.oid =
s.dbid JOIN pg_catalog.pg_authid AS a ON a.oid = s.refobjid AND s.refclassid =
'pg_authid'::regclass WHERE s.deptype = 'a'\gx
-[ RECORD 1 ]-------
type     | schema
schema   |
name     | public
identity | public
acl      |
{pg_database_owner=UC/pg_database_owner,=U/pg_database_owner,alice=U*C*/pg_databa
se_owner,bob=UC/pg_database_owner}
```

В примере пользователям **alice** и **bob** выданы привилегии на схему **public**.

Неудобное отслеживание зависимостей при удалении объектов сподвигает не использовать запутанные наборы прав доступа. Сложная структура прав ухудшает наблюдение за привилегиями, что считается фактором, снижающим защищенность.

https://docs.tantorlabs.ru/tdb/ru/18_1/be/functions-info.html#FUNCTIONS-INFO-OBJECT

Атрибут INHERIT и GRANT WITH INHERIT

- Атрибут INHERIT устанавливается по умолчанию
- Позволяет наследовать права, выданные на объекты баз данных
- Если роли установить атрибут NOINHERIT, то она не будет наследовать привилегии ролей, которые ей даны
- При выдаче роли можно явно задать
  - будут ли работать права грантованной роли у сессий грантуемого
  - может ли грантуемый переключиться в грантованную роль

```
postgres=# grant postgres to alice with inherit false, set true;
GRANT ROLE
postgres=# \c postgres alice
You are now connected to database "postgres" as user "alice".
postgres=> set role postgres;
SET
postgres=#
postgres=# grant postgres to bob with inherit true, set false;
GRANT ROLE
postgres=# \c postgres bob
You are now connected to database "postgres" as user "bob".
postgres=> set role postgres;
ERROR:  permission denied to set role "postgres"
```


Атрибут INHERIT и GRANT WITH INHERIT

Атрибут INHERIT устанавливается по умолчанию. Если роли установить атрибут NOINHERIT, то она не будет наследовать права на конкретные объекты базы данных ролей, которые ей грантованы и ей надо будет переключиться в грантованные роли, чтобы работать с их правами на объекты. Если роли установить атрибут NOINHERIT, то она перестанет по умолчанию наследовать права на объекты, которые есть у грантованных ей ролям (членом которых она является). Однако, это можно переопределить, указав опцию WITH INHERIT явно при выдачи роли командой GRANT ... WITH INHERIT true или WITH INHERIT false. Пример:

```
postgres=# grant postgres to alice with inherit false, set true;
GRANT ROLE
postgres=# \connect postgres alice
You are now connected to database "postgres" as user "alice".
postgres=> set role postgres;
SET
postgres=# select current_user,session_user,current_role,user,system_user;
 current_user | session_user | current_role |   user   | system_user
--------------+--------------+--------------+----------+-------------
 postgres     | alice        | postgres     | postgres |
postgres=# grant postgres to bob with inherit true, set false;
GRANT ROLE
postgres=# \connect postgres bob
You are now connected to database "postgres" as user "bob".
postgres=> set role postgres;
ERROR:  permission denied to set role "postgres"
```

Опция SET false не позволяет переключиться в роль и получить **право пользоваться её атрибутами** (например, SUPERUSER).

Атрибуты LOGIN, CREATEROLE, BYPASSRLS, REPLICATION, CREATEDB, SUPERUSER никогда не наследуются. Чтобы ими воспользоваться, необходимо переключиться на роль, имеющую этот атрибут, с помощью команды `SET ROLE`. Вернуться в исходную роль, с которой была создана сессия можно командами:

```
RESET ROLE; SET ROLE NONE; SET ROLE исходная_роль;
```

Функция **system_user** появилась в 16 версии, выдаёт имя внешнего пользователя или null.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/role-membership.html

Переключение сессии в другую роль и смена ролей

- `SET [SESSION | LOCAL] SESSION AUTHORIZATION роль;` переключает сессию в другую роль
- Команду может выполнить только суперпользователь
  - используется для того, чтобы суперпользователь мог переключить сессию в другого пользователя и затем вернуться в исходную
- Текущего пользователя можно изменить командой `SET ROLE`
  - проверка прав на объекты выполняется для текущего пользователя

```
postgres=# set session authorization alice;
postgres=> select current_user, session_user, user;
 current_user | session_user | user
--------------+--------------+-------
 alice        | alice        | alice
postgres=> set role bob;
postgres=> select current_user, session_user, user;
 bob          | alice        | bob
postgres=> reset role;
postgres=> select current_user, session_user, user;
 alice        | alice        | alice
postgres=> reset session authorization;
postgres=# select current_user, session_user, user;
 postgres     | postgres     | postgres
```


Переключение сессии в другую роль и смена ролей

Команда `SET [SESSION | LOCAL] SESSION AUTHORIZATION роль;` переключает сессию в другую роль. `LOCAL` используется только в открытой транзакции и переключает сессию до окончания транзакции.

Команду можно выполнить только, если сессия была изначально создана (аутентифицирована) суперпользователем. Эта команда может использоваться для того, чтобы суперпользователь мог переключить сессию в другого пользователя и затем вернуться в исходную сессию суперпользователя.

```
postgres=# set session authorization alice;
postgres=> select current_user, session_user, user;
 current_user | session_user | user
--------------+--------------+------
 alice        | alice        | alice
postgres=> set role bob;
postgres=> select current_user, session_user, user;
 bob          | alice        | bob
postgres=> set role pg_checkpoint;
postgres=> select current_user, session_user, user;
 pg_checkpoint | alice        | pg_checkpoint
postgres=> reset role;
postgres=> select current_user, session_user, user;
 alice        | alice        | alice
postgres=> reset session authorization;
postgres=# select current_user, session_user, user;
 postgres     | postgres     | postgres
```

`SET SESSION AUTHORIZATION` нельзя использовать в функции `SECURITY DEFINER`.

Текущего пользователя можно изменить командой `SET ROLE`. Проверка прав на объекты выполняется для **текущего пользователя.** `SET ROLE` переключит в любую роль, прямым или косвенным членом которой является роль, под которой была выполнена аутентификация.

Названия функций `current_user, current_role, user` являются синонимами. Эти функции вызываются без круглых скобок по стандарту SQL.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/functions-info.html
https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-set-session-authorization.html

Предопределённые (служебные) роли

- появились в PostgreSQL начиная с версии 14
- 15 ролей плюс псевдороль public
- может грантовать роль с атрибутами SUPERUSER или роль, имеющая право WITH ADMIN на грантуемую роль
- удалить служебные роли нельзя
- полезны чтобы не давать привилигею суперпользователя для выполнения востребованных действий

```
postgres=# \duS
                              List of roles
          Role name           |   Attributes
------------------------------+---------------
 pg_checkpoint                | Cannot login
 pg_create_subscription       | Cannot login
 pg_database_owner            | Cannot login
 pg_execute_server_program    | Cannot login
 pg_maintain                  | Cannot login
 pg_monitor                   | Cannot login
 pg_read_all_data             | Cannot login
 pg_read_all_settings         | Cannot login
 pg_read_all_stats            | Cannot login
 pg_read_server_files         | Cannot login
 pg_signal_backend            | Cannot login
 pg_stat_scan_tables          | Cannot login
 pg_use_reserved_connections  | Cannot login
 pg_write_all_data            | Cannot login
 pg_write_server_files        | Cannot login
```


Предопределённые (служебные) роли

До 14 версии предопределённых (predefined), то есть служебных, автоматически создаваемых при создании кластера, не было. Была только служебная роль public, которая включает в себя всех пользователей (ролей) кластера. Удалить эти служебные роли нельзя:

```
postgres=# drop role pg_checkpoint;
ERROR: cannot drop role pg_checkpoint because it is required by the database system
```

Эти роли может грантовать роль с атрибутами SUPERUSER или роль, имеющая право WITH ADMIN на грантуемую роль.

Единственным членом роли `pg_database_owner` всегда является текущая роль-владелец базы данных. `pg_database_owner` может владеть объектами и получать права на объекты. Имеет смысл давать права этой роли и делать её владельцем объектов, так как при клонировании базы или смене владельца базы не нужно будет менять привилегии и владение. Права, данные `pg_database_owner` (например в базе template1), приобретёт создатель новой базы, который её клонирует. По умолчанию, владеет схемой `public`, то есть владелец базы данных управляет использованием схемы `public` в своей базе.

`pg_signal_backend` имеет право выполнять функции `pg_cancel_backend(pid)` и `pg_terminate_backend(pid)`, которые прерывают выполнение команд или сессий, кроме сессий суперпользователей.

`pg_read_server_files`, `pg_write_server_files`, `pg_execute_server_program` дают право доступа к файлам и запуска программ под пользователем операционной системы, из под которого запущен экземпляр (postgres). Например, поменять содержимое файла `pg_hba.conf` или стереть файлы в директории `PGDATA`.

`pg_monitor`, `pg_read_all_settings`, `pg_read_all_stats` и `pg_stat_scan_tables` даются ролям для мониторинга и настройки производительности.

`pg_checkpoint` имеет право выполнять команду checkpoint;

`pg_maintain` имеет право выполнять команды `VACUUM`, `ANALYZE`, `CLUSTER`, `REFRESH MATERIALIZED VIEW`, `REINDEX`, `LOCK TABLE` для всех объектов, как будто имеет права `MAINTAIN` на эти объекты.

`pg_read_all_data`, `pg_write_all_data` имеют право читать и менять данные всех объектов (таблиц, представлений, последовательностей), как будто имеет права `SELECT`, `INSERT`, `UPDATE`, `DELETE` на эти объекты и права `USAGE` на все схемы.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/predefined-roles.html

Права на объекты

- При создании объекта назначается владелец
- По умолчанию владелец и суперпользователь имеют права на созданный объект
- Другим ролям нужны права
- Для каждого типа объекта есть свой набор прав
  - названия прав приведены в таблице
- Права выдаются и отзываются командами GRANT и REVOKE

право		типы объектов
SELECT	r	LARGE OBJECT, SEQUENCE, TABLE (и подобные им), COLUMN
INSERT	a	TABLE, столбец
UPDATE	w	LARGE OBJECT, SEQUENCE, TABLE, COLUMN
DELETE	d	TABLE
TRUNCATE	D	TABLE
REFERENCES	x	TABLE, COLUMN
TRIGGER	t	TABLE
CREATE	C	DATABASE, SCHEMA, TABLESPACE
CONNECT	c	DATABASE
TEMPORARY	T	DATABASE
EXECUTE	X	FUNCTION, PROCEDURE
USAGE	U	DOMAIN, FDW, FOREIGN SERVER, LANGUAGE, SCHEMA, SEQUENCE, TYPE
SET	s	PARAMETER
ALTER SYSTEM	A	PARAMETER
MAINTAIN	m	TABLE

Права на объекты

При создании объекта, ему назначается владелец. Владельцем становится та роль, чьи права использовались для создания объекта. Это может быть **current_user** - текущая роль, в которой работает сессия или унаследованная роль (которая была дана с **WITH INHERIT true**). Для большинства типов объектов по умолчанию владелец и суперпользователи имеют права на созданный объект. Например, право удалить этот объект.

Право изменять или удалять объект является неотъемлемым правом владельца объекта, его нельзя лишиться или передать другому. Это право, как и другие, наследуют роли, которым была выдана (GRANT) роль-владелец. Владельца объекта можно поменять. Это может делать суперпользователь или текущий владелец объекта командой ALTER, но только, если владелец может переключиться в роль нового владельца. Пример:

```
postgres=# alter database demo owner to bob;
ALTER DATABASE
postgres=# alter database demo owner to public;
ERROR:  role "public" does not exist
postgres=# revoke ALL on database demo from public;
REVOKE
postgres=# revoke connect on database demo from public;
REVOKE
```

Команда REVOKE не выдаёт ошибки, если отзываемой привилегии не было.

Псевдороль public нельзя назначить владельцем базы данных.

Владелец объекта может лишить себя прав на свой объект. Однако, владелец может управлять правами и предоставить себе права снова.

Чтобы разрешить использовать объект другим ролям, нужно дать им права на этот конкретный объект ("объектные привилегии").

Права выдаются (представляются) и отзываются командами GRANT и REVOKE.

Для каждого типа объекта (база данных, табличное пространство, параметр конфигурации, таблица, функция, последовательность и т.п.) - свой набор прав.

Ключевые слова, которые используются в командах GRANT и REVOKE: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE, USAGE, SET, ALTER SYSTEM, MAINTAIN.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/ddl-priv.html
https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-grant.html

Просмотр прав на объекты в psql

- Права выводятся в виде списка:
- кому_дана=привилегии/кто_дал
- Если перед значком "=" пусто, это означает public (предоставлено всем)
- Значок "*" после буквы означает, что право предоставлено с правом передачи WITH GRANT OPTION.
- Значок "+" в конце обозначает, что это не последний элемент и список продолжается на следующей строке

```
postgres=# \l
                  List of databases
   Name |  Owner   |   Access privileges
--------+----------+-----------------------
 demo   | postgres | postgres=CTc/postgres+
        |          | alice=C*c/postgres
postgres=# GRANT ALL PRIVILEGES ON ALL TABLES IN
SCHEMA public TO alice, bob WITH GRANT OPTION;
GRANT
```

Тип объекта	Все права	Права PUBLIC	Команда psql
DATABASE	CTc	Tc	\l
DOMAIN	U	U	\dD+
FUNCTION, PROCEDURE	X	X	\df+
FDW	U	нет	\dew+
FOREIGN SERVER	U	нет	\des+
LANGUAGE	U	U	\dL+
LARGE OBJECT	rw	нет	\dl+
PARAMETER	sA	нет	\dconfig+
SCHEMA	UC	нет	\dn+
SEQUENCE	rwU	нет	\dp
TABLE (и подобные им)	arwdDxtm	нет	\dp
COLUMN	arwx	нет	\dp
TABLESPACE	C	нет	\db+
TYPE	U	U	\dT+

Просмотр прав на объекты в psql

Список команд psql приведён в таблице на слайде. Например, для баз данных:

```
postgres=# \l
                  List of databases
   Name    |  Owner   | Encoding | .. |   Access privileges
-----------+----------+----------+----+-----------------------
 demo      | postgres | UTF8     | .. | postgres=CTc/postgres+
           |          |          | .. | alice=C*c/postgres
```

Права выводятся в виде списка элементов ("aclitem"), где каждый элемент обозначает:
кому_дана=привилегии/кто_дал

Если перед значком "=" пусто, то это означает public - предоставлены всем.

Значок "*" **после** буквы означает, что право предоставлено с правом передачи (**WITH GRANT OPTION**).

Значок "+" в конце обозначает, что это не последний элемент и список продолжается на следующей строке.

Пример выдачи привилегий:

```
postgres=# GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO alice, bob
WITH GRANT OPTION GRANTED BY postgres;
```

GRANTED BY нет смысла указывать, так как можно указать только текущего пользователя. Выдача привилегии от имени другого пользователя не реализована и присутствует для совместимости со стандартом SQL.

WITH GRANT OPTION даёт право роли-получателю давать получаемые права другим ролям. Псевдороли public нельзя дать право с GRANT OPTION.

Права удалять объект (DROP) нет, так как отозвать или дать его нельзя, оно принадлежит роли-владельцу объекта.

ALL PRIVILEGES или сокращённо ALL означает, что выдаются все допустимые для типа объекта привилегии.

Псевдороли public по умолчанию даются привилегии на базы данных (Temporary - создавать временные таблицы и другие временные объекты, **connect** - подсоединяться), **подпрограммы (eXecute** - выполнять), языки (**Usage** - создавать подпрограммы), типы данных, домены в момент создания объекта.

Привилегии по умолчанию (DEFAULT PRIVILEGES)

- `ALTER DEFAULT PRIVILEGES` позволяет задавать права, применяемые к объектам, которые будут создаваться в будущем для схем, таблиц, представлений, внешних таблиц, последовательностей, подпрограмм, типов (включают в себя домены)
- роль `public` получает права:
  - CONNECT и TEMPORARY (создание временных таблиц) для баз данных
  - EXECUTE для функций и процедур
  - USAGE для языков, типов данных, доменов

```
alter default privileges REVOKE ALL on routines from public;
alter default privileges REVOKE ALL on types from public;
revoke all on database demo from public;
revoke connect on database p2 from public;
\l
                List of databases
   Name    |  Owner   | .. |   Access privileges
-----------+----------+----+-----------------------
 demo      | postgres | .. | postgres=CTc/postgres+
           |          | .. | alice=c/postgres
 p2        | postgres | .. | =T/postgres           +
           |          | .. | postgres=CTc/postgres
revoke all on language plpgsql from public;
\dL+
                List of languages
  Name   |  Owner   | Trusted | .. |  Access privileges
---------+----------+---------+----+--------------------
 plpgsql | postgres | t       | .. | postgres=U/postgres
(1 row)
```


Привилегии по умолчанию (DEFAULT PRIVILEGES)

Команда `ALTER DEFAULT PRIVILEGES` позволяет задавать права, применяемые к объектам, которые будут создаваться в будущем. Команда не меняет права, назначенные существующим объектам. Можно задать **DEFAULT PRIVILEGES** для схем, таблиц, представлений, внешних таблиц, последовательностей, подпрограмм, типов (включают в себя домены). Задать **DEFAULT PRIVILEGES** для функций и процедур по отдельности нельзя: FUNCTIONS и ROUTINES для команды считаются равнозначными.

Вспомним, что роль `public` получает права: CONNECT и TEMPORARY (создание временных таблиц) для баз данных; EXECUTE для функций и процедур; USAGE для языков, типов данных, доменов. Владелец объекта может отозвать (REVOKE) эти права. Более удобно использовать команду `ALTER DEFAULT PRIVILEGES` для автоматического выполнения команды REVOKE, отзывающей у роли public привилегии сразу после создания подпрограммы и типа (распространяется на домены):

```
alter default privileges REVOKE ALL on routines from public;
alter default privileges REVOKE ALL on types from public;
```

Команда `ALTER DEFAULT PRIVILEGES` может выполнить не только команду отзыва, но и выдачи привилегий при создании объекта.

Для отзыва привилегий на базы данных и языки придется использовать команду REVOKE:

```
revoke all on database demo from public;
revoke connect on database p2 from public;
\l                List of databases
   Name    |  Owner   | .. |   Access privileges
-----------+----------+----+-----------------------
 demo      | postgres | .. | postgres=CTc/postgres+
           |          | .. | alice=c/postgres
 p2        | postgres | .. | =T/postgres           +
           |          | .. | postgres=CTc/postgres
revoke all on language plpgsql from public;
\dL+              List of languages
  Name   |  Owner   | Trusted | .. |  Access privileges
---------+----------+---------+----+--------------------
 plpgsql | postgres | t       | .. | postgres=U/postgres
```

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-alterdefaultprivileges.html

Защита на уровне строк (Row-level security, RLS)

- по умолчанию отключена
- RLS не является мандатным контролем доступа (MAC)
- если RLS включается, а разрешающих политик нет, то доступ запрещён
- политики создаются командой CREATE POLICY, например:
  `CREATE POLICY имя ON таблица AS PERMISSIVE FOR ALL TO роль USING (предикат);`
  - политик на таблице может быть несколько, они могут быть PERMISSIVE и/или RESTRICTIVE и могут комбинироваться AND и OR
- RLS включают на уровне таблиц командой:
  `ALTER TABLE имя [ENABLE | DISABLE | FORCE |NO FORCE ] ROW LEVEL SECURITY;`
- включение с опцией ENABLE:
  - RLS действуют на всех, кроме владельца и ролей с атрибутом SUPERUSER или BYPASSRLS
- RLS не применяется к проверкам ограничений целостности
  - ограничения целостности не могут нарушаться

Защита на уровне строк (Row-level security, RLS)

Защита на уровне строк по умолчанию отключена. Задает предикат (условие) по которому доступ к строкам ограничивается для пользователей. В Oracle Database аналогичная опция называется Fine-Grained Access Control (FGAC, детализированный контроль доступа), управляемый пакетом процедур с названием DBMS_RLS. Это одна из "опций для опций", так как аналогичный функционал можно реализовать с помощью представлений (views), что более просто и производительнее. RLS не является мандатным контролем доступа (Mandatory access control, MAC), который ограничивает доступ с помощью меток на каждой строке. В Oracle Database опция, аналогичная MAC, называется Label Security, которая появилась в 1998 году в версии 8i. MAC не добавляет функционала и ухудшает производительность, используется там, где нужна реализация формальных требований "защиты" данных. RLS и MAC действуют в дополнение к обычным правам доступа (Discretionary access control, DAC). Если обычных прав доступа к схеме и таблице нет, то доступа к таблице не будет.

Сначала создают политики командой CREATE POLICY. Например:

```
CREATE POLICY имя ON таблица AS PERMISSIVE FOR ALL TO роль USING (предикат);
```

Функции в предикате выполняются с правами пользователя, выполняющего запрос.

Политик может быть несколько, они могут быть PERMISSIVE и/или RESTRICTIVE и могут комбинироваться AND и OR.

Дальше RLS включают на уровне таблиц командой:

```
ALTER TABLE имя [ENABLE | DISABLE | FORCE |NO FORCE ] ROW LEVEL SECURITY;
```

В имени таблицы можно использовать символ шаблона "*".

Если RLS включается опцией ENABLE, то RLS действуют на всех, кроме владельца и ролей с атрибутом SUPERUSER или BYPASSRLS. Если RLS включается опцией FORCE, то RLS действует также и на владельца таблицы. Если RLS включается, а разрешающих политик нет, то доступ запрещён.

RLS не применяется к проверкам ограничений целостности. Это означает, что существуют косвенные пути проверить существование строки. Например, можно попытаться вставить повторяющееся значение в столбец, образующий первичный ключ. Если выдастся ошибка, можно догадаться, что строка существует.

Сложная структура политик, прав доступа нарушает принцип безопасности: простоту использования администраторами. Сложные конструкции создают ложное впечатление о защищенности и повышают вероятность ошибок, создающих бреши в защите.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-createpolicy.html
https://docs.tantorlabs.ru/tdb/ru/18_1/se/ddl-rowsecurity.html

Подсоединение к экземпляру

- Параметры начальной аутентификации устанавливаются в двух файлах `pg_hba.conf` и `pg_ident.conf`
- Файлы редактируются вручную
- содержимое файла `pg_hba.conf` можно в представлении `pg_hba_file_rules`
- Расположение файлов можно посмотреть параметрами конфигурации hba_file и ident_file:

```
postgres=# \dconfig *_file
                          List of configuration parameters
        Parameter         |                          Value
--------------------------+-------------------------------------------------------------
 config_file              | /var/lib/postgresql/tantor-se-1c-17/data/postgresql.conf
 enable_delayed_temp_file | off
 external_pid_file        |
 hba_file                 | /var/lib/postgresql/tantor-se-1c-17/data/pg_hba.conf
 ident_file               | /var/lib/postgresql/tantor-se-1c-17/data/pg_ident.conf
```


Подсоединение к экземпляру

При начальном соединении с экземпляром выполняется аутентификация клиента. Параметры начальной аутентификации устанавливаются в двух текстовых файлах `pg_hba.conf` (host-based authentication, аутентификации по имени узла) и `pg_ident.conf` (identification, файл сопоставления имён пользователей).

Расположение файлов можно посмотреть параметрами конфигурации hba_file и ident_file:

```
postgres=# \dconfig *_file
                          List of configuration parameters
        Parameter         |                          Value
--------------------------+-------------------------------------------------------
 config_file              | /var/lib/postgresql/tantor-se-1c-17/data/postgresql.conf
 enable_delayed_temp_file | off
 external_pid_file        |
 hba_file                 | /var/lib/postgresql/tantor-se-1c-17/data/pg_hba.conf
 ident_file               | /var/lib/postgresql/tantor-se-1c-17/data/pg_ident.conf
 ssl_ca_file              |
 ssl_cert_file            | server.crt
 ssl_crl_file             |
 ssl_dh_params_file       |
 ssl_key_file             | server.key
(10 rows)
```

По умолчанию, файлы располагаются в PGDATA и создаются при создании кластера.

Файлы редактируются вручную, команд для их редактирования нет.

Посмотреть содержимое файла `pg_hba.conf` можно в представлении `pg_hba_file_rules` , в нём отображается текущее содержимое файла. Представление полезно для проверки нет ли в фале опечаток. Если в столбце `error` непустое значение, то в строке файла есть ошибка.

Чтобы изменения в `pg_hba.conf` и `pg_ident.conf` подействовали, нужно перечитать конфигурацию, например, функцией

```
select pg_reload_conf();
```

В файлы можно включать содержимое других файлов директивами `include`, `include_if_exists`, `include_dir`. Например:

```
include_dir /var/lib/postgresql/tantor-se-1c-17/directory
```

https://docs.tantorlabs.ru/tdb/ru/18_1/se/client-authentication.html

Файл pg_hba.conf

- Формат файла - одна запись на строку
- Запись состоит из нескольких полей, разделённых пробелами и/или символами табуляции
- Содержимое полей может быть заключено в двойные кавычки
- записи (строки) ближе к началу файла превалируют, в отличие от файлов параметров (postgresql.conf)
- Текущее содержимое файла можно посмотреть через представление:

```
postgres=# select rule_number r, right(file_name, 11) file_name, line_number l, type, database, user_name
user, address, left(netmask, 15) netmask, auth_method auth, options opt, error from pg_hba_file_rules;
 r |  file_name  |  l  | type  |   database    | user  |  address  |     netmask     | auth  | opt | error
---+-------------+-----+-------+---------------+-------+-----------+-----------------+-------+-----+-------
 1 | pg_hba.conf | 117 | local | {all}         | {all} |           |                 | trust |     |
 2 | pg_hba.conf | 119 | host  | {all}         | {all} | 127.0.0.1 | 255.255.255.255 | trust |     |
 3 | pg_hba.conf | 121 | host  | {all}         | {all} | ::1       | ffff:ffff:ffff: | trust |     |
 4 | pg_hba.conf | 124 | local | {replication} | {all} |           |                 | trust |     |
 5 | pg_hba.conf | 125 | host  | {replication} | {all} | 127.0.0.1 | 255.255.255.255 | trust |     |
 6 | pg_hba.conf | 126 | host  | {replication} | {all} | ::1       | ffff:ffff:ffff: | trust |     |
(6 rows)
```


Файл pg_hba.conf

Формат файла - одна запись в строке файла. Комментарии начинаются на символ "**#**", пустые строки игнорируются. Запись может быть продолжена на следующей строке, для этого нужно завершить строку символом "****" (экранировать символ возврата каретки \r). Запись состоит из нескольких полей, разделённых пробелами и/или символами табуляции. Содержимое полей может быть заключено в двойные кавычки.

Записи просматриваются с начала файла и до конца, **записи (строки) ближе к началу файла превалируют**: если детали соединения подпадают под запись (строку), то эта строка определяет действие, а последующие строки не просматриваются.

В файлы конфигурации PostgreSQL можно включать содержимое других файлов директивами `include, include_if_exists, include_dir`. Путь к файлу или директории может задаваться как абсолютный или относительный, может заключаться в двойные кавычки. Для директивы `include_dir` будут включаться содержимое всех файлов в директории, название которых не начинается с точки и заканчивается на **`.conf`**

В `include_dir` заложена неоднозначность. Для записей важен порядок следования. Если в директории несколько файлов, то файл, включаемый первым будет превалировать. Файлы включаются по правилам сортировки языка "C": цифры идут перед буквами, а буквы в верхнем регистре - перед буквами в нижнем. Представление `pg_hba_file_rules` позволяет точно посмотреть порядок следования записей.

Чтобы подключиться к базе данных, пользователю нужны разрешения в `pg_hba.conf` и право CONNECT для базы данных. Вместо перечисления имён пользователей в файле, проще использовать привилегию CONNECT к базе данных, чтобы не раздувать содержимое файла. В PostgreSQL заложено неудобство для администраторов: по умолчанию, привилегия CONNECT даётся всем пользователям (`public`) и отключить это с помощью DEFAULT PRIVILEGES нельзя.

```
 select rule_number r, right(file_name, 11) file_name, line_number l, type, database, user_name
user, address, left(netmask, 15) netmask, auth_method auth, options opt, error from
pg_hba_file_rules;
 r |  file_name  |  l  | type  | database | user  |  address  |     netmask     | auth  | opt | error
---+-------------+-----+-------+----------+-------+-----------+-----------------+-------+-----+-------
 1 | pg_hba.conf | 117 | local | {all}    | {all} |           |                 | trust |     |
 2 | pg_hba.conf | 119 | host  | {all}    | {all} | 127.0.0.1 | 255.255.255.255 | trust |     |
```

Представление отображает содержимое файла на момент выполнения запроса и файл может быть ещё не применён (перечитан). Фигурные скобки - это массив.

Содержимое pg_hba.conf

- тип соединения:
  - local - соединение через UNIX-сокет
  - host - любые (с шифрованием и без) соединения по TCP/IP
- название базы данных:
  - all - все базы
- replication - подключение по протоколу физической репликации
- IP-адрес с маской подсети или без маски:
  - all - все адреса IPv4 и IPv6
  - samehost - с IP-адресов хоста, на котором работает экземпляр

```
postgres@tantor:~$ tail $PGDATA/pg_hba.conf
# TYPE  DATABASE        USER        IP-ADDRESS    IP-MASK     METHOD
local   sameuser        all                                   peer
local   samerole        all                                   md5
host    all             all         127.0.0.1/32              trust
host    db1,db2,@file1  +bob,@file2 10.0.0.0      255.0.0.0   scram-sha-256
host    "/^db\d{2,4}$"  /^.*desk$   samehost                  ident map=omicron
host    all             all         all                       reject
```


Содержимое pg_hba.conf

Записи в файле содержат:

1) тип соединения:

local - "локальное" (с того же самого узла) соединение через UNIX-сокет. Если поменять разрешения на файл сокета, то можно ограничить доступ к экземпляру от локальных пользователей операционной системы параметрами конфигурации `unix_socket_permissions` и `unix_socket_group`.

host - любые (с шифрованием и без него) соединения по TCP/IP. Вариации: `hostnossl`, `hostssl`, `hostgssenc` (gss = kerberos, с шифрованием), `hostnogssenc` (kerberos без шифрования).

2) название базы даннах:

`all` - все базы

`sameuser` - имя базы совпадает с именем роли, с которыми будет установлено соединение

`samerole` (`samegroup`) - имя базы совпадает с именем одной из грантованных ролей

`replication` - подключение по протоколу физической репликации (но не логической), имя базы по протоколу физической репликации не указывается

Имена баз данных и пользователей можно указывать через запятую. Если имя начинается со слэша, то после слэша регулярное выражение. Имена можно заключать в двойные кавычки. Если имя начинается на символ "@", то после него идёт имя файла, содержимое которого подставляется в этом месте. Несколько регулярных выражений и/или имён можно указать, разделяя их запятыми.

3) имя пользователя (роли):

`all` - любое имя

`+роль` - символ плюс означает любые пользователи, у которых есть указанная роль

4) IP-адрес:

для local отсутствует, для остальных типов соединения указывается IPv4, IPv6, CIDR IPv4 (через слэш число бит в маске сети)

`all` - все адреса IPv4 и IPv6

0.0.0.0/0 - все адреса IPv4

::0/0 - все адреса IPv6

`samehost` - с IP-адресов хоста, на котором работает экземпляр

Содержимое pg_hba.conf (продолжение)

- метод аутентификации:
- `trust` - установить соединение без проверок, в том числе пароля
- `reject` - безусловный отказ в соединении
- `peer` - имя пользователя операционной системы клиента должно совпадать с именем роли, есть опциональный параметр `map`
- `scram-sha-256` или `md5` - проверить пароль
- `gss` - аутентификация по протоколу kerberos
- `ldap` - аутентификация ldap-сервером
- `cert` - запрашивать у клиента сертификат SSL

```
postgres@tantor:~$ tail $PGDATA/pg_hba.conf
# TYPE  DATABASE        USER        IP-ADDRESS    IP-MASK     METHOD
local   sameuser        all                                   peer map=map1
local   samerole        all                                   md5
host    all             all         127.0.0.1/32              trust
host    db1,db2,@file1  +bob,@file2 10.0.0.0      255.0.0.0   scram-sha-256
hostssl "/^db\d{2,4}$"  /^.*desk$   localhost                 cert
host    all             all         1.1.0.0/16                reject
```


Содержимое pg_hba.conf (продолжение)

`samenet` - с IP-адреса в подсетей хоста, на котором работает экземпляр
имя хоста или имя домена можно указать, но не желательно, так как будет использоваться обратное разрешение имён, что приведёт к задержкам в установке соединений

5) Метод аутентификации, если подключение соответствует предыдущим полям записи:
`trust` - установить соединение без проверок, в том числе пароля.
`reject` - безусловный отказ в соединении
`peer` - только для соединений через UNIX-сокет. Имя пользователя операционной системы клиента должно совпадать с именем роли кластера, под которой устанавливается соединение . Есть опциональный параметр `map`.
`scram-sha-256` - проверяет пароль, который должен храниться в виде хэша `scram-sha-256`
`md5` - проверяет пароль, который должен храниться в виде хэша `scram-sha-256` или `md5`
`password` - не стоит использовать, так как пароль будет передаваться открытым текстом
`gss` - аутентификация по протоколу kerberos. Есть параметры `map`, `krb_realm`, `include_realm`.
`ldap` - аутентификация LDAP-сервером. Есть 13 параметров и два режима привязки (bind).
`cert` - запрашивать у клиента сертификат SSL, по умолчанию роль должна совпадать с CN, по это можно переопределить опциональным параметром `map`.
Также могут использоваться `radius`, `pam`, `ident`.

6) Параметры аутентификации (опционально). Параметры специфичны для методов аутентификации и указываются в формате параметр=значение.
Параметр `map` ссылается на строку в файле `pg_ident.conf`.
https://docs.tantorlabs.ru/tdb/ru/18_1/se/gauth-pg-hba-conf.html
https://docs.tantorlabs.ru/tdb/ru/18_1/se/client-authentication.html

Файл сопоставления имён pg_ident.conf

- Для методов **peer, gss, ident** можно сопоставить имя, возвращаемое службой аутентификации и роли кластера, под которой хочет установить сессию клиент
- на названия MAPNAME ссылается параметр map=map1 в записях файла pg_hba.conf
- Представление pg_ident_file_mappings позволяет посмотреть текущее содержимое файла:

```
postgres=# select map_number r, right(file_name, 13) file_name, line_number l,
map_name, sys_name, pg_username, error from pg_ident_file_mappings;
 r |   file_name   | l  | map_name | sys_name | pg_username | error
---+---------------+----+----------+----------+-------------+-------
 1 | pg_ident.conf | 73 | map1     | astra    | postgres    |
 2 | pg_ident.conf | 75 | map1     | astra    | alice       |
(2 rows)
postgres=# \! tail -n 4 $PGDATA/pg_ident.conf
# MAPNAME SYSTEM-USERNAME PG-USERNAME
map1      astra           postgres
# astra также может подключаться с ролью alice
map1      astra           alice
```


Файл сопоставления имён pg_ident.conf

Для методов peer, gss, ident можно сопоставить имя, возвращаемое службой аутентификации и роли кластера, под которой хочет установить сессию клиент.

Представление pg_ident_file_mappings позволяет посмотреть текущее содержимое файла:

```
postgres=# select map_number r, right(file_name, 13) file_name, line_number l,
map_name, sys_name, pg_username, error from pg_ident_file_mappings;
 r |   file_name   | l  | map_name | sys_name | pg_username | error
---+---------------+----+----------+----------+-------------+-------
 1 | pg_ident.conf | 73 | map1     | astra    | postgres    |
 2 | pg_ident.conf | 75 | map1     | astra    | alice       |
(2 rows)
postgres=# \! tail -n 4 $PGDATA/pg_ident.conf
# MAPNAME SYSTEM-USERNAME PG-USERNAME
map1      astra           postgres
# astra также может подключаться с ролью alice
map1      astra           alice
```

На названия MAPNAME ссылается параметр map=map1 в записях файла pg_hba.conf

Представление отображает содержимое файла на момент выполнения запроса и файл может быть ещё не применён (перечитан). Чтобы изменения в pg_hba.conf и pg_ident.conf подействовали, нужно перечитать конфигурацию, например, функцией

```
select pg_reload_conf();
```

В файл можно включать содержимое других файлов директивами include, include_if_exists, include_dir.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/auth-username-maps.html

Практика

1. Создание новой роли
2. Установка атрибутов
3. Создание групповой роли
4. Создание схемы и таблицы
5. Выдача роли доступа к таблице
6. Удаление созданных объектов
7. Расположение файлов конфигурации
8. Просмотр правил аутентификации
9. Локальные изменения для аутентификации
10. Проверка корректности настройки
11. Очистка ненужных объектов

Практика

Создание новой роли
Установка атрибутов
Создание групповой роли
Создание схемы и таблицы
Выдача роли доступа к таблице
Удаление созданных объектов
Расположение файлов конфигурации
Просмотр правил аутентификации
Локальные изменения для аутентификации
Проверка корректности настройки
Очистка ненужных объектов

7

7a

Физическое резервирование

Виды резервных копий

- Горячие - без простоя, без остановки экземпляра, без приостановки обслуживания клиентских сессий
- Холодные - на корректно (с финальной контрольной точкой) остановленном экземпляре
- Автономные или самодостаточные
- Инкрементальные

Виды резервных копий

Кластер баз данных физически состоит из файлов в файловой системе. Экземпляр не дублирует файлы, все файлы хранятся без дублей. Потеря любого файла может привести к потере данных, что обычно не допускается.

Файлы могут теряться и повреждаться по разным причинам. Например, злоумышленник или программа ("компьютерный вирус") могут стереть файлы кластера. Зеркалирование дисков в этом случае не поможет. В PostgreSQL есть много способов резервирования. Самое оптимальное с точки зрения простоты, стоимости, отказоустойчивости для типичного кластера - использовать физическую репликацию, которую мы рассмотрим в отдельной главе.

Резервные копии ("бэкапы") могут быть:

1)Горячими - без остановки экземпляра.

2)Холодными - если экземпляр корректно остановлен перед резервированием. Остановка экземпляра и выполнение резервирования означает простой в обслуживании, что нежелательно. Утилита резервирования `pg_basebackup` не делает холодные бэкапы.

3)Автономными или самодостаточными (self contained). Набор файлов, который достаточен для того, чтобы запустился экземпляр и дал доступ к образу данных на время, когда производилось резервирование. Такие копии могут периодически создаваться (например, раз в квартал) и храниться (удерживаться) желаемое время.

Для горячих физических резервных копий понятие согласованного состояния (consistent state) означает, что на копию наложены журнальные данные на момент окончания резервирования. Холодные резервные копии, если экземпляр был остановлен корректно, считаются согласованными. Горячую резервную копию можно согласовать, накатив (наложив, применив) на неё журнальные файлы (WAL-журналы) до момента окончания резервирования.

В любом случае, при запуске на согласованной копии будет искаться журнальный файл, содержащий запись о контрольной точке, на которую указывает управляющий файл (или файл `backup_label`). Если файл журнала отсутствует, экземпляр не запустится. Согласованность ускоряет запуск экземпляра.

Инкрементальные (синонимы: дифференциальные, кумулятивные, дельты) копии появились в 17 версии вместе с отслеживанием изменившихся блоков (summaries).

https://docs.tantorlabs.ru/tdb/ru/18_1/se/backup.html

Холодные бэкапы

- Экземпляр должен быть корректно остановлен
- Утилиты `pg_basebackup` и `wal-g` не бэкапят, если экземпляр остановлен
- Используются утилиты операционной системы cp, rsync, tar
- Недостаток - прерывание обслуживания приложений, нужно учитывать символические ссылки
- Пример создания холодного бэкапа, если нет символических ссылок:

```
pg_ctl stop
cp -pr $PGDATA/* $HOME/backup
sudo systemctl start tantor-se-server-18
```

- Пример создания горячего бэкапа утилитой pg_basebackup:

```
pg_basebackup -D $HOME/backup
```

- Пример создания горячего бэкапа утилитой wal-g:

```
wal-g backup-push $PGDATA
```


Холодные бэкапы

Холодное резервирование - это резервирование на корректно остановленном кластере. В результате получается автономная копия кластера. Автономная или самодостаточная (self contained) резервная копия включает в себя все файлы, которые нужны экземпляру для запуска.

Техника резервирования:

1) остановить экземпляр

2) скопировать `PGDATA`.

Наличие табличных пространств и символических ссылок (`pg_wal`) усложняет резервирование, теряется его единственное преимущество - простота и увеличивается вероятность ошибок.

Можно использовать снимки файловой системы или выполнять копирование на работающем экземпляре, а после остановки экземпляра обновлять файлы утилитой `rsync` в режиме подсчета контрольных сумм. При этом теряется преимущество холодного резервирования - простота. Более практично резервирование работающего кластера утилитой `pg_basebackup` или утилитой `wal-g`.

Холодная копия не отличается по возможностям её использования от копии, сделанной утилитами резервирования на работающем экземпляре. Созданную копию можно использовать с архивом журналов для полного восстановления или восстановления на момент в прошлом (point in time recovery).

В примере на слайде делается холодный бэкап и горячий. Холодный бэкап имеет недостаток: копируется всё содержимое директории `pg_wal` и много ненужных файлов журнала. Горячий бэкап утилитой `pg_basebackup` в директории `pg_wal` сохраняет только файлы журналов, которые нужны, чтобы бэкап был автономным.

Команда холодного бэкапа не учитывает, что могут быть символические ссылки и не сообщит, что директории, на которые они указывают не скопировала. Команда `pg_basebackup` попытается скопировать директории табличных пространств, выдаст ошибку и будет видно, что бэкап не сделан. Для копирования в другие директории есть параметр `--tablespace-mapping=OLDDIR=NEWDIR`.

При использовании утилиты резервирования `wal-g`, бэкап будет сделан без дополнительных параметров.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/backup-file.html

Что нужно резервировать

- Директорию PGDATA
  - временные файлы не резервируются
- Директории табличных пространств
- Файлы WAL от начала контрольной точки перед копированием файлов кластера до момента, на который нужно восстановиться
- Для полного восстановления нужно настроить "архив журналов"
  - утилитой `pg_receivewal`
  - и/или параметрами `archive_command` и `archive_mode=on`

Что нужно резервировать

Шаги резервирования:

1) Утилита резервирования `pg_basebackup` ждёт завершения контрольной точки или вызывает её (параметр `-c fast` или `--checkpoint=fast`)

2) С начала контрольной точки нужно сохранять файлы WAL. В журналах хранится история изменений файлов кластера. Чтобы восстановиться, нужно иметь все файлы WAL до того момента, на который нужно восстановить данные. Обычно, нужно восстановить на самый последний момент времени, чтобы не потерять ни одной транзакции. Нужно настроить архивирование WAL утилитой `pg_receivewal` и/или параметром `archive_command` и `archive_mode=on`.

Настройка архива журналов рассматривается дальше.

3) Копируются файлы кластера, то есть всё, что находится в `PGDATA` и директориях табличных пространств (кроме файлов, известных утилите как временные).

Журналы с момента начала контрольной точки до завершения резервирования могут копироваться, чтобы получилась автономная копия, могут не копироваться (`--wal-method=none` или `-X none`).

4) В корне директории бэкапа создаются файлы **`backup_label`** и **`backup_manifest`**.

5) Утилита выполняет системные вызовы **fsync** по каждому файлу в бэкапе или **sync** по файловой системе, где создан бэкап (если указать параметр **`--sync-method=syncfs`**, который появился в 17 версии). Это нужно, чтобы бэкап не был повреждён при выключении питания на хосте, куда сделан бэкап.

В результате резервирования получается "бэкап файлов кластера".

Для полного восстановления нужен ещё "архив журналов". Если не использовать архив журналов, что можно будет восстановиться только на момент окончания бэкапа, если бэкап автономный. Если бэкап не автономный (параметр `--wal-method=none` или `-X none`), то восстановиться без файлов журнала нельзя.

Ограничения при создании бэкапа

- Пока создаётся бэкап не стоит менять базы данных, на основе которых создаются новые базы данных
- После создания или удаления табличного пространства рекомендуется сделать бэкап
- Если нужно создатьавтономный бэкап, то нужно включить в бэкап журналы, которые включают изменения с момента начала контрольной точки до момента окончания копирования файлов кластера

Ограничения при создании бэкапа

Восстановиться с бэкапа можно не ранее, чем на момент завершения копирования файлов.

Что нельзя делать пока идёт создание бэкапа:

1) В процессе копирования можно создавать базы данных в резервируемом кластере, но не стоит вносить изменения в базы данных, на основе которых создаются новые базы, так как эти изменения могут попасть в создаваемые базы.

2) После создания или удаления табличного пространства рекомендуется сделать бэкап. Команда создания табличного пространства записывает в журнал абсолютный путь директории табличного пространства. Эта журнальная запись будет проигрываться при восстановлении и будет попытка создания символической ссылки на директорию.

3) Если нужно создать автономный бэкап, то нужно включить в бэкап файлы журналов, которые включают изменения с момента начала контрольной точки до момента окончания копирования файлов кластера. По умолчанию, `pg_basebackup` создаёт автономный бэкап. Утилита `wal-g` предупредит, что нужно настроить архивирование.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/continuous-archiving.html#CONTINUOUS-ARCHIVING-CAVEATS

Архив журналов

- Способы организации архива журналов:
- прием (pull) потока журнальных записей `pg_receivewal`

```
pg_receivewal --create-slot --slot=arch
pg_receivewal -D $HOME/archivelog --slot=arch --synchronous
```

- передача (push) командой в параметре `archive_command`

```
alter system set archive_command = 'wal-g wal-push %p >>
$PGDATA/log/archive_command.log 2>&1';
alter system set archive_mode=on;
```

- Команда копирования журналов из архива при восстановлении:

```
alter system set restore_command = 'wal-g wal-fetch %f %p >>
$PGDATA/log/restore_command.log 2>&1 || cp $HOME/archivelog/%f %p || cp
$HOME/archivelog/%f.partial %p';
```


Архив журналов

Резервная копия может быть автономной. Чтобы можно было восстановиться с неё на последний момент времени, нужно будет накатить на эту копию журнальные файлы со времени создания копии до последнего момента. По умолчанию кластер сохраняет в директории `PGDATA/pg_wal` файлы журналов для целей восстановления согласованности файлов кластера после аварийной остановки экземпляра, то есть с начала последней контрольной точки. Журнальные файлы могут удерживаться параметрами конфигурации долгое время, но директория `PGDATA/pg_wal` может быть не лучшим местом для хранения журналов, если это дорогостоящее устройство хранения, а также в целях защиты от удаления злоумышленниками. Бэкапы и журналы по возможности должны храниться на хосте к которому нет доступа резервируемого хоста, чтобы злоумышленник не смог стереть бэкапы.

Способы организации архива журналов:

1) утилитой `pg_receivewal`. Эта утилита может принимать журнальные данные без задержки. Недостатком является то, что нужно автоматизировать запуск утилиты и перезапускать её в случае сбоя. Пример:

```
pg_receivewal --create-slot --slot=arch
pg_receivewal -D $HOME/archivelog --slot=arch --synchronous
```

2) установив параметры `archive_command='команда'` и `archive_mode=on`. У этого способа есть недостаток - текущий файл журнала (в который пишут процессы экземпляра) начнет копироваться только, когда файл перестанет быть текущим. Если текущий файл потеряется, то транзакции будут потеряны, что неприемлемо.

Для копирования можно использовать команду `cp` или `wal-g`. Пример:

```
alter system set archive_command = 'wal-g wal-push "%p" >>
$PGDATA/log/archive_command.log 2>&1';
alter system set archive_mode=on;
```

Параметр для копирования файлов журнала из архива, в процессе восстановления:

```
alter system set restore_command = 'wal-g wal-fetch %f %p >>
$PGDATA/log/restore_command.log 2>&1 || cp $HOME/archivelog/%f %p || cp
$HOME/archivelog/%f.partial %p';
```

Процедура восстановления

- Остановить экземпляр
- Восстановить из полной резервной копии PGDATA и директории табличных пространств
- Применить инкрементальные бэкапы, если они создавались
- Создать файл `standby.signal` и установить значение параметра конфигурации `restore_command`, чтобы файлы журнала копировались и применялись из архива

```
pg_ctl stop -m immediate
rm -rf $PGDATA/*
wal-g backup-fetch $PGDATA LATEST
touch $PGDATA/standby.signal
echo "restore_command = 'wal-g wal-fetch %f %p >>
$PGDATA/log/restore_command.log 2>&1 || cp $HOME/archivelog/%f %p || cp
$HOME/archivelog/%f.partial %p'" >> $PGDATA/postgresql.auto.conf
```


Процедура восстановления

1) Обнаружение повреждения. Экземпляр при этом может быть аварийно остановлен и попытка запуска неудачна, либо экземпляр продолжает работать, но выдаёт ошибки при доступе к нужным приложению данным.

Если экземпляр не остановился, его нужно будет остановить. Можно попробовать остановить в режиме fast. Если не остановится в режиме `fast`, то остановить в режиме `immeidate`. При корректной остановке выполняется контрольная точка, если файлы исчезли, то контрольная точка не сможет выполнить запись в отсутствующие файлы.

2) Скопировать или переместить директорию `PGDATA/pg_wal` (или последние файлы, в которые писал экземпляр), если они доступны и файлы параметров (если редактировались вручную после бэкапа, с которого будет идти восстановление).

3) Удалить содержимое директории `PGDATA` и табличных пространств. Утилиты резервирования требуют пустые директории, чтобы из-за ошибки не затереть файлы в директории

4) Скопировать (restore) из бэкапа файлы кластера (`PGDATA` и табличных пространств).

5) Установить в параметре конфигурации `restore_command` команду копирования WAL-файлов из архива в директорию `PGDATA/pg_wal` (в том числе, файл `.partial`). Создать в `PGDATA` файл `standby.signal` . Можно скопировать файлы из архива в директорию `pg_wal` вручную.

Вместо файла `standby.signal` может использоваться файл `recovery.signal`, но с ним вероятность ошибок больше, а преимуществ он не даёт.

6) Скопировать последние, не попавшие в архив, файлы, сохраненные на 2 пункте.

Процедура восстановления (продолжение)

- Запустить экземпляр
- Проверить, что все журналы, которые есть, были применены
- Послать сигнал promote экземпляру, он откроется на чтение-запись

```
sudo systemctl start tantor-se-server-18
psql -c "select pg_last_wal_replay_lsn()"
psql -c "checkpoint"
psql -c "select pg_promote(true, 60)"
```


Процедура восстановления (продолжение)

7) Запустить экземпляр. Процесс **startup** обнаружит файл **`backup_label`** в корне `PGDATA` начнет восстановление (накат журналов) с LSN, указанного в нём, а не в файле **`pg_control`**. Файлы применяются в порядке их создания экземпляром, пропуски (gap) - отсутствие файла или повреждение блоков (повреждения распознаются, так как блоки журналов защищены контрольными суммами) критичны - перейти через них нельзя.

При накате проверяется, что к блоку данных может быть применена запись. В журналах присутствуют полные образы изменившихся блоков (`full_page_writes`) и, даже если блоки данных повреждены (torn "разорваны", расщеплены), они смогут восстановиться.

Обновляется файл **`pg_control`**, а файл **`backup_label`** переименуется в `backup_label.old`

В процессе восстановления могут выполняться точки рестарта, обновляющие управляющий файл **`pg_control`**.

8) Благодаря использованию файла `standby.signal` кластер не будет открыт на запись и можно будет убедиться, что все журнальные файлы применены.

9) Опционально, можно выполнить команду **`checkpoint`** для того, чтобы грязные буфера попытались записаться на диск и `pg_promote()` выполнился быстрее. Суммарное время выполнение то же самое, но выполняя `checkpoint` будет понятно, что долго выполняется именно контрольная точка.

В режиме восстановления выполняется restart point, которая не может выполняться чаще, чем выполнялись контрольные точки, которые имелись в журнале. Если restart point не сможет выполниться, то команда ничего не сделает.

10) После этого можно выполнить **`pg_ctl -t`** `таймаут` **`promote`** или функцию **`pg_promote(`**`true, таймаут`**`)`**. Результат вернётся после успешного перевода кластера в режим чтения-запись или, если по превышении таймаута. По умолчанию, таймаут 60 секунд.

После успешного promote линия времени увеличится на единицу.

Пример восстановления

- удобнее использовать файл `standby.signal`, а не `recovery.signal`
- в примере делается бэкап

```
cd $HOME/backup
pg_ctl stop -m immediate -D .
rm -rf $HOME/backup/*
pg_basebackup -c fast -D .
touch standby.signal
echo "restore_command = 'cp /var/lib/postgresql/tantor-se-
18/data/pg_wal/%f %p || cp /var/lib/postgresql/tantor-se-
18/data/%f.partial %p'" >> ./postgresql.auto.conf
echo "port = 5433" >> ./postgresql.auto.conf
chmod 750 $HOME/backup
pg_ctl start -D .
psql -p 5433 -c "select pg_last_wal_replay_lsn();"
psql -c "checkpoint"
pg_ctl promote  -t 60 -D .
```


Пример восстановления

Если есть файл `recovery.signal`, но не установлена команда в параметре **`restore_command`**, то накат журналов не происходит и экземпляр не запускается:

```
FATAL:  must specify "restore_command" when standby mode is not enabled
```

Если есть файл `recovery.signal`, установили параметр **`restore_command`** но в команде ошибка, то экземпляр перейдёт на новую линию времени, удалит файл `recovery.signal` и откроет кластер в режиме чтения-записи. Исправить ошибку в команде **`restore_command`** и повторить накат журналов не удастся. Нужно будет либо остановить экземпляр, стереть директорию и восстановить из бэкапа заново. В этом неудобство и опасность использования файла `recovery.signal`.

Если использовать файл **`standby.signal`**, то экземпляр, после наката журналов до достижения согласованности (накат журнала, созданного на момент завершения бэкапа) откроет кластер в режиме чтения. Можно подкладывать в директорию `pg_wal` журнальные файлы, исправлять параметр **`restore_command`** проверять применены ли все журнальные файлы, применён ли файл **`.partial`**, если он есть, останавливать и запускать экземпляр заново. Эти действия не повредят кластер. Опасность может быть только в том, что накатится недописанный до конца WAL-файл (**`.partial`**), потом будет найден более полный файл журнала с тем же названием (из директории `pg_wal` сбойнувшего мастера) то, чтобы его накатить, придётся восстановить (переименовав `backup_label.old`) файл `backup_label` и/или внести в него правильные значения (если вы уверены, что их знаете).

После проверок, что все файлы журналов применены, можно будет дать сигнал promote, будет создана новая линя времени, удалён файл **`standby.signal`** и кластер откроется в режиме чтение-запись.

В примере на слайде в **`restore_command`** журналы берутся из директории работающего кластера, а не архива журналов. Работающий кластер не формирует файлы **`.partial`**, они формируются только утилитой `pg_receivewal`. Команда копирования с переименованием файла **`.partial`** дана для примера.

Примечание:

Параметры конфигурации `recovery_target_*` (используются для неполного восстановления, задают момент, до которого нужно накатить журналы) работают со **`standby.signal`** точно так же, как с `recovery.signal`.

Если в `PGDATA` созданы оба файла, то превалирует **`standby.signal`**.

Применение журнальных записей (WAL)

- журнальные записи применяет только процесс startup
- управляется файлами `standby.signal`, `recovery.signal` и параметрами `restore_command` и `recovery_*`
- журнальная запись, с которой начинается применение (накат) берётся из текстового файла `PGDATA/backup_label` или бинарного управляющего файла `PGDATA/global/pg_control`
- процессу startup неизвестно какая журнальная запись формировалась последней перед сбоем кластера, процесс пытается применять все записи и файлы, которые имеются в `PGDATA/pg_wal`

Применение журнальных записей (WAL)

Накатывает (применяет, apply) файлы журналов только процесс **startup**. Отдельных команд и утилит для наката журналов нет. Процесс **startup** запускается при запуске экземпляра. Процесс ищет файл `backup_label` в корне `PGDATA` и берёт из этого файла LSN начала контрольной точки (название WAL-файла, где эта запись находится) и LSN окончания контрольной точки.

Если файла `backup_label` нет, то эти данные берутся из управляющего файла `pg_control`.

Процесс **startup** ищет файлы журнала в директории `PGDATA/pg_wal` и накатывает файлы, которые найдёт в этой директории. Файлы не дублируются (не зеркалируются).

Процессу **startup** неизвестно, какой файл содержит самые последние журнальные записи, которые формировались перед остановкой экземпляра. Поэтому, процесс startup накатив файл, пытается открыть следующий файл. В конце имени журнальных файлов число, которое увеличивается на единицу без пропусков. Если файла нет, то, при наличии файла `standby.signal` или `recovery.signal` процесс startup выполняет команду в параметре `recovery_command`. Если команда завершилась успешно и файл появился в результате выполнения команды, то файл накатывается.

Если файл не появился или при чтении 16-мегабайтного файла процесс **startup** обнаруживает, что не может накатить следующую журнальную запись (например, контрольная сумма неверна), то при наличии `standby.signal` процесс продолжает попытки выполнить `recovery_command` или перечитывает файл. Интервалы между попытками устанавливаются параметром `wal_retrieve_retry_interval` (по умолчанию 5 секунд). В тело файлов журналов может писать процесс `walreceiver` в то время, пока работает процесс startup, можно копировать в `pg_wal` файлы вручную. независимо от способа появления журнальных файлов и записей в них, процесс startupбудет их перечитывать.

Если вместо `standby.signal` использовался `recovery.signal`, то, по умолчанию, восстановление прекращается, выполняется контрольная точка, создаётся новая линия времени и экземпляр открывается на чтение-запись. Это можно поменять параметром **`recovery_target_action`**, но вместе с ним нужно установить один из параметров `recovery_target_*`, задающий момент, до которого нужно накатывать журнальные записи.

Если файлов `standby.signal` или `recovery.signal` не было, то команда `recovery_command` не выполняется, накатываются только те файлы, которые были в директории `pg_wal`. Параметр **`log_startup_progress_interval`** (10 секунд) задаёт интервал между записями в диагностический лог, чем занят процесс **startup**.

Линии времени

- При изменении линии времени в директории `PGDATA/pg_wal` создаются текстовые файлы `0000000N.history` с информацией о линиях времени, их нельзя стирать и они будут резервироваться
- Новая линия времени появляется, если выполнялось восстановление или реплика стала мастером
- Цель линий времени:
  - чтобы при восстановлении на момент в прошлом новые файлы журналов не затирали прежние
  - чтобы была возможность вернуться на прежние линии времени
- Файлы с линями времени может удалять утилита `pg_archivecleanup` запущенная с параметром `-b`

Линии времени

Новая линия времени появляется, если:
1) при запуске использовался файл `recovery.signal`, то есть выполнялось восстановление
2) экземпляр получил сигнал **promote**, то есть экземпляр вышел из режима восстановления.
Цель линий времени:
1) чтобы при восстановлении на момент в прошлом новые файлы журналов не затирали прежние
2) чтобы была возможность вернуться на прежние линии времени.
Процесс восстановления может сформировать название файла, так как знает номер линии времени, размер блока журнала, размер файла, LSN из управляющего файла или из файла `backup_label`.
При увеличении **линии времени**, в директории `PGDATA/pg_wal` создаются текстовые файлы `0000000N.history` с информацией о линиях времени.
Эти файлы не нужно удалять, они не занимают много места. Более новый файл включает в себя данные предыдущего файла. Эти файлы может удалять утилита `pg_archivecleanup`, запущенная с параметром `-b` (или `--clean-backup-history`).

При считывании следующей записи процесс восстановления в первую очередь ищет место, где должен быть размер журнальной записи. Если в этом месте нереальное значение, то прекращает восстановление, если реальное, то ищет место с контрольной суммой. Если контрольная сумма не совпадает, прекращает или приостанавливает восстановление.
Пример сообщения в файле сообщений экземпляра:
```
LOG:  invalid record length at CA/277E2A88: expected at least 24, got 0
```
процесс восстановления ожидал увидеть число не меньше 24 (минимальный размер заголовка журнальной записи в этой версии PostgreSQL), а увидел нули.

Файлы журнала предварительной записи (WAL)

- Находятся в `PGDATA/pg_wal`
- Не дублируются
- В них пишут серверные и фоновые процессы экземпляра
- могут располагаться вне PGDATA
- Размер текущего файла журнала такой же, как остальных
- `pg_switch_wal()` - переключение на следующий WAL-файл

```
postgres=# select * from pg_ls_waldir() order by 1;
            name            |   size   |      modification
----------------------------+----------+------------------------
 000000010000000000000005D | 16777216 | 2026-05-13 09:36:52+03
 000000010000000000000005E | 16777216 | 2026-05-13 09:41:59+03
 000000010000000000000005F | 16777216 | 2026-05-13 09:34:28+03
postgres=# select pg_current_wal_lsn()\gx
-[ RECORD 1 ]------+-----------
pg_current_wal_lsn | 0/5E0001C0
```


Файлы журнала предварительной записи (WAL)

В файлах содержатся журнальные записи переменной длинны. У записей есть заголовок размером от 24 байт (зависит от сборки PostgreSQL, может быть больше). Размер журнальной записи до 1Гб. Каждая журнальная запись переменной длинны защищена контрольной суммой, которая хранится в заголовке журнальной записи. Физически, запись в файлы журнала выполняется по 8Кб (`wal_block_size`). Если журнальная запись не кратна 8Кб, то в конец журнальной записи добавляются нулевые байты так, чтобы ее размер увеличился до ближайших 8Кб. На это место будет записана следующая журнальная запись. Если журнальные записи небольшой длинны, то запись может выполняться неоднократно в один и тот же блок.

При создании журнального файла ("WAL сегмента") ему назначается имя и размер. Чтобы операционная система физически выделила место для файла, процесс, создающий файл, выполняет запись пустых блоков вплоть до конца файла (16Мб), либо нулевого байта в самый конец файла (если `wal_init_zero=off`, используется для файловых систем copy-on-write). Зануление файла нужно, чтобы заранее было зарезервировано место в файловой системе и экземпляр не столкнулся с нехваткой места в файловой системе. Файлы журналов создаёт, удаляет, переименовывает и зануляет процесс **checkpointer** в конце контрольной точки, и только, если файлов не хватит их создают другие процессы.

Также, это улучшает отказоустойчивость: изменение размера файла это операция с метаданными файловой системы. В зависимости от настроек монтирования, файловая система может "журналировать только метаданные" (слово журналировать относится к файловым системам, в них тоже реализована логика защиты от пропадания питания), а при частом изменении размера файла (размер файла в файловой системе относится к метаданным) последние блоки файла могут быть потеряны, либо скорость записи в файл будет невысокой.

Файлы журналов можно расположить на диске, отличном от того, где находится остальные файлы кластера. Можно остановить экземпляр, переместить директорию pg_wal на другой диск и создать символическую ссылку `PGDATA/pg_wal` на новую директорию.

В каждый момент времени есть текущий файл журнала, куда процессы экземпляра пишут (или последний файл куда писали, если экземпляр погашен). Размер этого файла равен размеру других файлов журнала (по умолчанию, 16Мб).

Для переключения на запись в следующий журнальный файл используется функция **`pg_switch_wal()`**.

LSN (Log Sequence Number)

- Журнал можно представить себе как конкатенированный набор файлов журнала (большей части которые уже удалены) начиная с самого первого файла, сформированного при создании кластера
- LSN - 64битное число, монотонно возрастающее, указывающее адрес (смещение от первого байта первого файла журнала) с точностью до байта в журнале кластера
- Процессы экземпляра пишут в журнал записи переменной длинны, LSN используется для указания адреса начала журнальной записи
- физически запись в файлы осуществляется блоками по 8Кб (`wal_block_size`)

LSN (Log Sequence Number)

Процессы экземпляра пишут в файлы журнала записи переменной длинны. Адрес каждой записи обозначается 64-битным числом "LSN" (Log Sequence Number), которое представляет собой порядковый номер байта журнала со времени создания кластера (момента когда начала производиться запись в журнал).

Самый первый файл имеет название 000000010000000000000001.

Можно сказать, что LSN определяет "смещение от начала журнала" или "позицию в журнале предварительной записи". Можно также сказать, что LSN - монотонно возрастающее целое число, которое указывает на запись в журнале.

Значения LSN присутствуют во многих местах: блоках данных, управляющем файле, в самих записях журнала. По LSN можно восстановить название файла журнала, в котором содержится запись, на которую указывает LSN.

Название файлов журнала состоит из **трёх чисел по 8 символов**. Каждое число 32-битное, записывается в шестнадцатеричной форме. Максимальное число FFFFFFFF (32 единички в двоичном виде). Первое число это номер "линии времени" (`Time Line`, `TLI`, инкарнация). Это число увеличивается на единицу после восстановления для того, чтобы не допустить затирания старых файлов журнала.

При достижении максимальных значений сброса в ноль (LSN wrap) LSN и линий времени не предусмотрено. Максимальное значение LSN довольно большое: 16777216 терабайт.

Физически запись в журнальные файлы производится 8-килобайтными блоками. На размер блока указывает параметр конфигурации `wal_block_size`, который задан при сборке PostgreSQL и не меняется.

Журнальные записи защищены контрольными суммами и они не отключаются.

Размер файла журнала (WAL-сегмента), размер блока журнала, `TimeLineID` хранятся в управляющем файле кластера (`pg_control`), поэтому зная LSN можно определить название файла в котором содержится запись переменной длинны, на который указывает LSN.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/wal-internals.html

Названия журнальных файлов и LSN

- LSN представляют в виде двух 32-битных чисел, записанных в шестнадцатеричной форме (HEX), разделенных слэшем: XXXXXXXX/YYZZZZZZ
- XXXXXXXX - "старшие" 32 бита LSN
- YY - "старшие" 8 бит "младшего" 32-битного числа
- ZZZZZZ - смещение в файле 16-мегабайтного журнала относительно его начала
- Размер файла журнала определяет максимальный размер журнального буфера (wal_buffers)
- 0000000NXXXXXXXX000000YY - названия 16-мегабайтных файлов журнала

Названия журнальных файлов и LSN

Рассмотрим подробнее LSN. Возможно, вы задавались вопросом почему размер файлов журнала не очень большой, всего 16Мб.

В текстовом виде, который используется в файлах сообщений, опциях команд, функциях LSN представляют в виде двух 32-битных чисел, записанных в шестнадцатеричной форме (HEX), разделенных слэшем: XXXXXXXX/YYZZZZZZ. XXXXXXXX - "старшие" 32 бита LSN. Если размер файлов журнала 16Мб (значение по умолчанию), то YY - "старшие" 8 бит "младшего" 32-битного числа. ZZZZZZ - смещение в файле 16-мегабайтного журнала относительно его начала. Ведущие нули не выводятся: 00000001/0A000FFF будет выводиться как 1/A000FFF, что усложняет восприятие.

Максимальный размер файла журнала 1Гб, минимальный 1Мб и может принимать значения по степеням двойки (**16**, 32, 64, 128, **256**, 512 1024Мб). Например, если установить размер файла журнала в **256Мб**, то LSN будет выглядеть как XXXXXXXX/YZZZZZZZ. Если 1Мб (такой маленький размер не стоит использовать из-за того что `wal_buffers` будет не больше 1Мб), то: XXXXXXXX/YYYZZZZZ. У других значений размера файла такого наглядного деления по разрядам нет. Размер файла журнала определяет максимальный размер журнального буфера в разделяемой памяти экземпляра, который устанавливается параметром `wal_buffers`. По умолчанию, при размере `shared_bufers` больше `512Мб`, журнальный буфер выставляется в максимальное значение `16Мб`.

Размер журнальных файлов можно задать при создании кластера утилитой `initdb --wal-segsize=размер` или после создания кластера утилитой `pg_resetwal --wal-segsize=размер`.

Названия файлов журналов также зависят от размера файла. Для размера **16Мб** формат следующий: 0000000NXXXXXXXX000000YY. Вторые 8 символов - старшие 32 бита LSN, потом 6 **нулей**, потом 2 символа старших 8 битов младшего 32-разрядного числа. Для размера **256Мб** формат: 00000001XXXXXXXX00000YYY. первые 8 символов - номер перехода на новую линию времени.

Функции для работы с журналами

- **`pg_switch_wal()`** переключает запись на новый WAL-файл
- **`pg_create_restore_point('текст')`** создаёт в журнале запись LSN с текстовой меткой, которую можно использовать в параметре `recovery_target_name`
- **`pg_walfile_name('LSN')`** выдает название WAL-файла в котором есть запись LSN
- **`pg_current_wal_flush_lsn()`** LSN конца последней журнальной записи, которая считается надёжно сохранённой
- **`pg_wal_lsn_diff(LSN,LSN)`** байт между двумя LSN
- **`pg_last_wal_replay_lsn()`** какой LSN применён
- **`pg_last_wal_receive_lsn()`** какой LSN принят

Функции для работы с журналами

`pg_switch_wal()` переключает запись на новый WAL-файл, прежний не дописывается, хоть и имеет такой же размер, как и остальные файлы журналов.

`pg_create_restore_point('текст')` создаёт в журнале запись LSN с текстовой меткой. Функция возвращает LSN начала этой журнальной записи. Метку можно указывать в параметре recovery_target_name чтобы указать что нужно накатить журналы до записи с меткой. Если создать несколько меток с одним и тем же названием, тогда восстановление остановится как только оно наткнется на запись с этой меткой.

`pg_walfile_name('LSN')` выдает название WAL-файла, в котором должна встретиться запись с указанным LSN. Результат выдается расчётным способом на основе данных из управляющего файла. Что удивительно, на реплике не работает.

`pg_walfile_name_offset(LSN)` показывает не только расчётное имя файла, но и смещение в байтах относительно его начала.

`pg_current_wal_lsn()` показывает LSN **последнего** байта ("конца") последней журнальной записи, записанной в текущий файл журнала. До этого LSN включительно процессы в операционной системе должны видеть записанные, если будут читать файл журнала.

`pg_current_wal_flush_lsn()` LSN **последнего** байта последней журнальной записи, которая считается надёжно сохранённой (fsync или другой способ вернул результат). Определяет LSN до которого включительно должны сохраниться журнальные записи после пропадания питания.

`pg_current_wal_insert_lsn()` LSN **последнего** байта последней журнальной записи, которую сформировали процессы экземпляра в журнальном буфере и эта журнальная запись еще могла не попасть на диск. Используется процессами экземпляра для определения LSN своей записи, которую начнут формировать.

Утилитой командной строки **`pg_waldump`** `имя_файла` можно получить из WAL-файла в текстовом виде список LSN начала журнальных записей и их содержимое.

`pg_lsn` - тип данных. Для этого типа данных определено приведение типа `'литерал'::pg_lsn`, оператор вычитания или функция `pg_wal_lsn_diff(LSN,LSN)`, которым можно получить в байтах разницу между двумя LSN - объем журнальных данных.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/functions-admin.html#FUNCTIONS-ADMIN-BACKUP

Отсутствие потерь (Durability)

- Архитектура резервирования должна гарантировать полное восстановление зафиксированных транзакций
- Архитектура резервирования должна защищать от удаления бэкапов в случае взлома хоста с экземпляром кластера: инициатором бэкапов должен быть хост отличный от того на котором работает экземпляр кластера
- Текущий журнальный файл не должен быть точкой сбоя (single point of failure)
- Использование `pg_receivewal` или физической реплики - возможность гарантировать отсутствие потерь при повреждении текущего журнала

Отсутствие потерь (Durability)

Журналы можно забирать из "архива", но важным для полного восстановления является накат записей из самого последнего файла журнала, которые могли не успеть передаться в архив. Потеря даже одной зафиксированной транзакции обычно неприемлема (свойство Durability из свойств ACID транзакции). Архивы журналов не гарантируют что они содержат все транзакции, а последний журнал на диске поврежденного кластера может не сохраниться например в результате катаклизма (disaster: пожар, затопление, разрушение здания где находятся системы хранения файлов). Файл журнала в директории `PGDATA/pg_wal` не должен являться "точкой сбоя". Использование pg_receivewal и/или физической реплики с подтверждением ими фиксации транзакции обеспечит отсутствие потерь транзакций в случае полной потери хоста кластера со всеми дисковыми системами (disaster).

Подтверждение фиксации транзакций настраивается параметрами `synchronous_commit` и `synchronous_standby_names`.

Монтирование pg_wal на системах хранения с дублированием может защитить от сбоя диска, но не защитит от злоумышленника который может стереть файл журнала. В последнем случае можно задаться вопросом: нужно ли вести архивы или удерживать файлы в pg_wal? Технически архивы вести удобнее, чем настраивать удержание файлов в pg_wal. Также копирование в архивы освобождает место на дорогостоящем высокоскоростном устройстве, где располагают `pg_wal`. Также стоит принять во внимание, что в целях безопасности нужно, чтобы хост кластера не имел доступа к бэкапам и архивам журналов. Если злоумышленник получит доступ к хосту кластера, то первым делом злоумышленники удаляют все бэкапы. Хосты где хранятся бэкапы стоит физически отключать от сети (на уровне железа, сетевых портов) после выполнения резервирования чтобы в случае полного доступа к программным системам злоумышленник не смог стереть бэкапы и можно было бы восстановиться.

Достаточно ли физической реплики? В случае нестабильной работы хоста мастера есть теоретическая вероятность, что `walsender` передаст поврежденную журнальную запись на реплику. Такая запись может теоретически повредить реплику. Для защиты от этого можно использовать реплику с отложенным (на несколько часов) применением журнальных записей. Откладывание применения устанавливается параметром конфигурации `recovery_min_apply_delay` на реплике.

Утилита pg_receivewal

- подсоединяется по протоколу репликации
- принимает журнальные записи без задержек
- рекомендуется использовать слот репликации
- может получать журналы как с мастера, так и с физической реплики
- скрипты для автозапуска (**systemd**) отсутствуют
- может сжимать журнальные записи
- преимущество в том, что является инициатором соединения, что позволяет обеспечить безопасность
- текущему журнальному файлу даёт суффикс **.partial**

Утилита pg_receivewal

Утилита `pg_receivewal` подсоединяется по протоколу репликации и принимает поток журнальных записей по мере того, как они формируются на экземпляре и сохраняет принятые записи в файлы. Названия файлов и размеры такие же, как те, которые формируются экземпляром. Текущий файл утилита называет как `имя.partial`.

`pg_receivewal`, по умолчанию, накапливает журнальные данные в памяти, а сохраняет в файл при закрытии файла.

Утилита `pg_receivewal` может сжимать сохраняемые журналы (параметр `-Z` или `--compress`), но с записью в файл `.partial` есть нюанс с его размером, разжав этот файл нужно будет доводить его до размера в 16Мб, иначе процесс **startup** его не примет. Доступные алгоритмы сжатия: gzip и lz4.

Рекомендуется использовать слот репликации. Без слота репликации утилита при перезапуске может не получить часть журнальных файлов, в этом случае пройти через потерю при восстановлении не удастся. Отсутствие пропусков в журнальных записях важна. При использовании слота репликации утилита после рестарта запросит недостающие журнальные файлы.

Если нужно чтобы утилита записывала принятые данные без задержки, утилиту нужно запустить с параметром `--synchronous`.

Этот режим также должен использоваться, если утилита будет подтверждать транзакции в режиме синхронной фиксации, установленной параметром конфигурации `synchronous_commit` + `synchronous_standby_names`.

Zero data loss (RPO=0)

- `pg_receivewal --synchronous` может подтверждать транзакции
- Для гарантии отсутствия потерь транзакций при полной потере кластера используются параметры:
  - `synchronous_commit = {on, remote_write}`
  - `synchronous_standby_names`

```
pg_receivewal --create-slot --slot=arch
pg_receivewal -D $HOME/archivelog --slot=arch --synchronous
psql -c "alter system set synchronous_standby_names = pg_receivewal"
psql -c "alter system set synchronous_commit = remote_write"
psql -c "select pg_reload_conf()"
```


Zero data loss (RPO=0)

Утилита может гарантировать (а не только обеспечивать) отсутствие потерь транзакций при восстановлении (Zero data loss), он же нулевой Recovery Point Objective (RPO=0). Обеспечивает параметр `--synchronous` и накат файла `.partial` при восстановлении, а для гарантий нужно установить на пишущем экземпляре (primary, мастер) `synchronous_commit` + `synchronous_standby_names`.

Параметр `synchronous_commit` определяет, на каком этапе сохранения журнальной записи серверный процесс будет выдавать в сессии клиенту сообщение об успешной фиксации транзакции.

Значения параметра конфигурации `synchronous_commit`:

`remote_apply` - к утилите `pg_receivewal` неприменимо, только физические реплики могут накатить журнальную запись. Не стоит ставить даже на репликах, так как скорость фиксации транзакций резко падает, а согласованности мастера и реплики достичь всё равно не удастся - реплика может отдать данные раньше мастера.

`on` - значение по умолчанию. Транзакция подтверждается после того как `pg_receivewal` или реплика получит ответ от своей операционной системы, что она записала журнальные страницы на диск (выполнила fsync)

`remote_write` - процесс `pg_recievewal` или `wal receiver` реплики послал команду своей операционной системе записать журнальные блоки на диск.

Операционная система может задержать эти записи в своём страничном кэше и, если пропадет питание, то блоки будут потеряны.

Это значение - разумный выбор, если вероятность сбоя основного хоста и почти сразу резервного хоста мала, а значение `on` приводит к деградации производительности, которую нельзя устранить другими способами (например, параметром `commit_siblings`)

`local` - транзакция подтверждается после записи в локальный для кластера файл журнала и выполнения синхронизации записи (`fsync` - метод синхронизации, используемый по умолчанию)

`off` - не стоит ставить на уровне кластера. Может ставиться разработчиками приложений на уровне сессий или транзакций.

Если `synchronous_standby_names` пуст, то значения `remote_apply`, `on`, `remote_write` парметра `synchronous_commit` ведут себя `local`.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/app-pgreceivewal.html

Запуск pg_receivewal как службу

```
[Unit]
Description=postgres pg_receivewal
After=network.target
[Service]
Type=simple
Environment=PGDATA=/var/lib/postgresql/tantor-se-18/data
WorkingDirectory=/var/lib/postgresql/archivelog
ExecStartPre=/opt/tantor/db/18/bin/pg_receivewal --create-slot --if-not-exists
--slot=arch
ExecStart=/opt/tantor/db/18/bin/pg_receivewal -D /var/lib/postgresql/archivelog
--slot=arch --synchronous -w -v
Restart=on-failure
RestartSec=10s
User=postgres
Group=postgres
UMask=047
StandardOutput=append:/var/lib/postgresql/archivelog/pg_receivewal.log
StandardError=inherit
[Install]
WantedBy=multi-user.target
```


Запуск pg_receivewal как службу

Для постоянной работы pg_receivewal нужно запустить как службу. Файл службы не поставляется. Пример команд, которые автоматизируют запуск `pg_receivewal` как службы:

```
postgres@tantor:~$ mkdir $HOME/archivelog
touch $HOME/archivelog/pg_receivewal.log
sudo chown postgres:postgres /var/lib/postgresql/archivelog/pg_receivewal.log
sudo chmod 660 /var/lib/postgresql/archivelog/pg_receivewal.log
cat > $HOME/pg_receivewal.service << EOF
[Unit]
Description=postgres pg_receivewal # имя для systemctl service status
After=network.target # запустится после того, как поднимется сеть
[Service]
Type=simple # работает в foreground, так как не возвращает консоль
Environment=PGDATA=/var/lib/postgresql/tantor-se-18/data # переменная окружения
WorkingDirectory=/var/lib/postgresql/archivelog # директория запуска
ExecStartPre=/opt/tantor/db/18/bin/pg_receivewal --create-slot --if-not-exists --
slot=arch
ExecStart=/opt/tantor/db/18/bin/pg_receivewal -D /var/lib/postgresql/archivelog --
slot=arch --synchronous -w -v
Restart=on-failure
RestartSec=10s
User=postgres
Group=postgres
UMask=047 # для создаваемых процессом файлов
StandardOutput=append:/var/lib/postgresql/archivelog/pg_receivewal.log
StandardError=inherit #вывод stderr в файл
[Install]
WantedBy=multi-user.target # чтобы не запускалось в single mode
EOF
sudo cp $HOME/pg_receivewal.service /usr/lib/systemd/system/pg_receivewal.service
sudo chmod 644 /usr/lib/systemd/system/pg_receivewal.service
sudo systemctl daemon-reload
sudo systemctl enable pg_receivewal
sudo systemctl start pg_receivewal
sudo systemctl status pg_receivewal
```

Слот репликации

- три вида: физический, временный физический, логический
- используются для удержания журнальных файлов
- Если клиент не принимает журнальные данные (остановился), то файлы журналов будут удерживаться и заполнят всё свободное место в директории `PGDATA/pg_wal`. Чтобы этого не произошло стоит установить ограничение параметром **`max_slot_wal_keep_size`**
- представление **`pg_replication_slots`** содержит список всех слотов репликации

Слот репликации

При работе экземпляра генерируются журнальные записи и сохраняются в журнальных файлах. Кластер удерживает файлы журналов в целях восстановления после некорректной остановки экземпляра. Это файлы, в которых содержатся журнальные записи от момента начала последней завершенной контрольной точки, всегда удерживаются. Также параметрами кластера можно настроить сколько файлов будет удерживаться и условия удаления.

Слоты репликации используются для удержания журнальных файлов в целях физической и логической репликации, а также резервирования и создания реплики.

Клиенты (`pg_receivewal`, `pg_basebackup`, процессы `walreceiver`, `logical replication worker` экземпляров), подключающиеся по протоколу репликации, могут указывать имя слота репликации. Наличие слотов удерживает файлы журналов, которые не были получены с использованием этих слотов.

Слоты создаются и удаляются командами репликационного протокола, а также функциями и командами SQL. Физические слоты репликации создаются на кластере-мастере и относятся к нему. Каждая реплика использует свой слот. Временный слот существует только на время одной репликационной сессии и удерживает журналы только на время сессии.

Если при создании слота LSN не указан, то он устанавливается при первом подсоединении клиента. Если клиент не принимает журнальные данные (остановился), то файлы журналов будут удерживаться и заполнят всё свободное место в директории `PGDATA/pg_wal`. Чтобы этого не произошло стоит установить ограничение параметром **`max_slot_wal_keep_size`**.

Представление `pg_replication_slots` содержит список всех слотов репликации, существующих в данный момент в кластере баз данных, а также их текущее состояние.

Для создания физического или временного физического слота можно использовать функцию **`pg_create_physical_replication_slot('имя')`**.

Для удаления слота **`pg_drop_replication_slot ('имя')`**.

В 18 версии появился параметр конфигурации **`idle_replication_slot_timeout`** (по умолчанию 0), задаётся в секундах. Длительность неактивности слота можно посмотреть в `pg_replication_slots.inactive_since`.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/functions-admin.html#FUNCTIONS-REPLICATION

Утилита резервирования pg_basebackup

- создаёт резервную копию кластера на работающем экземпляре
- создать копию отдельных баз данных или объектов базы нельзя, только всего кластера
- использует безопасный режим резервирования "pull" - приём файлов, а не "push" - передачу файлов
- подсоединяется к экземпляру через TCP или Unix-сокет
- использует протокол репликации
- стоит использовать слоты репликации
- может резервировать реплику, не нагружая мастер
- прогресс в представлении `pg_stat_progress_basebackup`

Утилита резервирования pg_basebackup

Стандартная утилита резервирования PostgreSQL, остальные утилиты сторонние.

Утилита создаёт резервную копию всего кластера на работающем экземпляре. Не останавливает обслуживание, не блокирует работу пользовательских сессий.

Не может создавать копию отдельных баз данных, табличных пространств, объектов.

Подсоединяется к экземпляру через протокол TCP или Unix-сокет.

По умолчанию, создаёт два соединения по протоколу репликации, в котором есть команды для получения файлов из файловой системы сервера. Через первое соединение создаётся бэкап, одновременно через второе соединение начинают передаваться журнальные файлы. Для подсоединения к экземпляру по протоколу репликации можно дать отдельные разрешения пользователям кластера.

По умолчанию, создаёт резервную копию на том хосте, на котором запущена, но может создавать в директории резервируемого хоста (параметр `-t server:/директория`).

На время резервирования включает `full_page_writes`, если этот параметр был отключен.

По умолчанию, создаёт временный слот репликации для соединения, по которому передаёт журналы. При резервировании **стоит использовать временные или постоянные слоты репликации**, чтобы резервируемый кластер не удалил необходимые для восстановления журнальные файлы, пока создаётся бэкап.

После того, как резервирование закончено, утилита **переключает файл журнала** или дожидается его переключения (если бэкапит реплику), принимает журнал, на котором завершилось резервирование. Только после получения такого файла журнала, на котором завершилось резервирование, бэкап становится автономным.

Может выполнять резервирование, подсоединившись к экземпляру реплики (standby), не нагружая экземпляр основного (primary, мастера) кластера. Это называется "backup offloading". Бэкапы с реплики и мастера одинаковы.

Прогресс резервирования отражается в представлении `pg_stat_progress_basebackup`.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/app-pgbasebackup.html

Создание резервной копии

- **`pg_basebackup`** лучше запустить на хосте, где будет храниться резервная копия (резервный хост)
- параметр **`-D`** задает путь к директории куда бэкапить
  - Директория создаётся на том хосте, где запущена утилита
  - Если директория существует, то она должна быть пустой
- по умолчанию, создаёт два соединения к двум процессам `wal-sender`. По первому соединению передаются файлы с данными, по второму журналы
- Можно ограничить скорость резервирования файлов данных
- Полезный параметр `-P` или `--progress` покажет в какой фазе резервирования находится утилита

Создание резервной копии

Утилита `pg_basebackup` может создавать бэкапы в форматах plain и tar. Второй формат не рассматривается, всё написанное ниже относится к формату plain. Для создания архивов в сжатом виде лучше подходит утилита wal-g. У неё есть высокоэффективный алгоритм сжатия brotli.

Для создания бэкапа достаточно задать директорию параметром `-D директория` или `--pgdata=директория`. Если директория не существует, утилита её создаст и все директории в её пути, если они отсутствуют. Если директория существует, то она должна быть пустой, это защищает от перезаписи файлов, которые могут быть важны. Директория создаётся на том хосте, где запущена утилита.

Если в кластере были созданы табличные пространства (`PGDATA/pg_tblspc` содержит символические ссылки), то будут созданы директории, на которые указывают символические ссылки. То есть структура каталогов табличных пространств будет одинакова на кластере и хосте, где создаётся бэкап. Если бэкап создается на том же самом хосте, то нужно будет указать "мэппинг" - перечислить директории табличных пространств и куда их резервировать параметром:

`-T откуда=куда` или `--tablespace-mapping=откуда=куда`

Все директории указывать по их абсолютным путям, а не относительным. Можно перечислить лишние директории, ошибки не будет. Если же какую-то директорию не указать, то будет выдана ошибка, что директория не пустая. Символические ссылки, находящиеся внутри поддиректории `pg_tblspc` директории с бэкапом, будут указывать на новые директории.

Полезный параметр `-P` или `--progress` покажет в какой фазе резервирования находится утилита.

Параметр `-r скорость` или `--max-rate=скорость` позволяет ограничить скорость резервирования файлов данных, чтобы снизить нагрузку на ввод-вывод. Диапазон от 32 КБ/с до 1024 МБ/с. На скорость передачи журналов влияет только, если выбран метод передачи журналов `fetch`, который не имеет смысла использовать.

Создание резервной копии (продолжение)

- Ждёт завершения контрольной точки перед началом резервирования
  - можно выполнить контрольную точку максимально быстро, указав параметр `-c fast` или `--checkpoint=fast`
  - На реплике утилита не может инициировать контрольную точку и ждёт, пока она выполнится на мастере
- Создаёт файлы **`backup_manifest`** и **`backup_label`**
- Принимает файлы ("pull"), а не передаёт ("push")
  - Приём (pull) - **безопаснее**
  - `pg_receivewal` тоже работает в режиме "pull"

Создание резервной копии (продолжение)

В начале резервирования мастера утилита инициирует контрольную точку. По умолчанию, контрольная точка выполняется в соответствии со значением параметра `checkpoint_completion_target` чтобы не нагружать ввод-вывод, то есть ее длительность можно оценить как `checkpoint_timeout*checkpoint_completion_target`. Если хочется выполнить контрольную точку максимально быстро, можно использовать параметр `-c fast` или `--checkpoint=fast`. На реплике утилита не может инициировать контрольную точку и ждёт, пока она выполнится на мастере.

Параметром `-t` или `--target` можно (но не нужно) резервировать в директорию на хосте кластера, а также резервировать "в никуда" (`--target=blackhole`). Этот режим может использоваться для измерения производительности: какая часть времени резервирования тратится на чтение файлов.

Утилита создаёт файлы `backup_manifest, backup_label`. Файл `backup_label` содержит данные, которые перекрывают значения в файле `pg_control` бэкапа.

Утилита принимает файлы ("pull"), а не передаёт ("push"). Приём (pull) - **безопаснее**, так как при взломе хоста, на котором работает кластер, злоумышленник не сможет подсоединиться к хосту, на котором запускалась утилита `pg_basebackup` и был сохранён бэкап. Перед удалением кластера злоумышленники сначала ищут бэкапы и стирают их. Режим "push" **(когда резервируемый хост подсоединяется к хосту с бэкапами) не безопасен**. При использовании утилит резервирования, работающими в режиме "push", после создания бэкапа нужно изолировать хост с бэкапами от сети, чтобы избежать повреждения бэкапов. По той же причине не стоит использовать передачу журналов параметром `archive_command`, вместо этого стоит использовать утилиту `pg_receivewal`, которая работает в режиме "pull".

Утилита резервирует все директории и файлы, в том числе те, которые ей неизвестны. Поэтому не стоит хранить в `PGDATA` файлы, которые вы бы не хотели видеть в бэкапе, например файлы сообщений большого размера. Однако, утилита **не** резервирует файлы, которые ей известны как те, которые не нужно резервировать. Эти файлы описаны в документации: https://docs.tantorlabs.ru/tdb/ru/18_1/se/continuous-archiving.html

Параметр конфигурации `full_page_writes`

- по умолчанию включён
- не рекомендуется отключать
- защищает от разрыва (torn) блоков данных
- при резервировании реплики должен быть включён на мастере

Параметр конфигурации `full_page_writes`

По умолчанию, параметр `full_page_writes` включён. Это означает, что в журнал записывается всё содержимое (8Кб) каждого блока данных, при первом изменении этого блока в кэше буферов, после каждой контрольной точки. Размер блока данных 8Кб, размер страницы linux 4Кб. В случае пропадания питания в блок данных записаны 4Кб, а другие 4Кб не запишутся и останутся от прежней версии блока. Такой блок называется расщеплённым (fractured) или "разорванным" (torn). Контрольная сумма в блоке не совпадёт и он будет считаться повреждённым.

Почему при первом изменении блока, он записывается в журнал? Потому, что восстановление начинается с LSN начала контрольной точки, которая завершилась до момента начала резервирования. Из журнала в буферный кэш читаются целые образы блоков (а не из файла данных) и к ним применяются изменения из журнала. Если кэш буферов небольшой, то блоки записываются в файлы данных по мере необходимости. Если бы блок записывался при втором изменении, то в процессе восстановления в журнале сначала шла запись об изменении блока и он бы читался из файла данных где он может быть поврежденным, запись журнала не смогла примениться и восстановление бы остановилось с ошибкой типа:

```
PANIC: WAL contains references to invalid pages
```

Процесс восстановления не знает о том, что дальше в журнале может встретиться полный образ блока. При таких ошибках можно использовать `ignore_invalid_pages=on` (только, если `full_page_writes` был включён) надеясь на то, что полный образ блока встретится позже. Если `full_page_writes=off`, то использовать `ignore_invalid_pages` не стоит.

Наличие full page writes позволяет не зависеть от того, сбросит ли операционная система измененные страницы файлов данных на диск из своего кэша или от порядка сброса страниц. Операционная система работает с 4Кб блоками и записывает их в произвольном порядке. Вероятность получить большое число разорванных блоков при пропадании питания или сбое операционной системы высока и **отключать параметр не стоит**.

При резервировании с реплики, `full_page_writes` **должен быть включён на мастере**.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-wal.html#GUC-FULL-PAGE-WRITES

Инкрементальные бэкапы

- могут быть полезны, если размер кластера большой
- увеличивает сложность процедур резервирования, вероятность ошибки
- Процедура:
  - `alter system summarize_wal=on;` перечитать конфигурацию
  - создать полный бэкап утилитой `pg_basebackup`
  - инкрементальные бэкапы создаются утилитой `pg_basebackup` с парметром `-i манифест` или `--incremental=манифест`
  - для восстановления инкрементальные бэкапы накладываются на полный бэкап утилитой `pg_combinebackup`

Инкрементальные бэкапы

В 17 версии появились инкрементальные бэкапы утилитой pg_basebackup и асинхронное отслеживание изменений в блоках. Сначала создаётся полный бэкап, а затем резервируются только блоки, изменившиеся с момента одного из предыдущих бэкапов.

Для создания инкрементального бэкапа нужно:

1) до создания полного бэкапа, включить параметр **`summarize_wal`**, который запустит фоновый процесс `walsummarizer`. Процесс асинхронно создаёт файлы в директории `PGDATA/pg_wal/summaries`, которые покажут, какие блоки должны попасть в инкрементальный бэкап. Перезапуск экземпляра не нужен. Инкрементальные бэкапы могут делаться с реплики, тогда параметр нужно включать на ней. Параметр `summarize_wal` должен оставаться включённым с начала создания бэкапа, от которого будет отсчитываться инкрементальный бэкап. Файлы summaries очищаются через 10 дней (можно настроить параметром `wal_summary_keep_time`). Интервал должен превышать интервал времени между бэкапами. Размеры файлов summaried пренебрежимо малы, используется roaring bitmap. Есть функции, которыми можно получать данные об изменившихся блоках.

2) сделать полный бэкап

3) делать инкрементальные бэкапы, указывая в параметре `-i манифест` (или `--incremental=манифест`) утилиты `pg_basebackup`. В параметре нужно указать путь к `файлу манифеста` того бэкапа, относительно которого будет создаваться инкрементальный бэкап. (обычно последний бэкап). Эти бэкапы образуют звено цепочки бэкапов.

Периодичность бэкапов выбирается исходя из того, какой объем журнальных файлов за это время генерирует кластер. Объем журнальных файлов не зависит от объема кластера. После создания любой (и инкрементальной) копии, WAL-файлы можно удалять, освобождая место.

Недостаток инкрементального резервирования - бóльшая сложность процедур. Чем больше сложность, тем больше вероятность ошибиться в процессе восстановления.

Файлы манифеста позволяют проверять отсутствие повреждений в бэкапах.

Использование инкрементального бэкапа - его накат утилитой `pg_combinebackup` на полный бэкап. На полный бэкап можно наложить последовательность сделанных инкрементальных бэкапов. `pg_combinebackup` перед накатом проверит, что указанные ему бэкапы, образуют допустимую цепочку.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/continuous-archiving.html#BACKUP-INCREMENTAL-BACKUP

Пример инкрементального резервирования

```
pg_basebackup -D $HOME/backup/full1 -P -R -c fast -C --slot=r
39909/39909 kB (100%), 1/1 tablespace
pg_basebackup -D $HOME/backup/incr -P -R -c fast --slot=r -i
$HOME/backup/full1/backup_manifest
  6487/39891 kB (16%), 1/1 tablespace
pg_combinebackup --link $HOME/backup/full1 $HOME/backup/incr -o $HOME/backup/full2
pg_combinebackup: warning: manifest file
"/var/lib/postgresql/backup/incr/backup_manifest" contains no entry for file
"standby.signal"
pg_combinebackup: warning: --link mode was used; any modifications to the output
directory might destructively modify input directories
rm -rf $HOME/backup/full1
rm -rf $HOME/backup/incr
pg_basebackup -D $HOME/backup/incr -P -R -c fast --slot=r -i
$HOME/backup/full2/backup_manifest
  6487/39891 kB (16%), 1/1 tablespace
pg_combinebackup --link $HOME/backup/full2 $HOME/backup/incr -o $HOME/backup/full1
pg_combinebackup: warning: --link mode was used; any modifications to the output
directory might destructively modify input directories
rm -rf $HOME/backup/full2
rm -rf $HOME/backup/incr
```


Пример инкрементального резервирования

Перед созданием полного бэкапа, от которого будут отсчитываться инкрементальные, нужно включить параметр:

```
psql -qc "ALTER SYSTEM SET summarize_wal = on"
psql -qxc "SELECT pg_reload_conf()"
-[ RECORD 1 ]--+--
pg_reload_conf | t
```

Дальше создаётся полный бэкап в директорию `$HOME/backup/full1`.

После создания полного бэкапа можно создавать инкрементальные бэкапы. В пример на слайде инкрементальный бэкап создаётся в директории `$HOME/backup/incr`. При создании нужно в параметре `-i` указать путь к файлу `backup_manifest` в корне директории того бэкапа, от которого будет отсчитываться инкрементальный.

Дальше можно наложить утилитой `pg_combinebackup` инеременталный бэкап (или цепочку инкрементальных) на полный бэкап. В параметре `-o` указывается результатирующая директория. Параметр `--link` позволяет создать жёсткие ссылки, что ускоряет наложение.

Затем снова создаётся инкрементальный бэкап и всё повторяется. Директории `full1` и full2 чередуются.

В примере использовался параметр -R и можно проверить, что бэкап не повреждён, запустив на его директории экземпляр в режиме чтения:

```
echo "port=5433" >> $HOME/backup/full1/postgresql.auto.conf
pg_ctl start -D $HOME/backup/full1
```

Инкрементальное резервирование появилось в 18 версии и утилита `pg_combinebackup` не доделана, ей нужен файл `backup_label` в директории полного бэкапа, иначе она выдаст ошибку:

```
pg_combinebackup: error: could not open file
"/var/lib/postgresql/backup/full1/backup_label": No such file or directory
```

и придётся его откуда-то брать, например, `mv backup_label.old backup_label`

Также, утилита "не знает" о файле `standby.signal` и не копирует его.

Если бэкап не нужен, то его можно удалить:

```
pg_ctl stop -D $HOME/backup/full1
rm -rf $HOME/backup/full1
psql -qc "select pg_drop_replication_slot('r');"
```

Утилита pg_verifybackup

- проверяет файлы в бэкапах созданных утилитой `pg_basebackup`
- рассчитывает контрольные суммы файлов (CRC32C) и сравнивает со значениями в файле manifest_file
- сравнивает файлы со списком файлов в файле манифеста
- без файла манифеста не работает
- Проверяет наличие журнальных записей нужных для синхронизации файлов бэкапа - автономность бэкапа
- не проверяет файлы `postgresql.auto.conf, standby.signal, recovery.signal`

Утилита pg_verifybackup

Если нужно проверить, что файлы в бэкапе не повредились во время хранения, а также то, что во время резервирования были получены нужные для синхронизации журнальные файлы. Гарантий отсутствия повреждений утилита не даёт, их может дать только тестовое восстановление и последующая выгрузка данных на логическом уровне (`pg_dump` и `pg_dumpall`).

По умолчанию, `pg_basebackup` создает файл манифеста (`manifest_file`). Это текстовый файл в формате json с контрольными суммами для каждого зарезервированного файла по алгоритму CRC32C. Содержимое самого манифеста защищено контрольной суммой по алгоритму SHA256. Менять алгоритмы не нужно.

Если файл манифеста есть и не удалялся, то утилитой `pg_verifybackup` можно проверить, что файлы соответствуют манифесту, то есть не повредились в процессе хранения. Утилита выдаёт отчет по исчезнувшим, изменённым и добавленным файлам. Также утилита проверяет самодостаточность (автономность) бэкапа - можно ли по журнальным файлам (если не отказывались от их резервирования) синхронизировать резервную копию на момент завершения бэкапа. Проверка выполняется с помощью утилиты `pg_waldump`, проверяя наличие в зарезервированных файлах журнала нужных журнальных записей. Список нужных записей был передан экземпляром утилите `pg_basebackup` во время резервирования и помещен в файл манифеста.

Утилита не проверяет файлы `postgresql.auto.conf`, `standby.signal`, `recovery.signal` и наличие этих файлов.

В 18 версии у утилиты появилось новшество: она может проверять бэкапы в формате архивов tar.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/app-pgverifybackup.html

Утилита резервирования WAL-G

- Свободно распространяемая утилита командной строки для резервирования кластеров PostgreSQL
- Поставляется с Tantor Postgres
- Резервирует и восстанавливает с высокой скоростью в несколько потоков
- Использует всокоэффективный алгоритм сжатия бэкапов и архивных журналов
- Поддерживает протокол S3 и файловые системы
- Создаёт инкрементальные бэкапы
- Бэкапит данные и журналы в режимах push и pull

Утилита резервирования WAL-G

WAL-G - свободно распространяемая утилита командной строки для резервирования кластеров PostgreSQL. Поставляется с Tantor Postgres.

Преимущество - в многопоточности резервирования, высокоэффективном и не нагружающем процесср алгоритме сжатрия brotli, поддержке протокола S3 и файловой системы.

Есть свободно распространяемые сервера S3, например, rustfs.

WAL-G имеет возможность инкрементального резервирования, которое называет "дельта-копии".

Поддерживается проверка целостности файлов, настройка степени параллелизма для загрузки и скачивания из хранилища.

Утилита не реализует поточная передача журнальных записей, журналы передаются по файлам (WAL-сегментам), для этого можно использовать pg_receivewal.

Использует режим "push", но может резервировать кластер и получать файлы журналов (только целые файлы) по репликационному протоколу в режиме "pull", но только в один поток.

Написана на языке Go, как и свободно распространяемая утилита pgrwl (аналог pg_receivewal).

https://habr.com/ru/companies/tantor/articles/1029864/

https://docs.tantorlabs.ru/tdb/ru/18_1/se/wal-g.html

Демонстрация

- Изменение размера WAL файлов

Демонстрация

Изменение размера WAL файлов

Практика

1. Создание базовой резервной копии кластера
2. Запуск экземпляра на копии кластера
3. Файлы журнала
4. Проверка целостности резервной копии
5. Согласованная резервная копия
6. Удаление файлов журнала
7. Создание архива журнала утилитой pg_receivewal
8. Синхронная фиксация транзакций и pg_receivewal
9. Минимизация потерь данных транзакций

Практика

Создание базовой резервной копии кластера
 Запуск экземпляра на копии кластера
Файлы журнала
Проверка целостности резервной копии
Согласованная резервная копия
Удаление файлов журнала
Создание архива журнала утилитой pg_receivewal
Синхронная фиксация транзакций и pg_receivewal
Минимизация потерь данных транзакций

7

7b

Логическое резервирование

Логическое резервирование

- формирование текстового файла или файлов позволяющие воссоздать объекты
- Сохраняет объекты в том состоянии, в котором они были на начало выгрузки
- Данные выгружаются согласованно - на один момент
- Используются утилиты:
  - **pg_dump**
  - **pg_restore**
  - **pg_dumpall**
  - **psql**
  - команда **COPY**
  - команда утилиты `psql` **\copy**

Логическое резервирование

Резервирование на логическом уровне в PostgreSQL - это формирование текстового файла или файлов ("**дамп**"), позволяющих воссоздать объекты их данные, не отличающихся с точки зрения логики приложения от образа резервируемых объектов.

Данные и объекты, после восстановления из дампа, будут находиться в том состоянии, в котором они были на момент начала выгрузки.

Данные из базы данных выгружаются согласованно - на один момент.

Восстановления на самый последний момент нет.

В файле содержатся команды SQL или текст, на основе которого можно сгенерировать команды SQL.

Дампы делают:

1) команда `COPY TO`
2) команда `psql \copy to`
3) утилита командной строки `pg_dump`
4) утилита командной строки `pg_dumpall`

С дампов восстанавливают:

6) команда `COPY FROM`
7) команда `psql \copy from`
8) утилита командной строки `pg_restore`
9) утилита `psql`

Функционал логического резервирования и восстановления определяется параметрами этих утилит и команд.

Примеры использования

- Переход на новую основную версию
- Проверка целостности данных
- Получение скрипта воссоздания объектов
- Резервирование кластера и общих объектов кластера
- Выгрузка части кластера: отдельной базы данных, содержимого схемы и других объектов

Примеры использования

Логическое резервирование позволяет скопировать данные и/или объекты в другую базу данных на том же или другом кластере той же или другой версии и производителя.

Это используется:

1) Для перехода на новую основную (major, мажорную) версию PostgreSQL. Если время выгрузки и загрузки приемлемо, это оптимальный способ

2) Для гарантированной проверки того, что данные не повреждены. Только выгрузка на логическом уровне может это гарантировать

3) Для простой выгрузки содержимого отдельной базы данных. Физическое резервирование с помощью pg_basebackup не позволяет выгружать отдельно базы данных.

4) Перенести данные в другие системы хранения, например, СУБД других производителей или загрузить данные из сторонних источников

5) Получить текстовый командный файл (скрипт) для установки приложения.

6) Быстро и просто зарезервировать объекты и данные на любых уровнях (кластер, базы, объекты баз, глобальные объекты), получив целостную копию (на один момент времени)

7) Утилита pg_upgrade выгружает общие объекты кластера в процессе своей работы

Сравнение логического и физического резервирования

возможность	Л	Ф
выгружает содержимое отдельной базы данных	+	-
восстановление части объектов	+	-
восстановление на произвольный момент времени	-	+
не зависит от версии, сборки, производителя софта	+	-
обеспечение отказоустойчивости (Durability)	-	+
использует репликационный протокол	-	+
резервирование по сети	+	+
простота использования	+	+

Сравнение логического и физического резервирования

Логическое и физическое резервирование используются в разных целях. Физическое резервирование используется для того, чтобы иметь возможность восстановить данные на самый последний момент времени, то есть без потерь транзакций. Логическое резервирование это не способно сделать, оно может восстановить данные только на момент выгрузки.
Логическое резервирование не должно быть единственным способом резервирования для защиты от повреждения данных.

Логическое резервирование удобно для быстрого создания копии части кластера или переноса объектов между базами данных. Физическое резервирование создаёт копию всего кластера, а его размер может быть большим.

Одно из преимуществ логического резервирования PostgreSQL в том, что формат создаваемых файлов ("дамп") - текстовый со стандартными командами SQL, а не закрытый бинарный формат.

Команда COPY .. TO

- используется для высокоэффективной выгрузки и загрузки данных в одну таблицу
- Можно выгрузить содержимое таблицы или результат любой команды SQL, возвращающей данные
- Данные выгружаются в:
  - файл в файловой системе сервера
  - передаются на стандартный вход (`stdin`) утилите командной строки сервера
  - стандартный вывод `stdout`
- Если команда `COPY` вызывается через сетевое соединение, то стандартный вывод передается через сетевое соединение клиентской программе

Команда COPY .. TO

Особенности выгрузки:

Можно выгрузить, указав имя таблицы (но не представления) или любую команду SQL, возвращающую данные: `WITH`, `SELECT` (любой сложности), `VALUES`, команды с выражением `RETURNING (INSERT, UPDATE, DELETE)`.

Команду нужно заключать в круглые скобки, имя таблицы не нужно.

Если нужно выгрузить все строки, используется имя таблицы.

Если нужно выгрузить часть строк, то `SELECT` с `WHERE`.

Команда `VALUES` мало распространена, но это команда из стандарта SQL.

Вместо имени таблицы указывать имя представления нельзя, но имя представления можно использовать в команде SQL.

Параметры, которыми можно настроить формат и особенности выгрузки: кодировку, символы квотирования, экранирования, как обрабатывать NULL (пустые значения), нужно ли заключать текст в кавычки, нужно ли в первую строку выводить названия столбцов.

Синтаксис:

```
COPY таблица [( столбцы )] | (SELECT|VALUES|.RETURNING) TO 'файл'| PROGRAM
'команда' | STDOUT
 WITH (FORMAT csv|binary, DELIMITER 'символ', NULL 'маркер', HEADER true,
QUOTE 'символ', ESCAPE 'символ', FORCE_QUOTE (столбцы)|*, FORCE_NOT_NULL
(столбцы), FORCE_NULL (столбцы), ENCODING 'имя_кодировки);
```

Цветом помечены опции, которые можно указывать только при выгрузке, а не загрузке в таблицу.

Поддерживается две вариации синтаксиса команды `COPY` (для совместимости с PostgreSQL 9 и 7 версий). Вариации синтаксиса отличаются порядком следования ключевых слов. Это нужно знать, так как в книгах можно встретить примеры с таким синтаксисом. Формат **binary** обрабатывается быстрее, чем текстовый и **csv** форматы, но он менее переносим и выгрузить-загрузить можно только в точно тот же тип данных, а даже не в пределах семейства типов.

Команда `COPY` не входит в стандарт SQL она из языка QUEL, который использовался до перехода на язык SQL.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-copy.html

https://en.wikipedia.org/wiki/QUEL_query_languages

Команда COPY .. FROM

- Загрузка выполняется в одну таблицу
- Данные загружаются из:
  - файла на хосте где запущен экземпляр
  - стандартного вывода произвольной исполняемой программы
  - из стандартного ввода `stdin`
- В процессе загрузки можно сразу установить признак заморозки строк
- Можно задать опциональное выражение `WHERE`, в нём нельзя использовать подзапросы
- представление `pg_stat_progress_copy` - отслеживание работы команды и при загрузке и при выгрузке

Команда COPY .. FROM

Параметры, которыми можно настроить формат и особенности выгрузки:

```
COPY таблица [( столбцы )] FROM 'файл' | PROGRAM 'команда' | STDIN
WITH (FORMAT csv|binary, FREEZE true, DELIMITER 'символ', NULL 'маркер', DEFAULT
'выражение', HEADER true|match, QUOTE 'символ', ESCAPE 'символ', ENCODING
'кодировка', ON_ERROR ignore, LOG_VERBOSITY verbose, REJECT_LIMIT n)
[WHERE выражение];
```

Цветами помечены опции, которые можно указывать только при загрузке в таблицу, а не при выгрузке. Синим - параметры, появившиеся в 17 версии. Зелёным - в 18.

`FREEZE` в процессе загрузки ставит на строки признак заморозки, чтобы в будущем не обновлять блоки в целях заморозки. Таблица, в которую загружаются данные, должна быть создана или усечена в той же транзакции, в которой выполняется команда `COPY`

`HEADER match` используется для проверки того, что названия столбцов и их порядок (из первой строки загружаемых данных) совпадают со столбцами таблицы.

Можно задать опциональное выражение `WHERE`. В этом выражении нельзя использовать подзапросы. При вычислении выражений не видны изменения, которые вносит сама команда `COPY`. На последнее нужно обращать внимание только, если в выражении WHERE вызываются функции с уровнем изменчивости VOLATILE и ожидается, что они будут видеть изменения, а они не видят.

`DEFAULT` задаёт литерал. Если он встретится во входных данных, то будет вставлено значение по умолчанию, установленное в определении таблицы. Аналог: `insert into .. values (.., DEFAULT, ...)`

Пока команда `COPY` (в обоих вариантах `TO` и `FROM`) работает, можно отслеживать её работу через представление `pg_stat_progress_copy`.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-copy.html

Команда psql \copy

- **\copy** это команда psql
- синтаксис похож на команду COPY
- команда использует нижний регистр и чувствительна к нему
- набирается одной строкой
- ";" в конце команды ставить не нужно
- работает с файлами там, где запущена утилита psql, команда COPY работает с файлами через серверный процесс на хосте сервера
- для больших объемов данных COPY эффективнее

Команда psql \copy

\copy это команда psql. Синтаксис \copy похож на синтаксис команды COPY, но действия выполняет утилита psql. Отличия от COPY:

\copy набирается на одной строке, COPY можно набирать на нескольких строках

1) \copy .. from в формате CSV ошибочно обрабатывает единственное в строке значение \. как конец ввода и следующие строки не загружает

2) В COPY можно использовать переменные подстановки, раскрытие обратных кавычек (символ `). Для \copy конец строки всегда воспринимается как аргументы \copy, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек

3) Число обработанных строк \copy .. to stdout не отображает

4) При выполнении \copy ... to stdout вывод направляется в то же место, что и вывод команд утилиты psql. Для чтения/записи стандартного ввода/вывода psql, вне зависимости от источника текущей команды или параметра \o, можно использовать from pstdin или to pstdout

5) утилита psql работает с файлом на том хосте, где запущен psql. Это медленнее, чем работа серверного процесса с файлом. Для больших объёмов данных COPY эффективнее.

Поскольку stdin и stdout, при подсоединении по сети, направляются на клиента, команда COPY может работать с файлами на клиенте с помощью перенаправления ввода-вывода.

Вместо \copy .. to можно использовать COPY ... TO STDOUT и завершить её командой \g имя или \g | программа.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/app-psql.html#APP-PSQL-META-COMMANDS-COPY

Утилита pg_dump

- создаёт логическую резервную копию (дамп) базы данных или её части
- выгружает данные одной базы данных согласованно - на один момент времени
- для выгрузки данных из таблиц по умолчанию использует команду COPY
- выгружает в одном из четырёх форматов: **plain, custom, directory, tar**
- в формате **directory** может выгружать в несколько потоков
- для **custom** и **directory** параметром `-Z` можно выбрать алгоритм и уровень сжатия `zstd, lz4, gzip, none`
- можно не создавать файл, а использовать пайп (pipe):
  `pg_dump параметры | psql параметры`
  `pg_dump -F c параметры | pg_restore параметры`

Утилита pg_dump

`pg_dump` - выгружает содержимое базы данных. Утилита подсоединяется к одной базе данных, использует команды SELECT и COPY, устанавливает блокировки самого низкого уровня `ACCESS SHARE`, такие же, как команда `SELECT`. Блокировки нужны, чтобы во время выгрузки выгружаемые объекты не были удалены. Единственная блокировка, несовместимая с `ACCESS SHARE` это `ACCESS EXCLUSIVE`. Утилита выгружает данные согласованно, то есть на один момент времени, используя моментальный снимок. По умолчанию, использует высокоэффективную команду COPY, но может формировать и набор команд INSERT. Может выгружать отдельно команды создания объектов и отдельно данные.

Выгружает данные в одном из четырёх форматов:

1) `plain` - по умолчанию. Формируется скрипт с набором команд SQL. Для загрузки используется утилита `psql`. Основной недостаток - нельзя указать несколько процессов для одновременной выгрузки

2) `custom` - выгружает в в сжатом виде. Для восстановления используется утилита `pg_restore`, которая может читать формируемые файлы. Выгружать в несколько потоков нельзя, восстанавливать можно. Может использоваться с пайпом:
`pg_dump -F custom параметры | pg_restore параметры`

3) `directory` - создаёт каталог, в котором для каждой таблицы и `lo` будут созданы отдельные файлы и файл оглавления. Для восстановления используется утилита `pg_restore`. Можно указывать число процессов, которые одновременно будут выгружать данные - это основное преимущество по сравнению с форматом `custom`. Восстановление тоже можно проводить в несколько потоков.

4) `tar` - похож на `directory`, только не распараллеливается и не сжимается. Для восстановления используется `pg_restore`. Преимуществ по сравнению с форматом `directory` не имеет.

Использование пайпа (pipe, "канал") позволяет направлять `stdout` на `stdin` утилиты `psql` или `pg_restore` и перегружать данные без создания файла, что экономит место в файловой системе и ускоряет перегрузку данных, так как загрузка и выгрузка работают одновременно.

По умолчанию, для форматов `custom` и `directory` используется сжатие **`gzip`**. Можно выбирать алгоритмы параметром **`-Z gzip zstd`** или **`lz4`** (**`zstd`** и **`lz4`** появились в 16 версии) или отключить сжатие параметром **`-Z none`**.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/app-pgdump.html

Параллельная выгрузка

- возможна только в режиме directory
- количество рабочих процессов задаётся параметром **`-j`** или **`--jobs`**
- прервётся, если после начала выгрузки будут даны команды, устанавливающие эксклюзивную блокировку на выгружаемые объекты
- На время выгрузки устанавливаются блокировки **`ACCESS SHARE`** на все выгружаемые объекты
- При большом количестве объектов можно увеличить значения параметров **`max_locks_per_transaction`**, **`max_connections`**

Параллельная выгрузка

Время выгрузки линейно зависит от объема выгружаемых данных. При выгрузке данные обрабатываются на логическом уровне и узким местом может стать ядро центрального процессора, которое будет обслуживать сессию `pg_dump`. Выгрузку можно распараллелить - использовать рабочие процессы. Одну таблицу может выгружать только один рабочий процесс. Выгружать в параллельном режиме можно только в формате **`directory`**. Число рабочих процессов задаётся параметром `-j N` или `--jobs=N`. При выгрузке создастся `N+1` сессий с базой данных. Серверный процесс, обслуживающий `pg_dump` создаст моментальный снимок и экспортирует его. Рабочие процессы будут использовать этот моментальный снимок, чтобы выгрузка выполнилась на один момент времени (была согласованной).

В параллельном режиме, в начале работы, серверный процесс запрашивает блокировки уровня `ACCESS SHARE` на все объекты, которые будут выгружаться рабочими процессами. Это делается для того, чтобы нельзя было удалить объекты пока работает выгрузка. Число таких блокировок ограничено значением `max_locks_per_transaction * (max_connections + max_prepared_transactions)`. Если число выгружаемых объектов превышает это число, то серверный процесс выдаст ошибку о превышении количества блокировок и завершится не начав выгрузку. В этом случае можно посчитать число запланированных к выгрузке объектов и увеличить значение параметра `max_locks_per_transaction`.

С режимом `ACCESS SHARE` несовместимы только команды, устанавливающие блокировку самого высокого уровня - `ACCESS EXCLUSIVE` (монопольный, эксклюзивный доступ). Это команды `VACUUM FULL, DROP, ALTER, TRUNCATE, LOCK IN ACCESS EXCLUSIVE MODE, REFRESH MATERIALIZED VIEW`. Если какая-то сессия запросит блокировку на объект в режиме в монопольном режиме, то запрос на блокировку будет поставлен в очередь и не даст другим сессиям получить блокировку на объект до истечения значения параметра `lock_timeout`, если он был установлен. Любая попытка доступа к этому объекту будет вставать в очередь, вслед за эксклюзивной блокировкой. Так как рабочие процессы используют собственные сессии, то перед выгрузкой данных из объекта они запрашивают блокировку `ACCESS SHARE` и встают в очереди вслед за `ACCESS EXCLUSIVE`. Чтобы не допустить бесконечного ожидания, рабочие процессы запрашивают блокировку в режиме **`NOWAIT`**. Если рабочий процесс не сможет получить блокировку, то вся выгрузка прекратится.

В 17 версии появился параметр `--sync-method=syncfs` для режима `directory`.

Утилита pg_restore

- обрабатывает дампы в формате **custom, directory, tar**
- может создавать из архивов текстовый файл формата **plain**
- можно восстановить только определения объектов, не загружая данные указав параметр **--schema-only**
- есть вариант загрузки с помощью файла "оглавления" (TOC, title of contents):
  - сформировать файл оглавления архива параметром **--list**
  - закомментировать (удалить, переместить) строки объектов
  - загрузить архив указав оглавление **--use-list**
  - объекты, не указанные в оглавлении, не будут загружены

Утилита pg_restore

`pg_restore` восстанавливает базу данных или объекты из резервной копии, созданной утилитой `pg_dump` во всех режимах, кроме `text`. В режиме `text` создается файл, который выполняется утилитой `psql`, а не `pg_restore`.

Работает в трёх режимах:

1) **Режим загрузки.**

Если указан параметр `-d` имя или `--dbname=имя`, где значение параметра это имя базы данных или строка соединения, то `pg_restore` подключается к этой базе данных и восстанавливает содержимое архива в неё. Возможно распараллеливание загрузки (кроме формата `tar`). Параллельные процессы выполняют наиболее длительные действия, такие как загрузка данных в таблицы и создание индексов.

2) **Режим выдачи списка объектов.**

Если указан параметр `-l` или `--list`, то выводится список объектов архива (TOC, table of contents, оглавление) . Файл списка можно отредактировать, чтобы не загружать часть объектов. Отредактированный файл списка передаётся параметром `-L` файл или `--use-list=файл`

3) **Режим создания скрипта.**

Если параметры `-d` и `-l` не указаны, но указан `-f`, то создаётся скрипт с командами SQL. Формируемый `pg_restore` скрипт будет соответствовать выводу `pg_dump` в формате `plain`.

В 17 версии у утилит `pg_dump` и `pg_restore` появились параметры:

`--filter=файл`, в файле можно указать список объектов, которые нужно включить (include table) или исключить (exclude table) из выгрузки;

`--transaction-size=N` в транзакции будет обрабатывать до `N` объектов базы данных;

https://docs.tantorlabs.ru/tdb/ru/18_1/se/app-pgrestore.html

Возможности pg_restore

- можно восстановить только определения объектов не загружая данные
- загрузить объекты только части схем
- не восстанавливать права владения
- загружать в табличное пространство по умолчанию для базы данных
- восстановить часть секций секционированных таблиц
- восстановить только указанные таблицы, индексы, представления, последовательности, функции, процедуры, триггеры

Возможности pg_restore

Возможности утилиты `pg_restore`, которые задаются параметрами:

1) Параметром `-s` или `--schema-only` можно восстановить только определения объектов, не загружая данные. Позже можно загрузить сами данные, указав параметр `--data-only`. Загрузятся строки таблиц, lo, и установятся значения последовательностей. Имеет смысл использовать `--disable-triggers` для предварительного отключения триггеров перед загрузкой строк в таблицы.

2) Параметры `--clean --if-exists` перед созданием объекта генерируется команда `DROP IF EXISTS`. Без второго параметра выдаются информационные сообщения в `stderr` (обычно это не требуется)

3) `--create` создать базу данных. В параметре подсоединения `-d` нужно будет указать любую существующую базу для выдачи команды создания базы данных и подсоединения к созданной базе данных

4) `--exit-on-error` завершить работу, если возникнет ошибка. По умолчанию, утилита продолжает работать и в конце работы выдаёт число ошибок.

5) `-I` имя. Сгенерировать команду создания указанных индексов. Можно указывать параметр несколько раз, если нужно создать несколько индексов.

6) Для загрузки содержимого не всех, а только части схем можно использовать параметры `-n` или `-N`

7) `--no-owner` не восстанавливать права владения. Используется, если набор ролей в кластере отличается от тех, что были в исходном кластере

8) `-P` восстанавливать только указанные подпрограммы (процедуры и функции)

9) `-t` восстановить только перечисленные "relations" (таблицы, представления, материализованные представления, последовательности, внешние таблицы)

10) `-T` восстановить только указанные триггера

11) `-x` или `--no-privileges` или `--no-acl` не генерировать команды `GRANT, REVOKE`

12) `--section` восстановить секции таблиц

13) Параметром `--no-tablespaces` можно очищать команды `CREATE` от названий табличных пространств. Объекты будут загружаться в табличное пространство по умолчанию. Используется, если в кластере нет табличных пространств, которые были в исходном кластере.

Утилита pg_dumpall

- Создаёт единственный скрипт, позволяющий восстановить образ всего кластера
- Выгружает общие объекты кластера
- Последовательно запускает pg_dump в режиме plain для всех баз данных, которые нужно выгрузить
- Скрипт текстовый, содержит команды SQL
- Скрипт можно выполнить в psql
- В скрипте нет команды создания кластера, кластер нужно создать, запустить экземпляр и указать любую базу данных для начального подсоединения psql
- большая часть параметров утилиты относится к pg_dump, который будет запускать утилита
- выгружает в один поток
- можно использовать пайп (pipe):
  pg_dumpall параметры | psql параметры

Утилита pg_dumpall

Создаёт скрипт, позволяющий восстановить образ кластера, то есть все объекты кластера во всех базах данных и общие объекты. Скрипт содержит команды SQL, его можно выполнить в `psql` и восстановить все базы данных и их содержимое.

Утилита выгружает общие объекты кластера (роли, табличные пространства и права, выданные на параметры конфигурации) и последовательно запускает `pg_dump` для каждой базы данных в кластере в режиме `plain`.

Последовательно подключается к каждой из баз данных. Если используется аутентификация по паролю, может потребоваться вводить его несколько раз, поэтому удобно использовать аутентификацию, не требующую вводить пароль.

В скрипте нет команды создания кластера. При запуске сгенерированного скрипта, `psql` должен подсоединиться к экземпляру кластера, который уже должен быть создан.

Также нужно, чтобы директории табличных пространств находились по тем же путям, по которым они находились в исходном кластере. Создавать сами табличные пространства не обязательно - команды создания табличных пространств будут присутствовать в скрипте.

Утилита выгружает содержимое кластера в один поток.

`pg_dump` запускается последовательно и выгрузка из разных баз данных начинается в разное время. Содержимое каждой из баз данных выгружается согласованно - на момент запуска `pg_dump`.

Использование пайпа (pipe, "канал") позволяет направлять `stdout` на `stdin` утилиты `psql` или `pg_restore` и перегружать данные без создания файла, что экономит место в файловой системе и ускоряет перегрузку данных, так как загрузка и выгрузка работают одновременно.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/app-pg-dumpall.html

Возможности pg_dumpall

- имеет много параметров, большая часть из них является параметрами pg_dump
- можно выгрузить только общие объекты кластера --globals-only
- можно выгрузить только определения ролей --roles-only
- можно выгрузить только определения табличных пространств --tablespaces-only
- можно не выгружать часть баз данных --exclude-database=шаблон
- не добавлять в команды имена табличных пространств --no-tablespaces

Возможности pg_dumpall

Утилита имеет много параметров, но большая часть из них относится к утилите `pg_dump`, которую будет запускать `pg_dumpall`.

Параметр `-g` или `--globals-only` позволяет выгрузить общие объекты кластера: роли и определения табличных пространств. Параметр используется, когда нужно ускорить копирование содержимого кластера: сначала перегружаются роли и табличные пространства, а потом параллельно для каждой базы запускается выгрузка в желаемом режиме. Например, в параллельном: `pg_dump --format=directory --jobs=N`

`--clean` создаёт команды `DROP` для баз данных, ролей, табличных пространств. Полезен даже с пустым кластером, так как встроенные базы данных postgres и template1 будут пересозданы и получат свойства, которые эти базы имели в исходном кластере (параметры локализации). С этим ключом обычно используют `--if-exists`

`-r` или `--roles-only` выгружать только роли, без баз данных и табличных пространств

`-t` или `--tablespaces-only` выгружать только табличные пространства, без баз данных и ролей

`--exclude-database=шаблон` не выгружать базы данных с именами соответствующими шаблону

`--no-tablespaces` не добавлять в команды имена табличных пространств. С этим параметром все объекты будут созданы в табличном пространстве по умолчанию

Статистика не выгружается и команды её сбора не создаются. После загрузки можно её собрать не дожидаясь автоматического сбора.

Параметр `--binary-upgrade` предназначен для использования утилитой `pg_upgrade` (совместно с `--globals-only` или `--schema-only`), он позволяет сохранить названия файлов данных для объектов. Использование для иных целей не рекомендуется и не поддерживается.

Строки большого размера

- Типы данных text и bytea могут хранить данные размером до 1Гб
- Строки размером больше 1Гб могут требовать особого обращения
- Поля при выгрузке в текстовом виде могут увеличиваться в размерах

Строки большого размера

Первая проблема

Типы данных `text` и `bytea` могут хранить поля размером до 1Гб. В процессе выгрузки (`COPY`) или обработки данных любыми командами выделяется буфер, размер которого не может превышать 1Гб. По умолчанию команда COPY выводит значения полей в текстовом формате. В этом формате для символов типа перехода на новую строку, табуляции, забоя используются спецпоследовательности типа `\r \t \b` которые занимают два байта. В этом формате поле, содержащее спецсимволы может превысить 1Гб. При выгрузке **поля** `bytea` в текстовом виде, его размер также увеличивается и будет выдана ошибка:

```
ERROR:  out of memory
DETAIL:  Cannot enlarge string buffer containing 1073741822 bytes by 1 more bytes.
```

Вы этом случае можно использовать бинарный формат: `COPY .. TO .. WITH BINARY;`

Вторая проблема

При обработке **строк** память выделяется динамически, увеличиваясь на размер поля и при выгрузке строки может возникать ошибка:

```
ERROR:  out of memory
DETAIL:  Cannot enlarge string buffer containing 536870913 bytes by 536870912 more bytes.
```

При выгрузке любых типов данных, в том числе из lob размер строки не может превышать 1Гб. Такие поля придётся выгружать по частям: по столбцам; фильтруя строки и выгружая проблемные строки отдельно по полям.

Параметр утилиты `pg_dump -B` или `--no-large-objects` позволяет не выгружать `lo`. Для работы с lob имеются функции `lo_import()` и `lo_export()`.

Комментарий

При работе со строками большого размера серверные процессы могут пытаться выделять память больше 1Гб. Например, текущий размер строкового буфера 999Мб, идет попытка увеличить для обработки ещё ещё одного поля размером 1Гб, в операционную систему уходит запрос на выделение ещё 1Гб. Если физической памяти под этот 1Гб нет, то этот серверный процесс (или любой процесс) получает сигнал `9 (SIGKILL)` от `oom-kill`. Если физической памяти достаточно, то серверный процесс выдает клиенту `"ERROR: out of memory"` и продолжает работать.

Параметр enable_large_allocations

- параметр СУБД Tantor Postgres, который увеличивает размер StringBuffer с 1 гигабайта до 2 гигабайт

```
postgres=# select * from pg_settings where name like '%large%'\gx
name       | enable_large_allocations
setting    | off
category   | Resource Usage / Memory
short_desc | whether to use large memory buffer greater than 1Gb, up to 2Gb
context    | superuser
vartype    | bool
boot_val   | off
```

- может устанавливаться на уровне сессии и утилитами pg_dump, pg_dumpall

```
postgres@tantor:~$ pg_dump --help | grep alloc
  --enable-large-allocations   enable memory allocations with size up to 2Gb
```

- проблема возникает со строкой таблицы config приложений 1С:ERP, Комплексная автоматизация, Управление производственным предприятием

Параметр enable_large_allocations

параметр СУБД Tantor Postgres, который **увеличивает размер StringBuffer в локальной памяти процессов экземпляра с 1 гигабайта до 2 гигабайт**. Размер одной строки таблицы при выполнении команд SQL должен поместиться в StringBuffer. Если не поместится, то любому клиенту с которым работает серверный процесс, выдастся ошибка, в том числе утилитам `pg_dump` и `pg_dumpall`. Размер поля строки таблицы всех типов не может превышать 1Гб, но столбцов в таблице может быть несколько и размер строки может превысить и гигабайт и несколько гигабайт.

Утилита `pg_dump` может отказаться выгружать такие строки, так как она не использует опцию `WITH BINARY` команды `COPY`. Для текстовых полей непечатный символ, занимающий один байт будет заменен последовательностью печатных символов размером 2 байта (например, `\n`) и текстовое поле может увеличиться в размере до двух раз.

```
postgres=# select * from pg_settings where name like '%large%'\gx
name       | enable_large_allocations
setting    | off
category   | Resource Usage / Memory
short_desc | whether to use large memory buffer greater than 1Gb, up to 2Gb
context    | superuser
vartype    | bool
boot_val   | off
```

и у утилит командной строки:

```
postgres@tantor:~$ pg_dump --help | grep alloc
  --enable-large-allocations   enable memory allocations with size up to 2Gb
```

Параметр можно установить на уровне сессии. StringBuffer выделяется динамически в процессе обработки каждой строки, а не при запуске серверного процесса. Если таких строк нет, параметр не влияет на работу серверного процесса.

Такая проблема возникает со строкой таблицы config приложений 1С:ERP, Комплексная автоматизация, Управление производственным предприятием. Пример:

```
pg_dump: error: Dumping the contents of table "config" failed: PQgetResult() failed.
Error message from server: ERROR: invalid memory alloc request size 1462250959
The command was: COPY public.config
(filename, creation, modified, attributes, datasize, binarydata) TO stdout;
```

Демонстрация

- Обработка строк большого размера

Демонстрация

Обработка строк большого размера

Практика

1. Использование утилиты pg_dump
2. Формат custom и утилита pg_restore
3. Формат directory
4. Сжатие и скорость резервирования
5. Команда COPY

Практика

Использование утилиты pg_dump
Формат custom и утилита pg_restore
Формат directory
Сжатие и скорость резервирования
Команда COPY

8

8a

Физическая репликация

Физическая репликация

- Один основной (primary, master, ведущий) кластер баз данных - допускает внесение изменений в данные
- Один или несколько резервных (standby, ведомых) кластеров
- Резервные кластера (физические реплики или просто реплики) получают журнальные данные и применяют к своим файлам
- Реплики могут обслуживать запросы на чтение - режим горячего резерва (hot standby)
- Реплики являются физической резервной копией основного кластера, которая обновляется

Физическая репликация

До сих пор мы рассматривали работу с одним кластером, обслуживаемым одним экземпляром на одном хосте. Один хост может выйти из строя, как и центр обработки данных, в котором находится хост. Для высокой доступности (High Availability, HA) содержимого баз данных нужно использовать по крайней мере ещё один хост со своей системой хранения файлов и сделать так, чтобы при отказе первого хоста второй хост имел те же самые данные, что и первый и смог обслуживать приложения-клиенты.

В этой главе мы рассмотрим простое и наиболее распространённое решение обеспечения высокой доступности - репликацию изменений (журнальных записей) в данных на физическом уровне (страниц файлов данных) - "физическую репликацию".

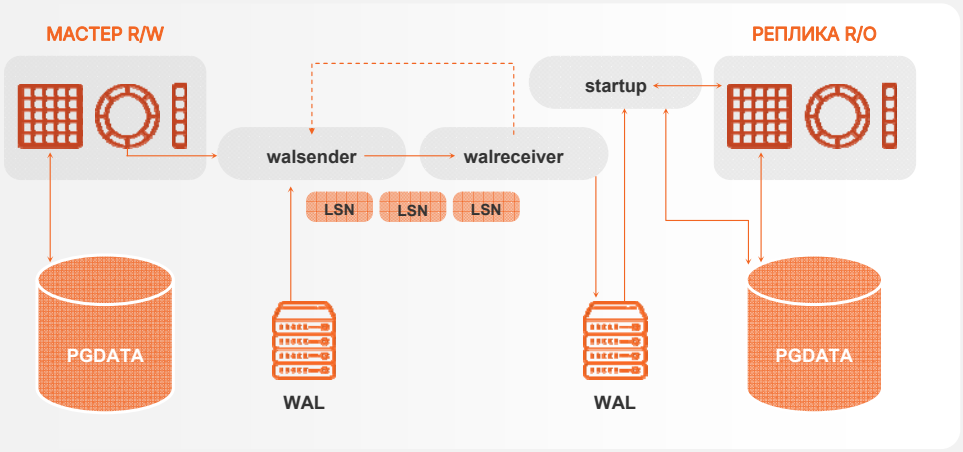
Модель использования: имеется кластер с которым работают клиентские приложения. Его называют основной (primary) или мастер (master, ведущий) кластер. Основной кластер только один в конфигурации, использующей физическую репликацию. Делается физическая резервная копия файлов этого кластера на резервный хост. Эта копия называется резервным (standby) кластером или физической репликой или просто "реплика". Настраивается передача журнальных данных на хост резервного кластера. Запускается экземпляр на резервном хосте. Экземпляр принимает и накладывает на файлы резервного кластера изменения. Таких резервных кластеров может быть несколько, они могут располагаться на разных хостах.

Резервный кластер обычно открывается в режиме для чтения данных (горячий резерв, hot standby) и может обслуживать запросы. При этом резервный кластер продолжает накладывать изменения на свои файлы и они становятся видны сессиям, подключённым к экземпляру, обслуживающему резервный кластер. На резервный кластер можно перенести долгие, аналитические запросы, которые обычно формируют отчёты.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/high-availability.html

Мастер и реплика

- мастер и реплики должны использовать ту же основную версию PostgreSQL
- кластер реплицируется целиком
- директории табличных пространств могут отличаться



Мастер и реплика

Мастер и реплики должны использовать ту же основную версию PostgreSQL. Кластер реплицируется целиком - со всеми базами данных. Исключить из репликации часть объектов нельзя. Директории табличных пространств могут отличаться, так как на директорию табличного пространства указывает только символическая ссылка в директории `PGDATA/pg_tblspc`.

В реплики нельзя вносить изменения, поэтому они не могут создавать свои собственные журнальные записи. Журнальные файлы реплики содержать журнальные записи мастера.

Реплики могут передавать журнальные записи мастера по протоколу репликации другим клиентам например, другим репликам. Это называется каскадная репликация (cascading replication). Реплики, получающие журнальные записи не от мастера, а от другой реплики, не могут подтверждать транзакции в синхронном режиме, их нельзя указывать в параметре `synchronous_standby_names`

Реплики и архив журнала

- Обычно используется поточная репликация с использованием слота репликации: процесс walreceiver подсоединяется к процессу walsender
- Может использоваться архив журналов, заполняемый командой в параметре `archive_command` или формируемый утилитой `pg_receivewal`
- Если реплика не сможет получить журнал через walreceiver, то использует команду `restore_command`

Реплики и архив журнала

Журнальные записи можно передавать на реплики всеми доступными способами. Реплика может забирать журнальные файлы из произвольной директории, например, директории куда, складывает полученные файлы утилита `pg_receivewal` или любая другая (например, указанная в параметре `archive_command`). Больше возможностей даёт получение журнальных записей по протоколу репликации с использованием слота репликации фоновым процессом экземпляра реплики, называющийся `walreceiver`.

Реплика может быть настроена на использование, как слота репликации, так и файлов журналов (параметр `restore_command` на реплике). Если реплика не сможет (по любым причинам) получить журнальную запись по протоколу репликации, она будет пытаться (с интервалом `wal_retrieve_retry_interval`, по умолчанию раз в 5 секунд) выполнить команду, указанную в `restore_command` и, если команда успешно выполнится, попробует прочесть файл журнала. При этом реплика будет пытаться восстановить соединение по протоколу репликации (с тем же интервалом `wal_retrieve_retry_interval`) и, если сможет получать журнальные записи по протоколу репликации, то будет его использовать. Если walsender не передаст ничего в течение `wal_receiver_timeout`, то сокет будет разорван и выполнится попытка заново подсоединиться.

Настройка мастера

- Настроить возможность подсоединения реплик по протоколу репликации
- Проверить значения параметров конфигурации:
  - **`wal_level`** по умолчанию replica
  - **`max_walsenders`** по умолчанию 10
  - **`max_replication_slots`** по умолчанию 10
  - **`max_slot_wal_keep_size`** по умолчанию -1 без ограничений, стоит установить ограничение

Настройка мастера

Основной кластер (мастер) скорее всего используется и успешно обслуживает клиентские приложения. Создать и настроить реплику можно без простоя обслуживания клиентов мастером. Реплика подсоединяется по протоколу репликации, нужно настроить параметры аутентификации для роли под которой будет подсоединяться реплика. Может потребоваться поменять значения параметров конфигурации кластера, которые не меняются без рестарта экземпляра, Параметры:

`wal_level` (по умолчанию `replica`) Должно быть значение `replica` или `logical`. При изменении значения требуется **перезапуск** экземпляра

`max_walsenders` (по умолчанию 10) Одна реплика использует одно соединение к walsender, но при сетевом сбое может переподсоединиться, при этом предыдущее соединение может существовать до `walsender_timeout`. `pg_basebackup` может использовать два соединения. При изменении значения требуется перечитывание параметров конфигурации.

`max_replication_slots` (по умолчанию 10). Должно быть не меньше числа существующих слотов, иначе экземпляр не запустится. Каждая реплика (независимо от каскадирования), pg_receivewal, pg_basebackup могут использовать по одному слоту. При изменении значения требуется **перезапуск** экземпляра.

`max_slot_wal_keep_size` (по умолчанию -1 без ограничений) Максимальный размер журнальных файлов, который может оставаться в каталоге pg_wal после выполнения контрольной точки для слотов репликации. Если реплика использует слот репликации и не подсоединяется к мастеру, то журнальные файлы удерживаются мастером для такого слота. Если не установлено ограничения, то файлы журнала заполнят всю файловую систему и экземпляр аварийно остановится. Чтобы этого не допустить, стоит установить ограничение. Однако, реплике придется получать файлы журналов откуда-то еще или реплику придётся удалить. Если реплика больше не нужна, нужно не забыть удалить её слот. При изменении значения требуется перечитать конфигурацию.

`walsender_timeout` (по умолчанию 60 секунд) Задаёт период времени, по истечении которого неактивные соединения по протоколу репликации разрываются. При изменении значения требуется перечитать конфигурацию.

`synchronous_standby_names` и `synchronous_commit` Конфигурируют после создания реплик для гарантий защиты от потерь транзакций при потере мастера. Можно менять без рестарта экземпляра.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-replication.html

Создание реплики

- создание резервной копии утилитой pg_basebackup
- параметр -R устанавливает конфигурационные параметры для реплики
- параметр **--create-slot** (-C) и **--slot=имя** (-S) создаёт слот для удержания WAL-журналов до запуска экземпляра реплики
- установить параметр конфигурации cluster_name
- проверить значения параметров конфигурации приведя их в соответствие с возможностями хоста реплики
- Настроить службу для автоматического запуска экземпляра реплики и запустить экземпляр реплики

Создание реплики

Реплика может работать на том же узле, что и мастер, но это не защищает от потери хоста, поэтому используется в целях обучения и тестирования. В остальных случаях, реплика должна работать на хосте, отличном от хоста мастера. Для упрощения настройки стоит использовать директории `PGDATA` и табличных пространств по тому же пути, как на мастере.

При создании реплики утилитой `pg_basebackup` удобно:

1) использовать параметры `-C (--create-slot)` и `-S (--slot=имя)` для создания постоянного слота репликации. Через этот слот будут передаваться журналы утилите `pg_basebackup`, а после окончания её работы слот не будет удалён, он будет удерживать журнальные файлы, чтобы мастер их не удалил до подсоединения реплики

2) использовать параметр `-R (--write-recovery-conf)` в файл `postgresql.auto.conf` реплики будут записаны параметры конфигурации:

а) `primary_conninfo` - адрес, по которому `pg_basebackup` подсоединялся к мастеру. Параметр указывает адрес и параметры сетевого соединения, с которыми процесс walreceiver экземпляра реплики будет подсоединиться к экземпляру мастера. walreceiver подсоединяется к walsender на экземпляре мастера

б) `primary_slot_name` имя слота репликации, который использовала утилита `pg_basebackup` и который удерживает журнальные файлы до подсоединения walreceiver реплики к мастеру. Параметр не действует, если кластер не является репликой или не задан `primary_conninfo`

в) в `PGDATA` создаётся файл `standby.signal`, наличие которого указывает процессу startup находиться в режиме постоянного восстановления и не останавливаться.

3) После создания реплики, установить на ней значение параметра `cluster_name`. Изменение значения требует перезапуск экземпляра реплики. Параметр устанавливает значение по умолчанию для опции `application_name` параметра `primary_conninfo`.

`application_name` устанавливает название реплики, которое может использоваться на мастере в параметре `synchronous_standby_names`. Также значение `cluster_name` будет выводиться в названии серверных процессов экземпляра, что удобно для мониторинга. Если значение `cluster_name` не установлено или пусто, то для `application_name` используется значение `walreceiver`.

4) Проверить и, если нужно, поменять значения в файлах параметров `postgresql.conf`, `pg_hba.conf` Эти файлы копируются с мастера и могут не подходить к хосту реплики. Например, на хосте реплики может быть меньше физической памяти, которая не сможет вместить `shared_buffers`. Если реплика находится на том же хосте что и мастер, то нужно изменить параметр `port`.

5) Создать службу для автоматического запуска экземпляра реплики и запустить экземпляр реплики через неё.

Слоты репликации

- Процесс walreceiver реплики использует физический слот
- список слотов - в представлении `pg_replication_slots`
- Функция создания `pg_create_physical_replication_slot('имя')`
- Удаление слота любого типа `pg_drop_replication_slot('имя')`
- Создание копии слота `pg_copy_physical_replication_slot('имя', 'имя_создаваемого')`

Слоты репликации

Причин не использовать слоты репликации нет. Слот использует и утилита `pg_receivewal` и реплики. Слоты бывают трёх видов: физический, временный физический, логический. Логический используется для логической репликации изменений в таблицах двух основных кластеров. Временные слоты используются в процессе создания автономной резервной копии, обычно предназначенной для создания клона или восстановления на момент окончания резервирования. Для передачи (трансляции) журнальных записей на кластера-реплики используются физические слоты репликации.

Физический слот репликации удобно создавать при создании бэкапа, который и будет представлять собой реплику. Это позволит "бесшовно" (без потери журнальных файлов в промежутке времени между окончанием бэкапа и запуском экземпляра реплики) запустить экземпляр реплики. При запуске экземпляра реплики запускается процесс `walreceiver`, который принимает журнальные записи и сохраняет их в `PGDATA/pg_wal` реплики. Также, запускается процесс startup, который накатывает содержимое директории `PGDATA/pg_wal` и периодически (параметр **`wal_retrieve_retry_interval`**) проверяет не появилось ли там что-то новое.

Функции работы с физическими слотами:

`pg_create_physical_replication_slot('имя', false, false)` - слоту нужно дать имя. второй параметр важен по умолчанию false - LSN резервируется при первом подключении клиента потоковой репликации. Если true, то LSN для этого слота репликации должен быть зарезервирован немедленно. Третий параметр по умолчанию false - слот физический постоянный, если true то временный.

`pg_drop_replication_slot('имя')` - удаляет слот любого типа

`pg_copy_physical_replication_slot('имя', 'имя_создаваемого', false)` - создаёт слот и инициализирует его LSN существующего слота. Используется, если при создании двух реплик используется один и тот же бэкап.

Список слотов репликации можно посмотреть в представлении `pg_replication_slots`.

Параметры конфигурации на репликах

- на мастере не играют роли, но могут быть заранее установлены
- многие не меняются без рестарта экземпляра реплики
- реплика обслуживает запросы (**`hot_standby=on`**)
- реплика применяет журнальные записи без задержки (**`recovery_min_apply_delay=0`**)
- `walreceiver_status_interval` по умолчанию 10 секунд. Обратная связь будут отправляться не чаще и горизонт событий баз данных на мастере будет сдвигаться не чаще, чем это значение
- `wal_retrieve_retry_interval` по умолчанию - 5 секунд. Время ожидания репликой поступления журнальных данных из любых источников

Параметры конфигурации на репликах

Часть параметров конфигурации настраивают работу реплики. В процессе эксплуатации конфигурации мастер-реплики одна из реплик может становиться мастером, а бывший мастер репликой. Это называют сменой ролей кластеров баз данных в физической репликации. Следующие параметры можно установить заранее на мастере и, при создании реплик, эти параметры будут использоваться на репликах:

`walreceiver_status_interval` по умолчанию 10 секунд. Обратная связь будут отправляться не чаще и горизонт событий баз данных на мастере будет сдвигаться не чаще, чем это значение.

`wal_retrieve_retry_interval` по умолчанию - 5 секунд. Время ожидания репликой поступления журнальных данных из любых источников (потоковая репликация, архив журналов, локальный pg_wal), прежде, чем повторять попытку получения (walreceiver отправляет запрос walsender и ждёт ответа, startup выполняет `restore_command`, startup читает `PGDATA/pg_wal`)

`recovery_min_apply_delay` по умолчанию ноль. Параметр будет рассмотрен позже.

`hot_standby` по умолчанию `on`. Определяет, можно ли будет подключаться к экземпляру и выполнять запросы или нельзя. Параметр играет роль только в режиме реплики или восстановлении. Значение влияет на поведение экземпляра при восстановлении и обслуживании реплики. Например, если `hot_standby=off`, то значение другого параметра `recovery_target_action=pause` действует как shutdown, а если `hot_standby=on`, то как promote. Параметр требует перезапуск экземпляра.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-replication.html#RUNTIME-CONFIG-REPLICATION-STANDBY

Конфликты на реплике

- обратная связь отключена (**hot_standby_feedback=off**)
- walreceiver_timeout по умолчанию 60 секунд
  - walreceiver реплики может обнаружить отсутствие ответа от walsender, и заново переподсоединиться
- max_standby_streaming_delay и max_standby_archive_delay по умолчанию равны 30 секунд. Максимально допустимое время задержки применения WAL

Конфликты на реплике

Если hot_standby=on, то действуют параметры:
hot_standby_feedback ("обратная связь") - по умолчанию off. Устанавливает будет ли walsender реплики (в режиме hot_standby=on так как при off запросов на реплике нет) сообщать walsenderу, от которого получает журналы, данные о запросах, которые выполняет в данный момент. При каскадной репликации данные от всех реплик (в каскаде) передаются мастеру. Мастер удерживает "горизонт событий баз данных" по самому длительному запросу (или транзакции в режиме REPEATABLE READ) среди всех реплик, на которых включена обратная связь. Это приводит к тому, что устаревшие версии строк не удаляются не только (авто)вакуумом, но и быстрой очисткой (HOT cleanup), зато благодаря этому запросы не реплике не прерываются и имеют возможность доработать и выдать все данные.
walreceiver_timeout по умолчанию 60 секунд. walreceiver реплики может обнаружить отсутствие ответа от walsender, и заново переподсоединиться.
max_standby_streaming_delay и max_standby_archive_delay по умолчанию равны 30 секунд. Максимально допустимое время задержки применения WAL.
Если процесс startup будет заблокирован запросом эти параметры определяют сколько он будет ждать перед тем, как прервёт блокирующие его запросы:
ERROR: canceling statement due to conflict with recovery
DETAIL: User query might have needed to see row versions that must be removed.
Типы конфликтов:
1) Конфликты моментальных снимков. Если на мастере вычищаются старые версии строк, нужные запросам на реплике.
2) Конфликты блокировок. startup воспроизводит блокировку ACCESS EXCLUSIVE на таблице, с которой работает запрос реплики.
3) Конфликты закреплений блока в буфере (BufferPin). Проигрывание заморозки и быстрой очистки блоков на реплике требует эксклюзивного закрепления блока процессом startup, а блок может быть закреплён потоком запросов на реплике.
4) Взаимоблокировка, конфликты при удалении табличных пространств и баз данных. Взаимоблокировки снимаются автоматически.
https://habr.com/ru/articles/1027704/

Горячая реплика

- по умолчанию включена (параметр **hot_standby=on**)
- реплика принимает соединения и выполняет запросы
- утилита **pg_basebackup** может выполнять резервирование реплики вместо мастера (backup offloading)
- временные таблицы на реплике нельзя использовать, расширение pg_variables можно

Горячая реплика

Физическая реплика обслуживается своим экземпляром. Реплика может использоваться для обслуживания команд не меняющих данные - запросов. При переносе читающей логике стоит учитывать, что гарантировать актуальность возвращаемых репликой данных нельзя. Если параметр конфигурации на мастере `synchronous_commit` установлен в значение `remote_apply`, то реплика своим сессиям может отдавать (гарантировать такое поведение нельзя) данные раньше чем мастере. То есть если есть две сессии от клиента к мастеру и реплике, в этих сессиях по времени одновременно даётся команда `SELECT` к строкам, которые только что были изменены транзакцией в параллельной сессии мастера, сессия с репликой может выдать измененные этой транзакций данные, а сессия мастера не выдать. Гарантировать синхронность отдачи тех же самых данных нельзя. Переносить на реплику всю читающую нагрузку не стоит. На реплику можно перенести часть логики приложения, которая строит отчёты и выполняет аналитические запросы. Это запросы, длительность выполнения которых существенно превышает репликационный лаг (задержку в передаче и накате журнальных записей) и он не играет роли для логики приложения.

По умолчанию параметр конфигурации `hot_standby=on` и физическая реплика работает в режиме горячего резерва - может обслуживать команды, не меняющие данные. Например, команды выборки `SELECT, WITH, COPY TO`, а также команды `BEGIN TRANSACTION, COMMIT, ROLLBACK` - эти команды нужны, чтобы иметь возможность выполнять запросы на один момент времени, что реализуется открытием транзакции на реплике в режиме `REPEATABLE READ`. Уровень `SERIALIZABLE` не поддерживается и не отличается для чтения от `REPEATABLE READ`:

```
ERROR:  cannot use serializable mode in a hot standby
HINT:  You can use REPEATABLE READ instead.
```

Результаты команд `COMMIT` и `ROLLBACK` не будут отличаться, они используются только для закрытия транзакции которая ничего не меняла. Использование временных таблиц невозможно.

Одна из полезных возможностей реплики - утилиты резервирования могут создавать резервные копии подсоединившись к реплике, тем самым можно снять нагрузку с мастера перенеся резервирование на реплику.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/hot-standby.html

Обратная связь с мастером

- обратная связь по умолчанию отключена
  `hot_standby_feedback=off`
- обратная связь удерживает горизонт очистки всего мастера в длительность самого долгого запроса на базах реплики или транзакции (уровень транзакций REPEATABLE READ)
- Вместо обратной связи можно использовать параметры
  - `max_standby_streaming_delay` и
  - `max_standby_archive_delay`

Обратная связь с мастером

По умолчанию `hot_standby_feedback=off` и мастер не принимает во внимание, что на репликах выполняются команды `SELECT`. Это значит, что на мастере могут выполняться команды `DROP`, передаваться на реплики, применяться процессом startup и SELECT обращавшийся к объекту его не найдёт и выдаст ошибку. А после `DROP DATABASE` на мастере и проведения этой команды на реплике, сессии на реплике с этой базой данных будут прерваны. Команды изменения объектов на мастере выполняются нечасто, да и смысла дорабатывать запросам если объект решили удалить нет. Практическое влияние на запросы на реплике оказывает вакуумирование (в том числе автоматическое) на мастере, вычищающее старые версии строк. Старые версии строк образуются после их удаления или обновления, но не после вставок. Запрос на реплике может прерваться, даже если вакуум не отрабатывал на таблице, а по причине обновления `HOT (Heap-Only tuples)`.

Если нужно, чтобы запросы на репликах выполнялись без ошибок можно:
1)установить значения параметров `max_standby_streaming_delay` и `max_standby_archive_delay` в длительность самого длинного запроса. Если запрос превысит это время, то он прервется с ошибкой не всегда, а только при наличии конфликта. Задержка в применении конфликтующих журнальных записей может увеличить отставание реплики от мастера ("лаг") вплоть до значений этих параметров. Все сессии на реплике будут выдавать данные с задержкой. Также если захочется сделать из реплики мастер, возможна задержка на применение журнальных записей для устранения лага.
2)включить обратную связь. Это повлияет на мастер - он не сможет очищать старые версии строк, так как запросы на репликах будут удерживать горизонт событий баз данных мастера. Удержание горизонта влияет на вакуумирование и быструю очистку HOT cleanup.

Мониторинг горизонта

- Оценка горизонта текущей базы данных:
  - **backend_xmin** из **pg_stat_activity**
  - Длительность самого долгого запроса или транзакции: **max(now()-xact_start)** из **pg_stat_activity**
  - Столбец **xmin** представления **pg_replication_slots** при использовании слотов репликации
  - Столбец **backend_xmin** представления **pg_stat_replication** на мастере - что получили walsender по обратной связи

```
select age(backend_xmin), extract(epoch from (clock_timestamp()-xact_start)) secs, pid, datname database,
state from pg_stat_activity where backend_xmin IS NOT NULL OR backend_xid IS NOT NULL order by
greatest(age(backend_xmin), age(backend_xid)) desc;
  age   |    secs     |  pid   | database |        state
--------+-------------+--------+----------+---------------------
 175455 | 1425.651346 | 255554 | postgres | idle in transaction
      1 |    0.001878 | 255547 | postgres | active
      1 |    0.001213 | 255626 | postgres | active
```


Мониторинг горизонта

Проверять сдвигается ли горизонт баз данных важно для оценки того сможет ли автовакуум эффективно очищать старые версии строк, а HOT выполнять очистку внутри страниц и оценивать последствия включения обратной связи.

Число отменённых запросов в базах данных реплики, с момента сброса статистики, можно просмотреть в представлении `pg_stat_database_conflicts` на реплике.

Представление не показывает приостановки работы процесса startup из-за ожидания завершения запросов или ожидания получения buffer pin.

Мониторинг запросов и транзакций на базах данных кластера:

```
select age(backend_xmin), extract(epoch from (clock_timestamp()-xact_start))
secs, pid, datname database, state from pg_stat_activity where backend_xmin IS
NOT NULL OR backend_xid IS NOT NULL order by greatest(age(backend_xmin),
age(backend_xid)) desc;
```

```
   age   |    secs     |  pid   | database |        state
---------+-------------+--------+----------+---------------------
  175455 | 1425.651346 | 255554 | postgres | idle in transaction
       1 |    0.001878 | 255547 | postgres | active
       1 |    0.001213 | 255626 | postgres | active
```

Представление `pg_replication_slots` содержит состояние всех слотов репликации. Столбец `xmin` содержит идентификатор старейшей транзакции, для которой должен удерживаться горизонт. Пример запроса:

```
select max(age(xmin)) from pg_replication_slots;
```

Представление `pg_stat_replication` на мастере содержит по одной строке для каждого `walsender`. Столбец `backend_xmin` содержит идентификатор старейшей транзакции ("xmin") реплики, если включена обратная связь (`hot_standby_feedback=on`).

https://docs.tantorlabs.ru/tdb/ru/18_1/se/monitoring-stats.html

Мониторинг горизонта

- горизонт баз данных кластера в количестве номеров транзакций, отстоящих от текущей:

```
select datname, greatest(max(age(backend_xmin)), max(age(backend_xid))) from pg_stat_activity
where backend_xmin is not null or backend_xid is not null group by datname order by datname;
```

- длительность самого долгого запроса или транзакции, который и удерживает горизонт:

```
select datname, extract(epoch from max(clock_timestamp()-xact_start)) from pg_stat_activity
where backend_xmin is not null or backend_xid is not null group by datname order by datname;
```

- удержание горизонта (удерживают на всех базах) физическими слотами репликации, если включена обратная связь (hot_standby_feedback=on

```
select max(age(xmin)) from pg_replication_slots;
select backend_xmin, application_name from pg_stat_replication order by  age(backend_xmin) desc;
```

- в самих репликах искать процессы, выполняющие команды, удерживающие горизонт можно так же, как и на мастере - запросом к `pg_stat_activity`

```
select age(backend_xmin), extract(epoch from (clock_timestamp()-xact_start)) secs, pid, datname database,
state from pg_stat_activity where backend_xmin IS NOT NULL OR backend_xid IS NOT NULL order by
greatest(age(backend_xmin), age(backend_xid)) desc;
```


Мониторинг горизонта (продолжение)

Наблюдать за горизонтом баз данных нужно для поиска причин, по которым он удерживается или не сдвигается долгое время.

Горизонт баз данных кластера в количестве номеров транзакций, отстоящих от текущей:

```
select datname, greatest(max(age(backend_xmin)), max(age(backend_xid))) from
pg_stat_activity where backend_xmin is not null or backend_xid is not null group
by datname order by datname;
```

Длительность самого долгого запроса или транзакции, который и удерживает горизонт:

```
select datname, extract(epoch from max(clock_timestamp()-xact_start)) from
pg_stat_activity where backend_xmin is not null or backend_xid is not null group
by datname order by datname;
```

Удержание горизонта (удерживают на всех базах) физическими слотами репликации, если включена обратная связь (`hot_standby_feedback=on`):

```
select max(age(xmin)) from pg_replication_slots;
select backend_xmin, application_name from pg_stat_replication order by
age(backend_xmin) desc;
```

В самих репликах искать процессы, выполняющие команды, удерживающие горизонт можно так же, как и на мастере - запросом к `pg_stat_activity`:

```
select age(backend_xmin), extract(epoch from (clock_timestamp()-xact_start))
secs, pid, datname database, state from pg_stat_activity where backend_xmin IS
NOT NULL OR backend_xid IS NOT NULL order by greatest(age(backend_xmin),
age(backend_xid)) desc;
```

Параметры `max_slot_wal_keep_size` и `transaction_timeout`

- `max_slot_wal_keep_size` по умолчанию -1 (без ограничений) максимальный размер журнальных файлов, который может оставаться в каталоге pg_wal после выполнения контрольной точки для слотов репликации
- `transaction_timeout` по умолчанию ноль (таймаут отключен). Позволяет отменить любую транзакцию или одиночную команду, длительность которой превышает указанный период времени, а не только простаивающую. Защищает от удержания горизонта базы данных и раздувания файлов (bloat)

Параметры `max_slot_wal_keep_size` и `transaction_timeout`

Чтобы место неограниченно не расходовалось, стоит проверить или установить значения следующих параметров.

`max_slot_wal_keep_size` по умолчанию -1 (без ограничений). Максимальный размер журнальных файлов, который может оставаться в каталоге `pg_wal` после выполнения контрольной точки для слотов репликации. Если слот активирован и клиент не подсоединяется, то журнальные файлы удерживаются. Если этим параметром не установлено ограничение, то файлы журнала заполнят всю файловую систему и **экземпляр аварийно остановится**. Серверный процесс, который не сможет записать в журнал данные, прервётся:

```
LOG:  server process (PID 6543) was terminated by signal 6: Aborted
```

Затем экземпляр попытается рестартовать:

```
LOG:  all server processes terminated; reinitializing
```

Чтобы не допустить нехватки места, стоит установить ограничение. Однако, реплике, которая не сможет получить журналы, а они будут стёрты, придётся получать файлы журналов откуда-то ещё или реплику придётся удалить и создать заново.

`transaction_timeout` по умолчанию ноль, таймаут отключен. Позволяет отменить не только простаивающую, но и любую транзакцию или одиночную команду, длительность которой превышает указанный период времени. Действие параметра распространяется как на явные транзакции (начатые с помощью команды BEGIN), так и на неявно начатые транзакции, соответствующие отдельному оператору. Параметр появился в СУБД Tantor версии 15.4.

Длительные транзакции и одиночные команды удерживают горизонт базы данных. Удержания горизонта базы данных не даёт вычищать старые версии строк и приводит к раздуванию (bloat) файлов объектов.

Параметры `statement_timeout + idle_session_timeout` не защищают от транзакций, состоящих из серии недолгих команд и коротких пауз между ними (например, длинная серия быстрых UPDATE в цикле). Для защиты от долгих команд SELECT может использоваться параметр `old_snapshot_threshold`. Его не стоит устанавливать на физических репликах. В 17 версии `old_snapshot_threshold` убран и `transaction_timeout` позволяет его заменить.

Параметры мастера, которые должны быть синхронизированы с репликами

- Если менять значения этих параметров на мастере, то на репликах значения пяти параметров должны быть не меньше
- Изменения в этих параметрах записываются в журнал
- если реплика обнаружит, что значение на мастере стало больше, то наложение журнальных записей приостановится или экземпляр реплики остановится
- Пять параметров:
- `max_connections, max_prepared_transactions, max_locks_per_transaction` ограничивают максимальное количество блокировок на уровне объектов
- `max_walsenders`
- `max_worker_processes`

Параметры мастера, которые должны быть синхронизированы с репликами

Часть параметров требует внимания. Если менять значения этих параметров на мастере, то на репликах эти значения должны соответствовать значениям на мастере. Так как роли мастер-реплика могут меняться, стоит значения этих параметров делать **одинаковыми** на всех кластерах, чтобы не следить за значениями после смены ролей. Если нужно увеличить значения этих параметров, сначала нужно увеличить на всех репликах, а затем внести изменения на мастере. Если нужно уменьшить значения этих параметров, сначала уменьшить на мастере, а затем уже менять значения на репликах.

Изменения в этих параметрах записываются в WAL. Если в процессе чтения принятых WAL процесс `startup` реплики обнаружит, что значение на мастере стало больше, чем в конфигурации его экземпляра, то если реплика открыта для чтения (параметр `hot_standby=on`), то в лог кластера запишется предупреждение и наложение журнальных записей приостановится. Если реплика не допускает подключений (`hot_standby=off`), то экземпляр реплики остановится и прекратит принимать журнальные записи, что может привести к проблеме при синхронной репликации.

Список параметров:

1) `max_connections, max_prepared_transactions, max_locks_per_transaction` эти параметры ограничивают максимальное количество блокировок объектов

2) `max_walsenders`

3) `max_worker_processes`

https://docs.tantorlabs.ru/tdb/ru/18_1/se/hot-standby.html

Смена ролей мастер-реплика

- запланированная когда мастер работоспособен называют switchover
- если мастер неработоспособен называют failover
- убедиться что мастер остановлен перед тем как дать сигнал одной из реплик стать мастером
- исключить или минимизировать потерю транзакций
- **`pg_last_wal_replay_lsn()`** - журнальная запись, которая была восстановлена последней
- **`pg_last_wal_receive_lsn()`** - последний LSN, полученный репликой

Смена ролей мастер-реплика

В физической репликации у одного из кластеров "роль" мастера (ведущего, основного), у остальных роли резервных серверов (реплик, ведомых). Менять роли можно:

1) когда мастер работоспособен, например, для запланированной остановки экземпляра мастера. Такую смену ролей называют switchover.

2) мастер неработоспособен. Такую смену ролей называют failover.

Перед проведением процедуры нужно:

1) исключить или минимизировать потерю транзакций. Для защиты от потери можно до сбоя настроить синхронную репликацию с подтверждением репликами транзакций и переключиться на ту реплику, которая имеет наибольший принятый и применённый LSN. Если синхронная репликация не использовалась, то стоит найти журнальные файлы мастера. Определить файл журнала, в который писал экземпляр мастера перед повреждением можно по управляющему файлу мастера утилитой `pg_controldata` или другими способами. Этот файл и другие, если они не были переданы на реплику можно скопировать в директорию `PGDATA/pg_wal` реплики и убедиться, что журнальные записи из него применены.

При синхронной репликации настройка может быть такой что транзакции подтверждает одна из реплик. При повреждении мастера может случиться так, что только одна реплика получила последнюю журнальную запись, а другие не получили. Если сделать мастером реплику не получившую последнюю журнальную запись, то могут быть потери транзакций.

Выяснить какая из реплик получила последнюю журнальную запись мастеру можно с помощью функций:

`pg_last_wal_receive_lsn()` - последний полученный LSN на реплике

`pg_last_wal_replay_lsn()` - журнальная запись, которая была восстановлена последней. Если `pg_is_in_recovery()` возвращает true значит это последняя наложенная журнальная запись На мастере функция выдаёт LSN на котором экземпляр мастера открывался после восстановления, а если он был корректно закрыт, то возвращает NULL. Мастером нужно назначать реплику, у которой больший LSN.

2) В каждый момент времени должен быть только один мастер. Если клиентам доступны два мастера и они принимают изменения ("split brain") от клиентов, то разобрать транзакции будет сложно. Чтобы избежать доступности двух мастеров нужно остановить экземпляр мастера перед тем как дать сигнал одной из реплик стать мастером.

Повышение реплики до мастера

- запустить утилиту **`pg_ctl promote`**
- вызвать функцию **`pg_promote()`**
- повышение до мастера увеличивает линию времени на единицу
- при переходе на новую линию времени в директории кластеров `PGDATA/pg_wal` создаётся текстовый файл `0000000N.history`

Повышение реплики до мастера

Для того, чтобы реплика стала мастером нужно ее продвинуть (promote, повысить). Это можно сделать двумя способами:

1) выполнить `pg_ctl promote`

2) вызвать функцию `pg_promote(boolean, integer)`. Первый параметр - нужно ли ждать завершения операции (по умолчанию true), второй параметр максимальное количество секунд ожидания (по умолчанию 60). Возвращает true если операция продвижения успешно выполнена.

Если удалить файл `standby.signal` и перезапустить экземпляр реплики, то перехода на новую линию времени не будет. В этом случае, утилитой `pg_rewind` на практике не удастся воспользоваться и придется пересоздавать бывший мастер. Удаление файла `standby.signal` не создает проблем только, если корректно остановить мастер.

После того, как появится новый мастер, можно будет поменять значения параметра `primary_conninfo` других реплик и бывшего мастера. Создать слоты репликации на новом мастере. Сделать из бывшего мастера реплику, для этого создать файл `standby.signal`.

Если экземпляр бывшего мастера был остановлен корректно и файлы кластера не повреждены, то достаточно запустить экземпляр кластера, не забыв создать файл `standby.signal`.

Если бывший мастер был остановлен некорректно, то его скорее всего захочется восстановить. Восстановить можно, пересоздав кластер: сделав бэкап утилитой `pg_basebackup -R`. Также, можно использовать утилиту `pg_rewind`, если продвижение нового мастера было выполнено с переходом на новую линию времени.

Файлы истории линий времени

- Находятся в директории `PGDATA/pg_wal`
- Имеют название `0000000N.history`
- Удалять эти файлы не стоит
- при повышении реплики до мастера создаётся новая линия времени и новый файл истории линий времени
- используются утилитами и процессами в целях резервирования и восстановления
- запрашиваются и передаются по протоколу репликации наравне с файлами журналов
- резервируются наравне с файлами журналов

Файлы истории линий времени

Каждый раз, когда создаётся новая линия времени, создаётся файл истории линий времени, сохраняющий метки от какой линии времени ответвилась новая линия и когда.

Новая линия времени создаётся при повышении реплики до мастера; при восстановлении из бэкапа на момент времени в прошлом, который можно задать одним из параметров: `recovery_target`, `recovery_target_lsn`, `recovery_target_name`, `recovery_target_time`, `recovery_target_xid`.

Файлы истории нужны, чтобы утилиты и процессы экземпляра могли найти название файла журнала, в котором содержится журнальная запись с нужной линией времени.

Файл истории линий времени это текстовый файл небольшого размера в директории `PGDATA/pg_wal` с названием `0000000N.history`. Можно добавлять в файл истории комментарии о том, как и почему была создана эта конкретная линия времени.

При создании нового файла в него сохраняется содержимое предыдущего файла истории той линии, на основе которой была создана новая линия времени.

Пример содержимого файла `00000003.history`

```
1       116/E30150E8    no recovery target specified
2       116/E30161E8    no recovery target specified
```

Удалять эти файлы не стоит. Пример ошибок связанных с отсутствием файлов:

```
pg_basebackup: could not send replication command "TIMELINE_HISTORY": ERROR:  could not open file "pg_wal/00000002.history": No such file or directory
pg_rewind -D /var/lib/postgresql/tantor-se-18-replica/data1 --source-server='user=postgres port=5432'
pg_rewind: connected to server
pg_rewind: error: could not open file "/var/lib/postgresql/tantor-se-18-replica/data1/pg_wal/00000004.history" for reading: No such file or directory
```

Запуск экземпляра после перехода на новую линию:

```
pg_ctl start -D /var/lib/postgresql/tantor-se-18-replica/data1
...
LOG:  unexpected timeline ID 2 in WAL segment 000000040000000116000000E3, LSN 116/E3016000, offset 90112
LOG:  invalid checkpoint record
PANIC:  could not locate a valid checkpoint record
LOG:  startup process (PID 7638) was terminated by signal 6: Aborted
```

https://docs.tantorlabs.ru/tdb/ru/18_1/se/continuous-archiving.html#BACKUP-TIMELINES

Утилита pg_rewind

- синхронизирует директорию `PGDATA` и табличных пространств с исходным кластером, в том числе заменяет файлы параметров
- Утилита ищет файл `0000000N.history` обоих кластеров с целью найти точку, в которой линии времени двух кластеров разошлись
- Читает журнальные файлы в `PGDATA/pg_wal`, начиная с последней контрольной точки перед моментом, когда линии времени разошлась и до текущего файла журнала того кластера, чью директорию утилита будет синхронизировать
- По журнальным записям определяет все блоки, в которые были внесены изменения
- Копирует эти блоки с другого кластера

Утилита pg_rewind

Утилита `pg_rewind` синхронизирует директорию кластера (`PGDATA` и директории табличных пространств) с директорией другого кластера (мастера или реплики), с которым они разошлись.

Для работы утилиты существенным является наличие линий времени. Утилита ищет файл `0000000N.history` (содержит историю создания линий времени) обоих кластеров с целью найти точку, в которой линии времени двух кластеров разошлись. Затем читает журнальные файлы в `PGDATA/pg_wal`, начиная с последней контрольной точки перед моментом, когда линии времени разошлась и до текущего файла журнала того кластера, чью директорию утилита будет синхронизировать. По журнальным записям определяет все блоки, в которые были внесены изменения. Затем копирует эти блоки с другого кластера.

Дальше утилита копирует все файлы, находящиеся в `PGDATA` (и табличных пространств), включая новые файлы данных, файлы журналов, `pg_xact`, файлы параметров, произвольные файлы.

Директории `pg_dynshmem`, `pg_notify`, `pg_replslot`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp`, `pg_subtrans`, `pgsql_tmp`, файлы `backup_label`, `tablespace_map`, файлы `pg_internal.init`, `postmaster.opts` и `postmaster.pid` не копируются. Утилита создаёт файл **`backup_label`** для перехода к накату журнала, начиная с контрольной точки до точки расхождения и устанавливает в файле `pg_control` `LSN` начала согласованного состояния.

Утилита копирует все файлы параметров, которые находятся в `PGDATA` исходного кластера. Если содержимое файлов параметров синхронизируемого кластера важно, то стоит сохранить эти файлы до запуска утилиты.

Утилита pg_rewind

- используется для того, чтобы вернуть в работу бывший мастер после незапланированной смены ролей
- смена ролей должна быть выполнена с переходом на новую линию времени
- требует включенных `full_page_writes`, подсчета контрольных сумм (или `wal_log_hints`)
- создаёт файл `backup_label` в котором указываются параметры для восстановления согласованности
- копирует все файлы параметров, которые находятся в `PGDATA` исходного кластера. Если содержимое файлов параметров синхронизируемого кластера важно, то стоит сохранить эти файлы до запуска утилиты

Утилита pg_rewind (продолжение)

Обычно, утилита используется для того, чтобы вернуть в работу бывший мастер в результате незапланированной смены ролей (failover). Чтобы полностью не пересоздавать бывший мастер и используется утилита `pg_rewind`.

Важно, чтобы при продвижении реплики была создана новая линия времени. Если этого не произойдёт, то `pg_rewind` будет искать самую последнюю линию времени, а она могла быть создана много времени назад и файлов журналов уже нет.

Если утилита не может записать в какой-либо файл, то прекращает работу. Если работа утилиты не завершилась удачно и повторные попытки запуска не привели к корректному завершению, то директорию синхронизируемого кластера нельзя использовать.

Использование параметров `-R --source-server='адрес'` упрощает конфигурирование: создаётся файл `standby.signal` и в конец `postgresql.auto.conf` добавляется параметр `primary_conninfo` с параметрами подключения.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/app-pgrewind.html

Процессы экземпляра реплики

- **postgres** основной процесс, запускает другие процессы, принимает соединения
- **checkpointer** выполняет точки рестарта
- **background writer** записывает грязные страницы на диск
- startup накатывает журнальные записи из файлов в `PGDATA/pg_wal`
- **walreceiver** принимает поток журнальных записей и записывает их в файлы в `PGDATA/pg_wal`

Процессы экземпляра реплики

На экземпляре реплики присутствуют процессы:
1) **postgres** - основной процесс. прослушивает сокеты, запускает процессы
2) **checkpointer**. Контрольные точки инициируются только на мастере. На реплике при получении журнальной записи о контрольной точке выполняется "точка рестарта". Если в процессе восстановления случится сбой, реплика сможет продолжить с последней точки рестарта
3) **background writer** записывает грязные страницы из буферного кэша на диск
4) **startup** - накатывает журнальные записи
5) **walreceiver**, который принимает журнальные данные от процесса **walsender** мастера
6) Могут присутствовать процессы расширений, например, **stats collector**, а также серверные процессы, обслуживающие сессии созданные с репликой

Повышение реплики до мастера происходит быстро, так как разделяемая память выделена, часть процессов запущена.

Отложенная репликация

- Можно установить задержку для реплики в применении журнальных записей
- по умолчанию задержки нет
- устанавливается параметром `recovery_min_apply_delay`
- Приём журнальных записей репликой (процесс `walreceiver`) выполняется без задержки, задерживается их накат
- параметр устанавливается на уровне экземпляра реплики, при изменении достаточно перечитать файлы параметров

Отложенная репликация

По умолчанию, реплика применяет полученные журнальные записи немедленно и с максимальной скоростью. Параметром `recovery_min_apply_delay` можно установить минимальную задержку, в соответствии с которой сессии реплики должны видеть данные. Параметр устанавливается на реплике и действует только на неё, а не на другие реплики. Задержка вычисляется как разница между меткой времени, записанной в журнальную запись на мастере и текущим временем на реплике. Если время на хосте мастера и реплики не синхронизировано и отличаются, то задержка вычисляется не точно - с учетом этой разницы.

Если реплика была только что создана и файлы реплики ещё не согласованы, то журнальные записи для согласования файлов применяются немедленно. Задержка начинает действовать с момента синхронизации реплики и дальше такой ситуации не возникает, так как файлы реплики остаются синхронизированными.

Приём журнальных записей репликой (процесс `walreceiver`) осуществляется без задержки. Журнальные файлы будут храниться в `PGDATA/pg_wal` реплики до тех пор, пока они не будут применены процессом `startup`. Чем больше задержка, тем больший объём WAL-файлов нужно накапливать и тем больше дискового пространства потребуется для каталога `PGDATA/pg_wal` на реплике.

При использовании обратной связи (параметр `hot_standby_feedback`) мастер не сможет очистить старые версии строк с как минимум на установленную задержку (плюс длительность запросов на реплике).

Если `synchronous_commit=remote_apply` на мастере и реплика единственная подтверждающая транзакции, то транзакции будут подвисать на время задержки.

Задержка применяется для журнальных записей содержащих `COMMIT`, остальные журнальные записи по возможности накатываются без задержки. Однако, журнальные записи не могут накатываться в произвольном порядке из-за зависимостей между транзакциями. Поэтому не стоит считать, что убрав задержку реплика быстро накатит журнальные записи.

Отложенная репликация управляется функциями. `pg_wal_replay_pause()` приостанавливает восстановление. Приостановку используют, если на мастере прошли нежелательные изменения и нужно принять решение что делать - выгрузить данные из реплики или, подкатив журнальные записи до желаемого момента, сделать из реплики мастер.

Восстановление повреждённых блоков данных с реплики

- Расширение **page_repair** содержит функцию **pg_repair_page** для восстановления повреждённых блоков
- за один вызов процедуры восстанавливается один блок
- можно восстановить блоки слоёв **main, vm, fsm**
- в процессе восстановления на отношение устанавливается монопольная блокировка
- блок запрашивается с реплики через соединение
- параметры соединения передаются функции

Восстановление повреждённых блоков данных с реплики

В Tantor Postgres имеется расширение `page_repair`.

