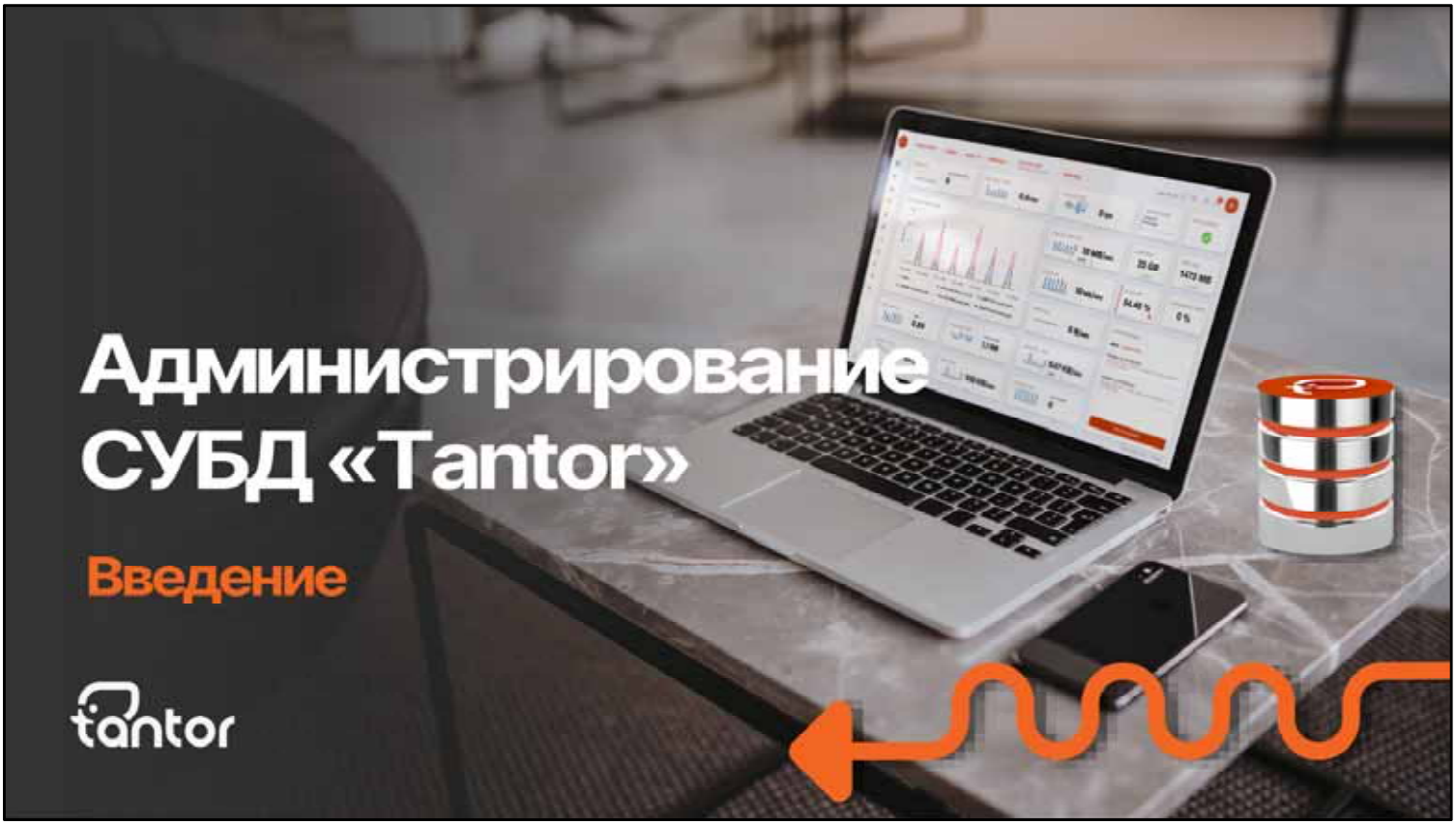
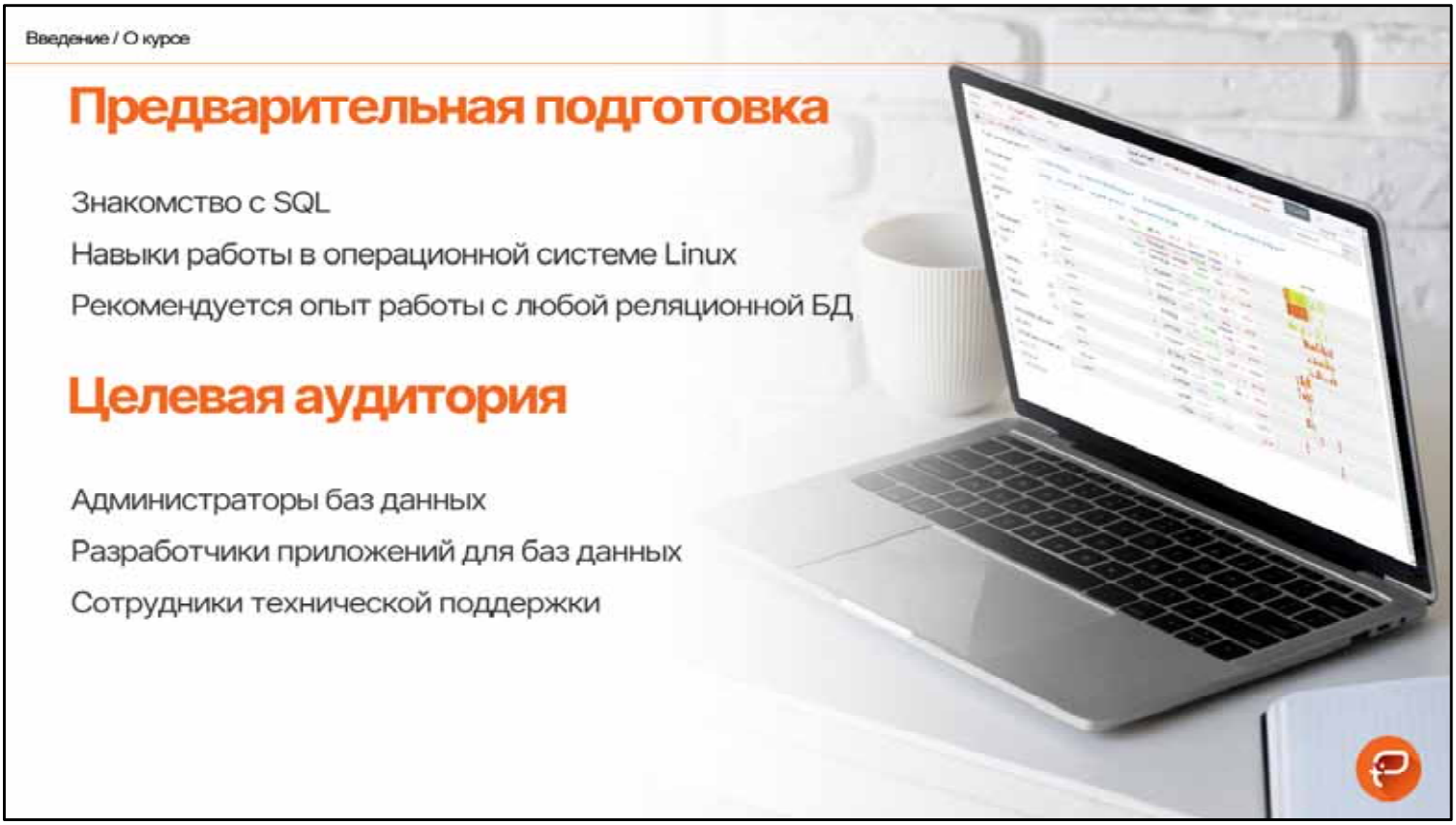

Администрирование
СУБД «Tantor»
Введение
tantor

Темы

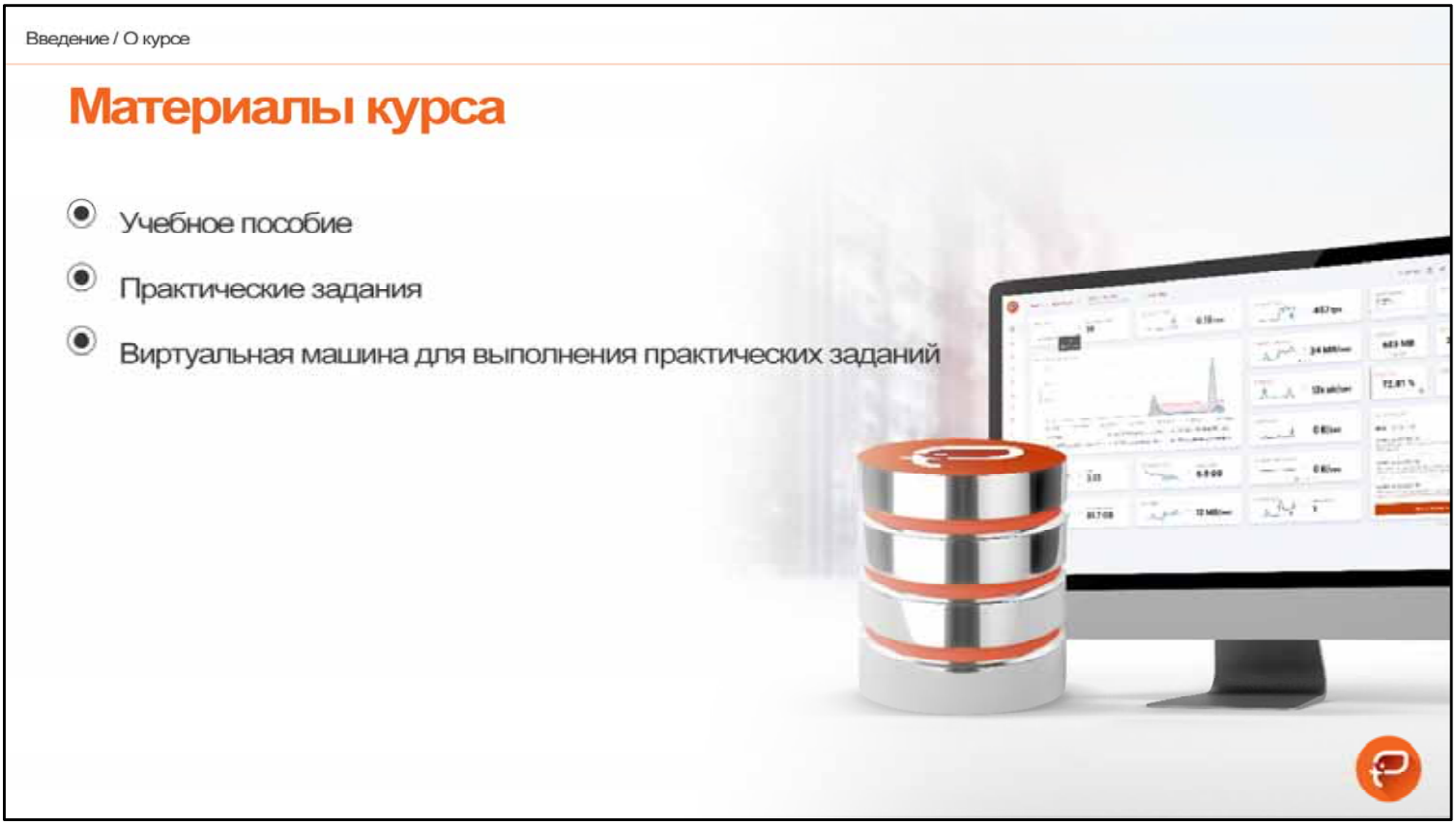
- О курсе
- Компания «Tantor Labs»
- СУБД «Tantor». Редакции



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`, `vi`; понимать и менять разрешения на файлы (команды `ls -al`, `chmod`, `chown`).

В курсе будут рассмотрены основные задачи администрирования баз данных семейства PostgreSQL на примере СУБД «Tantor» и особенности эксплуатации СУБД «Tantor».

Для успешного прохождения курса рекомендуется слушать инструктора, если будут возникать вопросы задавать их, читать текст практических заданий и самостоятельно их выполнять. При выполнении практических заданий рекомендуется набирать команды на клавиатуре, а не копировать в терминал из текста заданий. Ручной ввод команд, исправление опечаток, возникающих при наборе команд, изучение ошибок выдаваемых на неправильные команды позволяет лучше запомнить команды и смысл их использования. Ощущения «понимания» текста заданий недостаточно, при реальной работе важно вспомнить основные ключевые слова и возможности команд, чтобы быстро найти полный синтаксис. Копирование команд можно использовать, если вы с ними знакомы.



К материалам курса относятся:

1) Учебное пособие в форме книги в формате pdf, которое содержит теоретическую часть курса.

2) Практические задания в форме книги в формате pdf.

3) Виртуальная машина с установленной операционной системой Astra Linux и СУБД Tantor SE. 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.

После прохождения курса могут быть предложены вопросы для проверки знаний / итоговое тестирование.

Посмотрим из чего будет состоять наш курс:

- **Раздел 1. Установка и управление СУБД**
  - Установка СУБД Тантор
  - Управление экземпляром кластера баз данных
  - Утилиты управления кластером баз данных
  - Терминальный клиент
  - Графическая утилита pgAdmin
- **Архитектура**
  - Общие сведения
  - Структуры памяти
  - Многоверсионность
  - Регламентные работы
  - Выполнение запросов
  - Расширяемость
- **Конфигурирование**
- **Базы данных**
  - Логическая и Физическая реализация
- **Журналирование**
  - Диагностический журнал
- **Безопасность**
  - Модель безопасности, подключение и аутентификация
- **Резервное копирование**
  - Физическое и логическое резервирование
- **Репликация**
  - Физическая и логическая
- **Платформа «Tantor», обзор возможностей**
- **Расширения, программы, улучшения в ядре**

Длительность курса 5 дней.

Начало курса 10:00, окончание до 17:00. Перерыв на обед в 13:00-14:00. Перерывы на обед обычно начинаются по окончанию главы и совмещаются с практическим задание после главы, поэтому начало обеденного времени может сдвигаться на полчаса раньше или позже. В последний день курса теоретическая часть обычно заканчивается до 15:00. Время с 15:00 до 17:00 предназначено на завершение практик.

Курс разбит на главы (разделы), демонстрации от инструктора, практики.

Длительность глав от 20 до 40 минут. После большинства глав есть демонстрация, выполняемая инструктором. После демонстрации перерыв и время на самостоятельное выполнение практических заданий.

Время начала следующей главы или длительность перерыва, если он сделан в середине главы, устанавливает инструктор.

Перерывы совмещаются с практическими заданиями. Если при выполнении практических заданий возникают затруднения и самостоятельно устранить их не удаётся в течение 5-10 минут можно обратиться к инструктору.

Обычно практические задания выполняются после соответствующей главы. Для усвоения материала курса порядок выполнения задание до соответствующей ему главы или после — не важен. Если вы не успеваете выполнить задание за отведённое до следующей главы время, можно продолжить выполнение задания в следующем перерыве на практику. Если вы делаете задания быстро, то можно их выполнять до начала главы. Обычно в конце каждого дня, отводится время на выполнение практики и те, кто выполнил практику быстрее могут заканчивать обучение в этот день раньше.

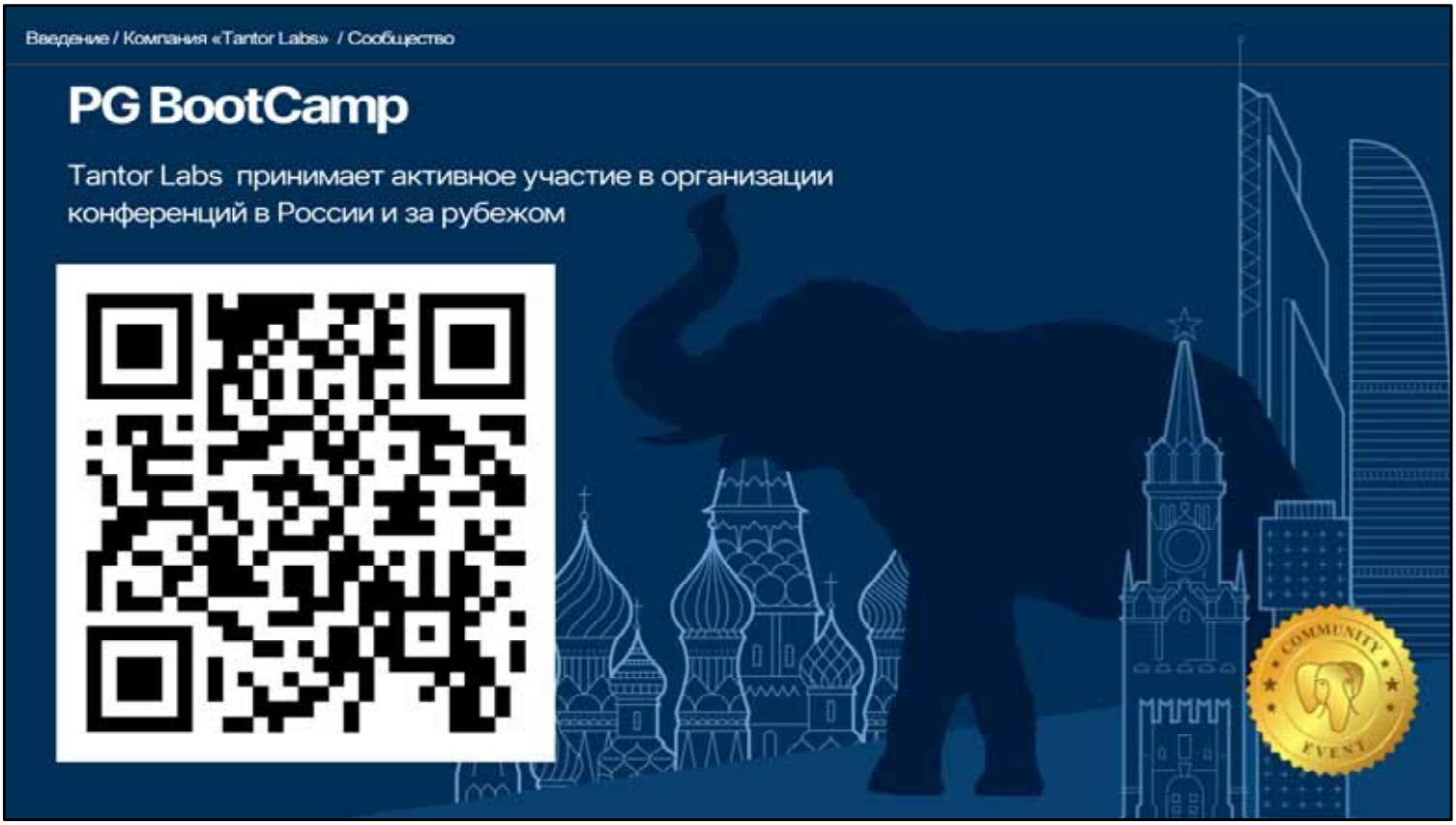
Темы

- О курсе
- Компания «Tantor Labs»
- СУБД «Tantor»: Редакции

«Тантор Лабс» в 2016 году стала работать на международном рынке профессиональных услуг по поддержке СУБД PostgreSQL.

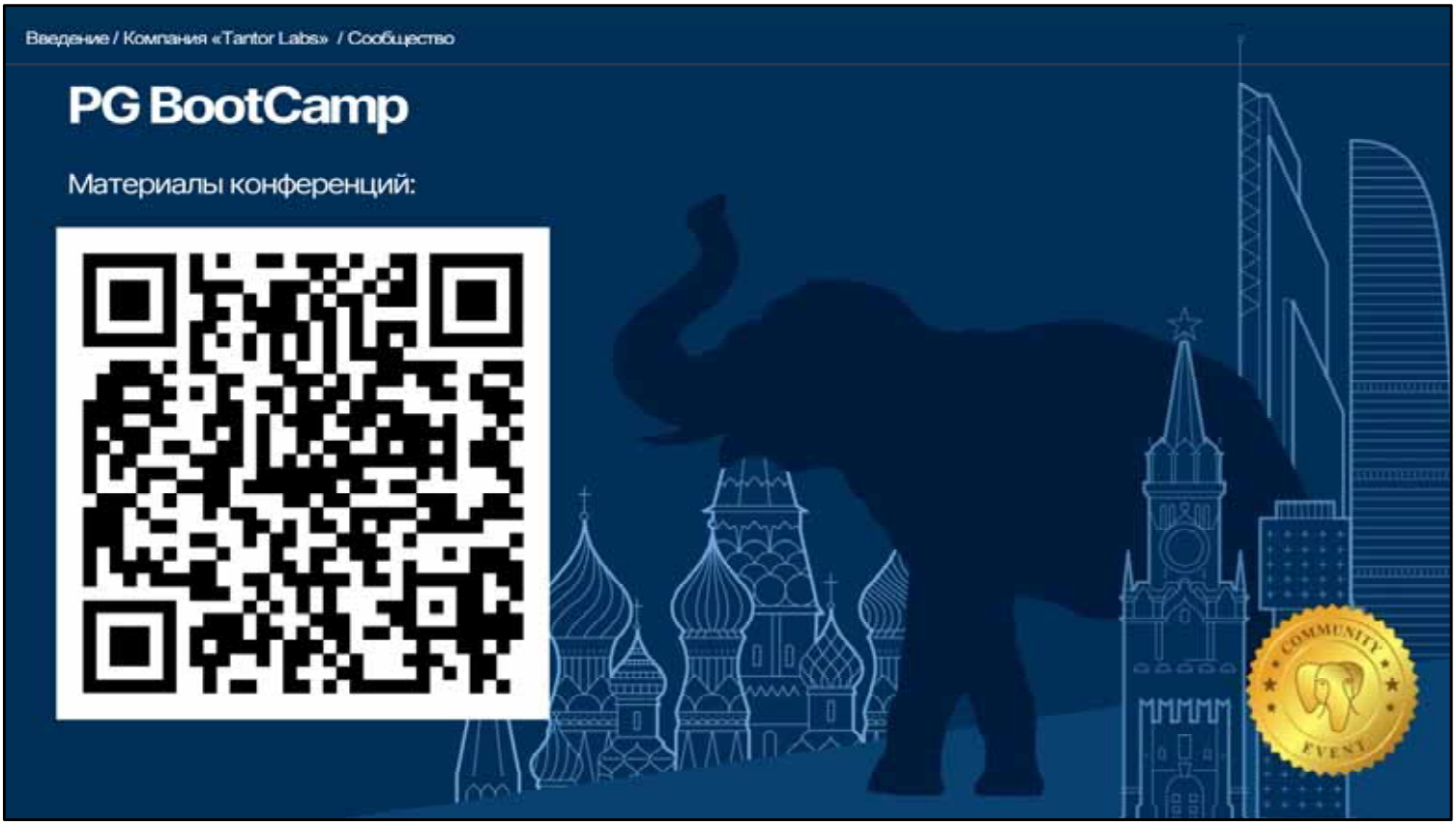
С 2021 года компания переключила свое внимание на российский рынок. «Тантор Лабс» активно участвует в сообществе PostgreSQL как в России, так и за рубежом.

С 2022 года компания Тантор входит в ПАО Группа Астра.



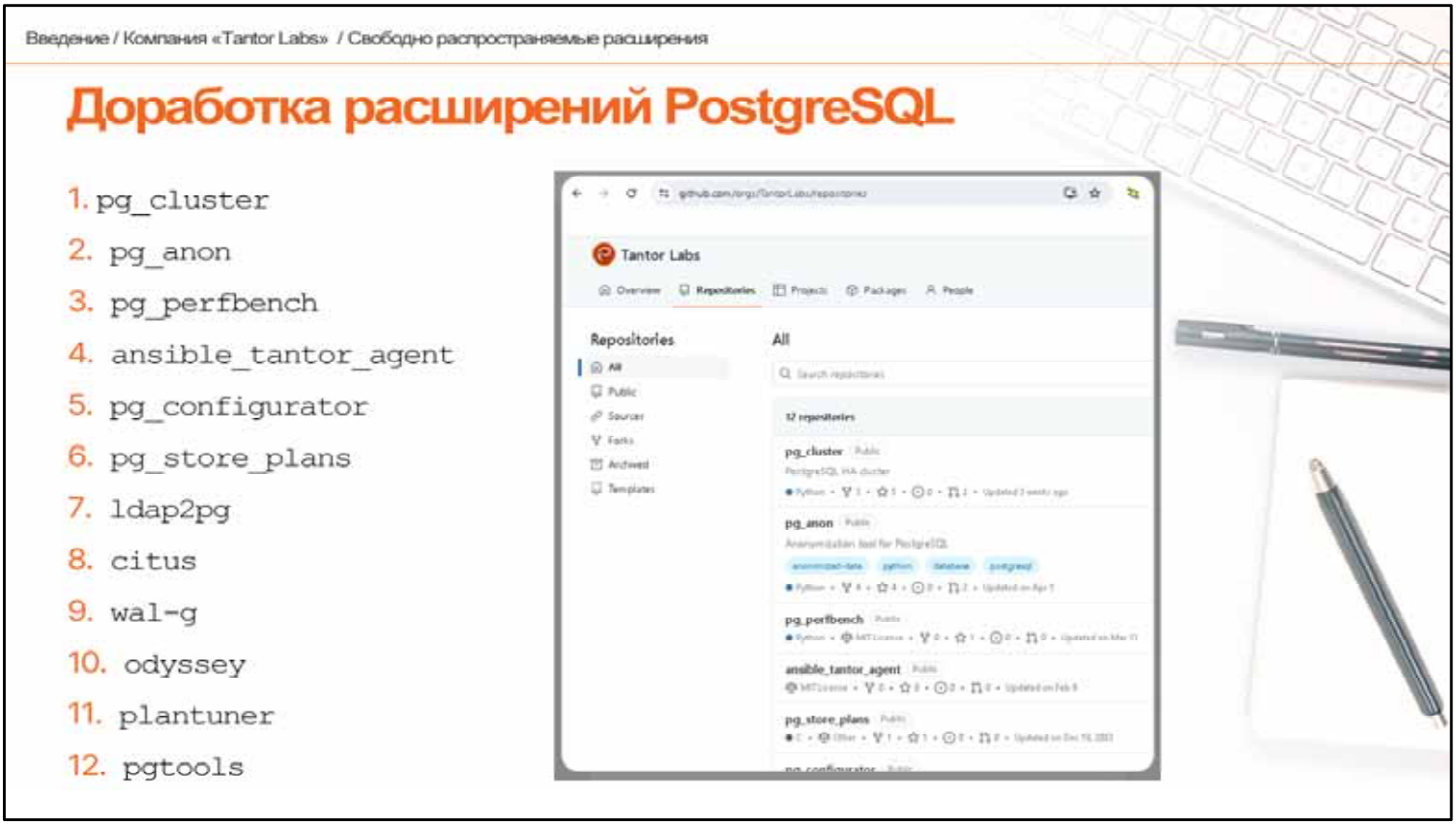
Тантор Лабс является активным участником организации конференций сообщества PostgreSQL в рамках глобальной инициативы PG BootCamp.

Заявить участие в конференции онлайн и оффлайн: https://pgbootcamp.ru/



Скачать материалы конференций: https://github.com/PGBootCamp

Посмотреть выступления https://www.youtube.com/@PGBootCampRussia



Сотрудники Тантор Лабс дорабатывают и создают расширения для СУБД PostgreSQL.

Репозитории расширений: https://github.com/orgs/TantorLabs

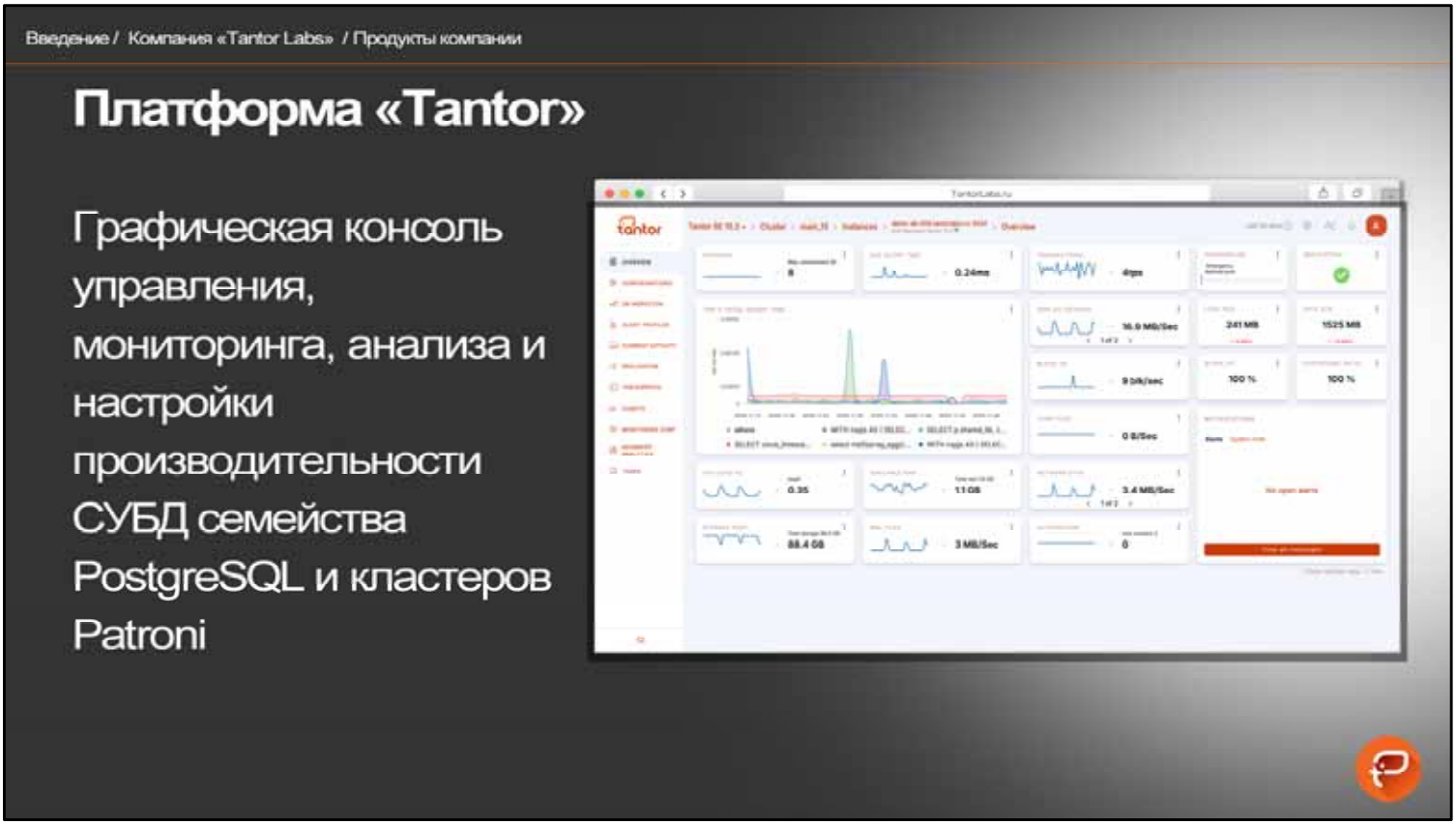
Список расширений:

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

Тантор Лабс производит СУБД «Tantor» и Платформу «Tantor», осуществляет их техническую поддержку.

СУБД «Tantor» оптимизирована для высоконагруженных корпоративных систем обеспечивает высокую производительность, надежность и отказоустойчивость.

СУБД «Tantor» включена в «Единый реестр российских программ для электронных вычислительных машин и баз данных».



Платформа «Tantor» — графическая консоль управления, мониторинга, анализа и настройки производительности СУБД семейства PostgreSQL и кластеров Patroni.

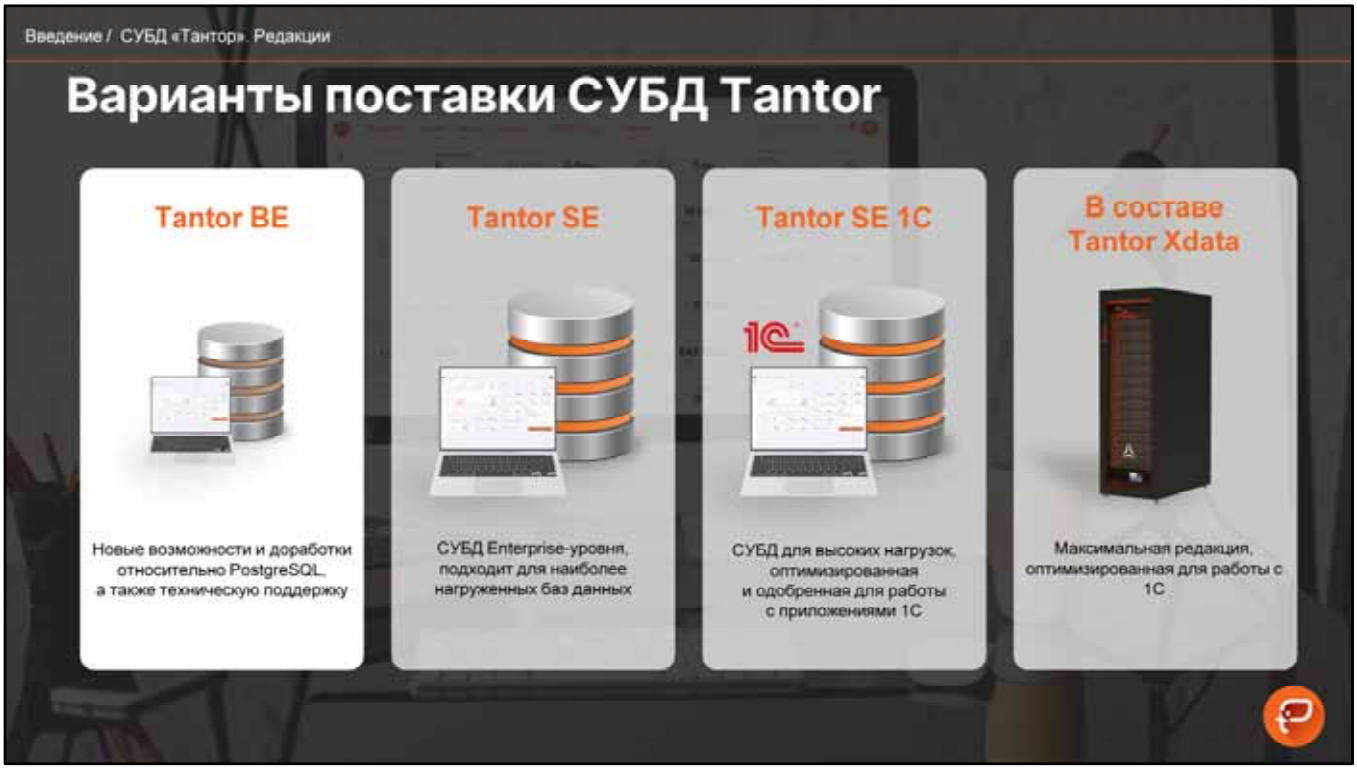
Включена в «Единый реестр российских программ для электронных вычислительных машин и баз данных». https://tantorlabs.ru/products/platform

Tantor XData — программно-аппаратный комплекс имеющий в своём составе Astra Linux Special Edition, СУБД «Tantor», Платформу «Tantor», программы обеспечения функционирования (Tantor Deploy Manager, Tantor Appliance Manager, Backman).

https://tantorlabs.ru/products/xdata

Темы

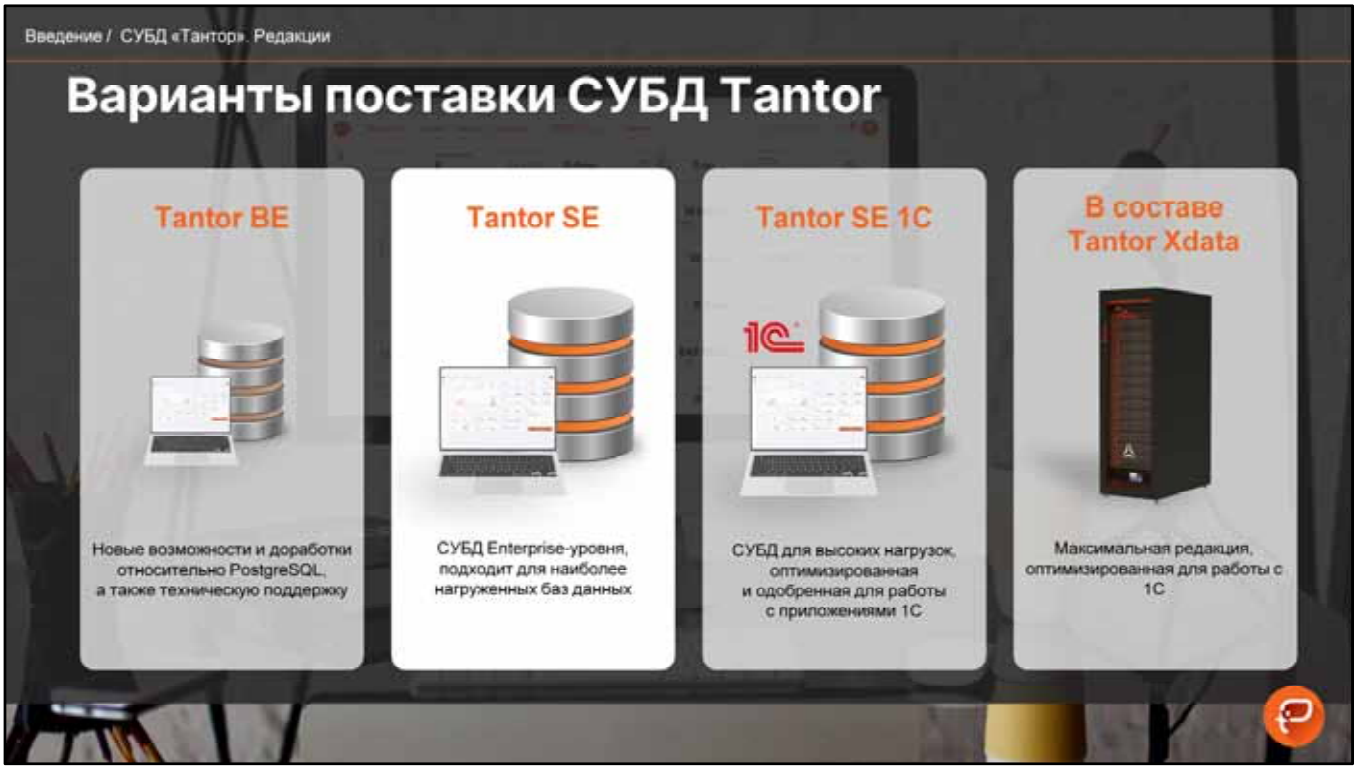
- О курсе
- Компания «Tantor Labs»
- СУБД «Tantor»: Редакции



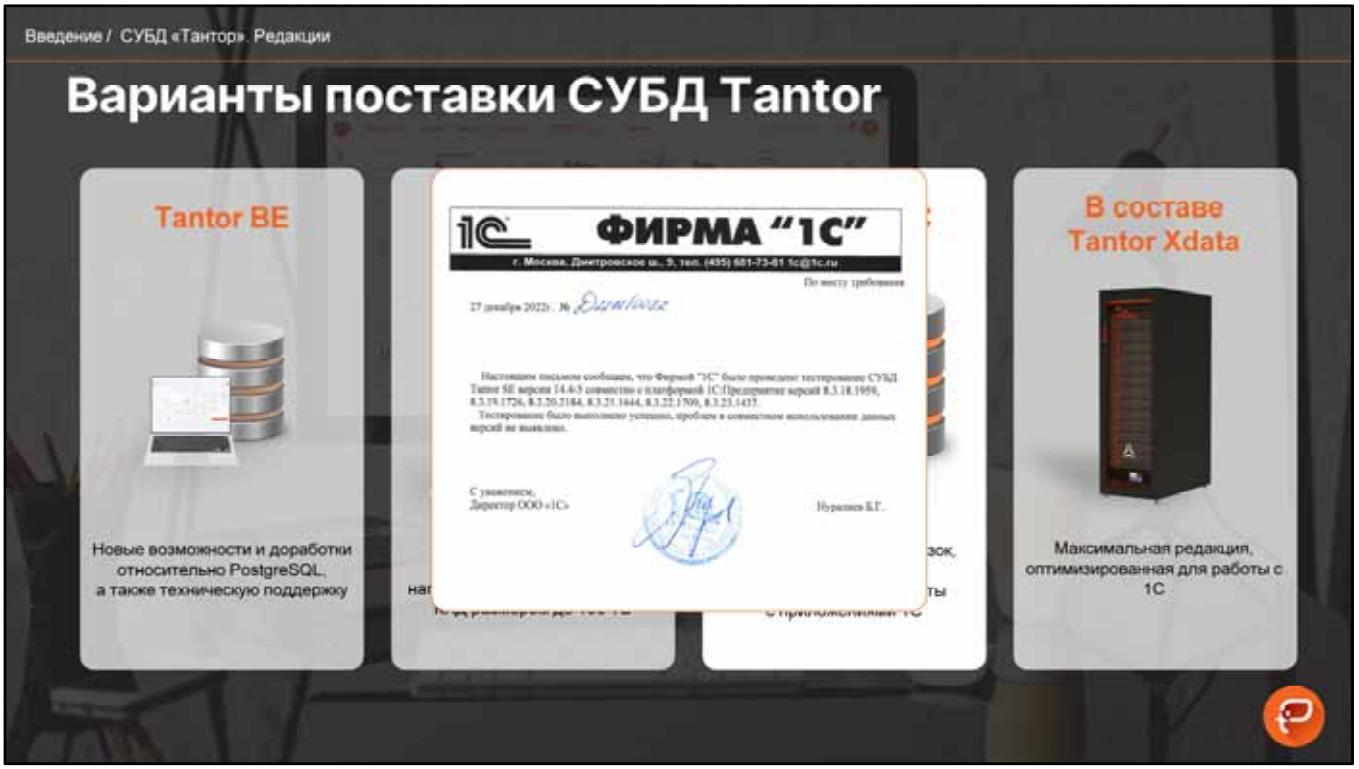
Варианты поставки СУБД Tantor (editions, редакции, сборки) спроектированы таким образом, чтобы покрыть весь спектр потребностей по хранению и обработке данных:

- Basic Edition включает в базовые улучшения по сравнению с PostgreSQL соответствующей версии, возможность получения технической поддержки Тантор Лабс и использования Платформы «Tantor» с СУБД «Tantor».

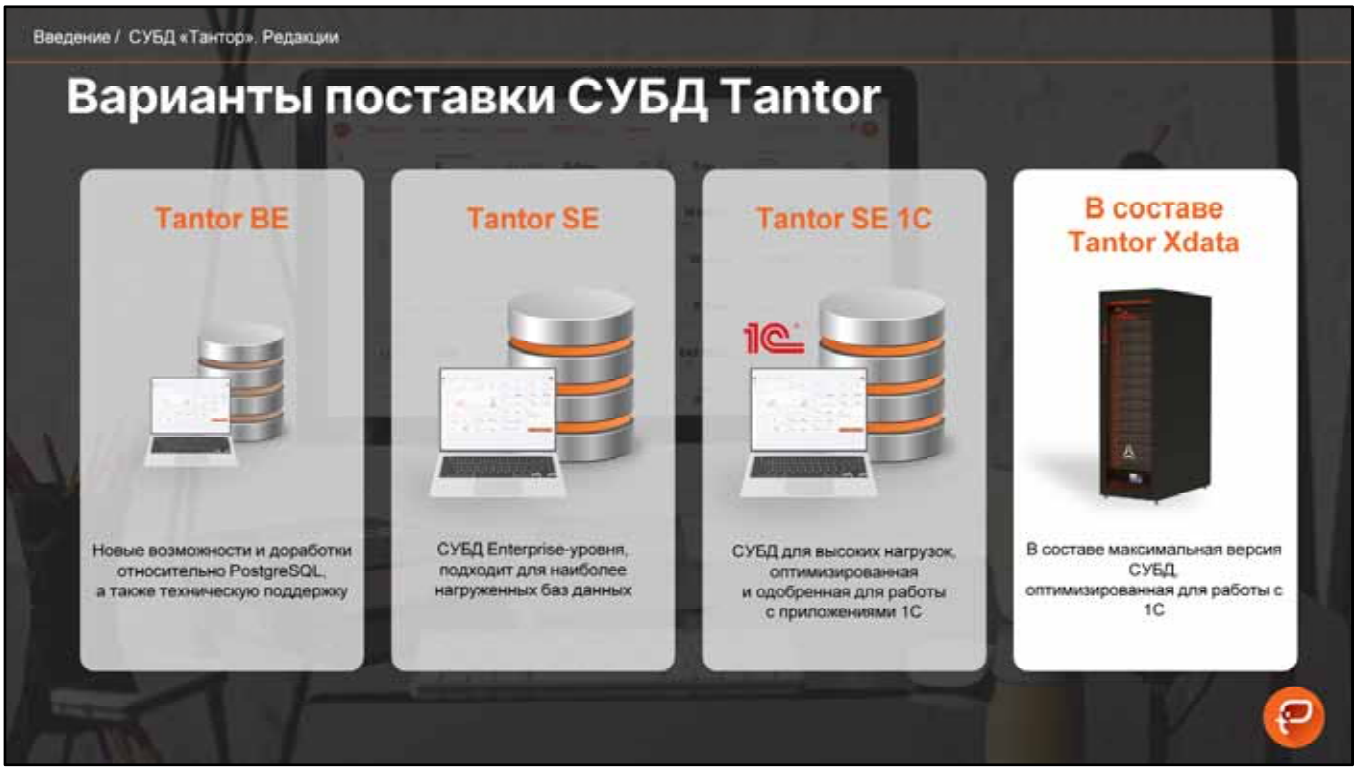
https://tantorlabs.ru/products/compare



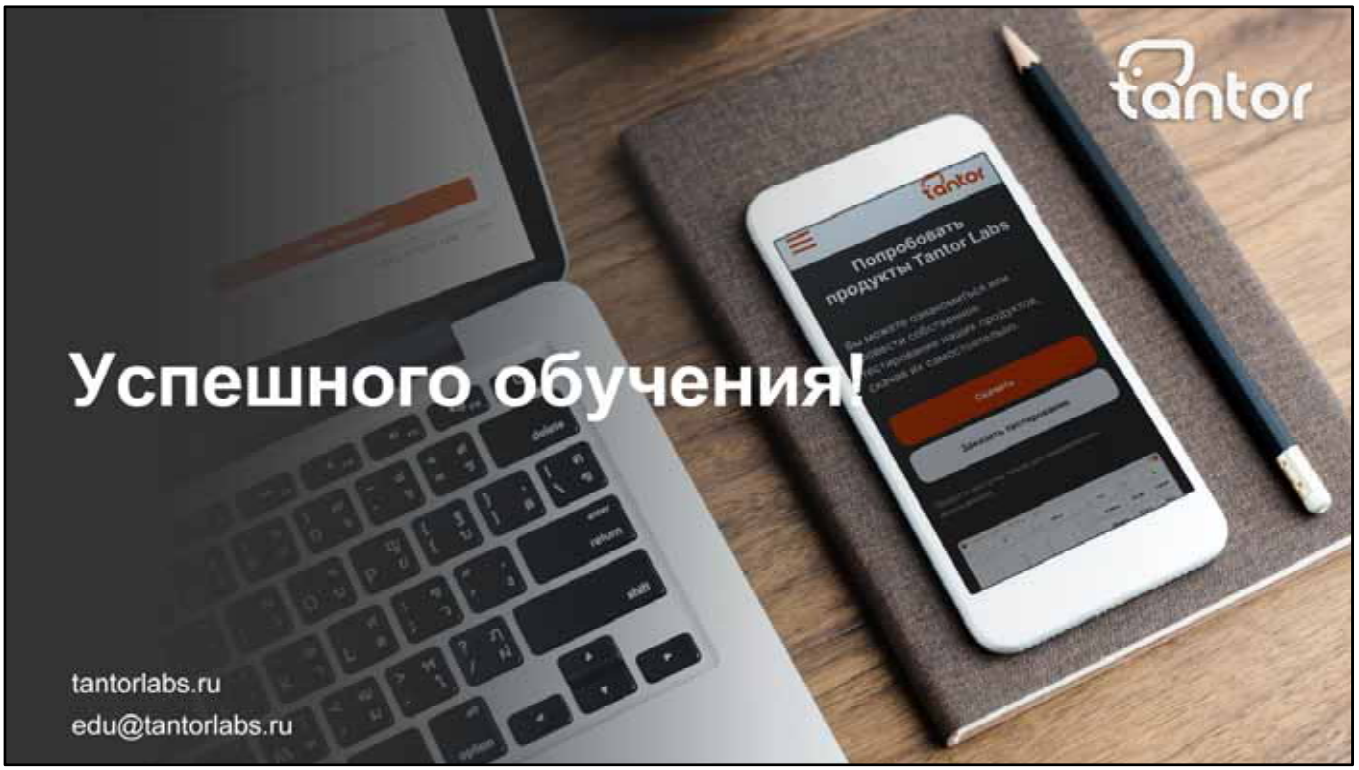
СУБД «Tantor Special Edition» (Tantor SE) оптимизирована и доработана для высоконагруженных систем, где кроме высокой производительности, важны надежность, отказоустойчивость, наличие удобных средств администрирования, профилирования и мониторинга нагрузки на экземпляр. Улучшения, добавленные в СУБД Tantor SE, разработаны с учетом многолетней практики эксплуатации промышленных приложений на СУБД PostgreSQL. Включает в себя оптимизации и расширения, позволяющие эффективно обслуживать кластера размером до 100 Тб без детальной настройки производительности. Часть новшеств добавляется в СУБД Tantor SE раньше, чем они появляются в PostgreSQL соответствующей версии.



- Special Edition 1С — это СУБД для высоких нагрузок, оптимизированная и одобренная для работы с приложениями 1С.



СУБД Tantor поставляется в составе Tantor Xdata — в этом случае СУБД идет в максимальной версии, с различными встроенными профилями нагрузки, в том числе оптимизированной для работы с 1С.


tantor
Попробовать продукты Tantor Labs
Успешного обучения!
tantorlabs.ru
edu@tantorlabs.ru

01. Установка и управление

Установка СУБД Тантор

Предварительные требования

Для установки нужны:

- поддерживаемая операционная система
- рекомендуется операционная система Astra Linux
- минимальные требования к оборудованию:
- 4 ядра CPU, 4Гб RAM, 40Гб SSD

СУБД Тантор поставляется в скомпилированном виде в виде пакетов для пакетного менеджера операционной системы. Перед установкой нужно свериться со списком операционных систем и их версий, для которых выпускается СУБД Тантор. В список поддерживаемых операционных систем входят:

Операционные системы с пакетным менеджером RedHat Packet Manager (rpm)

- Redos 7.3
- AltLinux
- Centos 7
- Oracle Linux 8
- Rocky 8, 9

Операционные системы с пакетным менеджером Debian (deb)

- линейка операционных систем AstraLinux
- Ubuntu 18, 20, 22
- Debian 10, 11.

Установка на другие операционные системы не поддерживается.

Оборудование:

Количество ядер центральных процессоров: по крайней мере 4;

Оперативная память: по крайней мере 4ГБ;

Свободное место на диске: не менее 40ГБ (плюс место под пользовательские данные которые планируется хранить). **Рекомендуется** использование SSD.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/install-binaries.html

Проверка возможности установки

- Дистрибутивы СУБД Тантор распространяются в виде файла для пакетных менеджеров Red Hat (rpm) и Дебиан (deb)
- В дистрибутивах указаны те пакеты, к функционалу которых могут обратиться утилиты и процессы экземпляра кластера
- в документации список зависимостей не указан, так как отличается для разных версий и сборок
- до установки можно получить список пакетов требующих 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 СУБД Тантор перечислены те библиотеки, к функционалу которых могут обратиться утилиты и процессы экземпляра кластера. 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 СУБД Тантор список зависимостей может отличаться, то в документации СУБД Тантор 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 СУБД Тантор можно использовать команды:

Для пакетного менеджера Дебиан: dpkg -I tantor*.deb

Для пакетного менеджера RedHat: rpm -qp --requires tantor-se-server-15-15.4.0.el7.x86_64.rpm

Ответ утилиты состоит из перечисления пакетов и, возможно, версий пакетов и библиотек. Например:

```
shadow-utils
grep
...
rpmlib(PayloadIsXz) <= 5.2-1
```

Значки <= и => указывают что требуются конкретные версии библиотек. В последней строке примера указаны ограничения версии пакетного менеджера, которые устанавливают проверку совместимости пакетного менеджера rpm для защиты от установки Тантор на несовместимую версию операционной системы.

Такая проверка может быть полезна для того, чтобы составить задачу на установку операционной системы. Запустить команду можно на любой операционной Linux где установлен пакетный менеджер rpm.

Для проверки перед установкой того, что зависимости выполнены можно использовать команду:

```
rpm -i --test  tantor*.rpm
```

Пример ошибки при проверке зависимостей дистрибутива Rocky 9 на Oracle Linux 7.

```
warning: tantor-be-server-15-15.4.2.el9.x86_64.rpm: Header V4 RSA/SHA512 Signature, key ID f24052f5: NOKEY
error: Failed dependencies:
...
 python3-libs is needed by tantor-be-server-15-15.4.2-0.x86_64
 rpmlib(PayloadIsZstd) <= 5.4.18-1 is needed by tantor-be-server-15-15.4.2-0.x86_64
```

Ошибки, связанные с **rpmlib** указывают на неподходящий дистрибутив.

Инсталлятор

- проверяет возможные конфликты библиотек и предлагает команду для их устранения
- добавляет адрес репозитария TantorLabs в список apt и yum
- Может скачать подходящий для операционной системы дистрибутив и установить его
- представляет собой текстовый скрипт:
- можно ознакомиться с тем какие действия выполняет инсталлятор
- легко выяснить в чем ошибки и исправить их

После установки необходимых пакетов, удаления конфликтующих можно скачать инсталлятор: https://**public.tantorlabs.ru**/db_installer.sh

После завершения загрузки можно поменять разрешения файловой системы чтобы скрипт мог выполняться.

Утилита скачает нужный дистрибутив.

Для скачивания коммерческих версий нужно установить переменные окружения:

```
export NEXUS_USER="имя"
export NEXUS_USER_PASSWORD="пароль"
export NEXUS_URL="nexus.tantorlabs.ru"
./db_installer.sh --edition=se
```

Возможные ошибки:

1) Конфликт может возникнуть, к примеру, если был установлен клиент (tantor-se-client-15_15.4.0_amd64.deb) так как пакет с сервером Тантор включает в себя библиотеки клиента. 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` Эта 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. Аутентификационные данные для скачивания дистрибутива позволяет только скачать коммерческие дистрибутивы и его сохранение в файле tantorlabs.list не рассматривается как брешь в защите.

3) У инсталлятора отсутствует параметр `--install-fromfile`

Используется версия инсталлятора ниже 23.12.21. Нужно скачать новую версию.

Локальная установка

- Проверить что версия инсталлятора не ниже 23.12.21 командой `./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

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

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

Затем можно запустить инсталлятор, задав желаемые параметры:

```
./db_installer.sh --edition=be --major-version=16 --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 Тантор 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=16 --from-file=./tantor-se-server-16_16.1.0_amd64.deb
```

Нужно указать параметром --major-version=16 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 и файлы
- устанавливаются нужные разрешения на директории, файлы и их владельцы

В процессе установки:
1)создается или модифицируется пользователь для СУБД Тантор, из под которого будут запускаться процессы и который будет являться владельцем директорий, относящихся к кластерам баз данных. Приме команды создания пользователя и именем postgres:
```
2)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.
3)создается директория `/opt/tantor/db/16` в которой располагаются исполняемые и вспомогательные файлы СУБД Тантор.
4)создается файл описателя службы `/usr/lib/systemd/system/tantor-se-server-16.service` для того чтобы можно было запускать экземпляр, обслуживающий кластер баз данных. Кластер баз данных представляет собой директорию в файловой системе хоста (синонимы: компьютер, узел, сервер) на котором было установлено программное обеспечение СУБД Тантор. Клиенты СУБД Тантор не имеют прямого доступа к файлам. 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" (instance) или экземпляром кластера баз данных Тантор.
Проверить статус службы можно командой
```
systemctl status tantor-se-server-16
```
5)создается файл `/etc/ld.so.conf.d/tantor-se-16.conf` Посмотреть список разделяемых библиотек, загруженных в кэш /etc/ld.so.cache можно командой `ldconfig -p | grep tantor`
Можно проверить что в переменной окружения `LD_PRELOAD` отсутствуют библиотеки, которые могут перекрыть библиотеки СУБД Тантор, так как `LD_PRELOAD` превалирует
6)создается директория `/var/run/postgresql` и файл `/usr/lib/tmpfiles.d/tantor-db.conf`
Файл используется стандартной службой очистки временных файлов. Директория является директорией по умолчанию для файлов Unix socket СУБД Тантор (параметр конфигурации (unix_socket_directories). В старых версих PostgreSQL использовалась директория "/tmp" которая по инерции может упоминаться в документации и руководствах.
Можно проверить, что в директории `/usr/lib/tmpfiles.d` отсутствуют другие файлы, которые могли остатья от предыдущих инсталляций postgresql, в которых указана та же самая директория, но с другими параметрами
```
systemctl status systemd-tmpfiles-*
 systemd-tmpfiles[]: /usr/lib/tmpfiles.d/tantor-db.conf:1: Duplicate line for path
"/run/postgresql", ignoring
```
7)создаётся директория `/var/lib/postgresql/tantor-`*`se`*`-16/data`
это директория по умолчанию для файлов кластера. Утилиты СУБД Tantor узнают о местоположении директории кластера либо ключём (параметром утилиты) -D , либо из переменной окружения PGDATA. Поэтому в устной речи эту директорию называют "PGDATA".
8)в конец файла `/var/lib/postgresql.bash_profile` добавляются строки
```
export PATH=$PATH
export PATH=/opt/tantor/db/16/bin:$PATH
```
для того чтобы зайдя под пользователем postgres можно было запускать утилиты управления кластером баз данных не указывая к ним путь.

После установки

- Установить переменную окружения PGDATA в профиле пользователя postgres
- создать кластер, если он не был ещё создан
- запустить экземпляр кластера
- перейти к настройке параметров кластера, настроить инструменты управления и мониторинга кластера (Платформа Тантор)

СУБД Тантор не имеет ограничений по количеству экземпляров запускаемых на одном узле. Однако, промышленные сервера баз данных обычно высоко нагружены и на одном узле обычно не запускают несколько экземпляров кластера баз данных. Несколько экземпляров на одном узле могут запускаться на время в процессе миграции на новую версию.

В некоторых пакетных дистрибутивах PostgreSQL имеются утилиты `pg_controlcluster`, `pg_createcluster`, являющиеся обёртками для стандартных утилит pg_ctl, initdb. Разработчики таких дистрибутивов предполагают, что это упрощает работу с несколькими кластерами на одном узле. СУБД Тантор эти утилиты не использует. Облачные дистрибутивы, где востребована работа большого числа экземпляров могут использовать другие сборки (синоним forks, "форки") PostgreSQL.

После установки можно сделать следующее:

1)Установить переменную окружения в файле `/var/lib/postgresql.bash_profile`

```
export PGDATA=/var/lib/postgresql/tantor-se-16/data
```

это упростит запуск утилит управления кластером, которые мы рассмотрим позднее pg_ctl, pg_controldata, pg_backup и других - им не нужно будет указывать параметр запуска -D путь_к_PGDATA.

2)создать кластер, если он не был ещё создан

3)запустить кластер `systemctl start tantor-se-server-16`

4)если автоматический запуск экземпляра был отключен (по умолчанию включён), то включить: `systemctl enable tantor-se-server-16`

5)Версию СУБД Tantor можно узнать функцией `tantor_version()`

6)перейти к настройке параметров кластера, настроить инструменты управления и мониторинга кластера Платформа Тантор, pgAdmin.

7)деинсталляция (например, предыдущей версии) выполняется пакетным менеджером. Для систем на основе Debian apt-get remove tantor-se-server-16

https://docs.tantorlabs.ru/tdb/ru/15_4/se/binary-download-execute.html

Создание кластера утилитой initdb

- Кластер создается утилитой командной строки initdb
- initdb запускается из под пользователя операционной системы postgres
- Перед запуском утилиты нужно создать директорию для файлов кластера
- выбрать настройки локализации
- ознакомиться с другими параметрами утилиты

Кластер создается утилитой командной строки initdb. Утилита может вызываться инсталлятором и утилитой `pg_ctl`.

initdb запускается из под пользователя операционной системы postgres.

Перед запуском утилиты нужно создать директорию, где будут находиться файлы создаваемого кластера PGDATA, проверить и при желании установить разрешения и права владения на эту директорию и те директории, в которых она находится, для пользователя postgres. При запуске экземпляра проверяются разрешения на саму директорию PGDATA:

1)разрешения должны быть `0700 (drwx --- ---)` или `0750 (drwx r-x ---)`

2)владельцем должен быть пользователь postgres.

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 выбирать для баз данных создаваемых после создания кластера:

1)LC_COLLATE правила сортировки текста

2)LC_CTYPE классификация символов (заглавные буквы, прописные буквы, символы цифр и другие классы символов)

3) схему кодирования символов LOCALE - третью часть значения после точки. Эта часть должна быть UTF8 или одной из поддерживающих кириллицу кодировок. Не все комбинации доступны и для однобайтных кодировок можно выбрать. Можно выбрать `ru_RU.iso88595` постольку поскольку она есть в поддерживаемых СУБД Тантор операционных системах.

Параметры локализации можно задать в параметрах initdb --locale=en_US.UTF8 --lc-collate=en_US.UTF8 --lc-ctype= en_US.UTF8 --locale-provider={libc|icu} --encoding=UTF8

Если не указывать параметры, то используются переменные окружения. Получить их список можно командой locale. Список допустимых комбинаций locale -a. Настроить dpkg-reconfigure locales. libc - стандартный провайдер. Параметр --encoding имеет смысл задавать, если в значении LOCALE нет кодировки (после точки) и есть несколько допустимых (совместимых) вариантов.

При выборе между UTF8 и `iso88595` (или `cp1251`) можно учитывать, что в UTF8 кириллические символы занимают больше места - два байта вместо одного. Однако, приложениям может требоваться хранение, например, фамилии клиента на его национальном языке. Про однобайтную кодировку koi8: её не стоит использовать из-за того, что бинарное сравнение символов не соответствует лингвистическому.

Параметры, на которые обратить внимание:

• -k -g явно включить контрольные суммы для блоков данных (для записей WAL они включены) и установить менее ограничивающие разрешения на создаваемые файлы и директории `0`755 (ноль означает, что это восьмеричное число).

• `--auth --auth-host --auth-local`

• -D путь к PGDATA

• `--wal-segsize` число в мегабайтах. По умолчанию 16Мб. Значение должно быть степенью двойки. Этот параметр **можно изменить** после создания кластера утилитой pg_resetwal --wal-segsize=новый_размер Задает размер файла журнала предзаписи (синонимы WAL-файл, WAL-сегмент). Меняют либо по причине большого количества файлов одной директории, либо по причине того, что максимальный размер буфера журнала в разделяемой памяти (wal_buffers) ограничен размером WAL-файла. Влияние размера WAL-буфера на производительность нелинейная.

Управление экземпляром кластера баз данных

Утилита управления экземпляром pg_ctl

- Основные действия: остановка, перечитывание конфигурационных файлов, проверка статуса экземпляра, а также возможен запуск экземпляра
- преимущество в простоте использования и получения результата выполнения команды
- запускается под пользователем postgres

`pg_ctl` - это утилита управления экземпляром. Преимущество утилиты - простота и удобство использования через командную строку. Интеграция с системными инструментами: pg_ctl может быть легко интегрирован с системными инструментами и сценариями автоматизации, что делает его полезным инструментом для автоматизации управления сервером баз данных PostgreSQL. pg_ctl обеспечивает мощный и гибкий способ управления сервером баз данных PostgreSQL, делая его одним из основных инструментов для администрирования PostgreSQL.

При обеспечении или получении технической поддержки позволяет точно выполнять короткие команды и получать результат их выполнения, которые могут даваться через мессенджеры в виде текстовых сообщений. Это одно из преимуществ консольных утилит по сравнению с графическими.

Основные команды, которые можно использовать с pg_ctl:

`start` - запуск экземпляра

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

Перед остановкой промышленного кластера рекомендуется выполнять контрольную точку, то есть давать команду checkpoint. Это уменьшает время на остановку.

restart - перезапуск, эквивалентен остановке и запуску, поэтому могут использоваться параметры которые задаются при остановке.

reload - перечитывает файлы конфигурации без остановки экземпляра

status - выводит статус экземпляра

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

Процесс postgres

- pg_ctl запускает процесс postgres, который порождает остальные процессы экземпляра
- Ключом `-o` утилиты `pg_ctl` можно передавать процессу postgres параметры командной строки и конфигурации кластера
- postgres `--single` позволяет использовать однопользовательский и однопроцессный режим. Режим используют для исправления содержимого кластера в сложных случаях повреждений
- для выхода из однопользовательского режима используется комбинация клавиш `Ctrl+d`

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 ключ **-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 16.1
backend> vacuum full
```

Для выхода из этого режима используется комбинация клавиш <Ctrl+d>.

Это не 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 ей, экземпляр запущенный pg_ctl утилитой systemctl не управляется
- является обёрткой вокруг pg_ctl
- ожидает запуска экземпляра 300 секунд (параметр `TimeoutSec=300` в файле-описателе службы)
- Запуск экземпляра:
  `systemctl start tantor-se-server-16`

В поддерживаемых СУБД Тантор операционных системах семейства Linux для запуска служб используется systemd. СУБД Танатор скомпилирован с опцией --with-systemd обеспечивающей поддержку всего функционала systemd. В дистрибутиве поставляется файл описания службы /usr/lib/systemd/system/tantor-se-server-16.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 убьет экземпляр, если он не запустится в течение этого времени. На промышленных серверах восстановление после сбоя по журналам может занять значительное время. Значение infinity в таких случаях рекомендуется и отключает логику таймаута.

Во время работы сервера его PID сохраняется в файле postmaster.pid в PGDATA. Это используется для предотвращения запуска нескольких экземпляров сервера в одном каталоге данных и также может использоваться для выключения сервера.

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

systemctl - главная команда для работы с systemd. По умолчанию запускается с правами root.

Запуск экземпляра:

systemctl start tantor-se-server-16.service

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

Если при экземпляра утилитой systemctl выдается ошибка:

```
Starting Tantor Special Edition database server 16...
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-16/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-16 в таком случае не может остановить экземпляр, **результат она не выдаёт и может возникнуть ложное впечатление, что экземпляр погашен.**

Проверить добавлен ли экземпляр в автозапуск можно командой `systemctl` **`is-enabled`** `tantor-se-server-16`

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

Экземпляр можно остановить командой pg_ctl stop.

Синтаксис команды:

```
pg_ctl stop [-D каталог_данных]
[-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. А этого можно ждать часами, при этом новые подсоединения невозможны, а это простой. В Oracle Database такой режим называется "shutdown normal". Таким образом, режим smart не практичен. Однако, в отличие от Oracle Database, после подачи сигнала на остановку в режиме smart можно подать сигнал на остановку в режиме fast. В Oracle Database же можно будет погасить экземпляр только в режиме "abort".

Поэтому 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 СУБД Тантор выполняется моментально, поэтому задержка в остановке связана, в основном, определяется длительностью выполнения контрольной точки.

fast режим остановки по умолчанию для остановки и через pg_ctl stop и через systmemctl stop

На промышленных 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 checkpint. 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, "защищаемые журналом предзаписи") по контрольной точке записываются в файлы, все файлы синхронизируются на один момент, информация об успешной остановке экземпляра записывается в **управляющий файл `pg_control`**. Это называется "корректной остановкой". При последующем запуске экземпляра по управляющему файлу определяется, что экземпляр был корректно остановлен и чтения журналов WAL не требуется.

Остановка экземпляра

- Экземпляр можно остановить командой `pg_ctl stop`
- Использование `pg_ctl` - наиболее удобный способ погасить экземпляр,
- Можно послать сигнал процессу postgres напрямую:
  `kill -INT `head -1 $PGDATA/postmaster.pid``
- Не рекомендуется посылать сигнал `SIGKILL` ни с одному процессу экземпляра
- `systemctl stop` не гарантирует остановки 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 "восстановлению" (путем повторного воспроизведения журнала WAL) при следующем запуске экземпляра. Рекомендуется использовать только в крайних случаях, например, подвисании (отсутствии дисковой активности, прогресса) остановки 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`
```

Обратите внимание, что кавычки обратные, а не апострофы.

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 СУБД Тантор 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/15_4/se/server-shutdown.html

Остановка экземпляра

- В журнале кластера при выполнении контрольных точек (параметр `log_checkpoints=on`) будут присутствовать сообщения типа:
  СООБЩЕНИЕ: начата контрольная точка: `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 постгрес нет команды `shutdown immediate`

В журнале кластера при выполнении контрольных точек (параметр log_checkpoints=on) будут присутствовать сообщения типа:
СООБЩЕНИЕ: начата контрольная точка: shutdown immediate

В постгрес нет команды `shutdown immediate`.

Текст "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) 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 по WAL)
xlog: контрольная точка вызвана достижением max_wal_size ("по размеру")
time: контрольная точка вызвана достижением checkpoint_timeout ("по времени")

Утилиты управления кластером баз данных

Утилиты управления (обёртки для команд SQL)

- находятся в директории `/opt/tantor/db/16/bin`
- путь к директории включен в переменную окружения PATH пользователя postgres в линукс
- Часть утилит командной строки - обёртки для команд SQL
- Утилиты и соответствующие им команды:
  `clusterdb - CLUSTER`
  `createdb - CREATE DATABASE`
  `createuser - CREATE ROLE`
  `dropdb - DROP DATABASE`
  `dropuser - DROP ROLE`
  `reindexdb - REINDEX`
  `vacuumdb - VACUUM`

В директории `/opt/tantor/db/16/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 администратора кластера выполняется не командами SQL (или удобнее выполнять) и для таких действий поставляются утилиты командной строки. Часть из них рассмотрим в течение курса.

Утилиты-оболочки (синоним обёртки, wrappers) для некоторых команд SQL (которые можно послать на выполнение утилитой psql). Иногда удобно выполнять действия в кластере баз данных скриптами командной строки и в таких скриптах удобно использовать утилиты-оболочки вместо написания вызова команды через psql:

`psql -c "КОМАНДА SQL"`

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

`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

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

`vacuum`**`lo`** - к вакуумированию (VACUUM) не имеет отношение. `vacuumlo` удобная для периодического запуска утилита удаления (вычистки) осиротевших больших объектов из баз данных кластера. Автоматизировать удаление осиротевших больших объектов можно разными способами (например, триггерами), эта утилита один из них. Расширение "lo" содержит функцию `lo_manage()` для использования в триггерах, предотвращающих появление осиротевших больших объектов.

Описание утилит:

https://docs.tantorlabs.ru/tdb/ru/15_4/se/reference-client.html

Утилиты управления резервированием

- pg_archivecleanup - используется на репликах (резервных кластерах)
- pg_basebackup - создаёт физические бэкапы
- pg_dump, pg_dumpall, pg_restore для логических бэкапов
- pg_receivewal - для создания поточных архивов WAL
- pg_resetwal для изменения размера WAL-сегментов

Часть утилит используется для выполнения важных задач по администрированию кластера и будут рассмотрены в курсе.

К резервированию кластера относятся утилиты:

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

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

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. Используется в процедурах обновления основной версии, переноса кластера на другие платформы, сборки, форки PostgreSQL. Представляет интерес параметр -g позволяющий выгружать общие объекты кластера.

pg_receivewal - используется для вытягивания по протоколу репликации содержимого файлов WAL (поточного архива). Обычно используется для организации хранения журналов WAL на узлах с бэкапами.

pg_recvlogical - инструмент для использования в логической репликации. Используется редко.

pg_resetwal очищает журнал WAL. Используется с параметром --wal-segsize для изменения размера WAL-сегментов, если захочется изменить их размер после создания кластера. Процедура требует аккуратности и знания что произойдёт с бэкапами и именами файлов WAL. Также для процедуры изменения размера WAL-сегментов критически важно, чтобы кластер был корректно остановлен. Меняют либо оп причине большого количества файлов одной директории, 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

https://docs.tantorlabs.ru/tdb/ru/15_4/se/reference-server.html

Утилиты управления (другие)

- pg_checksums - включение/отключение подсчета контрольных сумм блоков данных и проверка блоков данных кластера
- pg_rewind - для синхронизации кластеров, например, после аварийного переключения на физическую реплику и в процедурах апгрейда
- pg_upgrade - для перехода на новую основную версию СУБД Тантор
- pg_test_fsync - измеряет скорость записи в WAL сегменты в разных режимах
- pg_config - информация о параметрах инсталляции и сборки СУБД Тантор
- pg_controldata - выводит в текстовом виде содержимое управляющего файла кластера $PGDATA/global/**pg_control**

Утилиты для обновления версий СУБД Тантор и для настройки отказоустойчивости:

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

pg_upgrade - для апгрейда

pg_test_fsync - используется при настройке параметров записи в WAL сегменты, влияющих на отказоустойчивость

Полезные утилиты

pg_config - информация о параметрах инсталляции и сборки СУБД Тантор

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

https://docs.tantorlabs.ru/tdb/ru/15_4/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
- `vacuum_maintenance.py` и другие скрипты на языке Python, используются расширением для секционирования pg_partman `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

`pg_isready` проверка, что кластер принимает соединения, аналог psql -c "\q". Утилитой только 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.

`vacuum_maintenance.py` и другие скрипты на языке Python, используются расширением pg_partman секционирования ("партиционирования", partitioning) 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` - создает кластера

https://docs.tantorlabs.ru/tdb/ru/15_4/se/reference-server.html

Терминальный клиент psql

Терминальный клиент psql

- позволяет интерактивно вводить команды
- можно передавать команды и не интерактивно - команды могут быть взяты из файла или параметра командной строки
- есть файлы конфигурации
  глобальный
  `/opt/tantor/db/16/etc/postgresql/psqlrc`
  локальный по умолчанию `~/.psqlrc`
- Неинтерактивное выполнение команд:
  `psql -f файл_скрипта.sql`
  `psql -c "CREATE SCHEMA sh; CREATE TABLE sh.t (n numeric);"`
- описание параметров командной строки psql --help

В программном обеспечении СУБД Тантор есть 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 базах данных других производителей. На следующих слайдах рассматриваются особенности, которые могут казаться излишними, но именно они встречаются при повседневной работе и упрощают решение повседневных задач. 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 файл_скрипта.sql
psql -c "CREATE SCHEMA sh; CREATE TABLE sh.t (n numeric);"
```

У psql есть файлы конфигурации. Глобальный лежит в директории на которую указывает параметр pg_config --sysconfdir

для сборок Тантор это файл /opt/tantor/db/16/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 аналогичные файлы называются "glogin.sql"

Оба файла могут быть сделаны специфичными для версии psql путем добавления дефиса и идентификатора основной или минорной версии Tantor SE к имени файла, например ~/.psqlrc-16 или ~/.psqlrc-15.4.Все файлы применяются, но более специфичный файл превалирует.

С помощью этих файлов можно сделать работу в psql удобнее.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/app-psql.html#psql

psql: подключение к базе данных

- psql подключается к конкретной базе данных в кластере
- Для подсоединения к базе нужно пройти аутентификацию
- способы аутентификации рассматриваются в следующих главах
- Роль и пользователь синонимы и абсолютно одинаковые понятия
- Подключиться одновременно к нескольким базам даже расположенным в одном и том же кластере нельзя

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. СУБД Тантор 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 сопоставление 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` абсолютно одинаковы.

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 одновременно к нескольким базам даже из одного кластера кластере нельзя. Базы изолированы друг от друга с точки зрения безопасности и привилегий. Для одновременной работы с таблицами в разных базах данных даже если они находятся в одном кластере 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 ...`

psql: параметры подключения

Параметры командной строки для подсоединения:

- `-U` роль, по умолчанию имя пользователя операционной системы
- `-d` имя_базы, по умолчанию имя роли
- `-h` хост, по умолчанию `/var/run/postgresql`
- `-p` порт, по умолчанию 5432

Команды:

- Команда вывода деталей подсоединения:
  `\conninfo`
- переподсоединение:
  `\c` имя_базы имя_роли хост порт

  параметры текущего соединения можно передать, указав символ тире "-"
  `\c - - localhost`

Параметры командной строки psql которыми можно указать куда и под какой ролью подключаться:

-U роль или --username=роль значение по умолчанию имя пользователя операционной системы из под которого запущен psql

-d имя_базы или --dbname=имя_базы значение по умолчанию имя роли, 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`) то есть используется локальное соединение через Unix-сокет.

Если `psql` или другие утилиты выдают ошибку

```
could not connect to server: No such file or directory
          Is the server running locally and accepting
          connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

возможно запускается утилита старой версии (например, из пути /usr/bin/psql). Версия проверяется psql -V

помимо 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 порт тоже используется, так как директория одна и та же для всех кластеров. Если это директория в файловой системе, то в ней основной процесс postgres создаёт файл у которого суффикс является номером порта. Например, /run/postgresql/.s.PGSQL.**5432**

Также можно использовать сокращенный синтаксис `psql` параметры имя_базы имя_пользователя. Например, `psql postgres postgres`

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

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

Выдаётся имя роли, 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_роли 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, сохраняется прежнее.

Получение справки по командам psql

- 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 ней помощь
- по умолчанию если результат не помещается в окне терминала используется постраничный вывод
- для постраничного вывода используются утилиты операционной системы more или less, утилита less удобнее
- утилита устанавливается командой `\setenv PAGER 'less -XS'`
- режим постраничного вывода отключается командой `\pset pager off`
- в режиме постраничного вывода (после двоеточия):
  q - выход, `z` - вперёд, `b` - назад, h - помощь

После установки СУБД Тантор можно запустить на сервере psql без параметров и psql подсоединяется 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

Узнать какие команды SQL формирует psql чтобы исполнить команды, начинающиеся на \d (describe - получить описание объекта) можно установив параметр

`\set` `ECHO_HIDDEN` **`on`**

Если текст не помещается на экран используется функционал "постраничного выпода" (pager), вы увидите двоеточие.

По нажатии клавиши <ENTER> высветится еще одна строка.

Если нужно высветить следующую страницу, то после двоеточия нужно набрать символ "z"

Если вернуться к предыдущей странице - символ "b" (back)

если хочется прервать вывод можно набрать символ "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 psger off`

Постраничный вывод реализуется передачей результата вывода утилите less операционной системы.

История команд по умолчанию доступна по нажатию стрелок вверх/вниз на клавиатуре. История команд набранных интерактивно в psql хранится в файле `~/.psql_history` Его местоположение может быть переопределено переменными окружения `HISTFILE` или `PSQL_HISTORY`, но смысла в этом нет. Рядом с ~/.psql_history например лежит файл ~/.bash_history с историей команд терминала операционной системы. Названия файлов начинающиеся с точки в Линукс считаются "скрытыми" файлами. Например, команда `ls` без параметров не показывает такие файлы.

psql может запускаться с клиентской машины из сборок, отличных от сборок Тантор. `psql` работает лучше с серверами той же или более старой основной версии. При подключении к более новой версии СУБД могут отказаться работать команды psql (те который начинаются на обратный слэш).

Форматирование вывода psql

- По умолчанию psql выводит результат запроса с псевдографикой:

```
postgres=# select datname, datistemplate from pg_database;
  datname  | datistemplate
-----------+---------------
 postgres  | f
 template1 | t
 template0 | t
(3 rows)
```

- можно детально настроить формат вывода параметрами \pset и переключателями \a \t \x
- посмотреть текущие настройки форматирования: \pset
- отображение времени выполнения команды, удобно для настройки производительности
  `\timing`

Посмотреть текущие настройки форматирования можно набрав `\pset`

Если нужно повторять команду через интервалы времени, а такая потребность для мониторинга может возникать у администратора:

`\watch` секунд (выход CTRL+X)

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

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

По умолчанию столбцы разделяет вертикальная черта, можно установить другой символ, например, пробел:

```
\pset fieldsep ' '
```

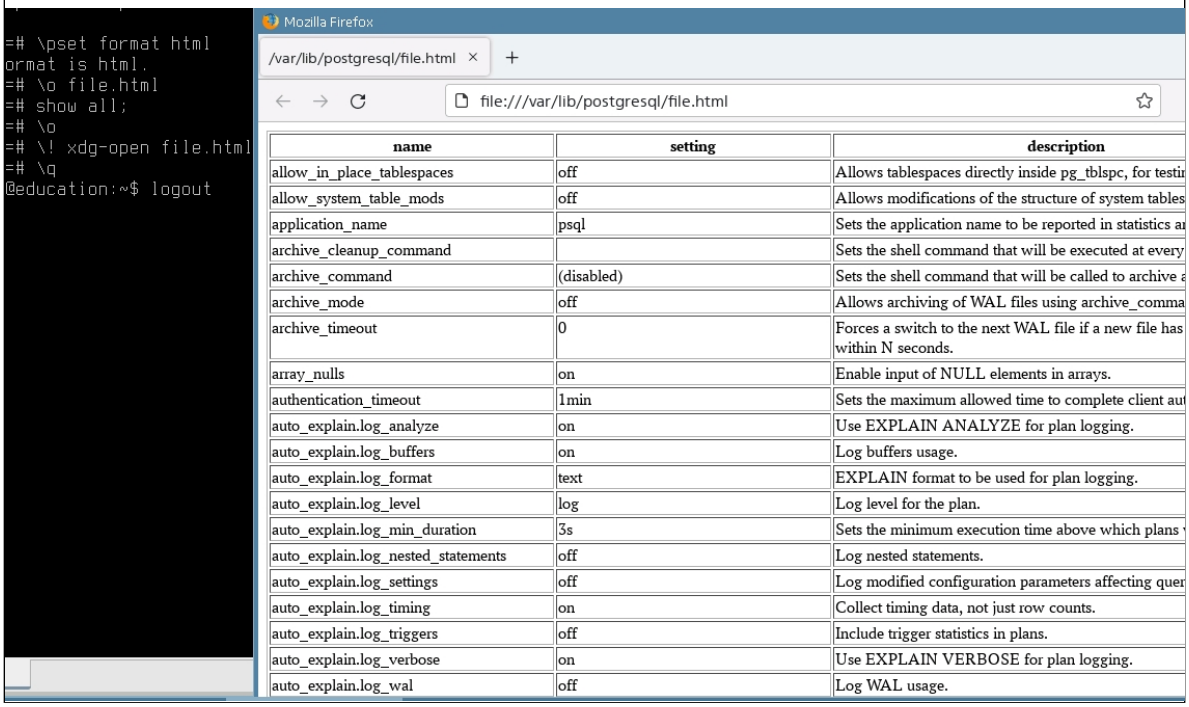
Отключение выравнивания и замена разделителя на нужный символ позволяет выводить результат выборки в формате, удобном для передаче программе работающей с таблицами.

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

```
postgres=# \timing
Timing is on.
postgres=# \timing
Timing is off.
```

Вывод результата в формате HTML

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



name	setting	description
allow_in_place_tablespaces	off	Allows tablespaces directly inside pg_tblspc, for testi
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 a
archive_cleanup_command		Sets the shell command that will be executed at every
archive_command	(disabled)	Sets the shell command that will be called to archive a
archive_mode	off	Allows archiving of WAL files using archive_comma
archive_timeout	0	Forces a switch to the next WAL file if a new file has within N seconds.
array_nulls	on	Enable input of NULL elements in arrays.
authentication_timeout	1min	Sets the maximum allowed time to complete client au
auto_explain.log_analyze	on	Use EXPLAIN ANALYZE for plan logging.
auto_explain.log_buffers	on	Log buffers usage.
auto_explain.log_format	text	EXPLAIN format to be used for plan logging.
auto_explain.log_level	log	Log level for the plan.
auto_explain.log_min_duration	3s	Sets the minimum execution time above which plans
auto_explain.log_nested_statements	off	Log nested statements.
auto_explain.log_settings	off	Log modified configuration parameters affecting quer
auto_explain.log_timing	on	Collect timing data, not just row counts.
auto_explain.log_triggers	off	Include trigger statistics in plans.
auto_explain.log_verbose	on	Use EXPLAIN VERBOSE for plan logging.
auto_explain.log_wal	off	Log WAL usage.

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

Вывод результата в расширенном формате

- переключается командой **\x**

```
postgres=# \x
Expanded display is off.
postgres=# \x
Expanded display is on.
postgres=# select * from pg_settings limit 1;
-[ RECORD 1 ]---+---------------------------------------------------------
name            | allow_in_place_tablespaces
setting         | off
unit            |
category        | Developer Options
short_desc      | Allows tablespaces directly inside pg_tblspc, for testing.
extra_desc      |
context         | superuser
vartype         | bool
source          | default
min_val         |
max_val         |
enumvals        |
boot_val        | off
reset_val       | off
sourcefile      |
sourceline      |
pending_restart | f

postgres=# \x
Expanded display is off.
```

Если строка результата имеет множество столбцов или длинные значения в полях, можно отображать данные построчно. Переключение вывода выполняется короткой командой \x

Вернуть обычный режим можно еще раз набрав **\x**

В запросе обычно указывается сортировка и ограничение количества выдаваемых строк: `ORDER BY` и `LIMIT`.

Приглашение к вводу команд (промпт) psql

- Зачем? избегать выдачи команд "не в том окне"
- Приглашение (промпт) можно менять командами `\set PROMPT1` и `\set PROMPT2`

```
postgres=# BEGIN TRANSACTION;
BEGIN
postgres=*#  select
postgres-*# tantor_version();
        tantor_version
------------------------------
 Tantor Special Edition 16.1.0
(1 row)

postgres=*# ffff;
ERROR:  syntax error at or near "ffff"
LINE 1: ffff;
        ^
postgres=!# COMMIT;
ROLLBACK
postgres=#
```

- "-" вторая и последующие строки (PROMPT2)
- "*" открыта и не зафиксирована транзакция
  "!" транзакция в состоянии сбоя и может только откатиться, даже если набрать `COMMIT;`

Нередки случаи, когда 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 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.

При промышленной эксплуатации удобно менять эти приглашения в файле `~\.psqlrc` чтобы видеть к какой базе кластера подсоединены.

Автофиксация транзакций и выполнение команд psql

- по умолчанию psql работает в режиме автоматической фиксации транзакции (AUTOCOMMIT)
- Изменение режима автофиксации:
  `\set AUTOCOMMIT on`
  `\set AUTOCOMMIT off`
- посмотреть переменные psql: команда `\set`
- очистка буфера набранных команд `\r`
- просмотр буфера или последней команды если буфер пуст `\p`
- ";"<ENTER> завершает команду SQL и посылает ее на выполнение серверному процессу

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

Командой `\set` можно посмотреть переменные psql. Часть переменных предопределена и управляет работой psql. Можно устанавливать на время до выхода из psql свои переменные и пользоваться ими как макросами.

Стоит отличать команды `\set \pset set`. Последняя относится к SQL и меняет параметры работы серверного процесса на уровне сессии (`set session`) или транзакции (`set local`). `\pset` это предопределённые параметры форматирования вывода psql.

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

В psql есть нестандартные команды `\g \gx \gexec \gset \g` которые могут заменять стандартно используемый символ ";" Эти нестандартные команды имеют широкие возможности, но их использование в скриптах делает скрипты непереносимыми - скрипты не смогут выполняться нигде, кроме psql.

Если не набрать ";" а просто набрать возврат каретки, то psql считает что команда многострочная и предыдущие строки накапливаются в буфере.

Если вы хотите очистить буфер, можно набрать `\r` (сокращение от `\reset`)

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

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

Режим автофиксации означает, что psql неявно после каждой команды которая работает в рамках транзакции (и вместе с такой командой) отправляет команду COMMIT;

Если вы хотите отключить режим автофиксации, вы можете отключить этот режим в системном файле `psqlrc` или в вашем файле `~/.psqlrc` или в своей сессии. За это отвечает параметр

```
\set AUTOCOMMIT on
\set AUTOCOMMIT off
```

Переменные psql

- устанавливаются командой `\set` имя значение
- Срок жизни до выхода из psql или до выполнения команды `\unset` имя
- есть переменные окружения операционной ситемы на которые реагирует psql
- их можно устанавливать в psql или в файлах параметров (`~/.psqlrc`)
  `\setenv PSQL_EDITOR /usr/bin/mcedit`

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

Переменные могут использоваться как макросы. Ссылаться на переменные можно префиксируя их символом двоеточия.

Пример:

```
postgres=# \set TEST1 'select user'
postgres=# :TEST1;
   user
----------
 postgres
(1 row)
postgres=# select * from (:TEST1);
```

По умолчанию для команд редактирования `\ef \ev \e` используется `vi`. Переопределить редактор можно установив переменную окружения

```
export PSQL_EDITOR=/usr/bin/mcedit
```

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

Либо находясь в psql дать команду `\setenv PSQL_EDITOR /usr/bin/mcedit`

Либо вставить команду `\setenv PSQL_EDITOR /usr/bin/mcedit` в файле `~/.psqlrc` или глобальном `/opt/tantor/db/16/etc/postgresql/psqlrc`

В разделе документации "Окружение" **https://docs.tantorlabs.ru/tdb/ru/15_4/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 том числе в файле `~/.psqlrc` или глобальном `/opt/tantor/db/16/etc/postgresql/psqlrc`. **Другими командами типа \set \pset \! export переменные окружения не устанавливаются**.

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
  ```

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 (скрппта) 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 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
```

Графическая утилита pgAdmin

pgAdmin

- свободно распространяемая графическая программа для работы с кластерами PostgreSQL
- работает с СУБД Тантор
- pgAdmin поддерживается в Астра Линукс

pgAdmin - свободно распространяемая графическая программа для работы с кластерами PostgreSQL, в том числе СУБД Тантор.

Утилита выпускаются 3 версии pgAdmin3 - оконный интерфейс, разработка завершилась в 2016 году и в 4 версии pgAdmin4 веб-интерфейс, но с удобной возможностью создать ссылку на десктопе. Многие администраторы и разработчики используют pgAdmin. Она позволяет использовать пошаговую отладку хранимых подпрограмм - является клиентским интерфейсом к функционалу свободно распространяемой библиотеки pldebugger реализующей серверную часть функционала отладчика.

pgAdmin3 не работает с PostgreSQL 15 и новее, так как при подсоединении обращается к столбцу datlastsysoid таблицы pg_database системного каталога кластера, который был удалён в 15 версии (https://pgpedia.info/postgresql-versions/postgresql-15.html)

pgAdmin4 может быть установлен в Астра Линукс 1.7 Эта версия основана на Debian 10 Buster (https://wiki.astralinux.ru/pages/viewpage.action?pageId=53646577).

Значит ищем дистрибутивы в поддиректории "buster" https://ftp.postgresql.org/pub/pgadmin/pgadmin4/apt/buster/dists/pgadmin4/main/binary-amd64/

pgadmin4-server_8.3_amd64.deb

pgadmin4-desktop_8.3_amd64.deb

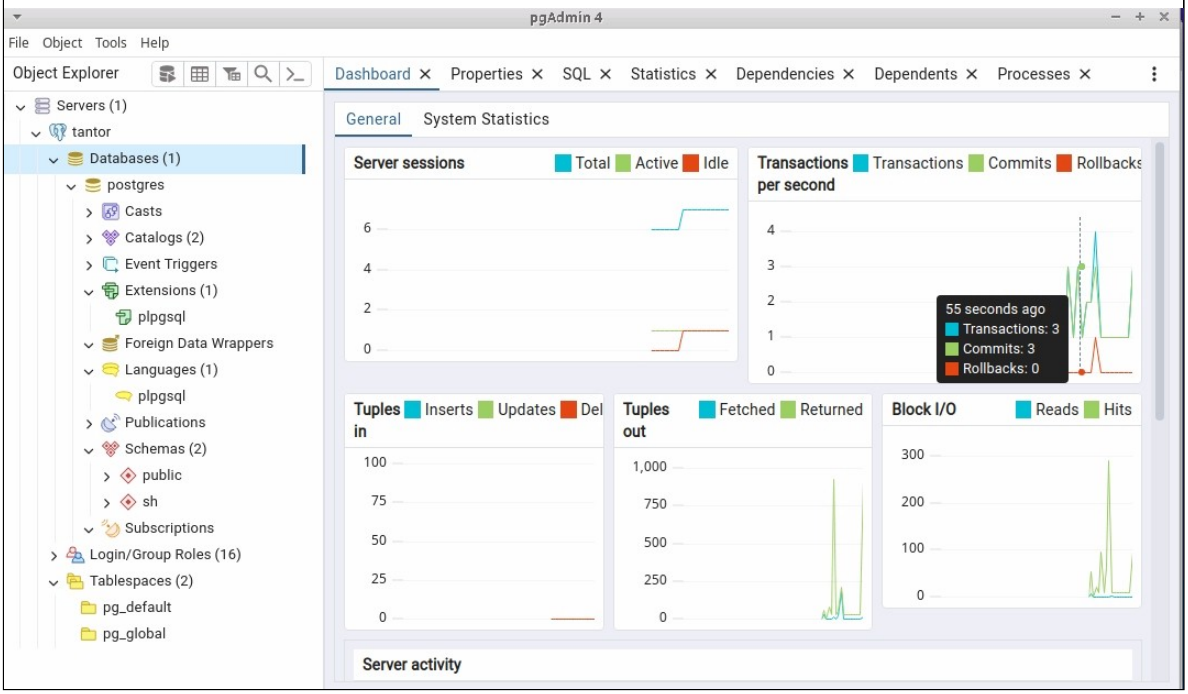
Устанавливать их кможно командами:

```
dpkg -i pgadmin4-server_8.3_amd64.deb
dpkg -i pgadmin4-desktop_8.3_amd64.deb
```

Запускать из меню Start-> Development ->pgAdmin 4

В меню pgAdmin4: в File->Preferences-> Paths -> Binary Paths--> PostgreSQL 16 установить путь `/opt/tantor/db/16/bin` чтобы из меню Tools можно было запускать "PSQL Tool".

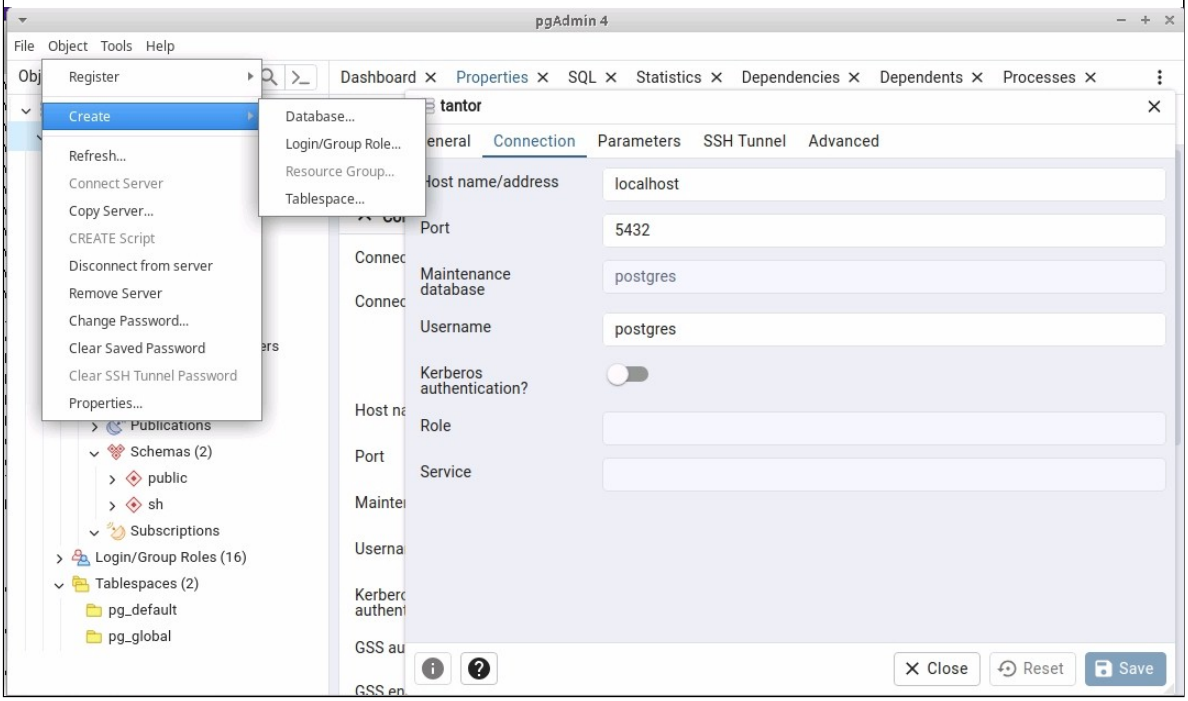
pgAdmin: Dashboard



Дальше на слайдах показаны снимки основных страниц pgAdmin.

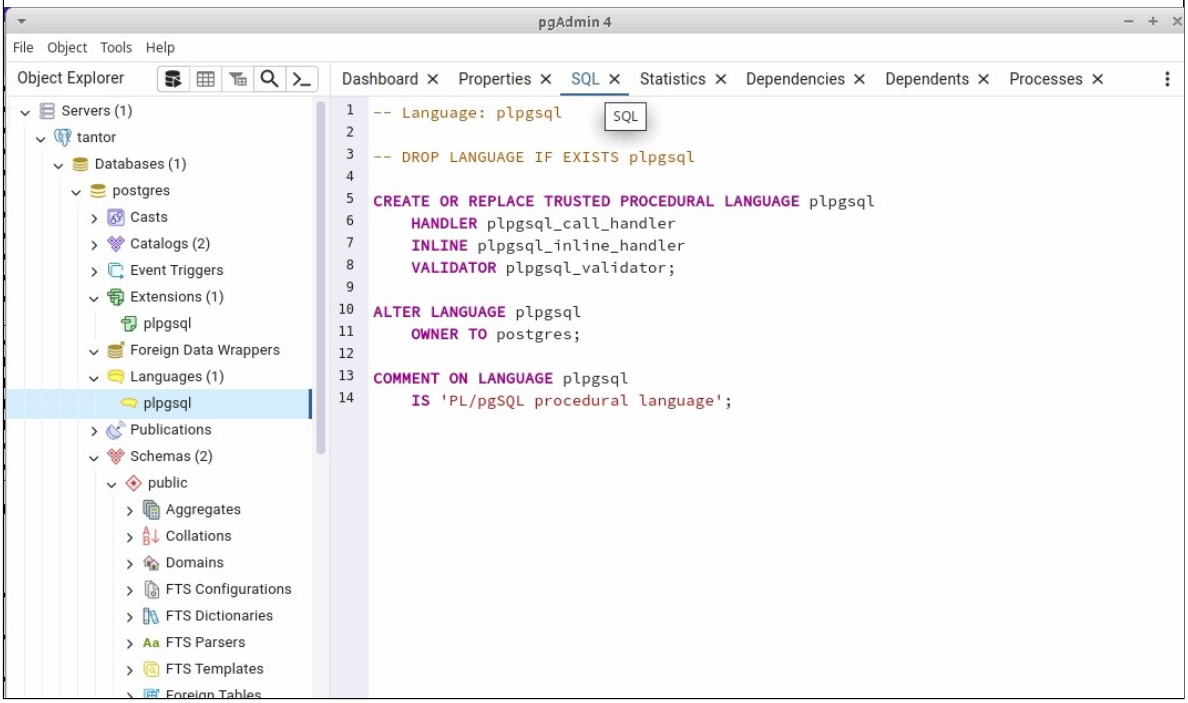
Это первая страница, которая показывается после подключения к экземпляру. Слева в Object Explorer показывается список общих объектов кластера: баз данных, ролей, табличных пространств.

pgAdmin: меню



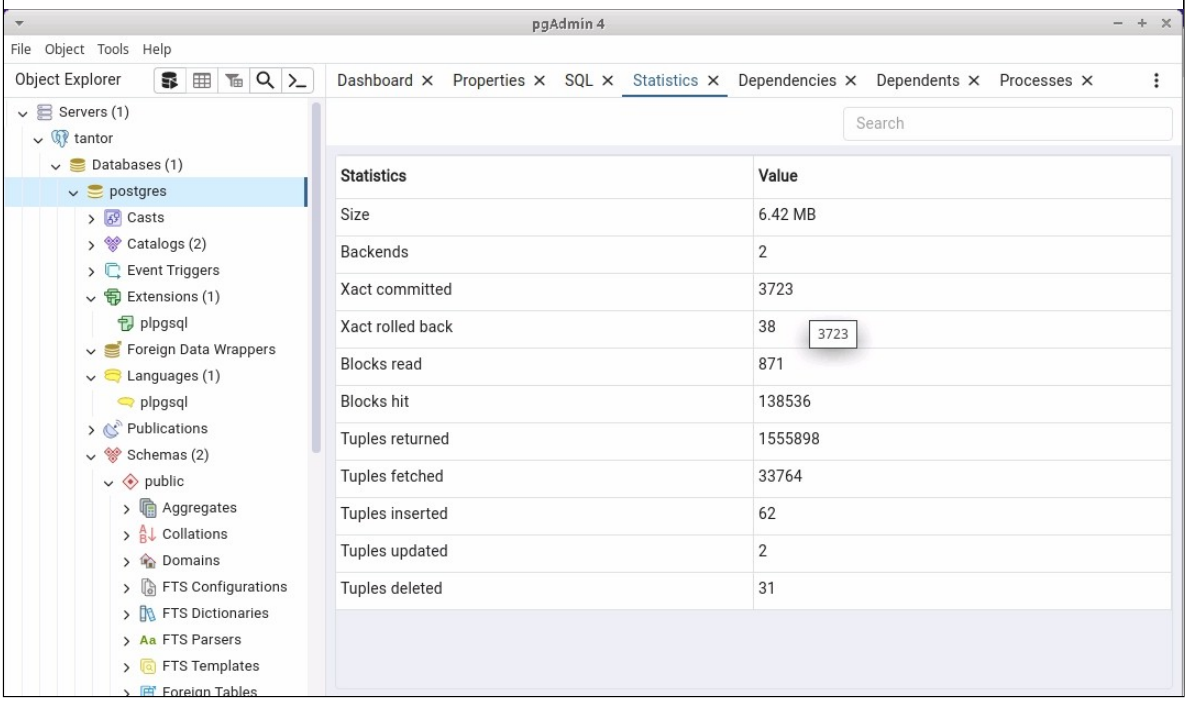
В меню можно вызвать визард создания объектов.

pgAdmin: закладка SQL команды создания объектов



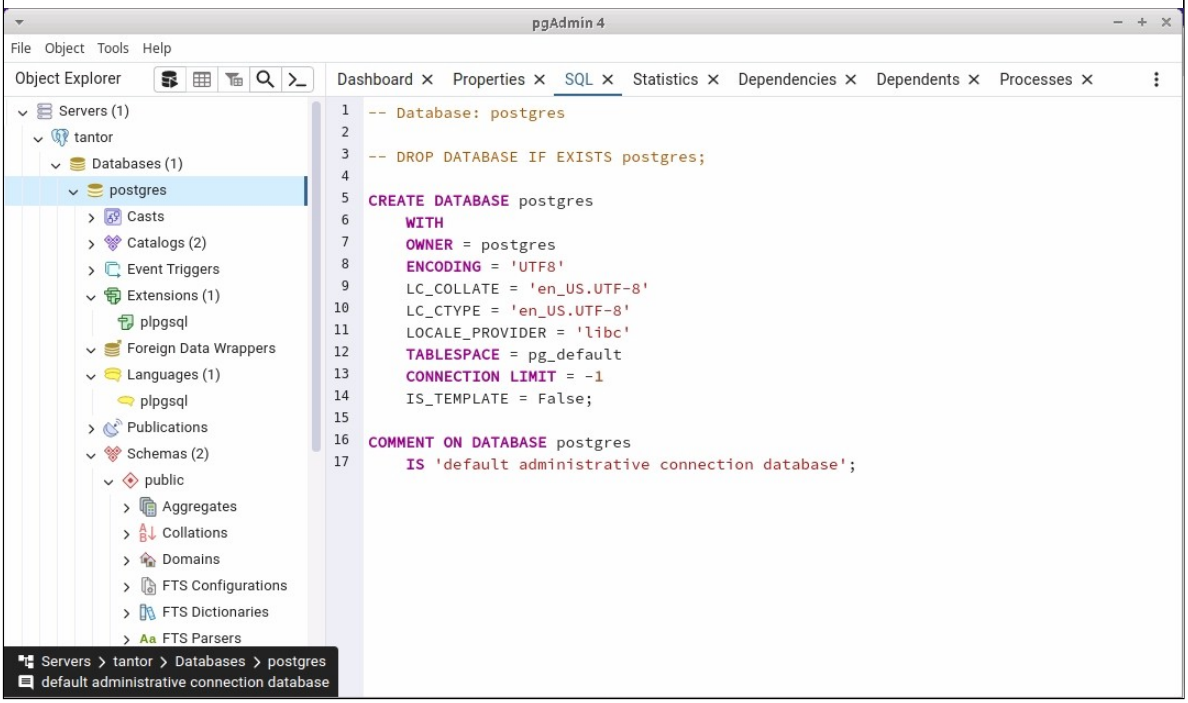
В Object Explorer можно кликнуть на объект и в правой части окна появится команда его создания

pgAdmin: закладка Statistics



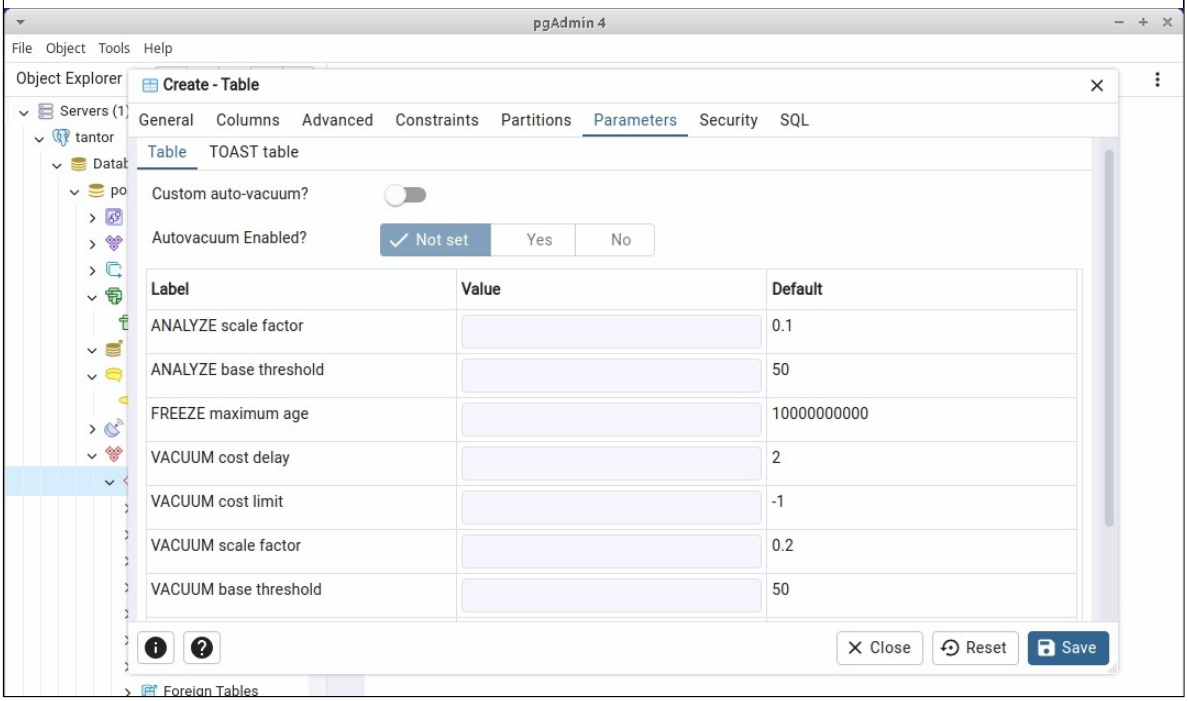
В pgAdmin есть страница на которой отображается статистика по объектам

pgAdmin: команда создания базы данных



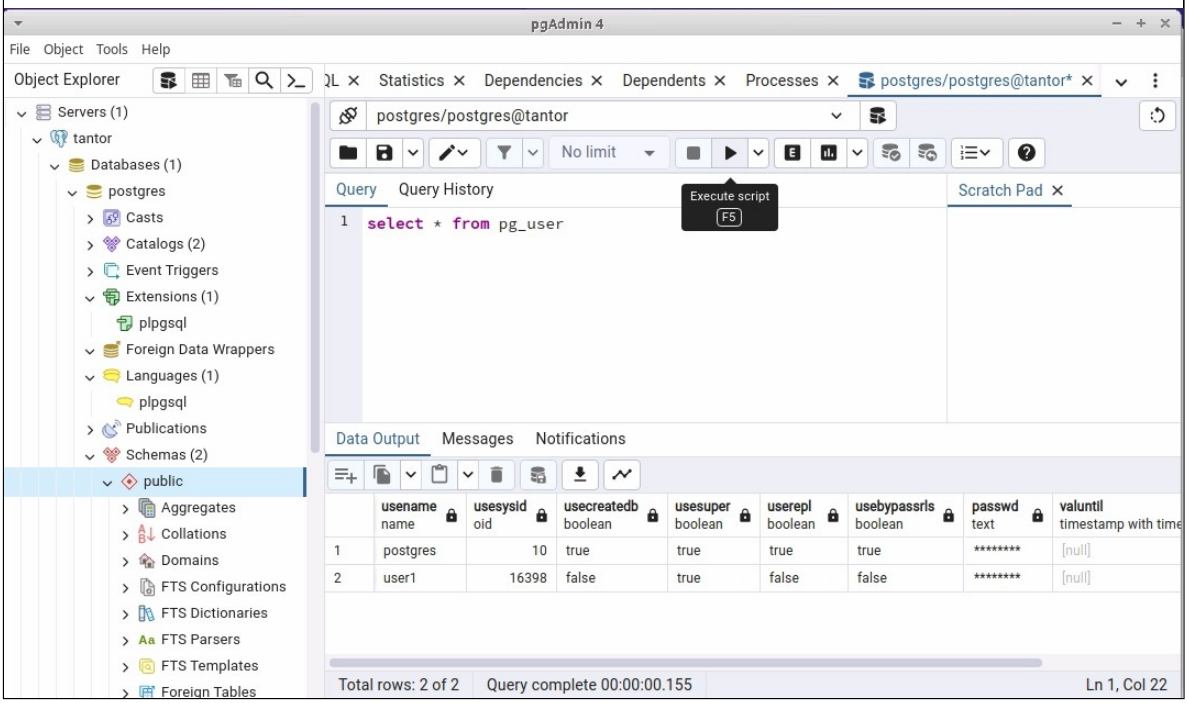
Пример команды создания базы данных

pgAdmin: визард создания таблицы



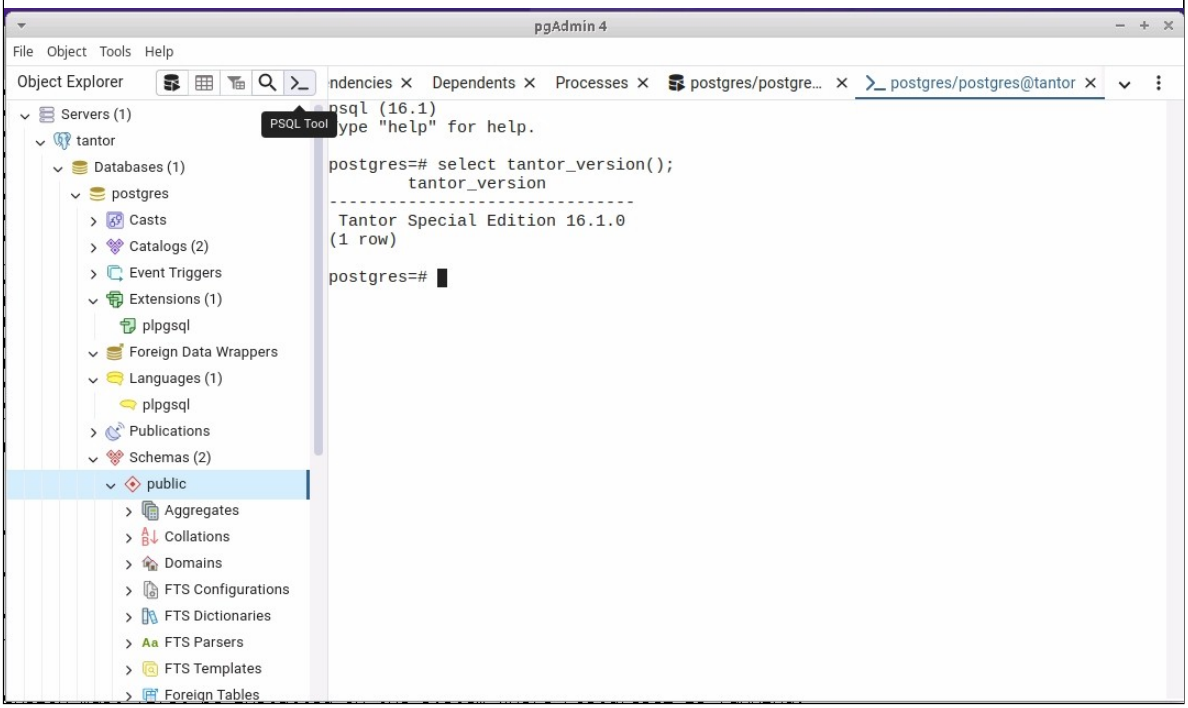
Команда создания таблицы синтаксически наиболее сложная из всех команд. В ней есть много опций. В pgAdmin есть визард создания таблицы, опции размещены на наборе закладок. На слайде показаны параметры работы автовакуума, которые можно настроить на уровне таблицы.

pgAdmin: Query Builder



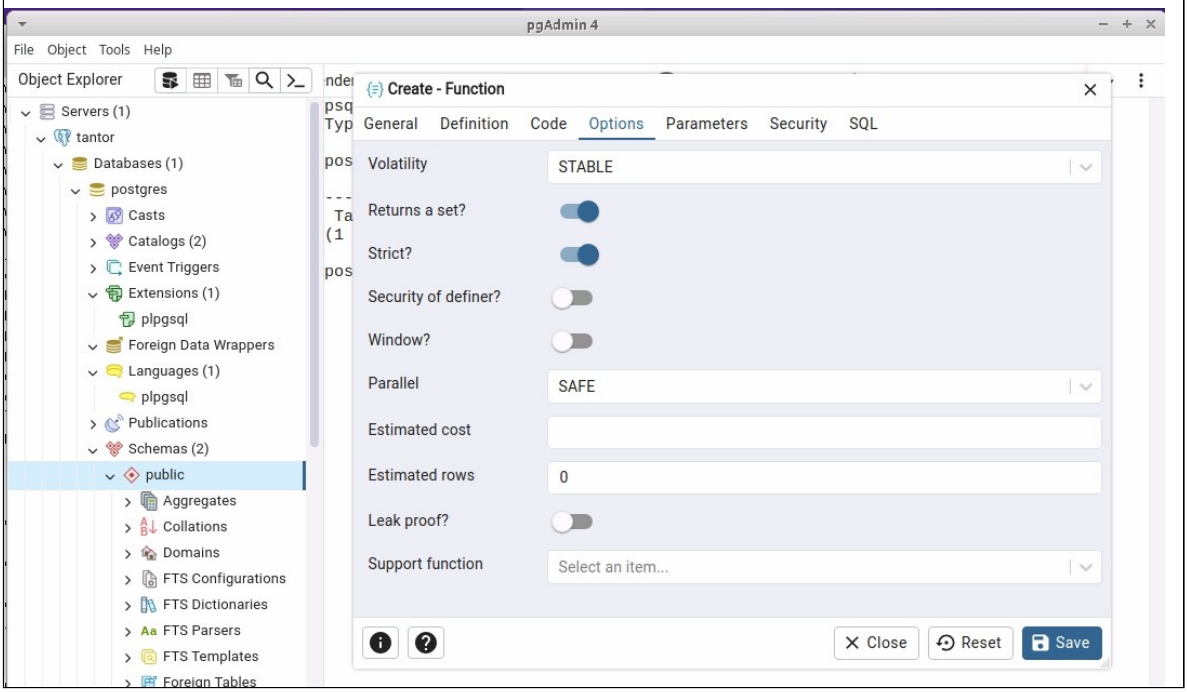
Графический редактор SQL-команд. Есть возможность в удобном виде просматривать результирующую выборку и сообщения серверного процесса.

pgAdmin: psql в окне



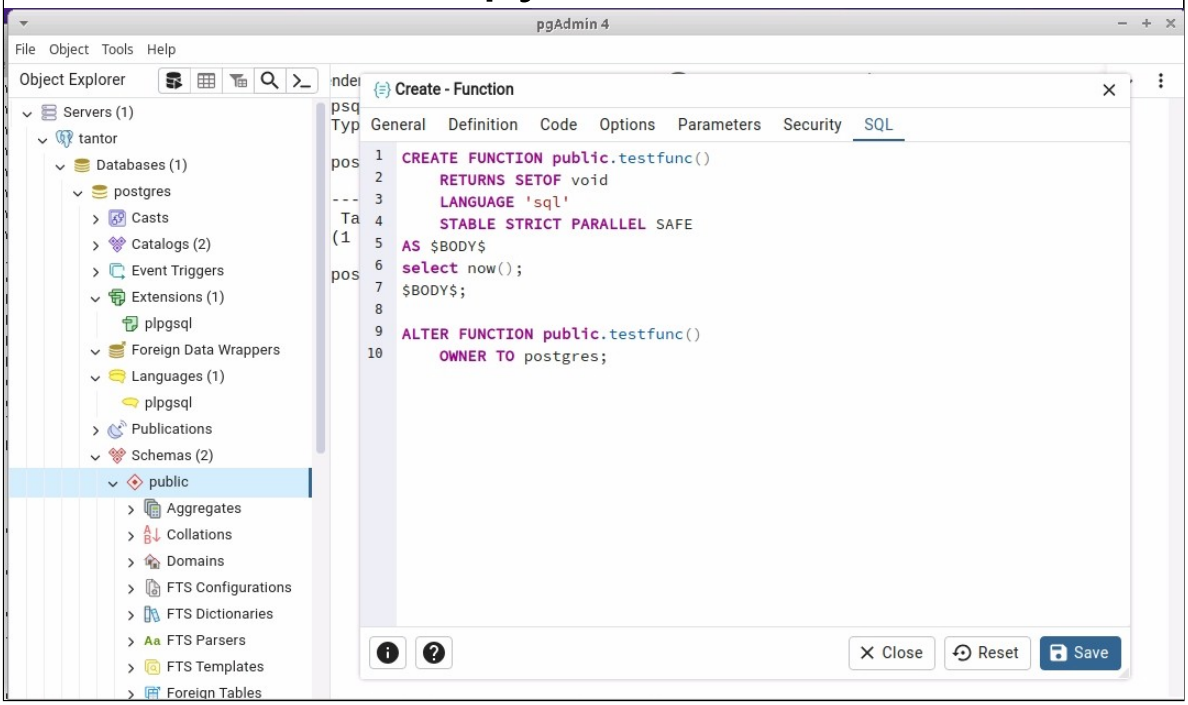
Также есть возможность запустить psql в окне pgAdmin. Это позволяет выполнять команды psql. Перед вызовом утилиты из пункта меню нужно настроить путь к директории утилит.

pgAdmin: визард создания функции



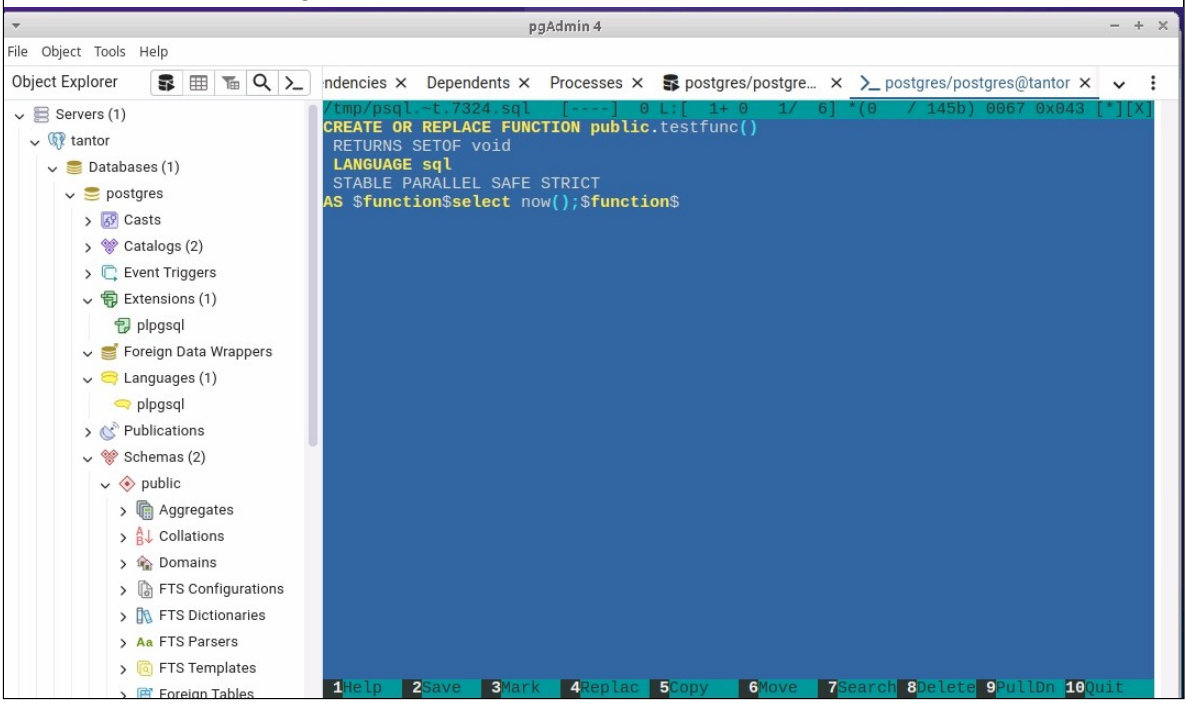
Визард создания подпрограмм. pgAdmin полезен не только для администраторов, но и для разработчиков

pgAdmin: результат создания функции



В результате работы визарда генерируется команда создания подпрограммы.

pgAdmin: редактирование функции в окне psql \ef



В качестве текстового редактора в psql был выбран mcedit. Этот сниок экрана иллюстрирует то, что psql запускающийся в окне pgAdmin полнофункционален.

Демонстрация

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

01. Демонстрация

Часть 1. Установка СУБД Тантор

В виртуальной машине курса предустановлена версияТантор SE для целей обучения.
Продемонстрируем установку СУБД Тантор.

1) Откроем терминал с правами root:
```
astra@tantor:~$ sudo bash
```

2) Выполним предварительные проверки.
Число ядер процессора (результат может отличаться от приведенных как пример значений)

```
root@tantor:/home/astra# cat /proc/cpuinfo | grep cores
cpu cores   : 2
cpu cores   : 2
```

Оперативной памяти

```
root@tantor:/home/astra# cat /proc/meminfo | grep Mem
MemTotal:        8130152 kB
MemFree:         3288668 kB
MemAvailable:    4781360 kB
```

Свободное место в точке монтирования "/"

```
root@tantor:/home/astra# df -HT | grep /$
/dev/sda1      ext4       41G   22G   18G  56% /
```

Свободно 18Гб

При промышленной эксплуатации рекомендуется иметь 4 ядра;
Оперативной памяти: по крайней мере 4ГБ;
Свободного места на системе хранения ("диске"): 40ГБ.

3) Скачаем инсталлятор:

```
root@tantor:/home/astra# wget https://public.tantorlabs.ru/db_installer.sh

https://public.tantorlabs.ru/db_installer.sh
Resolving public.tantorlabs.ru (public.tantorlabs.ru)... 84.201.157.208
Connecting to public.tantorlabs.ru (public.tantorlabs.ru)|84.201.157.208|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 18312 (18K) [application/octet-stream]
Saving to: 'db_installer.sh'

db_installer.sh             100%[=================================================>]  17,88K  -
-.-KB/s    in 0s

'db_installer.sh' saved [18312/18312]
```

4) Посмотрим разрешение на исполнение инсталлятора

```
root@tantor:/home/astra# ls -al db_installer.sh
-rw-r--r-- 1 root root 18353 db_installer.sh
```

5) Установка разрешения на исполнение инсталлятора

```
root@tantor:/home/astra# chmod +x db_installer.sh
```

6) Остановим основной кластер:

```
root@tantor:/home/astra# systemctl stop tantor-se-server-16
```

7) Остановим реплику:

```
root@tantor:/home/astra# systemctl stop  tantor-se-server-16-replica.service
```

8) Проверим версию инсталлятора и обратим внимание слушателей на выделенные параметры:

```
root@tantor:/home/astra# ./db_installer.sh --help
======================================================================
Usage: db_installer.sh [OPTIONS]

Installer version: 24.04.12

This script will perform installation of the Tantor DB on current host.
If the Tantor DB is already installed, no actions will be taken.

Available options:
  --help                    Show this help message.

----------------------------------------------------------------------
  --edition=                Set edition (be, se, se-1c, se-certified). "se" is default.
  --major-version=          Set major version (14, 15)
  --maintenance-version=    Set maintenance version (15.2.4).
                            By default latest version will be installed.

  --do-initdb               After installation run initdb with checksums.

  --package=                Set specific package (all, client, libpq5).
                            "all" is default.

----------------------------------------------------------------------
  --from-file=              Install package from local file (rpm, deb)
                            May be used with --do-initdb option

======================================================================
Example for commercial use
======================================================================

export NEXUS_USER="user_name"
export NEXUS_USER_PASSWORD="user_password"
export NEXUS_URL="nexus.tantorlabs.ru"

./db_installer.sh \
    --do-initdb \
    --major-version=15 \
    --edition=se

======================================================================
Example for evaluation use (without login and password)
Only for Basic Edition
======================================================================

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

./db_installer.sh \
    --do-initdb \
    --major-version=15 \
    --edition=be

======================================================================
Examples how to install from file
======================================================================

./db_installer.sh \
    --from-file=./packages/tantor-be-server-15_15.4.1.jammy_amd64.deb

./db_installer.sh \
    --do-initdb \
    --from-file=/tmp/tantor-be-server-15_15.4.1.jammy_amd64.deb
```

При создании кластера инсталлятором включается подсчет контрольных сумм для блоков данных.

9) Установка адреса расположения дистрибутивов

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

10) Проверим переменные окружения:

```
root@tantor:~# cat /var/lib/postgresql/.bash_profile
#export PATH=/opt/tantor/db/16/bin:$PATH
export PGDATA=/var/lib/postgresql/tantor-se-16/data
export LC_MESSAGES=ru_RU.utf8
unset LANGUAGE
```

Если нужно, чтобы сообщения утилит был локализован, то нужно отредактировать файл профиля:
командой unset убрать переменную окружения LANGUAGE и установить LC_MESSAGES
Если не закомментировать строку PATH, то кластер будет создан и запущен из под той сборки, которая присутствует в пути.

Инсталлятор скачан, порт по умолчанию 5432 свободен, адрес репозитория с дистрибутивами установлен. Можно приступить к инсталляции. Можно приступить к установке.

11) Установка со скачиванием дистрибутива и созданием базы данных:

```
root@tantor:/home/astra# ./db_installer.sh --edition=be --major-version=15 --do-initdb
Hit:1 http://dl.astralinux.ru/astra/stable/1.7_x86-64/repository-base 1.7_x86-64
InRelease
Get:2 https://ftp.postgresql.org/pub/pgadmin/pgadmin4/apt/buster pgadmin4 InRelease
[4,217 B]
Get:3 https://ftp.postgresql.org/pub/pgadmin/pgadmin4/apt/buster pgadmin4/main amd64
Packages [9,948 B]
Get:4 https://ftp.postgresql.org/pub/pgadmin/pgadmin4/apt/buster pgadmin4/main all
Packages [6,483 B]
Fetched 20.6 kB in 0s (41.4 kB/s)
Reading package lists... Done
OK
deb [arch=amd64] https://nexus-public.tantorlabs.ru/repository/astra-smolensk-1.7
smolensk main
Hit:1 http://dl.astralinux.ru/astra/stable/1.7_x86-64/repository-base 1.7_x86-64
InRelease
Get:2 https://nexus-public.tantorlabs.ru/repository/astra-smolensk-1.7 smolensk
InRelease [1,556 B]
Hit:3 https://ftp.postgresql.org/pub/pgadmin/pgadmin4/apt/buster pgadmin4 InRelease
Get:4 https://nexus-public.tantorlabs.ru/repository/astra-smolensk-1.7 smolensk/main
amd64 Packages [3,491 B]
Fetched 5,047 B in 0s (10.4 kB/s)
Reading package lists... Done
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  tantor-be-server-15
0 upgraded, 1 newly installed, 0 to remove and 2 not upgraded.
Need to get 18.0 MB of archives.
After this operation, 0 B of additional disk space will be used.
Get:1 https://nexus-public.tantorlabs.ru/repository/astra-smolensk-1.7 smolensk/main
amd64 tantor-be-server-15 amd64 15.6.0 [18.0 MB]
Fetched 18.0 MB in 1s (19.6 MB/s)
Selecting previously unselected package tantor-be-server-15.
(Reading database ... 210711 files and directories currently installed.)
Preparing to unpack .../tantor-be-server-15_15.6.0_amd64.deb ...
+ echo ------------------------------------------
------------------------------------------
+ echo 'tantor-be-server-15 is getting installed'
tantor-be-server-15 is getting installed
+ echo ------------------------------------------
------------------------------------------
+ getent group postgres
+ getent passwd postgres
```

```
++ getent passwd postgres
++ awk -F: '{print $6}'
+ current_home=/var/lib/postgresql
+ '[' /var/lib/postgresql '!=' /var/lib/postgresql ']'
+ mkdir -p /var/lib/postgresql
+ chown postgres:postgres /var/lib/postgresql
+ chmod 700 /var/lib/postgresql
+ mkdir -p /var/run/postgresql
+ chown postgres:postgres /var/run/postgresql
+ '[' '!' -d /usr/lib/tmpfiles.d ']'
+ echo 'D /run/postgresql 0755 postgres postgres - -'
+ tee /usr/lib/tmpfiles.d/tantor-db.conf
+ mkdir -p /etc/ld.so.conf.d
+ tee /etc/ld.so.conf.d/tantor-be-15.conf
+ echo /opt/tantor/db/15/lib
+ cat /etc/ld.so.conf.d/tantor-be-15.conf
/opt/tantor/db/15/lib
+ echo ------------------------------------------
------------------------------------------
+ set +vx
Unpacking tantor-be-server-15 (15.6.0) ...
Setting up tantor-be-server-15 (15.6.0) ...
+ echo ------------------------------------------
------------------------------------------
+ echo 'tantor-be-server-15 is getting installed'
tantor-be-server-15 is getting installed
+ echo ------------------------------------------
------------------------------------------
+ /usr/sbin/ldconfig
+ /bin/systemctl daemon-reload
+ '[' '!' -f /var/lib/postgresql/.bash_profile ']'
+ '[' -f /var/lib/postgresql/.bash_profile ']'
++ grep /opt/tantor/db/15/bin /var/lib/postgresql/.bash_profile
+ '[' -z '' ']'
+ echo 'export PATH=/opt/tantor/db/15/bin:$PATH'
+ chown postgres:postgres /var/lib/postgresql/.bash_profile
+ '[' '!' -d /var/lib/postgresql/tantor-be-15/data ']'
+ '[' '!' -d /var/lib/postgresql/data ']'
+ mkdir -p /var/lib/postgresql/tantor-be-15/data
+ chown postgres:postgres /var/lib/postgresql/tantor-be-15/data
+ chmod 700 /var/lib/postgresql/tantor-be-15/data
+ echo ------------------------------------------
------------------------------------------
+ set +vx
Файлы, относящиеся к этой СУБД, будут принадлежать пользователю "postgres".
От его имени также будет запускаться процесс сервера.

Кластер баз данных будет инициализирован со следующими параметрами локали:
  провайдер:   libc
  LC_COLLATE:  en_US.UTF-8
  LC_CTYPE:    en_US.UTF-8
  LC_MESSAGES: ru_RU.utf8
  LC_MONETARY: en_US.UTF-8
  LC_NUMERIC:  en_US.UTF-8
  LC_TIME:     en_US.UTF-8
Кодировка БД по умолчанию, выбранная в соответствии с настройками: "UTF8".
Выбрана конфигурация текстового поиска по умолчанию "english".

Контроль целостности страниц данных включён.

исправление прав для существующего каталога /var/lib/postgresql/tantor-be-15/data...
ок
создание подкаталогов... ок
выбирается реализация динамической разделяемой памяти... posix
выбирается значение max_connections по умолчанию... 100
выбирается значение shared_buffers по умолчанию... 128MB
выбирается часовой пояс по умолчанию... Europe/Moscow
создание конфигурационных файлов... ок
выполняется подготовительный скрипт... ок
выполняется заключительная инициализация... ок
сохранение данных на диске... ок
```

```
initdb: предупреждение: включение метода аутентификации "trust" для локальных подключений
initdb: подсказка: Другой метод можно выбрать, отредактировав pg_hba.conf или ещё раз запустив initdb с ключом -A, --auth-local или --auth-host.

