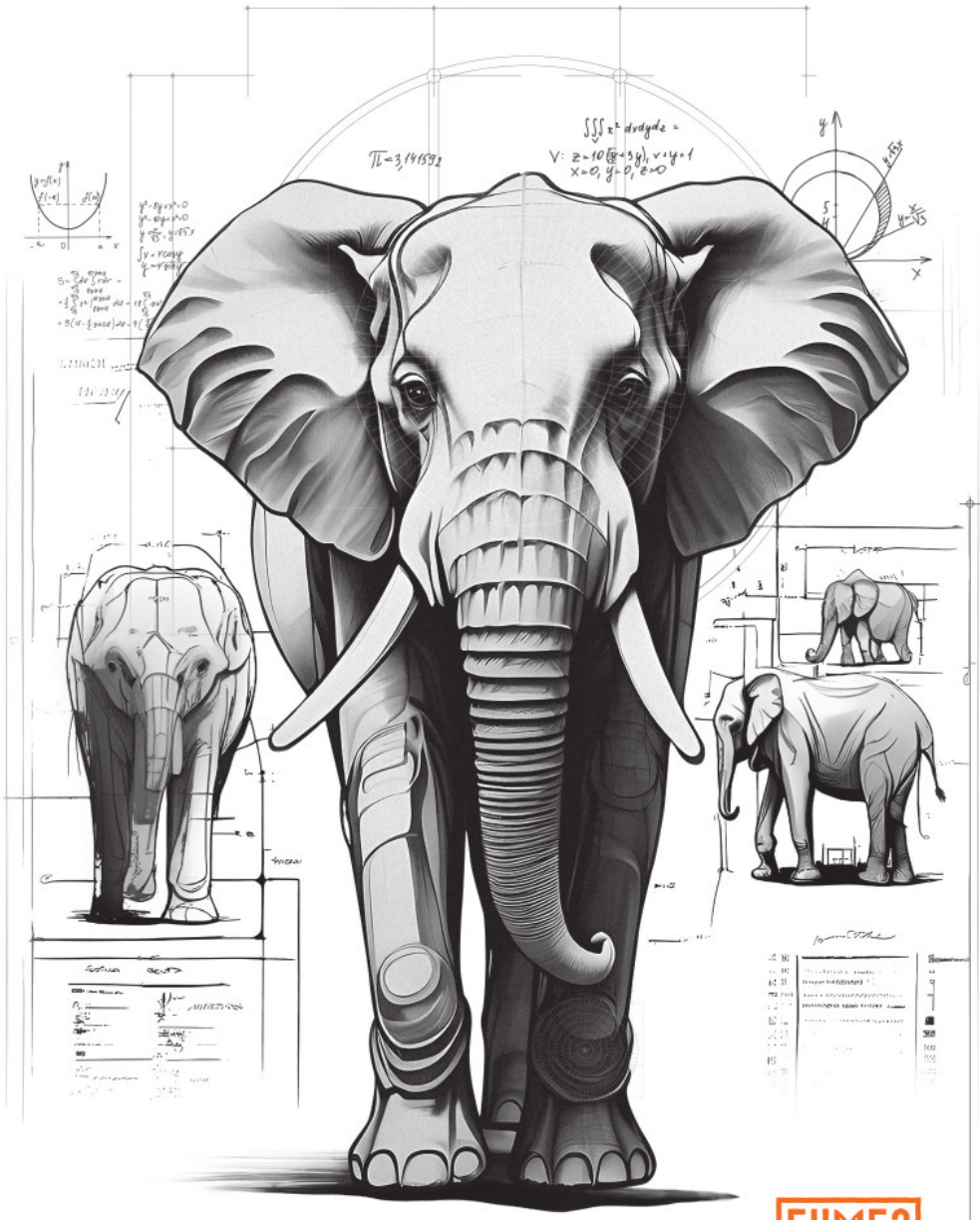
Алексей Лесовский

Мониторинг PostgreSQL



PostgresPro

БУМБА

Алексей Лесовский

Мониторинг PostgreSQL

PostgresPro БУМБА

Москва
2024

УДК 004.65
ББК 32.972.134
Л50

Лесовский А. В.

Л50 Мониторинг PostgreSQL / А. В. Лесовский. — М. : Бумба, 2024. — 247 с.

ISBN 978-5-907754-42-3

Мониторинг PostgreSQL составляет важную часть работы администратора, помогая отвечать на многие вопросы, связанные с производительностью. Эта книга всесторонне охватывает обширную тему мониторинга, соединяя в себе справочные материалы об инструментарии, практические приемы его использования и способы интерпретации полученных данных. Знание внутреннего устройства PostgreSQL и особенностей мониторинга, почерпнутое из этой книги, поможет в долгосрочной перспективе эффективно эксплуатировать СУБД и успешно решать возникающие задачи.

Для администраторов баз данных, системных администраторов, специалистов по надежности.

Сайт книги: github.com/lesovsky/postgresql-monitoring-book.

УДК 004.65
ББК 32.972.134

ISBN 978-5-907754-42-3

Оглавление

Содержание

Предисловие

К администрированию PostgreSQL я пришел случайно. Как системный администратор в отделе веб-разработки одной екатеринбургской компании, я поддерживал работу нескольких серьезных веб-проектов на Ruby on Rails. Стек технологий в этих проектах состоял из множества сложных и интересных инструментов, среди которых был и PostgreSQL.

За время работы в этой компании мне лишь изредка приходилось погружаться в тонкости функционирования СУБД. Моей первой серьезной задачей было обновление СУБД с версии 9.0 на 9.2 под нагрузкой и без остановки приложений. В то время я часто писал о своем техническом опыте в блогах и по результатам задачи также был написан пост.

Через какое-то время я стал работать в другой компании, где серверы баз данных отдельно поддерживались компанией-подрядчиком. Однако, уже имея за плечами хороший опыт работы с PostgreSQL, я брал инициативу на себя и самостоятельно выполнял часть задач. В результате компания-подрядчик пригласила меня к себе, и вместо системного администратора я стал администратором баз данных. Теперь каждый мой рабочий день был связан с PostgreSQL.

Другой моей сильной стороной было хорошее знание ОС Linux. Эти знания стали очень полезными при дальнейшем погружении и изучении PostgreSQL. Тема мониторинга всегда вызывала у меня живой интерес, и мне всегда нравилось наблюдать за тем, как работают системы. Когда я стал администратором баз данных, мой интерес к наблюдениям сместился в сторону PostgreSQL, и я стал разбираться с тем, как отслеживать характеристики работы СУБД. Результатом этого стало создание различных инструментов, начиная от простых SQL-скриптов и заканчивая плагинами к системам мониторинга и консольными утилитами. И, конечно же, бесчисленное количество постов. Как правило, они не были связаны между собой и объединялись лишь общей темой мониторинга. Понимая недолговечность постов и их разрозненность в интернете, мне захотелось соединить их в один большой материал. Так пришла идея написать книгу.

За те полтора года, что я пишу книгу, я проанализировал свой опыт, провел множество экспериментов, обобщил и собрал в одном месте большое количество материала. Однако и PostgreSQL не стоит на месте, продолжает развиваться. В СУБД появляются новые средства для мониторинга, так, например, после очередного релиза мне пришлось дополнять уже написанные главы.

В результате получилась книга, которая объединяет в себе справочные материалы об инструментах, практические приемы их использования и способы интерпретации полученных данных. Все эти знания являются концентрированным опытом, полученным за многолетнюю практику эксплуатации PostgreSQL. Если вы научитесь мониторить PostgreSQL, это значительно улучшит ваше понимание его работы и происходящих внутри процессов, а также повысит навыки администрирования, поиска и устранения проблем и оптимизации производительности.

Специалистам, занятым эксплуатацией PostgreSQL, пойдет на пользу знание инструментов, представленных в СУБД, умение их использовать и понимание тонкостей их применения. Сегодня оптимизация производительности СУБД — это одна из важнейших задач эксплуатации больших систем, а знание и навыки мониторинга позволяют успешно ее решать.

В сети доступно огромное количество информации для тех, у кого есть достаточно времени и желания искать, фильтровать и понимать ее. Весьма приятно читать новые статьи и узнавать какие-то ранее неизвестные детали. Особенно приятно знакомиться с изменениями на commitfest.postgresql.org и читать коммит-сообщения, еще до основного релиза одним из первых узнавать о новшествах, которые появятся в СУБД. Однако тема мониторинга очень обширна, и только объем и глубина книжного формата дают мне возможность полностью объяснить мониторинг PostgreSQL, затрагивая связанные с ним темы, а также дать ссылки для дальнейшего изучения, если что-то вызовет интерес. Надеюсь, что в этой книге я достиг своей цели.

Об этой книге

Я написал эту книгу для того, чтобы доходчиво и на реальных примерах объяснить, как на практике устроен мониторинг PostgreSQL. Обычно официальная документация PostgreSQL по встроенным средствам наблюдения и мониторинга изложена довольно сухо, поэтому цель этой книги — объяснить все детали на примерах, которые будут понятны каждому администратору СУБД.

Кому следует прочесть эту книгу?

Эта книга создана для администраторов баз данных, системных администраторов, специалистов по надежности и тех, кто просто интересуется администрированием PostgreSQL. Книга призвана осветить все тонкости темы мониторинга. В блогах можно найти множество публикаций, но большинство из них детально рассматривают лишь отдельные аспекты. Эта книга всесторонне охватывает мониторинг PostgreSQL, подготавливая читателя и помогая ему разобраться в большом разнообразии средств, встроенных в СУБД. Цель практически любого мониторинга — это выявление аномалий, предупреждение аварийных ситуаций и прогнозирование поведения системы в будущем. Используя мониторинг, можно лучше понять, как работает система, и дальше с помощью корректировки конфигурации и оптимизации рабочей нагрузки добиться увеличения производительности. Поэтому каждый, кто интересуется оптимизацией производительности, обязательно извлечет из этой книги полезную информацию и не только сделает выводы для себя, но и, возможно, наметит план изменений в администрируемых системах и получит полезный результат.

Как устроена эта книга

Книга состоит из девяти глав.

- В главе 1 дается общее представление о том, что такое *статистика активности СУБД*, почему она так важна и является основой мониторинга PostgreSQL. Это вводная глава с теоретическими основами.
- В главе 2 разбирается статистика, которая описывает процессы и события внутри СУБД, вызванные выполнением рабочей нагрузки и обслуживанием клиентов.
- В главе 3 подробно рассматриваются клиентские сеансы и проводится анализ рабочей нагрузки, которая создается приложениями.

- В главе 4 статистика активности изложена с точки зрения взаимодействия с объектами СУБД и происходящих при этом событий.
- Глава 5 посвящена области общей памяти и вводу-выводу, как процессу перемещения данных между различными подсистемами.
- В главе 6 рассматриваются журнал предзаписи и связанная с ним статистика.
- В главе 7 говорится о репликации, возможных проблемах при ее использовании и инструментах для отслеживания ее состояния.
- Глава 8 посвящена процессу очистки баз данных, с которым сталкивался каждый администратор PostgreSQL.
- В главе 9 рассказывается об инструментах отслеживания хода выполнения продолжительных операций, которые требуется периодически запускать при эксплуатации СУБД.

После изучения этой книги читатели приобретут четкое представление о мониторинге PostgreSQL, об имеющихся для этого средствах в СУБД и о том, как использовать их наиболее эффективно. Также книга поможет разобраться в способах поиска и устранения проблем, связанных с эксплуатацией PostgreSQL.

На кого рассчитана книга

Книга рассчитана на читателя с уровнем подготовки выше начального. Чтобы не увеличивать объем книги, подробное описание некоторых второстепенных вещей опущено в надежде, что читатель уже знаком с ними или способен найти ответ в поисковой системе или у ChatGPT. Предполагается, что читатель имеет следующие знания или навыки:

- базовые сведения об общем устройстве СУБД, например, в общих чертах представляет, что означает каждая из букв в аббревиатуре ACID, и знает, чем транзакция отличается от запроса;
- основы использования языка SQL и умение без подглядывания в документацию составить простой SELECT-запрос с условиями, соединениями или подзапросами; умение читать и понимать «чужие» запросы;
- базовые знания конкретно о PostgreSQL и общее представление о таких основных компонентах этой СУБД, как буферный кеш (shared buffers), журнал упреждающей записи, репликация и т. п.; превосходно, если читатель знаком с книгой The Internals of PostgreSQL, доступной по адресу www.interdb.jp/pg/index.html;
- некоторое представление о системе мониторинга Prometheus[1], о том, что такое метрики и метки[2];

[1] prometheus.io/docs/introduction/overview
[2] prometheus.io/docs/concepts/data_model

- минимальные навыки составления запросов PromQL/MetricsQL[1], так как в книге помимо SQL-запросов будут использоваться и примеры запросов на языках PromQL и MetricsQL;
- базовые навыки использования Grafana; желательно уметь создавать простые панели с графиками на основе запросов PromQL/MetricsQL, уметь редактировать легенду, единицы измерения, параметры отображения, цвета и т. д.;
- базовые навыки использования ОС Linux, такие как работа в терминале, установка пакетов и запуск программ;
- минимальные навыки использования инструментов Docker и Docker Compose; желательно уметь запускать и останавливать контейнеры, подключаться к ним и проверять их состояние.

Примеры кода

В книге часто приводятся примеры запросов на языках SQL и PromQL/MetricsQL, а иногда встречаются команды, которые необходимо выполнить в shell-оболочке или psql-клиенте. Для того чтобы отделить примеры запросов от обычного текста, используется `моноширинный шрифт`, а команды **набраны жирным**. Названия конфигурационных параметров даны *курсивом*.

Остались вопросы?

Поищите ответы в следующих источниках:

- Основная страница официальной документации на русском языке, посвященная встроенным средствам мониторинга PostgreSQL, доступна по адресу www.postgrespro.ru/docs/postgresql/current/monitoring.
- Основная страница официальной документации на английском языке, посвященная встроенным средствам мониторинга PostgreSQL, доступна по адресу www.postgresql.org/docs/current/monitoring.html.
- Официальная wiki-страница, посвященная различным инструментам для мониторинга PostgreSQL, доступна по адресу wiki.postgresql.org/wiki/Monitoring.

[1] prometheus.io/docs/prometheus/latest/querying/basics

Отзывы и пожелания

Я буду рад отзывам читателей. Расскажите, что вы думаете об этой книге, — что понравилось или, может быть, не понравилось. Отзывы важны для меня и будут полезны при подготовке обновлений этой книги. Вы можете написать отзыв или пожелание по адресу github.com/lesovsky/postgresql-monitoring-book/issues. То же самое относится и к найденным опечаткам, запросам на исправление неточностей и т. п.

Нарушение авторских прав

Пиратство в интернете (намеренное и непреднамеренное) по-прежнему остается насущной проблемой. Если вы столкнетесь с незаконной публикацией этой книги, пожалуйста, пришлите мне ссылку на интернет-ресурс, чтобы я мог связаться с его владельцем. Ссылку на подозрительные материалы можно оставить по адресу github.com/lesovsky/postgresql-monitoring-book/issues.

Об авторе

Алексей Лесовский (Alexey Lesovsky) — профессиональный администратор баз данных, системный администратор, разработчик программного обеспечения, devops-инженер. Почти 20 лет он занимается задачами эксплуатации больших и сложных систем, проектирования и разработки программного обеспечения. Вы можете найти Алексея Лесовского на GitHub по адресу github.com/lesovsky.

С уважением!

Танюша, эта книга посвящается тебе, ты больше, чем жена, ты лучше всех! Мы с тобой отличная команда ;)

Роман Алексеевич, помни: задача детей — стать лучше, чем их родители. Быстрее, выше, сильнее. Ты красавчик, я горжусь тобой!

Маруся, ты моя жемчужинка, спасибо тебе за улыбки, задорный смех и хорошее настроение. Ты всегда в моем сердце.

Андрей Фефелов, ты для меня пример стойкости и крепости духа, большая честь быть твоим другом.

Посвящаю книгу всем родным и близким, семье и родителям — за поддержку и веру. Спасибо друзьям и коллегам по сообществу. Спасибо всем, кто ходил на мои доклады, смотрел меня на YouTube и задавал каверзные вопросы в кулуарах и комментариях. Спасибо организаторам конференций за возможность живого выступления в зале — это незабываемый опыт. Спасибо тем, кто читал мои блог-посты и просил писать больше и чаще. Я написал больше, чем просто пост. Спасибо тебе, читатель, это все для тебя.

Отдельная благодарность Егору Рогову за бесценную помощь в подготовке книги. Вначале было сложно, и в процессе согласования я испытал всю гамму чувств — от отрицания до принятия, но в итоге остался очень доволен результатом. Надеюсь, нам еще удастся поработать вместе.

Глава 1

Обзор статистики

В этой главе мы рассмотрим:

- внутреннее устройство PostgreSQL;
- важность статистики активности;
- особенности статистики активности.

В этой главе дается общее представление о статистике активности и о том, почему эта статистика так важна. На примере внутреннего устройства PostgreSQL объясняется необходимость статистики для понимания работы внутренних процессов в СУБД и эффективного наблюдения за ними. Статистика, которую мы будем рассматривать на протяжении всей книги, является фундаментом для построения большинства инструментов мониторинга PostgreSQL.

1.1. «Postgres — это сервис»

Первую главу я хотел бы начать так же, как обычно начинаю свои доклады про мониторинг PostgreSQL. СУБД PostgreSQL — это продукт с долгой историей. За все время существования в нее было добавлено множество функций, и, конечно, это отразилось на ее внутреннем устройстве, которое схематично изображено в моем вольном представлении на рис. 1.1. Из схемы видно, что СУБД состоит из множества компонентов. Мало того, эти компоненты связаны между собой и постоянно взаимодействуют друг с другом в процессе обработки данных.

Если все сильно упростить, то СУБД можно рассматривать как сервис, который предоставляет две основные услуги:

1. Надежное хранение данных. Пользователи и сервисы загружают в СУБД свои данные и задача СУБД — обеспечить их прием и сохранность.

2. Обеспечение доступа к данным. Пользователям и сервисам могут потребоваться данные и СУБД предоставляет язык запросов для работы с ними.

Основные услуги могут расширяться дополнительными:

- разнообразие типов данных и гибкая организация схемы доступа;
- быстрый доступ к данным за счет индексов;

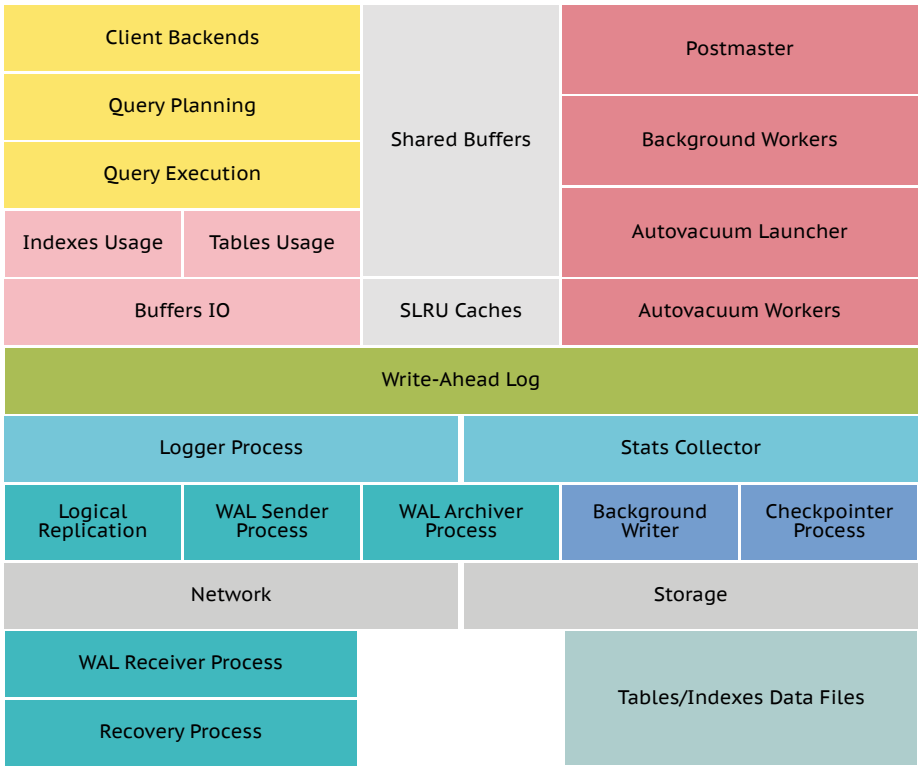


Рис. 1.1. Внутреннее устройство СУБД PostgreSQL

- конкурентный транзакционный доступ к данным с гарантиями атомарности, согласованности и изоляции (эти свойства входят в известную аббревиатуру ACID[1]).

Для реализации этих и многих других услуг СУБД опирается на различные подсистемы и механизмы.

1.2. Внутреннее устройство PostgreSQL

Давайте поверхностно, с минимально необходимым погружением в детали, рассмотрим основные подсистемы СУБД и работу внутренних процессов при выполнении таких регулярных задач, как:

- установка соединений и обслуживание сеансов;
- планирование и выполнение запросов;
- ввод-вывод при работе с данными;
- журналирование изменений;

[1] en.wikipedia.org/wiki/ACID

- репликация и архивирование журнала транзакций;
- фоновая синхронизация данных;
- журнал сообщений;
- автоочистка.

Установка соединений и работа сеансов

В своем обычном режиме СУБД работает как служба (программа-сервер) и ожидает подключений со стороны клиентов. Клиентом может быть как приложение, так и пользователь, подключающийся через клиентскую программу (psql, pgAdmin, DataGrip и др.). Для каждого клиента сервером СУБД создаются отдельные процессы операционной системы, которые в привычной для администраторов терминологии называются *бэкендами* (backend). В каждом таком процессе между клиентом и сервером СУБД устанавливается *сеанс* — двухсторонняя связь, позволяющая клиенту взаимодействовать с СУБД. В процессе создания сеанса СУБД выполняет аутентификацию клиента согласно указаниям в pg_hba.conf, инициализирует внутренние рабочие структуры и применяет различные настройки, влияющие на дальнейшую работу. Когда сеанс готов, клиент может отправлять серверу *команды*, включающие в себя как команды SQL, так и системные управляющие команды[1].

При эксплуатации СУБД важно иметь представление о подключенных клиентах, установленных сеансах и том, что происходит в этих сеансах. Поведение клиента и состояние сеанса в некоторых обстоятельствах могут негативно влиять на работу и производительность СУБД.

Запросы как базовая единица рабочей нагрузки

Запросы являются подмножеством команд и представляют собой базовую единицу рабочей нагрузки. Рабочую нагрузку можно представить как весь объем запросов, отправляемый всеми приложениями и выполняемый на стороне СУБД. Чтобы справляться с рабочей нагрузкой, серверу требуются ресурсы, такие как CPU, память, ввод-вывод и пространство на диске. В эксплуатации СУБД важно иметь точное представление о том, какие запросы исполняются СУБД и достаточно ли ресурсов для их эффективного выполнения. Статистика дает ответ на этот вопрос и предоставляет отправную точку для оптимизации запросов и производительности.

Планирование и выполнение запросов

Получив команду от клиента, сервер проверяет ее корректность. Чаще всего командой является SQL-запрос; в таком случае СУБД начинает его *планирование*. Планирование заключается в составлении оптимального плана для доступа к данным. Доступ и промежуточная обработка данных могут стоить по-разному (в смысле использования системных ресурсов), и задача

[1] www.postgrespro.ru/docs/postgresql/current/sql-commands

планирования сводится к составлению и выбору наиболее дешевого плана. План обычно принято представлять в виде графа, где каждый узел является вполне конкретной операцией над данными.

СУБД способна планировать параллельное выполнение некоторых операций, например чтение из таблицы или индекса. Если планом предполагается параллельное выполнение, для этого будут автоматически запущены *вспомогательные процессы* (background workers), которые возьмут на себя часть работы. После выполнения операции вспомогательные процессы также автоматически будут завершены.

Когда план выбран, сервер приступает к выполнению запроса согласно этому плану. Каждый узел плана — это конкретная операция над данными (например, над строками из таблиц). Когда запрос выполнен, результат возвращается клиенту. В качестве результата может выступать набор строк или тег команды (command tag), сообщающий об успешности выполнения. В случае запросов на изменение данных результат запроса должен быть записан в *журнале транзакций* (Write-Ahead Log, WAL). В зависимости от настроек СУБД это может происходить в синхронном или асинхронном режиме. В любом случае после выполнения запроса СУБД передает управление клиенту, и он может отправлять следующий запрос.

Ввод-вывод при выполнении запросов

При выполнении запросов практически вся работа с данными является буферизованной и происходит в памяти, выделенной процессам системой. Вся память, выделяемая СУБД, представляет собой набор сегментов, которые с точки зрения использования самой СУБД можно разделить на два типа областей памяти — локальные и общие.

Локальные области памяти — это сегменты, которые выделяются индивидуально для каждого процесса (в рамках сеанса), при этом процессы не имеют доступа к локальным сегментам друг друга. Среди таких сегментов можно выделить следующие характерные области:

- рабочая память процессов (см. параметр *work_mem*) — выделяется при необходимости для оперативного размещения данных при выполнении некоторых промежуточных операций в запросе (сортировка, исключение дубликатов (`DISTINCT`), соединение таблиц по алгоритмам merge join и hash join и др.). Если для выполнения операции рабочей памяти становится недостаточно, на диске создаются временные файлы, которые удаляются после завершения операции;
- временные буферы (см. параметр *temp_buffers*) — используются для работы с данными временных таблиц (temporary tables), которые существуют в рамках сеанса или вообще транзакции. Такие таблицы являются нежурналируемыми и часто применяются для сохранения промежуточных результатов;
- рабочая память для операций обслуживания (см. параметр *maintenance_work_mem*) — выделяется для таких операций, как `VACUUM`, `CREATE INDEX`, `REINDEX` и др. Фоновые процессы автоочистки используют собственную отдельную рабочую память (см. параметр *autovacuum_work_mem*).

Параметры конфигурации СУБД

В этой и следующих главах часто будут встречаться параметры конфигурации. Полный список всех параметров доступен в документации по адресу: postgrespro.ru/docs/postgresql/current/runtime-config.

Общая память (shared memory) — это один и, как правило, достаточно большой сегмент памяти, который выделяется один раз при запуске СУБД. Доступ к общей памяти имеют все процессы СУБД. В области общей памяти размещаются:

- буферный кеш для страниц таблиц и индексов;
- буферы WAL-журнала;
- данные журнала, который хранит состояние всех транзакций (commit log) и используется правилами видимости многоверсионного управления конкурентным доступом;
- служебные структуры для управления доступом (тяжелые и легкие блокировки, семафоры и т. п.);
- служебные структуры транзакций, необходимые для точек сохранений (savepoint)[1] и двухфазного подтверждения (two-phase commit)[2], при котором становится возможным успешное подтверждение даже в случае аварийного сбоя, предшествующего подтверждению;
- служебные структуры фоновых служб;
- серверная статистика;
- и многое другое.

Буферный кеш

Здесь и далее под термином «буферный кеш» мы будем называть именно общий буферный кеш, известный как shared buffers. В контексте повествования, где упоминаются локальные буферные кеши, это отмечено отдельно.

Кроме основной общей памяти, также могут создаваться и небольшие сегменты общей памяти для взаимодействия вспомогательных процессов в случае параллельного выполнения некоторых операций. Существование таких сегментов ограничено временем жизни вспомогательных процессов, а размер зависит от объемов передаваемых данных.

Работа с пользовательскими данными происходит примерно одинаково независимо от типа области данных, локального или общего. СУБД использует страничную организацию данных со *страницами* (page) фиксированного размера (8 КБ по умолчанию), что определяет минимальный объем ввода-вывода. В случае операций чтения или изменения данных процесс проверяет наличие необходимых данных в буфере, неважно, в локальном или общем. Если они

[1] postgrespro.ru/docs/postgrespro/current/sql-savepoint
[2] postgrespro.ru/docs/postgrespro/current/sql-prepare-transaction

есть, это считается успешной попыткой доступа (hit). Если данных в кеше не нашлось, то для продолжения работы их требуется загрузить с диска в кеш; это считается промахом кеша (miss, или read). Если к данным обращались ранее, они могут оказаться в страничном кеше операционной системы (page cache), откуда взять их будет быстрее, чем прочитать из основного хранилища. В случае обновления (`INSERT`, `UPDATE`, `DELETE`) страницы изменяются в кеше; такие страницы считаются грязными (dirtied), что указывает на необходимость их синхронизации с файлом данных в основном хранилище. Обычно синхронизацией страниц занимаются два фоновых процесса — checkpointer и background writer, но в некоторых случаях это может делать и клиентский процесс. Так бывает, когда процессу требуются страницы, которых нет в кеше, и, чтобы прочитать их с диска, процессу нужны свободные буферы, которых тоже нет. Чтобы освободить буфер под целевую страницу, процесс начинает поиск и вытеснение страницы, к которой давно не было обращений. Найденная страница может оказаться грязной, и тогда процесс сначала синхронизирует ее (written) и только потом освобождает буфер. В общем, получается не самая дешевая операция, особенно если приходится делать это часто.

Всего получаются четыре возможных случая:

- hit — наиболее быстрая и дешевая, но все-таки не бесплатная операция: страница находится в общей памяти;
- miss, или read — более дорогая операция: страницу необходимо прочитать в лучшем случае из страничного кеша, а в худшем — с диска;
- dirtied — также дорогая операция: страницу впоследствии нужно будет синхронизировать с основным хранилищем;
- written — еще одна дорогая операция: вместо отложенной фоновой синхронизации страница должна быть синхронизирована немедленно клиентским процессом (который выполняет запрос).

Описанная выше модель работы одинакова как для общего, так и для локального кеша. При работе с обычными и временными таблицами страницы в кеше всегда ассоциированы с файлом таблицы на диске. В случае использования рабочей памяти такого файла нет ровно до тех пор, пока не будет превышено ее ограничение (параметр *work_mem*). В этом случае создается временный файл, который удаляется после завершения операции. При нехватке рабочей памяти для операций обслуживания (автоочистка, создание индексов) СУБД повторно использует уже выделенную память без создания дополнительных файлов.

С точки зрения производительности очень хорошо, когда бóльшая часть данных находится в памяти и на любое обращение страницу можно найти в кеше, и хуже, когда приходится регулярно читать данные из основного хранилища или же вытеснять грязные страницы для освобождения буферов под новые страницы.

Журнал упреждающей записи — Write-Ahead Log

Перед тем как вернуть результат запроса, направленного на изменение данных, СУБД записывает эти изменения в *журнал предзаписи* (Write-Ahead Log). Еще можно встретить термин

журнал транзакций, однако чаще всего просто используется аббревиатура WAL. Запись в журнал требуется для обеспечения надежности и сохранения порядка всех изменений над данными и возможности восстановления после аварийного завершения СУБД. В таком случае при последующем запуске СУБД, используя журнал, воспроизведет последовательность изменений над теми данными, которые не были сброшены из буферного кеша в основное хранилище. Производительность СУБД зависит в том числе и от скорости работы с WAL-журналом, которую можно узнать с помощью статистики.

Журнал сообщений СУБД

После того как команда завершилась, клиенту передается результат выполнения, будь то набор строк, тег или вообще ошибка. Здесь может выполняться *протоколирование*, то есть сохранение служебной информации о выполнении команды. Эта информация может включать в себя данные о клиенте, текст и параметры запроса и т. п. Протоколирование дополняет статистику и также является важным источником информации о работе СУБД. Далее сервер готов к получению и выполнению следующей команды.

Репликация изменений

Помимо основной работы с клиентами, СУБД выполняет ряд служебных задач. Для этого существуют отдельные процессы, которые работают в фоновом режиме. Одной из таких задач является физическая *репликация*, которая построена на основе журнала упреждающей записи. Все изменения данных, которые попадают в журнал, читаются процессом walsender и по протоколу репликации передаются на реплики. На репликах процессы walreceiver принимают данные журнала и сохраняют содержимое на диск. Другой фоновый процесс — startup — читает полученные данные журнала и воспроизводит последовательность изменений на локальной копии данных. После этого изменения, пришедшие с основного сервера, становятся видимыми для запросов, выполняющихся на реплике.

Кроме физической репликации, PostgreSQL поддерживает и логическую. Физическая репликация предполагает передачу и применение всех изменений, в то время как логическая позволяет выборочно передавать данные на уровне отдельных таблиц и даже операций. Использование репликации может преследовать разные цели, например распределение рабочей нагрузки или отказоустойчивость и быстрое переключение при сбоях. Поэтому важно отслеживать состояние всех узлов, и в этом снова нам помогает статистика.

Архивирование журнала предзаписи

Похожей на репликацию является задача архивирования WAL-журнала. С точки зрения операционной системы WAL-журнал — это всего лишь последовательность файлов, называемых WAL-сегментами. Как только запись в очередной WAL-сегмент завершена, специальный процесс archiver может поместить этот сегмент в *архив*. Под архивом подразумевается отдельное

от СУБД хранилище, которое используется для задач резервного копирования и аварийного восстановления.

Фоновая синхронизация данных

Есть и другие фоновые задачи. Одна из них — это синхронизация измененных данных из буферного кеша с основным хранилищем. Для этого задействованы два процесса: background writer и checkpointer. Первый в непрерывном цикле ищет грязные страницы и записывает их на диск. Второй, checkpointer, выполняет *контрольные точки*. Контрольная точка — это отметка в WAL-журнале, которая указывает на то, что все изменения до этой отметки уже записаны в надежное хранилище. При выполнении контрольной точки процесс checkpointer записывает все грязные данные в отличие от background writer, который делает это выборочно. Другими словами, checkpointer, так же как и background writer, синхронизирует изменения из буферного кеша с диском, но дополнительно ставит особую отметку в WAL-журнале о том, что синхронизированы все грязные данные.

Поскольку журнал предзаписи — это история всех изменений, возникает вопрос: как долго следует хранить эту историю? Ответ дает процесс checkpointer: установка контрольной точки означает, что все предыдущие изменения надежно записаны в основное хранилище и все сегменты журнала, предшествующие этой контрольной точке, могут быть удалены. Таким образом, объем журнала предзаписи сохраняется более или менее постоянным и WAL-сегменты не накапливаются.

Автоочистка

Еще одной регулярной фоновой задачей является очистка таблиц и индексов от устаревших версий строк, так называемая *автоочистка* (autovacuum). Необходимость автоочистки является следствием реализации операций обновления данных и конкурентного доступа к ним. При изменении данных и конкурентной работе нескольких клиентов в таблицах могут возникать несколько версий одних и тех же строк. Со временем самые старые версии строки становятся неактуальными и могут быть удалены. Их удалением и занимается процесс автоочистки. Фоновый процесс autovacuum launcher с определенной периодичностью (см. *autovacuum_naptime*) запускает рабочие процессы autovacuum worker, которые выполняют очистку. Дополнительно рабочие процессы могут собирать статистику для планировщика о качественных и количественных характеристиках данных в таблицах. Эта статистика нужна для построения планов запросов. Неэффективная работа автоочистки в перспективе негативно влияет на производительность, и с помощью статистики активности можно наблюдать за работой не только очистки, но и других фоновых процессов.

Подводя итог, можно сказать, что в PostgreSQL есть много сложных процессов, которые влияют друг на друга и связаны между собой. При возникновении проблем администратору БД нужна информация о том, как работают те или иные процессы. Для отслеживания различных событий СУБД имеет внутренние инструменты, которые ведут учет и хранят статистику в специальной

области в общей памяти. До версии PostgreSQL 15 получением и хранением статистики занимался отдельный процесс stats collector. Статистика передавалась ему бэкендами по протоколу UDP, и затем она сохранялась на диск. С версии 15 статистика сохраняется сразу в общей памяти[1] и необходимость в отдельном процессе отпала. На протяжении всей книги мы будем использовать эту статистическую информацию для понимания работы СУБД, идентификации и устранения проблем.

1.3. Интерфейс статистики

Подсистема сбора статистики накапливает различную информацию, которая может понадобиться при устранении проблем. Чтобы воспользоваться этой информацией, достаточно иметь под рукой любой SQL-клиент, который поддерживает подключение к PostgreSQL. В качестве примера можно привести psql — это официальный клиент для PostgreSQL. Также можно использовать клиенты, которые являются частью продвинутых инструментов с расширенным набором функций для работы с СУБД. Примерами таких инструментов являются pgAdmin, DBeaver или DataGrip.

Вся статистика представлена в виде служебных данных СУБД, и для доступа к ней не нужно предпринимать дополнительных действий. Получить статистику можно с помощью служебных *функций*. Однако пользоваться ими не всегда удобно, поэтому получение статистики из функций организовано через системные *представления* (view), к которым можно выполнять SQL-запросы. Такие представления аналогичны таблицам, но не имеют физического слоя хранения данных, и при работе с ними пользователю доступны почти все (кроме записи) возможности языка SQL: соединения, агрегации, оконные функции, подзапросы и т. д.

Со временем при работе со статистикой у пользователя могут появиться «любимые» запросы, которые расширены дополнительной логикой, — в одном запросе может использоваться несколько представлений, возможно, в сочетании с функциями, результаты могут преобразовываться и форматироваться. С помощью SQL такие запросы можно оформить в отдельные представления. В дальнейшем это избавляет от необходимости искать или вспоминать текст запроса, писать его заново: достаточно лишь сделать запрос к представлению и получить готовый результат.

Одной из сильных сторон PostgreSQL является возможность создания *расширений* (extensions), специальных модулей, устанавливаемых в СУБД с целью добавления новой функциональности, без необходимости изменения основного исходного кода системы. Используя этот механизм, разработчики могут создать недостающую функциональность, а пользователи — относительно легко подключить ее и использовать. Часть статистики распространяется в виде расширений. Можно подключить эти расширения и дополнить статистику новыми данными. В некоторых главах мы будем также рассматривать статистику, поставляемую через расширения, что будет отмечено отдельно.

[1] git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=5891c7a8ed8f2d3d577e7eea34dacff12d7b6bbd

1.4. Статистика как отправная точка инструментов мониторинга

Статистика, поставляемая в виде функций и представлений, является прочным фундаментом, на основе которого можно строить более продвинутые инструменты для мониторинга и наблюдения за СУБД. Самый простой вариант — это такие же представления с расширенной статистикой, отформатированные для более удобного восприятия человеком. Примерами могут служить различные репозитории с SQL-скриптами[1]. Более сложные варианты могут встраивать в себя клиентское подключение к PostgreSQL и обеспечивать сбор, обработку, передачу и предоставление информации. По сути, это реализовано практически во всех агентах мониторинга и утилитах для администраторов БД. Задача таких агентов сводится к подключению к СУБД, снятию информации с помощью заготовленных запросов и отправке ее в систему мониторинга или отображению в собственном пользовательском интерфейсе. Пользователю лишь остается запустить агента и указать реквизиты подключения к СУБД (и, возможно, дополнительные параметры работы, специфичные для агента). Примерами таких инструментов являются pgAdmin, pgcenter, pg_activity, pgstats[2].

1.5. Особенности статистики

Простота интерфейса статистики влечет за собой и некоторые особенности. Одна из основных — это порог вхождения. Для использования статистики необходимо знание языка SQL на базовом уровне, чтобы делать простейшие `SELECT`-запросы. Для более сложных приемов, например для извлечения статистики из нескольких представлений, потребуется использование соединений, подзапросов и более продвинутых возможностей SQL. Другая особенность — это большой набор представлений и функций, что может вызвать сложности в запоминании имен представлений и их содержимого. Особенно сложно вспоминать имена и названия в аварийной ситуации, когда счет может идти на секунды. Также работа со статистикой усложняется тем, что пользователь должен работать в командном режиме: следует написать и отправить запрос, получить и проанализировать результат, при необходимости повторить. Обычно этот недостаток компенсируют организацией репозитория с набором скриптов, которые всегда под рукой или заранее установлены в БД.

Помимо psql PostgreSQL не предоставляет собственного продвинутого инструмента для работы со статистикой и представлениями. Его отсутствие является причиной появления множества сторонних инструментов работы со статистикой для администраторов БД. У каждого автора есть свое видение того, как следует отображать статистику, что и приводит к большому разнообразию среди инструментов и функциональности.

[1] github.com/dataegret/pg-utils/tree/master/sql
[2] wiki.postgresql.org/wiki/Monitoring

Отдельной особенностью и даже недостатком является отсутствие статистики по журналам сообщений. Чтобы работать с журналами, пользователю необходимо иметь к ним доступ. Чаще всего это файлы в операционной системе, где запущена СУБД. Некоторые расширения пытаются решить эту проблему, но, к сожалению, они не имеют официальной поддержки разработчиков PostgreSQL, что уменьшает их распространенность и усложняет использование.

1.6. Тестовое окружение

В приложении приведены описание и инструкция по развертыванию тестового окружения. Тестовое окружение используется на протяжении всей книги для практических задач и примеров. На основе тестового окружения будет демонстрироваться использование статистики и будут объясняться важные особенности. Тестовое окружение также будет полезно любознательным читателям, желающим поэкспериментировать, усвоить материал книги на практике и выйти за рамки основного повествования. При желании установку текстового окружения можно пропустить без значительного ущерба для понимания материала.

Резюме

- PostgreSQL — это сложное по внутреннему устройству ПО, состоящее из разных подсистем.
- При эксплуатации всегда есть риск возникновения различных проблем.
- Статистика активности позволяет отслеживать работу СУБД и дает администратору возможность упреждать проблемы.
- Статистика доступна с помощью SQL-функций и представлений.
- Для работы со статистикой потребуется знание языка SQL.
- Минималистичный интерфейс статистики является причиной многообразия инструментов для администраторов БД.
- Для рассмотрения практических аспектов статистики можно воспользоваться тестовым окружением.

Глава 2

Статистика активности

В этой главе мы рассмотрим:

- клиентскую активность;
- транзакционную активность;
- представления `pg_stat_activity`, `pg_locks` и `pg_stat_database`;
- основные состояния клиентских сеансов;
- ожидание и блокировки;
- бездействующие (idle) транзакции;
- как определять длительность запросов и транзакций;
- как определять время ожидания блокировок;
- дерево блокировок.

Статистика активности является ключевым элементом, необходимым для понимания происходящего в СУБД. С точки зрения операционной системы СУБД выглядит как черный ящик. Операционная система ничего не знает о таких внутренних сущностях СУБД, как пользователи, базы, сеансы, транзакции и т. п. Статистика активности позволяет заглянуть внутрь этого ящика и лучше понять происходящее внутри. Активность подразумевает любые процессы в СУБД, вызванные как внутренним взаимодействием подсистем, так и обслуживанием внешних клиентов. В этой главе мы подробно рассмотрим клиентские сеансы и все, что с ними связано: взаимодействие клиента и сервера, установка сеанса, выполнение запросов и транзакций, основные состояния сеанса и негативные сценарии, которые могут возникать во время сеанса, такие как бездействующие транзакции и блокировки. Умение отслеживать процессы, протекающие в СУБД, — это важный навык, необходимый для успешной эксплуатации приложений и СУБД. Для этого мы рассмотрим основные источники статистики и примеры ее использования.

2.1. Ключ к пониманию происходящего в СУБД

Я люблю говорить, что СУБД — это *сервис*. СУБД часто воспринимается как нечто большое и сложно устроенное внутри, но можно представить СУБД как небольшой и легко развертываемый микросервис (столь привычный веб-разработчикам). С этой точки зрения главная задача СУБД — принять и обработать запрос от клиента. Внутреннее взаимодействие сложных

компонентов можно считать второстепенным, так как оно не предполагает прямых действий со стороны клиента. В таком упрощенном случае есть лишь клиент и сервер. В качестве клиента выступает программа: это может быть приложение на Go, Python или Ruby, задание от Airflow или Celery или любимая IDE разработчика. В общем, что угодно, что может подключаться к СУБД и общаться с ней по ее протоколу. Сервером выступает СУБД, выполняющая команды клиента. Активность, создаваемая клиентом, формирует рабочую нагрузку. Объем рабочей нагрузки, с которой может справиться СУБД, определяет пропускную способность и, как следствие, общую производительность. Активность в СУБД можно измерить и проанализировать и в результате выявить аномалии, устранив которые можно увеличить производительность СУБД и приложений.

При анализе активности с точки зрения администрирования важно иметь количественную и качественную информацию о подключенных клиентах, например:

- сколько установлено сеансов и откуда они установлены;
- от имени каких пользователей и к каким базам установлены сеансы;
- в каких состояниях находятся сеансы;
- как долго сеансы находятся в этих состояниях;
- сколько выполняется запросов, от каких пользователей и в каких базах;
- как долго выполняются запросы;
- какие конкретно запросы выполняются в сеансах.

Вопросы к тому, что происходит в СУБД, могут быть самыми разными, в зависимости от задач администратора, его осведомленности и гипотез, выдвинутых в процессе поиска и устранения проблем. Вопросы могут затрагивать не только обработку запросов клиентов, но и работу фоновых служб. Ответы, полученные с помощью статистики активности, позволяют устранить проблемы или оптимизировать работу приложений для достижения более надежной работы и большей производительности.

2.2. Взаимодействие клиента и сервера

Для более полного понимания того, что представляет собой активность, давайте рассмотрим, как взаимодействуют между собой клиенты и СУБД. Взаимодействие строится по классической схеме «клиент — сервер». Сервер работает постоянно в фоновом режиме и ожидает подключений от клиентов. Клиент, будь то приложение или пользователь, подключается к серверу и после успешного подключения, следуя внутренней логике или желанию пользователя, формирует и отправляет команды серверу, ожидает их выполнения, получает ответ и обрабатывает его.

Со стороны PostgreSQL сервером выступает специальный процесс, который исторически принято называть postmaster. В задачи процесса входит прослушивание интерфейсов на предмет клиентских подключений, создание сеансов и запуск фоновых процессов для обслуживания

баз данных. При подключении клиента postmaster создает новый, дочерний процесс, внутри которого выполняются необходимая настройка и подготовка к сеансу. Когда сеанс готов, клиент может начинать отправку запросов.

Режимы работы

В зависимости от реализации клиента сеанс может подразумевать синхронный и асинхронный режимы работы. Большинство реализаций используют синхронный режим: клиент отправляет команду, ждет результат его выполнения и в это время не может отправлять другие команды. Возможен и асинхронный режим работы, но его реализация зависит от драйвера. Например, такая функциональность имеется в штатном драйвере libpq[1].

Ниже показаны процессы сервера СУБД, выведенные утилитой ps, с точки зрения операционной системы.

```
$ ps f -u postgres -o pid,cmd
    PID CMD
4150200 /usr/lib/postgresql/15/bin/postgres -c config_file=/etc/postgresql/15/main/postgresql.conf
4150206  \_ postgres: 15/main: logger
3979107  \_ postgres: 15/main: checkpointer
3979108  \_ postgres: 15/main: background writer
3979109  \_ postgres: 15/main: walwriter
3979110  \_ postgres: 15/main: autovacuum launcher
3979111  \_ postgres: 15/main: archiver last was 000000010000000400000096
3979112  \_ postgres: 15/main: logical replication launcher
1389347  \_ postgres: 15/main: walsender postgres 127.0.0.1(39540) streaming 4/97BB49B0
1865559  \_ postgres: 15/main: postgres pgbench [local] idle
1865560  \_ postgres: 15/main: postgres pgbench [local] SELECT
1865561  \_ postgres: 15/main: postgres pgbench [local] idle
1865562  \_ postgres: 15/main: postgres pgbench [local] SELECT
1865563  \_ postgres: 15/main: postgres pgbench [local] idle
1865564  \_ postgres: 15/main: postgres pgbench [local] SELECT
1865565  \_ postgres: 15/main: postgres pgbench [local] SELECT
1865566  \_ postgres: 15/main: postgres pgbench [local] SELECT
1865567  \_ postgres: 15/main: postgres pgbench [local] SELECT
1865568  \_ postgres: 15/main: postgres pgbench [local] idle
1865571  \_ postgres: 15/main: postgres pgbench [local] UPDATE
1865572  \_ postgres: 15/main: postgres pgbench [local] COMMIT
```

Процессы выведены в виде дерева «родитель — потомок». Главным из них является postmaster, который приходится родителем всем остальным процессам. Среди процессов-потомков есть процессы фоновых служб и процессы клиентских соединений, так называемые *бэкенды* (backend). Для наблюдения за работой СУБД со стороны операционной системы хорошо подходят такие утилиты, как top, htop и atop. С их помощью можно наблюдать за тем, как процессы операционной системы используют системные и операционные ресурсы, такие как

[1] postgrespro.ru/docs/postgresql/current/libpq-async

CPU, память, дисковый и сетевой ввод-вывод, пространство на диске. Стоит обратить внимание и на утилиты vmstat, dstat, nicstat и pidstat, iostat, входящие в состав пакета sysstat, — эти утилиты также могут быть полезны в оценке использования системных ресурсов.

Во время сеанса клиент общается с сервером посредством команд. SQL-запросы являются частным случаем таких команд и часто могут организовываться в *транзакции*. Транзакция представляет собой последовательность из нескольких запросов, которая в целом должна восприниматься как *одна* логическая операция. Начало транзакции объявляется командой `BEGIN`, для завершения могут использоваться команды `COMMIT` (`END`) или `ROLLBACK`. Транзакции являются важным механизмом СУБД, обеспечивающим возможность конкурентной работы множества клиентов. Транзакции обладают свойствами *атомарности*, *изоляции* и *согласованности*. Атомарность определяет, что в результате транзакции все ее операции выполняются вместе либо не выполняются совсем. В случае ошибки или отката транзакции результат всех операций отменяется до состояния, которое предшествовало началу транзакции. Здесь же проявляется и свойство согласованности, которое устанавливает, что атомарно выполненная (или отмененная) транзакция сохраняет базу данных в непротиворечивом, согласованном состоянии. Отдельные запросы (выполняемые без объявления блока `BEGIN` и `END`) на самом деле также являются транзакциями, но состоящими из одной команды. Свойство изоляции заключается в том, что результаты транзакции не видны другим, соседним транзакциям, до тех пор пока эта транзакция не будет зафиксирована, и то в зависимости от используемого уровня изоляции. Для более полного понимания этого вопроса следует ознакомиться с презентацией Брюса Момджяна MVCC Unmasked[1].

Несмотря на свойство изоляции, нельзя думать, что транзакции полностью независимы друг от друга. Возможность конкурентной работы подразумевает вероятность одновременного доступа к одним и тем же данным (строкам в таблице) со стороны нескольких транзакций. В случае операций чтения все просто, поскольку конкурентное чтение не вызывает конфликтов, почти[2]. С записью все становится чуть сложнее: конкурентная запись может вызывать конфликты, и такие операции должны быть *сериализованы*, то есть выстроены в строгую последовательность. Для сериализации доступа к объектам БД используются *блокировки*. Механизм блокировок позволяет ограничивать или запрещать одновременный доступ к ресурсу. Работа блокировок прозрачна для пользователя и в большинстве случаев не требует от него явных действий. Однако возможны ситуации, когда одновременно несколько клиентов пытаются установить несовместимую блокировку на один и тот же ресурс. В таком случае только один клиент сможет установить блокировку, а остальные образуют очередь и будут вынуждены ждать, когда блокировка будет снята.

В качестве промежуточного итога можно сделать вывод, что природа конкурентного доступа, свойства транзакций, механизм блокировок и некоторое стечение обстоятельств в совокупности могут приводить к ситуациям с негативными последствиями для СУБД и приложений. В зависимости от важности эксплуатируемой БД такие ситуации могут расцениваться как аварийные, так как могут привести к снижению производительности или к остановке запросов.

[1] momjian.us/main/writings/pgsql/mvcc.pdf
[2] postgrespro.ru/docs/postgresql/current/sql-select#SQL-FOR-UPDATE-SHARE

Одной из задач администратора БД является отслеживание, выяснение причин и предотвращение таких ситуаций. Далее в этой главе мы рассмотрим эти ситуации более подробно.

2.3. Источники информации об активности

Мы начнем с представлений pg_stat_activity, pg_locks и pg_stat_database. С исторической точки зрения возможности отображения активности развивались постепенно и со временем пришли к тому виду, в котором они есть на данный момент. В ранних версиях эти три представления содержали информацию, которая тематически не была связана между собой. Представления pg_stat_activity и pg_locks могли дополнять друг друга, а pg_stat_database не содержала той статистики, что будет рассматриваться далее. Сейчас все три представления содержат различную информацию, которую объединяет одна тема — определение активности. Все три представления помогают составить общую картину происходящего в СУБД:

- pg_stat_activity — статистика о процессах, подключенных клиентах и фоновых службах;
- pg_locks — статистика о блокировках, удерживаемых или ожидаемых активными процессами;
- pg_stat_database — кумулятивная статистика о клиентских сеансах в контексте отдельных баз данных.

Соглашение об именовании

Здесь и далее в тексте при указании представлений и их полей будет использоваться нотация *представление*.*поле*. Например, pg_stat_activity.pid указывает на поле pid в представлении pg_stat_activity.

Представление pg_stat_activity

Весь мой практический опыт поиска и устранения проблем говорит о том, что при возникновении самых разных подозрений на проблемы в работе СУБД pg_stat_activity — главный источник первичной информации. Если с точки зрения операционной системы администратор может увидеть только процессы, то pg_stat_activity помогает администратору заглянуть внутрь СУБД и посмотреть на работу этих же процессов, но с точки зрения самой PostgreSQL. В представлении содержится информация обо всех процессах СУБД, будь то клиентские процессы или фоновые службы, и каждая строка описывает отдельный серверный процесс. В ранних версиях PostgreSQL pg_stat_activity содержало информацию преимущественно о клиентских процессах. В следующих версиях была добавлена информация о фоновых службах. При этом клиентские процессы и фоновые службы отличаются характером работы с СУБД и часть информации, которая присуща клиентским процессам, для фоновых служб попросту отсутствует.

Но недостаток информации по фоновым процессам компенсируется другими представлениями, и pg_stat_activity остается наиболее ценным источником сведений о внутренней активности СУБД.

Посмотреть полное описание этого представления можно с помощью метакоманды \d+ pg_stat_activity.

Метакоманды

Клиент psql содержит набор метакоманд, полезных для получения справочной информации об объектах БД, как пользовательских, так и системных. Все метакоманды начинаются с символа обратной косой черты \, например \d. Для получения справки и списка всех метакоманд используйте метакоманду \?.

Ниже приведено сокращенное описание представления.

```
# \d pg_stat_activity
                    View "pg_catalog.pg_stat_activity"
      Column      |           Type           | Collation | Nullable | Default
------------------+--------------------------+-----------+----------+---------
 datid            | oid                      |           |          |
 datname          | name                     |           |          |
 pid              | integer                  |           |          |
 leader_pid       | integer                  |           |          |
 usesysid         | oid                      |           |          |
 usename          | name                     |           |          |
 application_name | text                     |           |          |
 client_addr      | inet                     |           |          |
 client_hostname  | text                     |           |          |
 client_port      | integer                  |           |          |
 backend_start    | timestamp with time zone |           |          |
 xact_start       | timestamp with time zone |           |          |
 query_start      | timestamp with time zone |           |          |
 state_change     | timestamp with time zone |           |          |
 wait_event_type  | text                     |           |          |
 wait_event       | text                     |           |          |
 state            | text                     |           |          |
 backend_xid      | xid                      |           |          |
 backend_xmin     | xid                      |           |          |
 query_id         | bigint                   |           |          |
 query            | text                     |           |          |
 backend_type     | text                     |           |          |
```

Вывод статистики в pg_stat_activity представляет собой таблицу на основе функции pg_stat_get_activity, где каждая строка описывает конкретный серверный процесс.

Давайте рассмотрим, какую полезную информацию можно получить из представления.

Информация о клиенте. Основные реквизиты соединения:

- client_addr — сетевой адрес, с которого выполнено подключение. Если соединение установлено через UNIX-сокет, значение будет отсутствовать (NULL);
- client_port — номер порта в удаленной системе, с которого установлено соединение;
- usename — имя пользователя, используемое при подключении;
- datname — имя базы данных, к которой выполнено подключение.

Для лучшей идентификации клиент при подключении может обозначить себя через отдельный идентификатор *application_name*. Это удобный способ отличать приложения в случаях, когда они подключены с одинаковыми реквизитами и с одного адреса. Поле backend_type позволяет отличать клиентские процессы от фоновых служб. В ранних версиях СУБД этого поля не было, и для определения клиентских подключений приходилось прибегать к различным уловкам — например, указывать в SQL-запросе дополнительное условие datname IS NOT NULL.

```
# SELECT
  row_number() OVER (ORDER BY pid) AS n,
  pid, backend_type, client_addr, application_name, usename, datname
FROM pg_stat_activity
LIMIT 15;
 n  |  pid   |         backend_type         | client_addr  | application_name | usename  | datname
----+--------+------------------------------+--------------+------------------+----------+----------
  1 |     65 | checkpointer                 |              |                  |          |
  2 |     66 | background writer            |              |                  |          |
  3 |     67 | walwriter                    |              |                  |          |
  4 |     68 | autovacuum launcher          |              |                  |          |
  5 |     69 | archiver                     |              |                  |          |
  6 |     71 | logical replication launcher |              |                  | postgres |
  7 |    117 | walsender                    | 192.168.64.5 | walreceiver      | replica  |
  8 | 209017 | client backend               |              | psql             | postgres | postgres
  9 | 224802 | client backend               |              | psql             | postgres | postgres
 10 | 230568 | client backend               | 192.168.64.9 | pgbench          | pgbench  | pgbench
 11 | 231751 | client backend               | 192.168.64.4 | pgbench          | maru     | pgbench
 12 | 231752 | client backend               | 192.168.64.4 | pgbench          | maru     | pgbench
 13 | 231753 | client backend               | 192.168.64.4 | pgbench          | maru     | pgbench
 14 | 231754 | client backend               | 192.168.64.4 | pgbench          | maru     | pgbench
 15 | 231755 | client backend               | 192.168.64.4 | pgbench          | maru     | pgbench
```

Этот пример наглядно показывает различия между фоновыми службами (строки с 1-й по 7-ю) и клиентскими процессами (строки с 8-й по 15-ю). Во-первых, отличить их можно по значению поля backend_type. Во-вторых, для фоновых служб могут отсутствовать значения других полей, например client_addr, application_name, usename и datname, поскольку фоновые процессы могут запускаться локально или не устанавливать соединений к БД.

Продолжительность сеанса, транзакции, запроса. В представлении есть информация, относящаяся ко времени начала сеанса и той активности, что происходит в сеансе:

- backend_start — время начала сеанса, то есть момент подключения клиента к СУБД;

- xact_start — время начала текущей транзакции;
- query_start — время начала текущего запроса — на случай, если это не единственный запрос в транзакции.

С помощью этих полей можно выявлять длительные запросы и транзакции, нехарактерные для конкретной рабочей нагрузки.

Состояние сеанса. Сеанс в течение своего жизненного цикла может находиться в разных *состояниях*. По состоянию можно определить, все ли в порядке с сеансом, и принять меры, если с ним что-то не так. На состояние сеанса указывает поле state. Также в поле state_change доступно время перехода в текущее состояние, по которому можно определить его продолжительность. Далее по этим полям мы будем отслеживать потенциально опасную активность в БД и ее продолжительность.

Ожидания и блокировки. Внутренняя механика СУБД предполагает использование разных механизмов синхронизации; хорошим примером являются блокировки, которые используются для сериализации доступа к объектам. Следствием использования таких механизмов является риск возникновения ожиданий на разных участках выполнения кода. Для отслеживания ожиданий СУБД предоставляет поля wait_event и wait_event_type, которые детально идентифицируют место, где возникло ожидание. С помощью полей state, wait_event_type и wait_event можно определять узкие места в работе СУБД, где происходит ожидание вместо выполнения полезной работы.

Идентификатор и текст запроса. В поле query указаны тексты выполняющихся в СУБД запросов. Поле ограничено размером буфера, который регулируется параметром *track_activity_query_size* (по умолчанию 1 КБ). Текст выявленного медленного запроса дает администратору отправную точку для оптимизации производительности. Запрос можно воспроизвести, получить план выполнения, проанализировать производительность и в результате наметить действия по оптимизации. Дополнительно для каждого запроса формируется уникальный идентификатор queryid, который в сочетании с полями usesysid и datid можно использовать для соединения с представлением pg_stat_statements, в результате чего будут получены дополнительные сведения о том, сколько ресурсов было использовано этим запросом и ему подобными.

Идентификаторы процессов. Все процессы в pg_stat_activity являются процессами операционной системы и имеют присвоенный ею уникальный идентификатор pid. Этот идентификатор доступен в статистике и является ключом, с помощью которого можно соединять pg_stat_activity с другими представлениями. Например, это может понадобиться для получения расширенной информации о блокировках или ходе выполнения отдельных операций.

СУБД умеет распределять выполнение частей запроса между несколькими процессами, ускоряя выполнение запросов и увеличивая эффективность использования многоядерных систем. При наблюдении важно отделять группу процессов, занятых выполнением параллельного запроса от всех остальных процессов или групп. Для этой цели служит поле leader_pid, которое у всех дочерних процессов указывает на pid родительского процесса.

Идентификаторы транзакций и горизонта видимости транзакций. Два идентификатора, указывающие на присвоенный номер транзакции backend_xid и транзакционный горизонт видимости backend_xmin, могут быть полезны при расследовании причин ненормального увеличения размеров таблиц и индексов из-за неэффективной работы автоочистки. Более подробно случаи использования этой статистики мы рассмотрим в главе 8, посвященной очистке.

Представление pg_locks

Представление pg_locks содержит информацию об удерживаемых блокировках. Оно может рассматриваться как самостоятельное, однако на практике часто используется совместно с pg_stat_activity, и оба представления обогащают итоговый результат.

Каждая строка pg_locks содержит информацию об объекте, на который установлена блокировка или для которого требуется установка блокировки, и о том, кто ее затребовал. Когда несколько процессов пытаются взять блокировку одного и того же объекта, этот объект будет показан в представлении несколько раз. При снятии блокировки с объекта информация из представления убирается. Объектами блокировки могут быть таблицы, отдельные строки или даже страницы в таблицах, идентификаторы транзакций, общие объекты СУБД. Некоторые действия, требующие блокировки, представлены отдельными объектами, например увеличение файлов таблицы или обновление таких метаданных, как pg_database.datfrozenxid. Также в представлении отображаются рекомендательные блокировки (advisory lock), смысл и назначение которых определяют сами приложения, но нет информации по легким блокировкам (lightweight lock, LWLock), которые работают на более низком уровне для контроля доступа к служебным структурам в памяти.

```
# \d pg_locks
                        View "pg_catalog.pg_locks"
       Column       |           Type           | Collation | Nullable | Default
--------------------+--------------------------+-----------+----------+---------
 locktype           | text                     |           |          |
 database           | oid                      |           |          |
 relation           | oid                      |           |          |
 page               | integer                  |           |          |
 tuple              | smallint                 |           |          |
 virtualxid         | text                     |           |          |
 transactionid      | xid                      |           |          |
 classid            | oid                      |           |          |
 objid              | oid                      |           |          |
 objsubid           | smallint                 |           |          |
 virtualtransaction | text                     |           |          |
 pid                | integer                  |           |          |
 mode               | text                     |           |          |
 granted            | boolean                  |           |          |
 fastpath           | boolean                  |           |          |
 waitstart          | timestamp with time zone |           |          |
```

Посмотреть полное описание pg_locks можно с помощью метакоманды \d+ pg_locks, а выше приведено его сокращенное описание. Это представление показывает табличную информацию на основе функции pg_lock_status.