```
postgres=# select * from pg_available_extensions where name like '%repair%';
    name     | default_ver| installed_ver|         comment
-------------+------------+--------------+-------------------------
 page_repair | 1.0        |              | Individual page reparing
postgres=# load 'page_repair';
```

При появлении поврежденной страницы данных на мастере, есть возможность забрать образ страницы с реплики, если она не повреждена на этой реплике. В базе данных мастера нужно установить расширение. Пример команды:

```
CREATE EXTENSION page_repair;
```

Расширение содержит две функции:

1) `pg_repair_page(table regclass, block_number bigint, connstr text)` Параметры функции: table имя таблицы, block_number номер поврежденного блока `connstr` - строка соединения к резервному серверу. Пример строки соединения можно взять из параметра конфигурации `primary_conninfo` на любой из реплик. На мастере этот параметр можно установить заранее на случай смены ролей.

2) `pg_repair_page(table regclass, block_number bigint, connstr text, fork text)` `fork` - название форка в котором нужно восстановить блок: `'main', 'fsm', 'vm'`.

Функция `pg_repair_page` запрашивает монопольную блокировку `ACCESS EXCLUSIVE` на объект, в котором будет восстанавливаться блок и ждёт, пока реплика не наложит журнальные записи мастера убрав отставание (лаг) от мастера. Если планируется восстановить несколько страниц, то можно получить блокировку заранее командой `LOCK TABLE`.

https://docs.tantorlabs.ru/tdb/ru/18_1/be/page_repair.html

Демонстрация

- Создание реплики и запуск её экземпляра

Демонстрация

Создание реплики и запуск её экземпляра

Практика

1. Создание реплики
2. Слоты репликации
3. Изменение имени кластера
4. Создание второй реплики
5. Выбор реплики на роль мастера
6. Подготовка к переключению на реплику
7. Переключение на реплику
8. Включение обратной связи
9. Утилита pg_rewind

Практика

Создание реплики
Слоты репликации
Изменение имени кластера
Создание второй реплики
Выбор реплики на роль мастера
Подготовка к переключению на реплику
Переключение на реплику
Включение обратной связи
Утилита pg_rewind

8

8b

Логическая репликация

Логическая репликация

- поддерживает репликацию изменений в строках обычных и секционированных таблиц
- Логическая и физическая репликация могут работать одновременно
- Создаётся два типа объектов: публикации и подписки
- Публикация включает в себя набор таблиц одной базы данных
- Набор таблиц можно менять без остановки репликации
- Реплицируются изменения, а не команды
- Одна и та же таблица быть источником и приемником изменений

Логическая репликация

При репликации происходит захват, передача, применение изменений в строках таблиц. При физической репликации изменения отслеживаются и применяются на физическом уровне - уровне файлов, страниц. При логической репликации отслеживаются изменения на уровне таблиц и их строк, то есть логических объектов. В логической репликации изменения применяются командами SQL, для строк - построчно.

При настройке логической репликации определяются наборы таблиц-"источников", изменения в которых нужно реплицировать. Эти наборы таблиц включаются в объект базы данных "публикация". Можно добавлять или исключать таблицы из "публикации" не пересоздавая её. Публикация - локальный объект базы данных и в неё можно включить только таблицы, находящиеся в её базе данных.

Захватываются не сами команды SQL, которые вносили изменения, а их последствия: по каждой строке, которую затронула команда захватывается идентификатор строки, тип действия со строкой (удалить, вставить, изменить) и значения полей, затронутых командой в этой строке. Такую логику называют "row-based replication". Существуют архитектуры "statement-based replication" (репликация команд), но этот тип репликации не используется для команд, обрабатывающих строки таблиц, так как имеет побочные эффекты.

Логическая и физическая репликация могут работать одновременно.

Логическая репликация использует архитектуру "публикации" (источник) и "подписки" (приёмник). При настройке репликации создаются одноимённые объекты в базах данных.

Логическая репликация развивается, в ней появляются новые возможности.

Новые возможности в 15 версии:
https://docs.tantorlabs.ru/tdb/ru/15_17/se/release-15.html#e-39-3-2-1-логическая-репликация
Новые возможности в 16 версии:
https://docs.tantorlabs.ru/tdb/ru/16_13/se/release-16.html#RELEASE-16-LOGICAL
Новые возможности в 17 версии:
https://docs.tantorlabs.ru/tdb/ru/17_9/se/release-17.html#RELEASE-17-LOGICAL
Новые возможности в 18 версии:
https://docs.tantorlabs.ru/tdb/ru/18_1/se/release-18.html#RELEASE-18-LOGICAL

Применение логической репликации

Используется для:

- переноса данных в архивные или аналитические базы данных
- консолидации данных: переноса данных из региональных баз данных в центральную
- организации резервной базы данных на случай потери основной
- снижения нагрузки на базу данных за счёт переноса части запросов на резервные базы данных

Применение логической репликации

Примеры использования, помимо перечисленных на слайде:
- распространение изменений в таблицах, используемых приложением как хранилища справочных данных из центральной базы в региональные.
- таблицы "событий" (event actions table) - запуск триггеров на вставку строк на стороне подписчика при вставке строки в таблицу в таблице публикации.
- таблица, в которую с какой-то частотой вставляются строки с текущим временем (timestamp) и на базе данных публикации для отслеживания лага и статуса работы приложения на стороне подписчиков (heartbeat table).
- репликация между разными основными версиями, сборками, форками PostgreSQL в целях миграции на них
- распространение данных клиентским приложениям, которые не должны иметь доступ к основной базе данных

https://docs.tantorlabs.ru/tdb/ru/18_1/se/logical-replication.html

Физическая и логическая репликация

Возможности логической репликации:

- можно реплицировать наборы таблиц, а не весь кластер целиком
- нечувствительность к основным версиям, платформам, сборкам
- передаются журнальные записи не всего кластера, а только по реплицируемым таблицам
- есть двунаправленная репликация
- журнальные данные могут передаваться с физической реплики

Физическая и логическая репликация

Преимущества логической репликации по сравнению с физической:
1)можно реплицировать наборы таблиц, а не весь кластер целиком
2)нечувствительность к основным версиям, платформам, сборкам программного обеспечения PostgreSQL физические повреждения не распространяются
3)нет лишнего трафика: передаются журнальные записи не всего кластера, а по таблицам включённым в публикацию
4)структура таблиц подписчика может отличаться от структуры таблиц публикации
5)гибкость: одна таблица может быть включена в несколько разных публикаций и подписок
6)есть двунаправленная репликация

Недостатки по сравнению с физической:
1)реплицируются только результаты выполнения команд `INSERT, UPDATE, DELETE, TRUNCATE` по конкретным таблицам. Не поддерживается репликация других типов объектов (внешних таблиц, представлений и других). Если на таблица, на которой выполняется команда TRUNCATE связана внешним ключом с таблицами, не включёнными в подписку, то на подписчике команда выдаст ошибку и репликация приостановится. Поддержку репликации состояния последовательностей планируется реализовать в 17 версии. Отсутствие "поддержки" означает, что текущее значение последовательности на подписчике (если она там есть) не меняется. Данные в автоинкрементальных столбцах (serial, bigserial) и генерируемых столбцах (`GENERATED .. AS IDENTITY`) реплицируются по значениям.
2)могут возникать "конфликты", что приведёт к приостановке проведения изменений в таблицах подписки
3)требует установить параметр `wal_level` в значение `replica` на уровне всего кластера, где находится публикация. Это приводит к значительному увеличению объема журнальных данных, особенно если многие таблицы имеют свойство `REPLICA IDENTITY FULL` и в таких таблицах часто удаляются или меняются строки
4)не поддерживается репликация `lo`. Это ограничение нельзя обойти, кроме как хранить данные в обычных таблицах, например, в столбцах с типом данных `bytea`.
https://docs.tantorlabs.ru/tdb/ru/18_1/se/logical-replication-restrictions.html

Идентификация строк

- для репликации только вставок строк (INSERT) и TRUNCATE идентификаторы строк не нужны
- для обновления и удаления строк в таблице должен быть:
- первичный ключ
- либо уникальный индекс и ограничения целостности NOT NULL на каждом из столбцов уникального индекса
- либо использовать все столбцы для идентификации строки

Идентификация строк

Логическая репликация реплицирует не текст команд SQL выполняемых над таблицами включенными в публикацию, а изменения в строках таблиц. Для `INSERT` идентификации не нужно и `REPLICA IDENTITY` может иметь любое значение. Для `UPDATE` и `DELETE` (и `MERGE` если хоть одна строка изменится или удалится) нужно идентифицировать строки, в которые будет вноситься изменения. Для идентификации нужно захватить и передать значения в столбцах даже если в самой команде на источнике эти столбцы не упоминались. Иногда это называют захватом значений полей до изменения (before image). Однако, before image - более широкое понятие. Before images могут использоваться для процедур разрешения конфликтов и для этого в before image могли бы включаться не только идентифицирующие строку столбцы, но и любые другие. В текущей версии функционала автоматического разрешения конфликтов нет и before image используются с целью идентификации строки.

Чтобы можно было реплицировать `UPDATE` и `DELETE`, а они реплицируются построчно, в таблицах публикации должен быть настроен "репликационный идентификатор", чтобы идентифицировать строки для изменения или удаления на стороне подписчика.

Самое простое как можно идентифицировать строки в таблицах - использовать значения первичных ключей и это значение по умолчанию.

Вместо первичного ключа (в том числе, составного) в качестве репликационного идентификатора можно назначить любой из уникальных индексов на таблице. Первичный ключ отличается от уникального тем, что в первичном ключе есть ограничение `NOT NULL на всех столбцах ключа`. При использовании уникальных индексов нужно добавить это ограничение на столбцы которые используются в этом ограничении. Использовать уникальные индексы имеет смысл, только если в таблице нет первичного ключа.

Без первичных ключей и уникальных индексов можно реплицировать `UPDATE` и `DELETE`, но тогда репликационным идентификатором придётся назначить все столбцы таблицы. Если в публикацию, в которой реплицируются операции `UPDATE` и `DELETE`, добавляется таблица без задания `REPLICA IDENTITY`, то транзакции с `UPDATE` и `DELETE` на источнике (а не на подписчиках) будут завершены с ошибкой.

Способы идентификации строк

- Если в таблице нет первичного ключа, нужно установить способ идентификации строк командой
  `ALTER TABLE имя REPLICA IDENTITY`
  - `DEFAULT;` использовать первичный ключ
  - `USING INDEX имя;` уникального, не частичного, не отложенного индекса, на всех индексируемых столбцах установить NOT NULL
  - `FULL;` передавать значения всех столбцов
  - `NOTHING;` установлено для таблиц системного каталога, для пользовательских таблиц не имеет смысла

Способы идентификации строк

Команды `INSERT` смогут выполняться без ошибок, им идентификатор не нужен и может быть любым. Одно из возможных значений `REPLICA IDENTITY NOTHING`. В документации описывается как "Записи не содержат информации о старой строке. Records no information about the old row" то есть в терминах "before image" и означает, что значения столбцов помимо указанных в команде не захватываются, но команды `UPDATE` и `DELETE` блокируются на источнике. Пример ошибки на источнике:

```
ALTER TABLE t REPLICA IDENTITY NOTHING;
UPDATE t SET t='b' WHERE id=2;
ERROR:  cannot update table "t" because it does not have a replica identity and publishes updates
HINT:  To enable updating the table, set REPLICA IDENTITY using ALTER TABLE
```

Значение `NOTHING` установлено по умолчанию для таблиц системного каталога (находящиеся в схеме `pg_catalog`).

Не нужно устанавливать `NOTHING` для обычных таблиц, это не даёт никаких преимуществ в том числе для начальной синхронизации, так как при синхронизации и при вставках идентифицировать строки не нужно.

На таблицах подписок требований по индексам нет, там индексы создаются для увеличения производительности.

Список таблиц в базе данных, которые не могут реплицировать `UPDATE` и `DELETE`, пока не будет создан первичный ключ или задан способ идентификации:

```
SELECT relnamespace::regnamespace||'.'||relname "table"
  FROM pg_class
  WHERE relreplident IN ('d','n') -- d первичный ключ, n никакие
    AND relkind IN ('r','p') -- r таблица, p секционированная
    AND oid NOT IN (SELECT indrelid FROM pg_index WHERE indisprimary)
    AND relnamespace <> 'pg_catalog'::regnamespace
    AND relnamespace <> 'information_schema'::regnamespace
ORDER BY 1;
```

Действия для создания логической репликации

- Установить значение параметра `wal_level=logical` на мастере и физической реплике к которой будет подсоединяться подписка (изменение параметра требует рестарт экземпляра)
- выбрать таблицы для репликации
- проверить наличие первичных ключей или выбрать способ идентификации строк
- создать таблицы в базах данных-приёмниках
- создать публикации на базе данных-источнике
- создать подписки на базах данных-приёмниках и выбрать параметры подписки

Действия для создания логической репликации

Для настройки логической репликации нужно:
1)проверить что параметр `wal_level=logical` на мастере и кластере-источнике
2)выбрать таблицы, которые будут включены в публикацию. Таблицы должны иметь первичный ключ. Если первичного ключа нет, то желательно проверить наличие уникального, не частичного, не отложенного индекса и наличие ограничения целостность `NOT NULL` для столбцов этого индекса и дать команду `ALTER TABLE имя REPLICA IDENTITY USING INDEX имя_индекса;`
Если эти условия не выполняются, можно дать команду `ALTER TABLE имя REPLICA IDENTITY FULL;` но в этом случае при изменении или удалении строк в журнал будут записываться старые значения всех полей таблицы, а размер строк может быть большим.
3)на базе данных в этом или другом кластере создать таблицы, которые будут принимать изменения. Скрипт создания таблиц можно получить утилитой `pg_dump` с параметром `--schema-only`. Функционал логической репликации не имеет возможности скопировать определения таблиц.
Команды DDL не реплицируются, определения и набор таблиц не синхронизируются. Изначальный набор таблиц в публикации можно скопировать утилитой `pg_dump` с параметром `--schema-only`. Последующие изменения набора таблиц и их определений нужно будет синхронизировать вручную. Схемы, и таблицы не обязательно должны быть абсолютно идентичными в базах данных публикации и подписчиков. Если определения таблиц в базе данных публикации не меняются, логическая репликация работает надёжно. Если меняется определение таблицы в публикующей базе данных, и команда на подписчике не может применяться, то выдаётся ошибка, репликация по всей подписке приостанавливается и можно провести вручную изменение определения таблицы и репликация восстановится без потерь данных. Во многих случаях можно сначала поменять определения таблиц на подписчике, а потом на публикуемой таблице и репликация не будет приостанавливаться.
4)На базе данных где находятся выбранные для репликации таблицы создать публикацию или публикации командой `CREATE PUBLICATION`. Публикация может включать в себя таблицы только своей базы данных
5)На базе с таблицами в которые будут реплицироваться изменения создать подписку или подписки командой `CREATE SUBSCRIPTION`.

Создание публикации

- выбрать таблицы, которые будут включены в одну публикацию
- Можно указать какие команды будут реплицироваться: вставки, изменение, удаление строк, усечение таблиц
- В публикацию можно включить:
  - все таблицы базы данных
  - все таблицы в схемах
  - список таблиц
- При задании списка таблиц можно указать набор столбцов и фильтр для строк

Создание публикации

После определения таблиц, которые должны быть включены в одну и ту же публикацию, так как эти таблицы одновременно используются в транзакциях или связаны внешними ключами или логически должны иметь согласованные по времени данные (в разных публикациях может быть разный временной лаг) можно дать команды создания публикаций.

Имя публикации должно быть уникальным в пределах своей базы данных. Создание публикации не запускает репликацию. Оно только определяет логику группировки и фильтрации для будущих подписчиков. Все таблицы, добавленные в публикацию, которая публикует операции UPDATE и/или DELETE, должны иметь определенный REPLICA IDENTITY. В противном случае, эти операции будут запрещены для этих таблиц. Для команды MERGE и , публикация будет публиковать INSERT, UPDATE или DELETE для каждой вставленной, обновленной или удаленной строки. Команды COPY публикуются в виде операций INSERT. Для команд MERGE и INSERT.. ON CONFLICT публикация будет публиковать фактическую операцию с каждой строкой.

В команде `CREATE PUBLICATION` можно указать:

1)`FOR ALL TABLES` - реплицирует изменения для всех таблиц в базе данных, включая таблицы, созданные в будущем

2)`FOR TABLES IN SCHEMA` - реплицирует изменения для всех таблиц в указанном списке схем, включая таблицы, созданные в будущем

3)`FOR TABLE` - список таблиц. Если перед именем таблицы указано слово `ONLY`, то в публикацию добавляется только эта таблица. Если слово `ONLY` не указано, то в публикацию добавляется таблица и все ее наследники. После имени таблицы можно указать названия столбцов, тогда реплицироваться будут только значения этих столбцов (и столбцы - идентификаторы строк). По умолчанию реплицируются все столбцы, включая те которые будут добавлены в будущем. Опцией `WHERE` можно задать фильтр чтобы публиковать изменения не во всех строках, а только тех изменений, которые удовлетворяют заданному условию. Список таблиц может быть пуст. Таблицы можно добавить позже командой ALTER PUBLICATION.

4)в опции `WITH ()` можно указать значения для двух опций. В опции publish какие действия со строками будут реплицироваться: `insert, update, delete, truncate`. Для секционированных таблиц есть опция `publish_via_partition_root`

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-createpublication.html

Создание подписки

- Подписка создаётся в базе данных где находятся таблицы-приёмники изменений
- Каждая подписка должна использовать свой слот логической репликации в базе данных-источнике
- Таблицы источника и приёмника и их столбцы сопоставляются по именам и должны быть одинаковыми
- В таблице-приёмнике могут быть дополнительные столбцы. Они могут заполняться значениями по умолчанию или триггерами

Создание подписки

После создания публикации, можно создавать подписки на базах данных с таблицами куда будут реплицироваться изменения. Подписки добавляются командой `CREATE SUBSCRIPTION` и могут быть приостановлены/возобновлены в любой момент командой `ALTER SUBSCRIPTION`, а также удалены командой `DROP SUBSCRIPTION`.

Для каждой подписки создаётся свой логический слот репликации в базе данных публикации. При создании подписки, по умолчанию, выполняется начальная синхронизация, то есть существующие строки в таблицах источника копируются в таблицы подписки, для этого используются дополнительные слоты логической репликации, которые удаляются после того как синхронизация будет выполнена.

Таблицы публикации сопоставляются с таблицами подписчика по именам. Репликация в таблицы с другими именами на стороне подписчика не поддерживается. Столбцы таблиц также сопоставляются по именам. Порядок столбцов в таблице подписчика может отличаться от порядка столбцов в публикации. Также могут не совпадать типы столбцов; достаточно только возможности преобразования текстового представления данных в целевой тип. Таблица-подписчик может содержать дополнительные столбцы, отсутствующие в публикуемой таблице. Такие столбцы будут заполнены значениями по умолчанию, заданными в определении целевой таблицы или триггерами.

Каждая активная подписка получает изменения из своего слота репликации созданного на публикующей стороне. Подписка и слот логической репликации могут управляться отдельно друг от друга. Например, нужно перенести таблицы подписчика другую базу данных (в том же кластере или другом) и активировать подписку там. Сначала командой `ALTER SUBSCRIPTION` разрывается связь подписки со слотом. Потом удаляется подписка, слот остаётся. Потом перегружаются данные в другую базу данных и создаётся подписка с параметром `create_slot=false`, и связывается с существующим слотом.

Также как и физические слоты, логические удерживают файлы журналов. Если слот не планируется использовать, его нужно обязательно удалять как физический, так и логический.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/logical-replication-subscription.html

Свойства подписки

- При создании подписки указывается строка соединения с базой публикации
- В одной подписке можно указать несколько публикаций если они в одной базе
- При создании подписки строки таблиц публикации можно скопировать в таблицы-приёмники
- Логическая репликация работает только через слот
- Слот может быть создан заранее и указан при создании подписки
- Слот может быть создан при создании подписки
- По умолчанию имя слота такое же как имя подписки

Свойства подписки

Команда `CREATE SUBSCRIPTION` создаёт подписку. Имя подписки уникально в пределах базы данных, где она создана. Параметры создания подписки:

1) `CONNECTION` 'строка' подключения к базе данных публикации
2) `PUBLICATION` имена публикаций через запятую
3) `WITH (параметр= значение, ...)`. Имеется больше десятка параметров, они описаны в документации. Основные параметры:

`connect` равно true по умолчанию. Нужно ли подключаться к базе данных публикации. Если поставить false, то параметры `create_slot, enabled, copy_data` тоже будут false

`create_slot = true` по умолчанию. Создавать ли слот логической репликации

`enabled = true` по умолчанию. Запускать ли подписку или оставить неактивной

`copy_data = true` по умолчанию. Будут ли копироваться существующие строки таблиц на которые оформляется подписка. При больших объемах данных копирование в один поток может занять значительное время

`slot_name` по умолчанию такое же как имя подписки. Стоит установить правила именования подписок, чтобы их имена были уникальными во всех кластерах. Если установить значение NONE, то нужно установить `enabled=false` и `create_slot=false`

`binary` по умолчанию false. Параметр позволяет ускорить начальную синхронизацию и репликацию за счёт меньшей совместимости

`streaming = off` по умолчанию, данные начинают передаваться подписке после фиксации транзакции. При значении on данные транзакций начинают передаваться немедленно и записываются на кластере с подпиской во временные файлы, а начинают применяться после фиксации транзакции в публикующей базе. При значении parallel, изменения начинают применяться немедленно фоновым процессом parallel worker. Если свободного процесса нет (их количество ограничивают параметры `max_logical_replication_workers` и `max_worker_processes`), то поведение как у значения on. Если транзакции обрабатывают большие объемы данных установка этих значений позволяет **уменьшить репликационный лаг**, так как изменения начинают передаваться и применяться без задержки. Уменьшение лага, которое можно ожидать 30-50%.

`synchronous_commit = off` по умолчанию. Переопределяет значение одноимённого параметра конфигурации для транзакций которыми применяются изменения в базе подписки. Значение off безопасно для логической репликации, так как если подписчик потеряет транзакции, то они будут повторно переданы.

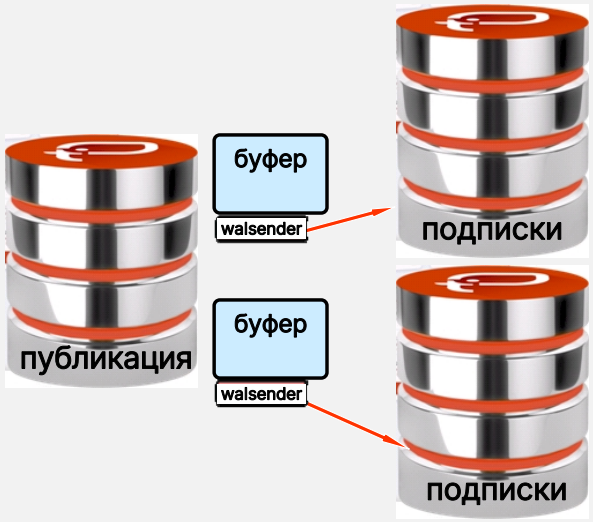
`disable_on_error = false` по умолчанию. Если установить true, то в случае обнаружения ошибки на стороне подписки, подписка переводится в состояние disabled. Если true, то делаются периодические попытки применить изменение, вдруг ошибка исчезнет

`origin = any` по умолчанию, публикация отправляет все изменения. Если используется двунаправленная репликация, то нужно установить `origin=NONE` для предотвращения зацикливания ("ping pong", эхо).

https://docs.tantorlabs.ru/tdb/ru/18_1/se/sql-createsubscription.html

Нагрузка на экземпляр

- Для каждой подписки запускается процесс **walsender**, выполняющий буферизацию и логическое декодирование



Нагрузка на экземпляр

Для каждой подписки на кластере-источнике запускается процесс `walsender`, по одному процессу **для каждой** подписки, которая в свою очередь использует отдельный слот репликации. Использование слота логической репликации обязательно. Их количество ограничивают параметры `max_wal_senders` и `max_replication_slots`. Изменение параметров требует перезапуск экземпляра. Процесс `walsender` читает файлы журнала, но в отличие от физической репликации не просто передаёт журнальные записи , а обрабатывает. Сначала процесс walsender **накапливает** в своей локальной памяти (`reorderbuffer`) изменения, сделанные каждой транзакцией. По умолчанию подписка создаётся с параметром `streaming=off`. Это означает, что должны реплицироваться только зафиксированные транзакции, для этого и используется буфер. Если объем изменений превысит значение `logical_decoding_work_mem` (по умолчанию консервативное значение `64Мб`), то изменения начнут записываются в файлы директории `PGDATA/pg_replslot/имя_слота`. Также, если накопленное количество изменений в одной транзакции превысит `4096`, то изменения этой транзакции тоже начнут записываться в файл. Значение выбрано достаточно большим, чтобы отсечь OLTP транзакции от транзакций с массовыми изменениями строк.

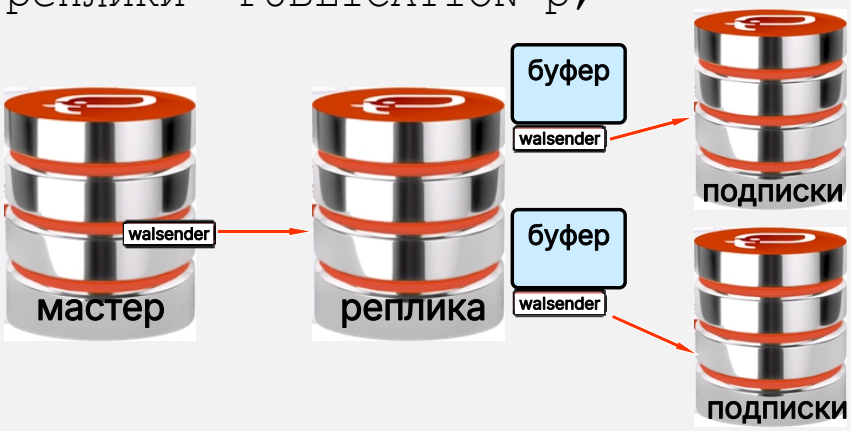
Логику буферизации можно поменять параметром конфигурации `debug_logical_replication_streaming`.

Накопленные в буфере данные по зафиксированным транзакциям (или незафиксированным при значениях `streaming = on или parallel`) передаются модулю вывода `pgoutput`. Модуль это отдельный процесс, а код, который выполняет walsender. На работу этого модуля влияет номер основной версии программного обеспечения подписчика, параметр `binary` подписки (по умолчанию `binary=off` и используется преобразование изменений в транзакции в виде текстовых строк); `streaming` - при значении parallel передаётся дополнительная информация: origin (именно модуль фильтрует транзакции, порождённые процессами логической репликации) и другие параметры, которые задаются в свойствах подписки.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/protocol-logical-replication.html#PROTOCOL-LOGICAL-REPLICATION-PARAMS

Получение журнальных данных с реплики

- При создании подписки достаточно указать адрес физической реплики:
```
CREATE SUBSCRIPTION s CONNECTION 'dbname=имя
host=реплики port=реплики' PUBLICATION p;
```



Получение журнальных данных с реплики

Знание архитектуры логической репликации позволяет понимать трудоёмкость обработки данных на кластере-источнике, оценивать нагрузку на память, процессора (из-за большого количества процессов walsender) и дисковый ввод-вывод (чтение файлов журналов каждым из процессов walsender и запись в файлы директории `PGDATA/pg_replslot`). Нагрузка на хост, где работают процессы walsender, обслуживающие логическую репликацию, может быть существенной.

Если имеются физические реплики, то разумно перенести на сторону физических реплик всю работу, выполняемую процессами walsender. В архитектуре логической репликации PostgreSQL основная обработка изменений выполняется walsender, а не на стороне приёмника.

Для получения данных с физической реплики нужно:

1)в публикации указать в параметре подсоединения адрес реплики

2)реплика должна быть горячей (`hot_standby=on`)

3)включить обратную связь (`hot_standby_feedback=on`), иначе автовакуум может очистить в таблицах системного каталога нужные подписке версии строк, слот перестанет работать, репликация остановится

4)между репликой и мастером нужно использовать физической слот репликации

Если в процессе выполнения команды `CREATE SUBSCRIPTION` долго не возвращается промпт, то можно на мастере выполнить функцию: `select pg_log_standby_snapshot()`. Для создания логического слота репликации нужен моментальный снимок (список всех активных транзакций на мастере). Реплика не имеет доступа к транзакциям на мастере и вынуждена ждать пока процесс `checkpointer` или `bgwriter` на мастере не запишут в журнал снимок. Если после выова функции промпт не возвращается, то значить используется начальная синхронизация строк таблиц (`copy_data = true`) и объем данных большой. Начальная синхронизация выполняется через дополнительно создаваемый слот логической репликации, который будет удалён когда копирование строк завершится.

Что произойдёт в случае, если мастер выйдет из строя и реплику сделают мастером? Логическая репликация продолжит работать без изменений. Слоты репликации (логические и физические) в Tantor Postgres сохраняются после смены ролей.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/logicaldecoding-explanation.html

Конфликты

- Конфликт - если процесс **logical replication worker** не может внести изменения из-за нарушения ограничений целостности или по другой причине
- репликация во всей подписке приостанавливается
- Автоматического разрешения конфликтов нет
- Устранить конфликт можно вручную изменив данные или пропустить (не применять) транзакцию, в которой выполняется команда, приведшая к ошибке
- Информация об ошибке попадает в логе кластера

Конфликты

Для каждой подписки запускается процесс logical replication worker. Этот процесс подсоединяется к процессу walsender по протоколу репликации и принимает поток декодированных модулем вывода изменений. Изменения проводятся командами `INSERT`, `UPDATE`, `DELETE` построчно: используя идентификатор строки `REPLICA IDENTITY`. Если генерируемые команды не могут внести изменения из-за нарушения ограничений целостности или по другой причине (например, срабатывает триггер и генерирует необработанное исключение, нет привилегий выполнить команду), то репликация во всей подписке приостанавливается и возобновится после устранения проблемы если значение параметра подписки `disable_on_error=false`. Возникновение ошибки называется "конфликт".

Если выполняется команда `UPDATE` или `DELETE`, а строка отсутствует (то есть обновлено или удалено ноль строк), то это не ошибка и конфликта нет, команда пропускается и репликация продолжает работать.

Функционала создать правила, по которым конфликт разрешается (автоматическое разрешения конфликтов) нет. Информацию об ошибке можно увидеть в логе кластера. В ошибке указан LSN содержащий COMMIT транзакции, к которой относится изменение, нарушающее ограничение.

Устранить конфликт можно вручную изменив данные или определение объекта: изменив строку с которой возник конфликт убрав ограничение целостности, отключив триггер, дав привилегии. Второй вариант: пропустить (не применять) транзакцию, в которой выполняется команда, приведшая к ошибке. Это делается командой `ALTER SUBSCRIPTION имя  SKIP (lsn = LSN)`. Когда пропускается вся транзакция (чей `LSN` с `COMMIT` указывается в команде), то пропускаются все изменения сделанные транзакцией, в том числе не нарушающие никаких ограничений.

Если параметр подписки `streaming=parallel`, то LSN неудачных транзакций может записываться в лог кластера. В этом случае можно изменить значение на on или off и возобновить репликацию.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/logical-replication-conflicts.html

Двунаправленная репликация

- Если менять значения этих параметров на мастере, то на репликах эти значения должны соответствовать значениям на мастере
- Изменения в этих параметрах записываются в журнал
- если реплика обнаружит, что значение на мастере стало больше, то наложение журнальных записей приостановится или экземпляр реплики остановится
- Список параметров:
- max_connections, max_prepared_transactions, max_locks_per_transaction ограничивают максимальное количество блокировок на уровне объектов
- max_walsenders
- max_worker_processes

Двунаправленная репликация

Двунаправленная репликация - два или более набора таблиц являются источниками изменений и приёмниками друг для друга. Направления репликации настраиваются независимо, но обычно настройки одинаковы. Для двух наборов таблиц создаётся две публикации и две подписки. Для трёх наборов - три и три.

При настройки двунаправленной репликации чем больше лаг, тем больше вероятность конфликтов. Чтобы избегать конфликтов используют горизонтальное или вертикальное "партиционирование". При горизонтальном на уровне приложения за каждым узлом закрепляют строки которые могут меняться или вставляться в таблицу. Например, в зависимости от значения в столбце таблицы. Например, две базы в двух городах. Сессии с базами меняют и вставляют строки относящиеся преимущественно к их городам. Ограничений со стороны базы данных нет и если приложение в одном городе перестанет работать, то клиенты могут быть направлены на приложение в другом городе и оно будет работать с любыми строками. При вертикальном "партиционировании", которое реже используется каждый узел может вносить изменения свой набор столбцов.

Цель двунаправленной репликации не увеличение производительности, а отказоустойчивость.

При использовании последовательностей для генерации значений первичного ключа в таблицах включенных в двунаправленную репликацию для двух узлов настраивают последовательности так, чтобы на одном узле последовательность выдавала чётные числа, на втором нечётные.

При настойке двунаправленной репликации необходимо на всех подписчиках указывать опцию `origin=NONE`.

Локальные команды (в локальных сессиях) имеют `origin= NONE`. Установка значения `NONE` означает, что подписке публикация будет пересылать изменения, которые не имеют origin, то есть внесённые локальными транзакциями, а не logical replication worker. Это позволяет избегать зацикливания в двунаправленной репликации.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/replication-origins.html

Демонстрация

- Однонаправленная репликация
- Двунаправленная репликация

Демонстрация

Однонаправленная репликация
Двунаправленная репликация

Практика

1. Репликация таблицы
2. Репликация без первичного ключа
3. Добавление таблицы в публикацию
4. Двунаправленная репликация

Практика

Репликация таблицы
Репликация без первичного ключа
Добавление таблицы в публикацию
Двунаправленная репликация

9

Обзор Платформы Tantor

Инструменты мониторинга

Универсальные системы мониторинга, неадаптированные для PostgreSQL

Grafana
okmeter
new relic
ZABBIX
MUNIN
cacti
DATADOG

Платформа "Tantor"

Инструменты мониторинга

Универсальные программы могиторинга: Zabbix, Grafana, OKMeter, New Relic, Munin, Cacti, Datadog. Из-за универсальности они не адаптированы для работы с особенностями PostgreSQL. Это означает, что они могут не предоставлять всю интересующую информацию в наиболее удобной форме.

Платформа Tantor специально разработана для мониторнига и управления PostgreSQL, Patroni и XData. Эта система знает, какие метрики наиболее важны для СУБД, как их собирать и как адекватно интерпретировать. В результате, вы получаете детализированный и полноценную картину о состоянии вашей базы данных: от производительности запросов до уровня загрузки дисков и использования памяти.

Использование специализированных программ дает возможность сделать качественный скачок в настройке и мониторинге производительности PostgreSQL. С этим инструментом у вас будет возможность не только быстро определять узкие места и проблемы в вашей системе, но и проводить более точную настройку параметров СУБД для достижения максимальной эффективности.

В заключении, выбор правильной системы мониторинга - это не просто вопрос удобства, это вопрос, который напрямую влияет на эффективность вашего приложения и, как следствие, на ваш бизнес. Поэтому инвестиции в специализированные инструменты часто окупаются в виде повышения производительности и уменьшения времени на устранение неполадок.

Платформа Tantor

- Функциональное ПО с графическим пользовательским интерфейсом, созданное для удобного администрирования экземпляров PostgreSQL и кластеров Patroni
- Необходима организациям, которые используют множество БД, каждая из которых обслуживает определенную информационную систему или сервис
- Российские компании переходят на российские аналоги

Платформа Tantor

Платформа - это функциональное ПО с графическим пользовательским интерфейсом, установленное, как правило в периметре заказчика, создано оно для удобного администрирования кластеров PostgreSQL.

С помощью Платформы "Tantor" можно управлять не только базой данных кластера Tantor, но и любой другой СУБД, основанной на PostgreSQL, включая классическую версию.

Платформа "Tantor" необходима организациям, которые используют множество баз данных, каждая из которых обслуживает определенную информационную систему или сервис. Так как каждая система имеет свои особенности, различные типы нагрузки и данные - это делает базу данных сложным элементом корпоративной информационной системы. Следовательно, на сотрудниках лежит большая ответственность за нормальное функционирование СУБД, а "Платформа Tantor" упрощает их ежедневную работу.

В связи с уходом международных ИТ вендоров с Российского рынка, большое количество Российских компаний, от малого до крупного бизнеса, и всех секторов экономики, переходят на российские аналоги.

Во всех компаниях, где есть ИТ службы и используются СУБД, возникает потребность администрировать большое количество систем управления баз данных.

https://docs.tantorlabs.ru/tp

Возможности Платформы Tantor

- Dashboard (панель): графики для наиболее важных метрик
- Конфигуратор: рекомендации по установке значений параметров PostgreSQL
- Анализатор планов запросов Tensor: встроенный анализатор планов запросов с рекомендациями исправления запросов и визуализацией планов
- Аналитика: анализ диагностических журналов PostgreSQL
- Управление кластерами Patroni: просмотр и изменение параметров, мониторинг репликации и событий failover
- Интеграция с системами
  - уведомлений
  - мониторинга: по REST API (Swagger/OpenAPI)
  - службами каталогов по протоколу LDAP
  - системами резервирования: RuBackup и Backman

Возможности Платформы Tantor

Dashboard (панель): готовые графики для наиболее важных метрик:, таких как соединения с базами данных, транзакции, попадание в буферный кэш, WAL, репликация, контрольные точки, блокировки; CPU, RAM, IO, место в файловых системах, сеть. Группировка и фильтрация метрик по экземплярам и пространствам.

Конфигуратор: рекомендации по установке значений параметров конфигурации PostgreSQL

Внесение изменений в значения параметров: генерация команд ALTER SYSTEM, хранение истории изменений, отмена и применение через агентов Платформы к экземплярам

Встроенный анализатор планов запросов от Tensor: В Платформу встроен лицензированный анализатор планов запросов от компании Tensor с рекомендациями исправления запросов и использования индексов, который не требует доступа в интернет и передачи запросов во внешние системы.

Повседневное обслуживание: планирование запуска VACUUM/VACUUM FULL/REINDEX/ANALYZE

Мониторинг и уведомления: установка пороговых значений (предупреждение, критическое значение, значение восстановления) для метрик, настройка уведомлений о превышении, маршрутизация получателей уведомлений по уровню важности

Интеграция с системами уведомлений: Мессенджеры (Telegram/e-mail), Triafly BI (экспорт для расширенной аналитики), SIEM (security information and event management, всего лишь передача уведомлений службе syslog).

Интеграция с системами мониторинга: по REST API (OpenAPI, тестирование на странице Swagger UI)

Интеграция со службами каталогов по протоколу LDAP: любые, если известны параметры, в документации даны примеры для FreeIPA, AldPro, Active Directory

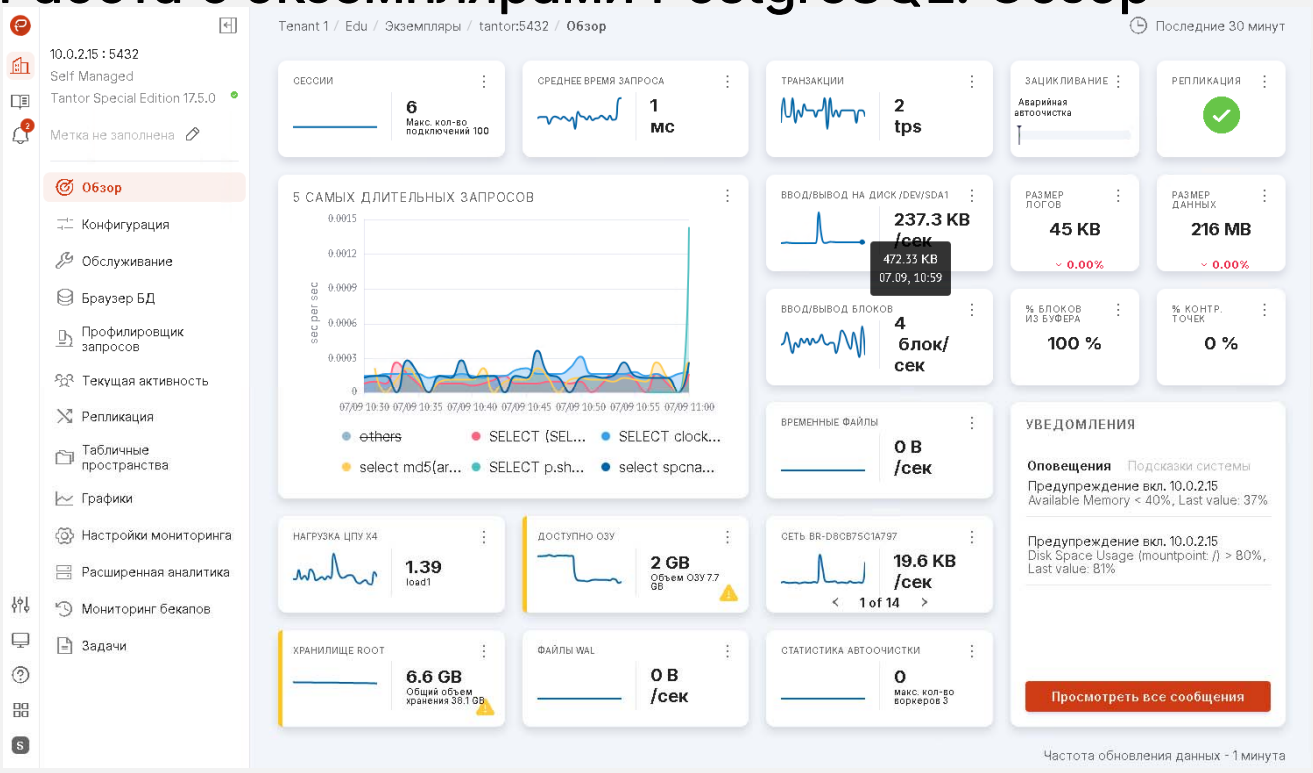
Управление кластерами Patroni: просмотр и изменение параметров, мониторинг репликации и событий failover

Интеграция с системами резервирования: RuBackup и Backman

Аналитика: анализ диагностических журналов PostgreSQL

Управление Tantor xData

Работа с экземплярами PostgreSQL: Обзор



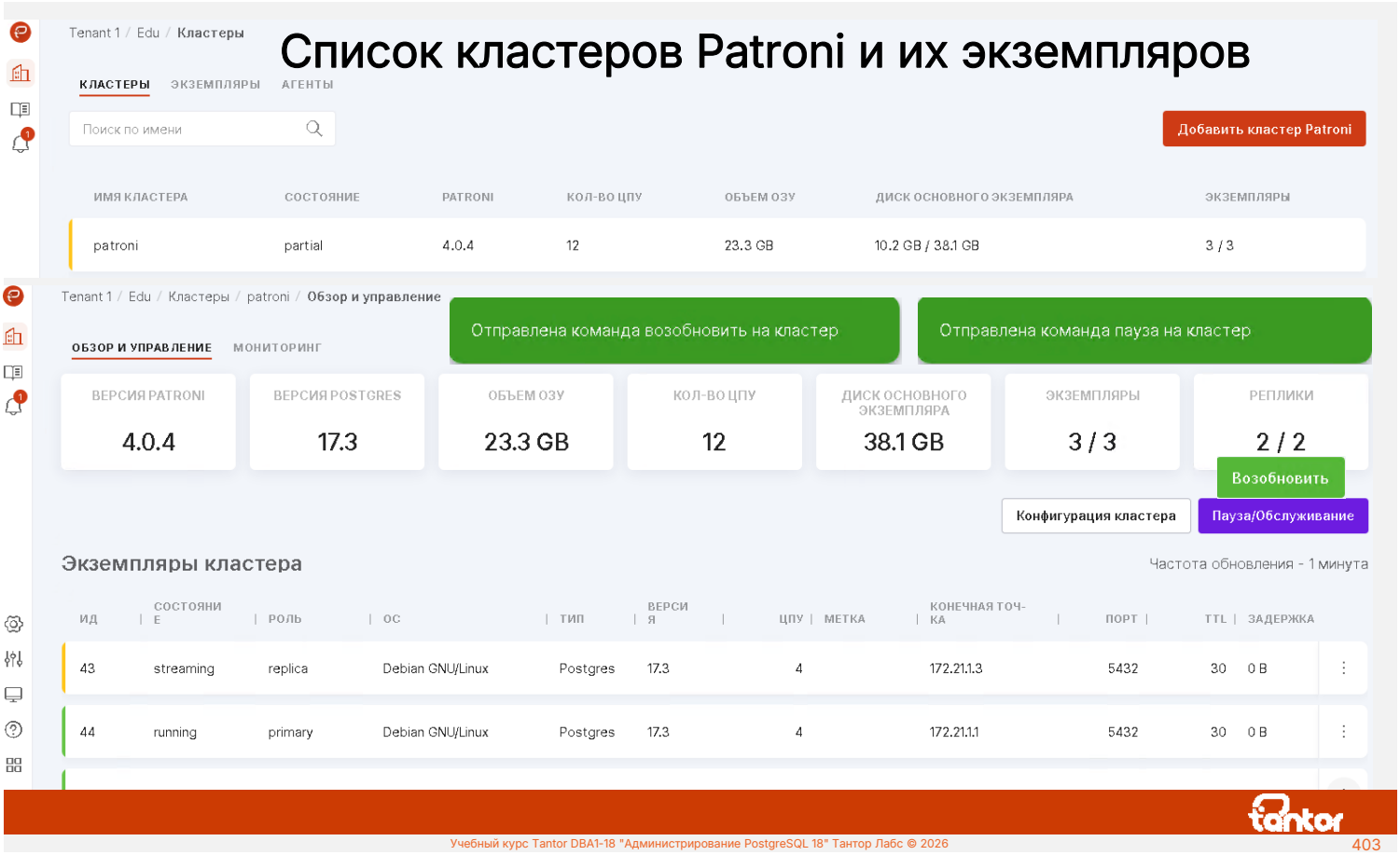
Работа с экземплярами PostgreSQL: Обзор

Страница "Обзор" экземпляра - основная, с которой начинают работу с экземпляром. На ней отображаются "плитки" (прямоугольные элементы графического интерфейса) с основными показателями работы экземпляра PostgreSQL.

В левой части показывается меню из 13 пунктов (**для реплик PostgreSQL пункт "Обслуживание" не показывается**) в 6 версии Платформы. В 5 версии Платформы было 12 пунктов меню, в 6 версии в меню был добавлен пункт: "Мониторинг бэкапов".

Меню закреплено. Если нужно спрятать меню, чтобы освободить пространство на экране, нужно кликнуть на стрелку в прямоугольнике, которая находится сверху справа на плитке меню:

https://docs.tantorlabs.ru/tp/6.1/instances/overview.html



Список кластеров Patroni и их экземпляров

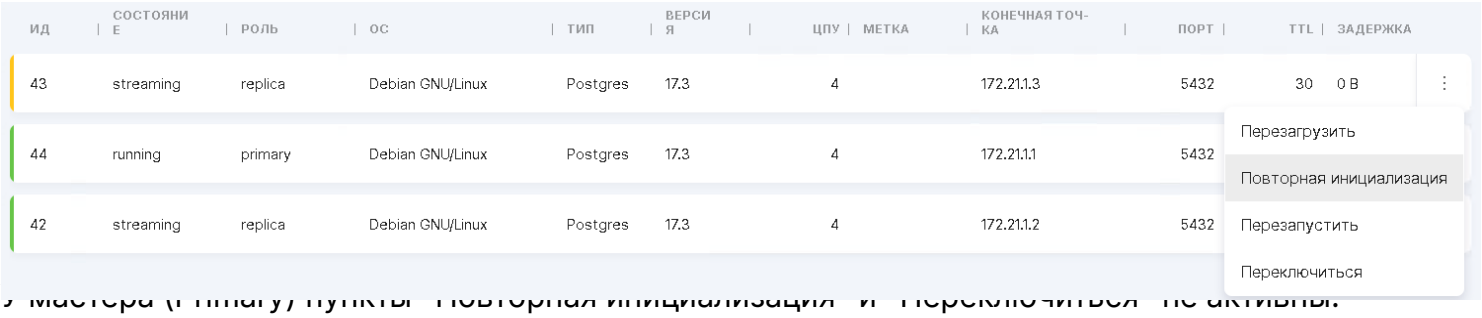
Кластер Patroni это экземпляр PostgreSQL, принимающий изменения (master, primary, основной, ведущий) и несколько физических реплик (standby, резервных). Patroni наблюдает за доступностью экземпляров и, если мастер не доступен, то из одной из реплик делает мастера.

На страницу Cluster Config можно попасть со страницы "Пространства", вкладка КЛАСТЕРЫ. На этой странице кликнуть на строку с именем кластера и в открывшейся странице нажать кнопку "Конфигурация кластера".

Кнопка "Пауза/Обслуживание" приостанавливает работу скрипта Patroni. Скрипт Patroni не будет пытаться что-либо менять в работе экземпляров PostgreSQL.

Кнопка "Возобновить" - возобновит работу скрипта Patroni.

Меню, которое открывается при нажатии на три точки у экземпляра PostgreSQL в списке "Экземпляры кластера":



У мастера (Primary) пункты "Повторная инициализация" и "Переключиться" не активны. Переключиться можно только на реплику. Повторно инициализировать мастер нельзя, это можно только после переключения на реплику.

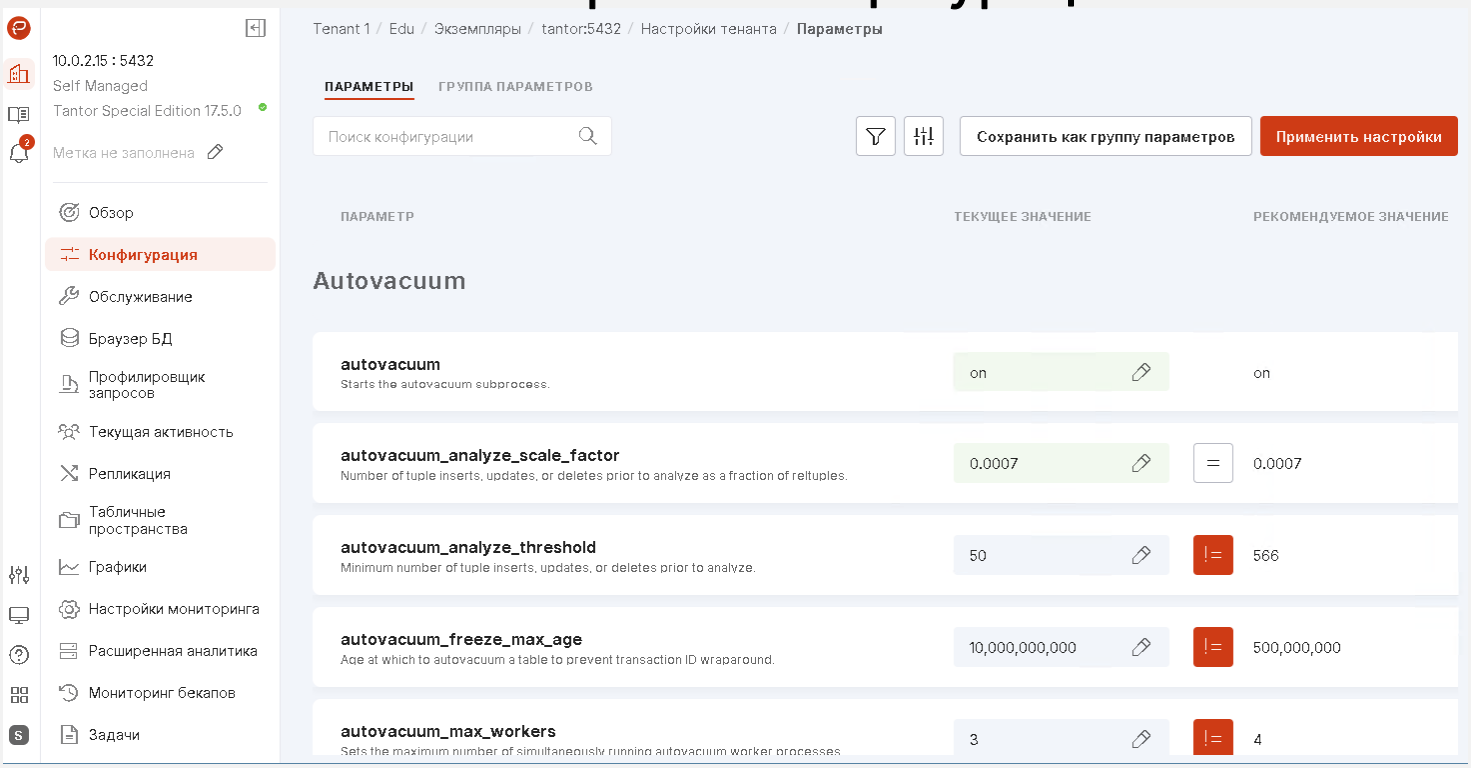
Переключение происходит по правилу: если есть синхронная реплика (synchronous_mode: true), то только её можно сделать мастером. Если синхронной реплики нет, то мастером можно сделать любую доступную реплику.

"Перезагрузить" означает перечитать конфигурацию, без перезапуска экземпляра. "Повторная инициализация" - пересоздание реплики скриптом Patroni.

При нажатии на кнопки "Пауза/Обслуживание" и "Возобновить" подтверждение не запрашивается, сразу появляется зелёнок всплывающее сообщение.

https://docs.tantorlabs.ru/tp/6.1/instances/ug_clusters_pages.html

Работа с экземплярами: Конфигурация



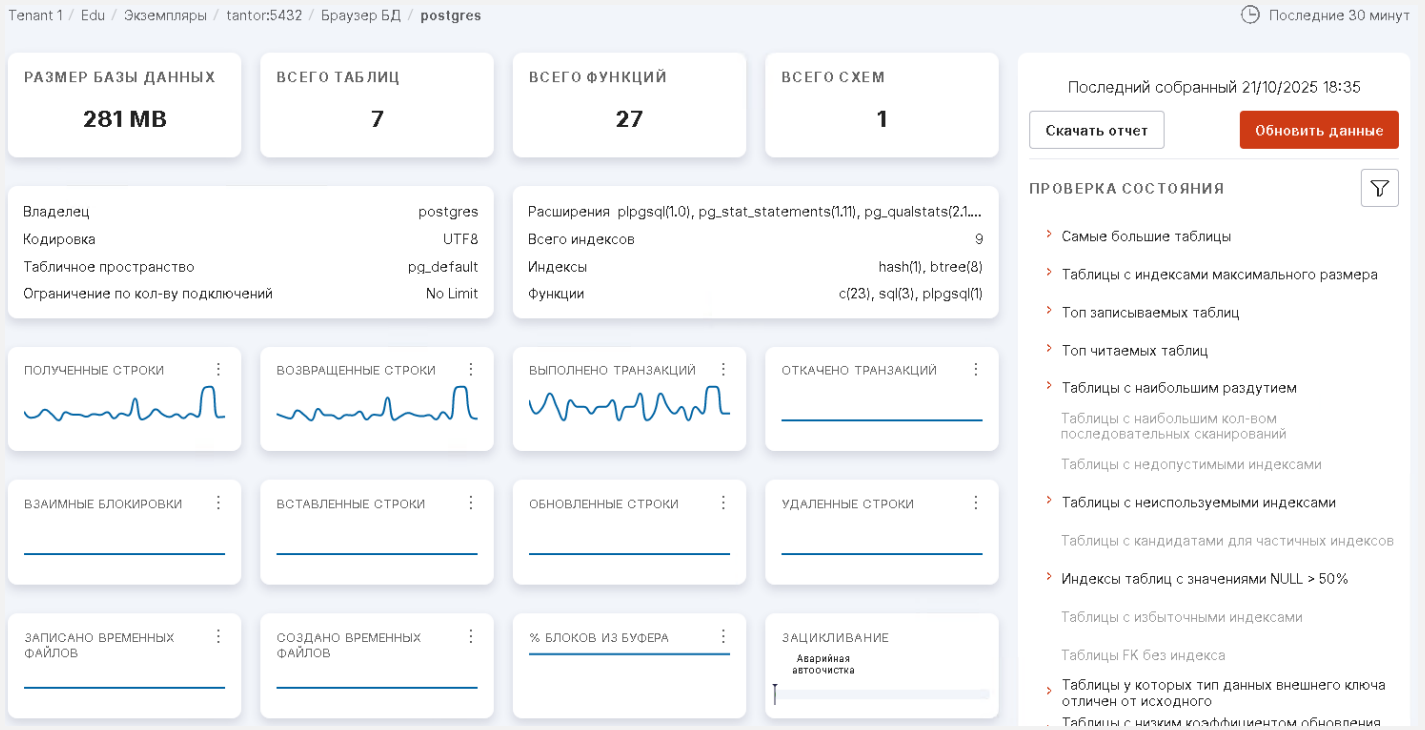
Работа с экземплярами: Конфигурация

На странице "Конфигурация" показываются параметры экземпляра PostgreSQL и даются рекомендации по установке значений параметров. Если текущее значение параметра отличается от рекомендуемого конфигуратором, встроенным в Платформу Tantor, то на это указывает значок "!=".

Можно создать группу параметров, связать её с несколькими экземплярами и применять значения параметров к ним.

https://docs.tantorlabs.ru/tp/6.1/instances/configurations.html

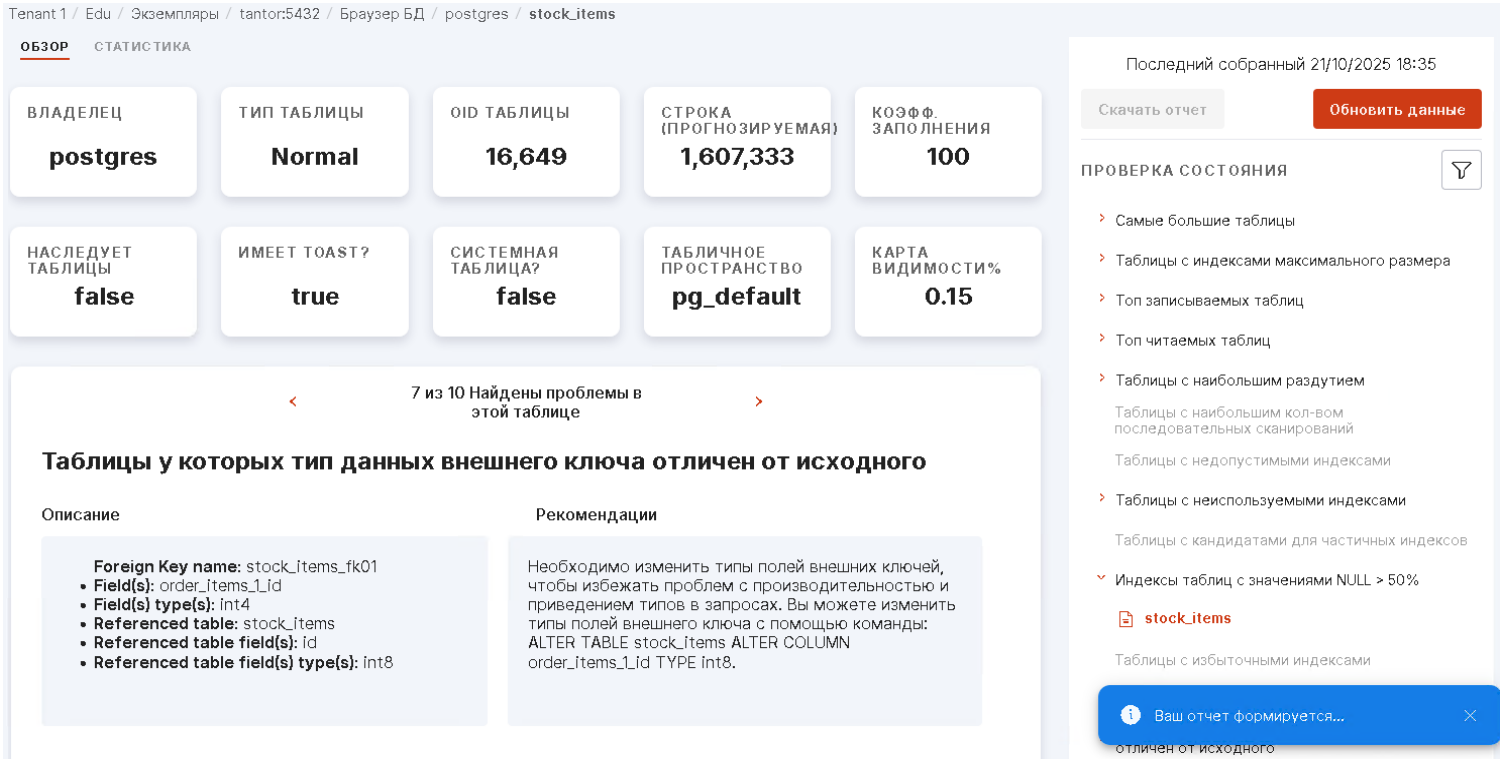
Браузер БД -> Аудит



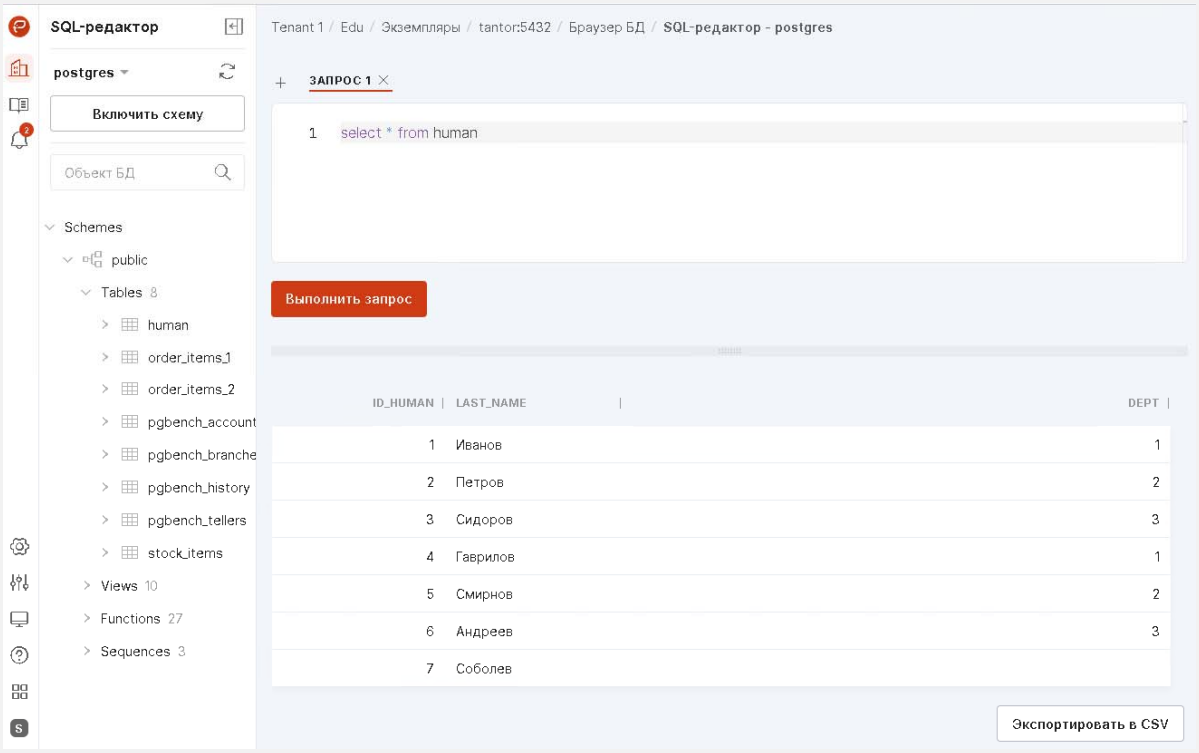
Работа с экземплярами: Браузер БД -> Аудит

На странице "Браузер БД" три кнопки у каждой из баз данных: Аудит, SQL-редактор, Схема данных.

Страница Аудит - одна из наиболее полезных. На этой странице проверяется структура объектов базы данных, выдаются типичные проблемы. Даются рекомендации по устранению проблем. Можно запустить команды Analyze и Vacuum Full.



Браузер БД -> SQL-редактор и Схема

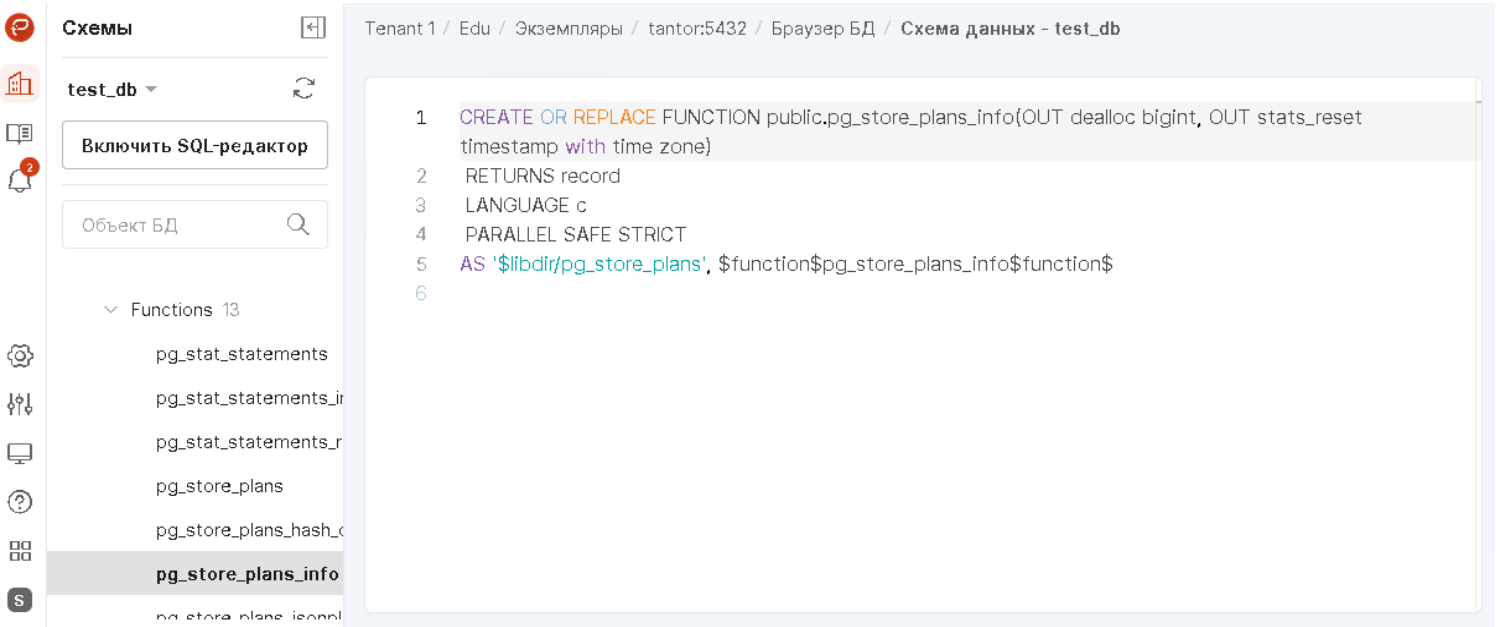


Браузер БД -> SQL-редактор и Схема

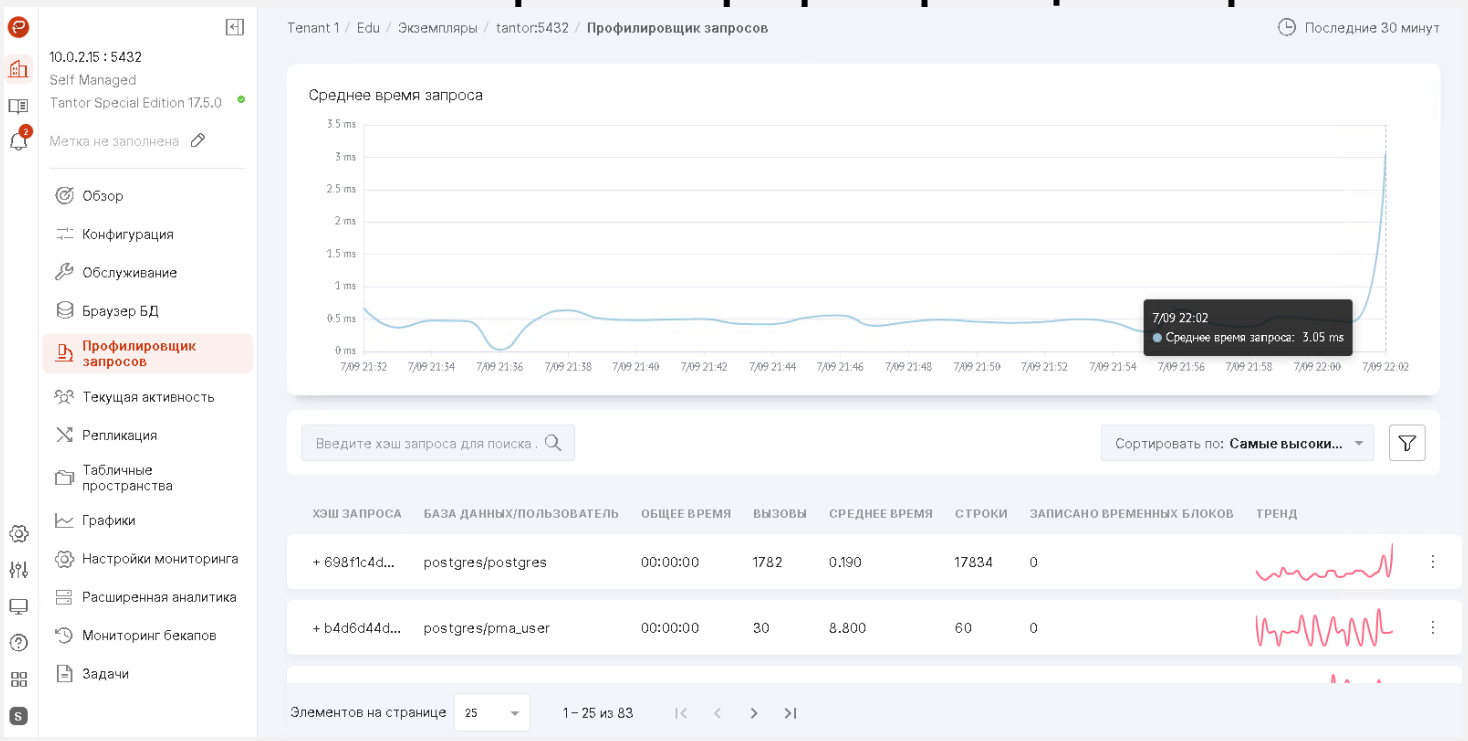
На странице SQL-редактора можно выполнять команды SQL, набирая их в окне браузера. При наборе команд есть автоматические подсказки.

На этих страницах можно переключаться между SQL-редактором и Схемой.

Схема показывает структуру выбранной таблицы, последовательности или команду создания представления, подпрограммы (функции или процедуры).



Работа с экземплярами: Профилировщик запросов



Работа с экземплярами: Профилировщик запросов

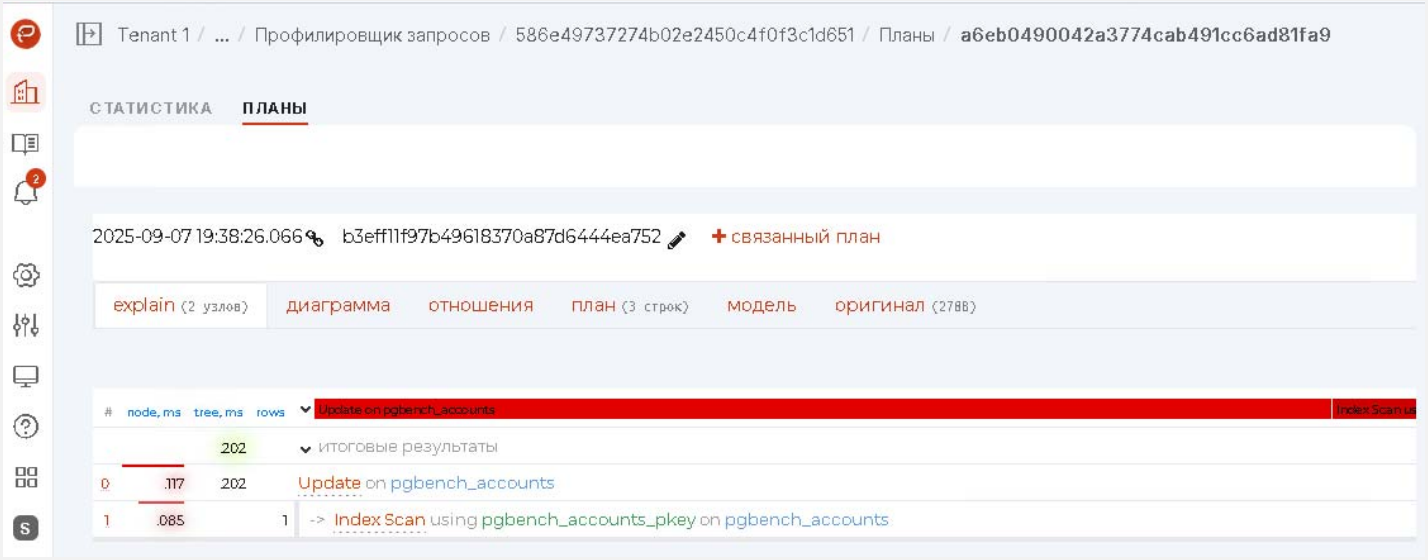
При клике на строку запроса появится страница с деталями выполнения запроса. Под графиком есть 7 точек, кликая на которые, можно переключить график на отображение:

Time Query/Second, Calls/Second, Rows/Second, CPU Time/ Second, IO Time/Second, Dirtied Blocks/Second, Temp Blocks(Write)/Second.

https://docs.tantorlabs.ru/tp/6.1/instances/query_profiler.html

Профилировщик запросов -> Планы

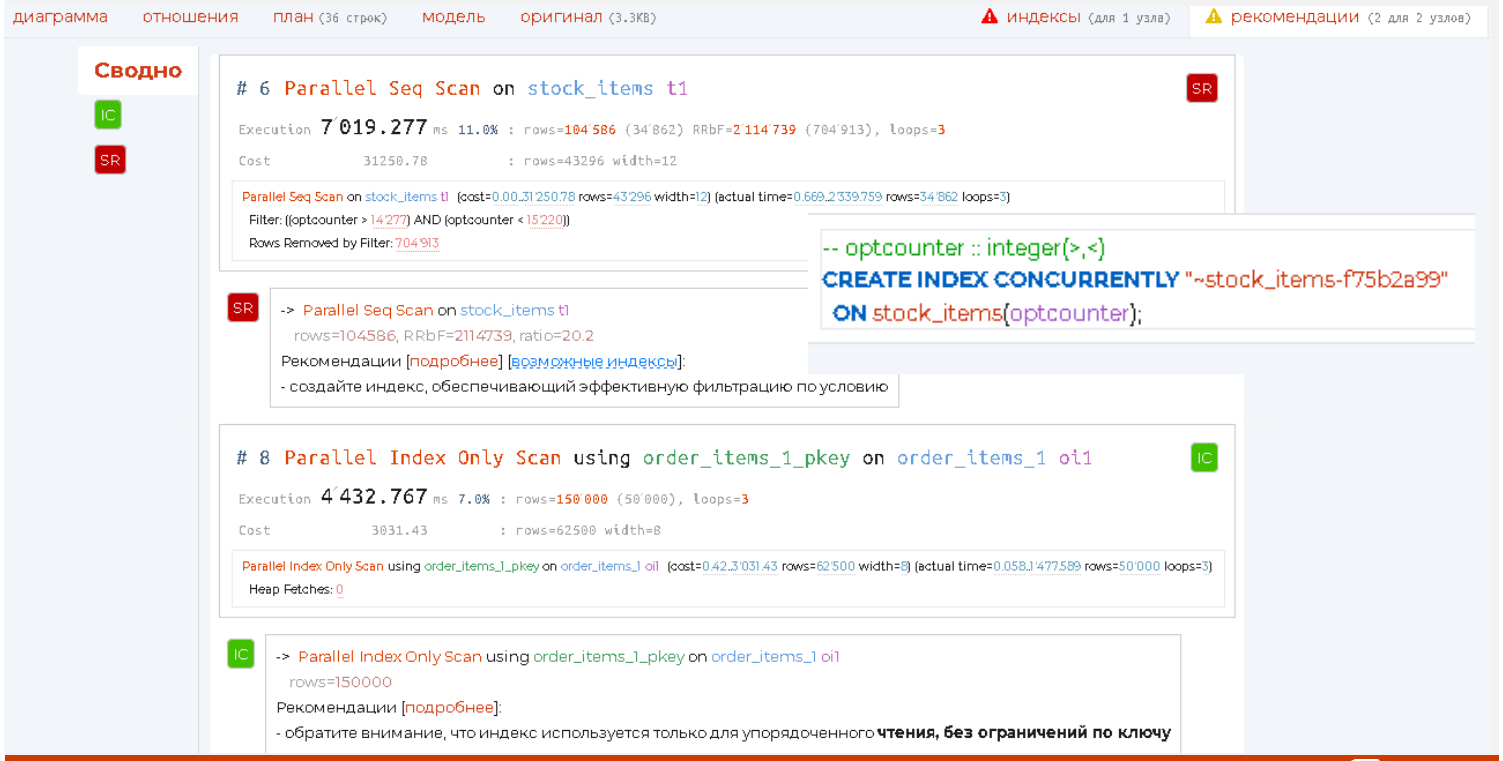
- В Платформу Tantor встроен полнофункциональный анализатор запросов Tensor
- Запросы не передаются на внешние сервера



Профилировщик запросов -> Планы

Профилировщик обеспечивает мониторинг параметров выполнения запросов и их планов в выбранном временном промежутке с использованием скользящего окна. Его цель - проведение анализа и выявление проблемных запросов в базе данных. Профилировщик не только отслеживает параметры выполнения, но и проводит анализ планов запросов, предоставляя подсказки для оптимизации. Это инструмент, который помогает идентифицировать и решать возможные проблемы с производительностью запросов.

Экземпляр: Профилировщик запросов: рекомендации



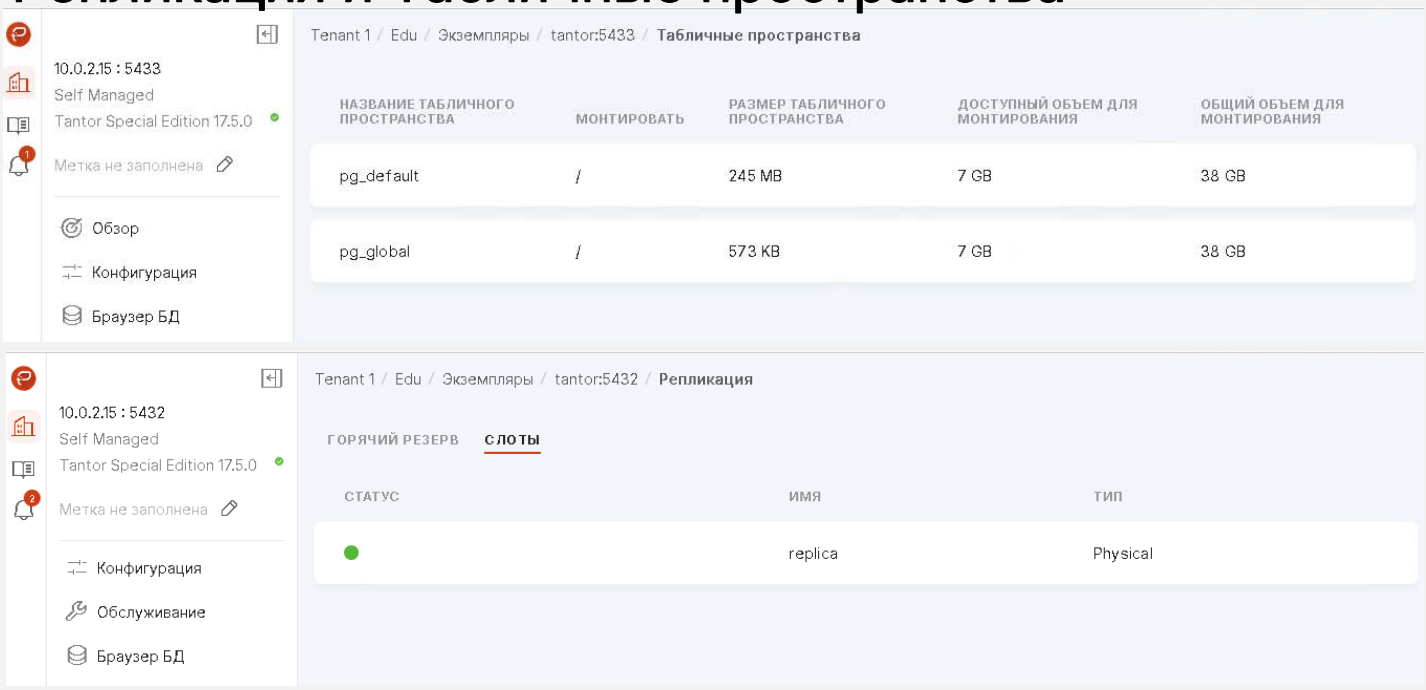
Экземпляр: Профилировщик запросов: рекомендации

Иконки рекомендаций в виде прямоугольников (IC - Index Create, SR - Service Recommendation) присутствуют на плане. При наведении на иконки откроются всплывающие окна с рекомендациями.

Также открыть страницу рекомендаций можно кликнув на ссылки "индексы" и "рекомендации".

В рекомендации по индексам даётся команда создания индекса, которую можно скопировать в буфер обмена.

Репликация и Табличные пространства



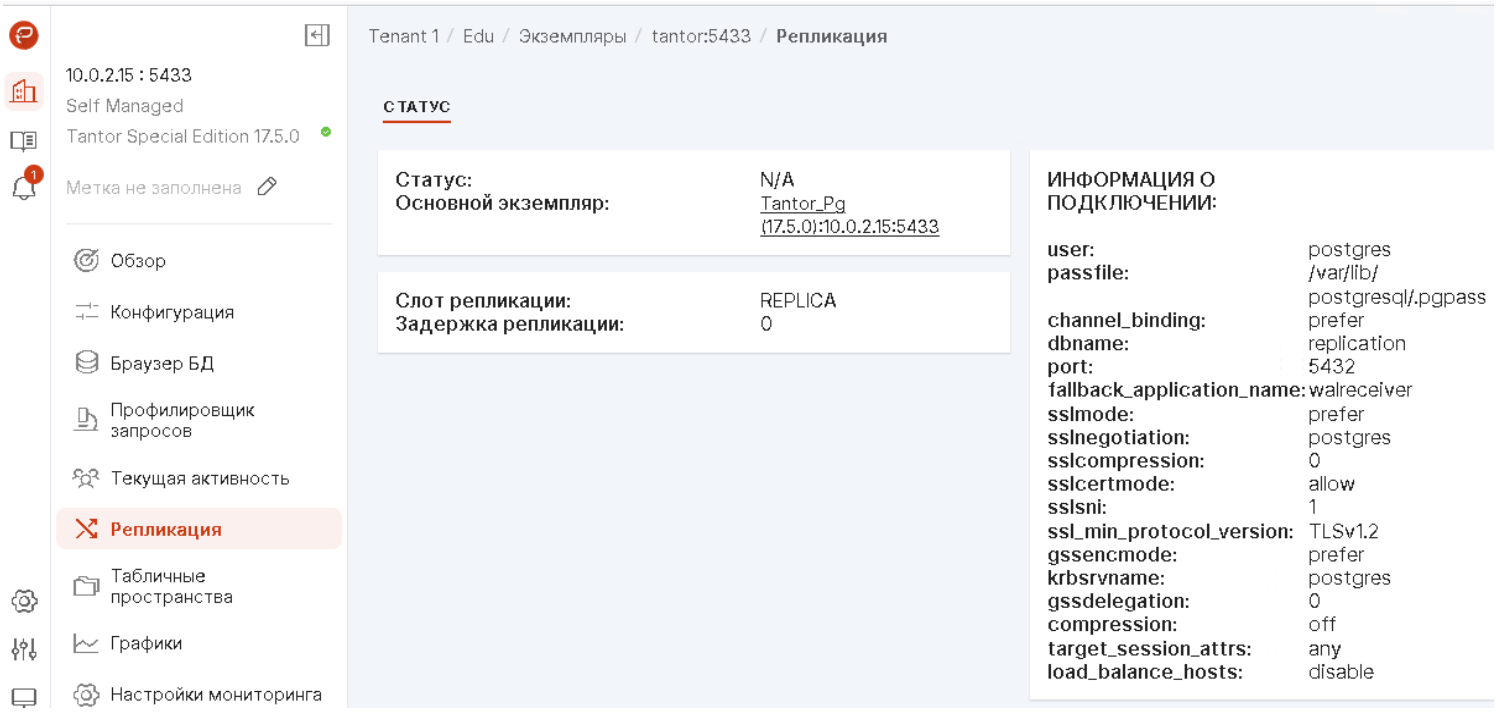
Репликация и Табличные пространства

Физические реплики мониторятся Платформой Tantor и у них есть свои страницы "Работа с экземплярами". Меню "Репликация" у физических реплик отлично от мастера и показывает "СТАТУС" репликации и адрес, откуда реплика получает журналы.

Пример данных "ИНФОРМАЦИИ О ПОДКЛЮЧЕНИИ", при установленных значениях параметров на реплике:

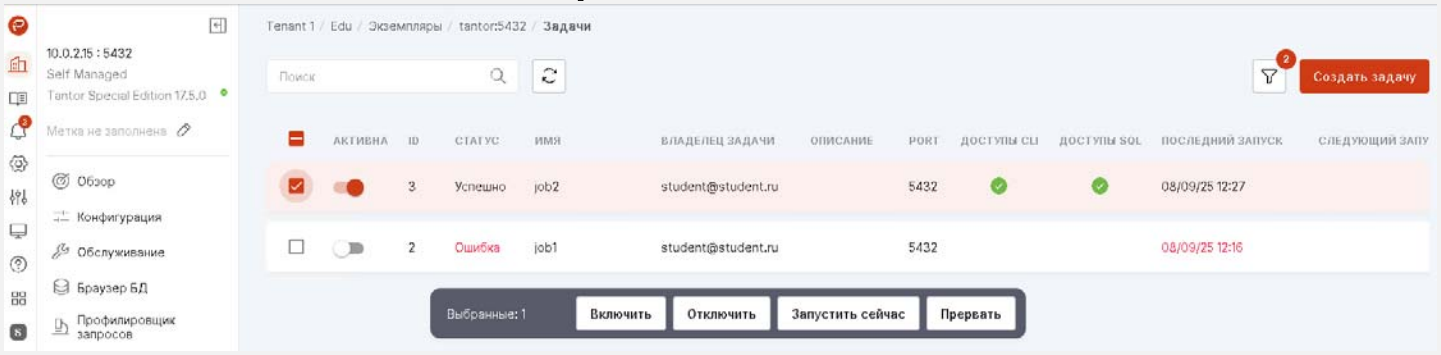
```
primary_conninfo = 'user=postgres port=5432'
primary_slot_name = 'replica'
```

По ссылке из поля "Основной экземпляр" можно перейти на страницу управления экземпляром мастера:



https://docs.tantorlabs.ru/tp/6.1/instances/replication.html

Работа с экземплярами: Задачи

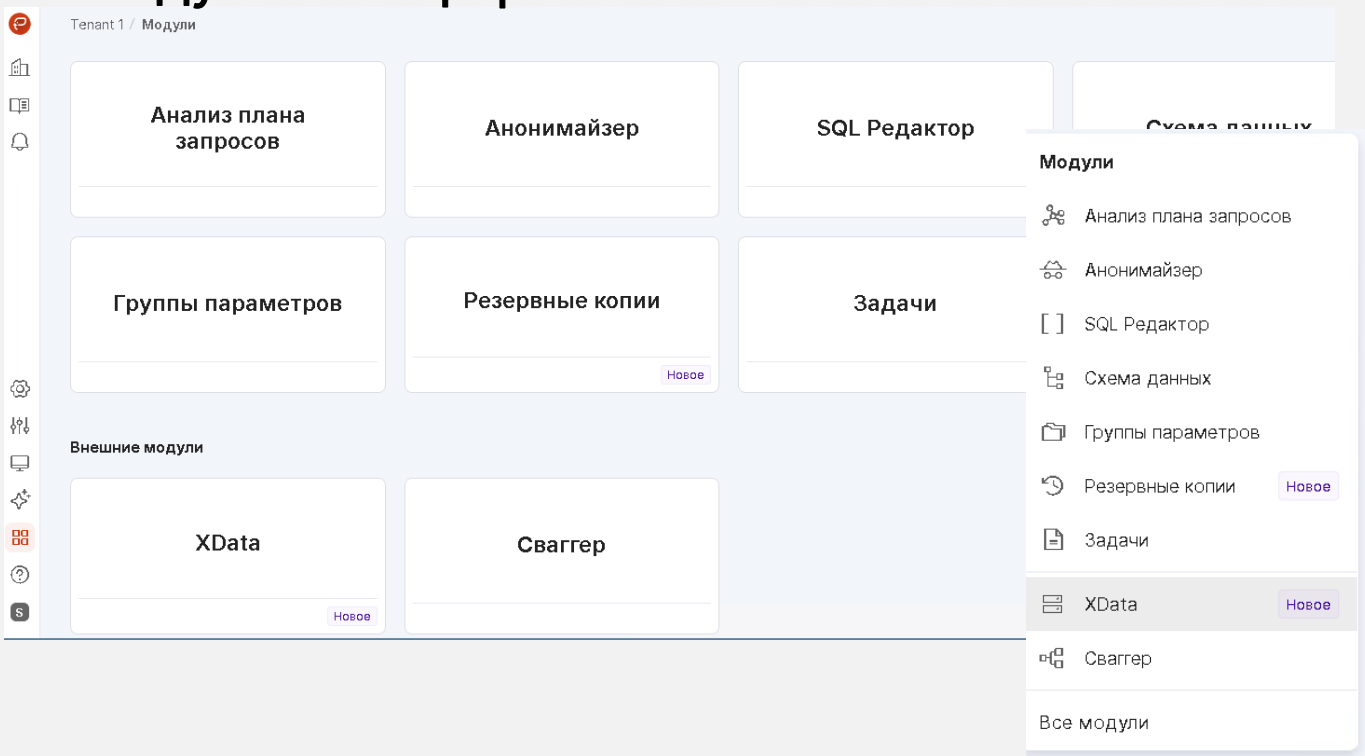


Работа с экземплярами: Задачи

В Платформе можно добавлять задачи, выполняемые по расписанию и указывать имя и пароль пользователя в операционной системе для запуска команд в linux и имя и пароль для запуска SQL-команд в базе данных. После добавления задачи нужно указать действия, которые будут выполняться. Действиями могут быть команды в linux и базе данных. Имя базы данных указывается при добавлении действия.

Добавив действие, можно запустить задачу немедленно.

Модули Платформы Tantor



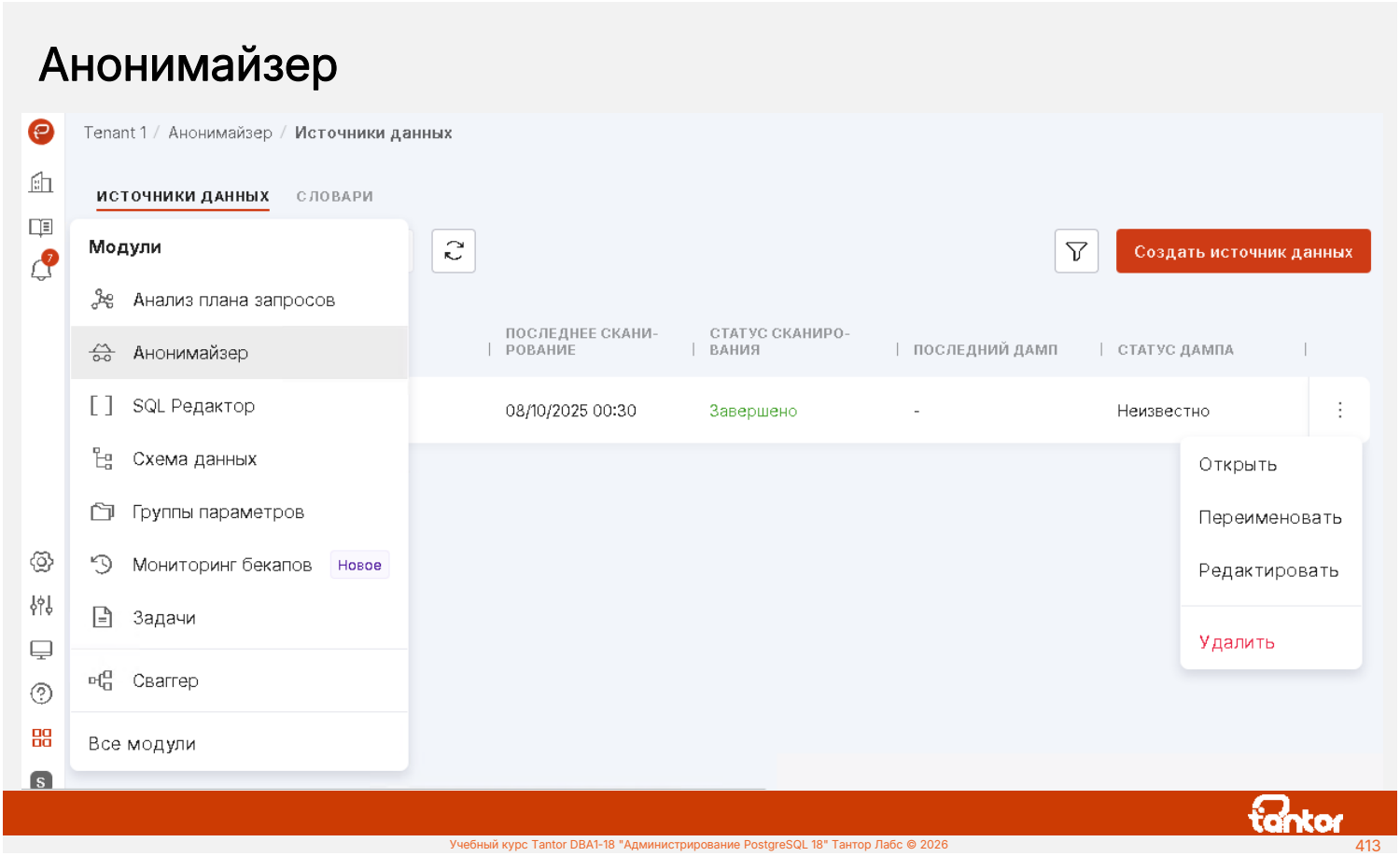
Модули Платформы Tantor

В версии 6.1 Платформы 9 модулей.

"SQL Редактор, Схема данных, Группы параметров, Мониторинг бэкапов, Задачи" могут открываться через страницу экземпляра PostgreSQL.

Модуль "Анализ планов запросов" позволяет вставить план запроса и визуализировать его. План не передаётся на внешние сервера, анализируется программным обеспечением Tensor, встроенным в Платформу Tantor.

"Swagger API" всего лишь открывает в новом окне браузера ссылку https://имя_хоста-Платформы/docs/ в которой высвечиваются страницы Swagger UI, описывающий (документирующий) REST API Платформы. Для доступа к этой странице аутентификация не нужна.



Анонимайзер

Анонимайзер - визуальный интерфейс к утилите pg_anon. Утилита есть как в Платформе Tantor, так и в Tantor Postgres. Анонимизация или маскировка данных - это:

1) поиск столбцов, в которых могут храниться конфиденциальные данные. Например, имена, телефоны, баланс счетов, адреса

2) Замена данных в таких столбцах на похожие, с сохранением связей между таблицами.

3) Выгрузка утилитой pg_dump данных в изменённом виде.

Анонимизация позволяет получить копию (дамп) таблиц, который можно передать подрядчикам для анализа работы программы, работающей с данными. Данные по объему похожи на реальные, но не содержат конфиденциальных данных.

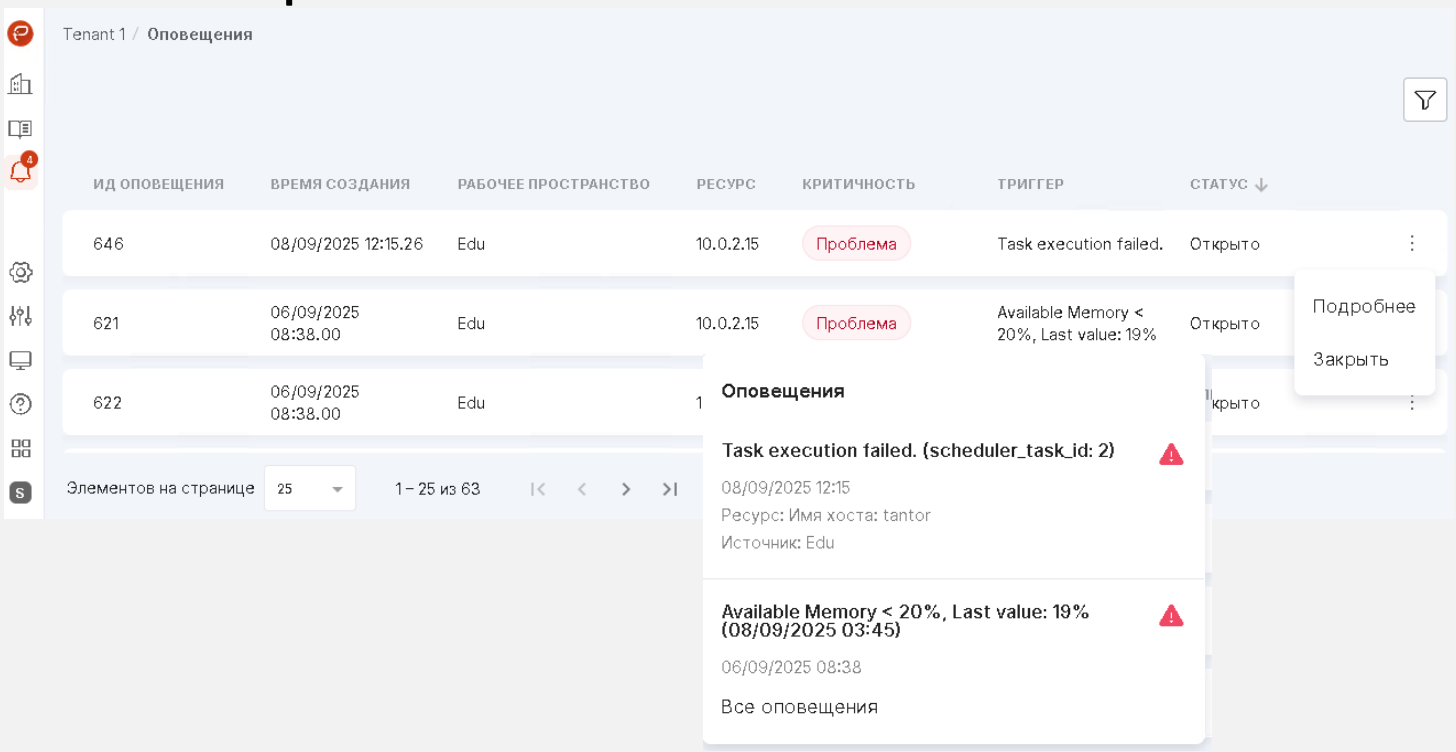
Анонимайзер аналогичен функционалу маскировки данных в Oracle Enterprise Manager Cloud Control.

Ссылка на анонимайзер находится в пункте меню Модули (четыре квадратика слева внизу линейки меню). На странице Анонимайзера две вкладки: ИСТОЧНИКИ ДАННЫХ и СЛОВАРИ.

pg_anon установлен в контейнере pg_anon Платформы. Устанавливать утилиту на хосты баз данных не обязательно.

Можно установить pg_anon и настроить REST-интерфейс по документации:
https://docs.tantorlabs.ru/tp/6.1/admin/pg_anon_stateless.html

Оповещения



Оповещения

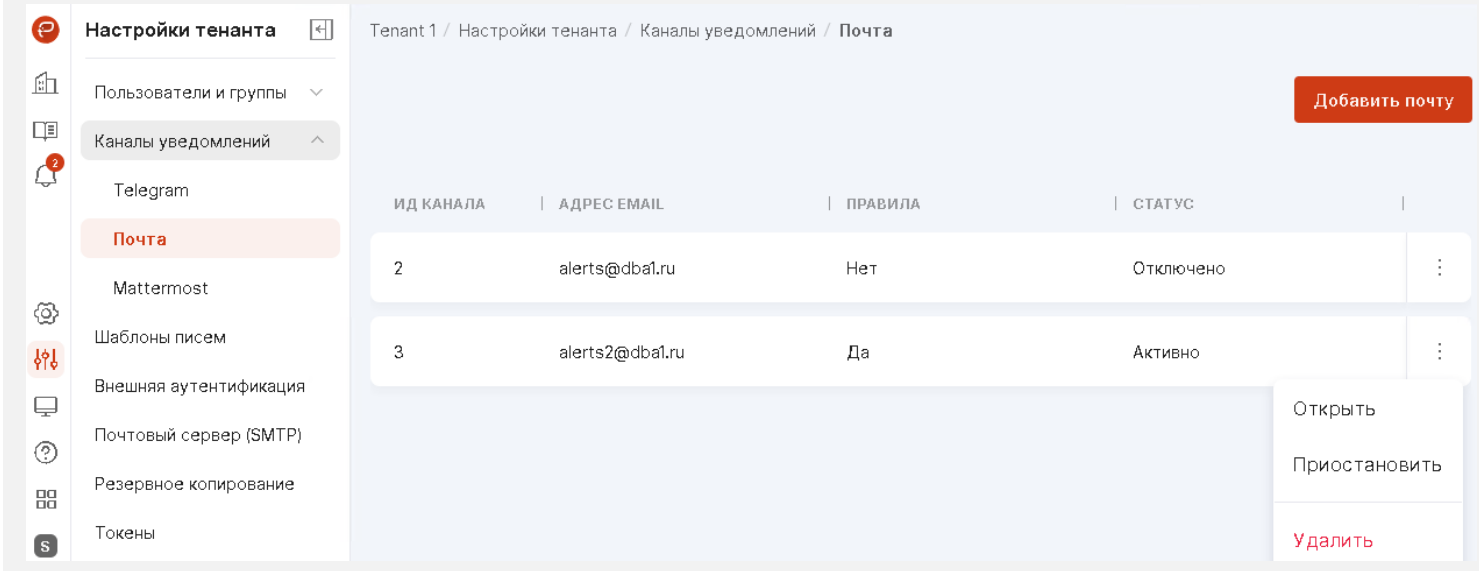
При нажатии на значок колокольчика появляется всплывающее окно с оповещениями (алёртами, уведомлениями). Если в этом окне кликнуть на "Все оповещения", то откроется страница со списком всех оповещений, в том числе "закрытых" (неактивных). Оповещение можно закрыть вручную, оно может закрыться автоматически при снижении значения метрики ниже уровня "восстановления".

Уведомления настраиваются на странице "Настройки тенанта".

https://docs.tantorlabs.ru/tp/6.1/instances/ug_alerts.html

Интеграция со службами сообщений

- уведомления об оповещениях могут передаваться по:
  - e-mail, Telegram, Mattermost



Интеграция со службами сообщений

О появлении оповещений можно посылать уведомления. В Платформе 6 версии поддерживается передача уведомлений:

1. по e-mail
2. в чат Telegram
3. в чат Mattermost.

Это позволяет оперативно получать информацию о происходящих событиях от Платформы.
https://docs.tantorlabs.ru/tp/6.1/admin/admin_reports.html

Курс по Платформе Tantor

- Возможности Платформы Tantor изучаются в учебном курсе PL6: Платформа Tantor 6
  - длительность курса 2 дня
- Темы курса:
  - Возможности Платформы Tantor
  - Архитектура Платформы Тантор (4 части)
  - Мониторинг (8 частей)
  - Конфигурирование и обслуживание PostgreSQL (7 частей)
  - Установка Платформы, Prometheus, Grafana (5 частей)

Курс по Платформе Tantor

Возможности Платформы Tantor изучаются в учебном курсе PL6: Платформа Tantor 6, длительностью 2 дня.

Темы курса:

1. Возможности Платформы Tantor
2. Архитектура Платформы Тантор (4 части)
3. Мониторинг (8 частей)
4. Конфигурирование и обслуживание PostgreSQL (7 частей)
8. Установка Платформы, Prometheus, Grafana (5 частей)

10

Возможности Tantor Postgres

Tantor Postgres - ветвь PostgreSQL

СУБД Tantor Postgres включает в себя:

- все возможности "ванильного" PostgreSQL
- возможности, которые появятся в будущих версиях PostgreSQL
- дополнительные расширения и утилиты
- изменения в ядре PostgreSQL, которые нужны для высоконагруженных СУБД и приложений, формирующих сложные запросы (1C:ERP)
- собственные доработки

Tantor Postgres - ветвь PostgreSQL

СУБД Tantor Postgres является ответвлением (fork, "форк") PostgreSQL и:
1) включает в себя все возможности "ванильного" (основная ветвь) PostgreSQL
2) включает возможности, которые появятся в будущих версиях PostgreSQL. Процесс принятия (коммита) изменений, добавляющих функционал (патчей) в основную ветвь PostgreSQL долгий и может занимать несколько лет. Изменения, которые полезны, не имеют недостатков добавляются в Tantor Postgres до появления в основной ветви PostgreSQL. Пример: расширение pg_uuidv7 появится в 18 версии PostgreSQL появилось в Tantor Postgres 16.8 версии; параметры, устанавливающие размеры SLRU буферов (transaction_buffers, subtransaction_buffers и др.), таймауты (transaction_timeout), появившиеся в основной ветке 17 версии были добавлены в Tantor Postgres 15 версии; расширенное использование инструкций SIMD процессоров, которое появится в 18 версии PostgreSQL появилось в 17 версии Tantor Postgres, а начало внедряться с 15 версии Tantor Postgres.
3) дополнительные расширения. В стандартно поставляемые (contrib) расширения добавляются расширения, которые легко переносить (rebase) на новые основные версии PostgreSQL: не имеющие компилируемого кода, с не очень большим объемом кода, имеющие не много взаимодействий с основным кодом или востребованный функционал. Много полезных расширений и утилит не включаются в основную ветвь, но добавлены в Tantor Postgres. Например, pg_hint_plan (подсказки оптимизатору), pg_columnar (клоночное хранение), pg_ivm (обновляемые материализованные представления), pg_background (использование фоновых процессов), утилиты pgcopydb, pgcompacttable, pg_repack
4) изменения в ядре PostgreSQL, которые нужны для высоконагруженных СУБД и сложные настолько, что добавление в основную ветвь откладывается много лет: 64-битный счетчик транзакций, автономные транзакции, доработки для совместимости с 1C:ERP и другими программами, формирующими сложные запросы.
5) собственные доработки кода PostgreSQL, расширения, утилиты. Доработки предлагаются в виде патчей сообществу, оформляются в виде проектов по свободным лицензиям (https://github.com/TantorLabs) авторами патчей. Авторами указываются разработчики патчей, сотрудники Тантор Лабс, которые их создали.
https://docs.tantorlabs.ru/tdb/ru/18_1/se/differences.html

Доработки в Tantor Postgres

- Доработки позволяют улучшить производительность и отказоустойчивость при промышленной эксплуатации
- Доработки вносятся так, чтобы Tantor Postgres минимально отличался от основной ветви PostgreSQL
  - выбирается реализация, которая имеет наибольшую вероятность появления в основной ветви
  - наименее меняющая код PostgreSQL и его настройки по умолчанию
- Администрируя Tantor Postgres можно применять опыт администрирования PostgreSQL
- Тантор Лабс избегает доработок, которые могли бы привязать приложения ("vendor lock") и затруднить работу приложений в ванильном PostgreSQL

Доработки в Tantor Postgres

Доработки позволяют улучшить производительность и отказоустойчивость при промышленной эксплуатации.

Доработки вносятся так, чтобы Tantor Postgres минимально отличался от основной ветви PostgreSQL: выбирается реализация, которая имеет наибольшую вероятность появления в основной ветви или наименее меняющая код PostgreSQL и его настройки по умолчанию. Например, не используются обёртки pg_controlcluster, изменения делаются отключаемыми (параметры конфигурации enable_group_by_reordering, enable_temp_memory_catalog и другие). Tantor Postgres стремится быть совместимым с PostgreSQL и не отличаться с точки зрения эксплуатации.

Тантор Лабс избегает доработок, которые могли бы привязать приложения ("vendor lock") и затруднить работу приложений в ванильном PostgreSQL.

Администрируя Tantor Postgres можно применять опыт администрирования PostgreSQL. Опыт администрирования Tantor Postgres пригодится для работы с PostgreSQL, в том числе будущих версий.

Дополнительные параметры конфигурации

- В Tantor Postgres 17.5 есть 15 дополнительных параметров, влияющих на создание и выбор планов запросов и функционал:

```
postgres=# \dconfig enable_*
               Parameter               | Value
---------------------------------------+------
enable_convert_exists_as_lateral_join | on
enable_convert_in_values_to_any       | on
enable_group_by_reordering            | on
enable_index_path_selectivity         | on
enable_join_pushdown                  | on
enable_self_join_removal              | on
```

```
backtrace_on_internal_error           | off
enable_delayed_temp_file              | off
enable_large_allocations              | off
enable_temp_memory_catalog            | off
libpq_compression                     | off
wal_sender_stop_when_crc_failed       | off
pg_stat_statements.sample_rate        | 1
pg_stat_statements.mask_const_arrays  | off
pg_stat_statements.mask_temp_tables   | off
```

- Параметры, появившиеся в PostgreSQL 17 версии:

```
allow_alter_system, commit_timestamp_buffers, huge_pages_status, io_combine_limit,
max_notify_queue_pages, ultixact_member_buffers, multixact_offset_buffers, notify_buffers,
restrict_nonsystem_relation_kind, serializable_buffers, subtransaction_buffers,
summarize_wal, sync_replication_slots, synchronized_standby_slots,
trace_connection_negotiation, transaction_buffers, transaction_timeout, wal_summary_keep_time
```


Дополнительные параметры конфигурации

Часть улучшений в ядре Tantor Postgres SE и SE 1C сделана отключаемой параметрами.

Параметры Tantor Postgres SE и SE 1C, влияющие на создание и выбор планов выполнения запросов:

```
postgres=# \dconfig enable_*
               Parameter               | Value
---------------------------------------+-------
 enable_convert_exists_as_lateral_join | on
 enable_convert_in_values_to_any       | on
 enable_group_by_reordering            | on
 enable_index_path_selectivity         | on
 enable_join_pushdown                  | on
 enable_self_join_removal              | on
```

Параметры Tantor Postgres SE и SE 1C, влияющие на функционал:

```
 backtrace_on_internal_error           | off
 enable_delayed_temp_file              | off
 enable_large_allocations              | off
 enable_temp_memory_catalog            | off
 libpq_compression                     | off
 wal_sender_stop_when_crc_failed       | off
 pg_stat_statements.sample_rate        | 1
 pg_stat_statements.mask_const_arrays  | off
 pg_stat_statements.mask_temp_tables   | off
```

Синим цветом выделены параметры, добавленные в версии 17.5, зелёным в 16 версии.

Параметры, появившиеся в 17 версии PostgreSQL:

```
allow_alter_system, commit_timestamp_buffers, huge_pages_status, io_combine_limit,
max_notify_queue_pages, ultixact_member_buffers, multixact_offset_buffers,
notify_buffers, restrict_nonsystem_relation_kind, serializable_buffers,
subtransaction_buffers, summarize_wal, sync_replication_slots,
synchronized_standby_slots, trace_connection_negotiation, transaction_buffers,
transaction_timeout, wal_summary_keep_time.
```

Параметры, удаленные в 17 версии: `db_user_namespace, old_snapshot_threshold, trace_recovery_messages`

Расширения Tantor Postgres SE и SE 1C

- Ядра Tantor Postgres SE и SE 1C унифицированы
  - все возможности и расширения SE 1C есть в SE
- Tantor Postgres SE и SE 1C включают:
  - расширения `credcheck`, `cube`, `fasttrun`, `fulleq`, `hypopg`, `mchar`, `page_repair`, `pg_cron`, `pg_hint_plan`, `pg_repack`, `pg_stat_kcache`, `pg_store_plans`, `pg_trace`, `pg_uuidv7`, `pg_wait_sampling`, `pgaudit`, `pgaudittofile`, `transp_anon`
  - библиотеки `dbcopies_decoding`, `oauth_base_validator`, `online_analyze`, `pg_query_id`, `pg_stat_advisor`, `plantuner`, `wal2json`
  - утилиты `pgcompacttable`, `pgcopydb`, `pg_diag`, `pg_repack`
- в отдельных пакетах поставляются программы: `pg_anon`, `wal-g`, `pg_configurator`, `pg_cluster`, `pg_diag_setup`, `pg_sec_check`
- Tantor Postgres SE дополнительно включает расширения:
  - `http`, `orafce`, `pgl_ddl_deploy`, `pgq`, `vector`, `pg_archive`, `pg_columnar`, `pg_ivm`, `pg_partman`, `pg_qualstats`, `pg_tde`, `pg_throttle`, `pg_variables`, `pg_background`

Расширения Tantor Postgres SE и SE 1C

Ядра Tantor Postgres SE и SE 1C унифицированы. Все возможности и расширения сборки Tantor Postgres SE 1C есть в сборке Tantor Postgres SE. В частности, 64-битный счетчик транзакций, автономные транзакции, оптимизирован алгоритм сжатия данных `pglz`, расширено использование инструкций SIMD центральных процессоров.

Часть изменений в ядре сделано добавлением опций в команды SQL: ориентир сбора статистики на уровне столбцов таблиц `ALTER TABLE t ALTER COLUMN c SET STATMULTIPLIER 100;` в дополнение к `SET STATISTICS`.

В пакет дистрибутива Tantor Postgres SE и SE 1C в дополнение к стандартным расширениям ванильного PostgreSQL добавлены

расширения: `credcheck`, `cube`, `fasttrun`, `fulleq`, `hypopg`, `mchar`, `page_repair`, `pg_cron`, `pg_hint_plan`, `pg_repack`, `pg_stat_kcache`, `pg_store_plans`, `pg_trace`, `pg_uuidv7`, `pg_wait_sampling`, `pgaudit`, `pgaudittofile`, `transp_anon`

библиотеки: `dbcopies_decoding`, `oauth_base_validator`, `online_analyze`, `pg_query_id`, `pg_stat_advisor`, `plantuner`, `wal2json`

утилиты: `pgcompacttable`, `pgcopydb`, `pg_diag`, `pg_repack`.

В стандартной поставке, в отдельных пакетах поставляются программы: `pg_anon`, `wal-g`, `pg_configurator`, `pg_cluster`, `pg_diag_setup`, `pg_sec_check`.

Тантор Лабс выпускает и поддерживает приложения, утилиты, расширения, не входящие в стандартную поставку СУБД Tantor Postgres (например, PostGIS, pgRouting) по отдельному соглашению ("сертификаты поддержки расширений"), так как перенос расширений на нужную версию СУБД, сборка под нужную операционную систему Linux, тестирование расширений, техническая поддержка сложны. Если расширение не требует доработки и переноса, то Тантор Лабс даёт инструкции для самостоятельной сборки.

В пакет дистрибутива Tantor Postgres SE добавлены расширения:
`http`, `orafce`, `pgl_ddl_deploy`, `pgq`, `vector`, `pg_archive` (дополнение к `pg_columnar`), `pg_columnar`, `pg_ivm`, `pg_partman`, `pg_qualstats`, `pg_tde`, `pg_throttle` (доработан для использования cgroup в linux), `pg_variables`, `pg_background`.

Синим цветом выделены параметры, появившиеся в 17.5 версии, зелёным в 16 версии Tantor Postgres.

Доработки, имеющиеся в Tantor Postgres **BE**, перечислены в документации:
https://docs.tantorlabs.ru/tdb/ru/18_1/be/differences.html

Параметры оптимизатора запросов

- В 17 версии Tantor Postgres появилось 6 параметров, которыми можно включать оптимизации планирования запросов
- Оптимизации позволяют существенно уменьшать время выполнения встречающихся на практике запросов
- Параметры добавлены, чтобы была гибкость настройки оптимизатора запросов
- По умолчанию оптимизации включены

```
postgres=# \dconfig enable_*
                Parameter                | Value
-----------------------------------------+------
enable_convert_exists_as_lateral_join | on
enable_convert_in_values_to_any       | on
enable_group_by_reordering            | on
enable_index_path_selectivity         | on
enable_join_pushdown                  | on
enable_self_join_removal              | on
```


Параметры оптимизатора запросов

В 17 версии Tantor Postgres появились параметры, которыми можно включать дополнительные оптимизации планировщика запросов. Эти оптимизации были созданы Тантор Лабс для устранения проблем с производительностью, возникающих в работе реальных приложений, в основном, 1С:ERP. При исследовании проблемы определялись запросы с неоптимальными планами. С оптимальными планами время выполнения запросов уменьшалось на порядки. По умолчанию оптимизации включены. Параметры добавлены, чтобы была гибкость в настройке планировщика и возможность быстро проверить эффективность оптимизаций.

`enable_convert_exists_as_lateral_join` позволяет планировщику преобразовывать подзапросы с EXISTS в латеральные SEMI JOIN, когда это возможно. Это преобразование может улучшить производительность в коррелированных подзапросах.

`enable_convert_in_values_to_any` включает оптимизацию планировщика, которая преобразует списки значений IN VALUES в конструкции ANY. Это может упростить планы запросов и обеспечить более эффективные пути выполнения.

`enable_group_by_reordering` будет ли планировщик запросов создавать план, который обеспечит группировку (GROUP BY) по столбцам, отсортированным в порядке столбцов, соответствующим столбцам, по которым дочерний узел плана возвращает данные в отсортированном виде. Например, при индексном сканировании. При отключенной оптимизации, планировщик учитывает порядок сортировки только для обслуживания ORDER BY (если это выражение присутствует).

`enable_index_path_selectivity` позволяет планировщику применять дополнительную оценку селективности при оценке путей соединения с использованием индексов. По умолчанию, планировщик выбирает составной индекс, созданный по меньшему числу столбцов, так как он меньше по размеру и не учитывает, что индексные записи указывают на большое число строк, которые не подходят под условие соединения. Параметр позволяет выбирать более подходящий индекс.

`enable_join_pushdown` позволяет планировщику перемещать внутренние соединения в подзапросы, когда это не изменит результат. Эта трансформация может позволить использовать более эффективные пути соединения.

`enable_self_join_removal` заменяет соединения таблиц самих с собой эквивалентными конструкциями, которые позволяют просканировать таблицу за один проход. Действует только на обычные (heap) таблицы.

Библиотека pg_stat_advisor

- автоматически обнаруживает запросы, где планировщик недооценивает или переоценивает число возвращаемых строк (actual rows отличается от планируемых rows)
- если коэффициент расхождения превышает заданный порог создает расширенную статистику на таблицу

```
set pg_stat_advisor.suggest_statistics_threshold = 0.33;
set pg_stat_advisor.min_duration = 0;
create table t (i int, j int);
insert into t select i/10, i/100 from generate_series(1, 1000000) i;
analyze t;
explain (analyze, buffers, timing off) select * from t where i = 100 and j = 10;
select pg_sleep(1);
\dX
                    List of extended statistics
 Schema |    Name    | Definition  | Ndistinct | Dependencies |   MCV
--------+------------+-------------+-----------+--------------+--------
 public | t_i_j_stat | i, j FROM t | defined   | defined      | defined
\! cat $PGDATA/log/postgresql-*.log | grep pg_stat_advisor
LOG:  pg_stat_advisor: successfully created extended statistics  from public.t
```


Библиотека pg_stat_advisor

pg_stat_advisor - библиотека автоматически обнаруживает запросы, где планировщик недооценивает или переоценивает число возвращаемых строк (actual rows отличается от планируемых rows, которые и сравниваются). Если `actual/planned` или `planned/actual >= pg_stat_advisor.suggest_statistics_threshold`, автоматически генерирует и выполняет команду `CREATE STATISTICS ON` по столбцам, после чего выполняет команду `ANALYZE` для обновления статистики. Тип статистики не указывается, поэтому создаются все виды статистики (`mcv, ndistinct, dependencies`). Команды по созданию статистики и обновлению статистики запускаются асинхронно фоновым рабочим процессом.

Библиотека загружается параметром `shared_preload_libraries`:

```
alter system set shared_preload_libraries = ..., pg_stat_advisor ;
```

Условия работы:

1. INSERT, UPDATE, DELETE не поддерживаются, только SELECT и WITH
2. Узел не является NestedLoop, MergeJoin, HashJoin
3. По временной таблице не создается
4. В WHERE указаны от 2 до 8 столбцов (включительно) из одной таблицы
5. Таблица была проанализирована и по крайней мере один столбец имеет ndistinct <> 1
6. Столбцы не покрываются составным индексом (для этого есть другая оптимизация `enable_index_path_selectivity`)
7. Параметр `pg_stat_advisor.min_duration >= 0` (значение по умолчанию -1)

```
set pg_stat_advisor.suggest_statistics_threshold = 0.33;
set pg_stat_advisor.min_duration = 0;
drop table if exists t;
create table t (i int, j int);
insert into t select i/10, i/100 from generate_series(1, 1000000) i;
analyze t;
explain (analyze, buffers, timing off) select * from t where i = 100 and j = 10;
-> Parallel Seq Scan on t (cost=0.00..10675.00 rows=1) (actual rows=3 loops=3)
select pg_sleep(1);
\dX
\! cat $PGDATA/log/postgresql-*.log | grep pg_stat_advisor
LOG:  pg_stat_advisor: successfully created extended statistics  from public.t
```

Патч предложен сообществу https://www.postgresql.org/message-id/aa034271-821c-42f3-92a1-b4112111c9c2%40tantorlabs.com

https://docs.tantorlabs.ru/tdb/ru/18_1/be/pg_stat_advisor.html

Параметры enable_temp_memory_catalog и enable_delayed_temp_file

- `enable_temp_memory_catalog` позволяет сохранять метаданные временных объектов в локальной памяти того процесса, который работает с ними и не вносить изменения в таблицы системного каталога
  - скорость доступа к метаданным уже созданной временной таблицы выше
  - способ хранения данных во временных объектах и способ доступа к данным не меняются
- `enable_delayed_temp_file` ускоряет работу с временными таблицами, позволяя не создавать файлы временных таблиц, пока хватает памяти локального буфера серверного процесса

Параметры конфигурации enable_temp_memory_catalog и enable_delayed_temp_file

При создании и удалении временных таблиц в PostgreSQL в таблицы системного каталога вносятся изменения несмотря на то, что временные таблицы доступны только тому процессу, в сессии которого они были созданы. Это может привести к раздуванию таблиц системного каталога и дополнительной нагрузке на экземпляр со стороны процессов автовакуума. Сильнее всего разрастаются pg_attribute, pg_class, pg_depend, pg_type.

Параметр **`enable_temp_memory_catalog`** позволяет сохранять метаданные временных объектов в локальной памяти того процесса, который работает с ними и не вносить изменения в таблицы системного каталога. Параметр **`enable_temp_memory_catalog`** устраняет изменение любых таблиц системного каталога при работе с временными таблицами. Параметр можно включить на разных уровнях, в том числе на уровне сессии.

При использовании параметра под хранение метаданных используется локальная память серверного процесса, вместо блоков таблиц системного каталога. Использование параметра не требует настройки параметров распределения памяти (work_mem, maintenance_work_mem). Скорость доступа к метаданным уже созданной временной таблицы выше, так как метаданные хранятся в локальной памяти серверного процесса и не требуется использование блокировок для доступа к таблицам и индексам системного каталога, что уменьшает конкуренцию. Параметр меняет способ хранения метаданных, способ хранения данных во временных объектах и способ доступа к данным не меняются, за исключением случая, когда в транзакции затрагиваются только временные таблицы. Если в транзакции не затрагиваются объекты постоянного хранения, то фиксация транзакции выполняется быстрее.

Параметр **`enable_delayed_temp_file`** ускоряет работу с временными таблицами (~15%), позволяя не создавать файлы временных таблиц, пока хватает памяти локального буфера серверного процесса.

Алгоритм сжатия pglz

- В Tantor Postgres оптимизирован алгоритм сжатия данных pglz
- Оптимизация удаляет потенциально избыточные операции, повышая скорость сжатия в 1.4 раза.
- Сжатие используется только для типов данных переменной ширины и используется по умолчанию для большинства типов данных, которые могут использовать сжатие
- Алгоритм сжатия pglz применяется по умолчанию для сжатия TOAST

```
postgres=# \dconfig *compress*
    Список параметров конфигурации
         Параметр          | Значение
---------------------------+----------
 default_toast_compression | pglz
 libpq_compression         | off
 wal_compression           | off
```


Алгоритм сжатия pglz

В Tantor Postgres оптимизирован алгоритм сжатия данных pglz. Оптимизация удаляет потенциально избыточные операции, повышая скорость сжатия в 1.4 раза.

Алгоритм сжатия pglz применяется по умолчанию для сжатия TOAST.

```
postgres=# \dconfig *compress*
    Список параметров конфигурации
         Параметр          | Значение
---------------------------+----------
 default_toast_compression | pglz
 libpq_compression         | off
 wal_compression           | off
(3 строки)
```

Сжатие используется только для типов данных переменной ширины (например, int фиксированной длины и не сжимается, text переменной длины и сжимается) и используется только тогда, когда режим хранения столбца MAIN или EXTENDED. EXTENDED является значением по умолчанию для большинства типов данных, поддерживающих хранение, отличное от PLAIN. Режим хранения можно установить командой:

```
ALTER TABLE имя ALTER COLUMN столбец SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN };
```

Алгоритм сжатия можно поменять на уровне столбца:

```
ALTER TABLE имя ALTER COLUMN столбец SET COMPRESSION {DEFAULT | pglz | lz4};
```

Технические детали оптимизаций кода алгоритма pglz в СУБД Tantor Postgres:

1) Используется более компактная хэш-таблица с индексами типа uint16 вместо указателей.
2) Игнорируется prev-указатель в хэш-таблице.
3) Используются более эффективные 4-байтные операции сравнения вместо 1-байтных.

Также макрофункции заменены обычными функциями (не влияет на производительность).

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-connection.html#GUC-LIBPQ-COMPRESSION

Параметр libpq_compression

- значение по умолчанию `off`
- определяет список поддерживаемых алгоритмом сжатия сетевого трафика
- допустимые значения: `off, on, lz4, zlib`
- можно задавать уровень сжатия, например:

```
alter system set libpq_compression =
'lz4:1,zlib:2';
```


Параметр libpq_compression

Параметр конфигурации Tantor Postgres `libpq_compression` включает поддержку сжатия в библиотеке `libpq`, реализованная новым параметром конфигурации `libpq_compression`. Функционал может использоваться клиентскими приложениями и драйверами, написанными на C или других языках, поддерживающих вызовы API к C.

Параметр `libpq_compression` может принимать следующие значения: `off, on, lz4, zlib`. По умолчанию `libpq_compression = off`.

Сжатие особенно полезно для импорта/экспорта данных с использованием команды COPY и для операций репликации (как физической, так и логической). Сжатие также может сократить время отклика для запросов, возвращающих большое количество данных (например, JSON, BLOB, текст и т.п.)

Этот параметр управляет доступными методами сжатия трафика между клиентом и сервером. Он позволяет отклонять запросы на сжатие, даже если сервер поддерживает эту функцию (например, из-за соображений безопасности или потребления процессорного времени). Для более точного контроля можно указать список разрешенных методов сжатия. Например, чтобы разрешить только методы lz4 и zlib, можно установить значение параметра в lz4, zlib. Также можно указать максимальный уровень сжатия для каждого метода, например, установив значение параметра в lz4:1,zlib:2, максимальный уровень сжатия для метода lz4 будет установлен 1, а для метода zlib 2. Если клиент запрашивает сжатие с более высоким уровнем сжатия, то будет установлен максимально допустимый уровень. По умолчанию, максимально возможный уровень сжатия для каждого алгоритма 1.

Появился начиная с версии 15.4 Tantor Postgres, в ванильном PostgreSQL 17 отсутствует.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/runtime-config-connection.html#GUC-LIBPQ-COMPRESSION

Параметр wal_sender_stop_when_crc_failed

- снижает вероятность передачи поврежденных журнальных записей на реплики
- по умолчанию отключён
- при включении процессы walsender будут проверять контрольные суммы журнальных записей перед передачей клиентам. Если контрольная сумма не совпадёт, то процессы попытаются прочесть запись из журнального буфера (WAL buffer). Если в журнальном буфере уже не будет записи или контрольная сумма не совпадёт, то `walsender` остановится

Параметр wal_sender_stop_when_crc_failed

Параметр конфигурации `wal_sender_stop_when_crc_failed` включает проверку контрольных сумм журнальных записей перед передачей их клиентам по протоколу репликации. Процесс walsender используется для передачи журнальных записей репликам и другим клиентам (`pg_recevewal`), читает WAL-сегменты из файловой системы. Журнальные записи защищены контрольными суммами, но по умолчанию walsender не проверяет контрольные суммы.

При установке значения параметра конфигурации `wal_sender_stop_when_crc_failed` в значение `true`, процессы `walsender` будут проверять контрольные суммы журнальных записей перед передачей клиентам. Если контрольная сумма не совпадёт, то процессы попытаются прочесть запись из журнального буфера (WAL buffer). Если в журнальном буфере не будет записи или контрольная сумма не совпадёт, то `walsender` остановится. Это позволит избежать распространения сбойных страниц на реплики и архивы WAL.

Параметр backtrace_on_internal_error

- Параметр относится к группе Developer Options, то есть не используется при промышленной эксплуатации
- Если этот параметр включен и возникает ошибка с кодом XX000 (internal_error), то трассировка стека записывается в диагностический журнал вместе с сообщением об ошибке
  - Это полезно при отладке внутренних ошибок, которые обычно не возникают в рабочей среде
- По умолчанию выключен.

Параметр backtrace_on_internal_error

Параметр относится к группе Developer Options, то есть не используется при промышленной эксплуатации. Если этот параметр включен и возникает ошибка с кодом XX000 (internal_error), то трассировка стека записывается в диагностический журнал вместе с сообщением об ошибке. Это полезно при отладке внутренних ошибок, которые обычно не возникают в рабочей среде. По умолчанию выключен.

Расширение uuid_v7

- расширение добавляет функцию **высокоскоростной** генерации uuid_v7
- uuid_v7 генерирует возрастающие значения, что **позволяет** использовать оптимизацию вставки в индекс btree
  - при очень частых вставках в сессии высокая скорость, не медленнее, чем с bigserial
  - структура индекса остаётся оптимальной и компактной
- uuid занимает 16 байт, bigint 8 байт
- пример использования:

```
create extension if not exists "uuid-ossp";
create extension if not exists pg_uuidv7;
create table tt1 (id uuid default uuidv7() primary key, data bigint);
create table tt2 (id bigint generated by default as identity primary key, data bigint);
select count(*), pg_indexes_size('tt1') from tt1;
```


Расширение uuid_v7

В PostgreSQL есть оптимизация вставки в индекс типа btree, позволяющая не спускаться с корня дерева индекса. Серверный процесс, который выполнил вставку в правый листовой блок, запоминает ссылку на него и при последующей вставке, если новое значение больше предыдущего (или пусто) и не проходит путь от корня до листового блока. Оптимизация используется при числе уровней в индексе начиная со второго (макрос в коде ядра PostgreSQL BTREE_FASTPATH_MIN_LEVEL).

При использовании в качестве уникального ключа типа uuid возрастающие значения генерирует `uuidv7()` и оптимизация работает. При использовании v4 (и других) оптимизации быстрой вставки не будет, так как вставляются случайные значения, а не возрастающие. Более того, вставка в разные листовые блоки индекса приводит к увеличению объема журнала за счёт записи в журнал большего числа полных образов страниц (full page images, FPI). Пример теста:

```
pgbench -i
echo "insert into tt1(data) values(1);" > txn.sql
create extension if not exists "uuid-ossp";
create extension if not exists pg_uuidv7;
create table tt1 (id uuid default uuidv7() primary key, data bigint);
vacuum analyze tt1;
pgbench -T 30 -c 16 -f txn.sql
select count(*), pg_indexes_size('tt1') from tt1;
drop table if exists tt1;
create table tt1 (id bigint generated by default as identity primary key, data bigint);
```

Скорость вставки сравнима: для uuidv7() tps = 1734, для bigint tps = 1707.

Размеры индексов по столбцу типа uuid больше, чем размер индекса по столбцу типа bigint из-за того, что размер поля типа uuid (16 байт) в два раза больше, чем размер поля bigint (8 байт). Для uuidv7 число строк в примере теста 97172, а размер индекса 3088384 байт, для bigint число строк 99050, а размер индекса 2236416 байт.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/pg_uuidv7.html

Расширение pg_tde (Transparent Data Encryption)

- Реализует "прозрачное" шифрование данных (Transparent Data Encryption)
- Прозрачность означает, что клиент получает и передает незашифрованные данные
- Не шифрует данные в памяти (в буферном кэше) и при передаче по сети
- Шифрование выполняется протоколами: AES, Magma, Kuznyechik, ChaCha20
- Физическая и логическая репликация поддерживаются
- Таблицы системного каталога не шифруются
- pg_rewind пока не работает с зашифрованным WAL
- WAL шифруются полностью

Расширение pg_tde (Transparent Data Encryption)

Реализует "прозрачное" шифрование данных (Transparent Data Encryption). Прозрачность означает, что клиент получает и передает незашифрованные данные. Смысл этой опции, чтобы при похищении файлов данных и журнальных (WAL) файлов кластера без похищения файлов (устройств) с ключами, доступа к зашифрованным данным не было. pg_tde не шифрует данные в памяти (в буферном кэше) и при передаче по сети. На Astra Linux пакет lib**gost**-astra настраивает OpenSSL автоматически и шифрование выполняется протоколами с симметричным ключом: AES, Magma, Kuznyechik, ChaCha20.

Можно зашифровать существующие таблицы:

```
ALTER TABLE t SET ACCESS METHOD tde_heap;
```

Параметром конфигурации можно установить, чтобы создаваемые таблицы шифровались:

```
ALTER SYSTEM SET default_table_access_method = tde_heap;
SET default_table_access_method = tde_heap;
```

Метод доступа tde_heap работает поверх метода доступа heap. В буферном кэше данные хранятся в нешифрованном виде.

Реализована только ротация основного ключа. Каждый файл шифруется поблочно (8Кб) своим ключом. Для ротации ключей, которыми шифруются файлы, пришлось бы перешифровывать файлы.

Особенности:

1) Физическая и логическая репликация поддерживаются.

2) Таблицы системного каталога не шифруются.

3) pg_rewind пока не работает с зашифрованным WAL, это будет реализовано в будущих версиях.

4) WAL-G не поддерживает отправку дельт WAL, если WAL зашифрован.

5) WAL шифруются полностью. Таблицы (в том числе временные) шифруются с зависимыми объектами: TOAST, индексами.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/pg_tde.html

Валидатор oauth_base_validator

- В Tantor Postgres 17 есть способ аутентификации oauth (OAuth 2.0), который появится в PostgreSQL 18 версии
- Этот способ использует внешнюю службу для аутентификации
- Метод oauth вставляется в четвертое поле строки в файле `pg_hba.conf`

```
#TYPE DATABASE USER ADDRESS METHOD
local all      all          oauth issuer="http://1.1.1.1:80/realms/a" scope="openid" map="o1"
```

- Для использования способа аутентификации oauth нужно на языке C написать "валидатор"
- В Tantor Postgres библиотека с валидатором поставляется
- Пример аутентификации в psql:

```
psql "user=alice dbname=postgres oauth_issuer=http://1.1.1.1:80/realms/a oauth_client_id=user1
oauth_client_secret=AbCdEf123GhIjKl"
Visit http://1.1.1.1:80/realms/a/device and enter the code: XYZX-XYZO
postgres=>
```


Валидатор oauth_base_validator

В Tantor Postgres 17 есть способ аутентификации oauth (OAuth 2.0), который появится в PostgreSQL 18 версии. Этот способ, подобный **radius**, использует внешнюю службу для аутентификации. Метод oauth вставляется в четвертое поле строки в файле `pg_hba.conf`.

Пример:

```
#TYPE DB  USER ADDR METHOD
local all all       oauth issuer="http://1.1.1.1:80/realms/a" scope="openid"
map="o1"
```

Имена могут сопоставляться через через `pg_ident.conf`:

```
# MAP SYSTEM-USERNAME                          PG-USERNAME
  o1  "0fc72b6f-6221-4ed8-a916-069e7a081d14" "alice"
```

Можно сопоставлять через код валидатора, если это реализовано в валидаторе. В этом случае вместо `map="o1"` в строку `pg_hba.conf` нужно вставить опцию `delegate_ident_mapping=1`

Для использования способа аутентификации oauth нужно на языке C написать "валидатор". Название библиотеки указывается в параметре конфигурации:

```
alter system set oauth_validator_libraries = 'oauth_base_validator';
```

В Tantor Postgres библиотека с валидатором поставляется.

Чтобы можно было использовать протокол http, нужно использовать переменную окружения:

```
export PGOAUTHDEBUG="UNSAFE"
```

и запустить клиента:

```
psql "user=alice dbname=postgres oauth_issuer=http://1.1.1.1:80/realms/a
oauth_client_id=user1 oauth_client_secret=AbCdEf123GhIjKl"
```

Выведется сообщение, куда зайти и какой код ввести

```
Visit http://1.1.1.1:80/realms/a/device and enter the code: XYZX-XYZO
```

После ввода кода на адресе внешней службы соединение будет установлено и **psql** выдаст приглашение:

```
postgres=>
```

Валидатор написан на языке C.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/oauth-base-validator.html

Библиотека credcheck

- Использует библиотеку, которая может загружаться на уровне кластера (параметр `shared_preload_libraries`) и для одной сессии (командой `LOAD credcheck`)
- При загрузке регистрирует 30 параметров конфигурации, которыми можно задать проверки сложности пароля, защиту от подбора пароля, параметры повторного использования паролей, список ролей на которые проверки не действуют и т.п.
- В расширении 8 функций и 2 представления
- Расширение срабатывает при создании роли, переименовании, изменения пароля, аутентификации

```
postgres=# \dconfig credcheck.*
              Parameter                | Value
---------------------------------------+-------
 credcheck.auth_delay_ms               | 0
 credcheck.encrypted_password_allowed  | off
 credcheck.max_auth_failure            | 0
...
 credcheck.whitelist                   |
(30 rows)
```


Библиотека credcheck

Использует библиотеку, которая может загружаться на уровне кластера (параметр `shared_preload_libraries`) и для одной сессии (командой `LOAD credcheck`). При загрузке регистрирует 30 параметров конфигурации, которыми можно задать проверки сложности пароля, защиту от подбора пароля, параметры повторного использования паролей, список ролей на которые проверки не действуют и т.п.