Готово. Теперь вы можете запустить сервер баз данных:

    /opt/tantor/db/15/bin/pg_ctl -D /var/lib/postgresql/tantor-be-15/data -l файл_журнала start

Created symlink /etc/systemd/system/multi-user.target.wants/tantor-be-server-15.service → /lib/systemd/system/tantor-be-server-15.service.
● tantor-be-server-15.service - Tantor Basic database server 15
   Loaded: loaded (/lib/systemd/system/tantor-be-server-15.service; enabled; vendor preset: enabled)
   Active: active (running) since 10:38:36 MSK; 26ms ago
     Docs: https://www.postgresql.org/docs/15/static/
  Process: 10564 ExecStartPre=/opt/tantor/db/15/bin/postgresql-check-db-dir ${PGDATA} (code=exited, status=0/SUCCESS)
  Process: 10566 ExecStart=/opt/tantor/db/15/bin/pg_ctl start -D ${PGDATA} -s -w -t ${PGSTARTTIMEOUT} (code=exited, status=0/SUCCESS)
 Main PID: 10568 (postgres)
    Tasks: 6 (limit: 4915)
   Memory: 16.9M
      CPU: 55ms
   CGroup: /system.slice/tantor-be-server-15.service
           ├─10568 /opt/tantor/db/15/bin/postgres -D /var/lib/postgresql/tantor-be-15/data
           ├─10569 postgres: checkpointer
           ├─10570 postgres: background writer
           ├─10572 postgres: walwriter
           ├─10573 postgres: autovacuum launcher
           └─10574 postgres: logical replication launcher

10:38:36 tantor systemd[1]: Starting Tantor Basic database server 15...
10:38:36 tantor pg_ctl[10566]: 2024-04-17 10:38:36.248 MSK [10568] LOG:  starting PostgreSQL 15.6 on x86_64-pc-linux-gnu, compiled by gcc (AstraLinuxSE 8.3.0-6) 8.3.0, 64-bit
10:38:36 tantor pg_ctl[10566]: 2024-04-17 10:38:36.249 MSK [10568] LOG:  listening on IPv6 address "::1", port 5434
10:38:36 tantor pg_ctl[10566]: 2024-04-17 10:38:36.250 MSK [10568] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5434"
10:38:36 tantor pg_ctl[10566]: 2024-04-17 10:38:36.254 MSK [10571] LOG:  database system was shut down at 2024-04-17 10:38:35 MSK
10:38:36 tantor pg_ctl[10566]: 2024-04-17 10:38:36.258 MSK [10568] LOG:  database system is ready to accept connections
10:38:36 tantor systemd[1]: Started Tantor Basic database server 15.


       tantor_version
-----------------------------
 Tantor Basic Edition 15.6.0
(1 row)


Installation successfully completed.
```

Если в переменной окружения PATH пользователя postgres (файлы профиля `/var/lib/postgresql/.bash_profile`) присутствовала директория другой сборки (`export PATH=/opt/tantor/db/16/bin:$PATH`), то кластер будет создан и запущен из под этой сборки (`Tantor Special Edition 16.1.0`)

12) Переключимся в пользователя postgres:

```
root@tantor:/home/astra# su - postgres
```

13) Проверяем, что путь к исполняемым файлам был добавлен в файл профиля пользователя postgres в конце файла:

```
postgres@tantor:~$ cat .bash_profile
#export PATH=/opt/tantor/db/16/bin:$PATH
export PGDATA=/var/lib/postgresql/tantor-se-16/data
export LC_MESSAGES=ru_RU.utf8
unset LANGUAGE
export PATH=/opt/tantor/db/15/bin:$PATH
```

14) Проверка что кластер работает:

```
postgres@tantor:~$ psql
psql (15.6)
Введите "help", чтобы получить справку.

postgres=# select version();
                                                  version
-----------------------------------------------------------------------------------------------------
 PostgreSQL 15.6 on x86_64-pc-linux-gnu, compiled by gcc (AstraLinuxSE 8.3.0-6) 8.3.0, 64-bit
(1 строка)

postgres=# \q
```

15) Вернемся в терминал root:

```
postgres@tantor:~$ exit
выход
root@tantor:/home/astra#
```

Демонстрация установки выполнена.

Часть 2. Деинсталляция

1) Остановим экземпляр кластера Тантор BE:

```
root@tantor:/home/astra# systemctl stop tantor-be-server-15
```

2) Запретим автоматический запуск службы:

```
root@tantor:/home/astra# systemctl disable tantor-be-server-15
Removed /etc/systemd/system/multi-user.target.wants/tantor-be-server-
15.service.
```

3) Посмотрим список установленного программного обеспечения Тантор:

```
root@tantor:/home/astra# apt list | grep tantor

WARNING: apt does not have a stable CLI interface. Use with caution in
scripts.

tantor-be-client-15/smolensk 15.6.0 amd64
tantor-be-libpq5-15/smolensk 15.6.0 amd64
tantor-be-server-14/smolensk 14.11.0 amd64
tantor-be-server-15/smolensk,now 15.6.0 amd64 [installed]
tantor-se-server-16/now 16.1.0 amd64 [installed,local]
```

4) Деинсталлируем то, что установили:

```
root@tantor:/home/astra# apt remove tantor-be-server-15/smolensk
Reading package lists... Done
Building dependency tree
Reading state information... Done
Selected version '15.6.0' (smolensk [amd64]) for 'tantor-be-server-15'
```

```
The following packages will be REMOVED:
  tantor-be-server-15
0 upgraded, 0 newly installed, 1 to remove and 2 not upgraded.
After this operation, 0 B of additional disk space will be used.
Do you want to continue? [Y/n] Y
(Reading database ... 213906 files and directories currently installed.)
Removing tantor-be-server-15 (15.6.0) ...
+ echo ----------------------------------------
+ echo 'tantor-be-server-15 is getting removed'
tantor-be-server-15 is getting removed
+ echo ----------------------------------------
+ /bin/systemctl --no-reload disable tantor-be-server-15
+ /bin/systemctl stop tantor-be-server-15
+ echo ----------------------------------------
----------------------------------------
+ set +vx
+ echo ----------------------------------------
+ echo 'tantor-be-server-15 is getting removed'
tantor-be-server-15 is getting removed
+ echo ----------------------------------------
+ /usr/sbin/ldconfig
+ /bin/systemctl daemon-reload
+ '[' -f /var/lib/postgresql/.bash_profile ']'
++ grep /opt/tantor/db/15/bin /var/lib/postgresql/.bash_profile
+ '[' '!' -z 'export PATH=/opt/tantor/db/15/bin:$PATH' ']'
+ sed -i 's|/opt/tantor/db/15/bin:*||g' /var/lib/postgresql/.bash_profile
+ sed -i '/^PATH=:*\($PATH\)*:*$/d' /var/lib/postgresql/.bash_profile
+ /usr/sbin/ldconfig
+ echo ----------------------------------------
+ set +vx
```

5) Проверим как изменился список установленного программного обеспечения Танатор:

```
root@tantor:/home/astra# apt list | grep tantor

WARNING: apt does not have a stable CLI interface. Use with caution in
scripts.

tantor-be-client-15/smolensk 15.6.0 amd64
tantor-be-libpq5-15/smolensk 15.6.0 amd64
tantor-be-server-14/smolensk 14.11.0 amd64
tantor-be-server-15/smolensk,now 15.6.0 amd64 [residual-config]
tantor-se-server-16/now 16.1.0 amd64 [installed,local]

Пакет деинсталлирован, но конфигурационные файлы были оставлены.
```

6) Посмотрим есть ли ещё пакеты residual config:

```
root@tantor:/home/astra# aptitude search ~c
c   tantor-be-server-15                                  - Tantor Basic
database server installation package
```

7) Удалим эти пакеты:

```
root@tantor:/home/astra# aptitude purge ~c
The following packages will be REMOVED:
  tantor-be-server-15{p}
0 packages upgraded, 0 newly installed, 1 to remove and 2 not upgraded.
Need to get 0 B of archives. After unpacking 0 B will be used.
Do you want to continue? [Y/n/?] Y
(Reading database ... 210713 files and directories currently installed.)
Purging configuration files for tantor-be-server-15 (15.6.0) ...
+ echo ----------------------------------------
----------------------------------------
```

```
+ echo 'tantor-be-server-15 is getting removed'
tantor-be-server-15 is getting removed
+ echo ----------------------------------------
+ /usr/sbin/ldconfig
+ /bin/systemctl daemon-reload
+ '[' -f /var/lib/postgresql/.bash_profile ']'
++ grep /opt/tantor/db/15/bin /var/lib/postgresql/.bash_profile
+ '[' '!' -z '' ']'
+ /usr/sbin/ldconfig
+ echo ----------------------------------------
+ set +vx
dpkg: warning: while removing tantor-be-server-15, directory
'/opt/tantor/db/15/lib' not empty so not removed
```

8) В директории `/opt/tantor/db/15/lib` лежит осиротевшая символическая ссылка:

```
root@tantor:/home/astra# ls -l --color /opt/tantor/db/15/lib/
total 0
lrwxrwxrwx 1 root root libzstd.so.1 -> libzstd.so.1.5.5
```

9) Удалим директорию:

```
root@tantor:/home/astra# rm -rf /opt/tantor/db/15
```

10) Директория созданного при инсталляции кластера BE не была удалена. Удалим её:

```
root@tantor:/home/astra# rm -rf /var/lib/postgresql/tantor-be-15
```

11) Проверим ещё раз список пакетов:

```
root@tantor:/home/astra# apt list | grep tantor

WARNING: apt does not have a stable CLI interface. Use with caution in
scripts.

tantor-be-client-15/smolensk 15.6.0 amd64
tantor-be-libpq5-15/smolensk 15.6.0 amd64
tantor-be-server-14/smolensk 14.11.0 amd64
tantor-be-server-15/smolensk 15.6.0 amd64
tantor-se-server-16/now 16.1.0 amd64 [installed,local]
```

12) Файл `/var/lib/postgresql/.bash_profile` выглядит так:

```
root@tantor:/home/astra# cat /var/lib/postgresql/.bash_profile
#export PATH=/opt/tantor/db/16/bin:$PATH
export PGDATA=/var/lib/postgresql/tantor-se-16/data
export LC_MESSAGES=ru_RU.utf8
unset LANGUAGE
export PATH=$PATH
```

13) Уберите из файла комментарий (знак #) и бесполезную строку.
Файл должен выглядеть так:

```
root@tantor:/home/astra# cat /var/lib/postgresql/.bash_profile
export PATH=/opt/tantor/db/16/bin:$PATH
export PGDATA=/var/lib/postgresql/tantor-se-16/data
export LC_MESSAGES=ru_RU.utf8
unset LANGUAGE
```

14) Запустим кластера 16 версии, которые останавливали:

```
root@tantor:/home/astra# systemctl start tantor-se-server-16
root@tantor:/home/astra# systemctl start tantor-se-server-16-replica
```

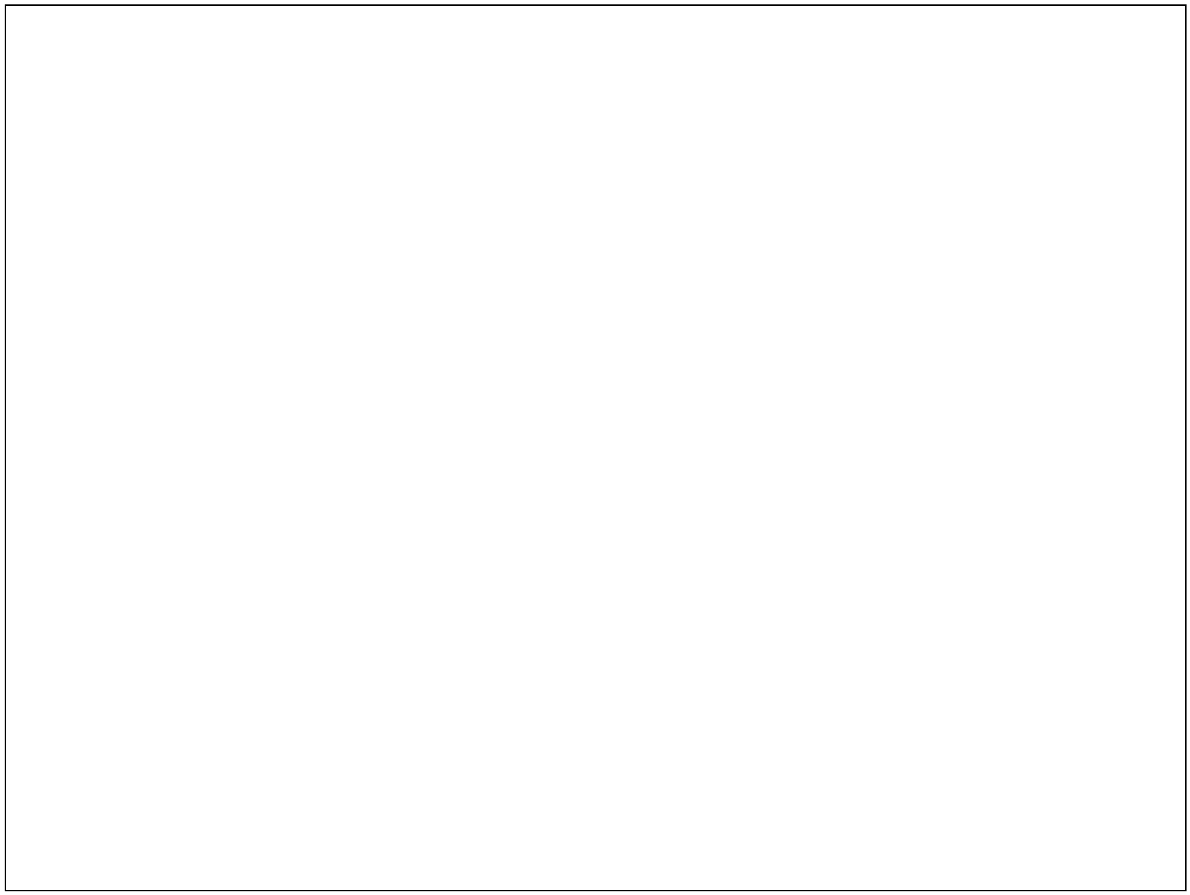
15) Переключимся в пользователя postgres и проверим что кластер работоспособен:

```
root@tantor:/home/astra# su - postgres
postgres@tantor:~$ psql
```

```
psql (16.1)
Введите "help", чтобы получить справку.

postgres=# select tantor_version();
        tantor_version
------------------------------
 Tantor Special Edition 16.1.0
(1 строка)

postgres=# \q
```

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

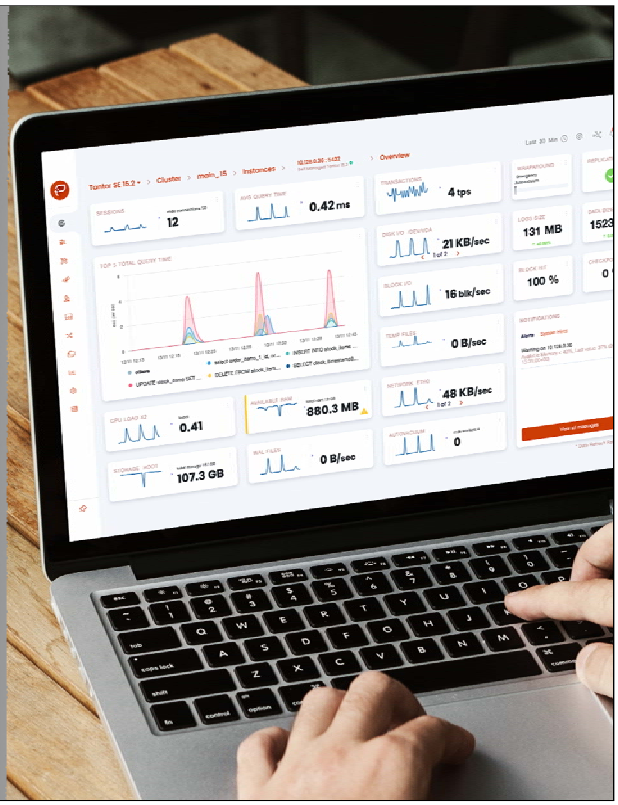


СУБД Tantor

Архитектура

Темы

- Общие сведения
  - Клиент-сервер
  - Транзакции
  - Фоновые процессы
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость



Определение

«Архитектур клиент-сервер»:

- ключевая модель взаимодействия в PostgreSQL
- один компьютер (сервер) предоставляет ресурсы, другие (клиенты) запрашивают их

«Сервер PostgreSQL»:

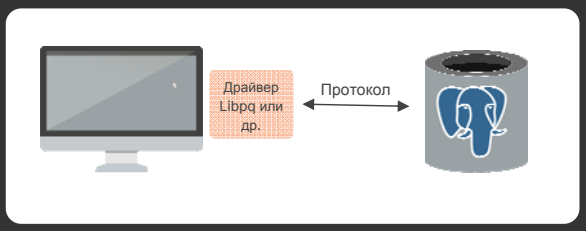
- отвечает за управление базой данных
- обрабатывает запросы от клиентов, обеспечивает безопасность и целостность данных

«Клиентские приложения»:

- приложения, взаимодействующие с сервером для операций с данными
- отправляют SQL-запросы, обрабатывают результаты

«Протоколы обмена данными»:

- используется TCP/IP для обмена данными
- между клиентами и сервером
- обеспечивает безопасность и эффективность взаимодействия



Архитектура клиент-сервер — это модель взаимодействия между компьютерами, где один компьютер (сервер) предоставляет ресурсы или услуги, а другие компьютеры (клиенты) запрашивают и используют эти ресурсы. В контексте СУБД, такой как PostgreSQL, это означает, что база данных управляется сервером, а клиентские приложения обращаются к серверу для выполнения операций с данными.

Клиенты — это приложения или пользовательские интерфейсы, которые взаимодействуют с сервером для выполнения операций с данными. Клиенты отправляют SQL-запросы серверу, получают результаты и обрабатывают их. Клиенты могут быть написаны на различных языках программирования и работать на различных устройствах.

Сервер базы данных (PostgreSQL сервер) — PostgreSQL сервер предоставляет доступ к базе данных и обрабатывает запросы от клиентов. Он управляет хранением данных, обеспечивает безопасность и целостность данных, управляет транзакциями и обеспечивает возможность одновременной работы нескольких клиентов. Сервер слушает определенный сетевой порт и ждет запросы от клиентов.

Клиентские приложения — Клиенты устанавливают подключение к серверу, и каждое подключение создает сеанс работы. В рамках сеанса клиент может отправлять SQL-запросы, а сервер выполняет их и возвращает результаты. Сеанс поддерживает состояние, что позволяет серверу отслеживать информацию о текущем состоянии подключенных клиентов.

Протоколы обмена данными — Для обмена данными между клиентами и сервером используются определенные протоколы, такие как TCP/IP. Обычно PostgreSQL использует протокол передачи данных по сети (TCP/IP), который обеспечивает безопасное и эффективное взаимодействие между клиентом и сервером.

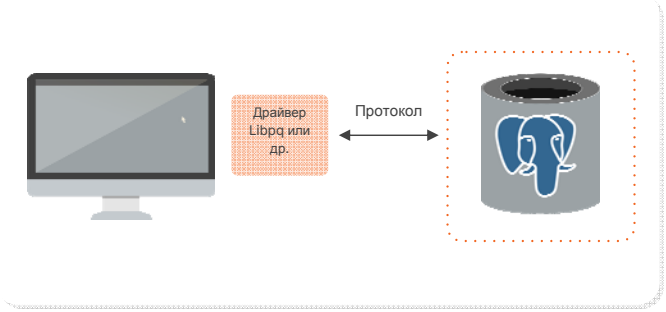
Подключение и сеансы:

Архитектура клиент-сервер обеспечивает многозадачность, так что несколько клиентов могут одновременно взаимодействовать с сервером. Каждый клиент работает в своем собственном сеансе, и сервер управляет выполнением их запросов параллельно.

Применение архитектуры клиент-сервер позволяет эффективно распределять роли и ответственности между сервером и клиентами, что делает систему более гибкой, масштабируемой и обеспечивает более эффективное управление данными в распределенной среде.

Функции сервера

- управление хранением данных
- обработка SQL-запросов
- управление транзакциями
- многопользовательская поддержка
- безопасность
- оптимизация запросов
- управление индексами
- работа с триггерами и хранимыми процедурами и функциями
- управление пользователями и ролями
- мониторинг и журналирование



Сервер PostgreSQL выполняет множество важных функций для эффективного управления базой данных и обеспечения безопасности, целостности данных и производительности. Основные функции сервера включают управление хранением данных, обработку SQL-запросов и транзакционное управление.

Управление хранением данных означает создание, изменение и удаление файлов данных на уровне операционной системы, связанных с таблицами, индексами и другими объектами базы данных.

Сервер PostgreSQL также ответственен за обработку SQL-запросов, поступающих от клиентских приложений. Это включает в себя анализ запросов, оптимизацию и выполнение операций непосредственно на уровне базы данных. Встроенный оптимизатор запросов выбирает наилучший план выполнения запроса, учитывая структуру данных, индексы и другие факторы, что повышает общую производительность.

Сервер PostgreSQL обеспечивает транзакционное управление, что позволяет объединять несколько операций в единую транзакцию, обеспечивая целостность данных. Этот механизм также дает возможность отката изменений в случае возникновения ошибок.

Кроме того, сервер обеспечивает многопользовательскую поддержку, параллельно обрабатывая запросы от различных клиентов и управляя конфликтами доступа к данным. Безопасность также играет ключевую роль, и сервер регулирует доступ пользователей с использованием механизмов аутентификации и авторизации.

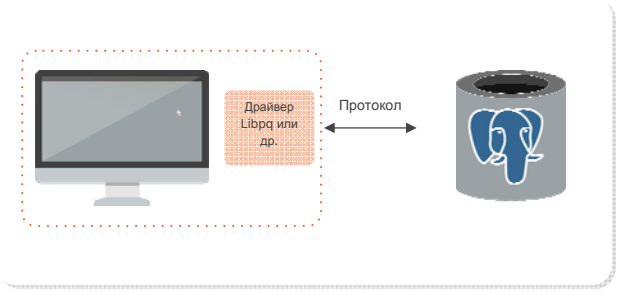
Сервер PostgreSQL также управляет индексами для ускорения поиска и выборки данных, а также предоставляет средства для создания и обслуживания триггеров и хранимых процедур, расширяя функциональность базы данных.

Дополнительно, сервер осуществляет управление пользователями и ролями, позволяя администраторам назначать права доступа. Функции мониторинга и журналирования также обеспечивают контроль за работой сервера и отслеживание изменений.

Все эти элементы в совокупности делают сервер PostgreSQL мощным инструментом для работы с базами данных, обеспечивая надежное и эффективное управление информацией в разнообразных сценариях использования.

Функции сервера

- установка соединения
- отправка SQL-запросов
- обработка результатов запросов
- управление транзакциями
- обработка ошибок
- работа с подготовленными запросами
- управление соединением
- обеспечение безопасности
- мониторинг и журналирование



Клиентское приложение взаимодействует с сервером PostgreSQL, выполняя различные функции, необходимые для работы с базой данных. Одной из основных функций является установка соединения с сервером, что обеспечивает возможность передачи SQL-запросов.

Клиент отправляет SQL-запросы, ожидая от сервера результатов их выполнения. Эти результаты, как правило, содержат данные, которые клиентское приложение должно обработать, отобразить пользователю или использовать в дальнейшей логике программы.

Важной задачей клиента является управление транзакциями, включая начало, фиксацию и откат транзакций в соответствии с логикой приложения. Помимо этого, клиент обрабатывает возможные ошибки, возникающие в результате выполнения запросов.

Для повышения производительности клиент может использовать подготовленные запросы, а также эффективно управлять соединениями с сервером. Это включает в себя открытие, закрытие и переиспользование соединений в соответствии с требованиями приложения.

Важно также отметить, что клиент может обеспечивать безопасность, включая аутентификацию пользователей и шифрование данных.

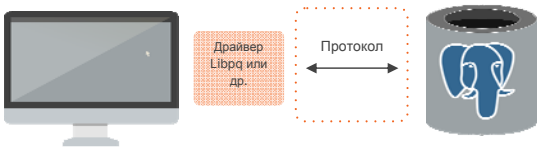
Наконец, клиент может осуществлять мониторинг работы приложения и журналирование для отслеживания действий пользователя и решения возможных проблем.

Таким образом, функции клиента включают в себя широкий спектр операций, направленных на эффективное взаимодействие с сервером PostgreSQL и обеспечение безопасности и производительности приложения.

Архитектура / Общие сведения / Клиент-сервер

Протокол подключения

- установка соединения
- безопасность и аутентификация
- управление сеансом
- поддержка различных функциональных возможностей
- оптимизация и эффективность
- поддержка различных клиентских приложений
- работа с дополнительными функциональными возможностями



Tantor SE — в библиотеку libpq добавлена поддержка сжатия.

Протокол подключения в базах данных, таких как PostgreSQL, является ключевым элементом для эффективного и безопасного обмена данными между клиентским приложением и сервером. Он обеспечивает ряд важных функций, начиная с установления соединения, где определены правила обмена необходимой начальной информацией.

Одним из важных аспектов протокола подключения является обеспечение безопасности и аутентификации. Он позволяет серверу проверить подлинность клиента, гарантируя, что взаимодействие происходит между доверенными сторонами, и обеспечивает защиту передаваемых данных.

Протокол также управляет сеансом работы, включая установку параметров сеанса и управление временем ожидания. Это позволяет эффективно управлять взаимодействием между клиентом и сервером, оптимизируя процессы передачи данных.

Определение формата запросов и ответов также является функциональной обязанностью протокола подключения. Он обеспечивает поддержку различных возможностей, таких как выполнение SQL-запросов, управление транзакциями и обработка ошибок.

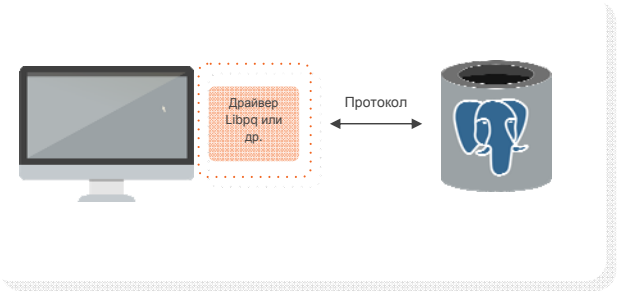
Протокол подключения может быть оптимизирован для эффективной передачи данных, включая использование сжатия, бинарных форматов и других техник для улучшения производительности. Такой подход позволяет снизить объем передаваемой информации и оптимизировать процессы взаимодействия.

Имея в виду расширенные функциональные возможности, протокол подключения поддерживает различные клиентские приложения, написанные на разных языках программирования и работающие на различных платформах.

Таким образом, протокол подключения является фундаментальным элементом для обеспечения стабильной, безопасной и эффективной связи между клиентскими приложениями и серверами баз данных в различных сценариях использования.

Драйвер libpq

- установка соединения
- отправка SQL-запросов
- обработка результатов
- управление транзакциями
- обработка ошибок
- подготовленные запросы
- параметры соединения и конфигурация
- безопасность и аутентификация
- многозадачность и асинхронность



Драйвер **libpq** — это библиотека на языке программирования «C», предназначенная для обеспечения эффективного взаимодействия клиентских приложений с сервером PostgreSQL.

Установка соединения является первоначальным шагом, где **libpq** предоставляет функции для указания параметров, таких как хост, порт, база данных, пользователь и пароль.

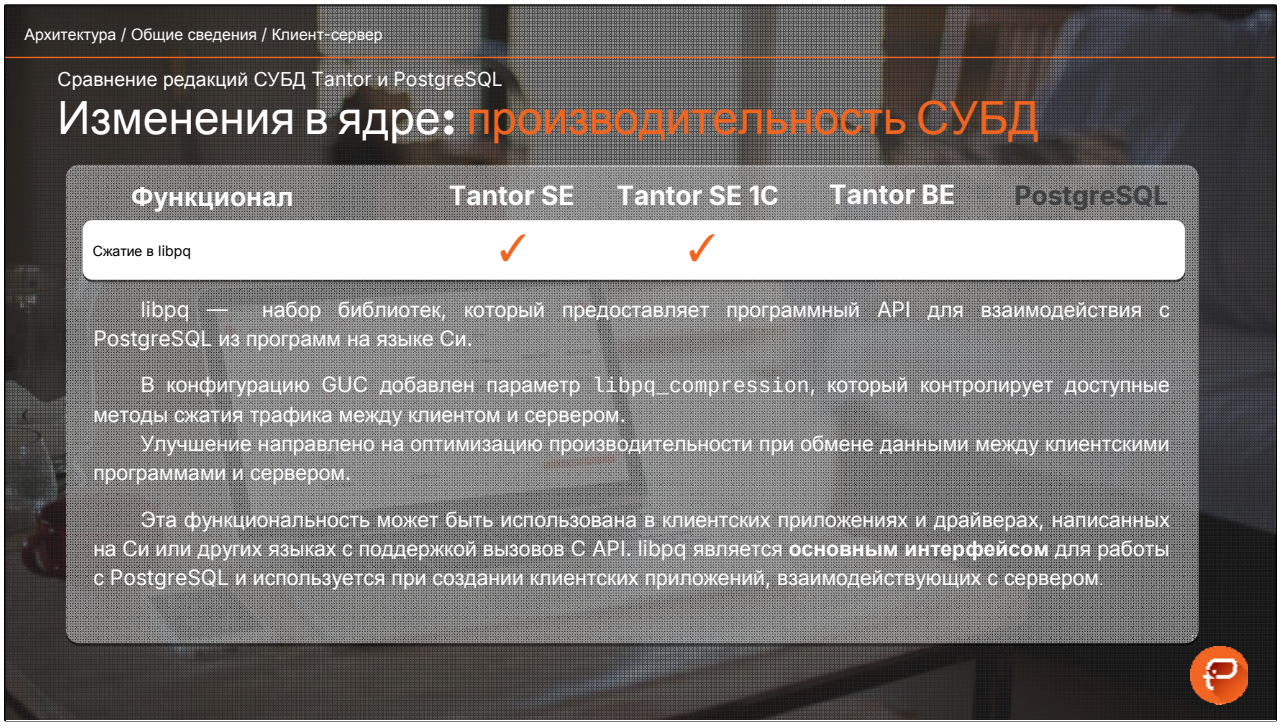
Одной из ключевых возможностей драйвера является отправка SQL-запросов на сервер PostgreSQL. Это важно для выполнения различных операций, включая выборку, вставку, обновление и удаление данных. Полученные после выполнения запросов результаты могут быть обработаны с использованием механизмов, предоставленных **libpq**. Здесь важно выделить момент эффективной обработки ошибок в случае их возникновения.

Для обеспечения целостности данных, драйвер **libpq** предоставляет функции для управления транзакциями. Это включает в себя начало, фиксацию и откат транзакции в зависимости от логики клиентского приложения.

Для повышения производительности, libpq поддерживает подготовленные запросы, что позволяет многократно использовать SQL-запросы с различными параметрами. Важно также отметить возможности настройки параметров соединения и конфигурации для оптимальной работы с базой данных.

Драйвер **libpq** обеспечивает взаимодействие с механизмами безопасности и аутентификации PostgreSQL, обеспечивая защиту соединения и данных. Кроме того, библиотека поддерживает асинхронное взаимодействие с сервером PostgreSQL, что является важной особенностью для обработки множества одновременных соединений.

В целом, **libpq** — важный инструмент для разработчиков на языке программирования «C», предоставляющий эффективные средства взаимодействия с базой данных PostgreSQL



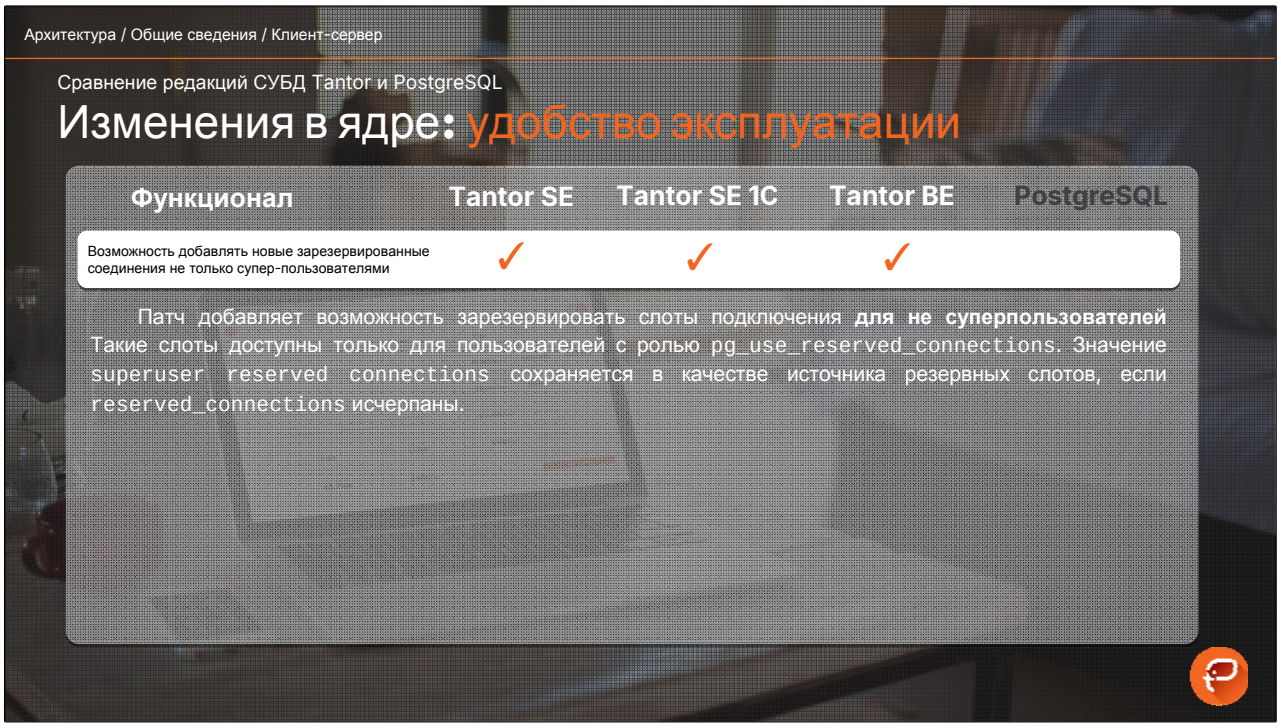
В TantorSE и Tantor SE 1C внедрено сжатие в `libpq`.

В библиотеку libpq добавлена поддержка сжатия.

В конфигурацию (GUC) добавлен параметр `libpq_compression`, который контролирует доступные методы сжатия трафика между клиентом и сервером. Улучшение направлено на оптимизацию производительности при обмене данными между клиентскими программами и сервером.

Поскольку `libpq` представляет собой C API для PostgreSQL, эту функциональность могут использовать клиентские приложения и драйверы, которые написаны на «C» или на других языках, поддерживающих вызовы C API.

Библиотека `libpq` является основным интерфейсом для работы с PostgreSQL и используется для создания «клиентских» приложений, которые взаимодействуют с сервером PostgreSQL.

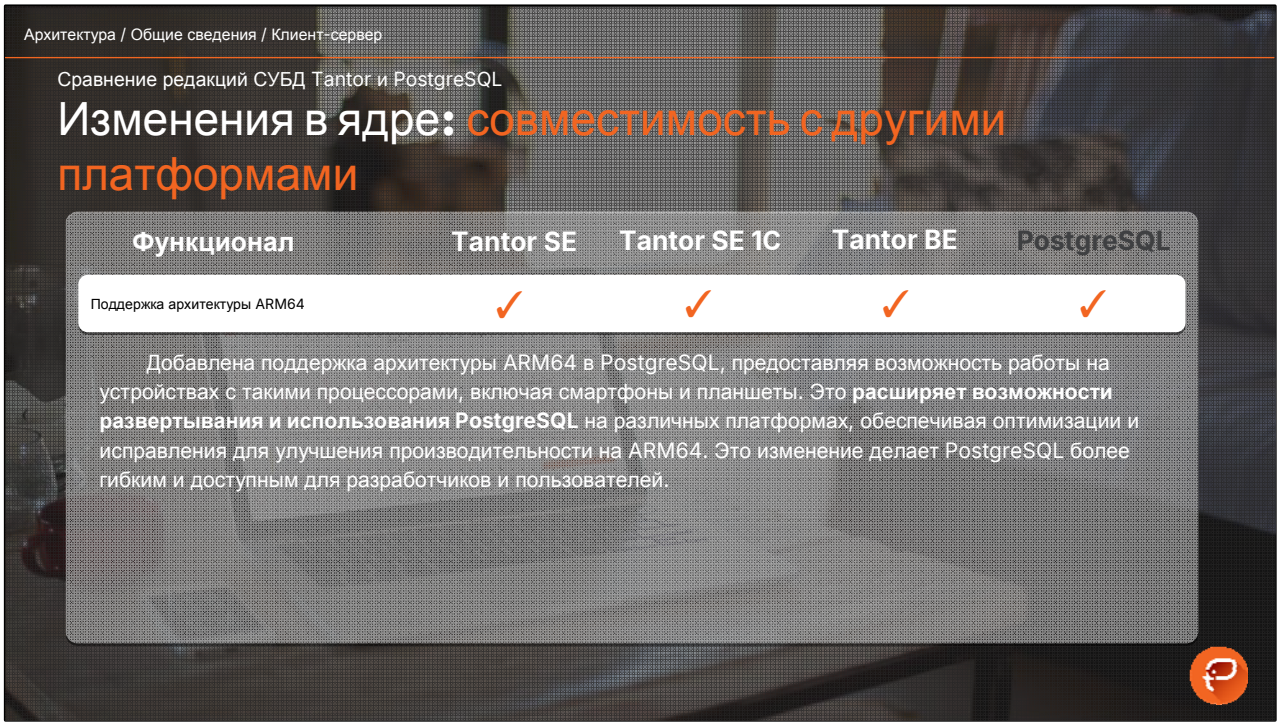


Для всех версий Tantor доступно добавление новых зарезервированных соединений для пользователей, не являющихся супер-пользователями.

Патч позволяет зарезервировать слоты подключения для не суперпользователей.

Слоты, зарезервированные через новый **GUC (Grand Unified Configuration)** `reserved_connections` (параметр postgresql.conf), доступны только пользователям с новой предопределенной ролью `pg_use_reserved_connections`.

Значение параметра `superuser_reserved_connections` остается в качестве окончательного резерва на случай, если `reserved_connections` были исчерпаны.



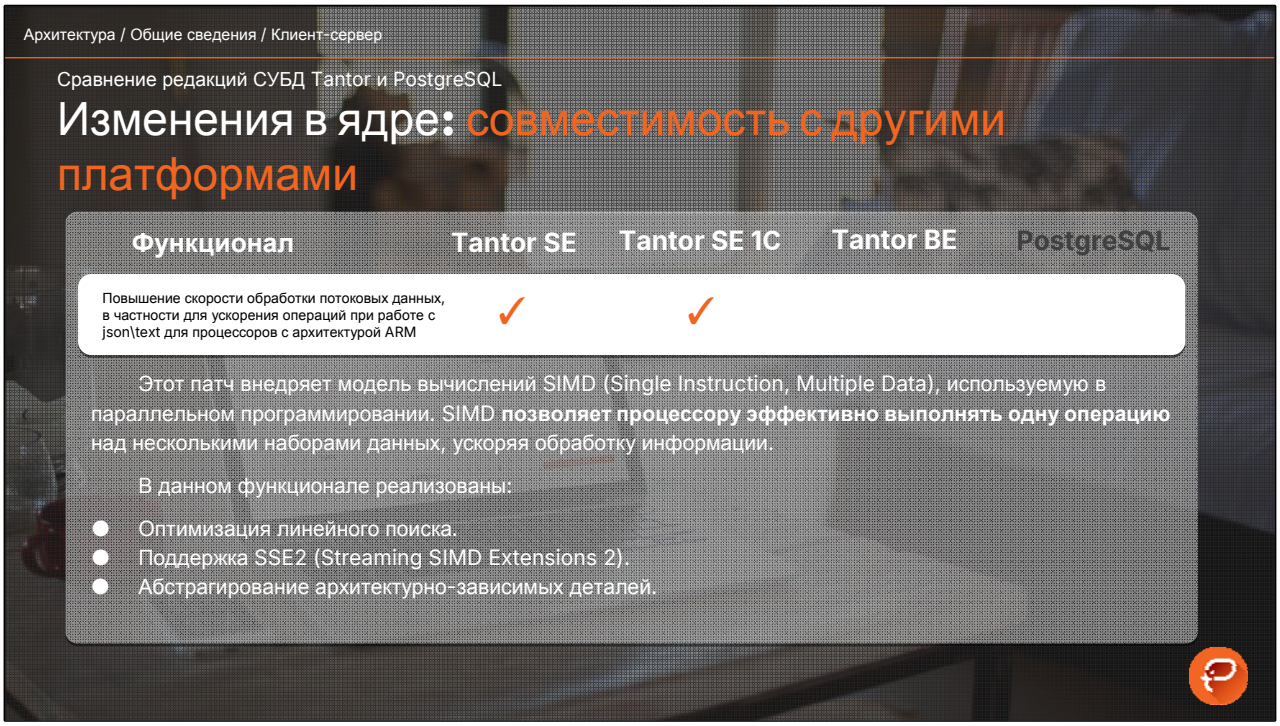
Во всех версиях Tantor и в PostgreSQL присутствует поддержка архитектуры ARM64.

Поддержка архитектуры ARM64 — это изменение, которое позволяет PostgreSQL работать на процессорах с архитектурой ARM64. Ранее PostgreSQL поддерживал только процессоры с архитектурой x86 и x86-64.

Поддержка архитектуры ARM64 в PostgreSQL открывает новые возможности для запуска PostgreSQL на устройствах с такими процессорами, например на смартфонах и планшетах. Это может быть полезно для разработки приложений, которые требуют локальной базы данных на таких устройствах.

Это изменение включает оптимизации и исправления ошибок, связанные с работой PostgreSQL на архитектуре ARM64. Также с PostgreSQL 14 произведены оптимизации для улучшения производительности на ARM64.

В целом, поддержка архитектуры ARM64 в PostgreSQL расширяет возможности развертывания и использования этой базы данных на различных устройствах и платформах. Это делает PostgreSQL еще более гибким и доступным для разработчиков и пользователей.



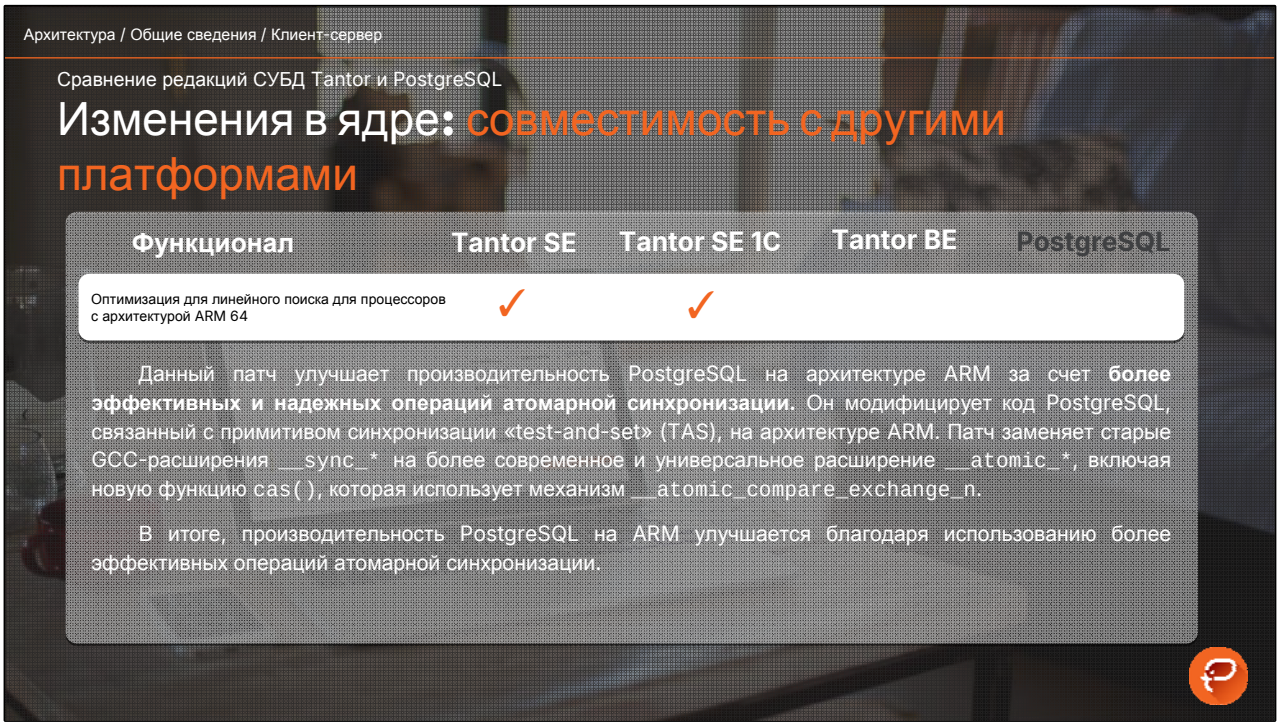
В редакциях Tantor SE и Tantor SE 1C в сравнении с ванильным PostgreSQL реализовано повышение скорости обработки потоковых данных, особенно для ускорения операций при работе с json\text для процессоров с архитектурой ARM.

Патч направлен на использование модели вычислений SIMD (Single Instruction, Multiple Data), которая используется в параллельном программировании. Основное преимущество SIMD заключается в том, что она позволяет процессору одновременно выполнять одну и ту же операцию сразу над несколькими наборами данных, что может значительно ускорить обработку данных.

Что реализовано в данном функционале:

1. Оптимизация линейного поиска. Добавлена новая функция, которая оптимизирует линейный поиск в данных, что может ускорить некоторые операции обработки данных.
2. Поддержка SSE2 (Streaming SIMD Extensions 2). Добавляются внутренние компоненты, которые позволяют использовать инструкции SSE2 там, где это возможно. Это может ускорить обработку данных, особенно при работе с текстовыми и JSON данными.
3. Абстрагирование архитектурно-зависимых деталей. Патч включает поддержку для архитектуры NEON. Это расширяет возможности использования SIMD на большем количестве платформ.
4. Поддержка SIMD NEON. Патч включает использование встроенных функций SIMD NEON, что повышает производительность на архитектурах, поддерживающих NEON.

Данные изменения улучшают производительность PostgreSQL за счет более эффективного использования ресурсов процессора при обработке данных. Особенно они могут быть полезны при работе с большими объемами данных или при выполнении сложных операций обработки данных, таких как обработка текстовых и JSON данных.



В редакциях Tantor SE и Tantor SE 1C также реализована оптимизация для линейного поиска для процессоров с архитектурой ARM 64.

Патч помогает повысить производительность PostgreSQL на машинах с архитектурой ARM, используя более эффективные и надёжные операции атомарной синхронизации.

Данный патч модифицирует код PostgreSQL, связанный с реализацией примитива синхронизации «test-and-set» (TAS), также известного как «atomic compare and swap» (CAS), на архитектуре ARM. Он меняет реализацию TAS с использованием старого GCC-расширения `__sync_*` на более современное и универсальное расширение `__atomic_*`, включая новую функцию `cas()`, которая использует `__atomic_compare_exchange_n` вместо предыдущего механизма TAS.

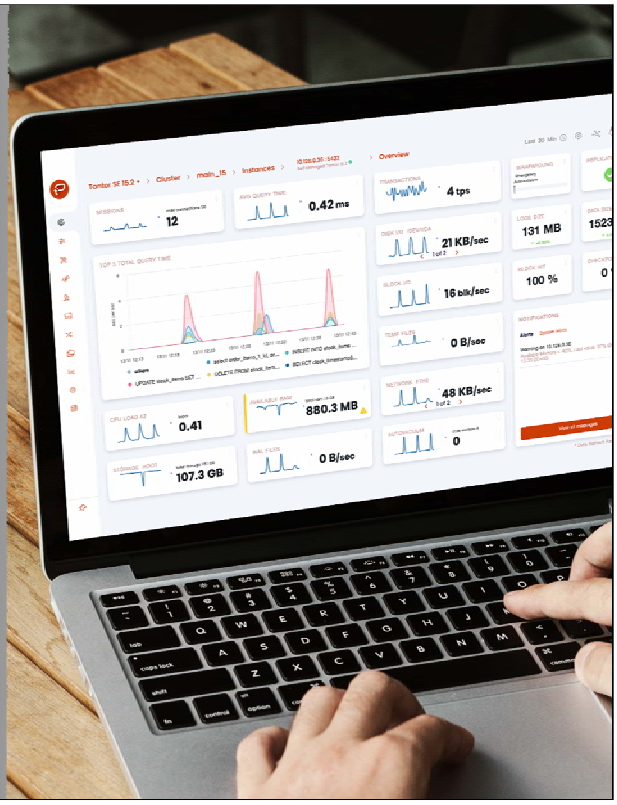
Темы

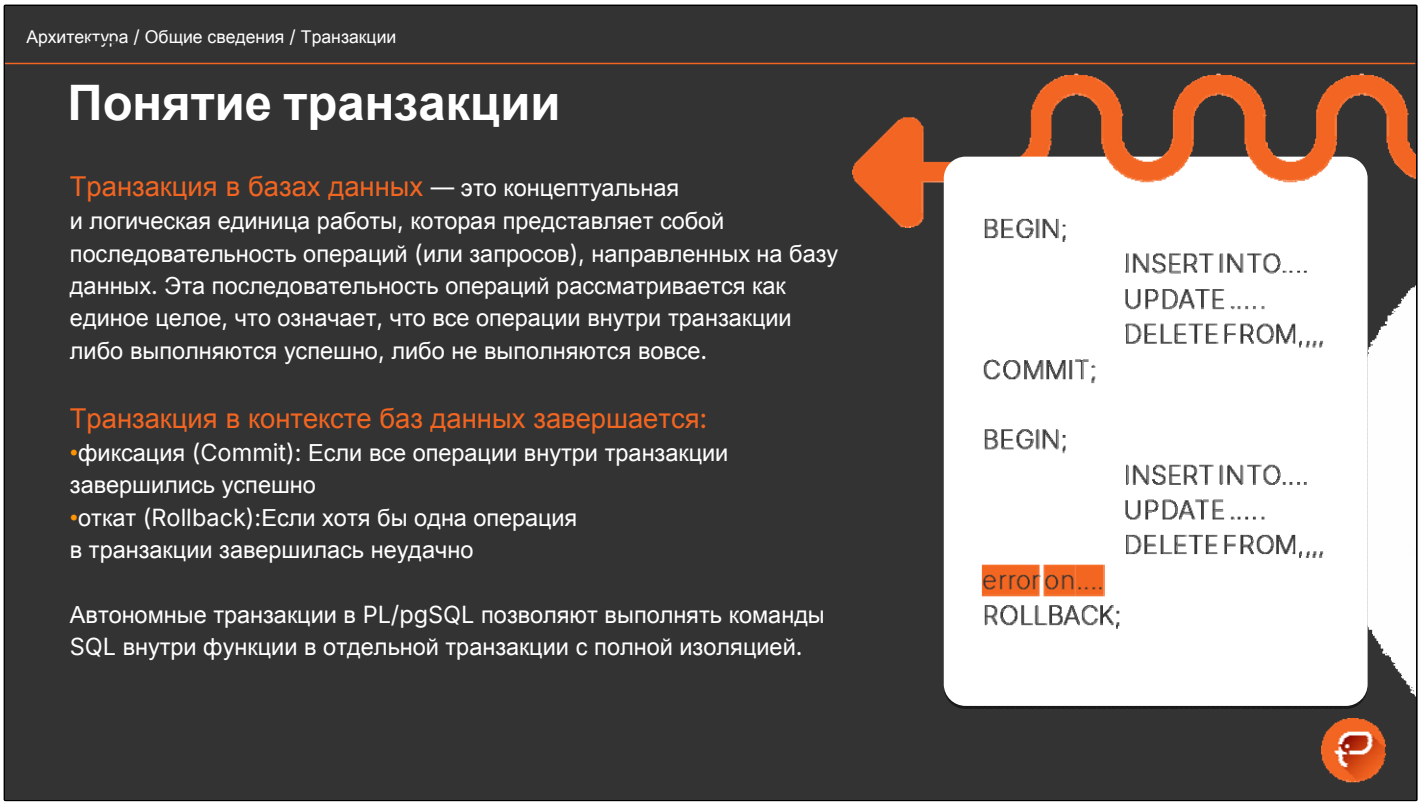
- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Клиент-сервер

Транзакции

Фоновые процессы





Транзакция в базах данных — это концептуальная и логическая единица работы, которая представляет собой последовательность операций (или запросов), направленных на базу данных. Эта последовательность операций рассматривается как единое целое, что означает, что все операции внутри транзакции либо выполняются успешно, либо не выполняются вовсе.

Транзакция в контексте баз данных завершается фиксацией (commit) или откатом (rollback).

●Фиксация (Commit): Если все операции внутри транзакции завершились успешно, и система прошла проверку на соответствие критериям атомарности, согласованности, изоляции и долговечности, то транзакция фиксируется. В этот момент все изменения, внесенные транзакцией, становятся постоянными и видимыми для других транзакций.

●Откат (Rollback): Если хотя бы одна операция в транзакции завершилась неудачно или система не прошла проверку на соответствие критериям, то транзакция откатывается. Все изменения, внесенные в рамках транзакции, отменяются, и база данных возвращается к предыдущему согласованному состоянию.

Таким образом, завершение транзакции происходит с фиксацией изменений в базе данных в случае успешного выполнения или с откатом изменений в случае неудачи. Этот момент является ключевым для обеспечения целостности и надежности данных в системе базы данных.

Также возможно использование автономных транзакций https://docs.tantorlabs.ru/tdb/ru/15_4/se/autonomous-transactions.html

Автономные транзакции в PL/pgSQL позволяют выполнять SQL-запросы внутри функции в отдельной транзакции с полной изоляцией.

Основные свойства транзакции:

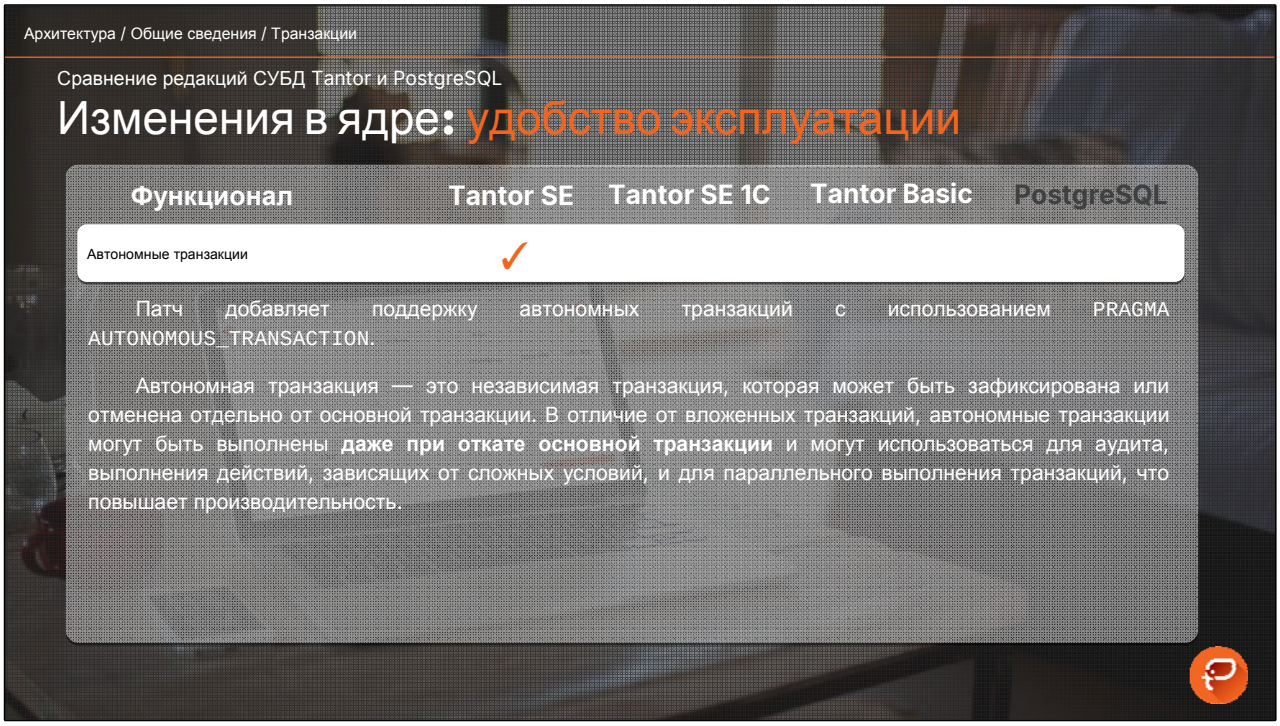
1.Атомарность (Atomicity): Транзакция является атомарной, что означает, что все операции внутри транзакции либо успешно выполняются, либо не выполняются вообще. Если хотя бы одна операция в транзакции завершается неудачно, то все изменения, внесенные предыдущими операциями, откатываются, восстанавливая базу данных в прежнее состояние.

2.Согласованность (Consistency): Транзакция должна приводить базу данных из одного согласованного состояния в другое. Если база данных была в согласованном состоянии до начала транзакции, она должна остаться в согласованном состоянии после ее завершения.

3.Изолированность (Isolation): Изоляция гарантирует, что выполнение одной транзакции не влияет на выполнение других транзакций. Другие транзакции не видят промежуточные изменения текущей транзакции, пока она не завершится.

4.Долговечность (Durability): После успешного завершения транзакции ее результаты остаются в базе данных даже в случае сбоя системы или отключения питания. Это гарантирует, что система может быть восстановлена к последнему согласованному состоянию после сбоя.

Примеры транзакций включают в себя операции вставки, обновления или удаления записей в базе данных. Транзакции обеспечивают надежность и целостность данных, предоставляя механизм для гарантированного выполнения группы операций как единого целого.



Поговорим об улучшениях в СУБД Тантор направленных на удобство использования.

Для версии SE улучшения включают в себя автономные транзакции.

Патч добавляет поддержку автономных транзакций с использованием синтаксиса `PRAGMA AUTONOMOUS_TRANSACTION`.

Автономная транзакция — это независимая транзакция, инициированная в рамках основной транзакции. В отличие от вложенных транзакций, которые могут быть зафиксированы только совместно с транзакцией, к которой они принадлежат, автономные транзакции должны быть зафиксированы или отменены до окончания основной (родительской) транзакции.

Вложенные транзакции используются для обработки ошибок и в хранимых процедурах. Автономные транзакции требуются для:

- осуществления аудита, когда следует отметить попытку выполнения главной транзакции;
- для выполнения действий, которые зависят от сложных условий и не должны влиять на состояние основной транзакции;
- действия должны оставаться в силе, даже если основная транзакция откатывается;
- параллельное выполнение транзакций с основной транзакцией, что может улучшить производительность в некоторых сценариях.

Темы

- Общие сведения
  - Клиент-сервер
  - Транзакции
  - Фоновые процессы
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

`postmaster` в PostgreSQL — это основной процесс управления сервером базы данных. Этот процесс отвечает за запуск и управление другими процессами PostgreSQL, в том числе рабочими процессами, обработкой запросов, управлением подключениями к базе данных и обеспечением целостности данных.

Когда вы запускаете PostgreSQL, процесс `postmaster` стартует первым. Он слушает определенный порт (по умолчанию 5432) и ожидает входящих запросов от клиентов. Когда клиент отправляет запрос на подключение к базе данных, **`postmaster`** обрабатывает это подключение, создает новый рабочий процесс (называемый **backend**), который будет обслуживать запросы этого клиента.

`postmaster` также отвечает за управление процессами фонового обслуживания, такими как архивирование журналов, проверка целостности базы данных, сбор статистики и другие административные задачи.

Важно отметить, что начиная с PostgreSQL версии 16 и выше, термин `postmaster` официально заменен на `postgres`, но в некоторых случаях оба термина могут использоваться.

В PostgreSQL выполняются ключевые служебные процессы, обеспечивающие эффективное управление и поддержание базы данных.

Все эти процессы функционируют в оперативной памяти (ОЗУ), обеспечивая нормальную работу системы.

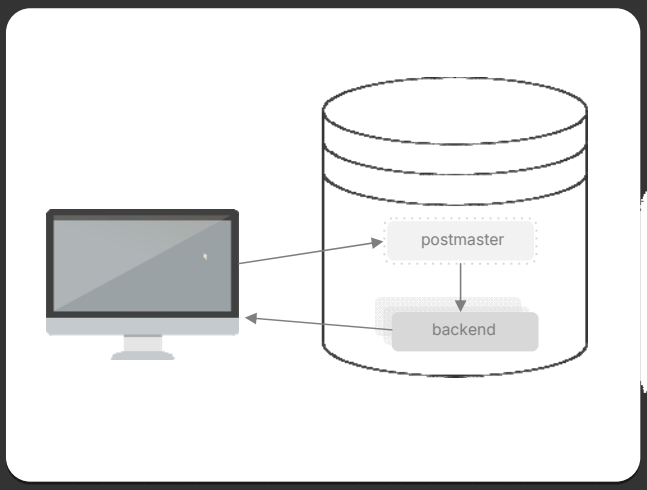
- **Управление подключениями:** Процесс следит за активными сессиями пользователей, обеспечивая безопасное управление доступом и координацию подключений.
- **Архивирование журнала транзакций**: Фоновый процесс регулярно сохраняет изменения данных в журнале, обеспечивая восстановление после сбоев и сохранение целостности данных.
- **Фоновая очистка**: Процесс удаляет устаревшие или удаленные записи, оптимизируя пространство в таблицах и улучшая производительность.
- **Мониторинг статистики**: Процесс собирает данные о работе базы данных, предоставляя ценные сведения для мониторинга и оптимизации производительности.
- **Фоновая запись:** Процесс записи изменений данных на диск обеспечивает устойчивость и предотвращает потерю данных при сбоях.
- **Управление сессиями репликации**: Процесс обеспечивает передачу изменений между мастер-сервером и репликами, поддерживая согласованность данных в распределенной среде.
- **Управление сетевыми подключениями**: Процесс обрабатывает входящие и исходящие соединения, обеспечивая стабильную коммуникацию между клиентами и сервером базы данных.

Эти служебные процессы взаимодействуют, обеспечивая надежное и эффективное функционирование PostgreSQL баз данных. Конкретное название и назначение пройдем позже.

Пользовательские процессы

В PostgreSQL пользовательские процессы часто называются «backend-процессами» или просто «backend».
Каждый подключенный клиент (пользователь) к базе данных запускает свой собственный backend-процесс на сервере PostgreSQL.

Tantor SE — увеличение производительности системы управления базами данных (СУБД) при большом количестве одновременных пользователей.



В PostgreSQL термин «пользовательские процессы» обычно относится к процессам, которые управляются и выполняются от имени конкретного пользователя базы данных. Когда пользователь подключается к PostgreSQL, сервер создает процесс для обслуживания этого подключения. Этот процесс работает от имени пользователя, который выполнил вход.

Пользовательские процессы выполняют запросы и другие операции в базе данных от имени соответствующего пользователя. Эти процессы обрабатывают SQL-запросы, выполняют транзакции, взаимодействуют с данными и выполняют другие задачи, связанные с обработкой запросов от пользователей.

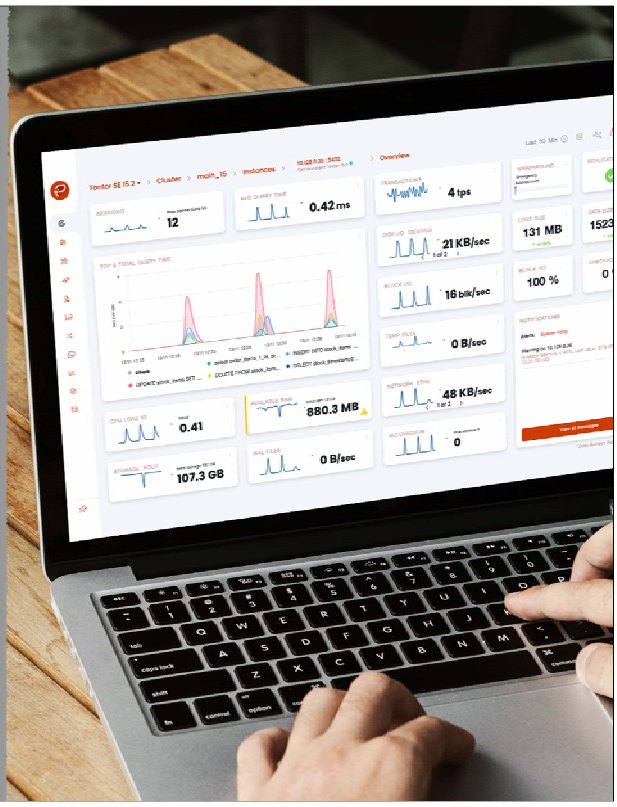
В PostgreSQL пользовательские процессы часто называются «**backend-процессами**» или просто «**backend**». Каждый подключенный клиент (пользователь) к базе данных запускает свой собственный backend-процесс на сервере PostgreSQL. Этот процесс отвечает за выполнение запросов и обработку других операций, предоставляя данные пользователю. Важно отметить, что управление пользователями, их правами доступа и другими аспектами безопасности также влияет на то, как пользовательские процессы взаимодействуют с базой данных. Каждый пользователь имеет определенные привилегии, определяющие, что он может делать в контексте базы данных.

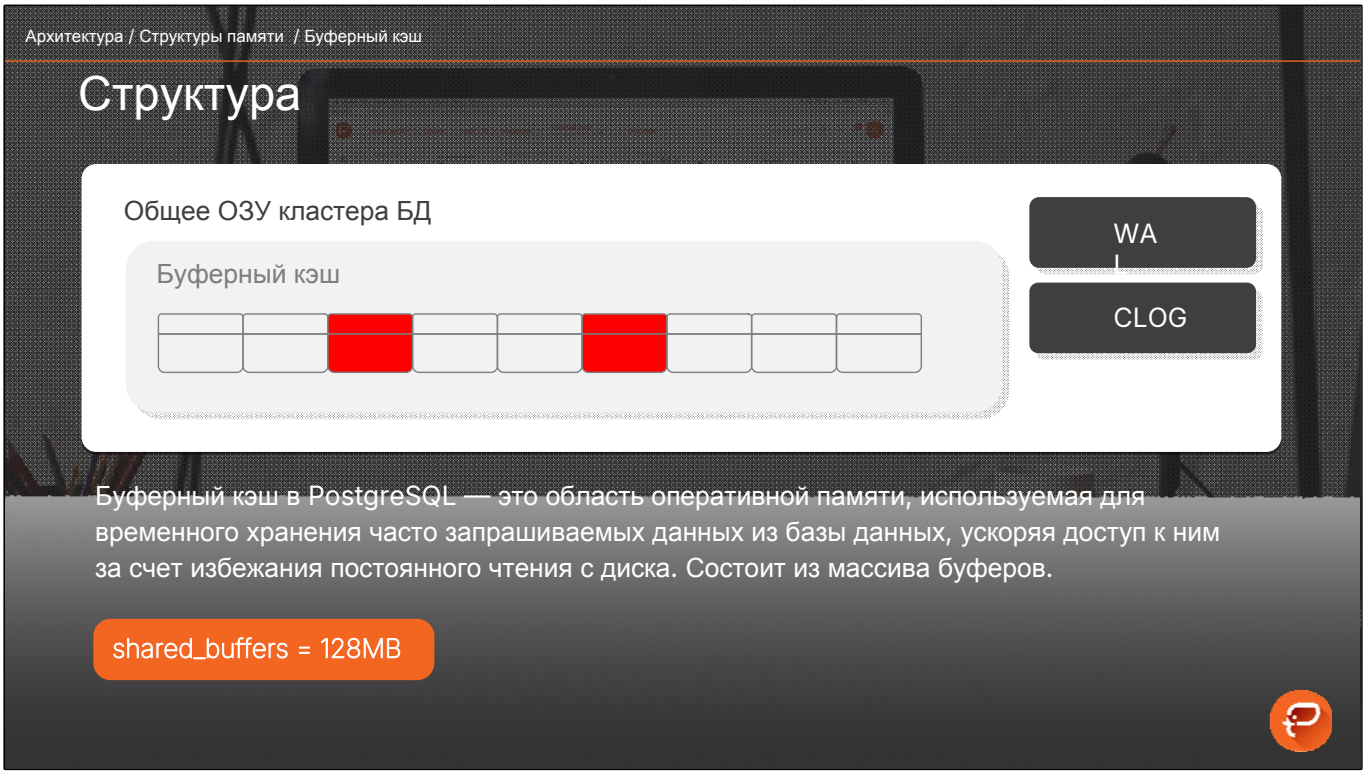
Tantor SE — увеличение производительности системы управления базами данных (СУБД) при большом количестве одновременных пользователей.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/differences.html

Темы

- Общие сведения
- Структуры памяти
  - Буферный кэш
  - WAL
  - Контрольная точка
  - Демонстрация
  - Практическая работа
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость





Область общей памяти предназначена для временного хранения данных таблиц, индексов и различных служебных структур. Термин «общая» отражает тот факт, что все процессы в системе управления базами данных имеют одинаковый уровень доступа к этой области, и изменения, внесенные одним из процессов, автоматически становятся доступными для всех остальных. Обычно основную часть этой общей памяти занимает буфер, также известный как буферный кеш, который служит для временного хранения пользовательских данных.

Буферный кэш в PostgreSQL — это область оперативной памяти, используемая для временного хранения часто запрашиваемых данных из базы данных, ускоряя доступ к ним за счет избежания постоянного чтения с диска. Состоит из массива буферов.

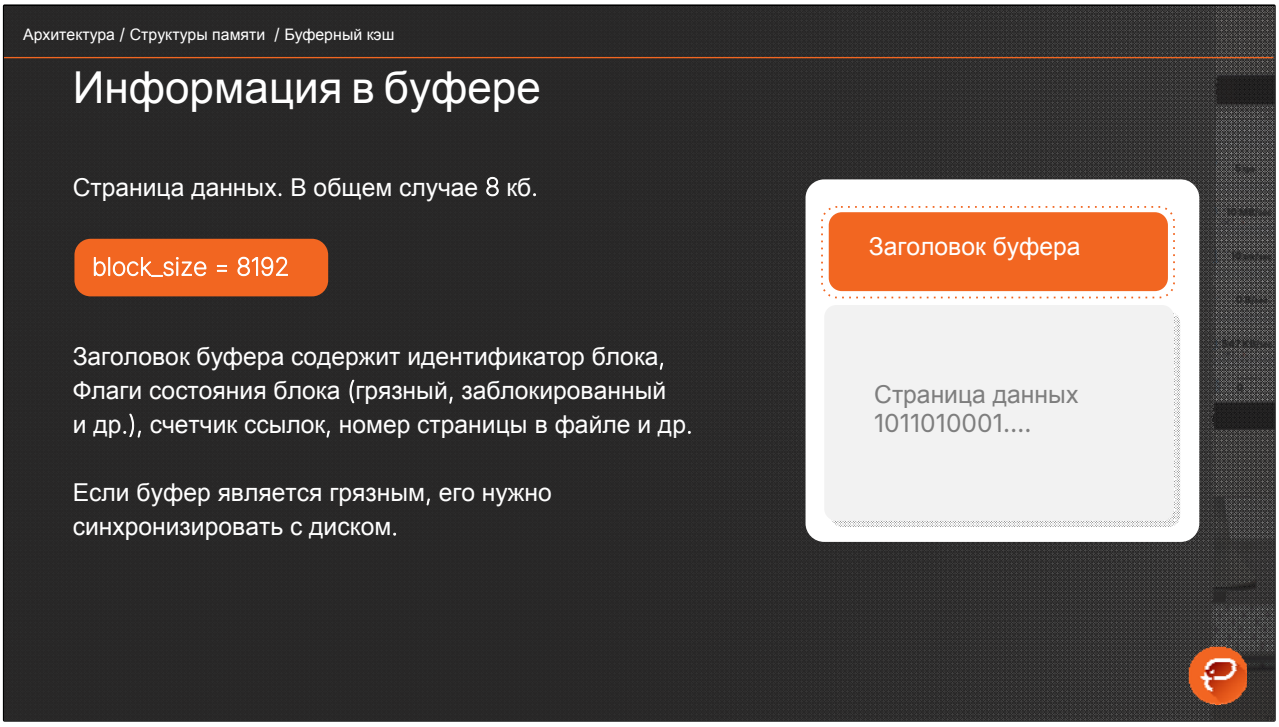
В PostgreSQL величина буферного кэша определяется параметром конфигурации под названием shared_buffers. Этот параметр задает количество памяти (в килобайтах), выделенной для буферного кэша.

Пример в конфигурационном файле postgresql.conf:

```
shared_buffers = 128MB
```

В данном случае, буферный кэш будет занимать 128 мегабайт оперативной памяти. Размер буферного кэша влияет на производительность системы: увеличение его размера может улучшить производительность при частых обращениях к данным, но слишком большой размер может привести к уменьшению производительности из-за больших накладных расходов на управление кэшем.

При настройке PostgreSQL важно учитывать общий объем доступной оперативной памяти и балансировать размер буферного кэша с другими параметрами, чтобы достичь оптимального сочетания производительности и эффективного использования ресурсов.



Заголовок буфера в PostgreSQL включает в себя метаданные, необходимые для управления содержимым буферного кэша. Метаданные обеспечивают информацию о состоянии и содержании блока данных в буфере.

Заголовок буфера содержит следующие элементы:

- **Идентификатор блока (Block Identifier)**: Уникальный идентификатор блока данных определяет, к какой таблице или индексу относится данный блок.
- **Флаги состояния блока (Block State Flags)**: Метаданные, указывающие на текущее состояние блока в буфере, такие как «заморожен» (frozen), «грязный» (dirty) или «заблокированный» (pinned).
- **Счетчик ссылок (Reference Counter)**: Число, отражающее количество активных ссылок на данный блок в буфере. Это может использоваться для определения, насколько часто блок используется и может быть вытеснен из кэша.
- **Номер страницы в файле (Page Number):** Информация о том, на какой странице файла данных или индекса находится данный блок.
- **Метка времени (Timestamp)**: Временная метка, указывающая время последнего доступа к данному блоку.

Эти элементы заголовка буфера предоставляют PostgreSQL информацию, необходимую для эффективного управления и использования буферного кэша.

Размер страницы данных устанавливается такой же, как и страница данных на диске. Проверить размер можно с помощью параметра `block_size`.

Если буфер является грязным, его нужно синхронизировать с диском.

В PostgreSQL, как и во многих других системах управления базами данных, механизмы вытеснения в буферном кэше используются для управления ограниченным объемом оперативной памяти и эффективного хранения часто используемых данных. В PostgreSQL применяется алгоритм **Least Recently Used (LRU)** для механизма вытеснения данных из буферного кэша.

В PostgreSQL каждый блок данных в буфере получает временную метку, отражающую момент последнего доступа к данному блоку. Когда система обнаруживает, что буфер заполняется и требуется освободить место, механизм LRU выбирает блок данных, который дольше всего не использовался . Такой блок данных считается менее актуальным, и его вытеснение имеет меньший потенциальный негативный эффект на производительность.

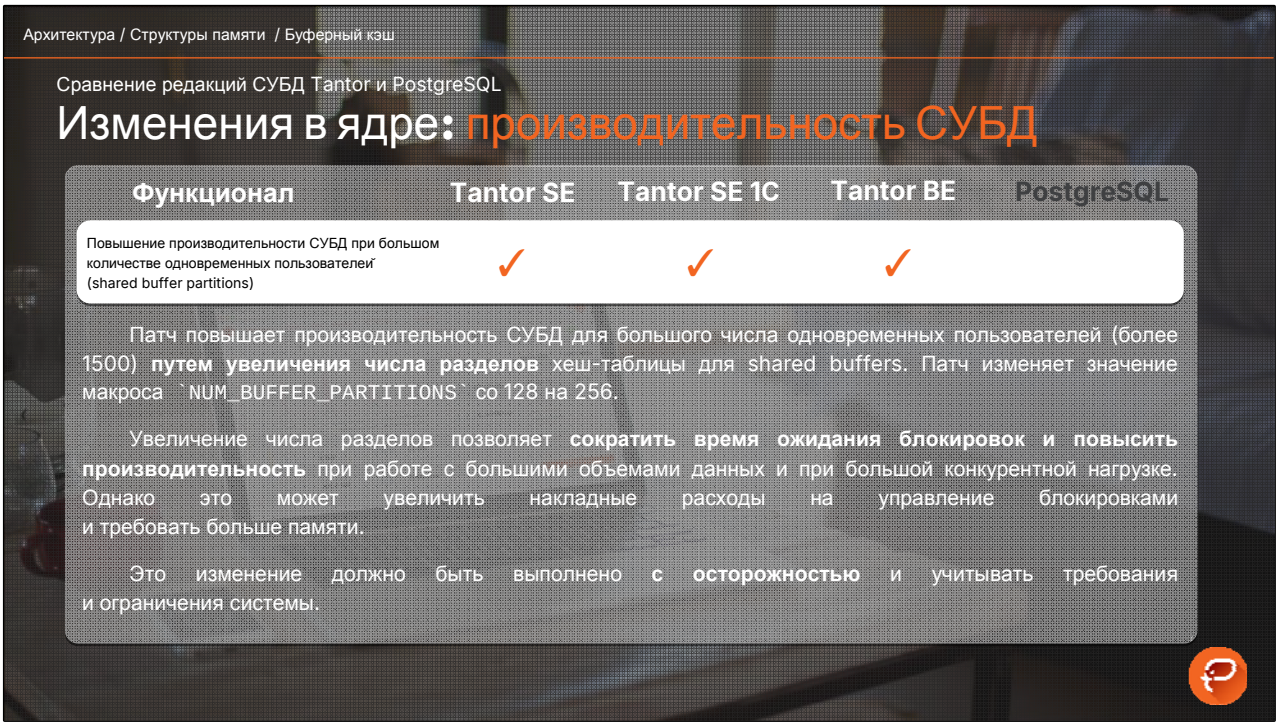
Когда буферный кэш начинает заполняться, механизмы вытеснения в PostgreSQL становятся активными, чтобы освободить место для новых данных. Это важный аспект управления буферным кэшем, чтобы сберегать оперативную память и эффективно использовать её ресурсы.

Несколько ключевых механизмов включают:

●**Least Recently Used (LRU)**: Этот подход основан на принципе, что менее используемые блоки данных более вероятно будут вытеснены. Каждый блок данных в буферном кэше получает временную метку последнего использования, и тот, который был использован давно, имеет больше шансов быть вытесненным при необходимости.

●**Clock Sweep**: Этот метод использует аналог часов, где каждому блоку данных присваивается бит-флаг. Процесс обходит блоки данных, сбрасывая флаги, и блок с определенным флагом вытесняется. Этот метод предоставляет баланс между эффективностью и простотой реализации.

Итак, когда процесс находит буфер с использованием по статистике ноль, то можно его вытеснить из буферного кэша для загрузки новой страницы. В примере буфер является грязным, поэтому перед вытеснением нужно сбросить его содержимое на диск. Только потом можно читать новую информацию.



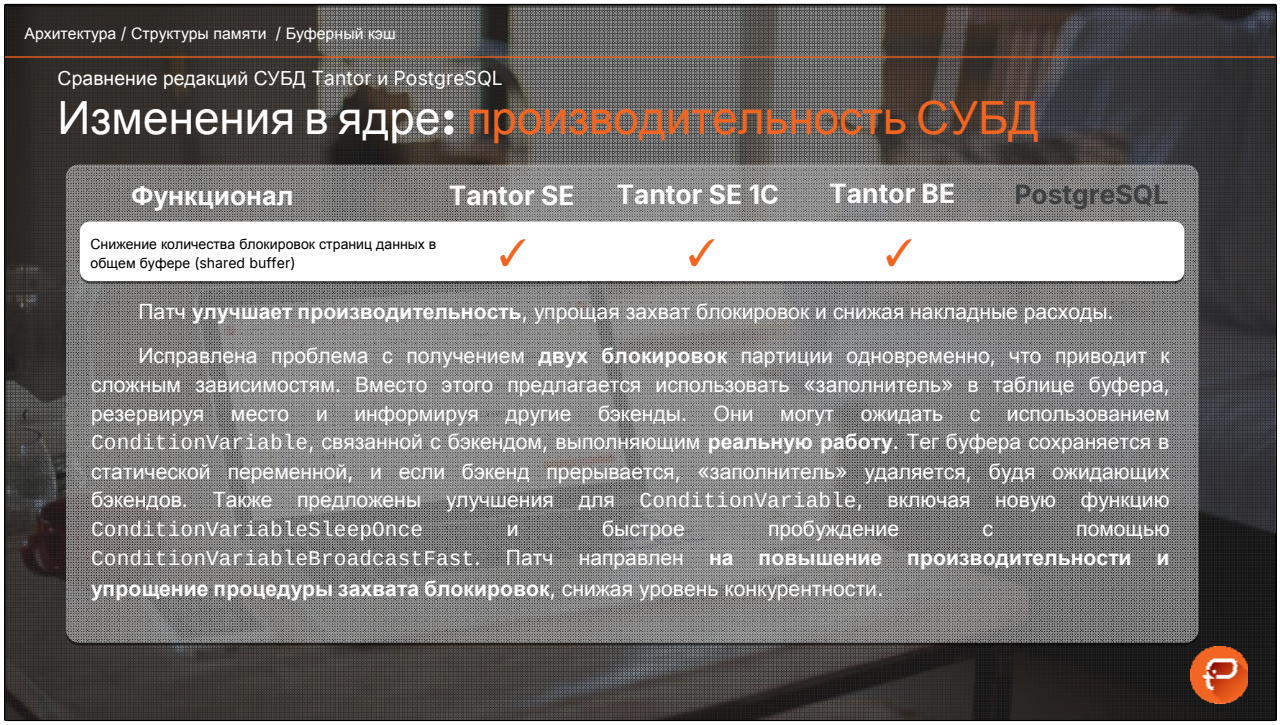
Во всех версиях всех СУБД Tantor улучшено функционирование СУБД при большом количестве одновременных пользователей (`shared buffer partitions`).

Патч повышает производительность СУБД при большом количестве одновременных пользователей (более 1500) путем увеличения числа разделов хеш-таблицы для общего буфера — shared buffers. Shared buffer — это основной буфер, который PostgreSQL использует для кэширования данных из дискового хранилища.

Этот патч изменяет значение макроса `NUM_BUFFER_PARTITIONS` со 128 на 256. Макрос определяет число разделов хеш-таблицы для общего буфера. Это позволяет сократить время ожидания блокировок, когда множество потоков или процессов пытаются получить доступ к данным в shared buffers.

Увеличение числа разделов может повысить производительность при работе с большими объемами данных и при большой конкурентной нагрузке, поскольку каждое разделение может быть заблокировано независимо. Это позволяет сократить время ожидания блокировок, когда множество потоков или процессов пытаются получить доступ к данным в shared buffer.

С другой стороны, увеличение числа разделов может повысить накладные расходы на управление этими блокировками и может потребовать больше памяти для хранения дополнительной метаинформации. Это изменение должно быть проведено с осторожностью и с учетом конкретных требований и ограничений системы.



Во всех версиях СУБД Tantor снижено количество блокировок страниц данных в общем буфере (shared buffer)

Патч направлен на улучшение производительности путем упрощения процедуры захвата блокировок и снижения накладных расходов, связанных с управлением блокировками.

1. Устранение проблемы с получением двух блокировок разделов одновременно. Это создает сложную цепочку зависимостей, которая может привести к проблемам при высоком уровне параллелизма.

2. Вместо этого предлагается вначале вставить «заполнитель» в таблицу буфера. Этот «заполнитель» будет резервировать место в таблице и информировать другие бэкенды о намерении выделить этот буфер.

3. Другие бэкенды могут ожидать, используя `ConditionVariable`, связанную с бэкендом, выполняющим реальную работу.

4. Тег буфера, над которым работает бэкенд, сохраняется в статическую переменную. Если бэкенд по какой-либо причине прерывается, он удаляет «заполнитель» и будит ожидающие бэкенды.

5. В патче также предлагаются некоторые улучшения `ConditionVariable` для повышения производительности:

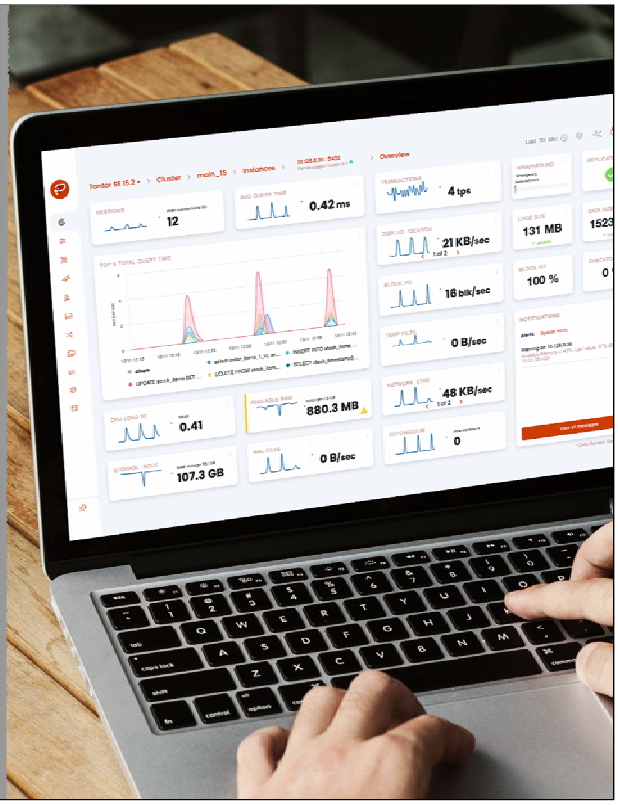
- Позволяет пропустить ConditionVariableCancelSleep(void), используя новую функцию ConditionVariableSleepOnce(cv). ConditionVariableSleepOnce не может быть использован в цикле, так как он не возвращает процесс обратно в `ConditionVariable`. Однако его безопасно использовать в данном случае, поскольку мы собираемся повторить попытку с полноценной функцией `LWLockAcquire+BufferLookup`, и скорее всего, она будет успешной.

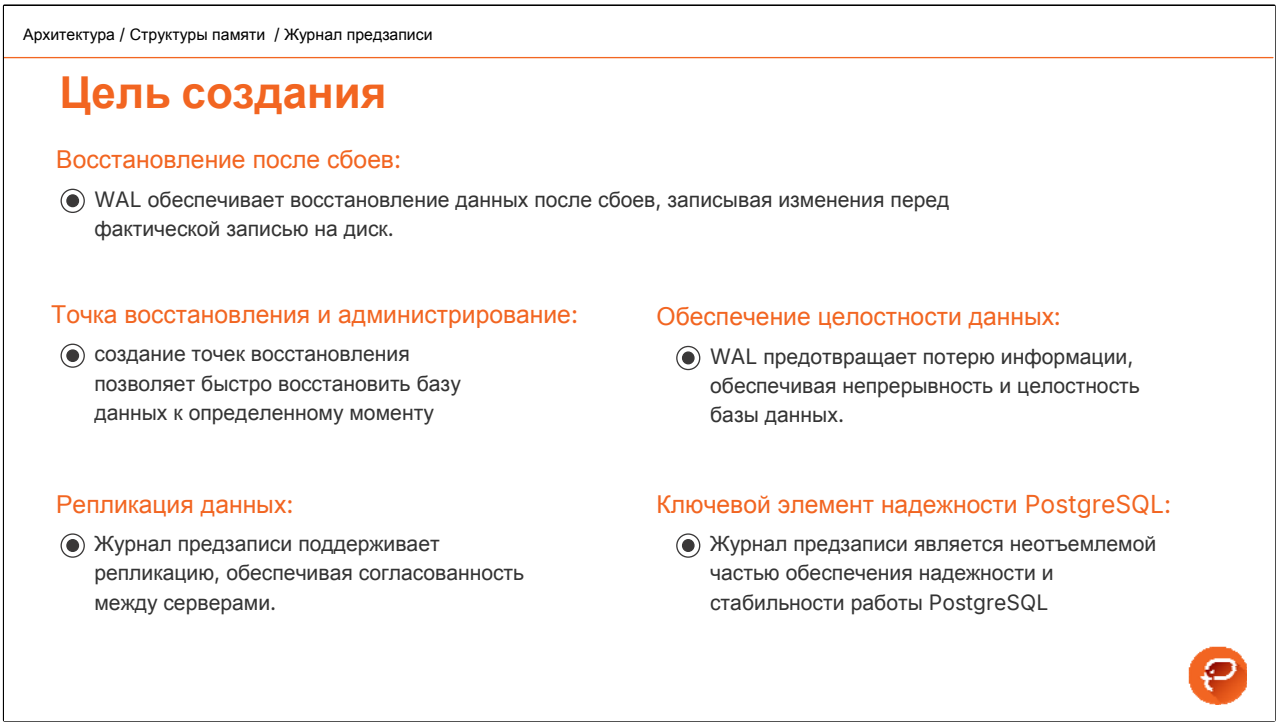
- Улучшение производительности за счет быстрого пробуждения процессов с помощью `ConditionVariableBroadcastFast`(cv). Итерации по одному вызывают множество синхронизированных атомарных записей на спин-блокировке `ConditionVariable`, что существенно влияет на производительность. Но пробуждение процессов пакетами не нарушает корректность работы, поэтому предлагается такой подход.

Патч направлен на улучшение производительности и упрощение процедуры захвата блокировок, снижая уровень конкурентности.

Темы

- Общие сведения
- Структуры памяти
  - Буферный кэш
  - WAL
  - Контрольная точка
  - Демонстрация
  - Практическая работа
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость





Буферный кэш, представляя собой временное хранилище данных в оперативной памяти, эффективно оптимизирует производительность систем управления базами данных (СУБД). Однако, при внесении изменений в данные, буфер становится «грязным», создавая потенциальные рассогласования с данными на диске. Это обстоятельство поднимает важный вопрос о том, как обеспечить целостность данных в случае сбоев.

Именно здесь на сцену выходит журнал предзаписи, или WAL (write-ahead log).

Этот механизм служит для регистрации всех изменений данных, произведенных в системе, до их фактической записи в постоянное хранилище. Такой подход позволяет восстанавливать данные в случае сбоя или аварии, предотвращая потерю важной информации.

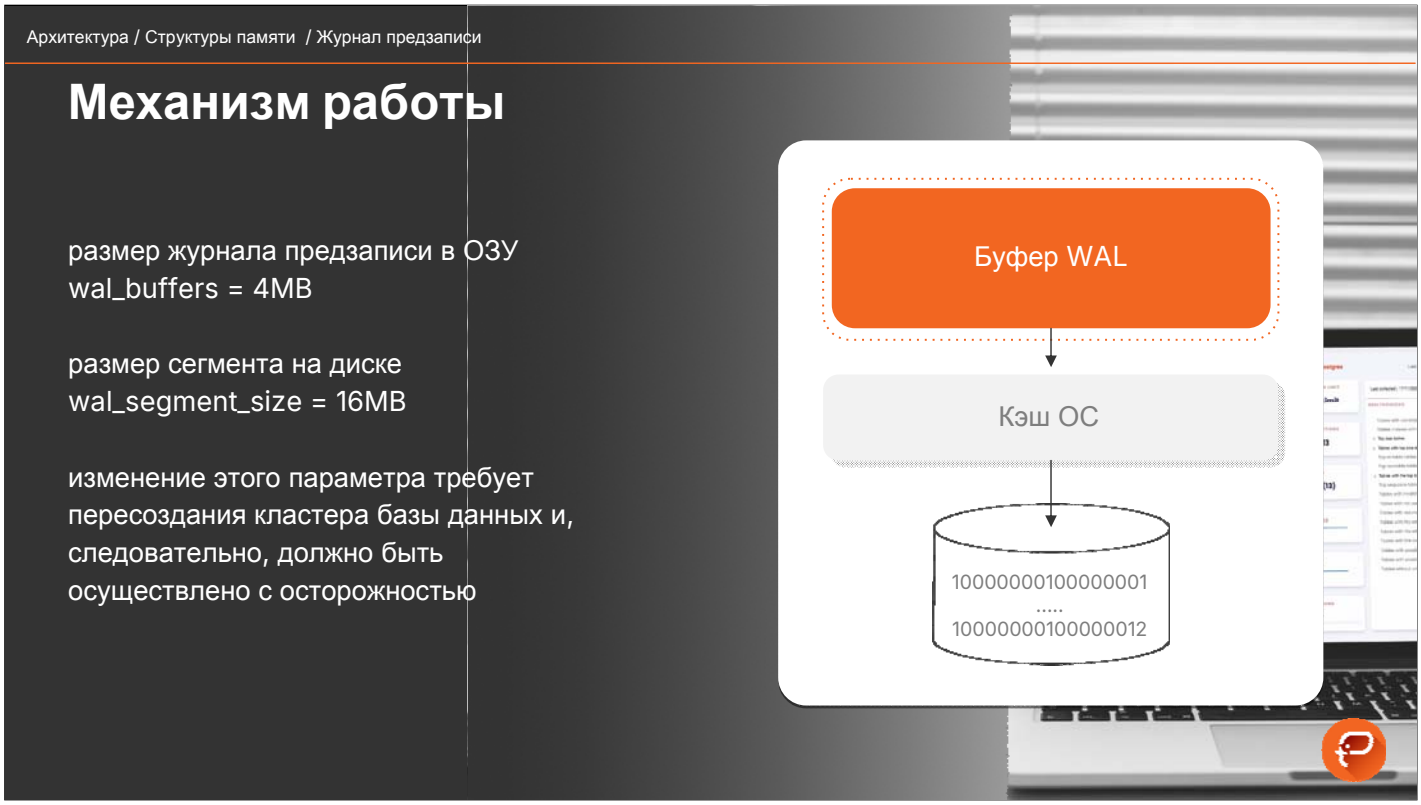
При этом стоит отметить, что буферный кэш, находясь в энергозависимой оперативной памяти, может стать уязвимым при сбоях электропитания. Для снижения риска потери данных в таких ситуациях, журнал предзаписи играет решающую роль, обеспечивая надежный механизм восстановления и поддержания целостности информации в системе управления базами данных.

Журнал предзаписи (WAL) в PostgreSQL служит ключевым механизмом для обеспечения надежности и целостности данных.

Он выполняет несколько важных функций в системе управления базами данных (СУБД) PostgreSQL:

- Восстановление после сбоев: Журнал предзаписи позволяет восстанавливать данные после сбоев или аварий. Все изменения данных сначала записываются в журнал, а затем применяются к постоянному хранилищу. В случае сбоя системы, PostgreSQL может использовать журнал для восстановления данных до момента сбоя, обеспечивая непрерывность и целостность базы данных.
- Точка восстановления: WAL также позволяет создавать точки восстановления (savepoints), что является полезным инструментом для администраторов баз данных. Точки восстановления позволяют быстро восстанавливать базу данных до определенного момента в прошлом, что может быть полезно при внесении критических изменений или обнаружении ошибок.
- Репликация: Журнал предзаписи играет важную роль в механизмах репликации PostgreSQL. Он используется для передачи и воспроизведения изменений данных на удаленных серверах, обеспечивая согласованность данных между мастер-сервером и его репликами.

Таким образом, журнал предзаписи в PostgreSQL обеспечивает не только восстановление данных после сбоев, но и поддерживает функции, такие как точки восстановления и репликация, что делает его важным элементом для обеспечения надежности и управления данными в системе.



Механизм работы журнала предзаписи

●Запись изменений: Когда пользователь или приложение выполняет операции вставки, обновления или удаления данных в PostgreSQL, эти изменения сначала записываются в буфер журнала предзаписи (WAL buffer) в оперативной памяти. Этот буфер — это временное хранилище, где система удерживает изменения перед их окончательной записью на диск. Записи в буфере содержат новые значения данных, старые значения (если применимо), и другие метаданные, такие как идентификаторы транзакций.

●Запись в постоянное хранилище: Периодически или при достижении определенного размера буфера, данные из буфера журнала предзаписи записываются в постоянное хранилище на диске. Этот этап называется «записью в WAL». Важно отметить, что запись происходит синхронно, что гарантирует, что изменения достигнут постоянного хранилища, прежде чем система продолжит выполнение других операций. Это обеспечивает устойчивость и непрерывность данных.

●Кольцевая структура буфера: Буфер журнала предзаписи имеет кольцевую структуру, что означает, что по мере заполнения он начинает перезаписывать старые записи. Такая структура обеспечивает оптимальное использование ограниченного пространства в оперативной памяти. Новые записи заменяют старые в кольцевой очереди, и процесс продолжается, что позволяет системе эффективно управлять объемом записей, сохраняя последние изменения в памяти и постепенно записывая их на диск.

Таким образом, этот связанный процесс обеспечивает надежность и устойчивость данных в PostgreSQL, отслеживая и храня изменения перед окончательной записью на долговременный носитель.

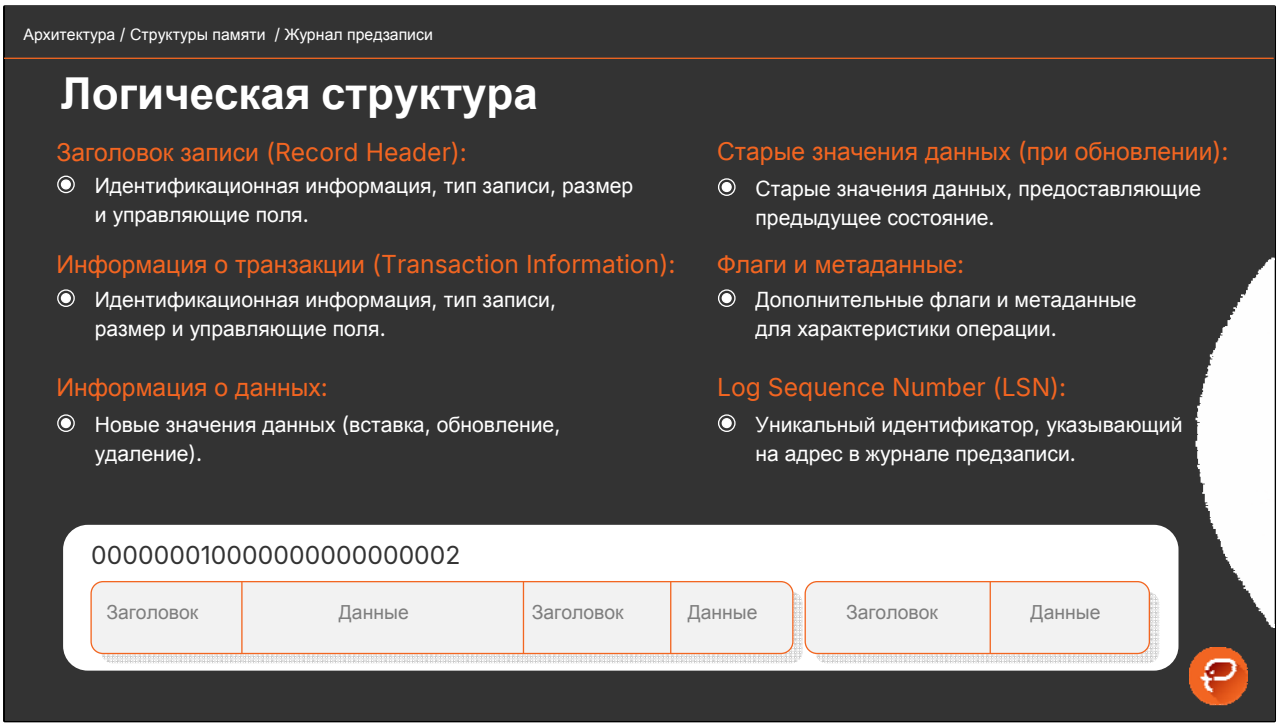
В PostgreSQL размер сегмента журнала предзаписи (WAL segment) на диске обычно составляет 16 мегабайт. Это фиксированный размер сегмента, который является стандартным для большинства установок по умолчанию.

Каждый сегмент представляет собой отдельный файл, в который записываются данные журнала предзаписи. Когда один сегмент заполняется, следующий сегмент начинает использоваться, обеспечивая кольцевую структуру для последовательного сохранения изменений.

Размер сегмента может быть настроен с помощью параметра конфигурации PostgreSQL **`wal_segment_size`**, который определяет размер сегмента в байтах.

Таким образом, если в конфигурации не указано иное, размер сегмента журнала предзаписи по умолчанию составляет 16 мегабайт

https://docs.tantorlabs.ru/tdb/ru/15_4/se/runtime-config-preset.html#GUC-WAL-SEGMENT-SIZE



Логическая структура журнала предзаписи (WAL) в PostgreSQL включает в себя концепцию Log Sequence Number (LSN), которая играет ключевую роль в отслеживании последовательности записей и состояния WAL.

Log Sequence Number (LSN):

Каждая запись в журнале предзаписи содержит уникальный идентификатор, называемый LSN. LSN представляет собой адрес в журнале, который указывает на место, где начинается данная запись.

LSN используется для упорядочивания и контроля согласованности записей. Система отслеживает последний примененный LSN, что позволяет точно восстанавливать данные после сбоев.

Кольцевая структура журнала и механизм учета LSN обеспечивают эффективное использование ресурсов и постоянную доступность для новых записей.

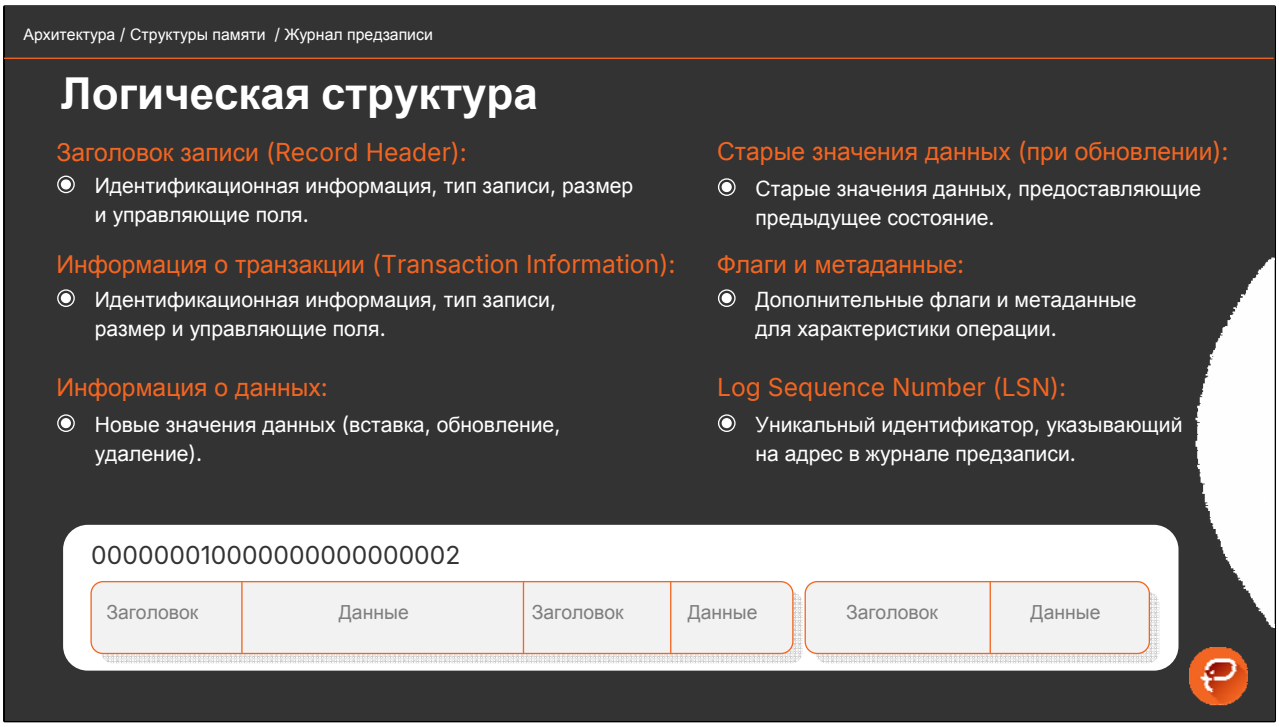
Log Sequence Number (LSN) в PostgreSQL измеряется в байтах. Он представляет собой уникальный адрес в журнале предзаписи (WAL), указывающий на конкретное место в последовательности записей.

LSN включает в себя два компонента:

●Segment Number (номер сегмента): Этот компонент указывает на номер сегмента (файла) в журнале предзаписи. Каждый сегмент обычно имеет размер 16 мегабайт.

●Offset (смещение внутри сегмента): Этот компонент указывает на смещение внутри конкретного сегмента, где начинается данная запись.

LSN позволяет системе PostgreSQL отслеживать порядок записей в WAL и обеспечивает точность и последовательность при восстановлении данных. Когда система применяет изменения из журнала к постоянному хранилищу, она фиксирует последний использованный LSN. В случае сбоя или перезапуска системы, она может использовать этот LSN для определения, с какой точки в журнале предзаписи начать восстановление. Это обеспечивает надежность и целостность данных, а также эффективное управление изменениями в системе.

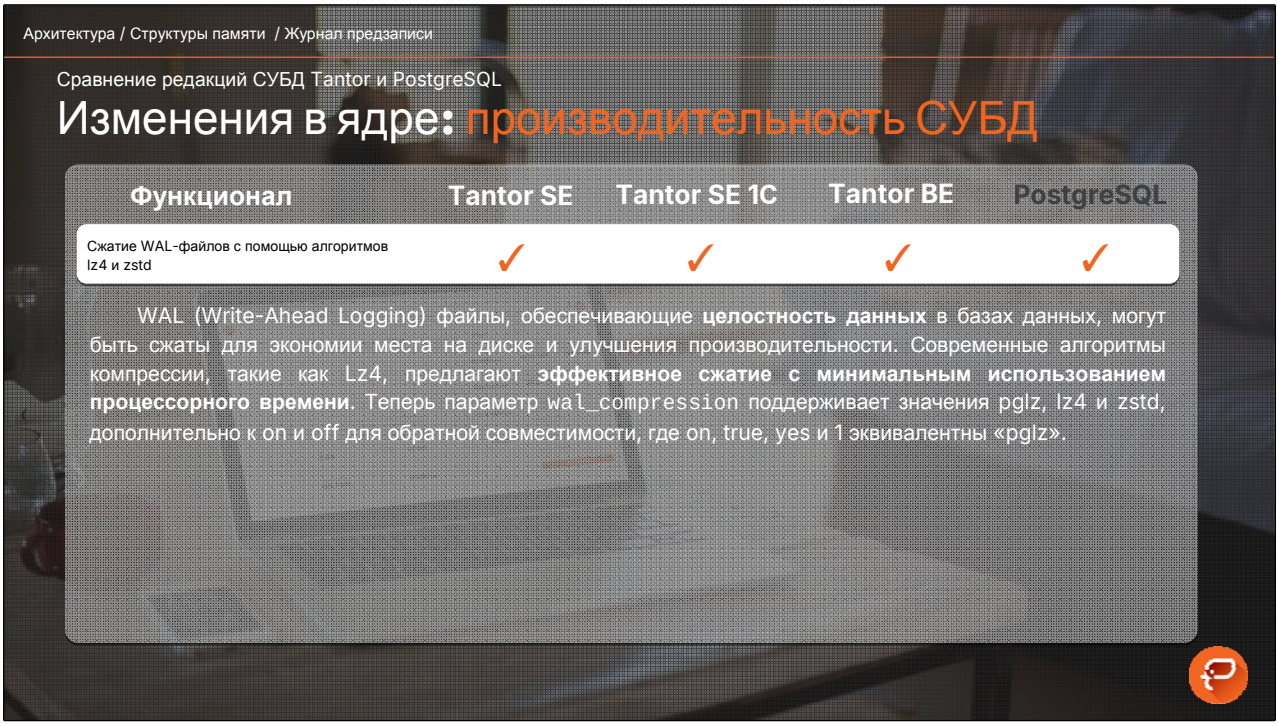


Запись в журнале предзаписи (WAL) в PostgreSQL содержит различные метаданные и информацию о событии или изменении, которое произошло в базе данных. Точное содержание записи зависит от типа операции, которая была зафиксирована.

Элементы, включаемые в запись журнала предзаписи:

- **Заголовок записи (Record Header)**: Идентификационная информация, такая как тип записи, размер записи и другие управляющие поля. Заголовок позволяет системе правильно интерпретировать содержимое записи.
- **Информация о транзакции** (Transaction Information): Идентификатор транзакции (XID) и другие метаданные о транзакции, такие как флаги и состояние. Эта информация позволяет системе отслеживать и контролировать изменения, связанные с определенной транзакцией.
- **Информация о данных**: Новые значения данных, которые были вставлены, обновлены или удалены. Эти данные могут включать новые значения полей, ключи и другую информацию, необходимую для восстановления данных в случае сбоев.
- **Старые значения данных** (при обновлении)
- **Флаги и метаданные:** Дополнительные флаги и метаданные, которые могут содержать информацию о характере операции, дополнительных параметрах или других событиях, связанных с изменением данных.
- **Log Sequence Number (LSN)**: Уникальный идентификатор, который указывает на адрес в журнале предзаписи, где начинается данная запись. LSN играет ключевую роль в упорядочивании и восстановлении записей.

Эти компоненты обеспечивают полноту и точность информации в записи журнала предзаписи, что позволяет PostgreSQL поддерживать надежность и целостность данных, а также обеспечивать восстановление после сбоев.

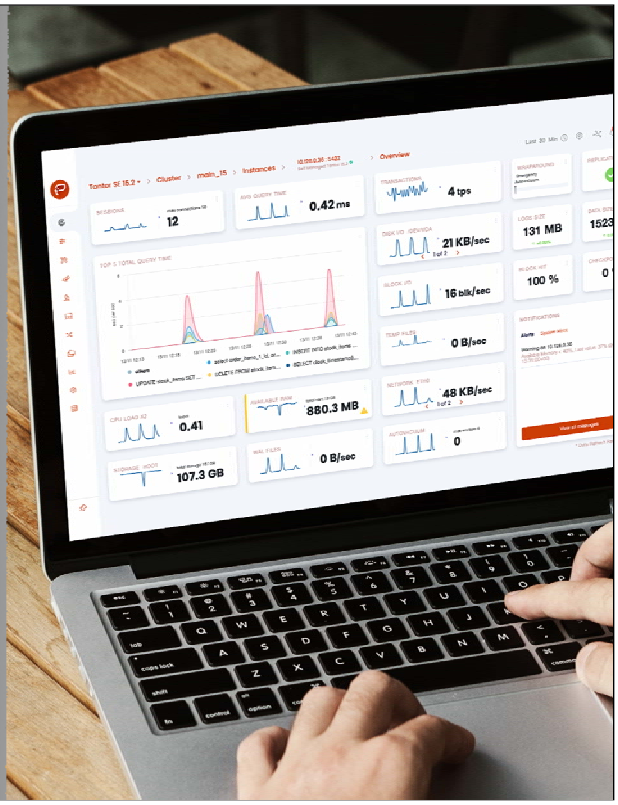


Сжатие WAL-файлов с использованием алгоритмов lz4 и zstd реализовано во всех версиях Tantor как и в классическом PostgreSQL.

WAL (Write-Ahead Logging) файлы используются в системах управления базами данных для обеспечения целостности данных. Сжатие WAL-файлов может помочь уменьшить использование дискового пространства и улучшить производительность. Современные алгоритмы компрессии стали предлагать гораздо лучшее сжатие при меньшем количестве тактов процессора. Lz4 хороший тому пример. Параметр wal_compression теперь может принимать значения pglz, lz4, и zstd, наряду со стандартными on и off, которые оставили в целях обратной совместимости, такие как on, true, yes и 1 эквивалентны «pglz».

Темы

- Общие сведения
- Структуры памяти
  - Буферный кэш
  - WAL
  - Контрольная точка
  - Демонстрация
  - Практическая работа
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость



Принцип работы

Контрольная точка в PostgreSQL — это процесс синхронизации данных между оперативной памятью и диском, фиксирующий момент времени для обеспечения целостности базы данных.

Процесс контрольной точки выполняет следующие шаги:

- Синхронизация буферов
- Фиксация контрольной точки
- Удаление устаревших сегментов журнала предзаписи

Параметры по умолчанию

checkpoint_timeout = 5min

Нахождение измененной информации в буферном кэше, особенно в хорошо настроенном, может занять продолжительное время.

Возникают следующие вопросы: сколько времени потребуется для восстановления из журнала предзаписи в таком случае, и сколько сегментов журнала предзаписи на диске будет необходимо?

Переполнение диска из-за недостатка места также может привести к техническому сбою.

На эти вопросы отвечает фоновый процесс контрольной точки.

Контрольная точка — механизм, который обеспечивает целостность базы данных и минимизирует время восстановления после сбоя. Работа фонового процесса контрольной точки заключается в периодической синхронизации данных между буферами в оперативной памяти и данными на диске.

Процесс контрольной точки выполняет следующие шаги:

- **Синхронизация буферов**: Фоновый процесс периодически смотрит на буферы в оперативной памяти, которые содержат измененные данные (грязные буферы), и записывает эти данные на диск. Это позволяет убедиться, что актуальные данные сохранены на долгосрочном носителе.
- **Фиксация контрольной точки**: После записи измененных данных на диск, процесс контрольной точки фиксирует момент времени, с которого можно начать восстановление базы данных. Полученный адрес начала контрольной точки сохраняется.
- **Удаление устаревших сегментов журнала предзаписи**: Поскольку мы теперь знаем адрес последней контрольной точки, устаревшие сегменты журнала предзаписи, которые предшествуют этой точке, могут быть безопасно удалены, освобождая место на диске.

Таким образом, фоновый процесс контрольной точки не только обеспечивает целостность данных, но и оптимизирует использование дискового пространства. Этот механизм является важной частью обеспечения надежности и эффективности работы PostgreSQL.

Интервал запуска фонового процесса контрольной точки в PostgreSQL устанавливается в конфигурационных файлах базы данных. Параметр, отвечающий за этот интервал, называется checkpoint_timeout. Этот параметр определяет время (в секундах), через которое процесс контрольной точки должен быть запущен повторно.

Например, если в конфигурации установлено значение `checkpoint_timeout = 15min`, то процесс контрольной точки будет запускаться каждые 15 минут.

Значение этого параметра зависит от требований к базе данных, нагрузки на неё и желаемого баланса между производительностью и частотой сохранения данных на диске

Восстановление из WAL

После технического сбоя, восстановление PostgreSQL происходит в несколько этапов:

- определение точки восстановления
- применение журнала предзаписи (WAL)
- синхронизация данных
- запуск фонового процесса контрольной точки

Подводим итоги.

Буферный кэш, как ключевой элемент архитектуры PostgreSQL, играет важную роль в повышении производительности системы. Его задача — временное хранение данных в оперативной памяти, что существенно ускоряет доступ к часто используемым данным.

Важным аспектом обеспечения надежности данных является журнал предзаписи (WAL), который действует как защитный механизм. Он фиксирует изменения данных в оперативной памяти сервера, позволяя восстановить состояние базы данных после возможного сбоя. Таким образом, комбинация буферного кэша и журнала предзаписи обеспечивает не только производительность, но и целостность данных.

Однако, когда происходит технический сбой, важно понимать, как система PostgreSQL осуществляет восстановление.

Процесс включает в себя **определение точки восстановления** — последней контрольной точки, сохраненной в журнале предзаписи. Это становится отправной точкой для восстановления.

Следующий шаг — **применение журнала предзаписи (WAL)**. Записи в журнале используются для восстановления изменений, произошедших после последней контрольной точки. Это позволяет системе вернуть базу данных к консистентному состоянию.

Далее, после восстановления из журнала предзаписи, система **синхронизирует данные** в оперативной памяти с данными на диске. Этот этап важен для обеспечения полной целостности данных и избежания несогласованности между различными хранилищами.

Наконец, для оптимизации производительности и управления дисковым пространством, система **запускает фоновый процесс контрольной точки**. Этот процесс регулирует интервалы сохранения контрольных точек и управляет использованием дискового пространства для эффективного функционирования PostgreSQL.

Эффективное восстановление зависит от правильной конфигурации параметров PostgreSQL, таких как интервалы запуска контрольной точки, а также от наличия достаточного количества журналов предзаписи и правильно настроенных стратегий резервного копирования данных. Все меры направлены на минимизацию потерь данных и обеспечение стабильной работы системы после сбоя.

Демонстрация

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

Структура памяти

Часть 1. Транзакция в psql

Откроем терминал Fly на рабочем столе:

```
astra@tantor:~$ psql
psql (16.1)
```

Введите "help", чтобы получить справку.

```
postgres=#
```

Создадим произвольную таблицу:

```
postgres=# CREATE TABLE a(id integer);
CREATE TABLE
```

Посмотрим что получилось:

```
postgres=# \dt a
        Список отношений
Схема  | Имя |   Тип    | Владелец
--------+-----+---------+----------
public | a   | таблица | postgres
(1 строка)
```

Откроем транзакцию:

```
postgres=# BEGIN;
BEGIN
```

Вставим первую строчку. Обратите внимание, что с помощью табуляции можно дописывать ключевые слова и даже сложные конструкции.

```
postgres=*# INSERT INTO a VALUES (1);
INSERT 0 1
```

Обратите внимание на появление звездочки в строке — это означает что идет транзакция.

Попробуем во втором терминале увидеть первую строчку таблицы.

Откроем второй терминал:

Загрузим psql

```
astra@tantor:~$ psql
psql (16.1)
Введите "help", чтобы получить справку.

postgres=#
```

Обратимся к таблице:

```
postgres=# SELECT * FROM a;
 id
```

```
----
(0 строк)
```

Убедились — пока мы не видим первой строчки. Видны только зафиксированные данные. Грязное чтение не допускается.

—----------

В первом терминале зафиксируем транзакцию.

```
postgres=*# COMMIT;
COMMIT
```

—————————————————

Во втором терминале обратимся к таблице еще раз.

```
postgres=# SELECT * FROM a;
 id
----
  1
(1 строка)
```

Теперь изменения таблицы зафиксированы.

Вывод — видны только те изменения, которые успешно зафиксированы.

Шаг 2. Список фоновых процессов.

Посмотрим где находится директория PGDATA, где находятся файлы кластера БД.

```
postgres=# SHOW data_directory;
            data_directory
---------------------------------------
 /var/lib/postgresql/tantor-se-16/data
(1 строка)
```

Выйдите в первом терминале из psql.

```
postgres=# \q
```

Чтобы посмотреть список процессов воспользуемся утилитой ps

```
astra@tantor:~$ sudo cat /var/lib/postgresql/tantor-se-16/data/postmaster.pid
466
/var/lib/postgresql/tantor-se-16/data
1713847705
5432
/var/run/postgresql
*
  1048641         0
ready
```

Возьмем PID = 466

```
astra@tantor:~$ sudo ps -o command --ppid 466
COMMAND
```

```
postgres: logger
postgres: checkpointer
postgres: background writer
postgres: walwriter
postgres: autovacuum launcher
postgres: logical replication launcher
postgres: walsender replicator ::1(34460) streaming 0/6DA71ED8
postgres: postgres postgres [local] idle
```

Жирным шрифтом показаны системные фоновые процессы, остальные пользовательские.

Список процессов можно увидеть также через представление `pg_stat_activity`.

Сделайте во втором терминале.

```
postgres=# SELECT pid, backend_type, backend_start
FROM pg_stat_activity;
```

```
  pid  |         backend_type          |         backend_start
-------+-------------------------------+-------------------------------
   527 | autovacuum launcher           | 2024-04-23 07:48:25.435889+03
   528 | logical replication launcher  | 2024-04-23 07:48:25.441432+03
   533 | walsender                     | 2024-04-23 07:48:25.472863+03
 25072 | client backend                | 2024-04-23 07:48:51.242631+03
 10977 | client backend                | 2024-04-23 08:06:17.871119+03
   520 | background writer             | 2024-04-23 07:48:25.403365+03
   519 | checkpointer                  | 2024-04-23 07:48:25.402941+03
   526 | walwriter                     | 2024-04-23 07:48:25.425135+03
(8 строк)
```

Шаг 3. Буферный кэш, команда EXPLAIN

Во втором терминале добавим строки в таблицу «a».

```
postgres=#INSERT INTO a SELECT id FROM generate_series(1,10000) AS id;
INSERT 0 10000
```

В первом терминале перезагрузите сервер

```
astra@education:~$ sudo systemctl restart tantor-se-server-16
```

Во втором терминале сделать реконнект:

```
postgres=# \c
```

Вы подключены к базе данных «postgres», как пользователь «postgres».

С помощью команды Explain посмотрите, откуда берется информация

```
postgres=# EXPLAIN (analyze, buffers)
SELECT * FROM a;
                                                 QUERY PLAN
-------------------------------------------------------------------------------
-----------------------
```

```
Seq Scan on a (cost=0.00..145.00 rows=10000 width=4) (actual
time=0.035..1.952 rows=10000 loops=1)
  Buffers: shared read=45
Planning:
  Buffers: shared hit=16 read=6 dirtied=3
Planning Time: 0.428 ms
Execution Time: 2.948 ms
(6 строк)
```

Обратите внимание на строку Buffers. Информация была взята с диска или через кэш операционной системы.

Сделайте эксперимент еще раз.

```
postgres=# EXPLAIN (analyze, buffers)
SELECT * FROM a;
                                                    QUERY PLAN
-------------------------------------------------------------------------------
-----------------------
 Seq Scan on a (cost=0.00..145.00 rows=10000 width=4) (actual
time=0.016..1.383 rows=10000 loops=1)
   Buffers: shared hit=45
 Planning Time: 0.063 ms
 Execution Time: 2.355 ms
(4 строки)
```

Информация изменилась. Теперь информация найдена в буферном кэше

Шаг 4. Журнал предзаписи. Где хранится?

В первом терминале выполните команду

```
astra@education:~$ sudo ls -l /var/lib/postgresql/tantor-se-16/data/pg_wal
итого 360452
-rw------- 1 postgres postgres 16777216 Apr 23 08:10 00000001000000000000006D
-rw-r----- 1 postgres postgres 16777216 Apr  3 14:28 00000001000000000000006E
-rw-r----- 1 postgres postgres 16777216 Apr  3 14:28 00000001000000000000006F
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 000000010000000000000070
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 000000010000000000000071
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 000000010000000000000072
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 000000010000000000000073
-rw------- 1 postgres postgres 16777216 Apr  3 14:42 000000010000000000000074
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 000000010000000000000075
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 000000010000000000000076
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 000000010000000000000077
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 000000010000000000000078
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 000000010000000000000079
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 00000001000000000000007A
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 00000001000000000000007B
-rw------- 1 postgres postgres 16777216 Apr  3 14:42 00000001000000000000007C
-rw------- 1 postgres postgres 16777216 Apr  3 14:42 00000001000000000000007D
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 00000001000000000000007E
```

```
-rw------- 1 postgres postgres 16777216 Apr  3 14:41 000000010000000000000007F
-rw------- 1 postgres postgres 16777216 Apr  3 14:42 00000001000000000000080
-rw-r----- 1 postgres postgres 16777216 Apr  3 14:42 00000001000000000000081
-rw------- 1 postgres postgres 16777216 Apr  3 14:42 00000001000000000000082
drwx------ 2 postgres postgres     4096 Apr  2 12:09 archive_status
```

Файлы журнала предзаписи находятся в директории `pg_wal`. Сегментами по 16 мегабайт.

Шаг 5. Контрольная точка.

Контрольная точка выполняется периодически, посмотрим во втором терминале какой интервал установлен.

```
postgres=# SHOW checkpoint_timeout;
checkpoint_timeout
--------------------
5min
(1 строка)
```

Контрольную точку можно запустить вручную.

```
postgres=# CHECKPOINT;
CHECKPOINT
```

В первом терминале посмотрим на файлы журнала предзаписи. Ненужные файлы удалены.

```
astra@education:~$ sudo ls -l /var/lib/postgresql/tantor-se-16/data/pg_wal
итого 802820
-rw------- 1 postgres postgres 16777216 Feb 14 07:40 0000000100000001000000A4
-rw------- 1 postgres postgres 16777216 Feb 14 07:40 0000000100000001000000A5
-rw------- 1 postgres postgres 16777216 Feb 14 07:40 0000000100000001000000A6
-rw------- 1 postgres postgres 16777216 Feb 14 07:40 0000000100000001000000A7
-rw------- 1 postgres postgres 16777216 Feb 14 07:40 0000000100000001000000A8
```

Шаг 6. Восстановление после сбоя.

Добавим во втором терминале новые строчки

```
postgres=# INSERT INTO a SELECT id FROM generate_series(1,10000) AS id;
INSERT 0 10000
```

Остановите кластер БД в режиме системного сбоя. Для начала определим PID процесса postmaster.

```
astra@education:~$ sudo cat /var/lib/postgresql/tantor-se-16/data/postmaster.pid
12563
/var/lib/postgresql/tantor-se-16/data
1713849023
5432
/var/run/postgresql
*
  1048641        24
```

```
ready
astra@education:~$ sudo kill -9 12563
```

Запустим экземпляр сервера

```
astra@education:~$ sudo systemctl start tantor-se-server-16
```

Немного тормозит. Идет восстановление.

Во втором окне посмотрим, сохранились ли вставленные строки

```
postgres=# \c
```

Вы подключены к базе данных «postgres» как пользователь «postgres».

```
postgres=# SELECT count(*) FROM a;
count
-------
 20000
(1 строка)
```

Очистим объекты во втором терминале.