Объект блокировки. С помощью следующего набора полей можно идентифицировать объект, связанный с блокировкой:

- relation, page, tuple — отношение, номер страницы и номер строки внутри страницы;
- virtualxid, transactionid — виртуальный и фактический номера транзакций;
- classid, objid, objsubid — идентификаторы объекта блокировки внутри системного каталога в случае, когда этот объект нереляционный.

В зависимости от объекта блокировки значения в некоторых полях могут отсутствовать (NULL).

Детали блокировки. Поля с дополнительной информацией играют важную роль в определении процессов, ожидающих получения блокировки:

- locktype — тип блокировки, который позволяет отличать блокировки друг от друга[1];
- database — идентификатор (OID) базы данных, в которой возникла блокировка;
- virtualtransaction — идентификатор виртуальной транзакции, которая удерживает или ждет блокировку;
- pid — идентификатор процесса ОС. Чаще всего это поле используется для соединения с pg_stat_activity и получения такой дополнительной информации, как имя пользователя, время начала запроса или транзакции, текст запроса, маркеры ожидания и т. д.;
- mode — режим блокировки, который позволяет определять совместимость с другими блокировками. Полный список режимов доступен в документации[2];
- granted — флаг, указывающий на факт удерживания блокировки (значение true) или ожидания ее получения (значение false);
- waitstart — время перехода в ожидание блокировки. Это поле является единственным источником достоверной информации о том, сколько времени процесс находится в ожидании блокировки. Однако стоит учитывать, что даже после того как началось ожидание блокировки (granted = false), значение поля в течение очень короткого периода может отсутствовать (NULL);
- fastpath — флаг, указывающий на факт взятия блокировки через интерфейс fast-path. Этот интерфейс давно объявлен устаревшим, и это поле актуально только в случаях эксплуатации приложений, которые его используют.

С помощью деталей блокировки можно отсеивать лишние блокировки и фокусироваться только на тех, что напрямую относятся к исследуемой проблеме, например связаны с конкретным сеансом, или имеют определенный тип, или находятся в ожидании дольше допустимого времени. Соединение с pg_stat_activity позволяет получить дополнительную информацию,

[1] postgrespro.ru/docs/postgresql/current/monitoring-stats#WAIT-EVENT-LOCK-TABLE
[2] postgrespro.ru/docs/postgresql/current/explicit-locking#LOCKING-TABLES

которая будет полезна для анализа и исправления проблемы на стороне прикладных приложений (имя пользователя и приложения, текст запроса).

Особенности pg_stat_activity и pg_locks

У обоих представлений, `pg_stat_activity` и `pg_locks`, есть одна важная особенность, которую я привык рассматривать скорее как недостаток. О ней всегда следует помнить при реализации мониторинга на основе этих представлений. Выводимая статистика отражает текущий момент и не является кумулятивной. Данные, взятые из этих представлений, являются *снимками* того, что происходило в момент обращения к представлениям. Неизвестно, что происходило в интервале между двумя снимками. На практике можно встретить системы, которые обслуживают десятки и сотни тысяч запросов в секунду. В промежутках между снимками может возникать большое количество кратковременных ожиданий, которые не попадут в снимки и не будут видны в мониторинге, но эти ожидания могут негативно сказываться на времени выполнения запросов. Так, отсутствие информации создает неполную картину о происходящем. При построении систем мониторинга и накоплении статистики на основе таких снимков объективность картины начинает сильно зависеть от частоты опроса статистики. На практике такой опрос выполняется с интервалом в несколько секунд и наиболее часто встречаются значения между 15 и 60 секундами. Но даже профилирование с частотой 1–10 миллисекунд не гарантирует полноты картины, к тому же добавляет накладные расходы на выполнение самого профилирования. Резюмируем: глядя на графики, построенные на основе представлений `pg_stat_activity` или `pg_locks`, вы должны помнить об этой особенности. Некоторые другие представления устроены подобным образом, и в дальнейшем, рассматривая их, я буду отмечать это отдельно.

Представление pg_stat_database

Каждая строка представления `pg_stat_database` содержит статистику об использовании конкретной базы данных. Также в представлении есть отдельная строка, которая содержит статистику по разделяемым, общим для всех баз объектам (часть из них принадлежат так называемому системному каталогу). Представление основано на семействе функций с префиксом `pg_stat_get_db_`, которые принимают в качестве аргумента идентификатор БД и возвращают одну метрику. Полное описание представления можно получить с помощью метакоманды `\d+ pg_stat_database`. Часть полей этого представления можно отнести к статистике клиентских подключений, что может помочь при мониторинге активности СУБД.

Представление хоть и содержит информацию о базах данных, однако некоторые сведения можно рассматривать в контексте, связанном с клиентами и их сеансами. С помощью этой информации можно отслеживать и количество подключенных клиентов, и то, насколько нормально приложения работают с СУБД. Воспользоваться можно следующими полями:

- `numbackends` — количество клиентских процессов, подключенных к БД. В строке со статистикой по разделяемым объектам значение будет отсутствовать (`NULL`). Это единственное

поле в представлении, которое показывает текущее значение. Все остальные поля содержат кумулятивную статистику за определенный период;

- `xact_commit` — количество транзакций, завершившихся фиксацией (`COMMIT`). Счетчик также учитывает успешное выполнение одиночных запросов в рамках неявных транзакций;
- `xact_rollback` — количество транзакций, завершившихся обрывом (`ROLLBACK`), в том числе по причине ошибок;
- `sessions_abandoned` — количество сеансов, принудительно завершенных по причине того, что соединение оставлено клиентом. Это может указывать на ситуации, когда клиентское приложение забывает о соединении и не закрывает его как положено (graceful close). Другой причиной могут быть сетевые проблемы, такие как ошибки передачи и потери пакетов, приводящие к нарушению работы TCP-сеансов;
- `sessions_fatal` — количество сеансов, которые были принудительно завершены по причине возникновения фатальных ошибок и невозможности продолжения работы. Такие ошибки могут происходить во время выполнения запросов или вызываться исключительными ситуациями на стороне сервера при взаимодействии между клиентом и сервером. В таком случае сервер не может продолжить сеанс и завершает его. Примером такого сценария может быть завершение сеанса из-за превышения тайм-аута, настроенного параметром *idle_in_transaction_session_timeout*;
- `sessions_killed` — количество сеансов, которые были принудительно завершены административным способом. Такое завершение сеанса может быть инициировано функцией `pg_terminate_backend` или самой СУБД, например при выключении или перезапуске сервера;
- `sessions` — суммарное количество сеансов, установленных с БД. Значение поля указывает на общее число установленных сеансов независимо от статуса их завершения, то есть включает в себя все значения и по остальным возможным статусам.

Начало и завершение сеанса обычно происходит по инициативе клиента. Часть рассматриваемых полей как раз указывают на количество сеансов, завершение которых было инициировано сервером СУБД. Завершение сеансов, таким образом, в некоторых ситуациях можно считать ненормальным. В процессе сеанса клиент отправляет отдельные команды или может объединять их в транзакции. Эти команды формируют рабочую нагрузку, и по полям `xact_commit` и `xact_rollback` можно в количественном выражении вычислить величину нагрузки в СУБД.

Другая полезная информация о работе сеансов относится к их продолжительности в разных состояниях. Речь о тех состояниях, что коротко упоминались при рассмотрении `pg_stat_activity` и более подробно рассмотрены чуть дальше. Однако набор полей является неполным, и часть информации придется вычислить с помощью простых арифметических операций. Нас интересуют следующие поля:

- `session_time` — суммарное время, проведенное всеми сеансами. Учет времени имеет некоторую особенность: время увеличивается только при переходе между состояниями. Поэтому можно ожидать появления больших скачков при наличии сеансов, которые могут

подолгу пребывать в одном и том же состоянии. Хорошим примером является оставленный надолго сеанс разработчика в IDE;

- active_time — время, проведенное сеансами в состояниях active и fastpath function call, которые соответствуют выполнению запросов;
- idle_in_transaction_time — время, проведенное в состояниях idle in transaction и idle in transaction (aborted).

Обратите внимание: среди перечисленных полей нет статистики о времени, проведенном в состоянии idle. Именно это значение придется считать самостоятельно, вычитая из общего времени время, проведенное в бездействующих транзакциях, и время выполнения запросов.

Все значения в этих полях выражаются в миллисекундах.

Статистика по базам данных, которую предоставляет pg_stat_database, имеет кумулятивный характер и накапливается с течением времени, что обеспечивает непрерывность ее учета без риска потерь, как это может быть в случае pg_stat_activity или pg_locks. С помощью отдельных функций администратор может сбрасывать статистику, чтобы начать процесс накопления заново. В представлении есть поле stats_reset, которое показывает время, когда статистика была сброшена. С его помощью можно вычислить, за какой интервал времени была накоплена статистика.

2.4. Подключенные клиенты

Сталкиваясь с новой ситуацией или незнакомым окружением, в первую очередь важно оценить общую картину происходящего, получить общее представление о том окружении, в котором предстоит найти и устранить возникшую проблему. Для меня в начале знакомства с СУБД важно понимать то, насколько активно используется система и кто ее пользователи. Обычно это информация о том, сколько и каких клиентов подключено к системе, из чего можно сделать примерный вывод о том, сколько ресурсов им может потребоваться, какую нагрузку они могут генерировать.

Есть два способа получить такую информацию. Первый способ — использовать представление pg_stat_database:

```
# SELECT datname, numbackends FROM pg_stat_database
  ORDER BY numbackends DESC;
  datname  | numbackends
-----------+-------------
 pgbench   |          32
 postgres  |           1
           |           0
 template1 |           0
 template0 |           0
```

В выводе запроса для каждой базы можно увидеть текущее количество клиентских процессов, подключенных к ней. Однако представление pg_stat_database показывает информацию о подключенных клиентах только в контексте баз данных, что может быть недостаточно для более детального анализа. Также обратите внимание на строку с пустым именем в результате запроса: это служебная строка, объединяющая в себе статистику по *разделяемым* объектам, которые доступны во всех БД, но при этом не принадлежат ни одной из них. Все такие объекты принадлежат системному каталогу.

Системный каталог

Системный каталог — это набор таблиц со служебной информацией, которая используется самой СУБД. Служебные данные включают в себя информацию о таблицах, полях, индексах, типах данных и т. д. Практически все DDL-команды типа CREATE, ALTER, DROP как раз оперируют данными системного каталога[1].

Второй способ получить информацию о подключенных клиентах, к тому же гораздо более подробную, — использовать представление pg_stat_activity:

```
# SELECT client_addr, usename, datname, count(*)
  FROM pg_stat_activity
  GROUP BY 1,2,3
  ORDER BY 4 DESC;
 client_addr  | usename  | datname  | count
--------------+----------+----------+-------
 192.168.64.8 | classic  | pgbench  |    19
 192.168.64.4 | maru     | pgbench  |    15
              |          |          |     5
 192.168.64.7 | serral   | pgbench  |     4
 192.168.64.5 | replica  |          |     1
 192.168.64.9 | pgbench  | pgbench  |     1
              | postgres |          |     1
              | postgres | postgres |     1
```

В этом примере клиенты сгруппированы по адресу подключения, имени пользователя и имени БД. Можно сделать следующие выводы:

- есть два адреса, откуда идет основная масса подключений;
- наибольшая часть подключений приходится на базу pgbench;
- с базой работают несколько пользователей.

Значение NULL для client_addr означает, что подключение выполнено через UNIX-сокет с того же узла, где запущен сервер СУБД, — скорее всего, это наше подключение через psql. Также можно видеть NULL-значения в полях usename и datname — обычно такие строки соответствуют фоновым службам. В полном выводе pg_stat_activity можно увидеть больше подобных строк.

[1] postgrespro.ru/docs/postgresql/current/catalogs

Обычно для отсеивания фоновых служб и отображения только клиентских сеансов используется условие backend_type = 'client backend'.

Способ для ранних версий PostgreSQL

Поле backend_type появилось в PostgreSQL 10, а в более ранних версиях можно прибегнуть к условиям вроде datname IS NULL. Как правило, фоновые процессы не подключаются к базам данных, за исключением процессов автоочистки.

Используя дополнительное условие, можно исключить фоновые процессы из выборки и получить информацию только по клиентским процессам:

```
# SELECT client_addr, usename, datname, count(*)
  FROM pg_stat_activity
  WHERE backend_type = 'client backend'
  GROUP BY 1,2,3
  ORDER BY 4 DESC;
 client_addr  | usename  | datname  | count
--------------+----------+----------+-------
 192.168.64.8 | classic  | pgbench  |    19
 192.168.64.4 | maru     | pgbench  |    11
 192.168.64.7 | serral   | pgbench  |     5
 192.168.64.9 | pgbench  | pgbench  |     1
              | postgres | postgres |     1
```

На практике при анализе проблем приходится использовать группировку и по другим полям и также применять разные условия в зависимости от поставленных вопросов и проверяемых гипотез.

Отслеживание клиентских сеансов

Получение информации с помощью SQL удобно, однако на практике часто приходится анализировать проблемы после того, как они уже исчезли, и нужны информация в исторической перспективе и динамика изменений во времени. Эти задачи решаются системами мониторинга, которые собирают, хранят и визуализируют статистику. С помощью мониторинга администратор имеет возможность рассматривать изменение статистики во времени с помощью графиков.

Для мониторинга необходимо всегда иметь в распоряжении информацию о подключенных клиентах, при этом информация должна быть достаточно подробной. Давайте рассмотрим практические примеры того, как может выглядеть статистика по клиентам и на какие вопросы она может ответить.

Для получения суммарного количества подключенных клиентов можно использовать следующий PromQL-запрос:

```
# sum(postgres_connected_clients_total{service_id="primary"})
```

График, который можно построить на основе этого запроса, продемонстрирован на рис. 2.1.

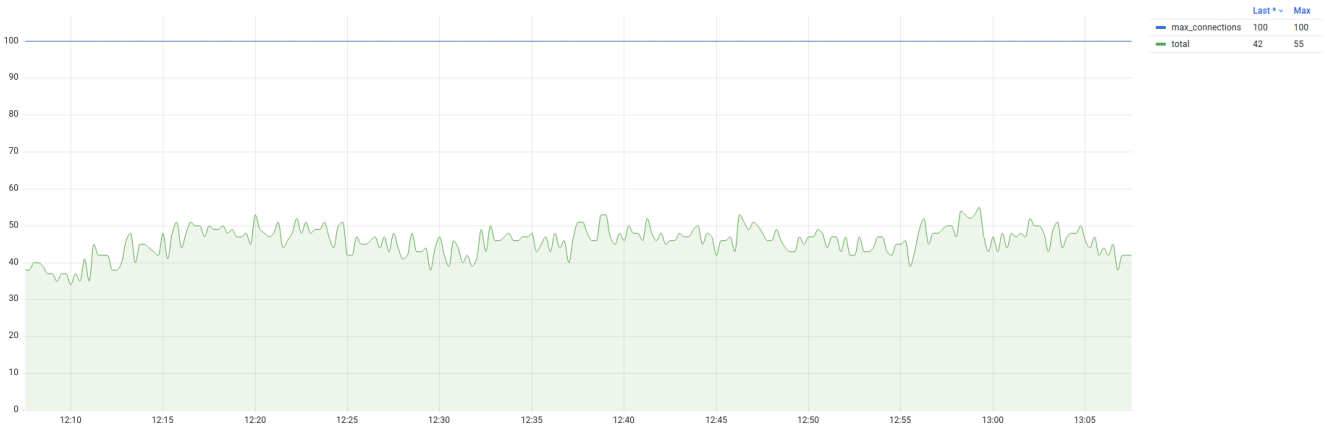


Рис. 2.1. Общее количество клиентов

На графике видно, как менялось количество подключений за последний час. В дополнение к основным метрикам указано и ограничение количества одновременных подключений, что наглядно дает понять, сколько еще подключений можно установить до достижения предела. Ограничение количества подключений задается параметром *max_connections* и по умолчанию равно 100. Из графика можно сделать вывод, что установлено чуть меньше половины от разрешенного ограничения и запас еще имеется.

Метрики параметров конфигурации

Параметры СУБД тоже доступны в виде метрик. Например, с помощью метрики `postgres_service_settings_info{name="max_connections"}` можно получить значение параметра *max_connections*.

Метрика `postgres_connected_clients_total` содержит в себе метки `address`, `user` и `database`, группировкой по которым можно воспользоваться, чтобы получить статистику в разных проекциях. Начнем с группировки по адресам:

```
# sum by (address) (postgres_connected_clients_total{service_id="primary"})
```

График будет выглядеть так, как показано на рис. 2.2. На нем видно, что бóльшая часть сеансов установлены с адресов 192.168.64.8 и 192.168.64.4. В окружениях с большим количеством экземпляров приложений такой график позволяет определять адреса, которые используют аномально много сеансов.

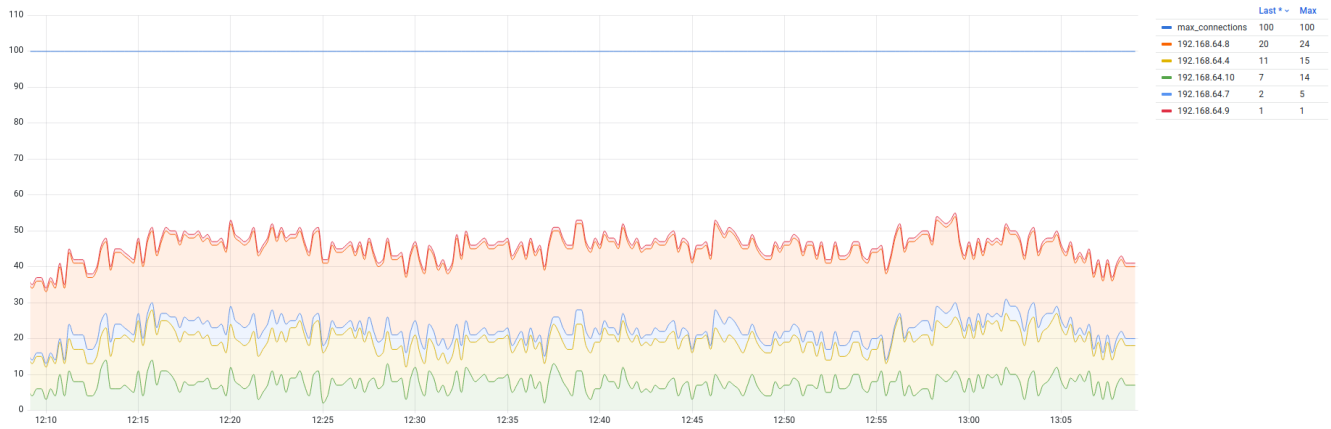


Рис. 2.2. Количество клиентов по адресам подключения

Теперь изменим условие группировки на имя пользователя, чтобы получить представление о том, какие пользователи подключаются к СУБД (рис. 2.3):

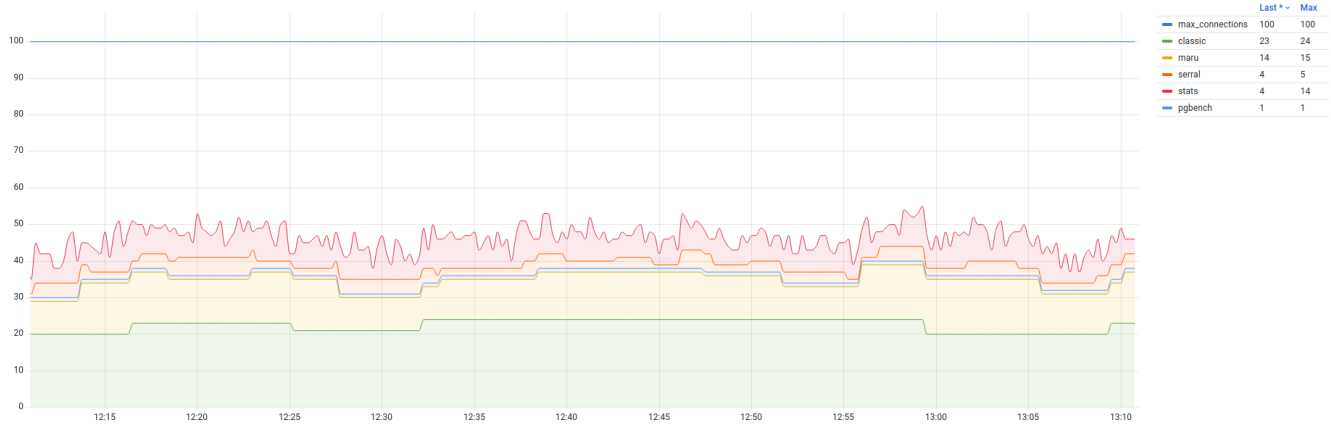


Рис. 2.3. Количество клиентов по именам пользователей

На графике видно, что большая часть сеансов выполнена от двух прикладных пользователей. Если присмотреться еще, то можно отметить, что выделяется пользователь `stats`. В тестовом окружении под этим пользователем подключается экспортер метрик для снятия статистики. Для экспортера, да и для любого другого агента мониторинга, такое количество соединений слишком велико, что является поводом для оптимизации кода.

Теперь взглянем на то, к каким базам данных выполнены подключения (рис. 2.4).

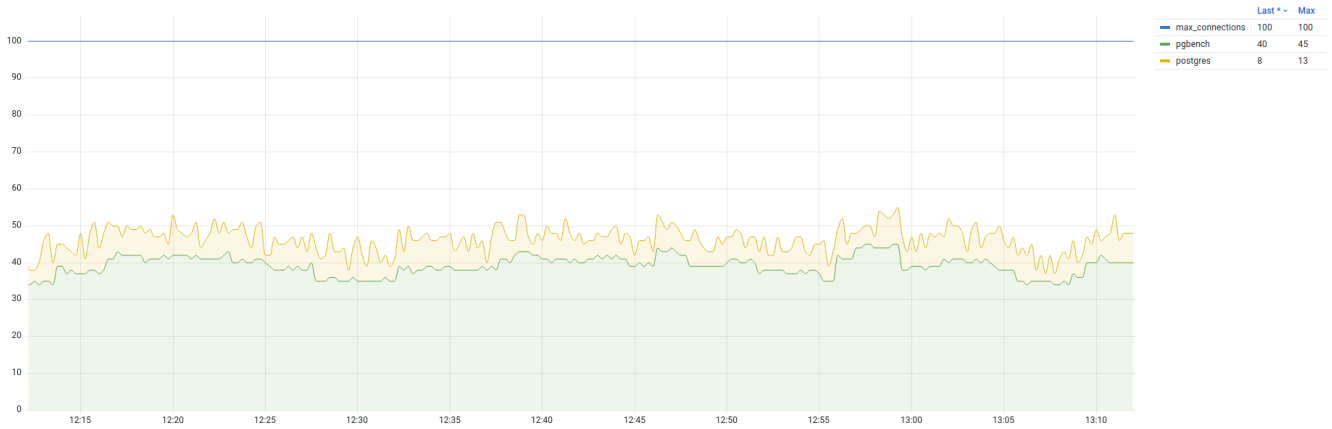


Рис. 2.4. Количество подключений к базам данных

Из приведенного графика становится очевидным, что почти все соединения установлены к БД pgbench и это основная БД, на которой сосредоточена рабочая нагрузка. Совсем небольшая часть соединений установлена к БД postgres; скорее всего, эти соединения также устанавливаются экспортером метрик.

В зависимости от информативности метрик и используемых меток можно строить и другие проекции. Например, при широком использовании *application_name* со стороны прикладных приложений эту информацию можно также экспортировать в метках метрик и вывести соответствующий график.

Транзакционная активность

Запросы — это базовая единица рабочей нагрузки. Их можно объединять в транзакции, и транзакция выполняется как единое целое: ошибка даже в одном запросе воспринимается как ошибка всей транзакции целиком. Механизм транзакций устроен так, что даже один запрос, не обернутый явно в команды управления транзакциями (`BEGIN` и `END`), также является транзакцией (состоящей из одной команды).

По количеству выполняемых транзакций можно сделать выводы о том, насколько активно используется база данных. Найти необходимую информацию можно в `pg_stat_database`. Она включает в себя поля `xact_commit` и `xact_rollback`, которые показывают количество зафиксированных и оборванных транзакций. Сумма этих полей и показывает транзакционную активность в базе данных:

```
# SELECT datname, xact_commit + xact_rollback AS xacts
  FROM pg_stat_database
  ORDER BY xact_commit + xact_rollback DESC;
  datname  |  xacts
-----------+---------
 pgbench   | 5786340
 postgres  |  357301
 template1 |    6116
           |      13
 template0 |       0
```

В тестовом окружении всего две активно используемые БД: pgbench и postgres. Также есть особая строка с отсутствующим именем базы данных, которая содержит статистику общих объектов системного каталога. Следующим запросом мы можем получить данные из мониторинга:

```
# rate(postgres_database_xact_commits_total{service_id="primary"} +
  postgres_database_xact_rollbacks_total{service_id="primary"}[1m])
```

Запрос считает количество выполненных транзакций в секунду, на его основе можно получить график (рис. 2.5), который показывает эту картину в динамике. На нем наглядно видно, что объем выполняемых транзакций в БД pgbench в несколько раз больше, чем в БД postgres.

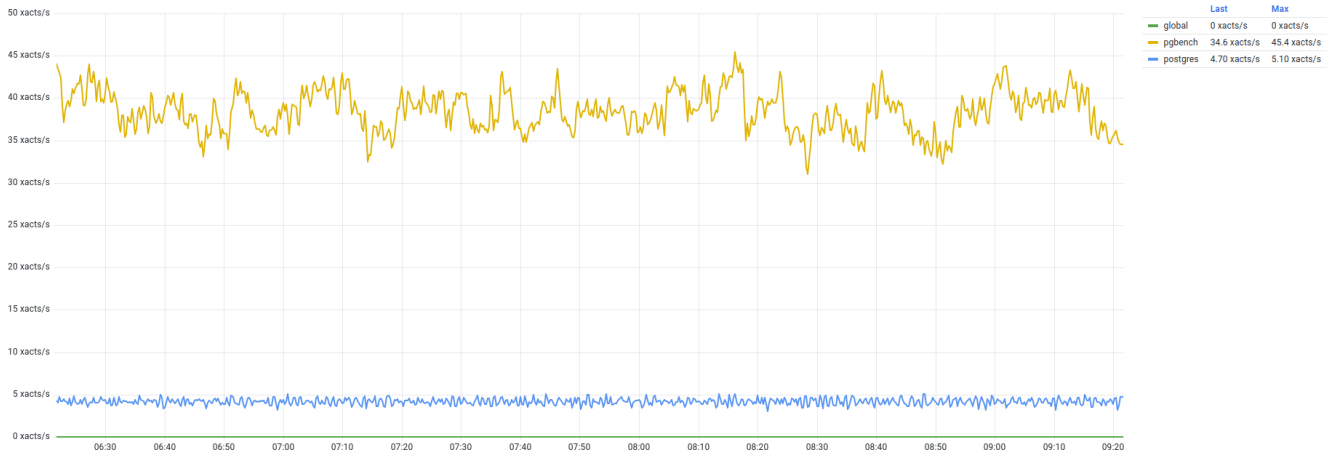


Рис. 2.5. Транзакционная активность: количество транзакций в секунду

График транзакционной активности полезен для поверхностного анализа и понимания текущей активности в СУБД, он хорошо подходит в качестве сводного графика для дежурных инженеров. Сильные колебания обычно являются признаком изменения количества экземпляров приложений или изменения в объеме рабочей нагрузки, вследствие чего может измениться и использование ресурсов со стороны СУБД.

Подсчет транзакций с помощью pg_stat_statements

Информацию о транзакциях можно получить с помощью `pg_stat_statements.query`, отслеживая команды начала и завершения транзакций (`BEGIN`, `COMMIT` и пр.). Однако у этого способа есть несколько недостатков. Во-первых, должно быть включено отслеживание служебных команд (*track_utility* = `on`). Во-вторых, расширение регистрирует только прямые вызовы команды `ROLLBACK`, следовательно, невозможно отслеживать откаты транзакций, произошедшие из-за ошибок. В-третьих, не всегда возможно отследить начало транзакции. Например, если клиент использует `\set AUTOCOMMIT off`, транзакция будет открываться неявно сразу после первой выполненной команды. Поэтому учет с помощью `pg_stat_statements` хоть и возможен, но менее удобен.

Статусы завершения сеансов

Нормальная работа сеанса предполагает, что клиент устанавливает соединение с СУБД, отправляет запросы и, когда необходимость в подключении исчезает, закрывает соединение и отключается. Однако возможно и аномальное поведение, при котором сеанс будет прерван:

- сброс соединения клиентом, СУБД или промежуточным сетевым устройством;
- возникновение ошибки в сеансе, после которой продолжение сеанса невозможно;
- принудительное завершение сеанса по инициативе администратора или СУБД.

Вполне возможны и другие, более экзотические варианты развития событий, но перечисленные выше причины встречаются наиболее часто. Аварийное завершение сеанса может происходить редко и незаметно — если сеанс прервется, СУБД запишет сообщение в журнал, приложение переустановит соединение и работа продолжится. Такое бывает из-за сетевых ошибок, тайм-аутов или потерь пакетов, когда нарушается работа TCP-соединения. Но есть и другая крайность: шквал ошибок, которые невозможно не заметить или игнорировать. Обычно это результат ошибок прикладного уровня, когда завершение сеанса происходит из-за ошибки в приложении. Например, после обрыва сеанса приложение может переустанавливать соединение и выполнять код с ошибкой, в результате чего сеанс сбрасывается и все повторяется снова без остановки. Такое поведение хорошо описывается термином *crash loop*, который не понаслышке знаком администраторам Kubernetes. При коротком (миллисекунды) интервале между ошибками приложение может создать шторм из попыток установки соединений. В лучшем случае это приведет к увеличению нагрузки на CPU на стороне СУБД (создание клиентских процессов обходится недешево). В худшем случае при наличии нескольких экземпляров приложения, которые ведут себя одинаково, можно исчерпать лимит подключений *max_connections* и сделать невозможным подключение других приложений к СУБД.

Статистика по статусам сеансов находится в `pg_stat_database`, получить ее можно следующим образом:

```
# SELECT datname, sessions, sessions_abandoned, sessions_fatal, sessions_killed
  FROM pg_stat_database
  WHERE sessions > 0;
 datname  | sessions | sessions_abandoned | sessions_fatal | sessions_killed
----------+----------+--------------------+----------------+-----------------
 postgres |  1154107 |                  0 |              0 |               0
 pgbench  |   383947 |                  0 |              0 |               0
```

Напомню, что `pg_stat_database` содержит кумулятивную статистику, поэтому среди значений можно наблюдать большие числа. На что нужно обращать внимание в этой статистике, это наличие аномальных статусов: `abandoned`, `fatal` и `killed`. В данном примере таких сеансов не зафиксировано, и это указывает на то, что на уровне сеансов приложения работают корректно и без ошибок. В этом примере можно обратить внимание на то, что к служебной БД `postgres` выполнено гораздо больше соединений, чем к прикладной БД `pgbench`. С помощью следующего PromQL-запроса можно построить график и посмотреть картину во времени:

```
# sum by (database) (
  increase(postgres_database_sessions_total{
  service_id="primary", database=~"(postgres|pgbench)"
  }[1m]))
```

В данном запросе количество сеансов, находящихся в любых статусах, суммируется по двум интересующим нас БД. Статистика является кумулятивной, и для наглядности изменений во времени стоит применить функцию `increase`, которая показывает изменение метрики за указанный интервал (в этом примере — одна минута).

На графике (рис. 2.6) видно, что довольно большое количество сеансов — примерно 90 в минуту — устанавливается с БД `postgres`. Это объясняется рабочей нагрузкой, характерной для экспортера метрик. Экспортер устанавливает несколько сеансов на время сбора статистики,

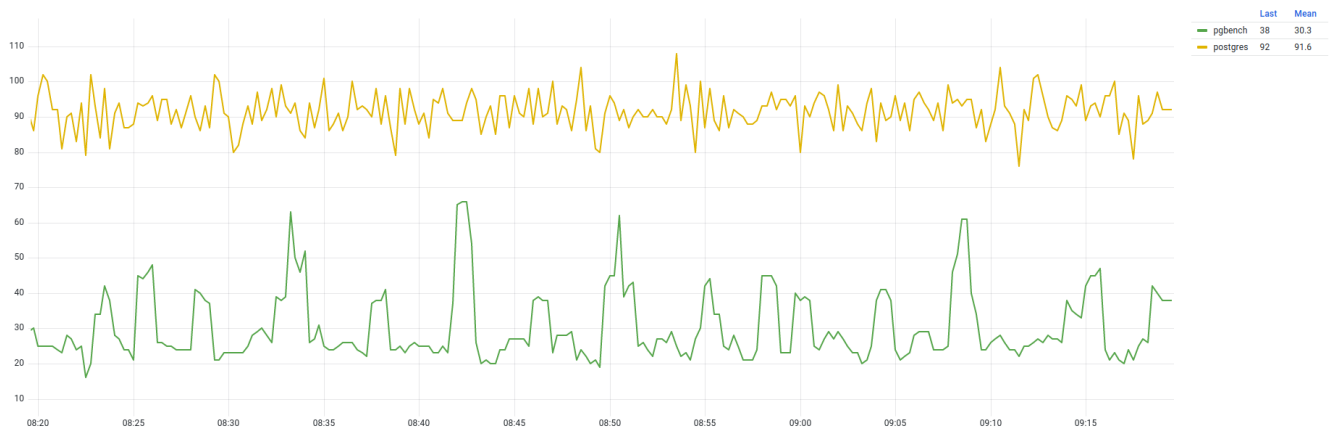


Рис. 2.6. Частота установления сеансов в минуту к базам данных

после чего завершает их. Такое поведение не очень эффективно в плане использования ресурсов, поскольку для каждого сеанса СУБД создает отдельный процесс и выполняет дорогостоящую инициализацию. С этой точки зрения выгодно установить и использовать несколько сеансов на постоянной основе, что является хорошей отправной точкой для оптимизации кода экспортера. Кроме экспортера в тестовом окружении работают прикладные приложения, которые перезапускаются каждые несколько минут с новыми параметрами нагрузки, и это заметно по резким пикам установки сеансов с БД pgbench. На практике реальные приложения могут перезапускаться редко, а установленные ими соединения могут продолжать работать в течение многих десятков минут и даже часов.

Пример из практики. Наличие аномальных статусов обычно является признаком проблемы или указывает на неэффективную работу приложения или окружения. Описываемый случай произошел, когда готовился материал этой главы.

Запрос на получение статусов сеансов показал, что значение в sessions_abandoned составляло 13 % от общего числа sessions и продолжало медленно увеличиваться. Это довольно большое значение, и я попытался выяснить причины такого количества оставленных сеансов. После отключения тестовой нагрузки счетчик продолжил расти, и подозрения упали на агента мониторинга. Эта проблема мне хорошо известна: она проявляется, если на стороне приложения не закрыть соединение должным образом. Имея доступ к исходному коду агента, я решил проверить все места, где открываются соединения. Нужно было убедиться, что соединения закрываются после использования, и я обнаружил одно из мест, где этого не делалось, что могло приводить к утечке соединений.

Для подтверждения постоянного характера проблемы я построил график (рис. 2.7) на основе следующего запроса:

```
# sum by (reason) (increase(postgres_database_sessions_total{
  service_id="primary",database="pgbench"
  }[1m]))
```

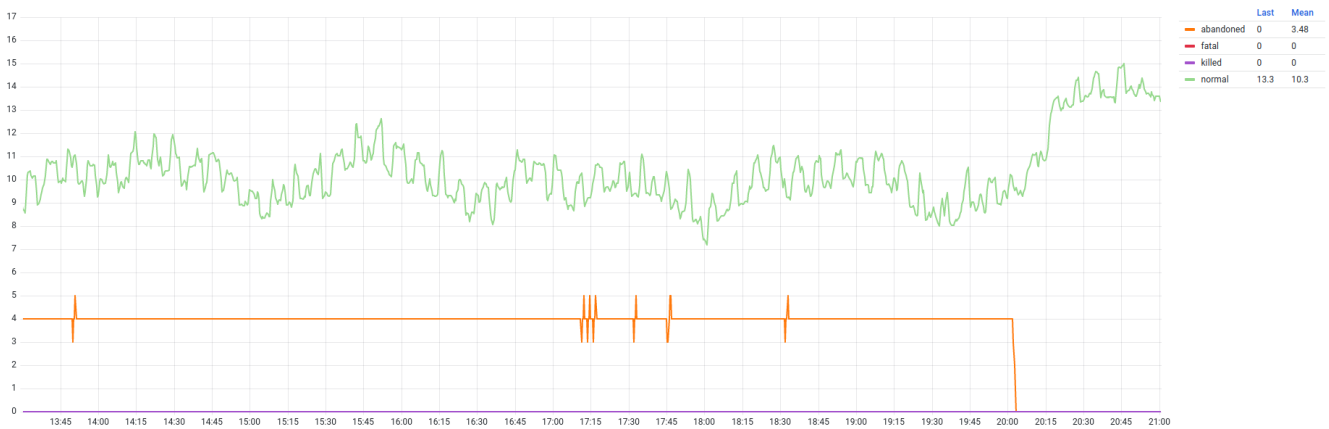


Рис. 2.7. Частота установления сеансов в минуту с группировкой по статусам

Результатом расследования стало исправление d098923[1]. Примерно в 20:00 версия приложения была обновлена, и проблема исчезла, что также подтвердилось графиком. Так мониторинг помог обнаружить и устранить ошибку в приложении агента.

2.5. Состояния сеансов

Клиентский процесс в своем жизненном цикле может находиться в разных состояниях, которые характеризуют происходящее в сеансе. Состояние процесса можно воспринимать как условный маркер, который помогает различать сеансы с нормальным и нежелательным поведением. Состояние отображается в `pg_stat_activity.state`, но при этом его можно увидеть только для клиентских соединений, а для фоновых процессов значение поля всегда отсутствует. Вероятно, это связано с тем, что «состояние» характеризует именно сеанс, а фоновые процессы не устанавливают сеансов, следовательно, и отображать нечего. Вполне возможно, в будущих версиях PostgreSQL ситуация изменится и появится какая-то информация о состоянии фоновых процессов. Давайте рассмотрим все состояния, в которых может находиться сеанс:

- `idle` указывает на состояние простоя. Если клиент не выполняет отправку команд, а сервер не занят выполнением запроса, то сеанс находится в холостом режиме и сервер ждет команды от клиента. В этом состоянии сеанс может находиться бóльшую часть своего времени.
- `active` указывает на активное состояние. Приняв команду от клиента, процесс начинает ее выполнение и переходит в активное состояние. Здесь важно отметить, что такое состояние включает в себя и состояние ожидания, когда процесс приостанавливает работу и ждет определенного события.
- `idle in transaction` указывает на состояние открытой транзакции, в которой ничего не происходит и процесс ждет от клиента команду на выполнение. Открыв транзакцию, приложение должно выполнить в ней набор команд и закрыть ее, однако по каким-то внутренним причинам между командами возникает пауза. Такое состояние в зависимости от характера выполняемых команд потенциально может привести к блокировкам и *раздуванию* таблиц и индексов из-за отложенной автоочистки. Следует отслеживать такие транзакции и устранять причины, приводящие к такому состоянию. Более подробно о бездействующих транзакциях можно узнать на с. 56.
- `idle in transaction (aborted)` указывает на состояние незавершенной транзакции, внутри которой произошла ошибка. Хорошим тоном считается, когда приложение полностью контролирует ход выполнения транзакции. Если выполнение любой из команд в транзакции привело к ошибке, приложение должно откатить транзакцию. При отсутствии управления транзакциями в таких ситуациях возникает риск *утечки ресурсов* (в виде слотов пула транзакций на стороне драйвера БД в приложении). Для самой СУБД пребывание сеанса

[1] github.com/lesovsky/pgscv/commit/d098923caef2d1a839df76ad9f441893640faed5

в таком состоянии является менее вредными, так как после ошибки все ресурсы и блокировки освобождаются, а вот для приложения это может быть чувствительно и способно привести к исчерпанию соединений во внутреннем пуле или появлению ошибок. Более подробно о бездействующих транзакциях можно узнать в подразделе на с. 56.

- `fastpath function call` указывает на выполнение функции через интерфейс fast-path[1]. Этот способ выполнения считается устаревшим, и использовать его не рекомендуется. На практике мне не приходилось встречаться с подобным состоянием, но допускаю, что давно разработанные и оставшиеся без поддержки приложения могут использовать эту функциональность.
- `disabled` не определяет конкретное состояние процесса, а лишь указывает на то, что отслеживание состояний сеансов отключено в конфигурации СУБД. Отслеживание обычно включено по умолчанию, и не рекомендуется выключать его, так как это снижает наблюдаемость и возможности мониторинга и отладки.

Существует еще одно важное состояние, которое при этом не отражается в поле `state`. Это *ожидание*. Это состояние неявно включено в состояние `active`, хотя, на мой взгляд, эти два состояния должны быть разделены и их следует рассматривать отдельно друг от друга. Активное состояние подразумевает действие и использование ресурсов, в то время как ожидание подразумевает бездействие и удерживание ресурсов от использования. Из этого следует, что активное состояние — это нормальное рабочее состояние системы, а ожидание — ненормальное, и администратор должен предпринимать действия, которые уменьшали бы ожидание.

Отслеживание состояний

Мониторинг клиентских сеансов по состояниям позволяет отслеживать выполнение рабочей нагрузки и появление в ней разных аномалий. Просмотреть все клиентские процессы и их состояния можно с помощью следующего SQL-запроса:

```
# SELECT state, count(*)
  FROM pg_stat_activity WHERE backend_type = 'client backend'
  GROUP BY 1 ORDER BY 2 DESC;
        state        | count
---------------------+-------
 idle                |    39
 active              |     3
 idle in transaction |     1
```

В этом примере с синтетической нагрузкой видно, что бóльшая часть соединений простаивает. В момент обращения к `pg_stat_activity` всего три процесса выполняют запросы, а один клиент открыл транзакцию и ожидает получения команды от клиента. Однако напоминаю, что состояние `active` неявно включает в себя и возможное ожидание, поэтому из результата непонятно

[1] postgrespro.ru/docs/postgresql/current/libpq-fastpath

точное состояние сеанса. Далее мы рассмотрим природу ожиданий и блокировок и варианты более точного определения состояния сеансов.

Ожидания и блокировки

Принцип блокировок заключается в ограничении или запрете одновременного доступа к ресурсу. В конкурентной среде следствием использования блокировок является появление очередей на доступ к заблокированному ресурсу. Нахождение в такой очереди напрямую влияет на время выполнения запроса или транзакции — выполнение будет заблокировано до тех пор, пока доступ не будет получен, а произойдет это только после того, как пройдет вся очередь и блокировка будет снята. Ожидание блокировок при выполнении запросов негативно сказывается на производительности: вместо того чтобы делать полезную работу, СУБД и приложения вынуждены ждать. Администратор обязан отслеживать, когда система тратит время на ожидание вместо полезной работы.

В случае клиентских процессов ожидание означает невозможность выполнять свою работу до тех пор, пока не освободится ресурс (объект) или не появится ожидаемое событие. Объектами могут являться таблицы или отдельные строки; легкие блокировки устанавливаются на ресурсы более низкого уровня. В качестве событий может выступать дисковый или сетевой ввод-вывод — синхронизация файлов, передача данных по сети и т. п.

Для отслеживания ожиданий используется представление `pg_stat_activity`. В СУБД потенциально очень много мест, где процесс может встать в ожидание. Вместо одного общего состояния важно знать, что именно стало причиной ожидания, поэтому поле `state` не очень подходит для этой задачи. В качестве инструмента СУБД предлагает так называемые *события ожидания* (wait event): они указывают на конкретные событие или ресурс, в ожидании которых находится процесс. В `pg_stat_activity` такие события ожидания представлены двумя полями:

- `wait_event_type` — общий тип ожидания, или, иными словами, обозначение группы, к которой относится ожидание;
- `wait_event` — конкретное событие или ресурс, в ожидании которых находится процесс.

Событие ожидания указывает на зафиксированный факт ожидания процессом определенного события или ресурса, однако они появились в версии 9.6. В более ранних версиях ожидание можно было определить по флагу `pg_stat_activity.waiting`, который указывал только на один из частных случаев ожидания — ожидание блокировки. События ожиданий расширили охват регистрируемых событий, которые могут стать причиной ожиданий, и заменили собой этот более простой флаг. Полный список всех регистрируемых ожиданий можно найти в документации в разделе, посвященном `pg_stat_activity`[1]. В качестве небольшого примера разберем результат запроса в тестовом окружении.

[1] postgrespro.ru/docs/postgresql/current/monitoring-stats#MONITORING-PG-STAT-ACTIVITY-VIEW

```
# SELECT wait_event_type ||'.'|| wait_event AS waiting, count(*)
  FROM pg_stat_activity
  WHERE wait_event_type IS NOT NULL AND backend_type = 'client backend'
  GROUP BY 1 ORDER BY 2 DESC;
      waiting       | count
--------------------+-------
 Client.ClientRead  |    30
 LWLock.WALWrite    |     7
 Lock.transactionid |     3
 IO.WALSync         |     1
```

Условием запроса исключены фоновые службы, их покрытие событиями ожиданий не самое большое, и в этом примере они нам не очень интересны. Для удобства работы с событиями ожиданий значения этих двух полей можно объединить в одно с помощью разделителя, например точки, так, чтобы оно сразу показывало и тип ожидания, и конкретное событие. Такое объединенное значение можно назвать *маркером ожидания*. В тестовом окружении есть несколько приложений, которые генерируют синтетическую нагрузку, и в приведенном результате запроса мы видим снимок ожиданий при выполнении этой нагрузки:

- `Client.ClientRead` — процесс ждет получения команды от клиента. Это обычное состояние для простаивающих процессов, в таком ожидании процессы могут находиться бóльшую часть своего существования.
- `LWLock.WALWrite` — ожидание легкой блокировки, необходимой для записи страницы WAL-журнала из буфера в памяти на диск. Запись в WAL осуществляется последовательно и часто это становится узким местом в производительности СУБД при высокой конкурентной нагрузке на запись.
- `Lock.transactionid` — ожидание завершения соседней транзакции. Часто случается, если транзакции пытаются изменить одни и те же данные.
- `IO.WALSync` — ожидание синхронизации записи WAL-сегмента на диск.

Из этого небольшого примера видно, что варианты ожиданий могут быть самыми разными и, более того, их влияние на синтетическую нагрузку тоже может быть разным. Например, в зависимости от производительности дисков (HDD, SSD, NVMe) скорость записи и синхронизации WAL будет разной, и это отразится на времени ожидания. Ожидание соседних транзакций зависит от характера рабочей нагрузки, количества конкурентных транзакций и продолжительности как самих транзакций, так и составляющих их команд. А вот ожидание команды от клиента никак не сказывается на нагрузке, сеанс находится в холостом режиме и может быть использован приложением немедленно, без задержек, как только потребуется. Таким образом, маркеры ожидания подсказывают администратору, куда тратится время, и дают отправную точку для возможных оптимизаций системы.

Однако маркеров ожидания довольно много, и не все из них следует воспринимать как критичное состояние, требующее немедленной реакции. Как я уже отметил, `Client.ClientRead` — это вполне безобидное ожидание, которое не нарушает работу системы.

За годы практики я и некоторые мои коллеги пришли к тому, что среди всех типов событий ожидания есть особая группа, которую можно выделить среди остальных и рассматривать как отдельное состояние, равноценное значениям из pg_stat_activity.state. Речь о группе событий с типом Lock: ожидания этой группы указывают на то, что выполнение команды остановлено и ожидает получения тяжелой блокировки. Группа включает в себя несколько ожиданий, которые более точно определяют причину ожидания. Одним из примеров может служить ожидание transactionid, указывающее на ожидание завершения соседней конкурентной транзакции. События группы Lock прямо указывают, что как процесс, так и клиентское приложение вместо выполнения полезной работы находятся в ожидании, и это само по себе негативно влияет на производительность приложения и СУБД.

Отслеживание состояний с учетом ожиданий

Для учета состояния ожиданий блокировок есть два способа. Первый и предпочтительный заключается в использовании событий ожидания. С помощью условия wait_event_type = 'Lock' процессы можно идентифицировать как находящиеся в ожидании блокировок:

```
# SELECT
    CASE WHEN wait_event_type = 'Lock'
        THEN 'waiting' ELSE state
    END AS state,
    count(*)
  FROM pg_stat_activity WHERE backend_type = 'client backend'
  GROUP BY 1 ORDER BY 2 DESC;
       state        | count
--------------------+-------
 idle               |    31
 waiting            |     3
 active             |     3
 idle in transaction |     1
```

Второй способ чуть сложнее и представляет, скорее, академический интерес. Способ заключается в соединении с pg_locks и использовании флага granted. Однако следует учесть, что для одного процесса в pg_locks будет несколько блокировок (несколько строк) и учитывать нужно только те, которые не захвачены:

```
# SELECT
  CASE WHEN NOT l.granted
      THEN 'waiting' ELSE a.state
    END AS state,
    count(*)
  FROM pg_stat_activity a
  LEFT JOIN pg_locks l ON a.pid = l.pid AND NOT l.granted
  WHERE a.backend_type = 'client backend'
  GROUP BY 1 ORDER BY 2 DESC;
```

```
        state         | count
----------------------+-------
 idle                 |    24
 active               |     8
 waiting              |     3
 idle in transaction  |     1
```

Результаты двух вариантов не совсем совпадают, потому что выполнены в разное время. Но первый вариант проще, поскольку позволяет обойтись без соединения.

Есть как минимум два способа реализовать метрики, показывающие состояние клиентов для мониторинга:

1. Отдельно показывать состояния на основе поля `state` и количество процессов со значением `Lock` в поле `wait_event_state`. В таком случае состояние `active` будет включать в себя процессы, находящиеся в ожидании блокировок.

2. Исключать процессы, находящиеся в ожидании с типом события `Lock`, из числа процессов в состоянии `active`.

Внутренняя реализация метрики `postgres_activity_connections_in_flight` использует второй способ и включает в себя метку `state`. Помимо стандартных состояний, метка может содержать и дополнительное состояние waiting, которое как раз указывает на процессы, находящиеся в ожидании блокировок:

```
# sum by (state)
  (postgres_activity_connections_in_flight{service_id="primary"})
```

На рис. 2.8 изображен пример графика на основе этого запроса.

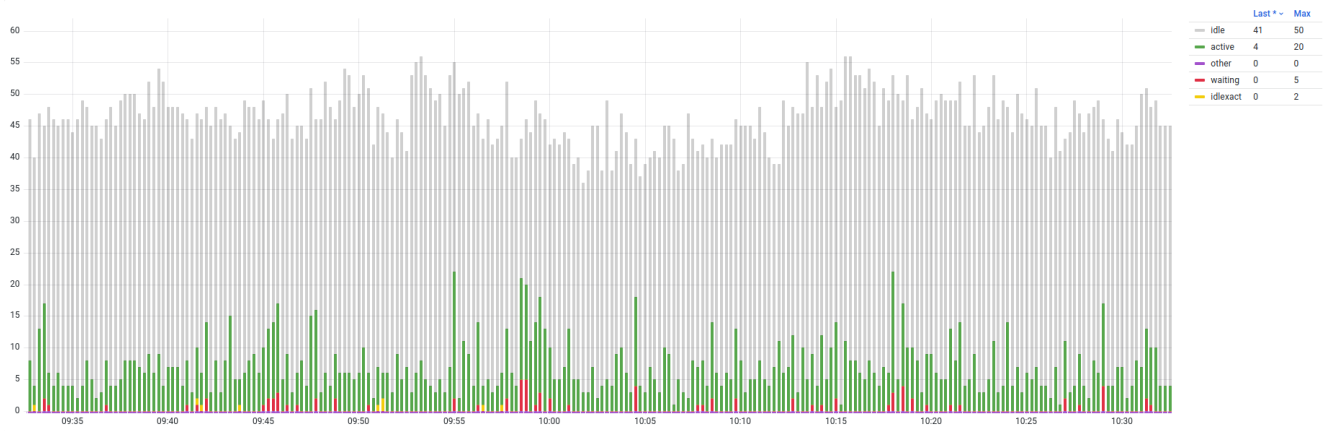


Рис. 2.8. Количество клиентов по состояниям

На графике есть процессы почти во всех возможных состояниях, в том числе и в ожидании блокировок. Объединяя возможности полей `state` и `wait_event_type`, можно собрать воедино информацию о состоянии клиентских сеансов.

Взаимоблокировки

Взаимоблокировка (deadlock) — это ситуация из нескольких блокировок, которая не может разрешиться сама собой, поскольку ее участники дошли до тупикового состояния, в котором все они заблокированы друг другом, перешли в ожидание и не могут продолжать работу. Для разрешения ситуации необходимо освободить часть блокировок путем принудительного завершения одного из участников взаимоблокировки.

В случае обычных блокировок ожидание может длиться непредсказуемо долго. Со взаимоблокировками все намного проще: их обнаружение и разрешение выполняется автоматически, без необходимости вмешательства со стороны администратора. Если блокировку не удалось взять за время *deadlock_timeout* (по умолчанию одна секунда), сеанс проверяет, не является ли он участником взаимоблокировки. В ходе проверки определяются участники конфликта. Если среди участников есть процесс очистки, то ему будет отправлен сигнал о завершении. Исключение составляют процессы очистки с целью предотвращения зацикливания счетчика транзакций (to prevent wraparound). Если процессов очистки нет, то сеанс, начавший проверку, принудительно отменяет выполнение своего собственного запроса, ради которого и потребовалось взять блокировки. Если запрос выполнялся в транзакции, вся транзакция становится ошибочной и ее следует завершить откатом. Остальные участники взаимоблокировки могут дальше продолжить работу, однако нет гарантий, что взаимоблокировка не возникнет снова.

На практике в самом простом варианте взаимоблокировки возникают при перекрестной попытке взять блокировки на одни и те же ресурсы из разных сеансов. Например, первая транзакция пытается захватить блокировку на ресурс, захваченный второй транзакцией, а вторая транзакция пытается захватить ресурс, взятый первой транзакцией. Основной вред взаимоблокировок заключается в том, что при их постоянном возникновении существенно уменьшается производительность (из-за снижения конкурентности) и появляется необходимость в повторе операции (запроса или транзакции) из-за ее принудительной отмены. Однако, учитывая автоматический характер разрешения взаимоблокировок, наносимый ими вред существенно меньше, чем в случае обычных блокировок, которые могут останавливать работу соседних сеансов на непредсказуемо долгое время.

К сожалению, универсального решения со стороны СУБД не существует, поскольку проблема заключается в неоптимальном порядке выполнения команд при конкурентной работе приложения с СУБД. Чтобы полностью решить проблему, необходимо изменить подход в работе с данными со стороны именно приложения.

Для отслеживания взаимоблокировок используют счетчик `pg_stat_database.deadlocks`, свой для каждой БД. В случае возникновения взаимоблокировки информация о ней записывается в журнал активности:

```
[UPDATE] ERROR:  deadlock detected
[UPDATE] DETAIL:
    Process 3498160 waits for ShareLock on transaction 809; blocked by process 3498137.
    Process 3498137 waits for ShareLock on transaction 808; blocked by process 3498160.
    Process 3498160: update t1 set a = 1 where a = 2;
    Process 3498137: update t1 set a = 2 where a = 1;
[UPDATE] HINT:  See server log for query details.
[UPDATE] CONTEXT:  while updating tuple (0,3) in relation "t1"
[UPDATE] STATEMENT:  update t1 set a = 1 where a = 2;
```

В зависимости от значения *log_line_prefix* содержимое журнала может дополняться и другой полезной информацией (время, пользователь, база данных, идентификаторы транзакций и пр.). В выводе есть ключевое сообщение о том, что зафиксировано появление взаимоблокировки — `deadlock detected`. Далее следуют информация о том, какой процесс находился в ожидании, каким процессом была вызвана блокировка, и тексты запросов, которые выполнялись в обоих сеансах в момент обнаружения взаимоблокировки. Здесь стоит учитывать, что это текущие запросы, а в реальности конфликтующая блокировка могла быть захвачена другим запросом той же транзакции, выполнение которого предшествовало текущему. Для полного анализа необходимо включать протоколирование всех запросов (*log_statement* = `all` или *log_min_duration_statement* = `0`), полностью записывать ход выполнения запросов в транзакции и проверять порядок их выполнения.

Бездействующие транзакции

В идеальном случае последовательность команд, составляющих транзакцию, должна выполняться непрерывно, без пауз или задержек — клиент, получив результат, должен немедленно отправить следующую команду. Однако на практике получается, что между отправкой команд приложение выполняет какую-то работу, в то время как СУБД ожидает от него продолжения (маркер ожидания `Client.ClientRead`). Транзакция в состоянии ожидания команды от клиента называется *бездействующей* (idle) транзакцией. Состояние простоя `idle in transaction` в абсолютном большинстве случаев возникает из-за клиента и лишь в редких случаях из-за окружения, в котором работает приложение (операционная система или сетевое окружение). С точки зрения приложения паузы между отправкой команд могут быть вызваны такими причинами, как:

- вычисления на основе результатов предыдущих команд;
- обращение к удаленному сервису (кеш, очередь, БД, API и т. п.);
- недостаток системных ресурсов и их выделение операционной системой (память);
- приостановка приложения при исчерпании установленного ограничения на использование ресурсов (Kubernetes requests/limits);
- возникновение ошибок и обработка исключений.

Это неполный список, и причины, по которым могут появляться простои в транзакциях, могут быть самыми разными и не всегда очевидными. В большинстве случаев ответственность за образование бездействующих транзакций возлагается на приложение. Хороший пример — обработка ошибок. В случае ошибки правильным действием со стороны приложения является завершение транзакции откатом. В худшем случае может возникать утечка транзакций — ситуация, когда приложение теряет контроль над транзакцией и она остается незавершенной в течение неопределенно долгого времени. Возможны и другие сценарии. Например, пользователь подключается к БД и выполняет какие-то команды. Пользователю может потребоваться начать транзакцию, выполнить в ней команду и затем сделать откат. Или IDE, в которой работает пользователь, при выполнении команд может неявно открывать транзакцию. Пользователь отвлекается на другую задачу, и ранее открытая транзакция остается незавершенной.

Пребывание клиентских сеансов в состоянии `idle in transaction` имеет свои негативные эффекты, и некоторые из них в зависимости от условий эксплуатации СУБД могут привести к аварии:

- Удерживание соединения с БД. Когда приложение открывает транзакцию, предполагается, что оно будет выполнять последовательность команд. В случае использования пулов соединений на стороне приложения соединение с открытой транзакцией не может использоваться другим потоком выполнения. При достижении ограничения на число одновременно открытых соединений на стороне пула, приложение заблокируется до освобождения хотя бы одного соединения.
- Устаревшие версии строк, *эффект раздувания* и падение производительности. При высокой интенсивности на запись размер таблиц и индексов растет из-за появления нескольких версий одних и тех же строк. Устаревшие версии строк не только занимают место на диске, но и негативно влияют на производительность. При чтении и поиске данных процессу приходится обрабатывать больше страниц в поисках нужной версии строки, что приводит к замедлению запроса и избыточному использованию ресурсов CPU, памяти и ввода-вывода. Для регулярного удаления устаревших версий строк и освобождения места используется автоочистка. Рабочие процессы автоочистки сканируют таблицы и индексы, находят версии строк, которые не нужны ни одной из транзакций, и удаляют их. Но старые версии строк могут понадобиться долгим транзакциям (в том числе бездействующим), поэтому автоочистка пропускает удаление таких версий. В случае же успешного удаления освобожденное место продолжает принадлежать таблице (или индексу). Получается, что размер таблиц и индексов увеличивается, но после очистки не уменьшается — это и называется эффектом раздувания. В особо запущенных случаях объем освобожденного пространства внутри таблиц и индексов может превышать общий объем полезных данных.
- Блокировки, исчерпание доступных подключений к СУБД. При высокой интенсивности на запись команды в транзакции могут удерживать блокировки строк. Соседние транзакции могут захотеть получить доступ к этим же строкам и при попытке взять блокировку будут ждать, пока блокировку не отпустит бездействующая транзакция. Ожидающие процессы не могут использоваться приложением. При высокой нагрузке приложение будет

пытаться открыть новые соединения, что может привести к росту очереди ожидающих и достижению ограничения на одновременно разрешенные подключения. При достижении этого ограничения приложение не сможет подключиться к БД, что приведет к невозможности работы и приложения, и СУБД, и других приложений, которые также хотели бы установить соединение с СУБД.

Таким образом, состояние `idle in transaction` является потенциально опасным, и долгое пребывание процессов в этом состоянии может привести к негативным последствиям. Такие транзакции нужно отслеживать и устранять. В тактическом плане такие транзакции можно завершать автоматически через включение тайм-аута *idle_in_transaction_session_timeout* в настройках СУБД. Эту настройку можно сделать как глобально, на уровне общей конфигурации СУБД, так и индивидуально, на уровне отдельных пользователей или баз данных. Другим вариантом может быть периодический запуск скриптов на основе функции `pg_terminate_backend` средствами утилиты cron. Такое решение может быть предпочтительным в случае, когда возможностей *idle_in_transaction_session_timeout* недостаточно и требуется более тонкая настройка тайм-аутов. Пример использования связки `pg_terminate_backend` и `pg_stat_activity`:

```
# SELECT pg_terminate_backend(pid)
  FROM pg_stat_activity
  WHERE usename = 'pgbench' AND application_name = 'pgbench'
    AND clock_timestamp() - coalesce(xact_start, query_start) > '00:01:00'::interval
    AND state ~ 'idle in transaction';
```

Такой запрос достаточно вызвать из psql. При необходимости список полей в `SELECT` или условия можно переопределить, а вывод результата сохранять, например, для анализа.

clock_timestamp vs now

Для определения длительности обычно принято вычитать отметку времени из текущего времени. В любой СУБД есть функция, которая выводит текущее время. В PostgreSQL такой функцией является хорошо известная `now`, однако ее особенностью является то, что она показывает время начала транзакции. Попробуйте вызвать `now` в транзакции несколько раз. Для целей же мониторинга рекомендуется использовать `clock_timestamp`, которая всегда показывает текущую отметку времени.