```
postgres=# LOAD 'credcheck';
postgres=# CREATE EXTENSION credcheck;
postgres=# \dconfig credcheck.*
               Parameter                | Value
----------------------------------------+-------
 credcheck.auth_delay_ms                | 0
 credcheck.encrypted_password_allowed   | off
 credcheck.max_auth_failure             | 0
 credcheck.no_password_logging          | on
 credcheck.password_contain             |
...
 credcheck.username_min_upper           | 0
 credcheck.username_not_contain         |
 credcheck.whitelist                    |
(30 rows)
```

Можно установить расширение, в котором есть 8 функций и 2 представления.

Расширение срабатывает при создании роли, переименовании, изменения пароля, аутентификации.

Параметром `credcheck.max_auth_failure` задаётся число неудачных попыток аутентификации перед блокировкой роли. Параметр `credcheck.auth_delay_ms` позволяет ввести задержку при неудачном вводе пароля, что защищает от подбора паролей. Для защиты от подбора паролей можно использовать стандартное расширение `auth_delay`, но такой способ защиты от подбора **усугубляет** DDOS атаки, так как серверные процессы на время задержки удерживая ресурсы, в отличие от блокировки ролей.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/credcheck.html
https://docs.tantorlabs.ru/tdb/ru/18_1/se/auth-delay.html

Расширения fasttrun и online_analyze

- Усечение временных таблиц обновляет таблицу pg_class
- Расширение fasttrun содержит функцию fasttruncate
  - у файлов не меняется название
  - таблицы системного каталога не меняются
  - может использоваться вместо команды TRUNCATE
  - работает только с временными таблицами
- После усечения или других команд можно собрать статистику
- Для автоматического сбора статистики можно использовать расширение online_analyze
  - расширение настраивается параметрами конфигурации

```
select fasttruncate('tt');
 INFO:  analyzing "pg_temp_5.tt"
 INFO:  "tt": scanned 0 of 0 pages, containing 0 live rows and 0 dead rows; 0 rows in sample, 0 estimated
total rows
 INFO:  analyze "tt" took 0.00 seconds
  fasttruncate
--------------

 (1 row)
```


Расширения fasttrun и online_analyze

Усечение (TRUNCATE) временной таблицы приводит к удалению и созданию файлов с новым названием, строка в pg_class обновляется. Старые версии строк не могут очищаться, если горизонт базы долго удерживается и pg_class и индексы раздуваются.

Расширение fasttrun состоит из одной функции `fasttruncate('имя')`. При использовании функции временная таблица усекается, у файлов название не меняется. Приложения 1С вместо команды TRUNCATE используют вызов этой функции. Функция работает только с временными таблицами:

```
select fasttruncate('t');
ERROR:  Relation isn't a temporary table
```

Для использования расширения нужно загрузить библиотеку и установить расширение:

```
alter system set shared_preload_libraries = fasttrun, fulleq, mchar;
create extension fasttrun;
```

После вставки или изменении строк во временных таблицах может быть полезным пересобрать статистику для планировщика. 1С Предприятие начиная с версии 8.3.13, выполняет команду ANALYZE после вставки строк во временную таблицу. Для других приложений, которые этого не делают, можно использовать расширение `online_analyze`. Загружать его для всех сессий не стоит, так как если статистика собирается отдельной командой, то автоматический сбор об этом не знает, повторяет то же самое действие, что приводит к лишнему потреблению ресурсов. Более того, статистика собирается синхронно приводит к замедлению выполнения команд, которые вызывают срабатывание расширения. Пример использования расширения на уровне сессии:

```
load 'online_analyze';
set online_analyze.enable = on;
set "online_analyze.verbose" = on;
set online_analyze.table_type = 'temporary';
```

Двойные кавычки у второго параметра нужны потому, что verbose зарезервированное слово. Этот параметр выполняет команду ANALYZE VERBOSE. После выполнения команды, приводящей к анализу, вызывающему команду передаются уведомления уровня INFO.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/fasttrun.html
https://docs.tantorlabs.ru/tdb/ru/18_1/se/online_analyze.html

Расширение mchar

- Добавляет поддержку типов mchar, mvarchar для совместимости с Microsoft SQL Server
- Для типов mchar и mvarchar определяет функции и операторы:
  - length()
  - substr(str, pos[, length])
  - || конкатенация с разными типами (mchar || mvarchar)
  - < <= = >= > сравнение без чувствительности к регистру (ICU)
  - &< &<= &= &>= &> сравнение с чувствительностью к регистру (ICU)
  - LIKE
  - SIMILAR TO
  - ~ (регулярные выражения)
- Добавлят неявное приведение типов mchar к mvarchar и обратно
- поддержку типов индексами b-tree и hash
- Использование индексов для выполнения оператора LIKE

Расширение mchar

Добавляет поддержку типов mchar, mvarchar для совместимости с Microsoft SQL Server.
Для типов mchar и mvarchar поддерживаются функции и операторы:
`length()`
`substr(str, pos[, length])`
`||` конкатенация с разными типами (`mchar || mvarchar`)
`< <= = >= >` сравнение без чувствительности к регистру (ICU)
`&< &<= &= &>= &>` сравнение с чувствительностью к регистру (ICU)
`LIKE`
`SIMILAR TO`
~ (регулярные выражения)
Неявное приведение типов mchar к mvarchar и обратно
Поддержку типов индексами b-tree и hash
Использование индексов для выполнения оператора LIKE
https://docs.tantorlabs.ru/tdb/ru/18_1/se1c/mchar.html

Расширение fulleq

- При использовании оператора "=" для сравнения значений, если хотя бы один из операндов имеет значение NULL, результатом будет NULL
- В приложениях 1С часто используется оператор "==", который возвращает true, когда операнды равны или оба имеют значение NULL, что удобно при работе с базами данных, особенно с 1С, где операторы и семантика работы с NULL отличаются от стандарта SQL
- Оператор “==” из расширения fulleq позволяет высокоэффективно выполнять сравнение значений с использованием нужной логики

Расширение fulleq

При использовании оператора "=" для сравнения значений, если хотя бы один из операндов имеет значение NULL, результатом будет NULL. В приложениях 1С часто используется оператор "==", который возвращает true, когда операнды равны или оба имеют значение NULL. Это удобно при работе с базами данных, особенно с 1С, где операторы и семантика работы с NULL отличаются от стандарта SQL.

Оператор "==" позволяет высокоэффективно выполнять сравнение значений с использованием нужной логики.

Оператор "==", примененный к двум операндам, возвращает true, если они равны или оба имеют значение NULL.

Оператор "==", примененный к двум операндам, возвращает false, если они не равны или один из них имеет значение NULL.

https://docs.tantorlabs.ru/tdb/ru/18_1/se1c/fulleq.html

Расширение orafce

- Расширение, предоставляющее функции, упрощающие миграцию кода приложений с Oracle Database
- orafce эмулирует часть функционала и пакетов Oracle Database
- Создаёт большое количество функций и других объектов и типов данных, которые работают так же, как их аналоги в Oracle Database
- Расширение упрощает миграцию кода приложений в Tantor Postgres с Oracle Database
- Расширение создаёт 15 схем, в которых находятся объекты расширения. Имена восьми схем соответствуют именам пакетов в Oracle Database

Расширение orafce

Расширение orafce есть в Tantor Postgres SE.

Расширение содержит функции, типы данных, которые похожи на те, что есть в Oracle Database.

Функции и операторы orafce эмулируют часть функций из часто используемых пакетов процедур Oracle Database.

Использование orafce уменьшает время на миграцию и снижает трудоёмкость миграции кода приложений.

При миграции с Oracle Database на Tantor Postgres в командах и программном коде могут использоваться функции, процедуры, типы данных, которые имеются в Oracle Database и отсутствуют в PostgreSQL и стандарте SQL. Переписывать код может быть достаточно трудоёмко, особенно если команд много.

Расширение orafce создаёт большое число функций, которые работают подобно одноимённым функциям и процедурам в Oracle Database.

Это наиболее распространённые подпрограммы, которые чаще всего используются в коде приложений, работающих с Oracle Database. Расширение не покрывает весь набор функций, также синтаксис вызова некоторых функций может отличаться, и не нужно предполагать, что команды SQL, которые выполнялись в Oracle Database, будут выполняться в postgres.

Цель расширения - упростить миграцию кода, дать возможность выполнения кода без существенных изменений, и постепенное переписывание и оптимизация выполнения команд SQL.

Функции из этого расширения могут быть полезными и сами по себе.

В Oracle Database программные единицы (функции и процедуры) находятся в "пакетах".

В postgres есть объект "схема", который обладает схожим функционалом, поэтому расширение создаёт довольно большое количество схем, имена которых соответствуют названиям пакетов в Oracle Database.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/orafce.html

Расширение http

- расширение http предоставляет возможность выполнять HTTP и HTTPS запросы прямо из SQL
- устанавливается в базу данных командой `create extension http;`
- можно создать триггер, обращающийся к веб-службе, передать данные и получить результат, который можно использовать в логике триггера
- pgsql-http может быть полезен в:
  - при интеграции с внешними API
  - в интерактивных приложениях
  - для обработки данных в реальном времени

Расширение http

Расширение http доступно в Tantor Postgres SE.

Устанавливается в базу данных командой `create extension http;`

Расширение http предоставляет возможность выполнять HTTP и HTTPS запросы прямо из SQL.

Например, можно создать триггер, обращающийся к веб-службе, передать данные и получить результат, который можно использовать в логике триггера. Использование протокола HTTP требует осторожности. В частности, следует избегать создания ситуаций, когда работа серверного процесса будет заблокирована из-за долгого ожидания ответа на HTTP-запрос.

Функционал pgsql-http может быть полезен в следующих задачах:

1) При интеграции с внешними API: в некоторых случаях удобнее работать по протоколу REST напрямую из базы данных, особенно когда получаемые от веб-службы данные требуется использовать в SQL-командах. Расширение pgsql-http позволяет делать это, поддерживая все основные методы протокола HTTP, включая GET, POST, PUT, DELETE, а также относительно новый метод PATCH.

2) В интерактивных приложениях: в некоторых сценариях использования, PostgreSQL может быть частью интерактивного веб-приложения, где база данных взаимодействует с пользователем посредством HTTP. http может использоваться для отправки запросов на сервер приложений и получения ответов на них.

3) Для обработки данных в режиме реального времени: позволяет обеспечить доступ к данным, которые постоянно обновляются и доступны клиентам по протоколу HTTP. С помощью http, можно запросить эти данные напрямую со стороны сервера базы данных.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/pgsql-http.html

Расширение pg_store_plans

- Расширение для отслеживания статистики выполнения SQL-запросов
- сохраняет полные планы запросов, а не только их статистику или текст
- использование pg_store_plans может увеличить нагрузку на систему из-за сбора и хранения дополнительных данных

Расширение pg_store_plans

Во всех версиях Tantor Postgres имеется расширение pg_store_plans.

Расширение предоставляет средства для отслеживания статистики плана выполнения всех запросов SQL.

Используется платформой Tantor для сбора статистики планов запросов.

В отличие от других инструментов, таких как auto_explain, pg_stat_statements, pg_stat_plans, расширение pg_store_plans способно собирать и хранить полные планы запросов, а не только статистику или текст запросов.

Позволяет анализировать, как запросы выполняются в системе.

Использование pg_store_plans может увеличить нагрузку на вашу систему из-за дополнительного сбора и хранения информации о планах запросов.

pg_store_plans:

1) автоматически сохраняет планы выполнения запросов, что позволяет исследовать, как запросы выполняются в вашей базе данных.

2) хранит планы запросов на протяжении длительного времени, что позволяет вам анализировать исторические данные и определять, как изменения в коде приложения или базе данных влияют на производительность запросов.

3) можно выявить медленные запросы и определить, какие операции в плане запроса занимают больше всего времени. Это может помочь в оптимизации запросов и улучшении производительности базы данных.

4) Совместимо с другими расширениями. pg_store_plans может использоваться вместе с другими расширениями, такими как pg_stat_statements и pg_qualstats.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/pg_store_plans.html

Расширение pg_variables

- Расширение, позволяющее создавать и использовать переменные в SQL-запросах
- по умолчанию не транзакционны, но могут быть транзакционными
- хранятся в памяти серверного процесса
- существуют только в текущем сеансе пользователя
- могут использоваться на физических репликах, временные таблицы не могут
- скорость работы сравнима с временными таблицами
- отсутствие накладных расходов (не требуют реальный номер транзакции, не создают файлы)
- альтернатива set_config()/current_setting()

Расширение pg_variables

В Tantor Postgres SE имеется расширение pg_variables.

Расширение pg_variables позволяет определять и использовать переменные внутри запросов SQL на сервере PostgreSQL.

Переменные могут быть использованы для хранения временных значений, обмена данными между функциями, хранения промежуточных результатов и т.д.

Предоставляет средства для отслеживания статистики плана выполнения для всех SQL-запросов, выполненных сервером Tantor.

Предоставляет функции для работы с переменными различных типов. Созданные переменные существуют только в текущем сеансе пользователя.

По умолчанию созданные переменные не являются транзакционными (т.е. на них не влияют команды BEGIN, COMMIT, ROLLBACK).

Расширение позволяет хранить в памяти серверного процесса значения переменных различных типов, в том числе: численных, текстовых, дата-временных, логического, jsonb, массивов, составных **типов**. Переменные доступны в рамках сессии.

Переменные могут использоваться как альтернатива временным таблицам. Можно работать с наборами значений с помощью функций pgv_select и pgv_insert. Скорость работы может быть выше, чем при работе с данными с помощью временных таблиц. Переменные **могут** использоваться на физических репликах, временные таблицы не могут. Переменные **могут** иметь составной тип, в том числе образ строк.

Расширение - альтернатива стандартным функциям set_config()/current_setting(), которые удобны для хранения нескольких строковых значений в памяти серверного процесса. Производительность работы этих функций аналогична расширению.

Накладные расходы отсутствуют: не требуется реальный номер транзакции, не используются файлы, не меняется содержимое таблиц системного каталога, не используют кэш операционной системы. Отсутствует деградация производительности при активном изменении значений переменных, характерном при активном изменении строк во временных таблицах. С помощью функции pgv_stats можно посмотреть сколько используется памяти.

Функционал аналогичен переменным пакетов, контекстов приложений (application contexts), PLS-таблиц (index by tables) в Oracle Database, поэтому расширение востребовано при миграции на СУБД Tantor Postgres.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/pg_variables.html

Производительность при использовании pg_variables

- При сравнении скорости доступа к таблицам в памяти, временным, обычным результаты не очевидны
- Расширение pg_variables не является стандартным и не пользуется популярностью из-за неудобного синтаксиса использования

```
 select * from pgv_select('bookings', 'tickets', '0005432020304'::char(13)) as (ticket_no character(13),
book_ref character(6), passenger_id character varying(20), passenger_name text, contact_data jsonb);
 Time: 0.281 ms
 select * from tickets where ticket_no='0005432020304';
 Time: 0.266 ms
 select book_ref from tickets where passenger_name like '%G IVANOV' limit 1;
 Time: 0.463 ms
 select book_ref from tickets1 where passenger_name like '%G IVANOV' limit 1;
 Time: 1.169 ms
 select book_ref from pgv_select('bookings', 'tickets', '0005432020304'::char(13)) as (ticket_no
character(13), book_ref character(6), passenger_id character varying(20), passenger_name text, contact_data
jsonb) where passenger_name like '%G IVANOV' limit 1;
 Time: 0.185 ms
```


Преимущества расширения pg_variables

С помощью функций расширения pg_variables можно сохранять как скалярные переменные, так и составные типы (образы строк). Поиск строк может выполняться полным сканированием или по хэшу. Структуры хранятся в локальной памяти процесса и смысла использовать другие способы типа btree нет.

```
 wget https://edu.postgrespro.com/demo-medium-en.zip
 zcat demo-medium.zip | psql
 psql -d demo
 create extension pg_variables;
 demo=# \o t.tmp
 \timing on \\
 select pgv_insert('bookings', 'tickets', tickets) from tickets;
 Time: 1634.973 ms (00:01.635)
 demo=# create temp table tickets1 as select * from tickets;
 Time: 557.808 ms
 select * from tickets1 where ticket_no='0005432020304';
 Time: 269.005 ms
 select * from tickets where ticket_no='0005432020304';
 Time: 0.266 ms
select * from pgv_select('bookings', 'tickets', '0005432020304'::char(13)) as (ticket_no character(13), book_ref
character(6), passenger_id character varying(20), passenger_name text, contact_data jsonb);
    ticket_no    | book_ref | passenger_id | passenger_name |                                contact_data
 ----------------+----------+--------------+----------------+-----------------------------------------------------
  0005432020304 | F5C81C   | 7257 672943  | OLEG IVANOV    | {"email": "oleg-ivanov_1984@postgrespro.ru",
"phone": "+70632852802"}
 (1 row)
 Time: 0.281 ms
```

Скорость выборки из таблицы в памяти чуть медленнее выборки из обычной таблицы по индексу. На временной таблице индекс не был создан. Если создать индекс типа btree:

```
 create index on tickets1(ticket_no);
 Time: 5615.559 ms (00:05.616)
 select * from tickets1 where ticket_no='0005432020304';
 Time: 0.302 ms
```

то скорость индексного доступа имеет широкий разброс и не отличается от обычной таблицы.

```
 select book_ref from tickets where passenger_name like '%G IVANOV' limit 1;
 Time: 0.463 ms
 select book_ref from tickets1 where passenger_name like '%G IVANOV' limit 1;
 Time: 1.169 ms
 select book_ref from pgv_select('bookings', 'tickets', '0005432020304'::char(13)) as (ticket_no character(13),
book_ref character(6), passenger_id character varying(20), passenger_name text, contact_data jsonb) where
passenger_name like '%G IVANOV' limit 1;
 Time: 0.185 ms
```

https://pgconf.ru/media//2019/02/08/zakirov-pg-variables-pgconf-ru-2019.pdf

Преимущества расширения pg_variables

- Временные таблицы нельзя использовать на репликах
- функции расширения pg_variables можно использовать на репликах точно так же, как на мастере
- pg_variables хранит данные только в локальной памяти серверного процесса и не использует временные файлы
- Переменные могут быть транзакционными и нет
- На сохраняемые объекты действует ограничение строкового буфера - 1Гб

```
select pgv_insert('bookings','t2', pgbench_branches) from pgbench_branches;
select * from pgv_select('bookings','t2',1) as (bid int, bbalance int, filler character(88));
 bid | bbalance | filler
-----+----------+--------
   1 |        0 |
select pgv_select('bookings','t2',1);
 pgv_select
------------
 (1,0,)
```


Преимущества расширения pg_variables

Временные таблицы нельзя использовать на репликах. Основное преимущество расширения pg_variables в том, что его возможности по хранению временных данных можно использовать на репликах точно так же, как на мастере. Благодаря этому, на репликах можно реализовать сложную аналитику, которой требуется хранение промежуточных данных и переносить её на реплики.

При этом нужно помнить, что pg_variables хранит данные **только в локальной памяти** серверного процесса и не использует временные файлы. На сохраняемые объекты действует ограничение строкового буфера 1Гб. Это не является проблемой, так как в других СУБД аналогичный функционал имеет аналогичные ограничения по памяти. Недостатком pg_variables можно считать неудобство (непривычность) использования, которые можно обходить. Например, функция выдаёт строки, вместо числа вставленных строк, что порождает сетевой трафик, если вызывать функцию с клиента:

```
select pgv_insert('bookings','t2', pgbench_branches) from pgbench_branches;
 pgv_insert
------------


(1 row)
```

При выборке составных типов приходится указывать детали структуры:

```
select * from pgv_select('bookings','t2',1) as (bid int, bbalance int, filler
character(88));
 bid | bbalance | filler
-----+----------+--------
   1 |        0 |
select pgv_select('bookings','t2',1);
 pgv_select
------------
 (1,0,)
```

Одно из преимуществ расширения в том, что можно создавать транзакционные переменные, то есть изменения значений могут меняться атомарно по фиксации транзакций, откатываться. По умолчанию, создаются нетранзакционные переменные.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/pg_variables.html

Расширение pg_stat_kcache

- дополняет pg_stat_statements и зависит от него
- собирает статистику linux, выполняя системный вызов getrusage после выполнения каждой команды
- в отличие от утилит операционной системы, расширение собирает статистики с детальностью до команды
- позволяет различать читался блок с диска или из страничного кэша
- использует два буфера в разделяемой памяти

```
 select * from (select *,lead(off) over(order by off)-off as diff from pg_shmem_allocations) as a where name
like 'pg_%';
           name            |    off    | size  | allocated_size |  diff
---------------------------+-----------+-------+----------------+---------
 pg_stat_statements        | 148162816 |    64 |            128 |     128
 pg_stat_statements hash   | 148162944 |  2896 |           2944 | 2188544
 pg_stat_kcache            | 150351488 |   992 |           1024 |    1024
 pg_stat_kcache hash       | 150352512 |  2896 |           2944 | 1373056
select name, setting, context, min_val, max_val from pg_settings where name like '%kcache%';
              name               | setting |  context   | min_val |  max_val
---------------------------------+---------+------------+---------+------------
 pg_stat_kcache.linux_hz         | 333333  | user       | -1      | 2147483647
 pg_stat_kcache.track            | top     | superuser |         |
 pg_stat_kcache.track_planning | off     | superuser |         |
```


Расширение pg_stat_kcache

Расширение дополняет pg_stat_statements и зависит от него. Отсутствует в стандартной поставке. Расширение работает стабильно и накладные расходы незначительны. Статистика shared_blks_read не различает находились ли страницы (размер 4Кб) из которых состоит блок (размер 8Кб) в страничном кэше linux или читались с диска. Расширение позволяет это различать, оно собирает статистику linux, выполняя системный вызов getrusage после выполнении каждой команды. Статистики, собираемые расширением, могут быть полезны для определения эффективности кэширования и возможных узких мест. Данные, собираемые системным вызовом, записываются в разделяемую память.

Вызов getrusage также используется параметром конфигурации log_executor_stats=on (по умолчанию отключён). Этот параметр конфигурации сохраняет собранную статистику операционной системы в диагностический лог кластера, что менее удобно для просмотра и необходимости следить за размером лога.

В отличие от утилит операционной системы, расширение собирает статистики с детальностью до команды. Число команд, по которым собирается статистика и размер структур разделяемой памяти определяется параметром pg_stat_statements.max (по умолчанию 5000), так как это расширение зависит от расширения pg_stat_statements.

Расширение использует два буфера в разделяемой памяти:

```
 select * from (select *,lead(off) over(order by off)-off as diff from
pg_shmem_allocations) as a where name like 'pg_%';
           name            |    off    | size  | allocated_size |  diff
---------------------------+-----------+-------+----------------+---------
 pg_stat_statements        | 148162816 |    64 |            128 |     128
 pg_stat_statements hash   | 148162944 |  2896 |           2944 | 2188544
 pg_stat_kcache            | 150351488 |   992 |           1024 |    1024
 pg_stat_kcache hash       | 150352512 |  2896 |           2944 | 1373056
```

Расширение имеет параметры:

```
\dconfig *kcache*
```

pg_stat_kcache.linux_hz (по умолчанию -1) устанавливается автоматически в значение параметра linux CONFIG_HZ и используется для компенсации ошибок сэмплирования. Менять не нужно.

pg_stat_kcache.track=top параметр - аналог pg_stat_statements.track

pg_stat_kcache.track_planning=off аналог pg_stat_statements.track_planning

Статистики, собираемые pg_stat_kcache

Статистики в представлениях pg_stat_kcache и pg_stat_kcache_detail:

- reads_blks reads, in 8K-blocks
- writes_blks writes, in 8K-blocks
- user_time user CPU time used
- system_time system CPU time used
- minflts page reclaims (soft page faults)
- majflts page faults (hard page faults)
- nswaps swaps
- msgsnds IPC messages sent
- msgrcvs IPC messages received
- nsignals signals received
- nvcsws voluntary context switches
- nivcsws involuntary context switches

```
alter system set shared_preload_libraries = pg_stat_statements, pg_wait_sampling, pg_stat_kcache;
create extension pg_stat_kcache;
\dx+ pg_stat_kcache
 function pg_stat_kcache()
 function pg_stat_kcache_reset()
 view pg_stat_kcache
 view pg_stat_kcache_detail
```


Статистики, собираемые pg_stat_kcache

Команды для установки расширения:

```
apt install clang-13
wget https://github.com/powa-team/pg_stat_kcache/archive/REL2_3_0.tar.gz
tar xzf ./REL2_3_0.tar.gz
cd pg_stat_kcache-REL2_3_0
make
make install
alter system set shared_preload_libraries = pg_stat_statements, pg_wait_sampling, pg_stat_kcache;
sudo systemctl restart  tantor-se-server-18.service
create extension pg_stat_kcache;
```

Расширение состоит из двух представлений и двух функций:

```
\dx+ pg_stat_kcache
 function pg_stat_kcache()
 function pg_stat_kcache_reset()
 view pg_stat_kcache
 view pg_stat_kcache_detail
```

Представление pg_stat_kcache_detail имеет столбцы: query, top, rolname и выдаёт данные с точностью до команды. Статистики выдаются из 14 столбцов для планирования и 14 столбцов для выполнения команд.

Представление pg_stat_kcache содержит суммарные статистики из pg_stat_kcache_detail, сгруппированные по базам данных:

```
CREATE VIEW pg_stat_kcache AS SELECT datname, SUM(столбцы) FROM pg_stat_kcache_detail WHERE top IS TRUE GROUP BY datname;
```

Статистики в обоих представлениях:

```
exec_reads       reads, in bytes
exec_writes      writes, in bytes
exec_reads_blks  reads, in 8K-blocks
exec_writes_blks writes, in 8K-blocks
exec_user_time   user CPU time used
exec_system_time system CPU time used
exec_minflts     page reclaims (soft page faults)
exec_majflts     page faults (hard page faults)
exec_nswaps      swaps
exec_msgsnds     IPC messages sent
exec_msgrcvs     IPC messages received
exec_nsignals    signals received
exec_nvcsws      voluntary context switches
exec_nivcsws     involuntary context switches
```

Расширение pg_wait_sampling

- входит во все сборки Tantor Postgres
- выдает статистику по событиям ожиданий всех процессов экземпляра
- для установки нужно загрузить библиотеку и установить расширение:

```
alter system set shared_preload_libraries = pg_stat_statements, pg_stat_kcache, pg_wait_sampling;
create extension if not exists pg_wait_sampling;
```

- библиотека pg_wait_sampling должна быть указана позже pg_stat_statements, чтобы расширение не перезаписало идентификаторы запросов (queryid)
- расширение использует фоновый процесс pg_wait_sampling collector
- процесс опрашивает состояние всех процессов экземпляра
- в расширение входят 4 функции и 3 представления:

```
\dx+ pg_wait_sampling
 function pg_wait_sampling_get_current(integer)
 function pg_wait_sampling_get_history()
 function pg_wait_sampling_get_profile()
 function pg_wait_sampling_reset_profile()
 view pg_wait_sampling_current
 view pg_wait_sampling_history
 view pg_wait_sampling_profile
```


Расширение pg_wait_sampling

Расширение входит во все сборки Tantor Postgres. Выдает статистику по событиям ожиданий всех процессов экземпляра. Для установки нужно загрузить библиотеку и установить расширение:

```
alter system set shared_preload_libraries = pg_stat_statements, pg_stat_kcache, pg_wait_sampling, pg_qualstats, pg_store_plans;
create extension if not exists pg_wait_sampling;
```

Библиотека pg_wait_sampling должна быть указана позже pg_stat_statements, чтобы pg_wait_sampling не перезаписывала идентификаторы запросов (queryid), которые используются pg_wait_sampling.

В расширение входят 4 функции и 3 представления:

```
\dx+ pg_wait_sampling
 function pg_wait_sampling_get_current(integer)
 function pg_wait_sampling_get_history()
 function pg_wait_sampling_get_profile()
 function pg_wait_sampling_reset_profile()
 view pg_wait_sampling_current
 view pg_wait_sampling_history
 view pg_wait_sampling_profile
```

Текущие события ожидания отображаются в представлении pg_stat_activity. Многие события ожидания длятся недолго и "поймать" их маловероятно. Расширение использует фоновый процесс pg_wait_sampling collector, который с частотой заданной параметром pg_wait_sampling.history_period или pg_wait_sampling.profile_period (по умолчанию 10 миллисекунд) опрашивает состояние всех процессов экземпляра, сохраняет pg_wait_sampling.history_size (по умолчанию 5000, максимальное значение определяется типом int4) событий в истории и группируется в "профиле" событий, доступном через представление pg_wait_sampling_profile.

История используется в кольцевом режиме: старые значения перезаписываются по кругу. Приложения могут сохранять собранную историю, запрашивая историю из представления:

```
select count(*) from pg_wait_sampling_history;
count
-----
5000
```

История событий ожидания

- расширение использует "сэмплрование" с частотой от 1 миллисекунды (по умолчанию 10 миллисекунд)
- история доступна через представление:

```
select count(*) from pg_wait_sampling_history;
count
-----
5000
```

- по умолчанию сохраняется история 5000 последних событий ожидания
- расширение также использует разделяемую память под хранение своих трёх структур:
  - очередь (MessageQueue) фиксированного размера 16Кб
  - память под список PID
  - память под идентификаторы команд (`queryid`), выполняющихся процессами

```
 select * from (select *, lead(off) over(order by off)-off as diff from pg_shmem_allocations) as a
where name like '%wait%';
        name        |    off    | size  | allocated_size | diff
--------------------+-----------+-------+----------------+-------
 pg_wait_sampling | 148145920 | 17536 |          17536 | 17536
```


История событий ожидания

Историю событий ожидания можно посмотреть через представление:

`\sv pg_wait_sampling_history`

```
CREATE OR REPLACE VIEW public.pg_wait_sampling_history AS SELECT pid, ts,
event_type, event, queryid FROM pg_wait_sampling_get_history()
pg_wait_sampling_get_history(pid, ts, event_type, event, queryid)
```

Функция `pg_wait_sampling_get_history()` выдает те же самые данные и не имеет входных параметров.

Получение данных о том, что в настоящее время выполняет процесс, с помощью опроса его состояния с какой-то частотой используется в Oracle Database ASH (Active Session History), являющейся частью AWR (Automatic Workload Repository).

На экземпляре с множеством активных сессий история на 5000 событий может перезаписываться за доли секунды. В истории сохраняются события ожиданий всех процессов. Если серверные процессы не сталкиваются с блокировками, то 99.98% событий ожидания будет заполнено фоновыми процессами и не связано с запросами. Например, при работе стандартного теста: **`pgbench -T 100`** среди 5000 событий в истории можно будет иногда увидеть одну строку:

```
select * from pg_wait_sampling_history where queryid<>0;
  pid  |             ts                | event_type |        event        |       queryid
-------+-------------------------------+------------+---------------------+---------------------
 53517 | 2035-11-11 11:18:19.676412+03 | IPC        | MessageQueueReceive | 6530354471556151986
```

Расширение также использует разделяемую память под хранение своих трёх структур:

```
select * from (select *, lead(off) over(order by off)-off as diff from
pg_shmem_allocations) as a where name like '%wait%';
        name        |    off    | size  | allocated_size | diff
--------------------+-----------+-------+----------------+-------
 pg_wait_sampling | 148145920 | 17536 |          17536 | 17536
```

Большая часть занята очередью (MessageQueue) фиксированного размера 16Кб, памятью под список PID, памятью под идентификаторы команд (`queryid`), выполняющихся процессами. Размер структуры под хранение списка PID процессов определяется максимальным числом процессов экземпляра. Число определяется параметрами конфигурации и примерно равно: `max_connections`, `autovacuum_worker_slots+1` (launcher), `max_worker_processes`, `max_wal_senders+5` (основных фоновых процессов). Память под queryid равна максимальному числу PID, умноженному на 8 байт (размер типа `bigint`, используемым `queryid`).

Расширение pg_background

- Расширение позволяет асинхронно (в фоновом режиме) выполнять произвольную команду и реализовать произвольные задачи, которые нужно выполнить приложению или администратору. Задачи будут выполняться фоновыми процессами экземпляра.
- Расширение содержит функции :
  - `pg_background_launch()`
  - `pg_background_result()`
  - `pg_background_detach()`

Расширение pg_background

Расширение pg_background имеется в Tantor Postgres.

Расширение позволяет выполнять асинхронно (в фоновом режиме) произвольные операции. С помощью расширения можно вручную реализовать произвольные задачи, которые нужно выполнять в фоновом режиме приложению или администратору. Задачи будут выполняться фоновыми процессами экземпляра. Расширение предоставляет программный интерфейс для запуска и взаимодействия с фоновыми процессами, устраняя необходимость использования низкоуровневого интерфейса взаимодействия с процессами, требующего программирования на языке C.

Расширение содержит функции:

pg_background_launch - принимает SQL-команду, которую пользователь хочет выполнить, и размер буфера очереди. Эта функция возвращает идентификатор фонового рабочего процесса.

pg_background_result - принимает идентификатор процесса в качестве входного параметра и возвращает результат выполненной команды через фоновый рабочий процесс.

pg_background_detach - принимает идентификатор процесса и отсоединяет фоновый процесс, который ожидает, чтобы пользователь прочитал его результаты.

Расширения pgaudit и pgaudittofile

- расширениями `pgaudit` и `pgauditlogtofile` можно направить сообщения о создании сессий и их длительности в отдельный файл или файлы аудита
- для работы расширений нужно загрузить их библиотеки:

```
alter system set shared_preload_libraries = pgaudit, pgauditlogtofile;
```

- Расширения работают независимо и параллельно с журналом кластера и управляются собственными параметрами, которые имеют префикс "`pgaudit.`"
- параметры `pgaudit.log_connections` и `pgaudit.log_disconnections` аналогичны одноимённым параметрам PostgreSQL и могут создавать аналогичные записи **в отдельном файле аудита**

Расширения pgaudit и pgaudittofile

При использовании параметров `log_connections` и `log_disconnections` сообщения записываются в журнал кластера. При промышленной эксплуатации, в журнал записывается множество сообщений. Логирование соединений не нужно для повседневного анализа и замусоривает общий журнал, затрудняя чтение более важных сообщений и желательно, чтобы логирование соединений, команд ddl и других команд выполнялось в не в журнал кластера, а в отдельный файл или файлы.

В СУБД Tantor есть расширения `pgaudit` и `pgauditlogtofile`, которыми можно направить сообщения о создании сессий и их длительности в отдельный файл или файлы аудита. Расширение `pgauditlogtofile` перенаправляет записи, создаваемые расширением `pgaudit` в отдельный файл или файлы. Без него записи попадают в журнал кластера. Расширение `pgauditlogtofile` зависит от расширения `pgaudit` и не работает без него. Для использования расширений достаточно загрузить две библиотеки:

```
alter system set shared_preload_libraries = pgaudit, pgauditlogtofile;
```

Библиотеки расширений регистрируют в экземпляре параметры конфигурации, которыми можно настроить что и куда будет логироваться. Расширения работают независимо и параллельно с журналом кластера и управляются собственными параметрами, которые имеют префикс "`pgaudit.`"

У расширения 18 параметров. 7 параметров относятся к библиотеке `pgauditlogtofile`, в том числе параметры `pgaudit.log_connections` и `pgaudit.log_disconnections`. Эти параметры аналогичны одноимённым параметрам PostgreSQL и могут создавать аналогичные записи, но только **в отдельном файле аудита, а не в журнале кластера, в чём большое преимущество этих параметров**. Преимущество перевешивает недостатки в виде необходимости загрузки двух библиотек и неудобств их использования. Параметры библиотеки устанавливаются только на уровне кластера, указание этих параметров в переменной окружения приводит к ошибке и невозможности соединиться, в отличие от стандартных параметров: **`export PGOPTIONS="-c pgaudit.log_connections=off"`**

```
psql
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL:  parameter "pgaudit.log_connections" cannot be changed now
```

Параметр `pgaudit.log_disconnections` , в отличие от параметра `log_disconnections` не может быть установлен при создании сессии.

Конфигурирование расширений pgaudit и pgaudittofile

- семь параметров относятся к библиотеке `pgauditlogtofile`
- чтобы создавался журнал аудита нужно установить параметр `pgaudit.log` как минимум в значение `'misc'`
  - `'none'` - файл журнала аудита не создаётся
  - `'role'` и `'ddl'` `pgaudit.log_connections` и `pgaudit.log_disconnections` не действуют

```
postgres=# \dconfig pgaudi*
       List of configuration parameters
            Parameter             |    Value
----------------------------------+----------
 pgaudit.log                      | none
 pgaudit.log_autoclose_minutes    | 0
 pgaudit.log_catalog              | on
 pgaudit.log_client               | off
 pgaudit.log_connections          | off
 pgaudit.log_directory            | log
 pgaudit.log_disconnections       | off
 pgaudit.log_filename             | audit-%F.log
 pgaudit.log_level                | log
 pgaudit.log_parameter            | off
 pgaudit.log_parameter_max_size   | 0
 pgaudit.log_relation             | off
 pgaudit.log_rotation_age         | 1d
 pgaudit.log_rotation_size        | 0
 pgaudit.log_rows                 | off
 pgaudit.log_statement            | on
 pgaudit.log_statement_once       | off
 pgaudit.role                     |
(18 rows)
```


Конфигурирование расширений pgaudit и pgaudittofile

Неудобство использования параметров расширений в том, что нужно установить параметр `pgaudit.log` как минимум в значение `'misc'`, чтобы создавался журнал аудита. Но значение `'misc'` заставляет логировать команды DISCARD, FETCH, CHECKPOINT, VACUUM, SET и раздувает журнал аудита. При значении по умолчанию `'none'` файл журнала не создаётся. При установке в значения `'role'` и `'ddl'` параметры `pgaudit.log_connections` и `pgaudit.log_disconnections` не действуют.

Установка расширения pgauditlogtoile командой бесполезна, так как в расширении нет обьектов:

```
create extension pgauditlogtofile;
\dx+ pgauditlogtofile
Objects in extension "pgauditlogtofile"
--------------------
(0 rows)
```

В расширение `pgaudit` входит два триггера и две триггерные функции:

```
event trigger pgaudit_ddl_command_end
event trigger pgaudit_sql_drop
function pgaudit_ddl_command_end()
function pgaudit_sql_drop()
```

Переменная подмены `'%F'` (или её эквивалент `%Y-%m-%d`) в названии журнала аудита и журнала кластера удобнее значения по умолчанию (`%Y%m%d_%H%M`) тем, что не создаёт отдельный файл при рестарте экземпляра. Новый файл создаётся раз в сутки. Пример установки значений:

```
alter system set pgaudit.log_filename = 'audit-%F.log';
alter system set log_filename = 'postgresql-%F.log';
```

Утилита pgcopydb

- Утилита автоматизирует копирование базы данных на другой кластер
- Типичный случай использования pgcopydb - миграция на новую основную версию PostgreSQL с минимизацией простоя
- Утилита реализует задачу распараллеливания с поточной передачей данных по логике "pg_dump -jN | pg_restore -jN" между двумя работающими кластерами, дирижируя этими утилитами
- Поддерживает параллельное создание индексов, отслеживание изменений и их применение, возобновление прерванной перегрузки, фильтрацию объектов
- Проект с открытым исходным кодом

Утилита pgcopydb

Утилита автоматизирует копирование базы данных на другой кластер. Типичный случай использования pgcopydb - миграция на новую основную версию PostgreSQL с минимизацией простоя. Утилита реализует задачу распараллеливания с поточной передачей данных по логике "pg_dump -jN | pg_restore -jN" между двумя работающими кластерами, дирижируя этими утилитами. Поддерживает параллельное создание индексов, отслеживание изменений и их применение, возобновление прерванной перегрузки, фильтрацию объектов.

pgcopydb - проект с открытым исходным кодом https://github.com/dimitri/pgcopydb
https://docs.tantorlabs.ru/tdb/ru/18_1/se/pgcopydb.html

Утилита pg_anon

Программа, написанная на языке python выполняет:

- поиск конфиденциальных данных в базе данных
- создание словаря на основе результатов поиска
- сохранение и восстановление с использованием словаря
- синхронизация содержимого или структуры указанных таблиц между исходной и целевой базами данных
- выгрузка данных с маскировкой (обезличиванием, анонимизацией) по заданным шаблонам

Утилита pg_anon

pg_anon - приложение, написанное на языке python.
Приложение выполняет:
Создание в базах данных схемы `anon_funcs`, которая содержит набор функций для маскировки (обезличивания, анонимизации) данных.
Поиск конфиденциальных данных в базе данных на основе словаря.
Создание словаря на основе результатов поиска (рекогносцировка).
Сохранение и восстановление с использованием словаря. Для разных баз данных можно предоставить отдельные файлы словаря.
Синхронизация содержимого или структуры указанных таблиц между исходной и целевой базами данных.
Приложение скачивается и устанавливается отдельно из пакета.
https://github.com/TantorLabs/pg_anon
Также есть расширение **transp_anon** - прозрачная анонимизация значений на лету результатов запросов от клиентов
Платформа Tantor имеет удобный и интуитивно понятный графический интерфейс к приложению pg_anon.
https://docs.tantorlabs.ru/tdb/ru/18_1/se/transp_anon.html

Утилита pg_configurator

- Представляет собой скрипт pg_configurator на языке python, устанавливаемый по пути /usr/bin
- предлагает рекомендуемые параметры конфигурации на основе характеристик аппаратных ресурсов, таких как доступная память, количество процессоров и дисковое пространство и т.д. Это позволяет оптимально использовать имеющиеся ресурсы и увеличить производительность экземпляра

Утилита pg_configurator

Приложение pg_configurator доступно для всех сборок СУБД Tantor Postgres. Приложение поставляется отдельно, виде пакета.

Представляет собой скрипт pg_configurator на языке python, устанавливаемый в директорию `/usr/bin`.

pg_configurator предлагает рекомендуемые параметры конфигурации на основе характеристик аппаратных ресурсов, таких как доступная память, количество процессоров и дисковое пространство и т.д. Оптимальные значения параметров конфигурации позволяют эффективно использовать имеющиеся аппаратные ресурсы.

Страница проекта: https://github.com/TantorLabs/pg_configurator

Веб-версия конфигуратора: https://tantorlabs.ru/pgconfigurator

Утилита pg_diag_setup.py

- Задача, решаемая утилитой: вызвать утилиту на хосте кластера баз данных, которая установит и настроит параметры расширений в соответствии с шаблоном и зарезервирует значения параметров, чтобы можно было восстановить изменения
- Проверяет доступность расширений через `pg_available_extensions`
- Обновляет значение `shared_preload_libraries` без перезаписи существующих библиотек
- Добавляет новые параметры в конец `postgresql.conf`
- Создает текстовый файл бэкапа со значениями конфигурируемых параметров с временной меткой
- Позволяет откатить изменения к любому бэкапу

Утилита pg_diag_setup.py

Представляет собой скрипт, написанный на языке python. Задача, решаемая утилитой: вызвать утилиту на хосте кластера баз данных, которая установит и настроит параметры расширений, в соответствии с шаблоном, и зарезервирует значения параметров, чтобы можно было восстановить изменения. Предполагается, что будут настраиваться параметры диагностических расширений типа pg_store_plans, pg_stat_statements, pg_stat_kcache, auto_explain, pg_buffercache, pg_trace, pg_wait_sampling.

Утилита не рестартует экземпляр после внесения изменений.

При запуске утилита:

1) читает файлы параметров конфигурации, с учетом параметров `include*`

2) создает список параметров с указанием файла-источника

3) читает свой файл параметров `default.yaml`, в котором указаны конфигурируемые расширения и их параметры конфигурации. Пример содержимого файла для расширения

```
pg_stat_statements:
  shared_preload_lib: pg_stat_statements
  create_cmd: CREATE EXTENSION pg_stat_statements
  params:
    pg_stat_statements.max: 10000
    pg_stat_statements.track: all
    pg_stat_statements.track_utility: "on"
    pg_stat_statements.track_planning: "off"
    pg_stat_statements.save: "on"
```

4) Проверяет доступность расширений через pg_available_extensions, подсоединившись к экземпляру через Unix-сокет;

5) Обновляет значение `shared_preload_libraries` командой ALTER SYSTEM без перезаписи существующих библиотек, устанавливает расширения (если удастся)

6) Добавляет новые параметры в конец `postgresql.conf`, помечая добавленные параметры комментарием "Added by pg_diag_setup"

8) Создает текстовый файл бэкапа со значениями конфигурируемых параметров с временной меткой

9) Позволяет откатить изменения к любому созданному утилитой бэкапу

Утилита pg_sec_check

- позволяет автоматизировать процесс проверки параметров безопасности, от настроек операционной системы до параметров конфигурации PostgreSQL
- создаёт отчеты о выявленных проблемах и рекомендации по их устранению
- проверки выполняются скриптами `.sql` и `.sh`
- результаты проверок валидируются скриптами на языке **Lua**, ими же формируются отчеты и рекомендации
- проверяет целостность своих файлов контрольными суммами
- В утилите имеется 68 проверок
- можно создавать собственные проверки путем написания скриптов `.sql .sh .lua`

Утилита pg_sec_check

Postgres Security Check - утилита предназначена для проведения аудита безопасности конфигураций баз данных PostgreSQL. Утилита позволяет автоматизировать процесс проверки параметров безопасности, от настроек операционной системы до параметров конфигурации PostgreSQL. По результатам проверок создаёт отчеты о выявленных проблемах и рекомендации по их устранению.

Возможность привязки проверок к версиям PostgreSQL (минимальная и максимальная поддерживаемые версии). Создает отчеты в форматах HTML, JSON на русском и английском языках. Контролирует целостность своих файлов с помощью проверки контрольных сумм.

Проверки выполняются (execute): скриптами .sql и .sh

Результаты проверок валидируются (validate) скриптами на языке Lua, ими же формируются отчеты и рекомендации.

Файл конфигурации утилиты тоже текстовый, в формате .json

В утилите имеется 68 проверок, позволяющие выявить типичные ошибки. Проверки описаны в виде скриптов, которые можно редактировать.

На примере поставляемых с утилитой скриптов можно создавать собственные проверки, для новых проверок нужно написать скрипты `.sql .sh .lua`

Написанна на языке Rust.

https://docs.tantorlabs.ru/tdb/ru/18_1/se/pg_sec_check.html

Утилита WAL-G (Write-Ahead Log Guard)

- утилита для создания шифрованных резервных копий кластера баз данных (полных и инкрементальных) и архивирования WAL-сегментов, их высокоэффективной отправки/получения по протоколу S3 "из" и "в" хранилища (облачные в сети предприятия или внешние) напрямую без создания промежуточных файлов в файловой системе.
- Используя утилиту WAl-G можно:
  - создавать резервные копии кластера и WAL-сегментов
  - восстанавливать кластер на выбранный момент времени в прошлом
  - управлять резервными копиями по протоколу S3: удалять ненужные бэкапы и связанные с ними файлы журналов

Утилита WAL-G (Write-Ahead Log Guard)

WAL-G (Write-Ahead Log Guard) - утилита командной строки для создания шифрованных резервных копий кластера баз данных и архивирования WAL-файлов, их эффективной отправки/получения в несколько потоков (с максимальной скоростью и минимальной нагрузке на процессора и память) по протоколу S3 "из" и "в" хранилища (облачные в сети предприятия или внешние) напрямую, без создания промежуточных файлов в файловой системе хоста. WAL-G разработана для эффективного резервирования использования WAL-сегментов, но также способна создавать резервные копии PGDATA кластера.

Утилита поставляется в пакетах deb или rpm. В пакете содержится единственный исполняемый файл WAL-G, который копируется в стандартную директорию с исполняемыми файлами `/opt/tantor/usr/bin`.

Пример установки параметров конфигурации для резервирования WAL-сегментов:

```
ALTER SYSTEM SET archive_command='wal-g wal-push "%p" >> ~/archive-command.log 2>&1';
ALTER SYSTEM SET restore_command='wal-g wal-fetch "%f" "%p" >> ~/restore_command.log 2>&1';
ALTER SYSTEM SET archive_mode=on;
```

Пример команды резервирования PGDATA:

```
wal-g backup-push $PGDATA >> ~/backup-push.log 2>&1
```

Пример команды восстановления из бэкапа (экземпляр должен быть остановлен):

```
wal-g backup-fetch $PGDATA LATEST
touch $PGDATA/recovery.signal
```

WAL-G может:

1) создавать резервные копии кластера и WAL-сегментов в режиме "push". Текущий WAL-сегмент не резервируется и утилита не может использоваться как единственное решение по обеспечению высокой доступности.

2) восстанавливать кластер на выбранный момент времени в прошлом. Из хранилища возможно восстановить WAL-сегменты кроме текущего (в который писали процессы экземпляра в момент остановки кластера). **Полное восстановление (без потери транзакций) возможно только, если текущий WAL-сегмент не был потерян**

3) управлять резервными копиями по протоколу S3: удалять бэкапы и связанные с ними файлы журналов

4) шифровать файлы перед передачей их в хранилище

Другие расширения

- dbcopies_decoding - библиотека 1С, предоставляет слоты логической репликации при копировании баз данных баз данных 1С
- vector - полная поддержка типа данных векторов большой размерности
- pg_partman - автоматизация поддержки секционированных таблиц
- pg_qualstats - ведёт статистику по предикатам, найденным в операторах WHERE и в предложениях JOIN
- pg_hint_plan - подсказки оптимизатору в запросах
- plantuner скрывает индексы от планировщика
- pg_cron - планировщик внутри экземпляра
- pg_throttle - ограничение объема читаемых строк в запросах, позволяющее снизить конкуренцию за ввод-вывод
- pg_trace - трассировки работающих запросов, может быть полезна для анализа запросов в 1С
- pg_ddl_deploy - расширение для логической репликации
- pqg - очередь в базе данных

Другие расширения

Описание расширений, которые не рассматривались:

dbcopies_decoding - библиотека 1С, предоставляет слоты логической репликации при копировании баз данных баз данных 1С

vector - полная поддержка типа данных векторов большой размерности: функции, операторы, индексная поддержка. Свободно распространяемый проект https://github.com/pgvector/pgvector

pg_partman - автоматизация поддержки секционированных таблиц https://github.com/pgpartman/pg_partman

pg_qualstats - ведёт статистику по предикатам, найденным в операторах WHERE и в предложениях JOIN https://github.com/powa-team/pg_qualstats

pg_hint_plan - подсказки оптимизатору в запросах https://github.com/ossc-db/pg_hint_plan

plantuner скрывает индексы от планировщика https://github.com/postgrespro/plantuner

pg_cron - планировщик внутри экземпляра

pg_throttle - ограничение объема читаемых строк в запросах, позволяющее снизить конкуренцию за ввод-вывод

pg_trace - трассировка работающих запросов SQL. Для получения трассировки нужен клиент, который подключится на порт фонового процесса и будет получать отладочную информацию в формате json. Пример использования для анализа запросов в 1С и пример клиента https://habr.com/ru/companies/tantor/articles/915256/

pg_ddl_deploy - расширение для логической репликации, реализующее захват команд DDL триггерами и репликацию DDL команд https://github.com/enova/pgl_ddl_deploy

pgq - очередь в базе данных от Skype. Обработчики (потребители) сообщений могут быть написаны на python и java https://github.com/pgq/pgq

Практика

1. Расширение orafce
2. Расширение pg_variables
3. Расширение page_repair
   1. Подготовка реплики
   2. Подготовка таблицы
   3. Восстановление страницы с помощью page_repair
   4. Обнуление страницы
4. Отладка подпрограмм
   1. Установка расширения из исходных кодов на примере pldebugger
   2. Отладка функции в pgAdmin
   3. Отладка подпрограмм в DBeaver
5. Обработка строк большого размера StringBuffer
6. Поиск осиротевших файлов

Практика

1.Расширение orafce
2.Расширение pg_variables
3.Расширение page_repair
4.Отладка подпрограмм
5. Обработка строк большого размера StringBuffer
6. Поиск осиротевших файлов

Практические задания к этой главе опциональны и выполняются, если остаётся время.