```
postgres=# DROP TABLE a;
DROP TABLE

postgres=# \dt
```

Отношения не найдены.

Практическая работа

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

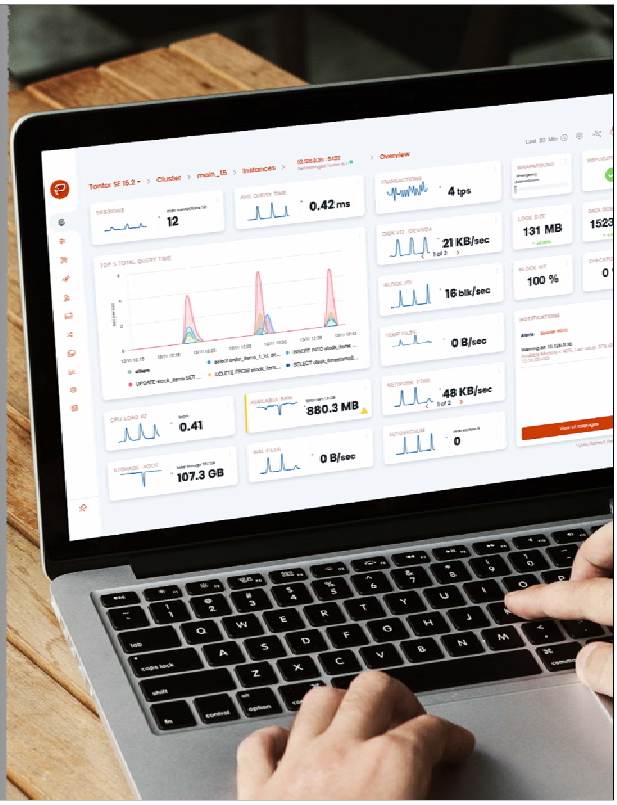
Дополнительное задание:

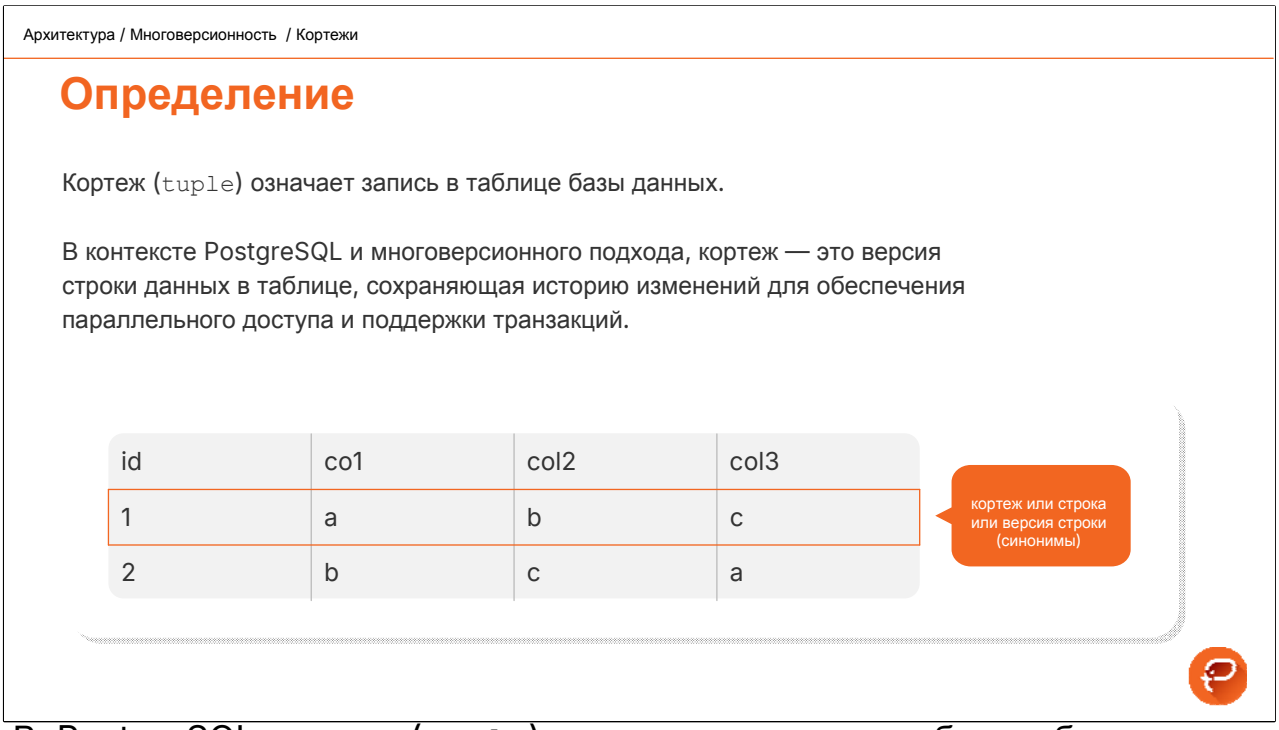
Сделать задание 1-6 с помощью pgAdmin

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Кортежи
Механизм
Снимок данных
Уровни изоляции
Фиксация и откат транзакции
Блокировки объектов
Блокировки строк
Демонстрация
Практическая работа





В PostgreSQL кортеж (`tuple`) означает запись в таблице базы данных. Каждая строка в таблице представляет собой кортеж, который содержит данные, соответствующие столбцам этой таблицы. Каждое поле кортежа соответствует определенному столбцу, а сам кортеж представляет собой упорядоченный набор значений.

В контексте баз данных кортеж и строка часто используются как синонимы. Таким образом, при обсуждении PostgreSQL кортеж можно рассматривать как строку данных в таблице.

В контексте многоверсионного подхода в базах данных, такого как подход **MVCC (Multi-Version Concurrency Control)**, кортеж может также относиться к версии строки данных в таблице, которая сохраняет историческую информацию о изменениях. Этот подход позволяет одновременно поддерживать несколько версий данных, что полезно для обеспечения консистентности данных в среде с параллельным доступом и транзакциями.

В PostgreSQL, например, MVCC используется для управления параллельным доступом к данным. Каждая транзакция видит свою собственную «версию» данных, что предотвращает конфликты и блокировки. Каждый кортеж в таблице имеет свою собственную историю изменений, и транзакции могут работать с разными версиями данных одновременно.

MVCC позволяет избежать проблем блокировок и повышает параллелизм в работе с данными. В таком случае, когда упоминается кортеж, это может относиться к конкретной версии строки данных, а не только к строке данных, как это понимается в контексте традиционного подхода к базам данных.

Заголовок версии строки

Версия строки данных содержит заголовок, который обычно включает в себя информацию о версии, видимости и флагах состояния строки.

xmin	xmax	Флаги состояния	Указатель (ctid)	Данные

В PostgreSQL версия строки данных содержит заголовок, который обычно включает в себя информацию о версии, видимости и флагах состояния строки. Этот заголовок представляет собой метаданные, необходимые для управления многоверсионным подходом.

Вот основные элементы, которые могут находиться в заголовке версии строки:

- **xmin (Transaction ID)**: поле отвечает за идентификацию транзакции, которая создала версию строки. В контексте MVCC, это является начальной транзакцией (ID транзакции, которая вставила или последний раз обновила строку).
- **xmax (Transaction ID)**: поле указывает на транзакцию, которая удалила строку или внесла последние изменения. Если строка не удалена или не обновлена, значение `xmax` будет равно `0`.
- **Флаги состояния (State Flags)**: флаги указывают на состояние строки в конкретной версии. Например, флаг «удалена» (DELETE_IN_PROGRESS) может указывать на то, что данная версия строки является результатом операции удаления. Другие флаги могут указывать на обновление, вставку и так далее.
- **Pointer (Указатель)**: Указатель указывает на фактические данные строки. Это может быть адрес в памяти или указатель на конкретный блок данных в таблице.

Элементы вместе обеспечивают систему управления версиями в PostgreSQL возможностью поддерживать одновременное выполнение транзакций без блокировок, что является ключевым преимуществом многоверсионного подхода (MVCC). В результате при параллельном доступе к данным каждая транзакция видит свою собственную версию данных, и система может управлять конфликтами и поддерживать целостность данных.

Transaction ID (XID) в PostgreSQL — уникальный идентификатор транзакции в системе управления базой данных. Каждая транзакция в PostgreSQL получает свой собственный уникальный номер XID, который идентифицирует ее в контексте базы данных.

Основное предназначение XID в базе данных — отслеживание и управление изменениями данных в транзакциях. Когда транзакция вносит изменения в данные (например, вставляет, обновляет или удаляет строки), эти изменения ассоциируются с уникальным номером транзакции, то есть XID.

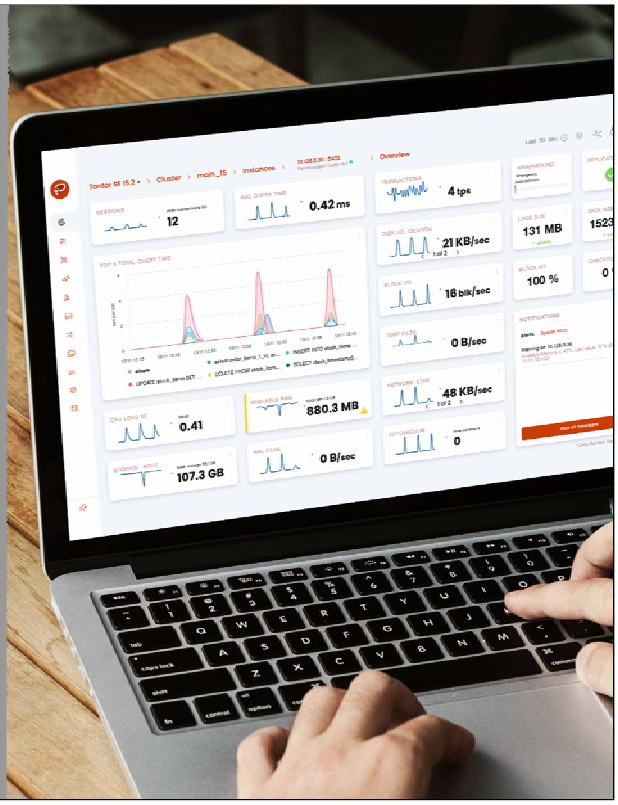
Важно отметить, что в PostgreSQL XID является 32-битным числом, и после достижения максимального значения происходит «обновление» (wraparound). Это может привести к ситуации, называемой «переполнением XID», которая требует выполнения операции очистки (`VACUUM`) для корректного функционирования базы данных.

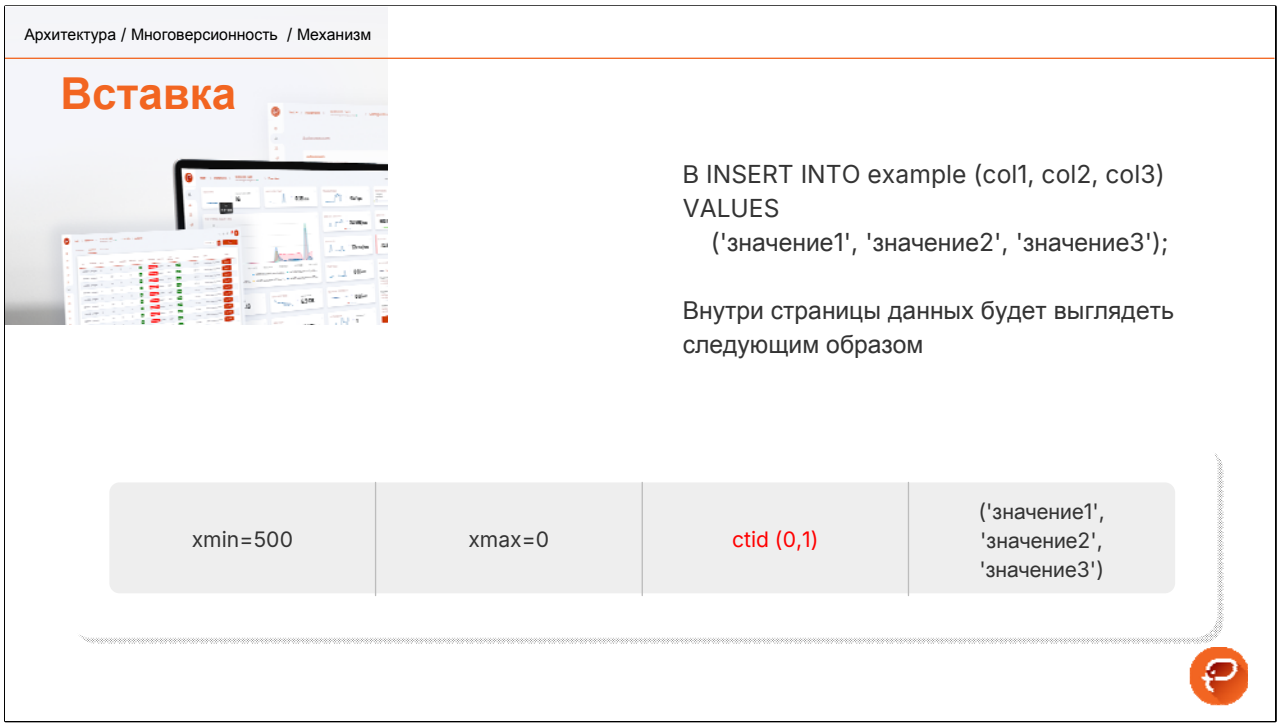
В СУБД Tantor (кроме BE) счетчик транзакций является 64-битным, что исключает переполнение. Подробнее об этом поговорим дальше по курсу.

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

- Кортежи
- Механизм
- Снимок данных
- Уровни изоляции
- Фиксация и откат транзакции
- Блокировки объектов
- Блокировки строк
- Демонстрация
- Практическая работа





Чтобы вставить данные в таблицу PostgreSQL, вам потребуется использовать оператор INSERT INTO. Давайте предположим, у вас есть таблица с именем «example_table» с колонками «column1», «column2» и «column3».

Вы можете вставить данные следующим образом:

```
INSERT INTO example (col1, col2, col3) VALUES
    ('значение1', 'значение2', 'значение3');
```

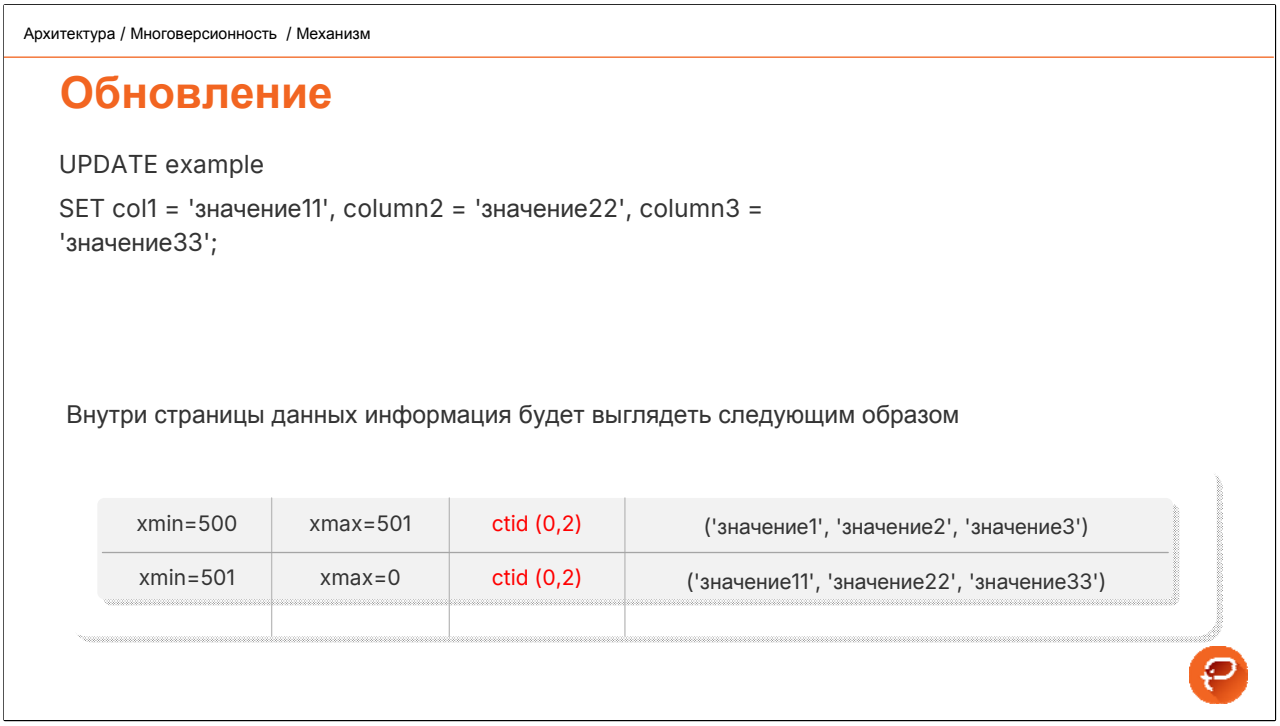
Предположим что текущий номер транзакции будет 500, тогда в заголовке версии строки будет xmin=500, xmax=0 а указатель будет показывать на ту же саму строку. Пока никаких версий нет.

`ctid` в PostgreSQL представляет собой системный столбец, который указывает на физическое местоположение строки в блоке таблицы.

Он состоит из двух чисел: (block_number, index_in_block), где `block_number` — номер блока, а `index_in_block` — относительный индекс строки в этом блоке.

Если `ctid` имеет значение (0,1), это означает, что соответствующая строка находится в блоке с номером 0, а её относительный индекс в этом блоке равен 1.

Важно отметить, что ctid может изменяться в результате операций обновления или удаления, так как PostgreSQL может создавать новые версии строк для управления многоверсионностью данных.



Для обновления строки в PostgreSQL используется оператор `UPDATE`. Пример обновления строки в таблице:

```
UPDATE example
SET col1 = 'значение11', column2 = 'значение22', column3 = 'значение33';
```

Итак при вставке получается что будет существовать одновременно две версии строки. Обе будут существовать до того момента, пока не будет уничтожено в результате регламентных работ. Старая строка будет называться исторической, или bloat.

В контексте баз данных, термин «bloat» (раздутие) обычно означает избыточное использование дискового пространства или другие виды избыточности, что может происходить в результате долгосрочного использования базы данных. Раздувание может привести к неэффективности использования ресурсов, таких как дисковое пространство и память, а также снизить производительность системы.

В PostgreSQL «bloat» может проявляться в различных аспектах:

Пространственное раздутие (Space bloat): Когда строки удаляются или обновляются, но не освобождается физическое дисковое пространство, происходит пространственное раздутие. Это может произойти, например, из-за того, что PostgreSQL использует механизм многоверсионности (MVCC).

Индексное раздутие (Index bloat): Если индексы не оптимизированы или не регулярно перестраиваются, они также могут страдать от раздувания. Это может привести к увеличению размера индекса и ухудшению производительности запросов.

Статистическое раздутие (Statistical bloat): Раздутие может произойти, если статистическая информация о таблицах и индексах устаревает или не точна, что может повлиять на эффективность выполнения запросов оптимизатором.

Удаление

DELETE FROM your_table
WHERE col3='значение33';

Внутри страницы данных будет выглядеть следующим образом

(0,1)	xmin=500	xmax=501	ctid (0,2)	('значение1', 'значение2', 'значение3')
(0,2)	xmin=501	xmax=502	ctid (0,2)	('значение11', 'значение22', 'значение33')

Чтобы удалить строку из таблицы в PostgreSQL, используйте оператор DELETE.

Пример:

```
DELETE FROM your_table
WHERE col3='значение33';
```

Для борьбы с «bloat» в PostgreSQL могут использоваться различные методы, такие как регулярная очистка (VACUUM), анализ и оптимизация индексов, а также обновление статистики. Однако важно проводить эти операции с осторожностью и, при необходимости, подходить к оптимизации базы данных с учетом конкретных требований и характеристик системы.

Эти и другие моменты будут постепенно будут раскрыты в курсе.

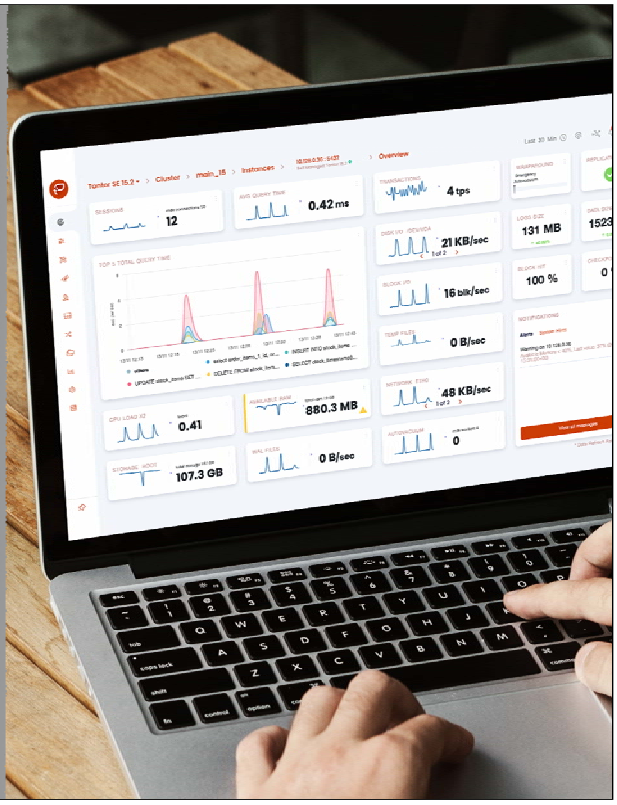
Но остается вопрос, а какую конкретно версию строки видит транзакция? Чтобы разобраться в вопросе, давайте поймем что такое снимок данных.

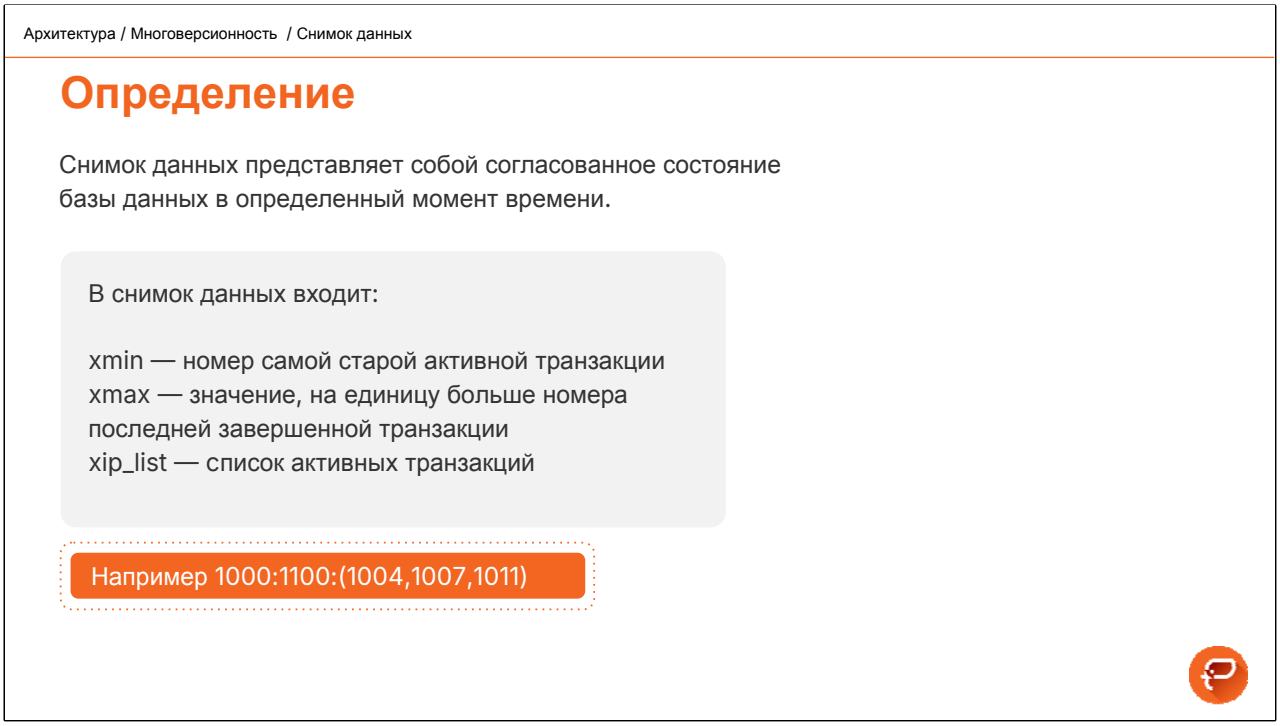
Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Многоверсионность:

- Кортежи
- Механизм
- Снимок данных
- Уровни изоляции
- Фиксация и откат транзакции
- Блокировки объектов
- Блокировки строк
- Демонстрация
- Практическая работа





Физически в страницах данных базы данных может существовать несколько версий одной и той же строки, и это объясняется механизмом управления версиями данных в PostgreSQL. Каждая транзакция, несмотря на наличие нескольких версий, видит лишь одну из них в соответствии с принципами изоляции транзакций.

Изменения данных, произведенные в транзакции, не сразу отражаются на самой строке в страницах данных. Вместо этого создаются новые версии строк, содержащие изменения. У каждой транзакции есть снимок данных, соответствующий ее моменту времени. Этот снимок, в свою очередь, формирует консистентное представление данных на определенный момент.

Таким образом, снимок данных в PostgreSQL является виртуальным состоянием базы данных, созданным по мере необходимости для каждой транзакции. Он обеспечивает изоляцию транзакций, предоставляя им уникальное видение данных, даже если физически в базе могут существовать несколько версий одной строки.

Снимок данных обеспечивает изоляцию транзакций, предоставляя им уникальное видение данных на определенный момент времени. Он является важным элементом для обеспечения согласованности и восстановления данных, особенно при параллельном выполнении множества транзакций.

Таким образом, снимок данных формируется на основе нескольких значений, зафиксированных в момент его создания:

- **Нижняя граница снимка,** обозначаемая как xmin, представляет собой номер самой старой активной транзакции. Все транзакции с меньшими номерами уже завершены (фиксированы), 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,** обозначаемая как xmax, представляет собой 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**, обозначаемый как xip_list (список транзакций в процессе выполнения), включает номера всех активных транзакций, за исключением виртуальных, которые не оказывают влияния на видимость данных.

Пример

Снимок данных 1 502:505:(502,504)

Снимок данных 2 502:514:(502,504,511)

Снимок данных 3 502:534:(502,504,511,521)

(0,1)	xmin=500	xmax=510	ctid (0,2)	версия 1
(0,2)	xmin=510	xmax=520	ctid (0,2)	версия 2

Давайте определим какую версию строки видит оператор в снимках данных:

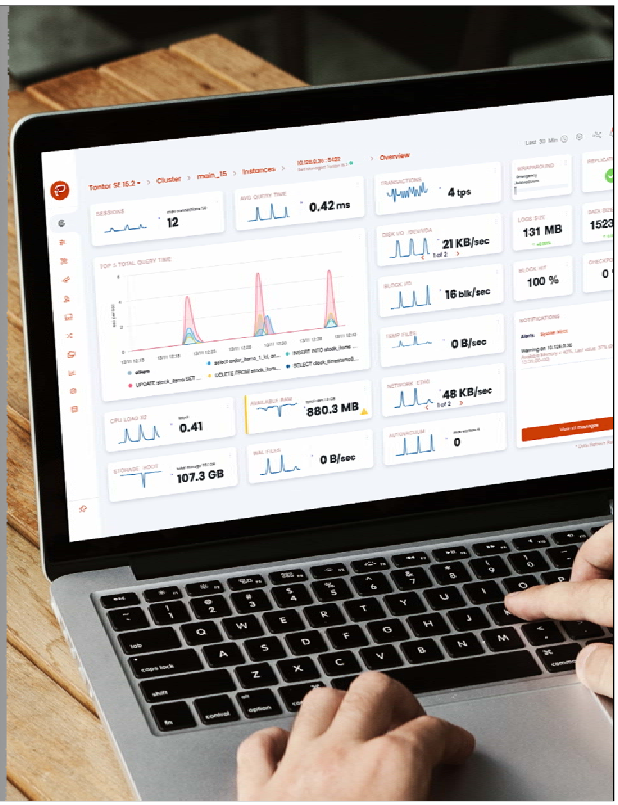
- Так как xmax снимка равен 505 который больше чем xmin версии строки и меньше xmax первой версии — значит видит первую версию.
- Снимок данных xmax снимка равен 514 который больше чем xmin версии строки и меньше xmax второй версии — значит видит вторую версию.
- Третий снимок данных не видит никакую версию потому что xmax снимка равен 534,а это уже больше xmax второй версии, а это значит что она уже удалена.

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Многоверсионность:

- Кортежи
- Механизм
- Снимок данных
- Уровни изоляции
- Фиксация и откат транзакции
- Блокировки объектов
- Блокировки строк
- Демонстрация
- Практическая работа



Архитектура / Многоверсионность / Уровни изоляции

Определение

Стандарт SQL определяет 4 уровня изоляции:

READ UNCOMMITTED
- Чтение неподтвержденных данных

READ COMMITTED
- Чтение подтвержденных данных

REPEATABLE READ
- Повторяемое чтение

SERIALIZABLE
- Сериализуемое выполнение

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

В стандарте SQL определены четыре уровня изоляции:

- `READ UNCOMMITTED` (Чтение неподтвержденных данных): Это самый низкий уровень изоляции. Он позволяет транзакциям видеть изменения, внесенные другими транзакциями, даже если эти изменения еще не были подтверждены (зафиксированы). В PostgreSQL не поддерживается.
- `READ COMMITTED` (Чтение подтвержденных данных): Этот уровень гарантирует, что транзакции будут видеть только те изменения, которые были подтверждены (зафиксированы). Это предотвращает чтение неподтвержденных данных, но не предотвращает фантомное чтение.
- `REPEATABLE READ` (Повторяемое чтение): Транзакции на этом уровне гарантируют, что все строки, считанные в рамках транзакции, будут оставаться неизменными до завершения транзакции. Это предотвращает как чтение неподтвержденных данных, так и фантомное чтение.
- `SERIALIZABLE` (Сериализуемое выполнение): Этот уровень предоставляет самую высокую степень изоляции. Он гарантирует, что выполнение транзакций будет таким, как если бы они выполнялись последовательно, без взаимодействия между ними. Это предотвращает чтение неподтвержденных данных, фантомное чтение и другие аномалии уровней ниже.

Важно отметить, что более высокие уровни изоляции могут повлиять на производительность из-за дополнительных блокировок и ограничений. При выборе уровня изоляции необходимо учитывать баланс между надежностью и производительностью в конкретном контексте приложения.

Read Commited

PostgreSQL не поддерживает уровень READ UNCOMMITTED (Чтение неподтвержденных данных) он совпадает с уровнем READ COMMITTED (Чтение подтвержденных данных).

Транзакции видят только те изменения, которые были зафиксированы (подтверждены).
Избегает чтения «грязных» данных, но может допускать фантомное чтение, когда другая транзакция добавляет новые строки в результаты запроса.

Возможны аномалии:

- Фантомные чтения (Phantom Reads)
- Неповторяющееся чтение (Non-Repeatable Reads)

Снимок данных создается в начале выполнения любого оператора

PostgreSQL не поддерживает уровень `READ UNCOMMITTED` (Чтение неподтвержденных данных) он совпадает с уровнем `READ COMMITTED` (Чтение подтвержденных данных). Можно указать только ради совместимости со стандартом.

В уровне изоляции «`READ COMMITTED`» в PostgreSQL, фантомные чтения представляют собой ситуации, когда транзакция видит изменения в наборе данных, которые были выполнены другой транзакцией после начала текущей транзакции. Это происходит из-за того, что «`READ COMMITTED`» позволяет видеть только подтвержденные (зафиксированные) изменения.

Вот пример сценария фантомного чтения на уровне «`READ COMMITTED`»:

- Транзакция A начинает чтение некоторого набора данных.
- Транзакция B вставляет новую строку или изменяет существующую строку в том же наборе данных и фиксирует изменения.
- Транзакция A продолжает чтение, и видит изменения, внесенные транзакцией B после начала своего чтения.

Таким образом, фантомные строки могут появиться в результате выполнения запроса в середине транзакции из-за изменений, внесенных другими транзакциями. Хотя уровень «`READ COMMITTED`» предотвращает чтение «грязных» данных, он не исключает фантомные чтения.

Обратите внимание, что уровень «`SERIALIZABLE`» обеспечивает более строгую изоляцию, предотвращая и фантомные чтения, и другие аномалии, за счет полного последовательного выполнения транзакций.

На этом уровне, каждый оператор работает со своим снимком данных, поэтому на уровне изоляции «`READ COMMITTED`» в PostgreSQL могут возникнуть следующие аномалии:

- Фантомные чтения (Phantom Reads): Это ситуация, когда транзакция видит новые строки, добавленные другой транзакцией, после начала своего выполнения.

Например, транзакция A начинает чтение данных, транзакция B добавляет новую строку, и транзакция A видит эту новую строку при последующем чтении.

- Неповторяющееся чтение (Non-Repeatable Reads): Это происходит, когда транзакция видит измененные значения в одной и той же строке при повторном чтении в рамках одной транзакции.

Например, транзакция A читает значение из строки, транзакция B изменяет это значение, и транзакция A, читая ту же строку снова, видит измененное значение.

и др.

Repeatable Read

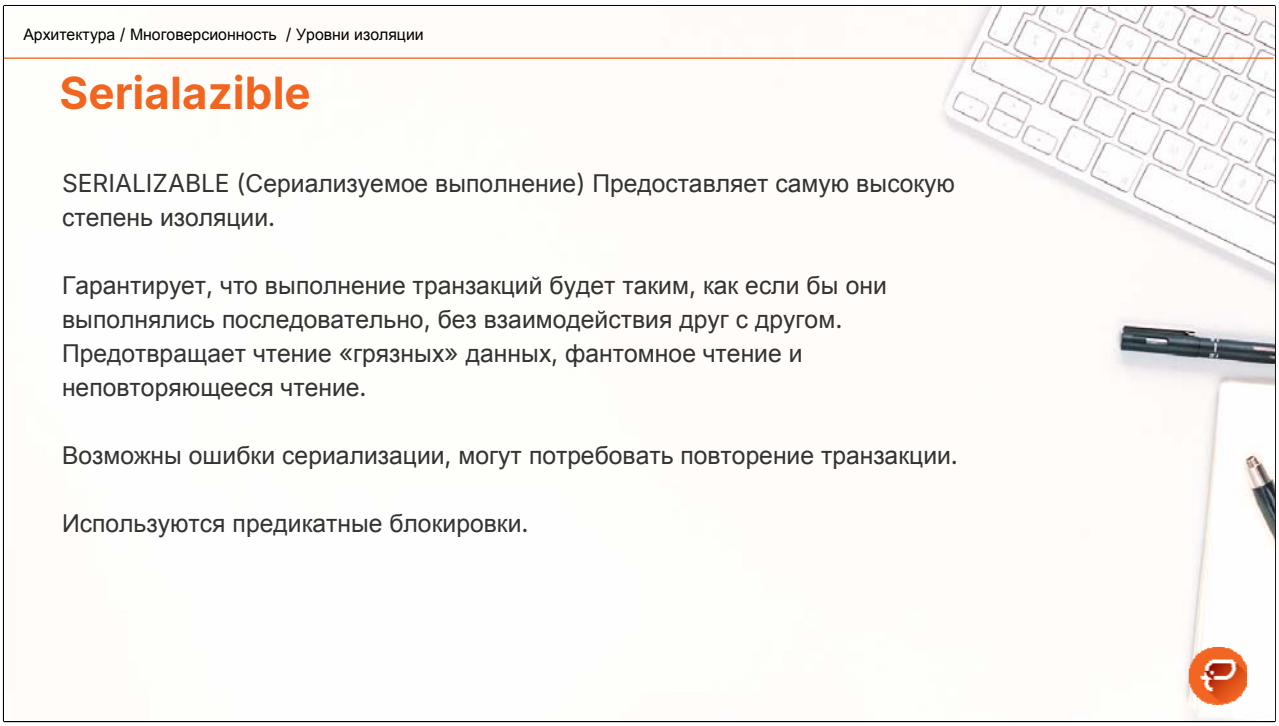
REPEATABLE READ (Повторяемое чтение):

Гарантирует, что все строки, считанные в рамках транзакции, останутся неизменными до завершения транзакции.
Предотвращает чтение «грязных» данных и фантомное чтение, но может допускать неповторяющееся чтение (non-repeatable read).

Возможны аномалии сериализации.

В контексте многоверсионного подхода - снимок данных делается при выполнении первого оператора транзакции и все остальные операторы пользуются одним снимком.

`REPEATABLE READ` (Повторяемое чтение):
Гарантирует, что все строки, считанные в рамках транзакции, останутся неизменными до завершения транзакции.
Предотвращает чтение «грязных» данных и фантомное чтение, но может допускать неповторяющееся чтение (non-repeatable read) и возможны аномалии сериализации
На уровне изоляции `REPEATABLE READ` в PostgreSQL фантомные чтения (Phantom Reads) и неповторяющиеся чтения (Non-Repeatable Reads) действительно предотвращаются. Транзакции на уровне REPEATABLE READ видят стабильный снимок данных на момент их начала, и изменения, произведенные другими транзакциями после начала текущей транзакции, не должны быть видны.
В контексте PostgreSQL это значит что снимок данных делается при выполнении первого оператора транзакции и все остальные операторы пользуются одним снимком



`SERIALIZABLE` (Сериализуемое выполнение) предоставляет самую высокую степень изоляции.

Гарантирует, что выполнение транзакций будет таким, как если бы они выполнялись последовательно, без взаимодействия друг с другом.

Предотвращает чтение «грязных» данных, фантомное чтение и неповторяющееся чтение.

Возможны ошибки сериализации, могут потребовать повторение транзакции.

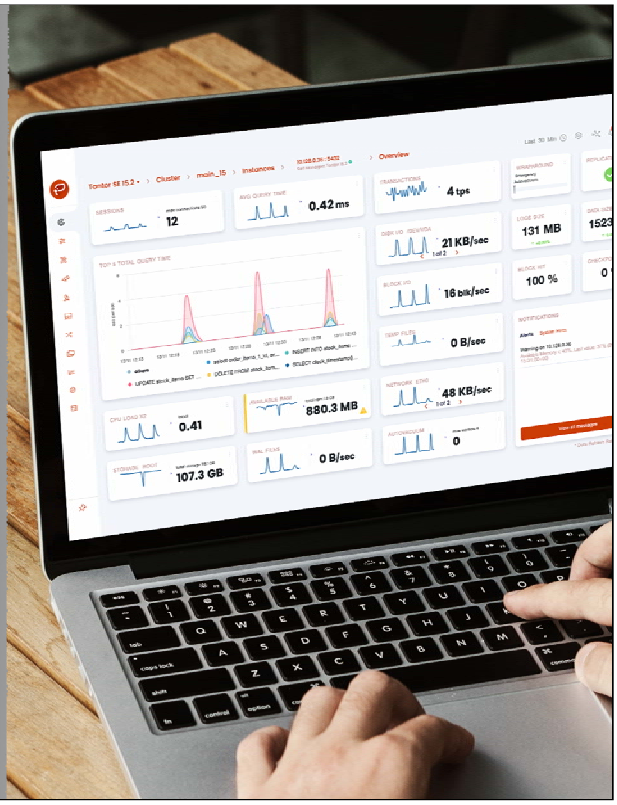
Уровень изоляции `SERIALIZABLE` в PostgreSQL обеспечивает самый высокий уровень изоляции между транзакциями, гарантируя, что выполнение транзакций будет таким, как если бы они выполнялись последовательно, без взаимодействия друг с другом. Одним из механизмов, используемых для достижения этой изоляции, являются предикатные блокировки (predicate locks).

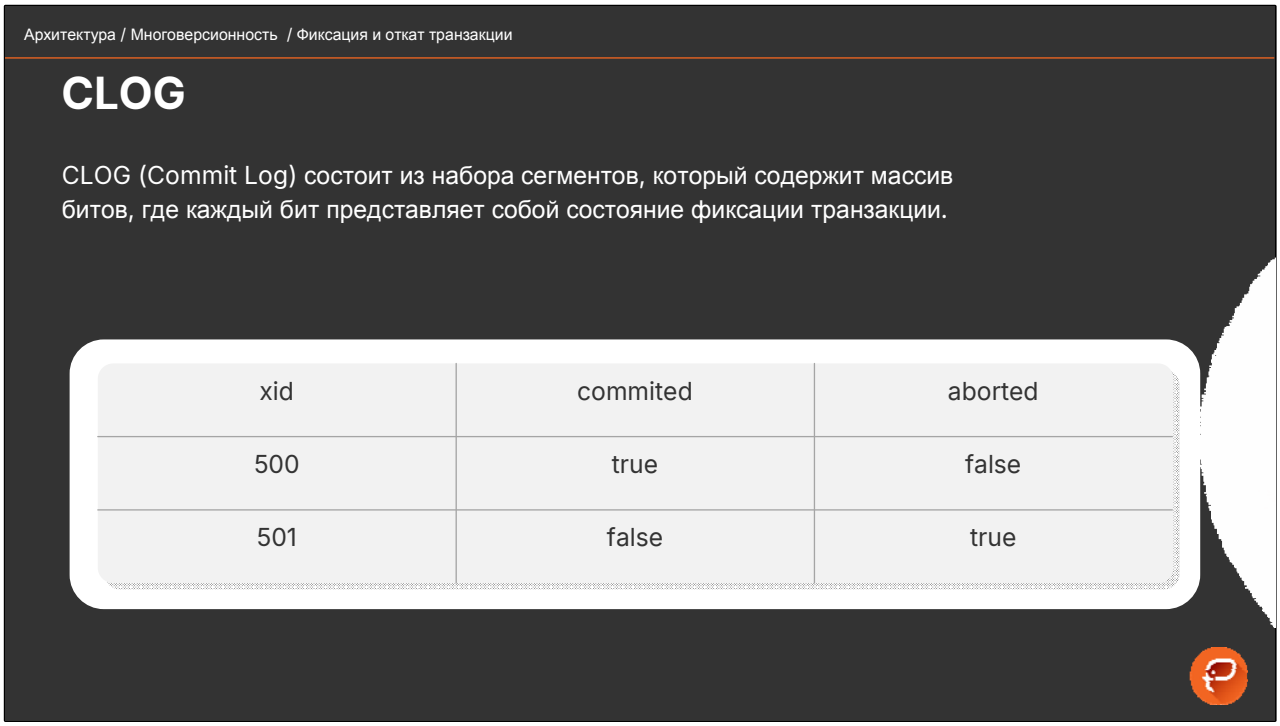
Предикатные блокировки в PostgreSQL позволяют предотвращать конфликты между транзакциями, связанными с условиями поиска (предикатами). То есть, если транзакция устанавливает блокировку на некоторый набор данных, соответствующий определенному условию (предикату), то другие транзакции, пытающиеся изменить данные, соответствующие этому предикату, будут заблокированы до завершения первой транзакции.

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Многоверсионность:
- Кортежи
- Механизм
- Снимок данных
- Уровни изоляции
- Фиксация и откат транзакции
- Блокировки объектов
- Блокировки строк
- Демонстрация
- Практическая работа





При фиксации или отмене транзакции, серверный процесс помимо формирования журнальной записи, устанавливает один из двух битов (как признак фиксации или отката) в разделяемой структуре памяти, называемой Commit Log.

`CLOG` (Commit Log) — это буфер, который используется для отслеживания информации о фиксации (commit) транзакций. Этот механизм необходим для поддержки многоверсионности (MVCC — Multi-Version Concurrency Control) в PostgreSQL.

`MVCC` — это техника, которая позволяет одновременно существовать нескольким версиям данных в базе данных. Когда происходит транзакция, она видит состояние данных на момент начала транзакции, и ей не мешают изменения, внесенные другими транзакциями после её начала.

`CLOG` сохраняет информацию о том, была ли транзакция зафиксирована (commit) или отменена (abort).

Это важно для того, чтобы система могла определить, какие версии данных видимы для каждой транзакции в системе. Когда транзакция фиксируется, её идентификатор сохраняется в CLOG, и эта информация используется для определения, какие данные она может видеть в будущем.

Кроме `CLOG`, в PostgreSQL также используются другие буферы для поддержки MVCC, такие как Transaction ID (XID) и MultiXact.

Эти механизмы совместно обеспечивают надежную и эффективную поддержку работы с транзакциями и версионностью данных.

`CLOG` (Commit Log) в PostgreSQL состоит из набора сегментов, которые являются файлами фиксированного размера. Каждый сегмент CLOG содержит массив битов, где каждый бит представляет собой состояние фиксации транзакции. Таким образом, каждый бит в `CLOG` соответствует отдельной транзакции.

Основной целью `CLOG` является отслеживание фиксации (commit) или отмены (abort) транзакции. Когда транзакция завершается, в соответствующий бит CLOG записывается информация о том, была ли транзакция успешно завершена (commit) или отменена (abort).

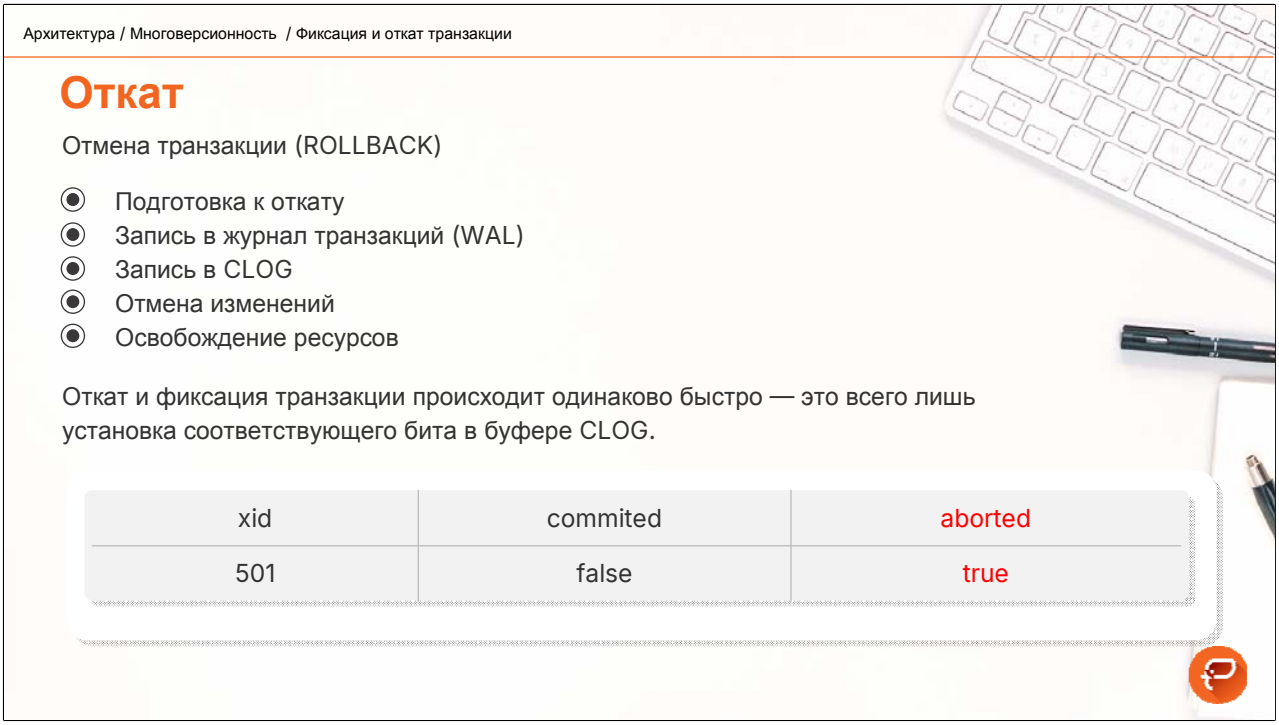
Использование битов вместо целых записей для каждой транзакции обеспечивает эффективное использование ресурсов и позволяет сэкономить место в CLOG. Каждый бит представляет собой минимальную единицу информации о фиксации транзакции, и весь CLOG может быть эффективно управляем в виде битовых массивов.

В PostgreSQL фиксация транзакции (commit) осуществляется следующим образом:

- **Подготовка к фиксации**: Когда клиентское приложение выполняет операцию COMMIT для завершения транзакции, PostgreSQL начинает процесс фиксации.
- **Запись в журнал транзакций** (WAL): Первым шагом является запись информации о фиксации транзакции в журнал транзакций (WAL). Это делается для обеспечения долговременной устойчивости данных и восстановления базы данных после сбоев. Журнал транзакций фиксирует факт завершения транзакции, включая идентификатор транзакции (Transaction ID) и другие необходимые метаданные.
- **Запись в CLOG**: После записи в WAL происходит фиксация в CLOG. В CLOG для каждой транзакции устанавливается бит, указывающий на успешную фиксацию (commit). Это позволяет системе определить, какие транзакции были успешно завершены и какие данные они могут видеть другие транзакции в системе.
- **Освобождение ресурсов**: После успешной записи в CLOG система освобождает ресурсы, занимаемые транзакцией, и завершает выполнение.

В случае отмены транзакции (ROLLBACK), вместо фиксации в CLOG записывается информация об отмене транзакции. Это также фиксируется в WAL, чтобы обеспечить целостность данных.

Этот процесс обеспечивает согласованность данных и поддерживает многоверсионность в PostgreSQL.



Когда транзакция отменяется (ROLLBACK) в PostgreSQL, соответствующая информация записывается в CLOG (Commit Log).

Изучим как происходит откат транзакции в CLOG.

- **Подготовка к откату**: Когда клиентское приложение выполняет операцию ROLLBACK для отмены транзакции, PostgreSQL начинает процесс отката.
- **Запись в журнал транзакций** (WAL): Первым шагом является запись информации об отмене транзакции в журнал транзакций (WAL). Это обеспечивает долговременную устойчивость данных и восстановление базы данных после сбоев.
- **Запись в CLOG**: После записи в WAL происходит откат в CLOG. Для каждой транзакции устанавливается бит, указывающий на отмену (abort). Это позволяет системе определить, что транзакция была отменена, и другие транзакции не должны учитывать её изменения.
- **Отмена изменений**: Система отменяет все изменения, внесенные транзакцией, чтобы вернуть данные к состоянию, которое было до начала транзакции.
- **Освобождение ресурсов**: После успешной записи в CLOG и отмены изменений система освобождает ресурсы, занимаемые транзакцией, и завершает её выполнение.

В результате выполнения этих шагов транзакция полностью отменяется, и данные возвращаются к состоянию, которое было до её начала.

Этот процесс обеспечивает согласованность данных в базе данных PostgreSQL при отмене транзакции.

Откат и фиксация транзакции происходит одинаково быстро - это всего лишь установка соответствующего бита в буфере CLOG.

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Многоверсионность:

- Кортежи
- Механизм
- Снимок данных
- Уровни изоляции
- Фиксация и откат транзакции
- Блокировки объектов
- Блокировки строк
- Демонстрация
- Практическая работа

Очередь

На уровне объектов используется честная очередь блокировок.

Система автоматически управляет конфликтами блокировок, ставя запросы в очередь и предоставляя блокировки в порядке очереди.

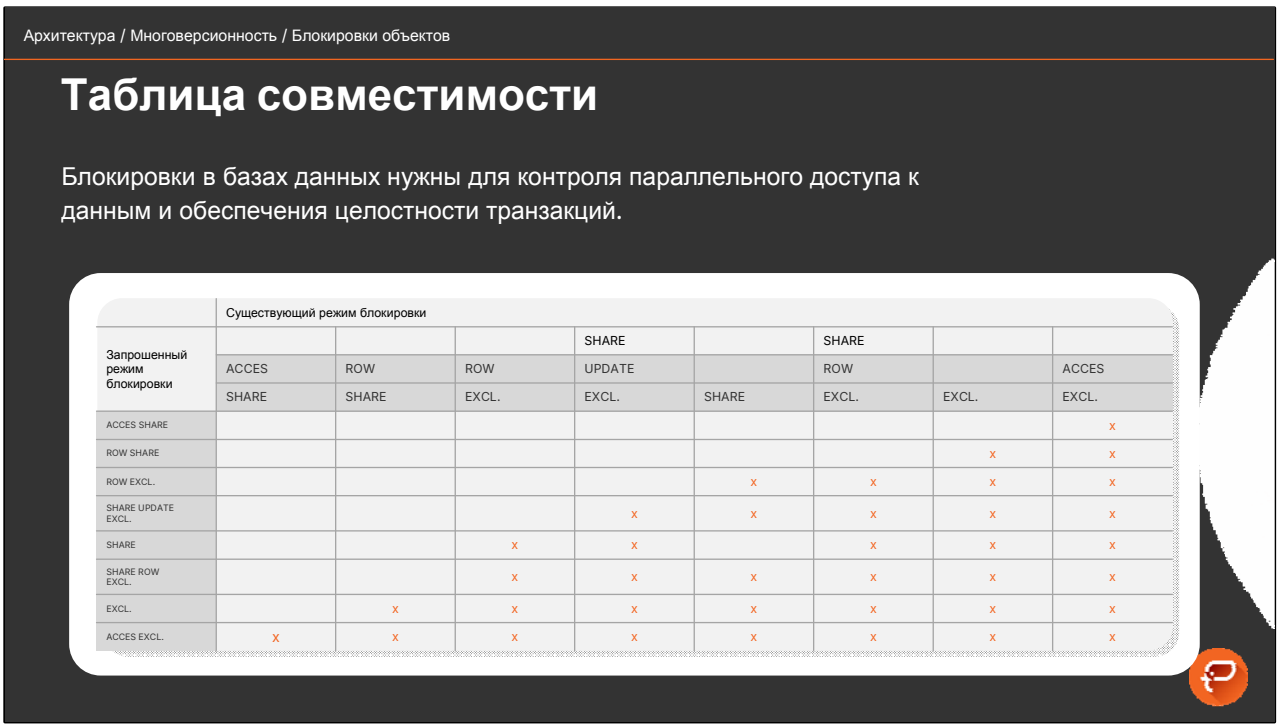
Механизм гарантирует согласованность данных, предотвращает конфликты и обеспечивает безопасность транзакций в среде с параллельным доступом к данным.

В контексте PostgreSQL «честная очередь блокировок» на уровне объектов означает, что когда несколько транзакций пытаются получить блокировку на один и тот же объект данных, СУБД PostgreSQL управляет этими блокировками в справедливой очереди.

Это обеспечивает порядок выполнения блокировок и предотвращает неравноправное обслуживание запросов.

Система автоматически управляет конфликтами блокировок, ставя запросы в очередь и предоставляя блокировки в порядке очереди.

Механизм гарантирует согласованность данных, предотвращает конфликты и обеспечивает безопасность транзакций в среде с параллельным доступом к данным.



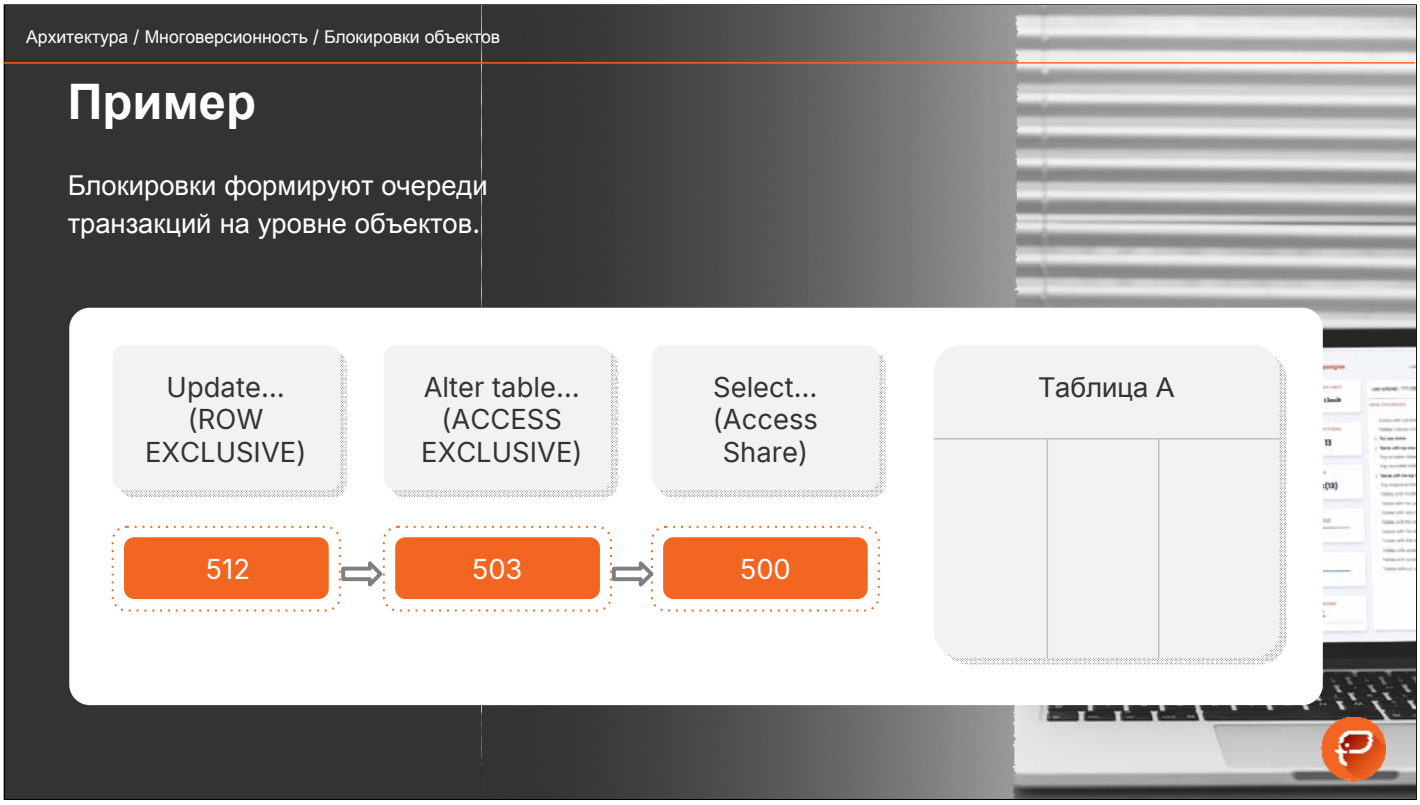
Блокировки в базах данных нужны для контроля параллельного доступа к данным и обеспечения целостности транзакций. Они предотвращают конфликты и несогласованное изменение данных, обеспечивая, что операции чтения и записи выполняются безопасно и атомарно.

СУБД Тантор предоставляет различные режимы блокировки для контроля параллельного доступа к данным в таблицах. Эти режимы могут использоваться для блокировки, контролируемой приложением, в ситуациях, когда `MVCC` не дает желаемого поведения. Кроме того, большинство команд Tantor SE автоматически получают блокировки соответствующих режимов, чтобы гарантировать, что ссылочные таблицы не будут удалены или изменены несовместимым образом во время выполнения команды.

Например, `TRUNCATE` не может быть безопасно выполнена одновременно с другими операциями на той же таблице, поэтому она получает блокировку ACCESS EXCLUSIVE на таблицу для обеспечения этого.

13.3.1. Таблица-уровень блокировок

https://docs.tantorlabs.ru/tdb/ru/15_4/se/explicit-locking.html



К примеру существует таблица А.

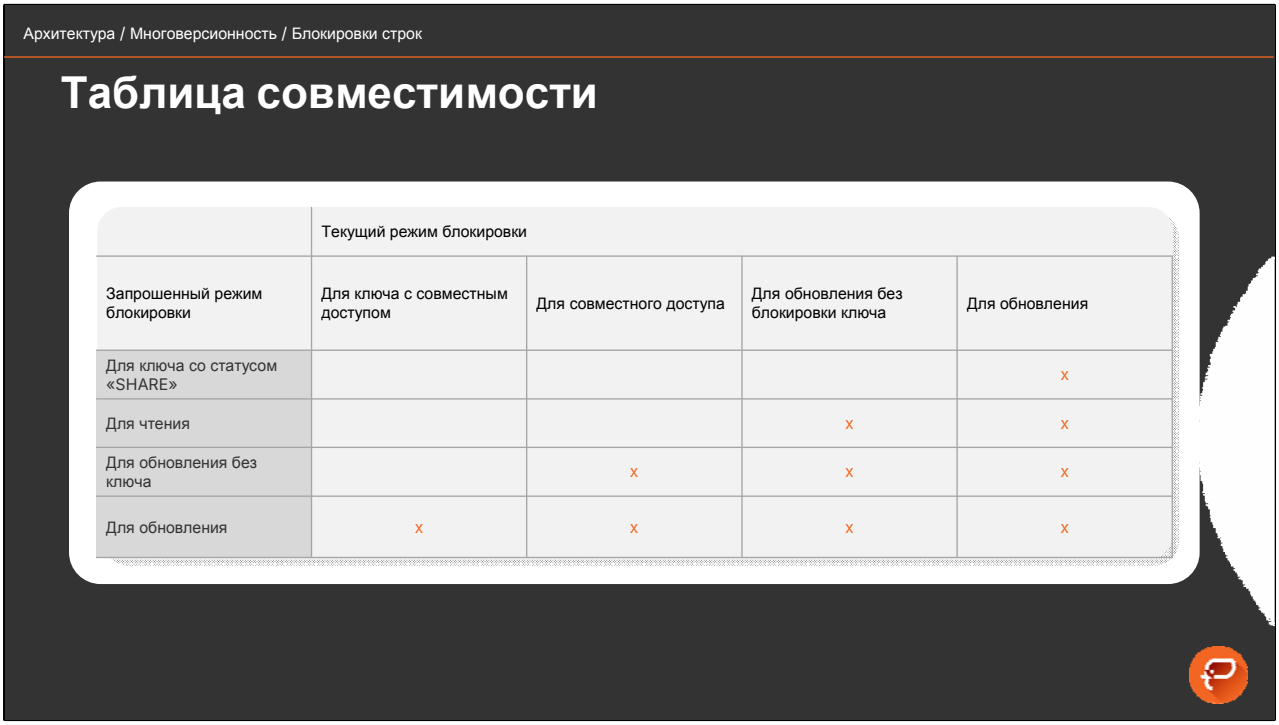
К ней происходит обращение в транзакции 500 с целью выборки данных `SELECT`. Накладывается блокировка Access Share.

Параллельно через некоторое время приходит с транзакции 503 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.

Транзакции будут ждать пока не отработает предыдущая, для произведения своих действий.

Уровень блокировк может быть совместим. К примеру если с обновлением строк придет обновление строк той же таблицы но других, эти транзакции могут делать свою работу параллельно.



В дополнение к блокировкам на уровне таблицы, существуют блокировки на уровне строк, которые автоматически используются Tantor SE в следующих контекстах.

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.

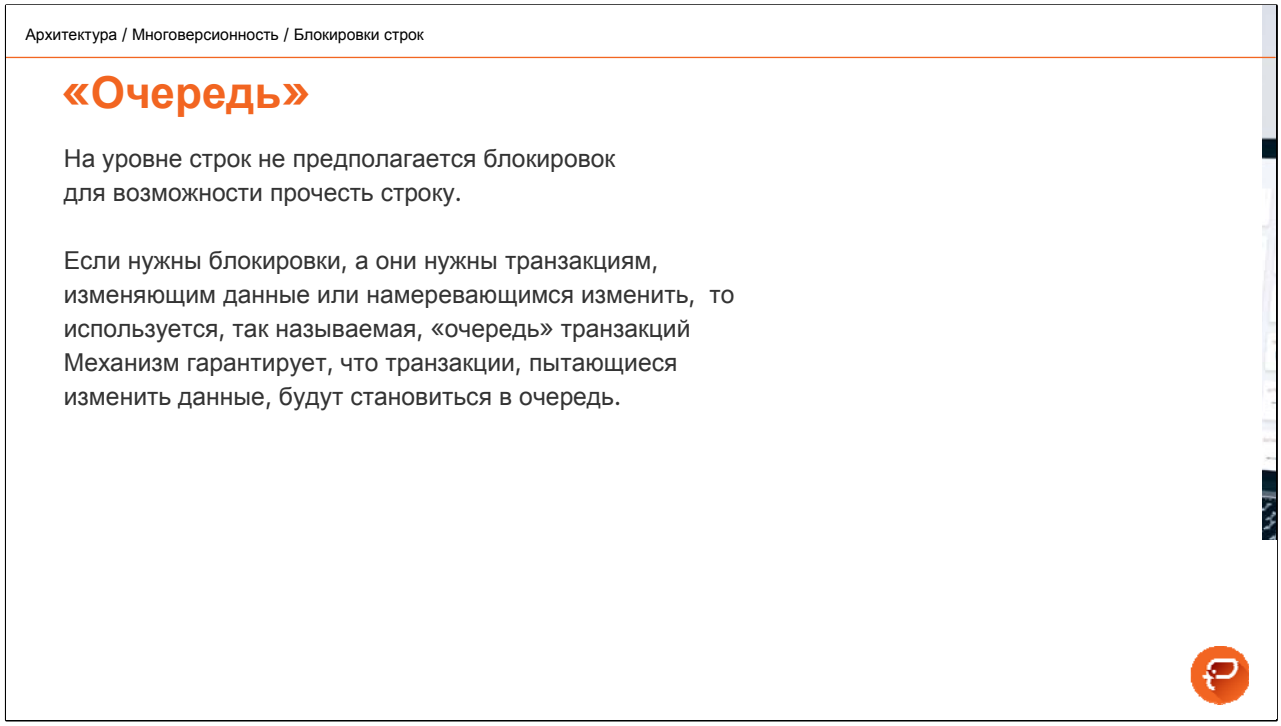
`FOR UPDATE`: Запрашивает блокировку строк для операций обновления, предотвращая их изменение или блокировку другими транзакциями до завершения текущей транзакции. Используется при выполнении операций `UPDATE`, `DELETE`, `SELECT FOR UPDATE` и подобных.

`FOR NO KEY UPDATE`: Аналогичен `FOR UPDATE`, но блокировка слабее, не влияя на команды `SELECT FOR KEY SHARE`.

`FOR SHARE`: Запрашивает общую блокировку строк для чтения, предотвращая изменение или блокировку другими транзакциями для операций `UPDATE`, `DELETE`, `SELECT FOR UPDATE` и подобных.

`FOR KEY SHARE`: Аналогичен `FOR SHARE`, но блокирует `SELECT FOR UPDATE` и не влияет на `SELECT FOR NO KEY UPDATE`.

Tantor SE не сохраняет информацию о измененных строках в памяти, и нет ограничений на количество заблокированных строк одновременно. Блокировка строки может вызвать запись на диск, например, SELECT FOR UPDATE изменяет строки для их пометки как заблокированные, вызывая запись на диск. 13.3.2. Блокировки на уровне строк https://docs.tantorlabs.ru/tdb/ru/15_4/se/explicit-locking.html#ROW-LOCK-COMPATIBILITY



Команды, которые только читают данные не устанавливают блокировки на уровне строк.

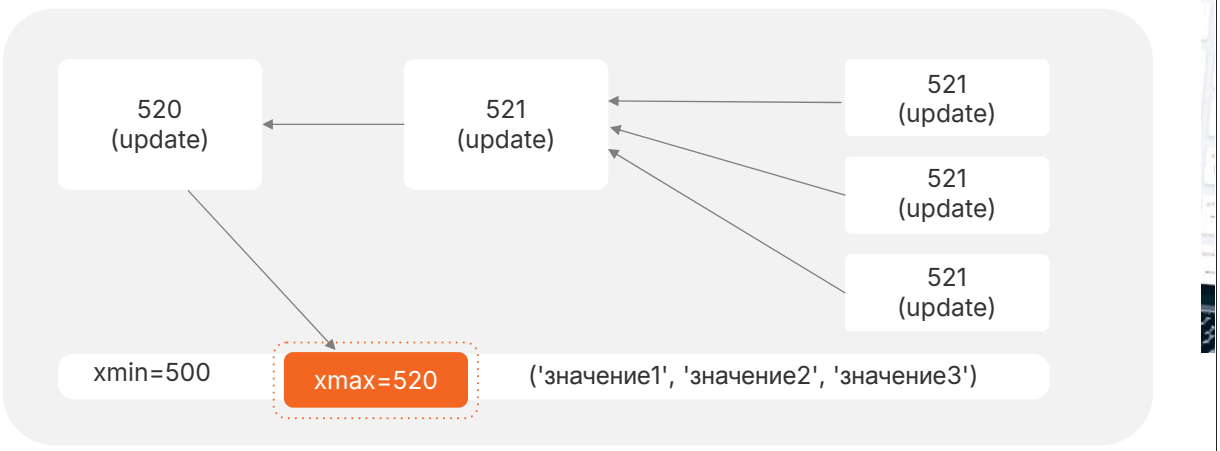
Транзакции, изменяющие данные, устанавливают блокировки на уровне строк.

Поскольку строки остаются заблокированными до конца транзакции или до отката (полностью или до точки сохранения), то транзакция, которая хочет получить блокировку какой-либо строки встаёт в очередь не «за строкой», а за транзакцией, которая эту строку заблокировала.

Поэтому можно сказать, что существует неупорядоченная «очередь» транзакций. Поскольку на уровне строк существуют разделяемые режимы блокирования, для обслуживания таких транзакций существует механизм «мультитранзакций» — набор транзакций, которые могут работать совместно.

Пример

«Очередь» транзакция при блокировке строки



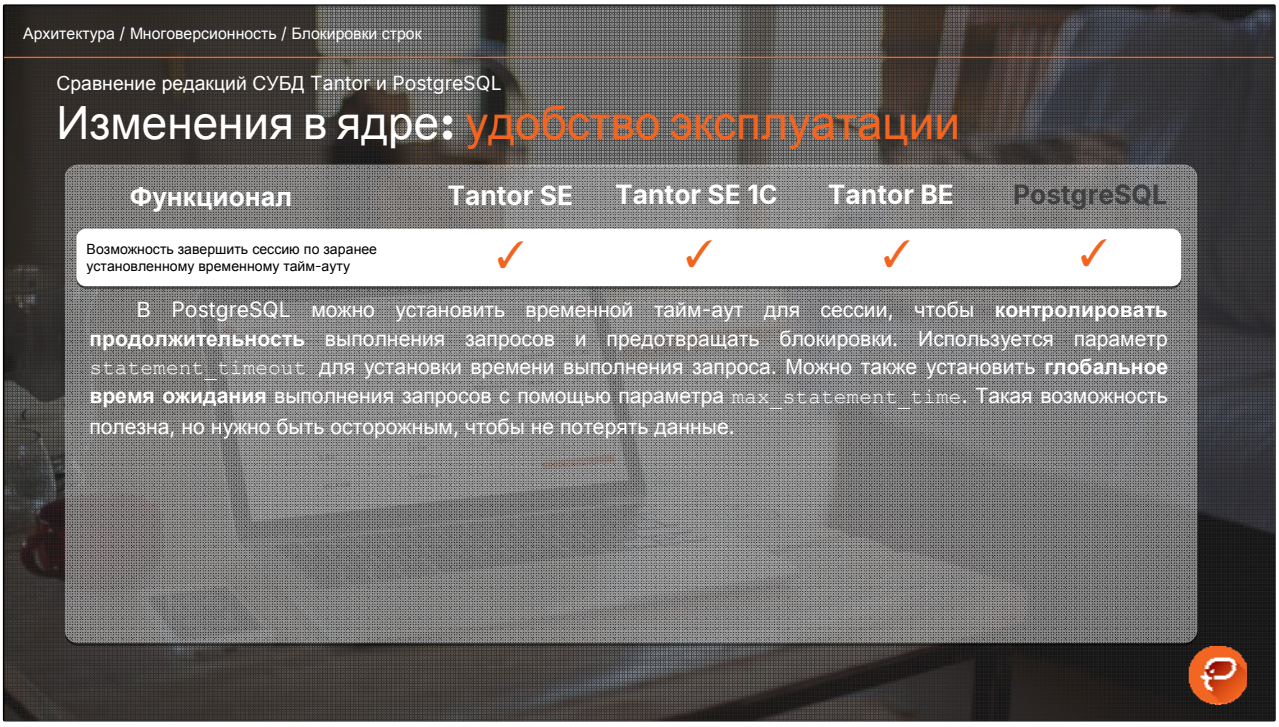
Признаком блокировки строки является заполнение поля xmax в заголовке версии строки.

Если придет транзакция не совместимая по уровню блокировки то она встает в очередь, пытаясь захватить номер транзакции 520.

Такое действие не предусмотрено — поэтому транзакция становится в очередь.

Остальные транзакции становятся в очередь за транзакцией 521.

В случае освобождения транзакции 520, транзакция 521 захватывает новую версию строки а следующая становится произвольная транзакция из «кучи» ожидающих транзакций. Можно сказать, что в очереди есть первый в очереди за получением блокировки и все остальные.



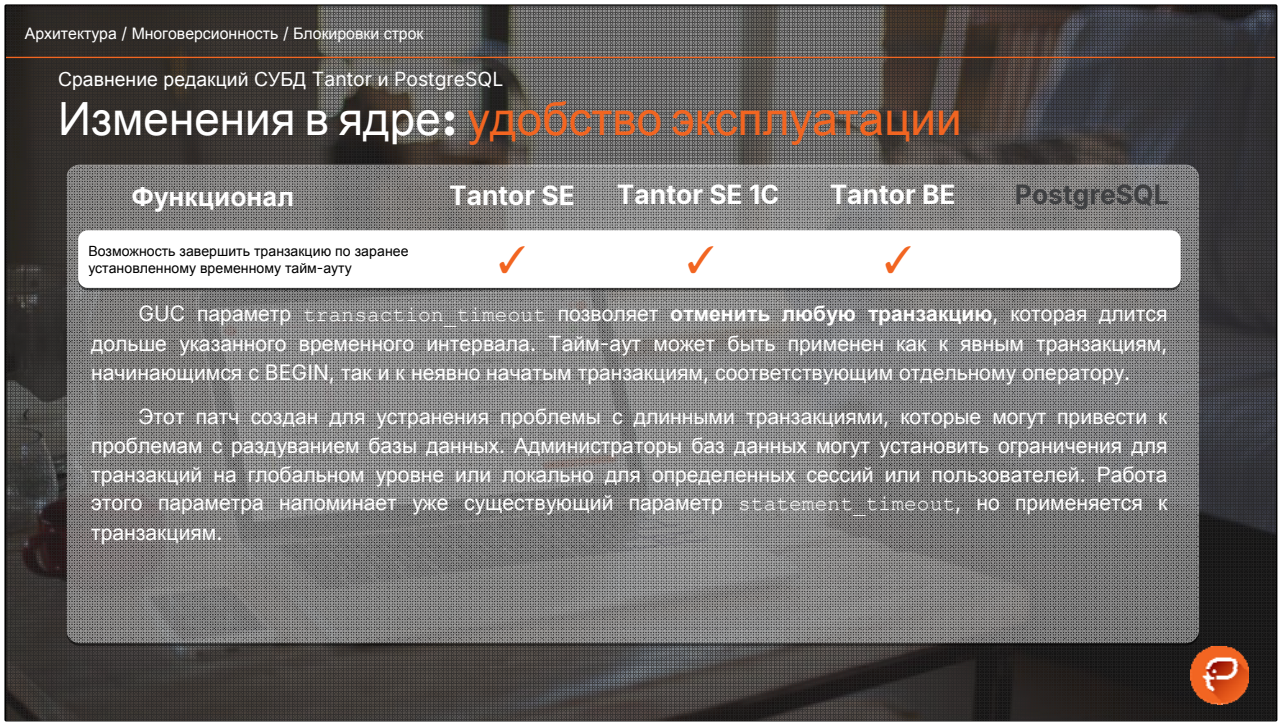
Возможность завершить сессию по заранее установленному временному тайм-ауту реализована и в версиях Tantor и PostgreSQL.

В PostgreSQL есть возможность установить временной тайм-аут для сессии, после которого она будет автоматически завершена. Эта функциональность может быть полезна для контроля продолжительности выполнения запросов или предотвращения блокировок и долгих операций.

Для установки временного тайм-аута для сессии в PostgreSQL можно использовать параметр конфигурации `statement_timeout`. Этот параметр определяет, сколько времени запрос может выполняться, прежде чем будет автоматически прерван.

Вы также можете установить глобальное время ожидания выполнения запросов с помощью параметра `max_statement_time`. Этот параметр устанавливает максимальное время выполнения для всех запросов в базе данных.

Эта возможность может быть полезной для предотвращения долгих запусков запросов или защиты от запросов, которые могут потенциально привести к блокировке или замедлению работы других сеансов. Однако ее следует использовать с осторожностью, чтобы убедиться в том, что установленное время не приведет к нежелательным прерываниям или потере данных.



Для всех версий СУБД Tantor добавлена возможность завершения транзакции по заранее установленному временному тайм-ауту

Добавлен GUC параметр `transaction_timeout`, позволяющий отменить любую транзакцию, которая продолжается дольше указанного интервала времени. Ограничение применяется как к явным транзакциям (начинающимся с BEGIN), так и к неявно начатым транзакциям, соответствующим отдельному оператору. Если это значение указано без единиц измерения, оно считается в миллисекундах. Значение нуль (по умолчанию) отключает тайм-аут.

Причины создания данного патча:

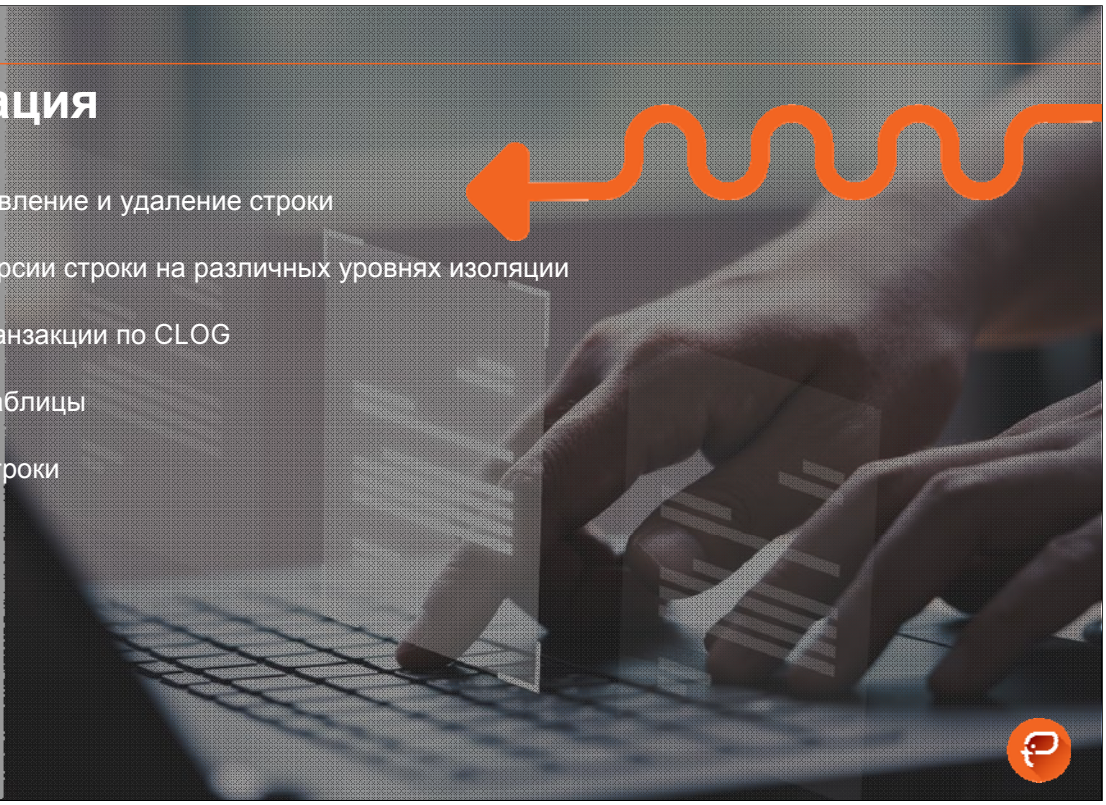
1.Существующие timeouts (а именно: `statement_timeout` + `idle_session_timeout`) не защищают от транзакций, состоящих из серии небольших операторов и коротких пауз между ними. Если происходит такое поведение (например, длинная серия быстрых UPDATE в цикле), то это может быть опасно, поскольку напрямую влияет на общее состояние БД (проблемы с раздуванием).

2.Администраторы баз данных могут свести к минимуму случаи длинных транзакций, устанавливая лимиты транзакций глобально, а также позволяя устанавливать их локально для определенных сеансов или для некоторых пользователей в редких случаях

Принцип работы напоминает statment_timeout, но в данном случае для транзакций.

Демонстрация

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



Многоверсионность

Часть 1. Вставка, обновление и удаление строки

Загрузим psql:

```
astra@tantor:~$ psql
psql (16.1)
Введите "help", чтобы получить справку.

postgres=#
```

Создадим произвольную таблицу.

```
postgres=# CREATE TABLE a(id integer);
CREATE TABLE
```

Посмотрим что получилось.

```
postgres=# \dt a
         Список отношений
 Схема  | Имя |   Тип   | Владелец
--------+-----+---------+----------
 public | a   | таблица | postgres
(1 строка)
```

Вставим первую строку в таблицу.

```
postgres=# INSERT INTO a VALUES(100);
INSERT 0 1
```

Посмотрим какой номер транзакции xmin.

```
postgres=# SELECT xmin, xmax, * FROM a;

xmin | xmax  | id
------+------+-----
 1567 |    0 | 100
(1 строка)
```

Получился 1567 это номер транзакции в которой была создана первая версия строки.

Начнем явную транзакцию.

```
postgres=# BEGIN;
BEGIN
```

Обновим первую строчку.

```
postgres=*# UPDATE a SET id = 200;
UPDATE 1
```

Обратимся и посмотрим что получилось

```
postgres=*# SELECT xmin, xmax, * FROM a;

xmin | xmax | id
------+------+-----
 1569 |    0 | 200
(1 строка)
```

Убедились в том, что транзакция видит свои изменения.

Как вы думаете что будет если обратиться в параллельной транзакции? id=100 или 200?

Во втором терминале обращаемся к таблице.

Загрузим `psql`.

```
astra@tantor:~$ psql
psql (16.1)
Введите "help", чтобы получить справку.
postgres=#
postgres=# SELECT xmin, xmax, * FROM a;

 xmin | xmax | id
------+------+-----
 1568 | 1569 | 100
(1 строка)
```

Обратите внимание, что xmax изменился, это значит что уже существует вторая версия строки но она еще не зафиксирована.

В первом терминале фиксируем транзакцию.

```
postgres=*# COMMIT;
COMMIT
```

Во втором терминале теперь видим вторую строку

```
postgres=# SELECT xmin, xmax, * FROM a;

 xmin | xmax | id
------+------+-----
 1569 |    0 | 200
(1 строка)
```

Теперь посмотрим как выглядит удаление. Откроем транзакцию в первом терминале.

```
postgres=# BEGIN;
BEGIN
```

Удаляем строчку.

```
postgres=*# DELETE FROM a;
```

```
DELETE 1
postgres=*# SELECT xmin, xmax, * FROM a;

xmin | xmax | id
------+------+----
(0 rows)
```

Первая транзакция не видит строчку,она удалена, но изменение пока не зафиксировано.

Во втором терминале:

```
postgres=# SELECT xmin, xmax, * FROM a;

xmin | xmax | id
------+------+-----
 1569 | 1570 | 200
(1 строка)
```

Строка еще видна но xmax опять изменился.

В первом терминале фиксируем транзакцию

```
postgres=*# COMMIT;
COMMIT
```

Во втором терминале теперь видим изменение;

```
postgres=# SELECT xmin, xmax, * FROM a;

 xmin | xmax | id
------+------+----
(0 rows)
```

Часть 2. Видимость версии строки на различных уровнях изоляции

Откроем первую транзакцию и вставим строку.

```
postgres=# BEGIN;
BEGIN
```

Посмотрим уровень изоляции.

```
postgres=*# SHOW transaction_isolation;

transaction_isolation
-----------------------
read committed
(1 строка)

postgres=*# INSERT INTO a VALUES(100);
INSERT 0 1
```

```
postgres=*# SELECT xmin, xmax, * FROM a;

xmin | xmax | id
------+------+-----
 1571 |    0 | 100
(1 строка)
```

Начнем вторую транзакцию во втором терминале и обратимся к таблице

```
postgres=# BEGIN;
BEGIN

postgres=*# SELECT xmin, xmax, * FROM a;
xmin | xmax | id
------+------+----
(0 строк)
```

Посмотрим уровень изоляции

```
postgres=*# SHOW transaction_isolation;

transaction_isolation
----------------------
read committed
(1 строка)
```

Пока новая строка не видна. Зафиксируем первую транзакцию

```
postgres=*# COMMIT;
COMMIT
```

Во втором окне повторно обратимся к таблице. Что увидим?

```
postgres=*# SELECT xmin, xmax, * FROM a;
xmin | xmax | id
------+------+-----
 1571 |    0 | 100
(1 строка)
```

Зафиксируем вторую транзакцию

```
postgres=*# COMMIT;
COMMIT
```

Изменения стали видны. Это и есть аномалия неповторяющегося чтения. Теперь в первом окне начнем транзакцию на уровне repeatable read; Вставим еще одну строку.

```
postgres=# BEGIN ISOLATION LEVEL REPEATABLE READ;
BEGIN
```

```
postgres=*# INSERT INTO a VALUES(200);
INSERT 0 1

postgres=*# SELECT xmin, xmax, * FROM a;
 xmin | xmax | id
------+------+-----
 1571 |    0 | 100
 1572 |    0 | 200
(2 строки)
```

Во второй транзакции обратимся к таблице в новой транзакции на том же уровне.

```
postgres=# BEGIN ISOLATION LEVEL REPEATABLE READ;
BEGIN

postgres=*# SELECT xmin, xmax, * FROM a;
xmin | xmax | id
------+------+-----
 1571 |    0 | 100
(1 строка)
```

Теперь фиксируем первую транзакцию.

```
postgres=*# COMMIT;
COMMIT
```

Обратимся во второй транзакции еще раз

```
postgres=*# SELECT xmin, xmax, * FROM a;
xmin | xmax | id
------+------+-----
 1571 |    0 | 100
(1 строка)
```

Изменения не видны. на этом уровне операторы транзакции работают только с одним снимком данных.

Зафиксируем вторую транзакцию.

```
postgres=*# COMMIT;
COMMIT
```

Часть 3. Состояние транзакции по CLOG

Откроем первую транзакцию и посмотрим после вставки состояние

```
postgres=# BEGIN;
BEGIN

postgres=*# INSERT INTO a VALUES(300);
INSERT 0 1
```

```
postgres=*# SELECT xmin, xmax, * FROM a;
xmin | xmax | id
------+------+-----
 1571 |    0 | 100
 1572 |    0 | 200
 1573 |    0 | 300
(3 строки)
```

Видим вставку третьей строки. посмотрим статус транзакции:

```
postgres=*# SELECT pg_xact_status('1573');
pg_xact_status
----------------
in progress
(1 строка)
```

Зафиксируем транзакцию и посмотрим статус.

```
postgres=*# COMMIT;
COMMIT

postgres=# SELECT pg_xact_status('1573');
 pg_xact_status
----------------
 committed
(1 строка)
```

Теперь посмотрим как поведет себя `CLOG` при откате транзакции.

```
postgres=# BEGIN;
BEGIN

postgres=*# INSERT INTO a VALUES(400);
INSERT 0 1

postgres=*# SELECT xmin, xmax, * FROM a;
 xmin | xmax | id
------+------+-----
 1571 |    0 | 100
 1572 |    0 | 200
 1573 |    0 | 300
 1574 |    0 | 400
(4 строки)

postgres=*# SELECT pg_xact_status('1574');
pg_xact_status
----------------
in progress
(1 строка)
```

```
postgres=*# ROLLBACK;
ROLLBACK

postgres=# SELECT pg_xact_status('1574');
pg_xact_status
----------------
aborted
(1 строка)


postgres=*# SELECT xmin, xmax, * FROM a;
xmin | xmax | id
------+------+-----
 1571 |    0 | 100
 1572 |    0 | 200
 1573 |    0 | 300
(3 строки)
```

Часть 4. Блокировки таблицы

В первой транзакции вставим новую строку и посмотрим блокировки с помощью `pg_locks`, для этого нам нужен pid обслуживающего процесса.

```
postgres=# SELECT pg_backend_pid();
pg_backend_pid
----------------
          12193
(1 строка)
```

Откроем транзакцию и обратимся к таблице

```
postgres=# BEGIN;
BEGIN

postgres=*# UPDATE a SET id = id + 1;
UPDATE 3

postgres=*# SELECT locktype, transactionid, mode, relation::regclass as obj
FROM pg_locks where pid = 12193;

  locktype    | transactionid |       mode       |   obj
--------------+---------------+------------------+----------
 relation     |               | AccessShareLock  | pg_locks
 relation     |               | RowExclusiveLock | a
 virtualxid   |               | ExclusiveLock    |
 transactionid |          1577 | ExclusiveLock    |
(4 строки)
```

Появилась блокировка на уровне таблицы `RowExclusiveLock` — накладывается в случае обновления строк.

Во втором окне построим индекс по таблице, предварительно посмотрим `pid` процесса.

```
postgres=# SELECT pg_backend_pid();
pg_backend_pid
----------------
          17210
(1 строка)

postgres=# CREATE INDEX ON a (id);
```

Транзакция подвисла. В первом терминале посмотрим что происходит во втором процессе.

```
postgres=*# SELECT locktype, transactionid, mode, relation::regclass as obj
FROM pg_locks where pid = 17210;
 locktype   | transactionid |     mode      | obj
------------+---------------+---------------+-----
 virtualxid |               | ExclusiveLock |
 relation   |               | ShareLock     | a
(2 строки)
```

Появилась блокировка `ShareLock` она не совместима `RowExclusiveLock` возникла блокировочная ситуация.

Зафиксируем первую транзакцию.

```
postgres=*# COMMIT;

COMMIT
```

Тут же срабатывает команда во втором окне.

```
CREATE INDEX
```

Часть 5. Блокировка строки

Начнем первую транзакцию:

```
postgres=# BEGIN;
BEGIN

postgres=*# UPDATE a SET id = id + 1 WHERE id=101;
UPDATE 1
```

Начнем вторую транзакцию:

```
postgres=# BEGIN;
BEGIN
postgres=*# UPDATE a SET id = id + 1 WHERE id=101;
```

Транзакция подвисла, сработала блокировка.

Зафиксируем первую транзакцию:

```
postgres=*# COMMIT;
COMMIT
```

Тут же срабатывает вторая.

```
UPDATE 0

postgres=*# COMMIT;
COMMIT
```

Обратите внимание обновление не произошло, теперь такой версии строки нет для обновления.

В первом терминале обратимся к таблице

```
postgres=# SELECT xmin, xmax, * FROM a;

xmin | xmax | id
------+------+-----
 1577 |    0 | 201
 1577 |    0 | 301
 1579 | 1580 | 102
(3 строки)
```

Удалим таблицу.

```
postgres=# DROP TABLE a;
DROP TABLE
```

Задание выполнено.

Практическая работа

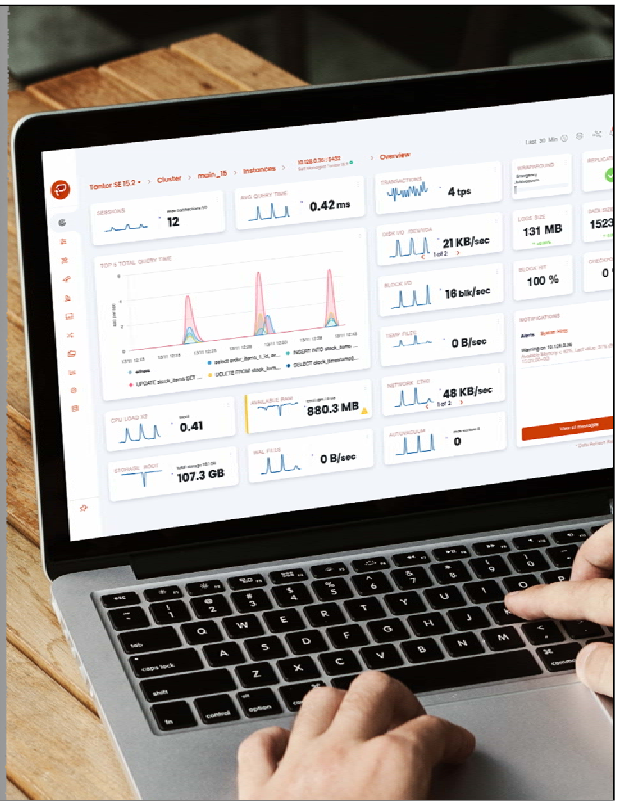
1. Вставка, обновление и удаление строки
2. Видимость версии строки на различных уровнях изоляции
3. Состояние транзакции по CLOG
4. Блокировка таблицы
5. Блокировка строки

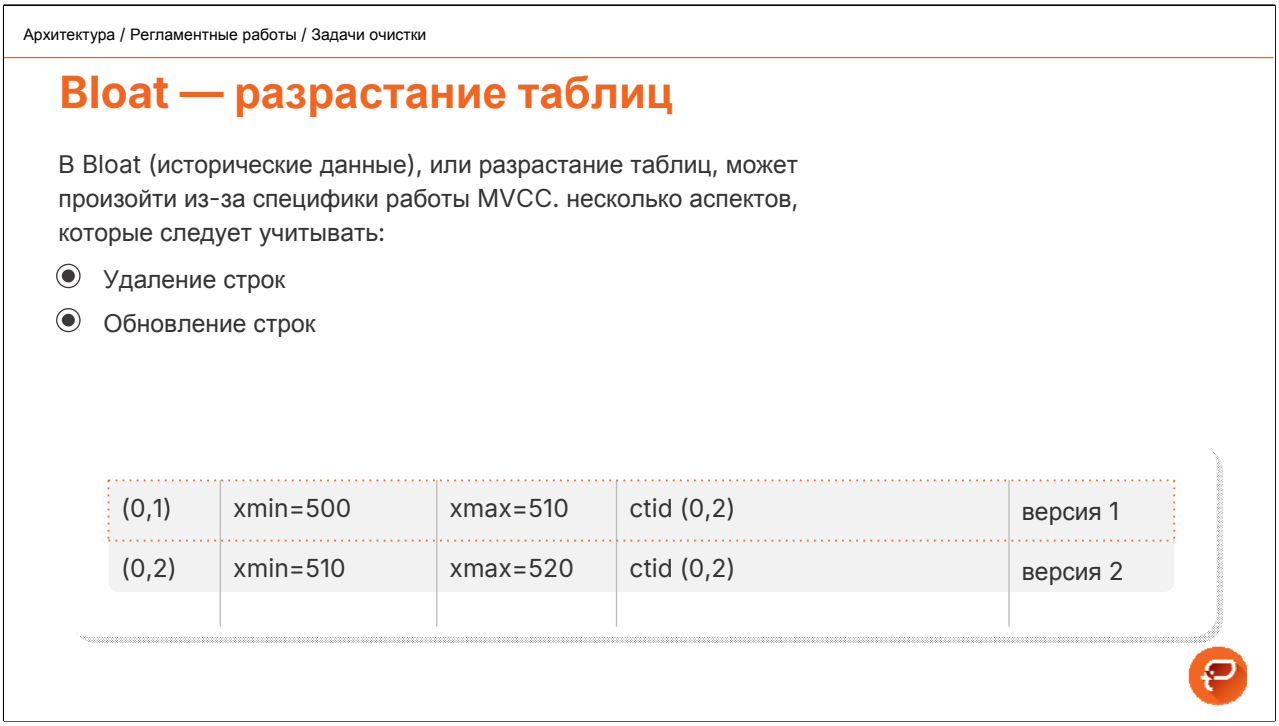
Дополнительное задание:

Сделать задание 1-5 с помощью pgAdmin

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
  - Задачи очистки
  - Автоочистка
  - Перестройка индексов
  - Заморозка
  - Демонстрация
  - Практическая работа
- Выполнение запросов
- Расширяемость





Bloat (исторические данные), или разрастание таблиц, происходит из-за специфики работы MVCC. Когда в PostgreSQL вносятся изменения в строки таблицы, старые версии строк сохраняются, чтобы обеспечить консистентность и изоляцию транзакций. Эти старые версии могут создавать дополнительный объем данных, что приводит к разрастанию таблицы.

Вот несколько аспектов, которые следует учитывать:

- **Удаление строк**: Когда строки удаляются, фактически они не удаляются из таблицы, а лишь помечаются как удаленные. Это приводит к тому, что старые версии строк остаются в системе, что может привести к увеличению размера таблицы.
- **Обновление строк**: Когда строки обновляются, PostgreSQL создает новую версию строки и сохраняет старую. Если такие операции выполняются часто, это может привести к увеличению размера таблицы.

Периодически нужно очищать исторические данные — еще одно название dead tuple.

Очистка строк

В процессе очистки исторических данных решаются задачи:

- Удаление исторических данных
- Обновление карты свободного пространства
- Обновление карты видимости

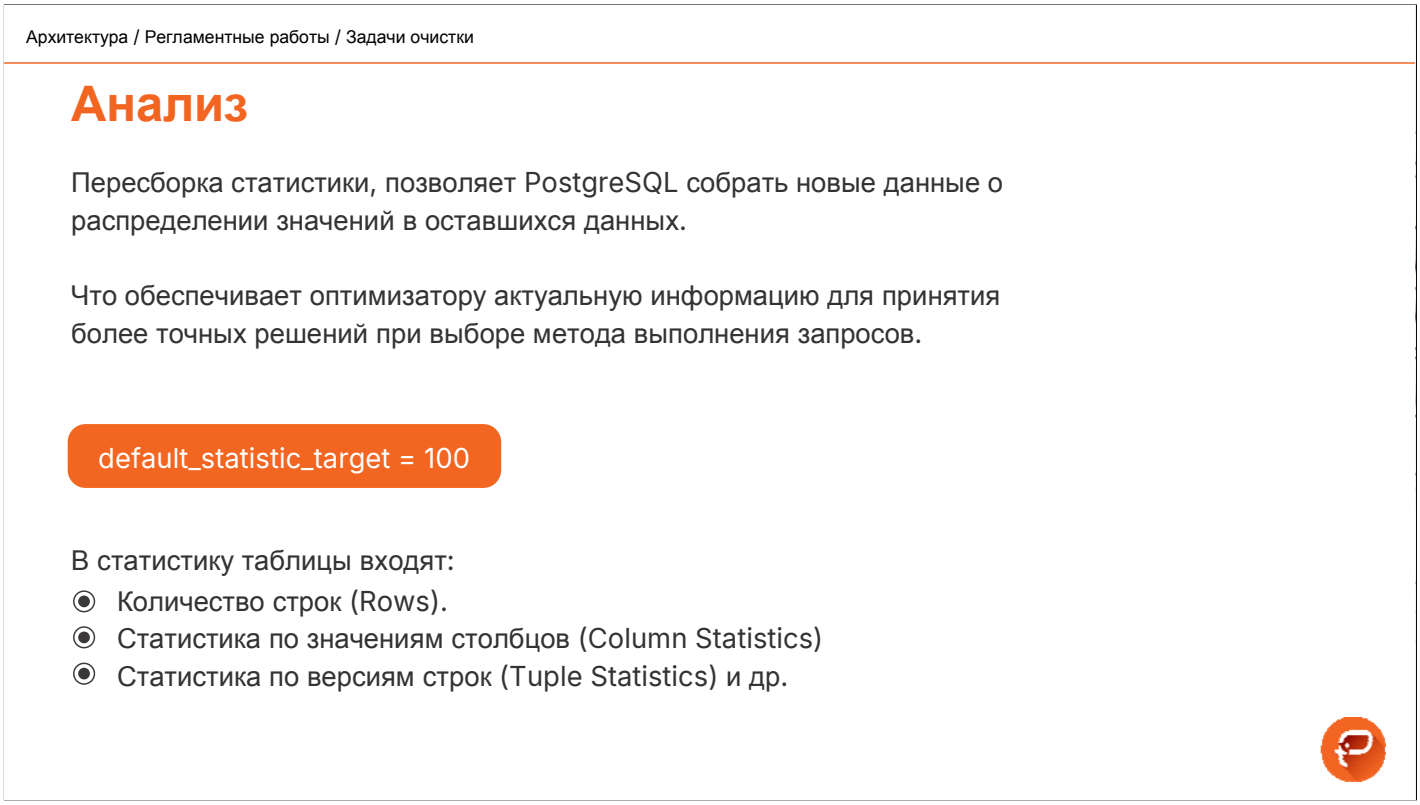
(0,1)	xmin=500	xmax=510	ctid (0,2)	версия 1
(0,2)	xmin=510	xmax=520	ctid (0,2)	версия 2

Давайте проанализируем, какие задачи решаются в процессе очистки исторических данных. В этом процессе выделяются несколько ключевых задач, и первая из них становится очевидной — необходимо удалить избыточные версии строк, оставляя только актуальные.

Задачу можно условно назвать «фазой один», включающей следующие подзадачи:

1. **Удаление исторических данных**: подзадача включает в себя активное удаление устаревших версий строк из базы данных, что важно для освобождения пространства и предотвращения накопления «мусора», который может привести к разрастанию таблицы.
2. **Обновление карты свободного пространства**: После удаления исторических данных необходимо обновить карту свободного пространства в базе данных, что позволяет эффективнее управлять доступным пространством и предотвращает фрагментацию данных.
3. **Обновление карты видимости**: обработанная страница данных таблицы становится «видимой», то есть в случае неизменности данных, ее можно в следующей итерации очистки не обрабатывать.

3 подзадачи представляют собой первую фазу очистки исторических данных. Однако, стоит отметить, что процесс может быть динамичным и требовать регулярного вмешательства для поддержания оптимальной производительности базы данных. Регулярное проведение операций очистки и обслуживания помогает предотвращать разрастание таблиц и обеспечивает эффективное управление данными.



После удаления устаревших данных первая фаза очистки может сделать текущую статистику устаревшей.

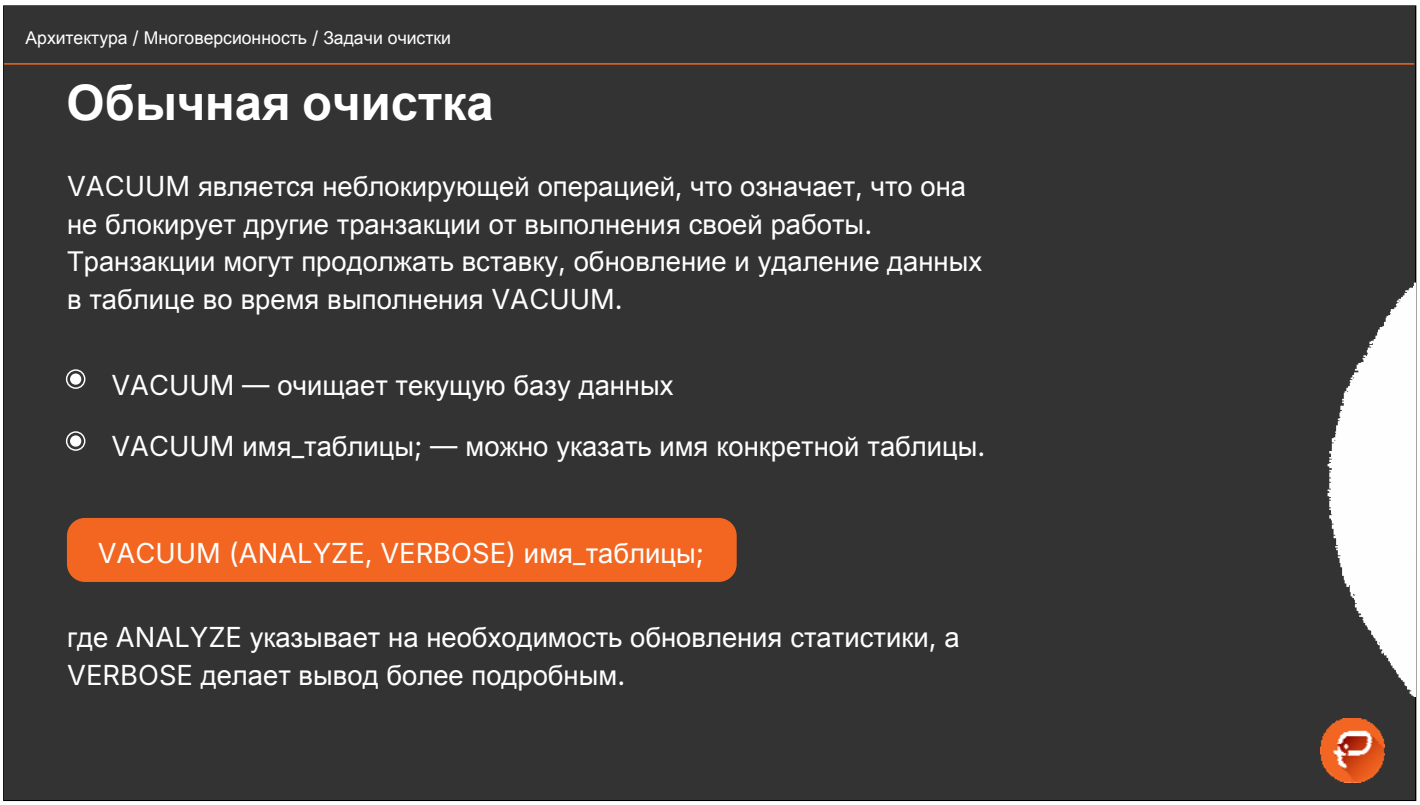
Вторая фаза, пересборка статистики, позволяет PostgreSQL собрать новые данные о распределении значений в оставшихся данных. Это обеспечивает оптимизатору актуальную информацию для принятия более точных решений при выборе метода выполнения запросов.

Использование параметра **`default_statistic_target`** позволяет настроить уровень детализации собираемой статистики. Вы можете увеличивать или уменьшать этот параметр в зависимости от требований вашей системы к точности планов выполнения запросов. Однако стоит помнить, что более высокая степень детализации требует больше ресурсов для сбора статистики, поэтому необходимо сбалансировать точность и производительность.

В PostgreSQL статистика таблицы включает в себя следующие основные аспекты:

- **Количество строк (Rows)**: Это показывает общее количество строк в таблице. Эта информация позволяет оптимизатору запросов оценить объем данных, с которым ему предстоит работать.
- **Статистика по значениям столбцов** (Column Statistics): Включает информацию о распределении значений в конкретных столбцах. Например, для числовых столбцов может быть рассчитана средняя, минимальная и максимальная значения, а также стандартное отклонение.
- **Статистика по версиям строк** (Tuple Statistics): Включает в себя информацию о физическом распределении кортежей в таблице. Это может быть полезно для оптимизации запросов с использованием различных операций объединения.

Более подробно про статистику поговорим позднее в разделе выполнения запросов.



Команда `VACUUM` в PostgreSQL не блокирует активности транзакций, но может влиять на параллельные запросы и доступ к данным во время выполнения.

В большинстве случаев, `VACUUM` является неблокирующей операцией, что означает, что она не блокирует другие транзакции от выполнения своей работы. Транзакции могут продолжать вставку, обновление и удаление данных в таблице во время выполнения `VACUUM`.

Команда `VACUUM` в PostgreSQL используется для очистки исторических данных в базе данных. Если у вас есть таблицы с большим количеством удаленных или устаревших строк, VACUUM может помочь освободить пространство и поддерживать более эффективное использование ресурсов.

Простая форма команды `VACUUM` может быть использована для выполнения очистки для всех таблиц в базе данных:

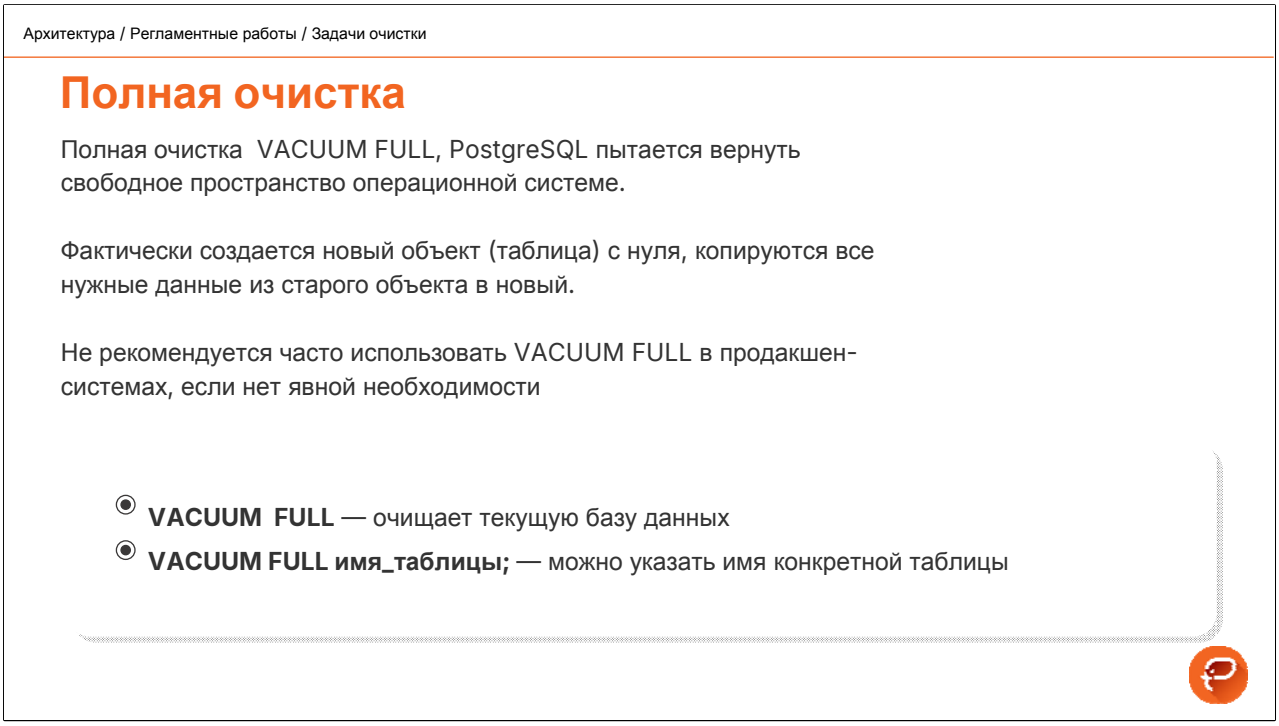
- `VACUUM` — очищает текущую базу данных
- `VACUUM` имя_таблицы; — можно указать имя конкретной таблицы.

Также у команды есть параметры к примеру:

```
VACUUM (ANALYZE, VERBOSE) имя_таблицы;
```

где `ANALYZE` указывает на необходимость обновления статистики, а `VERBOSE` делает вывод более подробным.

В примере очистка таблицы не происходит, но происходит пересбор статистики.



Обычная операция `VACUUM` в PostgreSQL обычно не возвращает свободное дисковое пространство операционной системе. Вместо этого, она помечает пространство, ранее занятое удаленными строками, как доступное для будущих операций внутри той же самой таблицы.

В случае полной очистки, такой как `VACUUM FULL`, PostgreSQL пытается вернуть свободное пространство операционной системе. Однако, как вы отметили, эта операция блокирует доступ к объекту, что может повлиять на возможность выполнения запросов к таблице в это время.

Использование `VACUUM FULL` следует оценивать осторожно, так как это более ресурсозатратная и блокирующая операция, и она может повлиять на производительность вашей базы данных.

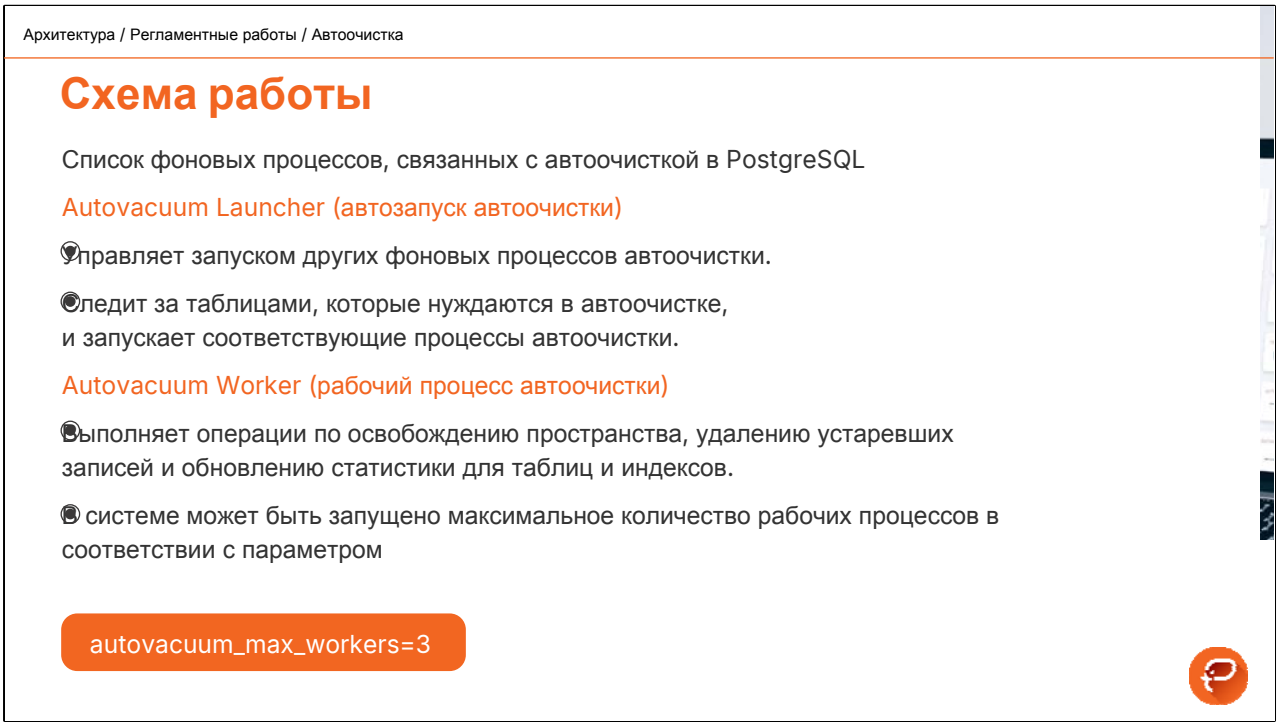
Также важно отметить, что в реальных сценариях обычно не требуется регулярно выполнять `VACUUM FULL`. Вместо этого, можно использовать автоматический механизм `AUTOVACUUM`, который управляет подобными операциями в более оптимальном режиме.

При выполнении операции `VACUUM FULL` фактически создается новый объект (таблица) с нуля, копируются все нужные данные из старого объекта в новый, и, по завершении операции, старый объект удаляется. Этот процесс называется «переписывание» таблицы.

Процесс переписывания при `VACUUM FULL` позволяет эффективно восстановить пространство, которое занимает таблица на диске. Однако, так как это создает новую копию таблицы и может быть ресурсоемким, обычно не рекомендуется часто использовать `VACUUM FULL` в продакшен-системах, если нет явной необходимости.

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
  - Задачи очистки
  - Автоочистка
  - Перестройка индексов
  - Заморозка
  - Демонстрация
  - Практическая работа
- Выполнение запросов
- Расширяемость



В PostgreSQL, фоновые процессы автоочистки играют ключевую роль в обеспечении эффективности баз данных. Они автоматически управляют освобождением пространства, оптимизацией таблиц и индексов, а также обновлением статистики для поддержания высокой производительности системы.

Вот более подробное описание двух основных фоновых процессов, связанных с автоочисткой:

- **Autovacuum Launcher (автозапуск автоочистки)** — процесс ответственен за координацию и контроль за запуском других фоновых процессов автоочистки. Его задача включает в себя мониторинг таблиц, требующих автоочистки, и запуск соответствующих рабочих процессов.
- **Autovacuum Worker (рабочий процесс автоочистки)** — когда таблица или индекс выявляет необходимость в автоочистке, один или несколько фоновых рабочих процессов автоочистки активируются. Эти процессы занимаются фактическим освобождением пространства, удалением устаревших записей и обновлением статистики, необходимой для оптимизации выполнения запросов.

Два ключевых фоновых процесса работают совместно, чтобы поддерживать целостность данных, эффективность запросов и предотвращать избыточное использование ресурсов базы данных. Важно отметить, что администратор базы данных может настраивать параметры автоочистки в зависимости от требований и характеристик конкретной системы для достижения оптимальной производительности.

Пример

Пример настройки процесса автовакуума:

```
autovacuum = on
autovacuum_max_workers =3
autovacuum_naptime = 1min

autovacuum_vacuum_scale_factor = 0.2
autovacuum_vacuum_threshold= 50

autovacuum_analyze_scale_factor = 0.1
autovacuum_analyze_threshold = 50
```

Процесс автовакуума проверяет статистику баз данных раз в 60 секунд
и в случае превышения порогов наличия исторических данных и автоанализа
производит очистку таблиц не более чем 3-мя процессами одновременно.

Настройка процесса автовакуума в PostgreSQL может включать в себя несколько параметров, которые определяют поведение этого процесса.

Вот несколько примеров настройки параметров:

●Включение/выключение автовакуума: `autovacuum` — параметр, который включает (`on`) или выключает (`off`) автовакуум. По умолчанию, этот параметр включен.

```
autovacuum = on
```

●Установка порога для запуска автовакуума: `autovacuum_vacuum_scale_factor` и `autovacuum_vacuum_threshold` - определяет порог и какой процент общего объема данных должен быть изменен, чтобы запустить автовакуум. Например, значение 0.2 означает, что автовакуум будет запущен, если 20% данных станут устаревшими или будут удалены.

```
autovacuum_vacuum_scale_factor = 0.2
autovacuum_vacuum_threshold= 50
```

●Количество рабочих процессов автовакуума: `autovacuum_max_workers` — определяет максимальное количество фоновых рабочих процессов автовакуума, которые могут выполняться одновременно.

```
autovacuum_max_workers = 3
```

●Параметры для анализа таблиц: `autovacuum_analyze_scale_factor` и `autovacuum_analyze_threshold` - определяют порог и масштаб для запуска процесса анализа таблицы.

```
autovacuum_analyze_scale_factor = 0.1
autovacuum_analyze_threshold = 50
```

Эти примеры позволяют настроить различные аспекты автовакуума в PostgreSQL в соответствии с требованиями вашей системы. Важно экспериментировать с этими параметрами и подстраивать их в зависимости от конкретных характеристик базы данных и её нагрузки.

`autovacuum_naptime` — параметр устанавливает минимальное время в секундах между итерациями автовакуума, может влиять на частоту запуска.

```
autovacuum_naptime = 1min
```

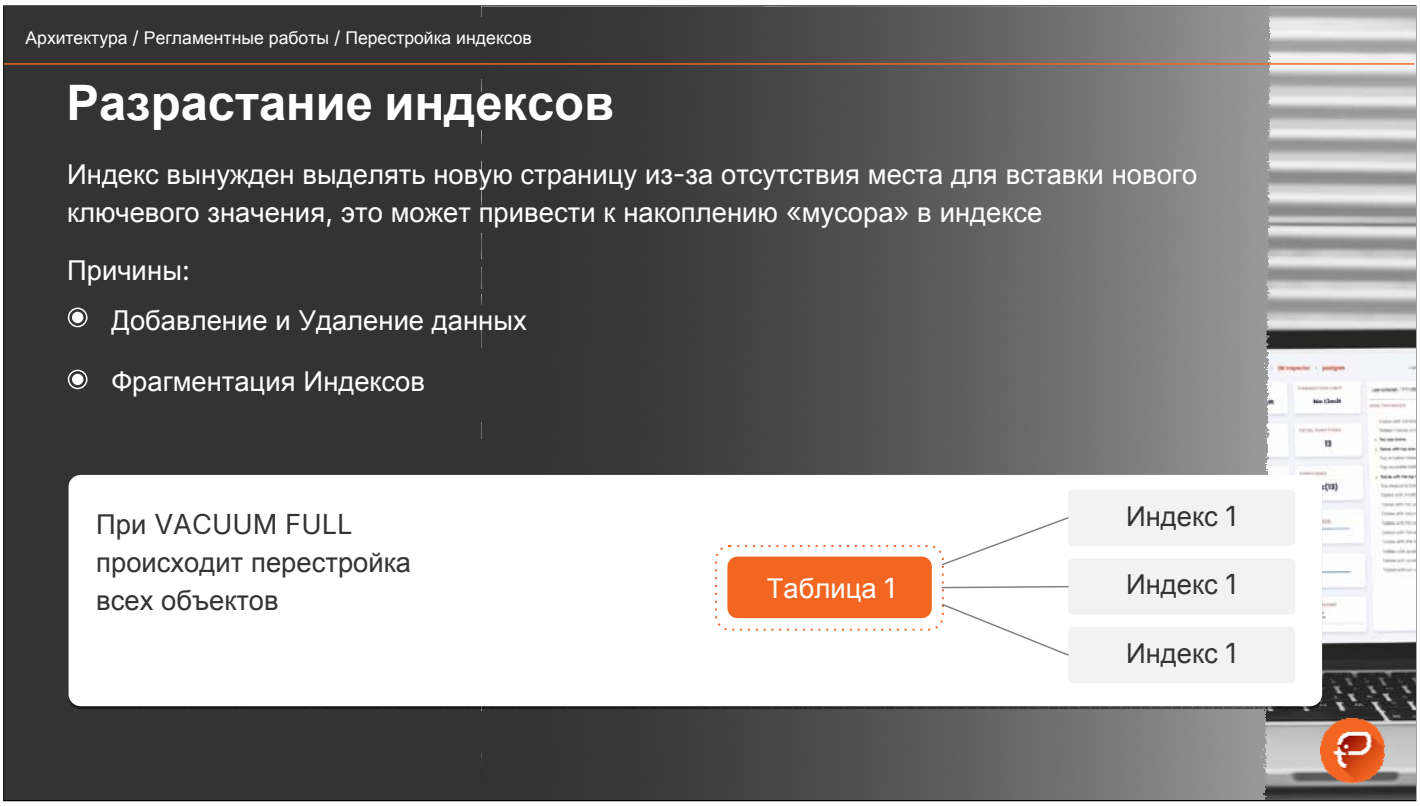
Процесс автовакуума проверяет статистику баз данных раз в 60 секунд и в случае превышения порогов наличия исторических данных и автоанализа производит очистку таблиц не более чем 3-мя процессами одновременно.

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Регламентные работы:

- Задачи очистки
- Автоочистка
- Перестройка индексов
- Заморозка
- Демонстрация
- Практическая работа



В PostgreSQL, когда таблицы накапливают все больше исторических данных, аналогичная ситуация может возникнуть и с индексами. Необходимость в управлении этими историческими данными в индексах также становится актуальной. Интересно отметить, что проблемы с разрастанием индексов могут возникнуть даже раньше, чем у самих таблиц, и достигнуть значений, которые негативно сказываются на производительности базы данных.

В PostgreSQL, когда индекс вынужден выделять новую страницу из-за отсутствия места для вставки нового ключевого значения, это может привести к накоплению «мусора» в индексе. Такие ситуации могут возникать при частых операциях добавления, обновления и удаления данных, особенно в больших таблицах.

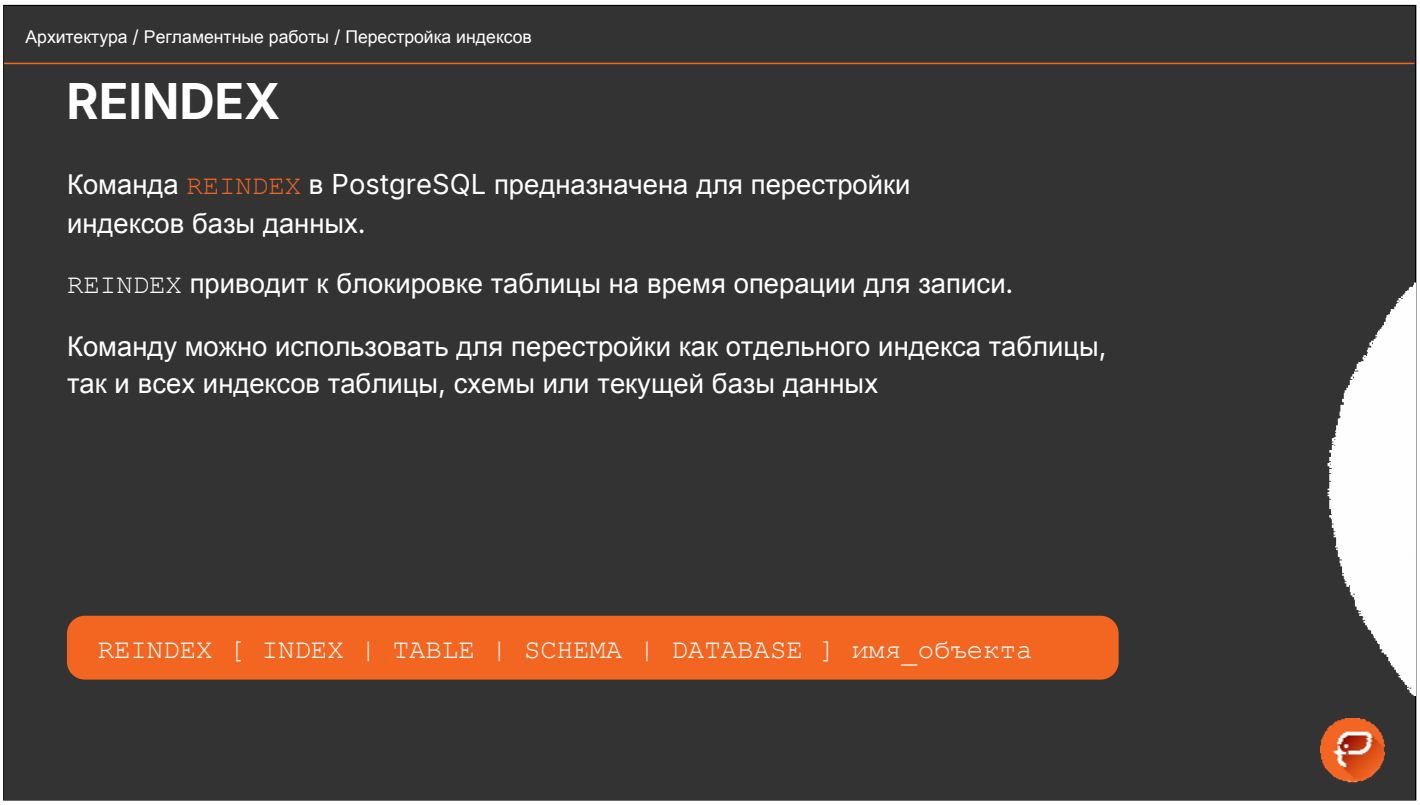
Проблема с «мусором» заключается в том, что эти выделенные страницы не всегда могут быть полностью освобождены и возвращены файловой системе. Вместо этого, они могут оставаться в индексе в виде свободных, но не освобожденных блоков.

Для эффективного решения этой проблемы в PostgreSQL предусмотрена команда `VACUUM FULL`, которая позволяет проводить перестройку индексов. Этот процесс становится неотъемлемой частью управления базой данных, позволяя удалить избыточные данные из индексов и восстановить их эффективность. Таким образом, внимательное внедрение стратегий по управлению историческими данными в индексах в PostgreSQL становится ключевым элементом поддержания оптимальной производительности и стабильности базы данных.

Разрастание индексов в базе данных может происходить из-за нескольких основных причин:

- **Добавление данных**: Когда в таблицу добавляются новые записи, индексы должны обновляться для учета новых значений. Это может привести к увеличению размера индексов.
- **Удаление данных**: Удаление записей из таблицы также требует обновления индексов. Однако, удаленные данные не всегда могут быть полностью освобождены, что может привести к накоплению «мусора» в индексах.
- **Обновление Значений**: Когда значения в колонках, используемых в индексах, обновляются, индексы также требуют обновления. Это может привести к увеличению размера индексов, особенно если такие операции обновления происходят часто.
- **Фрагментация Индексов**: Постепенная фрагментация индексов может произойти из-за частых операций вставки, обновления и удаления. Фрагментированные индексы могут замедлить производительность запросов.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/sql-reindex.html#reindex



Команда `REINDEX` в PostgreSQL предназначена для перестройки индексов базы данных. Это может быть полезно в ситуациях, когда производительность запросов страдает из-за фрагментации индексов или возникают другие проблемы с ними. Однако важно учесть, что выполнение команды `REINDEX` приводит к блокировке таблицы на время операции для записи.

Команда используется с различными параметрами в зависимости от того, что требуется перестроить.

Например, для перестройки индекса используется следующий синтаксис:

```
REINDEX INDEX имя_индекса;
```

Если необходимо перестроить всю таблицу, применяется следующая форма команды:

```
REINDEX TABLE имя_таблицы;
```

Также можно перестроить индексы в рамках конкретной схемы или даже всю базу данных:

```
REINDEX SCHEMA имя_схемы;
REINDEX DATABASE имя_базы_данных;
```

При использовании команды `REINDEX` важно выбирать оптимальное время для выполнения, чтобы минимизировать влияние на работу приложений. Рекомендуется проводить операцию в период низкой активности базы данных или внимательно планировать ее выполнение, чтобы избежать блокировок в критические моменты. Мониторинг активности с использованием pg_stat_activity поможет следить за процессом и предпринимать необходимые меры предосторожности.

Команду можно использовать для перестройки как отдельного индекса таблицы, так и всех индексов таблицы, схемы или текущей базы данных

```
REINDEX [ INDEX | TABLE | SCHEMA | DATABASE ] имя_объекта
```

https://docs.tantorlabs.ru/tdb/ru/15_4/se/sql-reindex.html#reindex

REINDEX CONCURENTLY

Восстановление индекса в PostgreSQL может нарушить обычную работу базы данных, блокируя таблицу от записей на время построения индекса

Чтобы избавится от блокировки на запись можно использовать `REINDEX.. CONCURENTLY`

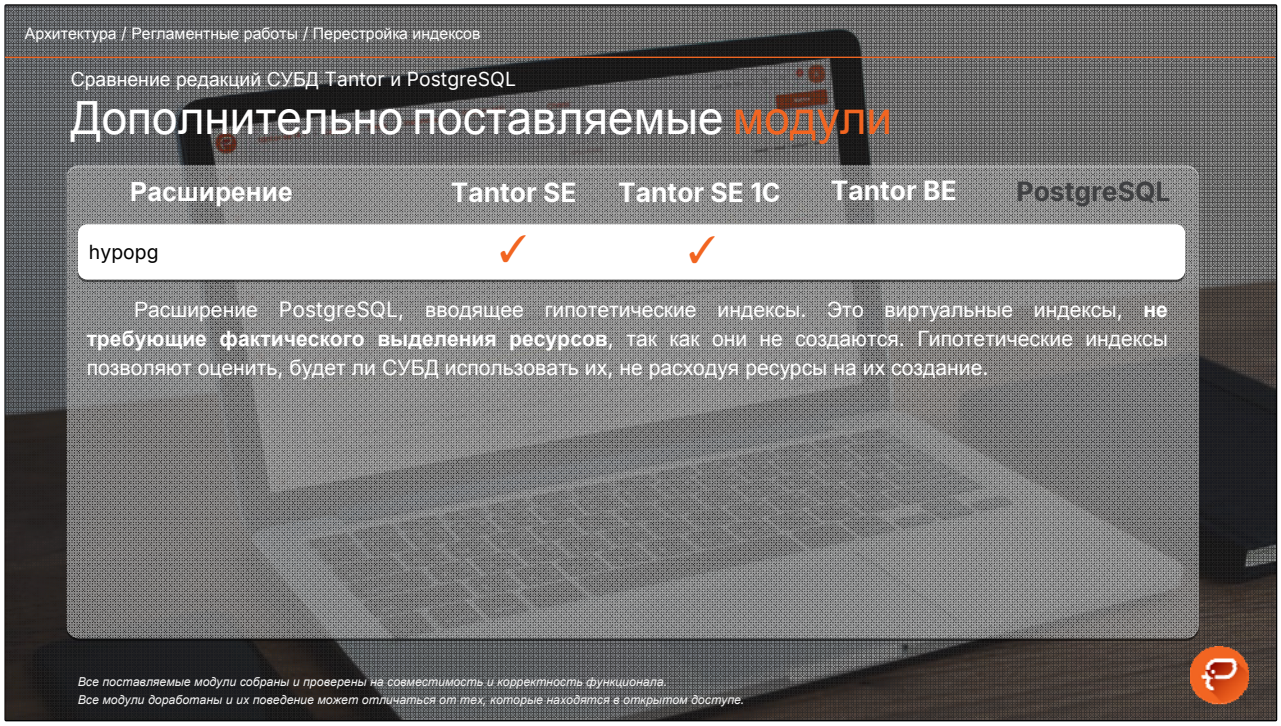
- `REINDEX INDEX ….  CONCURRENTLY;`
- `REINDEX TABLE ….  CONCURRENTLY;`

Перестройка индекса может привести к ошибке.

Восстановление индекса в PostgreSQL может нарушить обычную работу базы данных, блокируя таблицу от записей на время построения индекса. Это особенно критично в живой производственной базе данных, где большие таблицы могут занимать много часов для индексации. PostgreSQL поддерживает метод восстановления индексов с минимальной блокировкой записей, который позволяет продолжать нормальные операции во время восстановления. Этот метод требует больше работы и времени, но полезен в производственной среде.

Чтобы избавиться от блокировки на запись, можно использовать опцию `REINDEX CONCURRENTLY`. Однако, применение этого параметра может привести к увеличенной нагрузке на процессор, память и ввод-вывод, что может замедлить другие операции в системе. Также возможно, что перестройка индекса приведет к ошибке.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/sql-reindex.html#reindex

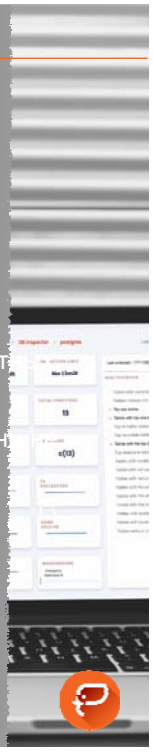


Модуль `hypopg` — доступен в версиях Tantor SE и Tantor SE 1C:

`hypopg` — это расширение PostgreSQL, добавляющее поддержку гипотетических индексов. Гипотетический или «виртуальный индекс» — это индекс, который на самом деле не существует, и следовательно, не требует ресурсов CPU или диска для его создания. Гипотетические индексы предоставляют возможность узнать будет ли СУБД использовать эти индексы или нет, не тратя ресурсы на их создание.

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 конкрет команд без создания индекса
- расширение позволяет скрыть любые существующие индексы в текущей сессии чтобы он влияли на планировщик
- гипотетические индексы создаются функцией `hypopg_create_index('CREATE INDEX.` которой передаётся текст команды создания индекса



Гипотетический или виртуальный индекс — это индекс, который не существует. 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 долгое. Расширение позволяет создать определение индексов, существующих только в текущей сессии и не влияющие 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 обычной командой `EXPLAIN`.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/hypopg.html

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
  - Задачи очистки
  - Автоочистка
  - Перестройка индексов
  - Заморозка
  - Демонстрация
  - Практическая работа
- Выполнение запросов
- Расширяемость

Переполнение счетчика

Для 32-битного счетчика транзакций (XID) в PostgreSQL максимальное значение равно 4,294,967,295.

Чтобы не происходило переполнение счетчика, старые версии строк замораживаются и старые данные поля xmin можно использовать повторно.

В более современных версиях PostgreSQL, таких как Tantor SE и других, используется 64-битный счетчик транзакций. Это увеличивает максимальное значение счетчика транзакций значительно, предотвращая переполнение и обеспечивая более долгий срок службы базы данных при высокой активности транзакций.

Переполнение счетчика транзакций и заморозка версий строк в PostgreSQL связаны через использование многоверсионности (MVCC).

MVCC в PostgreSQL предполагает создание различных версий строки для разных транзакций. Это позволяет транзакциям работать параллельно, избегая блокировок на чтение данных. Каждая транзакция видит свою собственную «замороженную» версию данных на момент ее начала.

Счетчик транзакций (Transaction ID, XID) используется 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 (в более поздних версиях PostgreSQL существуют механизмы для обработки переполнения, но об этом мы говорили ранее).

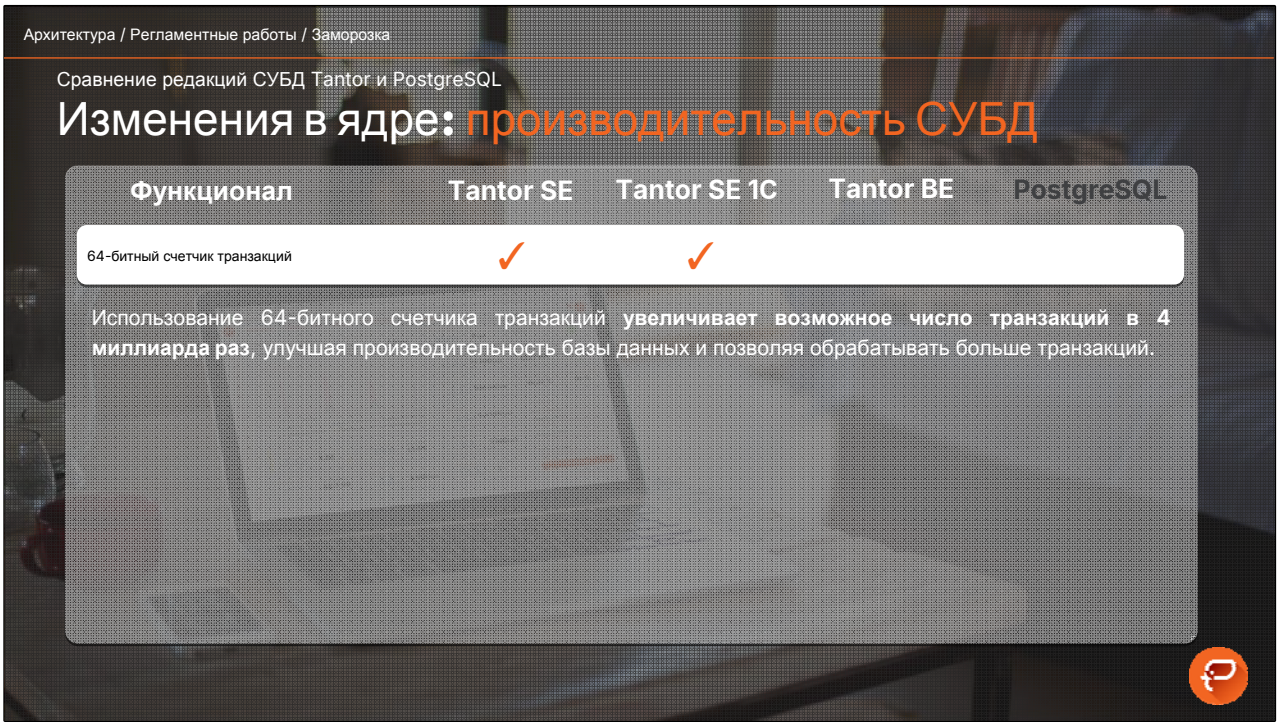
Если счетчик транзакций переполняется, это может привести к тому, что более новые транзакции будут восприниматься как более старые, что может вызвать ошибки и несогласованность в данных. Поэтому важно следить за возможностью переполнения счетчика транзакций и принимать соответствующие меры, такие как переход к более новым версиям PostgreSQL с 64-битным счетчиком транзакций.

Чтобы не происходило переполнение счетчика, старые версии строк замораживаются и старые данные поля xmin можно использовать повторно .

Заморозка версий строк (Freeze): В этой фазе VACUUM замораживает версии строк, устаревшие для транзакций с самыми высокими идентификаторами транзакций (XID). Таким образом, старые версии строк фактически «замораживаются» и могут быть использованы повторно. Это помогает предотвратить переполнение счетчика транзакций и обеспечивает более эффективное использование пространства.

Для 32-битного счетчика транзакций (XID) в PostgreSQL максимальное значение равно 4,294,967,295. Когда этот предел достигается, происходит переполнение, что может привести к проблемам согласованности данных, таким как несовместимость транзакций и ошибки при попытке доступа к данным.

В более современных версиях PostgreSQL, таких как Tantor SE и других, используется 64-битный счетчик транзакций. Это увеличивает максимальное значение счетчика транзакций значительно, предотвращая переполнение и обеспечивая более долгий срок службы базы данных при высокой активности транзакций.



Поговорим об изменениях в ядре и производительности СУБД:

В редакциях TantorSE и Tantor SE 1C внесены улучшения, направленные на повышение производительности СУБД, в частности, внедрен 64-битный счетчик транзакций.

При использовании 64-битного счетчика транзакций, количество возможных транзакций увеличивается до 2 в 32 степени, то есть примерно в 4 миллиарда раз, что существенно увеличивает время до переполнения счетчика.

Благодаря этому системе требуется меньше времени на «заморозку» данных, что позволяет улучшить общую производительность базы данных. С уменьшением необходимости в «заморозке» данных, уменьшается вероятность простоев в работе базы данных.

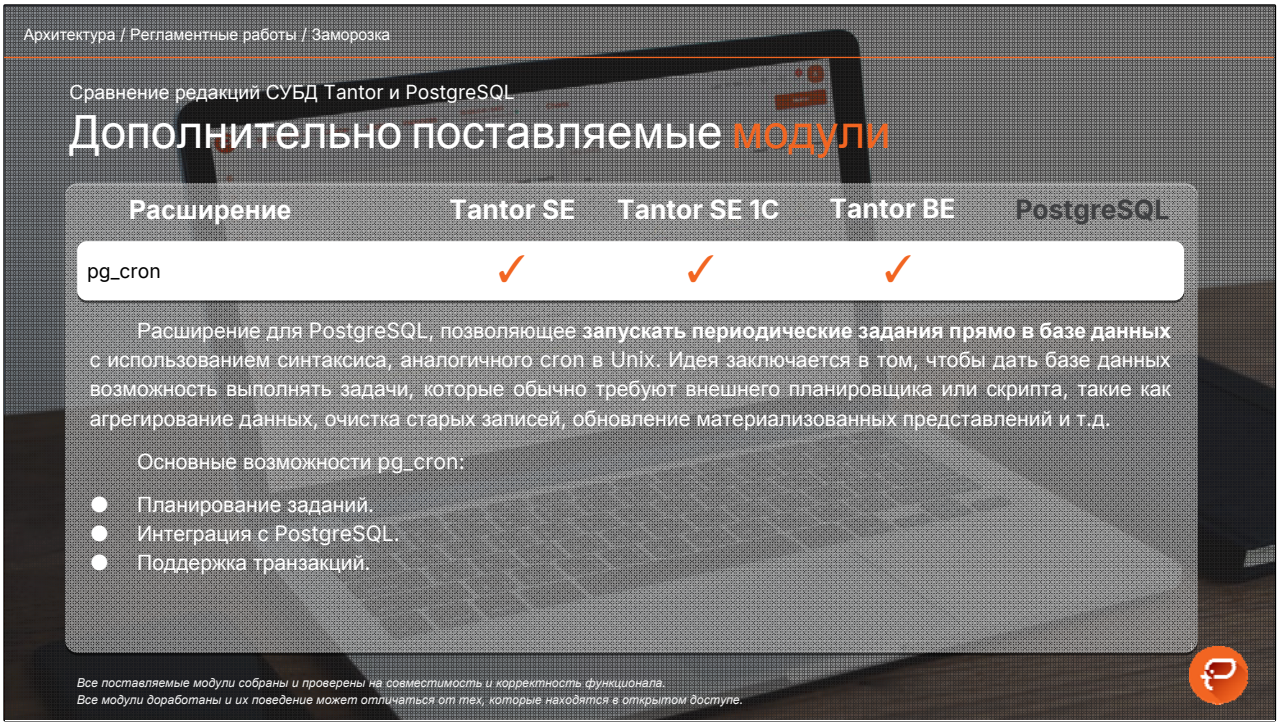
И вследствие этого система может обрабатывать гораздо большее количество транзакций, что делает ее более подходящей для больших и высоконагруженных баз данных.

В PostgreSQL можно явно вызвать третью фазу, заморозку версий строк (Freeze), командой `VACUUM FREEZE`. Это может быть полезно в ситуациях, когда вы хотите явно выполнить заморозку версий строк в определенный момент времени, независимо от автоматического выполнения VACUUM.

Команда `VACUUM FREEZE` замораживает версии строк для всех таблиц в базе данных, применяя третью фазу VACUUM ко всем устаревшим версиям строк. Это может быть полезно, например, когда вы заметили увеличение количества старых версий строк и хотите принудительно выполнить заморозку для управления счетчиком транзакций.

Применяется только в том случае, если счетчик транзакций 32-битный. Tantor SE использует 64 битный счетчик транзакций. В заморозке в таком случае нет необходимости.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/differences.html



Модуль `pg_cron` — также доступен во всех версиях Tantor.

`pg_cron` — это расширение, которое позволяет запускать периодические задания напрямую в базе данных, используя синтаксис, подобный cron в Unix.

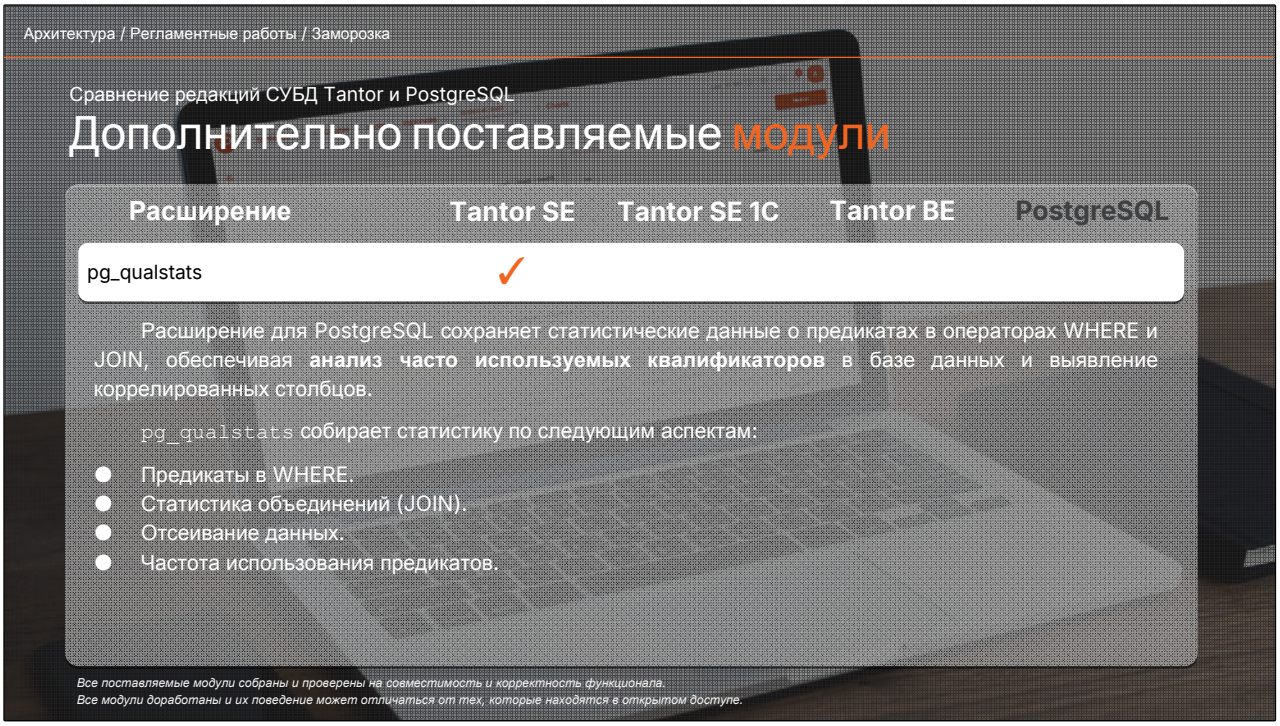
Основная идея заключается в том, чтобы позволить базе данных самой выполнять задачи, которые обычно требуют внешнего планировщика или скрипта, например, агрегирование данных, очистка старых записей, обновление материализованных представлений и т.д.

Основные возможности `pg_cron`:

●Планирование заданий. `pg_cron` может быть настроен для автоматического запуска SQL-заданий по расписанию. Это может быть очень полезно для регулярного обновления данных или выполнения обслуживающих операций, таких как очистка таблиц.

●Интеграция с PostgreSQL. Так как `pg_cron` является расширением PostgreSQL, он прекрасно интегрируется с системой управления базами данных. Это означает, что вы можете использовать его для автоматизации операций, которые вы бы обычно выполняли вручную в PostgreSQL.

●Поддержка транзакций. Задания, запускаемые с помощью `pg_cron`, могут использовать транзакции, что делает этот инструмент очень гибким для различных задач обслуживания базы данных.



Модуль `pg_qualstats` — доступен в версии Tantor SE.

Сохраняет статистические данные по найденным предикатам в операторах WHERE и предложениях JOIN.

Это позволит проанализировать наиболее часто выполняемые квалификаторы (предикаты) в вашей базе данных, а также идентифицировать коррелированные столбцы, определяя, какие столбцы чаще всего запрашиваются вместе.

pg_qualstats собирает статистику по следующим аспектам:

- Предикаты в WHERE: сбор статистики по условиям в секции WHERE запроса позволяет определить, какие условия часто используются и как они влияют на производительность.
- Статистика объединений (JOIN): понимание того, какие предикаты используются в операциях объединения, помогает определить, какие индексы могут быть полезны для улучшения производительности объединения.
- Отсеивание данных: статистика по предикатам, которые используются для отсеивания данных, помогает понять, какие данные часто запрашиваются и какие индексы могут быть полезны для улучшения производительности.
- Частота использования предикатов: понимание того, как часто используются определенные предикаты, может помочь в оптимизации запросов и индексов.

Демонстрация

1. Обычная очистка таблицы
2. Анализ таблицы
3. Перестройка индекса
4. Полная очистка

Регламентные работы

Часть 1. Обычная очистка таблицы

Загрузим psql

```
astra@tantor:~$ psql
psql (16.1)
Введите "help", чтобы получить справку.
postgres=#
```

Создадим произвольную таблицу:

```
postgres=# CREATE TABLE a
      (id integer primary key generated always as identity,
       t  char(2000)) WITH (autovacuum_enabled = off);
CREATE TABLE

postgres=# INSERT INTO a(t) SELECT to_char(generate_series(1,10000),'9999');
INSERT 0 10000
```

Посмотрим что получилось.