2.6. Время выполнения запросов и транзакций

Рабочая нагрузка состоит из постоянного потока запросов и транзакций, и чем быстрее они выполняются, тем больше пропускная способность системы. Наблюдая за рабочей нагрузкой

можно сфокусироваться на отдельных единицах выполнения — транзакциях и запросах, и отслеживать появление продолжительных запросов и транзакций, нехарактерных для этой рабочей нагрузки. Оптимизация времени выполнения положительно сказывается на производительности независимо от типа рабочей нагрузки. Неважно, OLTP, OLAP или HTAP, оптимизация всегда направлена на сокращение времени выполнения запросов и уменьшение объема ресурсов, необходимых для их выполнения. Время выполнения запросов зависит от таких факторов, как количество требуемых ресурсов, доступные СУБД ресурсы, сложность запроса, его параметры и план выполнения и т. п. Долгое выполнение команд может указывать на самые разные проблемы. Например, выбор планировщиком неэффективного плана может выражаться в долгом выполнении и избыточном использовании ресурсов. Долгие транзакции могут быть следствием плохого дизайна и проблем в управлении транзакциями на стороне приложения. С точки зрения приложений долгое выполнение операций в СУБД может приводить к тайм-аутам обработки запросов в самом приложении или эффекту зависания (если тайм-ауты не настроены), исчерпанию соединений к СУБД и аварийной остановке приложения. В регулярные задачи администратора входит отслеживание времени выполнения команд, расследование и устранение причин долгой работы. Зачастую устранением причин DBA занимается совместно с командами разработки приложений.

Для отслеживания продолжительности выполнения команд и транзакций в `pg_stat_activity` есть несколько полей:

- `xact_start` — время начала транзакции. Значение может отсутствовать (`NULL`), если в данный момент нет активной транзакции или в транзакции произошла ошибка и при этом транзакция не была закрыта, — состояние `idle in transaction (aborted)`;
- `query_start` — время начала текущего запроса (состояние `active`) или последнего выполненного запроса в сеансе (все прочие состояния).

Для получения информации о длительности выполнения запросов и транзакций можно использовать такой вариант запроса:

```
# SELECT pid,
  CASE WHEN wait_event_type = 'Lock' THEN 'waiting' ELSE state END AS state,
  (clock_timestamp() - xact_start) AS xact_age,
  (clock_timestamp() - query_start) AS query_age
FROM pg_stat_activity
WHERE (clock_timestamp() - xact_start > '00:00:00'::interval)
  OR (clock_timestamp() - query_start > '00:00:00'::interval
      AND state = 'idle in transaction (aborted)')
ORDER BY coalesce(xact_start, query_start);
  pid   |             state             |    xact_age     |    query_age
--------+-------------------------------+-----------------+-----------------
   3878 | idle in transaction (aborted) |                 | 00:00:04.264773
 242533 | active                        | 00:00:00.176216 | 00:00:00.174478
 241520 | active                        | 00:00:00.150633 | 00:00:00.149281
 242541 | waiting                       | 00:00:00.141747 | 00:00:00.139989
 241506 | waiting                       | 00:00:00.123768 | 00:00:00.122107
 241523 | active                        | 00:00:00.102864 | 00:00:00.101012
```

В этом запросе есть несколько особенностей, на которые стоит обратить внимание:

- Для определения состояния применяется расширенная логика с использованием событий ожидания. Процессы с типом события `wait_event_type = 'Lock'` расцениваются как процессы, ожидающие завершения конкурентных транзакций.
- Условие запроса отбирает именно транзакции, а не отдельные запросы: левая часть условия `OR` ищет активные транзакций, правая часть — отмененные (`aborted`), но не закрытые транзакции. И вот почему: запросы, даже если они выполняются отдельно, вне транзакций, на самом деле также используют транзакционный механизм и, по сути, являются транзакциями, состоящими из одной команды. В таком случае даже для обычных запросов заполняется поле `xact_start` (которое в этом случае равно значению `query_start`). Поэтому нет смысла концентрироваться на поиске запросов, они попадут в результат в любом случае.
- В правом от `OR` условии для поиска отмененных транзакций используется поле `state`. Смысл его использования заключается в том, что в случае возникновения ошибки внутри транзакции значение `xact_start` устанавливается в `NULL`, но сама транзакция при этом остается открытой. Чтобы не потерять такие транзакции, следует ориентироваться на `query_start` — время запроса, который завершился ошибкой.
- Изменяя значение в интервале, можно исключать из результата совсем короткие транзакции.

С помощью метрики `postgres_activity_max_seconds` можно получить максимальную длительность транзакции. Метрика содержит в себе несколько меток:

- `user` — пользователь, вызвавший активность;
- `database` — база данных, к которой подключен клиент;
- `type` — тип активности: пользовательский запрос или фоновое обслуживание (в случае автоочистки или сбора статистики для планировщика);
- `state` — состояние сеанса, согласно полю `state` с учетом состояния `waiting`.

С помощью меток можно выполнить агрегацию нескольких метрик в необходимой проекции. Например, следующим запросом можно отобразить долгую активность по конкретным пользователям независимо от типа активности, базы данных и состояния:

```
# max by (user) (postgres_activity_max_seconds{service_id="primary"})
```

На рис. 2.9 изображен пример графика на основе этого запроса. Из графика видно, что бóльшая часть запросов укладывается в интервал до 500 миллисекунд. Отдельно отметился процесс автоочистки, который выполнялся на тот момент около четырех секунд. Важно отметить, что эта метрика и график на ее основе показывают не абсолютно всю активность, происходящую в СУБД, а лишь ту, что была в момент опроса `pg_stat_activity`.

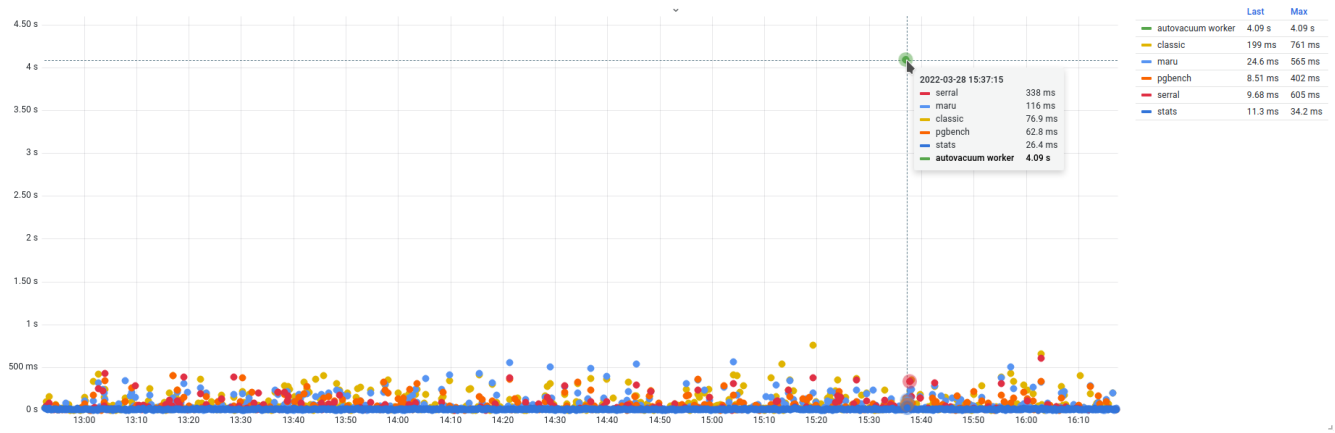


Рис. 2.9. Длительность выполнения транзакций

2.7. Отслеживание времени ожидания блокировок

Во время ожидания процесс не может выполнять полезную работу до тех пор, пока не исчезнет причина ожидания. Продолжительное ожидание само по себе неэффективно с точки зрения использования ресурсов и времени. Выявляя и устраняя участки, в которых происходит ожидание, можно ускорить выполнение рабочей нагрузки и общую производительность системы. Далее речь пойдет о времени, проведенном на блокировках, поскольку нет гарантированно точного способа определить время ожидания конкретного события.

Использование pg_locks.waitstart

Для оценки времени ожидания понадобится поле `pg_locks.waitstart`. Если процесс не находится в ожидании, то значение этого поля будет отсутствовать. Если процесс не смог взять блокировку, он переходит в ожидание, и это поле показывает время перехода в ожидание. Здесь есть неявная связь с другим полем этого же представления: на ожидание блокировки указывает не только `waitstart`, но и флаг `granted`. Однако они не полностью согласованы, и `waitstart` может в течение короткого периода содержать `NULL`, когда поле `granted` уже установлено в `false`.

При оценке времени ожидания можно использовать как минимум два подхода к расчету (на практике можно встретить и больше). Время ожидания всех процессов на текущий момент можно получить следующим запросом:

```
# SELECT
    a.pid, a.state, l.granted,
    a.wait_event ||'.'|| a.wait_event_type AS wait,
    clock_timestamp() - l.waitstart AS wait_age
  FROM pg_stat_activity a, pg_locks l
  WHERE a.pid = l.pid
    AND NOT l.granted;
   pid   | state  | granted |        wait        |    wait_age
---------+--------+---------+--------------------+-----------------
 2405179 | active | f       | transactionid.Lock | 00:00:00.284314
 2406471 | active | f       | transactionid.Lock | 00:00:00.277662
 2405394 | active | f       | transactionid.Lock | 00:00:00.018215
 2406467 | active | f       | transactionid.Lock | 00:00:00.060994
 2406472 | active | f       | tuple.Lock         | 00:00:00.240659
 2406466 | active | f       | transactionid.Lock | 00:00:00.125218
```

Первый вариант подсчета сводится к суммированию всех ожиданий всех процессов. В этом случае получается общее время ожидания, и чем оно больше, тем хуже ситуация, особенно в системах с большой конкурентностью. Вариант подходит для общей оценки того, сколько времени СУБД тратит на ожидание.

Второй вариант — это учет лишь максимального времени ожидания среди всех процессов. В этом случае получится картина только по одному процессу, который ждет дольше всех остальных. Вариант подходит для оперативного мониторинга, чтобы понимать, что в конкретный момент есть (или был) конкретный процесс, который находился в ожидании конкретный интервал времени.

Убрав в исходном запросе лишние поля и обернув его в CTE, можно получить оба значения и заодно посчитать количество ждущих процессов:

```
# WITH q AS (
    SELECT clock_timestamp() - l.waitstart AS wait_age
    FROM pg_stat_activity a, pg_locks l
    WHERE a.pid = l.pid AND NOT l.granted
  ) SELECT count(*),
    coalesce(max(wait_age), '0'::interval) AS max,
    coalesce(sum(wait_age), '0'::interval) AS sum
  FROM q;
 count |       max       |      sum
-------+-----------------+----------------
     6 | 00:00:00.284314 | 00:00:01.007062
```

В этом выводе видно, что в момент опроса было шесть процессов в ожидании, самый долгий ждал около 284 миллисекунд, а суммарно все клиенты СУБД прождали примерно одну секунду. В контексте синтетической нагрузки тестового окружения такие цифры могут казаться незначительными, однако в реальных окружениях значения могут быть совсем другого порядка. Своевременное обнаружение и сокращение времени ожидания позволит увеличить производительность приложений и СУБД.

Текущая реализация используемого агента по умолчанию собирает максимальное время ожидания среди всех процессов. Статистику ожиданий и график (рис. 2.10) можно получить с помощью запроса:

```
# postgres_activity_max_seconds{service_id="primary",state="waiting"}
```

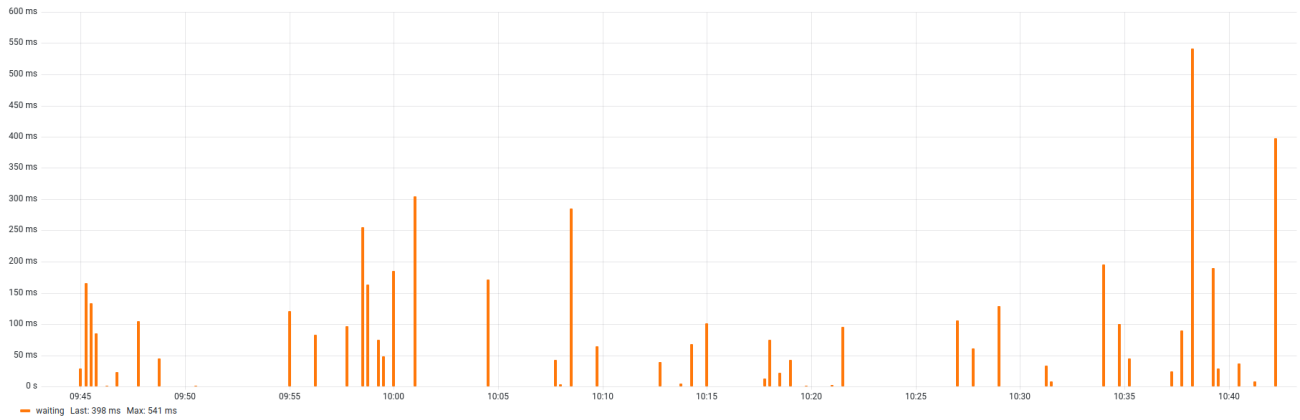


Рис. 2.10. Длительность ожидания блокировок

Ожидания блокировок не превышают секунды, и для тестового окружения это приемлемо. Но есть высокая вероятность, что на самом деле ситуация чуть хуже, чем это представляется из графика, так как `pg_locks` не аккумулирует статистику и исключает возможность получить полную картину по блокировкам.

Использование pg_stat_activity.state_change

В PostgreSQL до версии 14 отсутствовало поле `pg_locks.waitstart`, однако необходимость в отслеживании времени ожидания была всегда. В этих версиях СУБД можно использовать менее точный способ с использованием поля `pg_stat_activity.state_change`. Поле содержит отметку времени перехода состояния с предыдущего на текущее. Если ожидание включено в активное состояние, то при наличии у процесса маркера ожидания можно учесть время, проведенное в активном состоянии, как время, проведенное в ожидании. Недостаток способа заключается в том, что после перехода в активное состояние ожидание может начаться не сразу, а спустя какое-то время, но весь период будет зачтен как проведенный в ожидании. Это легко продемонстрировать на версии 14, где есть правильный источник `pg_locks.waitstart`.

Для демонстрации потребуется небольшая таблица, например та, что создается стандартным сценарием pgbench. В эксперименте эта таблица содержит 50 000 строк. Понадобится открыть

три сеанса к СУБД: первый и второй сеансы — для воспроизведения тестового сценария с блокировкой запроса, третий — для отслеживания статистики. В первом сеансе нужно открыть транзакцию и обновить последнюю строку в тестовой таблице. Транзакцию при этом следует оставить открытой:

```
# BEGIN;
# UPDATE pgbench_accounts SET abalance = abalance + 1 WHERE aid = 50000;
```

Во втором сеансе понадобится предварительно узнать pid процесса, после чего запустить полное обновление всей таблицы.

```
# SELECT pg_backend_pid();
2291758
# UPDATE pgbench_accounts SET abalance = abalance + 1;
```

В третьем сеансе следует взять статистику из pg_stat_activity и pg_locks. В качестве условия указываем вывод строк только для второго сеанса с идентификатором 2291758. Для удобства отслеживания после первого выполнения запроса стоит запустить метакоманду \watch 1, которая будет повторять запрос раз в секунду.

```
# SELECT
    a.pid, a.state, l.granted, a.wait_event ||'.'|| a.wait_event_type AS wait,
    (clock_timestamp() - a.state_change)::interval(0) AS state_age,
    (clock_timestamp() - l.waitstart)::interval(0) AS wait_age
  FROM pg_stat_activity a, pg_locks l
  WHERE a.pid = l.pid AND a.pid = 2291758;
```

В выводе запроса нас интересуют поля state_age и wait_age, которые и будут демонстрировать отличие способов в подсчете времени ожидания процесса. Полное обновление таблицы будет выполняться до тех пор, пока не будет достигнута последняя строка, которая ранее была обновлена во все еще незакрытой транзакции.

```
   pid   | state  | granted |      wait       | state_age | wait_age
---------+--------+---------+-----------------+-----------+----------
 2291758 | active | t       | WALWrite.LWLock | 00:00:08  |
 2291758 | active | t       | WALWrite.LWLock | 00:00:08  |
 2291758 | active | t       | WALWrite.LWLock | 00:00:08  |
 2291758 | active | t       | WALWrite.LWLock | 00:00:08  |
```

Здесь видно, что у процесса состояние active и время смены состояния state_age увеличивается. Значение granted = true указывает на то, что ожидания блокировок нет и запрос работает. Через какое-то время выполнение запроса остановится, и можно будет увидеть примерно следующее:

```
   pid   | state  | granted |        wait        | state_age | wait_age
---------+--------+---------+--------------------+-----------+----------
 2291758 | active | t       | transactionid.Lock | 00:00:14  |
 2291758 | active | t       | transactionid.Lock | 00:00:14  |
 2291758 | active | t       | transactionid.Lock | 00:00:14  |
 2291758 | active | f       | transactionid.Lock | 00:00:14  | 00:00:05
 2291758 | active | t       | transactionid.Lock | 00:00:14  |
 2291758 | active | t       | transactionid.Lock | 00:00:14  |
```

Появилась строка для блокировки, которую не удалось взять, с состоянием `granted = false` и отсчетом, начавшимся в `wait_age`. При этом значение `state` осталось прежним, `state_age` не сбросился и продолжает отсчитываться. В этом примере получается, что запрос выполнял полезную работу примерно девять секунд до начала ожидания блокировки. Фактически ожидание началось после девяти секунд работы запроса. Это поведение и демонстрирует отличие способов учета с помощью `waitstart` и `state_change`. Если вы решите повторить эксперимент, не забудьте в конце отменить или закрыть транзакцию, это позволит запросу во втором сеансе завершиться.

Используя учет времени на основе `state_change`, следует помнить о таком поведении. Например, в OLTP-нагрузке, преимущественно состоящей из коротких и быстрых запросов, оно не сильно исказит статистику, и этот способ подсчета может показаться приемлемым. А вот в случае более долгих OLAP-запросов (как в эксперименте выше) статистика может оказаться неточной, время ожидания будет несправедливо включать в себя время выполнения полезной работы, и способ подсчета может оказаться сомнительным.

2.8. Дерево блокировок

На практике часто бывает так, что в определенные моменты довольно много процессов находятся в ожидании блокировок, причем часть процессов оказываются заблокированы другими процессами. Если в такой ситуации найти источник блокировки и устранить его, то с высокой вероятностью остальные процессы смогут нормально продолжить работу. Однако, не имея подготовленных средств, в такой ситуации довольно сложно точно идентифицировать процесс, который стал причиной всех блокировок. Можно анализировать непосредственно вывод `pg_stat_activity` и `pg_locks`, но это неудобно и может занять много времени. Для определения источников блокировок нужно выявить зависимости между процессами и построить дерево зависимостей между заблокированными и блокирующими процессами. Такая визуализация позволит быстро определить источник блокировки и устранить его. Подход к решению таких проблем не является новым, и на просторах интернета можно найти разные реализации, которые позволяют выводить дерево блокировок. Такие запросы основываются на `pg_stat_activity` и `pg_locks` и, как правило, занимают несколько страниц, поэтому я не буду приводить их здесь. Такие запросы для удобства использования лучше всего оборачивать в представления.

В качестве отправной точки я взял этот запрос[1] и немного модифицировал его. Итоговый запрос можно найти в репозитории книги, в файле playground/scripts/locktree.sql[2]. Ключевой особенностью запроса является использование функции pg_blocking_pids. Функция принимает идентификатор процесса и возвращает список процессов, которые его блокируют. Это довольно удобно и позволяет избежать использования pg_locks, однако, согласно документации, частый вызов этой функции может негативно сказываться на производительности СУБД, так как функция получает кратковременный исключительный доступ к общему состоянию менеджера блокировок. Так или иначе, функция удобная, и использование ее в редких случаях поиска не должно быть проблемой. Не следует применять ее регулярно для нужд мониторинга, например для снятия метрик. В определенных условиях эксплуатации (высокая нагрузка, требования к задержкам), когда эта особенность даже для целей отладки оказывается неприемлемой, вместо pg_blocking_pids можно использовать pg_locks[3].

Давайте рассмотрим пару примеров получения дерева блокировок в тестовом окружении.

```
 pid  |    blocked_by     | state  |       wait        | wait_age | tx_age   | usename | datname | blkd |                                                                query
------+-------------------+--------+-------------------+----------+----------+---------+---------+------+-------------------------------------------------------------------------------------------
 2072376 | {}                | active | LWLock.WALWrite   |          | 00:00:00 | classic | pgbench |    1 | [2072376] END;
 2072427 | {2072376}         | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | maru    | pgbench |    0 | [2072427] . UPDATE pgbench_branches SET bbalance = bbalance + 3135 WHERE bid = 17;
 2072386 | {}                | active | LWLock.WALWrite   |          | 00:00:01 | classic | pgbench |    1 | [2072386] END;
 2072382 | {2072386}         | waiting | Lock.transactionid | 00:00:01 | 00:00:01 | classic | pgbench |    0 | [2072382] . UPDATE pgbench_branches SET bbalance = bbalance + 232 WHERE bid = 15;
 2072387 | {}                | active | LWLock.WALWrite   |          | 00:00:01 | classic | pgbench |    1 | [2072387] END;
 2072388 | {2072387}         | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | classic | pgbench |    0 | [2072388] . UPDATE pgbench_branches SET bbalance = bbalance + 3169 WHERE bid = 13;
 2072394 | {}                | active | LWLock.WALWrite   |          | 00:00:01 | classic | pgbench |    1 | [2072394] END;
 2072391 | {2072394}         | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | classic | pgbench |    0 | [2072391] . UPDATE pgbench_branches SET bbalance = bbalance + -2825 WHERE bid = 6;
 2072399 | {}                | active | LWLock.WALWrite   |          | 00:00:01 | classic | pgbench |    1 | [2072399] END;
 2072390 | {2072399}         | waiting | Lock.transactionid | 00:00:01 | 00:00:01 | classic | pgbench |    0 | [2072390] . UPDATE pgbench_branches SET bbalance = bbalance + -122 WHERE bid = 7;
 2072415 | {}                | active | LWLock.WALWrite   |          | 00:00:01 | maru    | pgbench |    1 | [2072415] END;
 2072418 | {2072415}         | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | maru    | pgbench |    0 | [2072418] . UPDATE pgbench_branches SET bbalance = bbalance + -3319 WHERE bid = 12;
 2072423 | {}                | active | IO.WALSync        |          | 00:00:01 | maru    | pgbench |    2 | [2072423] END;
 2072410 | {2072423}         | waiting | Lock.transactionid | 00:00:01 | 00:00:01 | serral  | pgbench |    0 | [2072410] . UPDATE pgbench_tellers SET tbalance = tbalance + 1404 WHERE tid = 18;
 2072416 | {2072423}         | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | maru    | pgbench |    0 | [2072416] . UPDATE pgbench_branches SET bbalance = bbalance + 1468 WHERE bid = 11;
 2072424 | {}                | active | LWLock.WALWrite   |          | 00:00:01 | maru    | pgbench |    4 | [2072424] END;
 2072408 | {2072424}         | waiting | Lock.transactionid | 00:00:01 | 00:00:01 | serral  | pgbench |    3 | [2072408] . UPDATE pgbench_branches SET bbalance = bbalance + -1605 WHERE bid = 8;
 2072392 | {2072408}         | waiting | Lock.tuple        | 00:00:00 | 00:00:00 | classic | pgbench |    2 | [2072392] .. UPDATE pgbench_branches SET bbalance = bbalance + -3889 WHERE bid = 8;
 2072409 | {2072408,2072392} | waiting | Lock.tuple        | 00:00:00 | 00:00:00 | serral  | pgbench |    1 | [2072409] ... UPDATE pgbench_branches SET bbalance = bbalance + 4620 WHERE bid = 8;
 2072419 | {2072409}         | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | maru    | pgbench |    0 | [2072419] .... UPDATE pgbench_tellers SET tbalance = tbalance + 2315 WHERE tid = 180;
```

Блокировки, представленные в первом примере, часто возникают при конкурентном обновлении данных. Каждая строка вывода описывает отдельный процесс и детализацию сеанса на основе pg_stat_activity. Давайте рассмотрим поля этого запроса:

- pid — идентификатор процесса;
- blocked_by — список процессов, которые блокируют текущий процесс. Рассчитывается на основе функции pg_blocking_pids;
- state — состояние процесса с дополненным псевдосостоянием waiting на основе условия wait_event_type = 'Lock';
- wait — маркер ожидания, составленный из wait_event_type и wait_event;
- wait_age — длительность ожидания процесса на основе pg_locks.waitstart;
- tx_age — длительность транзакции;

[1] postgres.ai/blog/20211018-postgresql-lock-trees
[2] github.com/lesovsky/postgresql-monitoring-book/blob/main/playground/scripts/locktree.sql
[3] dev.to/bolajiwahab/2022-01-13-postgresql-lock-trees-56e0

- `usename` — имя пользователя;
- `datname` — имя базы данных;
- `blkd` — количество других процессов, заблокированных данным;
- `query` — текст запроса. Содержит идентификатор процесса и его уровень в своей ветви дерева блокировок.

Все процессы можно условно разделить на группы, в которых есть основной источник блокировки, который никем не блокируется, и есть заблокированные процессы, которые могут также блокировать других участников группы. Давайте более внимательно рассмотрим последнюю группу, где главным процессом, который заблокировал остальных участников, является процесс с идентификатором 2072424. По тексту запроса видно, что это фиксация транзакции (`END`, или `COMMIT`), сам процесс находится в активном состоянии и его маркер ожидания — `LWLock.WALWrite`. Это указывает на то, что происходит запись в WAL-журнал, и пока она не завершится, клиент не получит возможность отправлять другие команды или открывать транзакции. Такое поведение характерно в режиме синхронной фиксации (см. *synchronous_commit*), который используется по умолчанию. По полю `blkd` можно увидеть, что фиксация транзакции блокирует еще четыре процесса, которые описаны в строках ниже. По полям `pid` и `blocked_by` можно проследить взаимосвязи процессов и кто кого блокирует. Также *глубину блокировки* удобно отслеживать по символу точки в поле `query`. По тексту запроса в этом же поле видно, что три процесса пытаются обновить одну и ту же строку по условию `WHERE bid = 8`. Вероятней всего, эта строка была обновлена в транзакции, которая фиксируется в данный момент, чем и вызвана очередь ожидания. Поля `state` и `wait` указывают на то, что все четыре процесса находятся в ожидании блокировок. Судя по полю `wait_age`, время ожидания составляет меньше одной секунды — это вполне приемлемо для тестовой рабочей нагрузки, а вот в производственной нагрузке даже такие короткие ожидания могут стать причиной задержек в приложениях. Поля `usename` и `datname` указывают, от имени каких пользователей и в какой базе данных возникли блокировки.

Рассмотрим еще один пример дерева блокировок, который может возникнуть в тестовом окружении.

```
 pid  | blocked_by | state   |        wait        | wait_age | tx_age   | usename  | datname | blkd |                                         query
------+------------+---------+--------------------+----------+----------+----------+---------+------+---------------------------------------------------------------------------------------------...
  161 | {}         | active  | IO.WALSync         |          | 00:00:00 | postgres | pgbench |   20 |      [161] TRUNCATE pgbench_history ;
 1923 | {161}      | waiting | Lock. relation     | 00:00:00 | 00:00:00 | serral   | pgbench |    0 |     [1923] . INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (15, 13, 1518573,...
 1924 | {161}      | waiting | Lock. relation     | 00:00:00 | 00:00:00 | serral   | pgbench |    2 |     [1924] . INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (46, 6, 589135, -...
 1925 | {161}      | waiting | Lock. relation     | 00:00:00 | 00:00:01 | serral   | pgbench |    1 |     [1925] . INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (106, 12, 100334,...
 1929 | {161}      | waiting | Lock. relation     | 00:00:00 | 00:00:00 | serral   | pgbench |    0 |     [1929] . INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (93, 18, 1330287,...
 1936 | {161}      | waiting | Lock. relation     | 00:00:00 | 00:00:00 | maru     | pgbench |    0 |     [1936] . INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (99, 7, 914556, 1...
 1942 | {161}      | waiting | Lock. relation     | 00:00:00 | 00:00:00 | maru     | pgbench |    3 |     [1942] . INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (61, 20, 669763, ...
 1957 | {161}      | waiting | Lock. relation     | 00:00:00 | 00:00:00 | classic  | pgbench |    0 |     [1957] . INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (104, 4, 880250, ...
 1963 | {161}      | waiting | Lock. relation     | 00:00:00 | 00:00:00 | classic  | pgbench |    0 |     [1963] . INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (162, 8, 1371725,...
 1964 | {161}      | waiting | Lock. relation     | 00:00:00 | 00:00:00 | classic  | pgbench |    0 |     [1964] . INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (69, 9, 71583, 18...
 1971 | {161}      | waiting | Lock. relation     | 00:00:00 | 00:00:01 | classic  | pgbench |    1 |     [1971] . INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (44, 14, 1331579,...
 1976 | {161}      | waiting | Lock. relation     | 00:00:00 | 00:00:00 | classic  | pgbench |    2 |     [1976] . INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (60, 17, 1238351,...
 955  | {1942}     | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | pgbench  | pgbench |    1 |      [955] .. UPDATE pgbench_branches SET bbalance = bbalance + -2769 WHERE bid = 20;
 1926 | {1976}     | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | serral   | pgbench |    0 |     [1926] .. UPDATE pgbench_branches SET bbalance = bbalance + -410 WHERE bid = 17;
 1927 | {1971}     | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | serral   | pgbench |    0 |     [1927] .. UPDATE pgbench_branches SET bbalance = bbalance + -4047 WHERE bid = 14;
 1930 | {1924}     | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | serral   | pgbench |    1 |     [1930] .. UPDATE pgbench_branches SET bbalance = bbalance + -3100 WHERE bid = 6;
 1938 | {1925}     | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | maru     | pgbench |    0 |     [1938] .. UPDATE pgbench_branches SET bbalance = bbalance + 608 WHERE bid = 12;
 1955 | {1976}     | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | classic  | pgbench |    0 |     [1955] .. UPDATE pgbench_branches SET bbalance = bbalance + 1745 WHERE bid = 17;
 1966 | {1942}     | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | classic  | pgbench |    0 |     [1966] .. UPDATE pgbench_branches SET bbalance = bbalance + -4835 WHERE bid = 20;
 1931 | {955}      | waiting | Lock.transactionid | 00:00:00 | 00:00:00 | serral   | pgbench |    0 |     [1931] ... UPDATE pgbench_tellers SET tbalance = tbalance + -4006 WHERE tid = 182;
 1972 | {1930}     | waiting | Lock.tuple         | 00:00:00 | 00:00:00 | classic  | pgbench |    0 |     [1972] ... UPDATE pgbench_branches SET bbalance = bbalance + -3771 WHERE bid = 6;
```

В этом примере источником блокировки выступает процесс с командой опустошения таблицы (TRUNCATE). Это довольно быстрая операция, заключающаяся в замене файла данных на пустой, однако сначала ей нужно дождаться завершения ранее запущенных запросов к целевой таблице. В момент опустошения таблица блокируется, поэтому все процессы, обращающиеся в нее, должны дождаться завершения команды. Поле blkd показывает, что процесс TRUNCATE (pid = 161) блокирует выполнение еще двадцати процессов. Примерно половина из них пытаются вставить строки в опустошаемую таблицу — у всех этих процессов в поле blocked_by указан идентификатор процесса TRUNCATE. Другая часть процессов — это операции обновления, которые, в свою очередь, ожидают выполнения вставок. Подтвердить это можно, сопоставив значения идентификаторов blocked_by с полем pid процессов, выполняющих вставку. Также можно сопоставить значения поля bid в запросах — оно будет совпадать в отдельных цепочках ожидающих запросов: например, процесс 955 обновляет строку с bid = 20 и ожидает завершения вставки в процессе 1942, который сам ожидает завершения команды TRUNCATE. Состояние и маркеры ожидания процесса, выполняющего команду TRUNCATE, показывают, что процесс активен и синхронизирует WAL-сегмент с файлом на диске (вызовом fsync или другим способом, который указан в параметре *wal_sync_method*). Еще одним способом определения причин ожиданий является исследование записей о блокировках в журнале сообщений СУБД (при условии, что включен параметр *log_lock_waits* и ожидание составило больше, чем значение параметра *deadlock_timeout*, по умолчанию равное одной секунде).

Оба рассмотренных случая относятся к рабочей нагрузке в тестовом окружении, которое характеризуется короткими транзакциями и такими же короткими блокировками. На практике все может быть намного сложнее, особенно если источником блокировок выступают бездействующие транзакции, которые могут удерживать блокировки непредсказуемо долго и потенциально приводить к аварийным ситуациям. Порядок действия в таких ситуациях, как правило, сводится к принудительному завершению процессов. В критической ситуации важно быстро сориентироваться и найти тот корневой процесс, завершение которого разрешит ситуацию. При недостатке информации часто приходится без разбора завершать множество процессов. В случае же использования запросов, подобных рассмотренному, можно с большей точностью определить и устранить источник блокировки, позволив остальным процессам продолжить работу.

Резюме

- Статистика клиентской активности позволяет понять, что происходит в СУБД.
- СУБД предоставляет три важных источника данных о клиентской активности: представления pg_stat_activity, pg_locks и pg_stat_database.
- И администраторы БД, и инструменты мониторинга получают данные из представлений с помощью SQL-запросов.
- Клиентский сеанс в процессе существования может пребывать в разных состояниях.

- Транзакционная активность помогает быстро оценить нагрузку в кластере баз данных.
- Важно вовремя отслеживать и устранять опасные ожидания и блокировки.
- Следует устранять или минимизировать нахождение транзакций в бездействующем состоянии.
- Продолжительность клиентской активности прямо влияет на производительность.
- Ожидание блокировок снижает производительность СУБД и приложений.
- Дерево блокировок позволяет быстро выявить источник блокировок.

Глава 3

Выполнение запросов и функций

В этой главе мы рассмотрим:

- зачем нужен мониторинг запросов;
- расширение `pg_stat_statements`;
- метаданные запроса;
- статистику планирования и исполнения запросов;
- практические примеры использования `pg_stat_statements`;
- сквозную идентификацию запросов по всем источникам статистики;
- примеры отчетов на основе `pg_stat_statements`;
- представление `pg_stat_statements_info`;
- представление `pg_stat_user_functions`.

В предыдущей главе мы рассмотрели клиентские процессы и сеансы, связанную с ними статистику и примеры ее использования для задач мониторинга. Далее, в этой главе, мы заглянем внутрь клиентских сеансов и рассмотрим способы анализа рабочей нагрузки. Запросы, отправляемые клиентом, определяют рабочую нагрузку; ее эффективное выполнение сервером СУБД определяет общую производительность системы. Статистика СУБД дает администратору возможность рассматривать нагрузку с разных точек зрения и разбирать общий поток рабочей нагрузки на отдельные запросы. Рассматривая запросы по отдельности, мы можем наметить отправные точки их оптимизации, что в итоге положительно скажется на общей производительности СУБД.

3.1. Зачем нужен мониторинг запросов

После установки соединения и настройки сеанса СУБД готова выполнять команды клиента. Посредством команд клиент может запрашивать и изменять данные (DML-команды), управлять объектами схемы (DDL-команды), выполнять служебные операции и т. п. Полный список команд можно найти в официальной документации[1]. Базовой единицей выполнения в рабочей нагрузке можно считать *запросы* (queries) — это, как правило, команды на извлечение и изменение данных: `SELECT`, `INSERT`, `UPDATE`, `DELETE`.

[1] postgrespro.ru/docs/postgresql/current/sql-commands.html

Команда, оператор, запрос?

Помимо термина *запрос*, в документации можно встретить термин *оператор* (statement). Это понятие имеет более широкий смысл: «оператор» охватывает весь диапазон существующих команд, в то время как «запрос» — это подмножество команд, которые возвращают результат или изменяют данные. Далее в этой главе вместо слов «команда» и «оператор» я буду использовать более привычный термин «запрос».

Запросы, их количество и типы, а также участвующие в них объекты БД формируют индивидуальный для каждой СУБД профиль рабочей нагрузки, для выполнения которой требуются ресурсы. Оптимизируя отдельные типы запросов, можно в конечном счете оптимизировать использование ресурсов самой СУБД и добиться существенного увеличения производительности (или сокращения избыточно выделенных ресурсов, которые ранее были необходимы).

Часто на практике приходится быть свидетелем попыток решить проблемы производительности поиском *магического* параметра в настройках, включение которого ускорит все и сразу. К сожалению, «ускорения», вызванные такими параметрами, жертвуют стабильностью или надежностью системы, так что нужно заранее хорошо оценивать риски. Более того, таким способом нельзя добиться качественного улучшения производительности и того вау-эффекта (ускорения на порядки), которого можно достичь только с помощью оптимизации запросов. Наиболее правильный путь — это рефакторинг запроса или в простом случае добавление индекса под конкретный тип запроса. Поэтому понимание того, какие запросы выполняются в системе, является ключевым пунктом для будущей оптимизации производительности.

Для оценки профиля рабочей нагрузки и выполняемых запросов используется несколько представлений:

- `pg_stat_statements` — представление с кумулятивной статистикой по запросам, предоставляемое одноименным расширением;
- `pg_stat_user_functions` — представление со статистикой выполнения пользовательских функций.

Помимо этих стандартных средств, есть и другие инструменты, которые разрабатываются сообществом на основе `pg_stat_statements`:

- `pg_stat_kcache`[1] — расширение собирает статистику о фактическом использовании системных ресурсов отдельными запросами клиентских процессов. Для учета статистики используется системный вызов `getrusage`[2], и это может накладывать некоторые ограничения на платформы, отличные от Linux;
- `pg_stat_monitor`[3] — расширение позиционируется как дополнение к `pg_stat_statements`, в котором есть расширенные средства для оценки запросов: агрегация статистики по интервалам, сбор планов, учет таблиц, участвующих в запросах, и многое другое.

[1] github.com/powa-team/pg_stat_kcache
[2] man7.org/linux/man-pages/man2/getrusage.2.html
[3] github.com/percona/pg_stat_monitor

Расширения добавляют отдельные представления, которые дополняют функциональность pg_stat_statements новыми возможностями. Однако я оставлю их рассмотрение в качестве домашнего задания читателю.

3.2. Расширение pg_stat_statements

Представление pg_stat_statements является одним из основных источников статистики по всем запросам, которые выполняются в СУБД. Представление служит интерфейсом к одноименному расширению, которое отслеживает стадии выполнения запросов и хранит накопленную статистику. Расширение сохраняет статистику по всем успешно выполненным запросам. В случаях, когда запрос завершился ошибкой (или был прерван администратором), статистика о нем не будет сохранена, что, пожалуй, можно считать некоторым недостатком при учете ресурсов.

По умолчанию расширение pg_stat_statements выключено, однако рекомендуется всегда его включать. Сделать это можно следующим образом:

1. Указать расширение в параметре *shared_preload_libraries*, после чего перезапустить сервер. Это необходимо для резервирования места в общей памяти, инициализация которой выполняется только при старте; получить место в уже выделенной общей памяти на лету во время работы СУБД невозможно.

2. После перезапуска СУБД расширение нужно установить в конкретной базе. Для этого подходит любая база: можно использовать базу данных postgres, которая всегда создается по умолчанию. Несмотря на то что расширение устанавливается в конкретную базу, сбор статистики осуществляется глобально для всех запросов независимо от того, в каких базах они работают. Установка выполняется с помощью команды CREATE EXTENSION pg_stat_statements;.

В тестовом окружении расширение pg_stat_statements уже установлено и настроено, поэтому дополнительного ничего делать не нужно. После установки расширения сбор статистики начинается автоматически. Убедиться в этом можно простым запросом к pg_stat_statements с подсчетом количества строк в представлении:

```
# SELECT count(*) FROM pg_stat_statements;
 count
-------
   113
```

На данный момент уже накоплена статистика по 113 *типам* запросов. Запросы, даже очень похожие друг на друга, могут отличаться значениями параметров. Хранить в статистике запросы со всеми значениями параметров избыточно — запросы могут содержать сотни и тысячи параметров и уникальных значений. Поэтому в процессе сбора для каждого запроса выполняется

нормализация — конкретные параметры заменяются пронумерованными заполнителями вида $1, $2, $3 и т. д. Нормализация позволяет отнести запрос к некоторому типу и учитывать объем использованных ресурсов для этого типа.

Вместе с основным представлением устанавливается еще одно — `pg_stat_statements_info` с метаданными о работе самого расширения; оба представления основаны на одноименных функциях. Вместе с представлениями в составе расширения есть служебная функция `pg_stat_statements_reset`, предназначенная для сброса накопленной статистики.

Помимо обязательного указания в *shared_preload_libraries*, расширение имеет несколько параметров конфигурации, которые позволяют подстраивать работу расширения под различные условия эксплуатации СУБД:

- *pg_stat_statements.max* — максимальное количество типов запросов, отслеживаемых расширением, что соответствует количеству строк в представлении. При регистрации большого количества запросов реже обновляемая часть статистики может быть удалена. Факты удаления можно отследить с помощью `pg_stat_statements_info.dealloc`. Значение параметра устанавливается только при старте сервера. На практике значения по умолчанию (5000) обычно достаточно, но на производственных серверах с большим разнообразием запросов его имеет смысл сразу увеличивать как минимум вдвое, чтобы избежать перезапуска СУБД в будущем.
- *pg_stat_statements.track* — уровень отслеживаемых запросов. Возможные значения:
  - `top` — отслеживать вызовы верхнего уровня, то есть запрошенные клиентом. Включено по умолчанию;
  - `all` — отслеживать все вызовы, включая вложенные в функции и процедуры;
  - `none` — отключить сбор и хранение статистики.

  При наличии хранимых процедур и функций имеет смысл устанавливать значение `all`, что позволит иметь более детальную статистику о выполняемой рабочей нагрузке.
- *pg_stat_statements.track_utility* — отслеживать ли статистику для служебных команд, отличных от DML-запросов вроде `SELECT`, `INSERT`, `UPDATE` и `DELETE`. Если на основе представления `pg_stat_statements` строятся отчеты производительности, включение этого параметра позволит получить более полную и точную картину о рабочей нагрузке. Но важно помнить, что отслеживаемые служебные команды также будут занимать место в памяти и может возникнуть риск удаления статистики из-за достижения *pg_stat_statements.max*.
- *pg_stat_statements.track_planning* — собирать ли статистику о планировании запросов. По умолчанию сбор выключен, так как может приводить к снижению производительности, особенно в случае, когда запросы с одинаковой структурой выполняются множеством клиентов (что характерно для OLTP-нагрузки), поскольку это вызывает конкуренцию за обновление одних и тех же элементов статистики.
- *pg_stat_statements.save* — сохранять ли статистику при перезапусках или выключении сервера. По умолчанию включено, и обычно эту настройку можно оставить без изменений.

Перечисленные параметры можно указать в основной конфигурации, в файле `postgresql.conf`, и для вступления в силу большинства из них достаточно перечитать конфигурацию (для изменения *pg_stat_statements.max* требуется перезапуск).

Больше подробностей, относящихся к работе расширения, можно найти в официальной документации[1].

Всю статистику в `pg_stat_statements` можно условно разделить на несколько категорий:

- запрос и его метаданные;
- статистика планирования (доступна с PostgreSQL 13 и `pg_stat_statements` 1.8);
- статистика выполнения;
- статистика использования ресурсов — здесь речь идет об использовании ресурсов с точки зрения СУБД, а с точки зрения операционной системы она может иметь несколько иной вид, о чем я упомяну отдельно;
- статистика записи в WAL-журнал (доступна с PostgreSQL 13 и `pg_stat_statements` 1.8);
- статистика JIT-компилятора (доступна с PostgreSQL 15 и `pg_stat_statements` 1.10).

Можно заметить, что разная статистика появляется в разных версиях расширения. При обновлениях основной версии СУБД следует также обновлять и расширения.

В следующих разделах мы шаг за шагом рассмотрим все категории статистики.

3.3. Метаданные запроса

Текст запроса и его метаданные позволяют отличать одни запросы от других и дают отправную точку поиска приложения, отправившего запрос. Запрос и его метаданные представлены несколькими полями:

- `userid` — идентификатор пользователя, от имени которого был выполнен запрос. Имя можно найти с помощью соединения с `pg_user` или функцией `pg_get_userbyid`;
- `dbid` — идентификатор базы данных, в которой был выполнен запрос. Имя можно найти с помощью соединения с `pg_database`. Функция для получения имени базы данных идентификатору отсутствует;
- `queryid` — идентификатор запроса, вычисляемый на основе его текста. На него не влияют имя пользователя или базы, и если аналогичный запрос выполняется в других базах или другими пользователями, то значения `queryid` для таких запросов будут совпадать;
- `query` — нормализованный текст запроса. Нормализация объединяет множество однотипных запросов с разными значениями параметров в один общий тип запроса, где конкретные значения заменены пронумерованными аргументами;

[1] postgrespro.ru/docs/postgresql/current/pgstatstatements.html

- toplevel — уровень вызова запроса: true для вызовов верхнего уровня и false, если вызов был вложен в процедуру или функцию. Если *pg_stat_statements.track* установлен в значение top, поле всегда равно true. При создании отчетов производительности этот флаг позволяет отфильтровать вложенные вызовы и не учитывать их статистику, поскольку она уже учтена в статистике запросов верхнего уровня. Без использования этого флага статистика может быть посчитана несколько раз, и отчет получится неверным.

С помощью метаданных появляется возможность выделить и сгруппировать статистику по отдельным пользователям, базам данных или группам:

```
# SELECT pg_get_userbyid(s.userid) AS username, d.datname, count(*)
  FROM pg_stat_statements s, pg_database d
  WHERE s.dbid = d.oid
  GROUP BY 1,2 ORDER BY 3 DESC;
 username | datname  | count
----------+----------+-------
 stats    | postgres |    31
 pgbench  | pgbench  |    12
 stats    | pgbench  |    10
 maru     | pgbench  |     9
 classic  | pgbench  |     9
 serral   | pgbench  |     9
 postgres | postgres |     6
```

3.4. Планирование запроса

Планирование — это отдельный этап, предшествующий выполнению запроса. На этом этапе планировщик рассматривает набор планов, которые могут отличаться сложностью исполнения и *стоимостью* (cost) — оценкой количества вычислительных ресурсов, необходимых для выполнения запроса. Стоимость измеряется в условных единицах, за базу принята стоимость чтения одной страницы при последовательном доступе. Среди построенных планов выбирается наиболее оптимальный, то есть имеющий наименьшую стоимость, согласно которому и будет выполнен запрос. В общем случае план зависит от сложности запроса, наличия подходящих индексов, настроек планировщика и того, насколько статистика соответствует реальным данным. Статистика планирования в pg_stat_statements позволяет отслеживать количество планирований и общее время, затраченное на планирование. Однако, как уже замечено, по умолчанию сбор статистики планирования выключен ввиду возможного снижения производительности при видах рабочей нагрузки, схожих с OLTP.

Статистика планирования в pg_stat_statements представлена следующими полями:

- plans — количество планирований запроса;
- total_plan_time — общее время, затраченное на планирование запросов данного типа;

- min_plan_time — наименьшая продолжительность планирования;
- max_plan_time — наибольшая продолжительность планирования;
- mean_plan_time — средняя продолжительность планирования;
- stddev_plan_time — величина стандартного отклонения для времени, затраченного на планирование.

Все значения времени указываются в миллисекундах.

Наибольший интерес в этой статистике вызывают время планирования и факты его значительных изменений, особенно в бóльшую сторону, когда запрос стал планироваться дольше при более или менее постоянном количестве планирований. Такое случается на практике, но, к счастью, это большая редкость.

Сценариев использования статистики планирования не так много, отправными точками могут служить следующие варианты:

- время планирования отдельных типов запросов может быть полезно для отслеживания аномалий планирования в случаях, когда объем выполняемых запросов не изменяется;
- величина стандартного отклонения показывает, насколько сильно время планирования «гуляет» относительно среднего времени планирования.

Давайте рассмотрим в качестве примера первый вариант. Для этого нам потребуется следующий запрос:

```
# topk_avg(5,
  sum by (queryid,query) (
    rate(postgres_statements_time_seconds_total{
           service_id="primary",mode="planning"
    }[1m]) + on(database,user,queryid) group_left(query)
        0 * postgres_statements_query_info{service_id="primary"}
  ), "other")
```

Подобные запросы будут и дальше использоваться в этой главе, поэтому его стоит детально разобрать:

- в качестве основной величины берется некоторая конкретная метрика, в данном случае это postgres_statements_time_seconds_total;
- относительно этой метрики считается частота изменений в минуту;
- для получения метки с текстом запроса (query) выполняется соединение с метрикой postgres_statements_query_info на основе общих меток database, user, queryid;
- величина postgres_statements_query_info равна единице, поэтому умножаем ее на ноль, чтобы не прибавлять лишнюю единицу к величине основной метрики;
- полученные метрики с частотой изменения суммируются с группировкой по меткам queryid и query, поскольку в тестовом окружении есть несколько пользователей, которые совместно выполняют одни и те же запросы;

- функция `topk_avg` считает средние величины за выбранный интервал временно́го ряда, сортирует их по убыванию и берет первые *K* значений (в данном случае пять), а остальные суммирует в шестую метрику с меткой `other`.

Полученный запрос выводит статистику по времени планирования (метка `mode="planning"`) пяти запросов, которые планировались наиболее долго. Время планирования остальных запросов объединено в шестой метрике с меткой `other`.

На основе запроса можно построить так называемый *график top-K* (рис. 3.1), который будет довольно часто появляться на протяжении всей книги. Удобство этого графика заключается в том, что он позволяет вывести и наибольшие величины, и сумму оставшихся.

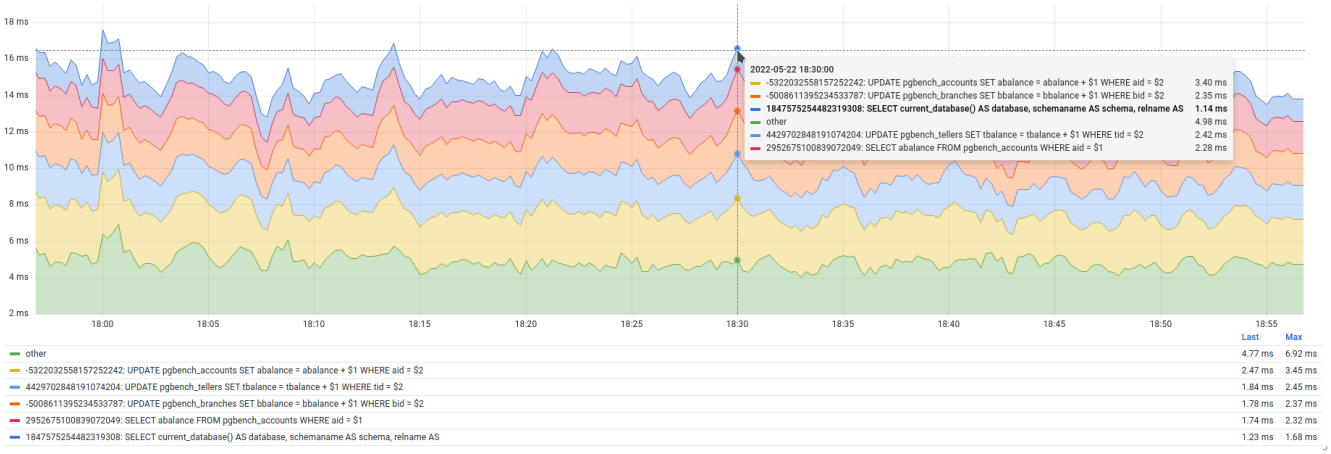


Рис. 3.1. Время планирования запросов

Каждая строка легенды содержит идентификатор и текст запроса, а также последнее и максимальное значение метрики. Для каждого типа запросов график показывает, сколько *машинного* времени СУБД тратит на планирование в одну секунду *реального* времени. В машинном времени суммируется продолжительность работы всех ядер, поэтому в нагруженных системах с многоядерными процессорами машинное время за одну реальную секунду легко может секунду превышать. График отображает изменение частоты именно *суммарного* времени планирования всех запросов конкретного типа, и эту особенность очень важно понимать. На графике отмечена конкретная точка — 18:30:00, и в эту секунду реального времени на планирование всех запросов всеми ядрами в сумме было затрачено около 16 миллисекунд (более подробная детализация приведена во всплывающей подсказке).

График времени планирования важно рассматривать вместе с графиком количества планирований на основе `pg_stat_statements.plans`. Поскольку величина затраченного времени прямо пропорциональна количеству планирований, при уменьшении (или увеличении) числа планирований затраченное время также уменьшится (или увеличится). Подозрительными можно считать случаи, когда время планирования изменилось, а количество планирований осталось прежним или вообще изменилось в обратную сторону.

3.5. Исполнение запроса

Стадия исполнения — это основной этап жизни запроса, в котором СУБД, используя системные ресурсы, выполняет его согласно плану и возвращает результат пользователю. Результатом может быть набор строк или тег (command tag). От скорости выполнения напрямую зависят текущая производительность и время отклика, ведь чем быстрее выполнится запрос и чем больше их можно выполнить за единицу времени, тем больше и общая производительность СУБД. Статистика исполнения запросов является наиболее полезной и востребованной, так как позволяет найти аномалии при выполнении запросов и сразу же начать оптимизацию (рефакторинг запроса, добавление индекса и т. п.), не погружаясь при этом глубоко в детали использования ресурсов.

Статистика исполнения представлена несколькими полями:

- `calls` — количество успешно исполненных запросов. По смыслу это поле близко к `plans`, однако оба поля являются независимыми, считаются отдельно друг от друга и могут содержать разные значения. Успешное планирование не гарантирует, что исполнение не завершится ошибкой. И наоборот, из успешного исполнения не следует, что перед ним выполнялось планирование: в случае подготовленных запросов (prepared statements) на этапе подготовки план выполнения может кешироваться;
- `rows` — общее количество строк, которое было возвращено (`SELECT`) или затронуто (`INSERT`, `UPDATE`, `DELETE`) в результате выполнения всех запросов данного типа;
- `total_exec_time` — общее время, затраченное на исполнение запросов данного типа. Это время включает в себя и время ожидания блокировок и событий;
- `min_exec_time` — наименьшая продолжительность исполнения;
- `max_exec_time` — наибольшая продолжительность исполнения;
- `mean_exec_time` — средняя продолжительность исполнения;
- `stddev_exec_time` — величина стандартного отклонения продолжительности исполнения;
- `blk_read_time` — общее время, затраченное на чтение данных и размещение их в буферном кеше;
- `blk_write_time` — общее время, затраченное на запись данных.

Время планирования

До версии `pg_stat_statements` 1.9 поля, показывающие время выполнения, назывались иначе и не содержали в своем имени слова `exec`. Переименование вызвано внедрением отслеживания планирования, из-за чего потребовалось добавить новые поля, а существующие переименовать. Например, поле `total_time` было переименовано в `total_exec_time`.

После включения и установки расширения сбор статистики осуществляется автоматически, за исключением статистики времени, затраченного на чтение и запись в буферный кеш (`blk_read_time` и `blk_write_time`). Для учета времени ввода-вывода необходимо включить параметр *track_io_timing*. Как и во всех остальных случаях, время выполнения считается в миллисекундах.

Статистика, связанная с исполнением запросов, позволяет понять состав рабочей нагрузки — типы запросов и их долю в общем объеме. Имея представление о запросах, можно сразу переходить к их оптимизации. Далее мы рассмотрим некоторые практические примеры использования статистики исполнения.

Запросы с наибольшим количеством вызовов. Количество вызовов можно получить в поле `calls` и с его помощью определить текущую производительность. Для ее вычисления нужно суммировать значения `calls` всех запросов:

```
# SELECT sum(calls) AS total_calls
FROM pg_stat_statements;
 total_calls
-------------
   173438464
```

На выходе получилось довольно большое число. Напомню, что статистика `pg_stat_statements` накапливается в течение определенного периода, и полученное число — это количество выполненных запросов за все время накопления статистики. Значение производительности всегда определяется за какой-то период и выражается, например, в запросах в секунду или минуту. Для подсчета производительности достаточно простой арифметики, и в контексте PostgreSQL ее можно вычислить с помощью SQL и функции `\watch` клиента psql. Другой вариант — использовать функции, которые предлагает язык запросов системы мониторинга:

```
# sum(rate(postgres_statements_calls_total{service_id="primary"})[1m])
```

Функция `rate` считает частоту изменения величины `postgres_statements_calls_total` в указанном интервале (одна минута). Затем подсчитывается сумма для всех метрик (на каждый тип запроса — отдельная метрика). Результатом запроса будет график, показанный на рис. 3.2.

На графике изображена динамика изменения частоты выполнения запросов, в легенде внизу подсчитаны агрегаты для интервала. Так, например, средняя частота выполнения — 258 запросов в секунду примерно за 12 часов.

Такой график дает поверхностное представление о текущей рабочей нагрузке, и при ее изменении можно увидеть и изменение в производительности, особенно в случае значительных колебаний. Исключив общее суммирование, можно получить подробную детализацию и определить долю отдельных типов запросов в общей производительности. Это дает понимание того, какие запросы выполняются чаще остальных и на каких запросах стоит сосредоточиться при оптимизации.

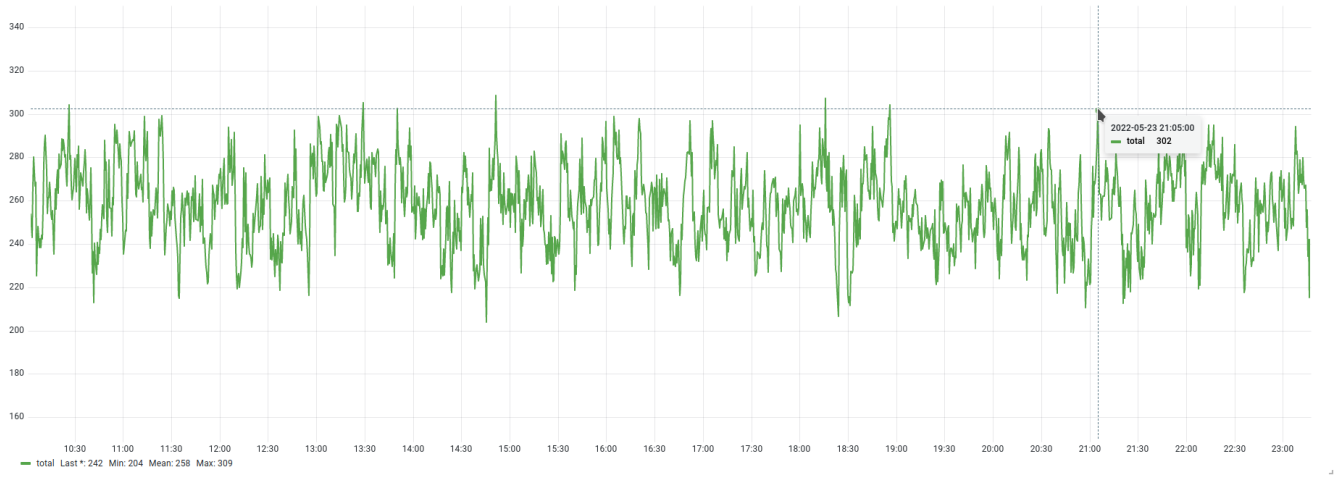


Рис. 3.2. Текущая производительность

Средствами SQL аналогичный результат можно получить следующим запросом:

```
# SELECT
    sum(calls) AS total_calls,
    left(query, 64) AS query_trunc
  FROM pg_stat_statements
  GROUP BY query
  ORDER BY sum(calls) DESC
  LIMIT 10;
 total_calls |                            query_trunc
-------------+------------------------------------------------------------------
    24525219 | SELECT abalance FROM pgbench_accounts WHERE aid = $1
    24525219 | BEGIN
    24525219 | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid =
    24525219 | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid =
    24525217 | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid =
    24525213 | INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
    24525213 | END
      270642 | SELECT setting FROM pg_settings WHERE name = $1
      225535 | SELECT datname FROM pg_database WHERE NOT datistemplate AND data
      135321 | SELECT extnamespace::regnamespace FROM pg_extension WHERE extnam
```

Результат запроса показывает десять наиболее часто выполняемых запросов. Однако здесь мы снова видим «большие» цифры, которые указывают на общую статистику, накопленную за весь период сбора. Это может быть полезно для построения отчетов, но в оперативной ситуации нагрузка непостоянна, и важно видеть изменение частоты выполнения запросов.

С помощью графиков можно увидеть эту картину в динамике, и для начала можем воспользоваться следующим запросом:

```
# topk_avg(5,
  sum by (queryid,query) (
    rate(postgres_statements_calls_total{service_id="primary"}[1m])
    + on(database,user,queryid) group_left(query)
        0 * postgres_statements_query_info{service_id="primary"}
  ), "other")
```

На основе запроса получается следующий график (рис. 3.3):

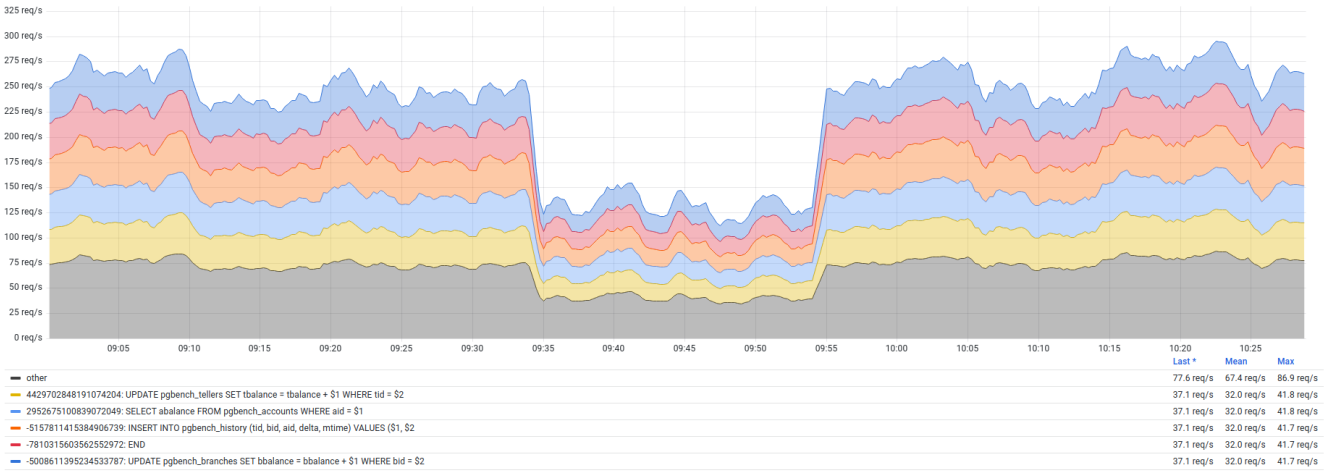


Рис. 3.3. Наиболее вызываемые запросы

На этом графике тоже показана текущая производительность, но уже с большей детализацией. Здесь видны те запросы, которые занимают наибольшую долю рабочей нагрузки и выполняются чаще остальных. Остальные запросы также присутствуют на графике в виде объединенной метрики с меткой `other`. В случае количественного изменения рабочей нагрузки, например при изменении количества экземпляров приложений (или одновременно работающих пользователей), это изменение будет заметно: просадка на графике — это следствие выключения одного из приложений. Однако важно понимать, что большое количество выполненных запросов не означает большого потребления ресурсов, и далее мы рассмотрим примеры, как получить информацию и об использовании ресурсов, и о тех запросах, что занимают существенную долю в рабочей нагрузке.

Запросы с наибольшим количеством затронутых строк. Сортировка запросов по количеству затронутых строк полезна при анализе OLTP-нагрузки. При таком виде нагрузки запросы обычно возвращают немного строк. Если запросы возвращают большое количество строк (тысячи и более), это может быть признаком ошибки в дизайне приложения: маловероятно, что клиенту, который выполнил запрос, нужны все эти тысячи строк. Возможно, в запросе не указано ключевое слово `LIMIT`, либо приложение обрабатывает строки своими силами, хотя это можно было сделать на стороне СУБД. Выявить такие запросы можно следующим образом:

```
# SELECT
    sum(rows) AS total_rows,
    left(query, 64) AS query_trunc
  FROM pg_stat_statements
  GROUP BY query
  ORDER BY sum(rows) DESC
  LIMIT 10;
 total_rows |                              query_trunc
------------+-----------------------------------------------------------------
   27968942 | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid =
   27968942 | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid =
   27968942 | SELECT abalance FROM pgbench_accounts WHERE aid = $1
   27968940 | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid =
   27968936 | INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
   17857314 | SELECT name, setting, unit, vartype FROM pg_show_all_settings()
    5415236 | SELECT d.datname AS database, pg_get_userbyid(p.userid) AS user,
    2768140 | SELECT coalesce(username, backend_type) AS user, datname AS datab
    2000000 | update pgbench_accounts set abalance = abalance + $1
     514620 | SELECT datname FROM pg_database WHERE NOT datistemplate AND data
```

Как и в случае с сортировкой по количеству выполненных запросов, в поле `total_rows` видим большие значения. В мониторинге можно увидеть эту картину в динамике с помощью запроса:

```
# topk_avg(5,
  sum by (queryid,query) (
    rate(postgres_statements_rows_total{service_id="primary"}[1m])
    + on(database,user,queryid) group_left(query)
      0 * postgres_statements_query_info{service_id="primary"}
  ), "other")
```

На рис. 3.4 показан соответствующий график:

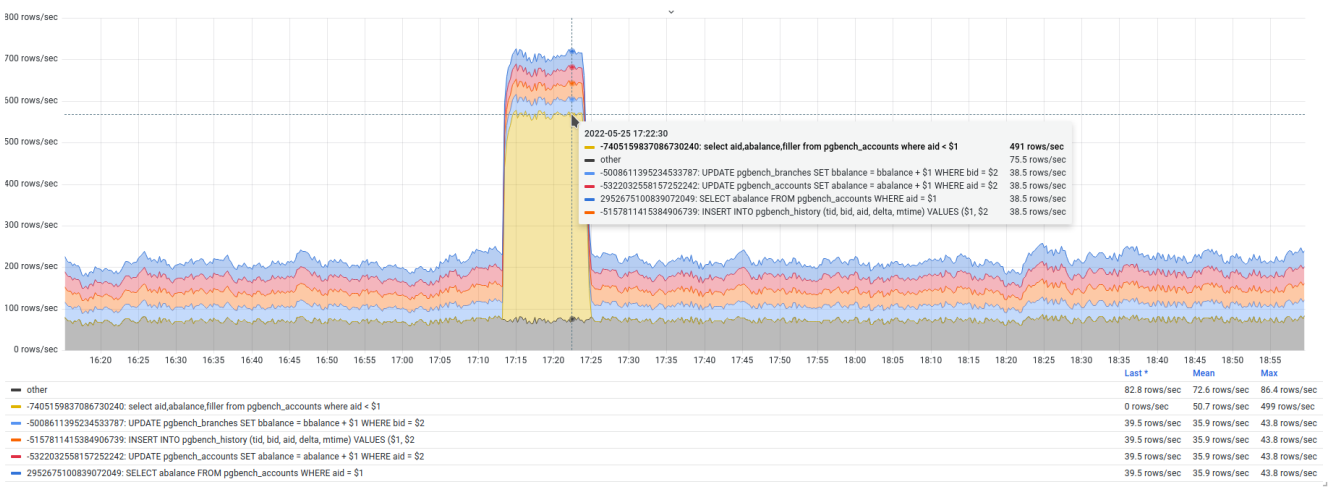


Рис. 3.4. Запросы, затрагивающие наибольшее количество строк

При стабильной рабочей нагрузке без сильных колебаний этот график может выглядеть более или менее ровно, при этом практически любой аномальный запрос выделяется довольно хорошо. В данном примере это можно наблюдать по простому запросу, который был запущен в цикле с помощью команды \watch 1 в psql:

```
# SELECT aid, abalance, filler
FROM pgbench_accounts WHERE aid < 500;
```

Этот запрос возвращает примерно 490 строк в секунду, в то время как его ближайшие соседи — в среднем по 35 строк, и это хорошо заметно на графике. На практике так обычно выглядят запросы с забытым ключевым словом LIMIT. В запросе, например, может использоваться сортировка, и для выполнения условий запроса и последующей сортировки СУБД просканирует большое количество строк и, более того, отправит их клиенту, что вызовет избыточный расход ресурсов не только на стороне СУБД, но и на стороне приложения. Если это какой-то регулярно выполняемый запрос, а не разовая задача, то нужно проанализировать запрос и его соответствие техническому заданию. Если запрос написан именно так, как и задумано, стоит проверить бизнес-требования функции, для которой используется этот запрос, — насколько оправданно обращение к такому большому количеству строк (даже для разбиения на страницы есть способы читать ровно столько, сколько нужно).

Запросы с наибольшим временем исполнения. Время исполнения является наиболее важной характеристикой запроса как элемента рабочей нагрузки. Уменьшение времени исполнения запроса приведет к увеличению производительности СУБД: за счет высвободившихся ресурсов можно будет выполнить дополнительные запросы. Для получения запросов, на выполнение которых было затрачено больше всего времени, можно использовать запрос, подобный тому, что использовался в предыдущих разделах:

```
# SELECT
    to_char(
        interval '1 millisecond' * sum(total_exec_time), 'HH24:MI:SS'
    ) AS exec_time,
    left(query, 64) AS query_trunc
  FROM pg_stat_statements
  GROUP BY query ORDER BY sum(total_exec_time) DESC LIMIT 10;
 exec_time |                          query_trunc
-----------+---------------------------------------------------------------
 105:41:06 | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid =
 14:14:38  | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid =
 04:00:12  | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid =
 00:53:14  | SELECT current_database() AS database, schemaname AS schema, fun
 00:45:37  | SELECT current_database() AS database, s1.schemaname AS schema,
 00:43:06  | SELECT current_database() AS database, schemaname AS schema, rel
 00:26:14  | SELECT coalesce(datname, $1) AS database, xact_commit, xact_roll
 00:25:45  | SELECT checkpoints_timed, checkpoints_req, checkpoint_write_time
 00:25:44  | SELECT pg_is_in_recovery()::int AS recovery, wal_records, wal_fp
 00:25:17  | SELECT archived_count, failed_count, extract($1 from now() - las
```

Напомню, что поле `total_exec_time` хранит время в миллисекундах, и в этом запросе оно для удобства преобразовано в привычный формат представления времени. Также для учета можно дополнительно включить и время планирования `total_plan_time`, получив суммарное время выполнения запроса, включая планирование.

В результате можно заметить существенно выделяющийся запрос: его суммарное время выполнения составило примерно 105 часов с отрывом от второго места в семь раз. Выполнение этого запроса и занимает большую часть рабочей нагрузки. Однако важным нюансом, который остался за кадром, является ответ на вопрос: а за какой интервал времени представлена эта статистика? Ведь 105 машинных часов в рамках одного дня или одного месяца — это совершенно разная рабочая нагрузка. Для получения ответа на этот вопрос можно обратиться к `pg_stat_statements_info.stats_reset`:

```
# SELECT now() - stats_reset FROM pg_stat_statements_info;
        ?column?
------------------------
 17 days 08:11:35.489621
```

В данном случае статистика накоплена за 17 дней; получается, что интересующий нас запрос в течение суток выполнялся примерно шесть часов. В принципе, это немного, учитывая, что остается 18 часов, а если умножить часы на количество доступных процессорных ядер, то и еще больше.

Давайте перейдем к мониторингу и получим время выполнения запросов в динамике:

```
# topk_avg(5,
  sum by (queryid,query) (
    rate(postgres_statements_time_seconds_total{
        service_id="primary",mode="executing"
    }[1m]) + on(database,user,queryid) group_left(query)
        0 * postgres_statements_query_info{service_id="primary"}
  ), "other")
```

На основе запроса можно получить график, изображенный на рис. 3.5.

График показывает изменение времени выполнения (метка `mode="executing"`) для отдельных запросов, и по нему видно, что бóльшую часть времени СУБД занята выполнением однотипных запросов. На выполнение остальных требуется существенно меньше времени. Также здесь следует обратить внимание на сильное колебание времени. Для интересующего нас запроса оно находится в диапазоне от 83 до 1900 миллисекунд, и это подозрительно. В более или менее стабильной рабочей нагрузке диапазон изменений времени выполнения должен быть существенно меньшим (стандартное отклонение всей выборки должно быть близко к нулю). Дело в том, что время выполнения включает в себя и время ожидания, а из предыдущей главы мы знаем, что в тестовой рабочей нагрузке у нас довольно часто возникают ожидания блокировок. Из-за ожидания соседних транзакций время выполнения запросов сильно колеблется, что и отражается на графике.

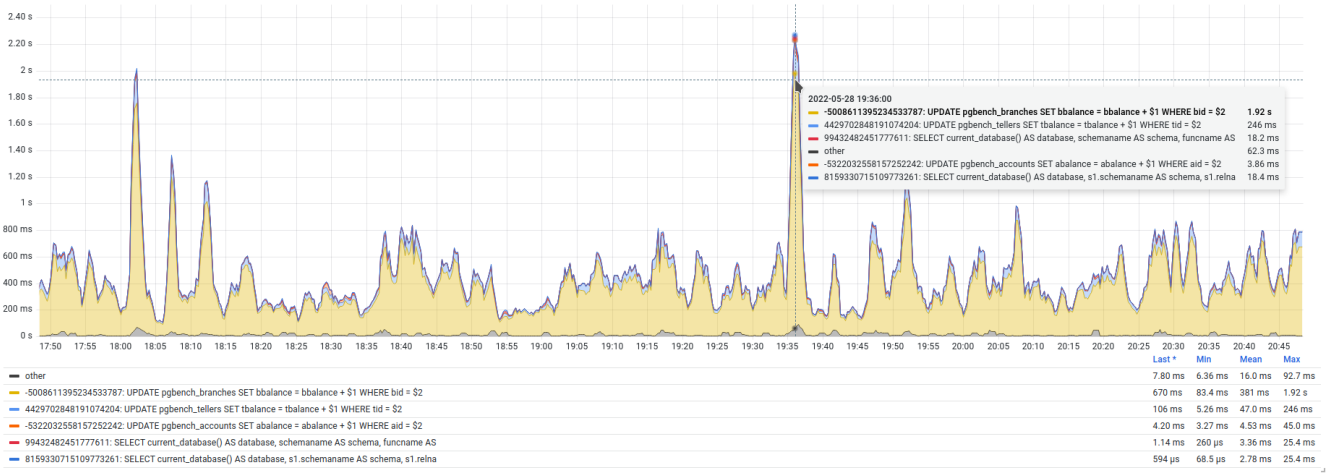


Рис. 3.5. Запросы с наибольшим временем выполнения

На практике этот график является одним из наиболее востребованных, поскольку позволяет быстро выявить запросы, которые требуют внимания, и сразу приступить к их оптимизации. Часто время выполнения запроса удается значительно сократить, просто построив индекс. В более сложных случаях приходится делать рефакторинг запроса или, как в случае с тестовой нагрузкой, предпринимать меры для уменьшения конкуренции за ресурсы.

Запросы с наибольшим временем ввода-вывода. Время, затраченное на блочный ввод-вывод, является еще одной важной характеристикой рабочей нагрузки. Она, в числе прочего, показывает, сколько времени СУБД тратит на чтение данных в буферный кеш (shared buffers) и на запись данных из него, и в какой-то мере говорит о том, насколько вообще достаточно буферного кеша. Если данных нет в кеше, СУБД обратится к операционной системе, что, скорее всего, приведет к чтению с диска. Обращение к диску будет медленнее, чем обращение к кешу в памяти, и это негативно отразится на времени выполнения запроса. Есть шанс, что нужные данные окажутся в страничном кеше (page cache), и тогда их чтение также будет быстрым, хоть и с некоторыми накладными расходами.

В `pg_stat_statements` есть несколько полей, которые отображают время, затраченное на блочный ввод-вывод:

- `blk_read_time` и `blk_write_time` учитывают время ввода-вывода, связанного со страницами в общем и локальных кешах;
- `temp_blk_read_time` и `temp_blk_write_time` учитывают время ввода-вывода, связанного со страницами во временных файлах.

Учет статистики ведется только при включенном параметре *track_io_timing*. По опыту, всегда рекомендуется его включать (однако накладные расходы могут быть на уровне 1 %). За счет этой статистики появляется возможность выявить те запросы, которым регулярно требуется доступ к диску, и запланировать оптимизацию или увеличение ресурсов.

Можно вычислять как отдельные статистики по чтению и записи, так и одну общую. Обычно на практике используется второй вариант:

```
# SELECT
    to_char(
        interval '1 millisecond' * sum(
            blk_read_time + blk_write_time + temp_blk_read_time + temp_blk_write_time
        ), 'HH24:MI:SS.MS'
    ) AS io_time,
    left(query, 64) AS query_trunc
  FROM pg_stat_statements
  WHERE blk_read_time + blk_write_time + temp_blk_read_time + temp_blk_write_time > 0
  GROUP BY query
  ORDER BY sum(blk_read_time + blk_write_time + temp_blk_read_time + temp_blk_write_time) DESC
  LIMIT 5;
   io_time    |                          query_trunc
--------------+---------------------------------------------------------------
 01:25:08.755 | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid =
 00:00:02.125 | INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
 00:00:00.121 | truncate pgbench_history
 00:00:00.050 | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid =
 00:00:00.003 | SELECT current_database() AS database, schemaname AS schema, fun
```

Бóльшую часть времени ввод-вывод выполняется на одном типе запросов. Это другой запрос, не тот, что встретился в предыдущей части и на выполнение которого тратилось больше всего времени. Рассматривая подобные запросы, нужно учитывать частоту их выполнения. Например, обращение к диску может быть приемлемо для запросов, которые выполняются редко и затрагивают холодные (архивные) данные. К регулярно выполняющимся запросам, наподобие запроса, обнаруженного в тестовом окружении, такое постоянное обращение к диску может считаться накладным и увеличивать время выполнения запросов. Устранив обращения к диску или хотя бы уменьшив время обращения, можно ускорить выполнение запросов. В зависимости от характера обращений есть различные варианты: увеличить буферный кеш, так, чтобы все необходимые горячие данные полностью помещались в него; модернизировать системные ресурсы (увеличить память, использовать более производительный накопитель, виртуальную машину и т. п.). Самый лучший вариант, конечно же, провести анализ запроса и оптимизацию.

В мониторинге запросы с наибольшим временем блочного ввода-вывода можно получить следующим запросом:

```
# topk_avg(5,
  sum by (queryid,query) (
    rate(postgres_statements_time_seconds_total{
        service_id="primary",mode=~"ioread|iowrite"
    }[1m]) + on(database,user,queryid) group_left(query)
        0 * postgres_statements_query_info{service_id="primary"}
  ), "other")
```

Обратите внимание, что в запросе для метки `mode` используется перечисление значений `ioread` и `iowrite`; при необходимости можно получить время только чтения или только записи.

Используя этот запрос, можно увидеть ввод-вывод в динамике (рис. 3.6):

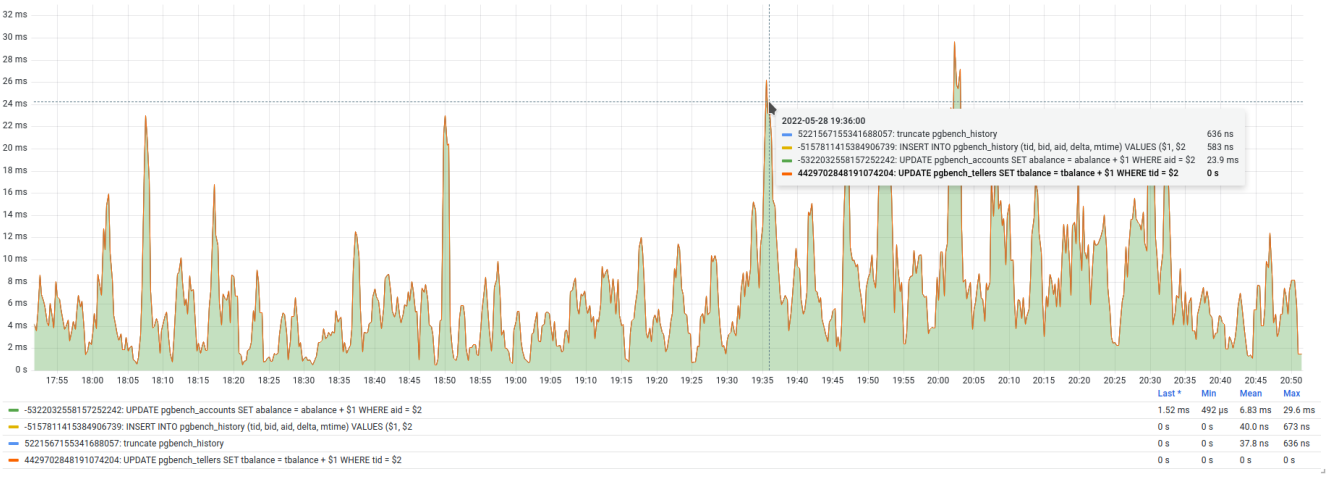


Рис. 3.6. Запросы с наибольшим временем ввода-вывода

Картина на графике сходится с результатом SQL-запроса. Только одному запросу постоянно требуется время на ввод-вывод, и в среднем это 6,83 миллисекунды за одну секунду реального времени. Может показаться, что это не так уж и много, однако стоит помнить, что дешевые обращения к страничному кешу ОС также расцениваются как блочный ввод-вывод. Можно провести эксперимент: сбросить страничный кеш и посмотреть, как изменится время ввода-вывода, когда СУБД будет вынуждена действительно обращаться к диску. В ОС Linux сбросить кеш можно следующей командой, выполненной от пользователя root (не делайте этого в производственных окружениях, это негативно скажется на производительности):

```
# echo 3 > /proc/sys/vm/drop_caches
```

Через пару минут, когда данные попадут в мониторинг, можно увидеть, как выросло время ввода-вывода до 1,16 секунды в пике (рис. 3.7). По мере продолжения рабочей нагрузки и наполнения буферного и страничного кешей время ввода-вывода будет постепенно уменьшаться и в какой-то момент окончательно придет к значению, которое было до сброса страничного кеша.

Запросы с наибольшим временем выполнения без учета ввода-вывода. Учитывая возможность получить время, потраченное на блочный ввод-вывод, можно с помощью простой арифметики получить и время, условно потраченное процессором на выполнение запроса. Условно, потому что полученная статистика хоть и будет указывать на время выполнения, но все-таки она не будет являться точным отражением того, сколько времени процессор провел в активном состоянии, выполняя этот тип запроса. Другой важный момент: это время будет включать в себя и время ожидания. Точную статистику по использованию процессора можно получить

Рис. 3.7. Время ввода-вывода после сброса страничного кеша

с помощью модуля pg_stat_kcache, использующего системный вызов getrusage. Однако это расширение не является официальным и устанавливается как отдельный пакет.

Итак, для получения статистики следует из общего времени total_exec_time вычесть время, потраченное на блочный ввод-вывод:

```
# SELECT
    to_char(
        interval '1 millisecond' * sum(total_exec_time - (
            blk_read_time + blk_write_time + temp_blk_read_time + temp_blk_write_time)),
        'HH24:MI:SS') AS cpu_time,
    left(query, 64) AS query_trunc
  FROM pg_stat_statements
  GROUP BY query
  ORDER BY sum(total_exec_time - (
    blk_read_time + blk_write_time + temp_blk_read_time + temp_blk_write_time)
  ) DESC LIMIT 10;
 cpu_time  |                          query_trunc
-----------+---------------------------------------------------------------
 105:41:06 | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid =
 14:14:38  | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid =
 02:35:03  | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid =
 00:53:14  | SELECT current_database() AS database, schemaname AS schema, fun
 00:45:37  | SELECT current_database() AS database, s1.schemaname AS schema,
 00:43:06  | SELECT current_database() AS database, schemaname AS schema, rel
 00:26:14  | SELECT coalesce(datname, $1) AS database, xact_commit, xact_roll
 00:25:45  | SELECT checkpoints_timed, checkpoints_req, checkpoint_write_time
 00:25:44  | SELECT pg_is_in_recovery()::int AS recovery, wal_records, wal_fp
 00:25:17  | SELECT archived_count, failed_count, extract($1 from now() - las
```

Здесь процессорное время практически совпадает с общим. Это значит, что СУБД тратит на блочный ввод-вывод относительно немного времени, всего около 1 %:

```
# SELECT
    sum(blk_read_time + blk_write_time + temp_blk_read_time + temp_blk_write_time
    ) / sum(total_exec_time) AS io_percent,
    sum(total_exec_time -
        (blk_read_time + blk_write_time + temp_blk_read_time + temp_blk_write_time)
    ) / sum(total_exec_time) AS cpu_percent
  FROM pg_stat_statements;
     io_percent      |    cpu_percent
---------------------+--------------------
 0.01101825786144295 | 0.9889817421385566
```

При использовании SQL для оценки времени выполнения запроса всегда полезно выводить обе статистики вместе, чтобы видеть, на что именно тратится время.

```
# SELECT
    to_char(
        interval '1 millisecond' * sum(total_exec_time),
        'HH24:MI:SS'
    ) AS exec_time,
    (100 * sum(
            blk_read_time + blk_write_time +
            temp_blk_read_time + temp_blk_write_time
        ) / sum(total_exec_time))::numeric(5,2)::text || ' / ' ||
    (100 * sum(total_exec_time - (
            blk_read_time + blk_write_time +
            temp_blk_read_time + temp_blk_write_time)
        ) / sum(total_exec_time))::numeric(5,2) AS "io / cpu, %",
    left(query, 48) AS query_trunc
  FROM pg_stat_statements
  GROUP BY query ORDER BY sum(total_exec_time) DESC LIMIT 10;
 exec_time |  io / cpu, %  |                  query_trunc
-----------+---------------+-----------------------------------------------
 105:50:01 | 0.00 / 100.00 | UPDATE pgbench_branches SET bbalance = bbalance
 14:15:46  | 0.00 / 100.00 | UPDATE pgbench_tellers SET tbalance = tbalance +
 04:00:25  | 35.47 / 64.53 | UPDATE pgbench_accounts SET abalance = abalance
 00:53:21  | 0.00 / 100.00 | SELECT current_database() AS database, schemanam
 00:45:44  | 0.00 / 100.00 | SELECT current_database() AS database, s1.schema
 00:43:12  | 0.00 / 100.00 | SELECT current_database() AS database, schemanam
 00:26:18  | 0.00 / 100.00 | SELECT coalesce(datname, $1) AS database, xact_c
 00:25:49  | 0.00 / 100.00 | SELECT checkpoints_timed, checkpoints_req, check
 00:25:48  | 0.00 / 100.00 | SELECT pg_is_in_recovery()::int AS recovery, wal
 00:25:21  | 0.00 / 100.00 | SELECT archived_count, failed_count, extract($1
```

Здесь сразу видно отношение времени, затраченного на ввод-вывод и CPU относительно друг друга, и запрос, в котором на ввод-вывод тратится примерно треть от общего времени выполнения, довольно хорошо бросается в глаза. Если этому типу запроса дать возможность находить нужные данные в буферном кеше, то из четырех часов суммарного выполнения высвободится примерно 80 минут. Это время можно потратить на выполнение других запросов и увеличить производительность.

Статистика планирования и выполнения — это не единственное, что есть в pg_stat_statements. В этом представлении также можно найти статистику ввода-вывода и записи WAL, однако ее рассмотрение мы продолжим в других главах.

3.6. Сквозная идентификация с queryid

Информацию по выполняемым в СУБД запросам можно получить из нескольких источников:

- pg_stat_activity — запросы, выполняемые в сеансах;
- pg_stat_statements — кумулятивная статистика по успешно выполненным запросам;
- журнал сообщений — запросы независимо от статуса их завершения (успех или ошибка);
- вывод команды EXPLAIN — план выполнения запроса.

Однако долгое время эти источники не были связаны друг с другом. Чтобы получить целостную картину о конкретном типе запроса, необходимо обработать данные из всех источников. Но в pg_stat_activity и журналах сообщений указываются полные тексты запросов, а в pg_stat_statements сохраняются нормализованные запросы, и сопоставить их было непростой задачей. Расширение pg_stat_statements умеет рассчитывать идентификатор запроса queryid, и начиная с версии 14 этот идентификатор стал доступен в остальных источниках — теперь выполнять идентификацию запроса по всем возможным источникам стало гораздо проще.

Расчет идентификатора зависит от параметра *compute_query_id*. Значение по умолчанию auto разрешает расширению pg_stat_statements (и аналогичным) автоматически вычислять идентификаторы. При включении *compute_query_id* идентификатор запроса будет отображаться в pg_stat_activity, журналировании через csvlog (или при указании формата в *log_line_prefix*) и в выводе EXPLAIN VERBOSE.

При реализации сквозной идентификации имена полей в представлениях получились разными — pg_stat_activity.query_id и pg_stat_statements.queryid.

3.7. Построение отчетов на основе pg_stat_statements

Для оценки эксплуатации системы за длительные периоды могут быть полезны сводные или суммарные отчеты. Как правило, эти отчеты в разных проекциях показывают, насколько эффективно система справляется с рабочей нагрузкой. С их помощью можно не только быстро оценить состояние системы, но и отследить динамику относительно предыдущих периодов. Такие отчеты можно строить на основе статистики по запросам из pg_stat_statements, учитывая ее накопительный характер.

Концепция такого отчета довольно проста: нужно определить суммарное использование ресурсов по каждой проекции в pg_stat_statements и затем для каждого отдельного запроса определить его вклад в эту сумму. Полученные доли использования ресурсов следует отсортировать по какому-либо критерию, например, по суммарному времени выполнения запроса.

Примеры скриптов для создания подобных отчетов можно найти в репозитории DataEgret[1]. Отчет содержит список запросов, выполнение которых заняло больше всего времени, обогащенный подробной информацией из pg_stat_statements. Вот пример такого отчета по запросам в тестовом окружении:

```
# \i /var/lib/postgresql/scripts/query_stat_total.sql
Output format is unaligned.
?column?
total time:     11:50:59 (IO: 1.29%)
total queries:  25,411,617 (unique: 62)
report for all databases, version 0.9.5 @ PostgreSQL 15beta1
tracking all 5000 queries, utilities on, logging 500ms+ queries

============================================================================================================
pos:1    total time: 05:12:05 (43.9%, CPU: 44.5%, IO: 0.0%)      calls: 1,800,650 (7.09%) avg_time: 10.40ms (IO: 0.0%)
user: classic    db: pgbench     rows: 1,800,650 (8.37%)         query:
UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2

============================================================================================================
pos:2    total time: 02:53:43 (24.4%, CPU: 24.8%, IO: 0.0%)      calls: 996,873 (3.92%)  avg_time: 10.46ms (IO: 0.0%)
user: maru       db: pgbench     rows: 996,873 (4.64%)   query:
UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2

============================================================================================================
pos:3    total time: 00:50:08 (7.1%, CPU: 7.1%, IO: 0.0%)        calls: 496,689 (1.95%)  avg_time: 6.06ms (IO: 0.0%)
user: pgbench    db: pgbench     rows: 496,689 (2.31%)   query:
UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2

... остальной вывод опущен ...
```

Отчет состоит из двух частей. Первая часть показывает общую информацию:

- суммарное время, затраченное на успешное выполнение всех запросов, причем отдельно отмечена доля ввода-вывода;
- общее количество успешно выполненных запросов с указанием количества уникальных типов запросов;
- версия отчета и версия PostgreSQL;
- настройки pg_stat_statements.

Вторая часть показывает запросы, отсортированные по общему времени выполнения. Каждый запрос сопровождается дополнительной информацией:

[1] github.com/dataegret/pg-utils/blob/master/sql/global_reports/query_stat_total_13.sql

- вклад запроса в общее время выполнения с разделением на CPU и ввод-вывод;
- количество вызовов запроса с указанием доли в общем объеме всех вызовов;
- среднее время выполнения запроса с отдельным учетом ввода-вывода;
- информация о пользователе, базе данных и количестве затронутых строк;
- нормализованный текст запроса.

Полученной информации достаточно, чтобы оценить доли запросов в рабочей нагрузке и при необходимости перейти к анализу производительности и рефакторингу. При желании скрипт можно модифицировать и добавить другую необходимую информацию, например период, за который взята статистика, идентификаторы запросов, использование ресурсов, объем сгенерированных журнальных записей и т. д.

В качестве другого примера можно взять вывод отчета из pgcenter, где для любого интересующего запроса из `pg_stat_statements` можно получить сводную информацию:

```
summary:
    total queries: 25,421,767
    total rows: 21,512,607
    total WAL: 25 GB
    total_time: 11:51:10, 100%
        total_plan_time: 00:26:34,  3.74%
        total_cpu_time: 11:15:27,  94.98%
        total_io_time: 00:09:08,  1.29%

query info:
    queryid:                              2368592290184574017
    username:                             classic,
    database:                             pgbench,
    calls (relative to total):            1,801,358,  7.09%,
    rows (relative to total):             1,801,358,  8.37%,
    WAL usage (relative to total):
        records:                          2,011,320,  10.88%
        full-page images:                 849,  0.03%
        bytes:                            144 MB,  0.56%
    total times (relative to total):      05:12:10.007,  43.89%
        planning:                         00:01:40.461,  6.30%
        cpu:                              05:10:29.546,  45.97%
        io:                               00:00:00.000,  0.00%
    average times (in-query distribution): 10.40ms,  100%
        planning:                         0.06ms,  0.54%
        cpu:                              10.34ms,  99.46%
        io:                               0.00ms,  0.00%

    query text:
        UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2
```

В фокусе отчета pgcenter находится всего лишь один выбранный запрос. В первой, верхней части отчета приведена сводная информация, а во второй части показаны статистика по выбранному запросу и его вклад в общую рабочую нагрузку:

- количество вызовов и затронутых строк с соответствующими долями;
- объем сгенерированных журнальных записей с детализацией по количеству записей, байтам и FPI-страницам (см. *full_page_writes*[1]);
- суммарное время выполнения всех запросов этого типа с детализацией по планированию, выполнению и вводу-выводу;
- среднее время выполнения запросов этого типа с детализацией;
- нормализованный текст запроса.

Сравнивая оба отчета, можно заметить, что они различаются предназначением. Отчет Data-Egret предлагает более полную картину о том, что выполняется в СУБД, и больше подходит для роли сводного отчета. Отчет pgcenter предлагает оценку использования ресурсов конкретным запросом и больше ориентирован на оперативный анализ и выявление деталей по подозрительным запросам.

При построении подобных отчетов и расчете суммарного использования ресурсов важно помнить, что расширение `pg_stat_statements` может собирать статистику выполнения не только верхнеуровневых, но и вложенных запросов, и это поведение регулируется параметром *pg_stat_statements.track*. В случае более детального отслеживания со значением `all` есть риск посчитать часть статистики дважды, например, при наличии запросов с функциями, внутри которых выполняются другие запросы или функции. Подсчет общей статистики будет включать статистику как самой функции, так и всех вложенных в нее запросов, что может сильно исказить результат. Для исключения этой проблемы следует учитывать флаг `toplevel`, который указывает на уровень выполнения и помогает исключить вложенные запросы (`toplevel = false`). К сожалению, флаг `toplevel` появился в версии 14, поэтому в более ранних версиях для отчетов рекомендуется установить *pg_stat_statements.track* = `top`.

3.8. Представление pg_stat_statements_info

В дополнение к основному представлению у расширения есть еще одно небольшое служебное представление — `pg_stat_statements_info`, состоящее всего из одной строки с двумя полями:

- `dealloc` — количество раз, когда расширение было вынуждено частично удалить статистику из-за достижения ограничения *pg_stat_statements.max*. При этом возникает риск потери некоторой статистики, что может искажать данные отчетов. В качестве обходного

[1] postgrespro.ru/docs/postgrespro/current/runtime-config-wal#GUC-FULL-PAGE-WRITES

пути источником данных может быть система мониторинга. Если ограничение еще не достигнуто, проблему можно решить на уровне конфигурации СУБД, увеличив значение *pg_stat_statements.max*, но это потребует перезапустить сервер.

- stats_reset — отметка времени последнего сброса статистики. Позволяет четко представлять, за какой интервал времени накоплена статистика по запросам.

Пример вывода:

```
# SELECT dealloc, now() - stats_reset AS age FROM pg_stat_statements_info;
 dealloc |           age
---------+------------------------
       0 | 18 days 11:51:27.117115
```

В приведенном примере видно, что существующей емкости достаточно и удаления статистики не происходило (как минимум за последние 18 дней).

3.9. Выполнение процедур и функций

Для реализации сложной логики при обработке данных можно использовать функции. Такие функции могут быть написаны на SQL и могут объединять в себе несколько запросов. Кроме SQL поддерживаются и такие процедурные языки (procedural language, PL), как PL/pgSQL, PL/Tcl, PL/Perl и PL/Python. Когда в рабочей нагрузке присутствуют запросы с вызовами функций, перед администратором возникает задача поиска медленных функций и их оптимизации. Для простейшего анализа производительности функций СУБД есть представление pg_stat_user_functions, которое содержит необходимый минимум статистики по выполнению функций. Для более детального профилирования функций придется использовать сторонние инструменты.

Основной сценарий использования pg_stat_user_functions — это оптимизация производительности не отдельных запросов, а приложений. С помощью представления можно выявить наиболее горячие функции, которые чаще всего вызываются целевым приложением, и оптимизировать их, тем самым улучшая производительность приложения.

Совместно с pg_stat_user_functions можно использовать и pg_stat_statements. Во втором представлении можно найти запросы, в которых вызывается функция, и по имеющимся полям получить информацию о времени выполнения и используемых ресурсах.

Для управления сбором статистики выполнения функций в конфигурации есть параметр *track_functions*, который может принимать одно из следующих значений (для вступления изменений в силу достаточно перезагрузки конфигурации):

- none — выключает отслеживание статистики (значение по умолчанию);
- pl — включает отслеживание только функций на процедурных языках;
- all — включает отслеживание всех функций, включая функции на языках SQL и C.

Стоит отметить, что SQL-функции, которые встраиваются (inline) в вызываемый запрос, не отслеживаются при любом значении параметра.

Представление pg_stat_user_functions содержит относительно небольшой объем данных:

- funcid — уникальный идентификатор функции;
- schemaname — имя схемы, которой принадлежит функция;
- funcname — имя функции, которое может быть неуникальным внутри схемы, поскольку допускаются перегруженные функции, отличающиеся набором входных параметров;
- calls — общее количество вызовов функции;
- total_time — общее время выполнения функции (в миллисекундах), включая и время выполнения вложенных функций, если таковые имеются;
- self_time — общее время выполнения самой функции (в миллисекундах) без учета времени выполнения вложенных функций.

Как я уже отмечал, основной сценарий использования этой статистики — это определение наиболее часто вызываемых функций и функций, выполнение которых занимает больше всего времени. В тестовой нагрузке на основе стандартного сценария pgbench отсутствуют вызовы функций, поэтому статистики будет немного и она будет включать в себя только вызовы функций из расширений:

```
# SELECT * FROM pg_stat_user_functions;
 funcid | schemaname |        funcname         | calls | total_time | self_time
--------+------------+-------------------------+-------+------------+------------
  16398 | public     | pg_stat_statements_info |     1 |      0.006 |      0.006
  16409 | public     | pg_stat_statements      | 14751 |  14246.418 |  14246.418
  16416 | public     | pg_buffercache_pages    | 28936 | 122401.529 | 122401.529
```

В этом запросе нет дополнительных условий или сортировок, так как я заранее знаю, что в статистике немного данных. Есть статистика только по трем функциям, принадлежащим двум расширениям: pg_buffercache и pg_stat_statements. Две из них запрашиваются агентом мониторинга, и в поле calls отражается количество вызовов. Время total_time и self_time для обеих функций совпадает, поскольку в них не используются вложенные вызовы других пользовательских функций. Суммарное время выполнения функций составило примерно 122 и 14 секунд. Функция pg_buffercache_pages вызывается в два раза чаще, однако ее общее время выполнения больше примерно в 8,5 раза, то есть ее выполнение обходится СУБД дороже.

Используя такой подход и сравнивая время выполнения функций, можно находить наиболее долгие. Следующим запросом можно получить статистику из мониторинга:

```
# increase(postgres_function_total_time_seconds_total{service_id="primary"}[1m])
```

В запросе используется `increase` вместо `rate`. Поскольку эти функции вызываются только агентом мониторинга, то очевидно, что вызовы происходят относительно редко, несколько раз в минуту, поэтому с этой точки зрения удобнее видеть, сколько времени тратится на выполнение функций за одну минуту (вместо одной секунды). В случае же производственных нагрузок, где выполнение функций может происходить чаще и конкурентно в нескольких сеансах, использование `rate` может быть более предпочтительным.

На рис. 3.8 отражается то, что можно было видеть в выводе запроса к `pg_stat_user_functions`, — выполнение функции `pg_buffercache_pages` занимает бóльшую часть времени:

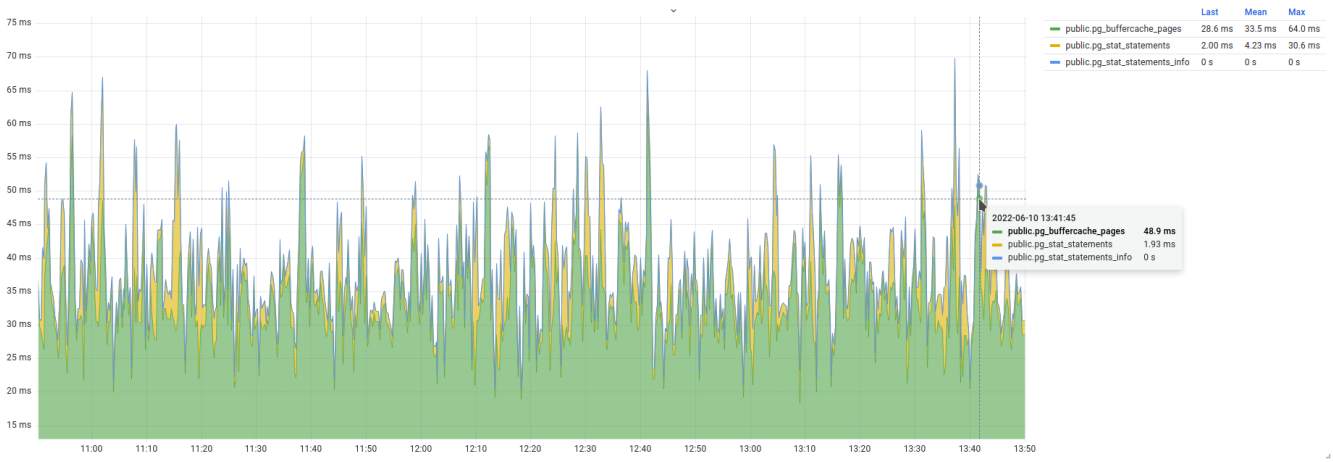


Рис. 3.8. Функции с наибольшим временем выполнения

В среднем за минуту выполнение этой функции занимает 33 миллисекунды, против 4 миллисекунд для функции `pg_stat_statements`.

Резюме

- Запросы являются базовой единицей выполнения рабочей нагрузки.
- Мониторинг запросов обязателен и важен для понимания рабочей нагрузки.
- Для анализа рабочей нагрузки используется расширение `pg_stat_statements`, которое выключено по умолчанию.
- Представление `pg_stat_statements` содержит статистику по планированию и выполнению запросов, утилизации ресурсов, времени ввода-вывода, генерации WAL.
- Представление `pg_stat_statements` является важным источником данных для графиков мониторинга и отчетов о работе СУБД.
- СУБД предоставляет сквозную идентификацию запросов для анализа статистики из разных источников.

- Представление `pg_stat_statements_info` содержит полезную информацию о работе расширения `pg_stat_statements`.
- Статистика выполнения пользовательских функций находится в `pg_stat_user_functions`.

Глава 4
Базы данных

В этой главе мы рассмотрим:

- иерархию объектов СУБД;
- кластер баз данных, табличные пространства, базы данных, схемы, таблицы и индексы;
- отношения;
- служебные структуры отношений — TOAST, карты видимости и свободного пространства;
- рабочую нагрузку в отношении таблиц и индексов;
- избыточное последовательное сканирование таблиц;
- ошибки и нежелательные события в кластере баз данных;
- функции для определения размеров объектов;
- функции вывода содержимого служебных каталогов кластера баз данных.

В предыдущей главе мы рассмотрели запросы как основу любой рабочей нагрузки, а также связанные с ними статистику и мониторинг активности. Другой стороной рабочей нагрузки являются пользовательские данные. В запросах указываются конкретные таблицы, из которых читаются и в которые сохраняются данные. В этой главе мы рассмотрим статистику активности с точки зрения обращений к объектам СУБД и проследим события, которые происходят в контексте именно этих объектов — баз данных, таблиц и индексов.

4.1. Иерархия объектов СУБД

Одной из главных задач СУБД является надежное хранение данных и обеспечение доступа к этим данным. На уровне СУБД средствами организации и хранения данных являются таблицы и другие типы объектов, каждый со своим предназначением. Все эти средства являются логическим слоем организации данных. Однако СУБД не является вещью самой в себе и вынуждена опираться на средства операционной системы, на ее собственные абстракции и ресурсы. С точки зрения операционной системы данные СУБД размещаются в файловой системе и используют блочные устройства. На физическом уровне все таблицы, индексы и многие другие абстракции СУБД — это всего лишь файлы и каталоги на диске.

Дальше мы рассмотрим, как в СУБД организовано хранение данных. Это поможет лучше понять качественный состав рабочей нагрузки (на какие объекты СУБД приходится нагрузка)

и более осознанно подойти к созданию мониторинга рабочей нагрузки с точки зрения использования объектов и данных. Опытные администраторы, хорошо представляющие внутреннюю структуру хранения, могут сразу перейти к разделу 4.2.

Кластер баз данных

Первое, что мы рассмотрим, — это концепция *кластера баз данных* (database cluster). Кластер представляет собой единый и неделимый набор баз данных и общий для них набор глобальных объектов. Свойство единости и неделимости означает, что весь набор существует вместе в рамках одного *экземпляра* СУБД (instance) — группы процессов, запущенных в пространстве операционной системы и имеющих область общей памяти. На одном сервере могут работать несколько экземпляров СУБД, если их TCP-порты не конфликтуют. Кластер баз данных не следует путать с кластером репликации. Кластер репликации обычно состоит из нескольких экземпляров СУБД, запущенных в независимых средах (отдельные физические, виртуальные серверы или контейнеры).

С точки зрения операционной системы кластер баз данных представляет собой набор каталогов и файлов. Инициализация кластера осуществляется командой initdb с указанием целевого каталога. Полученный каталог принято называть *каталогом данных* (data directory), он содержит в себе другие каталоги и файлы кластера. Исключением могут быть каталоги табличных пространств и WAL-журнала, которые могут быть вынесены за пределы каталога данных и связаны с основным каталогом через символические ссылки. Таким образом, каталог данных и все возможные внешние каталоги (табличные пространства и WAL) образуют хранилище данных кластера.

Ниже приведен пример каталога данных из тестового окружения. Каждый подкаталог играет свою значимую роль в функционировании кластера. Администратору важно знать назначение каждого подкаталога и его роль в работе СУБД.

```
# ls -l /var/lib/postgresql/data/
total 132
-rw-------    1 postgres postgres        3 Jul 31 08:52 PG_VERSION
drwx------    7 postgres postgres     4096 Jul 31 08:53 base
-rw-------    1 postgres postgres       46 Aug  2 00:00 current_logfiles
drwx------    2 postgres postgres     4096 Jul 31 08:55 global
drwx------    2 postgres postgres     4096 Jul 31 08:52 pg_commit_ts
drwx------    2 postgres postgres     4096 Jul 31 08:52 pg_dynshmem
-rw-------    1 postgres postgres     4991 Jul 31 08:52 pg_hba.conf
-rw-------    1 postgres postgres     1636 Jul 31 08:52 pg_ident.conf
drwx------    4 postgres postgres     4096 Aug  2 04:13 pg_logical
drwx------    4 postgres postgres     4096 Jul 31 08:52 pg_multixact
drwx------    2 postgres postgres     4096 Jul 31 08:52 pg_notify
drwx------    3 postgres postgres     4096 Jul 31 08:53 pg_replslot
```

```
drwx------    2 postgres postgres     4096 Jul 31 08:52 pg_serial
drwx------    2 postgres postgres     4096 Jul 31 08:52 pg_snapshots
drwx------    2 postgres postgres     4096 Jul 31 08:53 pg_stat
drwx------    2 postgres postgres     4096 Jul 31 08:53 pg_stat_tmp
drwx------    2 postgres postgres     4096 Aug  2 04:08 pg_subtrans
drwx------    2 postgres postgres     4096 Jul 31 08:52 pg_tblspc
drwx------    2 postgres postgres     4096 Jul 31 08:52 pg_twophase
drwx------    3 postgres postgres     4096 Aug  2 04:13 pg_wal
drwx------    2 postgres postgres     4096 Aug  2 01:13 pg_xact
-rw-------    1 postgres postgres       88 Jul 31 08:52 postgresql.auto.conf
-rw-------    1 postgres postgres    30146 Jul 31 08:52 postgresql.conf
-rw-------    1 postgres postgres       24 Jul 31 08:53 postmaster.opts
-rw-------    1 postgres postgres       94 Jul 31 08:53 postmaster.pid
```

pg_wal и pg_xlog

Зачем же нужно знать внутреннюю структуру каталога данных? Хорошим примером является история подкаталога WAL-журнала. Этот каталог критически важен для функционирования СУБД. При некоторых обстоятельствах его содержимое может неконтролируемо расти и занимать все больше и больше места на диске, но удаление каталога может привести к непредсказуемым последствиям. До версии 9.6 включительно каталог журнала WAL назывался pg_xlog, и неопытные администраторы нередко удаляли его, считая, что там хранятся какие-то «обычные текстовые логи», которыми можно пожертвовать, чтобы освободить место. В итоге удаление каталога приводило к печальным последствиям[1].

В версии 10 каталог WAL был переименован в pg_wal в попытке устранить двусмысленность и предотвратить случайные удаления[2]. Заранее зная роль и назначение отдельных подкаталогов внутри основного каталога данных, можно избежать попадания в подобные истории.

Подробнее об организации физического хранения баз данных можно узнать из официальной документации[3].

Табличные пространства

Следующей (внешней) частью каталога данных являются табличные пространства. С их помощью можно объединять независимые каталоги (размещенные на разных файловых системах и блочных устройствах) и использовать для нужд СУБД. После инициализации initdb в СУБД присутствует два табличных пространства:

- `pg_default` — табличное пространство по умолчанию, где будут создаваться все базы, таблицы, индексы. Это пространство размещается внутри основного каталога кластера. При инициализации в этом пространстве создаются две служебные базы — `template0`

[1] stackoverflow.com/questions/36900462/postgres-wont-start-after-deleting-pg-xlog-files
[2] www.depesz.com/2016/10/25/waiting-for-postgresql-10-rename-pg_xlog-directory-to-pg_wal
[3] postgrespro.ru/docs/postgresql/current/storage

и template1, которые используются как шаблоны для создания новых баз (см. команду CREATE DATABASE[1]);

- pg_global — служебное пространство, которое также размещается внутри основного каталога кластера. Здесь размещаются общие для всего кластера объекты системного каталога.

Получить список всех табличных пространств можно с помощью метакоманды \db+ или из таблицы pg_tablespace:

```
# \db+
                                    List of tablespaces
    Name    |  Owner   | Location | Access privileges | Options |  Size  | Description
------------+----------+----------+-------------------+---------+--------+-------------
 pg_default | postgres |          |                   |         | 372 MB |
 pg_global  | postgres |          |                   |         | 556 kB |
```

В выводе команды интерес представляет поле Location, которое содержит путь до каталога табличного пространства. Для пространств по умолчанию это поле содержит NULL (поскольку оба находятся в основном каталоге данных). Все остальные подключаемые пространства (как правило) находятся вне основного каталога данных. Они связаны с основным каталогом посредством символических ссылок, определенных в подкаталоге pg_tblspc.

Табличные пространства могут использоваться для размещения как баз данных, так и отдельных таблиц и индексов. Отдельно стоит отметить базы данных: размещение в конкретном табличном пространстве фактически означает, что в нем размещаются объекты системного каталога этой базы, а остальное содержимое может размещаться и в других табличных пространствах. Например, вполне допустимо существование базы в пространстве pg_default, в которой может находиться большая секционированная таблица; при этом секции и индексы этой таблицы за последние три месяца могут находиться в пространстве, размещенном в быстром SSD-хранилище, а более старые, архивные секции и индексы — в медленном HDD-хранилище. Таким образом, база размещена в одном пространстве, а часть ее данных — в двух других пространствах.

Табличные пространства и стоимость обслуживания

Практика показывает, что использование дополнительных пространств усложняет обслуживание СУБД. Следует помнить о существовании этих пространств и учитывать их в задачах обновления, настройки физической репликации и многих других. Также добавляются дополнительные операции при миграциях данных между табличными пространствами. Лучше до последнего избегать использования пространств и серьезно оценивать плюсы и минусы от их внедрения. На практике в большинстве систем достаточно тех табличных пространств, что создаются по умолчанию.

[1] postgrespro.ru/docs/postgresql/current/sql-createdatabase

Базы данных

Базы данных являются своего рода контейнерами для данных и обеспечивают их изоляцию. Таблицы и индексы одной базы недоступны в других базах (для обхода этого ограничения можно использовать такие средства, как `postgres_fdw` и `dblink`). Отдельные базы используются для логического разделения данных внутри одного кластера — данные нескольких не взаимосвязанных между собой приложений можно разделить и хранить в отдельных базах, но в рамках одного кластера баз данных.

Посмотреть список баз можно с помощью метакоманды `\l` (или в таблице `pg_database`):

```
# \l
                                                List of databases
   Name    |  Owner   | Encoding |  Collate   |   Ctype    | ICU Locale | Locale Provider |   Access privileges
-----------+----------+----------+------------+------------+------------+-----------------+-----------------------
 pgbench   | pgbench  | UTF8     | en_US.utf8 | en_US.utf8 |            | libc            |
 postgres  | postgres | UTF8     | en_US.utf8 | en_US.utf8 |            | libc            |
 template0 | postgres | UTF8     | en_US.utf8 | en_US.utf8 |            | libc            | =c/postgres          +
           |          |          |            |            |            |                 | postgres=CTc/postgres
 template1 | postgres | UTF8     | en_US.utf8 | en_US.utf8 |            | libc            | =c/postgres          +
           |          |          |            |            |            |                 | postgres=CTc/postgres
```

При инициализации кластера создаются три базы данных. Две базы — `template0` и `template1` — используются самой СУБД в качестве шаблонов при создании новых баз, а третья база — `postgres` — является обычной базой, пригодной для создания в ней таблиц. Как правило, БД `postgres` не используется для прикладных бизнес-задач, администраторы обычно устанавливают в нее вспомогательные скрипты и инструменты. Для размещения пользовательских данных рекомендуется создавать отдельные базы и давать им имена, которые ясно указывали бы на назначение БД или на суть хранимых там данных.

Все базы данных внутри основного каталога размещены в подкаталоге base. Каждая база размещена в каталоге, имя которого соответствует идентификатору в `pg_database.oid`.

Схемы

Внутри отдельной базы можно дополнительно разделить данные по *схемам* (schema). Схемы, по сути, являются *пространствами имен* (namespace) и позволяют задействовать дополнительный уровень изоляции и хранения объектов, но не предусматривают вложенности. Например, с помощью схем можно организовать хранение таблиц согласно разным предметным областям, версионирование функций и представлений. Массу интересных вариантов использования схем можно найти в презентации Б. Момджяна, посвященной микросервисам[1].

Любая база создается со схемой `public`, которая является схемой по умолчанию для всех создаваемых таблиц и индексов. Часто на практике достаточно одной этой схемы.

[1] momjian.us/main/writings/pgsql/microservices.pdf

Посмотреть список схем можно метакомандой \dn+ (или в таблице pg_namespace):

```
# \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                   |
```

Схемы являются внутренней абстракцией СУБД для логической группировки объектов. Они не имеют ассоциированных с ними файлов или каталогов в файловой системе.

Таблицы и индексы

Таблицы являются основными, можно даже сказать, главными объектами для размещения пользовательских данных, представленных в виде строк. Индексы являются дополнительной к таблице структурой и позволяют ускорить поиск отдельных строк таблицы.

Получить список таблиц можно с помощью метакоманды \dt+ или представления pg_tables:

```
# \dt+
                                      List of relations
 Schema |       Name       | Type  |  Owner  | Persistence | Access method |  Size   | Description
--------+------------------+-------+---------+-------------+---------------+---------+-------------
 public | pgbench_accounts | table | pgbench | permanent   | heap          | 256 MB  |
 public | pgbench_branches | table | pgbench | permanent   | heap          | 40 kB   |
 public | pgbench_history  | table | pgbench | permanent   | heap          | 0 bytes |
 public | pgbench_tellers  | table | pgbench | permanent   | heap          | 48 kB   |
```

В выводе представлены только пользовательские таблицы. Для просмотра всех таблиц, включая системные, следует указать модификатор S: команда примет вид \dtS+. Для индексов есть аналогичная метакоманда \di и представление pg_indexes (на основе таблицы pg_index):

```
# \di+
                                            List of relations
 Schema |         Name          | Type  |  Owner  |      Table       | Persistence | Access method | Size  | Description
--------+-----------------------+-------+---------+------------------+-------------+---------------+-------+-------------
 public | pgbench_accounts_pkey | index | pgbench | pgbench_accounts | permanent   | btree         | 43 MB |
 public | pgbench_branches_pkey | index | pgbench | pgbench_branches | permanent   | btree         | 16 kB |
 public | pgbench_tellers_pkey  | index | pgbench | pgbench_tellers  | permanent   | btree         | 16 kB |
```

Отдельно можно отметить таблицу pg_class, которая содержит список всех объектов в текущей базе и часто используется как вспомогательная при получении статистики по объектам базы. Далее в некоторых запросах мы будем обращаться к этой таблице.

Отношения и кортежи

Отношение (relation) — это обобщающий термин для любых объектов в базе данных, которые имеют имя и список атрибутов. Таблицы и индексы, внешние таблицы, представления (обычные и материализованные), последовательности (sequence), составные типы — все это можно назвать отношениями. Также в PostgreSQL существует термин *класс* — это устаревший синоним отношения, который остался со времен увлечения объектной ориентированностью. В самых ранних версиях PostgreSQL существовала таблица `pg_relation`, которую быстро переименовали в `pg_class`, а вот префикс `rel` у атрибутов так и остался.

В более общем виде отношение — это множество *кортежей* (tuple); например, результат запроса также является отношением. Кортеж представляет собой упорядоченный набор атрибутов. Порядок атрибутов задается при определении таблицы (или другого отношения), которая будет содержать кортежи. В этом случае кортеж часто называют строкой таблицы. Он также может определяться структурой результирующего множества; такие кортежи иногда называют записями. Часто этот термин принимает смысл «версия строки», так как относится скорее к физическому представлению данных внутри страницы, чем к логическому понятию строки.

И таблицы, и индексы (и некоторые другие типы отношений) представлены в виде отдельных файлов в каталогах БД. Если таблица содержит большое количество строк, файл таблицы (или индекса) автоматически сегментируется на файлы по одному гигабайту. Каждый такой файл-сегмент нумеруется по возрастанию.

Каждая таблица или индекс представлены несколькими файлами, так называемыми *слоями* (fork):

- Основной файл таблицы (main fork) содержит пользовательские данные в виде строк.
- Карта видимости (visibility map) предназначена для отслеживания страниц, строки в которых видны всем активным транзакциям. Карты видимости существуют только для таблиц. Файлы карт видимости имеют суффикс _vm. Карты видимости обновляются при очистке и обычно занимают немного места. Более подробное описание можно найти в документации[1].
- Карта свободного пространства (free space map) предназначена для отслеживания свободного места в таблице (или индексе) и тех страниц, что доступны для вставки новых строк. Так же как и карты видимости, карты свободного пространства обновляются при очистке и занимают немного места. Файлы этих карт имеют суффикс _fsm. Более подробное описание можно найти в документации[2].
- Файл инициализации (initialization fork) используется только для *нежурналируемых таблиц* (unlogged table) и индексов, изменения в которых не фиксируются в WAL-журнале. Такие таблицы не восстанавливаются после сбоя и пересоздаются путем копирования

[1] postgrespro.ru/docs/postgresql/current/storage-vm
[2] postgrespro.ru/docs/postgresql/current/storage-fsm

файла инициализации поверх основного файла (если таблица была большой и содержала несколько сегментов, они удаляются). В документации не так много информации по файлу инициализации[1].

На более низком уровне таблицы и индексы имеют страничную организацию и состоят из страниц (блоков) по 8 КБ. Каждая такая страница имеет служебные поля и место, зарезервированное под пользовательские данные, — строки, или кортежи. Индексы также имеют страничную организацию, но в зависимости от типа индекса (btree, hash, gin и пр.) могут по-разному структурировать место внутри страниц.

TOAST

Одним из ограничений страничной организации является то, что строки не могут выходить за пределы страниц, то есть в странице нельзя хранить строки длиной, превышающей ее размер. Для преодоления этого ограничения используется *методика хранения сверхбольших атрибутов* TOAST (The Oversized-Attribute Storage Technique), которая заключается в сжатии и/или разбиении длинных строк. Это происходит автоматически и прозрачно для конечного пользователя, хотя отмечу, что возможности для тонкой настройки TOAST имеются. Методика TOAST поддерживается только для тех типов данных, которые могут быть представлены атрибутами переменной длины (varlena, variable-length attribute), так как она не имеет смысла для типов фиксированного размера.

Если таблица имеет столбцы с типами переменной длины, с таблицей также будет связана дополнительная TOAST-таблица, в которой и будут размещаться данные, не вмещающиеся в обычные страницы. Как и обычные таблицы, TOAST-таблицы также используют страничную организацию с фиксированным размером страницы, но за счет дополнительного сжатия и разбиения значений на фрагменты методика позволяет распределять большие значения по нескольким страницам. Более подробное описание методики и то, как устроено размещение данных на диске и в памяти, можно прочитать в официальной документации[2].

Мы коротко рассмотрели основные концепции иерархии объектов СУБД. В дальнейшем это позволит нам лучше ориентироваться в статистике, особенно в вопросах, связанных с использованием дискового пространства и общего кеша.

4.2. События в кластере баз данных

В главе 3 рабочая нагрузка рассматривалась с точки зрения выполняемых запросов. Запросы условно можно представить как внешнюю силу, направленную от приложений на СУБД, а рабочую нагрузку — как внутреннюю работу, совершаемую СУБД под действием этой силы. Это

[1] postgrespro.ru/docs/postgresql/current/storage-init
[2] postgrespro.ru/docs/postgresql/current/storage-toast

позволяет взглянуть на рабочую нагрузку совершенно с другой стороны — например, с точки зрения объектов БД (таблиц и индексов), задействованных в рабочей нагрузке при обработке команд от приложений.

Рабочая нагрузка в отношении таблиц и индексов

Еще одной статистикой, которая также характеризует рабочую нагрузку, является статистика по использованию таблиц и индексов. Эта статистика доступна в двух вариантах: на уровне строк и на уровне блоков. Первый вариант представляет собой информацию о том, сколько строк затронуто в результате выполнения запросов. Статистика по строкам является логическим отражением рабочей нагрузки, поскольку указывает на использование логических абстракций СУБД; статистика по блокам, выраженная в страницах или байтах, является более близкой к физическому слою хранения данных и хорошо подходит для анализа ввода-вывода.

Для получения нужных данных потребуется несколько представлений. Получить общую картину в рамках кластера или отдельных БД можно в представлении `pg_stat_database`, в котором содержится набор необходимых полей:

- `tup_returned` и `tup_fetched` — показывают общее количество строк, затронутых при доступе к таблицам и индексам (не количество строк, отданных клиенту!).
  - Для индексов `tup_returned` показывает количество строк, возвращенных индексными методами доступа (Index Scan, Bitmap Index Scan, Index Only Scan), а `tup_fetched` показывает количество строк, прочитанных при этом из таблицы (но только для простого индексного доступа без битовой карты).
  - Для таблиц `tup_returned` показывает количество прочитанных из таблицы строк при последовательном сканировании (Seq Scan), а `tup_fetched` показывает количество строк, прочитанных из таблицы в случае доступа по битовой карте (Bitmap Heap Scan). Дело в том, что при объединении битовых карт (Bitmap And) статистику доступа нельзя отнести к какому-то конкретному индексу, поэтому она всегда учитывается в статистике по таблице.

  Из приведенного описания видно, что некоторые операции могут увеличивать оба счетчика одновременно, поэтому не стоит суммировать их значения. Также счетчики могут увеличиваться и при операциях обновления и удаления строк (ведь целевую строку тоже нужно найти и прочитать). В представлении `pg_stat_database` можно получить общую картину по затронутым строкам, а более детальную информацию можно получить, анализируя статистику отдельных таблиц и индексов.
- `tup_inserted`, `tup_updated` и `tup_deleted` — с этими полями все гораздо проще: они содержат количество строк, затронутых операциями `INSERT`, `UPDATE` и `DELETE`, без всяких тонкостей и нюансов.

Эта статистика также подходит для поверхностной оценки колебаний рабочей нагрузки. Из мониторинга ее можно получить с помощью следующих метрик:

- `postgres_database_tuples_returned_total`;
- `postgres_database_tuples_fetched_total`;
- `postgres_database_tuples_inserted_total`;
- `postgres_database_tuples_updated_total`;
- `postgres_database_tuples_deleted_total`.