```
postgres=# \d a
                                Table "public.a"
Column |      Type       | Collation | Nullable |           Default
-------+-----------------+-----------+----------+---------------------------
---
id     | integer         |           | not null | generated always as
identity
t      | character(2000) |           |          |
Indexes:
   "a_pkey" PRIMARY KEY, btree (id)
```

Обратите внимание создан первичный ключ и индекс.

Узнаем размер таблицы и индекса в байтах:

```
postgres=# SELECT pg_table_size('a');
pg_table_size
---------------
     20512768
(1 строка)


postgres=# SELECT pg_table_size('a_pkey');
pg_table_size
---------------
       245760
(1 строка)
```

Обновим 50% строк:

```
postgres=# UPDATE a set t= t || 'a' where id > 5000;
UPDATE 5000
```

Посмотрим размеры объектов .

```
postgres=# SELECT pg_table_size('a');
pg_table_size
---------------
     30752768
(1 строка)


postgres=# SELECT pg_table_size('a_pkey');

pg_table_size
---------------
       360448
(1 строка)
```

Они также увеличились. Очистим таблицу и индекс:

```
postgres=# VACUUM a;
VACUUM

postgres=# SELECT pg_table_size('a'); SELECT pg_table_size('a_pkey');
pg_table_size
---------------
     30760960
(1 строка)

pg_table_size
---------------
       360448
(1 строка)
```

Размер остался таким же. Еще раз обновим строки.

```
postgres=# UPDATE a set t= t || 'a' where id > 5000;
UPDATE 5000

postgres=# SELECT pg_table_size('a'); SELECT pg_table_size('a_pkey');
pg_table_size
---------------
     30760960
(1 строка)


pg_table_size
---------------
       360448
(1 строка)
```

Опять размер не изменился. Произошло это потому что было использовано очищенное пространство.

К примеру предположим, что пропущен цикл очистки

```
postgres=# UPDATE a set t= t || 'a' where id > 5000;
UPDATE 5000

postgres=# UPDATE a set t= t || 'a' where id > 5000;
UPDATE 5000

postgres=# SELECT pg_table_size('a'); SELECT pg_table_size('a_pkey');
pg_table_size
```

```
---------------
     51249152
(1 строка)


pg_table_size
---------------
       466944
(1 строка)
```

Размер объектов опять вырос.

```
postgres=# VACUUM a;
VACUUM

postgres=# SELECT pg_table_size('a'); SELECT pg_table_size('a_pkey');
pg_table_size
---------------
     51249152
(1 строка)


pg_table_size
---------------
       466944
(1 строка)
```

Даже после очистки размер не уменьшается

Часть 2. Анализ таблицы

Так как произошло несколько циклов обновлений, посмотрим насколько актуальна осталась статистика. Сначала обратимся к системному каталогу.

```
postgres=# SELECT reltuples FROM pg_class WHERE relname='a';
reltuples
-----------
     8333
(1 строка)
```

Получили что в таблице a у нас содержится 8333 строки.

Теперь обратимся к таблице.

```
postgres=# SELECT count(*) FROM a;
count
-------
10000
(1 строка)
```

Оказалось что строк больше.Статистика всегда приблизительна. Вызовем вторую фазу анализа.

```
postgres=# ANALYZE a;
ANALYZE
```

Теперь статистика стала более точной.

```
postgres=# SELECT reltuples FROM pg_class WHERE relname='a';

 reltuples
-----------
     10000
(1 строка)
```

Часть 3. Перестройка индекса

Посмотрим какой размер объектов:

```
postgres=# SELECT pg_table_size('a'); SELECT pg_table_size('a_pkey');
pg_table_size
---------------
     51249152
(1 строка)


pg_table_size
---------------
       466944
(1 строка)
```

Сейчас в таблице a один только индекс. Перестроим его.

```
postgres=# REINDEX TABLE a;
REINDEX

postgres=# SELECT pg_table_size('a'); SELECT pg_table_size('a_pkey');
pg_table_size
---------------
     51249152
(1 строка)


pg_table_size
---------------
       245760
(1 строка)
```

Размер индекса уменьшился, размер таблицы остался неизменным

Часть 4. Полная очистка

```
postgres=# VACUUM FULL a;
VACUUM
```

Поглядим размер объектов 5

```
postgres=# SELECT pg_table_size('a'); SELECT pg_table_size('a_pkey');

pg_table_size
---------------
     20488192
(1 строка)


pg_table_size
---------------
       245760
(1 строка)
```

Размер таблицы уменьшился.

Удалим таблицу.

```
postgres=# DROP TABLE a;
DROP TABLE
```

Задание выполнено.

Практика

1. Обычная очистка таблицы
2. Анализ таблицы
3. Перестройка индекса
4. Полная очистка
5. Расширение HypoPG

Дополнительное задание:

Сделать задание 1-4 с помощью pgAdmin

Спасибо!

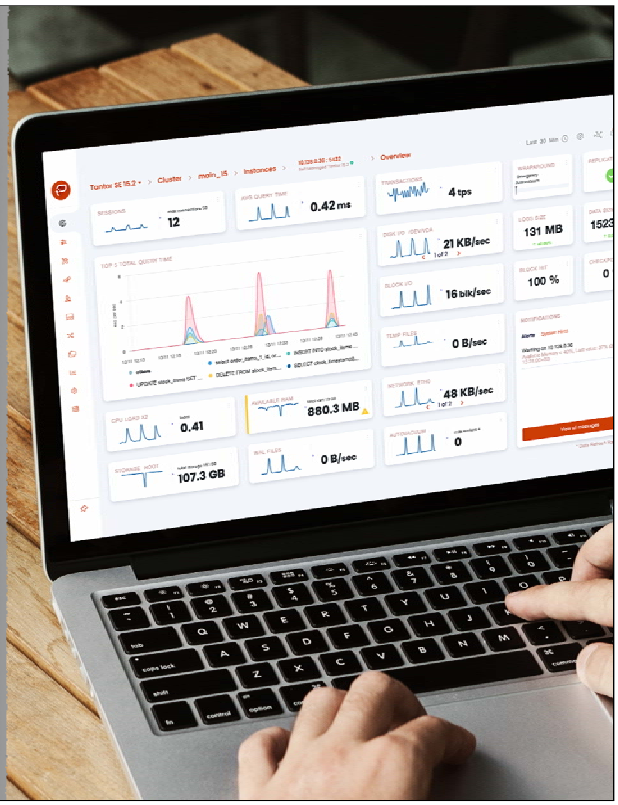
tantorlabs.ru
edu@tantorlabs.ru

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Выполнение запросов:

- Общие сведения
- Парсинг
- Планирование
- Выполнение
- Чтение плана запроса
- Оценка
- Статистика
- Демонстрация
- Практическая работа

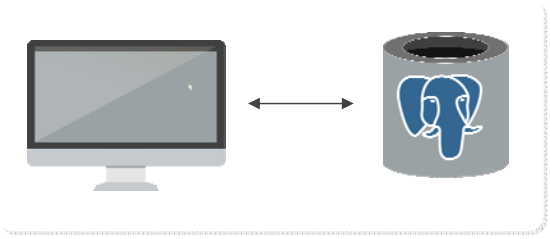


SQL — декларативный язык

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 следующий общий процесс:

- Парсинг (синтаксический и семантический разбор)
- Трансформация
- Планирование
- Выполнение запроса.



SQL (Structured Query Language) — это декларативный язык программирования, вам нужно описать, что вы хотите достичь, а не указывать, как это сделать шаг за шагом. В отличие от императивных языков программирования, где программа предоставляет последовательность операторов, которые выполняются по порядку, декларативные языки, такие как SQL, фокусируются на том, что должно быть достигнуто, оставляя оптимизацию и реализацию деталей исполнения системе управления базами данных.

В SQL, вы формулируете запросы, указывая, какие данные вы хотите получить или какие операции выполнить, но не говорите, каким образом система должна это сделать. Язык SQL более абстрактным и удобным для работы с данными, позволяя системе оптимизировать выполнение запросов и скрыть детали хранения данных.

Когда пользователь отправляет запрос SQL системе управления базами данных (СУБД) PostgreSQL, происходит следующий общий процесс:

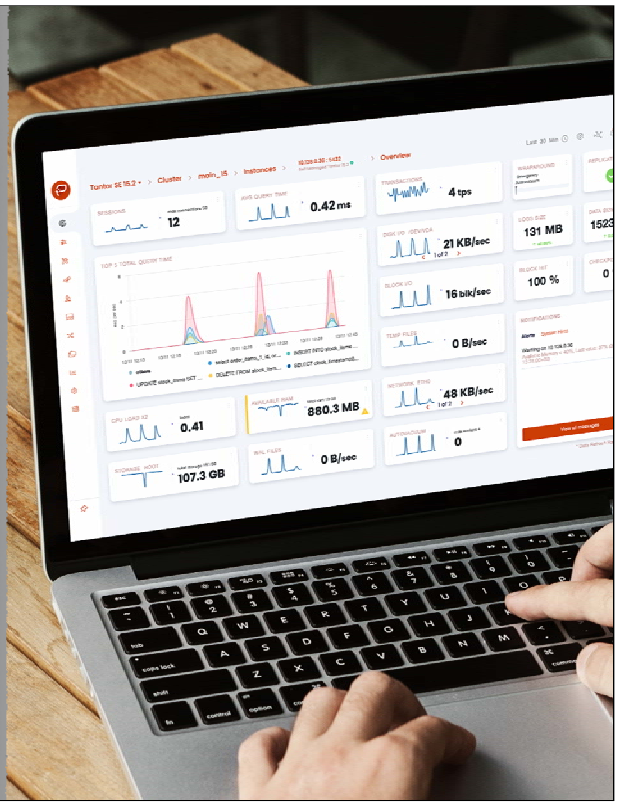
- **Парсинг** (синтаксический и семантический разбор): СУБД анализирует запрос пользователя, проверяет его синтаксис и осуществляет семантический анализ для понимания значения запроса.
- **Трансформация**: СУБД преобразует запрос во внутреннюю структуру данных, понятную для оптимизатора.
- **Планирование**: Оптимизатор создает оптимальный план выполнения запроса, решая, какие индексы использовать, как объединять таблицы и в каком порядке выполнять операции.
- **Выполнение запроса**: СУБД выполняет запрос, следуя оптимальному плану. Этот этап включает в себя чтение данных, их обработку и другие операции, необходимые для получения результата.

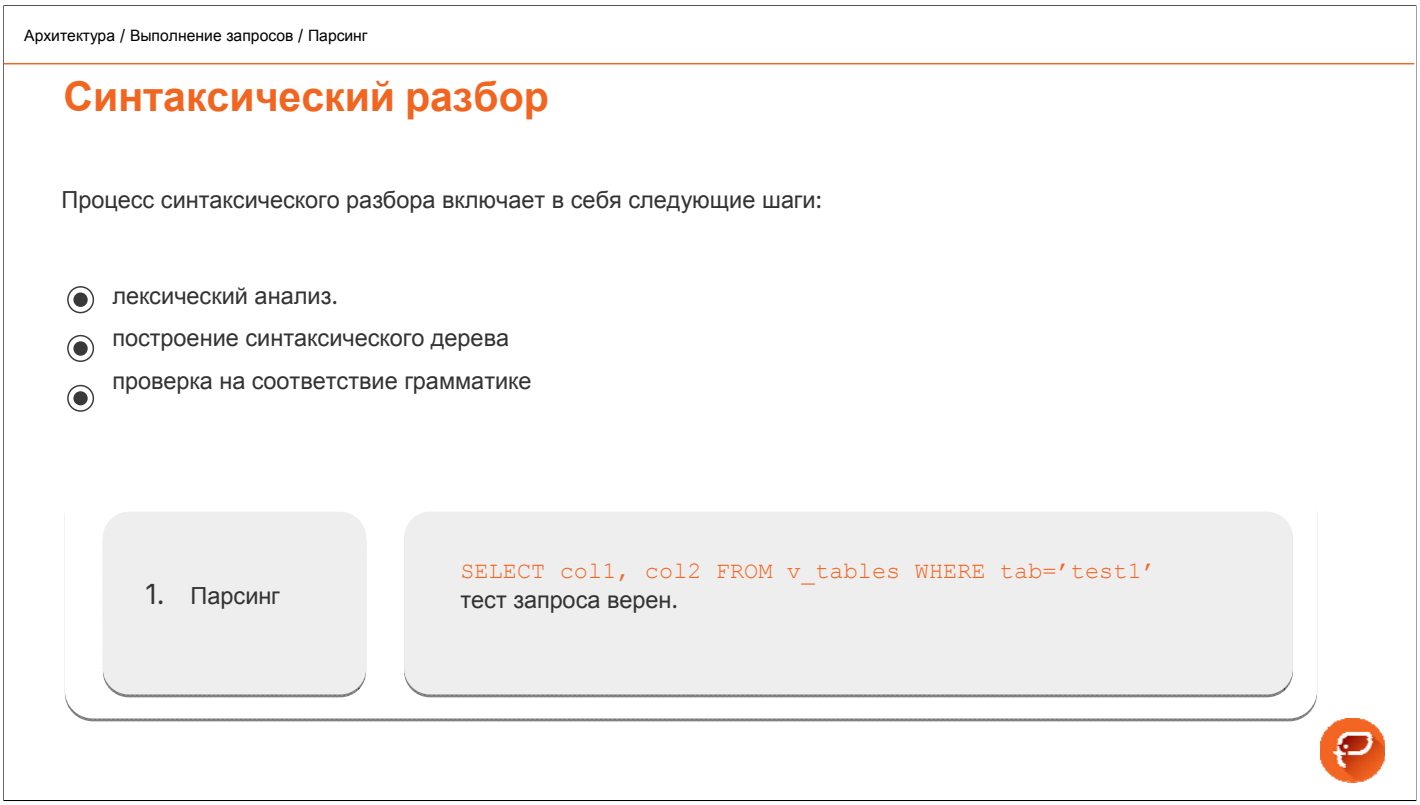
Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Выполнение запросов:

- Общие сведения
- Парсинг
- Планирование
- Выполнение
- Чтение плана запроса
- Оценка
- Статистика
- Демонстрация
- Практическая работа



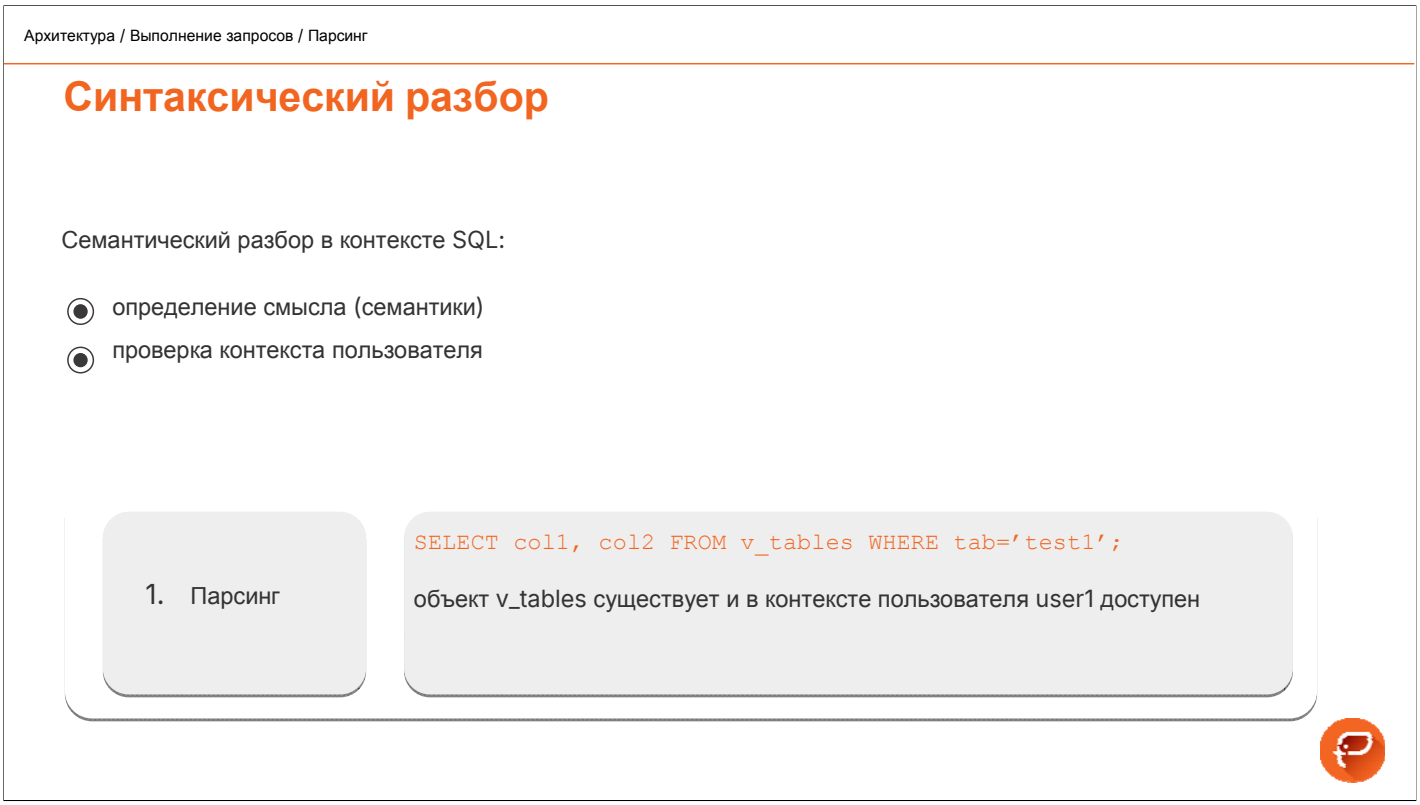


Синтаксический разбор, также известный как парсинг, представляет собой процесс анализа входящей последовательности символов или токенов для определения их структуры согласно определенным правилам грамматики языка. В контексте языков программирования или запросов, таких как SQL, синтаксический разбор используется для проверки того, соответствует ли введенный текст правильному синтаксису языка.

Процесс синтаксического разбора включает в себя следующие шаги:

- **Лексический анализ** (токенизация): Входящая строка разбивается на набор токенов, представляющих минимальные синтаксические единицы, такие как ключевые слова, операторы, идентификаторы и числа.
- **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.
- **Проверка на соответствие грамматике**: Синтаксический анализатор проверяет, соответствует ли построенное синтаксическое дерево правилам грамматики языка. Если нет, генерируется ошибка, указывающая на некорректность синтаксиса.

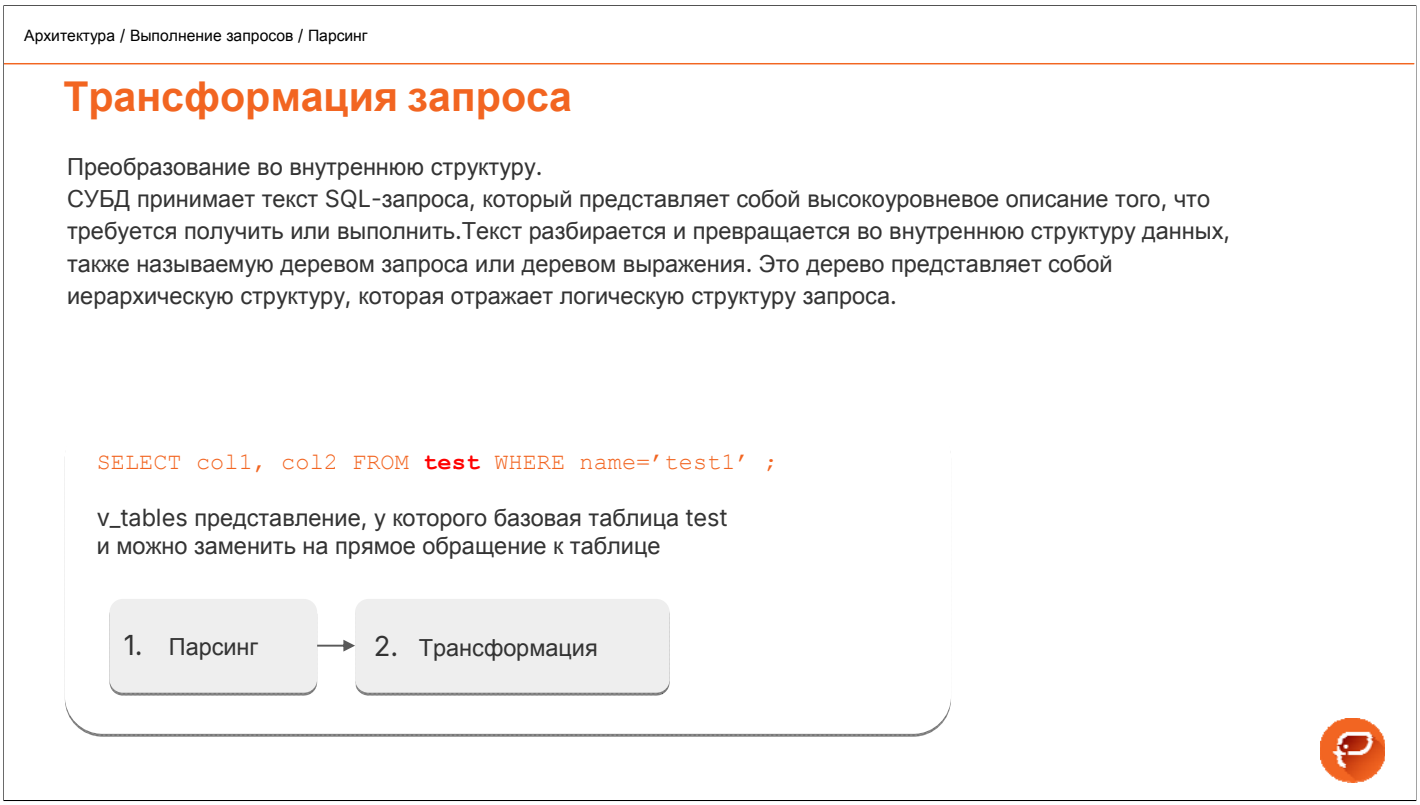
В случае SQL-запроса этот процесс гарантирует, что пользовательский запрос соответствует синтаксическим правилам SQL, что позволяет системе правильно интерпретировать и выполнить запрос.



Семантический разбор в контексте SQL:

- **Определение смысла (семантики)**: Этот этап парсинга SQL включает анализ смысла запроса, проверку существования таблиц, полей, согласованность типов данных и другие аспекты, связанные с корректным выполнением запроса.
- **Проверка контекста пользователя**: Учитывает контекст пользователя и прав доступа, проверяя, имеет ли пользователь необходимые разрешения для выполнения операций, указанных в запросе.

Таким образом, семантический разбор является частью более общего процесса парсинга, который включает в себя и синтаксический анализ (определение структуры запроса) и семантический анализ (определение смысла и контекста выполнения).



Этап трансформации в процессе выполнения SQL-запроса в СУБД представляет собой преобразование исходного запроса во внутреннюю структуру данных, которая легче поддается оптимизации и выполнению.

Давайте рассмотрим более подробно, что происходит на этом этапе. Преобразование во внутреннюю структуру — СУБД принимает текст SQL-запроса, который представляет собой высокоуровневое описание того, что требуется получить или выполнить.

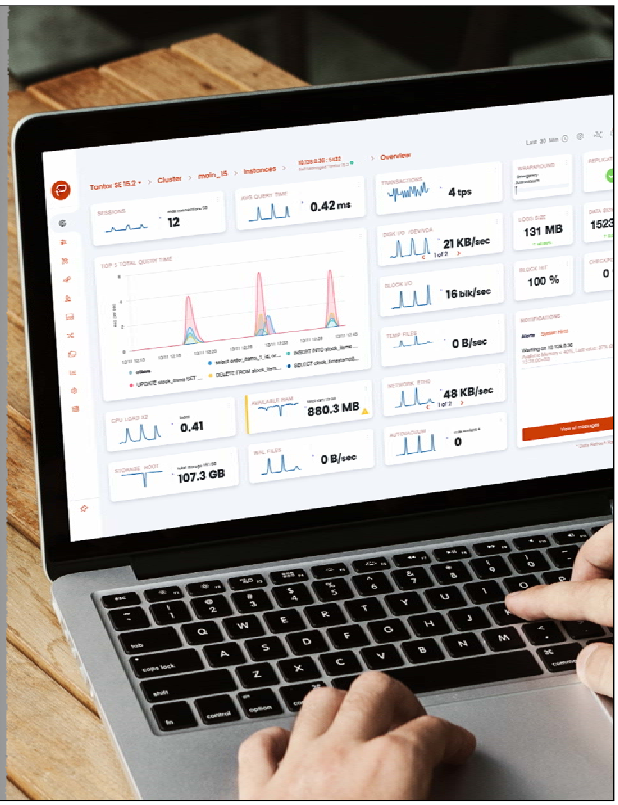
Текст разбирается и превращается во внутреннюю структуру данных, также называемую деревом запроса или деревом выражения. Это дерево представляет собой иерархическую структуру, которая отражает логическую структуру запроса.

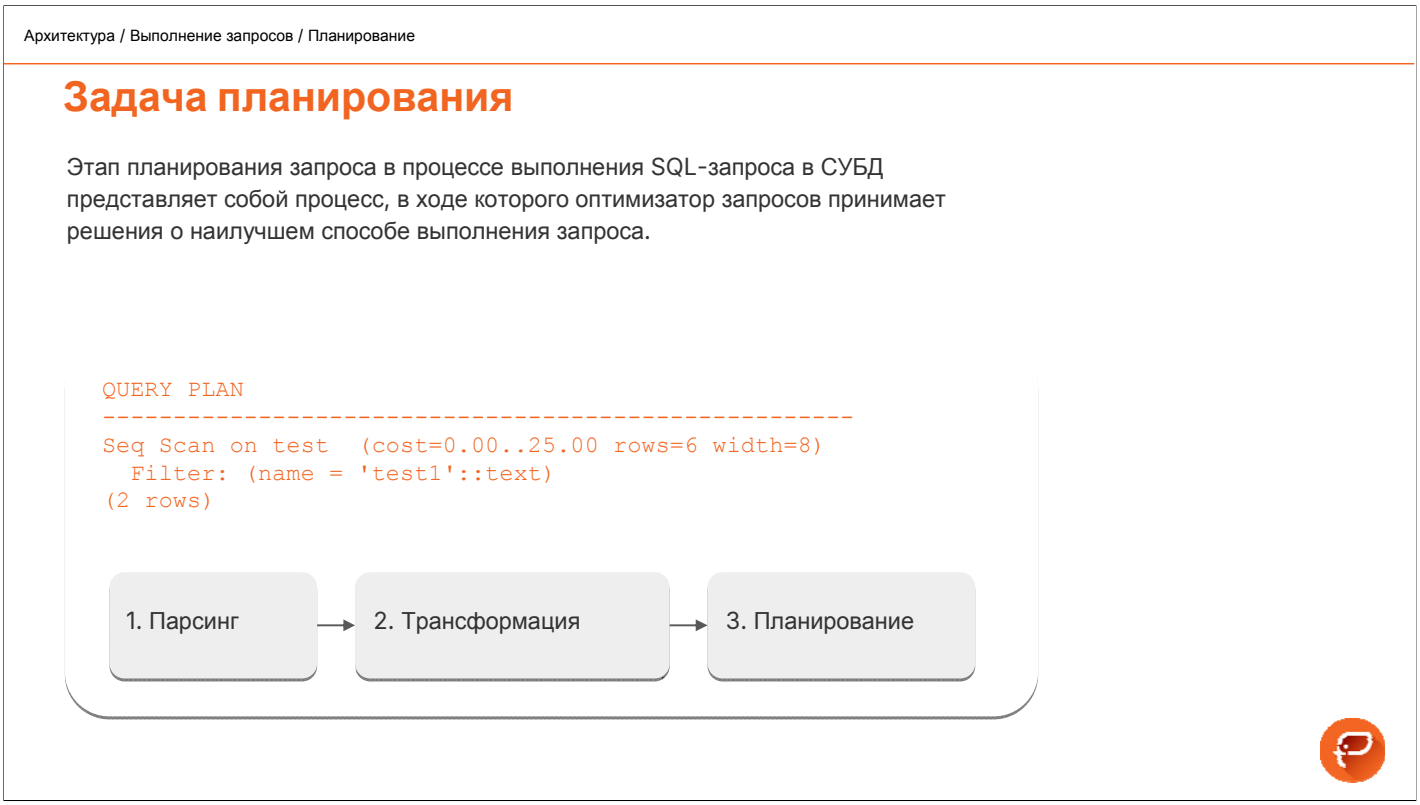
Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Выполнение запросов:

- Общие сведения
- Парсинг
- Планирование
- Выполнение
- Чтение плана запроса
- Оценка
- Статистика
- Демонстрация
- Практическая работа





Этап планирования запроса в процессе выполнения SQL-запроса в СУБД представляет собой процесс, в ходе которого оптимизатор запросов принимает решения о наилучшем способе выполнения запроса.

Давайте подробнее разберем этот этап:

- **Анализ структуры запроса**: Оптимизатор анализирует структуру запроса, который был преобразован на предыдущем этапе трансформации. Это включает в себя понимание, какие таблицы участвуют в запросе, какие условия и фильтры применяются, и какие операции необходимо выполнить.
- **Определение доступности индексов**: Оптимизатор решает, какие индексы могут быть использованы для ускорения выполнения запроса. Это включает в себя оценку затрат на использование индексов по сравнению с выполнением полного сканирования таблиц.
- **Выбор методов соединения таблиц**: В случае запросов, включающих соединение нескольких таблиц, оптимизатор решает, каким образом соединять данные. Это может включать в себя выбор между вложенным циклом, хеш-соединением или сортированным соединением в зависимости от структуры данных и объема информации.
- **Решение о порядке выполнения операций**: Оптимизатор определяет оптимальный порядок выполнения операций в запросе. Это включает в себя решение о последовательности операций фильтрации, сортировки и соединения, чтобы минимизировать затраты на доступ к данным и обработку.
- **Оптимизация вычислений и предикатов**: Оптимизатор рассматривает выражения и условия в запросе, оптимизируя их вычисления, чтобы уменьшить вычислительную сложность и повысить производительность.
- **Сбор статистики**: Перед принятием окончательного решения об оптимальном плане выполнения, оптимизатор может собирать статистику о данных, такую как количество строк в таблицах, распределение значений и другие метрики, которые помогут принять более информированное решение.

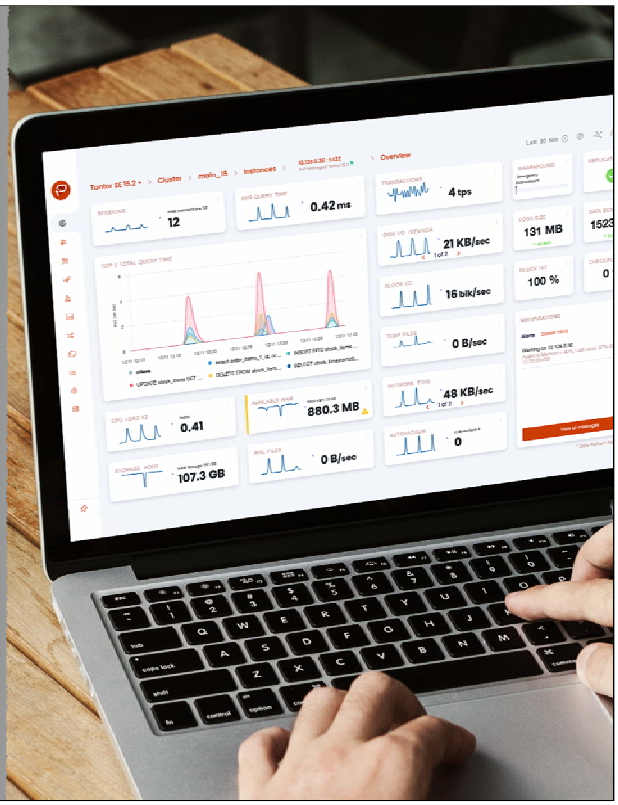
В результате этого этапа оптимизатор создает оптимальный план выполнения запроса, который затем используется для фактического выполнения запроса в базе данных.

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Выполнение запросов:

- Общие сведения
- Парсинг
- Планирование
- Выполнение
- Чтение плана запроса
- Оценка
- Статистика
- Демонстрация
- Практическая работа



Этап выполнения запроса — последний этап в обработке SQL-запроса в СУБД, где фактически выполняются операции, описанные в оптимальном плане, созданном на предыдущем этапе планирования.

Давайте более подробно рассмотрим этот этап:

- **Чтение данных**: В соответствии с оптимальным планом выполнения, СУБД начинает читать данные из соответствующих таблиц базы данных. Это может включать в себя использование индексов, сканирование полных таблиц или другие методы доступа к данным.
- **Обработка данных**: Полученные данные подвергаются обработке в соответствии с требованиями запроса. Это включает в себя выполнение операций фильтрации, сортировки, группировки, вычислений и других манипуляций с данными, описанных в запросе.
- **Выполнение соединений**: Если запрос включает в себя соединение нескольких таблиц, СУБД выполняет соединение данных в соответствии с определенным методом, который был выбран на этапе планирования.
- **Обработка групповых функций**: Если запрос содержит групповые функции, такие как `COUNT`, `SUM`, `AVG` и др., СУБД вычисляет эти значения для соответствующих групп данных в результате запроса.
- **Формирование результатов**: После завершения обработки данных, СУБД формирует результат запроса. Это может быть представлено в виде таблицы результатов или другого формата, который соответствует требованиям запроса.
- **Возврат результатов**: Сформированный результат отправляется обратно пользователю или приложению, и запрос считается успешно выполненным.
- **Освобождение ресурсов**: По завершении выполнения запроса, СУБД освобождает использованные ресурсы, такие как выделенная память или временные таблицы, чтобы поддерживать эффективность работы.

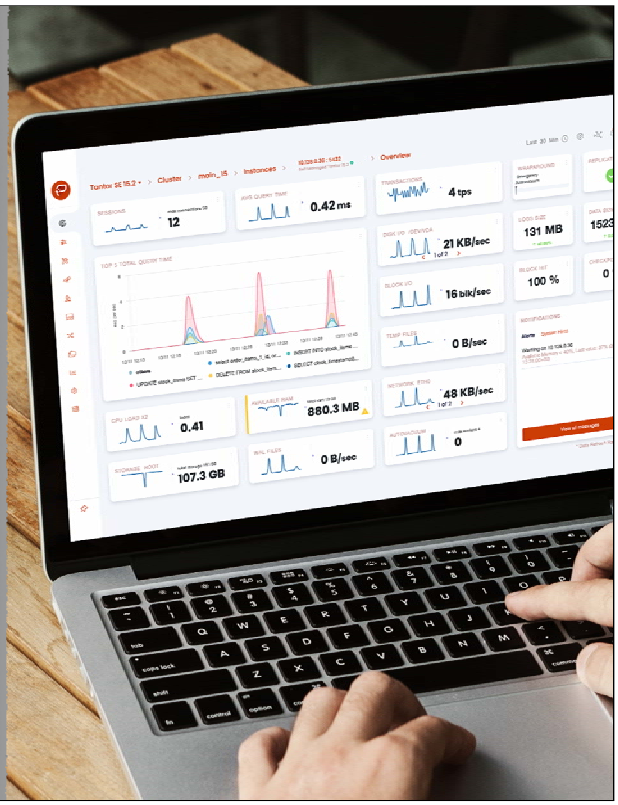
В итоге, на этапе выполнения запроса СУБД превращает абстрактный запрос на языке SQL в конкретные действия с данными в базе данных, в соответствии с планом, разработанным на более ранних этапах обработки.

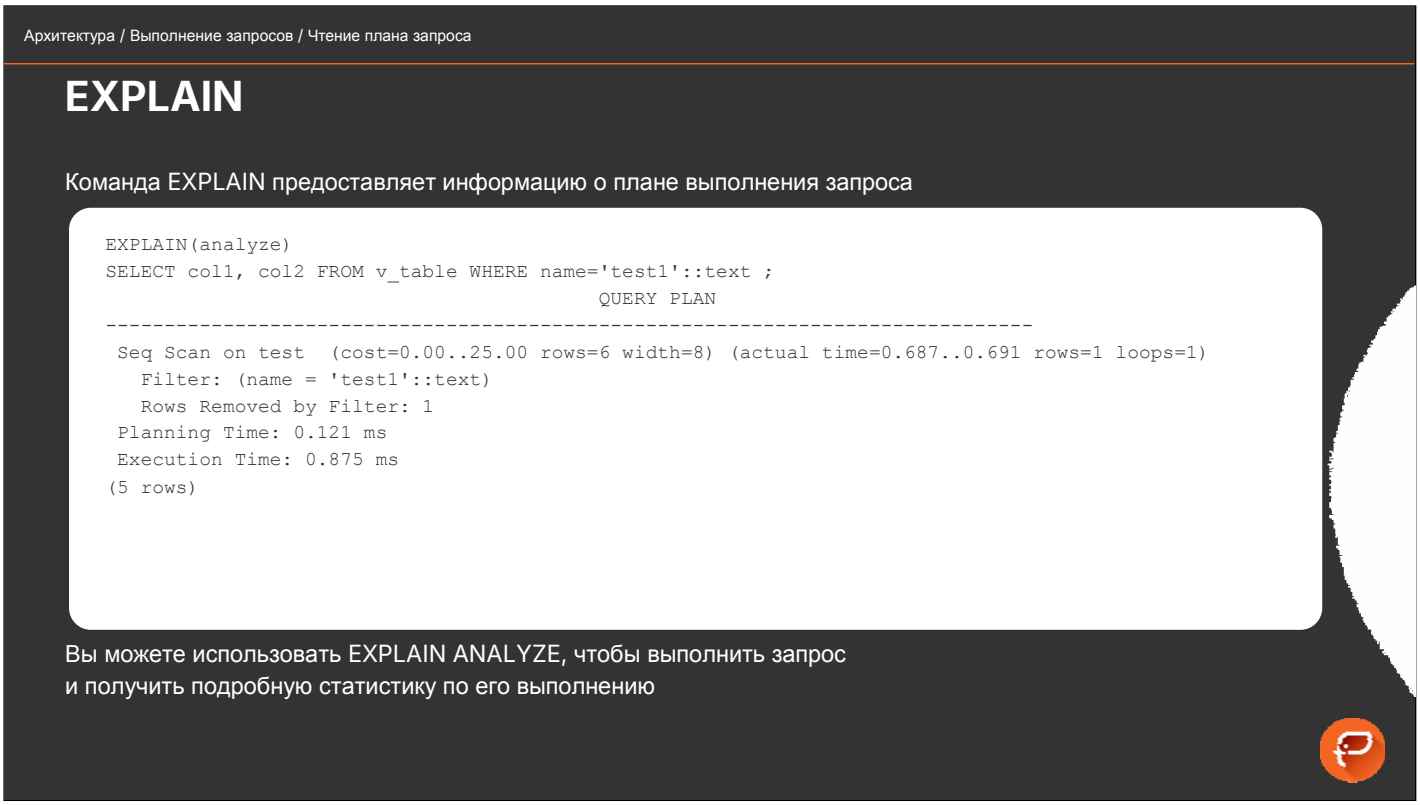
Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Выполнение запросов:

- Общие сведения
- Парсинг
- Планирование
- Выполнение
- Чтение плана запроса
- Оценка
- Статистика
- Демонстрация
- Практическая работа





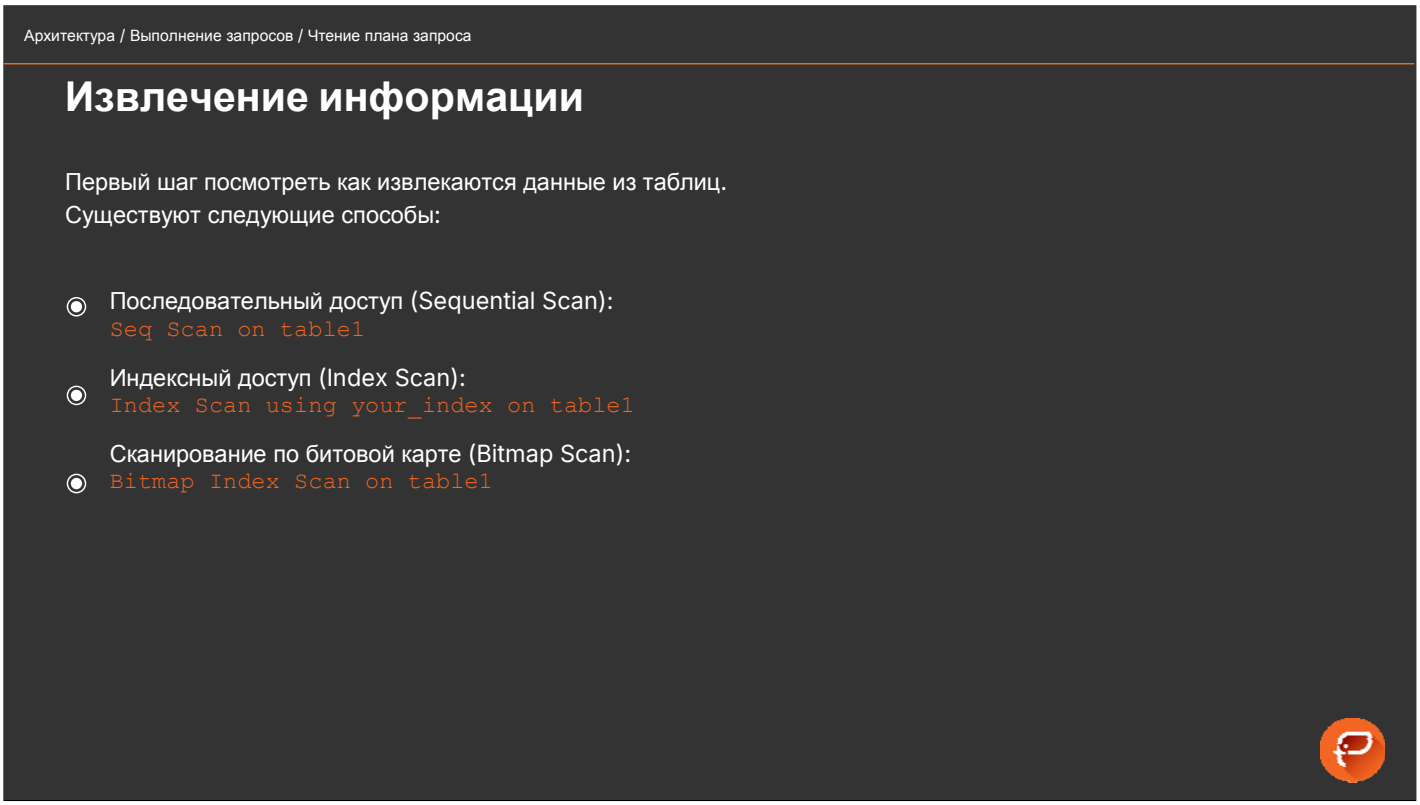
В СУБД, таких как PostgreSQL, команда `EXPLAIN` предоставляет информацию о плане выполнения запроса. Это мощный инструмент для анализа и оптимизации запросов. Команда `EXPLAIN` позволяет получить подробную информацию о том, как база данных собирается выполнить запрос.

Пример использования команды `EXPLAIN`:

```
EXPLAIN (analyze)
     SELECT col1, col2 FROM v_table WHERE name='test1'::text ;
                                                QUERY PLAN
     ----------------------------------------------------------------------
------------------------------
      Seq Scan on test   (cost=0.00..25.00 rows=6 width=8) (actual
time=0.687..0.691 rows=1 loops=1)
        Filter: (name = 'test1'::text)
        Rows Removed by Filter: 1
      Planning Time: 0.121 ms
      Execution Time: 0.875 ms
     (5 rows)
```

После выполнения этой команды, СУБД вернет план выполнения запроса, который может включать в себя информацию о порядке выполнения операций, использовании индексов, объединениях и других деталях. Это позволяет анализировать, как СУБД собирается извлекать данные и оптимизировать запрос при необходимости.

Примечание: Важно помнить, что команда `EXPLAIN` не фактически выполняет запрос, а лишь анализирует план выполнения. Вы можете использовать `EXPLAIN ANALYZE`, чтобы выполнить запрос и получить подробную статистику по его выполнению. Однако, выполнение `EXPLAIN ANALYZE` может занять больше времени, чем просто `EXPLAIN`, так как она фактически выполняет запрос.



В PostgreSQL существуют три основных способа доступа к данным (в контексте плана выполнения запроса):

- **Последовательный доступ** (`Sequential Scan`): Этот метод представляет собой простой последовательный перебор всех строк в таблице для поиска тех, которые соответствуют условиям запроса. Эффективен для запросов, которые должны вернуть большой объем данных, но может быть медленным для больших таблиц.

```
Seq Scan on your_table
```

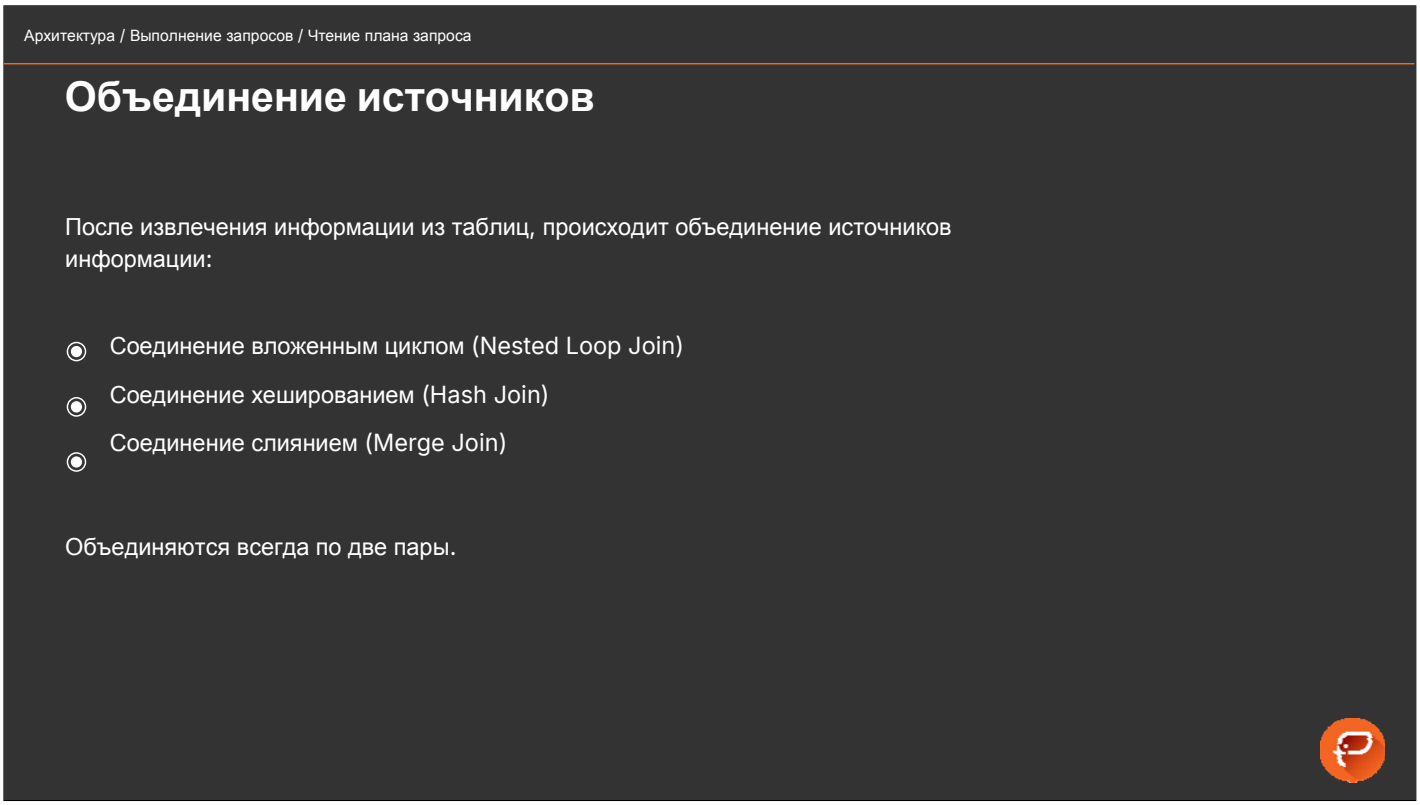
- **Индексный доступ** (`Index Scan`): Этот метод использует индексы для быстрого поиска и извлечения данных, соответствующих условиям запроса. Индексный доступ может быть более эффективным для запросов, которые фильтруют небольшой поднабор данных.

```
Index Scan using your_index on your_table
```

- **Сканирование по битовой карте** (`Bitmap Scan`): Этот метод используется в случае, когда несколько индексов могут быть использованы для выполнения запроса, и результаты объединяются с использованием битовой карты. Это может быть эффективным, когда несколько условий запроса могут быть улучшены с использованием разных индексов.

```
Bitmap Index Scan on your_table
```

Каждый из этих способов имеет свои преимущества и недостатки, и оптимизатор запросов PostgreSQL выберет подходящий метод в зависимости от структуры таблицы, условий запроса и других факторов.



После извлечения информации из таблиц, происходит объединение источников информации.

В PostgreSQL существуют три основных метода соединения источников информации в плане выполнения запроса:

- **Соединение вложенным циклом** (`Nested Loop Join`): Этот метод используется, когда одна таблица (внутренняя) последовательно просматривается для каждой строки из другой таблицы (внешней). Он эффективен для малых таблиц или когда нет подходящего индекса для соединения.

  `Nested Loop Join`

- **Соединение хешированием** (`Hash Join`): В этом методе обе таблицы подвергаются хешированию по ключевым столбцам, и затем происходит сравнение хешей. Это эффективно при больших объемах данных и при отсутствии индекса для соединения.

  `Hash Join`

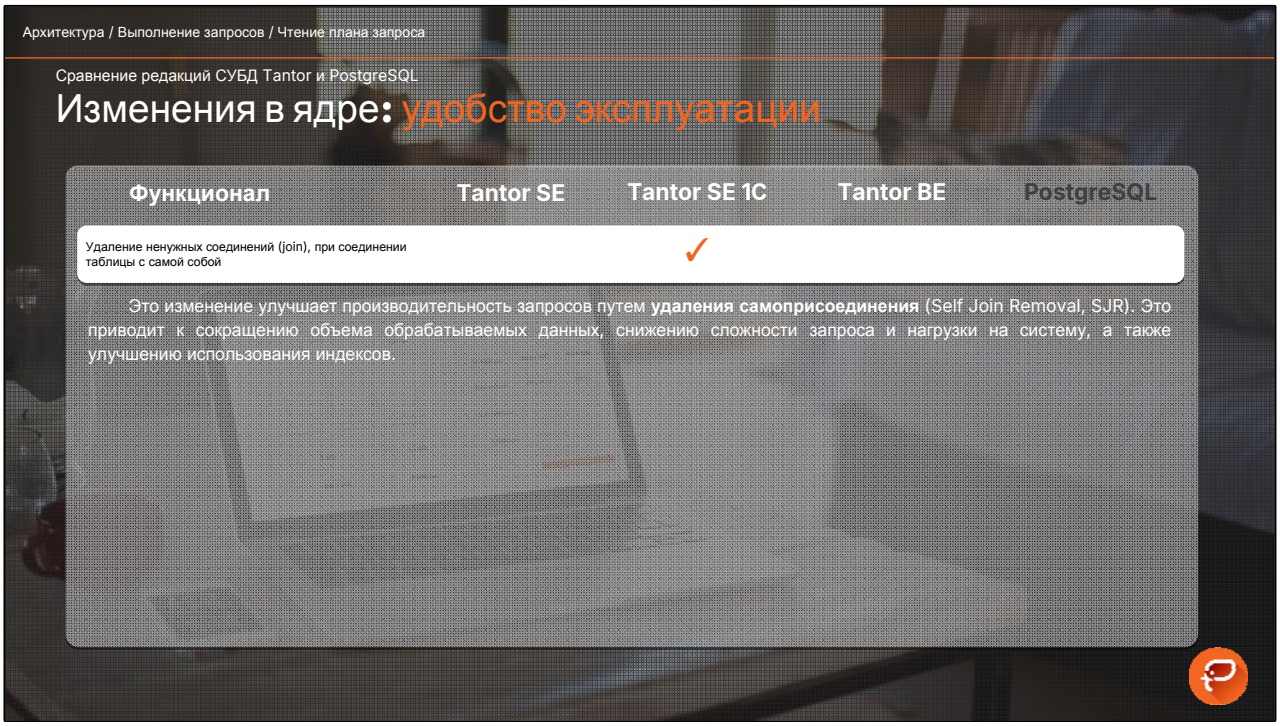
- **Соединение слиянием** (`Merge Join`): Этот метод предполагает, что обе таблицы уже отсортированы по ключу соединения, и происходит слияние двух отсортированных потоков данных. Он эффективен, когда обе таблицы упорядочены и когда данных достаточно для оправдания сортировки.

  `Merge Join`

Выбор конкретного метода зависит от размера таблиц, наличия индексов, структуры данных и других факторов.

Оптимизатор запросов PostgreSQL будет выбирать наиболее подходящий метод на основе статистики и текущих условий.

Объединяются всегда по две пары.



Для версии SE 1C доступно удаление ненужных соединений (join) при соединении таблицы с самой собой.

Данное изменение реализует удаление самоприсоединения (`Self Join Removal, SJR`), что в некоторых сценариях улучшает производительность запросов по нескольким причинам:

1. Уменьшение объема обрабатываемых данных. Самоприсоединение, как правило, увеличивает количество данных, которые необходимо обрабатывать при выполнении запроса. Заменой самоприсоединения на сканирование таблицы мы можем сократить объем обрабатываемых данных, что ускоряет выполнение запроса.
2. Уменьшение сложности запроса. Присоединение таблицы к себе может увеличить сложность запроса, что замедляет его выполнение. Удаление самоприсоединения приводит к упрощению запроса, что помогает ускорить его выполнение.
3. Уменьшение нагрузки на систему. Присоединение таблицы к себе может потребовать больше системных ресурсов, таких как процессорное время и память. Удаление самоприсоединения может помочь снизить нагрузку на систему, особенно при обработке больших объемов данных.
4. Улучшение индексного покрытия. Замена самоприсоединения на сканирование может улучшить использование индексов, что увеличивает производительность запроса.

Сортировка

В плане выполнения запроса в PostgreSQL могут применяться различные методы сортировки данных:

- Сортировка в памяти (In-Memory Sort)
  Sort Method: quicksort Memory: 25kB
- Внешняя сортировка (External Sort)
  Sort Method: external merge Disk: 2304kB
  и другие методы (top-N heapsort....).

Сортировка данных предоставляет контроль над порядком, в котором данные представляются или возвращаются в результате запроса, что может быть важным для корректного функционирования приложения или анализа данных.

В плане выполнения запроса в PostgreSQL могут применяться различные методы сортировки данных. Некоторые из них включают:

- **Сортировка в памяти (In-Memory Sort):** Этот метод применяется, когда данные могут быть помещены и отсортированы в оперативной памяти без необходимости дополнительных действий. Это может быть эффективно для небольших объемов данных.

```
Sort Method: quicksort Memory: 25kB
```

- **Внешняя сортировка (External Sort):** Если объем данных для сортировки слишком велик для помещения в оперативной памяти, применяется внешняя сортировка. Данные разбиваются на более мелкие части, которые сортируются отдельно, а затем объединяются.

```
Sort Method: external merge  Disk: 2304kB
```

Сортировка данных предоставляет контроль над порядком, в котором данные представляются или возвращаются в результате запроса, что может быть важным для корректного функционирования приложения или анализа данных.

Сортировка данных в базах данных выполняется по нескольким причинам, которые могут варьироваться в зависимости от конкретного контекста запроса или требований приложения:

- **Оператор `ORDER BY` в запросах**: Одним из основных случаев, когда требуется сортировка, является использование оператора `ORDER BY` в SQL-запросах. Оператор `ORDER BY` позволяет упорядочивать результаты запроса по одному или нескольким столбцам в определенном порядке (возрастающем или убывающем). Это полезно, когда требуется получить результаты запроса в определенной последовательности.

```
SELECT column1, column2 FROM your_table ORDER BY column1 ASC;
```

- **Обеспечение уникальности результатов**: Сортировка может использоваться для обеспечения уникальности результатов запроса. Порядок сортировки может быть также важен при определении уникальных значений.

```
SELECT DISTINCT column1 FROM your_table ORDER BY column1;
```

- **Слияние данных из разных источников**: В некоторых случаях сортировка используется при соединении данных из разных таблиц или подзапросов. Это может помочь оптимизировать процесс соединения и ускорить выполнение запроса.
- **Оптимизация использования индексов**: Сортировка может также играть роль в оптимизации использования индексов. Например, если запрос использует индекс, и данные в индексе уже отсортированы в нужном порядке, это может повысить производительность запроса.
- **Оконные функции (Window Functions)**: Некоторые функции в PostgreSQL, такие как оконные функции, могут требовать упорядоченности данных. Например, функции, связанные с расчетом скользящего среднего или ранжированием, могут использовать сортировку данных.

«Sort Method: top-N heapsort» в плане выполнения запроса указывает на то, что используется алгоритм сортировки «top-N heapsort» для получения первых N строк результата запроса.

Группировка

Узлы группировки в плане выполнения запроса могут включать:

GroupAggregate:
Этот узел отображает использование группировки и агрегации в запросе.
`GroupAggregate (cost=1000.00..1200.00 rows=100 width=12)`

HashAggregate:
Этот узел отображает использование хеширования при выполнении группировки и агрегации.
`HashAggregate (cost=1000.00..1200.00 rows=100 width=12) Hash Key: department`

Эти узлы отражают действия оптимизатора запросов для реализации группировки и агрегации данных в соответствии с запросом пользователя.

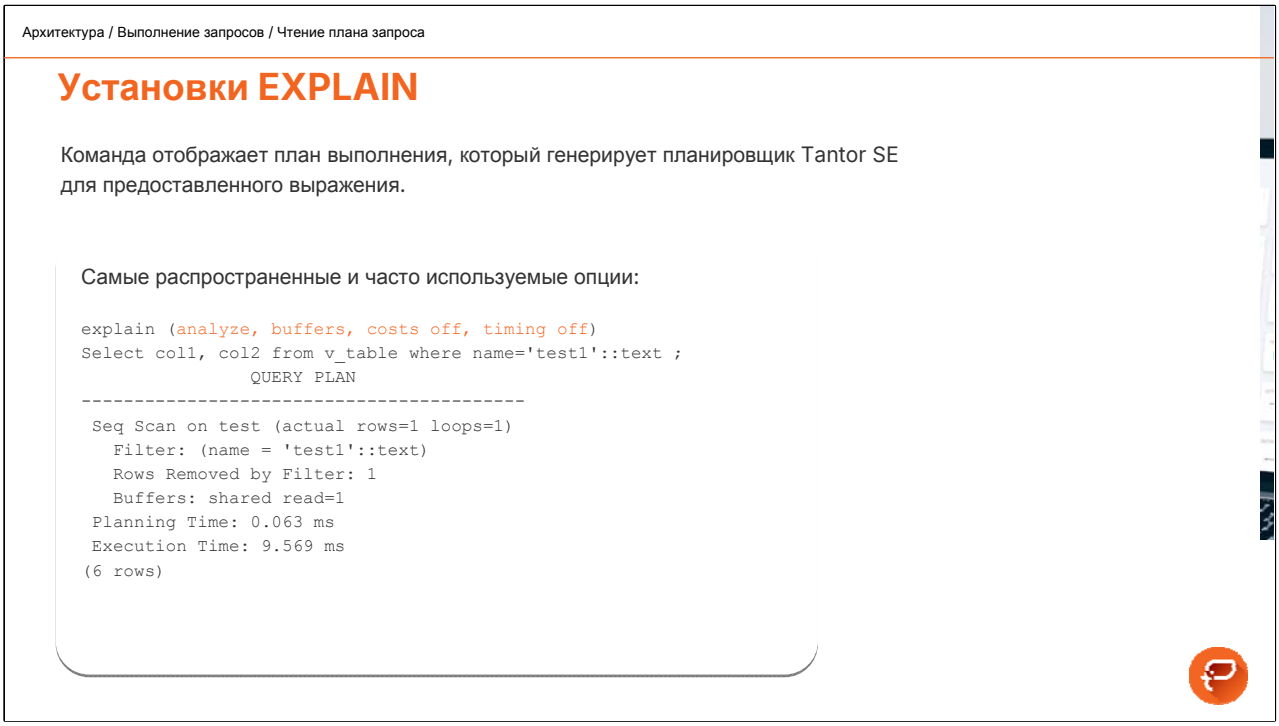
GroupAggregate: Этот узел отображает использование группировки и агрегации в запросе.

```
GroupAggregate (cost=1000.00..1200.00 rows=100 width=12)
```

HashAggregate: Этот узел отображает использование хеширования при выполнении группировки и агрегации.

```
HashAggregate (cost=1000.00..1200.00 rows=100 width=12)
Hash Key: department
```

Эти узлы отражают действия оптимизатора запросов для реализации группировки и агрегации данных в соответствии с запросом пользователя.



Команда отображает план выполнения, который генерирует планировщик Tantor SE для предоставленного выражения.

Самые распространенные и часто используемые опции для команды `EXPLAIN` в PostgreSQL включают:

- **`ANALYZE`:** Опция `ANALYZE` часто применяется для фактического выполнения запроса и включения статистики выполнения в вывод.
- **`VERBOSE` (или `VERBOSE ANALYZE`):** Опция `VERBOSE` добавляет более подробные сведения о каждой операции в плане выполнения, включая статистику и оценку стоимости. Она часто используется для получения дополнительной информации.
- **`FORMAT` (или `FORMAT JSON`):** Опция `FORMAT` часто применяется, чтобы выбрать формат вывода, например, текстовый (по умолчанию) или `JSON`. `FORMAT JSON` используется для получения вывода в формате `JSON`, который может быть удобен для автоматизированного анализа.
- **`COSTS` (или `COSTS ANALYZE`):** Опция `COSTS` широко используется для включения стоимости каждой операции в плане выполнения. Это полезно для анализа и оптимизации запросов.
- **`BUFFERS` (или `BUFFERS ANALYZE`):** Опция `BUFFERS` используется для включения информации о буферах, что позволяет анализировать использование кэша и операций ввода-вывода.

Эти опции предоставляют дополнительную информацию, которая может быть полезна при понимании и оптимизации выполнения запросов в PostgreSQL.

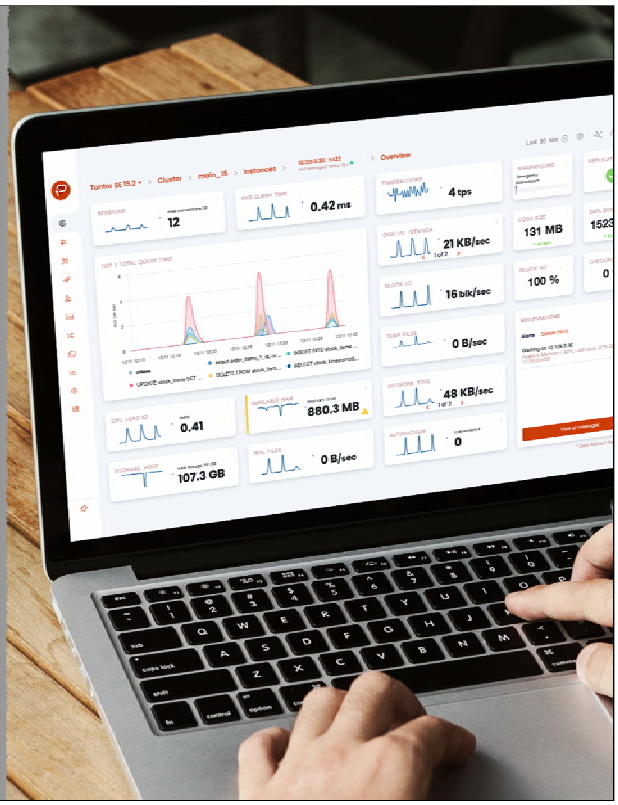
https://docs.tantorlabs.ru/tdb/ru/15_4/se/sql-explain.html#explain

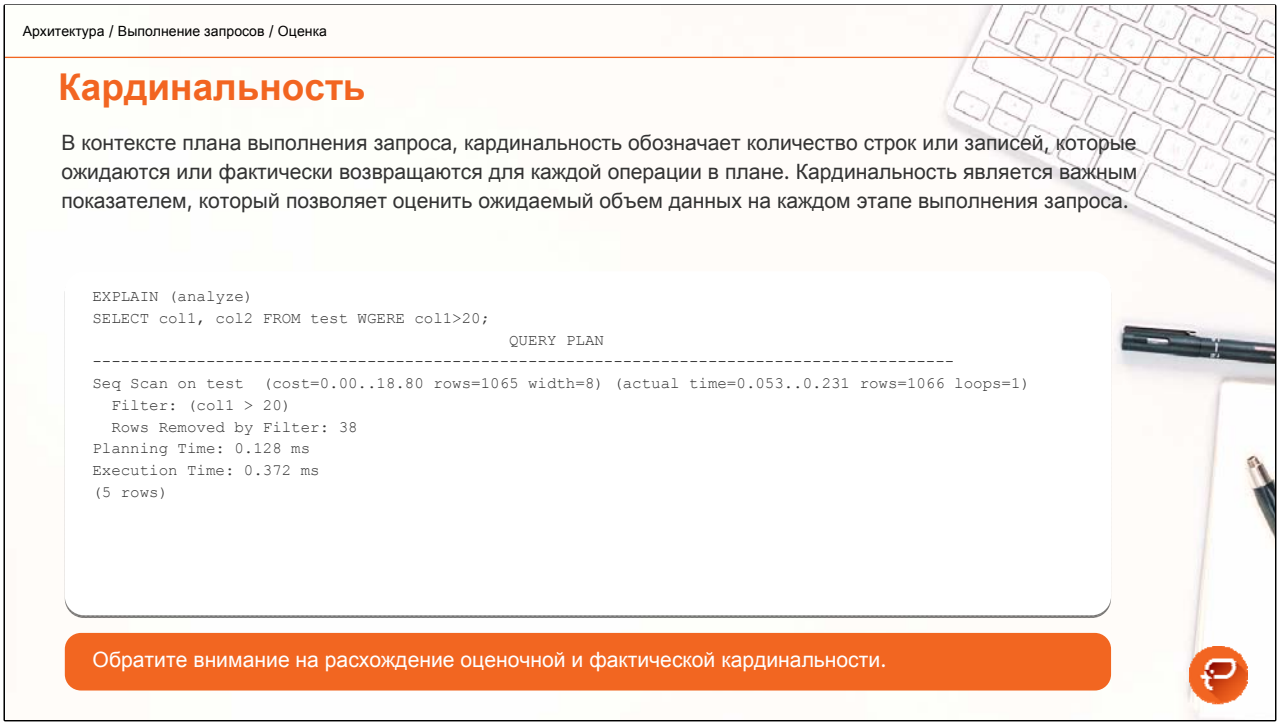
Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Выполнение запросов:

- Общие сведения
- Парсинг
- Планирование
- Выполнение
- Чтение плана запроса
- Оценка
- Статистика
- Демонстрация
- Практическая работа





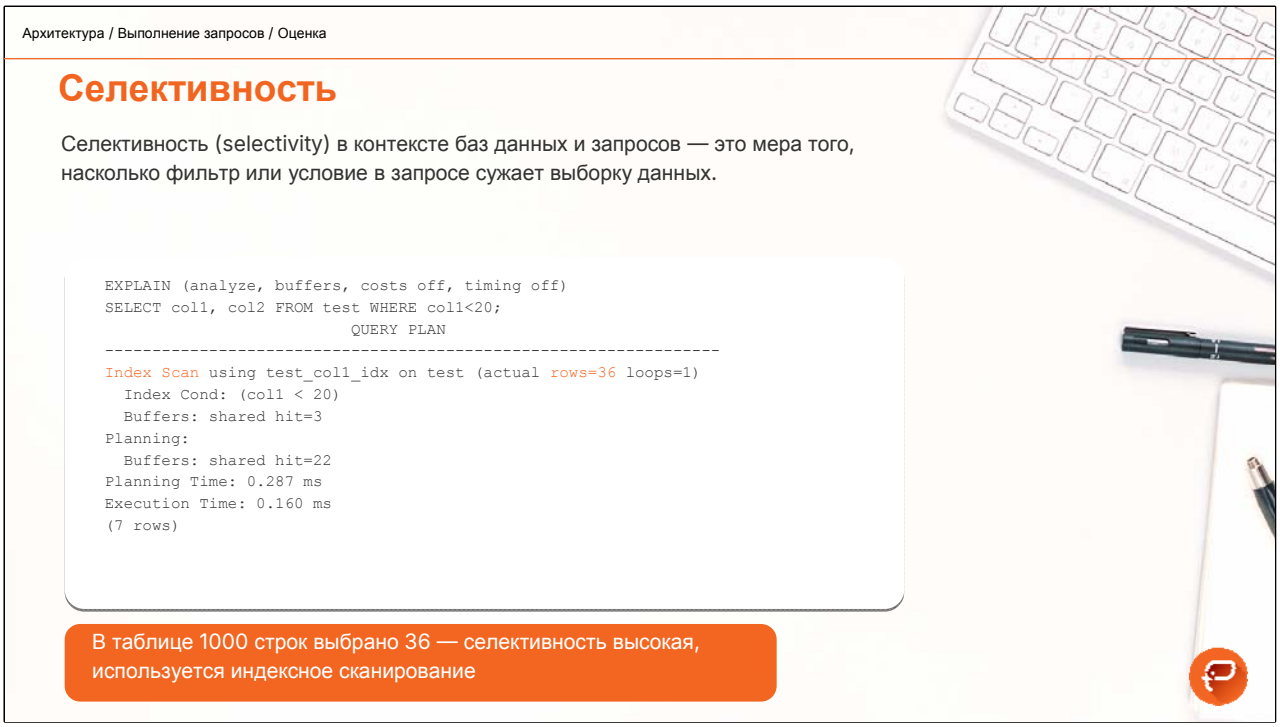
В контексте плана выполнения запроса, кардинальность обозначает количество строк или записей, которые ожидаются или фактически возвращаются для каждой операции в плане. Кардинальность является важным показателем, который позволяет оценить ожидаемый объем данных на каждом этапе выполнения запроса.

Типичные значения кардинальности включают:

- **Оценочная кардинальность (Estimated Cardinality)**: Оценочная кардинальность — это предварительное ожидание оптимизатора запросов по количеству возвращаемых строк для каждой операции в плане. Она часто видна в оценках строк (`rows`) в выводе команды `EXPLAIN`. Например, `rows=100` может означать, что ожидается 100 строк.
- **Фактическая кардинальность (Actual Cardinality)**: Фактическая кардинальность представляет собой реальное количество строк, возвращенных каждой операцией в плане после выполнения запроса. Она может быть видна в статистике выполнения, когда используется опция `ANALYZE` в команде `EXPLAIN`.

Кардинальность в плане выполнения запроса важна для анализа и оптимизации запросов.

Например, если оценочная кардинальность сильно отличается от фактической, это может быть признаком необходимости переоценки статистики или реорганизации запроса для улучшения производительности.



Селективность (selectivity) в контексте баз данных и запросов — это мера того, насколько фильтр или условие в запросе сужает выборку данных. Она показывает, какую часть данных в таблице можно ожидать после применения определенного условия.

Чем выше селективность, тем более эффективен индекс или другие методы доступа к данным, так как фильтр лучше сужает выборку. С другой стороны, низкая селективность может сделать использование индекса менее эффективным, так как большая часть данных удовлетворяет условию, и сканирование всей таблицы может быть более эффективным.

В контексте плана выполнения запроса в PostgreSQL, селективность может быть видна в статистике оценки (estimates) для каждой операции в плане. Например, оценка числа возвращаемых строк и оценка фильтрации по условиям запроса. Эти оценки могут помочь оптимизатору запросов выбирать наилучшие планы выполнения в зависимости от селективности условий.

Селективность показывает, какую часть данных в таблице можно ожидать после применения определенного условия.

Селективность измеряется в процентах строк, возвращаемых от общего количества строк источника данных.

К примеру, в таблице 100 строк возвращается 5 значит мы говорим о высокой селективности, 5%, если же селективность приближается к 100% это низкая селективность.

Чем выше селективность тем выше вероятность использования индекса.

Стоимость запроса

Оценка стоимости используется оптимизатором запросов для выбора наилучшего плана выполнения.

```
EXPLAIN (analyze)
SELECT col1, col2 FROM test WHERE col1>20;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..18.80 rows=1065 width=8) (actual time=0.053..0.231 rows=1066
loops=1)
   Filter: (col1 > 20)
   Rows Removed by Filter: 38
 Planning Time: 0.128 ms
 Execution Time: 0.372 ms
(5 rows)
```

В этом примере, cost=0.00..18.80 — это оценка стоимости для операции Seq Scan.

В контексте плана выполнения запроса в базе данных, стоимость выполнения запроса — числовая оценка ресурсов, необходимых для выполнения операций в запросе. Оценка стоимости используется оптимизатором запросов для выбора наилучшего плана выполнения.

Оценка стоимости может включать в себя различные параметры, такие как количество операций ввода-вывода, количество строк, использование индексов и другие метрики, отражающие затраты на выполнение запроса.

В примере, `cost=0.00..18.00` — это оценка стоимости для операции Seq Scan. Общая оценка стоимости для всего плана выполнения может быть видна в корневой строке вывода команды `EXPLAIN`.

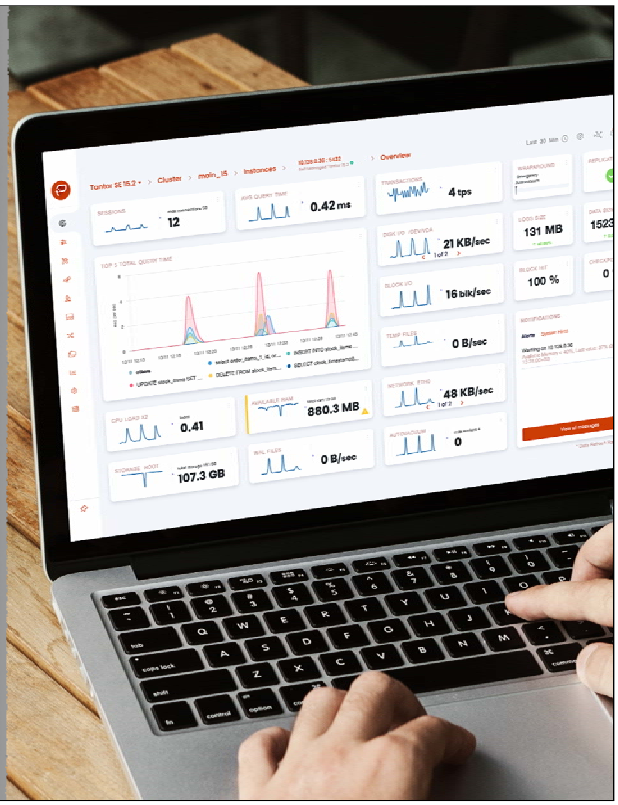
Оптимизатор запросов стремится выбрать такой план выполнения, который имеет наименьшую оценку стоимости, таким образом, обеспечивая наилучшую производительность выполнения запроса. Оценка стоимости является важным инструментом для принятия решений о том, как оптимизировать запросы и какие индексы или другие методы доступа к данным использовать.

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

Выполнение запросов:

- Общие сведения
- Парсинг
- Планирование
- Выполнение
- Чтение плана запроса
- Оценка
- Статистика
- Демонстрация
- Практическая работа



Источники статистики

Оптимизатор запросов использует статистику для принятия решений о том, как создавать план выполнения запроса.
Источники статистики pg_class и pg_index, pg_statistic (pg_stats), всевозможные иные представления.

Пример статистики таблицы:

```
postgres=# SELECT relpages, reltuples FROM pg_class where relname='test';
relpages | reltuples
----------+-----------
       5 |      1104
(1 row)
```

в таблице тест используется 5 страниц и 1104 актуальные строки.

Статистика обновляется автоматически во время выполнения операций `ANALYZE` или автоматически в фоновом режиме.

Оптимизатор запросов использует статистику для принятия решений о том, как создавать план выполнения запроса. В PostgreSQL статистика собирается и хранится для таблиц и индексов, и она включает в себя информацию о распределении данных, числе уникальных значений, размере таблиц и индексов, а также другие метрики.

Основные источники статистики, которые оптимизатор запросов может использовать:

- **Статистика PostgreSQL**: PostgreSQL автоматически собирает статистику о таблицах и индексах, чтобы помочь оптимизатору запросов принимать более информированные решения. Эта статистика включает в себя оценки числа строк, распределение значений в столбцах, информацию об индексах и другие характеристики. Статистика обновляется автоматически во время выполнения операций `ANALYZE` или автоматически в фоновом режиме.
- **`pg_statistic`**: Таблица `pg_statistic` содержит дополнительные статистические данные о значениях столбцов, такие как минимальные и максимальные значения, среднее, стандартное отклонение и др. Эта информация может быть использована для более точной оценки стоимости выполнения запроса.
- **`pg_class` и `pg_index`**: Таблицы `pg_class` и `pg_index` содержат информацию о размерах таблиц и индексов, а также о числе строк в таблицах и структуре индексов. Эта информация может быть использована оптимизатором для выбора метода доступа к данным, например, сканирование таблицы или использование индекса.

pg_statistic (pg_stats)

Таблица `pg_statistic` содержит статистическую информацию о данных в базе данных. Эти записи формируются в результате выполнения операции `ANALYZE` и служат для оптимизации запросов планировщиком. Важно отметить, что статистическая информация всегда представляет собой приблизительные значения, даже при актуальном их обновлении.

В таблице `pg_statistic` содержатся следующие статистические данные для каждого столбца. К примеру посмотрим какое количество NULL есть в третьем столбце таблицы.

```
SELECT stanullfrac  FROM pg_statistic WHERE starelid = 'test'::regclass AND staattnum =
3;
stanullfrac
-------------
  0.9981884
```

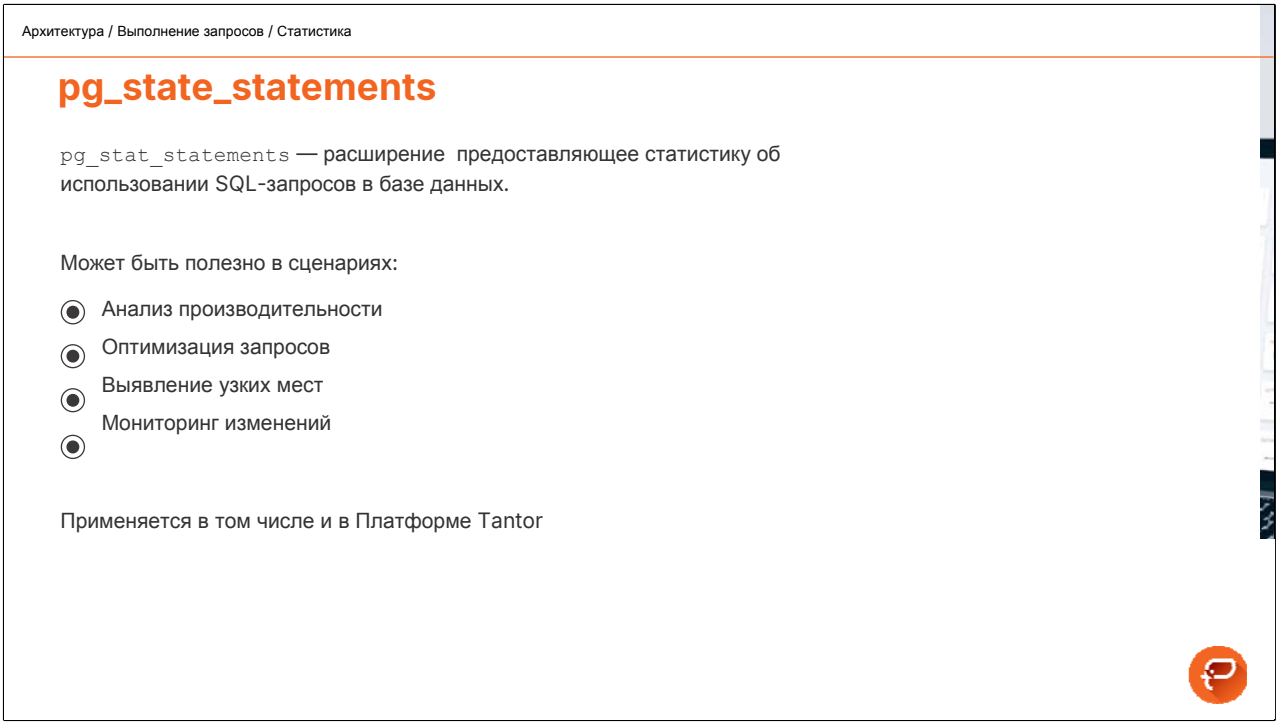
Таблица `pg_statistic` содержит статистическую информацию о данных в базе данных. Эти записи формируются в результате выполнения операции `ANALYZE` и служат для оптимизации запросов планировщиком. Важно отметить, что статистическая информация всегда представляет собой приблизительные значения, даже при актуальном их обновлении.

В таблице `pg_statistic` содержатся следующие статистические данные для каждого столбца. К примеру посмотрим какое количество `NULL` есть в третьем столбце таблицы

```
SELECT stanullfrac  FROM pg_statistic WHERE starelid =
'test'::regclass AND staattnum = 3;
stanullfrac
-------------
  0.9981884
```

Статистика о `null_frac` может быть использована оптимизатором для принятия решений о том, как лучше обрабатывать запросы, учитывая наличие `NULL` значений в столбце.

Более подробно в документации https://docs.tantorlabs.ru/tdb/ru/15_4/se/catalog-pg-statistic.html#pg-statistic



`pg_stat_statements` — расширение предоставляющее статистику об использовании SQL-запросов в базе данных. Оно отслеживает различные метрики, такие как общее количество выполненных запросов, время выполнения запросов, количество строк, затронутых каждым запросом, а также другие характеристики. Этот модуль полезен для анализа производительности и оптимизации запросов в приложении.

Вот несколько сценариев, когда `pg_stat_statements` может быть полезен:

Анализ производительности: Модуль `pg_stat_statements` позволяет администраторам баз данных и разработчикам получить общее представление о том, какие запросы чаще всего выполняются, каково их среднее время выполнения и какие запросы занимают больше всего ресурсов.

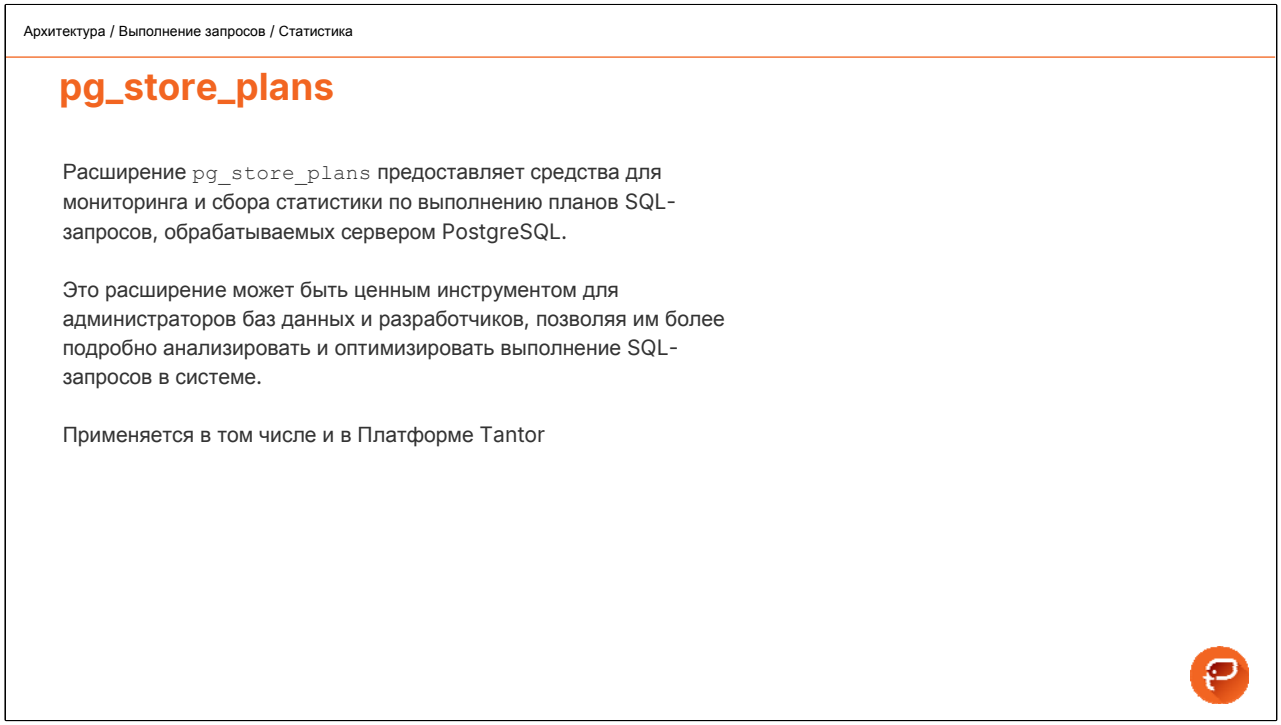
Оптимизация запросов: Путем анализа статистики запросов, можно выявить медленные запросы или запросы с высокой нагрузкой на базу данных. Это может помочь в оптимизации запросов или внесении изменений в схему базы данных для повышения производительности.

Выявление узких мест: Статистика запросов может помочь выявить узкие места в приложении, такие как запросы, которые выполняются слишком часто или занимают слишком много времени. Это может помочь в обнаружении проблем и улучшении общей производительности.

Мониторинг изменений: `pg_stat_statements` позволяет отслеживать изменения в производительности запросов с течением времени. Если какие-то запросы становятся медленнее из-за изменений в данных или индексах, это может быть замечено с помощью статистики.

Для активации `pg_stat_statements` в PostgreSQL, необходимо добавить или раскомментировать соответствующие строки в файле конфигурации PostgreSQL (postgresql.conf) и перезапустить сервер. После этого можно использовать предоставляемые представления, такие как `pg_stat_statements`, для анализа статистики запросов.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/pgstatstatements.html#f-48-pg-stat-statements

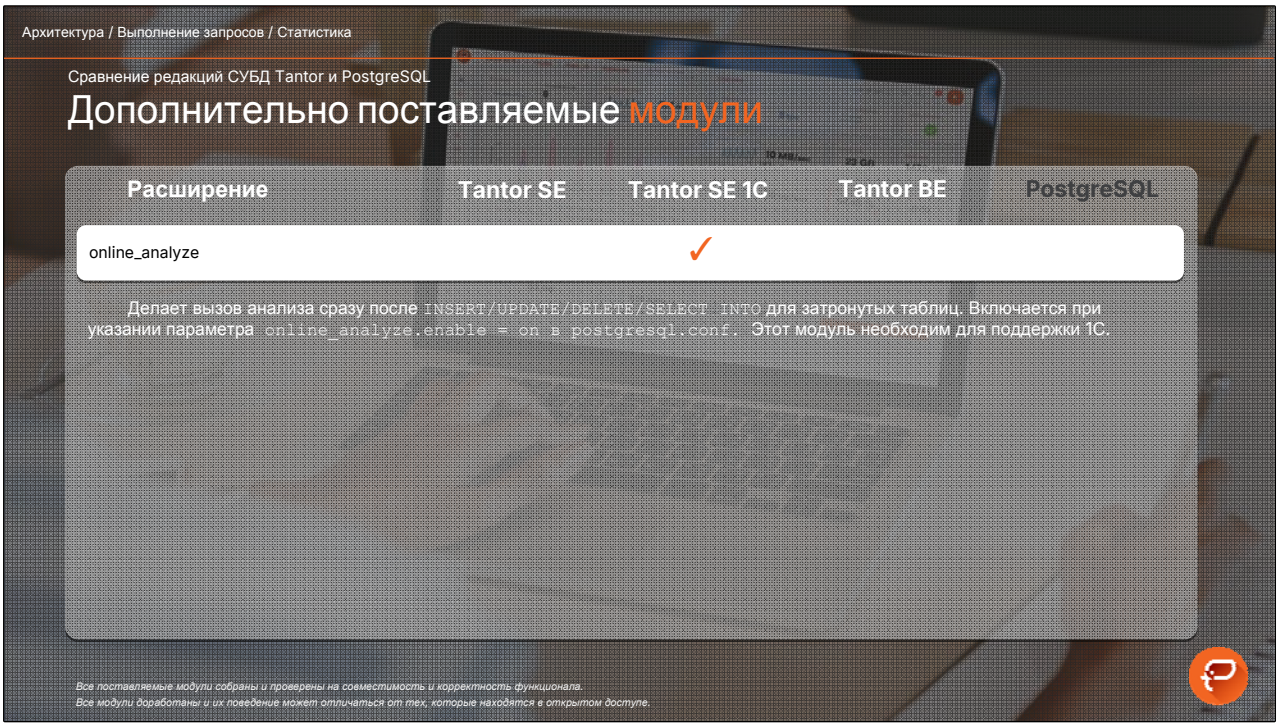


Расширение `pg_store_plans` предоставляет средства для мониторинга и сбора статистики по выполнению планов SQL-запросов, обрабатываемых сервером PostgreSQL.

После активации данного расширения, собранная статистика становится доступной через специальное системное представление с названием `pg_store_plans`. Представление содержит записи, каждая из которых представляет уникальный набор идентификаторов базы данных, пользователя и запроса. Информация в представлении может включать в себя различные аспекты выполнения запросов, такие как время выполнения, использование индексов, и другие статистические данные.

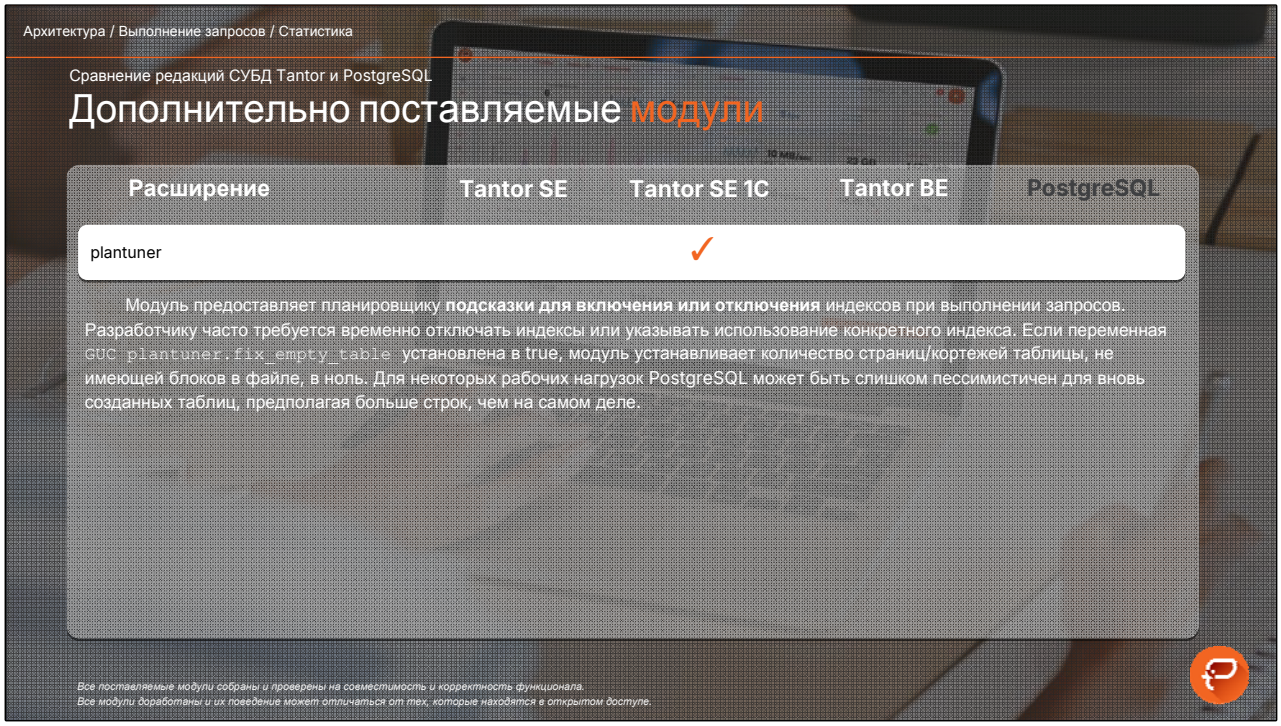
Расширение может быть ценным инструментом для администраторов баз данных и разработчиков, позволяя им более подробно анализировать и оптимизировать выполнение SQL-запросов в системе. Путем изучения статистики планов выполнения запросов можно выявить узкие места, оптимизировать индексы или решить проблемы производительности.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/pg_store_plans.html#f-50-pg-store-plans



Модуль `online_analyze`, также реализован в версии 1С.

Делает вызов анализа сразу после `INSERT/UPDATE/DELETE/SELECT INTO` для затронутых таблиц. Включается при указании параметра `online_analyze.enable = on` в `postgresql.conf`. Этот модуль необходим для поддержки 1С.

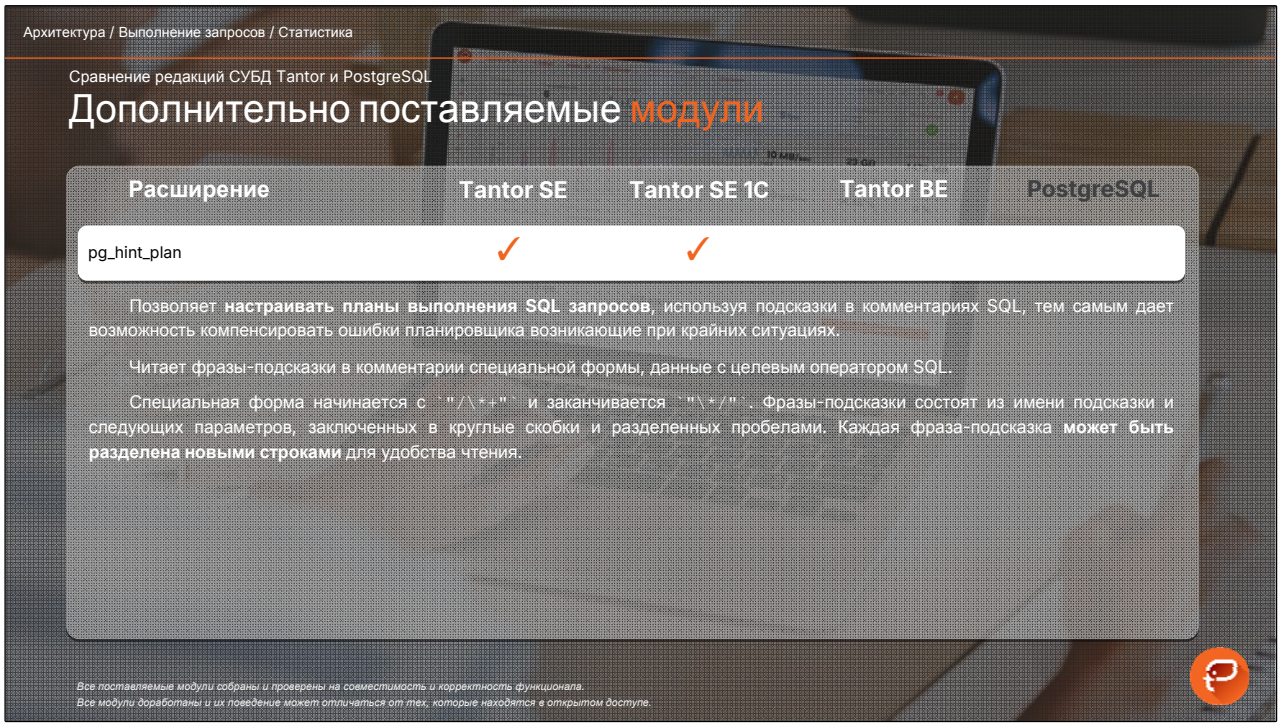


Модуль `plantuner` доступен в версии Tantor SE 1C.

Предоставляет планировщику подсказки, которые могут отключать или включать индексы для выполнения запросов.

Есть много ситуаций, когда разработчик хочет временно отключить определенные индексы, не удаляя их, или дать инструкцию планировщику использовать определенный индекс. Если переменная `GUC plantuner.fix_empty_table` установлена в true, то модуль устанавливает в ноль количество страниц/кортежей таблицы, которая не имеет блоков в файле.

Для некоторых рабочих нагрузок PostgreSQL может быть слишком пессимистичен для вновь созданных таблиц и предполагает гораздо больше строк в таблице, чем на самом деле есть.



Модуль `pg_hint_plan` — Доступен в версиях SE и SE 1c.

Позволяет настраивать планы выполнения SQL запросов, используя так называемые «подсказки» в комментариях SQL, тем самым давая возможность компенсировать ошибки планировщика возникающие при крайних ситуациях.

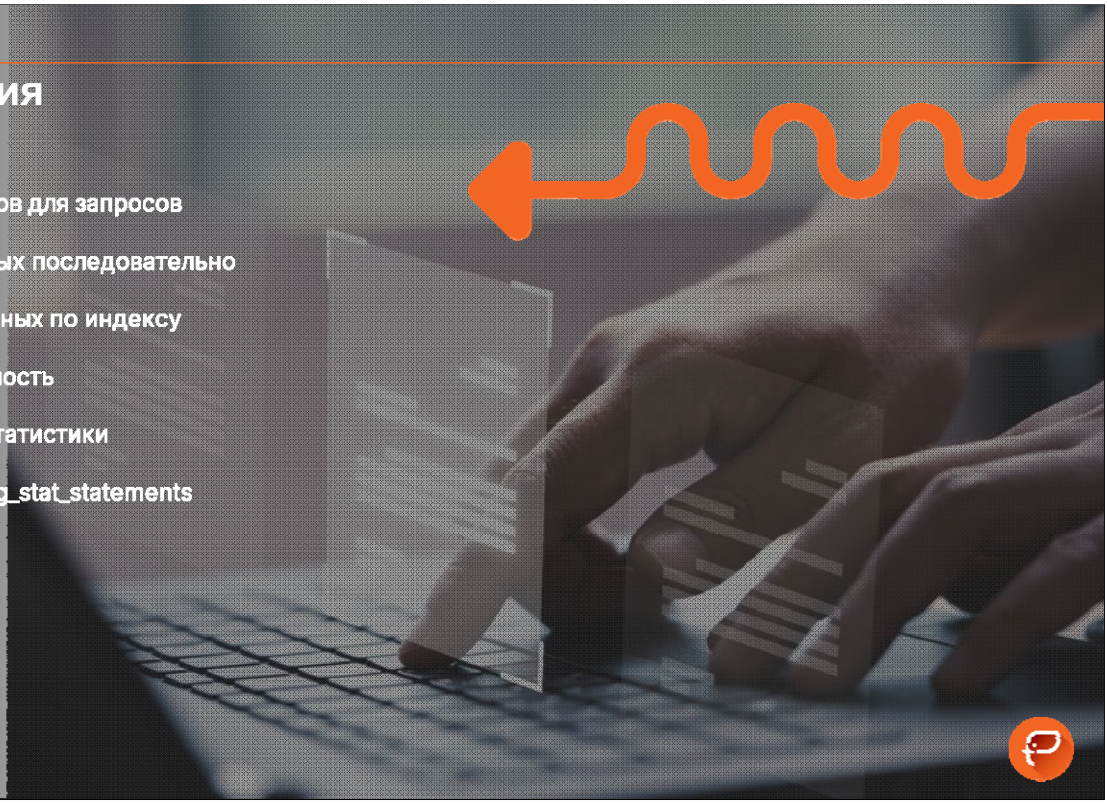
Читает фразы-подсказки в комментарии специальной формы, данные с целевым оператором SQL.

Специальная форма начинается и заканчивается с последовательности символов, которые вы видите на экране.

Фразы-подсказки состоят из имени подсказки и следующих параметров, заключенных в круглые скобки и разделенных пробелами. Каждая фраза-подсказка может быть разделена новыми строками для удобства чтения.

Демонстрация

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



Выполнение запросов

Часть 1. Создание объектов для запросов

Загрузим `psql`:

```
astra@tantor:~$ psql
psql (16.1)
Введите "help", чтобы получить справку.

postgres=#
```

Создадим новую таблицу и заполним данными.

```
postgres=# CREATE TABLE test (col1 integer, col2 integer, name text);
CREATE TABLE

postgres=# INSERT INTO test VALUES (1,2,'test1');
INSERT 0 1

postgres=# INSERT INTO test VALUES (3,4,'test2');
INSERT 0 1
```

Создадим представление над таблицей.

```
postgres=# CREATE VIEW v_table AS
             SELECT * FROM test;
CREATE VIEW

postgres=# SELECT col1, col2 FROM v_table WHERE name='test1'::text ;
 col1 | col2
------+------
    1 |    2
(1 строка)
```

Часть 2. Извлечение данных последовательно

С помощью команды `Explain` посмотрим план выполнения запроса.

```
postgres=# EXPLAIN
         SELECT col1, col2 FROM v_table WHERE name='test1'::text ;

                      QUERY PLAN
-------------------------------------------------------
Seq Scan on test  (cost=0.00..25.00 rows=6 width=8)
  Filter: (name = 'test1'::text)
(2 строки)
```

Видим, что использовалось последовательное чтение таблицы test. то есть представление было раскрыто, данные извлечены непосредственно с таблицы.

Давайте применим параметры `analyze и buffers`. Они показывают что запрос был выполнен реально и какое количество страниц было затронуто.

```
postgres=# EXPLAIN(analyze, buffers, costs off, timing off)
SELECT col1, col2 FROM v_table WHERE name='test1'::text ;
```

```
                QUERY PLAN
----------------------------------------
Seq Scan on test (actual rows=1 loops=1)
  Filter: (name = 'test1'::text)
  Rows Removed by Filter: 1
  Buffers: shared read=1
Planning Time: 0.063 ms
Execution Time: 9.569 ms
(6 строк)
```

Часть 3. Возвращение данных по индексу

Создадим индекс по столбцу col1.

```
postgres=# CREATE INDEX ON test (col1);
CREATE INDEX

postgres=# \d test
                          Таблица "public.test"
 Столбец |   Тип   | Правило сортировки | Допустимость NULL | По умолчанию
---------+---------+--------------------+-------------------+--------------
 col1    | integer |                    |                   |
 col2    | integer |                    |                   |
 name    | text    |                    |                   |
Индексы:
   "test_col1_idx" btree (col1)
```

Можно убедится, что формирование имени индекса производится автоматически

добавим информации к таблице.

```
postgres=# INSERT INTO test(col1,col2)
            SELECT generate_series(3,1003), generate_series(4,1004);
INSERT 0 1001
```

Посмотрим, что получится если будем выбирать малое количество строк. То есть, случай когда будет высокая селективность и маленькая кардинальность.

```
postgres=# EXPLAIN(analyze, buffers, costs off, timing off)
SELECT col1, col2 FROM test WHERE col1<20;

                          QUERY PLAN
---------------------------------------------------------------
Index Scan using test_col1_idx on test (actual rows=19 loops=1)
   Index Cond: (col1 < 20)
   Buffers: shared hit=3
 Planning:
   Buffers: shared hit=17
 Planning Time: 0.179 ms
 Execution Time: 0.117 ms
(7 строк)
```

Убедились, что используется индексный доступ.

Часть 4. Низкая селективность

Теперь отберем большое количество строк.

```
postgres=# SELECT count(*) FROM test;
count
-------
 1003
(1 строка)

Всего строк 1003

postgres=# EXPLAIN(analyze, buffers, costs off, timing off)
SELECT col1, col2 FROM test WHERE col1>20;
                QUERY PLAN
-------------------------------------------
 Seq Scan on test (actual rows=983 loops=1)
   Filter: (col1 > 20)
   Rows Removed by Filter: 20
   Buffers: shared hit=5
 Planning:
   Buffers: shared hit=3
 Planning Time: 0.157 ms
 Execution Time: 0.201 ms
(8 строк)
```

Отобрано 983 строки. что означает низкая селективность и высокая кардинальность. Убедились, что в этом случае индексный доступ становится дорогим и СУБД переходит к последовательному доступу.

Часть 5. Использование статистики

К примеру, при заполнении таблицы test третий столбец был не заполнен. Давайте посмотрим какой процент будет иметь значение NULL

Пересоберем статистику.

```
postgres=# ANALYZE test;
ANALYZE

postgres=# SELECT stanullfrac FROM pg_statistic WHERE starelid =
'test'::regclass AND staattnum = 3;
stanullfrac
-------------
  0.9981884
(1 row)
```

Как видно из таблицы `pg_statistic` 99%

Часть 6. Представление `pg_stat_statements`

Убедимся что представление установлено

```
postgres=# \dx pg_stat_statements
                                Список установленных расширений
      Имя          | Версия | Схема  |                              Описание
-------------------+--------+--------+-----------------------------------------------------------------
-----------
pg_stat_statements| 1.10    | public | track planning and execution statistics of all SQL
statements executed
(1 строка)
```

Посмотрим, какие столбцы есть в представлении

```
postgres=# \d pg_stat_statements
                        Представление "public.pg_stat_statements"
        Столбец         |          Тип          | Правило сортировки |
Допустимость NULL | По умолчанию
------------------------+-----------------------+--------------------+------------
-------+--------------
 userid                 | oid                   |                    |
|
 dbid                   | oid                   |                    |
|
 toplevel               | boolean               |                    |
|
 queryid                | bigint                |                    |
|
 query                  | text                  |                    |
|
 plans                  | bigint                |                    |
|
 total_plan_time        | double precision      |                    |
|
 min_plan_time          | double precision      |                    |
|
 max_plan_time          | double precision      |                    |
|
 mean_plan_time         | double precision      |                    |
|
 stddev_plan_time       | double precision      |                    |
|
 calls                  | bigint                |                    |
|
 total_exec_time        | double precision      |                    |
|
 min_exec_time          | double precision      |                    |
|
 max_exec_time          | double precision      |                    |
|
 mean_exec_time         | double precision      |                    |
|
 stddev_exec_time       | double precision      |                    |
|
 rows                   | bigint                |                    |
|
 shared_blks_hit        | bigint                |                    |
|
 shared_blks_read       | bigint                |                    |
|
 shared_blks_dirtied    | bigint                |                    |
|
 shared_blks_written    | bigint                |                    |
|
 local_blks_hit         | bigint                |                    |
|
 local_blks_read        | bigint                |                    |
|
 local_blks_dirtied     | bigint                |                    |
|
 local_blks_written     | bigint                |                    |
|
 temp_blks_read         | bigint                |                    |
|
```

```
 temp_blks_written        | bigint              |                           |
|
 blk_read_time              | double precision |                           |
|
 blk_write_time             | double precision |                           |
|
 temp_blk_read_time       | double precision |                           |
|
 temp_blk_write_time      | double precision |                           |
|
 wal_records                | bigint             |                           |
|
 wal_fpi                    | bigint             |                           |
|
 wal_bytes                  | numeric            |                           |
|
 jit_functions             | bigint             |                           |
|
 jit_generation_time      | double precision |                           |
|
 jit_inlining_count       | bigint             |                           |
|
 jit_inlining_time         | double precision |                           |
|
 jit_optimization_count | bigint              |                           |
|
 jit_optimization_time   | double precision |                           |
|
 jit_emission_count       | bigint             |                           |
|
 jit_emission_time         | double precision |                           |
|
```

Сбросим статистику применения представления

```
postgres=# SELECT pg_stat_statements_reset();
pg_stat_statements_reset
--------------------------

(1 строка)
```

Обратимся к таблице `test`

```
postgres=# EXPLAIN (analyze)
SELECT col1, col2 FROM test WHERE col1>20;
                                            QUERY PLAN

-------------------------------------------------------------------------------------
------
 Seq Scan on test   (cost=0.00..17.54 rows=984 width=8) (actual time=0.022..0.132
rows=983 loops=1)
   Filter: (col1 > 20)
   Rows Removed by Filter: 20
 Planning Time: 0.190 ms
 Execution Time: 0.234 ms
(5 строк)
```

С помощью представления `pg_stat_statements` посмотрим сколько времени заняло выполнение запроса и сколько страниц было задействовано.

```
postgres=# SELECT queryid, substring(query FOR 100) as query, total_exec_time as ms,
shared_blks_hit as blocks
```

```
from pg_stat_statements
WHERE query LIKE '%col1, col2%';

      queryid          |                     query                     |     ms
| blocks
----------------------+-----------------------------------------------+----------
+--------
 -3250261183448805182 | EXPLAIN (analyze)                            +| 0.491265
|     11
                      |   SELECT   col1,   col2   FROM   test   WHERE   col1>$1   |
|
(1 строка)
```

Практика

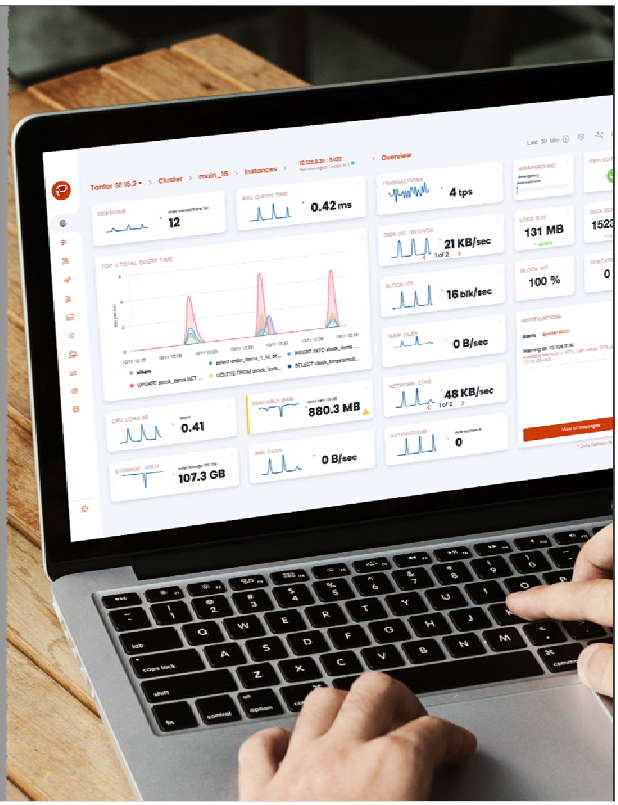
1. Создание объектов для запросов
2. Извлечение данных последовательно
3. Возвращение данных по индексу
4. Низкая селективность
5. Использование статистики
6. Представление pg_stat_statements

Дополнительное задание:

Сделать задание 1-6 с помощью pgAdmin

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость
  - Механизм использования
  - Обновление
  - FDW
  - Демонстрация
  - Практическая работа



Историческая справка

Расширяемость (extensibility) в контексте PostgreSQL означает способность системы быть легко расширяемой и адаптируемой под различные потребности пользователя.

Исторически, PostgreSQL активно разрабатывалась с упором на расширяемость:

- Типы данных и операторы
- Языки программирования
- Расширения
- Функциональные возможности
- Внешние источники данных
-

Новые возможности можно устанавливать без перекомпилирования ядра

Расширяемость (extensibility) в контексте PostgreSQL означает способность системы быть легко расширяемой и адаптируемой под различные потребности пользователя. PostgreSQL предоставляет механизмы для добавления новых функциональностей, типов данных, языков программирования, иных расширений и модулей. Расширяемость является важным аспектом для того, чтобы база данных могла эффективно справляться с разнообразными задачами и требованиями различных приложений.

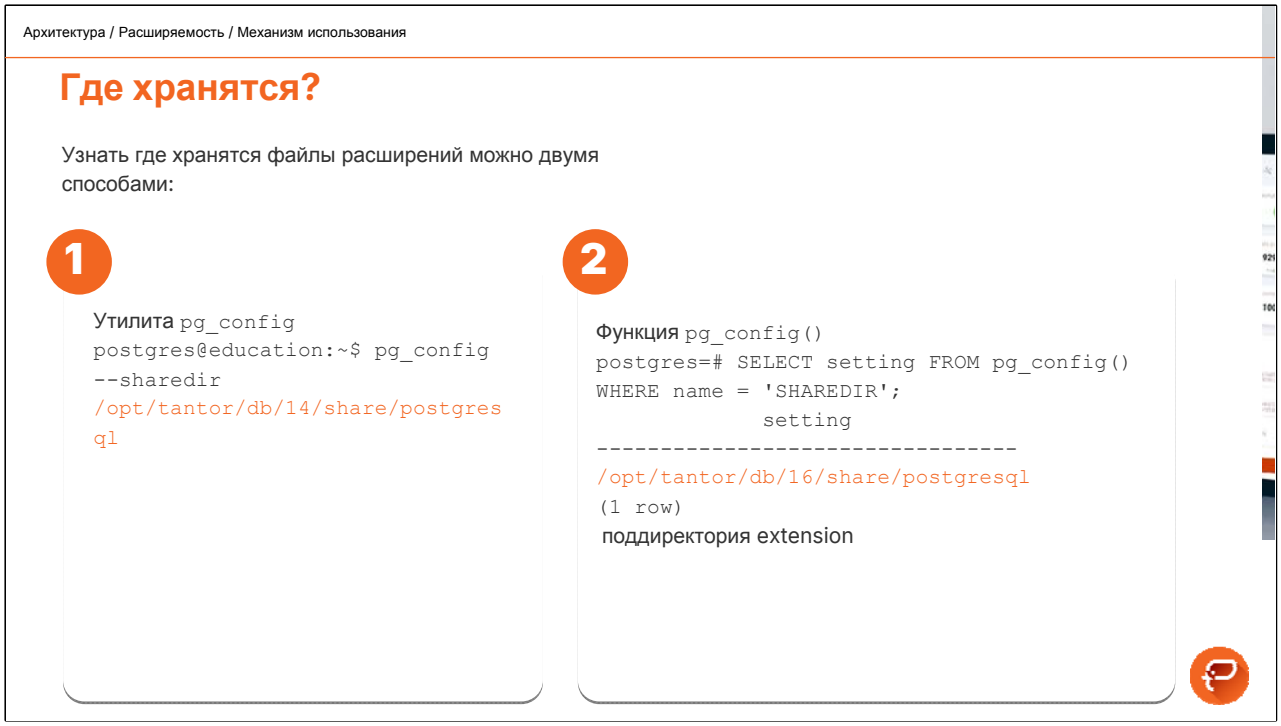
Исторически, PostgreSQL активно разрабатывалась с упором на расширяемость. В ранних версиях PostgreSQL, еще когда она называлась `POSTGRES`, создатель системы Майкл Стоунбрейкер (Michael Stonebraker) и его команда уделяли внимание расширяемости, предоставляя пользователю гибкость в определении новых типов данных, функций, и внешних языков программирования.

С течением времени PostgreSQL была доработана и улучшена множеством сообщества разработчиков. В результате этих усилий, PostgreSQL стала одной из самых расширяемых реляционных систем управления базами данных (СУБД).

Её расширяемость проявляется в следующих аспектах:

- **Типы данных и операторы:** PostgreSQL позволяет определять собственные пользовательские типы данных, операторы и функции, что обеспечивает гибкость в описании структуры данных.
- **Языки программирования**: В PostgreSQL можно встраивать и использовать различные языки программирования, включая `PL/pgSQL`, `PL/Tcl`, `PL/Perl`, `PL/Python`, и другие. Это позволяет разработчикам использовать тот язык, который наилучшим образом соответствует их задачам.
- **Расширения** (Extensions): Введение механизма расширений позволяет легко добавлять функциональность к PostgreSQL без изменения ядра. Расширения представляют собой логическую группировку объектов базы данных, которую можно установить или удалить как единое целое.
- **Функциональные возможности**: PostgreSQL предоставляет множество встроенных функций и возможностей, а также поддерживает использовани**е внешних библиотек и модулей.**
- **Внешние источники данных** (Foreign Data Wrappers — FDW): PostgreSQL предоставляет механизм Foreign Data Wrappers, который позволяет интегрировать данные из внешних источников, таких как другие базы данных, веб-сервисы или файлы, напрямую в запросы SQL. FDW обеспечивает возможность работать с данными, расположенными в удаленных источниках, как если бы они были локальными.

В результате всех этих возможностей PostgreSQL стала популярной СУБД для различных приложений, от небольших проектов до крупных корпоративных систем. Расширяемость PostgreSQL позволяет адаптировать базу данных под специфические требования различных предприятий и обеспечивает её эволюцию в соответствии с развивающимися технологическими потребностями.



Утилита `pg_config` в PostgreSQL предоставляет информацию о конфигурации вашей установки PostgreSQL и позволяет узнать директорию, где хранятся расширения.

- Если вам нужна конкретная информация о директории для какого-то определенного расширения, вы можете воспользоваться следующей командой:

```
postgres@education:~$ pg_config --sharedir
/opt/tantor/db/14/share/postgresql
```

Эта команда вернет путь к общей директории, где хранится информация, общая для всех баз данных PostgreSQL, включая расширения. Внутри этой директории, обычно, есть подкаталог `extension`, в котором могут храниться файлы расширений.

- Функция `pg_config()`

```
postgres=# SELECT setting FROM pg_config()
WHERE name = 'SHAREDIR';
              setting
-----------------------------------
/opt/tantor/db/16/share/postgresql
(1 row)
```

Расширение в PostgreSQL является механизм модификации системного каталога конкретной базы данных с которой идет работа в сессии, потому что вносит изменения в саму базу данных, расширяя её функциональность. Расширение может добавлять новые типы данных, функции, операторы, агрегатные функции и другие объекты базы данных.

При установке расширения в систему PostgreSQL происходит изменение системных каталогов и таблиц метаданных, чтобы система могла использовать новые возможности, предоставляемые расширением.

Во время установки PostgreSQL проверяет, что все зависимости расширения удовлетворены, и производит необходимые изменения в системных каталогах для обеспечения правильной работы расширения.

В PostgreSQL существует несколько команд для управления расширениями:

- **`CREATE EXTENSION:`**

Пример `CREATE EXTENSION my_extension;`

В этой команде my_extension — это имя устанавливаемого расширения. Если у расширения есть зависимости, PostgreSQL автоматически установит их.

- **`ALTER EXTENSION:`**

Пример `ALTER EXTENSION my_extension UPDATE TO '1.1';`

В этой команде `my_extension` — это имя установленного расширения, и `'1.1'` - это целевая версия. Эта команда также может использоваться для других операций с расширением, таких как добавление или удаление файла управления.

- **`DROP EXTENSION:`**

```
DROP EXTENSION my_extension;
```

В этой команде `my_extension` — это имя устанавливаемого расширения. При удалении расширения также будут удалены связанные с ним объекты базы данных.

Важно отметить, что управление расширениями в PostgreSQL может также включать в себя работу с файлами SQL для создания, обновления и удаления объектов расширения, а также с командами управления файлами и директориями, если вы создаете собственные расширения.

Кроме того, для просмотра информации о текущих расширениях в базе данных, вы можете использовать запрос к каталогу `pg_available_extensions` или `pg_extension`.

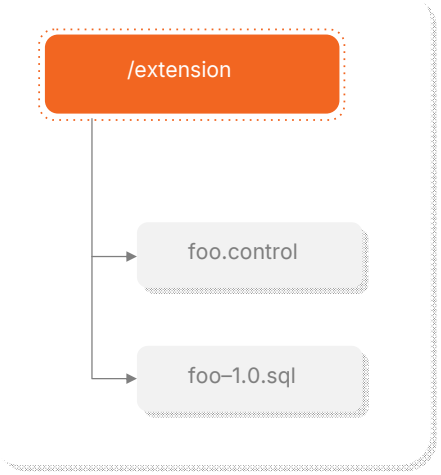
более подробно
https://docs.tantorlabs.ru/tdb/ru/15_4/se/extend.html

Необходимые и достаточные файлы

Для установки расширения минимально требуется:

- Управляющий файл `имя_расширения.control`
- Файл с объектами расширения `имя_расширения-1.0.sql`

Эти два файла являются минимальным набором для установки расширения. Управляющий файл содержит метаданные о расширении, а файл с объектами расширения содержит SQL-код для создания необходимых объектов базы данных, таких как функции, таблицы и т.д.



Для установки расширения минимально требуется

Управляющий файл `имя_расширения.control`. Этот файл содержит метаданные о расширении и имеет структуру аналогичную файлу postgresql.conf.

Файл с объектами `расширения имя_расширения-1.0.sql` (где 1.0 — версия расширения). Этот файл содержит SQL-код для создания объектов базы данных, необходимых для работы расширения. Это могут быть функции, таблицы, типы данных и другие объекты.

Эти два файла являются минимальным набором для установки расширения. Управляющий файл содержит метаданные о расширении, а файл с объектами расширения содержит SQL-код для создания необходимых объектов базы данных, таких как функции, таблицы и т.д.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/extend-extensions.html#id-1.8.3.19.11

Содержимое управляющего файла

Содержимое управляющего файла расширения

```
# расширение foo
comment = 'Пример расширения'
default_version = '1.0'
module_pathname = '$libdir/имя_расширения'
relocatable = true
schema = public
```

Эти параметры могут быть важными для расширений, которые предназначены для работы в различных сценариях и требуют гибкости в расположении и схеме.

Содержимое управляющего файла расширения (например, `foo.control`) должно включать минимально необходимые метаданные для правильной установки и работы расширения.

Вот пример простого управляющего файла:

```
# расширение foo
comment = 'Пример расширения'
default_version = '1.0'
module_pathname = '$libdir/имя_расширения'
relocatable = true
schema = public
```

Разберем, что означает каждый параметр:

- **`comment`:** Описывает расширение. Это произвольная строка, предоставляющая краткое описание того, что делает расширение.
- **`default_version`:** Указывает на версию расширения по умолчанию. В данном примере - `'1.0'`. Это может быть важно при установке и обновлении расширения, когда необходимо указать конкретную версию.
- **`module_pathname`:** Задает путь к динамической библиотеке расширения. Обычно используется `$libdir`, который представляет собой путь к директории, где хранятся динамические библиотеки PostgreSQL.

Дополнительные параметры:

- **`relocatable`:** Этот параметр указывает, может ли расширение быть перемещено в другую схему после установки. Если установлено значение `true`, расширение считается перемещаемым.
- **`schema`:** Этот параметр указывает схему, в которой должны создаваться объекты расширения. В данном примере указана схема `public`, но вы можете изменить её на другую, если это соответствует вашим требованиям.

Эти параметры могут быть важными для расширений, которые предназначены для работы в различных сценариях и требуют гибкости в расположении и схеме.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/extend-extensions.html#id-1.8.3.19.11

Содержимое скрипта

Пример файла foo–1.0.sql

#первым идет защита от применения скрипта без установки расширения

```
\echo Use "CREATE EXTENSION foo" to load this file. \quit
```

#затем идут операторы создания объектов и их заполнение

```
CREATE OR REPLACE FUNCTION …..
CREATE TABLE …..
CREATE INDEX …..
```

Вот пример файла SQL для расширения "foo", где первая строка содержит подсказку для пользователя:

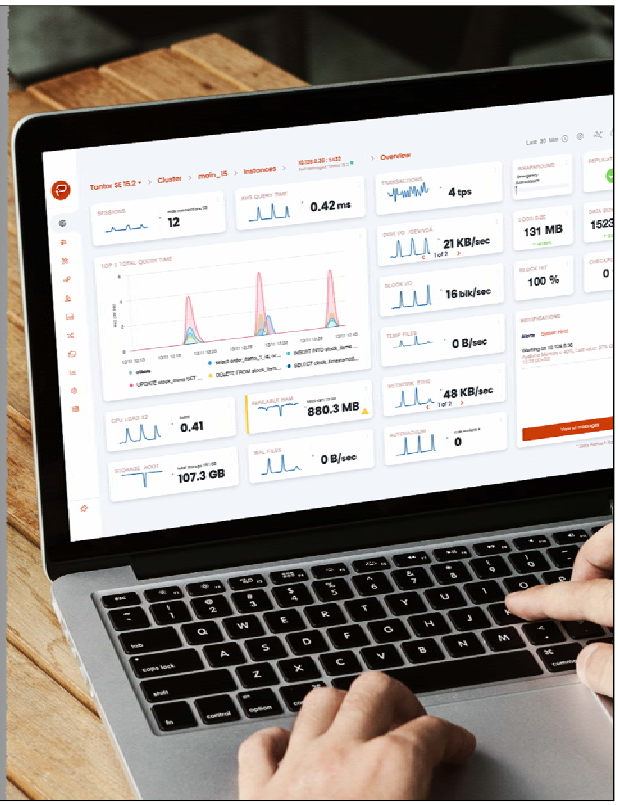
```
\echo Use "CREATE EXTENSION foo" to load this file. \quit
-- foo--1.0.sql
CREATE OR REPLACE FUNCTION foo_function() RETURNS text AS $$
BEGIN
  RETURN 'Hello, foo!';
END;
$$ LANGUAGE plpgsql;
CREATE TABLE foo_table (
  id serial PRIMARY KEY,
  name text
);
```

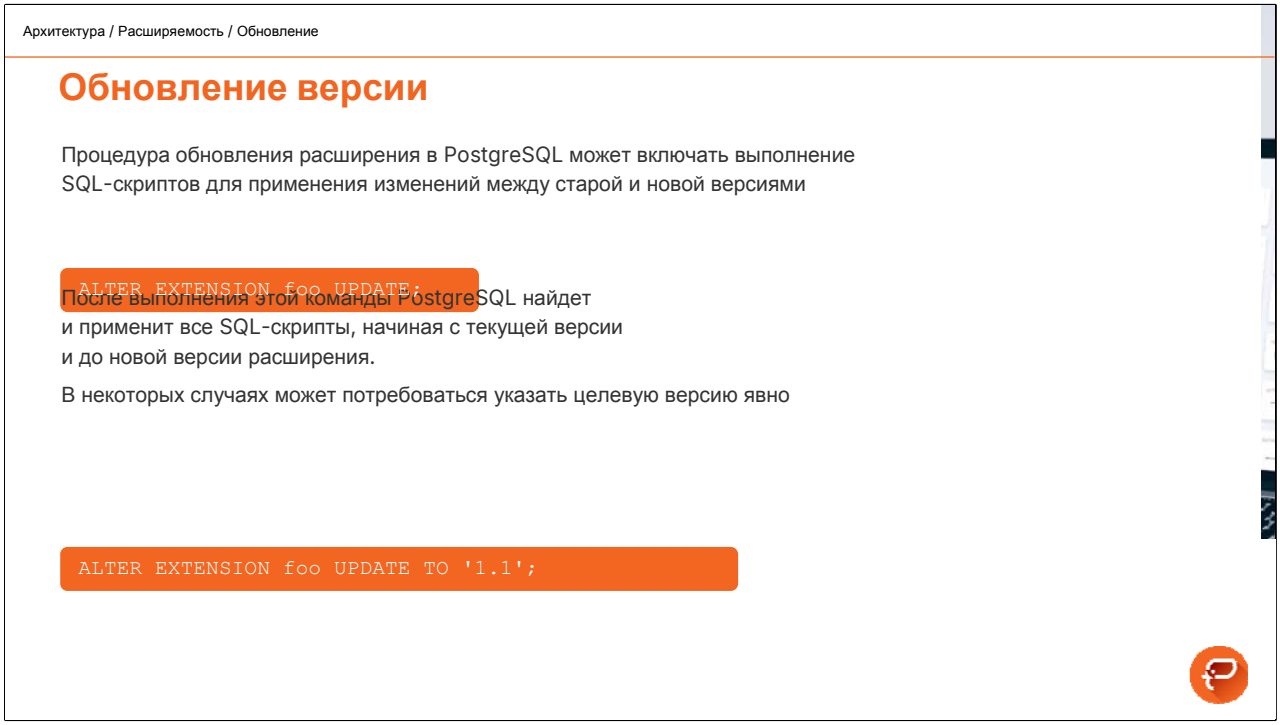
Этот файл содержит комментарий с помощью \echo, который предоставляет подсказку пользователю о том, как использовать данный файл. При помощи \quit завершается выполнение файла после его выполнения. Затем следует SQL-код для создания функции и таблицы в расширении "foo". В код может включаться также любые команды изменяющие объекты и т.д.

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

- Механизм использования
- Обновление
- FDW
- Демонстрация
- Практическая работа





Процедура обновления расширения в PostgreSQL может включать выполнение SQL-скриптов для применения изменений между старой и новой версиями.

Вот более подробный второй пункт:

Применение SQL-скриптов для обновления до текущей версии:

Если изменения расширения требуют применения дополнительных SQL-скриптов, вы можете использовать команду `ALTER EXTENSION ... UPDATE`. Эта команда выполняет все необходимые SQL-скрипты, связанные с обновлением версии расширения до текущей версии.

```
ALTER EXTENSION foo UPDATE;
```

После выполнения этой команды PostgreSQL найдет и применит все SQL-скрипты, начиная с текущей версии и до новой версии расширения.

В некоторых случаях может потребоваться указать целевую версию явно:

```
ALTER EXTENSION foo UPDATE TO '1.1';
```

Обязательно ознакомьтесь с документацией к вашему конкретному расширению, так как процедуры обновления могут варьироваться в зависимости от конкретных изменений и требований разработчиков.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/extend-extensions.html#id-1.8.3.19.14

Пути обновления

Представляют собой набор версий расширения и связанных с ними SQL-скриптов обновления, которые позволяют перейти от одной версии расширения к другой.

Для просмотра версий используется представление pg_available_extension_versions

```
SELECT name, version FROM
pg_available_extension_versions WHERE name = 'foo';
  name    | version
----------+---------
foo                  | 1.0
foo                  | 1.1
foo                  | 1.2
```

Функция pg_extension_update_paths позволяет посмотреть возможные пути обновления

```
source | target |        path
--------+--------+--------------------
 1.0     | 1.1     | 1.0--1.1
1.0      | 1.2     | 1.0--1.2
```

«Пути обновления расширения» (extension update paths) в PostgreSQL представляют собой набор версий расширения и связанных с ними SQL-скриптов обновления, которые позволяют перейти от одной версии расширения к другой. Эти пути обновления обеспечивают механизм для безопасного и последовательного обновления расширения в базе данных.

Для просмотра доступных путей обновления расширения в PostgreSQL можно воспользоваться системной функцией pg_available_extension_versions. Эта функция возвращает информацию о доступных версиях расширений и их путях обновления.

- Пример использования pg_available_extension_versions:

```
SELECT name, version FROM pg_available_extension_versions WHERE name = 'foo';
  name    | version
----------+---------
foo              | 1.0
foo              | 1.1
foo              | 1.2
(3 rows)
```

Этот запрос вернет информацию о версиях расширения "foo" и их путях обновления, если такие пути определены. Обратите внимание, что не все расширения могут иметь явно определенные пути обновления.

Функция pg_extension_update_paths позволяет посмотреть возможные пути обновления для расширения.

- Пример использования pg_extension_update_paths:

```
SELECT * FROM pg_extension_update_paths('foo');
```

Этот запрос вернет информацию о всех путях обновления для расширения "foo", включая те, которые не являются прямыми.

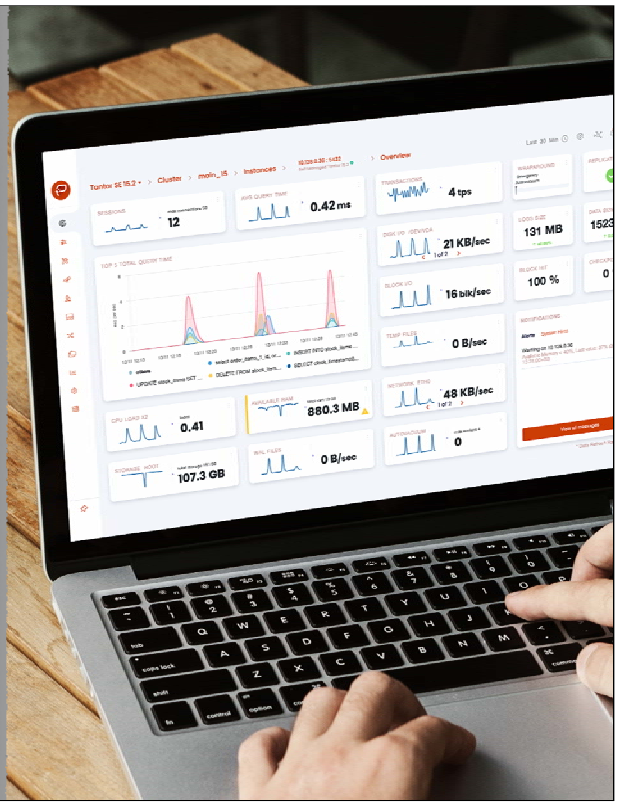
```
source | target |        path
--------+--------+--------------------
 1.0     | 1.1     | 1.0--1.1
1.0      | 1.2     | 1.0--1.2
```

На примере видно, что с версии 1.1 до версии 1.2 напрямую обновится нельзя. Нужно будет сначала удалить версию 1.1 а потом поставить версию 1.2

Темы

- Общие сведения
- Структуры памяти
- Многоверсионность
- Регламентные работы
- Выполнение запросов
- Расширяемость

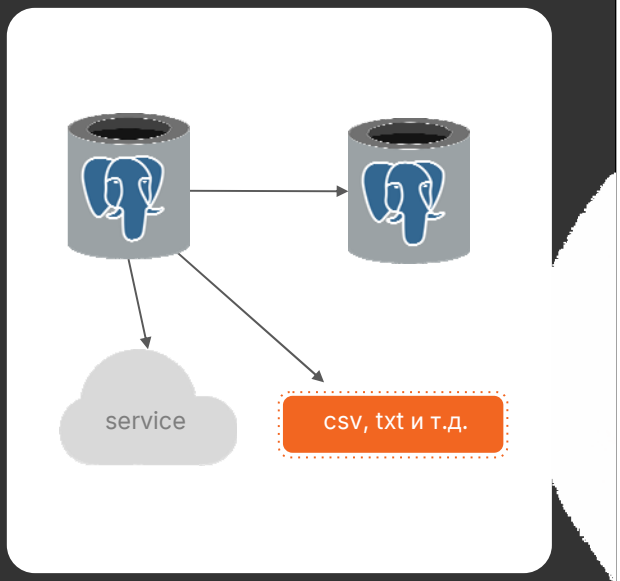
- Механизм использования
- Обновление
- FDW
- Демонстрация
- Практическая работа



Внешние источники информации

FDW в PostgreSQL означает «Foreign Data Wrapper» (Обертка для внешних данных).

Позволяет PostgreSQL взаимодействовать с данными,
которые хранятся вне самой базы данных
например в других базах данных, файлах CSV, веб-службах и т. д.



FDW в PostgreSQL означает «Foreign Data Wrapper» (Обертка для внешних данных). Это механизм, который позволяет PostgreSQL взаимодействовать с данными, которые хранятся вне самой базы данных PostgreSQL, например, в других базах данных, файлах CSV, веб-службах и т. д. FDW позволяет PostgreSQL создавать виртуальные таблицы, которые отображают данные из внешних источников, как если бы они были обычными таблицами в PostgreSQL.

Написание внешнего обертки данных
https://docs.tantorlabs.ru/tdb/ru/15_4/se/fdwhandler.html

Гетерогенные базы данных

В PostgreSQL существует несколько различных FDW, включая:

- `postgres_fdw` позволяет PostgreSQL взаимодействовать с другими экземплярами PostgreSQL

Сторонние разработки:

- `mysql_fdw`: Этот FDW позволяет PostgreSQL взаимодействовать с базами данных MySQL.
- `oracle_fdw`: Позволяет PostgreSQL взаимодействовать с базами данных Oracle.
- `sqlite_fdw`: Позволяет PostgreSQL работать с базами данных SQLite.
- `mongo_fdw`: Позволяет PostgreSQL взаимодействовать с базами данных MongoDB.
- `redis_fdw`: Этот FDW позволяет PostgreSQL работать с данными из базы данных Redis и т.д.

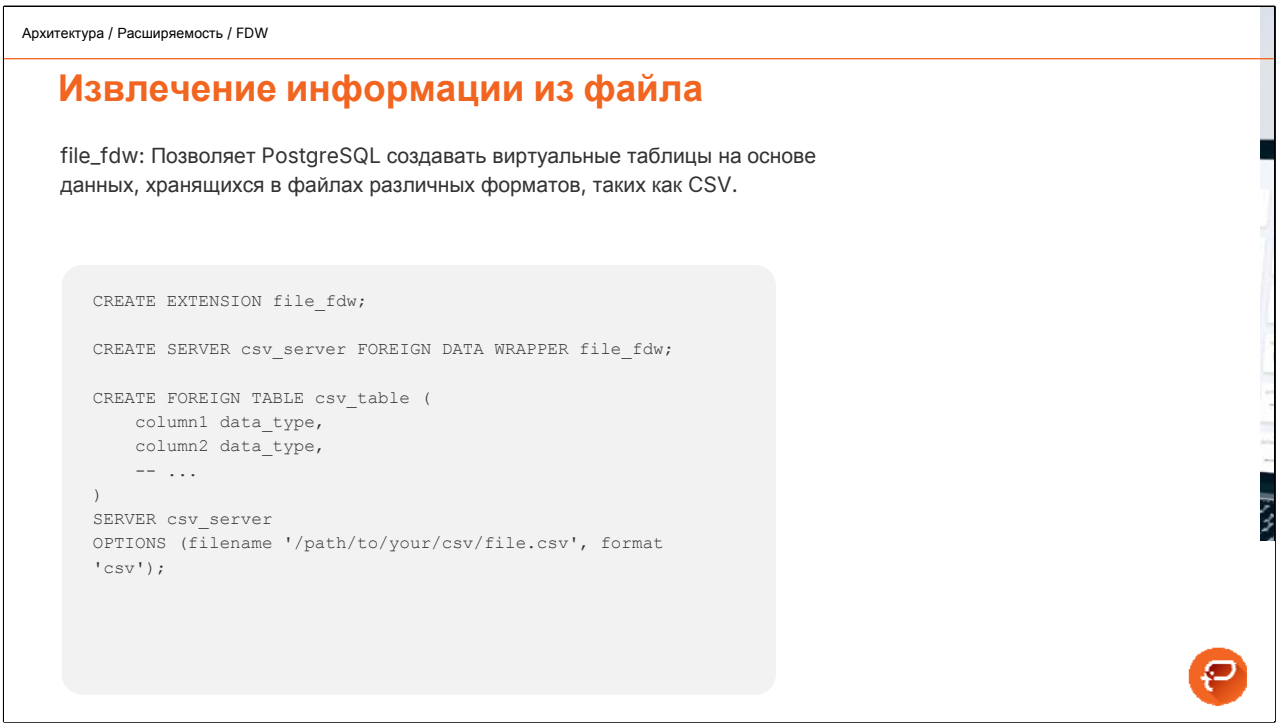
В PostgreSQL существует несколько различных FDW, включая:

- **`postgres_fdw`:** Этот FDW позволяет PostgreSQL взаимодействовать с другими экземплярами PostgreSQL. Это может быть полезно, например, для построения распределенных систем или для работы с данными, хранящимися на удаленных серверах PostgreSQL.

Также существуют обертки внешних данных сторонние:

- **`mysql_fdw`:** Этот FDW позволяет PostgreSQL взаимодействовать с базами данных MySQL.
- **`oracle_fdw`:** Позволяет PostgreSQL взаимодействовать с базами данных Oracle.
- **`sqlite_fdw`:** Позволяет PostgreSQL работать с базами данных SQLite.
- **`mongo_fdw`:** Позволяет PostgreSQL взаимодействовать с базами данных MongoDB.
- **`redis_fdw`:** Этот FDW позволяет PostgreSQL работать с данными из базы данных Redis.

И так далее. Каждый из этих FDW предоставляет интерфейс для взаимодействия с конкретным типом внешних данных. Вы можете выбрать подходящий FDW в зависимости от вашего конкретного случая использования и требований.



`file_fdw`: Позволяет PostgreSQL создавать виртуальные таблицы на основе данных, хранящихся в файлах различных форматов, таких как CSV.

Этот FDW предоставляет возможность создавать виртуальные таблицы на основе данных, хранящихся в файлах различных форматов, таких как CSV. Вы можете использовать его для чтения данных из CSV-файлов и представления их как таблиц в PostgreSQL.

Пример использования:

```
CREATE EXTENSION file_fdw;
CREATE SERVER csv_server FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE csv_table (
    column1 data_type,
    column2 data_type,
    -- ...
)
SERVER csv_server
OPTIONS (filename '/path/to/your/csv/file.csv', format 'csv');
```

https://docs.tantorlabs.ru/tdb/ru/15_4/se/file-fdw.html#f-19-file-fdw

Демонстрация

1. Определить в какой директории лежат файлы расширения
2. Посмотреть установленные расширения
3. Посмотреть доступные расширения
4. Установить и удалить произвольное обновление
5. Посмотреть доступные версии расширения.
   Обновить до актуальной версии
6. Обертки внешних данных

Расширяемость

Часть 1. Определить в какой директории лежат файлы расширения.

Перейдем под пользователя postgres.