Каждая метрика возвращает соответствующее значение затронутых строк для отдельной БД. Для наблюдения в динамике следует ориентироваться на частоту изменения величины, и для оценки общей картины метрики отдельных баз следует просуммировать.

Для графика (рис. 4.1) потребуется два запроса: каждая из метрик описывает отдельное действие, связанное со строкой.

```
# sum(rate(postgres_database_tuples_fetched_total{service_id="primary"}[1m]))
# sum(rate(postgres_database_tuples_returned_total{service_id="primary"}[1m]))
```

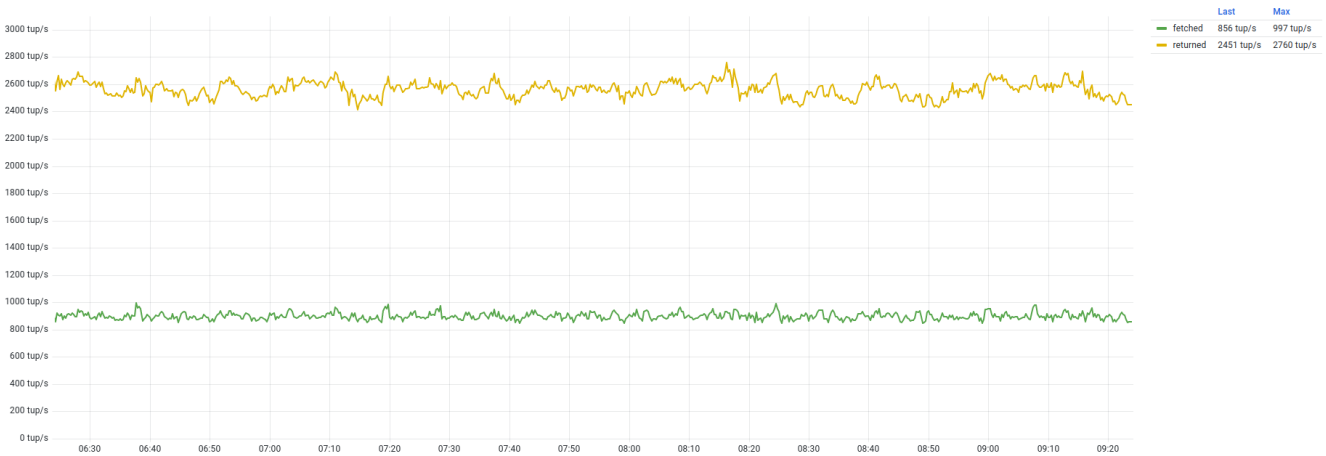


Рис. 4.1. Динамика затронутых (fetched, returned) строк

Аналогичную картину можно получить для строк, затронутых при операциях изменения. Для построения графика (рис. 4.2) потребуются три запроса:

```
# sum(rate(postgres_database_tuples_inserted_total{service_id="primary"}[1m]))
# sum(rate(postgres_database_tuples_updated_total{service_id="primary"}[1m]))
# sum(rate(postgres_database_tuples_deleted_total{service_id="primary"}[1m]))
```

Оба графика показывают ровную рабочую нагрузку без сильных колебаний. Это говорит о том, что со стороны приложений количество клиентов и объем запросов более или менее постоянны, и со стороны СУБД количество затрагиваемых строк при выполнении рабочей нагрузки также постоянно. Причиной сильных колебаний может быть как изменение количества экземпляров приложений и объема отправляемых запросов, так и внутренние причины, такие как

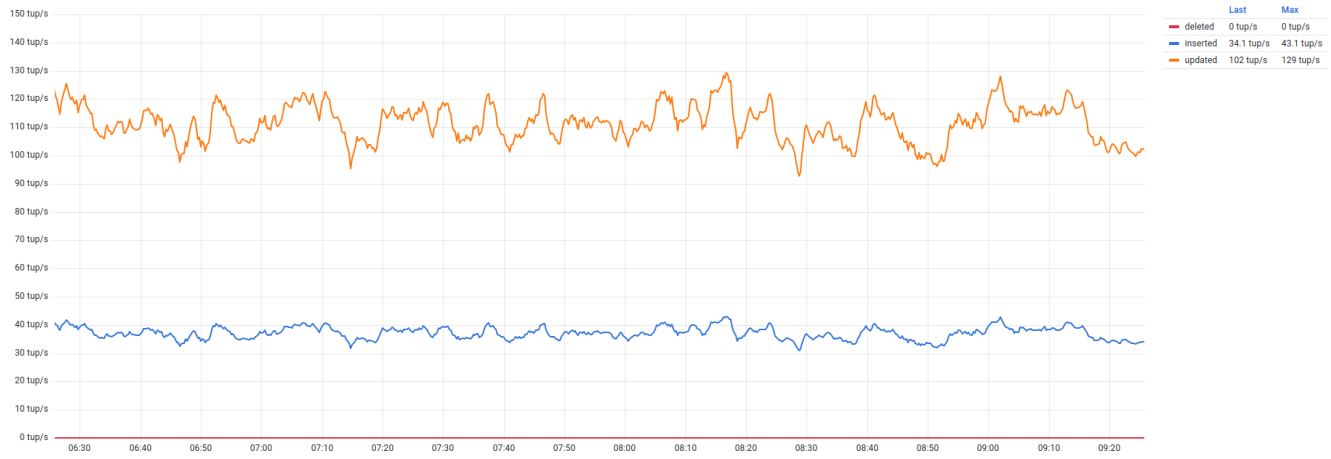


Рис. 4.2. Динамика вставленных, обновленных и удаленных строк

использование неоптимальных планов, которые могут приводить к изменению количества затрагиваемых строк и долгому выполнению запросов.

Для более детального анализа и понимания того, какие именно таблицы затронуты в нагрузке, следует использовать представления `pg_stat_user_tables` и `pg_stat_user_indexes`. В этих представлениях содержится больше информации об использовании именно таблиц и индексов. В `pg_stat_user_tables` интерес представляют следующие поля:

- `seq_scan` — количество операций последовательного доступа (sequential scan);
- `seq_tup_read` — количество строк, затронутых в результате выполнения всех операций последовательного доступа;
- `idx_scan` — количество операций индексного доступа (index scan);
- `idx_tup_fetch` — количество строк, затронутых индексным доступом;
- `n_tup_ins` — количество вставленных строк (обычно в результате выполнения `INSERT`-запросов);
- `n_tup_upd` — количество обновленных строк, включая и так называемые HOT-обновления (в результате выполнения `UPDATE`-запросов);
- `n_tup_del` — количество удаленных строк (в результате выполнения `DELETE`-запросов);
- `n_tup_hot_upd` — количество строк, обновленных с использованием механизма HOT (Heap-Only Tuples update) — оптимизации, направленной на уменьшение накладных расходов при обновлении строк. Ее суть состоит в том, что если в результате обновления значения атрибутов не изменились ни в одном индексе, то и вставку индексных указателей можно исключить. За счет этого обновление выполняется быстрее и с меньшими затратами

ресурсов. Количество HOT-обновлений также включено в поле n_tup_upd. Более подробно о механизме HOT можно почитать в некоторых публикациях[1, 2].

Похожая статистика есть и по индексам в представлении pg_stat_user_indexes:

- idx_scan — количество обращений к данному индексу. Это поле позволяет обнаруживать неиспользуемые индексы, что важно, поскольку их существование не обходится даром;
- idx_tup_read — количество элементов индекса, прочитанных при сканированиях по индексу (без обращения к таблице);
- idx_tup_fetch — количество живых строк таблицы, отобранных при простых сканированиях по индексу.

Стоит разобраться, как работают счетчики idx_tup_read и idx_tup_fetch в случае использования битовых карт индексов. При использовании нескольких индексов результаты сканирования могут объединяться логическим умножением или сложением, и, как следствие, становится невозможно связать выборки отдельных строк с конкретными индексами. В таких случаях увеличиваются значения pg_stat_user_indexes.idx_tup_read для задействованных индексов и счетчики pg_stat_user_tables.idx_tup_fetch для каждой таблицы, а значения pg_stat_user_indexes.idx_tup_fetch не меняются.

В представлениях pg_stat_user_tables и pg_stat_user_indexes собрана статистика по пользовательским таблицам и индексам. По системным объектам статистика собрана в аналогичных представлениях pg_stat_sys_tables и pg_stat_sys_indexes. Эти представления обращаются к двум другим, в которых собрана статистика по всем объектам вместе (pg_stat_all_tables и pg_stat_all_indexes), а они, в свою очередь, основаны на большом наборе функций, которые обращаются к отдельным счетчикам статистики. Перечисленные представления индивидуальны для каждой БД (в отличие от pg_stat_database, которое является общим для всего кластера баз данных).

Давайте рассмотрим возможные сценарии использования этой статистики:

- избыточный последовательный доступ;
- таблицы с наибольшим количеством вставок, обновлений и удалений.

Избыточный последовательный доступ. Последовательный доступ подразумевает обращение к таблице (чтение и, возможно, запись) ровно до того момента, как будет обработано необходимое количество строк. Это один из базовых методов доступа, который используется в случае, когда нужно обратиться к большому количеству строк, или при отсутствии подходящих индексов. В первом случае последовательное сканирование может использоваться в аналитических запросах или при массовых (bulk) изменениях, когда требуется прочитать большую часть таблицы или даже всю таблицу полностью. Во втором случае при отсутствии подходящего индекса последовательный доступ будет задействован даже при запросе всего одной строки, что может

[1] www.cybertec-postgresql.com/en/hot-updates-in-postgresql-for-better-performance
[2] www.interdb.jp/pg/pgsql07.html

привести к избыточному использованию ресурсов ввода-вывода и процессора (чтение и фильтрация строк).

На практике необходимо отслеживать количество операций последовательного доступа и понимать, действительно ли они необходимы. В случае аналитических запросов это оправданно, а для OLTP-запросов обычно не требуется большое количество записей, и последовательный доступ может обходиться дороже, чем использование индекса. Однако из этого не следует, что на каждое поле нужно создавать индекс, — обслуживание индексов не обходится бесплатно и при некоторых обстоятельствах может вызывать избыточный ввод-вывод. При добавлении нового индекса важно оценивать планы выполнения запросов и понимать, будет ли новый индекс использоваться и будет ли он полезен.

Для получения списка таблиц, которые читаются последовательно, можно использовать следующий запрос (подключение должно быть выполнено к БД pgbench):

```
# SELECT relname, seq_scan, seq_tup_read, idx_scan, idx_tup_fetch
  FROM pg_stat_user_tables
  WHERE seq_scan > 0
  ORDER BY seq_tup_read DESC;
     relname      | seq_scan | seq_tup_read | idx_scan | idx_tup_fetch
------------------+----------+--------------+----------+---------------
 pgbench_branches |  5684193 |    113683860 |        0 |             0
```

Запрос выводит статистику доступа к таблице с помощью последовательного и индексного доступов — количество операций и количество затронутых строк. В тестовой нагрузке не так много таблиц, и запрос выводит всего одну таблицу, доступ к которой осуществляется последовательно. На самом деле у таблицы есть индекс, который мог бы использоваться, но в таблице всего 20 строк (то есть вся таблица целиком помещается в один блок) и планировщик считает, что использование индекса обойдется дороже: ведь придется читать блоки не только таблицы, но и индекса. Поэтому он выбирает более дешевый план с последовательным доступом.

В производственных окружениях с разнообразной нагрузкой может оказаться больше таблиц, сканируемых последовательно, и, сравнивая их между собой, следует принимать во внимание некоторые показатели (для большего удобства их можно также выводить в запросе):

- среднее количество строк за одно сканирование — `seq_tup_read / seq_scan`. Чем больше строк в среднем, тем больше и ресурсов требуется при сканировании. Но это не значит, что все запросы читают именно среднее количество строк;
- доля последовательных обращений от общего числа обращений — `seq_scan / ( seq_scan + idx_scan )`. Чем больше доля, тем больше ресурсов используется при сканировании, однако стоит помнить про небольшие таблицы, которые могут полностью помещаться в общем кеше;
- размер таблицы в байтах позволяет сравнивать таблицы друг с другом. Например, устранение случаев последовательного доступа к большим таблицам может быть приоритетнее, чем к небольшим таблицам. Определение размеров таблиц и других объектов БД рассматривается в этой главе чуть дальше.

Получение этой и подобной статистики с помощью запросов удобно для построения отчетов за периоды, однако для оперативного анализа и поиска проблем важно видеть динамику изменений. Следующим запросом можно получить первые *K* таблиц по количеству строк, прочитанных при последовательном доступе:

```
# topk_avg(5,
    rate(postgres_table_seq_tup_read_total{service_id="primary"}[1m]),
    "other"
)
```

Если на основе этого запроса построить график, на нем будет показана всего одна таблица, что полностью соответствует результату SQL-запроса.

В качестве небольшого эксперимента можно выполнить разовый запрос с чтением всех строк другой таблицы:

```
# SELECT count(*) FROM pgbench_history;
 count
--------
 486693
```

Спустя некоторое время на графике появится характерный выброс (рис. 4.3).

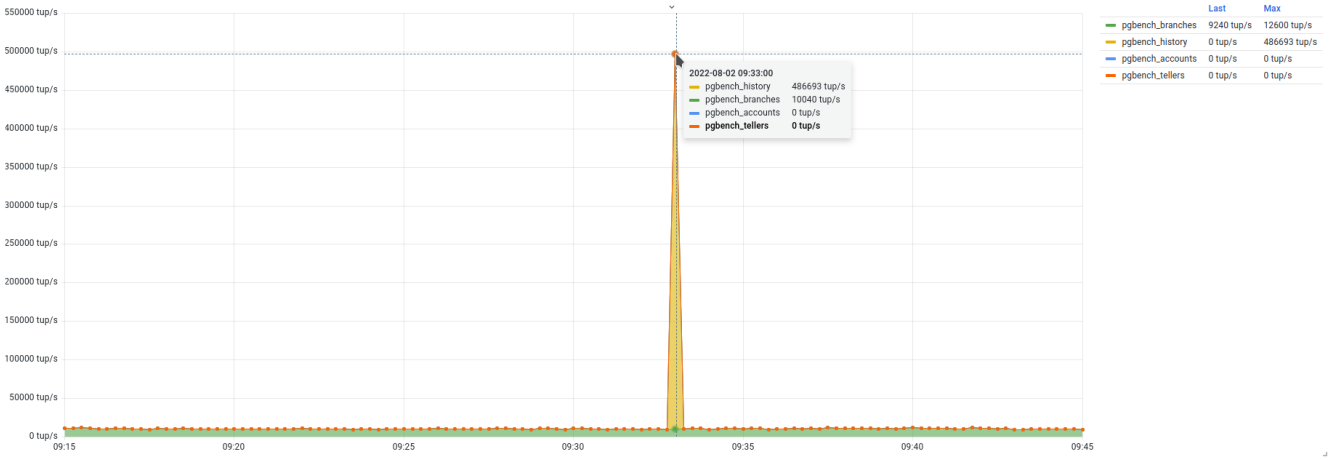


Рис. 4.3. Последовательный доступ

Таким образом, с помощью подобного графика можно оперативно отслеживать изменение в динамике последовательного доступа и реагировать должным образом (анализировать запросы к таблице, строить индексы и т. п.).

Таблицы с наибольшим количеством вставок, обновлений и удалений. Статистика использования таблиц является логическим развитием статистики строк из представления `pg_stat_database`. Если там мы видели общую статистику уровня баз данных, то с помощью `pg_stat_user_tables`

можно заглянуть внутрь отдельных БД и более подробно проанализировать таблицы, задействованные в рабочей нагрузке, и оценить объемы затронутых строк.

Представление pg_stat_user_tables является индивидуальным для каждой БД, поэтому, чтобы получить статистику по таблицам в БД pgbench, нужно подключиться именно к этой базе. Следующим запросом можно получить статистику затронутых строк:

```
# SELECT
    relname,
    seq_tup_read AS seq_read,
    idx_tup_fetch AS idx_fetch,
    n_tup_ins AS inserted,
    n_tup_upd AS updated,
    n_tup_del AS deleted
  FROM pg_stat_user_tables
  ORDER by relname;
     relname      |  seq_read | idx_fetch | inserted | updated | deleted
------------------+-----------+-----------+----------+---------+---------
 pgbench_accounts |   2000000 |  11506518 |  2000000 | 5753260 |       0
 pgbench_branches | 115105760 |         0 |       20 | 5753260 |       0
 pgbench_history  |    486693 |           |  5753260 |       0 |       0
 pgbench_tellers  |  40336000 |   5551581 |      200 | 5753260 |       0
```

В этом примере выведены все пользовательские таблицы в БД pgbench и общее количество строк, затронутое различными операциями; указывая интересующее поле в предложении ORDER BY, можно получить таблицы с наибольшим количеством затронутых строк. Однако в выводе снова появляются большие цифры, и на практике удобно отслеживать эти величины в динамике. Для этого можно воспользоваться следующими метриками (имена метрик аналогичны названиям полей):

- postgres_table_seq_tup_read_total;
- postgres_table_idx_tup_fetch_total;
- postgres_table_tuples_inserted_total;
- postgres_table_tuples_updated_total;
- postgres_table_tuples_deleted_total;
- postgres_table_tuples_hot_updated_total.

Следующим запросом можно получить таблицы с наибольшим количеством обновлений:

```
# topk_avg(5,
    rate(postgres_table_tuples_updated_total{service_id="primary"}[1m]),
    "other"
)
```

График (рис. 4.4) более наглядно выводит информацию о том, на какие именно таблицы приходится больше всего обновлений.

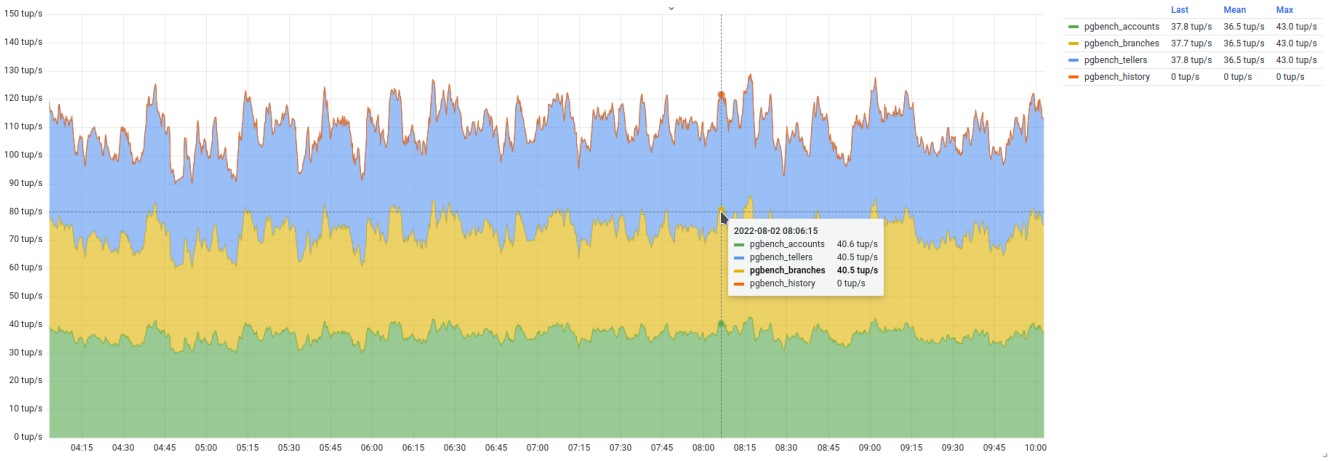


Рис. 4.4. Таблицы с наибольшим количеством обновленных строк

Меняя метрику в запросе, можно получить соответствующие графики для других показателей, например для вставленных строк (рис. 4.5). Используя такие графики на практике, можно оперативно отслеживать колебания в рабочей нагрузке, особенно в случае каких-то значительных изменений на стороне приложений.

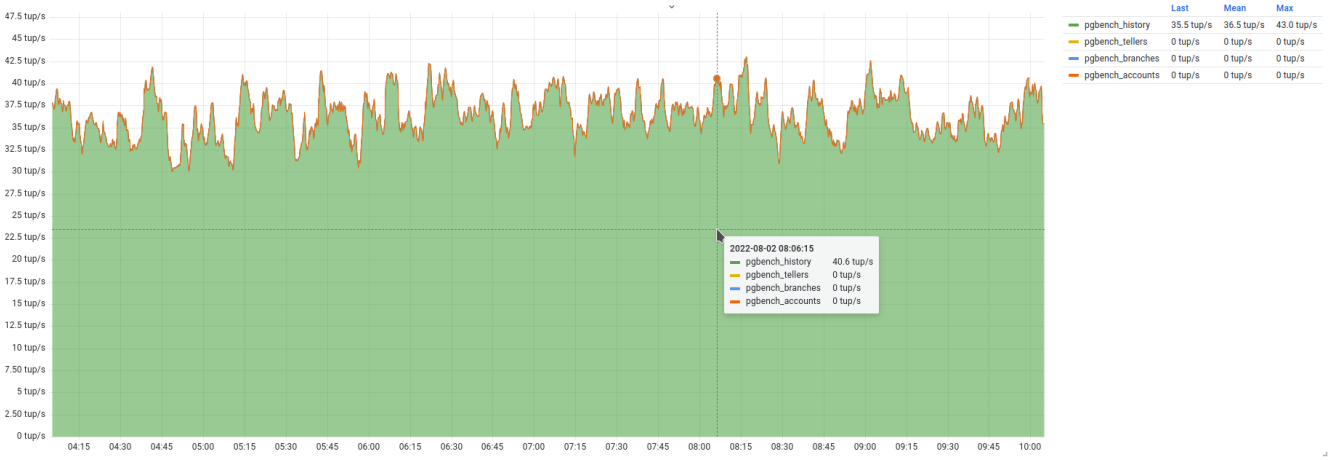


Рис. 4.5. Таблицы с наибольшим количеством вставленных строк

Ошибки и нежелательные события

Давайте снова вернемся к `pg_stat_database` и рассмотрим следующие поля:

- `xact_rollback` — это поле упоминалось при рассмотрении транзакционной активности. Оно указывает на количество транзакций, завершившихся откатом. Факт отката можно

расценивать как возникновение ошибки (за исключением случаев явного вызова команды ROLLBACK). Следовательно, большое число откатов может сигнализировать об ошибках при выполнении как отдельных запросов, так и транзакций, и xact_rollback можно использовать как счетчик таких ошибок. Типичными примерами могут служить нарушение уникальности при вставке строк, ошибки синтаксиса, неверное указание аргументов функции или обращение к несуществующим объектам. Однако кроме таких ошибок возможны и другие, которые могут возникать по причинам, не зависящим от приложения, но эту информацию можно получить только из журнала СУБД.

- conflicts — количество запросов, отмененных из-за конфликтов между выполнением запроса и воспроизведением WAL-журнала на узлах горячего резерва (при использовании репликации). Это отдельный класс ошибок, который более подробно будет рассмотрен в главе 7, посвященной репликации. В любом случае возникновение таких конфликтов также можно расценивать как ошибку.
- deadlocks — взаимоблокировки, их мы также рассматривали ранее в главе 2. Для разрешения взаимоблокировки какая-то из транзакций завершается принудительно, и это также можно расценивать как ошибку.
- checksum_failures и checksum_last_failure показывают количество ошибок при проверке контрольных сумм (data page checksums) и время последней зафиксированной ошибки. Контрольные суммы рассчитываются на основе содержимого страницы данных, пересчитываются при последующих изменениях и используются для выявления повреждения данных при обращениях к странице. Это один из наиболее неприятных типов ошибок; их появление (особенно в производственном окружении) следует расценивать очень серьезно и предпринимать все необходимые меры по исключению таких ошибок в будущем.
- sessions_abandoned, sessions_fatal, sessions_killed указывают на количество сеансов с ненормальным завершением. Эти поля мы также обсуждали в главе 2. Они указывают на то, что сеанс завершился ненормально, и это также можно рассматривать как ошибку.

Как вы могли заметить, все рассмотренные события по сути являются ошибками, и при нормальной и правильной эксплуатации СУБД и приложений их вообще не должно возникать. Хорошей практикой является использование графика, где все метрики по этим ошибкам собраны вместе:

- postgres_database_xact_rollbacks_total;
- postgres_database_conflicts_total;
- postgres_database_deadlocks_total;
- postgres_database_checksum_failures_total;
- postgres_database_sessions_total{reason=~"abandoned|fatal|killed"}.

В самом идеальном случае такой график должен быть «пустым», как на рис. 4.6. Любые выбросы должны привлекать внимание администратора БД и служить поводом разобраться с причинами.

Рис. 4.6. Ошибки и нежелательные события

Бóльшая часть перечисленных событий свидетельствует о проблемах со стороны приложений, и лишь ошибки контрольных сумм указывают на проблемы в СУБД. При долгой и продолжительной эксплуатации некоторые ошибки могут иметь хронический накопительный характер, и, если сформировать такой график, на нем уже может быть какой-то постоянный фон из ошибок. Тем более что некоторые классы ошибок могут восприниматься как нечто привычное и нормальное, так как обработка этих ошибок выполняется на стороне приложений (например, переустановка сеанса при обрыве или повтор запроса при взаимоблокировке). В таком случае часть событий, такие как откаты и конфликты репликации, можно игнорировать как менее важные и не влияющие значительно на производительность СУБД, а взаимоблокировки и ошибки контрольных сумм по-прежнему воспринимать серьезно — взаимоблокировки негативно влияют на производительность конкурентной работы, а ошибки контрольных сумм могут указывать на риск повреждения данных.

4.3. Функции для работы с объектами СУБД

Довольно часто вместе с основной статистикой по объектам БД хочется видеть и размеры этих объектов, чтобы понимать их масштаб относительно друг друга. Часто размер объекта определяет приоритет работ и сам подход к работе — большие объекты требуют большей аккуратности (нельзя допускать долгого удерживания блокировок, следует избегать побочной нагрузки), и конечная выгода в результате проведенных работ может быть больше. Размеры объектов важны и сами по себе, они являются одним из ключевых факторов при планировании емкости дискового пространства. При продолжительной эксплуатации СУБД данные будут добавляться, объекты будут увеличиваться и занимать все больше и больше места на диске.

Отслеживать размеры объектов можно как со стороны операционной системы, так и со стороны СУБД. Как известно, почти все объекты СУБД — это файлы и каталоги, и можно получить информацию об объеме этих объектов в файловой системе. С другой стороны, СУБД оперирует собственными абстракциями — табличными пространствами, базами данных, таблицами, индексами и т. д. В обоих случаях для получения размеров нам понадобятся так называемые *функции системного администрирования*[1]. Это большой набор вспомогательных функций для получения самой разной информации, которая может потребоваться администратору БД. Пока из всего набора функций нас интересуют две группы:

1. Функции управления объектами баз данных[2]. Большинство функций в этой группе используются для подсчета места, занимаемого различными объектами СУБД.
2. Функции доступа к файлам[3] предоставляют доступ к каталогам системы, на которой запущен сервер СУБД. Это мощный набор функций для исследования системы вообще; доступ к функциям имеют только суперпользователи и участники роли `pg_read_server_files`. Доступ к ним должен предоставляться очень аккуратно с учетом всех требований безопасности, поскольку к этим функциям неприменимы проверки привилегий, встроенные в СУБД. Так, например, выдача прав `EXECUTE` на функцию `pg_read_file` разрешает читать любые файлы, доступные серверному процессу СУБД; пользователь с правом выполнения такой функции может прочитать содержимое любой таблицы, включая и те, где могут содержаться конфиденциальные данные (например, `pg_authid`).

Определение размеров объектов СУБД

Среди функций управления объектами интерес представляют следующие:

- `pg_tablespace_size` — показывает размер указанного табличного пространства;
- `pg_database_size` — показывает размер указанной базы данных;
- `pg_table_size` — показывает общий размер указанной таблицы, включая TOAST-хранилище и служебные слои (vm, fsm, init);
- `pg_indexes_size` — показывает общий размер всех индексов указанной таблицы;
- `pg_total_relation_size` — показывает общий размер указанной таблицы (включая TOAST и служебные слои) и все индексы этой таблицы. Результат подсчета эквивалентен сумме функций `pg_table_size` и `pg_indexes_size`;
- `pg_relation_size` — показывает размер слоя отношения (таблицы или индекса), указанного в первом аргументе. Во втором аргументе можно указать имя интересующего слоя:
  - `main` — подсчитать размер основной структуры с данными;
  - `fsm` — подсчитать размер карты свободного пространства;
  - `vm` — подсчитать размер карты видимости.

[1] postgrespro.ru/docs/postgresql/current/functions-admin
[2] postgrespro.ru/docs/postgresql/current/functions-admin#FUNCTIONS-ADMIN-DBOBJECT
[3] postgrespro.ru/docs/postgresql/current/functions-admin#FUNCTIONS-ADMIN-GENFILE

Если не указывать второй аргумент, значение main предполагается по умолчанию. Для индекса размер карты видимости всегда будет нулевым.

- pg_size_pretty — вспомогательная функция, переводящая значение в байтах в текстовый вид с единицами измерения (bytes, kB, MB, GB, TB, PB).
- pg_size_bytes — вспомогательная функция, обратная pg_size_pretty: она позволяет пересчитать текстовое значение в байты.

Некоторые из этих функций существуют в двух вариантах и в качестве объекта могут принимать либо его числовой идентификатор OID, либо текстовое имя.

В запросах, выводящих статистику по объектам БД, эти функции удобно использовать для указания размеров объектов. Размеры таких составных объектов, как табличные пространства или базы данных, необходимы для оценки используемого места в хранилище и для планирования емкости. Размеры прочих объектов, таких как таблицы и индексы, обычно нужны при оценке использования места внутри базы данных или для оценки эффекта раздувания. Дальше мы коротко рассмотрим примеры использования функций.

Табличные пространства. В начале главы упоминалась метакоманда \db+, показывающая список табличных пространств. Большинство метакоманд на самом деле выполняют SQL-запросы к служебным таблицам системного каталога, соответственно, и результат, аналогичный выводу \db+, можно получить напрямую, используя SQL.

Запросы метакоманд

Чтобы получить текст запроса, который используется метакомандой, можно запустить psql с аргументом -E (--echo-hidden), который указывает psql включить отображение запросов при использовании встроенных метакоманд. В открытом сеансе отображение запросов можно включить с помощью команды \set ECHO_HIDDEN on.

Запрос, показанный ниже, работает как упрощенная версия метакоманды \db+:

```
# SELECT spcname, pg_tablespace_location(oid) AS path,
    pg_size_pretty(pg_tablespace_size(spcname)) AS size
  FROM pg_tablespace
  ORDER BY 2 DESC;
  spcname   | path |  size
------------+------+--------
 pg_default |      | 378 MB
 pg_global  |      | 556 kB
```

Для определения пути (поле path) используется функция pg_tablespace_location. В тестовом окружении используются табличные пространства по умолчанию; значения пути для них отсутствуют (NULL), так как эти пространства находятся в основном каталоге кластера БД. Для получения размера используется функция pg_tablespace_size, которая может принимать как числовой идентификатор OID, так и имя табличного пространства.

Базы данных. Для получения размеров баз данных используется функция pg_database_size. В качестве аргумента функция может принимать числовой идентификатор OID или имя базы данных. Пример запроса для получения размеров:

```
# SELECT t.spcname AS tablespace,
    d.datname AS dbname,
    pg_get_userbyid(d.datdba) AS owner,
    pg_size_pretty(pg_database_size(d.datname)) AS size
  FROM pg_database d
  JOIN pg_tablespace t ON d.dattablespace = t.oid
  ORDER BY pg_database_size(d.datname) DESC;
 tablespace |  dbname   |  owner   |  size
------------+-----------+----------+---------
 pg_default | pgbench   | pgbench  | 356 MB
 pg_default | postgres  | postgres | 7542 kB
 pg_default | template1 | postgres | 7534 kB
 pg_default | template0 | postgres | 7297 kB
```

Таблицы. Таблицы являются комплексными абстракциями, и, когда речь заходит об их размере, важно четко определять тот смысл, что вкладывается в термин «размер». Таблица состоит из основного слоя пользовательских строк и также включает в себя такие служебные слои, как карты видимости и свободного пространства. Кроме того, с таблицами могут быть ассоциированы индексы и TOAST-таблицы, которые с физической стороны являются отдельными сущностями (файлами на диске), но с логической не могут существовать без своей родительской таблицы. Для получения полного размера таблиц, включая все служебные, удобно использовать функцию pg_table_size. При необходимости учета индексов следует использовать функцию pg_total_relation_size (которая эквивалентна сумме pg_table_size и pg_indexes_size). Наконец, для определения размеров на уровне служебных слоев следует использовать pg_relation_size, которой можно указать конкретный служебный слой. Следующий запрос демонстрирует использование перечисленных функций:

```
# SELECT c.relname AS table,
    (SELECT count(*) FROM pg_index i WHERE i.indrelid = c.oid) AS n_idx,
    pg_size_pretty(pg_total_relation_size(c.oid)) AS total,
    pg_size_pretty(pg_relation_size(c.reltoastrelid)) AS toast,
    pg_size_pretty(pg_indexes_size(c.oid)) AS indexes,
    pg_size_pretty(pg_relation_size(c.oid, 'main')) AS main,
    pg_size_pretty(pg_relation_size(c.oid, 'fsm')) AS fsm,
    pg_size_pretty(pg_relation_size(c.oid, 'vm')) AS vm,
    pg_size_pretty(pg_relation_size(c.oid, 'init')) AS init
  FROM pg_class c
  WHERE c.relkind IN ('r', 'm')
    AND NOT EXISTS (
        SELECT 1 FROM pg_locks
        WHERE relation = c.oid AND mode = 'AccessExclusiveLock' AND granted
    )
    AND pg_total_relation_size(c.oid) > 500 * 2^10
  ORDER BY pg_total_relation_size(c.oid) DESC LIMIT 10;
```

```
      table       | n_idx |  total  |   toast    | indexes |  main  |   fsm   |     vm     |  init
------------------+-------+---------+------------+---------+--------+---------+------------+---------
 pgbench_accounts |     1 | 320 MB  |            | 43 MB   | 277 MB | 88 kB   | 16 kB      | 0 bytes
 pgbench_history  |     0 | 30 MB   |            | 0 bytes | 30 MB  | 24 kB   | 8192 bytes | 0 bytes
 pg_proc          |     2 | 1128 kB | 8192 bytes | 336 kB  | 768 kB | 0 bytes | 0 bytes    | 0 bytes
 pg_attribute     |     2 | 696 kB  |            | 208 kB  | 456 kB | 24 kB   | 8192 bytes | 0 bytes
 pg_rewrite       |     2 | 672 kB  | 512 kB     | 32 kB   | 112 kB | 0 bytes | 0 bytes    | 0 bytes
 pg_description   |     1 | 584 kB  | 0 bytes    | 224 kB  | 352 kB | 0 bytes | 0 bytes    | 0 bytes
```

В этом примере есть несколько особенностей:

- Вместо `pg_stat_user_tables` используется `pg_class`, так как в этой таблице содержится полный список объектов БД.
- Для получения нужных объектов используется фильтр `c.relkind IN ('p', 'r', 'm')`, который оставляет только обычные (`r`) и секционированные (`p`) таблицы, а также материализованные представления (`m`), таким образом исключая индексы, TOAST-таблицы, обычные представления, последовательности и прочие объекты.
- Исключение таблиц, заблокированных в исключительном режиме `AccessExclusiveLock`. Важным нюансом использования административных функций является уровень устанавливаемых блокировок. Для подсчета размеров объекта функциям требуется блокировка уровня `ExclusiveLock`[1]. При регулярном использовании таких функций, например в случае мониторинга, появляется риск конфликта с самой строгой исключительной (`AccessExclusiveLock`) блокировкой и вероятность встать в очередь. Для устранения такого риска заблокированные таблицы не выводятся; их размер можно будет получить в другой раз, когда блокировка будет снята.
- Выбираются только таблицы, размер которых превышает 500 КБ. Подобным условием удобно отсекать небольшие таблицы. Впрочем, в тестовом окружении вообще немного таблиц, а больших — и того меньше.
- Отдельным подзапросом к `pg_index` подсчитывается общее количество индексов для каждой таблицы.
- Для полноты картины добавлен подсчет размера init-слоев, хотя это актуально только при использовании нежурналируемых таблиц, которых, впрочем, нет в тестовом окружении.

Логическим улучшением такого запроса может быть представление служебных слоев в виде процентных отношений относительно общего размера таблицы: так будет удобнее видеть аномалии и перекосы (хотя я не припомню, чтобы мне встречались служебные слои больших размеров). Также можно вывести табличные пространства, которым принадлежат таблицы, однако для этого потребуется выполнить соединение с `pg_tablespace`.

Индексы. Размер конкретного индекса определяется с помощью функций `pg_relation_size` или `pg_total_relation_size`. Обе функции подходят как для таблиц, так и для индексов.

[1] postgrespro.ru/docs/postgresql/current/explicit-locking

```
# SELECT coalesce(tablespace, 'pg_default') AS tablespace,
    schemaname ||'.'|| tablename AS table,
    indexname AS index,
    pg_size_pretty(pg_relation_size(indexname::regclass)) AS size
  FROM pg_indexes
  ORDER BY pg_relation_size(indexname::regclass) DESC LIMIT 10;
 tablespace |           table           |              index               |  size
------------+---------------------------+----------------------------------+--------
 pg_default | public.pgbench_accounts   | pgbench_accounts_pkey            | 43 MB
 pg_default | pg_catalog.pg_proc        | pg_proc_proname_args_nsp_index   | 248 kB
 pg_default | pg_catalog.pg_description | pg_description_o_c_o_index       | 224 kB
 pg_default | pg_catalog.pg_attribute   | pg_attribute_relid_attnam_index  | 120 kB
 pg_default | pg_catalog.pg_attribute   | pg_attribute_relid_attnum_index  | 88 kB
 pg_default | pg_catalog.pg_proc        | pg_proc_oid_index                | 88 kB
 pg_default | pg_catalog.pg_depend      | pg_depend_depender_index         | 80 kB
 pg_default | pg_catalog.pg_depend      | pg_depend_reference_index        | 64 kB
 pg_default | pg_catalog.pg_amop        | pg_amop_fam_strat_index          | 48 kB
 pg_default | pg_catalog.pg_operator    | pg_operator_oprname_l_r_n_index  | 48 kB
```

Для удобства работы с индексами можно использовать представление pg_indexes, которое основано на pg_class и pg_index, однако в нем нет поля с уникальным идентификатором, что делает невозможным соединение с другими представлениями по OID. Для простоты запрос выполняется только к pg_indexes без соединения с pg_locks, однако риск конфликта с исключительной блокировкой остается, как и в случае с таблицами.

Оценка размеров объектов БД в системах мониторинга. Представление размеров в виде таблицы, как в SQL-запросах, хорошо подходит для отчетов. Для оперативного наблюдения за размерами объектов СУБД можно использовать следующие метрики:

- postgres_database_size_bytes;
- postgres_table_size_bytes;
- postgres_index_size_bytes.

На основе этих метрик можно сделать два типа графиков. Первый и самый простой вариант — показывать объекты с наибольшим размером. Это простая оценка того, как используется место самыми большими объектами БД, однако такой график не покажет аномального роста каких-то других таблиц — это будет заметно, только когда таблица увеличится в размерах настолько, что попадет в список самых больших. Рост может длиться долгое время, за которое проблема аномального увеличения могла бы быть уже исправлена. Вторым, дополнительным вариантом может быть использование графика с объектами, у которых происходит наибольшее изменение размера в интервале времени.

На примере таблиц давайте рассмотрим несколько случаев. Следующим запросом можно получить самые большие таблицы:

```
# topk_max(5, postgres_table_size_bytes{service_id="primary"}, "other")
```

На графике (рис. 4.7) это будет выглядеть следующим образом:

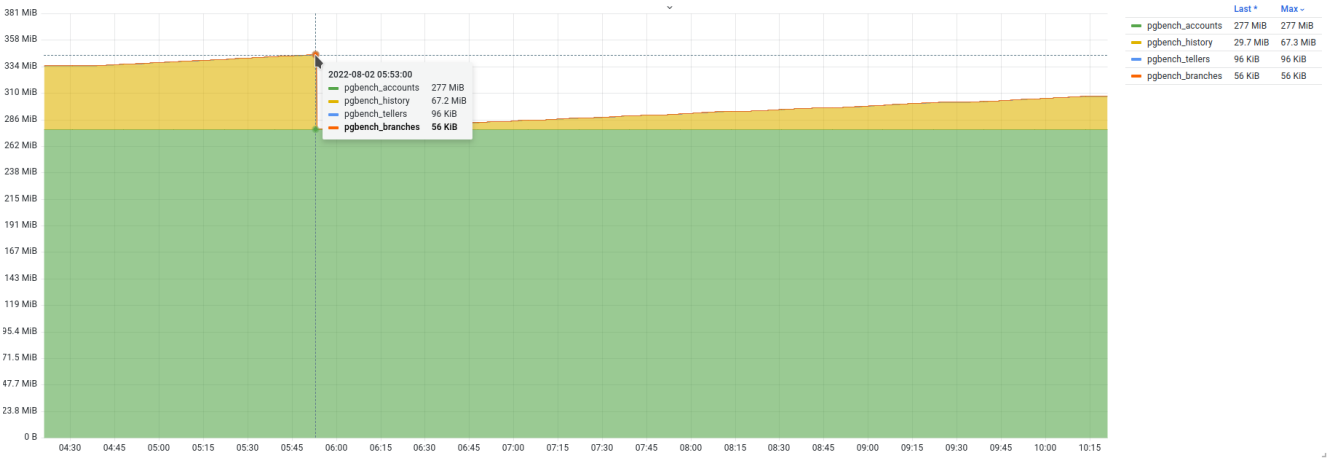


Рис. 4.7. Таблицы с наибольшим размером

Пример запроса для второго варианта с таблицами, у которых происходит наибольшее изменение размера:

```
# delta(postgres_table_size_bytes{service_id="primary"}[1h])
```

В этом запросе используется функция `delta`, которая считает разницу в значениях за интервал в один час. Следует аккуратно подходить к выбору интервала: даже один час может оказаться слишком длинным интервалом и сглаживать резкие изменения в размерах за короткие периоды. График, построенный по этой функции, показан на рис. 4.8:

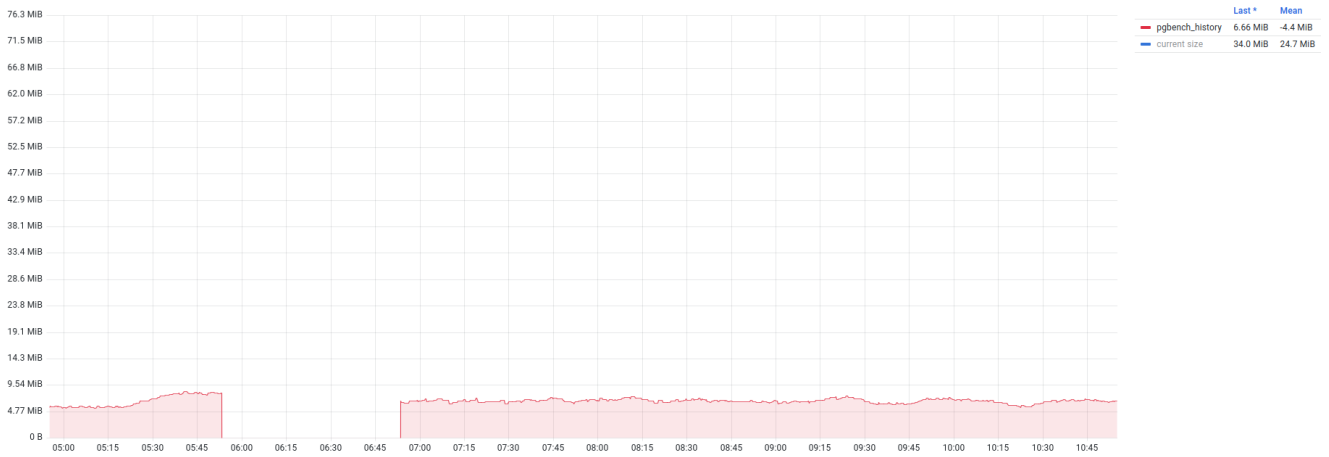


Рис. 4.8. Таблицы с наибольшими изменениями размеров

В этом примере видно, что таблица очищается и затем стабильно растет примерно на 5 МБ в час. Для большей наглядности на график можно наложить размер самой таблицы (рис. 4.9):

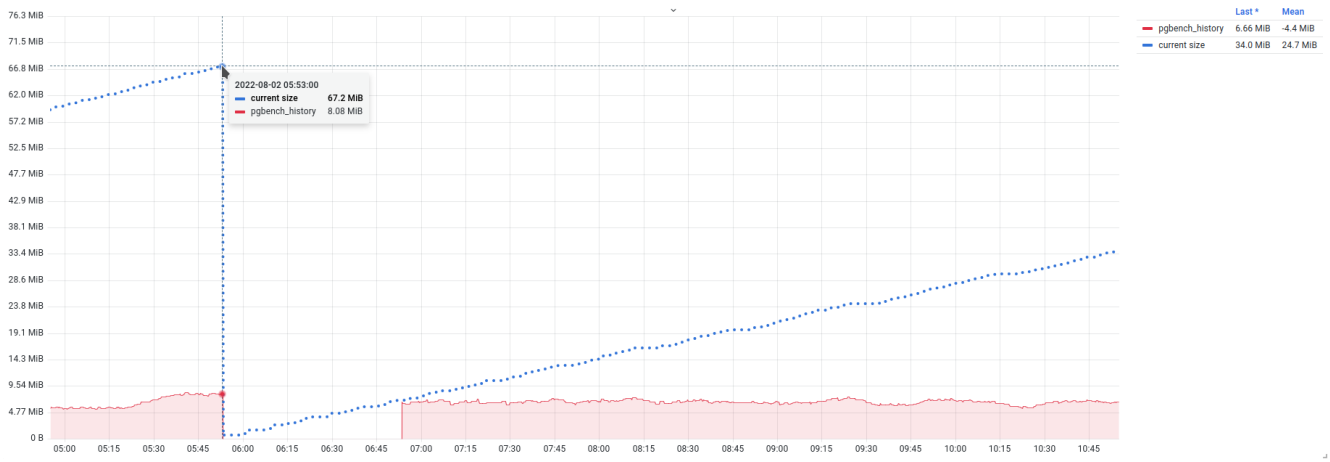


Рис. 4.9. Таблицы с наибольшим изменением размеров, включая актуальный размер

Размещение объектов в файловой системе

В некоторых случаях приходится обращаться к физическому уровню хранения таблиц и индексов, для чего необходимо найти физическое размещение объектов в файловой системе. Часто это бывает нужно в задачах анализа повреждений и восстановления данных. Для этого предназначено несколько функций:

- `pg_relation_filenode` — принимает идентификатор OID или имя объекта и возвращает номер так называемого *файлового узла* (filenode), который связан с указанным объектом. Файловым узлом называется основной компонент имени файла, используемого для хранения данных. Для большинства таблиц этот номер совпадает со значением `pg_class.relfilenode`, но для некоторых системных каталогов это значение равно нулю, и, чтобы узнать действительное значение, нужно использовать эту функцию. Если указанное отношение не хранится на диске, как, например, представления, данная функция возвращает `NULL`;
- `pg_relation_filepath` — возвращает путь к файлу отношения относительно основного каталога данных;
- `pg_filenode_relation` — является функцией, обратной `pg_relation_filenode`, и используется для поиска таблиц по номеру файлового узла (взятого из имени файла). Она возвращает идентификатор OID отношения по заданному OID табличного пространства и файловому узлу. Для таблицы в табличном пространстве по умолчанию в первом параметре можно передать ноль.

Помимо объектов, занятых в хранении пользовательских данных, есть и другие объекты СУБД, которые также хранятся на диске и при некоторых обстоятельствах могут занимать значительный объем. Для их просмотра и анализа также есть набор функций:

- pg_ls_logdir — содержимое каталога, где хранится журнал сообщений СУБД;
- pg_ls_waldir — содержимое каталога pg_wal, где хранятся сегменты WAL-журнала;
- pg_ls_archive_statusdir — содержимое каталога pg_wal/archive_status, где хранятся статусы архивирования отдельных WAL-сегментов;
- pg_ls_tmpdir — содержимое каталога, используемого для хранения временных файлов;
- pg_ls_logicalmapdir — содержимое каталога pg_logical/mappings;
- pg_ls_logicalsnapdir — содержимое каталога pg_logical/snapshots;
- pg_ls_replslotdir — содержимое отдельного каталога слота репликации в pg_replslot;
- pg_ls_dir — универсальная команда для вывода содержимого любого каталога (включая и те, что находятся за пределами основного каталога данных).

Наиболее универсальной из всех перечисленных в этом списке функций является pg_ls_dir. Функция выводит имена всех файлов и каталогов в указанном каталоге и, следовательно, позволяет просматривать любые каталоги на сервере, где запущена СУБД, включая и системные (в рамках прав доступа пользователя, под которым запущена СУБД):

```
# SELECT pg_ls_dir('/');
         pg_ls_dir
---------------------------
 lib
 opt
 bin
 run
 etc
 var
 tmp
 srv
 mnt
 media
 usr
 dev
 sbin
 proc
 root
 home
 sys
 docker-entrypoint-initdb.d
 .dockerenv
```

Хоть pg_ls_dir и является универсальной, остальными функциями пользоваться удобнее, поскольку не требуется запоминать точное расположение каталогов.

Следующая не менее интересная функция — это pg_read_file. Функция читает указанный текстовый файл и возвращает его содержимое. В примере ниже мы просто выводим результат, но его, конечно, можно передать другой функции для дальнейшей обработки. В качестве дополнительных параметров можно задавать смещение и длину для более точного указания диапазона чтения.

```
# SELECT pg_read_file('/etc/os-release');
                            pg_read_file
-----------------------------------------------------------------------
 NAME="Alpine Linux"                                                  +
 ID=alpine                                                            +
 VERSION_ID=3.16.0                                                    +
 PRETTY_NAME="Alpine Linux v3.16"                                     +
 HOME_URL="https://alpinelinux.org/"                                  +
 BUG_REPORT_URL="https://gitlab.alpinelinux.org/alpine/aports/-/issues"+
```

Прочитанное содержимое обрабатывается в кодировке базы данных, и, если символы оказываются недопустимыми для этой кодировки, возникает ошибка. В качестве обхода этой проблемы можно использовать функцию pg_read_binary_file, которая также предназначена для чтения, но читает произвольные двоичные данные и возвращает результат в значении типа bytea, а не text и, следовательно, обходит проверки кодировки. В сочетании с convert_from эту функцию можно применять для чтения текста в указанной кодировке и преобразования в кодировку базы данных:

```
# SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');
```

Следующая функция, заслуживающая особого внимания, — pg_stat_file, которая выводит атрибуты указанного объекта файловой системы:

- размер в байтах;
- время последнего обращения (atime, access time);
- время последнего изменения (mtime, modification time);
- время последнего изменения состояния (ctime, change time), только в Unix-системах;
- время создания (creation time), только в Windows;
- флаг, указывающий, что объект является каталогом.

Суммируем все сказанное выше: имея доступ к перечисленным функциям, можно прочитать содержимое любого каталога и файла в пределах прав пользователя, от имени которого запущена СУБД. По умолчанию доступ к функциям ограничен; при необходимости выдачи прав на эти функции риски безопасности следует серьезно взвесить и учесть.

Содержимое каталога с журналами сообщений СУБД. Функция pg_ls_logdir позволяет вывести содержимое каталога с журналами СУБД. Расположение каталога зависит от настроек конфигурации; использование функции позволяет сразу показать его содержимое в виде списка файлов и их атрибутов. Это довольно удобно, учитывая, что быстро определить, как и куда СУБД

пишет журнал сообщений, — задача не совсем тривиальная, и ответ к тому же зависит от версии системы.

```
# SELECT * FROM pg_ls_logdir() LIMIT 10;
        name        |  size   |      modification
--------------------+---------+------------------------
 postgresql-Mon.log | 1030651 | 2022-08-01 23:59:47+00
 postgresql-Tue.log |  333143 | 2022-08-02 06:03:18+00
 postgresql-Sun.log |  499410 | 2022-07-31 23:58:21+00
```

Функция выводит имя, размер и время последнего изменения (mtime) всех файлов (за исключением скрытых и специальных) в каталоге журналов сообщений СУБД. С точки зрения мониторинга можно получить полный объем, занимаемый журналами на диске:

```
# SELECT sum(size) FROM pg_ls_logdir();
   sum
---------
 1863443
```

Функция может вывести ошибку, если подсистема сбора журналов выключена (*logging_collector* = `off`) и каталога, указанного в *log_directory*, не существует. С некоторыми настройками журналы сообщений могут занимать много места, и, более того, они могут располагаться в каталоге кластера баз данных. Если журналы заполнят все свободное место в файловой системе, они станут причиной аварийной остановки СУБД.

По умолчанию доступ к функции `pg_ls_logdir` имеют только суперпользователи и члены группы `pg_monitor`, право на ее выполнение можно дать и другим пользователям.

Вывод содержимого каталога с WAL-журналами. Следующий каталог, не относящийся напрямую к пользовательским данным, но при этом имеющий тенденцию занимать много места, — это каталог WAL-журнала pg_wal. Обычно этот каталог размещается в основном каталоге кластера баз данных, но нередко его выносят наружу, а в самом каталоге кластера создают символическую ссылку. Делается это по соображениям производительности и использования места. Для вывода содержимого каталога WAL применяется функция `pg_ls_waldir`. Функция выводит имя, размер и время последнего изменения всех обычных файлов. Файлы с именами, начинающимися с точки, каталоги и другие специальные файлы не выводятся. По умолчанию доступ к этой функции имеют суперпользователи и члены группы `pg_monitor`, право на ее выполнение можно дать другим пользователям.

```
# SELECT * FROM pg_ls_waldir() ORDER BY 3 DESC LIMIT 5;
          name           |   size   |      modification
-------------------------+----------+------------------------
 000000010000002400000042 | 16777216 | 2022-08-02 06:03:04+00
 000000010000002400000041 | 16777216 | 2022-08-02 06:02:55+00
 000000010000002400000040 | 16777216 | 2022-08-02 06:01:50+00
 00000001000000240000003F | 16777216 | 2022-08-02 06:00:48+00
 00000001000000240000003E | 16777216 | 2022-08-02 05:59:53+00
```

При настроенном архивировании WAL-журнала в каталоге pg_wal создается каталог archive_status, который хранит статусы архивирования всех сегментов (требующих архивирования). Для получения его содержимого используется функция pg_ls_archive_statusdir, которая также выводит имя, размер и время последнего изменения всех обычных файлов в каталоге (файлы с именами, начинающимися с точки, каталоги и другие специальные файлы не выводятся).

```
# SELECT * FROM pg_ls_archive_statusdir() ORDER BY 3 DESC LIMIT 5;
                      name                       | size |      modification
-------------------------------------------------+------+------------------------
 000000010000002400000041.done                   |    0 | 2022-08-02 06:02:55+00
 000000010000002400000040.done                   |    0 | 2022-08-02 06:01:50+00
 00000001000000240000003F.done                   |    0 | 2022-08-02 06:00:48+00
 00000001000000240000003E.done                   |    0 | 2022-08-02 05:59:53+00
 000000010000000000000011.00000028.backup.done | |    0 | 2022-07-31 08:53:19+00
```

По умолчанию доступ к этой функции имеют только суперпользователи и члены группы pg_monitor, но право на ее выполнение можно дать и другим пользователям.

Обе функции — pg_ls_waldir и pg_ls_archive_statusdir — могут использоваться в целях мониторинга для получения информации о размере каталогов и очереди архивирования. Более подробно эта тема рассмотрена в главе 7, посвященной мониторингу репликации.

Вывод содержимого каталогов с временными файлами. Следующая функция предназначена для вывода содержимого каталогов с временными файлами. Функция pg_ls_tmpdir выводит имя, размер и время последнего изменения всех обычных файлов в каталоге для временных объектов. Таких каталогов может быть несколько, размещаются они в каталогах табличных пространств. Функция учитывает эту особенность и в качестве аргумента может принимать имя конкретного табличного пространства (если имя не указано, используется pg_default). Файлы с именами, начинающимися с точки, каталоги и другие специальные файлы не выводятся. По умолчанию доступ к функции имеют только суперпользователи и члены группы pg_monitor, но право на ее выполнение можно дать и другим пользователям.

Функция pg_ls_tmpdir может применяться в мониторинге для отслеживания временных файлов, однако есть несколько подходов и особенностей, которые следует учитывать. Более подробно тема мониторинга временных файлов рассмотрена в главе 5, посвященной вводу-выводу.

Прочие функции для вывода содержимого каталогов. Следующие функции относятся к выводу содержимого каталогов, задействованных в репликации и логическом декодировании (которое используется в логической репликации). Все следующие функции выводят имя, размер и время последнего изменения всех обычных файлов в целевых каталогах:

- pg_ls_logicalmapdir выводит содержимое pg_logical/mappings;
- pg_ls_logicalsnapdir выводит содержимое pg_logical/snapshots;

- `pg_ls_replslotdir` выводит содержимое pg_replslot/slot_name, где slot_name — это имя конкретного слота репликации.

Перечисленные функции могут применяться для мониторинга при использовании логической репликации или слотов репликации. Как обычно, файлы с именами, начинающимися с точки, каталоги и другие специальные файлы не выводятся. По умолчанию доступ к функции имеют только суперпользователи и члены группы `pg_monitor`, но право на ее выполнение можно дать и другим пользователям.

Резюме

- СУБД представляет множество абстракций для организации и хранения данных.
- Кластер баз данных — это единый и неделимый набор баз данных.
- Кластер баз данных, табличные пространства, таблицы и индексы являются каталогами и файлами в файловой системе.
- Базы данных являются изолированными друг от друга «контейнерами».
- Схемы являются пространствами имен для группировки объектов.
- Таблицы состоят из основного хранилища данных и вспомогательных служебных слоев.
- Таблицы имеют страничную организацию с фиксированным размером блока.
- TOAST-таблицы являются дополнительными к основным таблицам и предназначены для хранения больших значений.
- В каталоге кластера баз данных есть множество служебных каталогов, не связанных с хранением пользовательских данных.
- СУБД предоставляет администратору набор функций для работы с ее объектами.
- Статистика использования таблиц и индексов позволяет посмотреть на рабочую нагрузку еще с одной стороны.
- В рабочей нагрузке могут возникать ошибки и нежелательные события.
- Размер — важная характеристика объектов СУБД.

Глава 5

Область общей памяти и ввод-вывод

В этой главе мы рассмотрим:

- как анализировать содержимое общей памяти;
- представления `pg_buffercache` и `pg_shmem_allocations`;
- использование буферов в общем кеше;
- как анализировать память клиентских процессов;
- представление `pg_stat_slru`;
- как оценивать ввод-вывод на уровне объектов баз данных;
- как оценивать ввод-вывод на уровне выполнения запросов;
- что такое временные файлы и как оценивать их использование;
- как оценивать ввод-вывод процессов фоновой записи и процесса контрольной точки;
- представление `pg_stat_bgwriter`;
- особенности выполнения контрольных точек;
- как оценивать запись буферов различными процессами СУБД.

В предыдущих главах мы рассмотрели запросы и основные объекты СУБД. Запросы определяют рабочую нагрузку и направлены на объекты СУБД — таблицы и индексы. Для размещения пользовательских и системных данных и выполнения запросов СУБД нужна отдельная область в оперативной памяти. Память является быстрой, но энергозависимой, и для надежного хранения данных используются другие накопители, уступающие ей в скорости. Этим обусловлена необходимость перемещения данных между «быстрой» оперативной памятью и «медленным» основным хранилищем. Такое перемещение данных можно назвать довольно общим термином *ввод-вывод* (input/output). С точки зрения производительности важно понимать объем и скорость выполняемого ввода-вывода, ведь это напрямую влияет на производительность — чем быстрее данные перемещаются между областью в памяти и основным хранилищем, тем быстрее будут выполняться запросы и тем больше их можно выполнить. В этой главе мы рассмотрим средства, имеющиеся в СУБД, и способы оценки, которые могут потребоваться для мониторинга ввода-вывода с точки зрения объектов СУБД, фоновых процессов и выполнения запросов.

5.1. Анализ общей памяти

Область общей памяти используется для оперативного размещения данных таблиц, индексов и большого количества различных служебных структур. Общей она называется потому, что все процессы СУБД имеют к ней равный доступ и изменения, внесенные одним процессом, становятся доступными всем остальным. Как правило, самой большой частью общей памяти является *буферный*, или, как его еще называют, *общий кеш*, который используется для размещения пользовательских данных.

С буферным кешем почти никогда не возникает проблем ровно до тех пор, пока он способен вместить себя все данные из основного хранилища (основной каталог данных и табличные пространства). Однако, когда объем хранимых данных превышает объем общего кеша, появляется задача эффективного использования отведенного объема: как держать в нем нужные данные и замещать ненужные в случае нехватки места. Могут возникнуть вопросы: какие системные структуры находятся в общей памяти и сколько они занимают места? Какие объекты (таблицы, индексы) находятся в буферном кеше? Каковы размеры и доля этих объектов относительно друг друга? Насколько эффективно используется буферный кеш? Какой процент обращений заканчивается успехом? Как часто данные не удается найти и приходится обращаться к основному хранилищу? Для ответа на эти вопросы СУБД предлагает несколько инструментов, которые можно использовать как для разового анализа, так и для регулярного мониторинга:

- `pg_buffercache`[1] — отдельное расширение с представлением (на основе функции), которое показывает использование буферов общего кеша;
- `pg_shmem_allocations`[2] — представление с информацией о том, какие служебные структуры находятся в общей памяти.

С помощью этих представлений можно получить информацию о содержимом общей памяти с точки зрения как таблиц и индексов, так и внутренних служебных структур. Давайте более подробно рассмотрим оба инструмента и сценарии их использования.

Представление pg_buffercache

Расширение `pg_buffercache` предоставляет возможность анализа буферного кеша с точки зрения пользовательских объектов и помогает получить ответы на такие вопросы, как:

- страницы каких таблиц и индексов находятся в общем кеше;
- какое количество буферов используется для размещения таблиц и индексов, их процентное отношение относительно всего объема кеша;
- сколько свободных и занятых («чистых» или «грязных») буферов находятся сейчас в кеше.