```
astra@education:~$ sudo su - postgres
```

В командной строке воспользуемся утилитой `pg_config`.

```
postgres@education:~$ pg_config --sharedir

/opt/tantor/db/16/share/postgresql
```

Добавим к получившемуся пути `extension`.

```
postgres@education:~$ ls -l /opt/tantor/db/16/share/postgresql/extension/
итого 9768
-rw-r--r-- 1 root root    274 Apr 18  2023 adminpack--1.0--1.1.sql
-rw-r--r-- 1 root root   1535 Apr 18  2023 adminpack--1.0.sql
-rw-r--r-- 1 root root   1682 Apr 18  2023 adminpack--1.1--2.0.sql
-rw-r--r-- 1 root root    595  Apr 18  2023 adminpack--2.0--2.1.sql
-rw-r--r-- 1 root root    176 Apr 18  2023 adminpack.control
……
……
```

Загрузим psql.

```
postgres@tantor:~$ psql
psql (16.1)
Введите "help", чтобы получить справку.

postgres=#
```

Определим путь расширения с помощью функции `pg_config()`.

```
postgres=# SELECT setting FROM pg_config()
where name = 'SHAREDIR';

              setting
-------------------------------------
/opt/tantor/db/16/share/postgresql
(1 row)
```

Часть 2. Посмотреть установленные расширения

```
postgres=# \dx
                                        Список установленных расширений
        Имя          | Версия |   Схема    |                                Описание
---------------------+--------+------------+-------------------------------------------------------
-----------
 pg_stat_statements | 1.10   | public     | track planning and execution statistics of all SQL
statements executed
 pg_store_plans     | 1.6.4  | public     | track plan statistics of all SQL statements executed
 plpgsql            | 1.0    | pg_catalog | PL/pgSQL procedural language
(3 строки)
```

Часть 3. Посмотреть доступные расширения

Воспользуемся расширением `pg_available_extensions`

```
postgres=# SELECT * from pg_available_extensions;
        name        | default_version | installed_version |
comment
--------------------+-----------------+-------------------+------------------
------------------------------------------------------------------------------
 plpgsql            | 1.0             | 1.0               | PL/pgSQL
procedural language
 page_repair        | 1.0             |                   | Individual page
reparing
 pg_hint_plan       | 1.6.0           |                   |
 dblink             | 1.2             |                   | connect to other
PostgreSQL databases from within a database
 tcn                | 1.0             |                   | Triggered change
notifications
 pg_trgm            | 1.6             |                   | text similarity
measurement and index searching based on trigrams
 pg_buffercache     | 1.4             |                   | examine the
shared buffer cache


—————————————

 dict_xsyn          | 1.0             |                   | text search
dictionary template for extended synonym processing
 pg_variables       | 1.2             |                   | session variables
with various types
 old_snapshot       | 1.0             |                   | utilities in
support of old_snapshot_threshold
 pgcrypto           | 1.3             |                   | cryptographic
functions
 file_fdw           | 1.0             |                   | foreign-data
wrapper for flat file access
 amcheck            | 1.3             |                   | functions for
verifying relation integrity
 seg                | 1.4             |                   | data type for
representing line segments or floating-point intervals
 pg_background      | 1.2             |                   | Run SQL queries
in the background
(69 строк)
```

На данный момент установлено 69 расширений в текущем кластере БД. Их можно установить в любой БД.

Часть 4. Установить и удалить произвольное обновление

Например установим расширение `pg_surgery`.

```
postgres=# CREATE EXTENSION pg_surgery;
CREATE EXTENSION

postgres=# \dx
                                    Список установленных расширений
        Имя          | Версия |   Схема    |                        Описание
---------------------+--------+------------+-----------------------------------------
----------
 pg_stat_statements | 1.10   | public     | track planning and execution statistics of
all SQL statements executed
 pg_store_plans     | 1.6.4  | public     | track plan statistics of all SQL
statements executed
```

```
 pg_surgery         | 1.0     | public     | extension to perform surgery on a damaged
relation
 plpgsql            | 1.0     | pg_catalog | PL/pgSQL procedural language
(4 строки)
```

Посмотрим содержимое расширения.

```
postgres=# \dx+ pg_surgery
     Объекты в расширении "pg_surgery"
             Описание объекта
-----------------------------------------
 функция heap_force_freeze(regclass,tid[])
 функция heap_force_kill(regclass,tid[])
(2 строки)
```

Удалим расширение.

```
postgres=# DROP EXTENSION pg_surgery;
DROP EXTENSION
```

Часть 5. Посмотреть доступные версии расширения. Обновить до актуальной версии

Воспользуемся представлением `pg_available_extension_versions`

```
postgres=# SELECT name, version FROM pg_available_extension_versions WHERE
name = 'adminpack';
  name     | version
-----------+---------
 adminpack | 1.0
 adminpack | 1.1
 adminpack | 2.0
 adminpack | 2.1
(4 строки)
```

Для начала установим версию 1.0

```
postgres=# CREATE EXTENSION adminpack VERSION '1.0';

CREATE EXTENSION
postgres=# \dx adminpack
                    Список установленных расширений
    Имя     | Версия |   Схема    |                Описание
------------+--------+------------+----------------------------------------
 adminpack | 1.0    | pg_catalog | administrative functions for PostgreSQL
(1 строка)
```

Посмотрим содержимое расширения.

```
postgres=# \dx+ adminpack

    Объекты в расширении "adminpack"
             Описание объекта
-----------------------------------------
 функция pg_file_length(text)
 функция pg_file_read(text,bigint,bigint)
 функция pg_file_rename(text,text)
 функция pg_file_rename(text,text,text)
 функция pg_file_unlink(text)
 функция pg_file_write(text,text,boolean)
 функция pg_logdir_ls()
```

```
 функция pg_logfile_rotate()
(8 строк)
```

Посмотрим, можно ли расширение обновить до версии 2.1. Воспользуемся функцией `pg_extension_update_paths`.

```
postgres=# SELECT * FROM pg_extension_update_paths('adminpack');
source | target |        path
--------+--------+--------------------
 1.0    | 1.1    | 1.0--1.1
 1.0    | 2.0    | 1.0--1.1--2.0
 1.0    | 2.1    | 1.0--1.1--2.0--2.1
 1.1    | 1.0    |
 1.1    | 2.0    | 1.1--2.0
 1.1    | 2.1    | 1.1--2.0--2.1
 2.0    | 1.0    |
 2.0    | 1.1    |
 2.0    | 2.1    | 2.0--2.1
 2.1    | 1.0    |
 2.1    | 1.1    |
 2.1    | 2.0    |
(12 строк)
```

Обновим расширение до версии 2.1

```
postgres=# ALTER EXTENSION adminpack UPDATE;
ALTER EXTENSION

postgres=# \dx adminpack
                    Список установленных расширений
    Имя    | Версия |   Схема    |                Описание
-----------+--------+------------+----------------------------------------
 adminpack | 2.1    | pg_catalog | administrative functions for PostgreSQL
(1 строка)

postgres=# \dx+ adminpack
    Объекты в расширении "adminpack"
           Описание объекта
-----------------------------------------
 функция pg_file_rename(text,text)
 функция pg_file_rename(text,text,text)
 функция pg_file_sync(text)
 функция pg_file_unlink(text)
 функция pg_file_write(text,text,boolean)
 функция pg_logdir_ls()
(6 строк)
```

Как видите, содержимое расширения изменилось. Удалим расширение.

```
postgres=# DROP EXTENSION adminpack;
DROP EXTENSION
```

Часть 6. Обертки внешних данных

Посмотрим, какие есть обертки внешних данных (FDW).

```
postgres=# SELECT * FROM pg_available_extensions
           WHERE name LIKE '%fdw%';
    name     | default_version | installed_version |                    comment
```

```
-------------+----------------+------------------+-------------------------------
----------
 postgres_fdw | 1.1            |                  | foreign-data wrapper for remote
PostgreSQL servers
 file_fdw     | 1.0            |                  | foreign-data wrapper for flat
file access
(2 строки)
```

Воспользуемся оберткой внешней данных для подключения к СУБД PostgreSQL.

```
postgres=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION

postgres=# \dx postgres_fdw
                          Список установленных расширений
     Имя       | Версия | Схема  |                        Описание
--------------+--------+--------+-----------------------------------------------
--------
 postgres_fdw | 1.1    | public | foreign-data wrapper for remote PostgreSQL
servers
(1 строка)
```

Посмотрим какие есть базы данных.

```
postgres=# \l
                                                          Список баз данных
    Имя     | Владелец | Кодировка | Провайдер локали | LC_COLLATE  |
LC_CTYPE   | локаль ICU | Правила ICU |     Права доступа
-----------+----------+-----------+------------------+-------------+---------
----+------------+-------------+-----------------------
 postgres  | postgres | UTF8      | libc             | ru_RU.UTF-8 |
ru_RU.UTF-8 |            |             |
 template0 | postgres | UTF8      | libc             | ru_RU.UTF-8 |
ru_RU.UTF-8 |            |             | =c/postgres          +
           |          |           |                  |             |
|            |             | postgres=CTc/postgres
 template1 | postgres | UTF8      | libc             | ru_RU.UTF-8 |
ru_RU.UTF-8 |            |             | =c/postgres          +
           |          |           |                  |             |
|            |             | postgres=CTc/postgres
 test_db   | postgres | UTF8      | libc             | ru_RU.UTF-8 |
ru_RU.UTF-8 |            |             |
(4 строки)
```

Подключимся и вернем информацию с БД test_db.

Для начала создадим объект удаленного сервера.

```
postgres=# CREATE SERVER test FOREIGN DATA WRAPPER postgres_fdw
           OPTIONS (host 'localhost', port '5432', dbname 'test_db');
CREATE SERVER


postgres=# \des
         Список сторонних серверов
 Имя  | Владелец | Обёртка сторонних данных
------+----------+--------------------------
 test | postgres | postgres_fdw
(1 строка)
```

После этого создадим пользователя, под которым будет происходить подключение.

Отображение на пользователя может быть несколько.

```
postgres=# CREATE USER MAPPING FOR postgres SERVER test
OPTIONS ( user 'postgres', password 'postgres' );
CREATE USER MAPPING

postgres=# \deu
Список сопоставлений пользователей
 Сервер | Имя пользователя
--------+------------------
 test   | postgres
(1 строка)
```

После создадим таблицу к которой можно будет подключится.

```
postgres=# CREATE FOREIGN TABLE order_remote
( id bigint, name varchar(32))
server test
OPTIONS ( schema_name 'public', table_name 'order_items_1'
);
CREATE FOREIGN TABLE

postgres=# \det
   Список сторонних таблиц
 Схема  |   Таблица    | Сервер
--------+--------------+--------
 public | order_remote | test
(1 строка)
```

Обращаемся к этой таблице, как к обычной таблице

```
postgres=# SELECT * FROM order_remote LIMIT 10;
id | name
----+------
 0 |
 1 |
 2 |
 3 |
 4 |
 5 |
 6 |
 7 |
 8 |
 9 |
(10 строк)
```

Описание удаленной таблице можно получить как обычной.

```
postgres=# \d order_remote
                                Сторонняя таблица "public.order_remote"
 Столбец |          Тип          | Правило сортировки | Допустимость NULL | По умолчанию |
Параметры ОСД
---------+-----------------------+--------------------+-------------------+--------------+-------
--------
 id      | bigint                |                    |                   |              |
 name    | character varying(32) |                    |                   |              |
Сервер: test
Параметр ОСД: (schema_name 'public', table_name 'order_items_1')
```

Посмотрим откуда приходят данные.

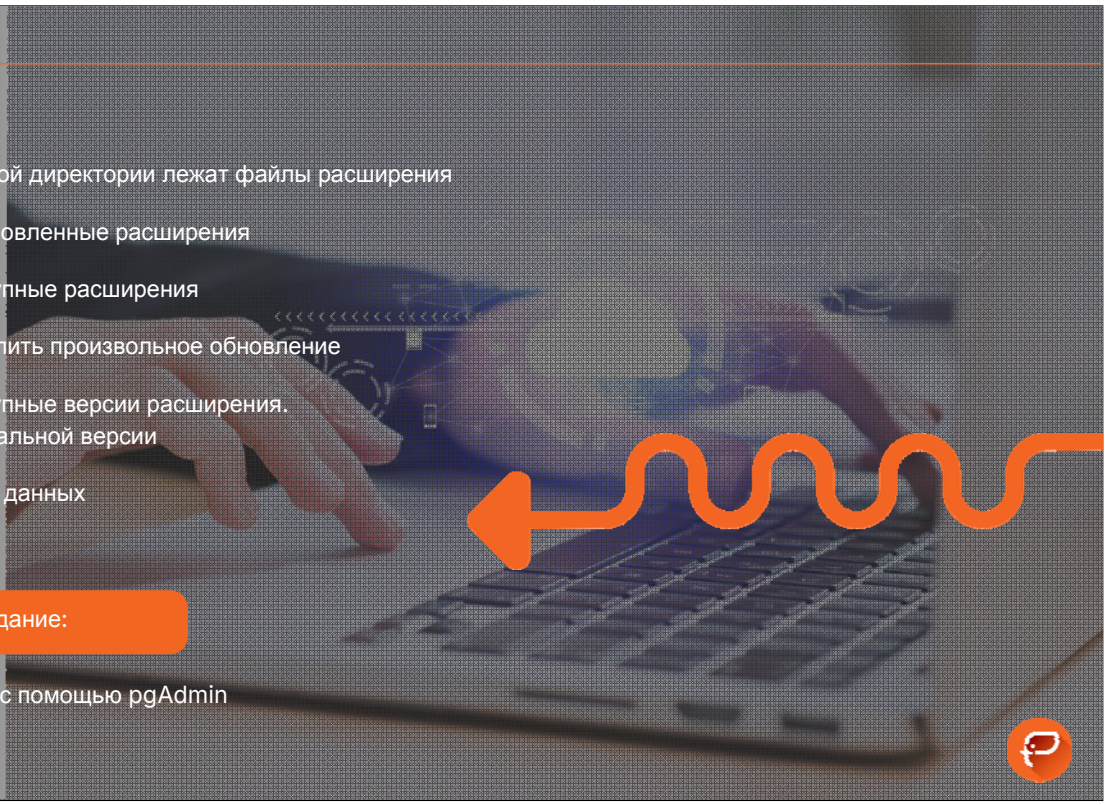
```
postgres=# EXPALIN SELECT * FROM order_remote LIMIT 10;
                            QUERY PLAN
------------------------------------------------------------------------
Foreign Scan on order_remote  (cost=100.00..100.42 rows=10 width=90)
(1 строка)
```

Очистим базу данных.

```
postgres=# DROP FOREIGN TABLE order_remote;
DROP FOREIGN TABLE

postgres=# DROP USER MAPPING FOR postgres server test;
DROP USER MAPPING

postgres=# DROP SERVER test;
DROP SERVER

postgres=# DROP EXTENSION postgres_fdw;
DROP EXTENSION
```

Практика

1. Определить в какой директории лежат файлы расширения
2. Посмотреть установленные расширения
3. Посмотреть доступные расширения
4. Установить и удалить произвольное обновление
5. Посмотреть доступные версии расширения.
   Обновить до актуальной версии
6. Обертки внешних данных



Дополнительное задание:

Сделать задание 1-6 с помощью pgAdmin

Спасибо!

tantorlabs.ru
edu@tantorlabs.ru

03. Конфигурирование

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

Существует около 370 параметров конфигурации, которые влияют на работу экземпляра.

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 (нечувствительно к регистру) и значение.

Значения параметров могут быть:
логическими (значение "bool" в столбце vartype представления pg_settings)
строковыми ("string")
целыми числами ("integer", "int64")
десятичными числами ("real")
числами ("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/15_4/se/runtime-config-custom.html

https://docs.tantorlabs.ru/tdb/ru/15_4/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` **`-c`** превалируют над установленными в файлах параметров
  `pg_ctl start` **`-o "-c`** `config_file=/opt/postgresql.conf`**`"`**

При создании кластера создаются два файла:
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` Он 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), то параметры, специфичные для резервного узла можно поместить в `postgresql.conf` вне `PGDATA` и этот файл не будет затёрт при восстановлении. Утилита `pg_rewind` также синхронизирует только директорию `PGDATA` и табличных пространств.

Просмотр параметров

- Команда `psql \dconfig`
  - можно искать по шаблону `\dconfig *`имя`*`
  - можно использовать клавишу табуляции
- Команда `SHOW` имя_параметра**;**
  - можно использовать клавишу табуляции
  - нужно посылать на выполнение набрав “**;**”
  - только один параметр
- функция `current_setting(`имя параметра`)`

Текущие значения параметров кластера можно и удобно просматривать командой psql **\dconfig** маска_параметра

Например, `postgres=# \dconfig *data_d*`

```
               List of configuration parameters
       Parameter       |                  Value
-----------------------+-----------------------------------
 data_directory        | /var/lib/postgresql/tantor-se-
16/data
 data_directory_mode | 0750
```

покажет значения параметров а которых встречается строка `data_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 “**;**” иначе она останется в буфере psql.

```
postgres=# show data_directory;
            data_directory
---------------------------------------
 /var/lib/postgresql/tantor-se-16/data
(1 row)
```

Очистить буфер psql можно командой `\r`

```
postgres=# \r
Query buffer reset (cleared).
```

Функция `current_setting(`имя параметра`)` – аналог команды `SHOW`.

Просмотр параметров

- Представления `pg_settings  pg_file_settings`
- Команда `SHOW ALL;`
  - аналог `SELECT * FROM pg_settings;`
  - нельзя отфильтровать параметры
- Файлы `postgresql.conf postgresql.auto.conf`

```
ls -CF /var/lib/postgresql/tantor-se-16/data
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 одного параметра на запущенном или остановленном экземпляре
  `postgres -C` имя_параметра

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.

просмотр одного параметра на запущенном или остановленном экземпляре:

`postgres -C` имя_параметра

https://docs.tantorlabs.ru/tdb/ru/15_2/se/config-setting.html

Изменение параметров конфигурации

Файл postgresql.conf

- основной файл, в котором хранятся параметры конфигурации кластера
- может содержать более 300 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 include и include_dir заданными внутри postgresql.conf

Файл postgresql.conf - основной файл, в котором хранятся параметры конфигурации кластера. Имеется около 370 параметров конфигурации плюс параметры расширений и разделяемых библиотек (*.so) загружаемых с помощью параметра конфигурации shared_preload_libraries

Файл создаётся утилитой initdb из файла примеров

```
/opt/tantor/db/16/share/postgresql$ ls -w 1 *.sample
pg_hba.conf.sample
pg_ident.conf.sample
pg_service.conf.sample
postgresql.conf.sample
psqlrc.sample
```

В файлы *.sample можно внести изменения. Закомментированные строки начинаются символом #

Утилита initdb вносит изменения в часть строк в зависимости от переданных ей параметров, переменных окружения установленных перед её запуском, внутренней логики. Изменения можно посмотреть сравнив файлы

```
diff postgresql.conf postgresql.conf.sample
65c65
< max_connections = 100                 # (change requires
restart)
---
> #max_connections = 100                        # (change
requires restart)
```

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 postgresql.auto.conf

- Расположен в PGDATA
- Параметры добавляются командой ALTER SYSTEM SET параметр = значение;
- Параметры убираются командой ALTER SYSTEM RESET параметр;
  ALTER SYSTEM RESET ALL;
- После ALTER SYSTEM нужно перечитать конфигурацию или перегрузить кластер

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 не может выполняться в рамках транзакции.

Применение изменений параметров конфигурации

- 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 -1 postgres
```

- Параметры заданные в `postgresql.auto.conf` перекрывают значения параметров `postgresql.conf`
- Если в `postgresql.conf` параметр указан несколько раз, применяется указанный ближе к концу файла

Для применения изменений (перечитывания) в текстовых файлах параметров конфигурации самое удобное использовать функцию

```
SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

Также можно использовать утилиту

```
pg_ctl reload -D /var/lib/postgresql/tantor-se-16/data
server signaled
```

Также можно послать сигнал SIGHUP (номер 1) основному процессу.

Например, послать сигнал процессам с названием postgres (синоним “postmaster”) всех запущенных кластеров:

```
killall -1 postgres
```

Параметры заданные в `postgresql.auto.conf` перекрывают значения параметров `postgresql.conf`

Если в `postgresql.conf` параметр указан несколько раз, применяется указанный 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`

Часть параметров конфигурации может быть изменена только ролью с атрибутом SUPERUSER

```
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

https://docs.tantorlabs.ru/tdb/ru/15_2/se/ddl-priv.html

https://docs.tantorlabs.ru/tdb/ru/15_2/se/view-pg-settings.html

Контекст параметров

- **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

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           |    18
 postmaster         |    64
 sighup             |    96
 superuser          |    44
 superuser-backend  |     4
 user               |   143
(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 утилиту `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/15_2/se/view-pg-settings.html

Уровни установки параметров

Параметры устанавливаются на уровнях:

- 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;

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/15_2/se/sql-alterdatabase.htm

https://docs.tantorlabs.ru/tdb/ru/15_2/se/sql-alterrole.html

Параметры устанавливаются на уровнях:

- 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};
```

На уровне транзакции значение меняется командой `SET LOCAL`.

Пример:

`SET work_mem to '16MB';` или `SELECT set_config('work_mem', '16MB', false);` где `false` – 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 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;
```

https://docs.tantorlabs.ru/tdb/ru/15_2/se/sql-set.html

https://docs.tantorlabs.ru/tdb/ru/15_2/se/sql-alterprocedure.html

Параметры хранения 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
- `ALTER TABLE` имя `ALTER COLUMN` имя `SET STATISTICS` число`;` перекрывает значение параметра конфигурации `default_statistics_target`

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`.

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>
```

https://docs.tantorlabs.ru/tdb/ru/15_2/se/sql-createtable.html
https://docs.tantorlabs.ru/tdb/ru/15_2/se/sql-altertable.html

Категории параметров

Категории параметров

- Категории - логическая классификация параметров для удобства
- Каегории описывают для чего предназначены параметры

```
select category, count(name)
 from pg_settings
 group by category order by 2 desc;
                   category                    | count
-----------------------------------------------+-------
 Client Connection Defaults / Statement Behavior |    31
 Developer Options                               |    25
 Resource Usage / Memory                         |    22
 Query Tuning / Planner Method Configuration     |    22
 Reporting and Logging / What to Log             |    21
 Preset Options                                  |    18
 Write-Ahead Log / Settings                      |    15
 Connections and Authentication / SSL            |    14
 Autovacuum                                      |    13
 Query Tuning / Planner Cost Constants           |    13
...
(42 rows)
```

Параметры логически разбиты на категории. Названия можно посмотреть:

```
select category, count(name) from pg_settings
group by category order by 2 desc;
                   category                    | count
-----------------------------------------------+-------
 Client Connection Defaults / Statement Behavior |    31
 Developer Options                               |    25
 Resource Usage / Memory                         |    22
 Query Tuning / Planner Method Configuration     |    22
 Reporting and Logging / What to Log             |    21
 Preset Options                                  |    18
 Write-Ahead Log / Settings                      |    15
 Connections and Authentication / SSL            |    14
 Autovacuum                                      |    13
 Query Tuning / Planner Cost Constants           |    13
 Reporting and Logging / Where to Log            |    12
 Client Connection Defaults / Locale and Formatting |    12
 Replication / Standby Servers                   |    11
...
(42 rows)
```

Большое количество параметров относятся к настройке производительности работы: настройке выполнения запросов, автовакуума.

https://momjian.us/main/writings/pgsql/administration.pdf (стр.21-65)

Опции для разработчиков

- Категория параметров `Developer Options`
- Не должны использоваться при промышленной эксплуатации
- Некоторые параметры могут помочь получить данные в сложных случаях повреждения страниц

Категория параметров `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 страницы обычно не даёт прочесть данные на этой странице и выборка прервется. Параметр позволяет пропустить содержимое страницы и продолжить работу с другими страницами. Это позволяет пропустить ошибку и извлечь строки из неповрежденных страниц.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/runtime-config-developer.html

Пользовательские настройки (Customized Options)

- параметры расширений, библиотек или просто произвольные параметры, в названии которых присутствует точка
- разработчики библиотек и расширений дают возможность стандартным образом настраивать их функционал

```
SET myapp.par1 = '20';
SET
SHOW myapp.par1;
 myapp.par1
------------
 20
(1 row)
```

Расширения и библиотеки, подгружаемые параметром `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 не подгрузится модуль. В частности, СУБД не может проверить допустимость значений параметров при их изменении например командой `ALTER SYSTEM` поэтому до загрузки библиотеки эта команда не может устанавливать неизвестные СУБД параметры, даже если в названии параметра есть точка. На уровне сессии их можно устанавливать. 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-16/data/postgresql.conf" line 834
 FATAL:  configuration file "/var/lib/postgresql/tantor-se-16/data/postgresql.conf"
contains errors
```

https://docs.tantorlabs.ru/tdb/ru/15_2/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`

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`.

Параметры специфичные для СУБД Тантор

Параметры специфичные для СУБД Тантор

- СУБД Тантор имеет улучшения
- Часть улучшений является расширениями, часть встроено в ядро
- Многие улучшения настраиваются параметрами конфигурации

Список параметров конфигурации:

```
autonomous_session_lifetime
commit_ts_buffers
libpq_compression
multixact_members_buffers
multixact_offsets_buffers
notify_buffers
serial_buffers
subtrans_buffers
transaction_timeout
wal_sender_stop_when_crc_failed
xact_buffers
```

Отличия СУБД Тантор от PostgreSQL описаны в разделе документации:
https://docs.tantorlabs.ru/tdb/ru/15_4/se/differences.html

autonomous_session_lifetime

- по умолчанию 300 минут
- реализуется пулом сессий
- обслуживают фоновые рабочие процессы

```
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)
```

Параметр autonomous_session_lifetime задает максимальное время жизни автономной сессии. Значение по умолчанию 300 минут (5 часов). Что это такое?

Автономные транзакции можно реализовать например через dblink к своей же базе, но проблема в производительности. СУБД Тантор обеспечивает высокоскоростную реализацию автономных транзакций благодаря своей архитектуре. 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. Таким образом не тратятся ресурсы на порождение-остановку процессов, обслуживающих автономные транзакции. Серверный и фоновый процессы обмениваются данными синхронно через разделяемую память. Параметр `autonomous_session_lifetime` заставляет пересоздавать автономные сессии, чтобы избежать возможные ошибки с неограниченным по времени использованием разделяемой памяти. Допускаются вложенные автономные транзакции. Для обслуживания вложенных автономных транзакций запускаются дополнительные (до сотни) фоновые процессы.

Пример как работает автономная транзакция:

```
CREATE TABLE tbl (a int);
CREATE OR REPLACE FUNCTION func() RETURNS void
LANGUAGE plpgsql
AS $$
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO tbl VALUES (1);
END;
$$;
START TRANSACTION;
SELECT func();
ROLLBACK;
SELECT * FROM tbl;
```

Реализация автономных транзакций предложена `TantorLabs` сообществу постгрес: https://www.postgresql.org/message-id/f7470d5a-3cf1-4919-8404-5c4d91341a9f@tantorlabs.com

commit_ts_buffers

- количество памяти, используемой для кеширования содержимого директории PGDATA/pg_commit_ts хранящей данные о времени фиксации транзакций
- Буфер используется ,если track_commit_timestamp=on (по умолчанию off)

Указывает количество памяти, используемой для кэширования содержимого директории PGDATA/pg_commit_ts хранящей данные времени фиксации транзакций. Если это значение указано без единиц измерения, оно принимается в блоках (8Кб). Значение по умолчанию - 0, что равно размеру размер разделяемого пула буферов деленному на 1К (shared_buffers/1024), но не менее 4 блоков. Этот параметр может быть установлен только при запуске экземпляра.

Буфер используется ,если track_commit_timestamp=on (по умолчанию off)

Обсуждение параметра:

https://www.postgresql.org/message-id/CA+hUKGKEdeVKpOqBxzRF+Z4=j0T+CA7ERrXni5De71Mm6-dBWA@mail.gmail.com

libpq_compression

- отключено по умолчанию на стороне сервера
- 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` включает поддержку сжатия в библиотеке `libpq`, реализованная новым параметром в 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

https://docs.tantorlabs.ru/tdb/ru/15_4/se/runtime-config-connection.html#GUC-LIBPQ-COMPRESSION

multixact_members_buffers и multixact_offsets_buffers

- задают количество разделяемой памяти, используемой для кеширования содержимого двух поддиректорий PGDATA/pg_multixact
- по умолчанию 8 блоков (64 килобайт)

multixact_offsets_buffers и multixact_members_buffers задают количество разделяемой памяти, используемой для кэширования содержимого двух поддиректорий PGDATA/pg_multixact. Если значение указано без единиц измерения, оно принимается в блоках (8Кб). Значение по умолчанию 8. Этот параметр может быть установлен только при запуске сервера.

Вакуумирование позволяет удалять старые файлы из подкаталогов pg_multixact/members и pg_multixact/offsets.

Мультитранзакции используются для поддержки блокировки строк несколькими транзакциями. Поскольку в заголовке кортежа есть только ограниченное место для хранения информации о блокировке, эта информация кодируется в виде “идентификатора нескольких транзакций”, или сокращенно multixact ID, когда более одной транзакции одновременно блокируют строку. Информация о том, какие идентификаторы транзакций включены в конкретный идентификатор мультитранзакции, хранится отдельно в подкаталоге pg_multixact.

В Tantor SE используются 64-битные идентификаторы транзакций, которые вряд ли до максимума своего значения и не требуют арифметики по модулю 32 для их сравнения. На уровне страницы проблема wrap around возможна, если какая-то сессия удерживает моментальный снимок, в котором накопилось более 4 миллиардов транзакций.

Проверить, что в кластере используются 64-битные идентификаторы транзакций самое простое по значениям по умолчанию для следующих 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.

Некоторые экземпляры сталкиваются с деградацией производительности и администраторы их эксплуатирующие вынуждены либо перекомпилировать PostgreSQL увеличивая в исходном коде размеры буферов и желают, чтобы размеры буферов ("SLRU кэши") конфигурировались параметрами (архитектура GUC, grand unified configuration).

https://www.postgresql.org/message-id/801D899E-9CC5-4FEE-AAA4-16D87853FD45@yandex-team.ru

notify_buffers

- задаёт количество 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.
```

`notify_buffers` задаёт количество общей памяти, используемой для кэширования содержимого `PGDATA/pg_notify`. Если значение указано без единиц измерения, оно принимается в блоках (8Кб). Значение по умолчанию 8. Этот параметр может быть установлен только при запуске сервера.

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.
```

serial_buffers

- задаёт количество общей памяти, используемой для кеширования содержимого PGDATA/pg_serial
- Значение по умолчанию 16 блоков по 8Кб

serial_buffers задаёт количество общей памяти, используемой для кэширования содержимого PGDATA/pg_serial. Если значение указано без единиц измерения, оно принимается в блоках (8Кб). Значение по умолчанию 16. Этот параметр может быть установлен только при запуске сервера.

pg_serial хранит информацию о завершенных транзакциях уровня SERIALIZABLE. Это один из SLRU буферов.

subtrans_buffers

- задаёт количество общей памяти, используемой для кеширования содержимого PGDATA/pg_subtrans
- подтранзакции могут массово порождаться, так как подтранзакция создается если в блоке plpgsql присутствует секция EXCEPTION

subtrans_buffers задаёт количество общей памяти, используемой для кэширования содержимого PGDATA/pg_subtrans. Если значение указано без единиц измерения, оно принимается в блоках (8Кб). Значение по умолчанию 8. Этот параметр может быть установлен только при запуске сервера.

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.

Если в программном коде множество вложенных блоков с секцией EXCEPTION (устанавливает неявную точку сохранения) на высоконагруженном экземпляре, возможно параметр subtrans_buffers поможет устранить узкое место.

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 отдельному оператору
- Значение ноль (по умолчанию) отключает таймаут
- Позволяет устанавливать лимит длительности транзакций

`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. Появился в версии СУБД Тантор 15.4 Если это значение указано без единиц измерения, оно считается в миллисекундах. Значение ноль (по умолчанию) отключает таймаут.

Длительные транзакции, одиночные `SELECT` (потому что используют снимок данных), удерживают горизонт событий базы данных. Это не даёт вычищать старые версии строк.

Для защиты от долгих `SELECT`ов может использоваться параметр `old_snapshot_threshold`. Его не стоит устанавливать на физических репликах. В 17 версии `old_snapshot_threshold` планируется убрать и `transaction_timeout` позволяет его заменить.

Однако, у `old_sanpshot_threshold` есть документированная особенность: если он установлен в значение больше ноля, то вакуум и автовакуум не усекает на последней фазе свое работы файлы, а значит не устанавливает блокировку `ACCESS EXCLUSIVE` хоть и на короткое время, но ожидание получения. этой блокировки может затянуться. Усекание файла будет происходить не на всех таблицах, а только если после вакуумирования ожидается освобождение более чем 1000 страниц (8 мегабайт).

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` также можно использовать для отката длительно работающих транзакций, что может быть актуально для приложений, которые сами устанавливают ограничение на выполнение транзакции, после которого приложению эти действия уже не нужны.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/runtime-config-resource.html

wal_sender_stop_when_crc_failed

- предотвращает передачу сбойных блоков журналов WAL на реплики и в поточные архивы

```
postgres=# \x
Expanded display is on.
postgres=# select * from pg_settings where name like '%crc_f%';
-[ RECORD 1 ]---+--------------------------------------------------
name            | wal_sender_stop_when_crc_failed
setting         | off
unit            |
category        | Query Tuning / Planner Method Configuration
short_desc      | Stop replication if the WAL checksum is corrupted.
extra_desc      |
context         | user
vartype         | bool
source          | default
min_val         |
max_val         |
enumvals        |
boot_val        | off
reset_val       | off
sourcefile      |
sourceline      |
pending_restart | f
```

СУБД Тантор защищается от повреждения данных на устройствах хранения, которые могут возникнуть из-за повреждения блоков от момента передачи блока операционной системе до момента последующего чтения этого блока. Это обеспечивается подсчетом контрольных сумм по простому, быстрому, но достаточному для обнаружения повреждений алгоритму `CRC-32`.

В СУБД Тантор по умолчанию при создании кластера в момент инсталляции включается расчет и проверка контрольных сумм блоков данных (вызывается утилита `initdb` с ключом `-k`). Если создавать кластер после инсталляции утилитой `initdb`, то по умолчанию (также как в PostgreSQL) кластер создается без подсчета контрольных сумм на блоках данных.

Подсчет контрольных сумм на записях WAL всегда включён.

Если `wal_sender_stop_when_crc_failed` установлено в значение `true`, то фоновый процесс walsender, который используется для передачи журнальных записей репликам и другим клиентам (`pg_recevewal`), читает WAL-сегменты из файловой системы. Если он обнаруживает неверную контрольную сумму, то пытается прочесть запись из буфера памяти (WAL buffer). Если попытка не увенчается успехом, `walsender` остановится. Это позволит избежать распространения сбойных страниц на реплики и архивы WAL. По умолчанию значение false.

xact_buffers

- задаёт количество общей памяти, используемой для кеширования 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)

xact_buffers задаёт количество общей памяти, используемой для кэширования содержимого PGDATA/pg_xact подкаталога о статусе фиксации транзакций. Если это значение указано без единиц измерения, оно принимается в блоках (8Кб). Значение по умолчанию - 0, что равно размеру размер разделяемого пула буферов деленному на 512 (shared_buffers/512), но не менее 4 блоков. Этот параметр может быть установлен только при запуске экземпляра.

Кэширование помогает быстро определить статус транзакции. Потребность в определении статуса последних транзакций и вплоть до горизонта событий 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.

Однако, статус фиксации транзакции может быть полезен и процессам вакуумирования. Статус фиксации использует два бита на транзакцию (зафиксирована COMMIT или отменена ROLLBACK), поэтому если autovacuum_freeze_max_age установлено на максимально допустимое значение в два миллиарда, размер pg_xact ожидается около полугигабайта, а pg_commit_ts - около 20 ГБ. Если это незначительно по сравнению с общим размером вашей базы данных, рекомендуется установить autovacuum_freeze_max_age на максимально допустимое значение, так как Единственным недостатком увеличения значения autovacuum_freeze_max_age (а также vacuum_freeze_table_age вместе с ним) является то, что подкаталоги pg_xact и pg_commit_ts кластера базы данных будут занимать больше места. Или установите autovacuum_freeze_max_age в зависимости от того, сколько места вы готовы выделить для хранения pg_xact и pg_commit_ts. (Значение по умолчанию, 200 миллионов транзакций, соответствует примерно 50 МБ для хранения pg_xact и около 2 ГБ для хранения pg_commit_ts).

Сохраняются статусы и подтранзакций. При фиксации или откате транзакции верхнего уровня статусы подтранзакций (два бита на каждую) также записываются в подкаталог pg_xact. При прерывании транзакции верхнего уровня все её подтранзакции также прерываются.

Предустановленные параметры

Параметры контекста `internal`

- 18 параметров в 16 версии
- только для чтения

```
postgres=# select name,setting||coalesce(unit, '') setting, trim(short_desc, 'Shows ') desc
postgres-# from pg_settings where context='internal' order by 1;
              name               |  setting  |                                                  desc
---------------------------------+-----------+---------------------------------------------------------
 block_size                      | 8192      | the size of a disk block.
 data_checksums                  | off       | ether data checksums are turned on for this cluster.
 data_directory_mode             | 0750      | the mode of the data directory.
 debug_assertions                | off       | ether the running server has assertion checks enabled.
 in_hot_standby                  | off       | ether hot standby is currently active.
 integer_datetimes               | on        | ether datetimes are integer based.
 max_function_args               | 100       | the maximum number of function arguments.
 max_identifier_length           | 63        | the maximum identifier length.
 max_index_keys                  | 32        | the maximum number of index keys.
 segment_size                    | 1310720kB | the number of pages per disk file.
 server_encoding                 | UTF8      | the server (database) character set encoding.
 server_version                  | 16.1      | the server version.
 server_version_num              | 160001    | the server version as an integer.
 shared_memory_size              | 145MB     | the size of the server's main shared memory area (rounded
 shared_memory_size_in_huge_pages | 73        | the number of huge pages needed for the main shared memory
 ssl_library                     | OpenSSL   | the name of the SSL library.
 wal_block_size                  | 8192      | the block size in the write ahead log.
 wal_segment_size                | 16777216B | the size of write ahead log segments.
(18 строк)
```

В 16 версии есть 18 параметров, значения которых нельзя поменять. Они не устанавливаются в файлах конфигурации и доступны только для чтения.

Часть параметров задана при сборке и устанавливает ограничения (лимиты) постгрес. 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/15_4/se/runtime-config-preset.html

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`
- `initdb -k` или `--data-checksums` задает подсчет контрольных сумм в страницах файлов данных
- 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`

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. В СУБД Тантор подсчет контрольных сумм включён по умолчанию, если кластер создается утилитой установки. Если утилита `initdb` запускается вручную, то она работает как в PostgreSQL и подсчет кластер создаётся без установки подсчета контрольных сумм.

Включить, выключить или проверить контрольные суммы файлов можно утилитой `pg_checksums`. 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: 0

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.

Зачем нужно знать эти параметры? 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/15_2/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/se-16/data" has invalid permissions
DETAIL:  Permissions should be u=rwx (0700) or u=rwx,g=rx (0750).
stopped waiting
```

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 (rwx --- ---)`.

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-16/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 которыми 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)
```

Историческая справка: в PostgreSQL 10 версии было одно допустимое значение 0700. До 10 версии ограничений не было. В 11 версии добавили 0750.

Размер блока данных

- размер 8192 байт = 8Кб задан при компиляции
- синоним “страница”
- при нахождении в памяти занимает буфер в буферном кэше

По умолчанию размер страницы (блока данных) составляет 8 килобайт (или 8192 байт). Размер блока данных задан при компиляции и в 16 версии не может быть изменён без перекомпиляции программного обеспечения. Он определяется макросом `BLCKSZ` который по умолчанию установлен в 8Кб (8192 байт) в файле `/opt/tantor/db/16/include/pg_config.h`

Узнать размер блока можно

```
pg_controldata | grep 'block size'
Database block size:                  8192
WAL block size:                       8192
```

или параметром конфигурации `block_size`

Размер блока данных определяет лимиты постгрес.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/limits.html

Ограничения

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

Размер блока данных 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 таблице будет превышать 32Тб, стоит использовать секционированные (partitioned) таблицы.

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.

```
cat pg_config_manual.h | grep FUNC_MAX_ARGS
#define FUNC_MAX_ARGS               100
```

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 буер при очередном увеличении своего размера выходит за этот лимит, то выдаётся ошибка: "`Cannot enlarge string buffer`".

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

Максимальная длина идентификаторов (например, имен 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. Если вам необходимо использовать более длинные идентификаторы, вам следует переосмыслить ваш дизайн базы данных

==========

идентификаторы превышающие этот размер 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 константой NAMEDATALEN-1 установленной при компиляции. Значение константы показывает параметр max_identifier_length

```
postgres=# show max_identifier_length;
 max_identifier_length
-----------------------
 63
(1 row)
```

Есть и другие ограничения, например, максимальное количество аргументов функции 100, параметров в запросе 65535.

Конфигурационные параметры

Конфигурационные параметры

- установлены при сборке и не меняются
- можно посмотреть:
- утилитой командной строки `pg_config`
- в представлении `pg_config`
- функцией `pg_config()`
- Параметр `SHAREDIR` определяет директорию с файлами расширений
- `PKGLIBDIR` указывает на директорию разделяемых библиотек по умолчанию
- `BINDIR` определяет директорию с исполняемыми файлами

"Конфигурационные параметры" (config) и "параметры конфигурации" (settings) созвучны, но это разные понятия.

Часть параметров задаётся при сборке (компиляции, линковке). Посмотреть большую часть параметров, заданных при сборке:

1) Утилитой командной строки `pg_config`
2) Функцией `select * from pg_config();`

Например, `SHAREDIR` определяет директорию с файлами расширений.

Вопрос, который может возникнуть: установил расширение, хочу посмотреть что в него входит. Самое простое найти текстовые файлы расширений и посмотреть в них команды и параметры. Где найти эти файлы? Ответ: управляющие файлы расширений лежат в в поддиректории `extension` директории `SHAREDIR`, путь к которой можно посмотреть командой: `/usr/share/postgresql/12/extension`.

Список управляющих файлов расширений:

```
ls $(pg_config --sharedir)/extension/*.control
```

Второй вопрос: загрузил разделяемую библиотеку, где лежит ее файл? Или - хочу загрузить библиотеку, куда скопировать её файл? Ответ: PKGLIBDIR указывает на директорию разделяемых библиотек по умолчанию (файлы с расширением .so). Библиотеки могут загружаться командой LOAD в сессии или параметром shared_preload_libraries.

```
pg_config --pkglibdir
/opt/tantor/db/16/lib/postgresql
```

`BINDIR` определяет директорию с исполняемыми файлами, путь которой добавляется в переменную окружения `PATH` пользователя `postgres` (файл `.bash_profile`) в процессе инсталляции `Tantor DB`.

```
cat .bash_profile
export PATH=$PATH
export PATH=/opt/tantor/db/16/bin:$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 `libpq`.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/libpq-pgservice.html

Демонстрация

- Просмотр параметров конфигурации

03. Демонстрация

Просмотр параметров конфигурации

1) Посмотрим названия столбцов в **представление** со списком конфигурационных параметров:

```
postgres=# \d pg_settings
                    Представление "pg_catalog.pg_settings"
     Столбец     |   Тип    | Правило сортировки | Допустимость NULL | По
-----------------+----------+--------------------+-------------------+---
 name            | text     |                    |                   |
 setting         | text     |                    |                   |
 unit            | text     |                    |                   |
 category        | text     |                    |                   |
 short_desc      | text     |                    |                   |
 extra_desc      | text     |                    |                   |
 context         | text     |                    |                   |
 vartype         | text     |                    |                   |
 source          | text     |                    |                   |
 min_val         | text     |                    |                   |
 max_val         | text     |                    |                   |
 enumvals        | text[]   |                    |                   |
 boot_val        | text     |                    |                   |
 reset_val       | text     |                    |                   |
 sourcefile      | text     |                    |                   |
 sourceline      | integer  |                    |                   |
 pending_restart | boolean  |                    |                   |
```

2) **Количество** параметров в текущей версии:

```
postgres=# select count(*) from pg_settings;
 count
-------
   404
(1 строка)
```

Около 400 параметров, включая параметры загруженных (параметр `shared_preload_libraries`) библиотек ("модулей").

3) Посмотрим какие типы значений параметров есть:

```
postgres=# select distinct unit, vartype from pg_settings order by unit;
 unit | vartype
------+---------
 8kB  | int64
 8kB  | integer
 B    | integer
 kB   | integer
 MB   | integer
 min  | integer
 ms   | integer
 ms   | real
 s    | integer
      | string
      | bool
      | enum
      | int64
      | integer
      | real
(15 строк)
```

4) Есть параметры с **единицами измерения** и без.
Посмотрим сколько параметров каждого типа есть:

```
postgres=# select unit, vartype, count(*) from pg_settings group by unit,
vartype order by 3;
 unit | vartype | count
------+---------+-------
 8kB  | int64   |     1
 ms   | real    |     2
 min  | integer |     3
 MB   | integer |     6
 B    | integer |     6
      | int64   |     8
 s    | integer |    10
 kB   | integer |    12
 8kB  | integer |    17
 ms   | integer |    23
      | real    |    24
      | enum    |    44
      | integer |    58
      | string  |    68
      | bool    |   122
(15 строк)
```

5) Посмотрим какие **два** параметра измеряются по умолчанию в **долях** миллисекунд:

```
postgres=# select name, setting from pg_settings where unit='ms' and
vartype='real';
             name              | setting
-------------------------------+---------
 autovacuum_vacuum_cost_delay | 2
 vacuum_cost_delay            | 0
(2 строки)
```

Это параметры, настраивающие задержку в работе процессов вакуумирования

6) Есть параметры типа enum. Посмотрим какие значения бывают для параметров этого типа:

```
postgres=# select distinct enumvals from pg_settings;
                          enumvals
-------------------------------------------------------------
{local,remote_write,remote_apply,on,off}
{md5,scram-sha-256}
{none,pl,all}
{pause,promote,shutdown}
{none,top,all}
{postgres,postgres_verbose,sql_standard,iso_8601}
{debug5,debug4,debug3,debug2,debug1,log,notice,warning,error}
{sysv,mmap}
{origin,replica,local}
{always,on,off}
{TLSv1,TLSv1.1,TLSv1.2,TLSv1.3}
{serializable,"repeatable read","read committed","read uncommitted"}
{disabled,debug5,debug4,debug3,debug2,debug1,log,notice,warning,error}
{auto,force_generic_plan,force_custom_plan}
{content,document}
{pglz,lz4}
{local0,local1,local2,local3,local4,local5,local6,local7}
{none,ddl,mod,all}
{none,top,all,verbose}
{text,xml,json,yaml}
{none,cache,snapshot}
{off,on,try}
{"",TLSv1,TLSv1.1,TLSv1.2,TLSv1.3}
```

```
{minimal,replica,logical}
{fsync,syncfs}
{auto,regress,on,off}
{shmem,file}
{safe_encoding,on,off}
{buffered,immediate}
{debug5,debug4,debug3,debug2,debug1,info,notice,warning,error,log,fatal,panic
}
{terse,default,verbose}
{debug5,debug4,debug3,debug2,debug1,info,notice,warning,log}
{base64,hex}
{posix,sysv,mmap}
{raw,text,json,yaml,xml}
{partition,on,off}
{fsync,fdatasync,open_sync,open_datasync}
{off,on,regress}
{pglz,lz4,zstd,on,off}
{escape,hex}
(41 строка)
```

Используются, в основном, английские слова, обозначающие названия технологий и алгоритмов. Например, алгоритмов сжатия `pglz,lz4,zstd`.

7) Какие **контексты** параметров есть:

```
postgres=# select distinct context from pg_settings;
      context
-------------------
 postmaster
 superuser-backend
 user
 internal
 backend
 sighup
 superuser
(7 строк)
```

Контекст указывает можно ли изменить значение пармаметра, если можно, то **каким образом**

8) Параметры расширений и библиотек имеют в названии **точку**.
Посмотрим параметр `plpgsql.variable_conflict`:

```
postgres=# show plpgsql.variable_conflict;
ERROR:  unrecognized configuration parameter "plpgsql.variable_conflict"
```

Параметр неизвестен. Неизвестные параметры можно устанавливать в `postgresql.conf`, но не командой `ALTER SYSTEM`.

9) Загрузим библиотеку расширения ("модуль"). Апострофы в строковых параметрах обязательны:

```
postgres=# load 'plpgsql';
LOAD
postgres=# show plpgsql.variable_conflict;
 plpgsql.variable_conflict
---------------------------
 error
(1 строка)
```

10) Посмотрим какие **параметры конфигурации** были зарегистрированы при загрузке модуля:

```
postgres=# show plpgsql.<TAB><TAB>
```

```
plpgsql.check_asserts        plpgsql.extra_errors
plpgsql.extra_warnings       plpgsql.print_strict_params
plpgsql.variable_conflict
```

также можно посмотреть командой:

```
postgres=# \dconfig plpgsql.*
     Список параметров конфигурации
          Параметр           | Значение
-----------------------------+----------
 plpgsql.check_asserts       | on
 plpgsql.extra_errors        | none
 plpgsql.extra_warnings      | none
 plpgsql.print_strict_params | off
 plpgsql.variable_conflict   | error
(5 строк)
```

Практика

1) Обзор параметров конфигурации
2) Параметры конфигурации с единицей измерения
3) Параметры конфигурации логического типа
4) Конфигурационные параметры
5) Файл служб

04a Логическая структура кластера

Кластер баз данных

- Кластер это набор или объединение баз данных
- В кластере есть общие объекты для всех баз данных
- Приложение подсоединяется к одной базе данных и имеет доступ к её объектам
- Кластер создаётся утилитой командной строки initdb
- Кластер создаётся в физическом месте хранения - директории, обозначаемой "`PGDATA`" по названию переменной окружения для утилит
- Базы данных в кластере можно создавать и удалять

Определения понятия "кластер баз данных" в документации дано следующим образом.

Кластер (объединение) баз данных или сокращённо "кластер" это базы данных, имеющие общие глобальные SQL-объекты и их общих статических и динамических метаданных. Кластер баз данных создаётся утилитой командной строки initdb. SQL-объект - любой объект, который может быть создан командой CREATE языка SQL. Глобальные SQL-объекты это роли, табличные пространства, источники репликации, подписки логической репликации, базы данных. Локальные SQL-объекты это те, которые не глобальные. Базы данных - именованный набор локальных SQL-объектов.

Определение аккуратное, но кажется непонятным, хотя бы потому, что "общие статические", а особенно "динамические метаданные" в списке определений не даны.

Если вы работали с Oracle Database, то аналог базы данных PostgreSQL ("постгрес") это Pluggable Database (PDB). Аналог кластера - multitenant container database. Корневой базы (CDB Root) в постгрес нет, для управления кластером постгрес можно подключиться к любой из баз данных. Аналог Oracle Seed PDB - это база данных template0 или template1.

Рассмотрим понятие "базы данных" с другой стороны. Приложению нужно хранить данные, оно их хранит в виде таблиц и других объектов. Для хранения приложение создаёт соединения с местом хранения, которое можно назвать базой данных. Место, где могут совместно располагаться (храниться, находиться) объекты, с которыми можно работать одновременно (например, соединять таблицы в одной выборке), является для приложения логическим местом для хранения - "базой данных".

Чтобы создать место хранения, нужно запустить утилиту 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 (понятия подсоединиться к кластеру нет, одно соединение оно же сессия подсоединяется к одной базе данных) нужно чтобы на хосте (он же сервер или компьютер) 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.

В Oraacle Database есть созвучная технология Real Application Cluster (RAC) - набор одного или нескольких экземпляров, обслуживающих CDB или non-CDB. RAC не является аналогом понятия "кластер баз данных" постгрес. В постгрес один кластер обслуживает один экземпляр (instance).

https://docs.tantorlabs.ru/tdb/ru/15_4/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 (instance) кластера баз данных постгрес - 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 постгрес то же самое, что экземпляр single-instance Oracle Database.

Детализируем: экземпляр это один процесс postgres (postmaster) набор серверных (обслуживающих, backend, foreground), вспомогательных (фоновых) процессов, которые используют разделяемую память чтобы обмениваться данными между собой и достигать синергии совместно используя структуры памяти которые располагаются в общей области памяти. На одном сервере могут работать несколько экземпляров СУБД, если не будет конфликта по номеру порта, в том числе в имени файла Unix-сокета.

Что за "порт"? Это число, которые 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 постгрес может обозначать много понятий, например семейство СУБД, к которому относится СУБД Тантор. СУБД Тантор - ответвление (fork) свободно распространяемого PostgreSQL, как и остальные forks: Enterprise DB, Postgres Pro Enterprise.

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

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
           |          |          |                 |             |
(3 rows)
```

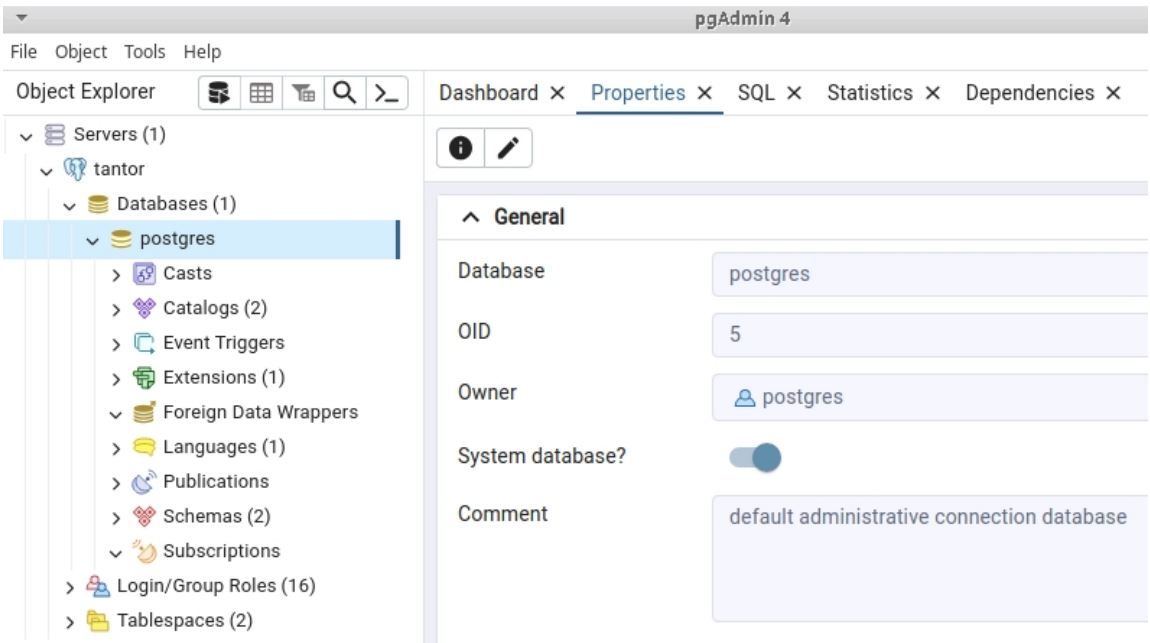
- `SELECT datname FROM pg_database;`

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;`

или графическими утилитами, например pgAdmin.



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

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 хочется создавать базы данных из командной строки.

В команде имеется около 15 параметров. Можно обратить внимание на следующие параметры:

OWNER можно указать имя роли, которая будет обладать привилегиями схожими с суперпользователем внутри этой базы данных.

Чтобы сделать роль владельцем быть суперпользователем или входить в эту роль прямо или косвенно. По умолчанию создатель становится владельцем базы.

`TEMPLATE` - имя базы содержимое которой вы скопируете. Это любая база, не обязательно имеющая свойство `IS_TEMPLATE`. По умолчанию используется template1. Но если вы захотите создать базу с параметрами локализации отличными от тех, что указаны 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 владелец. Также база со свойством шаблона не может быть удалена. Для удаления сначала 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 не меняются.

https://docs.tantorlabs.ru/tdb/ru/15_4/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

`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 постгрес и сразу по умолчанию стал использовать новую стратегию `WAL_LOG` при которой составляется список объектов и весь объем клонируемой базы проходит через журналы предзаписи. Причина появления новой стратегии в том, что прежняя `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 несущественно). Причина не только в одномоментной 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 изменяемы блок записывается в журнал полностью (8Кб) так как по умолчанию параметр `full_page_writes=on` (а отключать его небезопасно). Но если размер шаблонной базы больше значения параметра `max_wal_size`, то контрольная точка тоже будет выполнена и может быть даже неоднократно если размер базы больше значения параметра в несколько раз.

Если размер шаблона небольшой, например, меньше 16Мб (размер WAL-сегмента) или в несколько раз больше, то можно создавать базу данных. Если 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 например половины от `max_wal_size`, то `FILE_COPY` предпочтительнее.

Можно дать описание созданной 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/15_4/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

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 суперпользователь. Переименовать базу к которой подключены нельзя, нужно подключиться к любой другой базе.

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. 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). Знание о транзакционности команд имеет смысл, например, при установке или разработке расширений. Нетранзакционные команды не могут выполняться при установки расширения в базу данных.

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

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

- 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

Следующий объект, который мы рассмотрим - схемы. 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. 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/15_4/se1c/ddl-schemas.html

Создание и изменение схем

- Схемы не связаны с ролями, но имеют владельца
- При создании схемы можно назначить роль-владельца:
  `CREATE SCHEMA` имя_схемы `AUTHORIZATION` владелец;
- Можно сменить владельца:
  `ALTER SCHEMA` имя_схемы `OWNER TO` роль;
- Можно переименовать схему:
  `ALTER SCHEMA` имя_схемы `RENAME TO` имя;
- На схемы можно выдавать привилегии `CREATE`, `USAGE` ролям

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 роль;
```

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 постгрес аналогов "синонимов" нет, так как они не нужны.

На схемы можно выдавать привилегии CREATE и/или USAGE ролям. 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

- задаёт список схем, в которых ищется объект
- устанавливается параметром search_path который может быть изменён на любом уровне
- значение по умолчанию

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 перечисленных в пути поиска схем. Если схема не существует или на неё нет прав, то ищется в следующих по порядку схемах и 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, 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

В постгрес 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 сборки Tantor SE1C позволяет уменьшить проблемы при работе большим количеством 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

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()`. В PostgreSQL после имени функции без аргументов обязательно ставить круглые скобки "`()`". Однако для части функций, описанных в стандарте SQL к которым относится эта функция их ставить не обязательно, потому что в стандарте SQL круглые скобки опциональны. 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

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

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 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/15_4/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, схема системного каталога неявно присутствуют в начале пути поиска и создание временного объекта маскирует любую таблицу, представление, последовательность

```
postgres=# \dt pg_authid
            List of relations
   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_4,pg_catalog,public}
(1 row)
```

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.

Также, нужно знать, что 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/15_4/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 - 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`.

В Oracle Database аналог системного каталога 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/15_4/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:

1) базы данных
2) роли
3) табличные пространства
4) подписки логической репликации
5) источники логической репликации

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. В 16 версии СУБД Тантор 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 Тантор
- \? справка по командам psql
- Вносить изменения в таблицы системного каталога если, это не описано в документации нежелательно
- Зная архитектуру постгрес, понятия, термины вы сможете легко получать информацию командами psql

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 \dv**S**

Суффикс **S** в конце команд psql позволяет выдавать содержимое системного каталога, которые обычно по умолчанию не выдаются.

Более практична работа с 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 на которыми меняется поведение psql
\h [ИМЯ]      справка по SQL-команде; * - по всем командам
```

Зная архитектуру постгрес, 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

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
```

Чтобы получить данные из таблиц системного каталога может потребоваться соединить несколько таблиц. Строки таблиц системного каталога связаны через столбец oid, то есть число. В постгрес можно создавать типы данных (`CREATE TYPE`) приведения типов (`CREATE CAST`) . Этим пользуются и разработчики постгрес. 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 36 таблиц системного каталога, у которых они есть.

Часто используемые команды psql

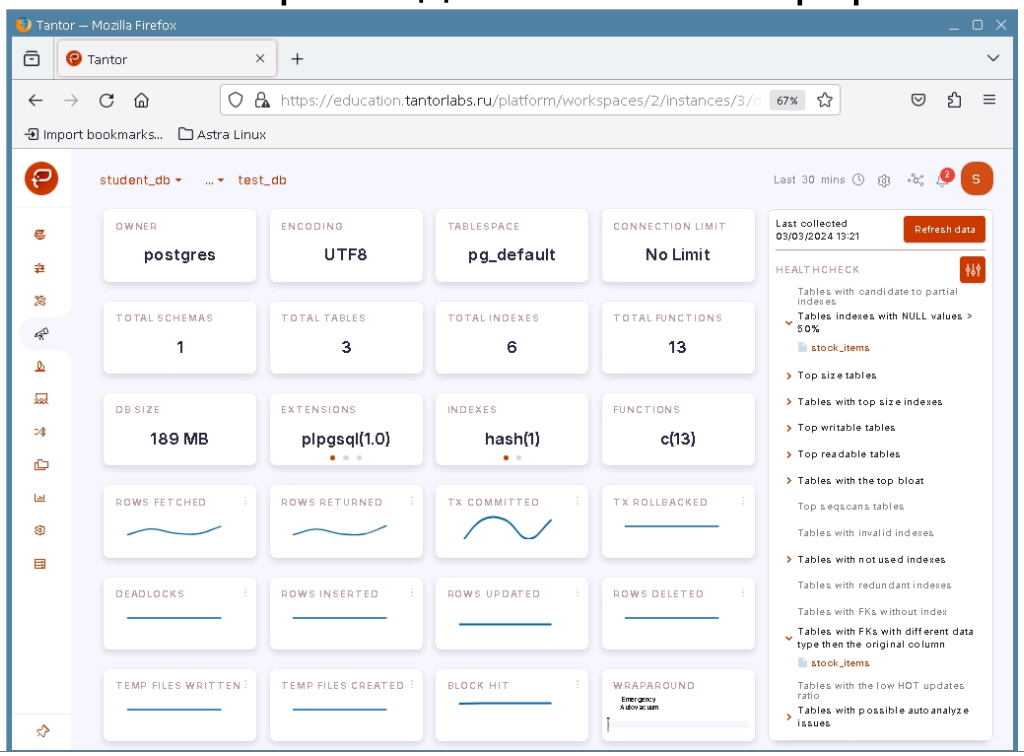
- `\l` - список баз данных
- `\du` `\dg` - список ролей кластера
- `\dn` - список схем базы
- `\db` - список табличных пространств
- `\dconfig *`имя`*` - список параметров конфигурации
- `\dfS pg*` - список системных функций и процедур, полезных для администрирования. Часть информации о работе экземпляра 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

l - список (**l**ist) баз данных
\d**u** или \d**g** - список ролей (**u**ser, **g**roup) кластера, \drg - назначения ролей ролям
\dn - список схем базы (**n**amespace)
\db - список табличных пространств
\d**config** *имя* - список параметров конфигурации (**config**) кластера
\ddp - список привилегий по умолчанию (**d**efault **p**rivileges). Это особый тип привилегий или отзывающих привилегий, специфичный для постгрес.
\d**f**S pg* - список системных функций (**f**unction) 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 постгрес позже функций, поэтому "f" используется и для процедур.
\d**vS** pg* - полезные системные (**S**ystem) представления (**v**iew)
\d**x** - список установленных расширений (e**x**tention)
\dy - список триггеров на события, которые обычно создают расширения или администраторы
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/15_4/se/**functions-admin**.html

Инспектор баз данных в Платформе



Инспектор Платформы показывает информацию о содержимом базы данных - количество и существенные для администрирования характеристики SQL-объектов. Также анализирует и выдает рекомендации по характеристикам объектов, которые могут являться проблемой.

Демонстрация

- Просмотр списка баз данных кластера
- Создание базы данных
- Переименование базы данных
- Ограничение на соединение с базой
- форматирование вывода psql

04a. Демонстрация

Логическая структура

1) Запустим стандартно поставляемую утилиту `oid2name`:

```
postgres@tantor:~$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
      5       postgres  pg_default
      4      template0  pg_default
      1      template1  pg_default
  16439        test_db  pg_default
```

Утилита, запущенная без параметров выдаёт список баз данных; название табличного пространства по умолчанию для каждой из баз данных; **oid** базы данных, который соответствует поддиректории в директории табличного пространства.

2) Подключимся к экземпляру:

```
postgres@tantor:~$ psql
psql (16.1)
Введите "help", чтобы получить справку.
```

3) Посмотрим список баз командой **\l**:

```
postgres=# \l
                                                 List of databases
   Name    |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    | ICU Locale | ICU Rules |   Access privileges
-----------+----------+----------+-----------------+-------------+-------------+------------+-----------+-----------------------
 postgres  | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |            |           |
 template0 | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |            |           | =c/postgres          +
           |          |          |                 |             |             |            |           | postgres=CTc/postgres
 template1 | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |            |           | =c/postgres          +
           |          |          |                 |             |             |            |           | postgres=CTc/postgres
 test_db   | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |            |           |
(4 строки)
```

4) Посмотрим что нам выдаст команда **\l** если добавить символ "+" означающий дополнительные данные:

```
postgres=# \l+
                                                                                List of databases
   Name    |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    | ICU Locale | ICU Rules |   Access privileges   |  Size   | Tablespace |                Description
-----------+----------+----------+-----------------+-------------+-------------+------------+-----------+-----------------------+---------+------------+--------------------------------------------
 postgres  | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |            |           |                       | 1757 MB | pg_default | default administrative connection database
 template0 | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |            |           | =c/postgres          +| 7337 kB | pg_default | unmodifiable empty database
           |          |          |                 |             |             |            |           | postgres=CTc/postgres |         |            |
 template1 | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |            |           | =c/postgres          +| 7401 kB | pg_default | default template for new databases
           |          |          |                 |             |             |            |           | postgres=CTc/postgres |         |            |
 test_db   | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |            |           |                       | 946 MB  | pg_default |
(4 rows)
```

Что добавилось?
Добавились столбцы с размером, табличным пространством по умолчанию, описанием.

5) Посмотрим список баз данных командой `SELECT`:

```
postgres=# SELECT datname FROM pg_database;
  datname
-----------
 postgres
 test_db
 template1
 template0
(4 rows)
```

6) Создадим базу данных командой SQL:

```
postgres=# CREATE DATABASE db01;
CREATE DATABASE
```

Показать, что можно стрелкой вверх на клавиатуре повторить предыдущие команды и убедиться, что новая база данных выводится.
На основе какой шаблонной базы была создана база данных `db01`?
На основе `template1`.

7) Переименуем базу:

```
postgres=# ALTER DATABASE db01 RENAME TO db02;
ALTER DATABASE
```

8) Убедимся что можем подсоединиться к базе db02:

```
postgres=# \c db02
Вы подключены к базе данных "db02" как пользователь "postgres".
db02=# \c postgres
Вы подключены к базе данных "postgres" как пользователь "postgres".
```

Помните о том, что нажимая клавишу табуляции <TAB> можно завешать команды.

9) Установим максимальное число подсоединений в ноль:

```
postgres=# ALTER DATABASE db02 CONNECTION LIMIT 0;
ALTER DATABASE
```

10) как пользователь с атрибутом `SUPERUSER` мы можем подсоединиться:

```
postgres=# \c db02
Вы подключены к базе данных "db02" как пользователь "postgres".
db02=# \c postgres
Вы подключены к базе данных "postgres" как пользователь "postgres".
```

11) воспользуемся свойством базы данных `ALLOW_CONNECTIONS`:

```
postgres=# ALTER DATABASE db02 ALLOW_CONNECTIONS false;
ALTER DATABASE
postgres=# \c db02
подключиться к серверу через сокет "/var/run/postgresql/.s.PGSQL.5432" не
удалось: FATAL:  database "db02" is not currently accepting connections
Сохранено предыдущее подключение
```

теперь мы не можем подсоединиться.

12) Посмотрим содержимое файла параметров `psql`:

```
postgres=# \! cat .psqlrc
\setenv PAGER 'less -XS'
\setenv PSQL_EDITOR /usr/bin/mcedit
```

13) В виртуальной машине для курса установили эти параметры. По умолчанию файл отсутсвует. Установим значение переменной в значение по умолчанию и посмотрим как будет выдаваться результат:

```
postgres=# \setenv PAGER 'more'
```

14) посмотрим список встроенных функций, которые полезны для администрирования:

```
postgres=# \dfS pg*
```

результат нечитаемый

15) настроим вывод и повторим:

```
postgres=# \pset format wrapped
postgres=# \dfS pg*
                                                                 List of functions
   Schema   |                     Name                     |     Result data type     |
Argument data types            | Type
------------+----------------------------------------------+--------------------------+----------
--------------------------------+------
 pg_catalog | pg_advisory_lock                             | void                     | bigint
| func
 pg_catalog | pg_advisory_lock                             | void                     | integer,
integer                        | func
 pg_catalog | pg_advisory_lock_shared                      | void                     | bigint
| func
 pg_catalog | pg_advisory_lock_shared                      | void                     | integer,
integer                        | func
```

Отображение поменялось. При использовании утилиты more клавишами <PgUp> и <PgDown> нельзя пользоваться.

16) Вернем формат в значение по умолчанию:

```
postgres=# \pset format aligned
Формат вывода: aligned.
```

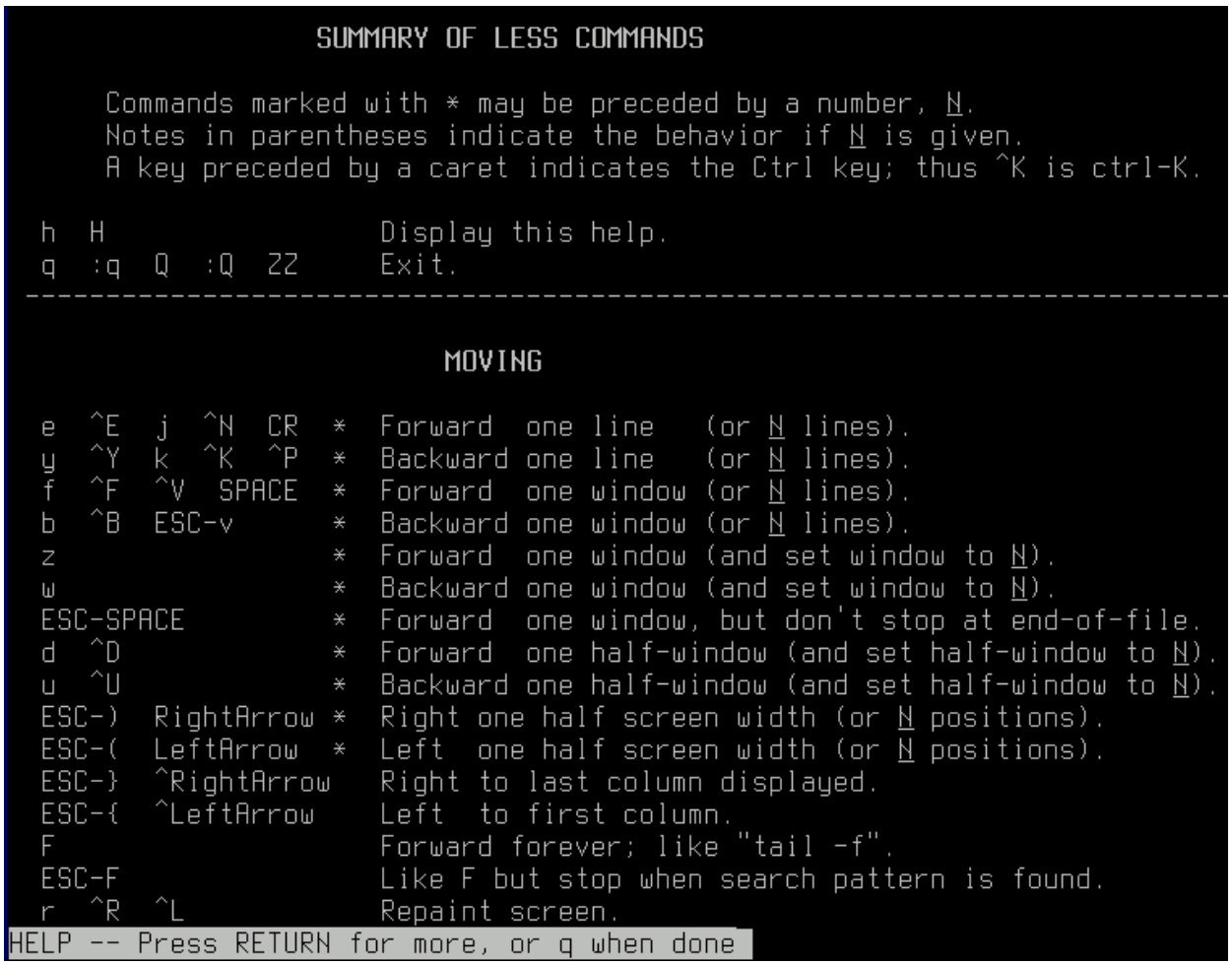
Вернем переменную, задающую программу постраничного вывода вместо использовавшейся утилиты "more":

```
postgres=# \setenv PAGER 'less -XS'
```

17) Повторим и убедимся что вывод стал читаемым можно использовать клавиши <PgUp> и <PgDown>:

```
postgres=# \dfS pg*
```

Нажмите клавиши <PgDn> клавишу <h> обратите внимание на то что высветилась подсказка по команде less, прочтите что для выхода из режима помощи можно нажать клавишу "q" и нажмите ее два раза<q><q>



18) Удалите созданную базу данных:

```
postgres=# drop database db02;
DROP DATABASE
```

Практика

1) Установка параметров конфигурации на различных уровнях
2) Установка пути поиска в функциях и процедурах

04b Физическая структура кластера

Директория файлов кластера

- 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

Файлы кластера баз данных хранятся в директории, которую называют `PGDATA` по имени переменной окружения операционной системы, которую обычно 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. Некоторые утилиты (`pg_resetwal`) в целях избежать случайный запуск с неправильно установленной переменной окружения требуют явного указания этого параметра.

Кластер может хранить файлы данных вне директории `PGDATA` с помощью "табличных пространств", которые мы рассмотрим дальше в этой главе.

По умолчанию инсталлятор СУБД Тантор создает директорию
`/var/lib/postgresql/tantor-se-16/data`
для хранения файлов кластера и файл служб
`/usr/lib/systemd/system/tantor-se-server-16.service`,
где указывает путь к этой директории. Параметрами **--edition** и **--major-version** инсталлятору можно задать другие значения. Остальные утилиты и программное обеспечение СУБД Тантор значений по умолчанию не имеют, так как на хосте могут существовать несколько равноправных кластеров. У каждого кластера своя директория `PGDATA`. Каждый кластер обслуживается одним экземпляром.

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. Для работы с файлами данных этот режим не стоит использовать.

PostgreSQL не дублирует (не мультиплексирует) файлы кластера. Отказоустойчивость работы с файлами должна обеспечиваться на более низких уровнях - файловой системы, оборудования.

PostgreSQL пользуется функционалом символических и жестких ссылок файловых систем. При администрировании `PGDATA` можно использовать точки монтирования, символические и жесткие ссылки.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/storage-file-layout.html

Директория PGDATA

- содержит файлы параметров
- поддиректории

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 Линукс. 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, остальные директории

https://postgrespro.ru/docs/postgresql/16/kernel-resources

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)

При работе экземпляра данными, не помещающимися в память процесса создаются временные файлы. Это файлы для временные таблицы, индексов на временные таблицы, а также файлы, создаваемые при выполнении в командах SQL сортировок, соединений наборов данных, создания индексов, если не хватит памяти процесса. Память ограничивается параметрами `work_mem` и `maintenance_work_mem`.

Временные файлы автоматически создаются и удаляются в табличных пространствах, имена которых перечислены в параметре конфигурации **`temp_tablespaces`**. По умолчанию пусто, тогда используется табличное пространство по умолчанию той базы данных, с которой работает процесс экземпляра.

Если в параметре **`temp_tablespaces`** указано несколько табличных пространств, процесс выбирает табличное пространство случайным образом при каждом создании временного объекта. Если объекты создаются в рамках транзакции, то для уменьшения задержек (на получение случайного значения) табличные пространства перебираются по кругу. Имена несуществующих табличных пространств или тех, которые не могут использоваться (нет привилегий) игнорируются и не вызывают ошибки.

Временные файлы для обслуживания команд SQL (соединения, сортировки и другие) создаются в поддиректории `pgsql_tmp` табличного пространства используемого серверным процессом для создания временных объектов. Имя временного файла pgsql_tmpPPP.NNN, где PPP - PID серверного процесса, а NNN число, уникальное в пределах одного процесса. Пример:

```
pg_tblspc/16385/PG_17.1_202501011/pgsql_tmp/pgsql_tmp12345.4
```

Поскольку временные файлы могут достигать больших размеров, их появление может быть нежелательным в табличных пространствах с постоянно хранимыми данными. Из-за больших объемов записи во временные файлы использование систем хранения на основе магнитных дисков (HDD) возможно будет предпочтительнее массивов хранения на основе чипов памяти (SDD), так как ресурс последних определяется объемом записываемых данных.

Временные объекты

- Параметр `temp_file_limit` используется для ограничения размера временных файлов используемых одним процессом для обслуживания команд, размер временных таблиц не ограничивает
- При создании временного объекта (таблицы, индекса на неё) создаются строки в таблицах системного каталога
- С временной таблицей может работать только серверный процесс, параллельные процессы с ними не работают
- При частом удалении и усечении временных таблиц могут может появляться много устаревших версий строк в таблицах системного каталога (pg_class, pg_attribute)

Ограничение на размер временных файлов используемых одним процессом можно установить параметром **`temp_file_limit`**. По умолчанию ограничение не установлено. Ограничивается объём временных файлов, которые создаются неявно при выполнении команд. При превышении ограничения команда прервётся. Этот параметр не ограничивает объем файлов явно (командой `CREATE TEMPORARY TABLE`) созданных временных таблиц.

Временные таблицы могут активно использоваться некоторыми 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 таблице `pg_attribute`. Таблицы системного каталога могут разрастись до десятков гигабайт. Для таких приложений можно использовать сборки, в которых имеются оптимизации работы с временными таблицами, например, Tantor SE1C.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/runtime-config-client.html#GUC-TEMP-TABLESPACES

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 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 ролям. 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

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/15_4/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 являются неотъемлемой частью кластера баз данных. Даже если они находятся не в PGDATA, они 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;`

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/15_4/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

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 с названием значения **`oid`** `(число)` табличного пространства. Эта 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:

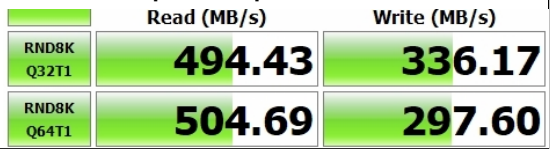
```
pg_controldata | grep down
Database cluster state: shut down
```

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-16`
9)Проверьте, что местоположение поменялось. Например, командой psql `\db`

Процедура появилась в версии 9.2 PostgreSQL.

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 `[, ... ] );`



В текущей версии СУБД Тантор 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 команд. Параметры представляют собой весовые коэффициенты, которые используются планировщиком для определения стоимости плана выполнения. Параметры влияют на оценку планировщика какой ресурс "дороже" дисковая подсистема или вычислительные мощности процессоров. Это может быть полезно, если табличное пространство расположено на системе хранения, которая быстрее или медленнее системы хранения, параметры которой установлены в конфигурационных файлах кластера. Одноимённые параметры конфигурации 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

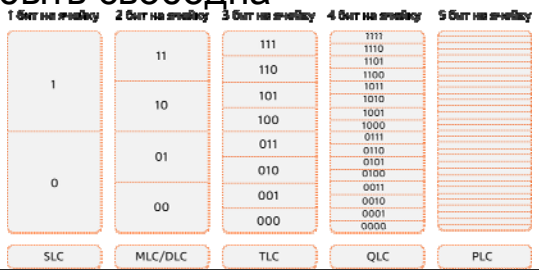
`random_page_cost (float)` - стоимость чтения блока с диска при произвольном доступе к блокам файлов. По умолчанию 4.0 Для SSD последовательный и произвольный доступ не отличаются по скорости, то есть random_page_cost можно сделать равным seq_page_cost. Уменьшение random_page_cost относительно seq_page_cost склоняет планировщик к методу доступа "Index Scan" вместо метода доступа "Seq Scan". Одновременное изменение значений обоих параметров меняет оценки стоимости дискового ввода-вывода относительно стоимости использования центральных процессоров.

`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 количество 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/15_4/se/runtime-config-resource.html

Работа с файлами журнала

- 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, ее можно смонтировать на отдельный раздел
- Рекомендуется включить сжатие образов страниц параметром `wal_compression`
- При использовании SSD большая часть раздела файловой системы должна быть свободна



В директории `PGDATA/pg_wal` создаются файлы журнала. 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 объёма 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 (tripple 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 discardom) 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. Пример ошибки, связанной с нехваткой места. Серверный процесс, который не смог записать в журнал данные 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
```

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` включенным зависит от алгоритма работы контроллера памяти SSD и файловой системы. Параметр `wal_init_zero` стоит отключить.

Параметр `wal_compression` по умолчанию отключен 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`.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/runtime-config-wal.html
https://wiki.archlinux.org/title/Solid_state_drive_(Русский)
https://en.wikipedia.org/wiki/Multi-level_cell

Основной слой

- 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

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 террабайт (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

- Карта видимости (`vm`)

- не создается для индексов
- заполняется 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

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.

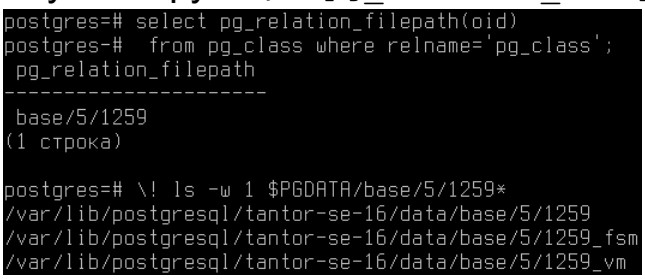
Для отношений (кроме индексов) создается слой "vm" (карта видимости и заморозки, visibility map). В файле этого слоя хранится по два бита на блок основного слоя таблицы. Единичка первого бита указывает, что в блоке основного слоя все строки самой последней версии (нет строк которые можно очистить). Этот 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/15_4/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)`



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 tBBB_FFF, где BBB - номер серверного процесса (B - Backend) обслуживающего сессию 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+` и функция `pg_tablespace_size()`
- размер баз данных `\l+` и функция `pg_database_size()`

```
postgres=# SELECT spcname, pg_size_pretty(pg_tablespace_size(oid)) FROM pg_tablespace;
 spcname   | pg_size_pretty
-----------+----------------
pg_default | 29 MB
pg_global  | 565 kB
2 строки)

postgres=# SELECT datname, pg_size_pretty(pg_database_size(oid)) FROM pg_database;
 datname  | pg_size_pretty
----------+----------------
postgres  | 7497 kB
test_db   | 7401 kB
template1 | 7561 kB
template0 | 7337 kB
4 строки)
```

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')` выдает 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

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 стоит смотреть в разделе документации по функциям для администрирования:
https://docs.tantorlabs.ru/tdb/ru/15_4/se/**functions-admin**.html

Перемещение объектов

- можно переместить файлы таблиц, индексов, материализованных представлений
- в новые файлы копируется содержимое файлов всех слоёв
- устанавливает монопольную блокировку несовместимую с `SELECT`
- Команда `ALTER` тип `ALL IN TABLESPACE` перемещает все объекты одного типа в текущей базе данных

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. На время перемещения используется место в директории табличного пространства куда перемещаются объекты. 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. Команды SELECT могут работать долго. 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.

Существует параметр lock_timeout которым можно задать максимальное время ожидания получения явной или неявной блокировки любой командой. 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 блокировокt. Этот параметр задает 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];`

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;

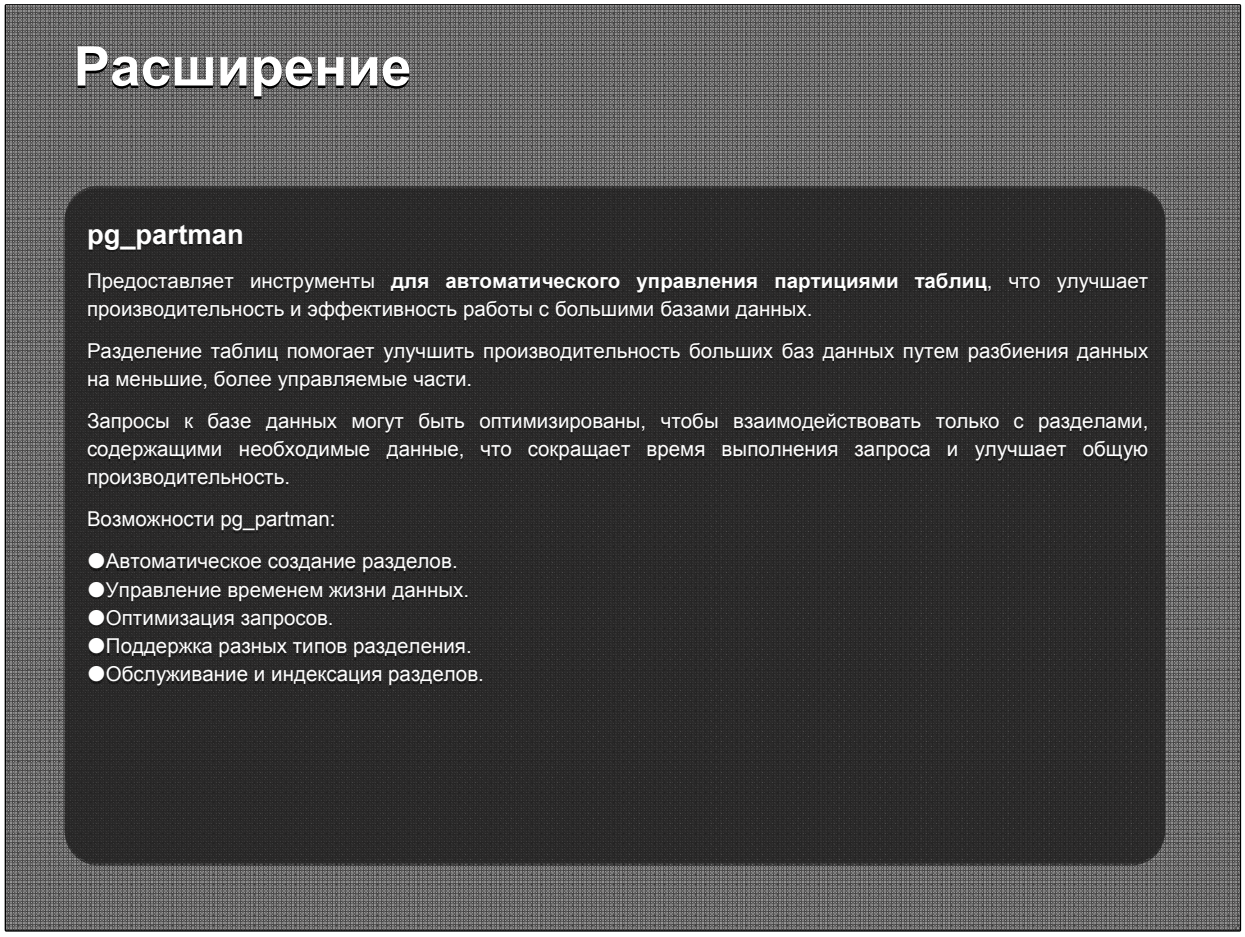
Эти команды не комбинируются друг с другом, выполняются по отдельности и требуют монопольной блокировки, но на короткое время.

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.

Расширения

Расширения	Tantor SE	Tantor SE 1C	Tantor BE	PostgreSQL
pg_partman	✓		✓	
pg_repack	✓	✓	✓	
pgcompacttable	✓	✓	✓	
ORC	✓			
		✓		



pg_partman предоставляет набор инструментов для автоматического управления секциями (partitions) таблиц, что улучшает производительность и эффективность работы с большими базами данных.

Разделение таблиц помогает улучшить производительность больших баз данных путем разбиения данных на меньшие, более управляемые части.

Запросы к базе данных могут быть оптимизированы, чтобы взаимодействовать только с теми разделами, которые содержат необходимые данные, что сокращает время выполнения запроса и улучшает общую производительность.

Преимущества pg_partman:

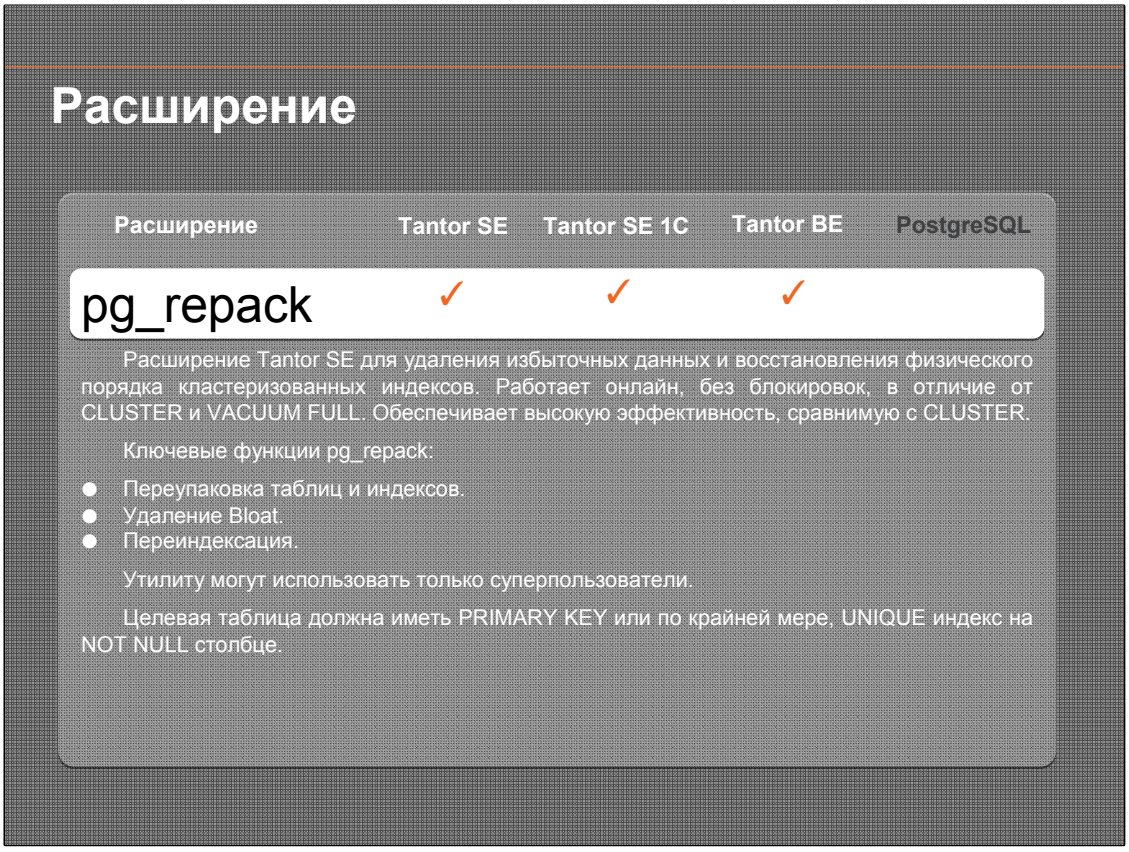
Автоматическое создание разделов: может автоматически создавать новые разделы на основе конфигурации, которую вы задаете.

Управление временем жизни данных: вы можете настроить, сколько времени данные должны оставаться в каждом разделе перед их удалением или перемещением в другое место.

Оптимизация запросов: помогает оптимизировать запросы к базе данных, позволяя СУБД Tantor быстрее находить и получать данные, так как они хранятся в меньших и более управляемых секциях.

Поддержка разных типов разделения: поддерживает несколько типов разделения, включая разделение по диапазону и разделение по значению.

Обслуживание и индексация разделов: предоставляет инструменты для индексации и обслуживания разделов, что способствует их эффективной работе.



Во всех версиях СУБД Tantor доступен pg_repack.

pg_repack - это расширение Tantor SE, которое позволяет удалять избыточные данные из таблиц и индексов, а также восстанавливать физический порядок кластеризованных индексов. В отличие от CLUSTER и VACUUM FULL, оно работает онлайн, не накладывая исключающую блокировку на обрабатываемые таблицы. pg_repack также эффективен, и производительность сравнима с использованием CLUSTER напрямую.

Ключевые функции pg_repack:

Переупаковка таблиц и индексов: удаляет мертвые строки и перестраивает таблицы и индексы для освобождения занятого ими пространства и улучшения производительности. Делает это без блокировки чтения или записи данных, что позволяет продолжать работу приложений без прерываний.

Удаление Bloat: <Bloat> - это ситуация, при которой пространство, занятое мертвыми строками, не возвращается обратно в систему. pg_repack может помочь устранить <bloat>.

Переиндексация: pg_repack также может перестроить индексы без блокировки, что повышает их эффективность и уменьшает время ответа на запросы.

Только суперпользователи могут использовать утилиту. Целевая таблица должна иметь PRIMARY KEY или по крайней мере, UNIQUE индекс на NOT NULL столбце.

Реорганизация и перемещение объектов pg_repack

- В СУБД Тантор есть расширение pg_repack
- Расширение нужно установить в базах данных
- Реорганизация файлов объектов с перемещением в другое табличное пространство или без запускается утилитой командной строки pg_repack
- В процессе реорганизации устанавливается блокировка ACCESS SHARE позволяющая выполнять команды VACUUM, ANALYZE, SELECT, DML
- В конце реорганизации на короткое время устанавливается монопольная блокировка

В СУБД Тантор всех сборок есть расширение pg_repack с помощью которого можно переместить объекты в другое табличное пространство не устанавливая на время работы монопольную блокировку на объекты. Вместо нее устанавливается блокировка самого щадящего уровня ACCESS SHARE. Такую блокировку устанавливают команды SELECT.

В конце перемещения на короткое время устанавливается монопольная блокировка. 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, он отсоединит серверные процессы функцией pg_terminate_backend().

Перемещение объектов в другое табличное пространство не основная задача pg_pepack, эта утилита реорганизует файлы объектов, делая структуру более компактной.

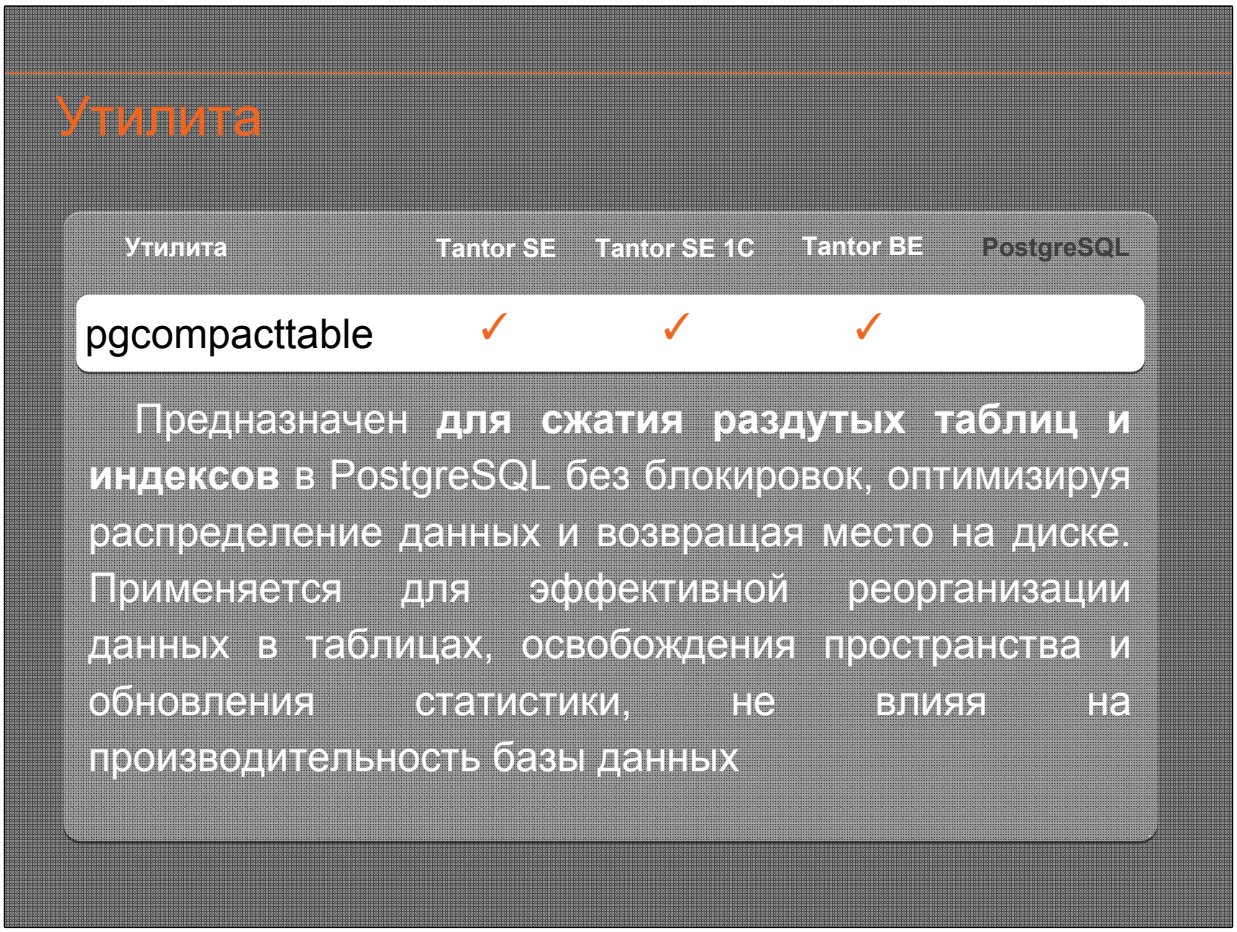
Можно задать количество параллельных сессий параметром --jobs, чтобы одновременно перестраивать несколько индексов на одной таблице в режиме полной реорганизации таблицы.

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. Весь объем переносимых данных проходит через WAL-журналы.

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/15_4/**be**/pg_repack.html



Инструмент для уменьшения размера раздутых таблиц и индексов без тяжелых блокировок. Предназначен для реорганизации данных в таблицах и перестроения индексов, чтобы вернуть место на диске без влияния на производительность базы данных.

Когда таблица в PostgreSQL получает много обновлений, удалений или вставок, она может стать <раздутой> и занимать больше места на диске.

pgcompacttable помогает компактно хранить данные таблицы, удалив неиспользуемые или устаревшие записи, обновляя статистику и упорядочивая физическое размещение данных.

Уменьшение размера файлов
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
- не уменьшает размеры файлов если установлен параметр old_snapshot_threshold

Утилита pgcompacttable поставлется с СУБД Тантор и находится в директории /opt/tantor/db/16/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.

4) при установленном параметре old_snapshot_threshold не может уменьшать размеры файлов, так же как и VACUUM не выполняет последнюю фазу vacuum_truncate. Это описано в документации к этому параметру.

Установка:

1)в базах данных нужно установить стандартное расширение pgstattuple:

```
CREATE EXTENSION pgstattuple;
```

2)Убрать параметр old_snapshot_threshold:

ALTER SYSTEM RESET old_snapshot_threshold;

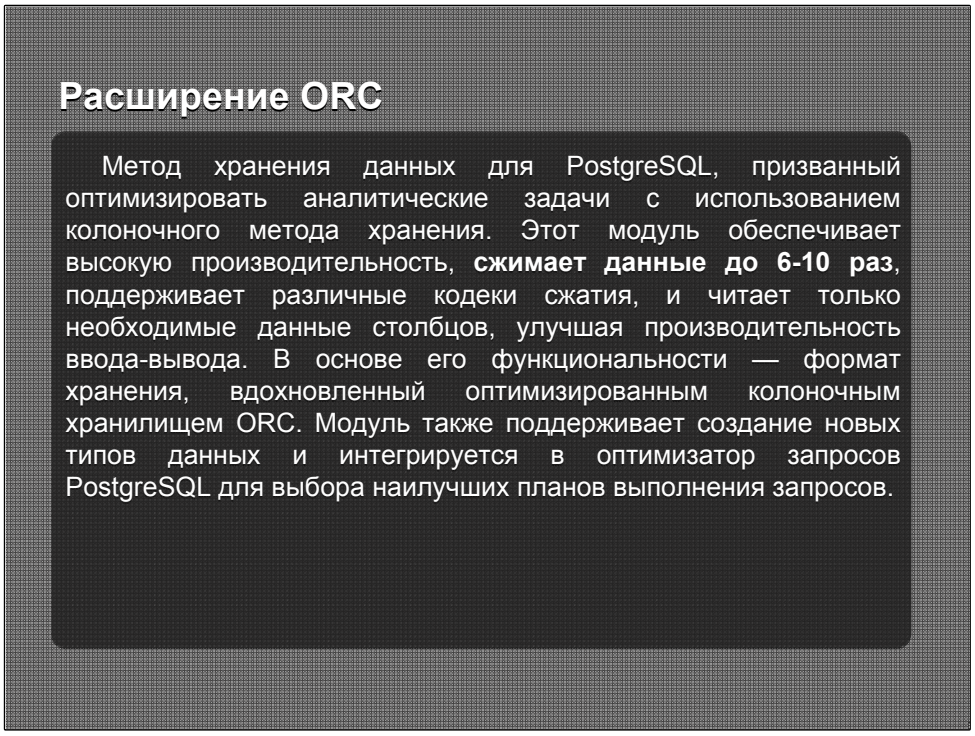
3)установить Perl: `apt-get install libdbi-perl libdbd-pg-perl`

или `yum install perl-Time-HiRes perl-DBI perl-DBD-Pg`

4)установить права на выполнение:

```
chmod 755 -R /opt/tantor/db/16/tools/pgcompacttable
```

https://docs.tantorlabs.ru/tdb/ru/15_4/se/pgcompacttable.html



Добавляет колоночный метод хранения данных с возможностью их сжатия для снижения объема ввода-вывода и достижения высокой производительности.

Подходит для append-only, например time series данных, и витрин корпоративных хранилищ

Колоночные хранилища предоставляют заметные преимущества для аналитических задач, где данные загружаются партиями.

Благодаря колоночной природе pg_columnar повышает производительность за счет чтения только соответствующих данных с диска, и оно может сжимать данные в 6-10 раз для сокращения требований к пространству для архивации данных.

Это расширение использует формат хранения данных, который вдохновлен ORC - форматом оптимизированного колоночного хранилища и предоставляет следующие преимущества:

Сжатие: сокращает объем данных в памяти и на диске на 2-4 раза. Может быть расширено для поддержки различных кодеков.

Проекции столбцов: Читает только данные столбцов, соответствующие запросу. Улучшает производительность для запросов, ограниченных вводом-выводом.

Skip indexes: хранит статистику минимального/максимального значения для групп строк и использует их для пропуска нерелевантных строк.

pg_columnar использует API оболочки внешних данных PostgreSQL, что позволяет хранить более 40 типов данных. Пользователь также может создавать новые типы и использовать их.

Оптимизатор запросов PostgreSQL использует данные pg_columnar для оценки различных планов запросов и выбора наилучшего.

Расширение ORC: введение

- 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

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. 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/15_4/se/storage-toast.html

ORC: 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: таблица `pg_am`

Идея колоночного способа хранения 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_internal.columnar_handler | t
```

Расширение создаёт схемы `columnar` и `columnar_internal` которые использует для хранения своих объектов.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/citus.html

ORC: особенности использования

- не поддерживаются `UPDATE` и `DELETE`
- поддерживаются команды: `TRUNCATE`, `INSERT` (в том числе одной строки), `COPY`
- параллельное сканирование не 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 псевдостолбцов `CTID`, `xmin`, `xmax`

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.

Формат columnar не поддерживает `UPDATE` и `DELETE`. Поддерживаются `TRUNCATE`, `INSERT` (в том числе одной строки), `COPY`. Это основное, что ограничивает применение этого способа хранения. При попытке выполнить неподдерживаемые команды выдастся ошибка:

```
DELETE FROM perf_columnar WHERE id=0;
ERROR:  UPDATE and CTID scans not supported for ColumnarScan
```

Псевдостолбца `CTID` в таблицах формата `columnar` нет.

TOAST с форматом `columnar` не используется, так как большие значения хранятся внутренне. Параллельное сканирование не реализовано - выборку выполняет один серверный процесс. 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.

ORC: настройки

- 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
- у расширения есть параметры с префиксом "`columnar.`"

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:

```
columnar.chunk_group_row_limit, columnar.compression_level,
columnar.stripe_row_limit, columnar.compression,
columnar.planner_debug_level
```

Можно устанавливать параметры на уровне таблиц. Параметры можно посмотреть в представлении `options` схемы `columnar`.

```
SELECT * FROM columnar.options;
-[ RECORD 1 ]---------+-------------
relation              | perf_columnar
chunk_group_row_limit | 10000
stripe_row_limit      | 150000
compression           | zstd
compression_level     | 3
```

Демонстрация

- Директория для временных файлов
- Перемещение директории табличного пространства

04b. Демонстрация

Физическая структура

Часть 1. Директория для временных файлов

1) Запустим стандартно поставляемую утилиту `oid2name`:

```
postgres@tantor:~$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
      5       postgres  pg_default
      4      template0  pg_default
      1      template1  pg_default
  16439        test_db  pg_default
```

Утилита, запущенная без параметров выдаёт список баз данных, название табличного пространства по умолчанию, **oid базы данных**, который соответствует **поддиректории** в директории табличного пространства.

Посмотрим какие **директории** есть в табличном пространстве **pg_default**:

```
postgres@tantor:~$ ls --color -w 1 $PGDATA/base
1
16439
4
5
pgsql_tmp
```

2) Зачем нужна директория **pgsql_tmp**?
Это поддиректория для временных файлов, которая создаётся в директории табличного пространства.
Для временных файлов лучше использовать отдельное табличное пространство, которое стоит отдельно создать.
Как устанавливается табличное пространство для временных файлов?
Параметром конфигурации **temp_tablespaces**
Дожидаться ответа слушателей не нужно, достаточно чтобы они задались вопросом в целях запоминания.

3) Перейдите в директорию `$PGDATA` и удобными средствами (`mc`) покажите директории и поддиректории и дайте короткий обзор что хранится в директориях и файлах

Часть 2. Перемещение директории табличного пространства

1) Создадим табличное пространство. Для этого создадим директорию:

```
postgres@tantor:~$ mkdir $PGDATA/u01
```

2) Проверим что пользователь postgres может читать-писать в неё:

```
postgres@tantor:~$ ls -al $PGDATA/u01
total 8
drwxr-xr-x  2 postgres postgres 4096 Mar 10 10:37 .
drwxr-x--- 20 postgres postgres 4096 Mar 10 10:37 ..
```

3) Запустим `psql`:

```
postgres@tantor:~$ psql
psql (16.1)
Введите "help", чтобы получить справку.
```

4) Попытаемся создать табличное пространство:

```
postgres=# CREATE TABLESPACE u01tbs LOCATION 'u01';
ERROR:  tablespace location must be an absolute path
```

Относительный путь не подходит, **нужно указать абсолютный**.

5) Укажем:

```
postgres=# CREATE TABLESPACE u01tbs LOCATION '/var/lib/postgresql/tantor-se-16/data/u01';
WARNING:  tablespace location should not be inside the data directory
CREATE TABLESPACE
```

Табличное пространство создано, но выдано предупреждение, что **не стоит** директорию `u01` располагать в `PGDATA`. Также не стоит располагать и другие директории (**например, логирования**), чтобы они с большим количеством ненужных файлов не попали в бэкап.

6) Создадим в табличном пространстве таблицу:

```
postgres=# CREATE TABLE t (id bigserial, t text) TABLESPACE u01tbs;
CREATE TABLE
```

7) Во втором окне терминала покажите, что появилось три файла, один из них размером 8192 байт, другие нулевого размера:

Перейдите в директорию табличного пространства и поддиректорию с oid базы данных:

```
postgres@tantor:~$ cd $PGDATA/u01/PG_16_202307071/5
postgres@tantor:~/tantor-se-16/data/u01/PG_16_202307071/5$ ls -l -w 1
итого 8
-rw-r----- 1 postgres postgres    0  374240
-rw-r----- 1 postgres postgres    0  374244
-rw-r----- 1 postgres postgres 8192  374245
```

Что это за файлы?
Это файл основного слоя таблицы `t`, основной слой её `TOAST` таблицы и `TOAST` индекса. `TOAST` таблица и индекс были созданы автоматически, так как есть столбец типа `text`.

8) Проверим к чему относится какой файл:

```
postgres@tantor:~/tantor-se-16/data/u01/PG_16_202307071/5$ oid2name -f 374240
From database "postgres":
  Filenode  Table Name
----------------------
    374240           t

postgres@tantor:~/tantor-se-16/data/u01/PG_16_202307071/5$ oid2name -f 374244
From database "postgres":
  Filenode       Table Name
---------------------------
    374244  pg_toast_374240

postgres@tantor:~/tantor-se-16/data/u01/PG_16_202307071/5$ oid2name -f 374245
From database "postgres":
  Filenode             Table Name
---------------------------------
    374245  pg_toast_374240_index
```

8-килобайтный файл относится к индексу.

9) Перенесем директорию с остановкой экземпляра:

```
postgres@tantor:~/tantor-se-16/data/u01/PG_16_202307071/5$ cd $PGDATA
postgres@tantor:~/tantor-se-16/data$ pg_ctl stop
waiting for server to shut down.... done
```

```
server stopped
postgres@tantor:~/tantor-se-16/data$ mv u01 ..
```

10) Посмотрим список символических ссылок на табличные пространства:

```
postgres@tantor:~/tantor-se-16$ ls -al $PGDATA/pg_tblspc
total 8
drwx------  2 postgres postgres 4096 Mar 10 10:40 .
drwxr-x--- 19 postgres postgres 4096 Mar 10 13:30 ..
lrwxrwxrwx  1 postgres postgres   41 Mar 10 10:40 32913 ->
/var/lib/postgresql/tantor-se-16/data/u01
```

На директорию `u01` указывает ссылка с названием **32913**. В вашем случае название файла ссылки будет другое.

11) Пересоздадим ссылку, чтобы указывала на уже перемещенную директорию:

```
postgres@tantor:~/tantor-se-16$ ln -fs $PGDATA/../u01 $PGDATA/pg_tblspc/32913
```

12) Убедимся, что символическая ссылка указывает на содержимое директории табличного пространства:

```
postgres@tantor:~/tantor-se-16/data$ ls  $PGDATA/pg_tblspc/32913

PG_16_202307071
```

13) Запустим экземпляр:

```
postgres@tantor:~/tantor-se-16/data$ sudo systemctl start tantor-se-server-
16.service
```

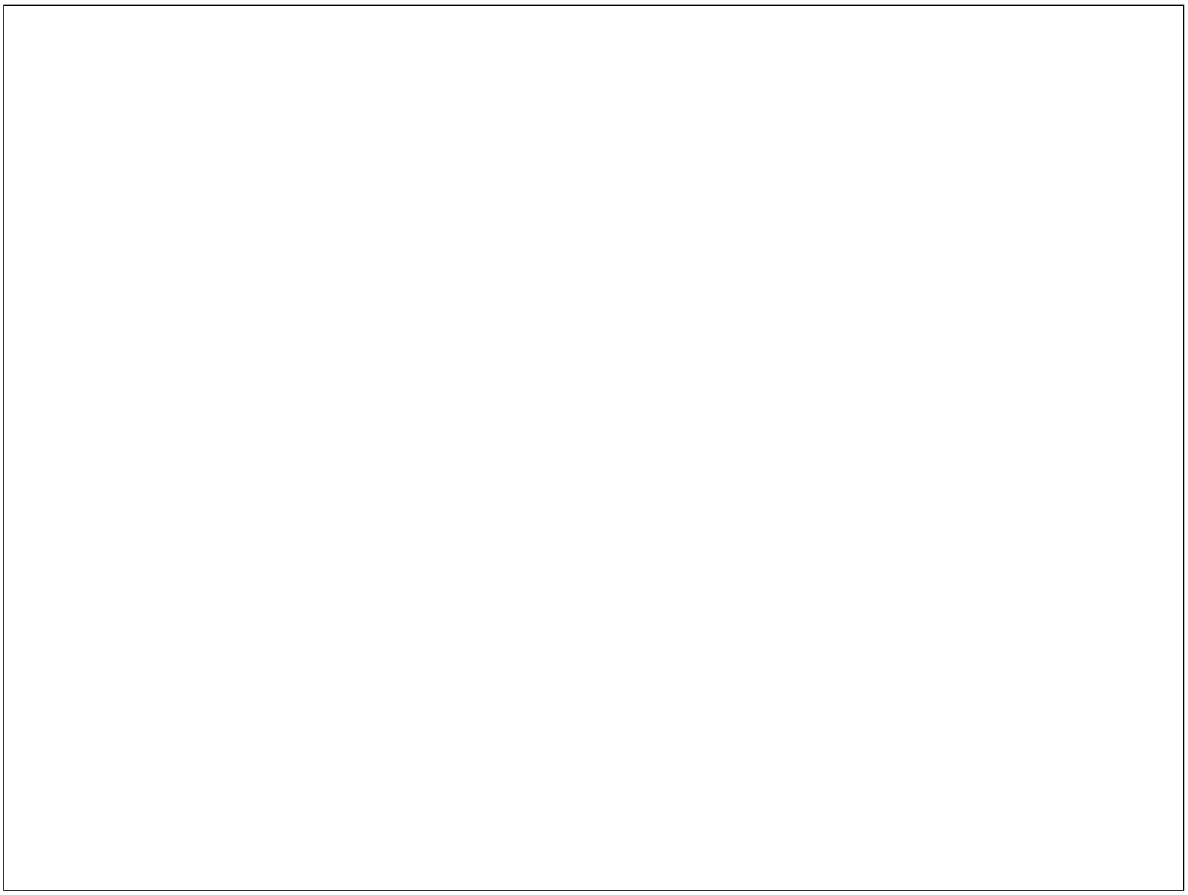
14) Переподсоединимся в окне `psql` и проверим, что содержимое таблицы доступно:

```
postgres=# \c
You are now connected to database "postgres" as user "postgres".
postgres=# select count(*) from t;
 count
-------
     0
(1 row)
```

Директория табличного пространства успешно перенесена.

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. Расширение `ORC` (Колоночно-ориентированный формат, Citus columnar)



СУБД Tantor

Журналирование

Темы

Журнал событий

- Конфигурирование
- Форматы
- Процесс logging_collector
- Демонстрация
- Практическая работа

Определение

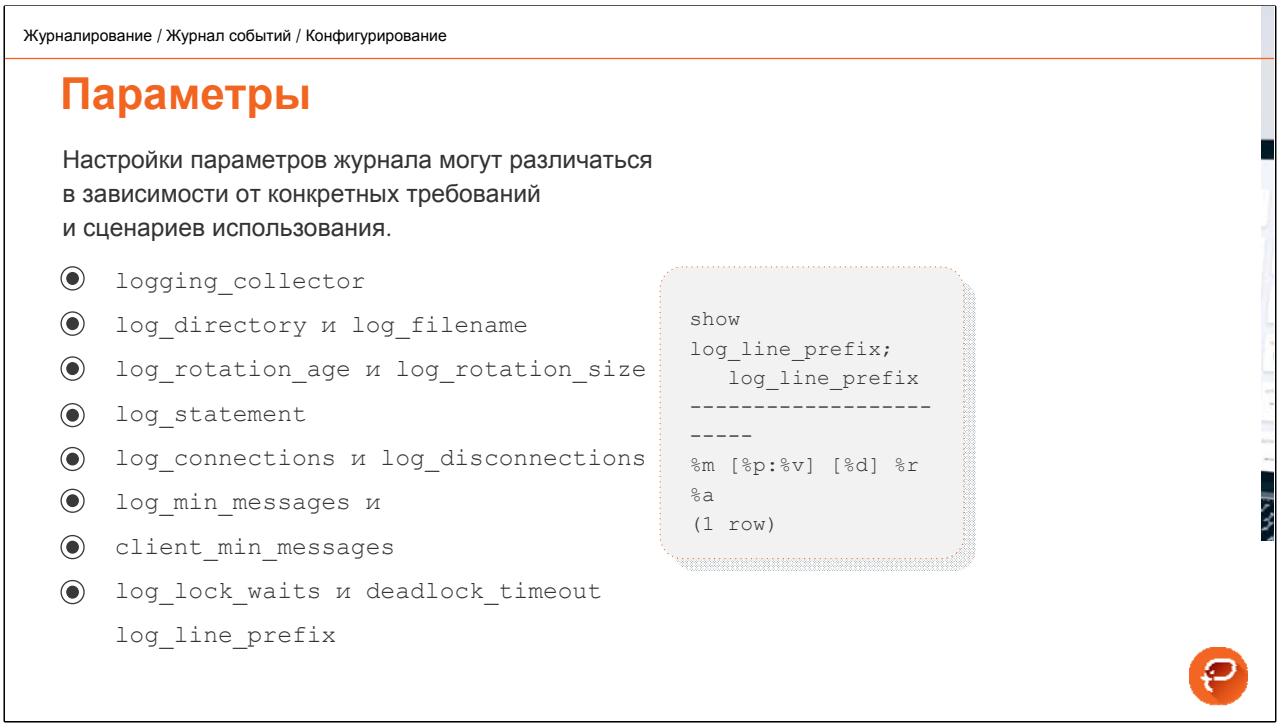
Журнал сообщений (или лог) предоставляет различную информацию о том, как база данных работает, и может быть полезен по многим причинам.

- Диагностика проблем
- Мониторинг производительности
- Безопасность
- Аудит и соответствие
- Восстановление после сбоев
- Мониторинг ресурсов

Журнал сообщений (или лог) PostgreSQL является важным инструментом для отслеживания и анализа деятельности системы баз данных.
Он предоставляет различную информацию о том, как база данных работает, и **может быть полезен по многим причинам**:

●**Диагностика проблем:** Журнал сообщений помогает выявлять и диагностировать проблемы в работе PostgreSQL. Это может быть связано с ошибками в SQL-запросах, проблемами сети, нехваткой ресурсов, блокировками и другими аспектами работы системы.
●**Мониторинг производительности:** Журнал позволяет отслеживать производительность базы данных. Анализ записей о времени выполнения запросов, блокировках и использовании ресурсов помогает оптимизировать работу базы данных и повысить ее эффективность.
●**Безопасность:** Журнал сообщений является важным инструментом для обеспечения безопасности данных. Он фиксирует важные события, такие как попытки неудачной аутентификации, что может помочь выявить попытки несанкционированного доступа.
●**Аудит и соответствие:** В некоторых случаях, особенно в секторах, подчиненных регулированию, ведение подробного журнала может быть необходимым для соответствия нормативам и требованиям.
●**Восстановление после сбоев:** Журнал сообщений может быть использован для восстановления данных после сбоев. Он может содержать информацию о том, какие транзакции были выполнены, и помочь восстановить базу данных после аварийного завершения работы.
●**Мониторинг ресурсов:** Журнал сообщений отслеживает использование ресурсов, таких как CPU, память, дисковое пространство, что позволяет операторам системы следить за общим состоянием сервера и своевременно реагировать на любые проблемы.

Общий доступ к журналу сообщений также может быть ограничен, чтобы обеспечить безопасность и конфиденциальность данных. Однако, важно уметь анализировать и использовать информацию из журнала для поддержания стабильной и производительной работы PostgreSQL.



Параметров журнала достаточно большое количество. Давайте посмотрим какие из них чаще всего настраиваются?

Настройки параметров журнала могут различаться в зависимости от конкретных требований и сценариев использования. Однако, некоторые параметры чаще всего настраиваются и привлекают внимание при работе с PostgreSQL.

Вот несколько из них:

- **logging_collector:** Включает или выключает сборщик журнала, который отвечает за запись логов в файл.
- **log_directory и log_filename:** Определяют директорию и имя файла для записи логов.
- **log_rotation_age и log_rotation_size:** Устанавливают условия для ротации (создания новых) лог-файлов по времени и размеру соответственно.
- **log_statement:** Задает уровень детализации логирования для SQL-запросов.
- **log_connections и log_disconnections:** Управляют логированием событий подключения и отключения от базы данных.
- **log_min_messages и client_min_messages:** Определяют минимальный уровень сообщений, который будет записан в лог.
- **log_lock_waits и deadlock_timeout:** Определяют, будут ли логироваться ожидания блокировок и таймаут для обнаружения взаимоблокировок.
- **log_line_prefix:** это параметр конфигурации в PostgreSQL, который определяет формат префикса, добавляемого к каждой строке в лог-файле.

Эти параметры представляют общий набор основных настроек, которые часто используются при настройке журнала PostgreSQL. Однако, конфигурация может быть адаптирована в зависимости от конкретных требований проекта и задач администрирования баз данных.

Запись `show log_line_prefix;` в PostgreSQL используется для отображения текущего значения параметра `log_line_prefix`.

В данном конкретном примере:

```
show log_line_prefix;
   log_line_prefix
------------------------
%m [%p:%v] [%d] %r %a
(1 row)
```

%m: Уровень сообщения (DEBUG5, DEBUG4, INFO, WARNING, ERROR, и так далее).
[%p:%v]: Идентификатор процесса PostgreSQL и номер версии протокола.
[%d]: Имя базы данных.
%r: Идентификатор транзакции.
%a: IP-адрес и порт клиента.

Таким образом, этот префикс будет включать уровень сообщения, идентификатор процесса и версию протокола, имя базы данных, идентификатор транзакции, а также IP-адрес и порт клиента для каждой строки в логе. Это делает логи более информативными и удобными для анализа в контексте деятельности PostgreSQL.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/runtime-config-logging.html#GUC-LOG-LINE-PREFIX

Расположение

Где и как хранится журнал сообщений сервера

- `log_directory`

Определяет каталог для файлов журнала,
по умолчанию - "log"

- `log_filename`

Устанавливает имена файлов журнала при включенном `logging_collector`.

Использует шаблон strftime для динамических имен файлов.
Значение по умолчанию - `"postgresql-%Y-%m-%d_%H%M%S.log"`.

Давайте выясним, где хранится журнал сообщений сервера и как он называется.
Для этого необходимо рассмотреть два ключевых параметра:

●log_directory (string): Если параметр logging_collector включен, то log_directory определяет каталог, в котором создаются файлы журнала. Этот путь может быть указан абсолютным или относительным каталогом данных кластера. Значение log_directory устанавливается в файле postgresql.conf или в командной строке сервера. По умолчанию, используется значение "log".
●log_filename (string): Когда параметр logging_collector включен, log_filename устанавливает имена файлов для создаваемых журналов. В значении параметра можно использовать символы подстановки (%) для автоматического наименования файлов. Обратите внимание, что если используются значения, зависящие от часового пояса, вычисления выполняются в зоне, указанной в параметре конфигурации log_timezone. Поддерживаемые символы подстановки аналогичны тем, которые перечислены в спецификации strftime Open Group. Значение по умолчанию - "postgresql-%Y-%m-%d_%H%M%S.log".

https://docs.tantorlabs.ru/tdb/ru/15_4/se/runtime-config-logging.html#GUC-LOG-LINE-PREFIX

Приемники сообщений

PostgreSQL обеспечивает различные методы регистрации сообщений сервера, включая следующие:

- `stderr`: Сообщения регистрируются в стандартный поток ошибок
- `csvlog`: Логи в формате CSV для легкого анализа
- `jsonlog`: Логи в формате JSON для структурированного вывода
- `syslog`: Сообщения отправляются в системный журнал операционной системы
- `eventlog` (Windows): Запись логов в Windows Event Log

```
show
log_destination;
 log_destination
-----------------
 stderr, csvlog
```

PostgreSQL обеспечивает различные методы регистрации сообщений сервера.

Логи в формате `JSON` обеспечивают структурированный вывод, что упрощает обработку данных с использованием инструментов обработки `JSON`.

`syslog`:

Сообщения отправляются в системный журнал операционной системы (`syslog`), который может быть настроен для централизованного сбора логов на удаленном сервере.

`eventlog (Windows)`: На операционной системе Windows доступен специфический метод записи логов в `Windows Event Log`. Это может быть полезно для интеграции с инфраструктурой мониторинга Windows.

Для указания предпочтительных методов регистрации сообщений, используется параметр `log_destination`. Значения этого параметра задаются списком, разделенным запятыми, в файле конфигурации postgresql.conf или в командной строке сервера.

Например:

```
log_destination = 'stderr,csvlog'
```

В данном случае сообщения будут регистрироваться и в `stderr`, и в формате `CSV`. Параметр `log_destination` позволяет гибко настраивать распределение логов в соответствии с требованиями системы мониторинга и анализа.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/runtime-config-logging.html#GUC-LOG-DESTINATION

Ротация

Применение ротации журнала в PostgreSQL:

- `log_rotation_age`:

Определяет максимальный возраст файла журнала в минутах до его ротации.
Пример: log_rotation_age = 60 (ротация каждый час).

- `log_rotation_size`:

Определяет максимальный размер файла журнала в мегабайтах до его ротации.

Пример: log_rotation_size = 10MB (ротация при размере 10 МБ).

Эти параметры позволяют гибко управлять ротацией журнала в зависимости от времени и размера, обеспечивая эффективное управление логами PostgreSQL.

Для применения ротации журнала в PostgreSQL, вы можете использовать параметры `log_rotation_age` и `log_rotation_size` в файле конфигурации `postgresql.conf`.

- `log_rotation_age`: Этот параметр определяет максимальный возраст файла журнала в минутах, после которого файл будет подвергнут ротации. Например, чтобы установить ротацию каждый час, вы можете указать:
`log_rotation_age = 60`

- `log_rotation_size`: Определяет максимальный размер файла журнала в мегабайтах, после которого будет произведена ротация.
Например, чтобы установить ротацию при достижении файла размера 10 МБ, вы можете указать:
`log_rotation_size = 10MB`

Эти параметры могут быть настроены в соответствии с вашими требованиями по управлению логами. Оба параметра могут быть использованы одновременно для управления как по времени, так и по размеру. Когда одно из условий (возраст или размер) выполняется, происходит ротация, и текущий лог-файл переименовывается, а новый создается.

Не забудьте перезапустить PostgreSQL после внесения изменений в файл конфигурации для применения новых параметров.

Чтение сообщений

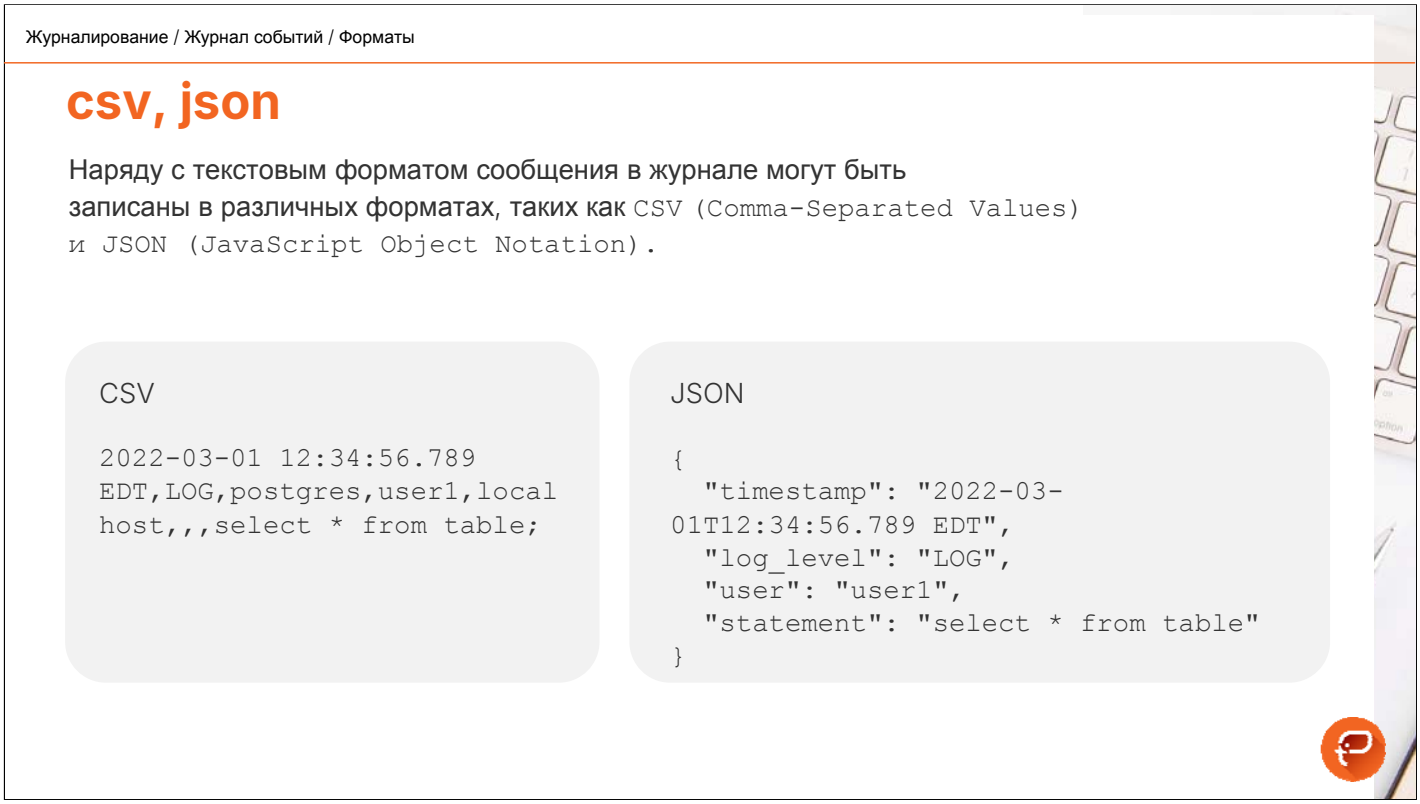
Просмотр текстовых логов:

- Использование стандартных текстовых лог-файлов
- Использование pgBadger
- Настройка централизованного сбора логов
- Применение систем мониторинга и управления

Для СУБД Tantor предпочтительным инструментом чтения и анализа журнала является Платформа

Чтение сообщений сервера в PostgreSQL рекомендуется осуществлять через просмотр лог-файлов, куда записываются соответствующие сообщения.
Здесь несколько способов и инструментов для этого:
●**Просмотр текстовых логов:** Стандартный текстовый лог-файл, который может быть настроен через параметры `log_directory` и `log_filename` в файле postgresql.conf, предоставляет читаемый формат сообщений.
Вы можете использовать текстовые редакторы, команды tail или cat для просмотра содержимого лог-файлов.
●**Использование** `pgBadger`: pgBadger - это утилита анализа логов PostgreSQL, которая генерирует красивые и информативные отчеты на основе лог-файлов.
Установка `pgBadger` и его использование позволяют легко просматривать и анализировать логи с дополнительной статистикой и графиками.
●**Настройка централизованного сбора логов**: Можно настроить централизованный сбор логов, отправляя их на удаленный сервер или используя системы централизованного логирования, такие как `ELK Stack (Elasticsearch, Logstash, Kibana)` или `Graylog`.
Эти инструменты обеспечивают удобный интерфейс для поиска, фильтрации и анализа логов. Централизованным сбором так же занимается Платформа `Tantor`.
●**Применение систем мониторинга и управления:** Многие системы мониторинга (например, `Prometheus, Grafana`) могут интегрироваться с `PostgreSQL`, предоставляя возможности по отслеживанию и визуализации логов.
Выбор способа зависит от требований, предпочтений и степени автоматизации, необходимой для анализа логов. Каждый из перечисленных способов имеет свои преимущества и может быть выбран в зависимости от конкретных потребностей вашего проекта.