[1] postgrespro.ru/docs/postgresql/current/pgbuffercache
[2] postgrespro.ru/docs/postgresql/current/view-pg-shmem-allocations

Для использования расширения достаточно установить его с помощью команды CREATE EXTENSION. После установки появятся соответствующие функция и представление. В тестовом окружении расширение уже установлено и готово к использованию.

```
# SELECT * FROM pg_buffercache LIMIT 10;
 bufferid | relfilenode | reltablespace | reldatabase | relforknumber | relblocknumber | isdirty | usagecount | pinn...
----------+-------------+---------------+-------------+---------------+----------------+---------+------------+-----...
        1 |        1262 |          1664 |           0 |             0 |              0 | f       |          5 |
        2 |        1260 |          1664 |           0 |             0 |              0 | f       |          5 |
        3 |        1259 |          1663 |           5 |             0 |              0 | f       |          5 |
        4 |        1259 |          1663 |           5 |             0 |              1 | f       |          5 |
        5 |        1259 |          1663 |           5 |             0 |              2 | f       |          5 |
        6 |        1259 |          1663 |           5 |             0 |              3 | f       |          5 |
        7 |        1249 |          1663 |           5 |             0 |              0 | f       |          5 |
        8 |        1249 |          1663 |           5 |             0 |              1 | f       |          5 |
        9 |        1249 |          1663 |           5 |             0 |              2 | f       |          5 |
       10 |        1249 |          1663 |           5 |             0 |              3 | f       |          5 |
```

Каждая строка описывает отдельный буфер; соответственно, чем больше размер буферного кеша, тем больше будет строк в представлении. На основе практического опыта отмечу, что не помню случаев, когда бы могла понадобиться информация по отдельно взятому буферу: обычно нужна статистика, сгруппированная по объектам, базам и таблицам. При составлении запросов к pg_buffercache важно учитывать, что расширение устанавливается в конкретную БД, но статистика описывает все буферы, в том числе и те, что ассоциированы с объектами в других БД. Для преобразования числовых идентификаторов в имена таблиц и других отношений можно выполнить соединение с pg_class, но такое преобразование будет работать только для объектов из данной БД. В таких случаях рекомендуется добавить условие, чтобы выводить буферы только для текущей БД, а не всех вообще.

Представление содержит несколько полей:

- bufferid — идентификатор конкретного буфера. У свободных (неиспользуемых) буферов значения всех полей, кроме bufferid, будут отсутствовать (NULL);
- relfilenode — номер файла отношения (ссылается на pg_class.relfilenode); по этому номеру объект можно найти в основном каталоге данных;
- reltablespace — идентификатор табличного пространства отношения (ссылается на поле pg_tablespace.oid);
- reldatabase — идентификатор базы данных (ссылается на pg_database.oid). Общие объекты, принадлежащие системному каталогу, будут иметь нулевое значение;
- relforknumber — идентификатор, указывающий на слой отношения (main, visibility map, free space map, init);
- relblocknumber — номер блока внутри отношения;

- `isdirty` — флаг, указывающий на то, что буфер является грязным и содержимое находящегося в нем блока отличается от содержимого блока в основном хранилище;
- `usagecount` — счетчик обращений к буферу, который используется алгоритмом Clock Sweep для вытеснения неиспользуемых страниц из общего кеша[1];
- `pinning_backends` — текущее количество закреплений буфера клиентскими процессами.

Закрепление буферов

Пока процесс читает страницу, буфер необходимо блокировать от попыток изменения другими процессами, но долгая блокировка буфера негативно скажется на конкурентной работе. Благодаря правилам видимости строк блокировка буфера нужна только на время чтения оглавления страницы; однако по-прежнему важно, чтобы страница не была вытеснена из буфера и содержимое страницы не изменилось кардинально. Чтобы ограничить спектр возможных действий с буфером, не блокируя его, используется так называемое *закрепление* (pin) буфера, которое позволяет другим процессам читать и изменять данные, но не позволяет вытеснять страницу из буфера или очищать ее.

Используя числовые идентификаторы, можно точно определить объекты (таблицы и индексы) в буферном кеше, их принадлежность к конкретным базам данных и даже с помощью функции `pg_relation_filepath` узнать их расположение в файловой системе.

На основе полей `isdirty`, `usagecount` и `pinning_backends` можно выделить набор состояний, в которых могут находиться буферы, и таким образом определить степень использования буферного кеша. После инициализации буферного кеша на старте СУБД бóльшая часть буферов не используется и не ассоциирована ни с одним из блоков (значение `bufferid` отсутствует). При выполнении запросов СУБД обращается к основному хранилищу и заполняет буферный кеш данными таблиц и индексов; буферы из свободного состояния переходят в занятое. В этом состоянии буферы ассоциированы с конкретными блоками таблиц и индексов. Такое состояние определяется полями `reltablespace`, `reldatabase` и `relfilenode`, которые вместе указывают на ассоциированный с буфером объект.

Также можно принять во внимание значение `usagecount`, которое показывает текущую интенсивность обращений к буферу. Этот счетчик увеличивается на единицу при каждом доступе (максимальное значение — 5) и уменьшается на единицу при каждом проходе алгоритма вытеснения. Блок вытесняется и заменяется другим блоком, если к моменту уменьшения счетчика его значение уже равно нулю. Таким образом, по значению `usagecount` (от 0 до 5) можно определить, насколько активно используются буферы.

Состояние занятого буфера можно детализировать. Так, например, к буферу могут обращаться клиентские процессы, читая или изменяя данные блока, на что указывает ненулевое значение `pinning_backends`. Если содержимое буфера было изменено, он становится грязным, и эти изменения должны быть перенесены в ассоциированный с буфером блок в основном хранилище. На грязное состояние указывает флаг `isdirty`.

[1] www.interdb.jp/pg/pgsql08.html#_8.4.4.

Таким образом, использование буферного кеша можно рассматривать как минимум в двух проекциях:

1. На верхнем уровне по свободным, используемым, закрепленным и грязным буферам.

2. На детальном уровне на основе счетчика обращений (usagecount), значения которого варьируются от 0 до 5.

При использовании pg_buffercache в регулярном мониторинге стоит учитывать, что представление показывает только снимок состояния буферов в момент опроса; состояние буферов между опросами остается неизвестным. Это поведение аналогично тому, как работают pg_locks и pg_stat_activity.

Влияние на производительность

До версии PostgreSQL 10 внутренний механизм pg_buffercache использовал блокировки для получения согласованного состояния всех буферов. Эти блокировки приводили к снижению производительности в системах с интенсивной рабочей нагрузкой. Начиная с версии 10 блокировки в менеджере буферов больше не устанавливаются. Поэтому обращения к pg_buffercache стали меньше влиять на обычную активность буферов, но набор результатов, полученный для всех буферов, в целом может оказаться несогласованным (что, как правило, некритично). Внутри каждого отдельного буфера согласованность информации по-прежнему гарантируется.

Давайте вернемся чуть назад, к SQL-запросу к pg_buffercache. Практически весь его вывод представлен числовыми идентификаторами; для получения более понятной информации потребуется соединить результат с дополнительными источниками информации и проделать некоторые преобразования.

```
# SELECT
    n.nspname,
    c.relname,
    count(*) AS buffers,
    pg_size_pretty(count(*) * 8192) AS bytes
  FROM pg_buffercache b
  JOIN pg_class c
    ON b.relfilenode = pg_relation_filenode(c.oid) AND
       b.reldatabase IN (
         0,
         (SELECT oid FROM pg_database WHERE datname = current_database())
       )
  JOIN pg_namespace n ON n.oid = c.relnamespace
  GROUP BY n.nspname, c.relname
  ORDER BY 3 DESC
  LIMIT 10;
```

```
  nspname   |            relname             | buffers |  bytes
------------+--------------------------------+---------+---------
 public     | pgbench_accounts               |    9902 | 77 MB
 public     | pgbench_accounts_pkey          |    5234 | 41 MB
 public     | pgbench_tellers                |     290 | 2320 kB
 public     | pgbench_history                |     198 | 1584 kB
 pg_catalog | pg_proc                        |      99 | 792 kB
 pg_catalog | pg_attribute                   |      38 | 304 kB
 pg_catalog | pg_class                       |      15 | 120 kB
 pg_catalog | pg_proc_proname_args_nsp_index |      14 | 112 kB
 pg_catalog | pg_operator                    |      14 | 112 kB
 pg_catalog | pg_statistic                   |      13 | 104 kB
```

Полученный результат стал более понятен: вместо числовых идентификаторов объектов теперь выводятся имена, статистика сгруппирована по объектам, а блоки просуммированы и дополнительно выражены в байтах. Теперь видно, что бóльшую часть в буферном кеше (в тестовом окружении значение *shared_buffers* установлено в 128 МБ) занимают таблица pgbench_accounts и ее индекс.

Преобразование в байты

В тестовом окружении мне заранее известен размер блока, и в примерах запросов используется константа 8192. Однако на практике размер блока может быть переопределен на этапе компиляции из исходных кодов, и в таких случаях преобразование будет неверным. Вместо константы можно использовать более универсальное решение и получать актуальный размер блока из конфигурации СУБД вызовом функции `current_setting('block_size')`. Такой способ предпочтителен, когда нужно выразить размер блоков в байтах, но при этом точный размер блока неизвестен.

Следующим запросом можно получить информацию об использовании буферного кеша и узнать соотношение объемов буферов, находящихся в разных состояниях.

```
# SELECT
    pg_size_pretty(count(*) FILTER (
        WHERE reldatabase IS NULL) * 8192) AS free,
    pg_size_pretty(count(*) FILTER (
        WHERE pinning_backends = 0 AND isdirty = 'f') * 8192) AS clean,
    pg_size_pretty(count(*) FILTER (
        WHERE pinning_backends > 0 AND isdirty = 'f') * 8192) AS "clean/pinned",
    pg_size_pretty(count(*) FILTER (
        WHERE pinning_backends = 0 AND isdirty = 't') * 8192) AS dirty,
    pg_size_pretty(count(*) FILTER (
        WHERE pinning_backends > 0 AND isdirty = 't') * 8192) AS "dirty/pinned"
  FROM pg_buffercache;
```

```
  free   | clean | clean/pinned | dirty  | dirty/pinned
---------+-------+--------------+--------+--------------
 0 bytes | 26 MB | 8192 bytes   | 102 MB | 40 kB
```

В выводе запроса видно, что бóльшая часть буферов является грязными. Это говорит о том, что вносится довольно много изменений, и это действительно так: в главе, посвященной запросам, мы выяснили, что в тестовой рабочей нагрузке преобладают запросы на обновление записей в pgbench_accounts. В тестовом окружении агент мониторинга использует подобный запрос для получения метрики postgres_shared_buffers_all_usage_bytes, на основе которой можно построить следующий график (рис. 5.1):

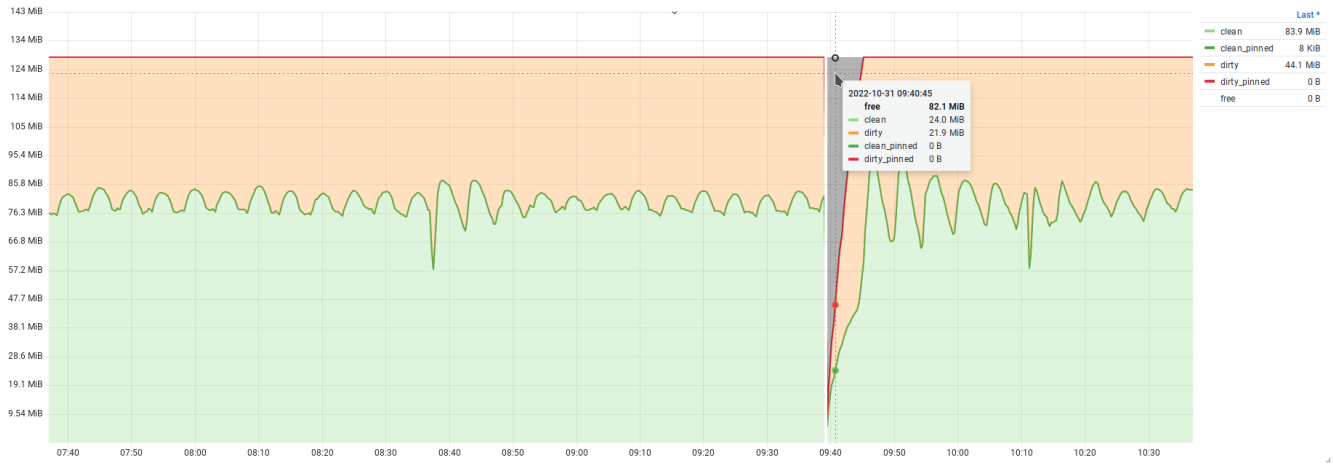


Рис. 5.1. Использование буферов в общем кеше

На графике запечатлен момент перезапуска контейнера с СУБД, при котором происходит сброс буферного кеша. Некоторое время в буферном кеше присутствуют свободные буферы, однако они довольно быстро заканчиваются.

Другим вариантом отображения может быть график, на котором статистика сгруппирована по именам баз данных. Так можно отслеживать заполненность буферного кеша страницами конкретных баз данных, что может быть полезно, если в кластере используется много баз.

Представление pg_shmem_allocations

Расширение pg_buffercache показывает данные только о буферах, используемых под кеш для таблиц и индексов. Но кроме буферного кеша, в общей памяти размещаются различные служебные структуры, доступные всем процессам СУБД, как клиентским, так и фоновым. Для структурного анализа общей памяти можно использовать представление pg_shmem_allocations.

```
# SELECT *
  FROM pg_shmem_allocations
  ORDER BY size DESC LIMIT 10;
        name         |    off    |   size    | allocated_size
---------------------+-----------+-----------+----------------
 Buffer Blocks       |   6843520 | 134217728 |      134217728
 <anonymous>         |           |   6710784 |        6710784
 XLOG Ctl            |     54144 |   4208200 |        4208256
                     | 149586304 |   1900160 |        1900160
 Buffer Descriptors  |   5794944 |   1048576 |        1048576
 CommitTs            |   4792192 |    533920 |         534016
 Xact                |   4263040 |    529152 |         529152
 Checkpointer Data   | 146862208 |    393280 |         393344
 Checkpoint BufferIds| 141323392 |    327680 |         327680
 Subtrans            |   5326336 |    267008 |         267008
```

Каждая строка описывает отдельный участок памяти, выделенный под конкретную служебную структуру. Самая большая структура в списке — область `Buffer Blocks`. Именно в ней размещается буферный кеш с данными пользовательских объектов, который можно детально исследовать с помощью `pg_buffercache`. Если посмотреть на полный вывод представления, то можно заметить, что в общей памяти размещается большое количество структур. Например, для версии 15 их около шестидесяти, а итоговое число может варьироваться в зависимости от конфигурации, используемых модулей и т. п. В самом представлении не так много полей:

- `name` — имя сегмента, по которому можно примерно понять его назначение. Сегмент, у которого отсутствует имя (`NULL`), представляет неиспользуемую память; сегменты с именем `<anonymous>` являются анонимными и могут использоваться для хранения информации о блокировках;
- `off` — смещение в области общей памяти, с которого начинается сегмент. У анонимных сегментов значение смещения отсутствует (`NULL`);
- `size` — размер сегмента в байтах;
- `allocated_size` — размер сегмента с учетом выравнивания (padding). Для свободной памяти и анонимных сегментов значения `size` и `allocated_size` всегда равны.

В повседневном мониторинге информация из `pg_shmem_allocations` не очень полезна. По большей части она может пригодиться в редких случаях поиска и устранения проблем или для контроля эффективности использования памяти при разработке и отладке новой функциональности СУБД.

5.2. Анализ памяти клиентских процессов

Еще два инструмента, которые также могут пригодиться для отладки и поиска проблем, — это представление `pg_backend_memory_contexts` и функция `pg_log_backend_memory_contexts`. Оба

инструмента предназначены для отображения информации по используемым сегментам памяти конкретного процесса, но поведение представления и функции отличаются друг от друга. Представление показывает распределение памяти только для процесса, ассоциированного с текущим сеансом, и его невозможно использовать, чтобы посмотреть раскладку памяти соседних процессов (которые, например, заняты выполнением долгого запроса). Для таких случаев следует использовать функцию pg_log_backend_memory_contexts, которая принимает в качестве аргумента идентификатор процесса и сбрасывает раскладку памяти в журнал СУБД.

```
# SELECT * FROM pg_backend_memory_contexts
  ORDER BY used_bytes DESC LIMIT 10;
         name           | ident |      parent      | level | total_bytes | total_nblocks | free_bytes | free_chunks
------------------------+-------+------------------+-------+-------------+---------------+------------+------------...
 CacheMemoryContext     |       | TopMemoryContext |     1 |     1048576 |             8 |     499872 |           2
 Timezones              |       | TopMemoryContext |     1 |      104120 |             2 |       2616 |           0
 TopMemoryContext       |       |                  |     0 |       97680 |             5 |      12232 |           7
 ExecutorState          |       | PortalContext    |     3 |       49208 |             4 |       4280 |           3
 WAL record construction |       | TopMemoryContext |     1 |       50208 |             2 |       6360 |           0
 MessageContext         |       | TopMemoryContext |     1 |       73728 |             4 |      30856 |           4
 TupleSort main         |       | ExecutorState    |     4 |       32824 |             2 |       6808 |           7
 Type information cache |       | TopMemoryContext |     1 |       24376 |             2 |       2616 |           0
 smgr relation table    |       | TopMemoryContext |     1 |       32768 |             3 |      12712 |           8
 Operator lookup cache  |       | TopMemoryContext |     1 |       24576 |             2 |      10752 |           3
```

В каждой строке содержится информация о конкретном участке памяти (так называемый *контекст*), метрики использования, а также дополнительные данные: родительский контекст, уровень вложенности, количество блоков (total_nblocks) и свободных фрагментов (free_chunks).

Применение этих инструментов может пригодиться скорее для отладки, поиска и устранения проблем, чем для регулярного мониторинга. Однако на данный момент (версия 15) это единственные инструменты, позволяющие детально проанализировать состав и использование памяти процессов СУБД.

5.3. Оценка использования SLRU-кешей

Осталось рассмотреть еще одну группу кешей, SLRU-кеши (Simple Least-Recently-Used). Это отдельные кеши, призванные ускорять доступ к некоторым служебным структурам, которые размещаются на диске в основном каталоге данных. Примерами таких служебных структур являются:

- структуры журнала фиксации транзакций (clog, commit log);
- структуры, обеспечивающие работу вложенных транзакций (subtransactions);
- структуры отслеживания времени фиксации транзакций (см. *track_commit_timestamp*);

- структуры, связанные с подсистемой уведомлений (команды LISTEN и NOTIFY);
- прочие структуры, использующие кеши на основе SLRU-алгоритма.

Для отслеживания использования SLRU-кешей применяется представление pg_stat_slru:

```
# SELECT * FROM pg_stat_slru;
      name       | blks_zeroed | blks_hit | blks_read | blks_written | blks_exists | flushes | truncates |         ...
-----------------+-------------+----------+-----------+--------------+-------------+---------+-----------+---------...
 CommitTs        |           0 |        0 |         0 |            0 |           0 |    1424 |         0 | 2022-11-0...
 MultiXactMember |           1 |        0 |         0 |            1 |           0 |    1424 |         0 | 2022-11-0...
 MultiXactOffset |           1 |        1 |         4 |            5 |           0 |    1424 |         0 | 2022-11-0...
 Notify          |           0 |        0 |         0 |            0 |           0 |       0 |         0 | 2022-11-0...
 Serial          |           0 |        0 |         0 |            0 |           0 |       0 |         0 | 2022-11-0...
 Subtrans        |        7522 |        0 |         0 |         7462 |           0 |    1424 |         0 | 2022-11-0...
 Xact            |         471 | 79554357 |        48 |         1889 |           0 |    1424 |         0 | 2022-11-0...
 other           |           0 |        0 |         0 |            0 |           0 |       0 |         0 | 2022-11-0...
```

Каждая строка содержит накопленную статистику использования конкретного SLRU-кеша (список кешей фиксирован):

- name — название SLRU-кеша;
- blks_zeroed — количество блоков, заполненных нулями во время инициализации;
- blks_hit — количество блоков, найденных в кеше;
- blks_read — количество блоков, которые потребовалось прочитать из основного хранилища на диске;
- blks_written — количество грязных блоков, которые потребовалось записать из кеша на диск;
- blks_exists — количество блоков, успешно найденных на диске;
- flushes — количество выполненных запросов на синхронизацию грязных блоков;
- truncates — количество выполненных запросов на удаление блоков ввиду их ненужности (при операциях обслуживания);
- stats_reset — отметка времени последнего сброса статистики.

Статистика использования SLRU-кешей пока может применяться только для мониторинга, поскольку конфигурация СУБД не предусматривает параметров, позволяющих регулировать их размеры (однако не исключено, что такие параметры могут появиться в будущем). Основной сценарий использования статистики — это оценка эффективности кешей.

5.4. Ввод-вывод в контексте объектов СУБД

Статистика pg_buffercache, рассмотренная ранее, позволяет анализировать только общую картину использования буферного кеша. Мониторинг буферного кеша с помощью этого представления не дает полной информации об объеме ввода-вывода, поскольку содержит информацию

только о тех объектах, что находятся в кеше в данный момент, и из поля зрения может пропасть множество других объектов БД, доступ к которым осуществлялся между снимками статистики. Для более полного учета ввода-вывода нужна статистика накопительного характера по всем объектам БД. Эта информация располагается в нескольких представлениях, с одним из которых нам уже приходилось сталкиваться:

- `pg_stat_database` содержит часть статистики ввода-вывода для отдельных баз данных;
- `pg_statio_all_tables` — статистика ввода-вывода по таблицам (включая TOAST и временные таблицы);
- `pg_statio_all_indexes` — статистика ввода-вывода по индексам;
- `pg_statio_all_sequences` — статистика ввода-вывода по последовательностям.

Базы данных

Представление `pg_stat_database` уже знакомо нам из предыдущих глав, и сейчас мы возвращаемся к нему, потому что оно содержит несколько полей с информацией о вводе-выводе в разрезе отдельных баз данных:

- `blks_hit` — количество блоков, найденных в общем и локальных кешах;
- `blks_read` — количество блоков, прочитанных с диска (включая найденные в страничном кеше ОС);
- `blk_read_time` — время, затраченное на чтение с диска, в миллисекундах;
- `blk_write_time` — время, затраченное на запись на диск, в миллисекундах.

Учет времени доступен только при включенном параметре *track_io_timing*. Для полноты картины не хватает только статистики по записанным и грязным блокам. На основе этих данных можно получить общее представление об эффективности использования кеша, однако важно помнить, что статистика содержит данные как по общему, так и по локальным кешам, что имеет особое значение при широком использовании временных таблиц.

Оценить эффективность и посчитать коэффициент попаданий в кеш можно следующим запросом:

```
# SELECT sum(blks_hit) / sum(blks_hit + blks_read) AS hit_ratio
  FROM pg_stat_database;
       hit_ratio
------------------------
 0.9847338704969256038 4
```

Полученная метрика показывает, насколько эффективно используется кеш. Значения, близкие к единице, говорят о том, что нужные данные успешно обнаруживаются в кеше. И наоборот, чем ближе значения к нулю, тем реже нужные данные обнаруживаются в кеше и тем чаще СУБД приходится обращаться к основному хранилищу, чтобы прочитать данные и затем загрузить

их в кеш, при этом вытесняя из кеша другие данные. При постоянной низкой эффективности кеша много времени тратится на дисковый ввод-вывод, и это негативно сказывается на общей производительности. К сожалению, здесь нет простого или универсального решения, и повысить эффективность кеша можно разными способами. Самый очевидный — это увеличение общего кеша, однако такой способ не всегда приводит к успеху. Более правильным способом является выявление тех запросов, которые осуществляют бóльшую часть ввода-вывода или тратят на это значительное время, и попытка их оптимизации (с помощью рефакторинга или добавления индексов). За счет такой оптимизации можно в десятки и сотни раз улучшить производительность запросов и увеличить эффективность использования кеша, не прибегая к изменению конфигурации.

Еще один способ использования статистики — это отслеживание времени, затраченного на ввод-вывод, с помощью `blk_read_time` и `blk_write_time` (рис. 5.2):

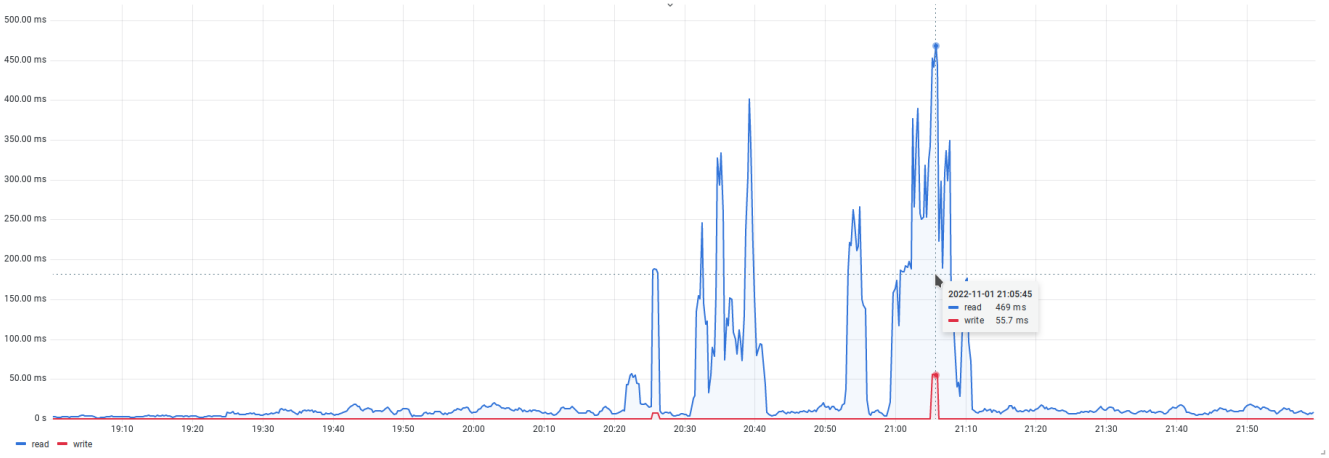


Рис. 5.2. Время, затраченное на чтение и запись блоков

Такой график хорошо подходит для общей оценки состояния СУБД и отражает моменты, когда СУБД была вынуждена тратить время на чтение и запись, в результате чего могли возникать просадки производительности при выполнении запросов.

Таблицы, индексы и последовательности

Для получения более детальной информации о происходящем вводе-выводе на уровне отдельных объектов СУБД можно воспользоваться следующими представлениями:

- `pg_statio_all_tables` содержит статистику по каждой таблице:
  - `heap_blks_read` — количество блоков, прочитанных с диска;
  - `heap_blks_hit` — количество блоков, найденных в кеше;
  - `idx_blks_read` — количество блоков всех индексов, принадлежащих этой таблице, прочитанных с диска;

  - `idx_blks_hit` — количество блоков всех индексов, принадлежащих этой таблице, найденных в кеше;
  - `toast_blks_read` — количество блоков TOAST-таблицы, прочитанных с диска;
  - `toast_blks_hit` — количество блоков TOAST-таблицы, найденных в кеше;
  - `tidx_blks_read` — количество блоков индексов TOAST-таблицы, прочитанных с диска;
  - `tidx_blks_hit` — количество блоков индексов TOAST-таблицы, найденных в кеше.
- `pg_statio_all_indexes` содержит статистику по каждому индексу:
  - `idx_blks_read` — количество блоков, прочитанных с диска;
  - `idx_blks_hit` — количество блоков, найденных в кеше.
- `pg_statio_all_sequences` содержит статистику по каждой последовательности:
  - `blks_read` — количество блоков, прочитанных с диска;
  - `blks_hit` — количество блоков, найденных в кеше.

Перечисленные представления содержат статистику по всем объектам, включая пользовательские и системные (которые принадлежат системному каталогу). Раздельную статистику по пользовательским и системным объектам можно найти в представлениях с префиксами `pg_statio_sys_` и `pg_statio_user_`.

Отдельного внимания заслуживает статистика попаданий в кеш (поля с суффиксом `_hit`). Когда блок найден в буферном кеше, ввода-вывода не происходит: блок уже находится в памяти, принадлежащей СУБД, так что достаточно прочитать в буфере нужную строку или даже ее часть. Важно понимать что статистика попаданий в кеш показывает объем найденных в кеше данных, но при этом реальный объем потребовавшихся данных может быть меньшим. При организации мониторинга нужно стараться избегать попыток прямого сравнения попаданий в кеш с объемом чтения и особенно представления обоих значений в байтах. Для оценки эффективности кеша лучше оценивать отношение попаданий в кеш к общему количеству обращений к данным (попаданиям в кеш и чтениям из хранилища). То же самое справедливо и в отношении статистики загрязнения буферов и записи страниц (поля `blks_dirtied` и `blks_written` в представлении `pg_stat_statements`): загрязнение блока при вставке, обновлении или удалении отдельных строк является лишь частичным изменением буфера и не приводит к вводу-выводу в отличие от записи блока непосредственно в основное хранилище. Этот нюанс важно учитывать при сведении метрик в один график или вычислении объемов ввода-вывода при составлении отчетов производительности и при выборе общих единиц измерения (блоков или байтов).

Другое важное замечание относится к индивидуальности этой статистики — базы данных являются своего рода изолированными контейнерами для таблиц, индексов и прочих объектов. Следовательно, представления содержат статистику только по тем объектам, которые принадлежат этой базе данных. Для сбора статистики по всем объектам в кластере баз данных необходимо подключаться по очереди к каждой отдельной БД.

Статистика по отдельным объектам дает детальную информацию по вводу-выводу, но обычно она требуется в исключительных случаях оптимизации или поиска узких мест в производительности. За годы моей практики случаи, когда требовалась такая подробная информация, можно пересчитать по пальцам одной руки. Во многих производственных окружениях может быть большое количество баз, таблиц и индексов. Мониторинг будет производить большой объем метрик, перед сбором и хранением которых важно оценить их практическую ценность.

Следующим запросом можно получить объем чтения, который приходится на таблицы базы (рис. 5.3 и 5.4):

```
# sum by (table)
    (rate(postgres_table_io_blocks_total{service_id="primary",access="read"}[1m]))
```

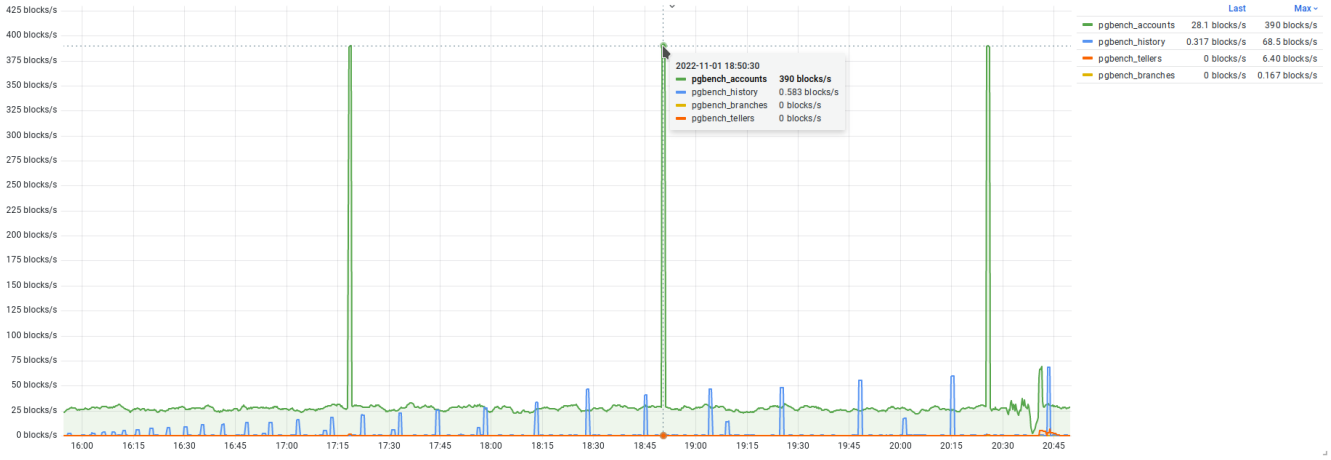


Рис. 5.3. Объем чтения по таблицам

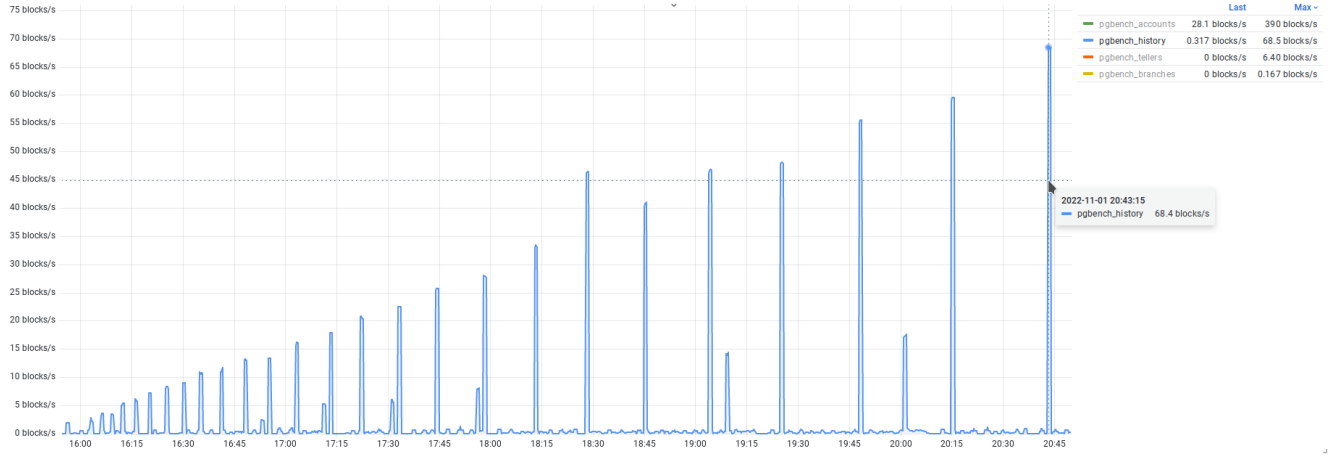


Рис. 5.4. Объем чтения таблицы pgbench_history

Объем ввода-вывода на первом графике (рис. 5.3) более или менее стабилен, но с точки зрения производственной эксплуатации все-таки могут возникнуть вопросы относительно периодических пиков, связанных с таблицей pgbench_accounts, и увеличения объема чтения таблицы pgbench_history, который менее заметен, чем пики, однако проявляется, если выключить остальные метрики (рис. 5.4).

Следующий запрос показывает количество блоков, найденных в кеше и прочитанных с диска для таблицы pgbench_accounts (самая используемая таблица в тестовом окружении; рис. 5.5):

```
# sum by (access) (rate(postgres_table_io_blocks_total{
        service_id="primary", table="pgbench_accounts"
  }[1m]))
```

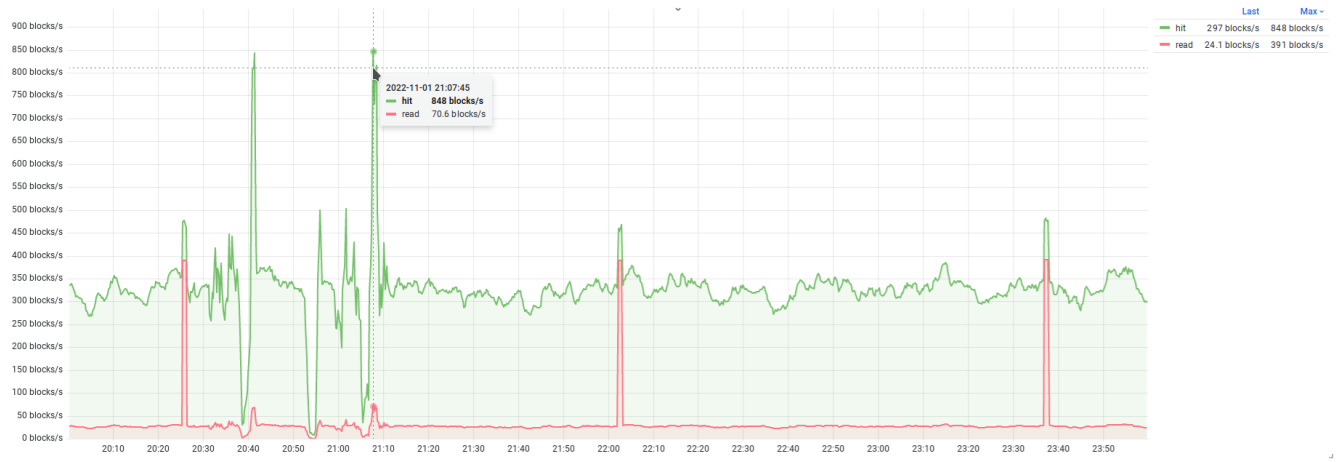


Рис. 5.5. Объем ввода-вывода для таблицы pgbench_accounts

На графике видно, что большая часть блоков читается из кеша и относительно небольшая часть читается из основного хранилища (или страничного кеша ОС), что в общем хорошо для производительности. И, конечно, важно отметить, что в качестве единиц измерения используются блоки без какого-либо преобразования.

5.5. Ввод-вывод в контексте выполнения запросов

Для анализа выполняемых запросов существует расширение pg_stat_statements, с которым мы успели познакомиться в предыдущих главах. Представление содержит массу полезной информации, в том числе и статистику ввода-вывода по каждому типу запроса. В отличие от pg_stat_database и pg_statio_* статистика по запросам более полная:

- shared_blks_hit — количество блоков, найденных в буферном кеше;

- shared_blks_read — количество блоков, прочитанных с диска или из страничного кеша ОС в общий буферный кеш;
- shared_blks_dirtied — количество блоков в общем буферном кеше, ставших грязными в результате внесения изменений;
- shared_blks_written — количество блоков в общем буферном кеше, содержимое которых было записано на диск.

Статистику дополняют аналогичные поля с данными по локальным кешам: local_blks_hit, local_blks_read, local_blks_dirtied и local_blks_written. В главе 3, посвященной запросам, рассмотрены поля blk_read_time и blk_write_time, которые показывают время, затраченное на ввод-вывод.

Статистику можно использовать в следующих сценариях:

- определение объемов ввода-вывода конкретными запросами;
- выявление запросов к временным таблицам и объем ввода-вывода, связанный с ними (для работы с временными таблицами используются локальные кеши, их размер определяется параметром *temp_buffers*);
- оценка эффекта от внесенных оптимизаций (как изменился объем ввода-вывода при выполнении конкретных запросов);
- оценка ввода-вывода запросов при определении приоритета для оптимизации (оптимизация запроса с бóльшим объемом ввода-вывода будет заметнее и эффективнее с точки зрения утилизации ресурсов).

Также возможны и другие сценарии использования в каких-то особых случаях поиска и устранения проблем производительности.

5.6. Временные файлы

Исполнение запросов, особенно сложных, состоит из нескольких шагов, соответствующих узлам плана. На этих шагах обычно происходит определенная работа с данными (выборка, фильтрация, сортировка, сравнение и т. д.). Для некоторых типов работ, таких как сортировка данных, построение хеш-таблицы для соединений и агрегаций или материализация CTE, может потребоваться временный буфер. Для него выделяется рабочая память размером *work_mem*, обычно небольшая — по умолчанию 4 МБ. При этом запросы могут оперировать значительно бóльшими объемами данных. В случае нехватки рабочей памяти создается временный файл на диске и часть данных переносится в него. После того как запрос выполнен, все потребовавшиеся для его выполнения временные файлы удаляются. Ожидаемая особенность использования временных файлов — замедление производительности запросов, поскольку необходимые

данные приходится записывать на диск и считывать с диска. С точки зрения производительности следует отслеживать все запросы, которые используют временные файлы и предпринимать шаги по их оптимизации с целью исключения необходимости ввода-вывода.

Статистика СУБД предлагает несколько источников для отслеживания временных файлов, которые могут пригодиться в разных сценариях:

- pg_stat_database содержит статистику по созданным временным файлам в контексте отдельных баз данных;
- pg_stat_statements содержит статистику по вводу-выводу во временные файлы в контексте выполнения запросов;
- в журналах сообщений СУБД также могут фиксироваться случаи использования временных файлов (см. *log_temp_files*), включая и тексты запросов.

Всем трем источникам присуща общая особенность: статистика использования фиксируется только в момент завершения запроса. Это неудобно, когда запрос выполняется долго и уже использует временные файлы, — обновление статистики произойдет только после его *успешного* завершения, а в случае аварийного завершения из-за тайм-аута (*statement_timeout* и т. п.) или принудительного завершения администратором (функцией pg_terminate_backend) фиксации не произойдет.

Далее мы рассмотрим все способы получения статистики по временным файлам.

Уровень баз данных

Представление pg_stat_database содержит поля temp_files и temp_bytes, которые указывают на суммарное количество временных файлов и их размер. Как известно, каждая строка pg_stat_database содержит накопленную статистику по конкретной БД, соответственно, указанные поля будут содержать кумулятивную статистику за все время накопления (время сброса статистики указано в stats_reset).

```
# SELECT datname, temp_files, pg_size_pretty(temp_bytes) AS temp_size
  FROM pg_stat_database WHERE temp_files > 0
  ORDER BY temp_bytes DESC;
 datname | temp_files | temp_size
---------+------------+-----------
 pgbench |          4 | 38 MB
         |          1 | 178 kB
```

Запросы, выполняемые в тестовом окружении, практически не используют временных файлов, однако какую-то статистику все же можно увидеть. Далее для экспериментов будет использоваться следующий запрос, для выполнения которого гарантированно потребуется временный файл (при значении *work_mem* по умолчанию):

```
# SELECT a.*, b.* FROM pg_class a, pg_class b ORDER BY random();
```

Запрос выполняется несколько секунд и создает временный файл размером около 60 МБ. Повторив запрос к pg_stat_database, можно обнаружить, что в БД postgres зафиксировано использование временного файла:

```
# SELECT datname, temp_files, pg_size_pretty(temp_bytes) AS temp_size
  FROM pg_stat_database
  WHERE temp_files > 0
  ORDER BY temp_bytes DESC;
 datname  | temp_files | temp_size
----------+------------+-----------
 postgres |          1 | 58 MB
 pgbench  |          4 | 38 MB
          |          1 | 178 kB
```

С помощью метрики postgres_database_temp_bytes_total можно получить график (рис. 5.6):

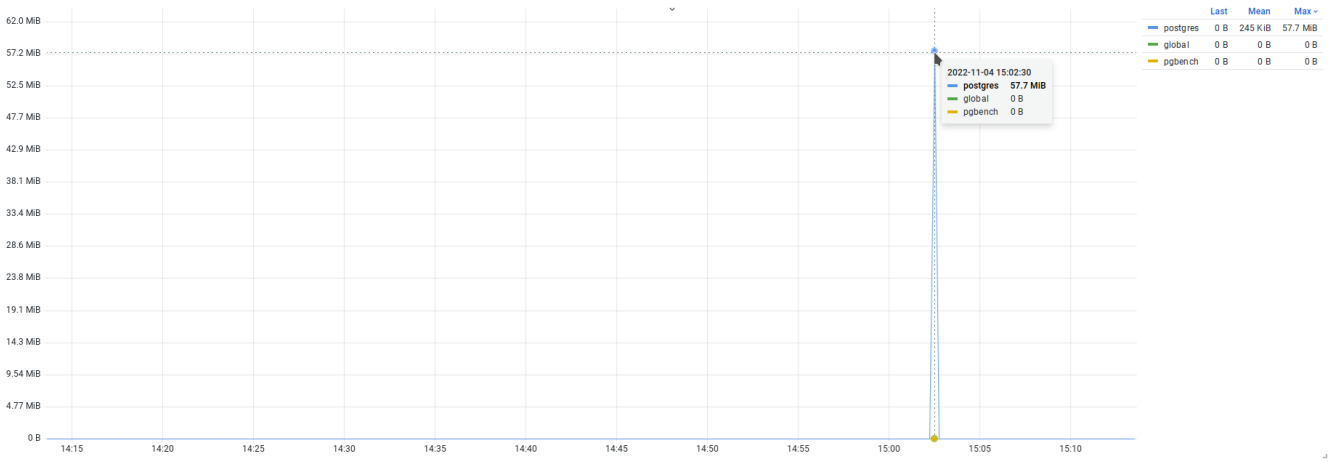


Рис. 5.6. Временный файл в БД postgres

Обычно такой график нужен для поверхностного отслеживания временных файлов: как только замечены критические уровни использования временных файлов, можно перейти к более детальному анализу с помощью pg_stat_statements или журналов сообщений.

Важной особенностью счетчика temp_bytes является фиксация итогового размера временного файла. Этот размер не следует путать с объемом ввода-вывода, связанного с файлом: чтение и запись в файл можно осуществлять многократно, и объем выполненного ввода-вывода может многократно превышать конечный размер файла.

Ввод-вывод при выполнении запросов

Расширение pg_stat_statements — это второй источник отслеживания временных файлов. Следующие столбцы одноименного представления показывают объем ввода-вывода запросов и время, затраченное на работу с временными файлами:

- temp_blks_read — количество блоков, прочитанных из временных файлов;
- temp_blks_written — количество блоков, записанных во временные файлы;
- temp_blk_read_time — время, затраченное на чтение временных файлов, в миллисекундах;
- temp_blk_write_time — время, затраченное на запись во временные файлы, в миллисекундах.

Отсутствие счетчиков _hits и _dirtied является нормальным, поскольку весь ввод-вывод осуществляется напрямую с файлом без кеширования. Более того, для работы с временным файлом используется техника, отличная от страничной, которая применяется при работе с буферными кешами (общим или локальными): ввод-вывод в этом случае работает с плотно упакованными строками без заголовков и только с нужными полями. В представлении также есть поле dbid, группировка по которому позволяет получить статистику по отдельным базам. Это может навести на мысли о сходстве с pg_stat_database, однако в нем ведется учет количества и размеров временных файлов, а в pg_stat_statements — объема ввода-вывода. Прямое сравнение значений из pg_stat_database и pg_stat_statements довольно бессмысленно.

```
# SELECT calls, total_exec_time,
    pg_size_pretty(temp_blks_read * 8192) AS temp_read_bytes,
    pg_size_pretty(temp_blks_written * 8192) AS temp_written_bytes,
    temp_blk_read_time, temp_blk_write_time,
    left(query, 32) as query
  FROM pg_stat_statements WHERE query ~ 'SELECT a.*, b.*';
-[ RECORD 1 ]-------+------------------------------------------------------------
calls               | 1
total_exec_time     | 3262.312213999999
temp_blks_read      | 115 MB
temp_blks_written   | 115 MB
temp_blk_read_time  | 96.354933
temp_blk_write_time | 257.18505
query               | SELECT a.*, b.* FROM pg_class a, pg_class b ORDER BY random()
```

С точки зрения мониторинга запросов полезны также графики top-K, позволяющие быстро выявить запросы, которые больше остальных используют временные файлы. Однако в тестовом окружении нет подходящих запросов и такой график не покажет ничего интересного.

Отслеживание в журнале сообщений

Следующий способ отслеживания временных файлов предполагает использование журнала сообщений СУБД. По умолчанию запись сообщений о временных файлах выключена, для ее включения следует установить пороговое значение с помощью параметра *log_temp_files*. Если размер временного файла превысит это значение, информация о нем, включая и текст запроса, будет сохранена в журнале сообщений. Записи об использовании временных файлов имеют узнаваемую сигнатуру с текстом temporary file и указанием пути до файла и его размера в байтах, в следующей строке будет указан текст запроса:

```
LOG: temporary file: path "base/pgsql_tmp/pgsql_tmp51374.0", size 60497920
STATEMENT: SELECT a.*, b.* FROM pg_class a, pg_class b ORDER BY random();
```

Следите за журналом

На практике произошел следующий случай. От системы мониторинга пришло уведомление о нехватке свободного места на диске. Довольно быстро выяснилось, что причина — в журналах сообщений: они стали занимать много места. При этом избыточный размер имели только журналы за последние два дня, размеры журналов за предыдущие дни были в десятки раз меньше. Анализ содержимого журналов показал, что в рабочей нагрузке появилось большое количество запросов с временными файлами. Но проблема была не в файлах, а в текстах запросов. Запрос содержал длинный `IN`-список, который автоматически генерировался приложением и не ограничивался по длине. В результате список содержал десятки тысяч `UUID`-значений, а текст запроса достигал нескольких мегабайт. При вставке записи об использовании временного файл дополнительно вставляется запись с текстом огромного запроса, что в результате и привело к увеличению объема журнала.

Отслеживание активных временных файлов

Недостаток отслеживания временных файлов с помощью представлений `pg_stat_database` и `pg_stat_statements` обусловлен тем, что статистика учитывается только после окончания запроса, который использовал временные файлы, а сами временные файлы к тому моменту уже удалены. Это нормально, если запросы выполняются быстро, однако в случае длительных запросов, работающих минутами, такое поведение может стать неприемлемым. В худшем случае количество и размер требуемых выполняющемуся запросу временных файлов могут увеличиваться до тех пор, пока не закончится все свободное место на диске, что может привести к аварийной остановке СУБД. Заметно будет только то, что заканчивается место, но по какой причине оно заканчивается, будет неочевидно. Размеры баз данных, таблиц и индексов при этом будут неизменны, а информация о временных файлах появится только после успешного завершения запроса.

Для эффективного отслеживания нужен способ обнаружения активных временных файлов, с которыми осуществляется работа в данный момент. Для этого нам потребуются функции для работы с файловой системой и некоторое знание об устройстве файловой иерархии каталога данных СУБД.

Начнем с устройства файловой системы. Для размещения временных файлов[1] СУБД руководствуется параметром *temp_tablespaces*, в котором указывается список табличных пространств: одно из них выбирается из этого списка случайным образом, и временный файл создается

[1] postgrespro.ru/docs/postgresql/current/storage-file-layout

в подкаталоге pgsql_tmp выбранного пространства. Однако по умолчанию, да и вообще в большинстве конфигураций, значение *temp_tablespaces* отсутствует (NULL); в таком случае используется табличное пространство по умолчанию для текущей базы данных. Обычно таким табличным пространством является pg_default, и временные файлы создаются в подкаталоге base/pgsql_tmp. Имена временных файлов имеют формат pgsql_tmpPPP.NNN, где PPP — идентификатор клиентского процесса, которому принадлежит файл, а NNN — порядковый номер файла, которым отличаются файлы, принадлежащие одному процессу.

Зная каталоги, в которых размещаются временные файлы, остается просмотреть их и с помощью функций доступа к файловой системе получить статистику по временным файлам.

Для эксперимента откроем два сеанса и в первом запустим уже известный запрос, создающий временные файлы, однако модифицируем его, добавив еще одно соединение для увеличения объема выполняемой работы:

```
# SELECT a.*, b.*, c.* FROM pg_class a, pg_class b, pg_class c ORDER BY random();
```

Запрос, выводящий нужную статистику, занимает несколько десятков строк, поэтому я не буду его приводить, однако его можно найти в каталоге scripts в тестовом окружении[1]. Во втором сеансе будем периодически запускать SQL-скрипт с запросом, который покажет не только увеличение размеров временного файла, но и появление новых сегментов:

```
# \i /var/lib/postgresql/scripts/active_temp_files.sql
  pid  |    query_age    |            filename             |  size   | last_modification |
-------+-----------------+---------------------------------+---------+-------------------+-------------------------...
 68403 | 00:02:06.101641 | base/pgsql_tmp/pgsql_tmp68403.0 | 1024 MB | 00:01:01.995852   | SELECT a.*, b.*, c.* FROM...
 68403 | 00:02:06.101641 | base/pgsql_tmp/pgsql_tmp68403.1 | 1024 MB | 00:00:10.995852   | SELECT a.*, b.*, c.* FROM...
 68403 | 00:02:06.101641 | base/pgsql_tmp/pgsql_tmp68403.2 | 455 MB  | 00:00:01.004148   | SELECT a.*, b.*, c.* FROM...
```

Дополнительно можно открыть еще один сеанс и понаблюдать за тем, в какой момент обновляется статистика в представлениях pg_stat_database и pg_stat_statements.

5.7. Ввод-вывод фоновых процессов

Среди процессов СУБД помимо клиентских процессов есть еще и фоновые службы, которые также осуществляют ввод-вывод. В этой главе рассмотрим процесс фоновой записи background writer и процесс checkpointer, выполняющий контрольные точки. Оба процесса берут на себя задачу записи изменений из буферного кеша в основное хранилище. Поскольку объем данных и количество необходимых операций ввода-вывода может быть довольно большим, процессы записывают данные не в момент внесения изменений, а асинхронно в фоновом режиме.

[1] github.com/lesovsky/postgresql-monitoring-book/blob/main/playground/scripts/active_temp_files.sql

Задача процесса фоновой записи заключается в записи грязных буферов из буферного кеша в основное хранилище. Процесс выполняет свою работу, чередуя ее с паузами. Перед началом очередной итерации выполняется оценка количества обращений к буферам, которое произошло за предыдущую итерацию, — чем больше обращений, тем больше буферов будет списано в этой итерации (см. *bgwriter_lru_multiplier*), но не более, чем указано в *bgwriter_lru_maxpages*[1]. Среди грязных буферов выбираются именно те, что будут вытеснены в ближайшем будущем, — для этого фактически повторяется алгоритм вытеснения, но без уменьшения счетчика использования. То есть процесс фоновой записи работает на упреждение и предотвращает попытки вытеснения грязных страниц со стороны клиентских процессов.

Процесс checkpointer выполняет контрольную точку, надежно записывая измененные данные в основное хранилище. В момент завершения контрольной точки гарантируется, что все изменения, произошедшие до начала контрольной точки, синхронизированы с диском. Контрольные точки позволяют обслуживать WAL-журнал, удаляя старые сегменты и освобождая место под новые. В процессе выполнения контрольной точки процесс сканирует буферный кеш, отмечая все грязные буферы как требующие синхронизации, а затем постепенно записывает их содержимое в основное хранилище. Выполнение этой работы тоже подразумевает ввод-вывод, объем которого напрямую зависит от количества грязных буферов. Исчерпывающую информацию о работе процесса контрольной точки можно получить из документации[2] и книг[3].

Представление pg_stat_bgwriter. Для отслеживания работы процессов фоновой записи и контрольной точки можно использовать представление pg_stat_bgwriter. Исторически обе задачи выполнялись одним процессом, но в версии 9.2 работа, связанная с контрольными точками, была вынесена в отдельный процесс, а статистика так и осталась в общем представлении. Представление содержит всего одну строку:

```
# TABLE pg_stat_bgwriter;
-[ RECORD 1 ]---------+------------------------------
checkpoints_timed     | 316
checkpoints_req       | 2
checkpoint_write_time | 84919068
checkpoint_sync_time  | 58509
buffers_checkpoint    | 1254815
buffers_clean         | 1801423
maxwritten_clean      | 0
buffers_backend       | 443469
buffers_backend_fsync | 0
buffers_alloc         | 2646195
stats_reset           | 2022-11-04 05:01:49.427562+00
```

Статистика относится не только к процессам фоновой записи и контрольной точки, но и к *фоновой записи* в целом. Давайте рассмотрим представление более подробно.

[1] postgrespro.ru/docs/postgresql/current/runtime-config-resource#RUNTIME-CONFIG-RESOURCE-BACKGROUND-WRITER
[2] postgrespro.ru/docs/postgresql/current/wal-configuration
[3] www.interdb.jp/pg/pgsql09.html#_9.7.

Причины наступления контрольных точек. Первые два поля описывают количество выполненных контрольных точек:

- `checkpoints_timed` — количество контрольных точек, выполненных по расписанию;
- `checkpoints_req` — количество контрольных точек, выполненных по необходимости.

Как видно, контрольные точки могут запускаться либо по расписанию, через интервал времени, указанный в параметре *checkpoint_timeout*, либо по необходимости, что означает внеплановый запуск при превышении объемом записи ограничения, установленного в параметре *max_wal_size*. Также к последней категории относятся контрольные точки, запущенные администратором с помощью команды `CHECKPOINT`, и контрольные точки, выполняемые при выключении сервера СУБД для синхронизации буферного кеша с основным хранилищем.

При настройке контрольных точек есть две стратегии. Первая стратегия полагается на выполнение контрольных точек по расписанию, когда выбирается относительно длинный интервал времени *checkpoint_timeout* и выполнение контрольной точки растягивается на этот интервал. Эта стратегия применима на оборудовании с низкой производительностью, и ее смысл состоит в распределение нагрузки от контрольных точек таким образом, чтобы ее выполнение меньше влияло на общую производительность. При этом появление контрольных точек по необходимости указывает на то, что нагрузка увеличилась и в WAL-журнал стало записываться больше данных еще до истечения настроенного интервала. У этой стратегии есть недостаток: если увеличивать интервал *checkpoint_timeout* и предел *max_wal_size*, то из-за больших объемов записи в WAL-журнал и редких контрольных точек восстановление после потенциальной аварии может занять много времени (так как при восстановлении после сбоя все изменения, записанные в WAL, нужно воспроизвести).

Вторая стратегия полагается на выполнение контрольных точек по необходимости на основе значения *max_wal_size* — как только объем записи в WAL-журнал достигнет указанного значения, будет запущено выполнение контрольной точки. Этот способ хорошо подходит для регулировки времени восстановления после сбоя: оно более предсказуемо, поскольку известно, какой объем WAL нужно воспроизвести, и значение *max_wal_size* можно подстроить под характеристики производительности оборудования. Но и у этой стратегии есть недостаток. Значение *max_wal_size* служит только ориентиром, а в реальности в WAL-журнал может быть записано больше изменений, что тоже отразится на времени восстановления.

Особенностью записи в WAL-журнал является то, что после выполнения контрольной точки первое изменение любого буфера в кеше приводит к записи всей страницы (full page write, FPW) из этого буфера в журнал (см. *full_page_writes*). Все последующие изменения в этом буфере будут журналироваться отдельно. Таким образом, при частом выполнении контрольных точек (неважно, по интервалу или по необходимости) будет возникать дополнительная нагрузка от записи полных образов страниц, и объем этой нагрузки зависит от размера буферного кеша и объема происходящих в нем изменений (операции `INSERT`, `UPDATE`, `DELETE`).

Можно отметить, что на практике большее распространение получила первая стратегия настройки, именно потому, что контрольная точка воспринимается как тяжелая операция и желательно проводить ее аккуратно с наименьшим влиянием на производительность остальных

процессов и выполняемых запросов. При такой стратегии увеличение `checkpoints_req` расценивается негативно и является поводом для оценки увеличения реальной нагрузки и ее влияния на общую производительность, для пересмотра конфигурации и последующего поиска причин возрастания нагрузки. В современных условиях, когда производительность дисковых подсистем сильно выросла благодаря SSD- и NVMe-носителям, актуальность такого подхода уже ставится под сомнение. В любом случае при выборе стратегии следует ориентироваться на метрики производительности подсистемы хранения и пропускной способности СУБД при выполнении запросов.

В мониторинге можно использовать график на основе этих метрик (рис. 5.7) и отслеживать пики появления «нежелательных» контрольных точек (в зависимости от выбранной стратегии настройки).

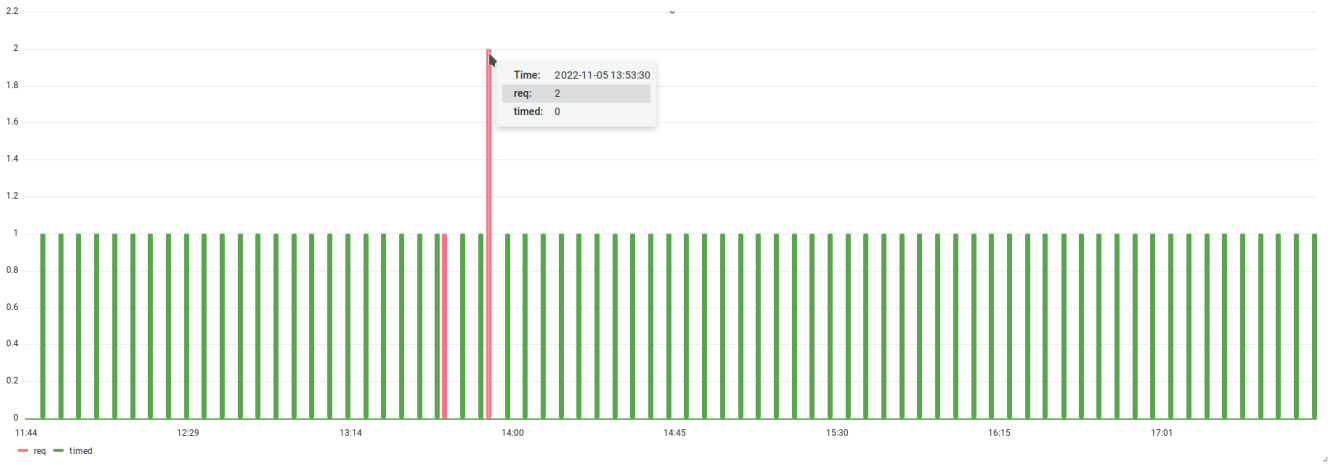


Рис. 5.7. Выполнение контрольных точек

График показывает, что бóльшая часть контрольных точек выполняются по расписанию, значит, объем записи в WAL-журнал укладывается в предел *max_wal_size*. Есть и контрольные точки, выполненные по необходимости: если бы это была производственная среда, это мог бы быть пик записи в WAL, но в тестовом окружении нагрузка стабильная, и, как следствие, объем записи в WAL тоже ровный, без резких всплесков. Для демонстрации эти контрольные точки были запущены с помощью команды `CHECKPOINT`.

Продолжительность выполнения контрольных точек. Выполнение контрольной точки можно условно разделить на два этапа: запись содержимого грязных буферов на диск и вызов синхронизации с основным хранилищем (см. *fsync*). Запись на первом этапе не дает надежной гарантии сохранности данных, поэтому необходим второй этап с явной синхронизацией (однако и здесь могут быть проблемы[1]). Разделение на этапы условное, поскольку выполнение контрольной точки устроено так, что запросы на синхронизацию выстраиваются в очередь, и, если очередь заполнена, запись приостанавливается, выполняется синхронизация уже записанных

[1] wiki.postgresql.org/wiki/Fsync_Errors

блоков, после чего запись возобновляется. Следующие поля как раз показывают время, затраченное на этих этапах (оба значения — в миллисекундах):

- `checkpoint_write_time` — время, затраченное на запись страниц;
- `checkpoint_sync_time` — время, затраченное на синхронизацию.

Запись страниц на первом этапе занимает бóльшую часть времени контрольной точки, а синхронизация является завершающим этапом и, как правило, выполняется существенно быстрее. Однако скорость синхронизации зависит от производительности дисковой подсистемы: на медленных носителях на этапе синхронизации могут образоваться очереди запросов ввода-вывода, что приведет к увеличению задержек и снижению производительности. Следовательно, медленная синхронизация — признак недостаточной производительности дисковой подсистемы. По значениям этих полей мы можем отслеживать время, требуемое на синхронизацию, и, если синхронизация становится долгой, стоит оценить влияние контрольных точек на производительность запросов (изменение времени выполнения запросов в момент синхронизации на контрольных точках).

Для отслеживания времени выполнения можно использовать график на основе метрики `postgres_checkpoints_seconds_total` (рис. 5.8):

Рис. 5.8. Продолжительность выполнения контрольных точек

На этом графике видно, что время выполнения контрольных точек ровное, но в интервале есть две контрольные точки, где синхронизация длится около 10 секунд. Это выделяется на фоне других контрольных точек и является достаточным поводом проявить интерес и разобраться, почему в редких случаях синхронизация выполняется так долго.

Запись буферов. Следующие поля позволяют анализировать объем данных, записанных как фоновыми, так и клиентскими процессами:

- `buffers_checkpoint` — количество буферов, записанных процессом контрольной точки;
- `buffers_clean` — количество буферов, записанных процессом фоновой записи;

- buffers_backend — количество буферов, записанных клиентскими процессами;
- buffers_alloc — количество буферов, выделенных для размещения страниц данных;
- maxwritten_clean — количество приостановок сброса грязных страниц процессом фоновой записи по причине достижения предела, указанного в *bgwriter_lru_maxpages*.

Процесс фоновой записи можно рассматривать как помощника процесса контрольной точки. Постоянно записывая грязные страницы, он уменьшает объем работы, необходимый для выполнения контрольной точки. Первые две метрики показывают объем работы, проделанный двумя этими фоновыми процессами. Третья метрика показывает количество буферов, записанных клиентскими процессами, включая и рабочие процессы автоочистки. Напомню, что, если процесс не смог найти нужный блок с данными в кеше, он вынужден прочитать этот блок с диска. Для этого обычно требуется найти буфер (buffers_alloc) и вытеснить из него имеющийся блок. Если найденный буфер окажется грязным, то дополнительно придется записать содержимое буфера на диск. Именно эту работу и проделывает заранее процесс фоновой записи.

Конфигурация СУБД подразумевает несколько араметров[1], которые регулируют работу процесса фоновой записи и, соответственно, влияют на объем выполняемой работы и производимой им нагрузки (в пределах одной итерации рабочего цикла). С помощью метрики maxwritten_clean можно оценивать, как часто процесс фоновой записи останавливался по причине достижения предела, указанного в *bgwriter_lru_maxpages*. Таким образом, рассматривая значения метрик buffers_backend и maxwritten_clean, можно оценивать эффективность процесса фоновой записи: чем меньше оба значения, тем лучше. При конфигурировании следует помнить, что работа процесса фоновой записи предполагает нагрузку на подсистему ввода-вывода. Со слишком агрессивными настройками процесс будет выполнять избыточный ввод-вывод, что может негативно сказаться на производительности других процессов.

Перечисленные метрики можно вывести в график (рис. 5.9). На нем видно, что процессы фоновой записи и контрольной точки создают более или менее стабильную и ровную нагрузку, а вот от клиентских процессов есть два пика запросов на запись. Стоит разобраться, справляется ли процесс фоновой записи (см. maxwritten_clean) или это был спонтанный всплеск в рабочей нагрузке.

Осталось рассмотреть еще два поля:

- buffers_backend_fsync — количество раз, когда клиентские процессы были вынуждены выполнить синхронизацию самостоятельно. Операции синхронизации обычно делегируется процессу контрольной точки, но могут быть выполнены и клиентскими процессами, что считается плохим признаком. За всю практику эксплуатацию СУБД мне ни разу не приходилось сталкиваться с тем, чтобы buffers_backend_fsync был больше нуля;
- stats_reset — уже хорошо известное по другим представлениям поле с отметкой времени сброса статистики. Обычно к этому полю не приходится обращаться, если статистика по фоновой записи и контрольным точкам складывается в систему мониторинга, но если

[1] postgrespro.ru/docs/postgresql/current/runtime-config-resource#RUNTIME-CONFIG-RESOURCE-BACKGROUND-WRITER

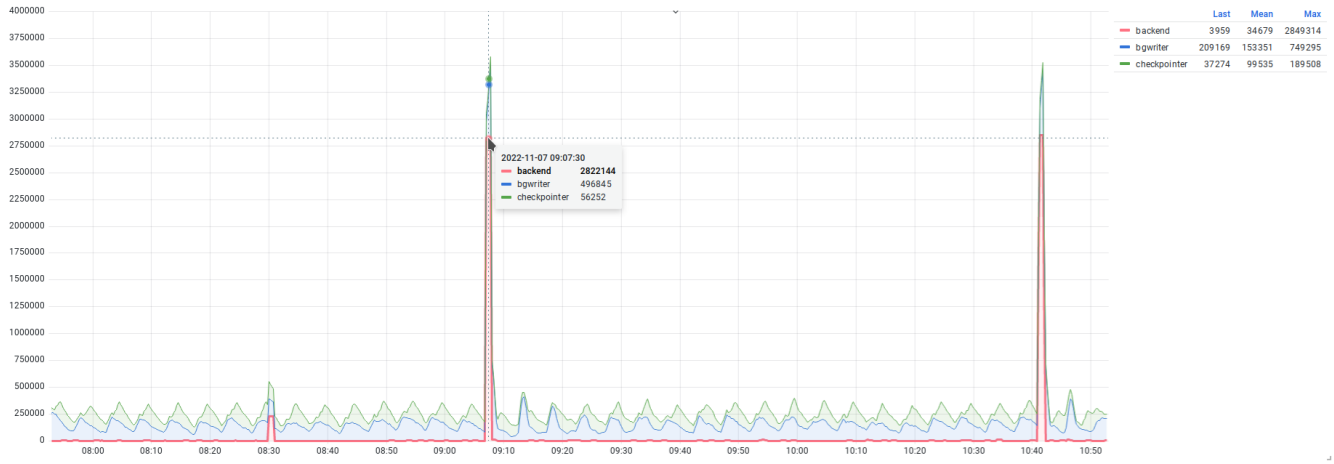


Рис. 5.9. Количество буферов, записанных разными процессами

статистика этого представления используется для построения сводных отчетов, то важно смотреть, чтобы статистика была накоплена за относительно длинный период, например 2–4 недели. Статистика за более короткие периоды может быть не очень показательной, а за слишком большие — сильно усредненной и не отражающей какие-либо всплески и возможные аномалии.

Завершая главу, стоит отметить, что некоторые другие представления, такие как `pg_stat_wal` или `pg_stat_replication_slots`, также содержат статистику ввода-вывода. Они в меньшей степени относятся к вводу-выводу, связанному с пользовательскими данными, однако могут служить источником информации о вводе-выводе на уровне экземпляра СУБД. Эти и другие представления будут рассмотрены в следующих главах, посвященных WAL-журналу и репликации.

Резюме

- Представление `pg_buffercache` позволяет оценить использование буферного кеша с разных сторон.
- Для анализа структуры общей памяти используется представление `pg_shmem_allocations`.
- Для анализа структуры памяти клиентских процессов используются представление `pg_backend_memory_contexts` и функция `pg_log_backend_memory_contexts`.
- Для оценки использования SLRU-кешей используется представление `pg_stat_slru`.
- Для оценки общего ввода-вывода можно использовать `pg_stat_database`.
- Детальную оценку ввода-вывода по конкретным объектам БД можно получить в соответствующих представлениях `pg_statio_*`.

- Для оценки ввода-вывода при выполнении запросов можно использовать представление `pg_stat_statements`.
- Для общей оценки временных файлов можно использовать `pg_stat_database`.
- Для оценки ввода-вывода, связанного с временными файлами, при выполнении запросов можно использовать `pg_stat_statements`.
- Отслеживать временные файлы можно с помощью журналов сообщений СУБД.
- Важно отслеживать активные временные файлы.
- Фоновый процесс записи и процесс контрольной точки являются источниками ввода-вывода, причем последний может оказывать существенное влияние на общую производительность.
- Для оценки ввода-вывода фоновой записи и процесса контрольной точки следует использовать представление `pg_stat_bgwriter`.

Глава 6
Журнал упреждающей записи

В этой главе мы рассмотрим:

- что такое журнал упреждающей записи;
- устройство журнала упреждающей записи в PostgreSQL;
- как отслеживать активность, связанную с журналом;
- представление `pg_stat_wal`;
- отслеживание записи в журнал запросами;
- архивирование сегментов журнала;
- представление `pg_stat_archiver`;
- способы отслеживания проблем при архивировании журнала;
- отслеживание очереди архивирования журнала.

В этой главе мы рассмотрим *журнал упреждающей записи*, он же *журнал предзаписи* (Write-Ahead Log), который в том или ином виде присутствует в большинстве СУБД. Журнал предзаписи в общем виде представляет собой историю всех изменений данных, происходящих в СУБД. В случае аварий такая история позволяет воспроизвести всю последовательность изменений до момента аварии и восстановить актуальное и согласованное состояние СУБД. В этой главе мы кратко рассмотрим устройство журнала (здесь и далее в этой главе под термином «журнал» будет подразумеваться именно WAL-журнал, а не журнал сообщений СУБД) и более подробно рассмотрим инструменты и способы отслеживания событий, связанных с журналом.

6.1. Write-Ahead Log — журнал упреждающей записи

Для гарантий надежности все изменения данных в СУБД должны быть записаны в надежное хранилище (обычно на диск). Нельзя допускать искажений, повреждений и тем более потери данных даже в случае сбоев в системе. Для реализации этих требований практически все СУБД используют журнал предзаписи. В PostgreSQL журнал представляет собой историю изменений данных, и СУБД в случае аварии использует журнал для воспроизведения этих изменений и достижения последней согласованной точки до момента аварии.

Экземпляр СУБД состоит из множества клиентских и фоновых процессов, которые работают с данными, размещенными в общей памяти, — к таким данным относятся и буферный

кеш с пользовательскими данными, и служебные структуры данных. Приложения отправляют запросы, а СУБД выполняет их. В случае запросов на чтение СУБД обращается к данным и возвращает необходимый набор строк; в случае запросов на изменение данных (или схемы) СУБД вносит соответствующие изменения. Все изменения происходят в общей памяти, затем в отложенном режиме процесс фоновой записи или контрольной точки синхронизирует изменения в кеше с основным хранилищем. Общая память СУБД является оперативной, и ее содержимое не защищено от системных сбоев вроде отказов питания. Также существует класс программных ошибок, при которых область общей памяти может быть повреждена и потребуется ее полная очистка и пересоздание — как правило, это ошибки сегментации (segmentation fault). К подобным последствиям приводит и аварийное завершение операционной системой каких-либо процессов СУБД. Обычно такие аварийные ситуации происходят внезапно, нельзя подготовиться к ним заранее и ждать их наступления в известный момент. Получается, что СУБД нужен инструмент, позволяющий восстановить те изменения данных, которые произошли в общей памяти, но из-за аварии не были записаны в основное хранилище.

Однако СУБД может эксплуатироваться очень долго, и хранить абсолютно всю историю изменений невозможно (либо это требует значительных или даже неадекватных экономических вложений). Для повторного использования файлов журнала и поддержания его в разумных объемах применяются контрольные точки: все изменения, предшествующие такой точке, гарантированно записаны в основное хранилище данных, которое считается надежным. Отметки об успешном завершении контрольных точек также записываются в журнал, после чего часть журнала до контрольной точки может быть удалена. В случае аварии остается воспроизвести только те изменения, которые были записаны в журнал после контрольной точки, и прийти к состоянию до момента аварии.

На практике журнал представляет собой набор файлов (сегментов) внутри подкаталога pg_wal в основном каталоге данных:

```
# ls -l /var/lib/postgresql/data/pg_wal/
total 196616
-rw-------    1 postgres postgres       341 Nov  4 05:03 000000010000000000000011.0001F7A0.backup
-rw-------    1 postgres postgres  16777216 Nov 18 05:26 0000000100000004B000000C2
-rw-------    1 postgres postgres  16777216 Nov 18 05:27 0000000100000004B000000C3
-rw-------    1 postgres postgres  16777216 Nov 18 05:28 0000000100000004B000000C4
-rw-------    1 postgres postgres  16777216 Nov 18 05:28 0000000100000004B000000C5
-rw-------    1 postgres postgres  16777216 Nov 18 05:29 0000000100000004B000000C6
-rw-------    1 postgres postgres  16777216 Nov 18 05:30 0000000100000004B000000C7
-rw-------    1 postgres postgres  16777216 Nov 18 05:31 0000000100000004B000000C8
-rw-------    1 postgres postgres  16777216 Nov 18 05:32 0000000100000004B000000C9
-rw-------    1 postgres postgres  16777216 Nov 18 05:24 0000000100000004B000000CA
-rw-------    1 postgres postgres  16777216 Nov 18 05:20 0000000100000004B000000CB
-rw-------    1 postgres postgres  16777216 Nov 18 05:25 0000000100000004B000000CC
-rw-------    1 postgres postgres  16777216 Nov 18 05:23 0000000100000004B000000CD
drwx------    2 postgres postgres      4096 Nov 18 05:31 archive_status
```

Все сегменты журнала имеют имя из 24 цифр в шестнадцатеричном формате без расширения. Имя состоит из трех октетов (на примере сегмента `0000000100000004B000000CD`):

1. 00000001 — идентификатор линии времени (timeline id). Линии времени используется механизмом восстановления на точку во времени (Point-in-Time Recovery, PITR)[1, 2].
2. 0000004B — идентификатор верхнего диапазона сегментов со значениями от 00000000 до FFFFFFFF.
3. 000000CD — идентификатор нижнего диапазона сегментов со значениями от 00000000, а последний номер зависит от размера сегмента, заданного при инициализации основного каталога.

Каждый сегмент имеет фиксированный размер (см. *wal_segment_size*), по умолчанию 16 МБ. Внутри каждый сегмент поделен на блоки по 8 КБ (см. *wal_block_size*). Размер сегмента можно менять при инициализации кластера (параметр --wal-segsize). Каждый блок содержит WAL-записи (records) в непечатаемом двоичном формате, описывающие отдельные изменения данных. Одна такая запись содержит достаточно информации для воспроизведения (повторного применения) одного конкретного изменения в случае сбоя системы. Записи могут иметь разные типы и относиться к изменениям самых разных объектов. Для анализа содержимого журнала можно использовать утилиту pg_waldump[3] или расширение pg_walinspect[4]. Оба инструмента предоставляют одинаковую функциональность, но первый предназначен для работы только в среде терминала, а расширение предоставляет SQL-интерфейс с более богатыми возможностями для анализа результатов:

```
# SELECT * FROM pg_get_wal_records_info('4B/DF000000','4B/E0000000') LIMIT 14;
  start_lsn  |   end_lsn   |  prev_lsn   |   xid    | resource_manager | record_type | record_length | main_data_length | fpi_length |          ...
-------------+-------------+-------------+----------+------------------+-------------+---------------+------------------+------------+----------...
 4B/DF000350 | 4B/DF0003F8 | 4B/DEFFE370 | 43601290 | Heap             | HOT_UPDATE  |           163 |               14 |          0 | off 88 xma...
 4B/DF0003F8 | 4B/DF000448 | 4B/DF000350 | 43601290 | Heap             | HOT_UPDATE  |            78 |               14 |          0 | off 22 xma...
 4B/DF000448 | 4B/DF000498 | 4B/DF0003F8 | 43601290 | Heap             | HOT_UPDATE  |            74 |               14 |          0 | off 111 xm...
 4B/DF000498 | 4B/DF0004E8 | 4B/DF000448 | 43601290 | Heap             | INSERT      |            79 |                3 |          0 | off 43 fla...
 4B/DF0004E8 | 4B/DF000518 | 4B/DF000498 | 43601290 | Transaction      | COMMIT      |            46 |               20 |          0 | 2022-11-18...
 4B/DF000518 | 4B/DF0024E8 | 4B/DF0004E8 |        0 | Heap2            | PRUNE       |          8119 |                8 |       8060 | latestRemo...
 4B/DF0024E8 | 4B/DF002590 | 4B/DF000518 | 43601291 | Heap             | HOT_UPDATE  |           163 |               14 |          0 | off 104 xm...
 4B/DF002590 | 4B/DF0025E0 | 4B/DF0024E8 | 43601291 | Heap             | HOT_UPDATE  |            78 |               14 |          0 | off 71 xma...
 4B/DF0025E0 | 4B/DF002630 | 4B/DF002590 | 43601291 | Heap             | HOT_UPDATE  |            74 |               14 |          0 | off 133 xm...
 4B/DF002630 | 4B/DF002680 | 4B/DF0025E0 | 43601291 | Heap             | INSERT      |            79 |                3 |          0 | off 27 fla...
 4B/DF002680 | 4B/DF0026B0 | 4B/DF002630 | 43601291 | Transaction      | COMMIT      |            46 |               20 |          0 | 2022-11-18...
 4B/DF0026B0 | 4B/DF004680 | 4B/DF002680 |        0 | Heap2            | PRUNE       |          8119 |                8 |       8060 | latestRemo...
 4B/DF004680 | 4B/DF004728 | 4B/DF0026B0 | 43601292 | Heap             | HOT_UPDATE  |           163 |               14 |          0 | off 78 xma...
 4B/DF004728 | 4B/DF004778 | 4B/DF004680 | 43601292 | Heap             | HOT_UPDATE  |            78 |               14 |          0 | off 179 xm...
```

Каждая отдельная запись имеет уникальный идентификатор LSN (log sequence number). Идентификатор можно использовать как *позицию* в WAL-журнале, по которой можно определить местоположение записи в журнале. На примере записи 4B/DF0026B0:

- 4B — идентификатор верхнего диапазона (второй октет имени файла сегмента);
- DF — идентификатор нижнего диапазона (третий октет имени файла сегмента);
- 0026B0 — смещение внутри файла сегмента.

[1] www.interdb.jp/pg/pgsql10.html#_10.3.1.
[2] postgrespro.ru/docs/postgresql/current/continuous-archiving
[3] postgrespro.ru/docs/postgresql/current/pgwaldump
[4] postgrespro.ru/docs/postgresql/current/pgwalinspect

Определить сегмент, в котором находится конкретная запись, можно с помощью функции pg_walfile_name, передав ей LSN:

```
# SELECT pg_walfile_name('4B/DF008B78');
     pg_walfile_name
--------------------------
 00000001000000040000000DF
```

При активной эксплуатации СУБД и постоянном изменении данных в журнал вставляются новые записи и текущая позиция (выраженная в LSN) постоянно смещается вперед. После вставки добавленные записи следует надежно записать и в основное хранилище. Получить текущую позицию записи журнала можно с помощью нескольких функций:

- pg_current_wal_insert_lsn — позиция последней вставленной в журнал записи;
- pg_current_wal_lsn — позиция последней сохраненной на диск записи;
- pg_current_wal_flush_lsn — позиция последней сохраненной и синхронизированной с диском записи.

При полном заполнении сегмента вставка начинается в следующий по порядку сегмент.

Объем WAL-журнала ограничен параметрами конфигурации и при нормальной эксплуатации количество сегментов колеблется в определенном диапазоне. При пиковых нагрузках, особенно связанных с изменением данных, могут создаваться дополнительные сегменты, отчего размер журнала может увеличиваться. При возвращении нагрузки к нормальным значениям и после выполнения контрольной точки размер журнала скорректируется — излишние сегменты будут либо удалены, либо переименованы для использования в будущем; таким образом объем журнала уменьшится до размеров, заданных в конфигурации СУБД. В следующем листинге по отметкам времени можно заметить, что запись идет в сегмент 00000001000000040000000FC, а сегменты с бóльшими номерами, но меньшим временем зарезервированы под использование в ближайшем будущем — как только активный сегмент будет заполнен, СУБД переключится на запись в следующий сегмент.

```
# ls -l /var/lib/postgresql/data/pg_wal/
total 196616
-rw-------    1 postgres postgres       341 Nov  4 05:03 000000010000000000000011.0001F7A0.backup
-rw-------    1 postgres postgres  16777216 Nov 18 06:21 00000001000000040000000F5
-rw-------    1 postgres postgres  16777216 Nov 18 06:22 00000001000000040000000F6
-rw-------    1 postgres postgres  16777216 Nov 18 06:23 00000001000000040000000F7
-rw-------    1 postgres postgres  16777216 Nov 18 06:24 00000001000000040000000F8
-rw-------    1 postgres postgres  16777216 Nov 18 06:25 00000001000000040000000F9
-rw-------    1 postgres postgres  16777216 Nov 18 06:26 00000001000000040000000FA
-rw-------    1 postgres postgres  16777216 Nov 18 06:27 00000001000000040000000FB
-rw-------    1 postgres postgres  16777216 Nov 18 06:28 00000001000000040000000FC <<< активный сегмент
-rw-------    1 postgres postgres  16777216 Nov 18 06:17 00000001000000040000000FD
-rw-------    1 postgres postgres  16777216 Nov 18 06:16 00000001000000040000000FE
```

```
-rw-------    1 postgres postgres  16777216 Nov 18 06:19 000000010000004B000000FF
-rw-------    1 postgres postgres  16777216 Nov 18 06:18 000000010000004C00000000
drwx------    2 postgres postgres      4096 Nov 18 06:27 archive_status
```

Запись в журнал может осуществляться как клиентом, так и фоновым процессом walwriter. По умолчанию все изменения в журнал пишутся клиентским процессом. Это может происходить при подтверждении транзакции командой `COMMIT` или при заполнении WAL-буфера (см. *wal_buffers*). Такая запись называется синхронной, и, перед тем как отправить следующую команду или запрос, клиент вынужден дождаться завершения записи в журнал. Настройки СУБД позволяют переопределить поведение и использовать асинхронное подтверждение транзакций (см. *synchronous_commit*). В таком режиме запись в журнал делегируется процессу walwriter, который работает в фоновом режиме. Перекладывая задачу записи в журнал на фоновый процесс, клиентские процессы могут не дожидаться завершения записи и сразу отправлять серверу следующую команду. Однако следует помнить, что в таком режиме в случае сбоев есть риск потери последних транзакций, которые еще не были записаны процессом walwriter. Настройка подтверждения транзакций имеет еще несколько режимов, действующих при использовании репликации. Более подробно с ними можно ознакомиться в документации[1]. Можно найти и исчерпывающую информацию о работе WAL-журнала[2, 3].

6.2. Отслеживание активности в журнале

Журнал упреждающей записи является важной частью СУБД, и администратору необходимо иметь представление об активности, связанной с ним. Запись в журнал влияет на продолжительность транзакций, поэтому для хорошей производительности СУБД важно, чтобы запись проходила быстро и без задержек. Может возникнуть и ряд других вопросов: каковы объемы записи в журнал? Сколько времени уходит на запись? Какой объем записи в журнал выполняют те или иные типы запросов?

Для отслеживания активности, связанной с журналом, имеется несколько инструментов:

- pg_waldump и `pg_walinspect` — инструменты для детального анализа журнала, которые больше подходят для отладки и поиска проблем или для образовательных целей;
- `pg_stat_wal` — общая статистика связанной с журналом активности;
- `pg_stat_statements` — это хорошо зарекомендовавшее себя расширение содержит также статистику, связанную с журналом, по отдельным типам запросов;
- `EXPLAIN ANALYZE` — средство для вывода плана запроса, которое показывает дополнительную информацию о записи в журнал при указании параметра `WAL`.

[1] postgrespro.ru/docs/postgresql/current/wal-async-commit
[2] postgrespro.ru/docs/postgresql/current/wal-configuration
[3] www.interdb.jp/pg/pgsql09.html

С точки зрения регулярного мониторинга особенно интересны представления pg_stat_wal и pg_stat_statements, поэтому рассмотрим их более подробно.

Представление pg_stat_wal

Представление pg_stat_wal содержит всего одну строку:

```
# TABLE pg_stat_wal;
-[ RECORD 1 ]----+------------------------------
wal_records      | 322508636
wal_fpi          | 46090986
wal_bytes        | 393680012729
wal_buffers_full | 15545
wal_write        | 48958771
wal_sync         | 48846599
wal_write_time   | 5697276.907
wal_sync_time    | 486414839.104
stats_reset      | 2022-11-04 05:01:49.427562+00
```

Статистика носит накопительный характер и включает следующую информацию:

- wal_records — общее количество записей, отправленных в журнал;
- wal_fpi — общее количество записанных FPI-страниц (full page image);
- wal_bytes — общий объем данных, записанный в журнал, в байтах;
- wal_buffers_full — общее количество раз, когда данные в журнал были записаны из-за заполнения WAL-буфера;
- wal_write — количество операций ввода-вывода для записи данных из WAL-буфера;
- wal_sync — количество запросов на синхронизацию WAL-сегментов с основным хранилищем;
- wal_write_time — время, затраченное на запись содержимого WAL-буфера в основное хранилище, в миллисекундах. Может включать в себя время, затраченное на синхронизацию, если параметр *wal_sync_method* установлен в open_datasync или open_sync;
- wal_sync_time — время, затраченное на синхронизацию сегментов в основное хранилище, в миллисекундах;
- stats_reset — отметка времени сброса статистики.

Статистика по времени отслеживается только при включенном параметре *track_wal_io_timing*. На основе данных представления можно отслеживать активность, связанную с журналом, объемы и продолжительность записи и факты, указывающие на нехватку места в WAL-буфере, что может послужить поводом к пересмотру настройки параметра *wal_buffers*.

На основе метрик по этой статистике можно построить графики общей WAL-активности на уровне экземпляра СУБД.

На рис. 6.1 изображен график с количеством обычных записей и FPI-страниц, попадающих в журнал. Можно отметить, что объем записи в журнал стабильный, без значительных всплесков. Также можно отметить постоянный и пилообразный поток записи FPI-страниц. Продолжительность пиков укладывается в пятиминутный интервал, что соответствует значению *checkpoint_timeout* по умолчанию. Выполнение принудительных контрольных точек визуально никак не сказывается на объеме записи FPI-страниц, и на графике это остается незаметным.

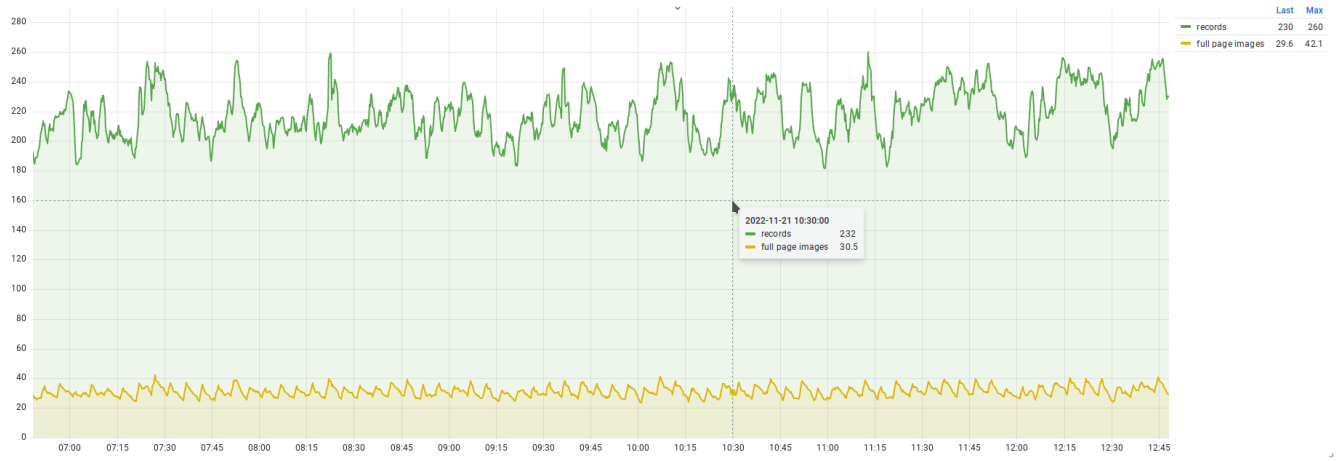


Рис. 6.1. Количество обычных записей и FPI-страниц в журнале

На рис. 6.2 изображен график объема записи в журнал в байтах, и здесь нагрузка также умеренно ровная. Но, если присмотреться, этот график, так же как и предыдущий, напоминает пилу с пятиминутными интервалами. Подобный график можно использовать для визуализации объема записи в журнал.

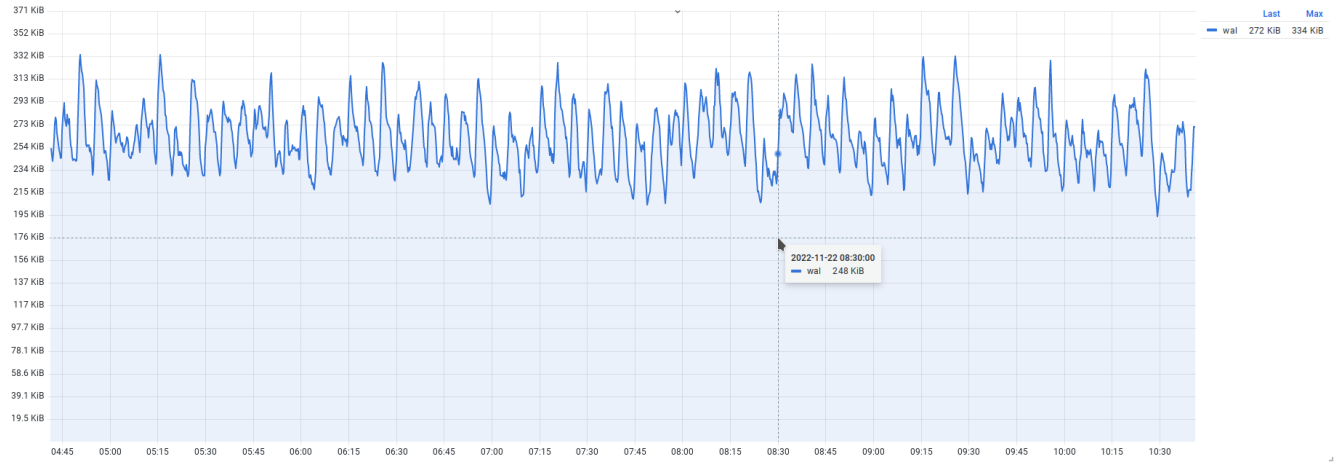


Рис. 6.2. Объем записи в журнал в байтах

График на рис. 6.3 показывает количество операций записи данных из буфера в журнал и последующих синхронизаций сегментов. В левой части графика количество операций записи совпадает с количеством синхронизаций. Это связано с тем, что запись и последующая синхронизация происходят при завершении каждой транзакции. Рядом показано количество подтвержденных транзакций (синяя линия commits), практически совпадающее с остальными метриками. Дальше, в правой части графика, при неизменном потоке транзакций количество синхронизаций сегментов сильно уменьшилось и затем через какое-то время вернулось на прежний уровень. Количество операций записи тоже уменьшилось, хотя и не так сильно. В этот период был включен режим асинхронного подтверждения транзакций (*synchronous_commit* = `off`). В этом режиме задачи по записи в журнал и синхронизации сегментов перекладываются на процесс walwriter, который делает это согласно своим настройкам.

Важной информацией является и время, затраченное на запись и синхронизацию (рис. 6.4).

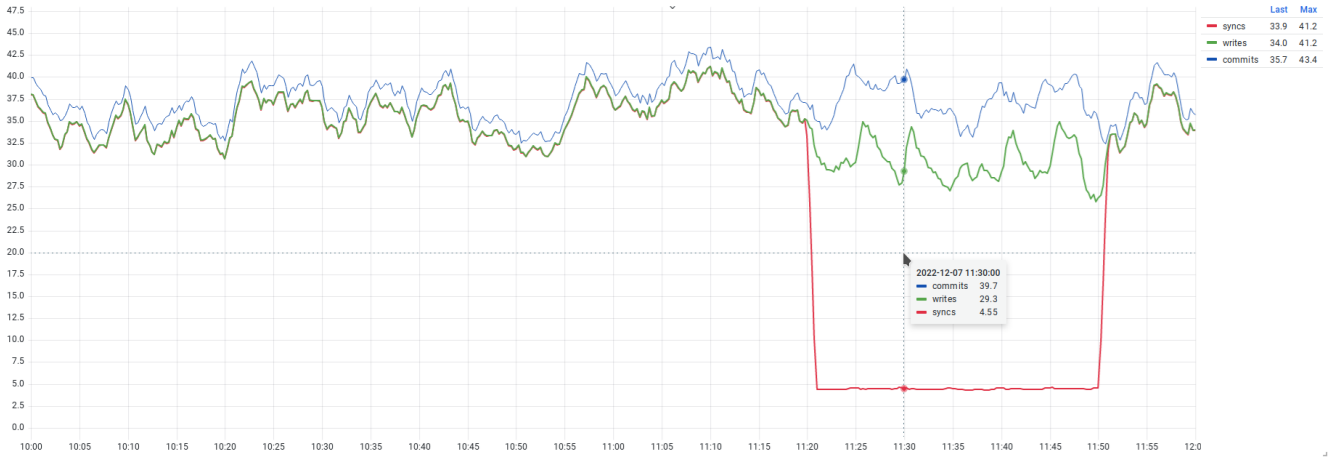


Рис. 6.3. Количество операций записи из WAL-буфера, синхронизаций сегментов и подтвержденных транзакций

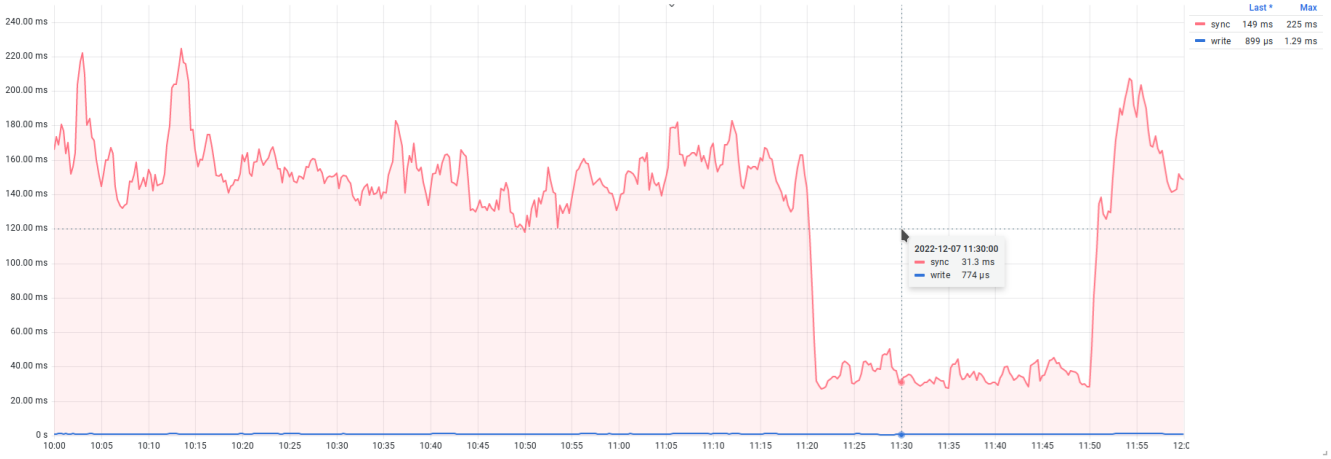


Рис. 6.4. Время, затраченное на запись в журнал и синхронизацию сегментов

На таком графике важно отслеживать пики и выяснять причины всплесков, когда на запись в журнал СУБД тратит больше времени, чем обычно.

Отключение синхронного подтверждения транзакций повлияло также и на время. Когда запись в журнал происходит в асинхронном режиме фоновым процессом walwriter, вместе с уменьшением количества синхронизаций уменьшается и время, затрачиваемое на синхронизацию сегментов. Отключение синхронного подтверждения транзакций часто используется как оптимизация производительности за счет перекладывания обязанности подтверждения транзакций с клиентских процессов на фоновый процесс. Высвободившееся время СУБД может использовать для выполнения еще большего числа запросов, что увеличивает общую пропускную способность СУБД.

Представление pg_stat_statements

Если представление pg_stat_wal содержит общую статистику уровня СУБД, то в представлении pg_stat_statements можно найти аналогичную статистику в контексте отдельных типов запросов:

- wal_records — общее количество записей, отправленных в журнал;
- wal_fpi — общее количество FPI-страниц, записанное в журнал;
- wal_bytes — общий объем данных, записанный в журнал, в байтах.

В перечисленных полях содержится статистика записи для каждого конкретного типа запросов. С ее помощью можно проанализировать состав WAL-журнала и ответить на вопрос о том, какие запросы пишут в журнал больше остальных:

```
# SELECT
    pg_size_pretty(sum(wal_bytes)) AS wal_volume,
    left(query, 64) AS query_trunc
  FROM pg_stat_statements
  GROUP BY query
  ORDER BY sum(wal_bytes) DESC
  LIMIT 10;
 wal_volume |                          query_trunc
------------+---------------------------------------------------------------
 678 GB     | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid =
 10196 MB   | INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
 8219 MB    | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid =
 7805 MB    | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid =
 9313 kB    | SELECT abalance FROM pgbench_accounts WHERE aid = $1
 1394 kB    | truncate pgbench_history
 536 kB     | vacuum pgbench_tellers
 375 kB     | select count(*) from pgbench_branches
 250 kB     | vacuum pgbench_branches
 129 kB     | SELECT current_database() AS database, schemaname AS schema, fun
```

Результат показывает, что наибольший объем записи в журнал генерирует запрос, связанный с обновлением строк в pgbench_accounts (678 ГБ), с большим отрывом опережающий запрос, занявший второе место. Получить данные из мониторинга можно следующим запросом:

```
# topk_avg(5,
  sum by (queryid,query) (
    rate(postgres_statements_wal_bytes_all_total{
        service_id="primary"
    }[1m]) + on(database,user,queryid) group_left(query)
        0 * postgres_statements_query_info{service_id="primary"}
  ), "other")
```

График на рис. 6.5 подтверждает результат запроса к pg_stat_statements: в динамике видно, что запрос на обновление pgbench_accounts пишет в среднем примерно 500 КБ в секунду и по объему записи значительно опережает остальные запросы.

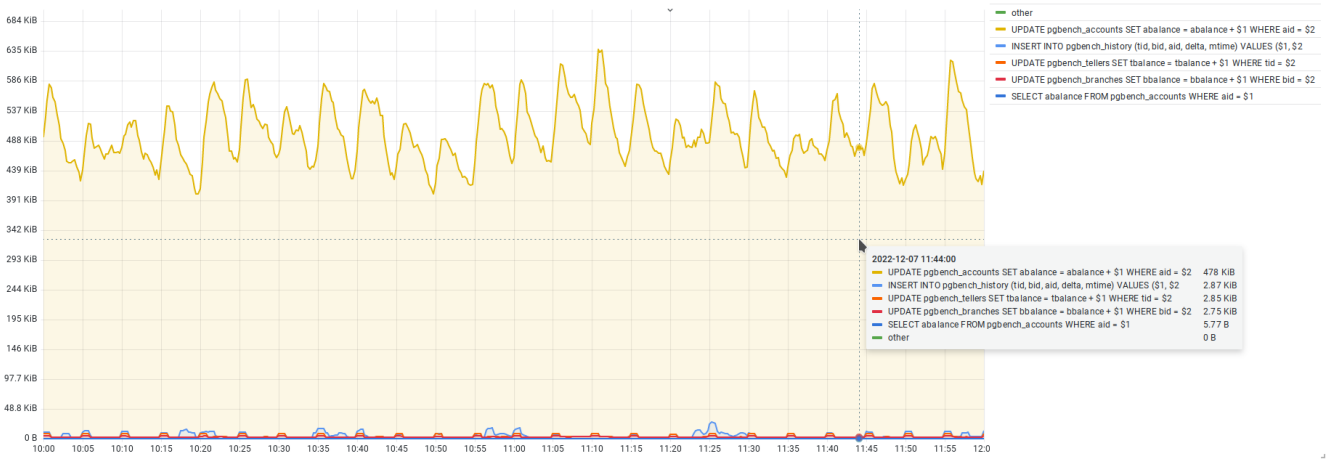


Рис. 6.5. Запросы, пишущие в журнал больше остальных

6.3. Архивирование журнала

Обслуживая журнал, СУБД поддерживает непрерывную историю изменений в целях защиты от потенциальных системных сбоев. Напомню, что после сбоя СУБД может автоматически восстановить согласованное состояние данных с помощью проигрывания изменений. Существование журнала дает также возможность резервного копирования СУБД. Выполнив резервное копирование файлов СУБД вместе с сегментами журнала, впоследствии из созданной резервной копии можно восстановить состояние СУБД до произвольной точки — сначала восстанавливая файлы, затем воспроизводя изменения по журналу. Такой подход связан с некоторыми сложностями, но в целом имеет много положительных сторон.

При таком подходе требуется постоянное архивирование сегментов журнала в отдельное от СУБД хранилище[1]. По умолчанию архивирование сегментов журнала выключено и требует отдельной настройки. При включении архивирования СУБД запустит фоновый процесс archiver. Когда запись в отдельный сегмент журнала завершается и он готов для архивирования, archiver запускает команду архивирования из параметра *archive_command* и ждет ее завершения. Если команда завершилась ошибкой, через некоторое время процесс предпримет еще одну попытку заархивировать сегмент и будет повторять такие попытки до тех пор, пока команда не завершится успехом. Очевидно, что это может привести к ситуации, когда из-за «сломанного» архивирования начнет увеличиваться количество сегментов, и в какой-то момент это может привести к исчерпанию свободного места на диске и аварийной остановке СУБД при попытке записи в журнал. При таких рисках у администратора появляется еще одна обязанность — наблюдение за работой архивирования, отслеживание потенциальных ошибок и их устранение.

Представление pg_stat_archiver

Для отслеживания процесса архивирования есть представление `pg_stat_archiver`, которое содержит всего одну строку со следующими полями:

- `archived_count` — общее количество сегментов, отправленных в архив;
- `last_archived_wal` — имя последнего сегмента, отправленного в архив;
- `last_archived_time` — отметка времени последней отправки сегмента в архив;
- `failed_count` — количество неудачных попыток отправки в архив;
- `last_failed_wal` — имя сегмента, при отправке которого произошла последняя ошибка;
- `last_failed_time` — отметка времени, когда произошла последняя ошибка при отправке в архив;
- `stats_reset` — отметка времени, когда была сброшена статистика.

В тестовом окружении настроено псевдоархивирование: когда наступает время отправки сегмента в архив, вызывается команда /bin/true, которая ничего не делает и лишь возвращает код успешного завершения. Тем не менее это позволяет увидеть статистику архивирования:

```
# TABLE pg_stat_archiver;
-[ RECORD 1 ]------+------------------------------
archived_count     | 45711
last_archived_wal  | 000000010000000B20000009D
last_archived_time | 2022-12-07 06:43:05.091157+00
failed_count       | 21
last_failed_wal    | 000000010000004600000073
last_failed_time   | 2022-11-17 05:52:27.826036+00
stats_reset        | 2022-11-04 05:01:49.427562+00
```

[1] postgrespro.ru/docs/postgresql/current/continuous-archiving

Чтобы понять, как лучше использовать эту статистику, важно понимать классы проблем, которые могут возникнуть при архивировании журнала:

- ошибки передачи данных или отказ в обслуживании на стороне архива — в этом случае процесс архивирования не может передать сегмент в хранилище;
- ошибка при выполнении команды архивирования, например из-за ошибки в скрипте или его отсутствия;
- зависание процесса (по каким-то неизвестным причинам), ответственного за непосредственную отправку в архив;
- зависание процесса archiver — это маловероятный сценарий, однако его нельзя исключать полностью.

В первых двух сценариях необходимо отслеживать количество неудачных попыток архивирования. А если архивирование уже сломалось, важно понимать и то, как долго оно находится в таком состоянии. Для оценки хорошо подходят поля `failed_count` и `last_archived_time`. При возникновении ошибок счетчик `failed_count` начнет увеличиваться (а рост `archived_count` остановится). Факт увеличения количества ошибок более красноречив, чем отсутствие попыток архивирования, ведь система может быть не под нагрузкой и простаивать, а отсутствие успешных попыток архивирования может быть нормальным. То же самое справедливо и для времени: возраст `now() - last_archived_time` будет увеличиваться до тех пор, пока архивирование не будет восстановлено, и тоже является хорошим маркером наличия проблем с архивированием.

На рис. 6.6 изображен график с количеством успешных и неудачных попыток архивирования. Полоса неудачных попыток указывает на то, что в течение 10 минут архивирование не работало, сегменты при этом продолжали копиться. После того как проблема была устранена, наблюдается небольшой всплеск успешной отправки в архив накопившихся сегментов.

С проблемами вроде зависаний дело обстоит сложнее: в этой ситуации команда запустилась, но не может завершиться, а процесс архивирования ожидает ее завершения. В таком случае `archived_count`, `failed_count` и `last_failed_time` остаются без изменений, но при этом возраст `now() - last_archived_time` начинает расти, что указывает на остановку архивирования. На рис. 6.7 изображен график, демонстрирующий ту же самую остановку архивирования, что и на рис. 6.6. В среднем архивирование сегментов выполняется каждые 54 секунды, но был период, когда время увеличилось до 11 минут.

Ориентироваться на возраст последней успешной отправки удобно в активных системах с интенсивной и постоянной записью в журнал, где архивирование сегментов происходит регулярно. Для систем с малой активностью, эпизодической и небольшой записью в журнал, где между архивированием даже двух сегментов может проходить много времени, большой возраст `now() - last_archived_time` может и не указывать на проблему. Конечно, для принудительного архивирования можно использовать тайм-аут (см. *archive_timeout*), по истечении которого выполняется переключение на запись в следующий сегмент журнала, а предыдущий отправляется в архив.

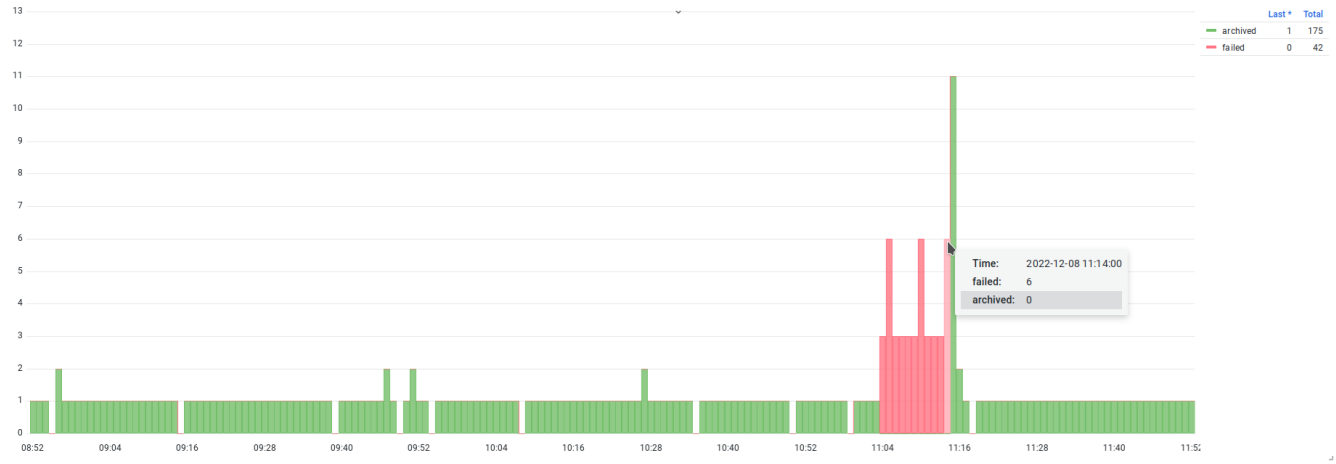


Рис. 6.6. Успешные и неудачные попытки отправки сегментов в архив

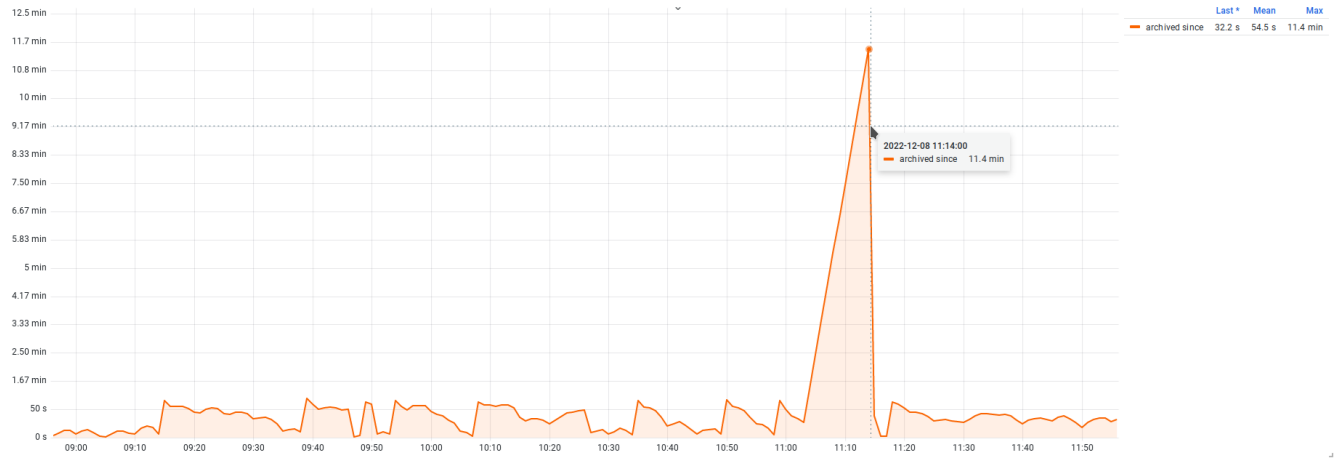


Рис. 6.7. Время с момента последней успешной отправки сегмента в архив

Очередь архивирования

Ранее рассмотренные способы оценки состояния архивирования опираются на конечный результат: был ли сегмент отправлен в архив. Существует еще один подход к отслеживанию работоспособности архивирования. Он учитывает необходимый объем работы и заключается в отслеживании количества сегментов в очереди на отправку. Если очередь пустая — значит, все хорошо, если очередь растет — значит, есть проблемы. Для реализации способа необходимо некоторое знание об устройстве файловой иерархии основного каталога данных.

Внутри каталога pg_wal есть подкаталог archive_status с набором файлов, описывающих статус архивирования конкретных сегментов.

```
# ls -l /var/lib/postgresql/data/pg_wal/archive_status/
-rw-------    1 postgres postgres        0 Nov  4 05:03 000000010000000000000011.0001F7A0.backup.done
-rw-------    1 postgres postgres        0 Dec  8 06:00 0000000100000B7000000D7.done
-rw-------    1 postgres postgres        0 Dec  8 06:01 0000000100000B7000000D8.done
-rw-------    1 postgres postgres        0 Dec  8 06:02 0000000100000B7000000D9.done
-rw-------    1 postgres postgres        0 Dec  8 06:04 0000000100000B7000000DA.done
-rw-------    1 postgres postgres        0 Dec  8 06:05 0000000100000B7000000DB.done
-rw-------    1 postgres postgres        0 Dec  8 06:06 0000000100000B7000000DC.done
-rw-------    1 postgres postgres        0 Dec  8 06:06 0000000100000B7000000DD.done
-rw-------    1 postgres postgres        0 Dec  8 06:08 0000000100000B7000000DE.done
```

В этом каталоге нас интересуют файлы с расширениями ready и done. Статус ready указывает на то, что соответствующий сегмент готов для архивирования, а статус done — на то, что сегмент был успешно отправлен в архив. Таким образом, чтобы получить число сегментов в очереди на отправку, нам нужно лишь подсчитать количество файлов с расширением ready. Размер очереди можно перевести в байты (умножить на размер сегмента, по умолчанию 16 МБ) и получить представление о том, сколько места на диске занимают сегменты, требующие архивирования. Для подсчета потребуется функция `pg_ls_archive_statusdir` либо ее более универсальный аналог `pg_ls_dir`:

```
# SELECT count(*)
  FROM pg_ls_archive_statusdir()
  WHERE name ~'.ready';
 count
-------
     0
```

В тестовом окружении настроено псевдоархивирование командой /bin/true, при вызове которой не происходит реального копирования, однако и такое архивирование отражается в статистике в `pg_stat_archiver`. В целях экспериментов архивирование можно «сломать», указав вместо /bin/true команду /bin/false.

Перед началом эксперимента можно выполнить запрос к статистике и зафиксировать образец того, как выглядит рабочее состояние архивирования:

```
# SELECT *, now() - last_archived_time AS last_archived_age
  FROM pg_stat_archiver;
-[ RECORD 1 ]------+------------------------------
archived_count     | 47058
last_archived_wal  | 0000000100000B7000000E0
last_archived_time | 2022-12-08 06:10:14.254712+00
failed_count       | 0
last_failed_wal    |
last_failed_time   |
stats_reset        | 2022-11-04 05:01:49.427562+00
last_archived_age  | 00:00:09.227001
```

Теперь следует изменить параметр *archive_command*:

```
# ALTER SYSTEM SET archive_command TO '/bin/false';
ALTER SYSTEM
# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
```

Понаблюдаем за тем, что теперь происходит в статистике:

```
# SELECT *, now() - last_archived_time AS last_archived_age
  FROM pg_stat_archiver;
-[ RECORD 1 ]------+------------------------------
archived_count     | 47061
last_archived_wal  | 000000010000000B7000000E3
last_archived_time | 2022-12-08 06:13:06.971114+00
failed_count       | 75
last_failed_wal    | 000000010000000B7000000E4
last_failed_time   | 2022-12-08 06:17:23.115775+00
stats_reset        | 2022-11-04 05:01:49.427562+00
last_archived_age  | 00:04:21.452935
```

Счетчик `archived_count` остановился, а `failed_count` начал расти; время с момента последней успешной попытки архивирования в добавленном поле `last_archived_age` также растет и достигло четырех минут. Проверим, что в каталоге статусов:

```
# SELECT *
  FROM pg_ls_archive_statusdir() ORDER BY modification;
                      name                       | size |      modification
-------------------------------------------------+------+------------------------
 000000010000000000000011.0001F7A0.backup.done |    0 | 2022-11-04 05:03:21+00
 000000010000000B7000000E0.done                 |    0 | 2022-12-08 06:10:14+00
 000000010000000B7000000E1.done                 |    0 | 2022-12-08 06:11:08+00
 000000010000000B7000000E2.done                 |    0 | 2022-12-08 06:12:06+00
 000000010000000B7000000E3.done                 |    0 | 2022-12-08 06:13:06+00
 000000010000000B7000000E4.ready                |    0 | 2022-12-08 06:14:15+00
 000000010000000B7000000E5.ready                |    0 | 2022-12-08 06:15:17+00
 000000010000000B7000000E6.ready                |    0 | 2022-12-08 06:16:19+00
 000000010000000B7000000E7.ready                |    0 | 2022-12-08 06:17:29+00
```

Появились файлы с расширением ready, можно их посчитать и определить количество сегментов в очереди на архивирование:

```
# SELECT count(*)
  FROM pg_ls_archive_statusdir() WHERE name ~'.ready';
 count
-------
     4
```

Соответствующие измерения можно отразить на графике (рис. 6.8):

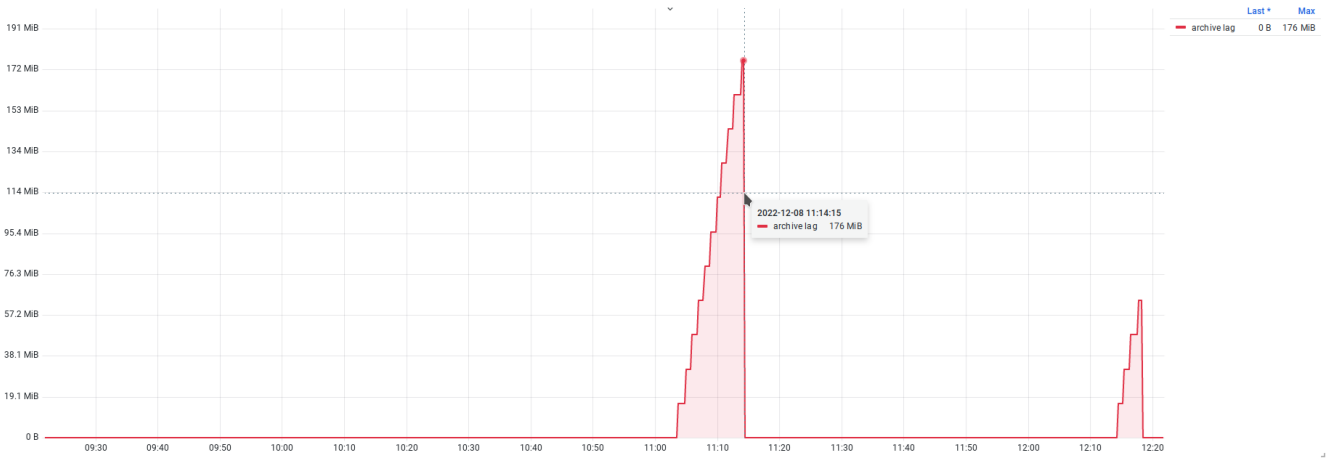


Рис. 6.8. Очередь архивирования в байтах

Способ с размером очереди является универсальным, не зависит от активности и объемов записи в журнал и позволяет отслеживать как ошибки в архивировании, так и случаи, связанные с зависанием.

После экспериментов не забываем откатить конфигурацию:

```
# ALTER SYSTEM RESET archive_command;
# SELECT pg_reload_conf();
```

Резюме

- Журнал упреждающей записи — важный компонент любой СУБД, который хранит историю всех изменений.
- Журнал представляет собой непрерывную последовательность файлов-сегментов.
- Для детального анализа содержимого журнала используются утилита pg_waldump и расширение `pg_walinspect`.
- Для отслеживания общей активности, связанной с журналом, используется представление `pg_stat_wal`.
- `pg_stat_statements` содержит статистику записи в журнал для конкретных запросов.
- Журнал также используется для резервного копирования и репликации.
- Журнал можно архивировать для задач резервного копирования и восстановления.
- Для отслеживания архивирования используется представление `pg_stat_archiver`.
- Отслеживание очереди архивирования — наиболее универсальный способ выявления проблем.

Глава 7

Репликация

В этой главе мы рассмотрим:

- репликацию и ее устройство;
- инструменты отслеживания репликации;
- представления `pg_stat_replication` и `pg_stat_wal_receiver`;
- отставание репликации;
- слоты репликации;
- представление `pg_replication_slots`;
- публикации и подписки;
- представление `pg_stat_replication_slots`;
- представления `pg_stat_subscription` и `pg_stat_subscription_stats`;
- конфликты восстановления и представление `pg_stat_database_conflicts`.

В предыдущей главе мы рассмотрели журнал упреждающей записи, один из важных компонентов СУБД, обеспечивающий гарантии сохранности данных. Кроме того, на использовании WAL-журнала основана репликация. С помощью репликации становится возможным построение комплексных и распределенных конфигураций, которые позволяют обеспечить резервирование, отказо- и катастрофоустойчивость и распределение нагрузки. Использование репликации и кластерных конфигураций усложняет администрирование, требует ответственного отношения и навыков настройки и эксплуатации. В этой главе мы кратко рассмотрим устройство репликации в PostgreSQL и варианты ее использования, а также более детально погрузимся в инструменты, которые предоставляет СУБД для отслеживания работы кластеров и механизма репликации.

7.1. Обзор репликации

Репликация — это процесс синхронизации данных между двумя и более узлами. Обычно различают два вида репликации: физическую и логическую. Физическая репликация предполагает передачу физических изменений блоков данных без анализа содержимого передаваемых изменений. Логическая репликация является более сложной и включает в себя анализ,

фильтрацию и выборочную передачу изменений. В минимальной конфигурации в репликации участвует два узла: один — в роли *основного* (primary), являясь источником изменений, второй — в роли *реплики*, принимая изменения с основного узла и воспроизводя их у себя. В зависимости от типа репликации терминология может изменяться: например, в случае физической репликации реплики обычно называют *резервными*, или *запасными* (standby) узлами. В более сложных схемах репликации могут участвовать несколько основных узлов, типы репликации могут комбинироваться, и каждый из отдельных узлов может реплицировать изменения на другие узлы. Основное применение репликация находит в задачах распределения нагрузки на несколько узлов и обеспечения отказоустойчивости: при аварии на основном узле можно переключить реплику в основной режим и продолжить обслуживать рабочую нагрузку с минимальным простоем (downtime) системы в целом. Физическая репликация получила распространение в силу своей универсальности и применимости в обоих сценариях. Логическая же репликация применяется для выборочной передачи набора данных, например в случае шардирования (sharding). В этой главе мы по большей части сосредоточимся на рассмотрении именно физической репликации, а логическую затронем в меньшей степени.

Оба типа репликации в PostgreSQL устроены схожим образом. Сначала выполняется начальная синхронизация данных с основного узла на реплику. Дальше реплика подключается к основному узлу, и все изменения, попадающие в WAL-журнал основного узла, передаются на реплику по протоколу репликации. Реплики, в свою очередь, также могут передавать изменения на другие узлы; таким образом можно выстраивать каскадные конфигурации.

Репликация по умолчанию является асинхронным процессом: изменения на основном сервере фиксируются независимо от наличия реплик. Репликацию можно переключить в синхронный режим (см. *synchronous_commit*), в котором транзакция на основном узле фиксируется только после подтверждения ее получения резервным узлом. В зависимости от значения параметра *synchronous_commit* подтверждение может отсылаться после сохранения полученных журнальных записей на диск (`remote_write`), синхронизации их с диском (`on`, по умолчанию) или воспроизведения принятых изменений (`remote_apply`). Синхронный режим позволяет в аварийных случаях свести потери зафиксированных транзакций к нулю. Однако использование синхронной репликации, особенно в режиме `remote_apply`, может значительно сказаться на производительности, и, более того, отказ реплики может привести к отказу в обслуживании на основном узле: все операции по фиксации транзакций будут ожидать доставки журнальных записей на реплику. Поэтому решение использовать синхронную репликацию должно быть взвешенным и обоснованным. В любом случае все изменения, записанные в журнал основного узла, появляются на репликах с некоторой *задержкой* (replication lag); ее величина зависит от таких факторов, как производительность узлов, сетевое окружение или текущая рабочая нагрузка.

Процесс репликации инициируется репликой. Процесс walreceiver проверяет состояние локального WAL-журнала и отправляет на основной узел запрос на передачу журнальных записей. Если у основного узла есть в распоряжении запрашиваемые записи, то на нем запускается

фоновый процесс walsender, который в дальнейшем и будет отвечать за потоковую передачу журнала. Оба процесса, walsender и walreceiver, поддерживают между собой постоянное соединение, и, как только в журнале появляются новые данные (включая данные еще незафиксированных транзакций), walsender передает их процессу walreceiver. Процесс walreceiver ожидает передачи данных. Получив журнальные записи, он сохраняет их в локальный каталог pg_wal основного каталога данных, после чего другой фоновый процесс, startup, *воспроизводит* (replay) переданные изменения на локальной копии данных. Как только изменения применены, обновленные данные становятся видны запросам, выполняющимся на реплике. Такой механизм репликации зарекомендовал себя как простой и надежный.