Для СУБД `Tantor` предпочтительным инструментом чтения и анализа журнала является Платформа



В PostgreSQL, наряду с текстовым форматом сообщения в журнале могут быть записаны в различных форматах, таких как CSV (Comma-Separated Values) и JSON (JavaScript Object Notation).

Вот краткое описание каждого формата:

- CSV (Comma-Separated Values): В формате CSV каждая запись в логе представлена как строка, где значения разделены запятыми. Этот формат облегчает анализ с использованием стандартных инструментов обработки CSV.

Пример записи:

```
2022-03-01 12:34:56.789 EDT,LOG,postgres,user1,localhost,,,select * from table;
```

Как включить:

```
log_destination = 'csvlog'
```

- JSON (JavaScript Object Notation): В формате JSON каждая запись представлена в виде объекта JSON. Это обеспечивает структурированный и читаемый вывод, который легко обрабатывать инструментами обработки JSON.

Как включить:

```
log_destination = 'jsonlog'
```

```
{
  "timestamp": "2022-03-01T12:34:56.789 EDT",
  "log_level": "LOG",
  "user": "user1",
  "host": "localhost",
  "database": "postgres",
  "statement": "select * from table"
}
```

Выбор между CSV и JSON зависит от предпочтений и требований к структурированию данных. JSON обеспечивает более гибкий и читаемый формат, поддерживающий структурированный анализ данных, в то время как CSV может быть предпочтителен при использовании стандартных инструментов обработки CSV.

Гарантия доставки сообщения

Если установлен фоновый процесс сборщика сообщений Tantor SE гарантирует, что все сообщения дойдут до приемника сообщений.

```
show
logging_collector;
 logging_collector
-------------------
 on
```

`logging_collector` - это параметр конфигурации `PostgreSQL`, который управляет активацией сборщика журналов. Сборщик журналов - это фоновый процесс, который отслеживает и захватывает сообщения журнала, отправленные в стандартный поток ошибок (`stderr`), и перенаправляет их в файлы журналов.

Сборщик журналов выполняет роль фонового процесса, работающего параллельно с основным процессом `PostgreSQL`. Его основная задача - перехватывать и сохранять сообщения, которые обычно выводятся в stderr, вместо их непосредственного вывода в консоль.

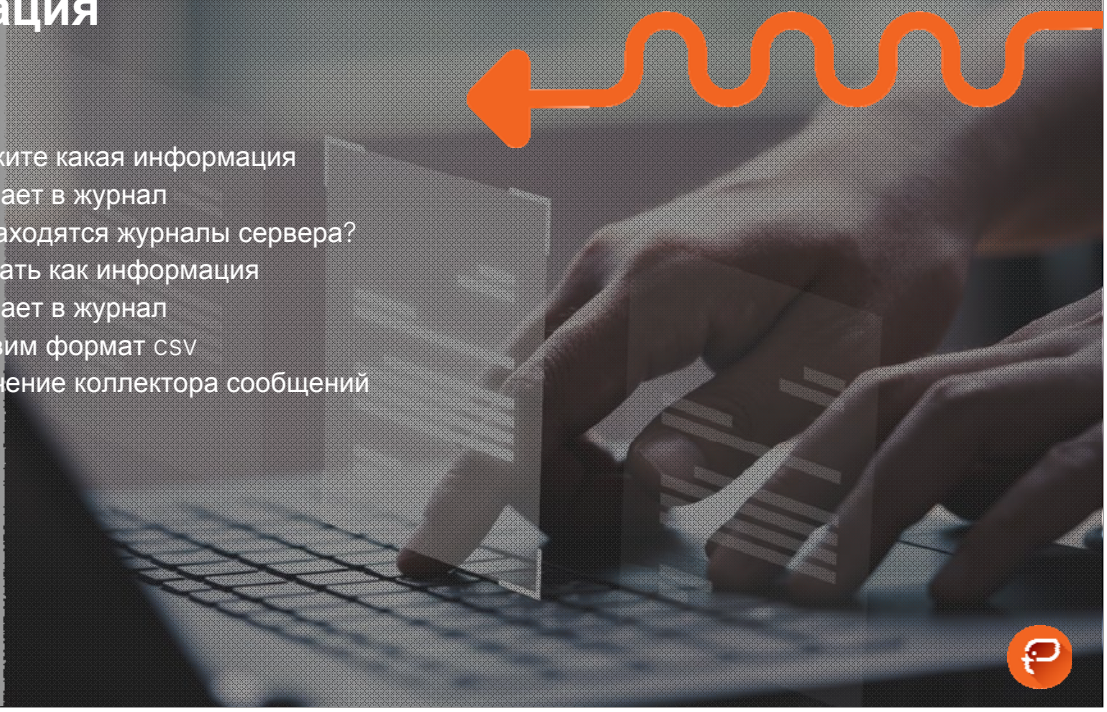
Параметр `logging_collector` может быть установлен только при запуске сервера. Это означает, что для активации или деактивации сборщика журналов необходимо изменить параметр в конфигурационном файле (`postgresql.conf`) и перезапустить `PostgreSQL`. Также можно задать в командной строке при запуске сервера

Итак, `logging_collector` предоставляет эффективный механизм сбора и хранения логов в `PostgreSQL`, улучшая контроль над журналированием и обеспечивая более полную информацию о событиях и ошибках.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/runtime-config-logging.html#GUC-LOGGING-COLLECTOR

Демонстрация

1. Покажите какая информация попадает в журнал
2. Где находятся журналы сервера?
3. Показать как информация попадает в журнал
4. Добавим формат csv
5. Включение коллектора сообщений



Часть 1. Какая информация попадает в журнал

Загрузим `psql`:

```
astra@tantor:~$ psql
psql (16.1)
Введите "help", чтобы получить справку.
postgres=#

postgres=# SHOW log_line_prefix;
   log_line_prefix
-----------------------
%m [%p:%v] [%d] %r %a
(1 row)
```

%m: Уровень сообщения (`DEBUG5, DEBUG4, INFO, WARNING, ERROR`, и так далее).

[%p:%v]: Идентификатор процесса PostgreSQL и номер версии протокола.

[%d]: Имя базы данных.

%r: Идентификатор транзакции.

%a: IP-адрес и порт клиента.

Часть 2. Расположение журналов сервера

1) Посмотрим путь до журналов

```
postgres=# SHOW log_directory;
 log_directory
---------------
 log
(1 row)
```

Какая маска у файлов журнала?

```
postgres=# SHOW log_filename;
          log_filename
------------------------------
 postgresql-%Y-%m-%d_%H%M%S.log
(1 row)
```

Где находятся данных кластера БД?

```
postgres=# SHOW data_directory;
            data_directory
---------------------------------------
 /var/lib/postgresql/tantor-se-14/data
(1 row)
```

2) Посмотрим содержимое папки журнала.

```
postgres=#\! ls -l /var/lib/postgresql/tantor-se-14/data/log
total 148228
-rw------- 1 postgres postgres     1115 Jul  3  2023 postgresql-2023-07-
03_130021.log
-rw------- 1 postgres postgres     1112 Jul  3  2023 postgresql-2023-07-
03_130033.log
-rw------- 1 postgres postgres     6545 Jul  3  2023 postgresql-2023-07-
03_162937.log
—------------------------------
```

3) Посмотрим содержимое любого журнала.

```
postgres=# \! tail -n 10  /var/lib/postgresql/tantor-se-
14/data/log/postgresql-2023-07-03_130021.log
2023-07-03 13:00:21.009 MSK [23506] LOG:  listening on IPv6 address "::1",
port 5432
```

```
2023-07-03 13:00:21.012 MSK [23506] LOG:  listening on Unix socket
"/var/run/postgresql/.s.PGSQL.5432"
2023-07-03 13:00:21.017 MSK [23506] LOG:  listening on Unix socket
"/tmp/.s.PGSQL.5432"
2023-07-03 13:00:21.021 MSK [23508] LOG:  database system was shut down at
2023-07-03 13:00:19 MSK
2023-07-03 13:00:21.027 MSK [23506] LOG:  database system is ready to accept
connections
2023-07-03 13:00:33.293 MSK [23506] LOG:  received fast shutdown request
2023-07-03 13:00:33.297 MSK [23506] LOG:  aborting any active transactions
2023-07-03 13:00:33.299 MSK [23506] LOG:  background worker "logical
replication launcher" (PID 23514) exited with exit code 1
2023-07-03 13:00:33.300 MSK [23509] LOG:  shutting down
2023-07-03 13:00:33.328 MSK [23506] LOG:  database system is shut down
```

Убедились, что маска соответствует.

Часть 3. Как информация попадает в журнал

```
postgres=# CREATE TABLE t (id integer);
CREATE TABLE
postgres=# \! tail -n 10  /var/lib/postgresql/tantor-se-
14/data/log/postgresql-2024-02-29_100938.log
2024-02-29 07:49:54.753 MSK [5289:8/30] [postgres] [local] psql LOG:
statement: create table t (id integer);
```

Часть 4. Добавление формата csv

1) Посмотрим параметр.

```
postgres=# SHOW log_destination;
log_destination
-----------------
 stderr
(1 row)
```

2) Изменим параметр и перечитаем конфигурацию.

```
postgres=# ALTER SYSTEM SET log_destination = 'stderr,csvlog';
ALTER SYSTEM
postgres=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

3) Посмотрим что параметр успешно применяется.

```
postgres=# SHOW log_destination;
 log_destination
-----------------
 stderr,csvlog
(1 row
```

4) Вставим новое значение в таблицу `t`.

```
postgres=# INSERT INTO t VALUES(1);
INSERT 0 1
```

5) Посмотрим содержимое файла.

```
postgres=# \! ls -l /var/lib/postgresql/tantor-se-14/data/log/*csv
-rw------- 1 postgres postgres 30749 Feb 29 08:02 /var/lib/postgresql/tantor-
se-14/data/log/postgresql-2024-02-29_080122.csv
```

6) Добавился формат данных csv.

```
postgres=#\! tail -n 1 /var/lib/postgresql/tantor-se-14/data/log/postgresql-
2024-02-29_080122.csv
```

```
2024-02-29 08:08:54.580
MSK,"postgres","postgres",9199,"[local]",65e01024.23ef,3,"idle",2024-02-29
08:03:32 MSK,5/325,0,LOG,00000,"statement: insert into t
values(1);",,,,,,,,,"psql","client backend",,0
```

7) Сравним с содержимым обычного журнала.

```
postgres=# INSERT INTO t VALUES(1);
INSERT 0 1
postgres=# \! tail -n 1 /var/lib/postgresql/tantor-se-14/data/log/postgresql-
2024-02-29_080122.log
2024-02-29 08:12:02.631 MSK [9199:5/326] [postgres] [local] psql LOG:
statement: insert into t values(1);
postgres=#
```

Часть 5. Включение коллектора сообщений

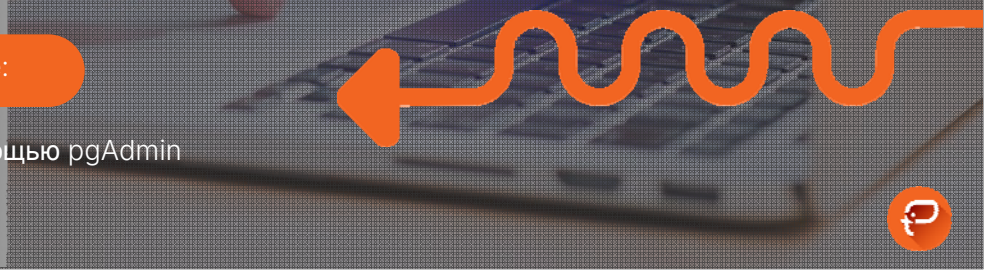
```
postgres=# show logging_collector;
 logging_collector
-------------------
 on
(1 row)
```

Удалим ненужные объекты:

```
postgres=# DROP TABLE t;
DROP TABLE
postgres=# ALTER SYSTEM SET log_destination = 'stderr';
ALTER SYSTEM
postgres=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

Практическая работа

1. Покажите какая информация попадает в журнал
2. Где находятся журналы сервера?
3. Показать как информация попадает в журнал
4. Добавим формат csv
5. Включение коллектора сообщений



Дополнительное задание:

Сделать задание 1-5 с помощью pgAdmin

Спасибо!

tantorlabs.ru
edu@tantorlabs.ru

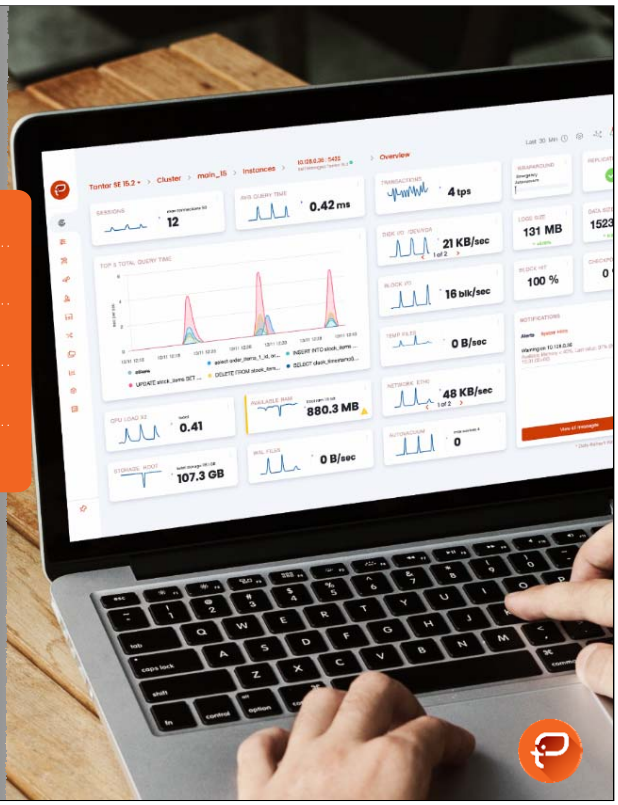
СУБД Tantor

Безопасность

Темы

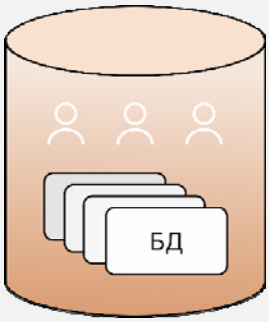
- Ролевая модель безопасности
- Подключение и аутентификация

- Роли
- Привилегии
- Политика защиты строк
- Демонстрация
- Практическая работа



Роль

- Роль в PostgreSQL определяет права доступа к базе данных и ее объектам
- Роли могут иметь различные привилегии, включая создание, изменение и удаление объектов
- Роль пользователя в PostgreSQL не связана с внешним пользователем операционной системы
- Создание роли пользователя в PostgreSQL не создает учетную запись в ОС, и наоборот
- Роль - это объект СУБД создается на уровне кластера БД




```
CREATE ROLE имя_роли;
```

```
DROP ROLE имя_роли;
```

В PostgreSQL роль (`role`) представляет собой концепцию, аналогичную пользователям или группам пользователей в других системах управления базами данных (СУБД). Роль в PostgreSQL определяет права доступа к базе данных и объектам в ней, таким как таблицы, представления, процедуры и другие объекты. Роли могут иметь различные привилегии, такие как создание, изменение или удаление объектов базы данных, а также права на выполнение определенных операций.

Роль - это объект в базе данных, создаваемый на уровне кластера. Это означает, что после создания роль становится видимой в любой базе данных в этом кластере.

Роли в PostgreSQL предоставляют гибкий и мощный механизм управления доступом к данным и объектам базы данных.

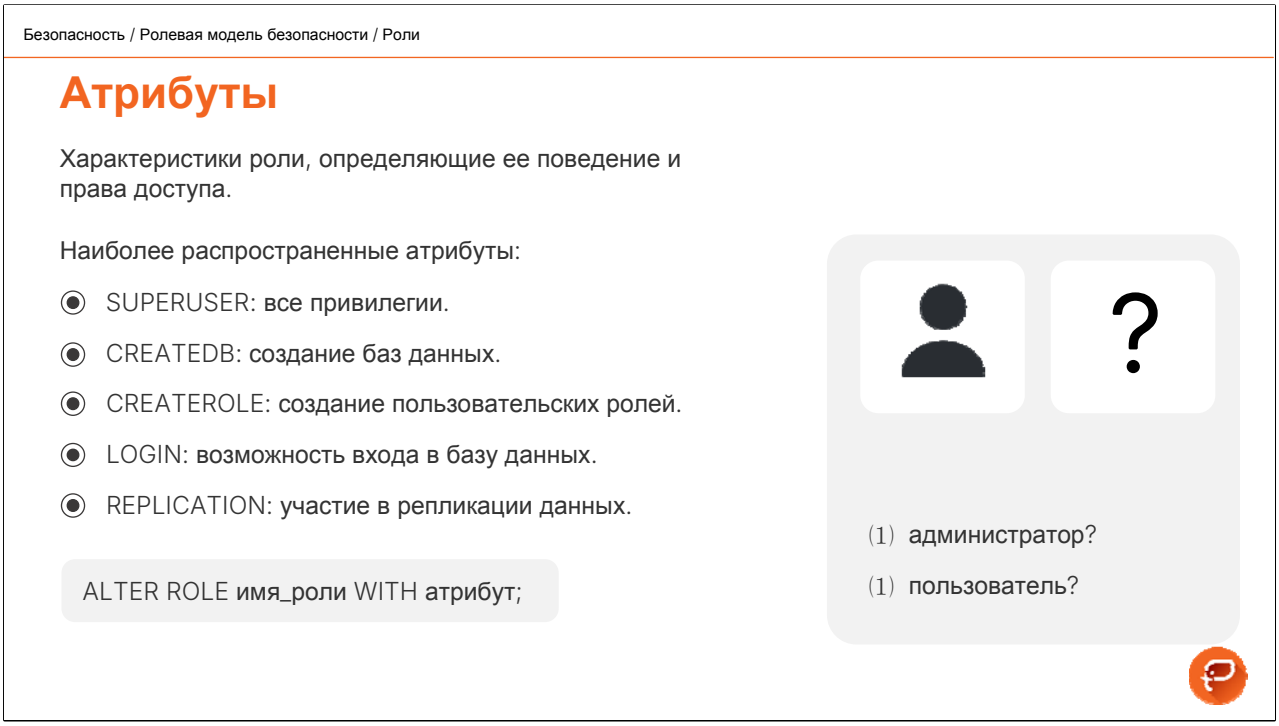
В PostgreSQL роль пользователя (`User Role`) не имеет прямой связи с внешним пользователем операционной системы (ОС). То есть, создание роли пользователя в PostgreSQL не предполагает автоматического создания соответствующей учетной записи в ОС, и наоборот. Роли пользователей в PostgreSQL управляются в рамках самой базы данных и не имеют прямого отображения на учетные записи ОС.

Это означает, что для входа в базу данных пользователь должен явно аутентифицироваться через механизм аутентификации PostgreSQL, например, с помощью имени пользователя и пароля базы данных, не зависящего от учетной записи операционной системы.

Таким образом, роли пользователей в PostgreSQL обеспечивают гибкое управление доступом к данным и объектам базы данных в пределах самой СУБД, независимо от внешних источников аутентификации.

Для создания и удаления роли в PostgreSQL вы можете использовать SQL-команды `CREATE ROLE` и `DROP ROLE`

https://docs.tantorlabs.ru/tdb/ru/15_4/se/database-roles.html



В PostgreSQL атрибуты роли представляют собой характеристики или свойства, которые могут быть назначены ролям пользователей для управления их поведением и правами доступа в базе данных. Вот некоторые из наиболее распространенных атрибутов ролей:

`SUPERUSER`: Роль с этим атрибутом обладает всеми привилегиями в базе данных и может выполнять любые операции.

`CREATEDB`: Роль с этим атрибутом может создавать новые базы данных.

`CREATEROLE`: Роль с этим атрибутом может создавать новые роли пользователей.

`LOGIN`: Роль с этим атрибутом может входить в базу данных. Обычно это используется для обозначения того, что роль является ролью пользователя, которая может выполнять аутентификацию.

`REPLICATION`: Роль с этим атрибутом может участвовать в процессе репликации данных между серверами баз данных.

Есть и другие атрибуты, будут упомянуты в контексте курса

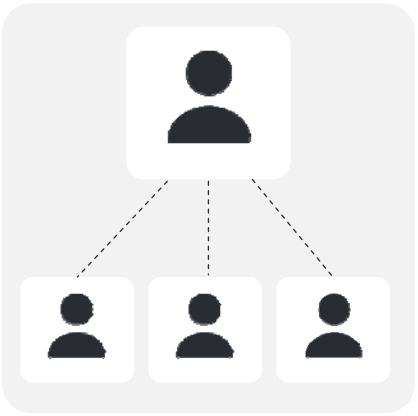
Как вариант атрибуты - это свойства объекта роль

Для установки атрибутов роли в PostgreSQL вы можете использовать команду `ALTER ROLE` с указанием имени роли и необходимого атрибута `ALTER ROLE` имя_роли `WITH` атрибут;

https://docs.tantorlabs.ru/tdb/ru/15_4/se/role-attributes.html

Групповая роль

- Групповые роли создаются путем назначения одной роли в качестве члена другой роли.
- Добавлять другие роли в эту группу с помощью команды GRANT.
- GRANT имя_групповой_роли TO имя_роли;
- Отзыв делается командой REVOKE
- REVOKE имя_групповой_роли FROM имя_роли;
- Переключение на другую роль
- SET ROLE имя_роли.



Групповая роль в PostgreSQL - это роль, которая объединяет одну или несколько других ролей вместе. Она позволяет управлять правами доступа и привилегиями для нескольких пользователей одновременно, обеспечивая более удобное и гибкое управление безопасностью базы данных.

Добавлять другие роли в эту группу с помощью команды `GRANT`. Это делается путем назначения роли членом другой роли:

```
GRANT имя_групповой_роли TO имя_роли;
```

Отзыв делается командой `REVOKE`

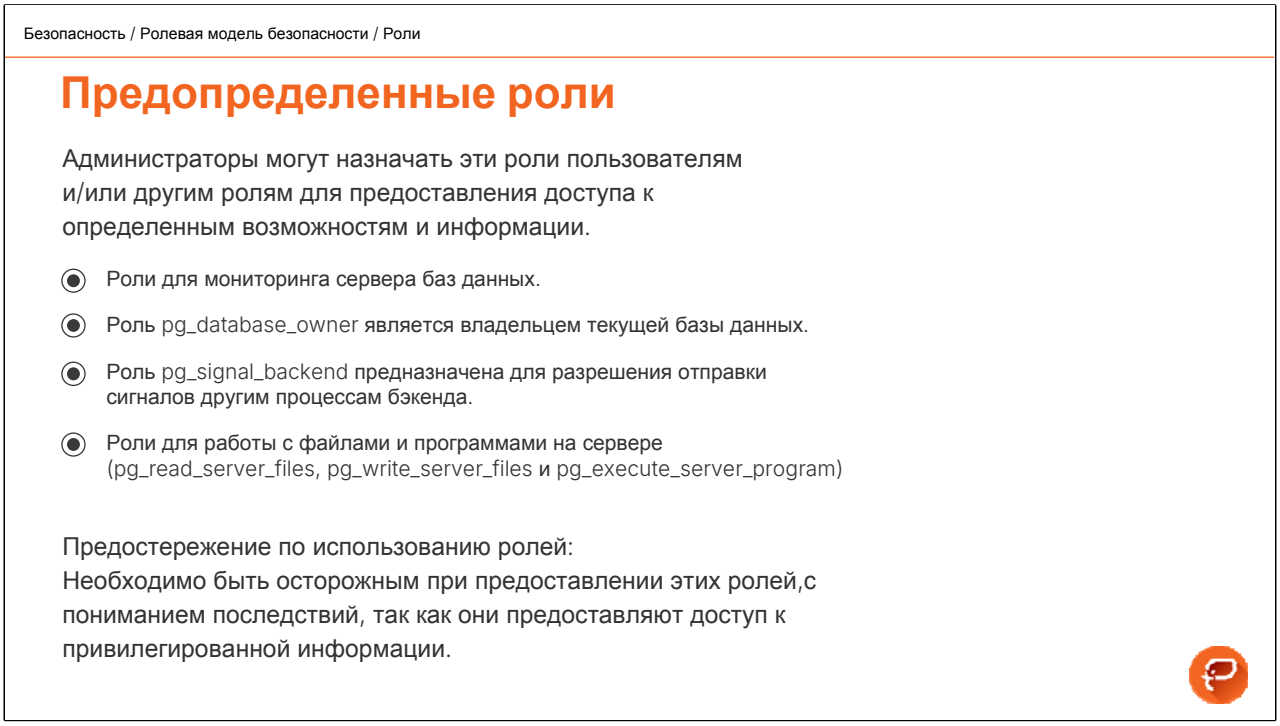
```
REVOKE имя_групповой_роли  FROM имя_роли;
```

Переключение на другую роль позволяет текущему пользователю или роли временно стать другой ролью с правами доступа и привилегиями этой роли. Это делается с помощью команды

```
SET ROLE имя_роли.
```

После того, как роли были добавлены в группу, они наследуют права доступа и привилегии, назначенные этой групповой роли. Это означает, что если вы назначите какие-либо права доступа или привилегии групповой роли, то все её члены автоматически наследуют эти права. Таким образом, групповая роль обеспечивает единое управление доступом для всех её членов.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/role-membership.html



СУБД предоставляет набор предопределенных ролей для предоставления доступа к привилегированным возможностям и информации.

Администраторы могут назначать эти роли пользователям и/или другим ролям для предоставления доступа к определенным возможностям и информации.

1.Роли для мониторинга сервера баз данных:

В PostgreSQL существуют роли, такие как `pg_monitor`, `pg_read_all_settings`, `pg_read_all_stats` и `pg_stat_scan_tables`, которые предназначены для облегчения настройки роли администратора для мониторинга сервера баз данных.

Эти роли предоставляют общие привилегии для чтения различных настроек конфигурации, статистики и другой системной информации.

1.Роль `pg_database_owner`:

Роль `pg_database_owner` является владельцем текущей базы данных.

Она может управлять объектами в базе данных и получать предоставление привилегий доступа.

`pg_database_owner` не может быть членом другой роли и не имеет неявных членов.

1.Роль `pg_signal_backend`:

Роль pg_signal_backend предназначена для разрешения отправки сигналов другим процессам бэкенда.

Это позволяет администраторам разрешать доверенным ролям отправлять сигналы для отмены запросов или завершения сессий.

1.Роли для работы с файлами и программами на сервере:

Роли `pg_read_server_files`, `pg_write_server_files` и `pg_execute_server_program` предоставляют доступ к файлам и возможность запуска программ на сервере базы данных от имени пользователя, от имени которого работает база данных.

Эти роли обходят проверки разрешений на уровне базы данных и могут использоваться для получения привилегий суперпользователя.

Предостережение по использованию ролей:

Необходимо быть осторожным при предоставлении этих ролей, чтобы они использовались только там, где это необходимо, и с пониманием последствий, так как они предоставляют доступ к привилегированной информации.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/predefined-roles.html

public

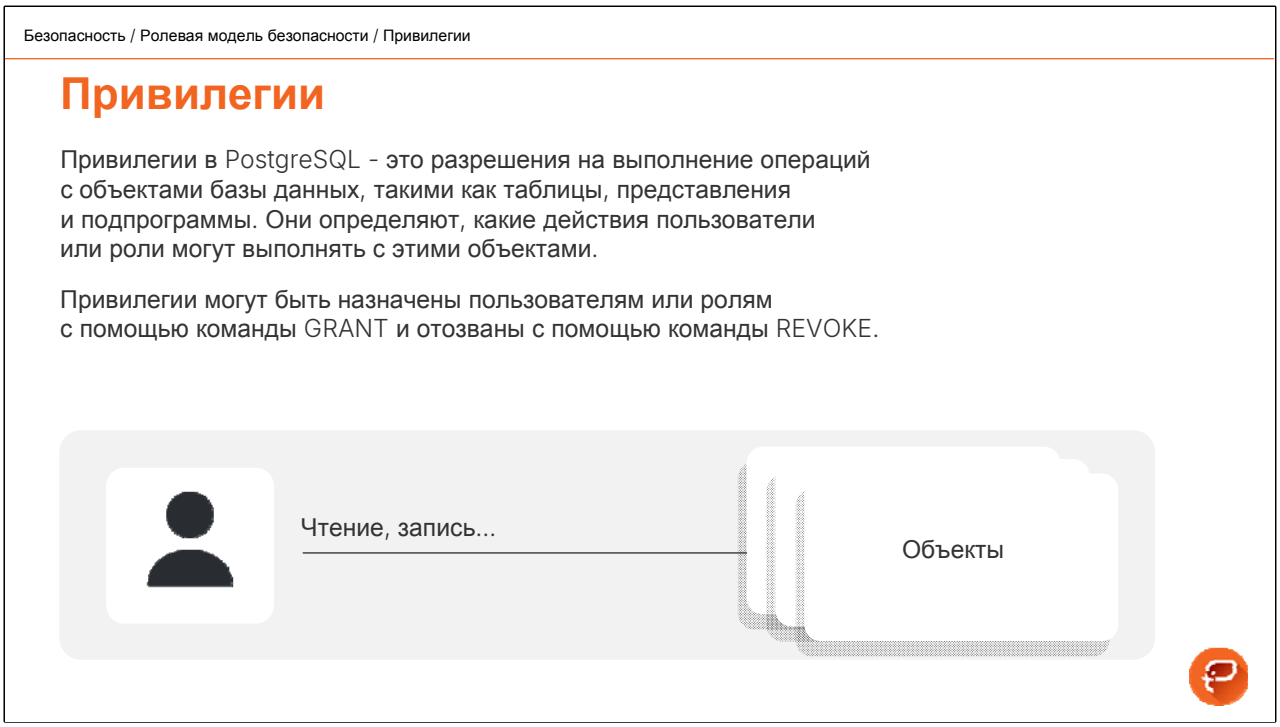
Роль "public" включает в себя все остальные роли по умолчанию

Хотя роль "public" обеспечивает удобное и простое управление доступом к объектам базы данных, следует помнить о потенциальных рисках безопасности.

Предоставление привилегий через роль "public" может привести к широкому доступу к данным и функциональности, что может повысить риск утечки информации или злоупотребления привилегиями.

Предупреждение! Отзыв привилегий у роли "public" может быть сложным и потребует тщательного тестирования, чтобы убедиться, что изменения не повредят функциональность системы и не нарушают доступ пользователей,

Роль "`public`" включает в себя все остальные роли по умолчанию. Это означает, что если вы назначаете какие-либо привилегии для роли "`public`", то эти привилегии будут распространяться на всех пользователей и роли в базе данных, которые не имеют явно определенных прав доступа.
Роль "`public`" является глобальной ролью по умолчанию для всех пользователей, если для них не указаны специальные права доступа. Это обеспечивает простоту и единообразие в управлении доступом, поскольку права, назначенные для "`public`", применяются ко всем объектам базы данных для всех пользователей, если не определены другие специфические права.
Таким образом, роль "`public`" является ключевым элементом в обеспечении открытого или общего доступа к объектам базы данных по умолчанию, если не требуется явное ограничение доступа для конкретных пользователей или ролей.
Хотя роль "`public`" обеспечивает удобное и простое управление доступом к объектам базы данных, следует помнить о потенциальных рисках безопасности. Предоставление привилегий через роль "`public`" может привести к широкому доступу к данным и функциональности, что может повысить риск утечки информации или злоупотребления привилегиями.
Отзыв привилегий у роли "`public`" может быть сложным и потребует тщательного тестирования, чтобы убедиться, что изменения не повредят функциональность системы и не нарушают доступ пользователей, которым привилегии были предоставлены. Поэтому рекомендуется использовать принцип минимальных привилегий и явно назначать права доступа только тем пользователям и ролям, которым они действительно необходимы.



Привилегии в PostgreSQL представляют собой разрешения на выполнение определенных действий с объектами базы данных, такими как чтение, запись, изменение или удаление данных. Они могут быть назначены пользователям, ролям или группам пользователей, чтобы контролировать их доступ к данным и функциональности базы данных.

Эти привилегии распространяются на различные объекты базы данных, включая таблицы, представления, подпрограммы (например, функции и хранимые процедуры), а также на другие элементы, такие как базы данных и схемы.

Привилегии позволяют определить, кто может читать, записывать, изменять или удалять данные из определенных таблиц, кто может выполнять определенные операции с функциями или процедурами, и так далее.

Привилегии могут быть назначены пользователям или ролям с помощью команды `GRANT` и отозваны с помощью команды `REVOKE`.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/ddl-priv.html#id1

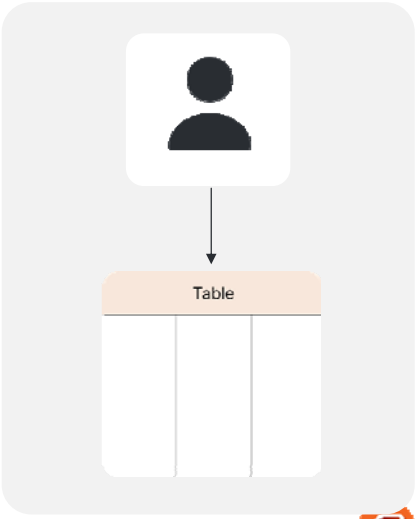
Привилегии таблицы, представления

У таблицы в PostgreSQL могут быть следующие привилегии:

- SELECT: Разрешает чтение данных из таблицы.
- INSERT: Разрешает добавление новых строк в таблицу.
- UPDATE: Разрешает изменение существующих строк в таблице.
- DELETE: Разрешает удаление строк из таблицы.
- REFERENCES: Разрешает ссылаться на таблицу внешним ключом.
- TRIGGER: Разрешает создание триггеров на таблицу.
- TRUNCATE: Разрешает очистку (удаление всех строк) таблицы.

На уровне столбца таблицы можно назначить привилегии SELECT INSERT UPDATE REFERENCE

На уровне представления есть только две привилегии SELECT и REFERENCES



У таблицы в PostgreSQL могут быть следующие привилегии:

`SELECT`: Разрешает чтение данных из таблицы.
`INSERT`: Разрешает добавление новых строк в таблицу.
`UPDATE`: Разрешает изменение существующих строк в таблице.
`DELETE`: Разрешает удаление строк из таблицы.
`REFERENCES`: Разрешает ссылаться на таблицу внешним ключом.
`TRIGGER`: Разрешает создание триггеров на таблицу.
`TRUNCATE`: Разрешает очистку (удаление всех строк) таблицы.

На уровне столбца таблицы можно назначить привилегии `SELECT`, `INSERT`, `UPDATE`, `REFERENCE`, что расширяет возможности ролевой модели безопасности

В PostgreSQL привилегии на уровне представления ограничены двумя типами:

`SELECT`: Разрешает чтение данных из представления.
`TRIGGER`: Разрешает создание триггеров на представление.

Привилегии БД, схемы

В PostgreSQL привилегии для базы данных и схемы могут включать следующие

Для базы данных:

- CONNECT: Разрешает подключение к базе данных.
- CREATE: Разрешает создание новых объектов в базе данных.
- TEMPORARY: Разрешает создание временных объектов в базе данных.

Для схемы:

- CREATE: Разрешает создание новых объектов в схеме.
- USAGE: Разрешает использование объектов в схеме.

В PostgreSQL для базы данных и схемы можно устанавливать следующие привилегии.

Для базы данных:

`CONNECT`: Разрешает подключение к базе данных.
`CREATE`: Разрешает создание новых объектов в базе данных.
`TEMPORARY`: Разрешает создание временных объектов в базе данных.

Для схемы:

`CREATE`: Разрешает создание новых объектов в схеме.
`USAGE`: Разрешает использование объектов в схеме.

Привилегии определяют, какие действия пользователи могут выполнять с базой данных и схемой, и они могут быть назначены на соответствующем уровне для управления доступом к данным и функциональности базы данных.

Привилегии подпрограмм

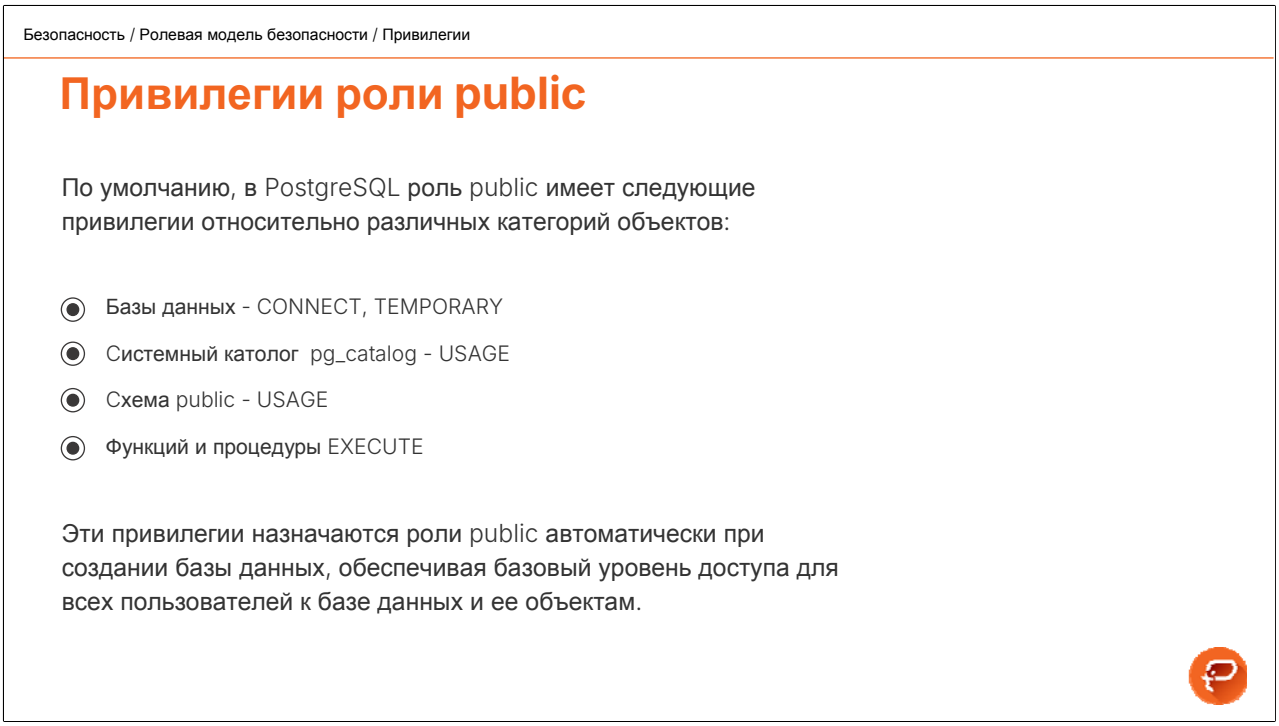
```
В PostgreSQL привилегии для подпрограмм
(например,функций и хранимых процедур)

EXECUTE: Разрешает выполнение подпрограммы.

Опция контекста безопасности имеет два значения:

SECURITY DEFINER: Выполнение подпрограммы от имени
ее владельца,обладающего привилегиями.
SECURITY INVOKER: Выполнение подпрограммы от имени
пользователя,вызывающего ее.

CREATE FUNCTION secure_function() RETURNS void AS $$
BEGIN
    -- операторы
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

Опция контекста безопасности имеет два возможных значения:

`SECURITY DEFINER`: Когда подпрограмма (функция, процедура или представление) запускается с опцией `SECURITY DEFINER`, она выполняется с привилегиями владельца объекта, а не пользователя, который вызывает эту подпрограмму. Это позволяет подпрограмме иметь доступ к объектам базы данных и выполнять операции от имени владельца, что может быть полезно для выполнения привилегированных операций.

`SECURITY INVOKER`: Когда подпрограмма запускается с опцией `SECURITY INVOKER` (по умолчанию), она выполняется с привилегиями пользователя, который вызывает эту подпрограмму. Это означает, что подпрограмма выполняется с правами доступа вызывающего пользователя, а не с привилегиями владельца объекта.

Использование `SECURITY DEFINER` следует рассматривать с осторожностью, так как это может представлять потенциальные угрозы безопасности, особенно если неправильно настроить права доступа к подпрограмме или контроль за ее выполнением.



По умолчанию, в PostgreSQL роль public имеет следующие привилегии относительно различных категорий объектов

Для баз данных:

`CONNECT`: Это разрешает пользователям подключаться к данной базе данных.
`TEMPORARY`: Позволяет создавать временные таблицы в этой базе данных.

Для схемы public:

`CREATE`: Это дает возможность пользователям создавать новые объекты (такие как таблицы, представления и прочие) в схеме public.
`USAGE`: Обеспечивает доступ к существующим объектам в схеме public.

Для системных схем (pg_catalog и information_schema):

`USAGE`: Предоставляет возможность просматривать объекты в этих системных схемах, такие как таблицы, представления и функции.

Для подпрограмм (функций и процедур):

`EXECUTE`: Это дает право на выполнение функций и процедур.

Эти привилегии назначаются роли public автоматически при создании базы данных, обеспечивая базовый уровень доступа для всех пользователей к базе данных и ее объектам.

Привилегии по умолчанию

ALTER DEFAULT PRIVILEGES - это команда в SQL, используемая в PostgreSQL для изменения привилегий по умолчанию, которые будут автоматически назначены к определенным типам объектов базы данных, создаваемым в будущем.

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public
   GRANT SELECT ON TABLES TO PUBLIC;
```

`ALTER DEFAULT PRIVILEGES` - команда в SQL, используемая в PostgreSQL для изменения привилегий по умолчанию, которые будут автоматически назначены к определенным типам объектов базы данных, создаваемым в будущем. Позволяет администраторам базы данных настроить стандартные права доступа для новых объектов, что обеспечивает более гибкое управление безопасностью данных в системе.

Назначение привилегий по умолчанию:

`ALTER DEFAULT PRIVILEGES` позволяет установить привилегии, которые будут применяться к объектам, созданным в будущем.

Привилегии по умолчанию применяются только к объектам, которые будут созданы вами или ролями, к которым вы принадлежите.

В настоящее время команда позволяет изменять привилегии по умолчанию для схем, таблиц (включая представления и внешние таблицы), последовательностей, функций и типов (включая домены).

По умолчанию привилегии для каждого типа объекта обычно предоставляют все разрешения, которые можно предоставить владельцу объекта, и могут также предоставлять некоторые привилегии для `PUBLIC`. Однако это поведение можно изменить, изменив глобальные привилегии по умолчанию с помощью `ALTER DEFAULT PRIVILEGES`.

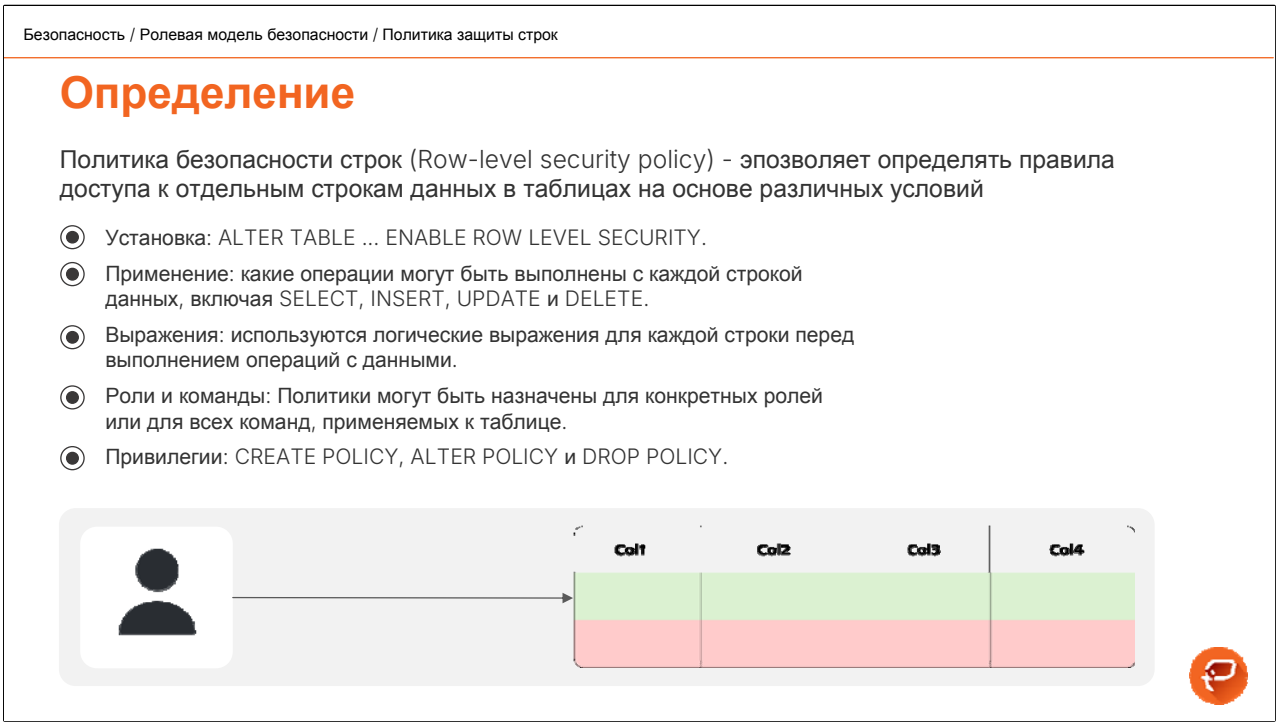
Все привилегии по умолчанию, указанные для схемы, добавляются к глобальным привилегиям по умолчанию для конкретного типа объекта. Это означает, что отзыв привилегий `REVOKE` для схемы имеет смысл только для отмены эффектов предыдущего предоставления привилегий `GRANT` для схемы.

Предположим, вы хотите автоматически предоставлять пользователям привилегию на `SELECT` для всех таблиц, созданных в будущем в схеме public. Для этого вы можете использовать следующую команду:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT SELECT ON TABLES TO PUBLIC;
```

Эта команда устанавливает привилегию `SELECT` по умолчанию для всех таблиц, созданных в будущем в схеме `public`, для роли `PUBLIC`, что позволит всем пользователям выполнять операцию `SELECT` на новых таблицах без необходимости явного назначения привилегий на каждую таблицу.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/sql-alterdefaultprivileges.html#alter-default-privileges



Политика безопасности строк (`Row-level security policy`) - это механизм в PostgreSQL, который позволяет определять правила доступа к отдельным строкам данных в таблицах на основе различных условий. Она расширяет стандартные системы управления доступом, позволяя ограничивать доступ к строкам данных в зависимости от их содержимого или контекста запроса.

Основные аспекты политики безопасности строк:

Установка: Политика безопасности строк устанавливается для конкретной таблицы с помощью команды `ALTER TABLE ... ENABLE ROW LEVEL SECURITY.`

Применение: Политика определяет, какие операции могут быть выполнены с каждой строкой данных, включая `SELECT, INSERT, UPDATE и DELETE.`

Выражения: Для определения доступа к строкам используются логические выражения, которые оцениваются для каждой строки перед выполнением операций с данными.

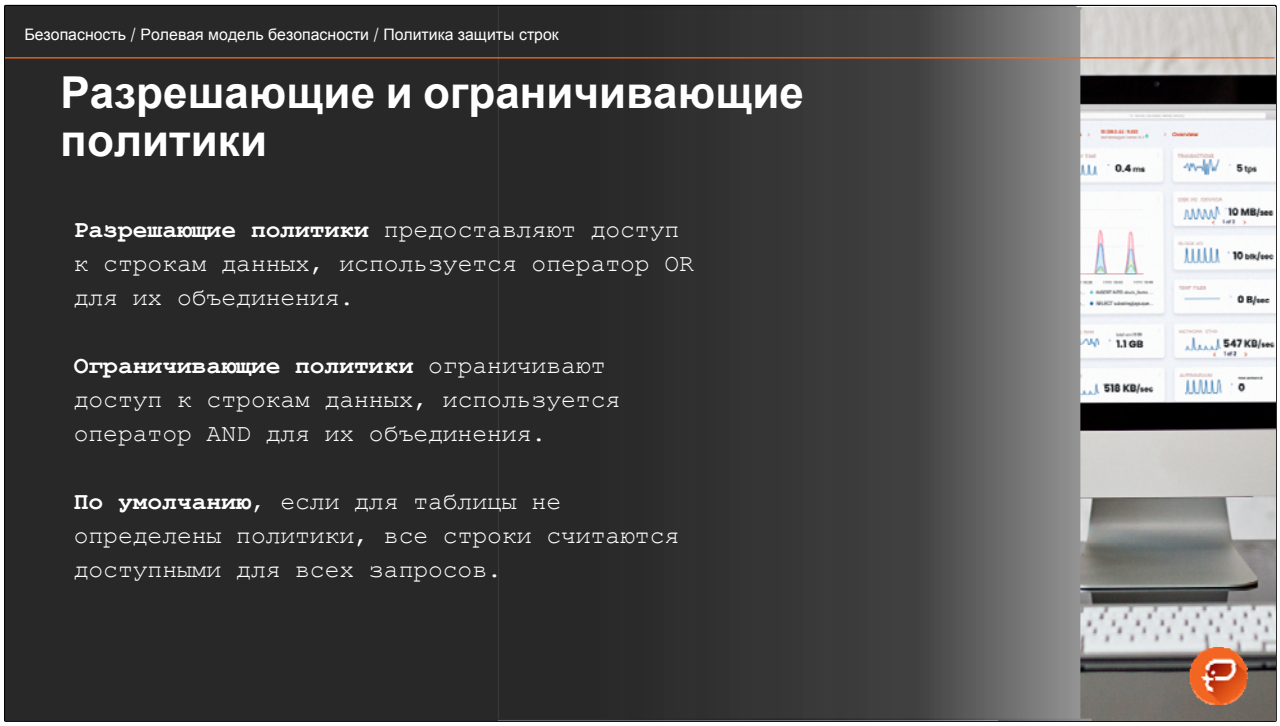
Роли и команды: Политики могут быть назначены для конкретных ролей или для всех команд, применяемых к таблице.

Привилегии: Суперпользователи и роли с атрибутом `BYPASSRLS` обходят политику безопасности строк, а владельцы таблицы могут выбирать, подвергаться ли им или нет.

Управление: Создание, изменение и удаление политик осуществляется с помощью команд `CREATE POLICY, ALTER POLICY и DROP POLICY.`

Политика безопасности строк позволяет гибко настраивать доступ к данным на уровне строк, обеспечивая более тонкое управление безопасностью в базе данных.

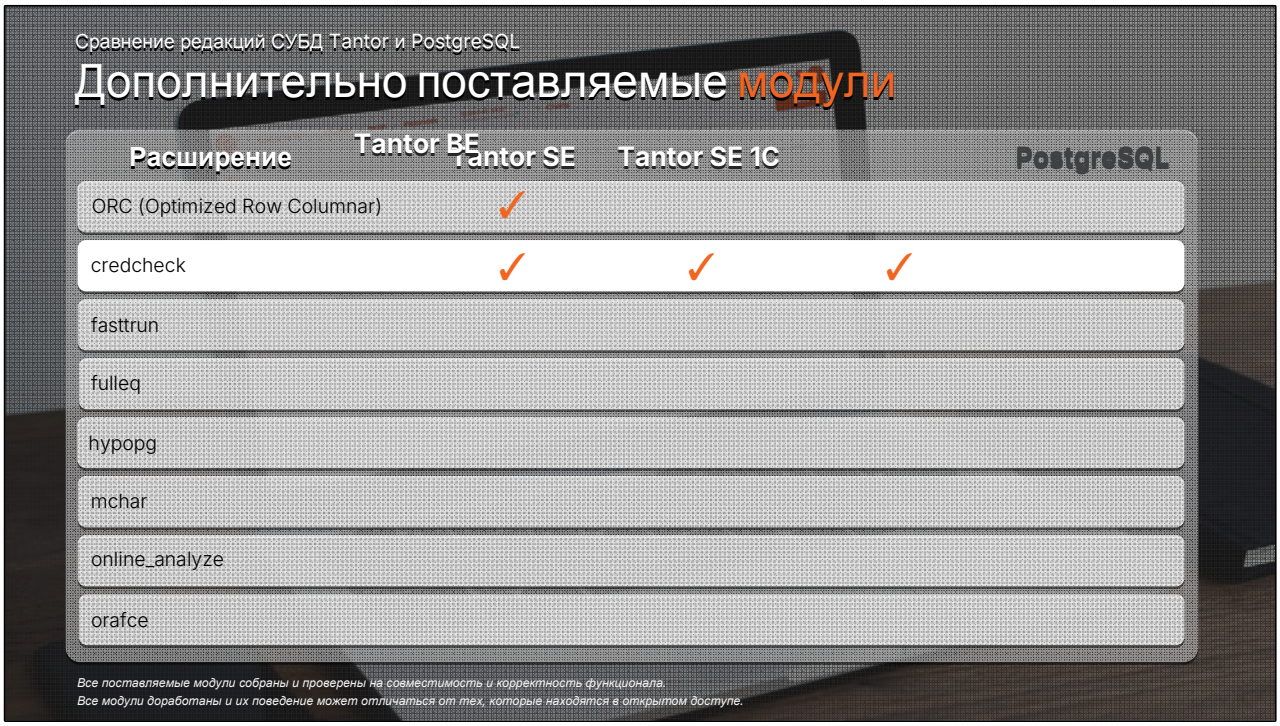
https://docs.tantorlabs.ru/tdb/ru/15_4/se/ddl-rowsecurity.html



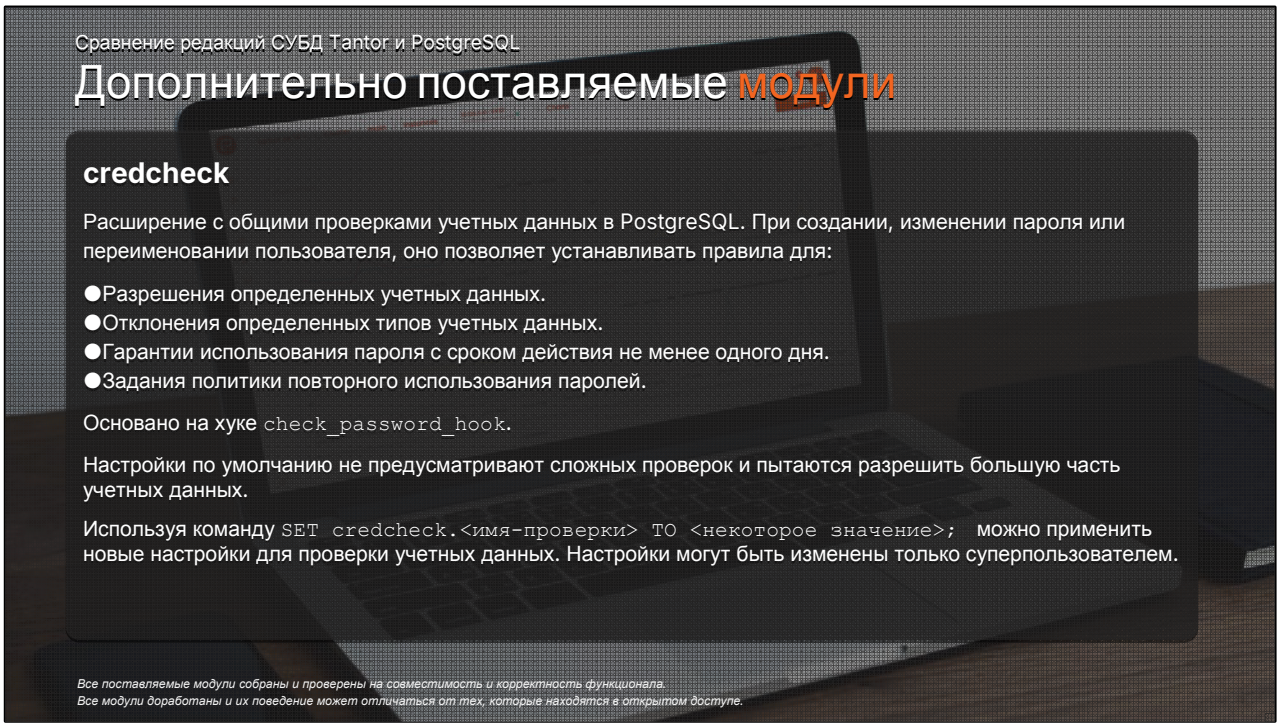
Каждая политика безопасности строк в PostgreSQL имеет уникальное имя и применяется к конкретной таблице. Это позволяет определять несколько политик для одной таблицы с различными правилами доступа. Поскольку политики применяются на уровне запроса к данным, они объединяются с использованием операторов OR или AND, в зависимости от их характера.

Каждая политика для таблицы должна иметь уникальное имя. Даже если несколько таблиц имеют одинаковые политики, имена политик в разных таблицах должны различаться.
Разрешающие и ограничивающие политики в PostgreSQL определяют, какие строки данных будут доступны для пользователей в рамках политики безопасности строк.
Разрешающие политики: Эти политики предоставляют доступ к строкам данных в таблице. Когда применяются несколько разрешающих политик к запросу, строки данных, соответствующие хотя бы одной из политик, будут доступны пользователю. При объединении разрешающих политик используется логический оператор OR, что означает, что если хотя бы одна из политик разрешает доступ к строке, она будет доступна для пользователя.
Ограничивающие политики: Эти политики ограничивают доступ к строкам данных в таблице. Если применяются несколько ограничивающих политик к запросу, строки данных, которые соответствуют всем условиям ограничений, будут доступны пользователю. При объединении ограничивающих политик используется логический оператор AND, что означает, что строка данных будет доступна только в том случае, если она удовлетворяет всем условиям всех примененных политик.

Каждая политика определяет правила доступа к строкам данных, и эти правила применяются при выполнении запросов к таблице. По умолчанию, если для таблицы не определены политики, все строки считаются доступными для всех запросов.



Модуль `credcheck` (кредчек) — доступен во всех версиях Tantor

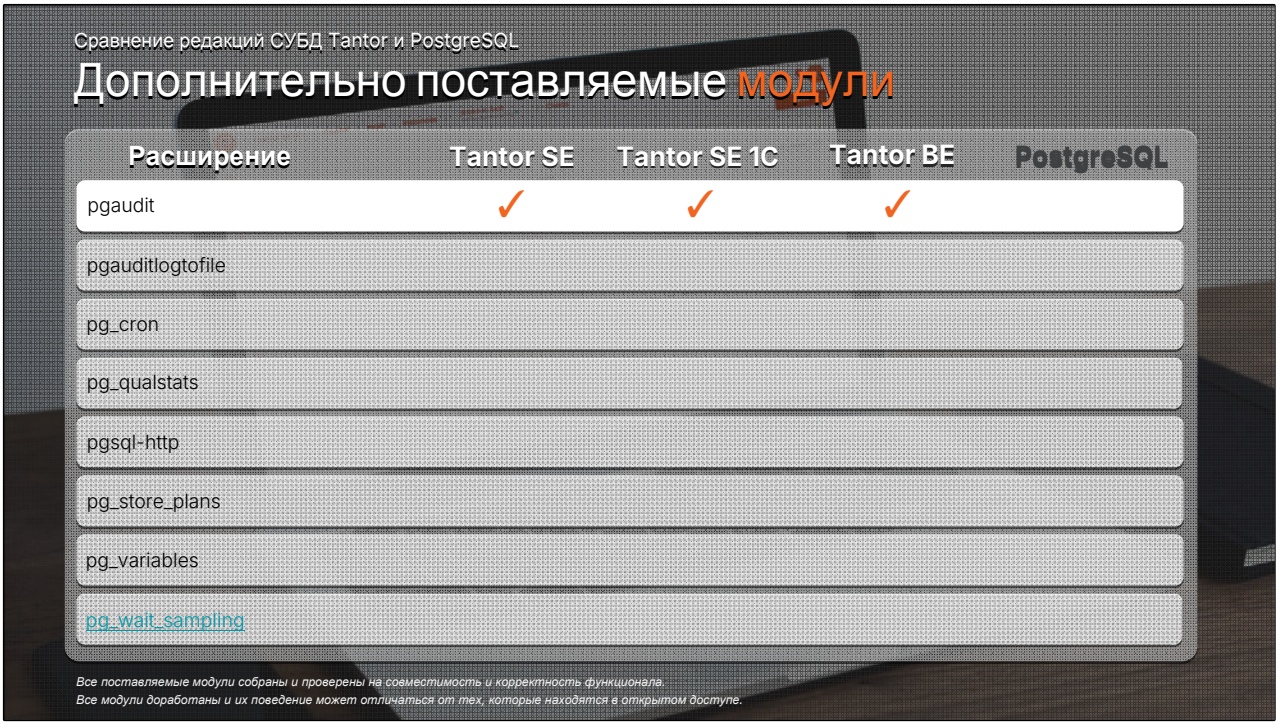


Расширение `credcheck` предоставляет несколько общих проверок учетных данных, которые выполняются при создании пользователя, при смене пароля и переименовании пользователя. Используя это расширение, можно определить набор правил:

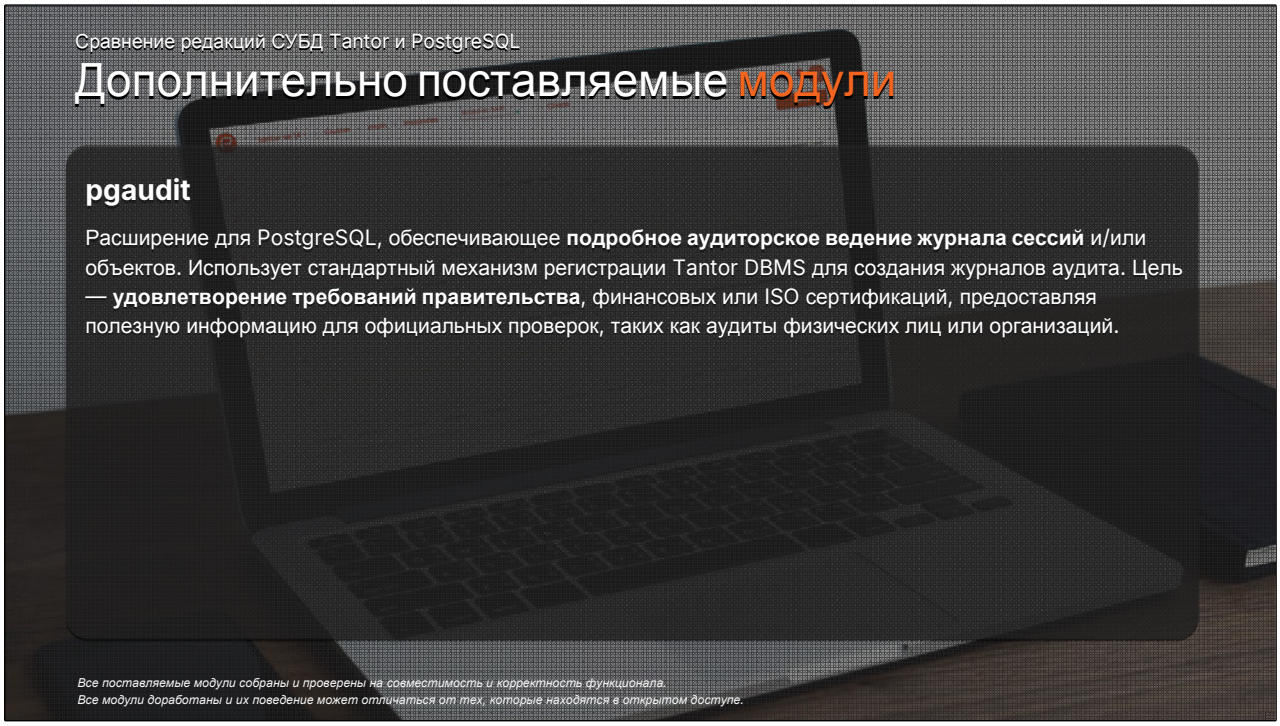
- разрешить определенный набор учетных данных
- отклонить определенный тип учетных данных
- обеспечить использование даты истечения с минимумом одного дня для пароля
- определить политику повторного использования паролей

Это расширение разработано на основе хука check_password_hook в PostgreSQL.

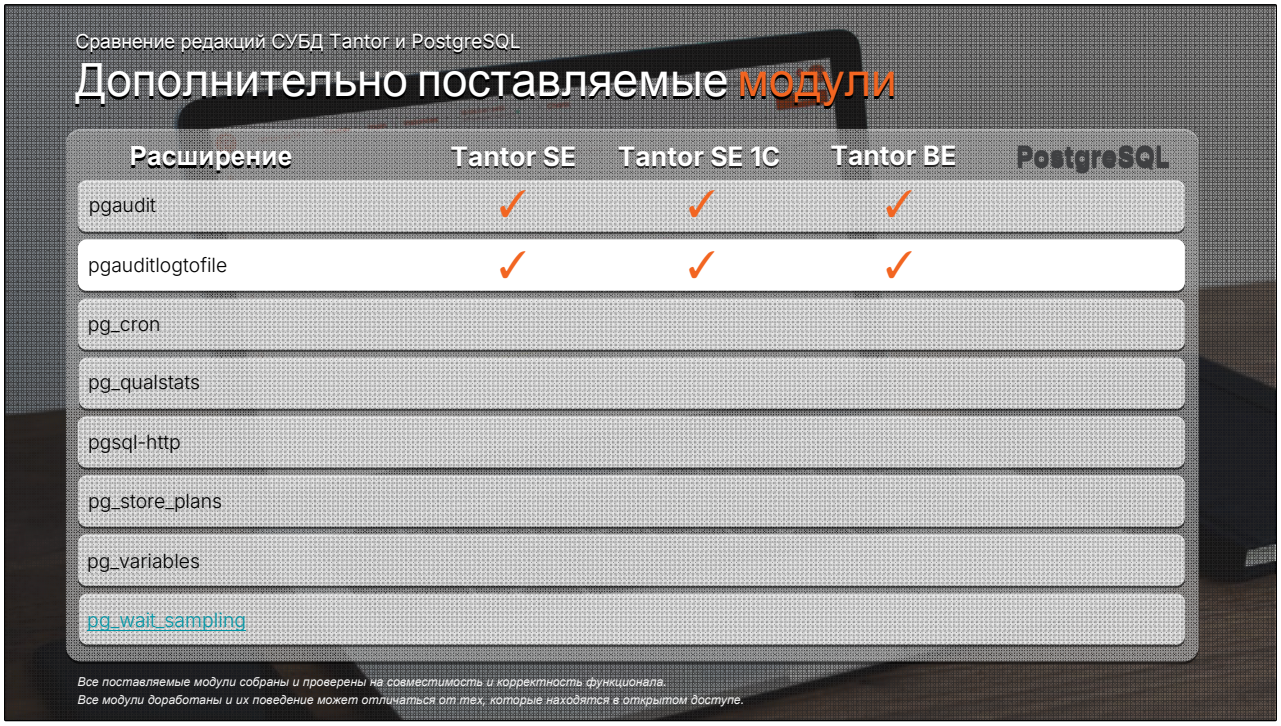
`credcheck` предоставляет все проверки в виде настраиваемых параметров. Настройки по умолчанию не предусматривают сложных проверок и пытаются разрешить большую часть учетных данных. Используя команду указанную на слайде можно применить новые настройки для проверки учетных данных. Настройки могут быть изменены только суперпользователем.



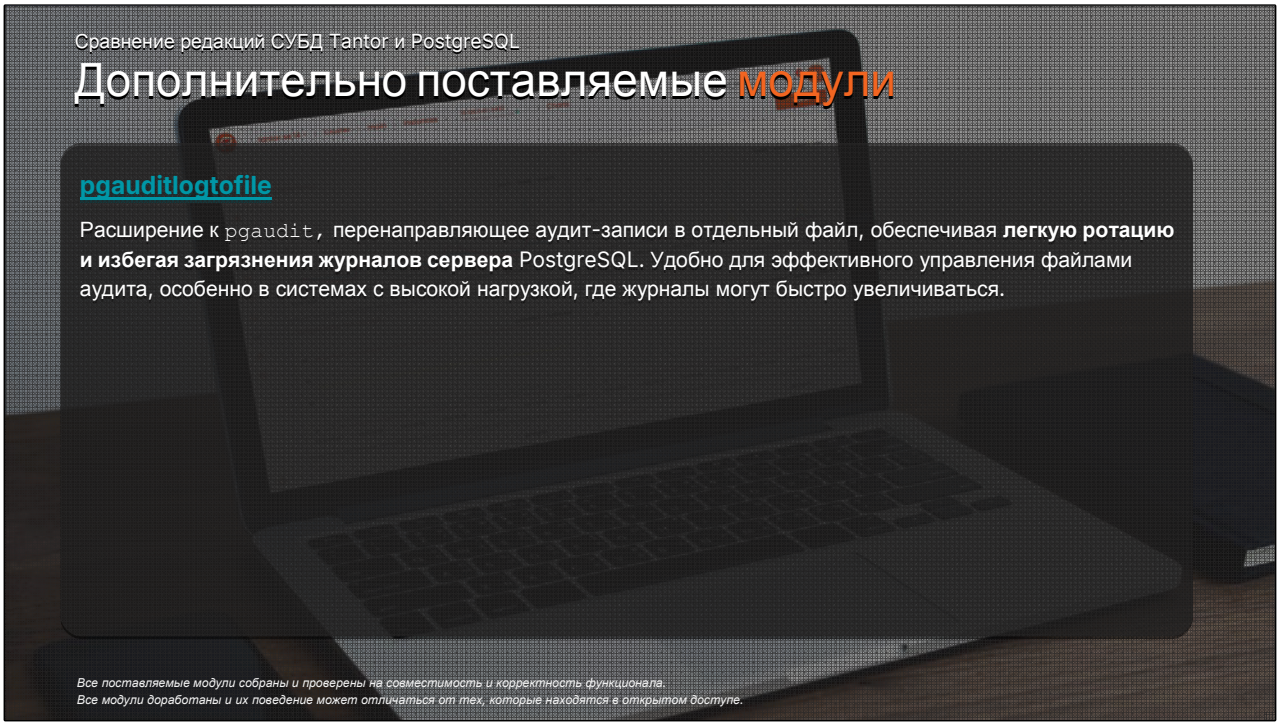
Модуль `pgaudit` — Доступен во всех версиях Tantor



Расширение `pgAudit` предоставляет детальное аудиторское ведение журнала сессий и/или объектов через стандартное средство ведения логов PostgreSQL. Обеспечивает подробное журналирование сессий и/или объектов через стандартный механизм регистрации `Tantor DBMS`. Целью pgAudit является предоставление пользователям Tantor DBMS возможности создания журналов аудита, которые часто требуются для соответствия правилам правительства, финансовым или ISO сертификациям. Информация, собранная pgAudit, правильно называется «журналом регистрации» или «журналом аудита».



Модуль `pgauditlogtofile` — Доступен во всех версиях Tantor



Дополнение к `pgAudit`, которое перенаправляет строки аудита в отдельный файл, вместо использования журналирования сервера PostgreSQL. Это позволяет нам иметь файл аудита, который можно легко ротировать, не загрязняя журналы сервера этими сообщениями. Журналы аудита в системах с высокой нагрузкой могут очень быстро расти. Это расширение позволяет автоматически ротировать файлы на основе заданного количества минут.

Демонстрация

- Создать нового пользователя
- Установить атрибуты
- Сделаем групповую роль
- Создать схему и таблицу
- Выдать роли доступ к таблице
- Удалить созданные объекты

Шаг 1. Создать новую роль

Загрузим инструмент psql

```
astra@alse-vanilla-gui:~$ sudo su - postgres

postgres@alse-vanilla-gui:~$ psql
psql (16.1)
Введите "help", чтобы получить справку.

postgres=#
```

Создадим новую роль

```
postgres=# CREATE ROLE user1;
CREATE ROLE
```

Посмотрим какие есть роли в СУБД

```
postgres=# \du
                                   Список ролей
    Имя роли    |                                Атрибуты
----------------+-------------------------------------------------------
--------------------
 anon_test_user | Суперпользователь
 pma_user       | Суперпользователь, Создаёт роли
 postgres       | Суперпользователь, Создаёт роли, Создаёт БД,
Репликация, Пропускать RLS
 replicator     | Репликация
 user1          | Вход запрещён
```

Шаг 2. Установить атрибуты

postgres=# **ALTER ROLE user1 LOGIN CREATEDB;**
ALTER ROLE

```
postgres=# \du
                                   Список ролей
    Имя роли    |                                Атрибуты
----------------+-------------------------------------------------------
--------------------
 anon_test_user | Суперпользователь
 pma_user       | Суперпользователь, Создаёт роли
 postgres       | Суперпользователь, Создаёт роли, Создаёт БД,
Репликация, Пропускать RLS
 replicator     | Репликация
 user1          | Создаёт БД
```

Шаг 3. Сделаем групповую роль.

Предположим что нам нужна роль, под которой можно только подключаться к кластеру, а под второй создавать БД но нельзя делать соединения к БД.

Создадим вторую роль

```
postgres=# CREATE ROLE user2;
CREATE ROLE
```

```
postgres=# ALTER ROLE user2 LOGIN;
ALTER ROLE
```

Отзовем лишний атрибут

```
postgres=# ALTER ROLE user1 NOLOGIN;
ALTER ROLE

postgres=# \du
                                   Список ролей
    Имя роли    |                                  Атрибуты
----------------+-----------------------------------------------------------
--------------------
 anon_test_user | Суперпользователь
 pma_user       | Суперпользователь, Создаёт роли
 postgres       | Суперпользователь, Создаёт роли, Создаёт БД,
Репликация, Пропускать RLS
 replicator     | Репликация
 user1          | Создаёт БД, Вход запрещён
 user2          |
```

Дадим право вхождения в группу user1 роли user2

```
postgres=# GRANT user1 TO user2;
GRANT ROLE
```

Проверим условия задания

Первая роль не может входить в кластер БД

```
postgres=# \c - user1
подключиться к серверу через сокет "/var/run/postgresql/.s.PGSQL.5432"
не удалось: ВАЖНО:  для роли "user1" вход запрещён
Сохранено предыдущее подключение
```

Входим под второй ролью

```
postgres=# \c - user2
Вы подключены к базе данных "postgres" как пользователь "user2"
```

Пытаемся создать базу данных под второй ролью

```
postgres=> CREATE DATABASE  dat1;
ОШИБКА:  нет прав на создание базы данных
```

Переключаем роль на первую

```
postgres=> SET ROLE user1;
SET
```

Теперь создать базу данных можно

```
postgres=> CREATE DATABASE  dat1;

CREATE DATABASE
```

Вернемся к роли user2

```
postgres=> RESET ROLE;
RESET
```

Подключаемся к БД dat1

```
dat1=> \c dat1
Вы подключены к базе данных "dat1" как пользователь "user2".
```

Шаг 4. Создать схему и таблицу

```
dat1=> CREATE SCHEMA sch1;
CREATE SCHEMA
```

Посмотрим кто владелец схемы

```
dat1=> \dn+
                                                List of schemas
 Name  |       Owner       |           Access privileges            |      Description
-------+-------------------+----------------------------------------+-----------------------
public | pg_database_owner | pg_database_owner=UC/pg_database_owner+| standard public schema
       |                   | =U/pg_database_owner                   |
sch1   | user2             |                                        |
(2 строки)
```

```
dat1=> CREATE TABLE sch1.a1 (id integer PRIMARY KEY GENERATED ALWAYS
AS IDENTITY, str text);
CREATE TABLE
```

Посмотрим описание таблицы

```
dat1=>\d sch1.a1
                              Таблица "sch1.a1"
 Столбец |   Тип   | Правило сортировки | Допустимость NULL |
По умолчанию
---------+---------+--------------------+-------------------+---------
--------------------
 id      | integer |                    | not null          |
generated always as identity
 str     | text    |                    |                   |
Индексы:
    "a1_pkey" PRIMARY KEY, btree (id)
```

Посмотрим разрешения на таблицу

```
dat1=>\dp sch1.a1
                                                    Права доступа
 Схема | Имя |   Тип   | Права доступа | Права для столбцов | Политики
-------+-----+---------+---------------+--------------------+----------
 sch1  | a1  | таблица |               |                    |
(1 строка)
```

Пока нет ни у какой роли, кроме суперпользовательской.

Шаг 5. Выдать роли доступ к таблице

Создадим еще одну роль

```
dat1=> \c - postgres
Вы подключены к базе данных "dat1" как пользователь "postgres"

dat1=# CREATE ROLE user3 LOGIN;
CREATE ROLE
```

Попробуем получить доступ к таблице a1

```
dat1=# \c - user3
Вы подключены к базе данных "dat1" как пользователь "user3".

dat1=> \dn

    Список схем
  Имя   |     Владелец
--------+------------------
 public | pg_database_owner
 sch1   | user2

(2 rows)

dat1=> SELECT * FROM sch1.a1;
ОШИБКА:  нет доступа к схеме sch1
СТРОКА 1: SELECT * FROM sch1.a1;
```

В доступе отказано нет привилегий на схему

dat1=> **\c - postgres**
Вы подключены к базе данных "dat1" как пользователь "postgres"

dat1=> **GRANT USAGE on SCHEMA sch1 TO user3;**
GRANT

```
dat1=> \dn+ sch1
               Список схем
 Имя  | Владелец | Права доступа  | Описание
------+----------+----------------+----------
 sch1 | user2    | user2=UC/user2+|
      |          | user3=U/user2  |
(1 строка)


dat1=> \c - user3
You are now connected to database "dat1" as user "user3".

dat1=> SELECT * FROM sch1.a1;
ОШИБКА:  нет доступа к таблице a1
```

Теперь отказ из-за отсутствия привилегий на таблице a1

dat1=> **\c - postgres**
Вы подключены к базе данных "dat1" как пользователь "postgres"

```
dat1=> GRANT SELECT, INSERT (str) ON TABLE sch1.a1 to user3;
```

```
GRANT

dat1=> \dp sch1.a1
                                Права доступа
 Схема | Имя |   Тип   |    Права доступа    | Права для столбцов |
Политики
-------+-----+---------+---------------------+--------------------+---
-------
 sch1  | a1  | таблица | user2=arwdDxt/user2+| str:               +|
       |     |         | user3=r/user2       |   user3=a/user2    |
(1 строка)


dat1=> \c - user3
Вы подключены к базе данных "dat1" как пользователь "user3"

dat1=> SELECT * FROM sch1.a1;
id | str
----+-----
(0 строк)
```

Теперь все в порядке. Доступ предоставлен в рамках выданных привилегий.

Проверим вставку в столбец

```
dat1=> INSERT INTO sch1.a1 (str) VALUES ('первая запись');
INSERT 0 1

dat1=> SELECT * FROM sch1.a1;
id |      str
----+---------------
 1 | первая запись
(1 row)
```

Проверим вставку в первый столбец

```
dat1=> INSERT INTO sch1.a1 OVERRIDING SYSTEM VALUE values (2);
ОШИБКА:  нет доступа к таблице a1
```

Не хватает привилегий

Удаление также строк и объекта невозможно - нужно быть владельцем или суперпользователем

```
dat1=> DELETE FROM sch1.a1;
ОШИБКА:  нет доступа к таблице a1
dat1=> DROP TABLE sch1.a1;
ОШИБКА:  нужно быть владельцем таблицы a1
```

Шаг 6. Удалить созданные объекты

Удалим схему

```
dat1=> \c - user2
Вы подключены к базе данных "dat1" как пользователь "user2".

dat1=> DROP SCHEMA sch1;
```

```
ОШИБКА:  удалить объект схема sch1 нельзя, так как от него зависят
другие объекты
ПОДРОБНОСТИ:  таблица sch1.a1 зависит от объекта схема sch1
ПОДСКАЗКА:  Для удаления зависимых объектов используйте DROP ...
CASCADE.
```

Схема не пуста, можно удалить каскадом

```
dat1=> DROP SCHEMA sch1 CASCADE;
ЗАМЕЧАНИЕ:  удаление распространяется на объект таблица sch1.a1
DROP SCHEMA
```

переключимся на другую базу данных и и удалим dat1

```
dat1=> \c postgres
Вы подключены к базе данных "postgres" как пользователь "user2".

postgres=> DROP DATABASE dat1 (force);
DROP DATABASE
```

Для удаления ролей воспользуемся суперпользовательской ролью

```
postgres=> \c - postgres
Вы подключены к базе данных "postgres" как пользователь "postgres".

postgres=# DROP ROLE user1, user2, user3;
DROP ROLE
```

Практическая работа

- Создать нового пользователя
- Установить атрибуты
- Сделаем групповую роль
- Создать схему и таблицу
- Выдать роли доступ к таблице
- Удалить созданные объекты

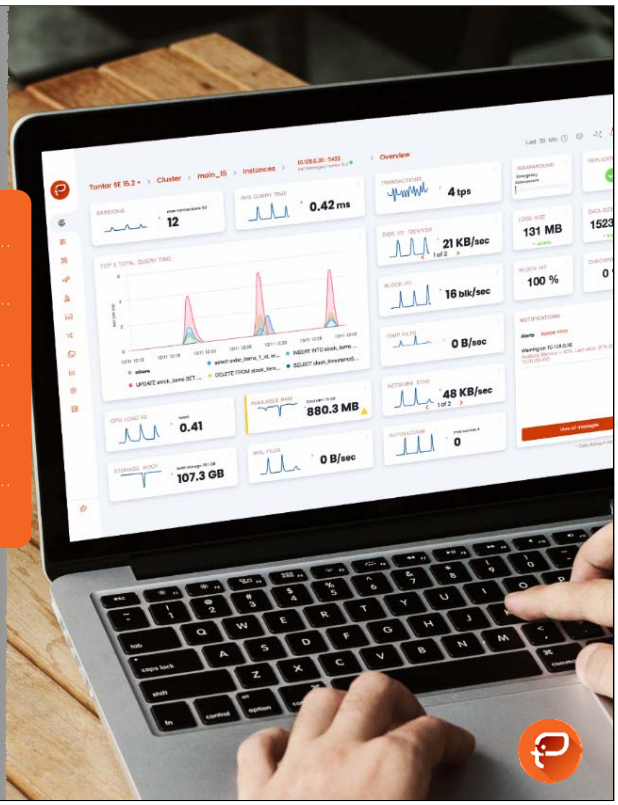
Дополнительное задание:

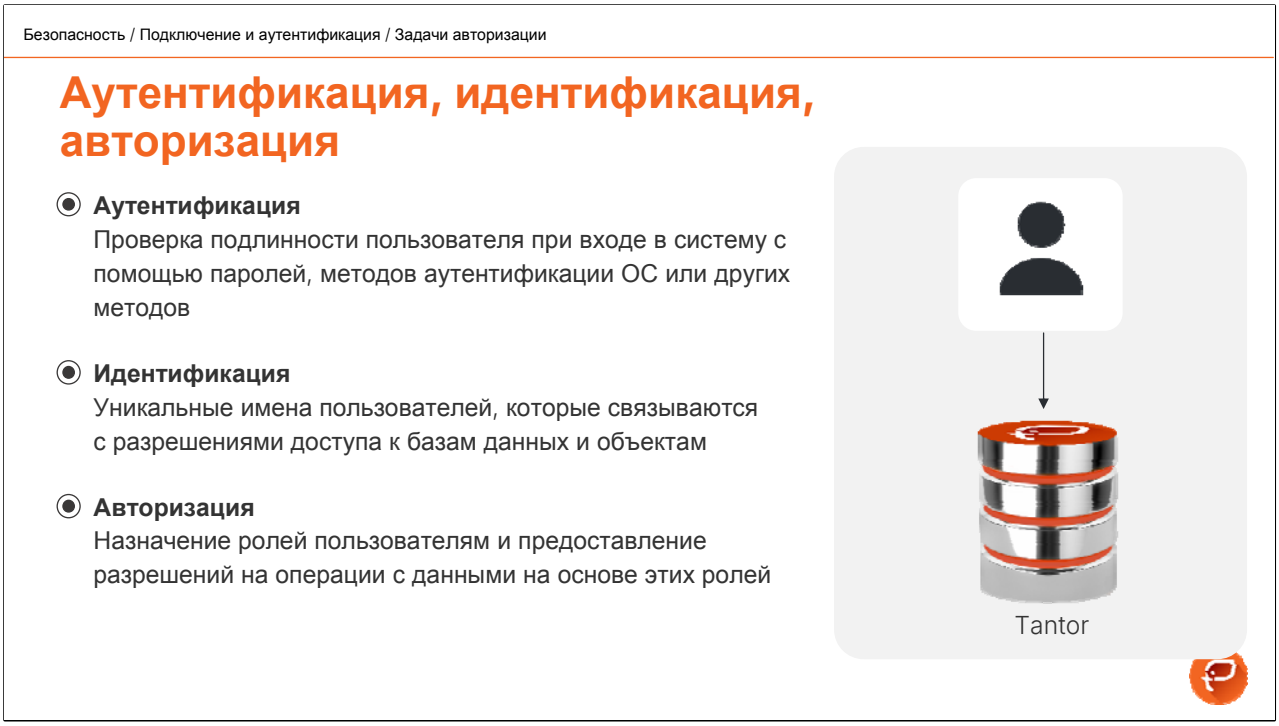
Сделать задание 1-6 с помощью pgAdmin

Темы

- Ролевая модель безопасности
- Подключение и аутентификация

- Задачи авторизации
- Конфигурирование
- pg_hba
- pg_ident
- Демонстрация
- Практическая работа





Прежде чем разберемся как устроено подключение к СУБД PostgreSQL, давайте более внимательно рассмотрим три задачи, а именно Аутентификация, идентификация, авторизация

1 Идентификация в PostgreSQL:

В PostgreSQL идентификация обычно осуществляется с помощью уникальных имен пользователей. Каждый пользователь в базе данных имеет свой собственный уникальный идентификатор, который используется для связывания с соответствующими разрешениями и правами доступа к базам данных и их объектам. Идентификация также может включать другие атрибуты, такие как пароль или метод аутентификации операционной системы, чтобы подтвердить личность пользователя.

2. Аутентификация в PostgreSQL:

В PostgreSQL аутентификация может осуществляться различными способами, включая использование паролей, методов аутентификации операционной системы, а также других методов, таких как аутентификация `GSSAPI (Generic Security Services Application Programming Interface)`. При аутентификации пользователь предоставляет учетные данные, которые затем сравниваются с данными, хранящимися в базе данных, для проверки их подлинности.

3. Авторизация в PostgreSQL:

Авторизация в PostgreSQL осуществляется с помощью назначения ролей пользователям и предоставления соответствующих разрешений на выполнение определенных операций с базами данных, таблицами, представлениями и другими объектами. После того как пользователь успешно прошел процесс аутентификациии, система определяет его права доступа на основе назначенных ему ролей и правил авторизации.

Безопасность / Подключение и аутентификация / Конфигурирование

Файлы конфигурации. Где найти?

Файл pg_hba.conf используется для определения правил аутентификации

Расположение

show hba_file;

```
               hba_file
-----------------------------------------------------
 /var/lib/postgresql/tantor-se-16/data/pg_hba.conf
```

Вы можете открыть этот файл с помощью текстового редактора для просмотра и изменения правил аутентификации, определенных в нем

Файл pg_hba.conf является одним из основных файлов конфигурации PostgreSQL. Он используется для определения правил аутентификации, которые управляют тем, как клиенты могут подключаться к серверу PostgreSQL и как их идентификация будет проверяться. Этот файл определяет различные параметры аутентификации, такие как методы аутентификации, адреса клиентов, базы данных, для которых правило действительно, и пользователи, которые могут использовать это правило.
Поскольку файл `pg_hba.conf` играет ключевую роль в настройке безопасности и аутентификации в PostgreSQL, внесение изменений в этот файл должно производиться осторожно, чтобы избежать уязвимостей и обеспечить безопасность базы данных.

```
postgres=# show hba_file;
                    hba_file
-----------------------------------------------------
 /var/lib/postgresql/tantor-se-16/data/pg_hba.conf
```

Команда SHOW hba_file; в PostgreSQL выводит путь к файлу pg_hba.conf, который используется для настройки правил аутентификации в PostgreSQL. В вашем случае путь к файлу `pg_hba.conf` указан как `/var/lib/postgresql/tantor-se-16/data/pg_hba.conf`.
Таким образом, файл pg_hba.conf для вашего экземпляра PostgreSQL находится в указанном пути. Вы можете открыть этот файл с помощью текстового редактора для просмотра и изменения правил аутентификации, определенных в нем.

Чтение файла конфигурации

Посмотреть правила подключения можно с помощью представления

```
select rule_number, type, database, user_name, auth_method from
pg_hba_file_rules();
 rule_number | type  |   database    | user_name  | auth_method
-------------+-------+---------------+------------+-------------
           1 | local | {all}         | {pma_user} | md5
           2 | local | {all}         | {all}      | trust
           3 | host  | {all}         | {all}      | trust
           4 | host  | {all}         | {all}      | trust
           5 | local | {replication} | {all}      | trust
           6 | host  | {replication} | {all}      | trust
           7 | host  | {replication} | {all}      | trust
```

```
 postgres=# select rule_number, type, database, user_name, auth_method from
pg_hba_file_rules();
  rule_number | type  |   database    | user_name  | auth_method
 -------------+-------+---------------+------------+-------------
            1 | local | {all}         | {pma_user} | md5
            2 | local | {all}         | {all}      | trust
            3 | host  | {all}         | {all}      | trust
            4 | host  | {all}         | {all}      | trust
            5 | local | {replication} | {all}      | trust
            6 | host  | {replication} | {all}      | trust
            7 | host  | {replication} | {all}      | trust
```

Этот вывод показывает содержимое файла `pg_hba.conf` для вашего экземпляра PostgreSQL. Давайте разберем каждую строку:

`rule_number` (Номер правила): Это порядковый номер правила в файле pg_hba.conf.

`type` (Тип соединения): Определяет тип подключения, для которого действует данное правило. В вашем случае есть два типа: local (для локальных соединений) и host (для удаленных соединений).

`database` (База данных): Указывает, для каких баз данных действует данное правило. `{all}` указывает, что правило применяется ко всем базам данных.

`user_name` (Имя пользователя): Определяет, для каких пользователей действует данное правило. `{all}` указывает, что правило применяется ко всем пользователям. В случае строк типа local указано конкретное имя пользователя, например, `{pma_user}`.

`auth_method` (Метод аутентификации): Указывает метод аутентификации, который будет использоваться для проверки подлинности клиента. Например, `md5` используется для парольной аутентификации, а `trust` означает, что подключение разрешено без проверки пароля.

Эти правила определяют, какие пользователи и базы данных могут получить доступ к вашему серверу PostgreSQL и с какими методами аутентификации. Важно правильно настроить эти правила для обеспечения безопасности и функциональности вашей базы данных.

Также в представлении есть и другие поля

`file_name` (Имя файла): Это имя файла `pg_hba.conf`, в котором определено данное правило.

`line_number` (Номер строки): Это номер строки в файле `pg_hba.conf`, где определено данное правило.

`address` (Адрес клиента): Это IP-адрес или доменное имя клиента, для которого применяется правило. Обычно используется в правилах типа host.

`netmask` (Маска подсети): Это маска подсети, которая определяет диапазон IP-адресов, для которых действует правило. Обычно используется в правилах типа host.

`options` (Дополнительные опции): Это дополнительные параметры правила, которые могут использоваться для настройки дополнительных настроек аутентификации.

`error` (Ошибка): Это текстовое описание любой ошибки, возникшей при обработке данного правила.

Поля аутентификации

Тип соединения (type): Определяет, является ли правило для локальных или удаленных соединений.

База данных (database): Указывает, для каких баз данных действует данное правило.

Имя пользователя (user_name): Определяет, для каких пользователей действует данное правило.

Адрес клиента (address): Это IP-адрес или доменное имя клиента, для которого применяется правило.

В файле конфигурации PostgreSQL `pg_hba.conf` определяются правила аутентификации, которые управляют тем, как клиенты могут получить доступ к серверу PostgreSQL и как будет проверяться их подлинность. Эти правила играют ключевую роль в обеспечении безопасности базы данных, позволяя администраторам гибко настраивать доступ к различным базам данных и определять методы аутентификации для различных пользователей и клиентских соединений.

Разбор полей правил аутентификации в файле `pg_hba.conf`:

`type` (Тип соединения):

- Поле определяет тип соединения, для которого действует данное правило аутентификации:
- `local`: Определяет правило для локальных соединений к PostgreSQL. Это могут быть соединения через UNIX-сокеты или локальные TCP-соединения.
- `host`: Определяет правило для удаленных соединений к PostgreSQL. Это могут быть соединения через TCP/IP сеть. Когда указывается тип host, также используются другие поля, такие как address, чтобы указать, какие клиентские адреса допускаются.

`database` (База данных):

- Поле определяет для какой базы данных действует данное правило аутентификации. Может содержать конкретное имя базы данных или быть установлено как `{all}`, что означает, что правило применяется ко всем базам данных.

`user_name` (Имя пользователя):

- Поле определяет, для каких пользователей действует данное правило аутентификации. Может содержать конкретное имя пользователя или быть установлено как `{all}`, что означает, что правило применяется ко всем пользователям.

`address` (Адрес клиента):

- Поле указывает IP-адрес или доменное имя клиента, для которого применяется данное правило аутентификации. Обычно используется только в правилах типа `host`, чтобы указать, с каких адресов разрешено соединение с сервером PostgreSQL.

Заключение:

Понимание каждого из этих полей помогает администраторам настраивать безопасность и функциональность своей базы данных в соответствии с требованиями безопасности и доступа к данным. Правильная настройка правил аутентификации в файле `pg_hba.conf` является важным шагом для обеспечения безопасности и надежности PostgreSQL сервера.

Протоколы аутентификации внутренние

- trust (доверие): Позволяет подключаться к базе данных без запроса пароля,
- reject (отказ): Принудительно отклоняет все попытки подключения к базе данных
- md5: Подобно методу пароля, пользователь должен предоставить пароль для аутентификации, однако он передается в зашифрованном виде, используя алгоритм MD5 для хеширования пароля.
- scram-sha-256 : использует алгоритм HMAC для аутентификации, который основан на комбинации хэширования SHA-256 и случайного набора символов (соль).
- ident: отправляет запрос на сервер идентификации (обычно это служба ident) на клиентском компьютере для проверки, есть ли у пользователя разрешение на подключение к базе данных.

Протоколы аутентификации внутренние в контексте базы данных PostgreSQL описывают способы проверки подлинности пользователей и клиентских соединений, которые выполняются непосредственно на уровне PostgreSQL, без использования внешних систем аутентификации. Вот некоторые из внутренних протоколов аутентификации, поддерживаемых в PostgreSQL:

1. `trust` (доверие):

Позволяет подключаться к базе данных без запроса пароля, что делает его ненадежным для использования в любом окружении, где безопасность имеет значение.

2. `reject` (отказ):

Отклоняет все попытки подключения к базе данных без каких-либо дополнительных проверок или запросов к пользователям, предназначенным для временного отключения доступа или предотвращения всех попыток подключения.

3. `md5` (MD5-хеширование пароля):

Подобно методу пароля, пользователь должен предоставить пароль для аутентификации, однако он передается в зашифрованном виде, используя алгоритм MD5 для хеширования пароля. Это более безопасный метод, чем простое хранение пароля в зашифрованном виде.

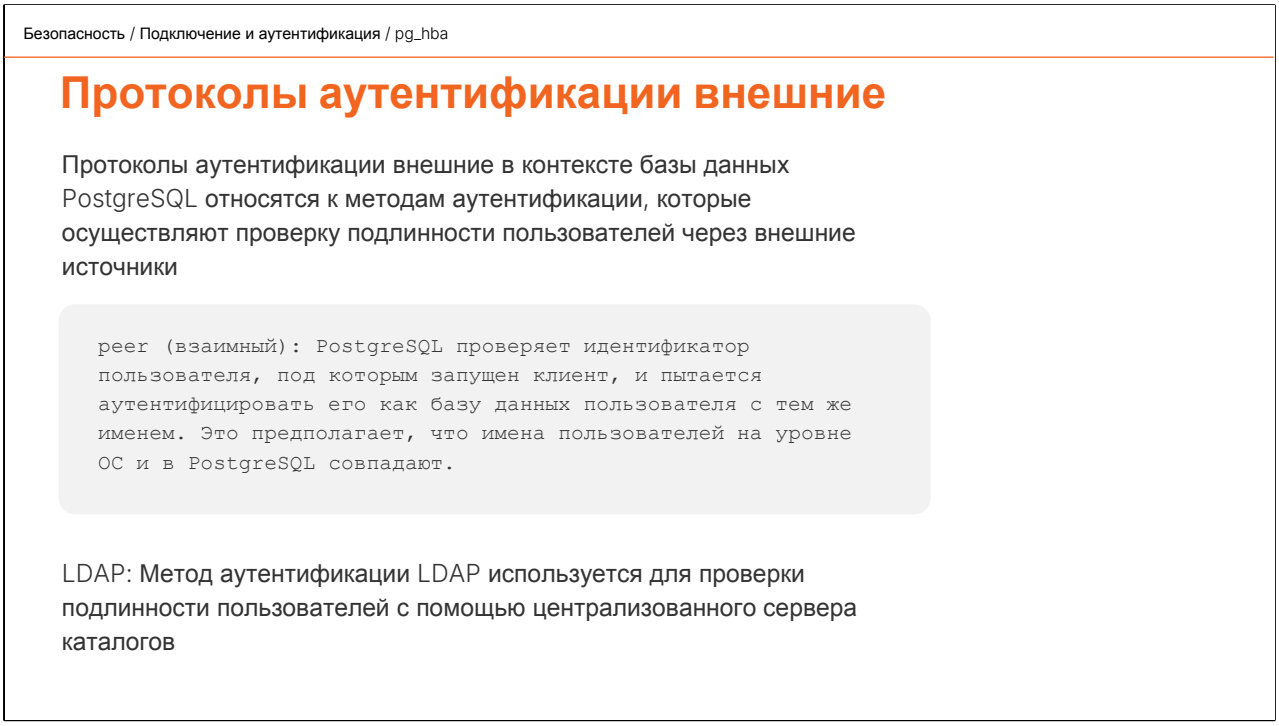
4. `scram-sha-256` (Salted Challenge Response Authentication Mechanism):

Этот метод аутентификации предлагает более безопасное и современное решение, чем md5. Он использует алгоритм HMAC (Hash-based Message Authentication Code) для аутентификации, который основан на комбинации хэширования SHA-256 и случайного набора символов (соль).

5. `ident`:

●При использовании этого метода PostgreSQL отправляет запрос на сервер идентификации (обычно это служба ident) на клиентском компьютере для проверки, есть ли у пользователя разрешение на подключение к базе данных. Этот метод часто используется в ситуациях, где необходимо подтверждение идентичности пользователя на удаленной системе.

Эти внутренние протоколы аутентификации обеспечивают базовый уровень безопасности и контроля доступа к данным в PostgreSQL. Они позволяют администраторам баз данных выбирать подходящий метод в зависимости от требований безопасности и удобства использования.



Протоколы аутентификации внешние в контексте базы данных PostgreSQL относятся к методам аутентификации, которые осуществляют проверку подлинности пользователей через внешние источники, такие как операционная система или централизованные службы аутентификации, такие как LDAP или Kerberos. Вот несколько примеров внешних протоколов аутентификации, поддерживаемых в PostgreSQL:

`peer` (взаимный):

Метод аутентификации используется для локальных подключений к серверу PostgreSQL на основе операционной системы. PostgreSQL проверяет идентификатор пользователя, под которым запущен клиент, и пытается аутентифицировать его как базу данных пользователя с тем же именем. Это предполагает, что имена пользователей на уровне ОС и в PostgreSQL совпадают.

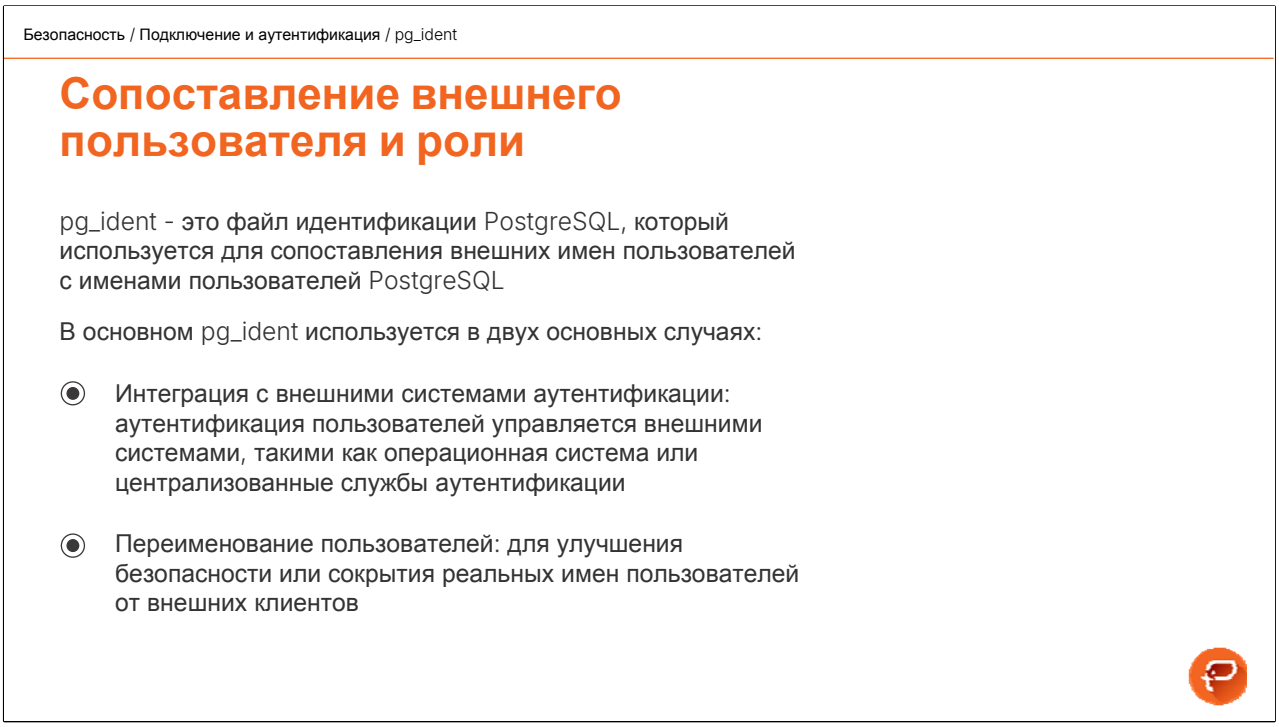
`LDAP` (Lightweight Directory Access Protocol):

Метод аутентификации `LDAP` используется для проверки подлинности пользователей с помощью централизованного сервера каталогов, который поддерживает протокол `LDAP`. PostgreSQL может взаимодействовать с сервером `LDAP` для проверки имени пользователя и пароля перед предоставлением доступа к базе данных.

`Kerberos`:

Протокол аутентификации `Kerberos` обеспечивает безопасную аутентификацию на основе обмена ключами между клиентом и сервером. При использовании этого метода PostgreSQL взаимодействует с сервером Kerberos для проверки подлинности пользователей перед разрешением доступа к базе данных.

Внешние протоколы аутентификации обеспечивают более гибкие и безопасные методы проверки подлинности пользователей в PostgreSQL, чем внутренние методы, такие как пароль или md5. Они позволяют интегрировать базу данных PostgreSQL в существующие системы аутентификации и централизованные службы управления идентификацией.



`pg_ident` - это файл идентификации PostgreSQL, который используется для сопоставления внешних имен пользователей с именами пользователей PostgreSQL. Этот файл позволяет настраивать соответствие между внешними именами пользователей, которые поступают от клиентов, и именами пользователей PostgreSQL, которые используются для аутентификации и авторизации на сервере баз данных.

В основном `pg_ident` используется в двух основных случаях:
Интеграция с внешними системами аутентификации:
Когда PostgreSQL используется в окружениях, где аутентификация пользователей управляется внешними системами, такими как операционная система или централизованные службы аутентификации (например, `LDAP или Kerberos`), `pg_ident` позволяет сопоставить идентификационные имена, используемые этими системами, с именами пользователей PostgreSQL.

Переименование пользователей:
`pg_ident` также может использоваться для переименования пользователей PostgreSQL для улучшения безопасности или сокрытия реальных имен пользователей от внешних клиентов. Например, вы можете использовать `pg_ident` для сопоставления внешнего имени "`webuser`" с реальным именем пользователя PostgreSQL "`app_user`".

В обоих случаях `pg_ident` дает администраторам баз данных гибкость настройки аутентификации и авторизации пользователей в PostgreSQL в соответствии с требованиями и стандартами безопасности и интеграции с внешними системами.

Пример

К примеру хотим чтобы локальный пользователь astra мог подключаться только как user1

Файл pg_hba.conf

```
# TYPE  DATABASE      USER        ADDRESS
METHOD
local   all         all                     ident   map=map1
```

Файл pg_ident.conf

```
# MAPNAME     SYSTEM-USERNAME      PG-USERNAME
map1   astra  user1
```

Давайте подробно рассмотрим ваш пример.

Файл `pg_hba.conf`:

```
# TYPE   DATABASE         USER             ADDRESS                  METHOD
local    all              all                                       ident
map=map1
```

Эта строка в файле `pg_hba.conf` означает, что для всех баз данных и всех пользователей, пытающихся подключиться локально (через сокеты UNIX), будет использоваться метод аутентификации "`ident`" с использованием маппинга `map1`.

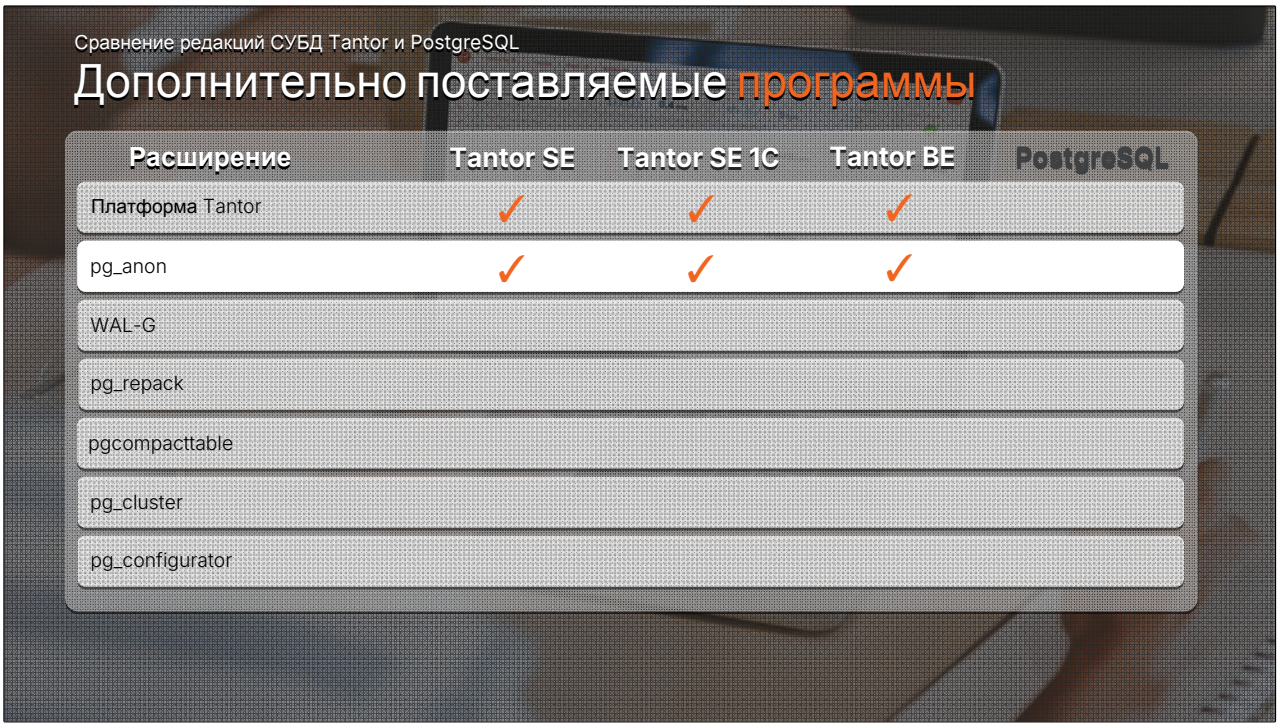
Файл `pg_ident.conf`:

```
# MAPNAME        SYSTEM-USERNAME          PG-USERNAME
map1            astra                   user1
```

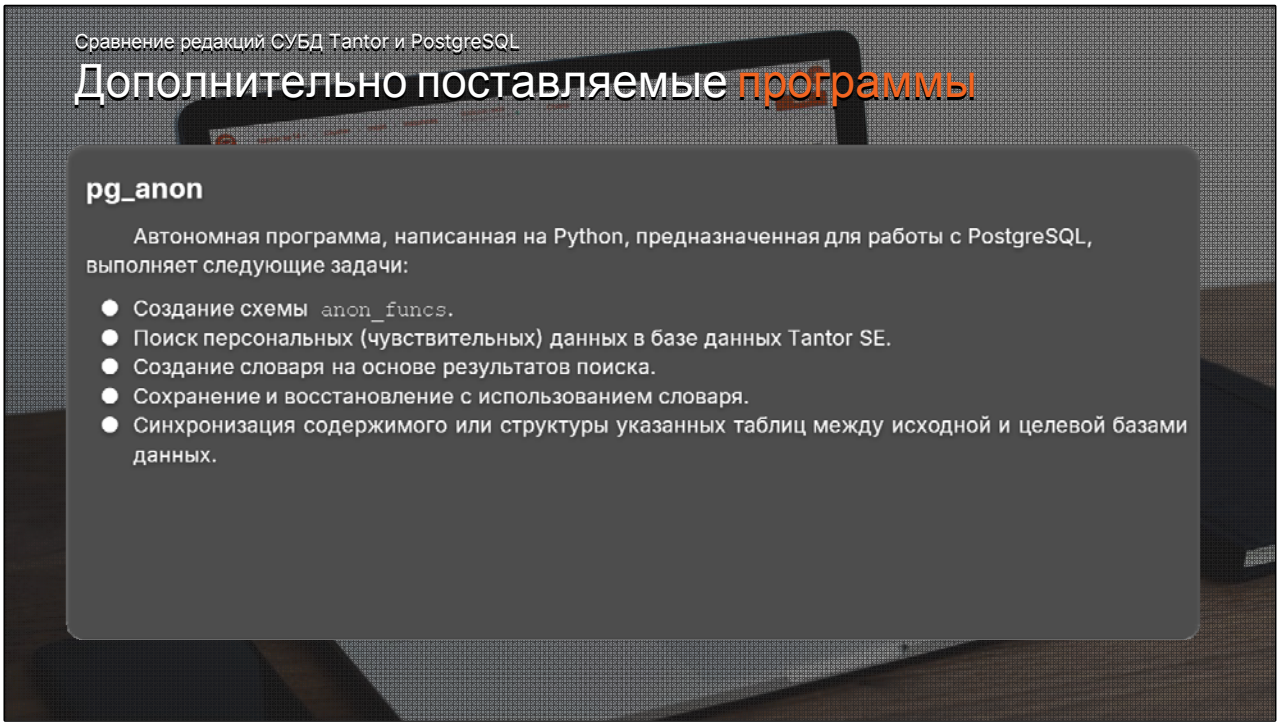
Эта строка в файле pg_ident.conf определяет маппинг map1, который говорит PostgreSQL, что внешний пользователь "astra" должен быть аутентифицирован как пользователь PostgreSQL "`user1`".

Теперь, когда локальный пользователь "`astra`" пытается подключиться к PostgreSQL, сервер PostgreSQL смотрит на файл `pg_ident.conf`, чтобы узнать, какой пользователь PostgreSQL должен быть сопоставлен с внешним пользователем "`astra`". В данном случае, внешнее имя пользователя "`astra`" будет сопоставлено с пользователем PostgreSQL "`user1`".

Таким образом, локальный пользователь "`astra`" сможет подключиться к базе данных только как пользователь PostgreSQL "`user1`".



Во всех версиях СУБД Tantor доступен `pg_anon`

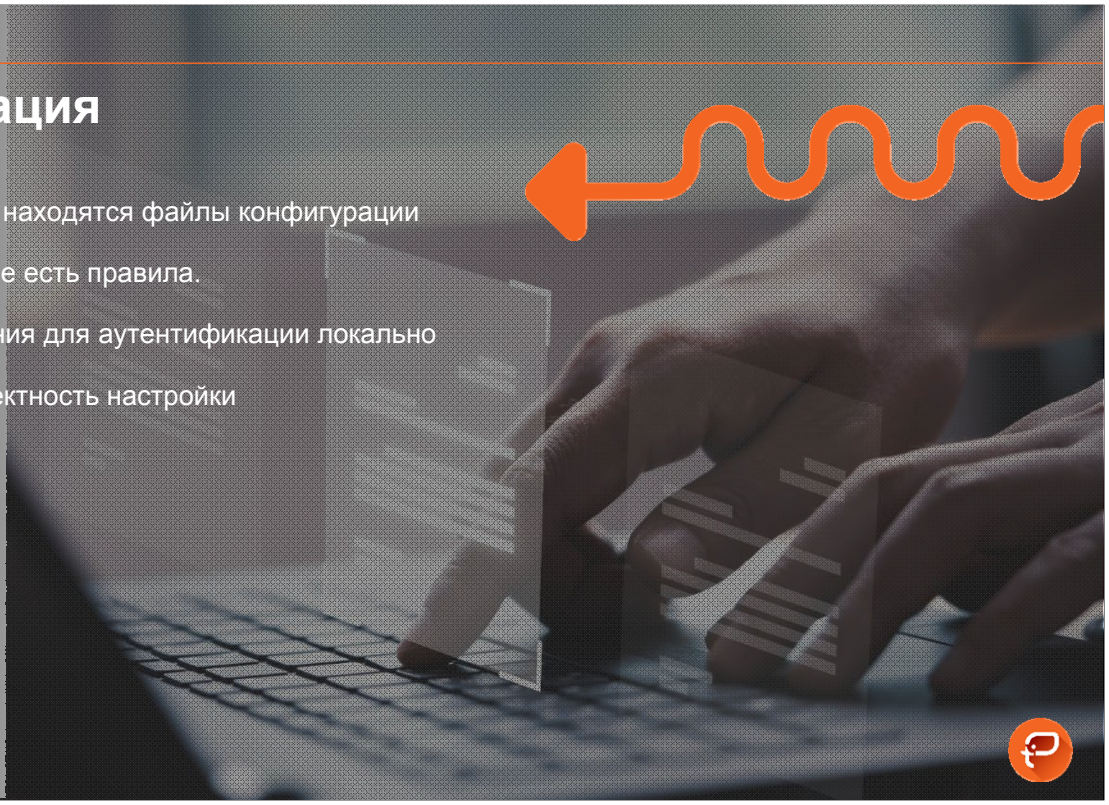


`pg_anon` — это автономная программа, написанная на Python, предназначенная для работы с PostgreSQL (начиная с версии 9.6 и выше), которая выполняет следующие задачи:

- Создание схемы `anon_funcs`, которая содержит библиотеку функций для анонимизации
- Поиск персональных (чувствительных) данных в базе данных `Tantor SE` на основе мета-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

Демонстрация

- Посмотреть где находятся файлы конфигурации
- Посмотрим какие есть правила.
- Внесем изменения для аутентификации локально
- Проверим корректность настройки



Шаг 1. Посмотреть где находятся файлы конфигурации
Загрузим инструмент psql

```
astra@alse-vanilla-gui:~$ sudo su - postgres

postgres@alse-vanilla-gui:~$ psql

psql (16.1)
Введите "help", чтобы получить справку.
```

Посмотрим место расположение конфигурационного файла.

```
postgres=# SHOW hba_file;
                    hba_file
-----------------------------------------------
/var/lib/postgresql/tantor-se-16/data/pg_hba.conf
(1 строка)
```

можно посмотреть правила подключения с помощью представления pg_hba_file_rules

postgres=# **\d pg_hba_file_rules;**

```
                Представление "pg_catalog.pg_hba_file_rules"
   Столбец    |   Тип    | Правило сортировки | Допустимость NULL | По
умолчанию
--------------+---------+--------------------+-------------------+------------
--
 rule_number | integer |                    |                   |
 file_name   | text    |                    |                   |
 line_number | integer |                    |                   |
 type        | text    |                    |                   |
 database    | text[]  |                    |                   |
 user_name   | text[]  |                    |                   |
 address     | text    |                    |                   |
 netmask     | text    |                    |                   |
 auth_method | text    |                    |                   |
 options     | text[]  |                    |                   |
 error       | text    |                    |                   |
```

Шаг 2. Посмотрим какие есть правила.

```
postgres=# SELECT rule_number, type, database, user_name, auth_method FROM
pg_hba_file_rules();
rule_number | type  |   database    | user_name  | auth_method
------------+-------+---------------+------------+-------------
           1 | local | {all}         | {pma_user} | md5
           2 | local | {all}         | {all}      | trust
           3 | host  | {all}         | {all}      | trust
           4 | host  | {all}         | {all}      | trust
           5 | local | {replication} | {all}      | trust
           6 | host  | {replication} | {all}      | trust
           7 | host  | {replication} | {all}      | trust
           8 | host  | {all}         | {all}      | md5
(8 строк)
(7 rows)
```

Шаг 3. Внесем изменения для аутентификации локально

astra@alse-vanilla-gui:~$ **export PATH=$PATH**
astra@alse-vanilla-gui:~$ **export PATH=/opt/tantor/db/16/bin:$PATH**
astra@alse-vanilla-gui:~$ **echo $PATH**

/opt/tantor/db/16/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games

Любым редактором внесем две строки

Файл pg_hba.conf

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     ident    map=map1
```

Файл pg_ident.conf

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME
map1   astra  user1

postgres@alse-vanilla-gui:~$ psql

psql (16.1)
Введите "help", чтобы получить справку

postgres=# SELECT pg_reload_conf();
pg_reload_conf
----------------
t
(1 row)
```

создадим двух пользователей user1 и 2

```
postgres=# CREATE ROLE user1 LOGIN;
CREATE ROLE
postgres=# CREATE ROLE user2 LOGIN;
CREATE ROLE

postgres=# \du
                                   Список ролей
    Имя роли    |                                 Атрибуты
----------------+-----------------------------------------------------------
-------------
 anon_test_user | Суперпользователь
 pma_user       | Суперпользователь, Создаёт роли
 postgres       | Суперпользователь, Создаёт роли, Создаёт БД, Репликация,
Пропускать RLS
 replicator     | Репликация
```

Посмотрим есть ли ошибки в конфигурации

```
postgres=# \d pg_ident_file_mappings;
             Представление "pg_catalog.pg_ident_file_mappings"
   Столбец   |   Тип   | Правило сортировки | Допустимость NULL | По
умолчанию
-------------+---------+--------------------+-------------------+------------
--
 map_number  | integer |                    |                   |
 file_name   | text    |                    |                   |
 line_number | integer |                    |                   |
 map_name    | text    |                    |                   |
 sys_name    | text    |                    |                   |
 pg_username | text    |                    |                   |
 error       | text    |                    |                   |


postgres=# SELECT * FROM pg_ident_file_mappings;
```

```
map_number |                        file_name                         |
line_number | map_name
| sys_name | pg_username | error
-----------+----------------------------------------------------------+----------
---+----------
+----------+-------------+-------
          1 | /var/lib/postgresql/tantor-se-16/data/pg_ident.conf |
74 | map1
| astra    | user1       |
(1 строка)
```

postgres**=# SELECT rule_number, type, database, user_name, auth_method, address, options, error FROM pg_hba_file_rules();**

```
rule_number | type  |   database    | user_name  | auth_method |  address  |
options  | er
ror
-------------+-------+---------------+------------+-------------+-----------
+----------+---
----
           1 | local | {all}         | {all}      | peer        |           |
{map=m1} |
           2 | local | {all}         | {pma_user} | md5         |           |
|
           3 | local | {all}         | {all}      | trust       |           |
|
           4 | host  | {all}         | {all}      | trust       | 127.0.0.1 |
|
           5 | host  | {all}         | {all}      | trust       | ::1       |
|
           6 | local | {replication} | {all}      | trust       |           |
|
           7 | host  | {replication} | {all}      | trust       | 127.0.0.1 |
|
           8 | host  | {replication} | {all}      | trust       | ::1       |
|
           9 | host  | {all}         | {all}      | md5         | 0.0.0.0   |
|
(9 строк)
```

Проверим корректность настройки

```
astra@alse-vanilla-gui:~$ psql -U user2 -d postgres
psql: ошибка: подключиться к серверу через сокет
"/var/run/postgresql/.s.PGSQL.5432" не удалось: ВАЖНО:  пользователь "user2"
не прошёл проверку подлинности (Peer)


astra@alse-vanilla-gui:~$ psql -U user1 -d postgres
psql (16.1)
Введите "help", чтобы получить справку.

postgres=> \q
```

5. Очистка лишних объектов

```
postgres=# DROP ROLE user1, user2;
DROP ROLE
```

Удалить строки

Файл pg_hba.conf

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     ident   map=map1
```

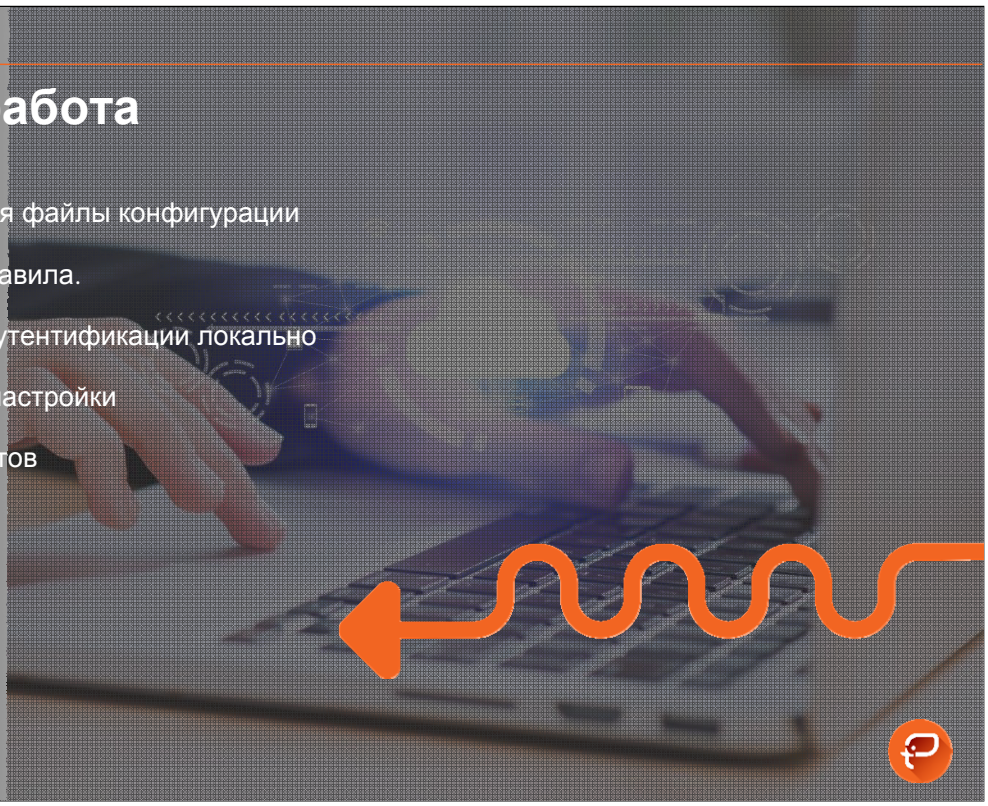
Файл pg_ident.conf

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME
map1    astra   user1
```

```
postgres=# SELECT pg_reload_conf();
pg_reload_conf
----------------
t
(1 row)

postgres=# SELECT rule_number, type, database, user_name, auth_method,
address, options, error  FROM  pg_hba_file_rules();
rule_number | type  |   database    | user_name  | auth_method |  address  |
options | error
-------------+-------+---------------+------------+-------------+-----------
+---------+-------
          1 | local | {all}         | {pma_user} | md5         |           |
|
          2 | local | {all}         | {all}      | trust       |           |
|
          3 | host  | {all}         | {all}      | trust       | 127.0.0.1 |
|
          4 | host  | {all}         | {all}      | trust       | ::1       |
|
          5 | local | {replication} | {all}      | trust       |           |
|
          6 | host  | {replication} | {all}      | trust       | 127.0.0.1 |
|
          7 | host  | {replication} | {all}      | trust       | ::1       |
|
(7 строк)

postgres=# select * from pg_ident_file_mappings;
map_number | file_name | line_number | map_name | sys_name | pg_username |
error
------------+-----------+-------------+----------+----------+-------------+--
-----
(0 строк)
```

Практическая работа

- Посмотреть где находятся файлы конфигурации
- Посмотрим какие есть правила.
- Внесем изменения для аутентификации локально
- Проверим корректность настройки
- Очистка ненужных объектов



Спасибо!

tantorlabs.ru
edu@tantorlabs.ru

07a Физическое резервирование

Виды резервных копий

- Горячие - без простоя, без остановки экземпляра, без приостановки обслуживания клиентских сессий
- Холодные - на корректно остановленном экземпляре
- Автономные или самодостаточные

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 СУБД Тантор есть много способов резервирования. Самое оптимальное с точки зрения простоты, стоимости, отказоустойчивости для типичного кластера это 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 обычно нежелательно. Однако может встречаться на практике. Кластер использует одну или несколько точек монтирования. Если файловая система поддерживает моментальные копии, то на остановленном кластере их можно сделать по точкам монтирования за короткое время. Время простоя кластера будет невелико.

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) определенное 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 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 он есть). Если файл журнала отсутствует, экземпляр не запустится. Согласованность лишь ускоряет запуск экземпляра.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/backup-file.html

Инкрементальные копии

- могут быть полезны, если размер кластера большой
- дополнительно поставляемое клиентское приложение WAL-G поддерживает "дельта-копии"
- увеличивает сложность процедур резервирования

Также резервирование может быть инкрементальным. В стандартной поставке СУБД Тантор имеется утилита резервирования `pg_basebackup`, она не поддерживает инкрементальное резервирование. Есть расширение `pg_probackup`, которое поддерживает инкрементальное резервирование и восстановление. Также есть клиентская программа WAL-G которая поддерживает создание "дельта-копий" (дельта от слова разница, дифференциальные копии то же самое что инкрементальные). Инкрементальное резервирование оправдано в случае когда обычное резервирование длится долгое время или создаёт повышенную нагрузку на ввод-вывод (чтение и запись файлов, ресурсы "железа"). Долгим временем может считаться, например, если экземпляр ночью испытывает небольшую нагрузку, то резервирование не успевает завершиться за ночь. Логика инкрементального резервирования в том, что создается один раз полная копия всех файлов данных и исходя из того, что за сутки (или желаемые интервалы между резервированием) успевает измениться небольшая часть блоков файлов данных, то достаточно зарезервировать только изменившиеся блоки. Объем будет небольшой. Чтобы резервирование шло быстро и не читались все блоки файлов данных (а это половина времени резервирования) должно использоваться расширение, которое будет на работающем экземпляре отслеживать изменения в блоках данных. Если полученный бэкап небольшого размера можно наложить на полный бэкап, это называется инкрементально-обновляемая резервная копия. Делается один раз полный бэкап на что тратится время, а потом раз в сутки (или другой периодичностью) делается инкрементальный бэкап, накладывается на полный и после этого удаляется. Периодичность 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 журнальные файлы можно удалять, тем самым освобождая место, а также ускоряя восстановление если оно потребуется. Ускорение достигается за счет того, что не придется накатывать журналы, объем которых может быть значительным.

Недостаток инкрементального резервирования - бОльшая сложность процедур. Чем больше сложность, тем больше вероятность ошибиться в том числе в процессе непредвиденного сбоя и необходимости выполнять процедуру восстановления.

Есть еще одно понятие это частичные копии, например, отдельных баз данных, возможны с помощью `pg_probackup`, но это экспериментальная возможность, сильно усложняющая резервирование и восстановление и смысла не имеет, так как можно создать несколько кластеров и частичная копия не потребуется. Архитектурно частичные копии могут быть инкрементальными или полными, то есть эти понятия не взаимозависимы.

Что резервировать

Резервируются:

- бинарные файлы с данными
- журнал предзаписи
- текстовые файлы параметров кластера

Не резервируются:

- временные файлы
- файлы которые пересоздаются при запуске экземпляра

В кластере имеются:
1)файлы данных - бинарные файлы с размером блока 8Кб. В них содержатся данные объектов базы данных, системного каталога и служебные файлы (управляющий файл, файлы `xact` (устаревшее название с`log`) и другие). Сюда же относятся и директории, в которых располагаются файлы и символические ссылки.
2)WAL-журнал или "журнал предзаписи" или просто "журнал" (устаревшее название `xlog`). Слово "журнал" может использоваться и для обозначения текстовых файлов сообщений экземпляра. В в этой главе будем использовать его для обозначения журнала предзаписи. Он состоит из файлов по умолчанию имеющих размер 16Мб. В журнал записываются изменения в блоках файлов данных и с частотой контрольных точек могут записываться полные образы изменившихся с момента предыдущей контрольной точки блоков данных.
3)текстовые файлы параметров и другие файлы и директории, находящиеся в PGDATA не распознаваемые утилитами резервирования как временные. Резервируются для удобства вместе с файлами данных.
4)временные файлы. Их можно не резервировать, так как их срок жизни не дольше сессии, после чего они могут быть удалены. Если утилита резервирования может распознать такие файлы, она их в бэкап не включает.

Логика резервирования в следующем. Делается копия файлов данных и других файлов. Это можно делать утилитами операционной системы, но это неудобно так как надо предусмотреть выполнение контрольной точки ли получения о ней данных. Обычно используют утилиту `pg_basebackup` или утилиты расширений, которые автоматизируют как резервирование, так и восстановление. Дальше копятся или передаются в безопасное место журналы, создаваемые с момента завершения контрольной точки перед резервированием и до самого последнего момента, пока работает экземпляр (если хочется восстановиться до последнего момента, а это обычно необходимое требование для сколько-нибудь важных данных). Таким образом создаются "бэкапы файлов кластера" и "архив журналов".

https://docs.tantorlabs.ru/tdb/ru/15_4/se/continuous-archiving.html

Процедура восстановления

- Остановить экземпляр
- Восстановить из резервной копии её содержимое
- Накатить журнальные файлы
- Запустить экземпляр

Восстановление:
1)обнаружение повреждения. Экземпляр при этом может быть аварийно остановлен и попытка запуска не удастся. Либо продолжать работать, но выдавать ошибки при доступе к части данных. Если экземпляр не остановился, его нужно будет остановить в аварийном режиме, так как вряд ли он остановится корректно.
2)Восстановить из бэкапа всё что в нём есть. Если размер кластера большой, можно пытаться восстанавливать только те файлы, которые отличаются. Гарантию даст только подсчет контрольных сумм каждого файла и их сравнение. Если утилита восстановления имеет такие режимы можно их использовать. Ручное использование например `rsync` (в режиме подсчета контрольных сумм) может увеличить время восстановления, та как к нему добавляется время на планирование и ввод команд восстановления ("think time"). Оптимально иметь командные файлы и простые инструкции на случай сбоев, это уменьшит время восстановления за счет устранения времени на раздумья и вероятные ошибки.
3)Накатить (наложить, применить) журнальные файлы. Файлы применяются в строгом порядке их создания экземпляром, пропуски (gap) - отсутствие файла или повреждение блоков (они распознаются, так как блоки журналов защищены контрольными суммами) в его теле критичны - перейти через них нельзя. Поэтому журналы можно называть "цепочкой" журнальных файлов (WAL сегментов), так как повреждения звена цепочки ее разрывает. Есть техники восстановления с пропусками, но не стоит на них полагаться. Если пропусков нет, но запись не может быть применена к файлу данных (файл отсутствует), то накат также останавливается. Записи журналов применяются к блокам данных в строгом порядке. При применении проверяется, что к блоку данных может быть применена запись. В журналах по умолчанию присутствуют полные образы изменившихся блоков (`full_page_writes`) и даже если блоки данных повреждены (torn "разорваны", расщеплены), они смогут восстановиться.

Файлы журнала предзаписи

- находятся в PGDATA/pg_wal
- файлы журнала не дублируются
- размер текущего файла журнала такой же как остальных
- в файлы журнала пишут серверные и фоновые процессы экземпляра

Наличие файлов журнала с момента начала контрольной точки которая инициируется в начале резервирования является критичным для восстановления. Каким образом организовано хранение файлов журнала?

В директорию PGDATA/pg_wal процессы экземпляра кластера производят запись в файлы журнала. Файлы экземпляром кластера не дублируются. Можно сказать что каждый WAL-файл в этой директории представлен в одном экземпляре (не экземпляре кластера, а экземпляре файла). В каждый момент времени есть текущий файл журнала, куда процессы экземпляра пишут (или последний файл куда писали, если экземпляр погашен). Размер этого файла всегда равен размеру других файлов (по умолчанию 16Мб) потому, что при создании следующего файла ему сразу задаётся размер. Размер задаётся либо (параметр wal_init_zero) командой записи последнего байта в файл размером wal_segment_size, либо командами записи пустых блоков вплоть до размера wal_segment_size. Это нужно, чтобы заранее было зарезервировано место в файловой системе и экземпляр не столкнулся с нехваткой места, а также для отказоустойчивости и быстроты: 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 это метаданные) в случае пропадания питания последние блоки файла либо будут потеряны, либо скорость записи в файл будет невысокой.

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`)

В файлы журнала процессы экземпляра пишут записями 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). Можно сказать, что LSN определяет "смещение от начала журнала" или "позицию в журнале предзаписи". 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.

Самый первый файл имеет название 000000010000000000000001. Название состоит из трёх частей по 8 знаков. Каждое число 32-битное, записывается в шестнадцатеричной форме. Максимальное число FFFFFFFF (32 единички в двоичной записи). Первое число это номер "линии времени" (`Time Line`, `TLI`, ветвь времени, инкарнация). Это число увеличивается на единицу при открытии экземпляром кластера после процедуры восстановления для того, чтобы не допустить затирания файла журнала, так как восстановление не всегда попадает на 16-мегабайтную границу журнала, LSN на который восстановили кластер может указывать на байт в середине файла. В этом случае до восстановления в файле первая половина нужные записи, вторая - байты со значением ноль.

При достижении максимальных значений сброса в ноль (LSN wrap) LSN и линий времени не предусмотрено. Максимальное значение LSN довольно большое: 16777216 террабайт.

Физически запись в журнальные файлы производится 8-килобайтными блоками. На размер блока указывает параметр конфигурации wal_block_size, который задан при сборке СУБД Тантор и не меняется. Журнальные записи защищены контрольными суммами.

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/15_4/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 16мегабайтных файлов журнала

Рассмотрим подробнее LSN. Возможно вы задавались вопросом почему размер файлов журнала по умолчанию 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, 62, 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.

Процесс восстановления

- startup - процесс восстановления файлов кластера с помощью журнала
- при изменении линии времени в директории PGDATA/pg_wal создаются текстовые файлы 0000000**N**.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 и чтобы была возможность вернуться на прежние линии времени

В процессе полного восстановления, информация о том, какая журнальная запись была сформирована самой последней перед падением экземпляра никуда не записывается. Процесс восстановления последовательно читает журнальные записи и если видит, что следующая запись в журнале содержит "мусор", то прекращает восстановление. 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 восстановление. Пример сообщения в файле сообщений экземпляра:

```
LOG:  invalid record length at CA/277E2A88: expected at least 26, got 0
```

процесс восстановления ожидал увидеть число не меньше 26 (минимальный размер журнальной записи), а увидел нули.

Процесс восстановления может сформировать название файла, так как знает номер линии времени, размер блока журнала, размер файла, LSN из управляющего файла. При изменении линии времени в директории PGDATA/pg_wal создаются текстовые файлы 0000000**N**.history с информацией о линиях времени. В журнале, формировавшемся на прежней линии времени данных о новой линии нет, так как экземпляр прежней линии времени, который формировал файл журнала, остановился.

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
- `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

Посмотрим описание полезных функций для работы с журналами. В документации есть похожие описания, но сформулированные иначе (может даже показаться, что менее понятно).

`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.

`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/15_4/se/functions-admin.html#FUNCTIONS-ADMIN-BACKUP

Холодное резервирование

- экземпляр должен быть корректно остановлен
- утилита `pg_basebackup` не может резервировать кластер, если экземпляр погашен
- используются утилиты операционной системы `cp, rsync, tar`
- Копируется целиком PGDATA и директории табличных пространств
- создается автономная копия, на которой можно запустить экземпляр
- преимущество - простота
- основной недостаток - прерывание обслуживания клиентов кластера

Холодное резервирование это резервирование на корректно остановленном кластере. В результате получается автономная копия кластера. Автономная или самодостаточная (self contained) это включающая в себя все файлы, чтобы мог запуститься экземпляр и дать доступ к данным.

Техника резервирования: определяется местоположение PGDATA, директории табличных пространств (символические ссылки в `PGDATA/pg_tblspc`), экземпляр корректно останавливается, проверяется что процессы экземпляра не остались в памяти и найденные директории копируются любыми утилитами.

Особенности подробно описаны в документации. Например, можно использовать снимки файловой системы, если она позволяет или предварительно выполнять копирование, а после остановки кластера обновлять файлы утилитой `rsync` в режиме подсчета контрольных сумм. Основное преимущество холодного резервирования - простота при этом теряется. Более практично резервирование работающего кластера, например, утилитой `pg_basebackup`.

Созданную копию можно использовать с накапливаемыми журнальными файлами (архивом журналов), например, для полного восстановления.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/backup-file.html

full_page_writes

- по умолчанию включён
- защищает от разрыва (torn) страниц
- не рекомендуется отключать
- включается на время резервирования утилитой `pg_basebackup`
- при резервировании реплики должен быть включён на мастере

По умолчанию параметр `full_page_writes` включён. Это означает, что в журнал записывается всё содержимое (8Кб) каждого блока данных ("страницы") при первом изменении этого блока (меняется в буфере кэша буферов) после каждой контрольной точки. Это большой объем данных. Зачем этот объем данных нужен? Размер блока 8Кб, а размер блока HDD, SDD, файловой системы не всегда 8Кб, а чаще 4Кб. В случае пропадания питания в блок данных, а они не дублируются, может быть записаны 4Кб, а другие 4Кб не запишутся и останутся от прежней версии блока. Такой блок называется расщепленный или "разорванный" (torn). Контрольная сумма в блоке не совпадёт и он будет считаться повреждённым. Блок может относиться к объекту системного каталога и экземпляр может не запуститься.

Почему при первом изменении страницы она записывается в журнал, а не при втором? 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 на свои места 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 образ страницы. При таких ошибках можно использовать ignore_invalid_pages=on (только если full_page_writes был включён) надеясь на то, что целый образ страницы 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, а это кэш на запись или от порядка сброса блоков. Операционная система может работать с 4Кб блоками в своем кэше и записывать их в произвольном порядке. Если хоть где-то на пути от процесса экземпляра до сектора диска (или контроллера плашки SSD) используется блок не 8Кб, а меньше, то вероятность получить большое число разорванных блоков при пропадании питания или сбое операционной системы высока и отключать параметр не стоит.

При резервировании реплики full_page_writes должен быть включён на мастере.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/runtime-config-wal.html#GUC-FULL-PAGE-WRITES

Утилита pg_basebackup

- создаёт резервную копию кластера на работающем экземпляре
- создать копию отдельных баз данных или объектов базы нельзя, только всего кластера
- использует протокол репликации

Создаёт резервную копию всего кластера на работающем экземпляре. Не останавливает обслуживание, не блокирует работу пользовательских сессий.

Создаёт копию всего кластера. Не может создавать копию отдельных баз данных или объектов.

Утилита подсоединяется к экземпляру через сетевое соединение или Unix-сокет. По умолчанию создаёт два соединения по протоколу репликации, в котором нет обычных команд SQL, но есть команды для получения файлов из файловой системы сервера. Через первое соединение создаётся бэкап, через второе параллельно начинают передаваться журнальные файлы.

Для подсоединения к экземпляру по протоколу репликации можно дать отдельные разрешения ролям кластера. Утилита входит в стандартную поставку.

Утилита по умолчанию создаёт резервную копию на том хосте, на котором запущена. Запускаться может на хосте, отличном от хоста где работает экземпляр. Кластер будет резервироваться через сетевое соединение.

На время резервирования включает `full_page_writes`, если параметр был отключен.

По умолчанию создаёт временный слот репликации. При резервировании нужно использовать временные или постоянные слоты репликации, чтобы пока создаётся бэкап кластер не удалил необходимые для восстановления этого бэкапа журнальные файлы.

После того как резервирование закончено утилита переключает журнал или дожидается его переключения, принимает журнал на котором завершилось резервирование. Только после получения файла журнала на котором завершилось резервирование получается автономный бэкап.

Может выполнять резервирование подсоединившись к экземпляру, обсуживающую физическую реплику (копию) кластера, не нагружая экземпляр основного (primary, мастера) кластера. Это называют "backup offloading".

Инкрементальное резервирование отсутствует, создаётся копия всего кластера.

Для наблюдения за процессом резервирования есть представление pg_stat_progress_basebackup.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/app-pgbasebackup.html

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`

Зачем использовать эту утилиту? Если хочется проверить, что файлы в бэкапе не повредились во время хранения, а также что во время резервирования были получены нужные для синхронизации журнальные файлы. Вероятность их неполучения мала, поэтому проверять бэкапы после их создания нет необходимости. Гарантий утилита не даёт, их даёт 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.

Стоит ли отключать создание файла манифеста при резервировании? Не стоит. Если не менять алгоритмы, то лишней нагрузки нет.

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 их наличие.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/app-pgverifybackup.html

Архив журналов

Способы организации архива журналов:

- копирование WAL-сегментов командой в параметре `archive_command`
- прием потока журнальных записей и сохранение утилитой `pg_receivewal`
- сторонние расширения (barman)
- использование физических реплик

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)установив параметр `archive_command='команда'` и `archive_mode=on`. У этого метода есть недостаток - текущий файл журнала (в который пишут процессы экземпляра) начнет копироваться только когда он перестанет быть текущим. Если текущий файл потеряется, то данные в нем будет неоткуда взять и будут потеряны транзакции. Это обычно неприемлемо.

2)утилитой `pg_receivewal`. Эта утилита может принимать журнальные данные без задержки. Недостатком является то, что нужно автоматизировать запуск утилиты и перезапускать в случае сбоя.

3)использовать сторонние расширения, автоматизирующие резервирование

4)использовать физические реплики

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 единственной точкой сбоя (single point of failure)
- Использование `pg_receivewal` или физической реплики - возможность гарантировать отсутствие потерь при повреждении текущего журнала

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.

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-обёртки для автозапуска (systemd) отсутствуют
- может сжимать журнальные записи
- преимущество в том, что является инициатором соединения, что позволяет обеспечить безопасность
- может подтверждать транзакции в режиме синхронной фиксации
- текущему журнальному файлу даёт суффикс `.partial`

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, чтобы не было путаницы утилита называет как имя`.partial`

`pg_receivewal` по умолчанию накапливает журнальные данные в памяти, а сохраняет в файл при закрытии файла. Если нужно чтобы утилита записывала принятые данные без задержки, утилиту нужно запустить с параметром `--synchronous`. Этот же режим должен использоваться если утилита будет подтверждать транзакции в режиме синхронной фиксации, установленной параметром synchronous_commit. Этот параметр определяет после завершения какого уровня обработки журнальной записи процесс будет выдавать клиенту сообщение об успешной фиксации COMMIT COMPLETE.

Значения:

`remote_apply` - к `pg_receivewal` неприменимо, только физические реплики могут подтвердить транзакцию. Не стоит ставить, так как скорость фиксации транзакций резко падает.

`on` - значение по умолчанию. Транзакция подтверждается после того как pg_receivewal или реплика получит ответ от своей операционной системы, что она записала журанльные 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 их в своём кэше файловой системы и если пропадет питание, то блоки могут быть потеряны. Это значение разумный выбор, если вероятность сбоя основного хоста и следом за ним резервного мала, а значение on приводит к деградации производительности, которую нельзя устранить другими способами (например, параметром `commit_siblings` или заменой `fsync` на стороне `pg_recievewal`)

`local` - транзакция подтверждается после записи в локальный файл журнала и `fsync` (метод используемый по умолчанию)

`off` - не стоит ставить на уровне кластера. Может ставиться разработчиками приложений на уровне сессий или транзакций.

Если `synchronous_standby_names` не задано, то это эквивалентно local и текущий журнал - единственная точка сбоя.

Утилита может сжимать сохраняемые журналы.

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.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/app-pgreceivewal.html

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

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, они безусловно 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').

https://docs.tantorlabs.ru/tdb/ru/15_4/se/functions-admin.html#FUNCTIONS-REPLICATION

Создание базовой резервной копии

- pg_basebackup лучше запустить на хосте где нужно создать резервную копию
- параметр -D задает путь к директории
- по умолчанию создаёт два соединения к двум процессам wal-sender. По первому соединению передаются файлы с данными, по второму журналы
- Можно ограничить скорость резервирования первого соединения
- инициирует контрольную точку перед началом копирования
- создаёт файлы `backup_manifest` и `backup_label`

Утилита `pg_basebackup` может создавать бэкапы в форматах plain и tar. Второй формат не рассматриваем, всё написанное ниже относится к формату plain.

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. По умолчанию 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 данных, чтобы снизить нагрузку на ввод-вывод. Диапазон от 32 КБ/с до 1024 МБ/с. На скорость передачи журнальных данных влияет, только если выбран метод передачи журналов `fetch`, который не имеет смысла использовать.

В начале резервирования мастера (основной кластер, не физическая реплика) 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`

Параметром -t или --target можно (но не нужно) резервировать в директорию на хосте кластера, а также резервировать "в никуда" (`--target=blackhole`). Последний режим может использоваться для измерения производительности: какая часть времени резервирования тратится на чтение файлов.

Утилита резервирует директории и файлы, которые ей неизвестны. Поэтому не стоит хранить в PGDATA файлы, которые вы бы не хотели видеть в бэкапе, например файлы сообщений большого размера.

Утилита создаёт файлы `backup_manifest, backup_label`. Файл `backup_label` содержит те же данные, что и в файле `pg_control` бэкапа

Утилита **не** резервирует файлы, которые известно что не нужно резервировать. Такие файлы описаны в документации:

https://docs.tantorlabs.ru/tdb/ru/15_4/se/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP-DATA

WAL-G

- дополнительно поставляемое с СУБД Тантор клиентское приложение
- Используется для резервирования кластера и файлов журналов по протоколу S3
- поддерживает создание "дельта-копий" (инкрементальные копии)
- поддерживает ограничение скорости резервирования, проверку целостности файлов, настройку уровня параллелизма для загрузки и выгрузки файлов

WAL-G дополнительно поставляемое с СУБД Тантор клиентское приложение.

Используется для резервирования кластера и файлов журналов по протоколу S3, который используют и облачные хранилища файлов. Для резервирования во внутренней сети доступно программное обеспечение, например, minio.

Одно из преимуществ утилиты - возможность инкрементального (второе название - дифференциального) резервирования. В WAL-G это называется "дельта-копии". Они хранят страницы файлов, изменившиеся с предыдущего бэкапа.

Поддерживается троттлинг - ограничение скорости чтения файлов и скорости загрузки в место хранения, проверка целостности файлов, настройка степени параллелизма для загрузки и скачивания из хранилища файлов.

Не поддерживается поточная передача журнальных записей, журналы передаются по файлам (WAL-сегментам).

Если хранилище поддерживает протокол S3 и не поддерживает монтирование по сети (обычно, NFS), то `WAL-G` может быть полезен.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/wal-g.html

Демонстрация

- Изменение размера WAL файлов

07a. Демонстрация

Изменение размера WAL файлов

1) Корректно остановим экземпляр кластера:
```
postgres@tantor:~$ pg_ctl stop
ожидание завершения работы сервера.... готово
сервер остановлен
```

2) Проверим, что остановка выполнена корректно:
```
postgres@tantor:~$ pg_controldata | grep state
postgres@tantor:~$ pg_controldata | grep Состояние
Состояние кластера БД:                выключен
```

Переключение раскладки: справа внизу окна виртуальной машины кликнуть мышкой на Eng.

3) Сохраним значения из управляющего файла для последующего сравнения со значениями, которые изменятся:
```
postgres@tantor:~$ pg_controldata > 16MB.txt
```

3) Меняем размер WAL-сегментов с 16Мб на 25Мб:
```
postgres@tantor:~$
pg_resetwal --wal-segsize=256 /var/lib/postgresql/tantor-se-16/data
Журнал предзаписи сброшен (Write-ahead log reset)
```

4) сохраним значения из управляющего файла для сравнения:
```
postgres@tantor:~$ pg_controldata > 256MB.txt
```

5) сравним:

```
postgres@tantor:~$ diff 16MB.txt 256MB.txt
5,8c5,8
< Последнее обновление pg_control:      02:29:38 PM MSK
< Положение последней конт. точки:      9/1199D828
< Положение REDO последней конт. точки: 9/1199D828
< Файл WAL с REDO последней к. т.:      000000010000000900000011
---
> Последнее обновление pg_control:      02:34:53 PM MSK
> Положение последней конт. точки:      9/30000028
> Положение REDO последней конт. точки: 9/30000028
> Файл WAL с REDO последней к. т.:      000000010000000900000003
23c23
< Время последней контрольной точки:    02:29:38 PM MSK
---
> Время последней контрольной точки:    02:34:53 PM MSK
30c30
< Значение wal_level:                   replica
---
> Значение wal_level:                   minimal
42c42
< Байт в сегменте WAL:                  16777216
---
> Байт в сегменте WAL:                  268435456
```

Пример на английском языке:

```
< pg_control last modified:             12:43:57 AM MSK
< Latest checkpoint location:           115/BE000F70
< Latest checkpoint's REDO location:    115/BE000F70
< Latest checkpoint's REDO WAL file:    0000000100000115000000BE
---
> pg_control last modified:             12:48:17 AM MSK
```

```
> Latest checkpoint location:           115/D0000028
> Latest checkpoint's REDO location:    115/D0000028
> Latest checkpoint's REDO WAL file:    00000001000001150000000D
23c23
< Time of latest checkpoint:            12:43:57 AM MSK
---
> Time of latest checkpoint:            12:48:17 AM MSK
30c30
< wal_level setting:                    replica
---
> wal_level setting:                    minimal
42c42
< Bytes per WAL segment:                16777216
---
> Bytes per WAL segment:                268435456
```

значение **minimal** поменяет своё значение после запуска экземпляра.

5) Попытаемся запустить экземпляр:

```
postgres@tantor:~$ pg_ctl start
ожидание запуска сервера....
[10094] ВАЖНО:  "min_wal_size" должен быть минимум вдвое больше
"wal_segment_size"
[10094] СООБЩЕНИЕ:  система БД выключена
 прекращение ожидания
pg_ctl: не удалось запустить сервер
Изучите протокол выполнения.

waiting for server to start....
[10094] FATAL:  "min_wal_size" must be at least twice "wal_segment_size"
[10094] LOG:  database system is shut down
 stopped waiting
pg_ctl: could not start server
Examine the log output
```

Мы не учли, что от размера WAL сегментов может что-то зависеть.

6) Установим значение параметра:

```
postgres@tantor:~$ echo "min_wal_size=512MB" >> $PGDATA/postgresql.auto.conf
```

7) Запустим экземпляр:

```
postgres@tantor:~$ pg_ctl start

ожидание запуска сервера....
[10962] СООБЩЕНИЕ:  передача вывода в протокол процессу сбора протоколов
[10962] ПОДСКАЗКА:  В дальнейшем протоколы будут выводиться в каталог "log".
 готово
сервер запущен
```

8) В `psql` переключим файл журнала:

```
postgres@tantor:~$ psql
postgres=# select pg_switch_wal();
 pg_switch_wal
---------------
 115/D000015A
(1 row)

postgres=# select pg_switch_wal();
 pg_switch_wal
```

```
---------------
 115/E000008A
(1 row)

[8505] LOG:  checkpoint starting: wal
```

Теперь после слэша меняется не два символа, а один. Остальные символы укажут на смещение в 256-мегабайтном файле

```
LOG:  checkpoint complete: wrote 4 buffers (0.0%); 0 WAL file(s) added, 0
removed, 0 recycled; write=0.002 s, sync=0.001 s, total=0.270 s; sync
files=3, longest=0.001 s, average=0.001 s; distance=524288 kB,
estimate=524288 kB; lsn=115/F00000B8, redo lsn=115/F0000070
```

9) Выйдем из `psql`, остановим кластер и вернем обратно размер журнала:

```
postgres=# \q
postgres@tantor:~$ pg_ctl stop
[8504] LOG:  received fast shutdown request
ожидание завершения работы сервера....
[8504] LOG:  aborting any active transactions
[8504] LOG:  background worker "logical replication launcher" (PID 8510)
exited with exit code 1
[8505] LOG:  shutting down
[8505] LOG:  checkpoint starting: shutdown immediate
[8505] LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s)
added, 0 removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.008 s;
sync files=0, longest=0.000 s, average=0.000 s; distance=0 kB,
estimate=471859 kB; lsn=115/F0000198, redo lsn=115/F0000198
[8504] LOG:  database system is shut down
готово
сервер остановлен
```

10) Проверяем корректность остановки:

```
postgres@tantor:~$ pg_controldata | grep state
postgres@tantor:~$ pg_controldata | grep Состояние
Состояние кластера БД:                выключен
```

11) Меняем размер обратно на 16Мб:

```
postgres@tantor:~$ pg_resetwal --wal-segsize=16 /var/lib/postgresql/tantor-
se-16/data
Журнал предзаписи сброшен
```

12) Запустим экземпляр через службы:

```
postgres@tantor:~$ sudo systemctl start tantor-se-server-16
```

13) Проверяем как изменилось содержимое выдаваемых LSN:

```
postgres@tantor:~$ psql
psql (16.1)
Type "help" for help.

postgres=# select pg_switch_wal();
 pg_switch_wal
---------------
 116/15A
(1 row)

postgres=# select pg_switch_wal();
 pg_switch_wal
```

```
---------------
 116/100008A
(1 row)
```

14) LSN может выводиться коротким, как в данном примере. Почему LSN был с виду "коротким" **116/15A**? И в **116/100008A** после слэша 7 символов, а не 8.
Потому, что название WAL-сегмента приняло значение ноль в конце.
Реальное значение: **116/0000015A** и **116/0100008A**

```
postgres@tantor:~$ ls $PGDATA/pg_wal
000000010000011600000000  000000010000011600000001  000000010000011600000002
000000010000011600000003  archive_status
```

15) Посмотрим какие записи есть в файлах журнала (выберите несколько):

```
postgres@tantor:~$ pg_waldump 000000010000011600000000

rmgr: XLOG        len (rec/tot):    148/   148, tx:                  0, lsn: 116/00000028, prev
0/00000000, desc: CHECKPOINT_SHUTDOWN redo 116/28; tli 1; prev tli 1; fpw true; xid 35741; oid
390998; multi 502936; offset 2034077; oldest xid 723 in DB 1; oldest multi 1 in DB 1;
oldest/newest commit timestamp xid: 0/0; oldest running xid 0; shutdown
rmgr: XLOG        len (rec/tot):     56/    56, tx:                  0, lsn: 116/000000C0, prev
116/00000028, desc: PARAMETER_CHANGE max_connections=100 max_worker_processes=8
max_wal_senders=10 max_prepared_xacts=0 max_locks_per_xact=64 wal_level=replica wal_log_hints=off
track_commit_timestamp=off
rmgr: Standby     len (rec/tot):     68/    68, tx:                  0, lsn: 116/000000F8, prev
116/000000C0, desc: RUNNING_XACTS nextXid 35741 latestCompletedXid 35740 oldestRunningXid 35741
rmgr: XLOG        len (rec/tot):     26/    26, tx:                  0, lsn: 116/00000140, prev
116/000000F8, desc: SWITCH
```

```
postgres@tantor:~$ pg_waldump 000000010000011600000001

rmgr: Standby     len (rec/tot):     68/    68, tx:                  0, lsn: 116/01000028, prev
116/00000140, desc: RUNNING_XACTS nextXid 35741 latestCompletedXid 35740 oldestRunningXid 35741
rmgr: XLOG        len (rec/tot):     26/    26, tx:                  0, lsn: 116/01000070, prev
116/01000028, desc: SWITCH
```

```
postgres@tantor:~$ pg_waldump 000000010000011600000002

rmgr: Standby     len (rec/tot):     68/    68, tx:                  0, lsn: 116/02000028, prev
116/01000070, desc: RUNNING_XACTS nextXid 35741 latestCompletedXid 35740 oldestRunningXid 35741
rmgr: XLOG        len (rec/tot):     26/    26, tx:                  0, lsn: 116/02000070, prev
116/02000028, desc: SWITCH
```

Текущий файл журнала (03):

```
postgres@tantor:~$ pg_waldump  000000010000011600000003

rmgr: Standby     len (rec/tot):     68/    68, tx:                  0, lsn: 116/03000028, prev
116/02000070, desc: RUNNING_XACTS nextXid 35741 latestCompletedXid 35740 oldestRunningXid 35741
rmgr: Standby     len (rec/tot):     68/    68, tx:                  0, lsn: 116/03000070, prev
116/03000028, desc: RUNNING_XACTS nextXid 35741 latestCompletedXid 35740 oldestRunningXid 35741
rmgr: XLOG        len (rec/tot):    148/   148, tx:                  0, lsn: 116/030000B8, prev
116/03000070, desc: CHECKPOINT_ONLINE redo 116/3000070; tli 1; prev tli 1; fpw true; xid 35741;
oid 390998; multi 502936; offset 2034077; oldest xid 723 in DB 1; oldest multi 1 in DB 1;
oldest/newest commit timestamp xid: 0/0; oldest running xid 35741; online
rmgr: Standby     len (rec/tot):     68/    68, tx:                  0, lsn: 116/03000150, prev
116/030000B8, desc: RUNNING_XACTS nextXid 35741 latestCompletedXid 35740 oldestRunningXid 35741
pg_waldump: error: error in WAL record at 116/3000150: invalid record length at
116/3000198: expected at least 26, got 0
```

или на русском языке:

```
pg_waldump: ошибка: ошибка в записи WAL в позиции 116/3000150: неверная длина
записи в позиции 9/A3000198: ожидалось минимум 26, получено 0
```

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

07b Логическое резервирование

Логическое резервирование

- формирование текстового файла или файлов позволяющие воссоздать объекты
- Сохраняет объекты в том состоянии, в котором они были на начало выгрузки
- Используются утилиты:

`pg_dump`
`pg_restore`
`pg_dumpall`
`psql`
команда `COPY`
команда `psql \copy`

Резервирование на логическом уровне в постгрес - это формирование текстового файла или файлов, позволяющих воссоздать объекты их данные, не отличающихся с точки зрения логики приложения от образа резервируемых объектов. Данные и объекты после восстановления находятся в том состоянии, в котором они были на момент начала выгрузки. В файле содержатся команды SQL или текст, на основе которого можно сгенерировать команды SQL.

Инструменты логического резервирования:

1) команда `COPY TO`
2) команда `psql \copy to`
3) утилита командной строки `pg_dump`
4) утилита командной строки `pg_dumpall`

Инструменты восстановления с логической копии:

1) команда `COPY FROM`
2) команда `psql \copy from`
3) утилита командной строки `pg_restore`
4) `psql`

Функционал логического резервирования и восстановления определяется параметрами этих инструментов и он достаточно детализирован под любые потребности.

Примеры использования

- Переход на новую основную версию
- Смена параметров кластера, которые не меняются после его создания
- Проверка целостности данных
- Получение скрпита воссоздания объектов
- Резервирование объектов
- Выгрузка части кластера: отдельной базы данных, содержимого схемы и других объектов

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. Зачем это может потребоваться?

1) Для перехода на новую основную версию СУБД Тантор. Если время выгрузки и загрузки приемлемо, это оптимальный способ
2) Смена параметров кластера или базы данных, которые не меняются без пересоздания кластера или базы данных
3) Для гарантированной проверки того, что данные не повреждены. Только выгрузка на логическом уровне может это гарантировать
4) Для простой выгрузки содержимого отдельной базы данных. Физическое резервирование с помощью pg_basebackup не позволяет выгружать отдельно базы данных. В pg_probackup эта опция экспериментальная
5) Перенести данные в другие системы хранения, например, СУБД других производителей или загрузить данные из сторонних источников
6) Получить текстовый командный файл (скрипт) для установки приложения.
7) Быстро и просто зарезервировать объекты и данные на любых уровнях (кластер, базы, объекты баз, глобальные объекты), получив целостную копию (на один момент времени)

Сравнение логического и физического резервирования

возможность	Л	Ф
выгрузить 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	+	+

Логическое и физическое резервирование имеют разные цели использования. 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. Поэтому логическое резервирование не должно рассматриваться как единственный способ резервирования.

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 постгрес 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 команду SQL, возвращающей данные
- Данные выгружаются в:
  а) файл в файловой системе сервера
  б) передаются на стандартный вход (`stdin`) утилите командной строки сервера
  в) стандартный вывод `stdout`. Если команда COPY вызывается через сетевое соединение, то стандартный вывод передается через сетевое соединение клиентской программе

Особенности выгрузки:
1)Можно выгрузить указав имя таблицы (но не представления) или любую команду SQL, возвращающая данные: WITH, SELECT (любой сложности), команда VALUES, команды с выражением RETURNING (INSERT, UPDATE, DELETE). Команду нужно заключать в круглые скобки, имя таблицы не нужно. Обычно используется имя таблицы если нужно выгрузить все строки, либо SELECT с фразой WHERE, если нужно выгрузить часть строк. Команда VALUES мало распространена, но это стандартная команда SQL.
2)Вместо имени таблицы указывать имя представления нельзя, но имя представления можно использовать в команде SQL
3)Есть десять параметров, 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:

```
 COPY таблица [( столбцы )]
 | (SELECT|VALUES|.RETURNING)
TO 'файл'|PROGRAM 'команда'
 | STDOUT
WITH (
FORMAT text | csv | binary
DELIMITER 'символ'
NULL 'маркер'
HEADER true | false
QUOTE 'символ'
ESCAPE 'символ'
FORCE_QUOTE (столбцы)|*
FORCE_NOT_NULL (столбцы)
FORCE_NULL (столбцы)
ENCODING 'имя_кодировки);
```

```
=# COPY pg_authid
-#  TO PROGRAM 'gzip > file.gz';
COPY 19
=# COPY (WITH RECURSIVE
(#  t(n) AS ( SELECT 1
(#  UNION ALL
(#  SELECT n+1 FROM t
(# )
(# SELECT n FROM t LIMIT 1
(# )
-# TO stdout;
1
=# COPY (SELECT
(#  usename FROM pg_user)
-#  TO stdout;
replicator
```

Синим 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 может обрабатываться быстрее, чем текстовый и CSV форматы, но он менее переносим и выгрузить-загрузить можно только в тот же тип данных, а не в пределах семейства типов. Команда COPY не входит в стандарт SQL и специфична для PostgreSQL.
https://docs.tantorlabs.ru/tdb/ru/15_4/se/sql-copy.html

Команда COPY .. FROM

- Загрузка выполняется в одну таблицу
- Данные загружаются из:

1) файла на хосте где запущен экземпляр
2) стандартного вывода произвольной исполняемой программы
3) из стандартного ввода `stdin`

- В процессе загрузки можно сразу установить признак заморозки строк
- Можно задать опциональное выражение `WHERE`. В выражении нельзя использовать подзапросы
- `pg_stat_progress_copy` - отслеживание работы команды и при загрузке и при выгрузке

Есть девять параметров, которыми можно настроить формат и особенности выгрузки:

```
COPY таблица [( столбцы )]
FROM 'файл'|PROGRAM 'команда'
 | STDIN
WITH (
FORMAT text | csv | binary
FREEZE true | false
DELIMITER 'символ'
NULL 'маркер'
DEFAULT 'выражение'
HEADER true | MATCH
QUOTE 'символ'
ESCAPE 'символ'
ENCODING 'имя_кодировки)
[WHERE выражение];
```

Синим цветом помечены опции, которые можно указывать только при загрузке в таблицу, а не при выгрузке.

• FREEZE в процессе загрузки ставит на строки признак заморозки. В этом случае нет необходимости в будущем 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 в первой строке загружаемых данных и в таблице совпадают. Может использоваться как дополнительная проверка того, что столбцы не перепутаны при выгрузке-загрузке.

• Можно задать опциональное выражение WHERE. В этом выражнии 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 (и ТО и FROM) работает, можно отслеживать её работу через представление `pg_stat_progress_copy`.

В psql имеется команда **\copy**. Синтаксис \copy похож на COPY, но действия выполняет утилита psql. Отличие от COPY в том, что с файлом работает psql на том хосте где запущен psql, а не серверный процесс. Поскольку stdin и stdout при подсоединении по сети направляются на клиента, команда COPY может работать с файлами на клиенте с помощью перенаправления ввода-вывода.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/sql-copy.html

Команда \copy

- \copy это команда psql
- синтаксис похож на COPY
- набирается на одной строке
- 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

\copy это команда psql. Синтаксис \copy похож на синтаксис команды COPY, но действия выполняет утилита psql. Отличия от COPY:

1)\copy набирается на одной строке, COPY можно набирать на нескольких строках

2)\copy .. from в формате CSV ошибочно обрабатывает единственное в строке значение \. как конец ввода и следующие строки не загружает

3)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

4)Количество обработанных строк \copy .. to stdout не отображает

5)При выполнении \copy ... to stdout вывод направляется в то же место, что и вывод psql команд. Для чтения/записи стандартного ввода/вывода psql, вне зависимости от источника текущей команды или параметра \o, можно использовать from pstdin или to pstdout

6)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/15_4/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, 0`
- можно не создавать файл, а использовать пайп (pipe):
  `pg_dump параметры | psql параметры`
  `pg_dump -F c параметры | pg_restore параметры`

`pg_dump` - утилита создания логической резервной копии содержимого базы данных.Утилита подсоединяется к одной базе данных так же как `psql` , использует обычные команды SQL, устанавливает блокировки самого низкого уровня ACCESS SHARE, такие же как команда SELECT. Эта блокировка нужна, 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. Утилита может выгружать данные параллельно несколькими процессами, согласованность на один момент времени при этом сохраняется. Для этого используется стандартный функционал - экспорт моментального снимка. Для выгрузки по умолчанию использует высокоэффективную команду COPY, но может формировать и набор команд INSERT. Может выгружать определения объектов без данных и наоборот. Имеет гибкие настройки, позволяющие детально выбирать какие типы объектов выгружать.

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 архивном формате, по умолчанию сжатом. 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 lob будут созданы отдельные файлы и файл оглавления. Для восстановления используется утилита pg_restore. Можно указывать количество потоков, которые одновременно будут выгружать данные - это основное преимущество по сравнению с форматом custom. Восстановление также 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.

`pg_dump` на низком уровне выполняет команды SELECT.

Использование пайпа (pipe, "канал") позволяет направлять stdout на stdin утилиты psql и перегружать данные не создавая файл, что может сэкономить место в файловой системе.

По умолчанию для форматов custom и directory используется сжатие. Сжатие может существенно (в десятки раз) замедлять выгрузку. Можно использовать более быстрый алгоритм -Z zstd или отключить сжатие параметром -Z 0

https://docs.tantorlabs.ru/tdb/ru/15_4/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`

Время выгрузки почти линейно зависит от объема выгружаемых данных. Поскольку в процессе выгрузки данные обрабатываются на логическом уровне, то узким местом может стать ядро центрального процессора, которое будет обслуживать `pg_dump`. Для уменьшения времени выгрузки может использоваться выгрузка в несколько потоков. Выгрузку будут выполнять серверный процесс и рабочие процессы. Одну таблицу может выгружать один рабочий процесс. Выгружать в параллельном режиме можно только в формате directory. Другие три формата выгружают в один поток. Количество потоков нужно задать 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. Количество таких блокировок ограничено значением `max_locks_per_transaction * (max_connections + max_prepared_transactions)`. Если число выгружаемых объектов (таблиц) превышает это число, то серверный процесс выдаст ошибку о превышении количества блокировок и завершится не начав выгрузку. В этом случае можно посчитать количество запланированных к выгрузке таблиц и увеличить max_locks_per_transaction на основном кластере. На всех репликах значения вышеуказанных параметров должны быть не меньше, чем на мастере.

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`, а также команды, которые могут устанавливать эту блокировку на короткое время в конце своей работы. Если какая-то сессия запросит блокировку на объект в режиме в монопольном режиме, то запрос на блокировку будет поставлен в очередь и не де 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.

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 специфический 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

`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 вполняется цтилитой psql, а не `pg_restore`.

Работает в трёх режимах:

1)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. Если указать ключ, а значение не указывать, то будет использоваться переменная окружения `PGDATABASE`. Если и переменная не задана, то в качестве имени базы будет взято имя пользователя операционной системы. В этом режиме возможна загрузка в несколько сессий для входных файлов дампа в формате custom или directory. Формат tar не поддерживает параллельную загрузку. Параллельные процессы выполняют наиболее длительные действия, такие как загрузка данных в таблицы и создание индексов

2)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)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`. Скрипт генерируется одним процессом.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/app-pgrestore.html

Обзор возможностей 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, триггера

Посмотрим основные 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, lob, 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 ней
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

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 для параметров конфигурации) и последовательно запускает pg_dump для каждой базы данных в кластере в режиме plain. Для подсоединения требуется подключаться к экземпляру несколько раз к каждой из баз данных. Если используется аутентификация по паролю, то 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 можно создать утилитой initdb. 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 и перегружать данные не создавая файл, что может сэкономить место в файловой системе

https://docs.tantorlabs.ru/tdb/ru/15_4/se/app-pg-dumpall.html

Функционал 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

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. 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 ключём 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 ее 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

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 lob. 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.

Демонстрация

- Обработка срок большого размера

07b. Демонстрация

Обработка строк большого размера

1) Выполните команды:

```
drop table if exists t2;
create table t2 (c1 text, c2 text);
insert into t2 (c1)
VALUES (repeat('a', 1024*1024*512));
update t2 set c2 = c1;
select * from t2;
```

При выполнении команды `select` появится ошибка:

```
ERROR:  out of memory
DETAIL:  Cannot enlarge string buffer containing 536870922 bytes by 536870912
more bytes.

[31089] ERROR:  out of memory
[31089] DETAIL:  Cannot enlarge string buffer containing 536870922 bytes by
536870912 more bytes.
[31089] STATEMENT:  select * from t2;
```

При выборке в строковый буфер выбиралось значение поля c1 плюс 10 байт. Для выборки значения второго поля `c2` буфер пытался увеличиться на его размер поля `c2`.

2) Попробуем с меньшими полями:

```
drop table if exists t1;
create table t1 (c1 text, c2 text, c3 text, c4 text);
insert into t1 (c1) VALUES (repeat('a', 1024*1024*256));
update t1 SET c2=c1;
update t1 SET c3=c1;
update t1 SET c4=c1;
select * from t1;
```

Появится ошибка:

```
ERROR:  out of memory
DETAIL:  Cannot enlarge string buffer containing 805306386 bytes by 268435456
more bytes.
```

При выборке в строковый буфер выбирались значения полей `c1, c2, c3`. Буфер достиг размера трёх полей плюс 18 байт. При увеличении размера буфера на размер поля c4 возникла ошибка превышения границы 1Гб.

3) Выполните команду:

```
postgres=# COPY t2 TO '/tmp/test';
ERROR:  out of memory
DETAIL:  Cannot enlarge string buffer containing 536870913 bytes by 536870912
more bytes.
```

В логе кластера будут сообщения:

```
20:17:50.015 MSK [31089] ERROR:  out of memory
20:17:50.015 MSK [31089] DETAIL:  Cannot enlarge string buffer containing
536870913 bytes by 536870912 more bytes.
20:17:50.015 MSK [31089] STATEMENT:  COPY t2 TO '/tmp/test';
```

Возникла та же самая ошибка.

4) Строки больше 1Гб можно выгрузить по отдельным полям:

```
postgres=# COPY t2 (c1) TO '/tmp/test';
COPY 1
postgres=# \! rm /tmp/test
```

5) Выполните:

```
drop table if exists t2;
create table t2 (c1 text);
insert into t2 (c1) VALUES (repeat(E'a\n', 357913941));
COPY t2 TO '/tmp/test';
```

Появится ошибка:

```
postgres=# COPY t2 TO '/tmp/test';
ERROR:  out of memory
DETAIL:  Cannot enlarge string buffer containing 1073741822 bytes by 1 more
bytes.
```

Было превышено на 1 байт ограничение на память строкового буфера.

В логе кластера будут сообщения:

```
20:23:51.783 MSK [31089] ERROR:  out of memory
20:23:51.783 MSK [31089] DETAIL:  Cannot enlarge string buffer containing
1073741822 bytes by 1 more bytes.
20:23:51.783 MSK [31089] STATEMENT:  COPY t2 TO '/tmp/test';
```

Размер поля - треть гигабайта с округлением в меньшую сторону.
При выгрузке в текстовом виде содержимое поля будет выглядеть так:
`a\na\na\na\n` и размер поля увеличится в три раза до 1073741823 байт, что на 1 байт превышает максимальную границу.

6) При использовании формата `binary` поле можно выгрузить:

```
postgres=# COPY t2 TO '/tmp/test' WITH BINARY;
COPY 1
postgres=# \! rm /tmp/test
```

7) удалите таблицы:

```
drop table t1;
drop table t2;
```

Примечание:

Если на виртуальной машине не хватает физической памяти для выделения буфера обработки строк, то экземпляр может аварийно остановиться.

Следующий пример можно не выполнять:

```
drop table if exists t2;
create table t2 (c1 text, c2 text);
insert into t2 (c1) values (repeat('a', 1024*1024*1024-69));
```

В процессе выполнения команды **insert**, если успеть, то можно во втором окне показать как менялся объем памяти:

```
postgres@tantor:~$ free -b -w

postgres@tantor:~/tantor-se-16/data/base/5$ free -b -w
              total        used        free      shared     buffers
cache   available
Mem:     8325275648  3656470528  2537848832  1463402496    77914112
2053042176  2886438912
Swap:             0           0           0
```

```
               total        used        free      shared     buffers
cache   available
Mem:     8325275648  5789761536   412213248  1463402496    80195584
2043105280   761610240
Swap:             0           0           0
```

Использование памяти **увеличилось примерно на 2Гб (2125635584 байт)**. Свободной памяти осталось 400Мб.

```
update t2 set c2 = c1;
select * from t2;

сервер неожиданно закрыл соединение
        Скорее всего сервер прекратил работу из-за сбоя
        до или в процессе выполнения запроса.
Подключение к серверу потеряно. Попытка восстановления неудачна.
Подключение к серверу потеряно. Попытка восстановления неудачна.
!?> \q
postgres@tantor:~$ psql
psql (16.1)
Введите "help", чтобы получить справку.
postgres=# drop table t2;
DROP TABLE
```

Такая ошибка возникнет при нехватке физической памяти. Серверный процесс пытается выделить чуть меньше **4Гб** памяти, а свободной памяти в данном примере 2.5Гб. **oom-kill** (out of memory killer) убил серверный процесс. Процесс `postgres` остановил все процессы и запустил фоновые процессы.

Сообщения в логе кластера:

```
[31030] LOG:  server process (PID 31038) was terminated by signal 9: Killed
[31030] DETAIL:  Failed process was running: COPY t1 TO '/tmp/test' WITH BINARY;
[31030] LOG:  terminating any other active server processes
[31030] LOG:  all server processes terminated; reinitializing
[31039] LOG:  database system was interrupted; last known up at 19:58:59 MSK
[31042] FATAL:  the database system is in recovery mode
Failed.
[31039] LOG:  database system was not properly shut down; automatic recovery in
progress
[31039] LOG:  redo starts at 116/CE344C0
[31039] LOG:  invalid record length at 116/DF34798: expected at least 26, got 0
[31039] LOG:  redo done at 116/DF34770 system usage: CPU: user: 0.02 s, system: 0.12
s, elapsed: 0.15 s
[31040] LOG:  checkpoint starting: end-of-recovery immediate wait
[31040] LOG:  checkpoint complete: wrote 2105 buffers (12.8%); 0 WAL file(s) added, 0
removed, 0 recycled; write=0.025 s, sync=0.003 s, total=0.031 s; sync files=25,
longest=0.001 s, average=0.001 s; distance=17408 kB, estimate=17408 kB;
lsn=116/DF34798, redo lsn=116/DF34798
[31030] LOG:  database system is ready to accept connections
```

Сообщение в журнале операционной системы:

```
tantor kernel: oom-kill:constraint=CONSTRAINT_NONE,nodemask=(null),
cpuset=user.slice, mems_allowed=0,global_oom,task_memcg=/system.slice/tantor-
se-server-16.service, task=postgres,pid=4647,uid=997
```

В примере **oom-kill** посылает сигнал 9 (`SIGKILL`) серверному процессу, но он может послать этот сигнал и другим процессам, которые выделили много памяти. Процесс `postgres` останавливает все процессы и снова запускает процессы, как при запуске экземпляра.

Практика

1. **Использование утилиты pg_dump**
2. **Формат custom и утилита pg_restore**
3. **Формат directory**
4. **Сжатие и скорость резервирования**
5. **Команда COPY**

08a 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

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.

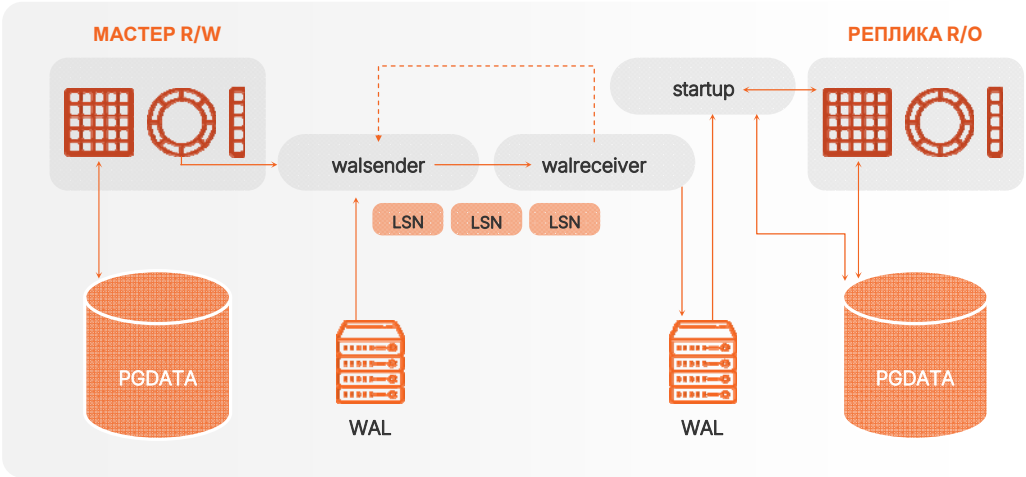
В этой главе мы рассмотрим простое и наиболее распространённое решение обеспечения высокой доступности - репликацию изменений (журнальных записей) в данных на физическом уровне (страниц файлов данных) - "физическую репликацию".

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/15_4/se/high-availability.html

Мастер и реплика



Мастер и реплики должны использовать ту же основную версию СУБД Тантор. 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 каскадирование (cascading replication). Реплики, получающие журнальные записи не от мастера 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 утилитой `pg_receivewal` (со слотом репликации)
- реплика может использовать и слот репликации и если не сможет получить через него файл журнала, то директорию с архивом журналов если она есть

Передавать на реплики журнальные записи можно всеми доступными способами. Например, 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`). Б**о**льшие возможности даёт получение журнальных записей по протоколу репликации с использованием слота репликации, так же как и утилита `pg_receivewal`. Только вместо `pg_receivewal` используется фоновый процесс 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 выполнит команду, указанную в `restore_command` и если команда успешно выполнится попробует прочесть файл журнала. При этом реплика будет пытаться восстановить соединение по протоколу репликации и если сможет получать журнальные записи по протоколу репликации, то будет это делать.

Настройка основного кластера

- Настроить возможность подсоединения реплик по протоколу репликации
- Установить 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

Основной кластер (мастер) уже 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 конфиурации.

`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/15_4/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

После настройки мастера можно приступить к созданию 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. При промышленной эксплуатации реплика должна работать на хосте, отличном от хоста мастера. Для упрощения настройки стоит использовать директории `PGDATA` и табличных пространств такие же как на мастере, но это необязательно.

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 не завершать восстановление, а находиться в режиме постоянного восстановления, кластер не открывается на запись. При наличии файла recovery.signal действует standby.signal.

3) После создания реплики установить на ней значение параметра конфигурации `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) Настроить службу для автоматического запуска экземпляра реплики и запустить экземпляр реплики

Слоты репликации

- используются репликами
- 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')`

Причин не использовать слоты репликации нет. Слот использует и утилита `pg_receivewal` и реплика. 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 резервный кластер. 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_wa`l 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`

Дополнительные параметры конфигурации на репликах

- на мастере не играют роли, но могут быть заранее установлены
- многие не меняются без рестарта экземпляра реплики
- по умолчанию:
- включён горячий резерв (`hot_standby=on`)
- обратная связь отключена (`hot_standby_feedback=off`)
- реплика применяет журнальные записи без задержки (`recovery_min_apply_delay=0`)

Часть параметров конфигурации настраивают работу экземпляра который обслуживает реплику. 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. Эти 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, чем раз в этот интервал. Горизонт событий баз данных на мастере будет сдвигаться не чаще чем этот интервал.

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. Будет рассмотрен позже.

hot_standby по умолчанию on. Определяет можно ли будет подключаться к экземпляру и выполнять запросы (горячий резерв) или нельзя (тёплый резерв). 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 меняется только с перезапуском экземпляра. 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 `walsender`у, 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 механизмом `HOT - Heap-Only Tuples`, зато благодаря этому запросы не реплике не получают ошибку "`snapshot too old`" (слишком старая моментальная копия) 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 равно 30 секунд. Максимально допустимое время задержки применения WAL

https://docs.tantorlabs.ru/tdb/ru/15_4/se/runtime-config-replication.html#RUNTIME-CONFIG-REPLICATION-STANDBY

Горячая реплика

- включен по умолчанию параметром `hot_standby=on`
- реплика принимает соединения и выполняет запросы
- `pg_basebackup` может выполнять резервирование реплики вместо мастера
- временные таблицы на реплике использовать нельзя

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/15_4/se/hot-standby.html

Обратная связь

- обратная связь по умолчанию отключена
  `hot_standby_feedback=off`
- Можно использовать параметры `max_standby_streaming_delay` и `max_standby_archive_delay`

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 оптимизацию HOT.

Мониторинг горизонта событий

- 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

Проверять сдвигается ли горизонт событий 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. Для мониторинга удобнее использовать Платформу Тантор. Информацию можно получить из представлений системного каталога.

Количество отменённых запросов в базах данных реплики с момента сброса статистики можно просмотреть в представлении `pg_stat_database_conflicts` на реплике.

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 баз данных кластера:

```
select datname, extract(epoch from max(now()-xact_start)) from
pg_stat_activity WHERE backend_xmin IS NOT NULL OR backend_xid IS NOT
NULL group by datname order by datname;
```

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`). Пример запроса:

```
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 backend_xmin, backend_xid, pid, datname, 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;
```

https://docs.tantorlabs.ru/tdb/ru/15_4/se/monitoring-stats.html

Параметры которые должны быть синхронизированы

- 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`

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/15_4/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 полученный LSN на реплике

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`

Для того чтобы реплика стала мастером (promote, продвинуть или повысить) можно 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` может рассматриваться только как способ сменить роли с корректной остановкой мастера перед продвижением реплики.

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

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-16-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-16-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-16-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/15_4/se/continuous-archiving.html#BACKUP-TIMELINES

Утилита pg_rewind

- синхронизирует директорию PGDATA и табличных пространств с исходным кластером, в том числе заменяет файлы параметров
- 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

```
postgres@tantor:~/tantor-se-16-replica/data1$ cat backup_label
START WAL LOCATION: 116/E3016108 (file 000000020000116000000E3)
CHECKPOINT LOCATION: 116/E3016108
BACKUP METHOD: pg_rewind
BACKUP FROM: standby
START TIME: 2024-03-29 12:02:10 MSK
```

Утилита pg_rewind синхронизирует директорию кластера (PGDATA и директории табличных пространств) с директорией другого кластера, на основе которого был создана синхронизируемая директория.

Для работы утилиты существенным является наличие ветвления 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 история линии времени разошлась и до текущего файла журнала того кластера, чью директорию будет синхронизировать ("целевой" кластер). 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, 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. Если содержимое файлов параметров важно, то стоит их сохранить до запуска утилиты, чтобы после ее работы восстановить.

Обычно утилита используется для того, чтобы вернуть в работу бывший мастер в результате незапланированной смены ролей (failover). Если бывший мастер не успеет передать хотя бы одну журнальную запись на реплику, которая станет мастером, то бывший мастер не может работать репликой. Чтобы полностью не пересоздавать бывший мастер и используется утилита pg_rewind.

Существенным является, 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/15_4/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`

```
~$ ps -o pid,command --ppid `head -n 1 /var/lib/postgresql/tantor-se-16/data/postmaster.pid`

master: checkpointer
master: background writer
master: walwriter
master: autovacuum launcher
master: logical replication launcher
master: walsender postgres [local] streaming 116/E1000358
~$
~$ ps -o pid,command --ppid `head -n 1 /var/lib/postgresql/tantor-se-16-replica/data1/postmaster.pid`

replica1: checkpointer
replica1: background writer
replica1: startup recovering 0000000100000116000000E1
replica1: walreceiver streaming 116/E1000358
```

На экземпляре реплики присутствуют процессы:

1) postgres - основной процесс. прослушивает сокеты, запускает процессы
2) checkpointer. Контрольные точки инициируются только на мастере. На реплике 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 кеша 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`
- параметр устанавливается на уровне кластера, при изменении достаточно перечитать файлы параметров

```
\df pg*replay*

 |              Имя
-+-------------------------------
 | pg_get_wal_replay_pause_state
 | pg_is_wal_replay_paused
 | pg_last_wal_replay_lsn
 | pg_last_xact_replay_timestamp
 | pg_wal_replay_pause
 | pg_wal_replay_resume
```

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 (утилитой `pg_basebackup`) 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.

Важно помнить, что 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). Нужно с осторожностью использовать обратную связь при использовании задержки.

Также если `synchronous_commit=remote_apply` на мастере и реплика единственная в конфигурации или должна использоваться для подтверждения транзакций, то все транзакции будут подвисать на время установленной 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. Например, pg_wal_replay_pause() позволяет приостановить восстановление. Его 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.

Расширение page_repair

page_repair

Модуль **для восстановления поврежденных страниц** из резервных данных сервера репликации в PostgreSQL. Позволяет экономить время, восстанавливая **только отдельные поврежденные страницы,** без необходимости восстанавливать всю базу данных. Функция **pg_repair_page** принимает имя таблицы, номер поврежденного блока и строку подключения к ведомому серверу. Может использоваться для восстановления объектов, таких как карта свободного пространства и другие. Внимание: функция устанавливает блокировку на таблицу и может ожидать синхронизации с ведущим сервером.

Все поставляемые модули собраны и проверены на совместимость и корректность функционала.
Все модули доработаны и их поведение может отличаться от тех, которые находятся в открытом доступе.

Расширение `page_repair` доступно во всех версиях СУБД Tantor.

```
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';
```

Модуль для восстановления отдельных поврежденных страниц с использованием резервных данных с сервера репликации. Позволяет сэкономить время на восстановление т.к. не требует восстановления всех данных, а только отдельных страниц

Расширение page_repair предоставляет функцию pg_repair_page (table regclass, block_number bigint, connstr text).

- table и block_number - это имя таблицы и номер поврежденного блока соответственно.
- connstr - это строка подключения к ведомому серверу.

Ведомый сервер, к которому можно подключиться с помощью connstr, должен иметь тот же системный идентификатор, что и сервер, на котором выполняется данная функция. Эта функция может быть выполнена на ведущем сервере и только суперпользователем. Если вы хотите восстановить другие объекты, такие как карта свободного пространства, карта видимости, вы можете использовать функцию pg_repair_page (table regclass, block_number bigint, connstr text, forkname text), где forkname может быть main, fsm или vm.

Функция pg_repair_page устанавливает AccessExclusiveLock на целевую таблицу и может ожидать, пока ведомый сервер не сравняется с ведущим сервером. Эта функция не пытается восстановить страницу, которая отмечена как грязная в общем буфере, поскольку грязная страница будет выгружена на диск и тем самым могла бы исправить поврежденную страницу.

Восстановление повреждённых страниц

- Расширение `page_repair` содержит функции `pg_repair_page` для восстановления повреждённых страниц
- за один вызов процедуры восстанавливается одна страница
- можно восстановить страницы слоёв `main, vm, fsm`
- нужна монопольная блокировка на объект
- страница запрашивается с реплики через соединение
- параметры соединения передаются функции

В СУБД Тантор всех сборок имеется расширение `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/15_4/**be**/page_repair.html

Изменения в ядре: производительность СУБД

standby fadvise

Добавлен патч в PostgreSQL для улучшения производительности через предварительное чтение блоков WAL (Write-Ahead Log) с использованием системного вызова posix_fadvise() с параметром POSIX_FADV_WILLNEED. Это оптимизирует чтение с диска, предварительно загружая данные в кэш. Также внедрено новое событие ожидания WAIT_EVENT_WAL_PREFETCH, позволяющее более детально анализировать производительность и определить, улучшает ли предварительное чтение WAL общую эффективность. Описаны также функции InitWalRecovery(), PerformWalRecovery(), и EndWalRecovery(), а также уточнено использование posix_fadvise() для оптимизации доступа к файлу данных.

Для всех версий СУБД Tantor доступна о**птимизация и ускорение процесса восстановления каскадной репликации (standby fadvise)**.

Патч для повышения производительности путем добавления предварительного чтения для WAL (Write-Ahead Log) блоков в PostgreSQL, используя системный вызов `posix_fadvise()` с параметром `POSIX_FADV_WILLNEED`. Это указывает операционной системе, что определенные блоки данных скоро потребуются, что позволяет ей оптимизировать чтение с диска, упреждающе загружая данные в кэш.

Также в этом патче добавляется новое событие ожидания `WAIT_EVENT_WAL_PREFETCH` для отслеживания времени, затраченного на предварительное чтение WAL. Это позволяет администраторам БД более детально анализировать производительность системы и определять, улучшает ли предварительное чтение WAL общую производительность

Программы могут использовать posix_fadvise для объявления намерений осуществить доступ к файлу данных по особому шаблону в скором будущем, тем самым позволяя ядру выполнить некоторые операции по оптимизации.

advice применяется к (не обязательно существующей) области, начинающейся с offset, длиной len байтов (или до конца файла, если len равно 0) внутри файла, на который ссылается fd.

xlogrecovery файл содержит функции, управляющие восстановлением WAL.

- InitWalRecovery() инициализирует систему для аварийного восстановления, восстановления архива или режима ожидания, в зависимости от параметров конфигурации и состояния контрольного файла и возможного файла метки резервной копии.
- PerformWalRecovery() выполняет реальное воспроизведение WAL, вызывая процедуры повторного выполнения, специфичные для rmgr.
- EndWalRecovery() выполняет проверки завершения восстановления и действия по очистке, а также подготавливает информацию, необходимую для инициализации WAL для записи.

Помимо этих трех основных функций есть еще много функций опроса состояния восстановления и управления процессом восстановления.

Дополнительно поставляемые программы

Расширение	Tantor SE	Tantor SE 1C	Tantor BE	PostgreSQL
pg_cluster	✓	✓	✓	

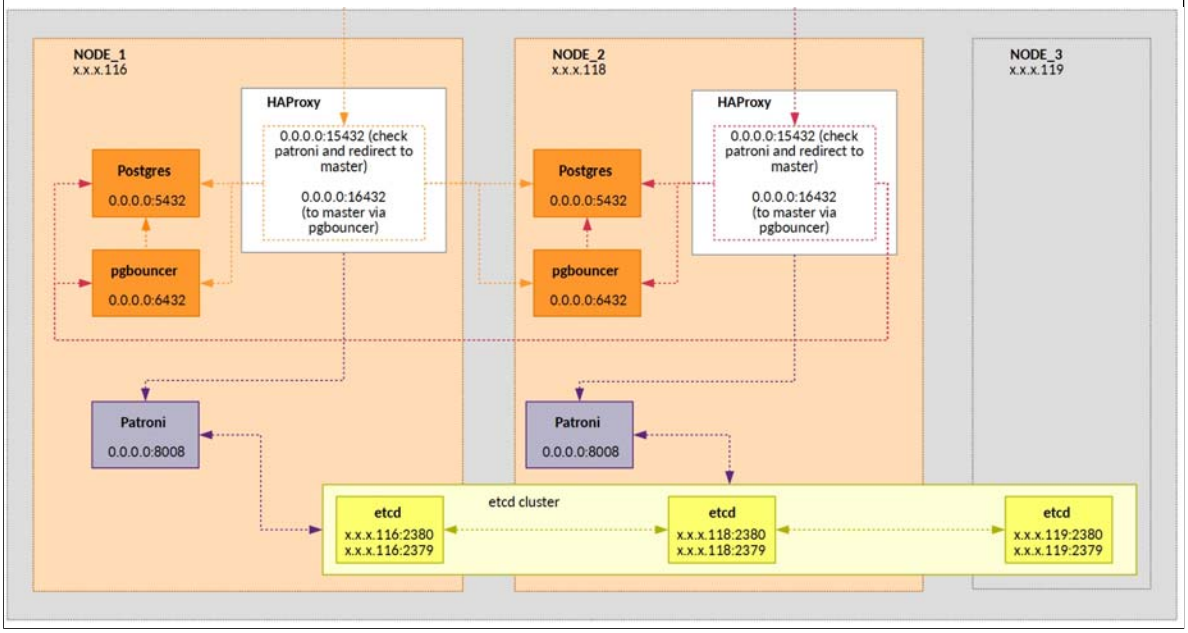
Разработан ansible playbook для **упрощения развертывания отказоустойчивых решений** на базе СУБД Tantor с использованием кластеризации Patroni под управлением OS Astra Linux, с целью минимизации трудозатрат администратора

Во всех версиях СУБД Tantor доступен pg_cluster.

Для минимизации трудозатрат администратора нами был разработан готовый ansible playbook для развертывания отказоустойчивых решений на базе СУБД Tantor под управлением OS Astra Linux посредством кластеризации Patroni.

Программа `pg_cluster`

- набор инструкций Ansible для создания кластера PostgreSQL с высокой доступностью на базе Patroni



Дополнительно поставляемая с СУБД Тантор программа `pg_cluster` это набор инструкций Ansible playbook для создания кластера PostgreSQL с высокой доступностью на базе Patroni. Минимизирует трудозатраты администратора.

Доступна для всех сборок СУБД Тантор.

Страница с описанием утилиты:
https://github.com/TantorLabs/pg_cluster/

Демонстрация

- Создание и запуск реплики

08a. Демонстрация

До выполнения демонстрации проверьте есть ли табличные пространства:

```
postgres=# \db
                          List of tablespaces
    Name    |  Owner   |                    Location
------------+----------+-----------------------------------------------
 pg_default | postgres |
 pg_global  | postgres |
 u01tbs     | postgres | /var/lib/postgresql/tantor-se-16/data/../u01
(3 rows)
```

Если есть созданные ранее табличные пространства, то удалите их. Если табличное пространство не содержит объектов, то оно удалится командой:

```
postgres=# drop tablespace u01tbs;
DROP TABLESPACE
```

Если не удалится, так как есть объекты, то список отношений в текущей базе данных можно получить командой:

```
SELECT n.nspname, relname
FROM pg_class c
 LEFT JOIN pg_namespace n ON n.oid = c.relnamespace,
 pg_tablespace t
WHERE relkind IN ('r','m','i','S','t') AND
 n.nspname <> 'pg_toast' AND t.oid = reltablespace AND
 t.spcname = 'u01tbs';
```

удалить объекты и потом удалить табличное пространство.

Создание физической реплики

1) Сделаем бэкап с параметрами
-P показывает прогресс резервирования;
-C или **--slot** создает слот;
-R создает файлы конфигурации для реплики:

```
postgres@tantor:~$ pg_basebackup -D /var/lib/postgresql/tantor-se-16-replica/data1 -P -R -C --slot=replica1
```

Если резервирование прервать, то нужно будет удалить директорию:

```
rm -rf /var/lib/postgresql/tantor-se-16-replica/data1
```

и слот на мастере:

```
select pg_drop_replication_slot('replica1');
```

2) После успешного создания бэкапа нужно установить порт для экземпляра реплики. Обязательно две угловые скобки, если будет одна, то файл затрётся:

```
echo "port=5433" >> /var/lib/postgresql/tantor-se-16-replica/data1/postgresql.auto.conf
```

3) Можно запустить реплику:

```
pg_ctl start -D /var/lib/postgresql/tantor-se-16-replica/data1
```

```
ожидание запуска сервера....
[446] СООБЩЕНИЕ:  передача вывода в протокол процессу сбора протоколов
[446] ПОДСКАЗКА:  В дальнейшем протоколы будут выводиться в каталог "log".
 готово
сервер запущен
```

4) На мастере посмотрим, что **физический слот репликации создан** и **активен**:

```
postgres@tantor:~$ psql

postgres=# select * from pg_replication_slots;
 slot_name | plugin | slot_type | datoid | database | temporary | active |
-----------+--------+-----------+--------+----------+-----------+--------+-
 replica1  |        | physical  |        |          | f         | t      |
(1 строка)
```

5) Посмотрим ещё одно представление для мониторинга репликации:

```
postgres=# select * from pg_stat_replication \gx
-[ RECORD 1 ]----+------------------------------
pid              | 12236
usesysid         | 10
usename          | postgres
application_name | walreceiver
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 12:00:59.907801+03
backend_xmin     |
state            | streaming
sent_lsn         | 116/E1000070
write_lsn        | 116/E1000070
flush_lsn        | 116/E1000070
replay_lsn       | 116/E1000070
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
reply_time       | 12:07:11.962288+03
```

Имя приложения по умолчанию **walreceiver**.

6) Подключиться к реплике:

```
postgres=# \q
postgres@tantor:~$ psql -p 5433
```

7) Проверим название слота:

```
postgres=# \dconfig primary_slot_name
List of configuration parameters
     Parameter     |  Value
-------------------+----------
 primary_slot_name | replica1
(1 строка)
```

8) Посмотрим значение параметра `cluster_name`:

```
postgres=# \dconfig cluster_name
List of configuration parameters
  Parameter   | Value
--------------+-------
 cluster_name |
```

значение параметра пусто, поэтому **application_name=walreceiver**

9) Посмотрим значение параметра `primary_conninfo`:

```
postgres=# show primary_conninfo \gx
```

```
-[ RECORD 1 ]-
primary_conninfo | user=postgres passfile='/var/lib/postgresql/.pgpass'
channel_binding=prefer port=5432 sslmode=prefer sslcompression=0
sslcertmode=allow sslsni=1 ssl_min_protocol_version=TLSv1.2 gssencmode=prefer
krbsrvname=postgres gssdelegation=0 compression=off target_session_attrs=any
load_balance_hosts=disable
```

Значнение было сгенерировано автоматически утилитой `pg_basebackup` при использовании параметра **`-R`** на основе параметров с которыми утилита подсоединялась к экземпляру с которого создавала бэкап.

10) Удалим реплику и слот репликации:

```
postgres=# \q
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/tantor-se-16-
replica/data1
ожидание завершения работы сервера....
 готово
сервер остановлен
postgres@tantor:~$ rm -rf /var/lib/postgresql/tantor-se-16-replica/data1
postgres@tantor:~$ psql -c "select pg_drop_replication_slot('replica1')"
 pg_drop_replication_slot
--------------------------

(1 строка)
```

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`

08b Логическая репликация

Обзор

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

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_4/se/release-15.html#id-1.11.6.15.5.4.3

Новые возможности в 16 версии:
https://postgrespro.com/docs/postgresql/16/release-16#RELEASE-16-HIGHLIGHTS

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

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/15_4/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

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 `lob`. Это ограничение нельзя обойти, кроме как хранить данные в обычных таблицах, например, в столбцах с типом данных `bytea`.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/logical-replication-restrictions.html

Идентификация строк

- для репликации только вставок строк (INSERT) 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

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 - они 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

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;
```

Настройка репликации

Шаги по созданию логической репликации

- Установить значение `wal_level=logical`
- 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

Для настройки логической репликации нужно:
1)проверить что параметр `wal_level=logical` на 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 столбцов этих строк, 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

После определения таблиц, которые должны быть включены в одну и ту же публикацию постольку поскольку 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.

Почему не создать одну публикацию? Средний размер строк в таблицах может отличаться. Если есть таблицы со строками большого объема, то изменения в этих строках будут передаваться подписчикам, а пропускная способность сети может быть ограничена и обработка и применение к таблицам подписки всех переданных изменений по всем таблицам при большом объеме изменений может в какие-то диапазоны времени создавать значительный репликационный лаг. Если Разбить таблицы например на две публикации, в одну включить таблицы с небольшими строками, то по ним репликационный лаг может быть меньшим и колебаться в небольших пределах. Универсальных средств определить как разбивать таблицы на несколько публикаций нет, нужно знать как приложение работает с таблицами.

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/15_4/se/sql-createpublication.html

Подписка

- 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

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. Целевая таблица может также 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/15_4/se/logical-replication-subscription.html

Создание 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-приемники
- 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

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

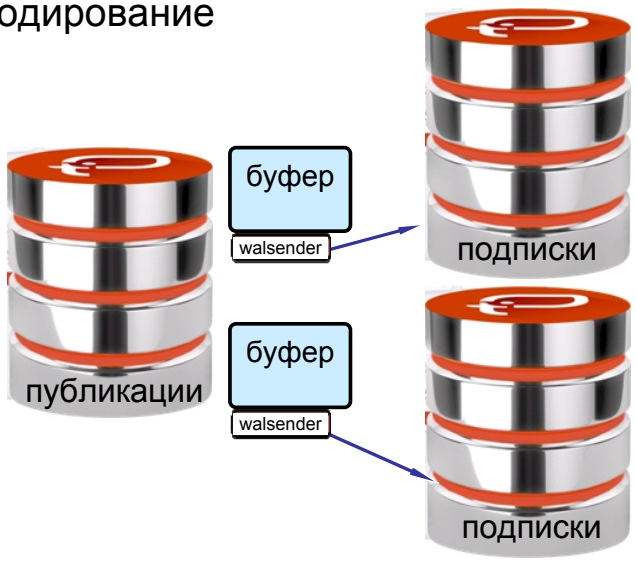
origin = any по умолчанию, публикация отправляет все изменения. Если используется двунаправленная репликация, то нужно установить origin=NONE для предотвращения зацикливания ("ping pong", эхо).

https://docs.tantorlabs.ru/tdb/ru/15_4/se/sql-createsubscription.html

Особенности

Нагрузка на экземпляр

- Для каждой подписки запускается процесс `walsender`, выполняющий буферизацию и логическое декодирование



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`.

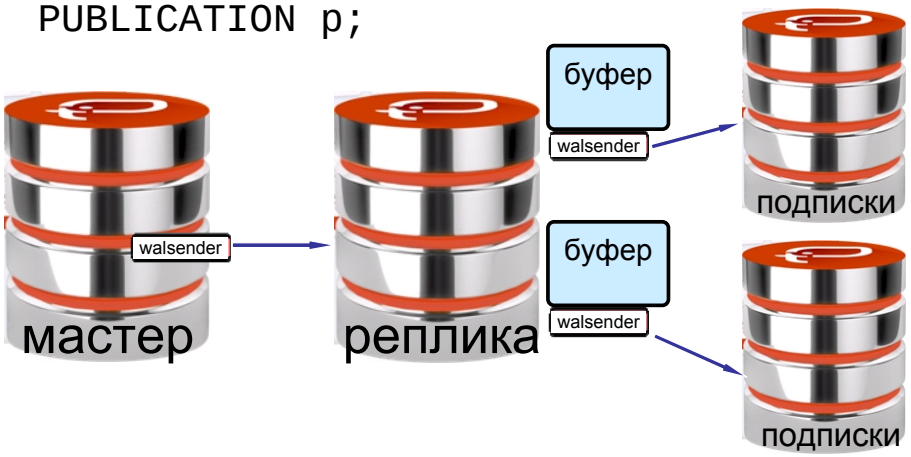
Накопленные в буфере данные по зафиксированным транзакциям (или незафиксированным при значениях `streaming = on` или `parallel`) передаются модулю вывода `pgoutput`. Модуль это отдельный процесс, а код, который выполняет walsender. На работу этого модуля влияет номер основной версии программного обеспечения 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://postgrespro.com/docs/postgresql/16/protocol-logical-replication#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;
```



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 постгрес 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 и 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 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 СУБД Тантор сохраняются после смены ролей.

https://postgrespro.com/docs/postgresql/16/logicaldecoding-explanation

Конфликты

- Конфликт - если `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

Для каждой подписки запускается процесс logical replication worker. Этот процесс подсоединяется к процессу walsender по протоколу репликаиии 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/15_4/se/logical-replication-conflicts.html

Двунаправленная репликация

- Две или более таблицы являются источниками изменений и приёмниками друг для друга
- Публикации и подписки настраиваются отдельно по каждому направлению репликации, настройки обычно зеркальны
- Чем больше лаг, тем больше вероятность конфликта
- Увеличивает отказоустойчивость, а не производительность
- На подписках установить параметр `origin=NONE`

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/15_4/se/replication-origins.html

Расширение pgq

•Расширение для создания
высокопроизводительной
не заблокированной очереди с простым SQL API
•Позволяет асинхронное взаимодействие
компонентов системы через «сообщения»,
обрабатываемые в удобное время
•Подходит для асинхронной репликации данных
и балансировки нагрузки
•Требует программирования для поддержки
конкретных сценариев работы

Особенности:

•Транзакционные очереди
•Продюсеры и консьюмеры
•Перенаправление и повторная попытка

Расширение pgq присутствует только в Tantor SE.

Расширение, предоставляет универсальную высокопроизводительную не заблокированную очередь с простым API, основанным на функциях SQL для создания двунаправленной логической репликации.

pgq позволяет разным компонентам системы взаимодействовать друг с другом асинхронно через «сообщения», которые помещаются в очередь и затем обрабатываются в более удобное для приложения-потребителя время.

pgq полезен для асинхронной репликации, балансировки нагрузки и других задач, где требуется гибкая и надежная система сообщений.

pgq - низкоуровневое расширение и для многих сценариев может потребоваться создание программного кода для поддержки логики приложения.

Особенности:

●Транзакционные очереди: Все операции с очередью (вставка, извлечение, удачное завершение обработки, удаление) являются транзакционными. Это означает, что вы можете полагаться на ACID-гарантии, которые предоставляет PostgreSQL, при работе с очередью.

●Продюсеры и консьюмеры: pgq позволяет нескольким процессам (в том числе разных экземпляров) быть "потребителями" событий из очереди, при этом события могут быть добавлены в очередь с помощью "продюсеров". Это позволяет легко масштабировать и распределять обработку событий.

●Перенаправление и повторная попытка: pgq позволяет настроить повторную попытку обработки событий в случае сбоя, а также перенаправлять события в другие очереди.

Демонстрация

1) Однонаправленная репликация
2) Двунаправленная репликация

08b. Демонстрация

Часть 1. Однонаправленная репликация

1) Подсоединимся к базе данных мастера и создадим таблицу, которую будем реплицировать:

```
postgres@tantor:~$ psql
postgres=# create table t (t text);
CREATE TABLE
```

2) Посмотрим список таблиц, для которых не задан способ идентификации строк:

```
postgres=#  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;

  table
----------
 public.demo2
 public.hypo
 public.t
 utl_file.utl_file_dir
(4 строки)
```

По этим таблицам могут реплицироваться только вставки строк (`INSERT`) и `TRUNCATE`

3) Установим параметр конфигурации `wal_level=logical`. Изменение параметра требует перезапуск экземпляра:

```
postgres=# alter system set wal_level=logical;
ALTER SYSTEM
postgres=# \q
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/tantor-se-16/data
postgres@tantor:~$ sudo systemctl start tantor-se-server-16
```

4) Создадим публикацию:

```
postgres@tantor:~$ psql
postgres=# CREATE PUBLICATION t for TABLE t WITH (publish=
'insert,truncate');
```

Репликация UPDATE и DELETE рассматривается в практике.

5) создадим определение таблицы t в какой-нибудь базе данных этого же кластера

```
postgres=# \! pg_dump -t t --schema-only | psql -d test_db
```

Список баз можно посмотреть командой `\l`

6) Создадим слот логической репликации в базе источника

```
postgres=# select pg_create_logical_replication_slot('s','pgoutput');
 pg_create_logical_replication_slot
------------------------------------
 (s,9/BC0739E8)
(1 строка)
```

7) В базе-приёмнике создадим подписку и укажем имя слота

```
postgres=# \q
postgres@tantor:~$ psql -d test_db
psql (16.1)
Введите "help", чтобы получить справку.

test_db=# CREATE SUBSCRIPTION t CONNECTION 'dbname=postgres user=postgres'
PUBLICATION t WITH (origin=none, create_slot=false, slot_name=s);
CREATE SUBSCRIPTION
```

Слот создали отдельно, потому что если публикация и подписка в одном и том же кластере, то создание подписки подвиснет на создании слота. Можно создать репликацию даже между таблицами той же самой базы данных, но в разных схемах, так как имена таблиц должны быть одинаковыми.

8) можно проверить что вставка строк из одной базы в другую реплицируется.
Запустите другой тепминал или переключитесь в другое окно терминала.

```
postgres@tantor:~$ psql
postgres=# insert into t values ('a');
```

9) В первом окне терминала проверьте, что строка реплицировалась:

```
test_db=# select * from t;
 t
---
 a
(1 строка)
```

10) Аналогично проверьте, что реплицируется команда `TRUNCATE`:

```
postgres=# truncate t;
TRUNCATE TABLE

test_db=# select * from t;
 t
---
(0 строк)
```

Часть 2. Двунаправленная репликация

1) Создадим репликацию в обратном направлении с зеркальными настройками.
Единственно имя слота должно быть уникальным в конфигурации.

```
test_db=# select pg_create_logical_replication_slot('reverses','pgoutput');
 pg_create_logical_replication_slot
------------------------------------
 (reverses,9/BC0817D8)
(1 строка)

test_db=# CREATE PUBLICATION t for TABLE t WITH (publish= 'insert,truncate');
CREATE PUBLICATION
```

2) В другом окне:

```
postgres=# CREATE SUBSCRIPTION t CONNECTION 'dbname=test_db user=postgres'
PUBLICATION t WITH (origin=none, create_slot=false, slot_name=reverses);
WARNING:  subscription "t" requested copy_data with origin = NONE but might
copy data that had a different origin
ПОДРОБНОСТИ:  The subscription being created subscribes to a publication
("t") that contains tables that are written to by other subscriptions.
```

```
ПОДСКАЗКА:  Verify that initial data copied from the publisher tables did not
come from other origins.
CREATE SUBSCRIPTION
```

Педупреждение говорит о том, что при создании подписки данные будут скопированы из таблиц публикующей базы данных. Если в табицах подписчика уже есть эти строки и строки синхронизированы, то стоило бы создавать подписку с параметром **copy_data=off**.
В обоих таблицах нет ни одной строки, поэтому разницы нет.
Использование параметра **copy_data=off** рассматривается в практике. В демонстрации показывается пример предупреждения.

3) Проверим, что репликация работает в обе стороны:

```
test_db=# insert into t values ('b');


postgres=# insert into t values ('a');
INSERT 0 1
postgres=# select * from t;
 t
---
 b
 a
(2 строки)
```

4) Удалим все строки:

```
postgres=# delete from t;
DELETE 2
```

5) Удаление не реплицируется, потому что в публикации указали **publish= 'insert,truncate'**

```
test_db=# select * from t;
 t
---
 b
 a
(2 строки)
```

6) Вставим строку:

```
postgres=# insert into t values ('a');
INSERT 0 1
```

7) Проверим, что строка вставилась:

```
postgres=# select * from t;
 t
---
 a
(1 строка)
```

8) Проверим какие строки есть в таблице второй базе:

```
test_db=# select * from t;
 t
---
 b
 a
 a
(3 строки)
```

Возникла рассинхронизация. Строки на второй таблице не удаляются, но при этом новые строки вставляются. На первой таблице удалили 2 строки, потом вставили одну и получилась одна строка. На второй таблице две строки осталось и добавилась еще одна строка, получилось три строки.

9) Удалим объекты:

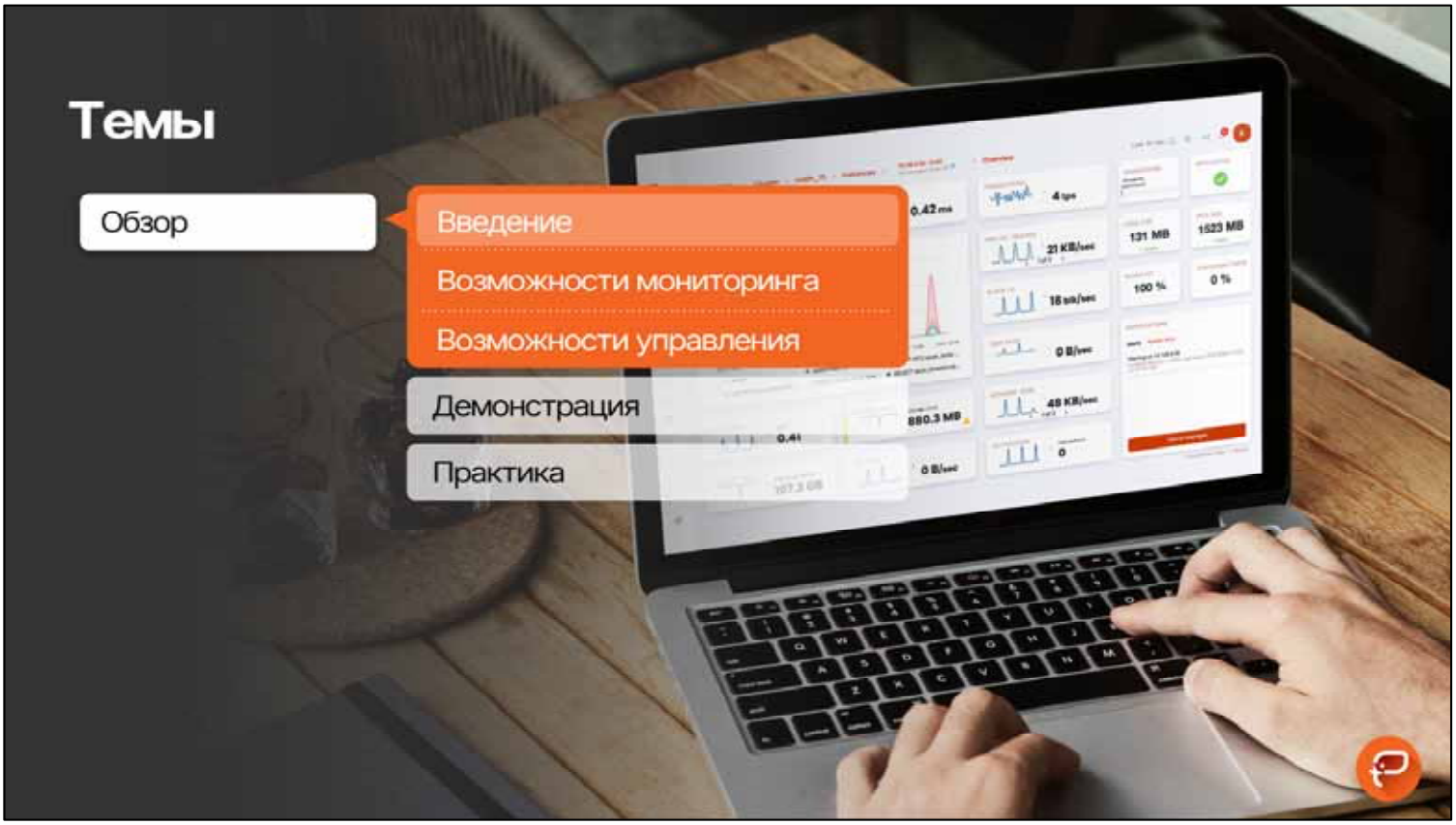
```
test_db=# drop subscription t;
NOTICE:  dropped replication slot "s" on publisher
DROP SUBSCRIPTION
test_db=# drop publication t;
DROP PUBLICATION
test_db=# drop table t;
DROP TABLE
test_db=# \c postgres postgres /var/run/postgresql 5432
Вы подключены к базе данных "postgres" как пользователь "postgres".
postgres=# drop publication t;
DROP PUBLICATION
postgres=# drop subscription t;
NOTICE:  dropped replication slot "reverses" on publisher
DROP SUBSCRIPTION
postgres=# drop table t;
DROP TABLE
```

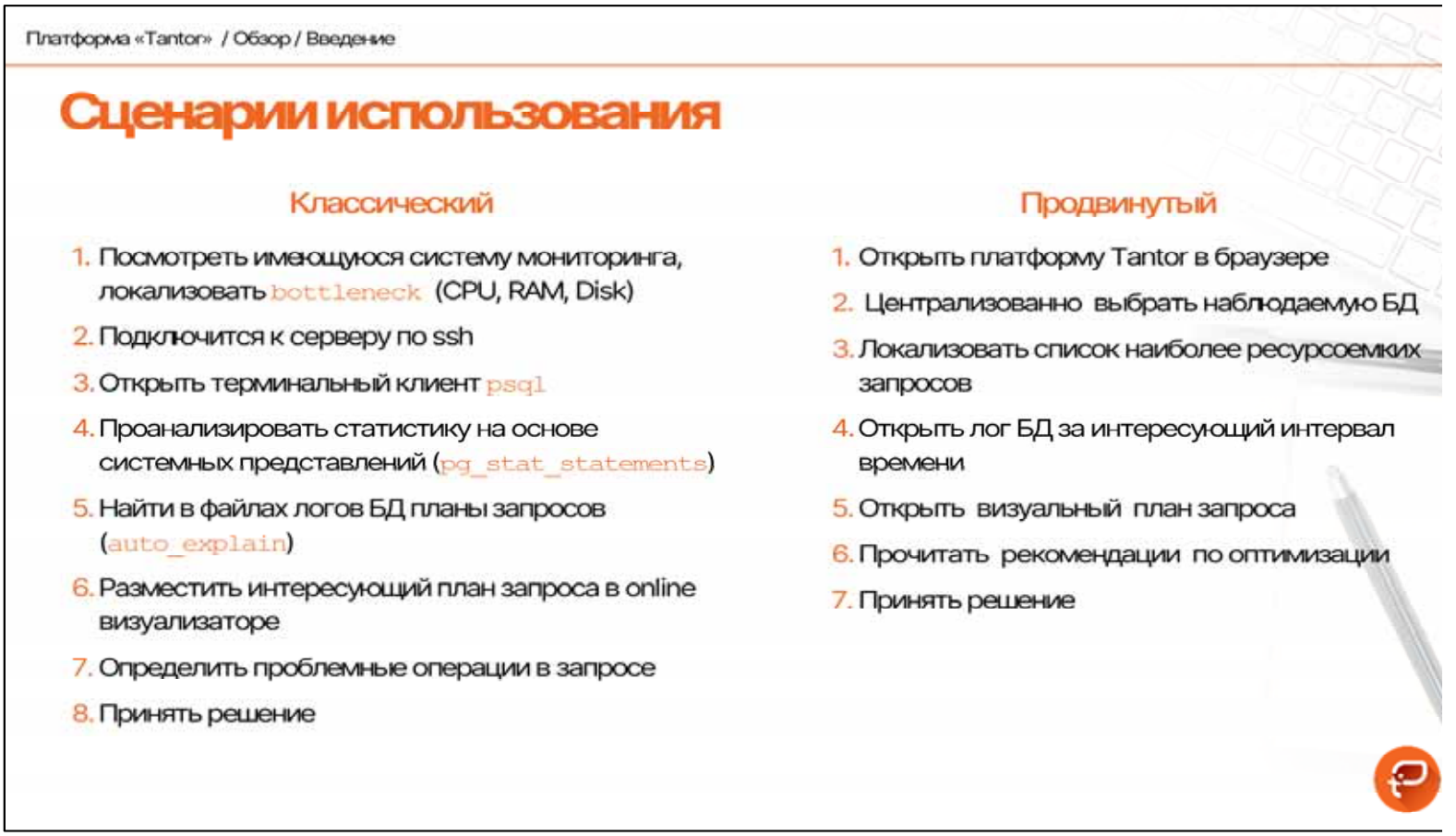
Практика

1. Репликация таблицы
2. Репликация без первичного ключа
3. Добавление таблицы в публикацию
4. Двунаправленная репликация


СУБД Tantor
Платформа «Tantor»
tantor

Темы


Обзор
Введение
Возможности мониторинга
Возможности управления
Демонстрация
Практика



Рассмотрим, как на практике могут выглядеть два подхода (классический и продвинутый) к диагностике и оптимизации производительности СУБД PostgreSQL.

Классический сценарий

Представьте, что у вас есть система мониторинга, которая зарегистрировала резкий спад

В современном мире мониторинг производительности СУБД является критически важным компонентом для обеспечения надежности и эффективности любого приложения или сложной инфраструктуры.

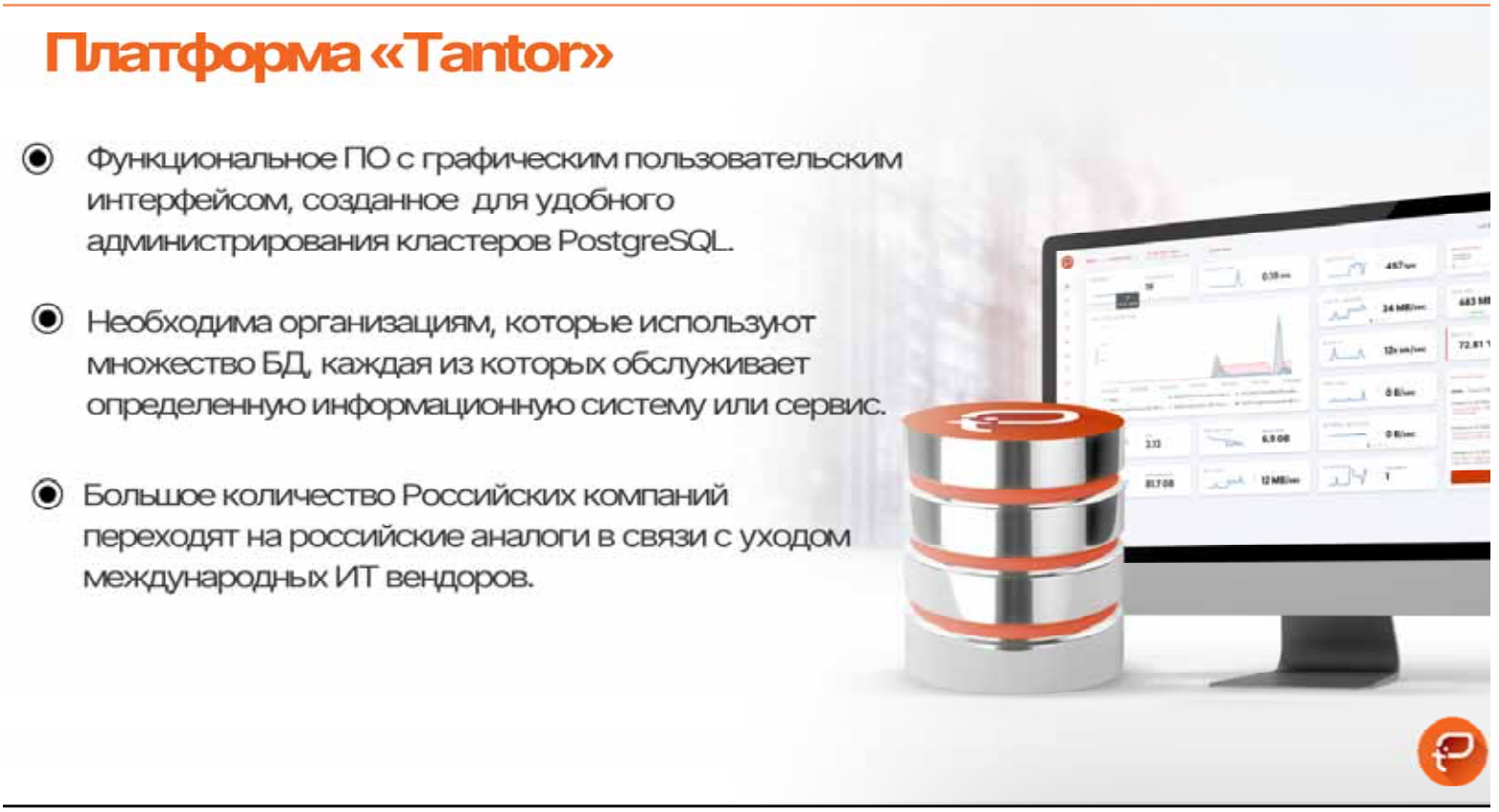
PostgreSQL — одна из наиболее мощных и гибких СУБД, но её потенциал можно полностью раскрыть только с помощью правильных инструментов мониторинга.

Существует много универсальных систем мониторинга, таких как Zabbix, Grafana, OKMeter, New Relic, Munin, Cacti и Datadog. Они предоставляют широкий спектр функций для мониторинга различных аспектов инфраструктуры и приложений. Однако, именно из-за этой «универсальности» они часто не адаптированы для работы со специфическими особенностями и метриками PostgreSQL. Это означает, что они могут не предоставлять всю интересующую вас информацию в наиболее удобной форме, что может стать препятствием для оптимизации производительности.

С другой стороны, есть специализированные решения, такие как Платформа «Tantor», которая разработана специально для мониторинга PostgreSQL. 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 специализированного решения типа Платформа «Tantor» 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 — это функциональное ПО с графическим пользовательским интерфейсом, установленное, как правило в периметре заказчика, создано оно для удобного администрирования кластеров PostgreSQL.

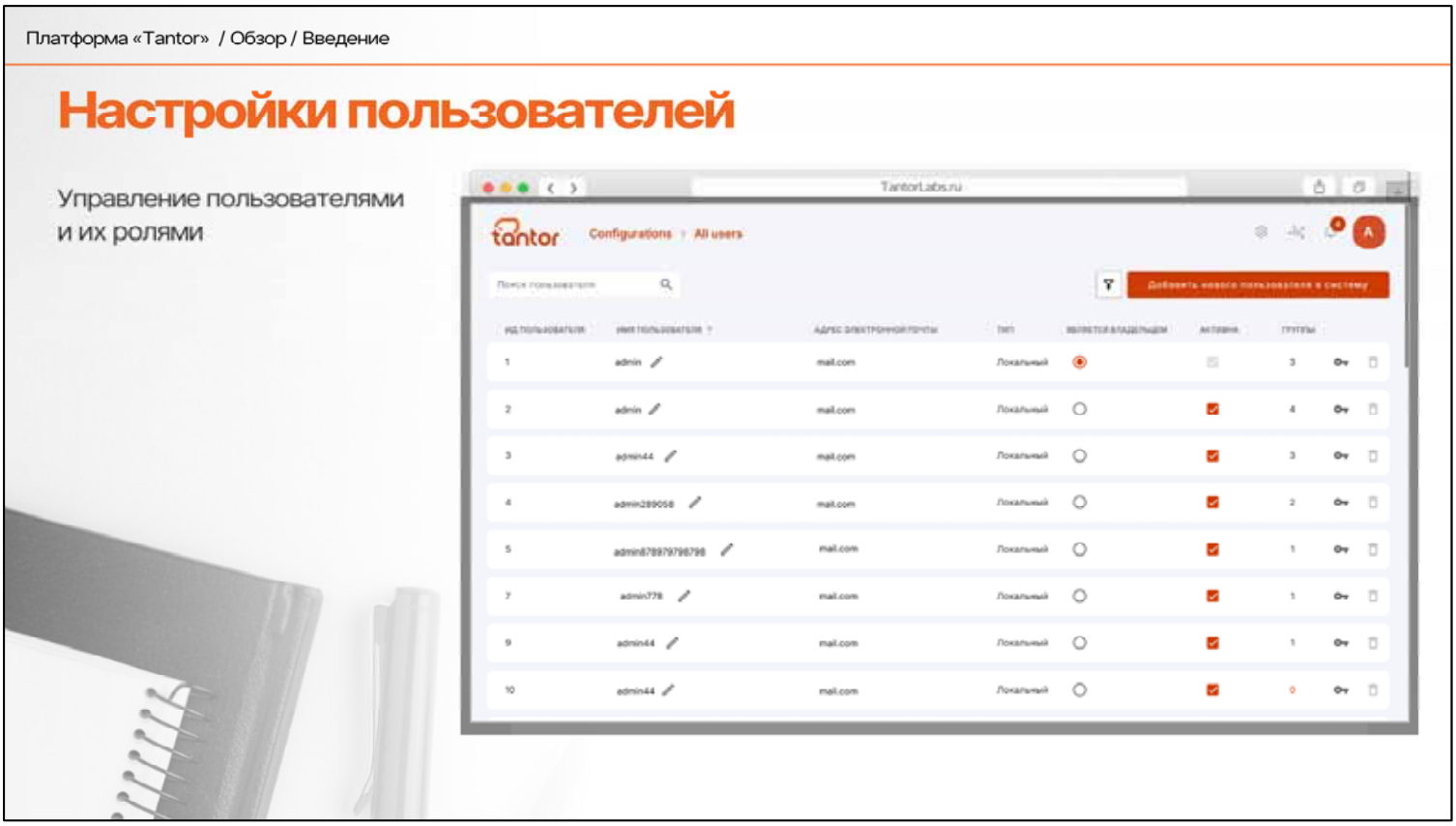
С помощью Платформы «Tantor» можно управлять не только базой данных кластера Tantor, но и любой другой СУБД, основанной на PostgreSQL, включая классическую версию.

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.

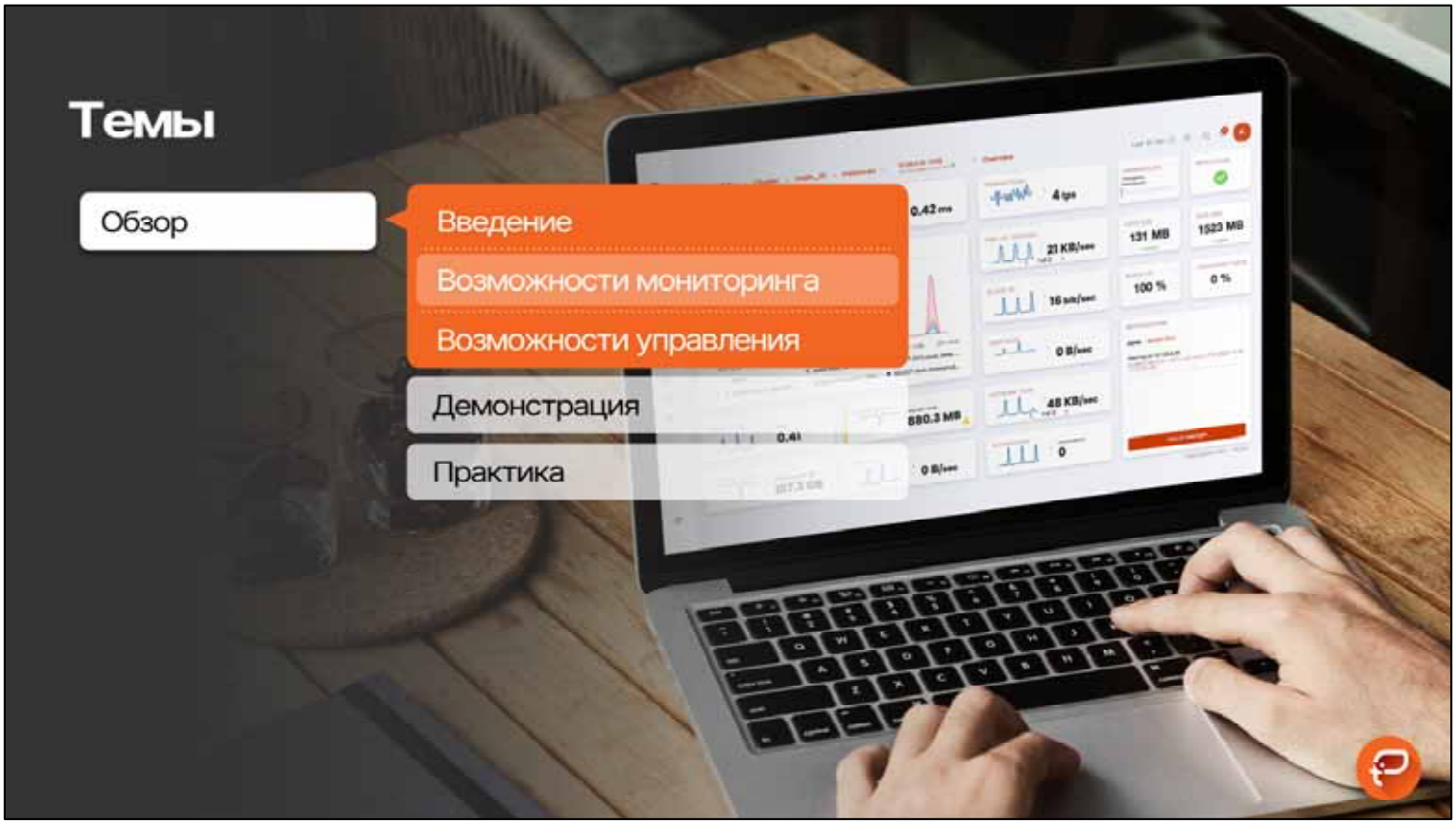
https://docs.tantorlabs.ru/tp/4.0/index.html

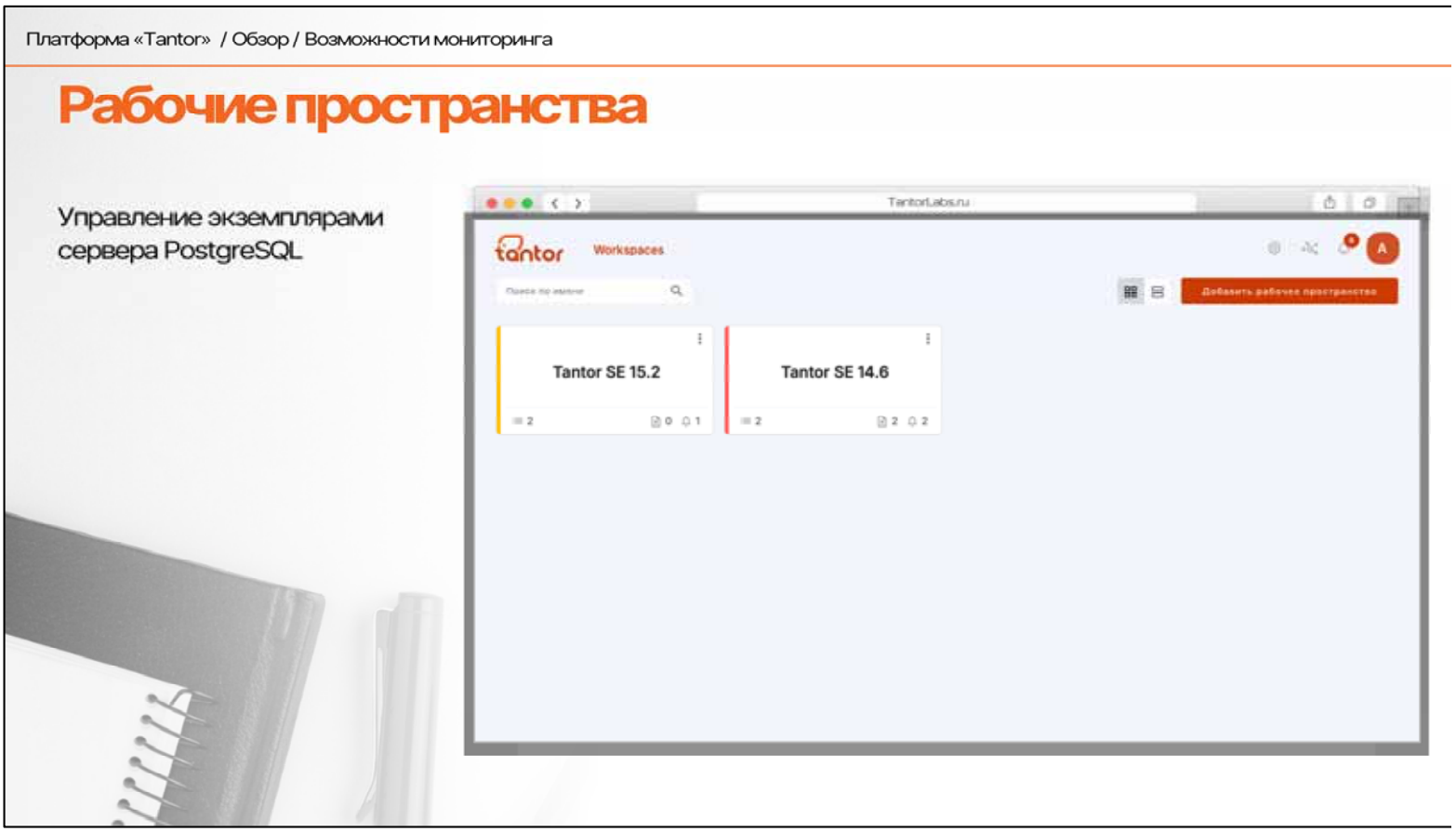


На Платформе Tantor управление пользователями и группами организовано через модуль администрирования, который позволяет выполнять различные операции. Основные функции включают добавление, активацию, деактивацию и удаление пользователей. Также доступны опции для управления правами пользователя, включая передачу прав администратора. Дополнительно, можно управлять группами пользователей, добавляя новые группы и управляя их членством, а также интегрировать систему с Active Directory для более глубокой интеграции с корпоративными учетными записями.

https://docs.tantorlabs.ru/tp/4.0/admin_users.html

Темы


Обзор
Введение
Возможности мониторинга
Возможности управления
Демонстрация
Практика



Рабочее пространство в Платформе Tantor — это веб-интерфейс, предназначенный для управления экземплярами сервера PostgreSQL. Это позволяет пользователям организовывать и управлять различными экземплярами баз данных, которые находятся в одном или нескольких рабочих пространствах.

В рамках рабочего пространства можно создавать новые экземпляры, администрировать существующие, а также проводить различные операции управления и мониторинга. Каждый экземпляр сервера PostgreSQL в рабочем пространстве предназначен для управления одной или несколькими базами данных. Создание нового рабочего пространства требует роли пользователя с административными правами и производится через интерфейс Платформы с помощью специальной формы ввода.

https://docs.tantorlabs.ru/tp/4.0/glossary.html

Дополнительный материал:

Основное преимущество использования рабочих пространств на Платформе Tantor заключается в возможности централизованного мониторинга и управления всеми кластерами баз данных PostgreSQL предприятия. Это позволяет обеспечить единое управление всеми ресурсами баз данных, что критически важно для крупных организаций, стремящихся к оптимизации и повышению эффективности своих IT-операций.

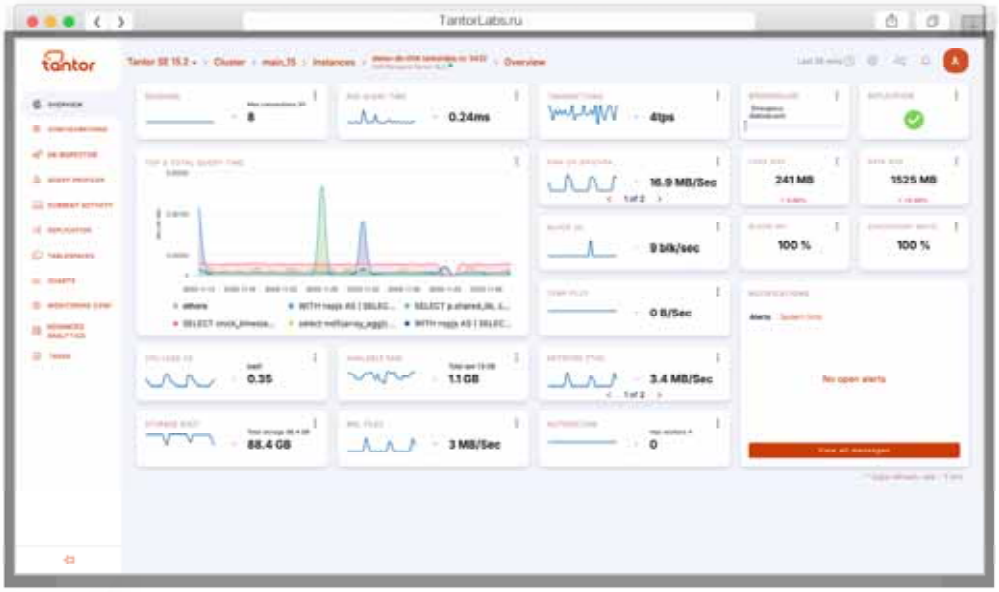
Централизованное управление упрощает процессы настройки, мониторинга и обслуживания различных экземпляров PostgreSQL, обеспечивая:

1. Быстрый доступ ко всей необходимой информации о состоянии каждого экземпляра и кластера.
2. Проактивное управление и мониторинг, позволяющее своевременно реагировать на возможные проблемы или изменения в работе баз данных.
3. Упрощение процессов масштабирования и добавления новых экземпляров и кластеров без необходимости значительной перенастройки системы.

Аспекты делают рабочие пространства незаменимым инструментом для обеспечения надежности, доступности и производительности баз данных в корпоративной среде.

Обзор экземпляра

Консолидация всей важной информации об инстансе PostgreSQL и её представление в упрощенном виде

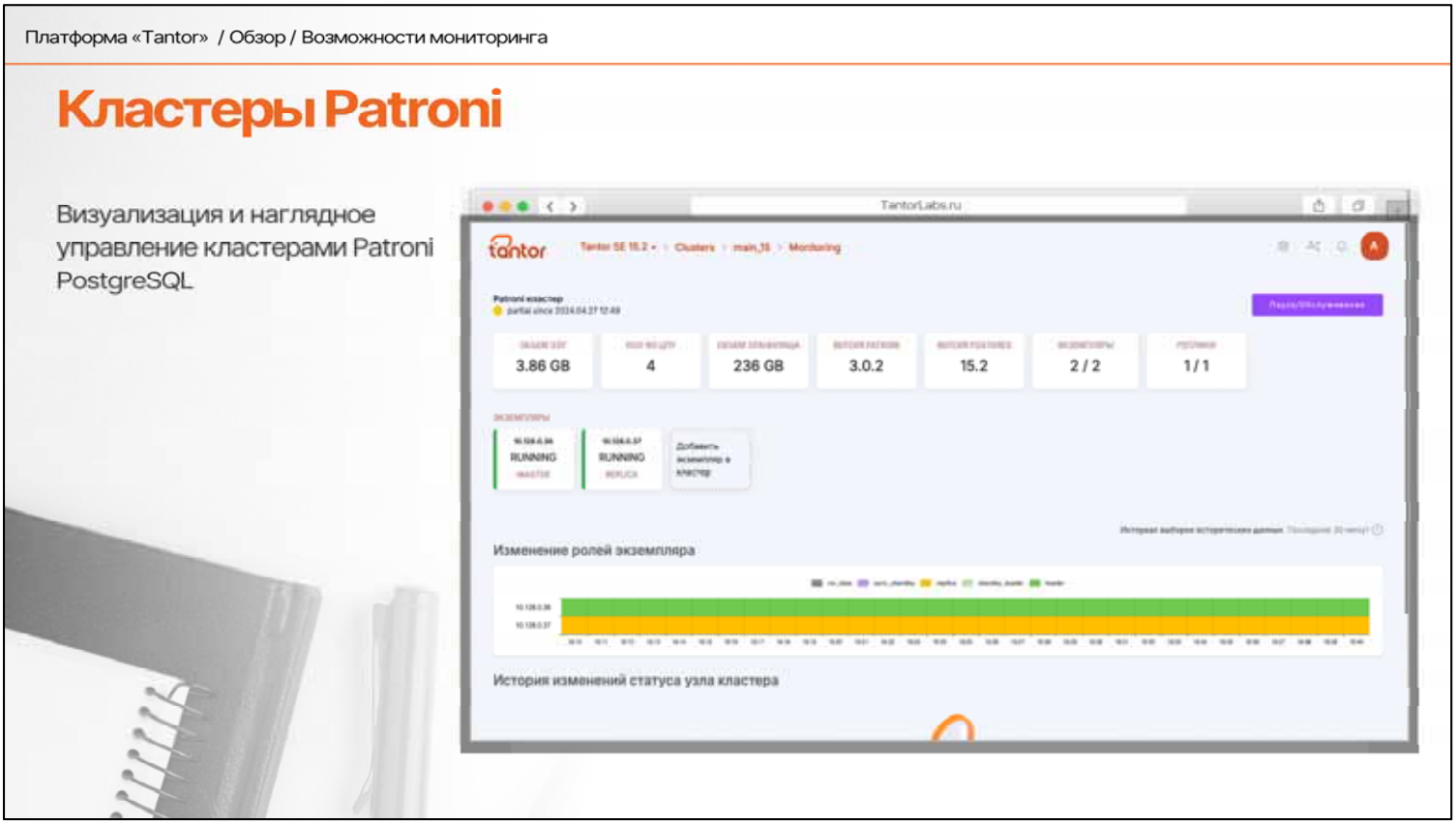


На странице обзора экземпляра в Платформе, плитки предоставляют визуальное представление ключевой информации об экземпляре базы данных. Каждая плитка дает обзор определенного аспекта экземпляра, например, загрузку CPU, использование памяти, дисковое пространство, а также активные сессии и процессы. Плитки могут быть настроены для отображения данных за различные временные интервалы и включают всплывающие подробности при наведении курсора, что позволяет быстро получить детальную информацию без необходимости перехода на другие страницы.

На странице Экземпляра открывается меню с модулями для управления и мониторинга экземпляра базы данных. Меню включает следующие модули:

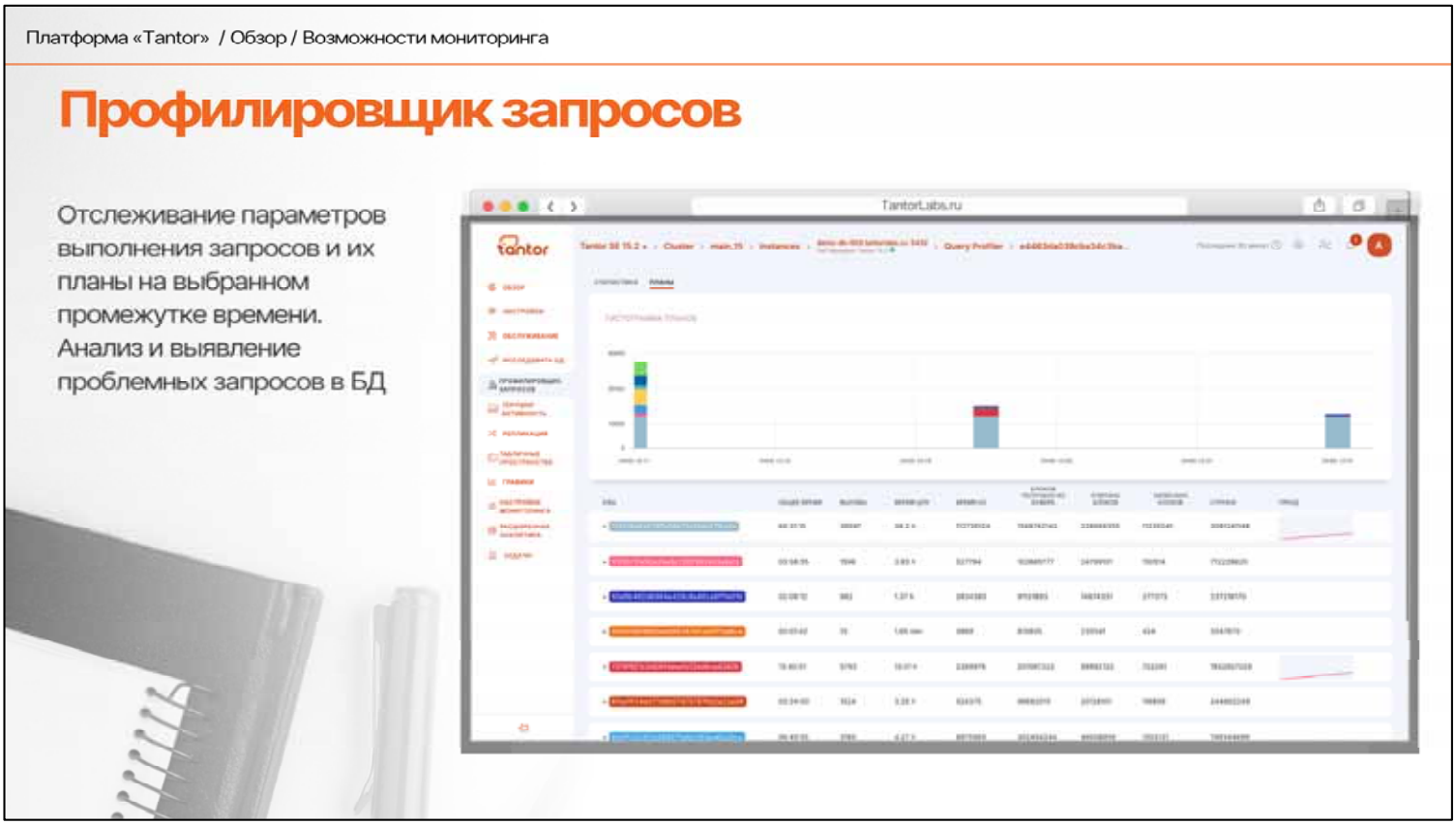
Overview: Общие сведения и показатели работы экземпляра.
Configuration: Настройки экземпляра.
Maintenance: Инструменты обслуживания.
DB Inspector: Инструменты для анализа структуры баз данных.
Query Profiler: Профилировщик для анализа запросов.
Current Activity: Отображение текущей активности.
Replication: Управление настройками репликации.
Tablespaces: Управление табличными пространствами.
Charts: Графики мониторинга.
Monitoring Config: Настройки системы мониторинга.
Advanced Analytics: Инструменты для продвинутого анализа данных.
Tasks: Планировщик для запуска отложенных задач на экземплярах СУБД.

https://docs.tantorlabs.ru/tp/4.0/instances/overview.html



Работа с кластерами Patroni на Платформе «Tantor» охватывает различные аспекты управления и мониторинга. Вкладка «Clusters» позволяет видеть все кластеры в рабочем пространстве, их статус, версию Patroni, а также ресурсы, такие как CPU и память. Можно переходить к подробной информации о каждом кластере, мониторить кластеры, настраивать их, а также управлять обслуживанием. Для каждого кластера доступны функции, такие как просмотр, мониторинг, пауза/обслуживание и настройка. Информация о кластерах включает данные о каждом экземпляре, их роли и статусы.

https://docs.tantorlabs.ru/tp/4.0/ug_clusters_pages.html

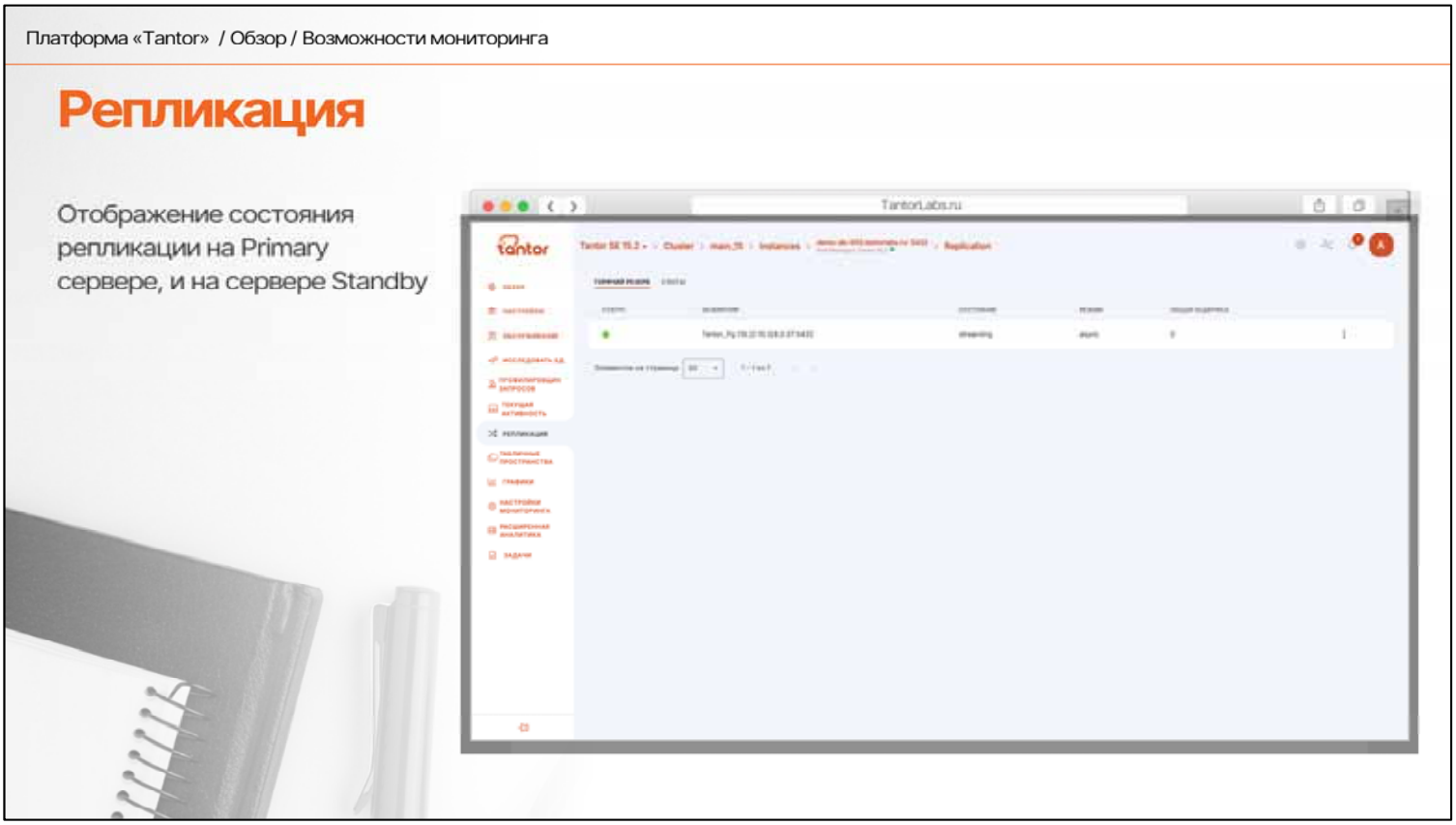


Профайлер запросов (Query Profiler) на Платформе «Tantor» использует расширение `pg_stat_statements` для сбора статистики по запросам PostgreSQL. Основная функция этого инструмента — идентификация и анализ медленных запросов. Профайлер позволяет выбирать различные временные интервалы для отображения данных, предлагает детальные метрики такие как время выполнения запроса, количество вызовов, время CPU и IO. Он также включает визуализации, такие как графики, и предоставляет возможность просмотра деталей запроса, включая текст и план выполнения.

Профайлер запросов на Платформе «Tantor» позволяет пользователям глубоко анализировать производительность SQL-запросов. Это помогает в оптимизации запросов путем предоставления данных о времени выполнения, использовании памяти, а также количестве и типах операций ввода-вывода. Используя эту информацию, разработчики и администраторы баз данных могут выявлять и устранять неэффективные запросы, что приводит к улучшению общей производительности системы.

Кроме того, Профайлер запросов обладает возможностью сравнения производительности запросов за разные периоды, что позволяет отслеживать влияние внесенных изменений в код или структуру данных на производительность базы данных. Эта функция становится неоценимым инструментом при проведении тестирования изменений и оценке их эффективности.

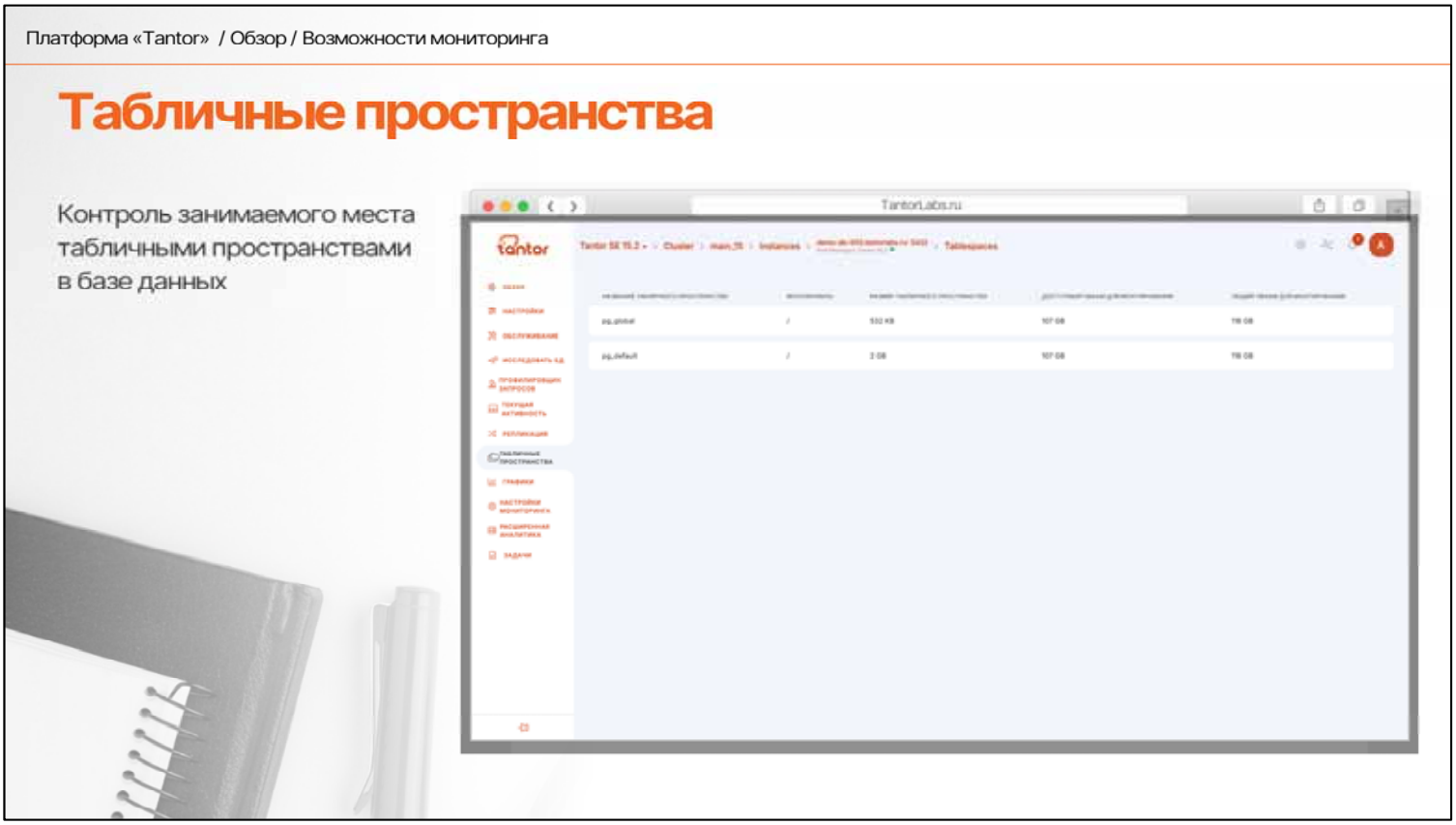
https://docs.tantorlabs.ru/tp/4.0/instances/query_profiler.html



Модуль «Репликация» на Платформе «Tantor» предоставляет подробный мониторинг репликации PostgreSQL. Он включает две основные вкладки: «STANDBY'S» для управления репликами и «SLOTS» для управления слотами репликации. На вкладке «STANDBY'S» пользователь может видеть список реплик с их статусами и доступом к детальной информации о каждой реплике. Вкладка «SLOTS» отображает список слотов репликации с их активностью и статусом. Эти инструменты помогают оптимизировать процесс репликации и обеспечивают надежную синхронизацию данных.

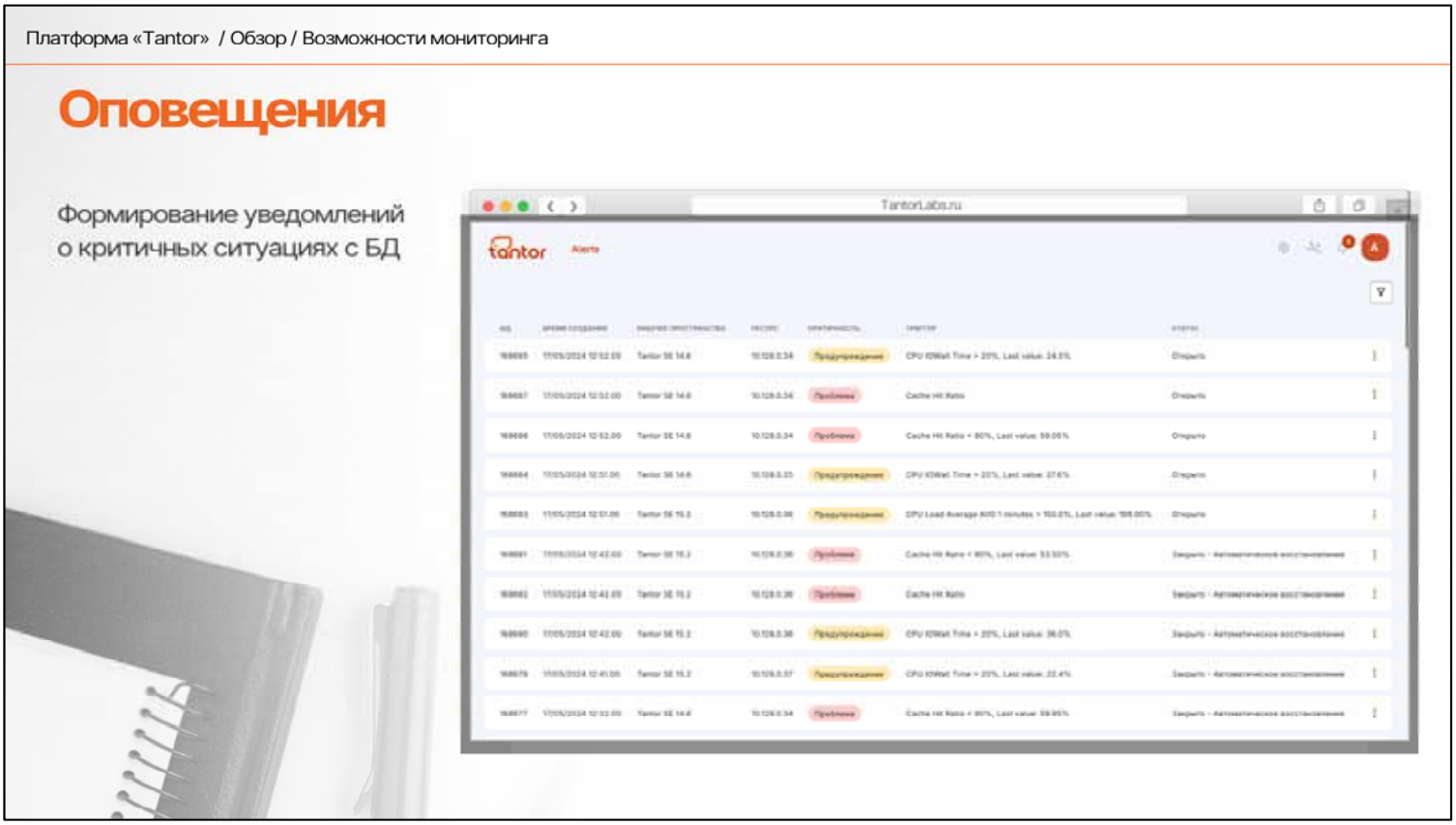
Дополнительно, модуль «Репликация» на Платформе позволяет мониторить параметры репликации в реальном времени, что включает отслеживание задержек репликации и статусов синхронизации. Это ключевой элемент для поддержания целостности данных и минимизации рисков потери данных в случае сбоя основного сервера. Эффективное управление репликацией способствует обеспечению высокой доступности и надежности баз данных.

https://docs.tantorlabs.ru/tp/4.0/instances/replication.html



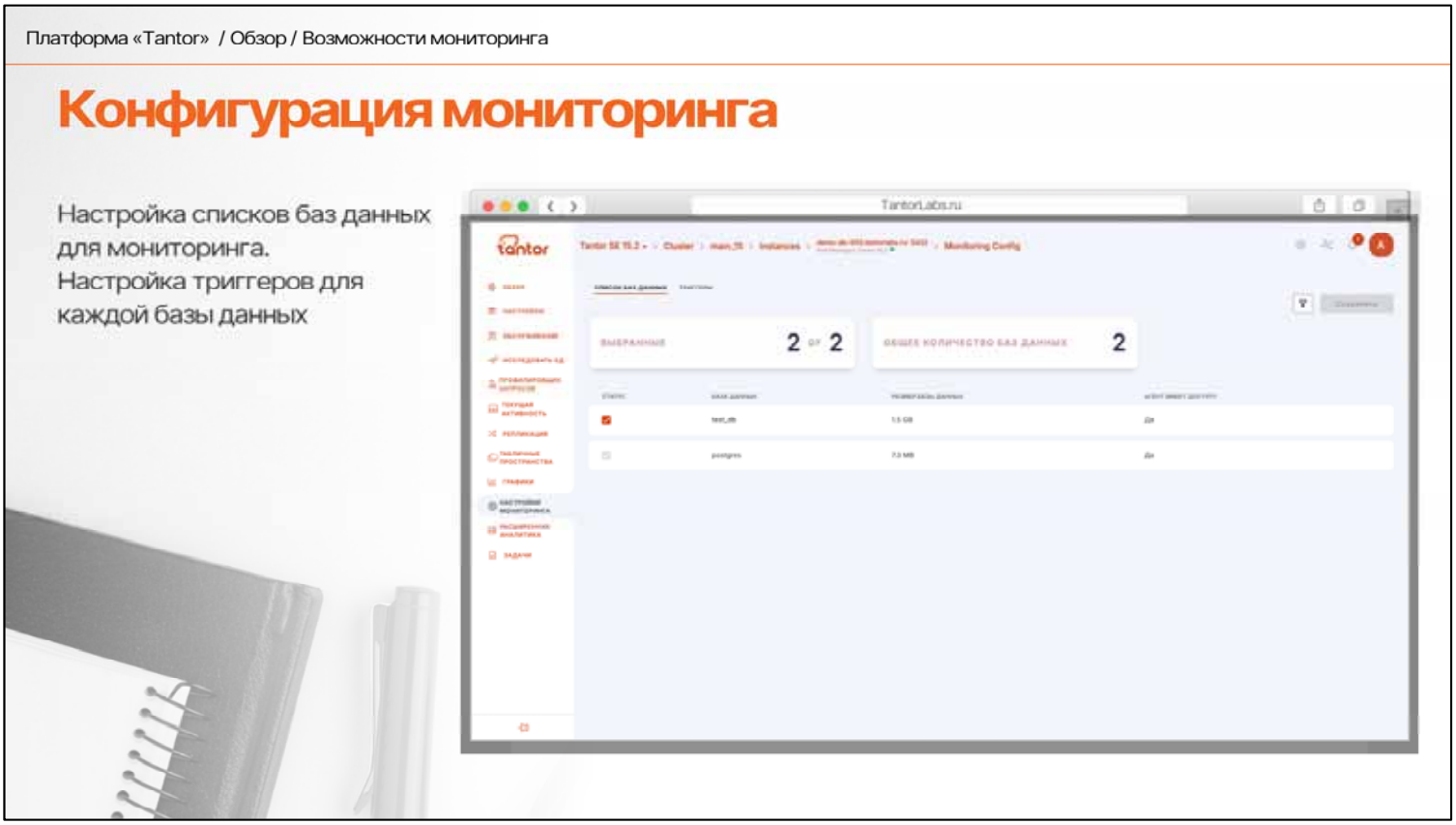
Модуль «Табличные пространства» создан для эффективного контроля за использованием места табличными пространствами в базе данных. Он осуществляет сопоставление размера диска с размером каждого табличного пространства, что обеспечивает более детальное представление об объеме занимаемого пространства каждой части базы данных.

https://docs.tantorlabs.ru/tp/4.0/instances/tablespaces.html

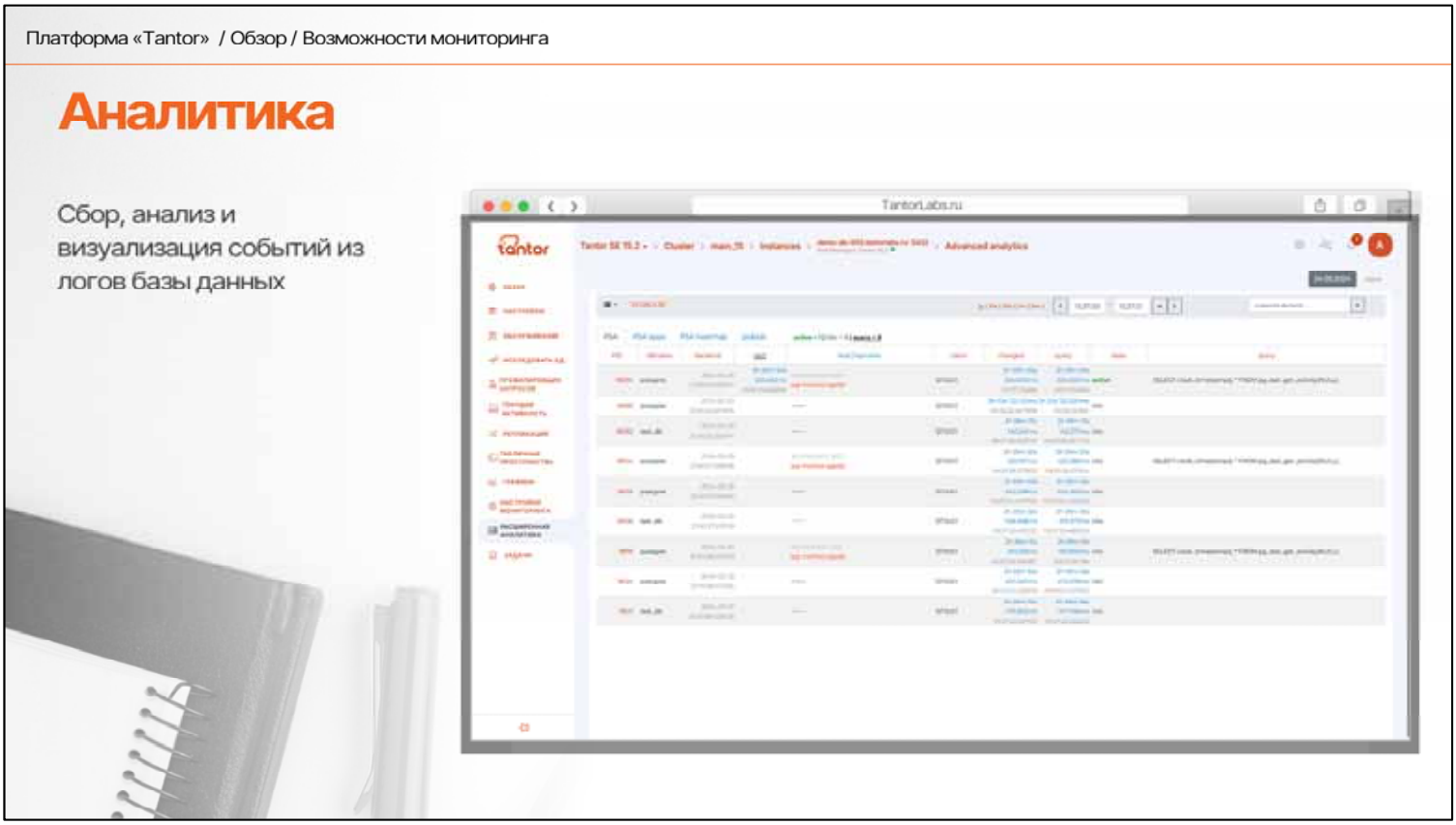


Модуль «Оповещения» разработан для создания уведомлений о критических ситуациях в базе данных. Он обеспечивает возможность мониторинга изменений статуса оповещений, позволяя оперативно реагировать на важные события в базе данных. Этот инструмент обеспечивает эффективное уведомление о возможных проблемах, что позволяет быстро и точно реагировать на изменения в работе базы данных.

https://docs.tantorlabs.ru/tp/4.0/ug_alerts.html



На Платформе «Tantor» модуль конфигурации мониторинга позволяет настроить слежение за базами данных, управлять триггерами оповещений и настраивать условия для оповещений на основе собранных метрик PostgreSQL. Это включает выбор баз данных для мониторинга, установку триггеров для предупреждений, проблем и восстановления, а также сохранение изменений для активации настроенных параметров мониторинга. Инструмент предоставляет возможность детально настроить параметры мониторинга для эффективного управления производительностью и безопасностью баз данных.
https://docs.tantorlabs.ru/tp/4.0/instances/monitoring_config.html



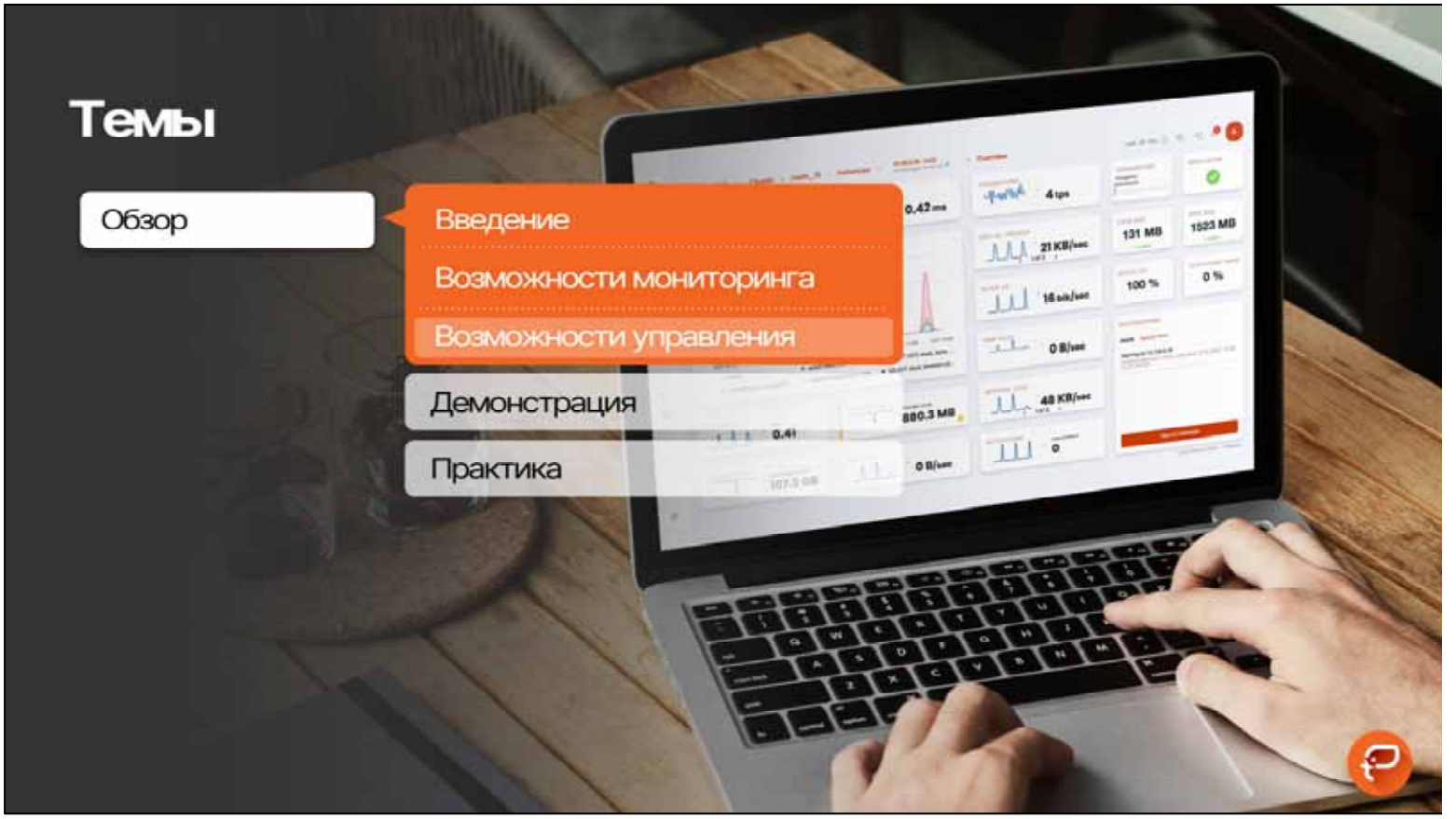
Расширенная аналитика на Платформе «Tantor» позволяет детально анализировать работу серверов баз данных. Она охватывает анализ медленных запросов, блокировок, ошибок, системных действий и логов. Включает разделы для мегазапросов и диагностики проблем с запросами, позволяя пользователям отслеживать производительность и находить узкие места. Интерфейс предлагает графики и сводки по хостам, обеспечивая удобство мониторинга и аналитики в режиме реального времени.

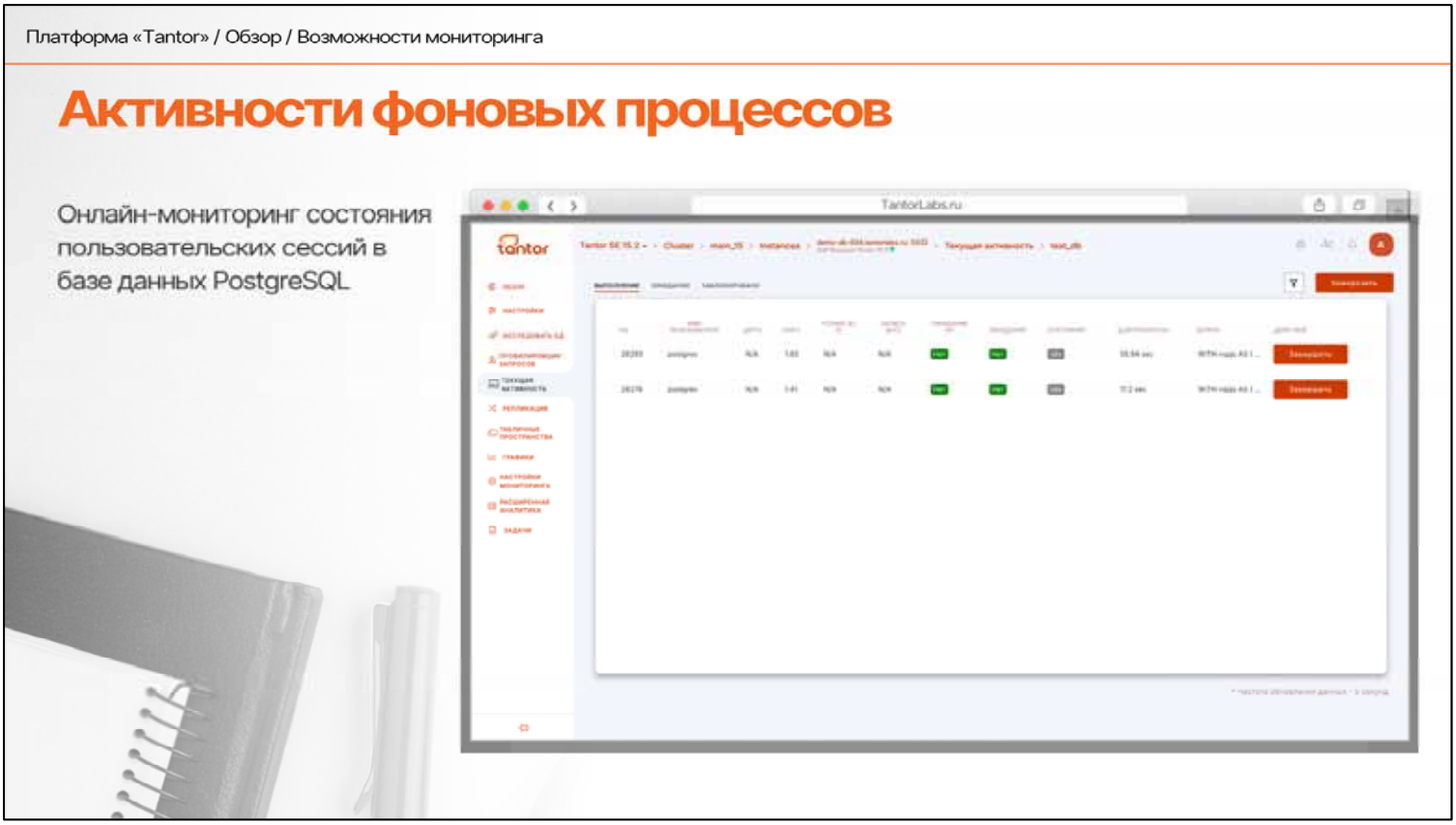
https://docs.tantorlabs.ru/tp/4.0/instances/pg_monitor/pg_monitor.html

Темы

Обзор

- Введение
- Возможности мониторинга
- Возможности управления
- Демонстрация
- Практика

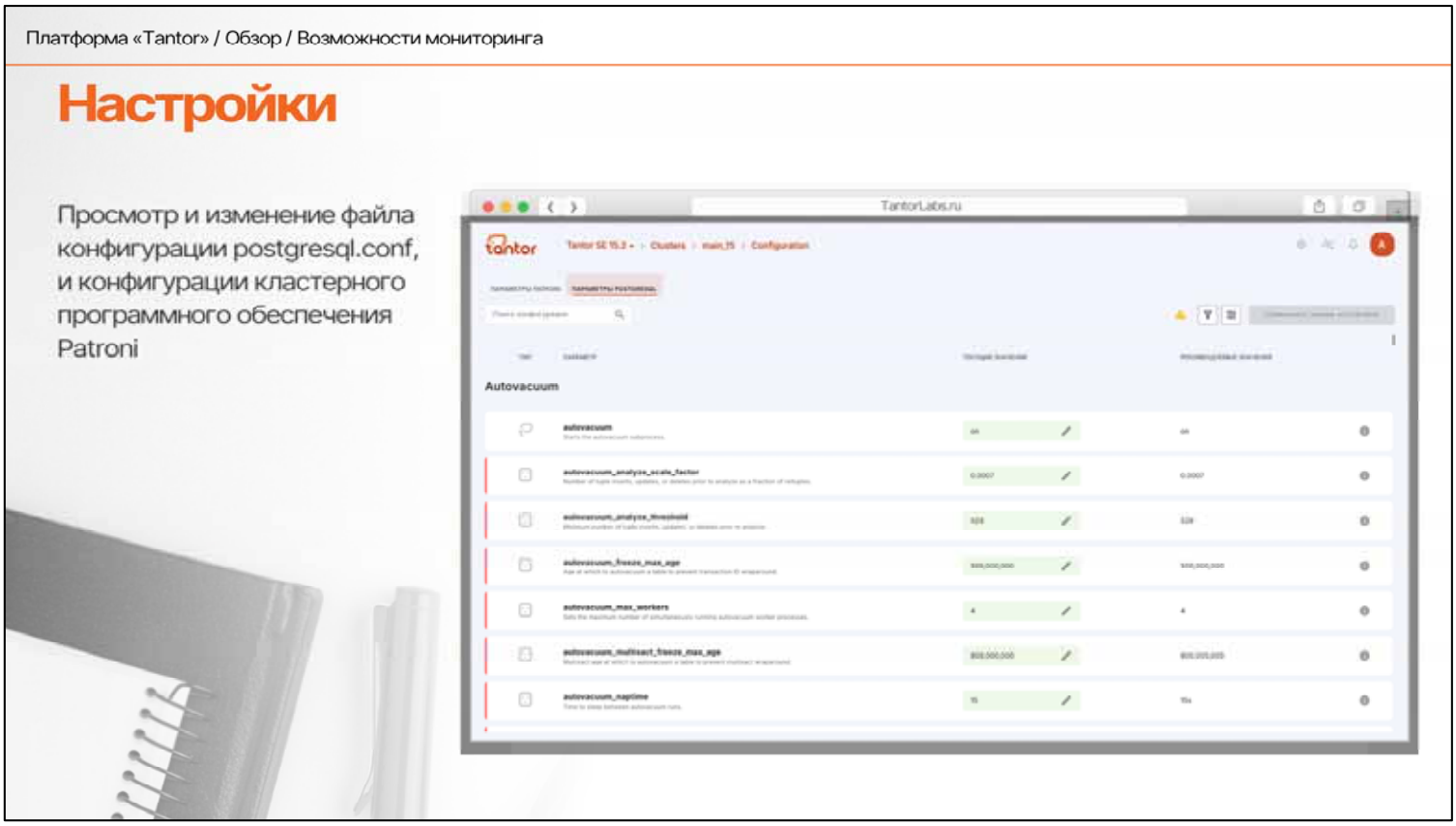




Модуль «Текущая активность» на Платформе «Tantor» отображает детальную информацию о пользовательских и системных процессах баз данных в реальном времени. Это включает мониторинг активных, ожидающих и блокирующих сессий. Для каждого процесса показываются такие параметры, как использование CPU, памяти, скорость чтения/записи, и состояние процесса. Функция «TERMINATE» позволяет завершить процессы напрямую через интерфейс. Все данные обновляются каждые пять секунд и могут быть «заморожены» для удобства анализа.

Модуль «Текущая активность» также предлагает удобные фильтры для сортировки и поиска процессов по различным параметрам, таким как имя пользователя, база данных или состояние сессии. Это делает его мощным инструментом для быстрого выявления и решения проблем в базах данных, особенно в высоконагруженных средах, где немедленное вмешательство может предотвратить длительные простои или сбои системы.

https://docs.tantorlabs.ru/tp/4.0/instances/activity.html



Модуль «Настройки» на Платформе «Tantor» предоставляет удобный интерфейс для просмотра и изменения файла конфигурации postgresql.conf, а также конфигурацию кластерного программного обеспечения Patroni. Он автоматически рекомендует оптимальные значения для различных параметров, позволяет применить эти значения и требует перезагрузку или перезапуск экземпляра для вступления изменений в силу. Интерфейс также предоставляет цветовую кодировку для легкой идентификации статуса параметров: по умолчанию, измененные и требующие перезагрузки или перезапуска. Пользователи могут фильтровать параметры по статусу и категории для удобства управления.

В дополнение, модуль «Настройки» обеспечивает возможность сохранения и восстановления предыдущих конфигураций, что позволяет легко откатить изменения в случае необходимости. Эта функция является особенно ценной при тестировании новых настроек в производственной среде, гарантируя, что можно безопасно экспериментировать, минимизируя риски сбоев.
https://docs.tantorlabs.ru/tp/4.0/instances/instance_configuration.html

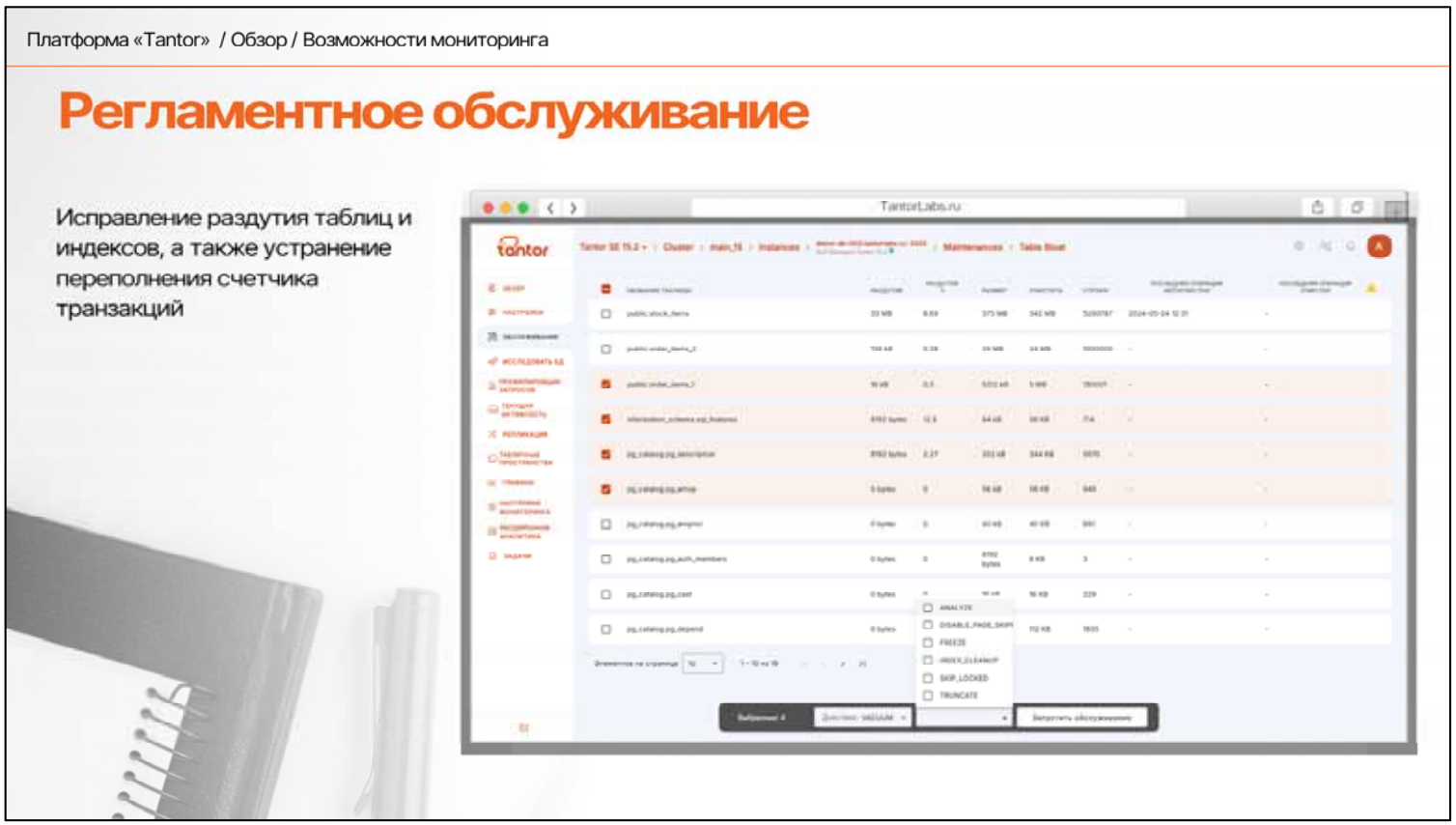
Анализ схемы данных

Анализ схемы данных базы PostgreSQL

Модуль «Инспектор Баз Данных» на Платформе «Tantor» позволяет проводить анализ схемы данных базы PostgreSQL. Используя HEALTHCHECKS, можно определить потенциальные проблемы, такие как неиспользуемые или избыточные индексы, таблицы с большим размером или неоптимальными настройками. Пользователю доступны детали каждой проблемы, с возможностью применения рекомендуемых изменений прямо через интерфейс. Этот инструмент помогает улучшить производительность базы данных и обеспечить её корректную конфигурацию.

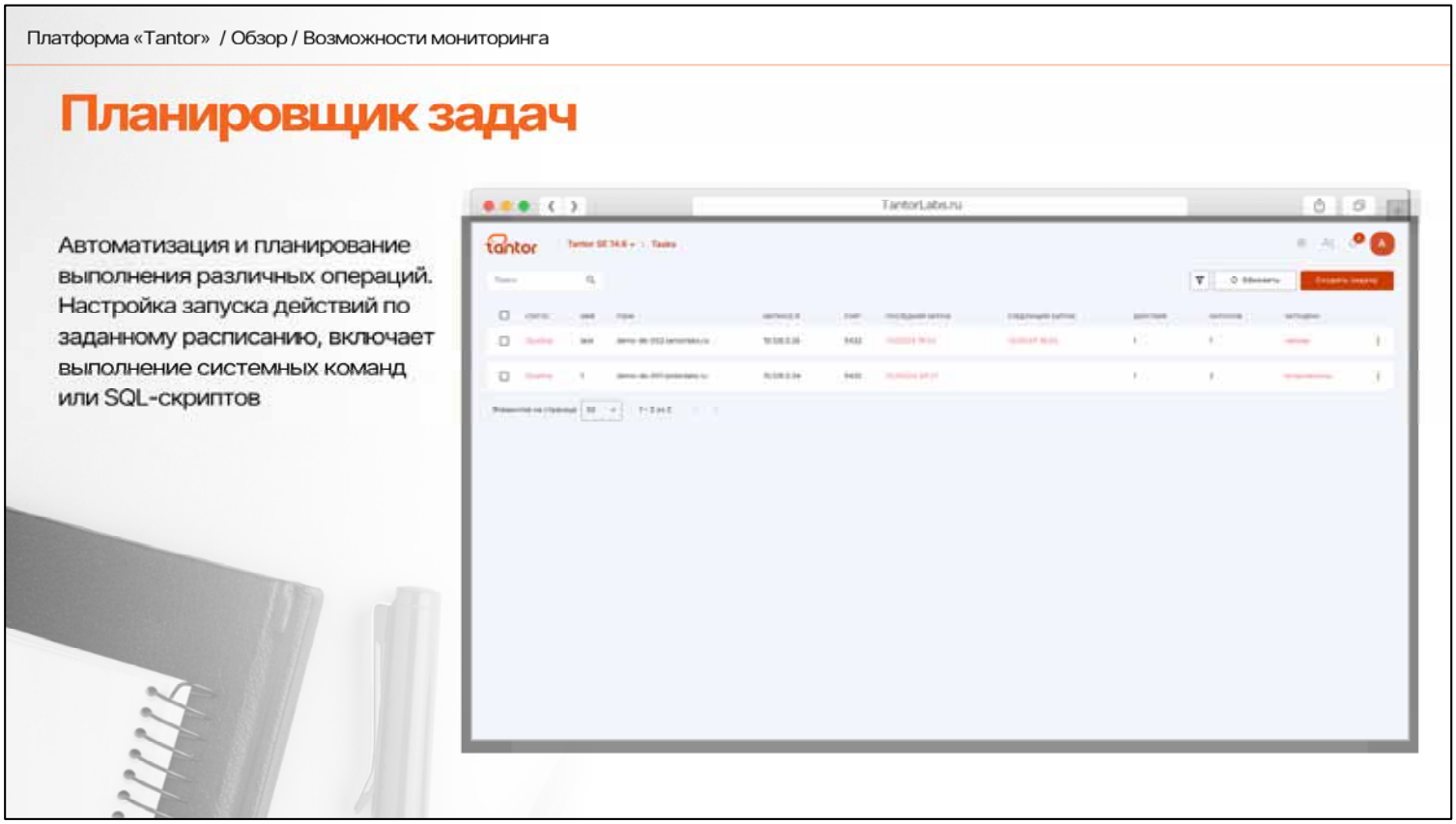
В дополнение к основным функциям, «Инспектор Баз Данных» на Платформе позволяет администраторам просматривать детальные характеристики каждой таблицы и индекса, включая информацию о количестве строк, занимаемом пространстве и активности чтения/записи. Это предоставляет полезные данные для оптимизации и реорганизации данных, повышая общую производительность базы данных.

https://docs.tantorlabs.ru/tp/4.0/instances/DB_inspector.html



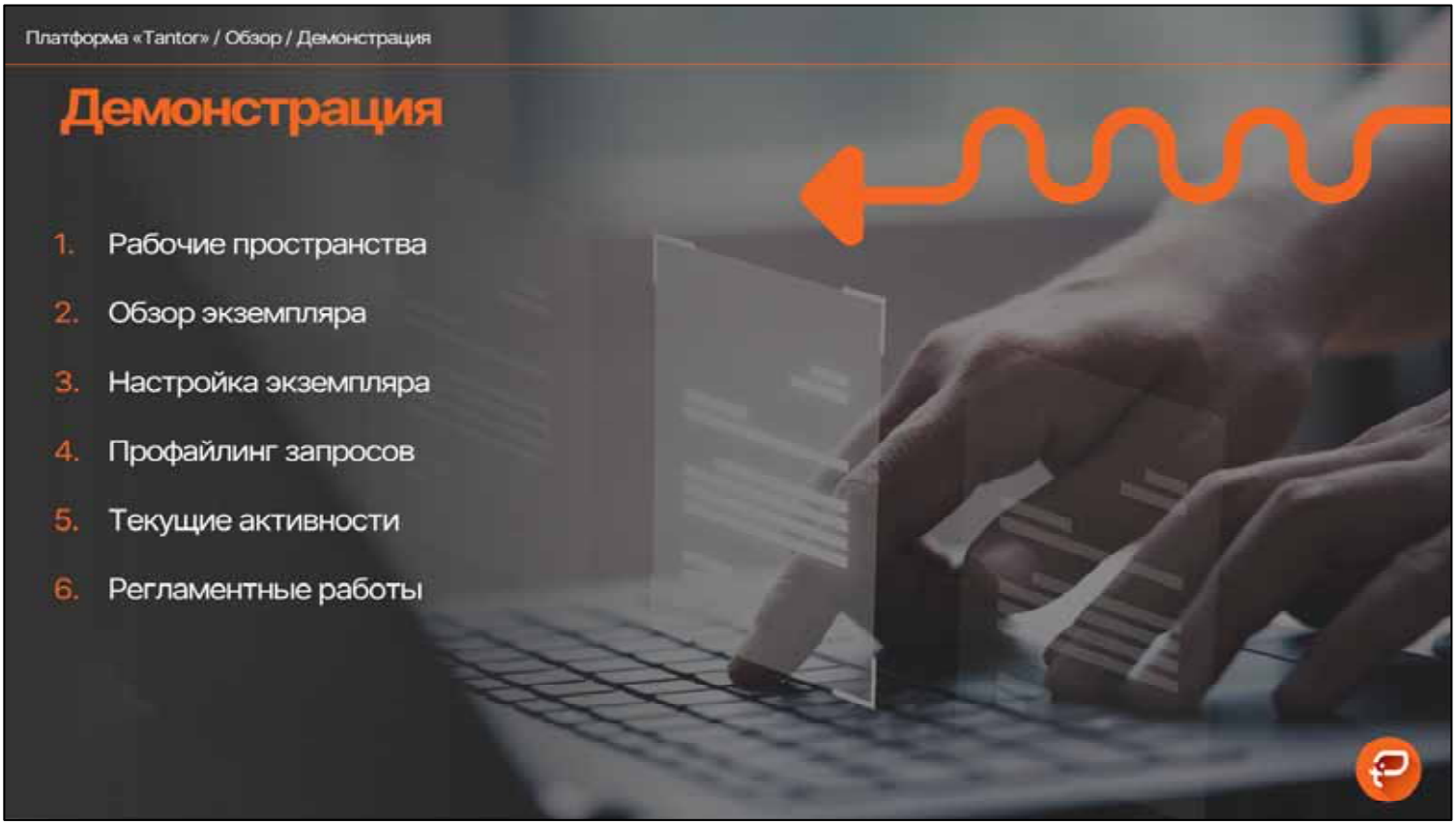
В модуле «Обслуживание» Платформы «Tantor» можно управлять задачами по поддержанию баз данных PostgreSQL, такими как исправление раздутия таблиц и индексов, а также устранение переполнения счетчика транзакций. Пользователи могут выбирать конкретные действия для исправления проблем и запускать команды VACUUM, REINDEX, или ANALYZE для оптимизации производительности базы данных. История обслуживания предоставляет доступ к подробностям и результатам прошлых операций.

На странице «Обслуживание» можно также настроить автоматические регламентные задачи для повышения эффективности операций обслуживания. Это включает возможность планирования задач на регулярной основе, что помогает поддерживать базу данных в оптимальном состоянии без необходимости постоянного вмешательства администратора.
https://docs.tantorlabs.ru/tp/4.0/instances/maintenance.html



Модуль «Задачи» на Платформе позволяет пользователям автоматизировать и планировать выполнение различных операций. С его помощью можно настроить запуск действий по заданному расписанию, что включает выполнение системных команд или SQL-скриптов. Эта функциональность идеально подходит для автоматизации рутинных задач управления базами данных, таких как обновления, резервное копирование, или процедуры очистки, обеспечивая повышение эффективности и снижение вероятности ошибок за счет автоматизации процессов.

https://docs.tantorlabs.ru/tp/4.0/admin_scheduler.html



Часть 1. Рабочие пространства

1) Войти в Платформу по локальной ссылке https://education.tantorlabs.ru/platform/login
2) Ввести учетные данные: student@student.ru пароль Student123!
3) Открыть рабочее пространство «Tantor»
4) Открыть экземпляр «demo».
5) На странице «Обзор» продемонстрировать индикаторы.

Часть 2. Обзор экземпляра

Показать выпадающее окно «Сессии», «Нагрузка ЦПУ», «Доступно ОЗУ», «Сеть», «Блок из буфера».

Часть 3. Настройка экземпляра

1) Открыть страницу «Настройки» → «Страницы настройки кластера» → «ПАРАМЕТРЫ POSTGRESQL».
2) Изменить параметр `autovacuum_analyze_scale_factor`.
3) Нажать на кнопку «Применить новые настройки».
4) Показать возможные параметры фильтра.

Часть 4. Профайлинг запросов

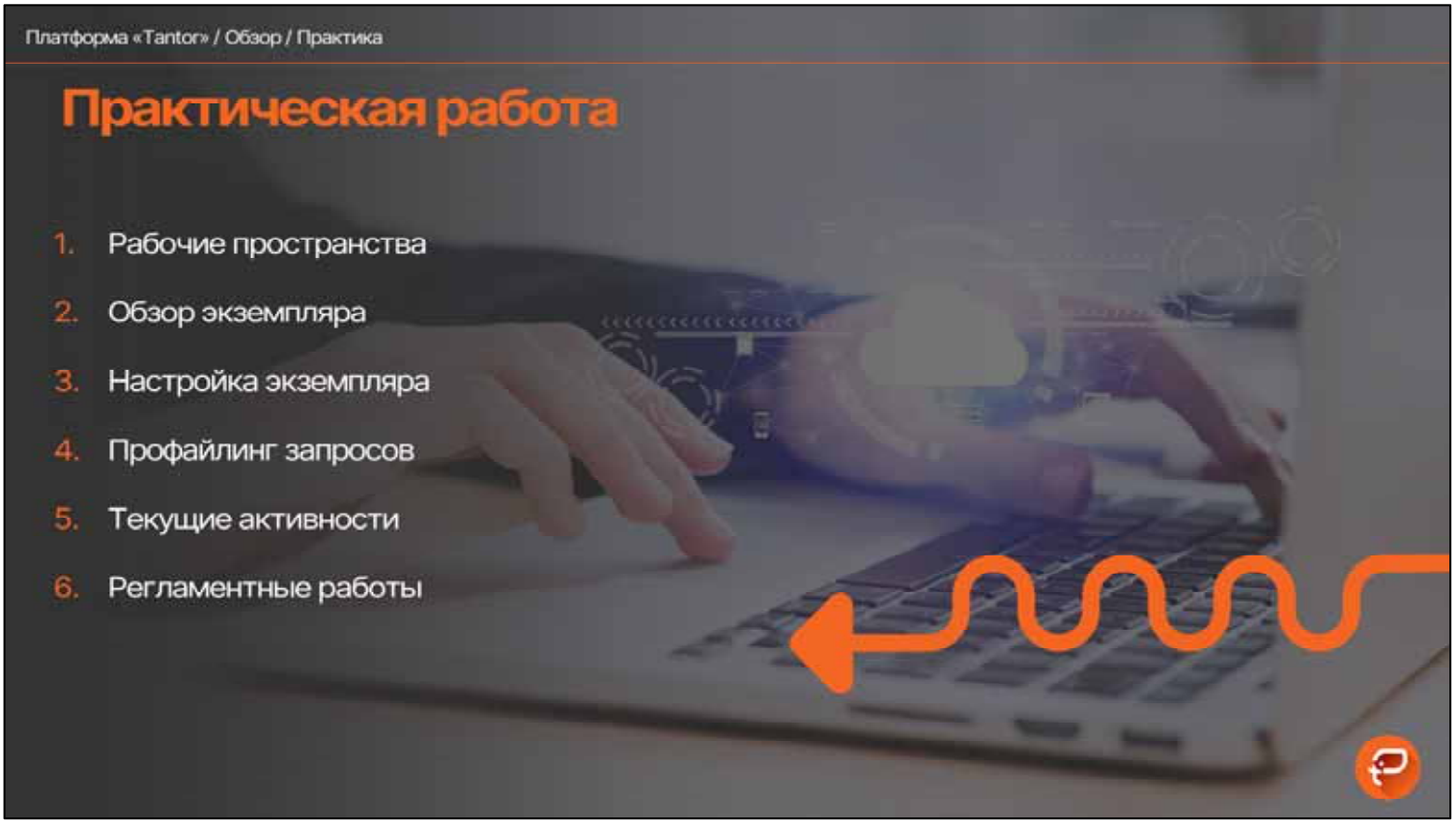
1) Открыть страницу «Профилировщик запросов».
2) Выбрать запрос с самым большим значением «Записано временных блоков».
3) Нажать на поле «Хэш запроса» → посмотреть основную статистику запроса.
4) Перейти на закладку «Планы».
5) Выбрать план → нажать на любое поле правее «Хэш запроса».
6) Продемонстрировать графический план запроса.

Часть 5. Текущие активности

1) Открыть страницу «Текущая активность»
2) Выбрать БД postgres.
3) Продемонстрировать текущие подключение на вкладке «Выполнение»

Шаг 6. Регламентные работы

1) Открыть страницу «Обслуживание».
2) Выбрать БД test_db.
3) Выбрать «Раздутие индексов».
4) Отсортировать по убыванию колонку «КОЭФФ. РАЗДУТИЯ %»
5) Выбрать самую первую таблицу, Действие «Reindex».
6) Нажать кнопку «Запустить обслуживание».
7) Нажать на кнопку «Запустить обслуживание».
8) Нажать на ссылку «История».
9) Посмотреть результаты запуска.



Часть 1. Рабочие пространства

1) Войдите в Платформу по локальной ссылке https://education.tantorlabs.ru/platform/login
2) Введите учетные данные: student@student.ru пароль Student123!
3) Откройте рабочее пространство «Tantor»
4) Откройте экземпляр «demo».
5) На странице «Обзор» изучите индикаторы.

Часть 2. Обзор экземпляра

Изучите выпадающее окно «Сессии», «Нагрузка ЦПУ», «Доступно ОЗУ», «Сеть», «Блок из буфера».

Часть 3. Настройка экземпляра

1) Откройте страницу «Настройки» → «Страницы настройки кластера» → «ПАРАМЕТРЫ POSTGRESQL».
2) Измените параметр `autovacuum_analyze_scale_factor`.
3) Нажмите на кнопку «Применить новые настройки».
4) Изучите возможные параметры фильтра.

Часть 4. Профайлинг запросов

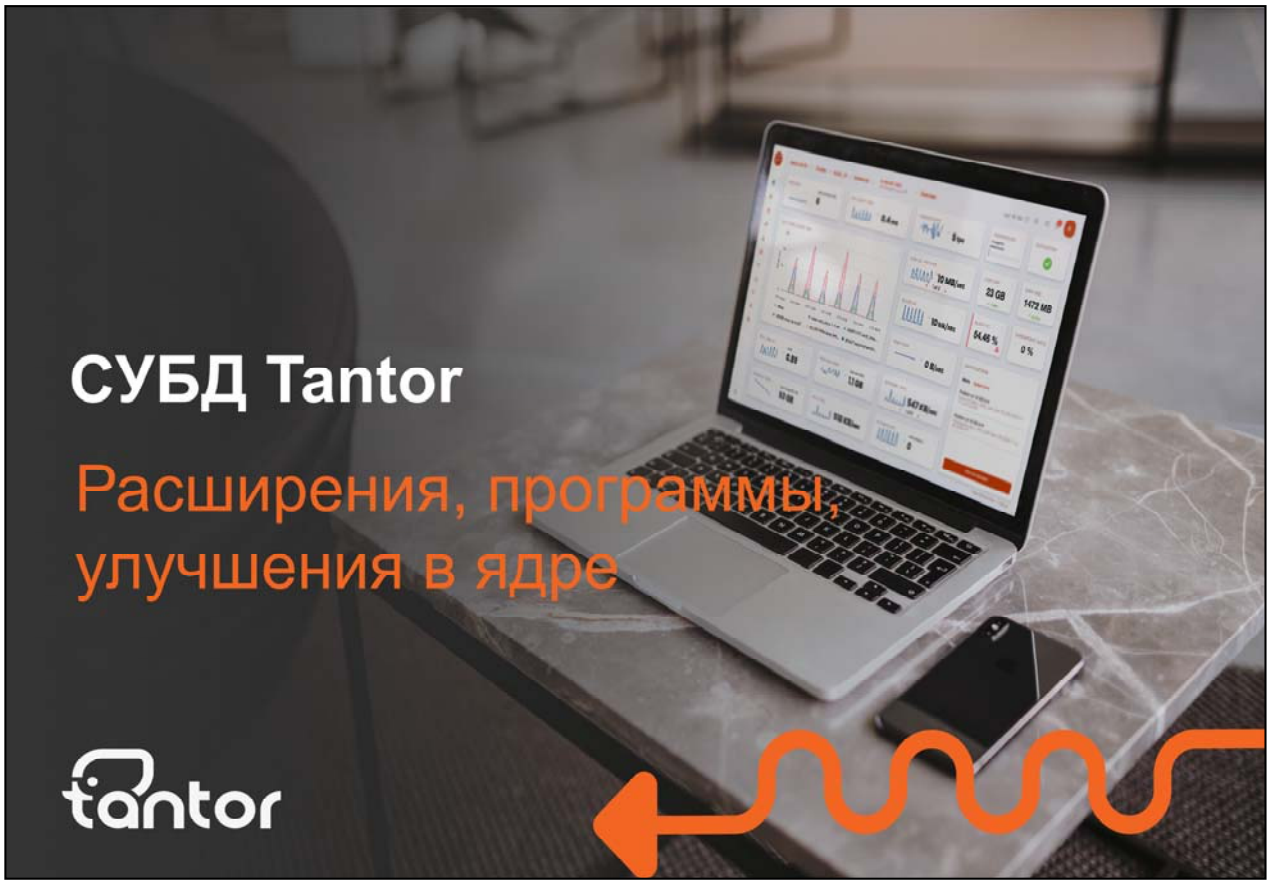
1) Откройте страницу «Профилировщик запросов».
2) Выберете запрос с самым большим значением «Записано временных блоков».
3) Нажмите на поле «Хэш запроса» → изучите основную статистику запроса.
4) Перейдите на закладку «Планы».
5) Выберете план → нажмите на любое поле правее «Хэш запроса».
6) Изучите графический план запроса.

Часть 5. Текущие активности

1) Откройте страницу «Текущая активность».
2) Выберете БД postgres.
3) Изучите текущие подключения на вкладке «Выполнение».

Шаг 6. Регламентные работы

1) Откройте страницу «Обслуживание».
2) Выберете БД test_db.
3) Выберете «Раздутие индексов».
4) Отсортируйте по убыванию колонку «КОЭФФ. РАЗДУТИЯ %»
5) Выберете самую первую таблицу, Действие «Reindex».
6) Нажмите кнопку «Запустить обслуживание».
7) Нажмите на кнопку «Запустить обслуживание».
8) Нажмите на ссылку «История».
9) Изучите результаты запуска.



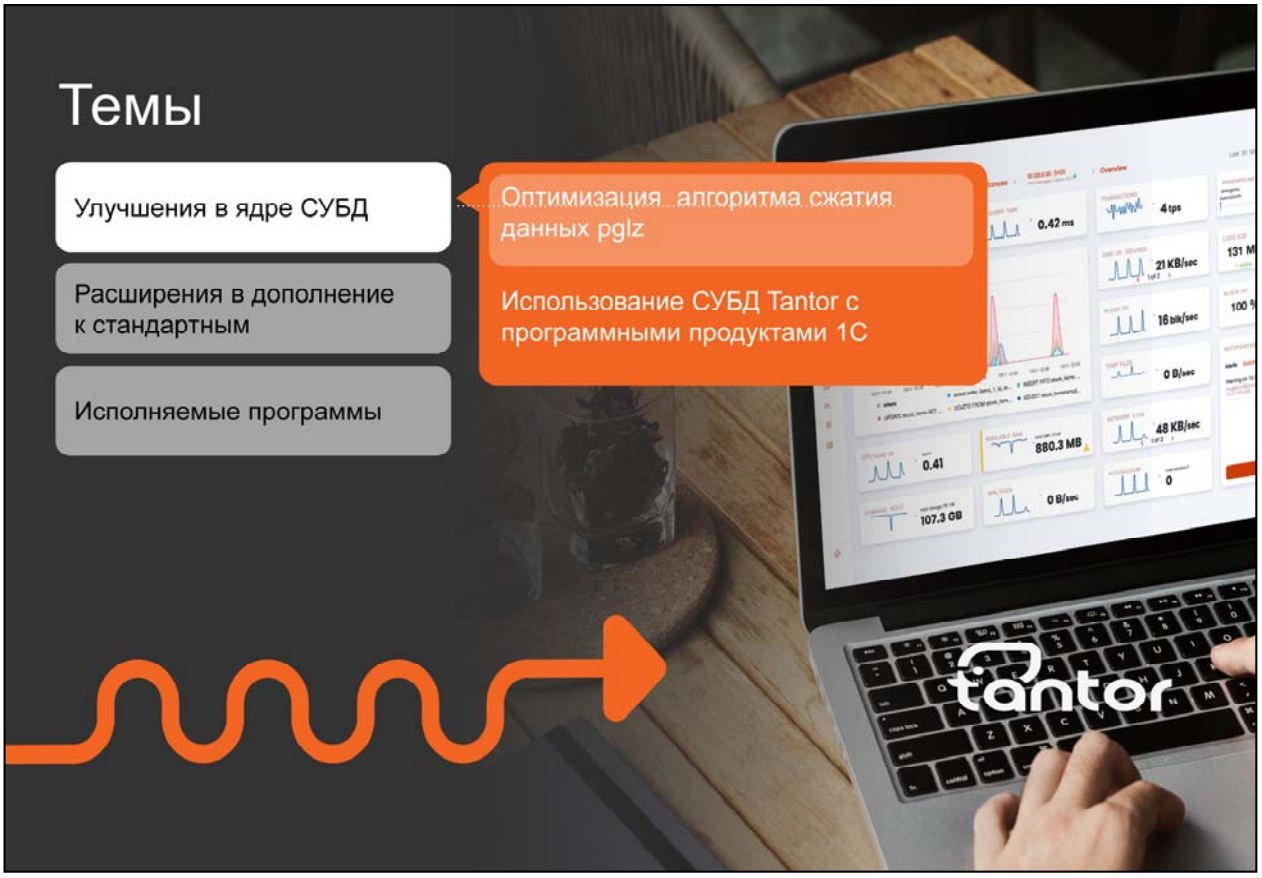
В этой главе рассматриваются улучшения, которые имеются в СУБД Tantor относительно PostgreSQL:

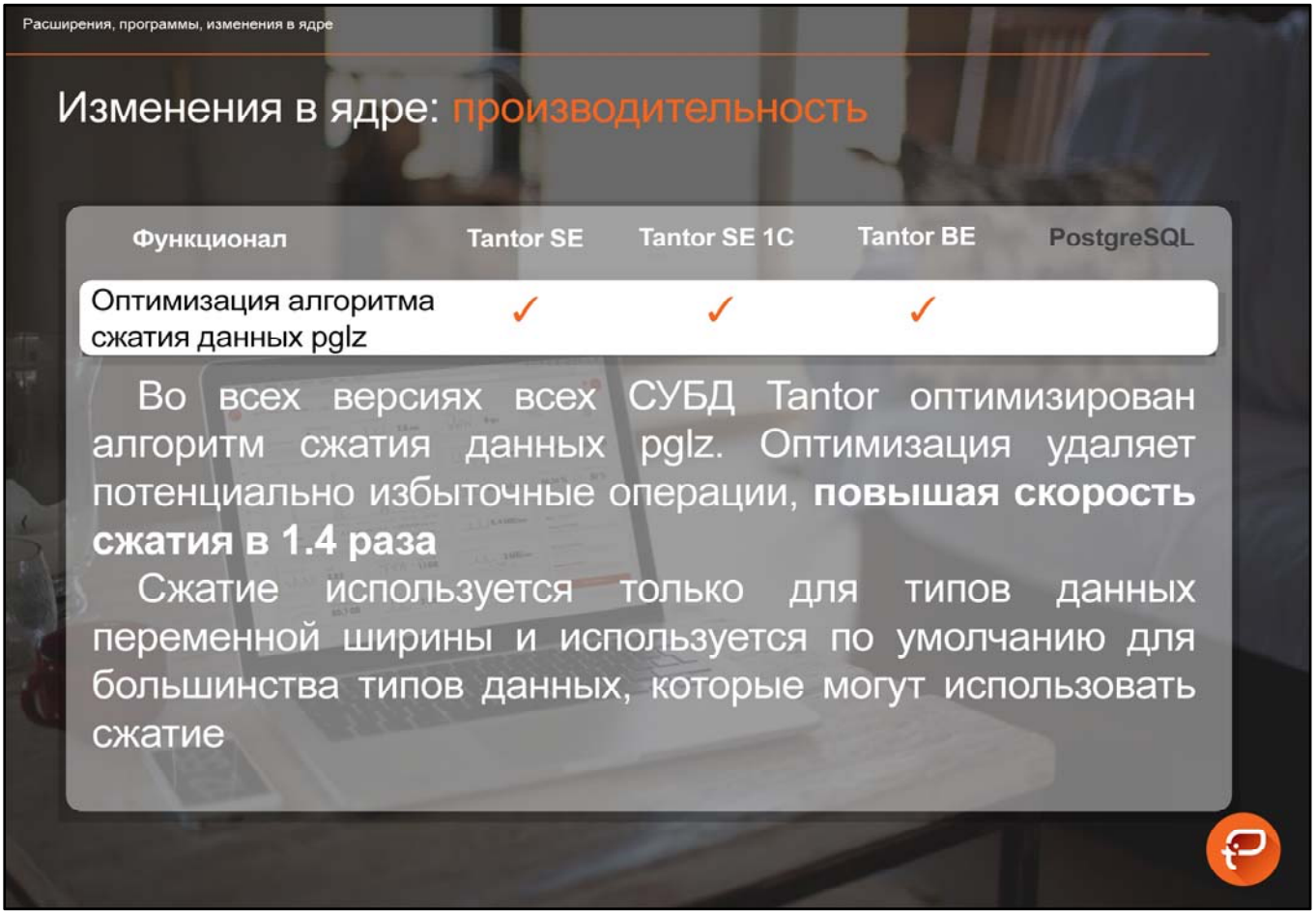
1) Изменения в ядре
2) Расширения, поставляемые с СУБД Tantor
3) Приложения (компьютерные программы, утилиты)

Практические задания к этой главе опциональны и выполняются, если остаётся время.

Темы

- Улучшения в ядре СУБД
  - Оптимизация алгоритма сжатия данных pglz
- Расширения в дополнение к стандартным
  - Использование СУБД Tantor с программными продуктами 1С
- Исполняемые программы





Во всех версиях всех СУБД Tantor оптимизирован алгоритм сжатия данных pglz. Оптимизация удаляет потенциально избыточные операции, **повышая скорость сжатия в 1.4 раза**.

Алгоритм сжатия `pglz` применяется по умолчанию.

```
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, integer фиксированной длинны и не сжимается, text переменной 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};
```

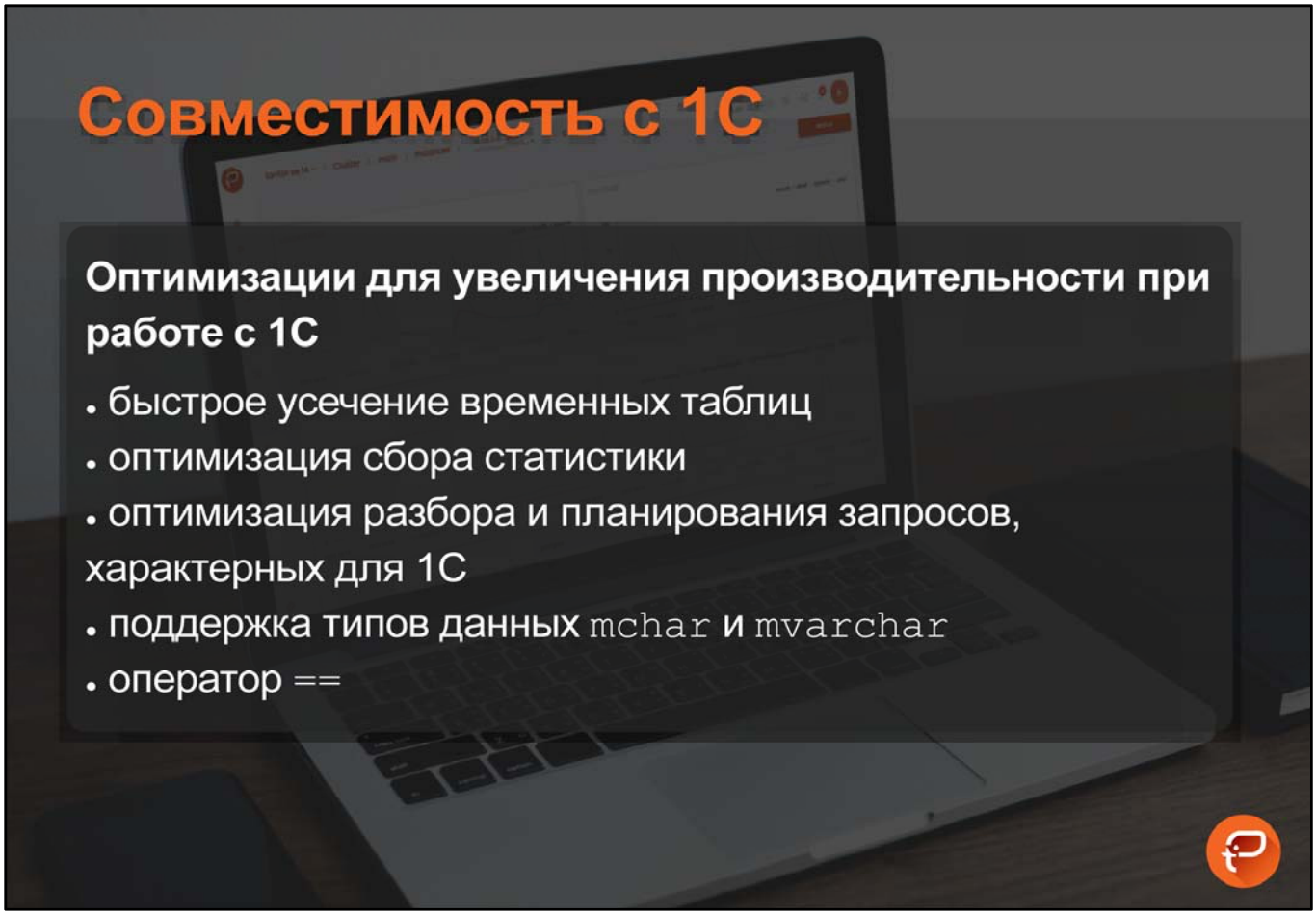
Технические детали оптимизаций кода алгоритма pglz в СУБД Tantor:

1) Используется более компактная хэш-таблица с индексами типа `uint16` вместо указателей

2) Игнорируется `prev`-указатель в хэш-таблице

3) Используются более эффективные 4-байные операции сравнения вместо 1-байтных.

Также макрофункции заменены обычными функциями (не влияет на производительность).



Для использования с программными продуктами 1С выпускается сборка СУБД Tantor SE 1С, которая включает расширения:

1) fasttrun: создаёт функцию `fasttruncate('имя_таблицы')` для использования вместо команды `TRUNCATE`. При использовании этой функции не вносятся изменения (не порождается версия строки) в таблицы системного каталога (`pg_class`).

2) fulleq: создаёт оператор ==, который возвращает true, когда операнды равны или оба имеют значение NULL

3) mchar: поддержка типов данных mchar и mvarchar имеющихся в Microsoft SQL Server и использующихся в 1С. Поддержка включает в себы набор операторов, функций и использования библиотек ICU для независимого от платформы сравнения и сортировки значений.

4) online_analyze: выполняет сбор статистики для оптимизатора (эквивалент команды `ANALYZE`) после выполнения команд `INSERT, UPDATE, DELETE, SELECT INTO` в том числе по временным таблицам (последнее существенно для 1С). Имеет набор параметров которыми можно настраивать когда будет вызываться сбор статистики.

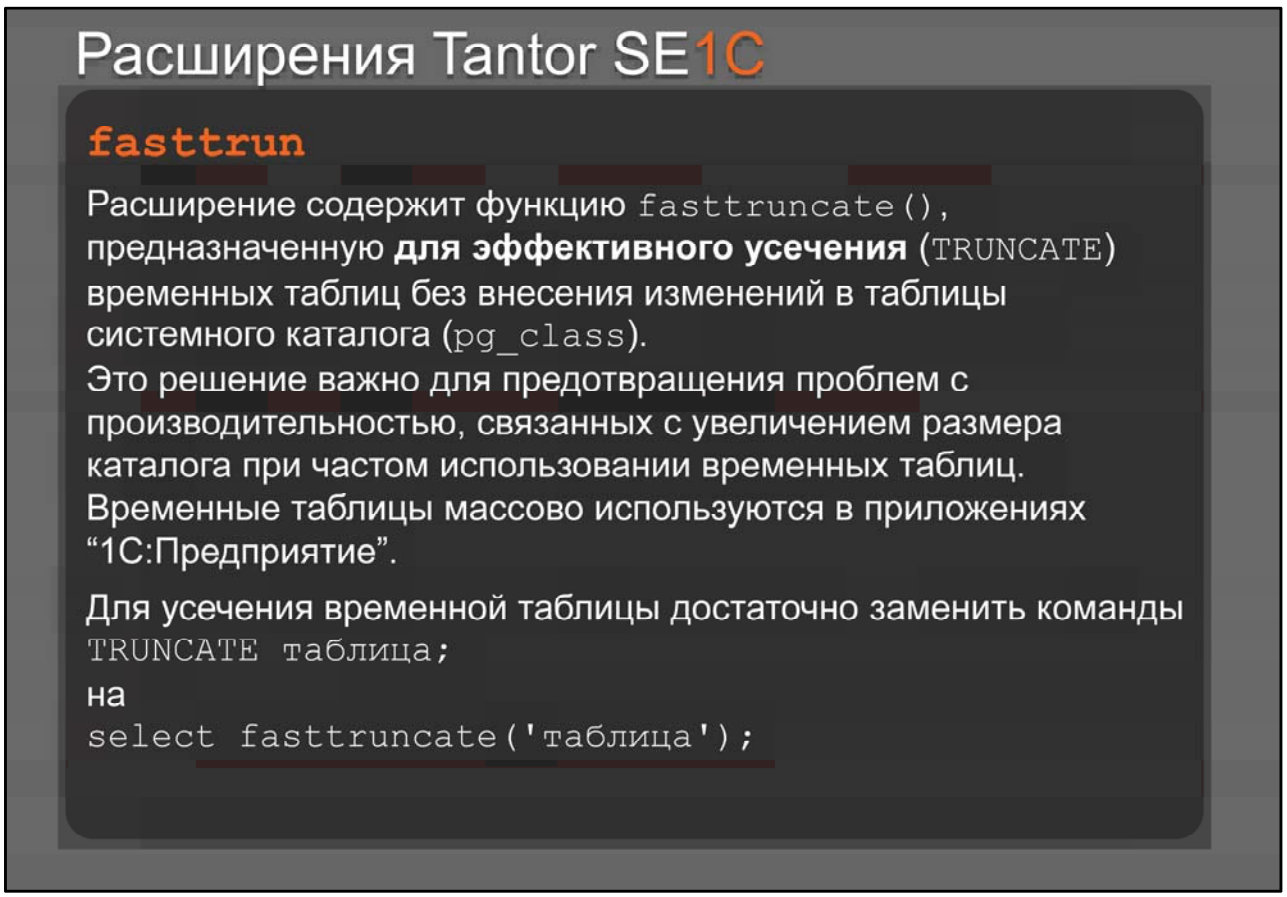
Также имеются улучшения:

1) Переписывание запроса для удаления избыточных соединений, если в команде имеется соединение таблицы с самой собой. Соединение таблиц с самой собой может встречаться при автоматической генерации запросов, либо в результате изменения логики хранения данных.

2) Оптимизация разбора запросов с операторами `LIKE` и поддержка этого оператора для типов данных, использующихся в 1С.

3) Изменения в логике планировщика для создания оптимальныхх планов запросов, типичных для 1С.

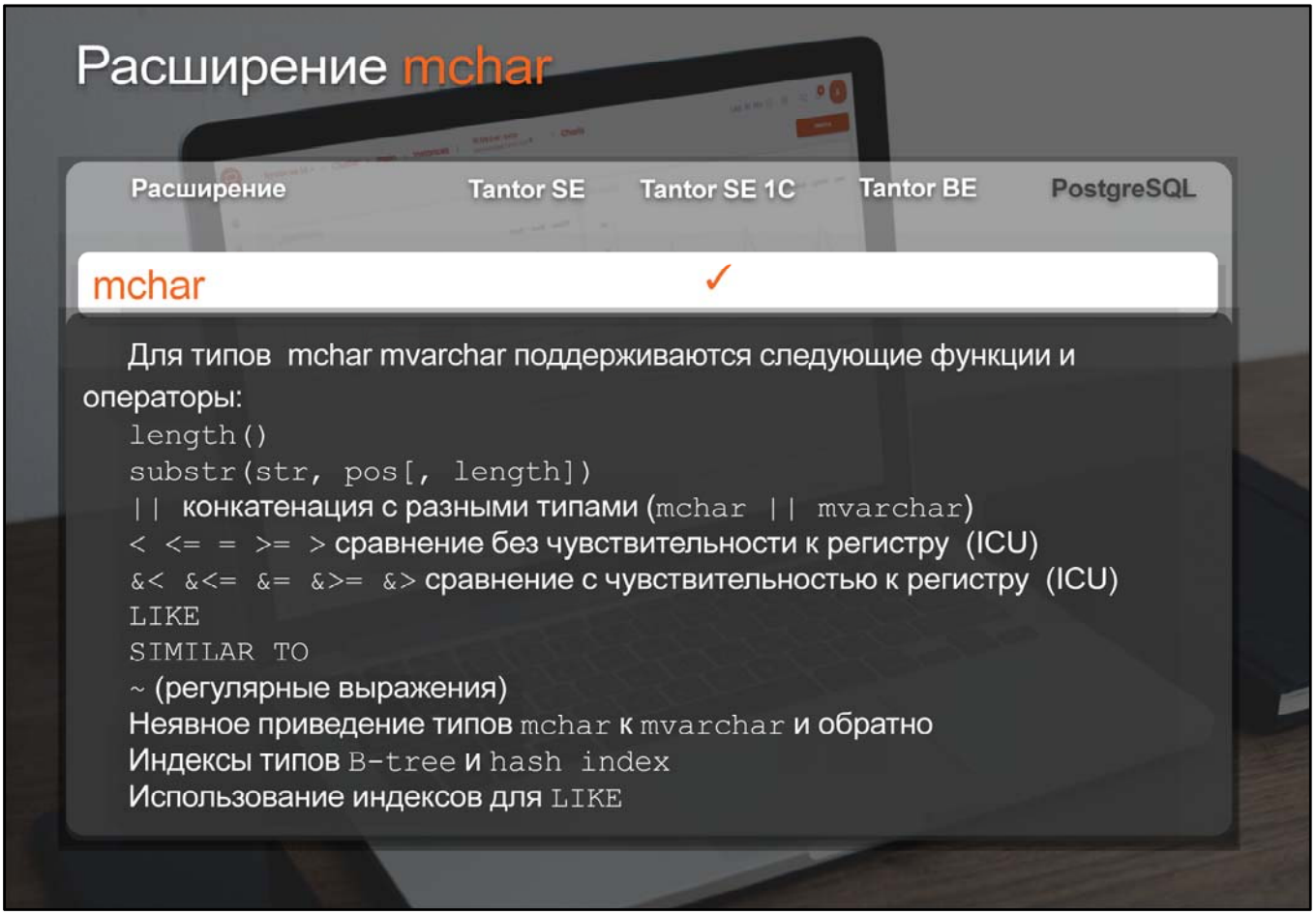
https://docs.tantorlabs.ru/tdb/ru/15_4/se1c/online_analyze.html



Расширение содержит функцию `fasttruncate`, предназначенную для эффективного усечения (`TRUNCATE`) временных таблиц без внесения изменений в таблицы системного каталога (`pg_class`). Это решение важно для предотвращения проблем с производительностью, связанных с увеличением размера каталога при частом использовании временных таблиц. Временные таблицы массово используются в приложениях “1С:Предприятие”.

- Для использования функционала нужно установить в базу данных расширение `fasttruncate`.
- Для усечения временной таблицы достаточно заменить команды

```
TRUNCATE таблица;
на
select fasttruncate('таблица');
```



Расширение **mchar**, доступно в версии СУБД tantor SE1C:

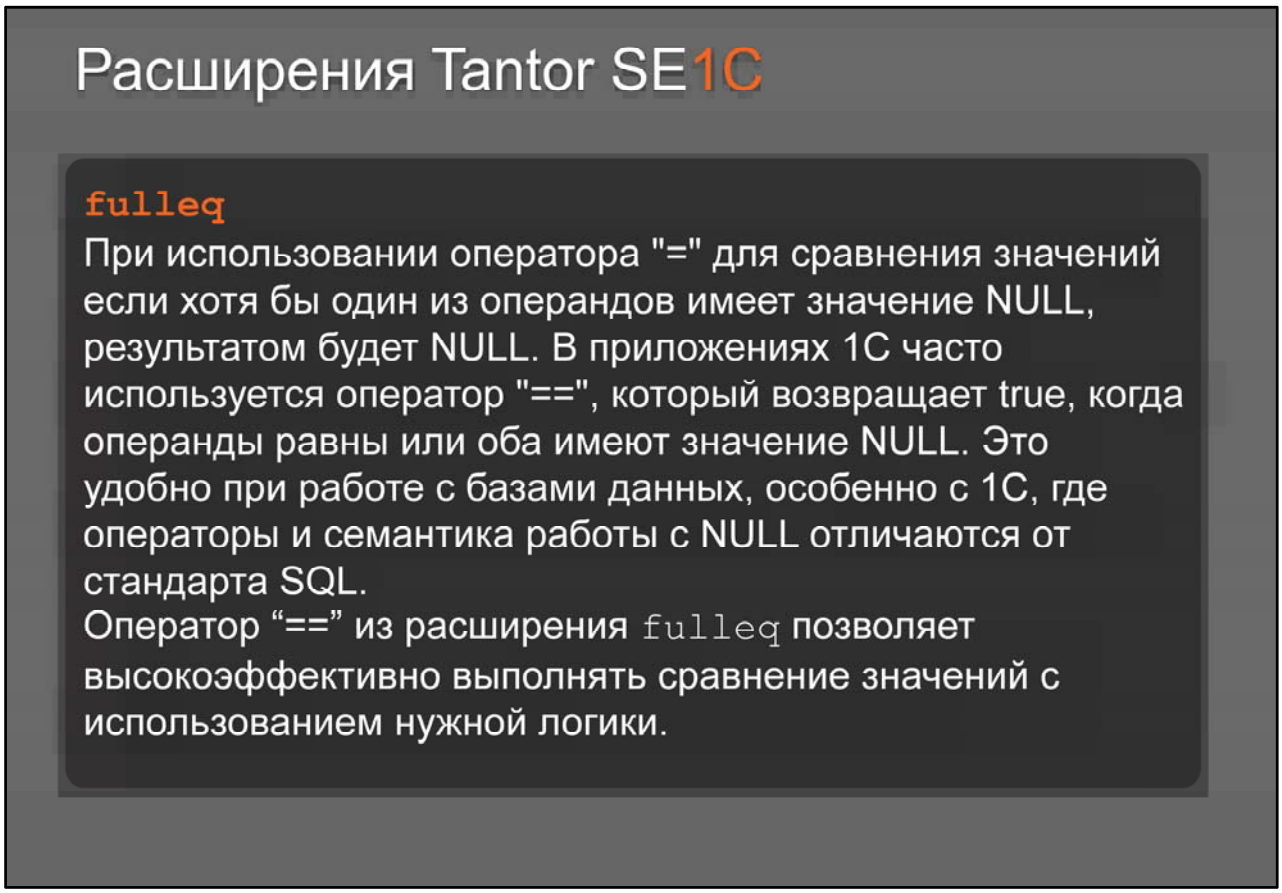
Предоставляет дополнительные типы данных для совместимости с Microsoft SQL server (MS SQL).

Добавляет поддержку типов mchar mvarchar для совместимостим с ms sql.

Для типов `mchar, 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
Индексы типов b-tree и hash index
Использование индексов для выполнения оператора LIKE
```

https://docs.tantorlabs.ru/tdb/ru/15_4/se1c/mchar.html



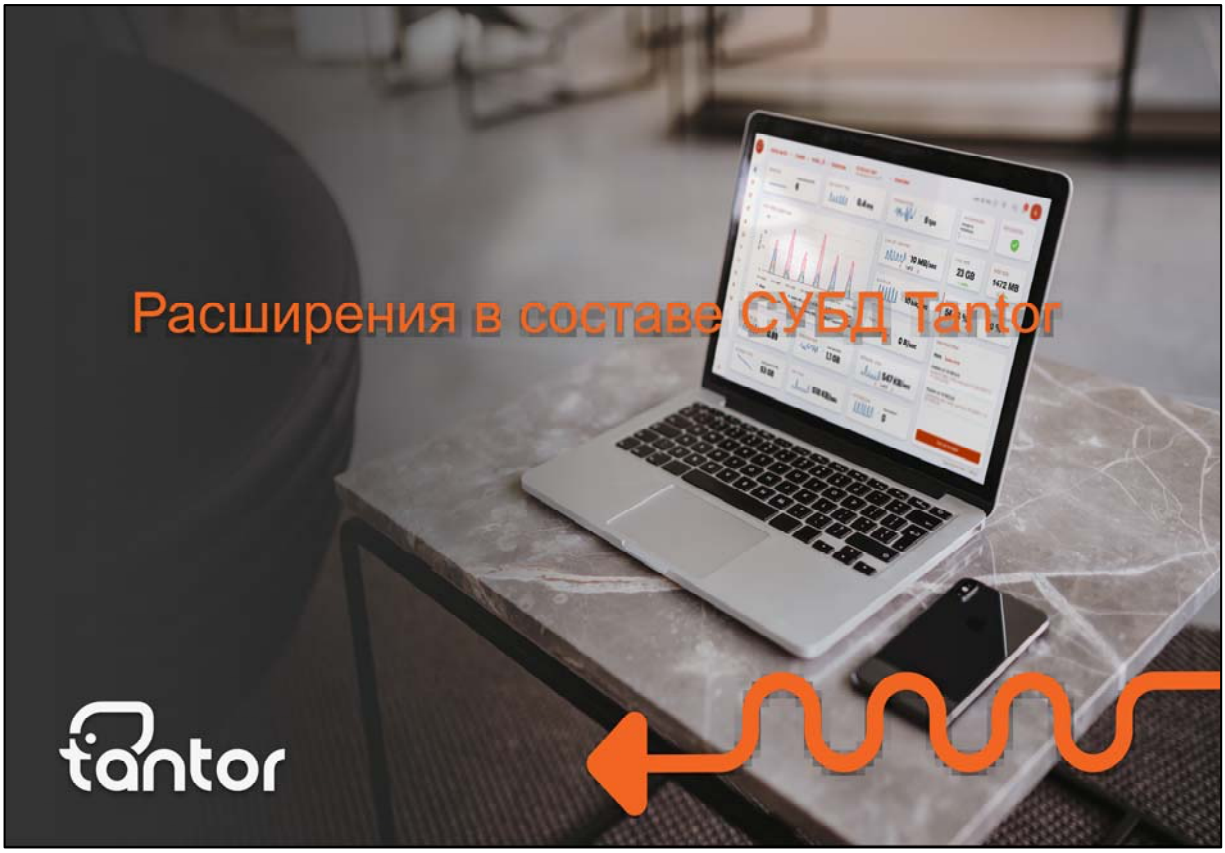
При использовании оператора "=" для сравнения значений если хотя бы один из операндов имеет значение NULL, результатом будет NULL. В приложениях 1С часто используется оператор "==", который возвращает true, когда операнды равны или оба имеют значение NULL. Это удобно при работе с базами данных, особенно 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.

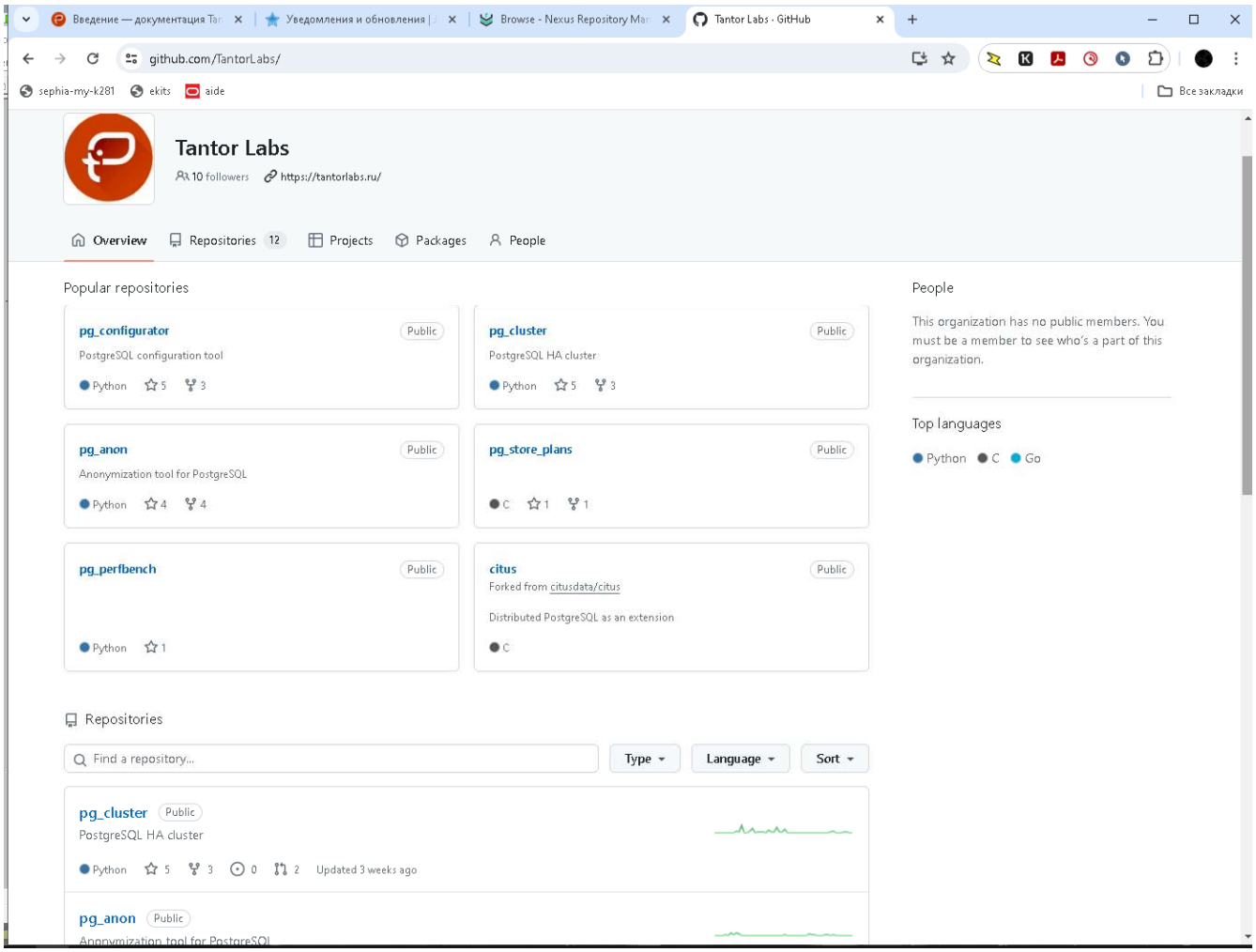
Оператор "==", примененный к двум операндам, возвращает false, если они не равны или один из них имеет значение NULL.

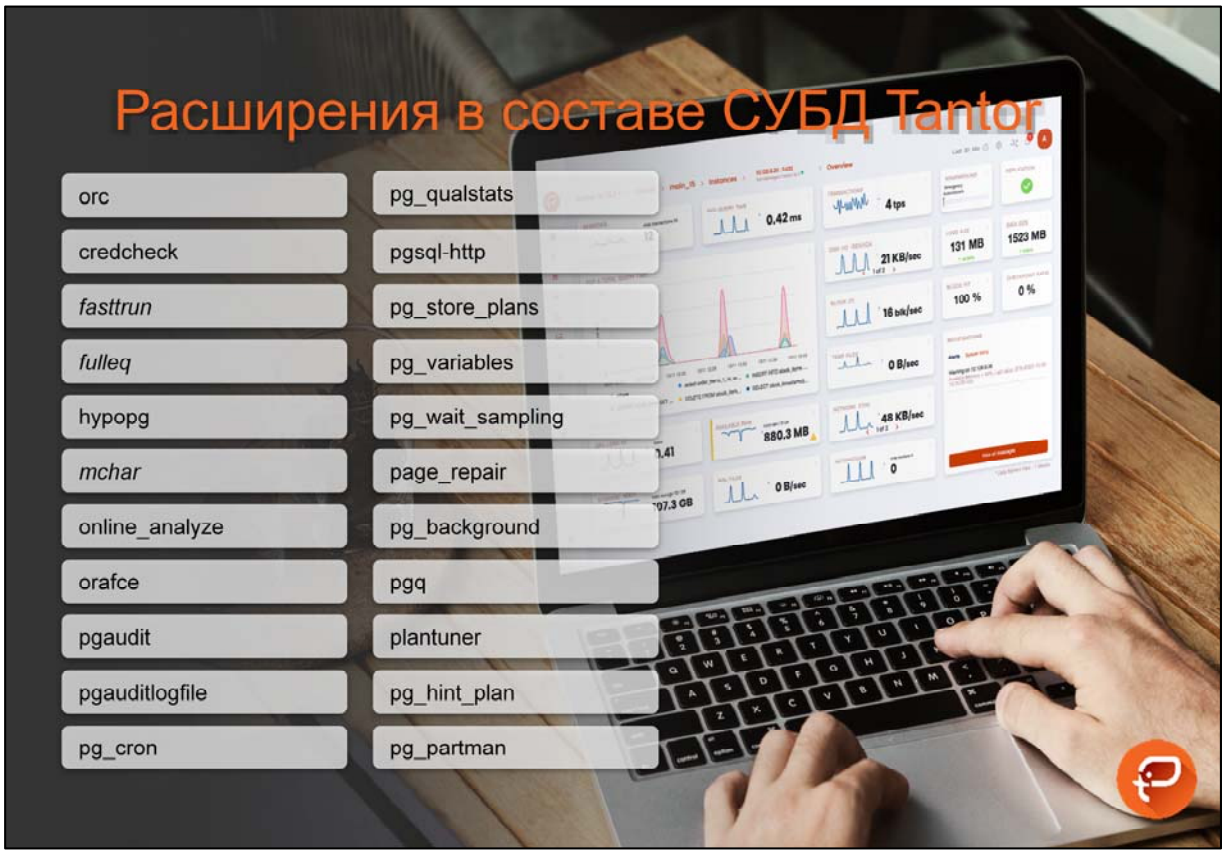
https://docs.tantorlabs.ru/tdb/ru/15_4/se1c/fulleq.html



ТанторЛабс участвует в жизни сообщества постгрес. Часть разработок ТанторЛабс, рассматриваемых далее доступно на https://github.com/TantorLabs/

ТанторЛабс является организатором конференций PG BootCamp https://pgbootcamp.ru/ в рамках в рамках глобальной инициативы сообщества.



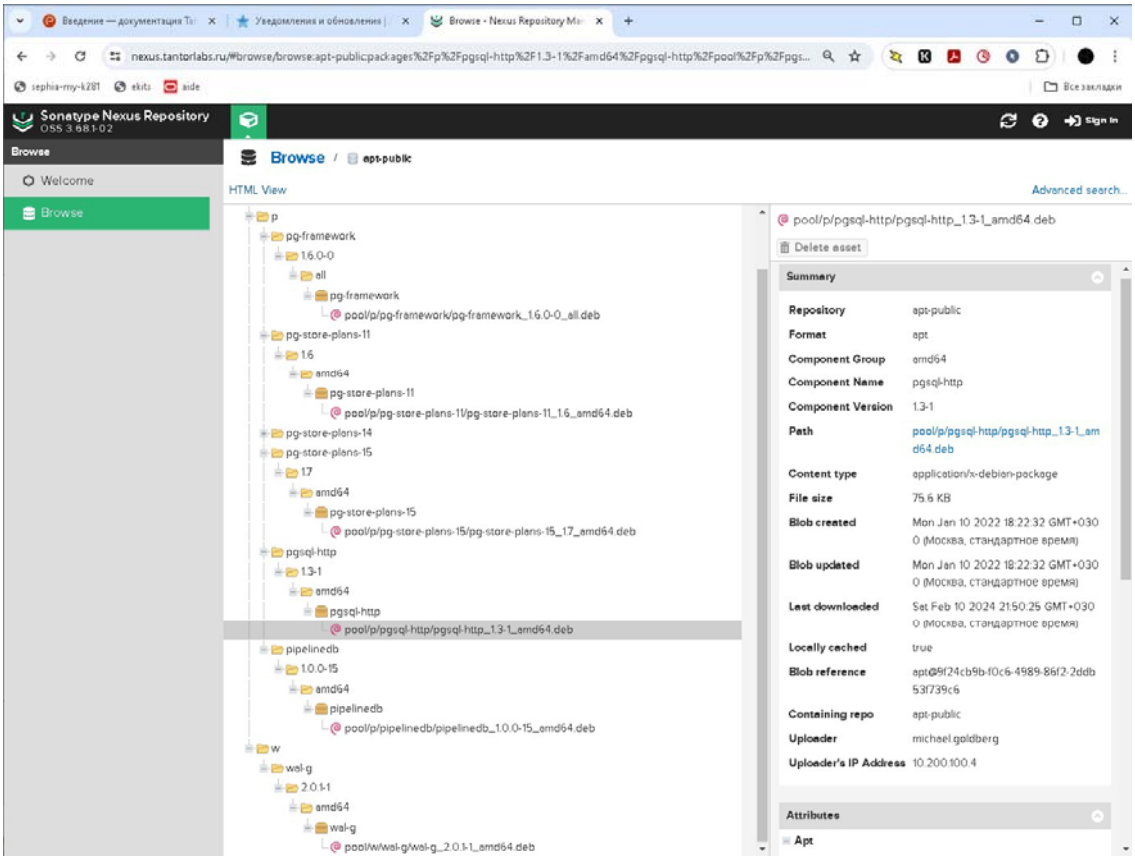


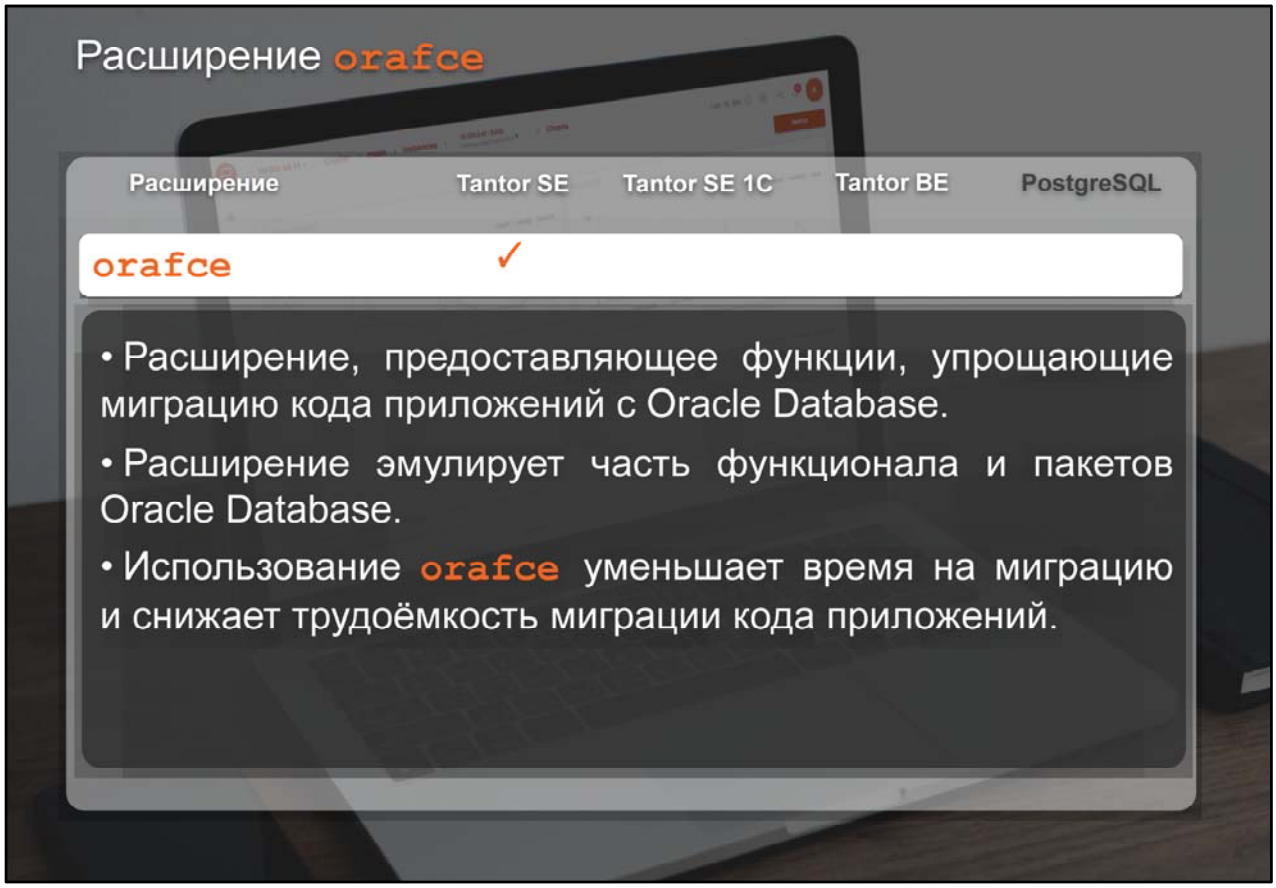
С PostgreSQL поставляются стандартные расширения. Все они присутствуют в СУБД Tantor соответствующих сборок.

Помимо них вместе с программным обеспечением СУБД Tantor поставляются дополнительные расширения и приложения.

Большая часть расширений (перечислены на слайде) не требуют отдельной установки (deb или rpm), находятся в едином пакете дистрибутива СУБД Tantor и могут использоваться так же, как используются стандартные расширения PostgreSQL.

Также имеются расширения и утилиты (приложения), которые распространяются отдельно в виде пакета или архива. Такие расширения и приложения могут скачиваться из репозиториев ТанторЛабс.



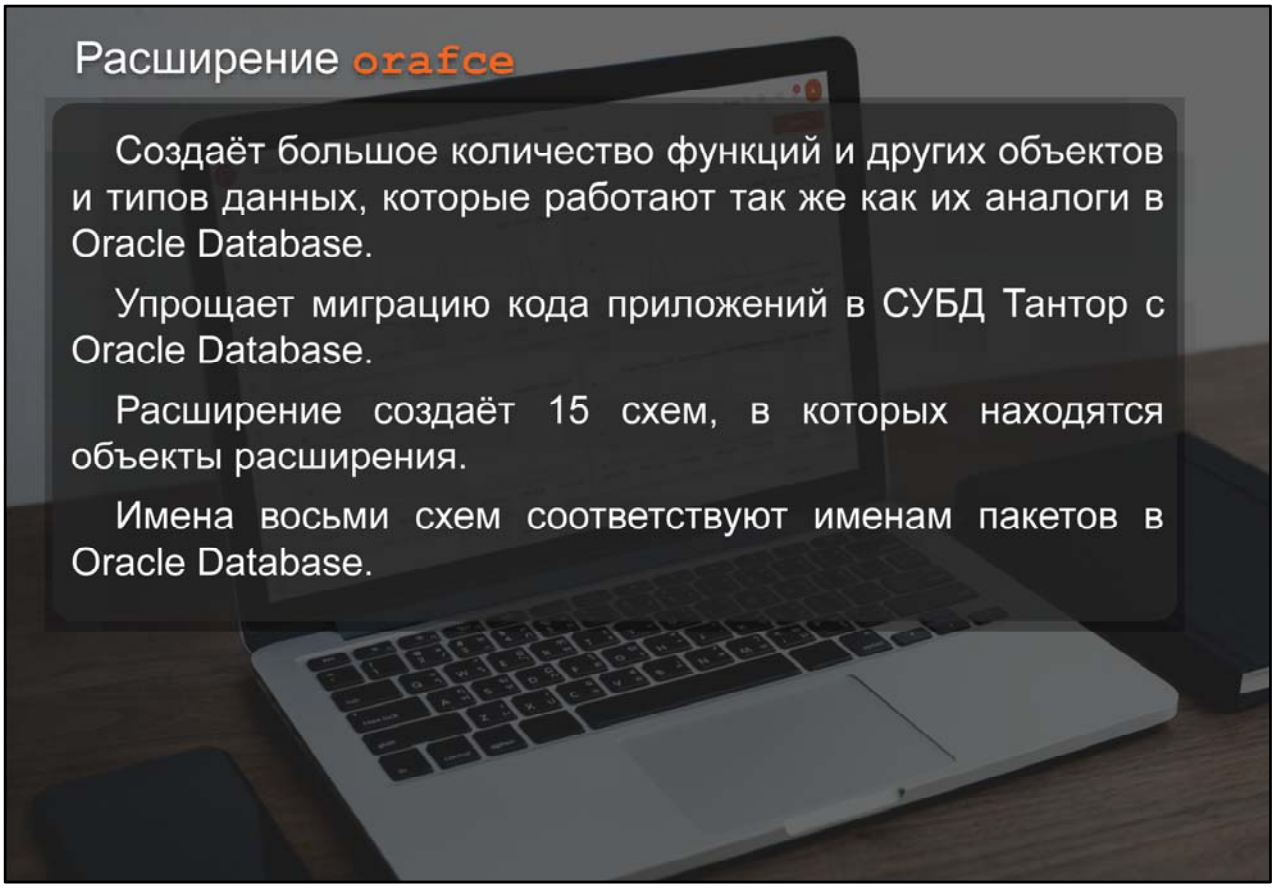


Расширение orafce присутствует в сборке Tantor SE.

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 СУБД Oracle Database на СУБД Тантор в командах и программном коде могут использоваться функции, процедуры, типы данных, которые имеются в Oracle Database и отсутствуют в постгрес и стандарте SQL. Переписывать код может быть достаточно трудоёмко, особенно если команд много.

Расширение **`orafce`** создаёт большое количество 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 постгрес.

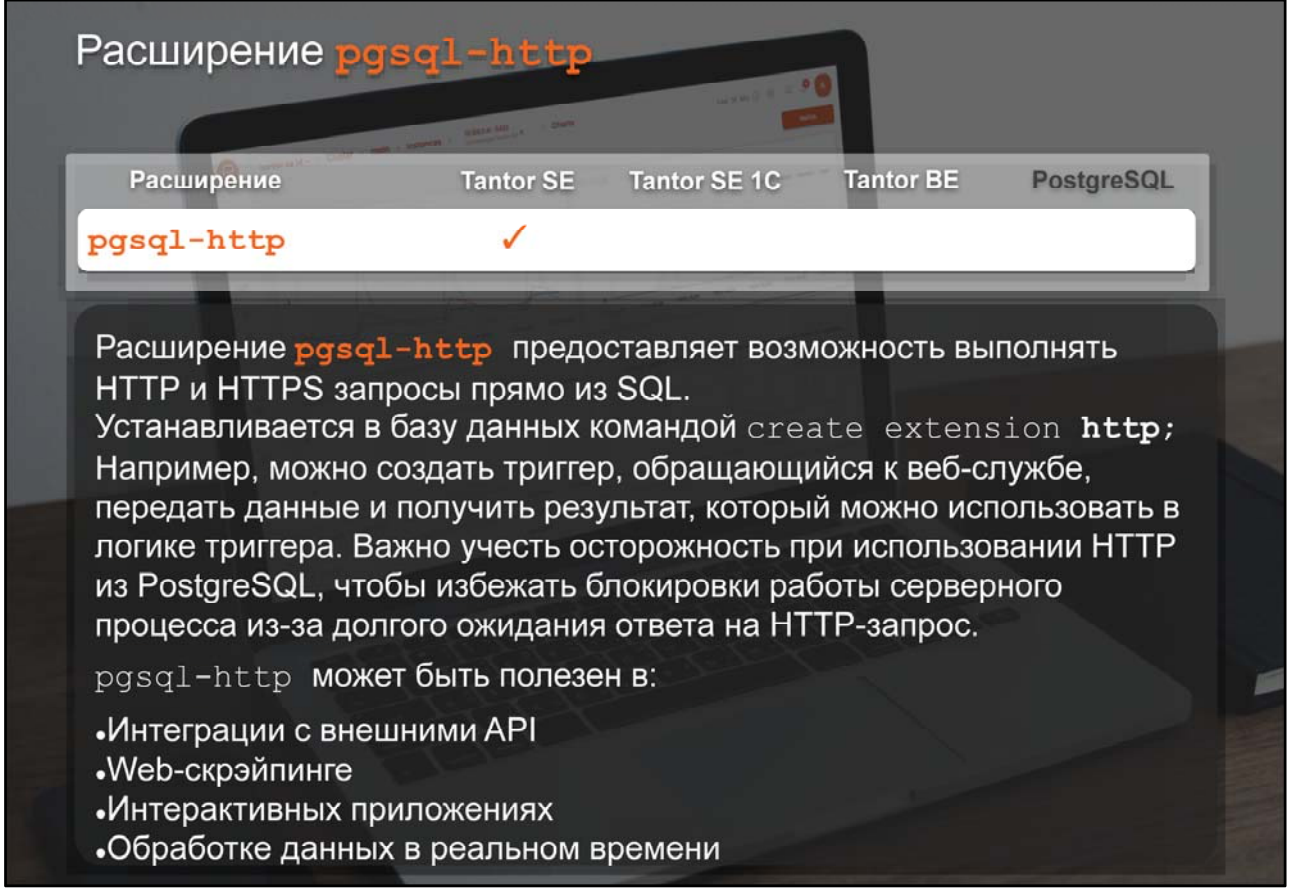
Цель расширения - упростить миграцию кода, дать возможность выполнения кода без существенных изменений и постепенное переписывание и оптимизацию выполнения команд SQL.

Также функции из этого расширения могут быть полезными и сами по себе.

В Oracle Database программные единицы (функции и процедуры) находятся в "пакетах".

В постгрес 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/15_4/se/orafce.html



Расширение `pgsql-http` доступно только в сборке Tantor SE.

Устанавливается в базу данных командой `create extension http;`

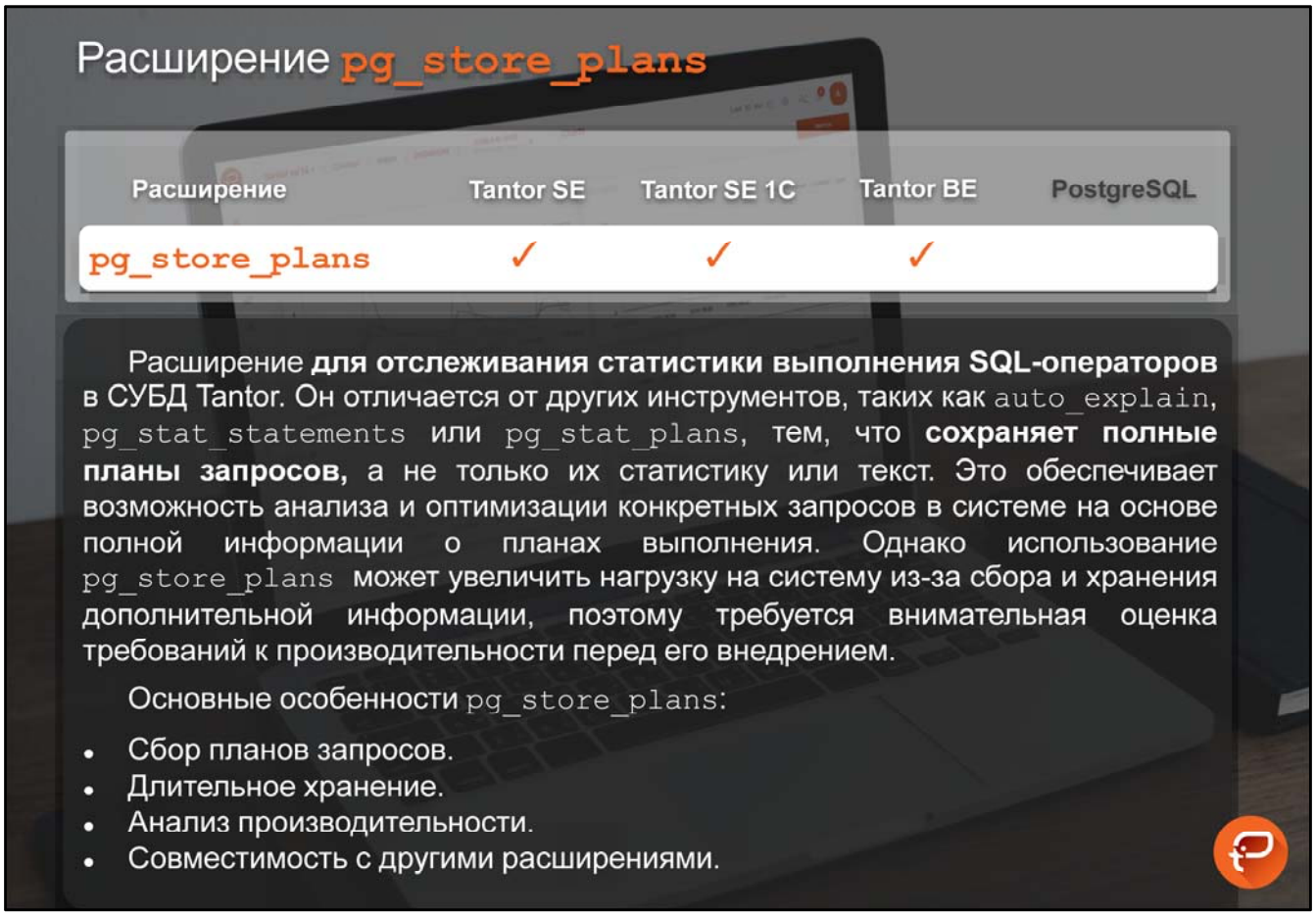
Расширение `pgsql-http` предоставляет возможность выполнять HTTP и HTTPS запросы прямо из SQL.

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 задачах:

1) **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) **Web-скрэйпингом:** Функции расширения `pgsql-http` могут быть использованы для извлечения данных с веб-страниц напрямую в PostgreSQL, что может быть полезно для аналитики и других задач.
3) **Интерактивные приложения:** В некоторых сценариях использования PostgreSQL может быть частью интерактивного веб-приложения, где база данных взаимодействует с пользователем посредством HTTP. `pgsql-http` может использоваться для отправки запросов на сервер приложений и получения ответов на них.
4) **Обработка данныых в режиме реального времени:** позволяет обеспечить доступ к данным, которые постоянно обновляются и доступны клиентам по протоколу HTTP. С помощью `pgsql-http`, можно запросить эти данные напрямую со стороны сервера базы данных.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/pgsql-http.html



Во всех версиях СУБД Tantor имеется доработанное ТанторЛабс расширение **pg_store_plans:**

Расширение предоставляет средства для отслеживания статистики плана выполнения всех операторов SQL, выполняемых СУБД Tantor.

Используется платформой Tantor для сбора статистики планов запросов.

В отличие от других инструментов, таких как auto_explain, pg_stat_statements или pg_stat_plans, pg_store_plans способен собирать и хранить полные планы запросов, а не только статистику или текст запросов.

Это позволяет вам анализировать, как конкретные запросы выполняются в системе, и оптимизировать их на основе этой информации.

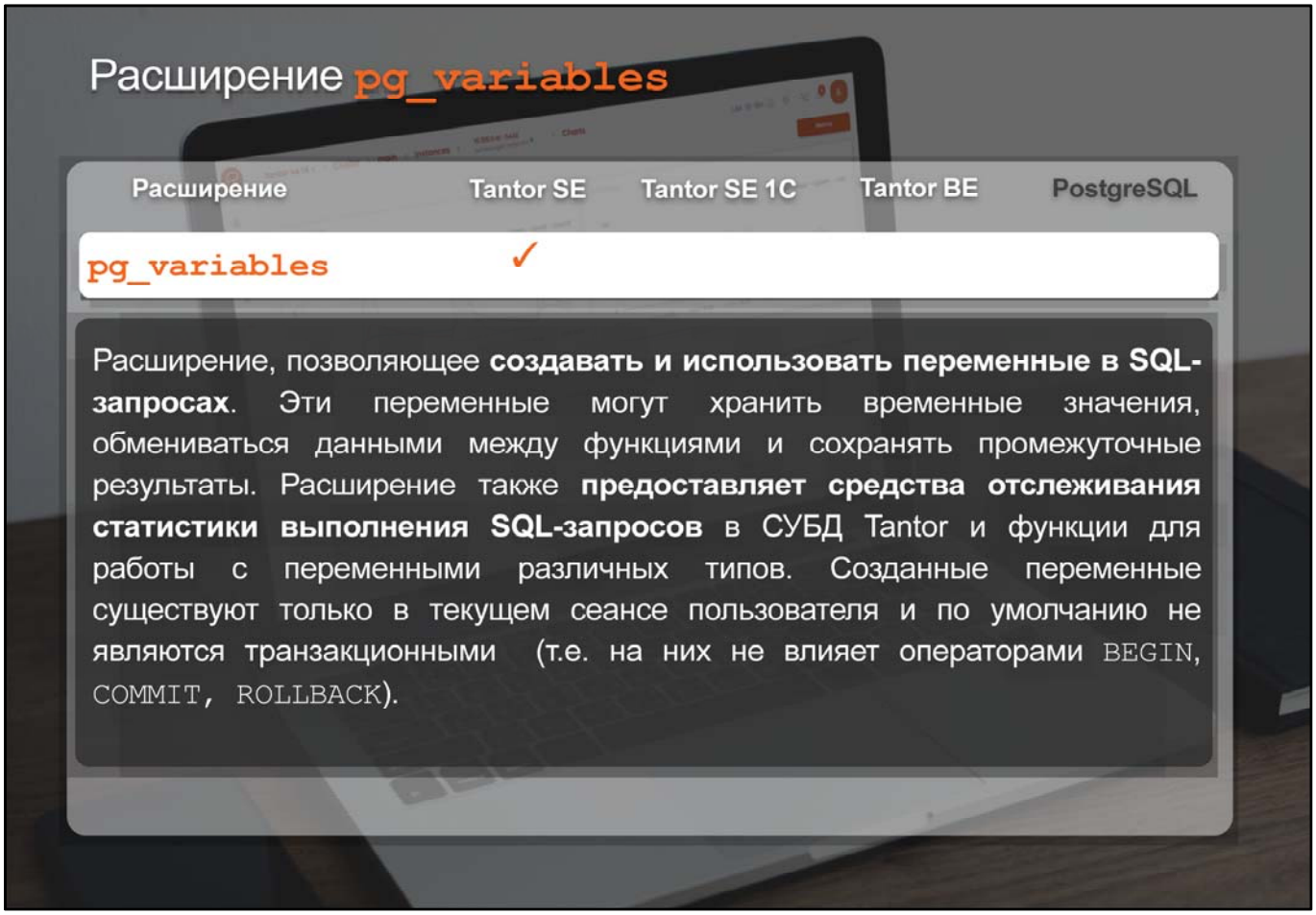
Использование pg_store_plans может увеличить нагрузку на вашу систему из-за дополнительного сбора и хранения информации о планах запросов.

Поэтому перед его использованием следует тщательно оценить требования к производительности.

Основные особенности pg_store_plans:

1) Сбор планов запросов: pg_store_plans автоматически сохраняет планы выполнения запросов, что позволяет исследовать, как запросы выполняются в вашей базе данных.
2) Длительное хранение: pg_store_plans хранит планы запросов на протяжении длительного времени, что позволяет вам анализировать исторические данные и определять, как изменения в вашем приложении или базе данных влияют на производительность запросов.
3) Анализ производительности: с помощью pg_store_plans 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) Совместимость с другими расширениями: pg_store_plans может быть использован в сочетании с другими расширениями PostgreSQL для более глубокого анализа производительности, такими как pg_stat_statements или pg_qualstats.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/pg_store_plans.html



В Tantor SE имеется расширение `pg_variables`.

Расширение pg_variables позволяет определять и использовать переменные внутри запросов SQL на сервере PostgreSQL.

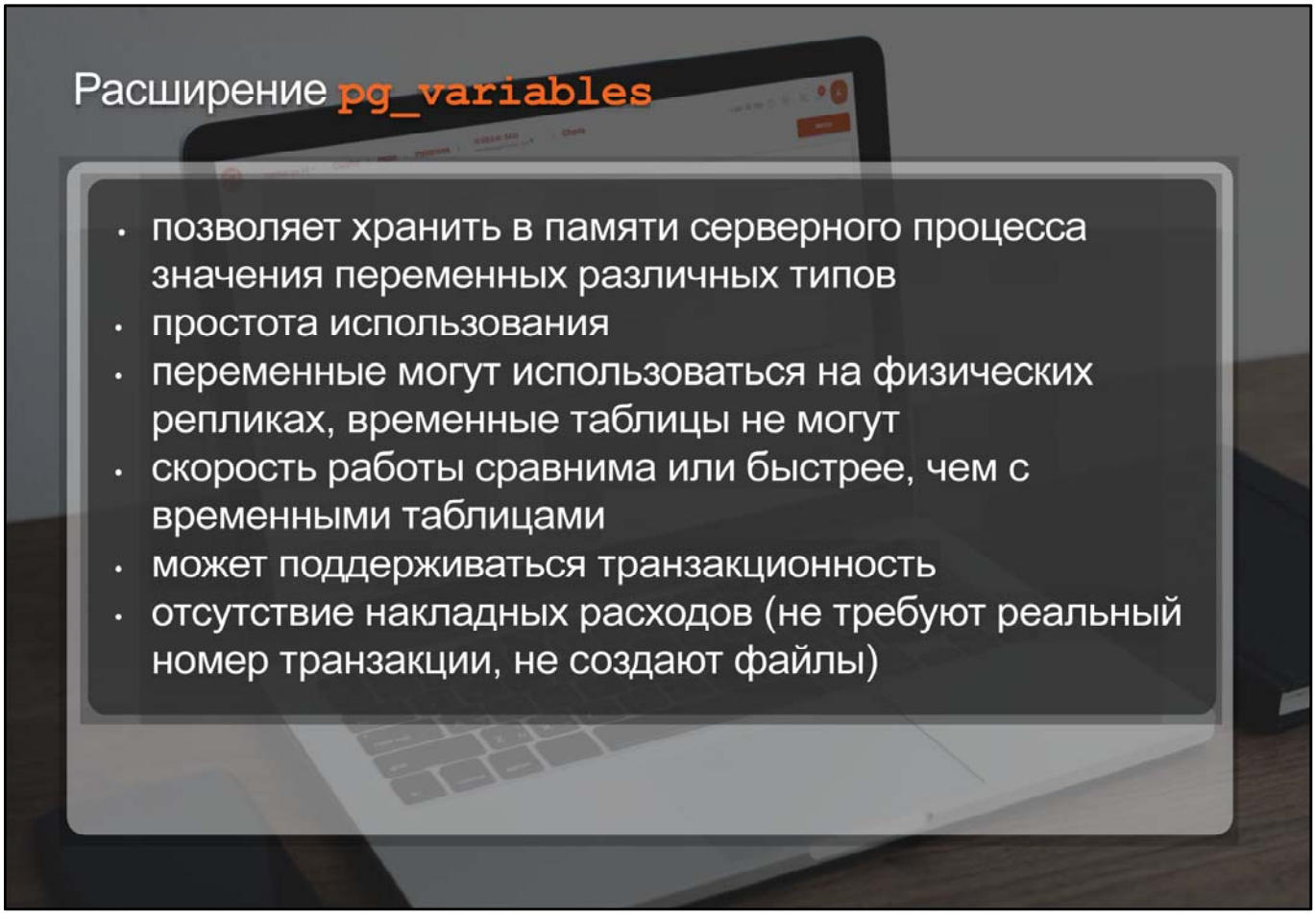
Переменные могут быть использованы для хранения временных значений, обмена данными между функциями, хранения промежуточных результатов и т.д.

Предоставляет средства для отслеживания статистики плана выполнения для всех SQL-запросов, выполненных сервером Tantor.

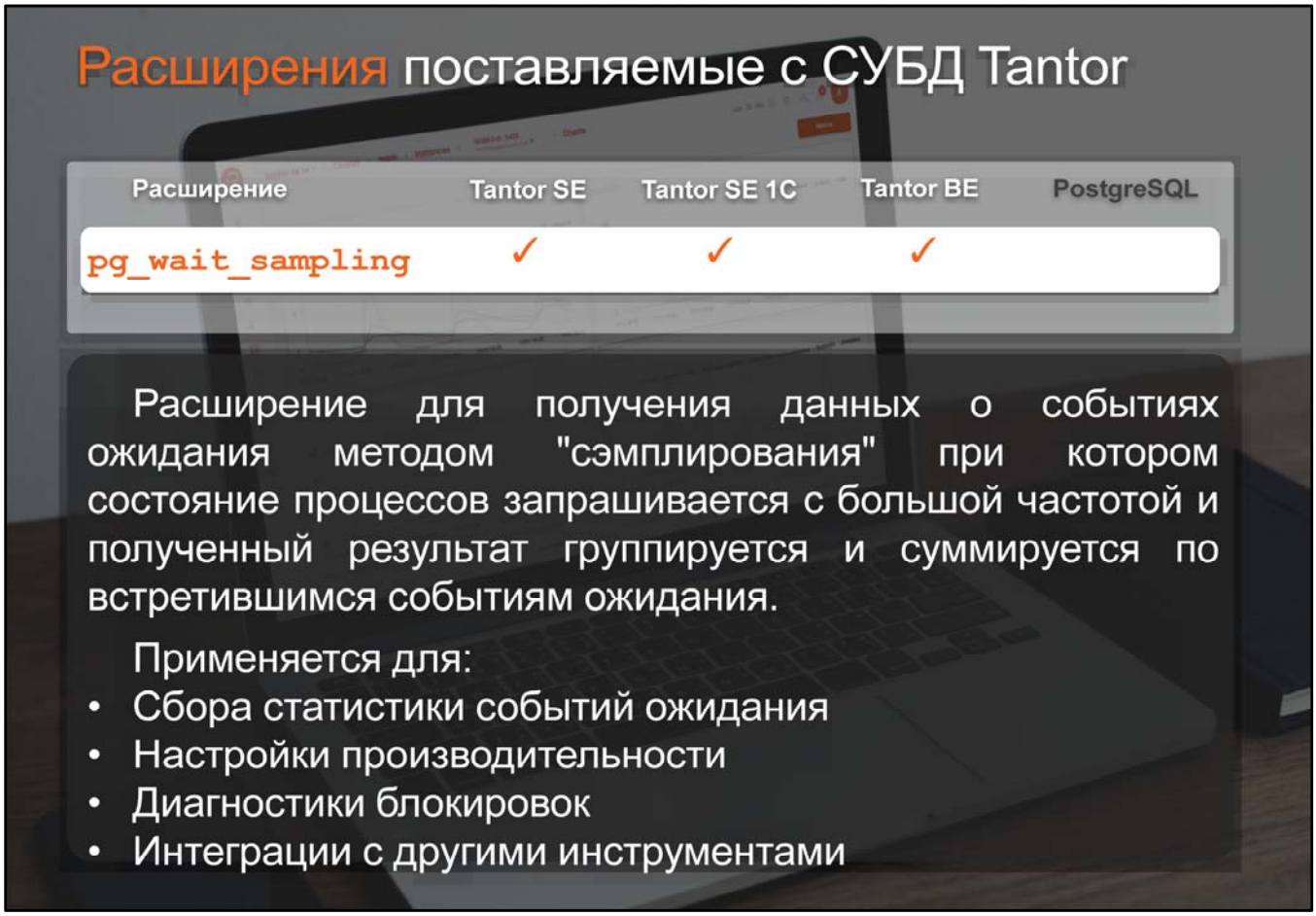
Предоставляет функции для работы с переменными различных типов. Созданные переменные существуют только в текущем сеансе пользователя.

По умолчанию созданные переменные не являются транзакционными (т.е. на них не влияют команды `BEGIN, COMMIT, ROLLBACK`).

```
astra@tantor:~$ locate pg_variables
/opt/tantor/db/16/lib/postgresql/pg_variables.so
/opt/tantor/db/16/lib/postgresql/bitcode/pg_variables
/opt/tantor/db/16/lib/postgresql/bitcode/pg_variables.index.bc
/opt/tantor/db/16/lib/postgresql/bitcode/pg_variables/pg_variables.bc
/opt/tantor/db/16/lib/postgresql/bitcode/pg_variables/pg_variables_record.bc
/opt/tantor/db/16/share/postgresql/extension/pg_variables--1.0--1.1.sql
/opt/tantor/db/16/share/postgresql/extension/pg_variables--1.0.sql
/opt/tantor/db/16/share/postgresql/extension/pg_variables--1.1--1.2.sql
/opt/tantor/db/16/share/postgresql/extension/pg_variables--1.2.sql
/opt/tantor/db/16/share/postgresql/extension/pg_variables.control
```



Расширение позволяет хранить в памяти серверного процесса значения переменных различных типов, в том числе численных, текстовых, датавременных, 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.
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 памяти используется.
Функционал аналогичен переменным пакетов и контекстов приложений (application contexts) в Oracle Database, поэтому расширение может использоваться при миграции приложений на СУБД Тантор.
https://docs.tantorlabs.ru/tdb/ru/15_4/se/pg_variables.html



Расширение `pg_wait_sampling доступно` во всех сборках СУБД Tantor.

В экземпляре одновременно работает много фоновых и северных процессов. Процессы либо активны, либо находятся в ожидании событий, таких как блокировки, операции дискового ввода-вывода, получения команд от клиента и множество других событий ожидания.

Процессы не сохраняют историю того, какие события они ожидают, так как накладные расходы на сохранение на порядки больше, чем трудоемкость операции. Можно получить только состояние процесса опросив его. Длительность многих событий ожидания может быть короткой, а частота большой. Для получения данных о событиях ожидания используется метод "сэмплирования": состояние процессов запрашивается с большой частотой и полученный результат суммируется по встретившимся событиям ожидания.

`pg_wait_sampling` это расширение для сбора статистики событий ожидания без больших накладных расходов.

pg_wait_sampling применяется для:

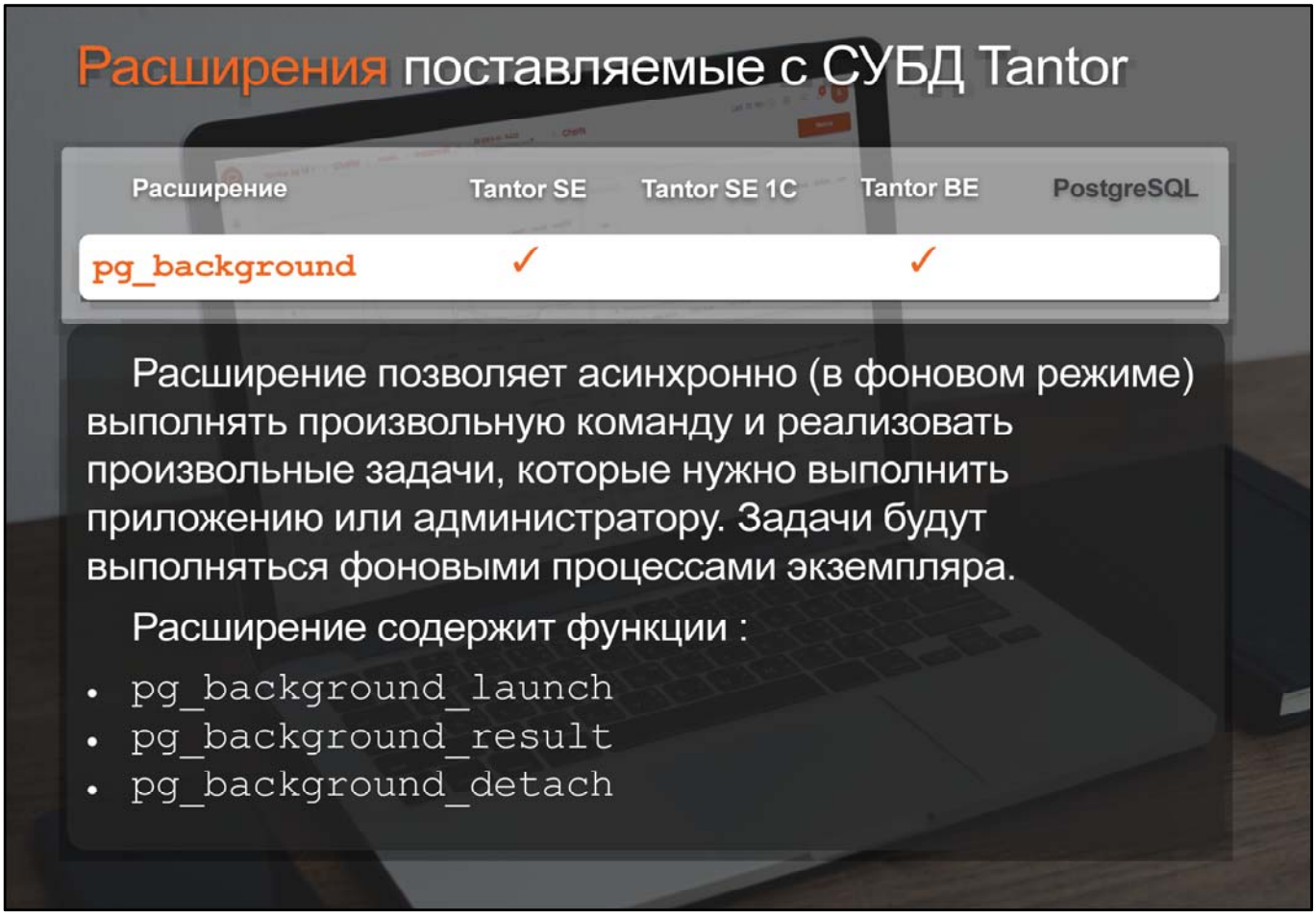
• Сбора статистики событий ожидания: `pg_wait_sampling` собирает статистику о том, какие процессы ждут и чего они ждут. Это может помочь определить, какие операции вызывают задержки.

• Настройки производительности: Используя информацию, собранную `pg_wait_sampling`, можно определить проблемные области в производительности и найти способы улучшения.

• Диагностики блокировок: `pg_wait_sampling` может помочь в диагностике и решении проблем с блокировками, показывая, какие процессы ожидают освобождения блокировок.

• Интеграции с другими инструментами: `pg_wait_sampling` может быть интегрирован с другими инструментами мониторинга и производительности для более глубокого анализа.

https://docs.tantorlabs.ru/tdb/ru/15_4/se/pg_wait_sampling.html

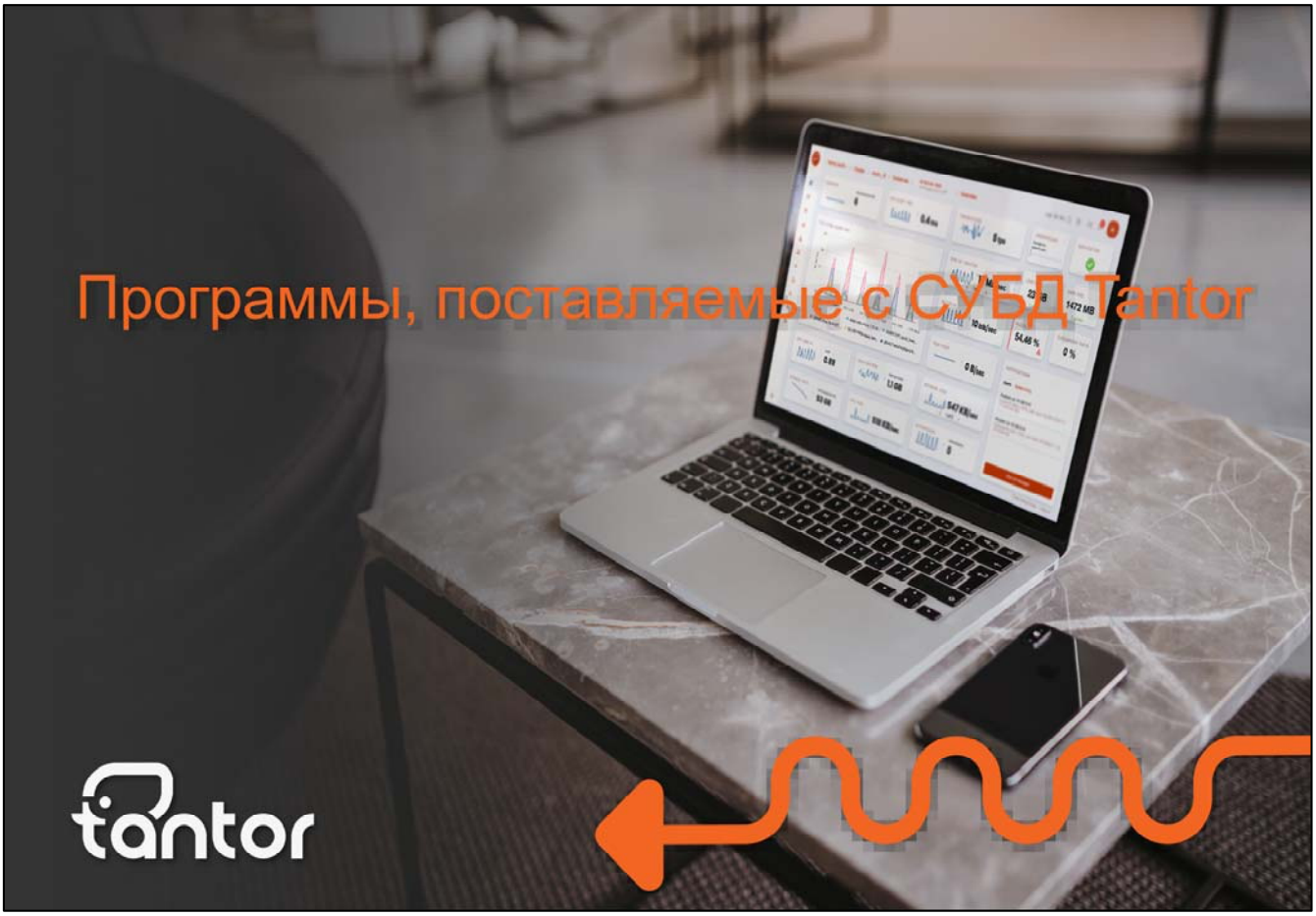


Расширение `pg_background` имеется в версиях Tantor SE и Tantor BE.

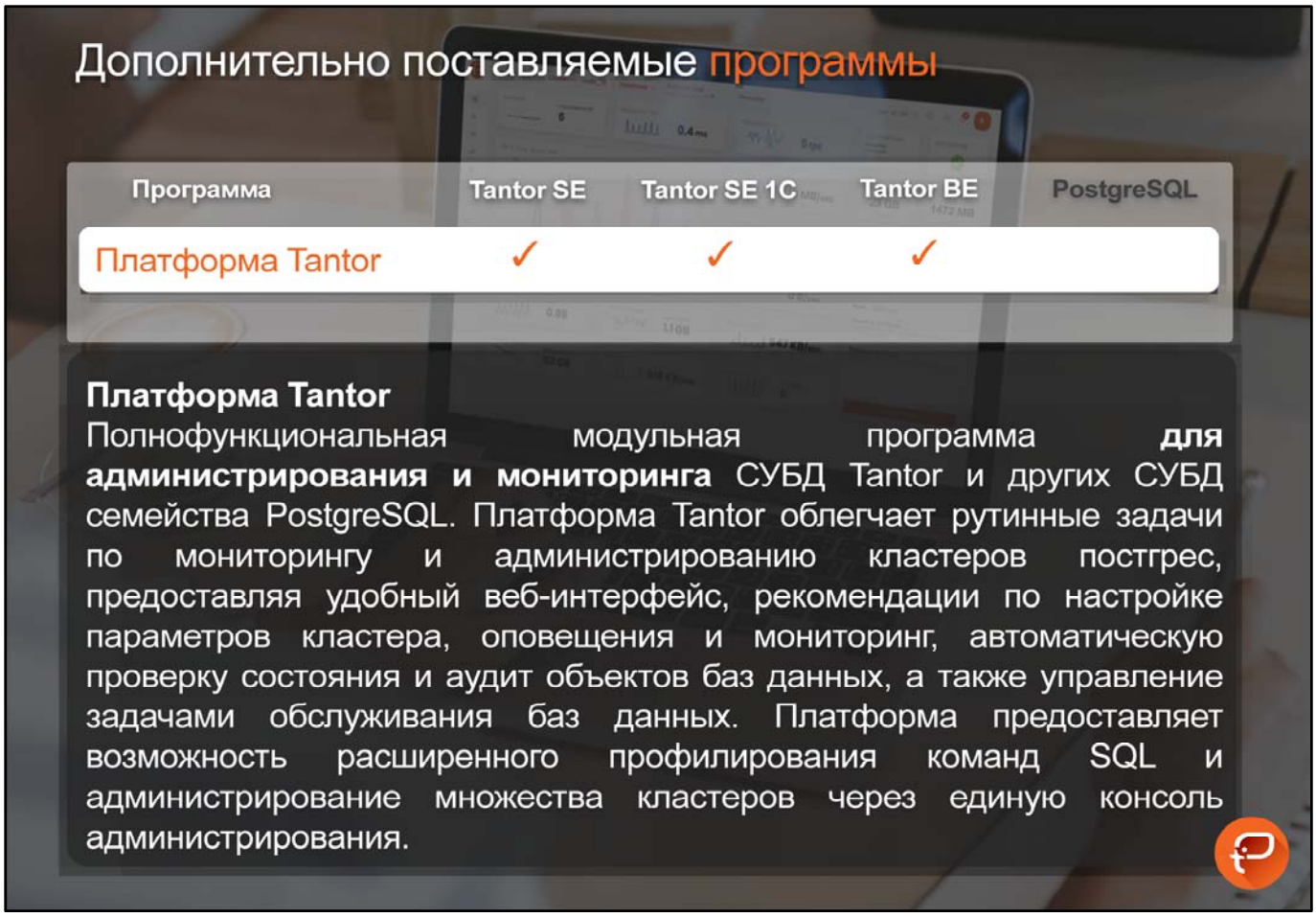
Расширение позволяет выполнять асинхронно (в фоновом режиме) произвольные операции. С помощью расширения можно вручную реализовать произвольные задачи, которые нужно выполнять в фоновом режиме приложению или администратору. Задачи будут выполняться фоновыми процессами экземпляра. Расширение предоставляет 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 процесса фонового рабочего.
- `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.


Программы, поставляемые с СУБД Tantor
tantor

Программы, поставляемые с СУБД Tantor:
Платформа Тантор
`WAL-G`
`pg_configurator`
`pg_anon`
`pg_cluster`
`pg_repack`
`pgcompacttable.`
Программы `pg_repack, pgcompacttable` входят в дистрибутив СУБД Tantor, остальные скачиваются и устанавливаются из отдельных пакетов.

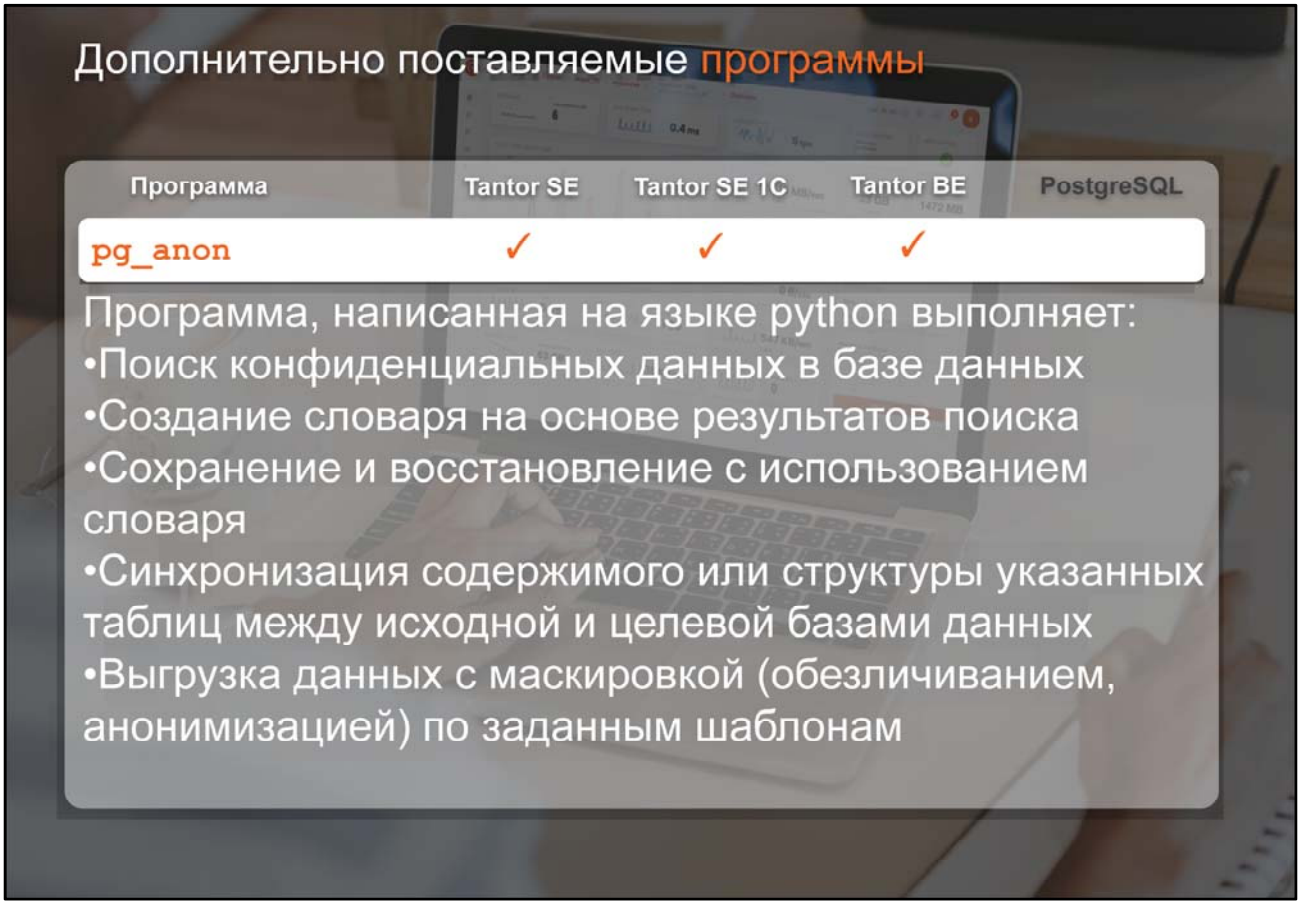


Полнофункциональная модульное приложение **для мониторинга и администрирования** СУБД Tantor и других СУБД семейства PostgreSQL. Платформа Tantor облегчает рутинные задачи по мониторингу и администрированию кластеров постгрес, предоставляя удобный веб-интерфейс, рекомендации по настройке параметров кластера, оповещения и мониторинг, автоматическую проверку состояния и аудит объектов баз данных, а также управление задачами обслуживания баз данных. Платформа предоставляет возможность расширенного профилирования команд SQL и администрирование множества кластеров через единую консоль администрирования.

Платформа Tantor может использоваться как основной инструмент мониторинга и администрирования всех СУБД основанных на коде PostgreSQL в сети предприятия и организации.

Платформа упрощает ежедневную работу с СУБД и значительно упрощает управление СУБД. Функциональность программного обеспечения включает: простой и функциональный пользовательский интерфейс, автоматическую адаптивную настройку кластера, обзор системы, оповещения и мониторинг, автоматическую проверку состояния и аудит схемы, разрешение задач обслуживания через графический интерфейс, расширенный профилирование запросов и администрирование множества кластеров через единую консоль администрирования.

Программные продукты, обладающие аналогичным функционалом обычно называют "Enterprise manager".



`pg_anon` доступно для всех сборкок СУБД Tantor.

`pg_anon` - приложение, написанное на языке python.

Приложение выполняет следующие задачи:

1) Создаёт в базах данных схему `anon_funcs`, которая содержит набор функций для маскировки (обезличивания, анонимизации) данных
2) Поиск конфиденциальных данных в базе данных на основе словаря
3) Создание словаря на основе результатов поиска (рекогносцировка)
4) Сохранение и восстановление с использованием словаря. Для разных баз данных можно предоставить отдельные файлы словаря
5) Синхронизация содержимого или структуры указанных таблиц между исходной и целевой базами данных

Приложение скачивается и устанавливается отдельно из пакета. Например, `pg-anon-tantor-all_0.1.1-1jammy1_all.deb`

Wal-G (Write-Ahead Log Guard) - это утилита для создания шифрованных резервных копий кластера баз данных (полных и инкрементальных) и архивирования WAL-сегментов, 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, в том числе инкрементальные ("дельта-копии").

Утилита поставляется в пакетах `deb` или `rpm`, скачиваемых их соответствующего репозитория. Например, из публичного:

https://nexus.tantorlabs.ru/repository/apt-public/pool/w/wal-g/wal-g_2.0.1-1_amd64.deb

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`:

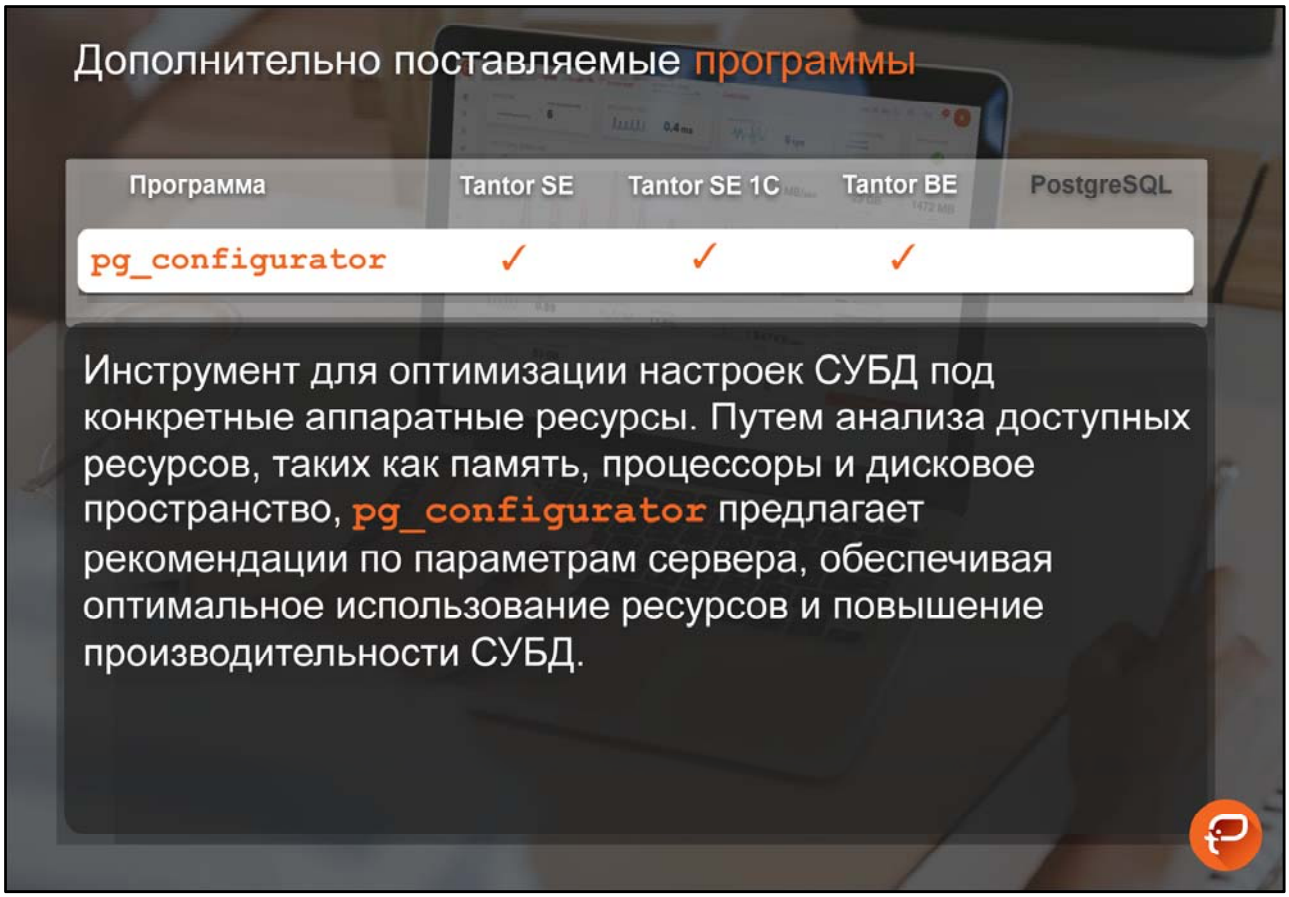
```
wal-g backup-push $PGDATA >> ~/backup-push.log 2>&1
```

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. 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 ненужные администратору бэкапы и связанные с ними файлы журналов.
4) шифровать файлы перед передачей их в хранилище.



Приложение **pg_configurator** доступно для всех сборок СУБД Tantor. Приложение поставляется отдельно. Например в виде архива `pg-configurator-tantor-all_22.10.17-1astra1.7-1_all.deb`

Представляет собой скрипт `pg_configurator` на языке pyton, устанавливаемый по пути `/usr/bin`.

pg_configurator - инструмент оптимизации настроек СУБД под конкретные аппаратные ресурсы, профиль нагрузки и требования поставки, чтобы обеспечить наилучшую производительность и эффективность работы СУБД, учитывая доступные ресурсы системы и особенности профиля нагрузки, с которым сервер будет работать.

pg_configurator предлагает рекомендуемые параметры конфигурации на основе анализа аппаратных ресурсов, таких как доступная память, количество процессоров и дисковое пространство и т.д. Это позволяет оптимально использовать имеющиеся ресурсы и увеличить производительность сервера.

https://github.com/TantorLabs/pg_configurator

Практическое задание

1. Расширение `orafce`
2. Расширение `pg_variables`
3. Расширение `page_repair`
   - Подготовка реплики
   - Подготовка таблицы
   - Восстановление страницы с помощью `page_repair`
   - Обнуление страницы
4. Отладка подпрограмм
   - Установка расширения из исходных кодов на примере `pldebugger`
   - Отладка функции в pgAdmin
   - Отладка подпрограмм в DBeaver
5. Обработка строк большого размера и строковый буфер

tantor