На основе передачи WAL-записей реализованы оба вида репликации — и физическая, и логическая. Поскольку в WAL-журнале фиксируются изменения на уровне отдельных физических блоков, физическая репликация и заключается в передаче изменений физических блоков. Эти изменения описывают атомарный переход блока из одного физического состояния в другое без уточнения, какие именно пользовательские данные были изменены (таблицы, строки или отдельные поля). Таким образом, процессы walsender и walreceiver обеспечивают непрерывную и последовательную передачу потока изменений между узлами, а процесс startup применяет все изменения, без пропусков и строго в порядке следования журнальных записей. Для оценки работы физической репликации достаточно отслеживать состояние процессов walsender и walreceiver и объем данных передаваемых основным узлом на реплики.

Логическая репликация более сложна и позволяет реплицировать изменения выборочно. В PostgreSQL эта функциональность реализована механизмом *публикаций*[1] и *подписок*[2]. В случае логической репликации исчезают понятия основного и резервного узлов, поскольку узлы логической репликации могут совмещать обе роли; для обозначения источника и получателя изменений используются термины *сервер публикации* и *сервер подписки*. Сервер публикации декодирует поток журнала и выявляет объекты, которые были изменены. В результате декодирования появляется возможность фильтровать поток записей и реплицировать изменения (все или определенный набор операций) отдельных объектов. Выбранные изменения передаются серверу подписчика, который сохраняет их локально и затем применяет транзакции в том порядке, в котором они были зафиксированы на стороне сервера публикации (а не в порядке следования записей в WAL, как в случае физической репликации).

При физической репликации могут использоваться так называемые *слоты репликации*, а в случае логической репликации они являются обязательными. Слоты позволяют удерживать необходимый объем WAL-сегментов, исключая возможность их переработки в ходе выполнения контрольных точек. Слоты являются важной частью механизма репликации и требуют внимания и контроля со стороны администратора: для мониторинга логической репликации нужно отслеживать не только передачу изменений, но и состояние слотов репликации.

[1] postgrespro.ru/docs/postgresql/current/sql-createpublication
[2] postgrespro.ru/docs/postgresql/current/sql-createsubscription

При эксплуатации кластеров репликации могут возникнуть разные проблемы, способные повлиять как на работоспособность кластера, так и на производительность отдельных узлов:

- отказ, недоступность узлов и невозможность передачи журнала;
- задержка при передаче журнала из-за недостаточной производительности узлов СУБД или среды передачи;
- недостаточная производительность при воспроизведении журнала на репликах и, как результат, отставание от основного узла;
- конфликты при воспроизведении журнала, приводящие к отставанию или ошибкам при выполнении запросов;
- аварийное завершение работы СУБД из-за нехватки места на диске в связи с накоплением сегментов журнала.

Как видно из приведенного списка, потенциальных проблем предостаточно. Некоторые из них могут приводить к деградации или даже к полному отказу в обслуживании с непрогнозируемым временем восстановления. Поэтому в обязанности администратора входит отслеживание состояния кластера и работоспособности отдельных узлов, а также предупреждение проблем, способных нарушить нормальную работу. Дальше мы рассмотрим инструменты, которые предлагает СУБД для отслеживания репликации и выявления связанных с ней проблем.

7.2. Инструменты отслеживания репликации

Для отслеживания репликации есть несколько инструментов, и в зависимости от способа репликации могут понадобиться только некоторые из них:

- `pg_stat_replication` — отслеживание процессов walsender и процедуры передачи журнала на реплики;
- `pg_stat_wal_receiver` — отслеживание процесса walreceiver и процедуры приема журнала;
- `pg_replication_slots` — отслеживание слотов репликации, как физических, так и логических;
- `pg_stat_replication_slots` — статистика использования логических слотов репликации;
- `pg_stat_subscription` — статистика работы подписчиков;
- `pg_stat_subscription_stats` — статистика ошибок в работе подписчиков;
- `pg_stat_database_conflicts` — статистика конфликтов восстановления с разделением по причинам их возникновения.

Далее мы рассмотрим эти инструменты и возможные сценарии их использования.

Представление pg_stat_replication

Основной узел является источником всех изменений и может реплицировать данные больше чем на один узел. Скорость отправки изменений и скорость их применения на репликах могут различаться, что выражается в величине отставания репликации и степени актуальности данных. Для определения отставания репликации и других характеристик передачи журнала можно использовать представление pg_stat_replication. В каждой его строке содержится статистика работы отдельного процесса walsender. Представление pg_stat_replication показывает текущие данные аналогично тому, как работает представление pg_stat_activity:

```
# TABLE pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
pid              | 39
usesysid         | 16431
usename          | replica
application_name | walreceiver
client_addr      | 172.22.0.3
client_hostname  |
client_port      | 46666
backend_start    | 2022-12-14 03:54:32.228266+00
backend_xmin     |
state            | streaming
sent_lsn         | D8/EF27AFC0
write_lsn        | D8/EF27AFC0
flush_lsn        | D8/EF27AFC0
replay_lsn       | D8/EF27AFC0
write_lag        | 00:00:00.000159
flush_lag        | 00:00:00.002637
replay_lag       | 00:00:00.002807
sync_priority    | 0
sync_state       | async
reply_time       | 2022-12-16 04:36:36.495495+00
```

В тестовом окружении настроен кластер репликации с одним запасным узлом; соответственно, в представлении будет всего одна строка. Набор полей таков:

- pid — идентификатор процесса walsender в операционной системе;
- usesysid, usename — идентификатор и имя пользователя СУБД, от которого выполнено подключение со стороны резервного узла;
- application_name — дополнительный идентификатор приложения, который можно указать при подключении к основному узлу;
- client_addr, client_hostname, client_port — сетевые реквизиты со стороны резервного узла: адрес, имя узла и номер порта;
- backend_start — отметка времени подключения и создания процесса walsender;

- backend_xmin — транзакционный горизонт реплики, переданный основному узлу через обратную связь (см. *hot_standby_feedback*);
- state — текущее состояние процесса walsender, определяющее и общее состояние репликации:
  - startup — walsender находится в процессе запуска;
  - catchup — реплика пытается «догнать» основной узел;
  - streaming — это основной режим работы: walsender передает поток изменений на реплику, после того как реплика успешно «догнала» основной узел;
  - backup — walsender находится в процессе передачи резервной копии;
  - stopping — walsender находится в процессе выключения.

На основе следующих полей, показывающих состояние отправки и приема журнала, можно определить величину отставания конкретной реплики и объем работы, который ей предстоит проделать, чтобы полностью устранить это отставание:

- sent_lsn — позиция журнала, записи до которой отправлены на реплику;
- write_lsn — позиция журнала, записи до которой приняты, но еще не синхронизированы процессом walreceiver;
- flush_lsn — позиция журнала, записи до которой уже синхронизированы процессом walreceiver;
- replay_lsn — позиция журнала, записи до которой воспроизведены процессом startup.

Как уже упоминалось, изменения на репликах могут появляться с задержкой. Величина задержки (отставание репликации) в идеале должна быть близка к нулю, но это не всегда возможно. Отставание еще можно расценивать как меру работы, которую необходимо проделать. Однако основной узел не стоит на месте, и в его журнал записываются новые и новые изменения. Объем появляющихся изменений может варьироваться; соответственно, отставание реплики тоже будет непостоянным. Увеличение отставания может выражаться в том, что результаты выполнения запросов к основному узлу и реплике будут расходиться, и это следует учитывать при проектировании приложений, которые получают данные с реплик.

Чтобы «догнать» основной узел, реплике требуется: а) получить все изменения с него, б) записать их в локальное хранилище, в) синхронизировать и г) воспроизвести над локальной копией данных. Используя вышеперечисленные поля с позициями журнала, можно вычислить расстояние в байтах между позициями и получить представление о том, в каком месте образовалась наибольшая задержка. Для этого потребуются уже известная функция pg_current_wal_lsn и оператор вычитания, который поддерживается типом данных pg_lsn:

- pg_current_wal_lsn - sent_lsn — объем данных к отправке, который состоит из всех журнальных записей, накопленных на основном узле, но еще не переданных на реплику;

- sent_lsn - write_lsn — объем данных, отправленных с основного узла, но еще не записанных на стороне реплики;
- write_lsn - flush_lsn — объем данных, записанных, но еще не синхронизированных на реплике;
- flush_lsn - replay_lsn — объем данных, готовых для воспроизведения процессом startup.

Полученный объем выражается в байтах и позволяет довольно наглядно представить масштабы отставания реплик. На практике отставание часто возникает на этапе отправки или воспроизведения журнала. Часто это происходит из-за того, что в результате изменения больших объемов данных генерируется много журнальных записей и процессы walsender или walreceiver не справляются с передачей или процессу startup не хватает производительности, чтобы своевременно воспроизводить весь объем журналов.

Следующая группа полей позволяет сразу отслеживать отставание в секундах, показывая приблизительное время, которое потребуется реплике, чтобы «догнать» основной узел:

- write_lag — время, прошедшее между локальной синхронизацией журнала и получением уведомления о том, что изменения записаны (но еще не синхронизированы) на реплике;
- flush_lag — время, прошедшее между локальной синхронизацией журнала и получением уведомления о том, что изменения записаны и синхронизированы на реплике;
- replay_lag — время, прошедшее между локальной синхронизацией журнала и получением уведомления о том, что изменения записаны, синхронизированы и применены;
- reply_time — отметка времени о получении последнего служебного сообщения от реплики, в котором и содержатся данные по обработке журнала.

Данные поля позволяют оценить задержку, возникающую при подтверждении транзакции в случае синхронной репликации, когда параметр *synchronous_commit* установлен в одно из значений — remote_write, on или remote_apply.

Остаются еще два поля: sync_priority и sync_state. При асинхронной репликации sync_state всегда равно async. В случае синхронной репликации узлы могут находиться в разных состояниях и выполнять в кластере разные роли. Значение sync означает, что в данный момент реплика работает как синхронная. Если используется синхронная репликация, *основанная на приоритетах*, значение potential говорит о том, что реплика работает в асинхронном режиме, но является кандидатом на переход в синхронный режим (в этом случае sync_priority показывает приоритет данного узла). Если же используется синхронная репликация, *основанная на кворуме*, значение quorum означает, что реплика входит в минимально необходимое число узлов, подтвердивших получение изменений. Больше информации о настройке синхронной репликации можно получить из документации[1].

[1] postgrespro.ru/docs/postgrespro/15/runtime-config-replication#GUC-SYNCHRONOUS-STANDBY-NAMES

С точки зрения поиска и устранения проблем интересными являются поля с адресом, позициями обработки журнала и задержками, которые позволяют точно идентифицировать реплики (особенно если их несколько) и определить, на каких этапах обработки журнала возникают задержки. Поля с позициями журнала можно сразу преобразовать в величину задержек:

```
# SELECT
    client_addr,
    state,
    pg_current_wal_lsn() - replay_lsn AS total_lag_bytes,
    pg_current_wal_lsn() - sent_lsn AS pending_bytes,
    sent_lsn - write_lsn AS write_lag_bytes,
    write_lsn - flush_lsn AS flush_lag_bytes,
    flush_lsn - replay_lsn AS replay_lag_bytes,
    write_lag,
    flush_lag,
    replay_lag
  FROM pg_stat_replication;
-[ RECORD 1 ]----+----------------
client_addr      | 172.22.0.3
state            | streaming
total_lag_bytes  | 17152
pending_bytes    | 0
write_lag_bytes  | 0
flush_lag_bytes  | 17152
replay_lag_bytes | 0
write_lag        | 00:00:00.000151
flush_lag        | 00:00:00.000151
replay_lag       | 00:00:00.000151
```

Представление позволяет оценивать величину задержки как в байтах, так и в секундах, показывая, какой объем данных нужно обработать реплике, чтобы «догнать» основной узел, и сколько времени это займет. В выводе запроса видно, что отставание реплики незначительно и составляет несколько микросекунд, в байтах задержка составляет всего около 17 КБ, при этом данные уже переданы и записаны на реплику, их осталось только синхронизировать и воспроизвести.

На основе величин задержек можно собирать метрики и строить графики, которые будут отображать отставание реплик во времени (рис. 7.1 и 7.2).

Обратите внимание на легенды обоих графиков. Отставание в байтах представлено более подробно и содержит информацию об объеме журнала, который еще не был отправлен на реплику. Второй любопытный факт связан с тем, что в выбранный момент оба графика показывают разные максимумы отставания. Например, большая часть отставания в байтах находится на этапе записи изменений, в то время как отставание в секундах — уже на этапах синхронизации и воспроизведения. Особенно хорошо это заметно на небольших значениях. Общая картина обоих графиков показывает, что отставание часто появляется на этапах синхронизации и воспроизведения. Это говорит о том, что реплика чуть хуже справляется со своей частью работы.

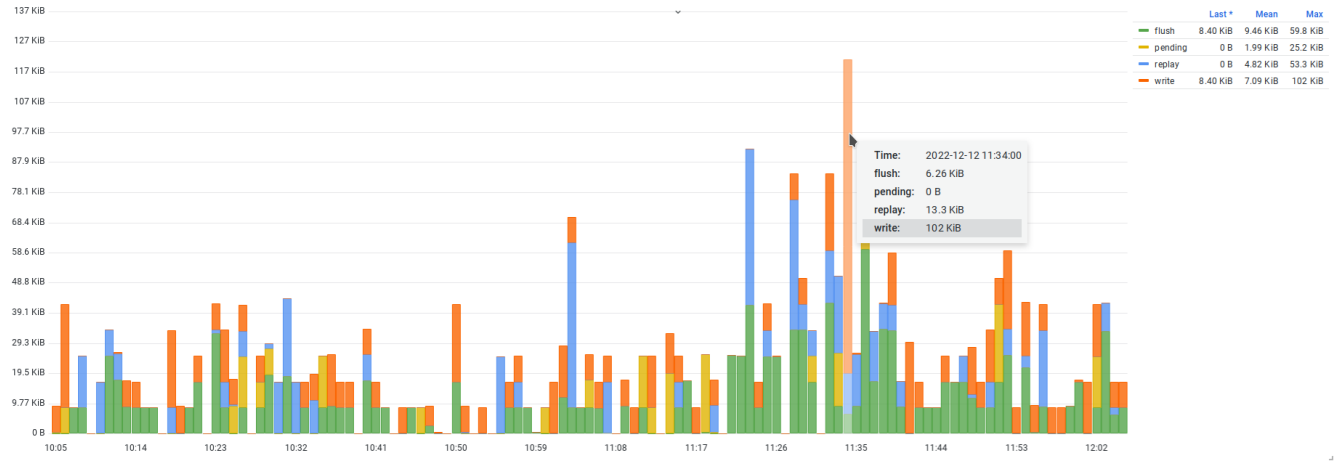


Рис. 7.1. Отставание реплики в байтах

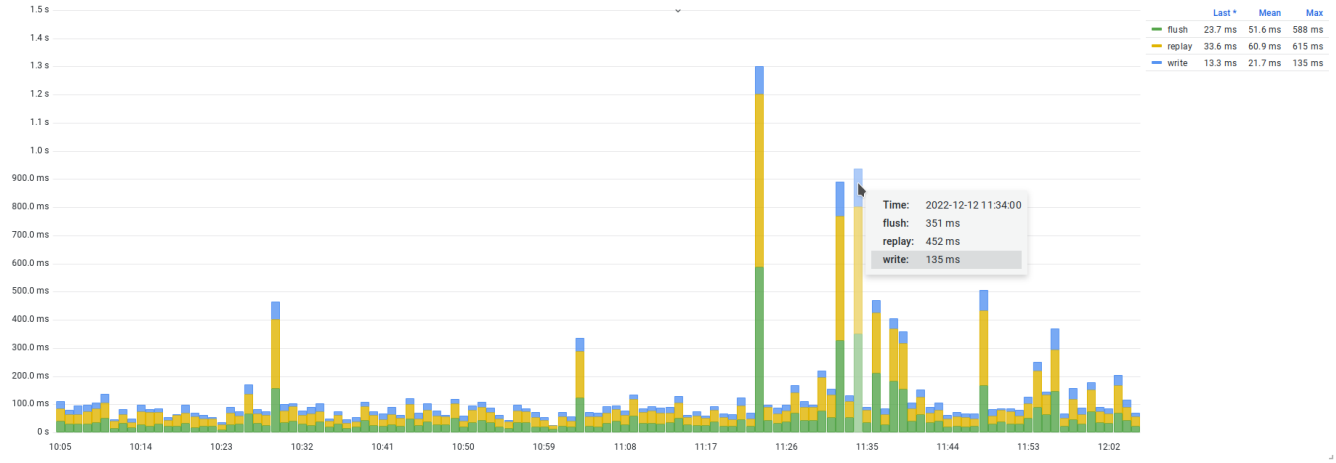


Рис. 7.2. Отставание реплики в секундах

Представление pg_stat_wal_receiver

Следующий способ отслеживания репликации доступен на резервных узлах с помощью представления `pg_stat_wal_receiver`. Представление отражает состояние репликации со стороны процесса walreceiver и содержит следующую информацию:

- `pid` — идентификатор процесса walreceiver в операционной системе;
- `status` — состояние процесса walreceiver;
- `receive_start_lsn` — позиция журнала на момент старта процесса walreceiver;
- `receive_start_tli` — линия времени в журнале на момент старта процесса walreceiver;

- written_lsn — позиция журнала, соответствующая последним записанным, но еще не синхронизированным данным;
- flushed_lsn — позиция журнала, соответствующая последним записанными и синхронизированным данным. Эта позиция используется как отправная точка при запуске процесса walreceiver;
- received_tli — номер линии времени, соответствующий последним записанным и синхронизированным данным. Линия времени используется как отправная линия при запуске процесса walreceiver;
- last_msg_send_time — время отправки последнего сообщения, полученного от основного узла;
- last_msg_receipt_time — время поступления последнего сообщения, полученного от основного узла;
- latest_end_lsn — позиция журнала, отправленная репликой основному узлу в последнем сообщении;
- latest_end_time — время последней позиции в журнале, отправленной основному узлу в последнем сообщении;
- slot_name — имя слота, используемого в данном сеансе репликации;
- sender_host — имя основного узла, с которым установлено соединение;
- sender_port — номер сетевого порта основного узла, с которым установлено соединение;
- conninfo — строка подключения к основному узлу, используемая процессом walreceiver.

В представлении нет информации о воспроизведении журнала, поскольку это задача процесса startup. Процесс walreceiver занимается только приемом и записью журнала. В целях мониторинга это представление практически не используется, поскольку все данные можно получить с основного узла через pg_stat_replication.

Помимо pg_stat_wal_receiver, есть несколько функций, которые позволяют получить дополнительную информацию о журнале:

- pg_last_wal_receive_lsn — последняя позиция журнала, полученного с основного узла;
- pg_last_wal_replay_lsn — последняя позиция журнала, записи до которой воспроизведены процессом startup;
- pg_is_in_recovery — находится ли СУБД в режиме восстановления (это рабочий режим реплики, так как она постоянно восстанавливается на основе журнала).

Функцию pg_is_in_recovery удобно использовать для определения, является узел основным или резервным. Функции pg_last_wal_receive_lsn и pg_last_wal_replay_lsn подходят для определения объема журнала на реплике, который необходимо воспроизвести, однако эти функции нельзя использовать для расчета отставания, поскольку они не учитывают тот объем журнала, который находится на основном сервере и должен быть передан на реплику. В случае отказа основного узла и его полной недоступности (при аппаратной неисправности, сетевой

недоступности или из-за катастрофы) эти функции можно использовать для определения узла, который наилучшим образом подходит на роль основного: сравнивая позиции журнала, следует выбрать узел, который успел принять больше журнальных записей или воспроизвести бóльшую их часть.

Слоты репликации и pg_replication_slots

При нормальной работе репликации основной узел отправляет журналы на реплику, а реплика воспроизводит изменения на локальной копии данных. Однако могут возникнуть обстоятельства, при которых реплика не будет успевать воспроизводить изменения, или даже не будет успевать получать и записывать журнал. Например, из-за увеличенной нагрузки на основном узле генерируется большой объем журнала, пропускной способности сети становится недостаточно, сегменты журнала начинают скапливаться на основном узле и отставание реплики все увеличивается. Или передача журналов прерывается из-за нарушения сетевой связности между основным узлом и репликой, и репликация останавливается. В ранних версиях PostgreSQL основной узел никак не отслеживал необходимость реплики в тех или иных сегментах журнала — если реплика отключилась, сервер продолжает работу и при выполнении контрольной точки может удалить те журналы, которые еще не были переданы на реплику. Если реплика вновь подключится и запросит журнал с известной ей позиции, а на основном узле таких сегментов журнала уже нет, то репликация не сможет продолжиться. Решением этой проблемы являлась либо загрузка необходимых сегментов из архива, либо повторная инициализация реплики. Оба варианта имеют свои минусы: содержать архив сегментов может быть накладно по ресурсам, а инициализация может занимать много времени. Для минимизации проблемы можно использовать параметр *wal_keep_size* (ранее назывался *wal_keep_segments*), который указывает СУБД удерживать от переработки дополнительный объем журнала как раз на случай подобных отказов реплик. Другой, современный способ решить проблему — использовать слоты репликации.

Слоты репликации гарантируют удерживание сегментов журнала в рамках конкретного сеанса репликации до тех пор, пока они не будут переданы на реплику. Однако при продолжительном отключении реплики и большом приросте новых сегментов возникает риск полного исчерпания свободного места, что может привести к аварийному завершению СУБД. В качестве защитной меры появился параметр *max_slot_wal_keep_size*, позволяющий указать предельный объем, при достижении которого СУБД начнет удалять сегменты, и напоминающий улучшенный вариант *wal_keep_size*. В любом случае независимо от выбранных настроек остается риск либо полного и необратимого отставания реплики (с возможностью восстановления только с помощью WAL-архива), либо аварийного завершения СУБД из-за исчерпания места.

Поэтому применение слотов репликации добавляет администратору задачу отслеживания их состояния и использования. Для этого есть представление `pg_replication_slots`, которое содержит информацию обо всех слотах репликации, по одной строке на каждый:

```
# TABLE pg_replication_slots;
-[ RECORD 1 ]-------+------------
slot_name           | standby
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | t
active_pid          | 39
xmin                |
catalog_xmin        |
restart_lsn         | D3/94ADE4C8
confirmed_flush_lsn |
wal_status          | reserved
safe_wal_size       |
two_phase           | f
```

Обычно этого представления достаточно для оценки состояния слотов и определения факта существования проблем:

- `slot_name`, `slot_type`, `plugin` — поля, идентифицирующие слот: его имя, тип и плагин для декодирования потока WAL-записей (указываются только для логических слотов);
- `datoid`, `database` — идентификатор и имя базы данных, с которой ассоциирован слот (только для логических слотов;
- `temporary` — флаг, указывающий на то, что слот является временным. Состояние временных слотов не сохраняется на диске, в случае ошибки или завершения сеанса они автоматически удаляются;
- `active` — флаг, указывающий на то, что слот является активным и используется в данный момент. Это один из главных атрибутов слота, по которому можно определить, что потребитель слота жив;
- `active_pid` — идентификатор процесса сеанса, который использует слот в данный момент;
- `xmin` — идентификатор транзакции, определяющий горизонт возможной очистки версии строк;
- `catalog_xmin` — аналогичный горизонт для версий строк в объектах системного каталога;
- `restart_lsn` — позиция в журнале, начиная с которой записи еще могут понадобиться потребителю в рамках работы со слотом. Все журнальные записи с этой позиции не могут быть переработаны в ходе выполнения контрольной точки до тех пор, пока их объем не превысит *max_slot_wal_keep_size*;
- `confirmed_flush_lsn` — позиция журнала, записи до которой потребитель уже гарантированно получил (только для логических слотов);
- `wal_status` — состояние файлов журнала, которые требуются для этого слота:

  - reserved — требуемый объем журнала укладывается в ограничение *max_wal_size*;
  - extended — объем журнала превышает ограничение *max_wal_size*, но сегменты все еще удерживаются слотом или укладываются в ограничение *wal_keep_size*;
  - unreserved — для слота больше не сохраняются требуемые сегменты журнала и некоторые из них будут удалены при следующей контрольной точке. Это говорит о том, что потребитель не успевает вычитывать изменения, но, если скорость увеличится, это состояние еще может вернуться к reserved или extended;
  - lost — необходимые сегменты журнала удалены, и данный слот уже нельзя использовать;
- safe_wal_size — объем изменений в байтах, при записи которого в журнал слот окажется в состоянии lost (при отсутствии потребителя). Значение может быть не указано (NULL), если слот уже потерян или ограничение *max_slot_wal_keep_size* не установлено;
- two_phase — флаг, указывающий на то, что слот используется для декодирования подготовленных транзакций (только для логических слотов).

При мониторинге слотов важными показателями являются:

- Разница между текущей позицией журнала (значение функции pg_current_wal_lsn) и позицией confirmed_flush_lsn. По сути, это объем данных в очереди на отправку подписчику, определяющий его отставание. Чем больше это значение, тем больше накоплено журналов и тем больше времени потребуется подписчику для обработки изменений.
- Величина горизонта транзакций (на основе полей xmin и catalog_xmin), который учитывается процессами очистки при удалении устаревших версии строк.
- Доступность сегментов журнала для потребителя слота (wal_status), которую СУБД оценивает самостоятельно.
- Величина safe_wal_size, указывающая предельный объем журнала, при исчерпании которого появляется риск переработки журнала и невозможности использовать слот.

Так, например, в тестовом окружении репликация работает с использованием слота и отставание можно отслеживать с помощью pg_replication_slots (рис. 7.3):

Можно поставить небольшой эксперимент — установить *max_slot_wal_keep_size* и выключить реплику — и посмотреть, как меняется статистика. Некоторое время СУБД будет удерживать сегменты, однако, после того как объем журнала превысит *max_slot_wal_keep_size*, СУБД переработает часть сегментов, что сделает слот недоступным для дальнейшего использования. Итак, включаем *max_slot_wal_keep_size*:

```
# ALTER SYSTEM SET max_slot_wal_keep_size TO '1GB';
# SELECT pg_reload_conf();
```

Далее выключаем контейнер, в котором запущена реплика:

```
# cd playground
# docker-compose stop standby
```

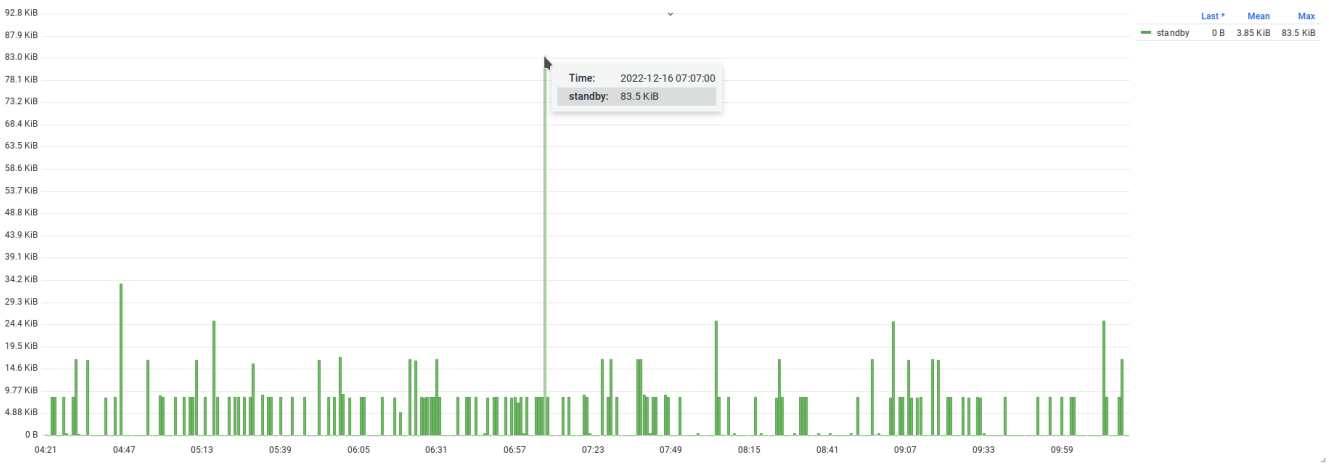


Рис. 7.3. Объем журнала, удерживаемый слотом репликации

Теперь посмотрим содержимое `pg_replication_slots`:

```
# SELECT
    slot_name, active,
    pg_current_wal_lsn() - restart_lsn AS backlog,
    wal_status, safe_wal_size
  FROM pg_replication_slots;
 slot_name | active |  backlog  | wal_status | safe_wal_size
-----------+--------+-----------+------------+---------------
 standby   | f      | 353427008 | reserved   |     722212616
```

Из-за выключения резервного узла через слот перестали передаваться изменения, и слот стал неактивным (`active = f`). При этом рабочая нагрузка никуда не делась, и новые изменения продолжают записываться в WAL-журнал, в результате чего растет отставание потребителя (поле `backlog`). Вместе с ростом отставания начинает уменьшаться запас объема журнала `safe_wal_size`, при этом состояние журнала пока `reserved`, что указывает на то, что требуемый объем журнала укладывается в *max_wal_size*.

Можно запустить метакоманду `\watch 60` и понаблюдать за тем, как отставание будет увеличиваться со временем и в какой-то момент `safe_wal_size` станет отрицательным.

```
Tue Jan  3 14:21:54 2023 (every 60s)

 slot_name | active |  backlog   | wal_status | safe_wal_size
-----------+--------+------------+------------+---------------
 standby   | f      | 1115080952 | unreserved |     -39441328
```

```
Tue Jan  3 14:22:54 2023 (every 60s)

 slot_name | active | backlog | wal_status | safe_wal_size
-----------+--------+---------+------------+---------------
 standby   | f      |         | lost       |
```

Если не включить вовремя реплику, слот будет потерян, что и произошло в этом эксперименте. В журнале сообщений можно найти запись о том, что слот стал недействительным в процессе выполнения контрольной точки:

```
2023-01-03 14:18:10.111 UTC 21 LOG: checkpoint starting: time
2023-01-03 14:22:40.095 UTC 21 LOG: invalidating slot "standby" because its restart_lsn E3/DDE30AB8
exceeds max_slot_wal_keep_size
2023-01-03 14:22:40.458 UTC 21 LOG: checkpoint complete: wrote 3545 buffers (21.6%); 0 WAL file(s)
added, 57 removed, 6 recycled; write=269.931 s, sync=0.015 s, total=270.348 s; sync files=10,
longest=0.009 s, average=0.002 s; distance=78162 kB, estimate=81638 kB
```

Для возвращения реплики в строй придется повторно инициализировать ее (поскольку в тестовом окружении нет WAL-архива):

```
# docker-compose rm standby
# docker volume rm playground_standby_data
# docker-compose up -d
```

После пересоздания контейнера статус слота покажет, что слот снова начал использоваться, как и прежде:

```
# SELECT
    slot_name, active,
    pg_current_wal_lsn() - restart_lsn AS backlog,
    wal_status, safe_wal_size
  FROM pg_replication_slots;
 slot_name | active |  backlog  | wal_status | safe_wal_size
-----------+--------+-----------+------------+---------------
 standby   | f      |         0 | reserved   |    1090234536
```

Публикации и подписки

Логическая репликация использует модель публикаций и подписок. На одном узле можно опубликовать изменения, происходящие в одной или нескольких таблицах, а на другом — подписаться на получение этих изменений и воспроизводить их. Как и в случае физической репликации, механизм публикаций и подписок реализован на основе передачи журнала упреждающей записи. Процесс репликации инициируется на стороне подписки: при создании подписки запускается фоновый рабочий процесс logical replication worker, который подключается по указанному адресу к серверу публикации. На стороне публикации запускаются два процесса walsender, один из которых выполняет начальное копирование целевых таблиц и завершается,

а второй остается и занимается передачей WAL-журнала. Если серверов подписки несколько, для каждого из них будет запущен отдельный процесс walsender, но не больше, чем указано в параметре *max_wal_senders*. Дальше начинается процесс декодирования WAL-журнала, при котором происходят разбор журнальных записей, извлечение данных об изменении отдельных строк таблиц и их группировка по отдельным транзакциям. В результате подписчику передаются данные зафиксированных транзакций. Для декодирования используется буфер, размер которого определяется параметром *logical_decoding_work_mem* (64 МБ по умолчанию). При нехватке этого буфера отправитель может принять решение о вытеснении содержимого буфера на диск или немедленной отправке части декодированных данных подписчику. Это решение принимается на основе параметра `streaming`, указанного при создании подписчика. По умолчанию потоковая передача выключена, и транзакция сначала полностью декодируется на стороне отправителя, и только затем декодированные данные отправляются подписчику. Узнать больше об устройстве, настройке и работе логической информации можно в официальной документации[1].

В тестовом окружении не используется логическая репликация и нет ни публикаций, ни подписок, однако давайте рассмотрим инструменты для отслеживания их работы.

Представление `pg_stat_replication_slots` содержит информацию только о слотах логической репликации и может быть использовано для оценки объемов выполняемой работы. Каждая строка представления содержит статистику по отдельному логическому слоту:

- `slot_name` — уникальный идентификатор слота;
- `spill_txns` — общее количество транзакций, вытесненных на диск из-за нехватки рабочей памяти при декодировании журнала. Учитываются как транзакции верхнего уровня, так и вложенные транзакции (subtransaction);
- `spill_count` — общее количество фактов вытеснения транзакций на диск при декодировании журнала. Счетчик увеличивается каждый раз при вытеснении транзакции, одна и та же транзакция может быть вытеснена несколько раз;
- `spill_bytes` — общий объем декодированных данных в байтах, вытесненных на диск при декодировании журнала;
- `stream_txns` — общее количество активных транзакций, данные которых отправлялись подписчику из-за нехватки рабочей памяти при декодировании журнала. Таким образом передаваться могут только данные транзакций верхнего уровня (данные вложенных транзакций не передаются в потоке отдельно), поэтому счетчик не учитывает вложенные транзакции;
- `stream_count` — общее количество раз, когда данные активных транзакций отправлялись подписчику из-за нехватки рабочей памяти при декодировании журнала. Этот счетчик увеличивается каждый раз, когда данные транзакций передаются в потоковом режиме; одна и та же транзакция может быть передана таким способом несколько раз;

[1] postgrespro.ru/docs/postgresql/current/logical-replication

- stream_bytes — общий объем декодированных данных в байтах, переданных подписчику в потоковом режиме;
- total_txns — общее количество декодированных транзакций, отправленных подписчику. Учитываются только транзакции верхнего уровня, но не вложенные транзакции. Счетчик учитывает как транзакции, передаваемые в потоковом режиме, так и вытесненные транзакции;
- total_bytes — общий объем декодированных данных в байтах, отправленных подписчику. Счетчик учитывает данные транзакций, передаваемых как в потоковом режиме, так и вытесненных;
- stats_reset — отметка времени сброса этой статистики.

Счетчики помогают оценить объем работы и ввода-вывода, связанного с логическим декодированием журнала, и настроить параметр *logical_decoding_work_mem*.

Следующий инструмент — это представление pg_stat_subscription со статистикой подписок, которое позволяет отслеживать процесс получения журнала со стороны отправителя:

- subid, subname — уникальный идентификатор и имя подписки;
- pid — процесс, обслуживающий подписку;
- relid — идентификатор отношения, для которого в данных момент выполняется начальная синхронизация. Значение будет отсутствовать (NULL), если процесс работает в основном рабочем режиме, в котором только применяются изменения;
- received_lsn — последняя позиция журнала, записи до которой приняты подписчиком;
- last_msg_send_time — время отправки последнего сообщения, полученного подписчиком;
- last_msg_receipt_time — время поступления последнего сообщения, полученного со стороны публикующего узла;
- latest_end_lsn — последняя локальная позиция в журнале, подтвержденная подписчиком публикующему узлу;
- latest_end_time — время последней локальной позиции в журнале, подтвержденное подписчиком публикующему узлу.

На практике pg_stat_subscription может пригодиться для отслеживания состояния подписок и скорости обработки приходящих изменений.

Следующий инструмент — это представление pg_stat_subscription_stats, которое помогает отслеживать появление ошибок на стороне подписки. При репликации изменений, связанных с операциями обновления и удаления (команды UPDATE и DELETE), используется так называемый *репликационный идентификатор*, который, как правило, основан на значениях первичного ключа таблицы. Вместо первичного ключа может использоваться любой другой уникальный ключ или вообще вся строка. Если при репликации изменений происходит нарушение уникальности репликационного идентификатора, например при попытке вставить строку с идентификатором, который уже присутствует в таблице, то возникает *конфликт*, который

приводит к ошибке. Автоматическое разрешение конфликтов отсутствует; процесс синхронизации или применения изменений останавливается и требует ручного вмешательства администратора. Чтобы таких конфликтов не возникало, при начальной синхронизации таблицы на стороне подписчика должны быть пустыми, а при действующей репликации следует избегать внесения изменений на стороне подписки. Но даже при соблюдении этих условий полезно отслеживать появление ошибок с помощью представления `pg_stat_subscription_stats`, в котором есть следующие поля:

- `subid`, `subname` — уникальный идентификатор и имя подписки;
- `apply_error_count` — общее количество ошибок, возникших при применении изменений в режиме действующей репликации;
- `sync_error_count` — общее количество ошибок, возникших во время начальной синхронизации таблиц;
- `stats_reset` — отметка времени сброса этой статистики.

Когда механизм публикаций и подписок был добавлен, подобные ошибки фиксировались только в журнале сообщений, который приходилось обрабатывать и анализировать. После добавления `pg_stat_subscription_stats` отслеживать подобные ошибки стало проще.

7.3. Конфликты восстановления

Один из сценариев применения физической репликации связан с распределением нагрузки таким образом, что часть запросов от приложений выполняются на основном узле, а часть запросов (на чтение) — на репликах, благодаря чему более эффективно используются вычислительные ресурсы всего кластера СУБД.

В процессе воспроизведения изменений на реплике, выполняющей нагрузку на чтение, существует вероятность того, что применение очередной журнальной записи приведет к невозможности продолжить выполнение запроса. Такая ситуация называется *конфликтом восстановления* (recovery conflict). Перед СУБД встает вопрос: продолжить выполнение запроса, но при этом остановить процесс применения изменений (то есть приостановить репликацию), или отменить запрос и применить изменения. СУБД поступает следующим образом: применение изменений приостанавливается на величину *max_standby_streaming_delay* (по умолчанию 30 секунд), чтобы дать запросу возможность выполниться. Если запрос не успел выполниться за это время, то СУБД принудительно завершает его и возобновляет применение изменений. Важно отметить, что во время отсчета этой задержки СУБД продолжает получать от основного узла новые журнальные записи, но не применяет их. В лучшем случае запросы, задерживающие репликацию, появляются редко и отставание, накопленное за время выполнения таких запросов, будет быстро наверстываться. В худшем же случае применение журнальных записей может постоянно откладываться из-за большого количества одновременно выполняемых запросов: в этом случае после завершения одного конфликтующего запроса та же журнальная

запись может вступить в конфликт уже со следующим запросом, из-за чего отставание может накапливаться.

В случае принудительной отмены запроса приложение получит ошибку, которая будет также зафиксирована в журнале сообщений:

```
ERROR:  canceling statement due to conflict with recovery
DETAIL: User query might have needed to see row versions that must be removed.
```

Конфликты восстановления могут возникать и при восстановлении из архива, когда вместо потоковой репликации сегменты журнала копируются из WAL-архива. В таком случае величина задержки определяется значением *max_standby_archive_delay*, по умолчанию равным тем же 30 секундам.

Как поступать с такими ошибками, зависит от бизнес-требований: важно определить, что является наименьшим из двух зол — отставание от основного узла или прерывание запросов. Если важно выполнение запросов без ошибок, то имеет смысл увеличить величину задержки *max_standby_streaming_delay*, что потенциально может приводить к еще большему отставанию реплики. Если же, наоборот, отставание реплики недопустимо (например, реплика является основным кандидатом для аварийного переключения), то ошибки можно игнорировать или обрабатывать их на стороне приложения, повторяя сбойные запросы на основном узле. Другим частым решением является использование отдельной реплики с очень большой величиной допустимой задержки и потенциально большим отставанием. Такая реплика используется преимущественно для выполнения продолжительных запросов и не участвует в процедурах аварийного переключения.

Помимо журнала сообщений, конфликты восстановления можно отследить с помощью `pg_stat_database.conflicts`: этот счетчик учитывает все типы конфликтов для каждой из существующих баз данных. Более детально причины конфликтов можно отследить в представлении `pg_stat_database_conflicts`. Это представление актуально только для реплик, поскольку конфликты происходят только там. Представление содержит список баз и счетчики для каждого типа конфликтов:

- `datid`, `datname` — уникальный идентификатор и имя базы данных;
- `confl_tablespace` — общее количество запросов, отмененных по причине удаления табличного пространства;
- `confl_lock` — общее количество запросов, отмененных по причине истечения тайм-аута блокировки;
- `confl_snapshot` — общее количество запросов, отмененных по причине устаревания снимка данных;
- `confl_bufferpin` — общее количество запросов, отмененных по причине закрепления буфера в общем кеше;
- `confl_deadlock` — общее количество запросов, отмененных по причине возникновения взаимоблокировки.

Обычно конфликты восстановления относят к ошибкам и при появлении ненулевых значений в `pg_stat_database.conflicts` детали и причины конфликтов расследуются уже с помощью журнала сообщений (какие именно запросы вызвали конфликты) и представления `pg_stat_database_conflicts` (какие именно конфликты возникали).

Причинами конфликтов могут быть разные события, возникающие на основном узле. На практике чаще всего встречается отмена запросов из-за очистки: изменения, вызванные очисткой устаревших версий строк, попадают на реплику, в то время как на ней выполняется запрос, которому все еще нужны эти версии (счетчик `confl_snapshot`). Как правило, это продолжительные аналитические запросы, обрабатывающие большие объемы данных, из-за чего они могут выполняться непредсказуемо долго и требовать большого объема ресурсов. Отмена и повторение таких запросов может обходиться дорого как по ресурсам, так и по времени ожидания результата. Для уменьшения вероятности отмены запросов есть две меры, которые можно принимать как по отдельности, так и вместе. Первая — это увеличение величины допустимой задержки *max_standby_streaming_delay*. Вторая — включение обратной связи *hot_standby_feedback*. С помощью обратной связи процесс walsender будет получать от реплики необходимый ей горизонт и учитывать его при удалении устаревших версий строк (этот горизонт виден в `pg_stat_replication.backend_xmin`, если слот не используется, или в `pg_replication_slots.xmin`, если используется). В этом случае конфликтующих изменений не возникнет (подробно о том, что такое транзакционный горизонт, можно узнать в главе 8, посвященной очистке). Однако ни один из этих способов не дает 100 % гарантии, что запрос будет выполнен, и оказывают определенное негативное влияние. Увеличение задержки может приводить к росту отставания, но, если время выполнения запроса превысит задержку, запрос все равно будет отменен (в крайнем случае можно заставить реплику ждать бесконечно, установив значение параметра в –1). Обратная связь гарантирует защиту только от очистки, но за счет откладывания очистки таблицы будут раздуваться; кроме того, остаются и другие причины, по которым могут произойти конфликт и последующая отмена запроса. Поэтому идеального решения проблемы не существует и следует отталкиваться от бизнес-требований о допустимости отмены запросов и возможной величине отставания. В качестве компромиссного решения для аналитических запросов можно сделать отдельную реплику с большим значением допустимой задержки, но не использовать ее в выборах основного узла. Тогда большое отставание будет допустимо; при этом можно отказаться от использования обратной связи и минимизировать эффекты раздувания.

Резюме

- Репликация практически всегда используется в производственной среде.
- Репликация работает на основе передачи WAL-журнала.
- Репликация бывает физической и логической.
- `pg_stat_replication` — основной инструмент отслеживания репликации.

- Для отслеживания репликации на репликах используется `pg_stat_wal_receiver`.
- Отставание репликации — самая часто возникающая проблема.
- Величину отставания можно измерять в байтах или секундах.
- Слоты репликации позволяют исключить риск необратимого отставания реплик, но добавляют риск исчерпания места на диске.
- Для отслеживания слотов репликации используется представление `pg_replication_slots`.
- Логическая репликация использует модель публикаций и подписок.
- Для отслеживания слотов логической репликации можно использовать представление `pg_stat_replication_slots`.
- Для отслеживания подписок можно использовать представления `pg_stat_subscription` и `pg_stat_subscription_stats`.
- Конфликты восстановления могут возникать при выполнении запросов на репликах.
- Для отслеживания конфликтов восстановления можно использовать представления `pg_stat_database` и `pg_stat_database_conflicts`.

Глава 8
Очистка

В этой главе мы рассмотрим:

- что такое очистка и зачем она нужна;
- живые и мертвые строки;
- фоновый процесс автоочистки;
- при каких условиях и как часто выполняется очистка;
- как посмотреть очередь таблиц на обработку;
- счетчик транзакций и предотвращение ошибок, связанных с его зацикливанием;
- транзакционный горизонт;
- как определить количество доступных идентификаторов транзакций;
- эффект раздувания;
- как определить, насколько раздулись таблицы и индексы;
- способы отслеживания активных процессов очистки.

В предыдущих главах рассмотрена часть фоновых процессов СУБД, в этой продолжено их рассмотрение. Глава посвящена процессу очистки, более известной как *очистка* (vacuum). Далее в тексте я буду использовать термин «очистка», подразумевая оба его варианта, как ручной, так и автоматический; в контексте автоматической очистки будет использоваться термин «автоочистка».

Очистка представляет собой реализацию сборщика мусора (garbage collector) и является важной частью СУБД. Ее эффективная работа напрямую влияет на производительность. В этой главе в необходимом объеме будет рассмотрен механизм очистки, аспекты его работы, требующие отслеживания и, конечно же, необходимые для мониторинга инструменты СУБД.

8.1. Введение в очистку

Чтобы обеспечить высокую производительность конкурентной работы клиентов при значительных нагрузках, PostgreSQL реализует модель MVCC[1] (Multiversion Concurrency Control). Эта модель минимизирует количество блокировок и позволяет читающим процессам не блокировать пишущие, а пишущим — читающие.

[1] momjian.us/main/writings/pgsql/mvcc.pdf

При выполнении запросов и транзакций СУБД оперирует *снимками* (snapshot). Снимок определяет видимое для транзакции состояние данных и включает только зафиксированные на момент создания снимка изменения, а момент создания выбирается исходя из установленного *уровня изоляции* (isolation level) транзакций. Конкурентное выполнение транзакций (и запросов) предполагает существование нескольких снимков, в рамках которых может осуществляться не только чтение, но и изменение данных.

В модели MVCC PostgreSQL строки не изменяются на месте (in-place); вместо этого различные операции выполняют действия с версиями строк:

- операция `INSERT` вставляет первую версию новой строки;
- операция `UPDATE` вставляет новую версию строки, а предыдущую версию отмечает как удаленную;
- операция `DELETE` отмечает версию строки как удаленную.

Таким образом, для одной и той же строки может существовать несколько версий — одна актуальная, она же живая (live), и, возможно, несколько мертвых (dead). Строки, помеченные как удаленные, продолжают некоторое время храниться в тех же страницах и файлах данных, поскольку все еще могут потребоваться другим активным транзакциям (то есть входят в снимки, используемые этими транзакциями). В случае высокой активности на запись может появляться все больше и больше версий строк, помеченных удаленными, что сказывается на размерах таблиц и индексов. С постепенным завершением транзакций мертвые строки в конечном счете становятся не нужными ни одной из транзакций, и их можно безопасно удалить, освободив место для новых строк. Более подробно о работе механизма MVCC можно прочитать в соответствующей главе книги The Internals of PostgreSQL[1].

Постепенно становясь ненужными, мертвые версии строк продолжают занимать пространство в страницах, что приводит к избыточным операциям ввода-вывода и расходу ресурсов, которого хотелось бы избежать. Так перед СУБД возникает необходимость в удалении устаревших версий строк, которые больше никому не нужны. Этой задачей и занимается очистка. В ранних версиях PostgreSQL очистка была реализована в виде отдельной команды `VACUUM`, которую администратор был должен запускать вручную или с помощью внешнего планировщика задач (обычно cron). Помимо очистки, у команды `VACUUM` есть несколько дополнительных функций: сбор статистики планировщика, полная блокирующая очистка, заморозка строк и др. Больше информации о работе команды очистки можно узнать из документации[2]. Впоследствии была реализована автоматическая версия очистки в виде фоновых процессов, запускающихся по необходимости.

Первый процесс автоочистки autovacuum launcher поддерживает список баз данных и принимает решение, когда нужно запустить очистку в конкретной базе. Когда в этом возникает необходимость, autovacuum launcher отправляет сигнал головному процессу postmaster, и тот,

[1] www.interdb.jp/pg/pgsql05.html
[2] postgrespro.ru/docs/postgrespro/15/sql-vacuum

в свою очередь, запускает рабочий процесс autovacuum worker. Рабочий процесс очистки узнает имя целевой базы, подключается к ней и строит список таблиц, подлежащих обработке. Обрабатывая по очереди таблицы из списка, рабочий процесс сканирует их на предмет устаревших версий строк, не нужных ни одной из активных транзакций, и освобождает место как в самих таблицах, так и в индексах. Освобожденное пространство может использоваться для вставки новых строк. Если в ходе очистки свободное пространство образовалось в конце файла таблицы, то очистка может отсечь хвостовые пустые страницы (truncate heap) и таким образом уменьшить файл таблицы.

При некоторых видах рабочих нагрузок возникают случаи, когда в таблицах и индексах содержится относительно небольшое количество живых строк, а все остальное пространство освобождено и при этом не используется. Если при этом участки пустого пространства физически находятся в разных частях файла, то очистка не может высвободить эти участки и уменьшить сам файл. Процесс образования таких участков и их постепенное увеличение называют *раздуванием* (bloat). В самом безобидном случае следствием эффекта раздувания является впустую занимаемое пространство, а в худшем — снижение производительности ввода-вывода из-за избыточных операций, нехватки буферного кеша и снижения эффективности индексного доступа из-за образования лишних уровней у индексов. Поэтому при эксплуатации СУБД необходимо отслеживать излишнее раздувание таблиц и индексов и сокращать его.

Подводя некоторый итог, можно сказать, что с точки зрения эксплуатации администратор БД должен отслеживать эффективность работы автоочистки, создаваемый ею уровень нагрузки и долю мертвых строк в таблицах, и при необходимости корректировать настройки очистки и уменьшать образовавшееся раздувание.

В современных реалиях с ростом производительности систем хранения важность отслеживания уровня нагрузки от очистки сильно уменьшилась. Если раньше, в эпоху HDD-дисков, это было актуально, поскольку очистка могла значительно влиять на производительность запросов, то сейчас, при использовании SSD- и NVMe-носителей, это негативное влияние сильно уменьшилось или вообще сошло на нет.

8.2. Особенности очистки на практике

Чтобы получить представление о том, что именно следует отслеживать в работе очистки, давайте рассмотрим практические особенности ее работы. Очистка запускается только при наступлении определенных событий, и в идеале не должна запаздывать. Но в случае запаздывания важно представлять себе объем накопившейся работы.

Когда выполняется автоочистка?

Автоочистка обрабатывает таблицу, когда объем мертвых строк в ней превышает некоторое допустимое количество. Устаревшие версии строк появляются из-за обновлений и удалений;

при выполнении этих операций количество устаревших версий постоянно отслеживается, и, как только оно превысит допустимое значение, таблица добавляется в список на обработку. Предельное значение вычисляется на основании следующей информации:

- `pg_class.reltuples` — общее количество строк в таблице согласно последнему выполнению очистки;
- `pg_stat_all_tables.n_dead_tup` — общее количество мертвых строк, которое обновляется при операциях обновления и удаления;
- *autovacuum_vacuum_threshold* — параметр, определяющий базовое пороговое значение, то есть минимальное количество мертвых строк в таблице, необходимое для ее очистки (по умолчанию 50 строк);
- *autovacuum_vacuum_scale_factor* — параметр, определяющий долю общего числа строк, на которую увеличивается базовое пороговое значение *autovacuum_vacuum_threshold* (по умолчанию 0,2, что означает 20 % от `pg_class.reltuples`).

Согласно значениям по умолчанию, для срабатывания автоочистки количество мертвых строк в таблице должно превышать 50 + 20 % от общего числа строк. Значение параметра *autovacuum_vacuum_scale_factor* `= 0.2` часто воспринимается как консервативное и в современных конфигурациях принято указывать меньшее значение в диапазоне от 0,01 до 0,05, тем самым вызывая автоочистку чаще и при гораздо меньшем количестве мертвых строк. Для очень больших таблиц (с большими индексами) даже небольшое значение параметра *autovacuum_vacuum_scale_factor* при переводе в строки может оказаться большим, и рабочей памяти может не хватить для размещения всех указателей на мертвые строки. Это очень сильно повышает стоимость процедуры очистки, так как при этом приходится полностью сканировать и очищать индексы несколько раз. В таких случаях для отдельных таблиц можно устанавливать индивидуальные параметры очистки с помощью параметров хранения[1].

Теперь, зная условие срабатывания автоочистки, можно написать запрос для определения таблиц, у которых количество мертвых строк превышает допустимое значение. Запрос выводит список пользовательских таблиц с некоторой статистикой:

- `relation` — имя таблицы. При желании можно вывести также имя схемы;
- `reltuples` — приблизительное количество строк в таблице. Это значение обновляется каждый раз после выполнения автоочистки, команд `VACUUM`, `ANALYZE` и некоторых DDL-команд вроде `CREATE INDEX`;
- `live_tup` — количество живых строк в таблице;
- `dead_tup` — количество устаревших, мертвых строк в таблице;
- `boundary` — допустимое количество мертвых версий строк. Когда количество мертвых версий превысит это значение, таблица должна обработаться автоочисткой;
- `av_cnt` — общее количество выполненных операций автоочистки на таблице;
- `since_last_av` — время, прошедшее с момента выполнения последней автоочистки;

[1] postgrespro.ru/docs/postgresql/current/sql-createtable#SQL-CREATETABLE-STORAGE-PARAMETERS

- av_need — признак того, что количество мертвых версий превышает допустимое значение и таблице требуется обработка;
- dead_ratio — процентное отношение количества мертвых версий к общему числу строк в таблице.

```
# SELECT
    *,
    av.dead_tup > av.boundary AS av_need,
    CASE WHEN reltuples > 0
      THEN round(100.0 * av.dead_tup / reltuples)
      ELSE 0
    END AS n_dead_ratio
  FROM
    (SELECT
       c.relname AS relation,
       c.reltuples AS reltuples,
       pg_stat_get_live_tuples(c.oid) AS live_tup,
       pg_stat_get_dead_tuples(c.oid) AS dead_tup,
       round(current_setting('autovacuum_vacuum_threshold')::integer +
             current_setting('autovacuum_vacuum_scale_factor')::numeric * c.reltuples
       ) AS boundary,
       pg_stat_get_autovacuum_count(c.oid) AS av_cnt,
       now() - pg_stat_get_last_autovacuum_time(c.oid) AS since_last_av
     FROM pg_class c
     LEFT JOIN pg_namespace n ON (n.oid = c.relnamespace)
     WHERE c.relkind = 'r'
     AND n.nspname NOT IN ('pg_catalog', 'information_schema')
    ) AS av
  ORDER BY av_need, dead_tup DESC;
    relation      |  reltuples   | live_tup | dead_tup | boundary | av_cnt |  since_last_av  | av_need | dead_ratio
------------------+--------------+----------+----------+----------+--------+-----------------+---------+------------
 pgbench_accounts | 1.998889e+06 |  1998889 |   161911 |   399828 |      0 |                 | f       |          8
 pgbench_history  |       957729 |   973640 |        0 |   191596 |    114 | 01:02:37.853934 | f       |          0
 pgbench_tellers  |          200 |      200 |     1361 |       90 |   2849 | 00:00:36.65028  | t       |        680
 pgbench_branches |           20 |       20 |     1160 |       54 |   2849 | 00:00:36.660004 | t       |       5800
```

По результатам запроса можно сделать несколько наблюдений:

- есть две таблицы (pgbench_tellers и pgbench_branches), которым требуется очистка: количество мертвых версий в них существенно превышает допустимое значение, о чем говорит признак av_need = t;
- значения av_cnt и since_last_av показывают, что две эти таблицы регулярно очищаются, но при сложившейся рабочей нагрузке до следующего запуска очистки в таблицах успевает накопиться большое количество мертвых версий (см. dead_ratio);
- для оценки значения since_last_av важно знать значение параметра *autovacuum_naptime*[1] (по умолчанию одна минута). Этот параметр определяет интервал, с которым процесс

[1] postgrespro.ru/docs/postgresql/current/runtime-config-autovacuum#GUC-AUTOVACUUM-NAPTIME

autovacuum launcher отправляет сигнал на запуск рабочих процессов очистки в каждую базу данных. В выводе запроса есть две таблицы с признаком необходимости очистки. Значения since_last_av для этих двух таблиц укладываются в минуту: это значит, что таблицы хоть и должны быть обработаны, но задержки при этом нет. И наоборот, если бы при наличии признака очистки значение since_last_av превышало значение *autovacuum_naptime*, это указывало бы на то, что очистка опаздывает;

- есть таблица с большим количеством строк, которая при этом ни разу не подвергалась обработке (pgbench_accounts), поскольку объем мертвых строк ни разу не достигал порогового значения.

Можно сделать вывод: в тестовом окружении не так много таблиц, и автоочистка справляется со своей работой.

Дополнительно стоит отметить, как вычисляется допустимое значение (boundary). За основу берутся общие параметры конфигурации, однако СУБД предоставляет возможность указывать эти параметры индивидуально для отдельных таблиц с помощью так называемых параметров хранения (storage parameter)[1]. Они имеют приоритет перед общими параметрами конфигурации, и их следует учитывать при расчете порогового значения. В тестовом окружении у таблиц нет явно определенных параметров хранения, поэтому запрос их не учитывает.

Статистика выполнения очистки

В ранее рассмотренном запросе была использована функция pg_stat_get_autovacuum_count, которая возвращает количество обработок таблицы автоочисткой. Эта и еще несколько похожих функций используются в представлении pg_stat_all_tables, на котором также основаны представления pg_stat_user_tables и pg_stat_sys_tables. На основе этих функций в представлениях отображаются несколько полей, указывающих на работу как автоматической, так и ручной очистки:

- last_vacuum — время, когда таблица в последний раз очищалась командой VACUUM;
- last_autovacuum — время, когда таблица в последний раз была обработана автоочисткой;
- vacuum_count — общее количество обработок таблицы командой VACUUM;
- autovacuum_count — общее количество обработок таблицы автоочисткой.

Сбор статистики для планировщика

Набор похожих полей есть и для операций сбора статистики планировщика. Сбор статистики может выполняться либо с помощью вызова команды ANALYZE, либо как дополнительная операция при вызове команды VACUUM (с указанием параметра ANALYZE), либо рабочим процессом автоочистки.

[1] postgrespro.ru/docs/postgresql/current/sql-createtable#SQL-CREATETABLE-STORAGE-PARAMETERS

Следующий запрос позволяет отслеживать регулярность очистки:

```
# SELECT
    schemaname ||'.'|| relname AS relation,
    vacuum_count AS vacuum,
    now() - last_vacuum AS since_last_vacuum,
    autovacuum_count AS autovacuum,
    now() - last_autovacuum AS since_last_autovacuum
  FROM pg_stat_user_tables;
        relation         | vacuum | since_last_vacuum | autovacuum | since_last_autovacuum
-------------------------+--------+-------------------+------------+-----------------------
 public.pgbench_tellers  |      5 | 07:54:41.804464   |       2869 | 00:00:29.026952
 public.pgbench_branches |      5 | 07:54:41.805567   |       2869 | 00:00:29.035908
 public.pgbench_history  |      0 |                   |        114 | 01:22:30.721539
 public.pgbench_accounts |      0 |                   |          0 |
```

В выводе запроса отражена статистика и по ручным, и по автоматическим очисткам. Основная работа по очистке выполняется автоочисткой, и изредка выполняется команда `VACUUM` (специфика рабочей нагрузки).

Собрав статистику выполнения операций очистки со всех БД, можно получить общую картину по всему экземпляру и вывести ее в мониторинг (рис. 8.1):

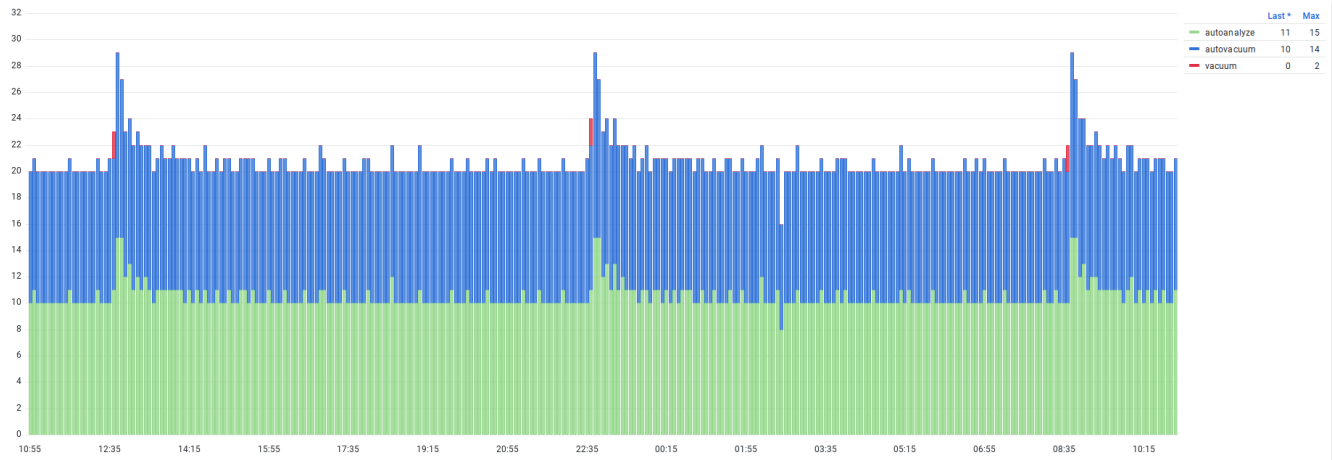


Рис. 8.1. Частота выполнения операций очистки и анализа

График на рис. 8.1 показывает количество операций очистки и анализа, выполненных в течение пяти минут. За показанные сутки количество операций держится на одном уровне без больших колебаний, однако можно заметить некоторую цикличность с интервалом 10 часов. В случае корректировок настроек автоочистки такой график помогает понять, как сделанные изменения отразились на ее работе. Значительные изменения в рабочей нагрузке, особенно связанные с объемом записи, также могут повлиять на частоту выполнения служебных операций, и последствия этих изменений также будут заметны на графике.

8.3. Счетчик транзакций и предотвращение ошибок, связанных с его зацикливанием

У каждой транзакции, будь это даже одиночная команда, есть уникальный идентификатор. Это может быть виртуальный идентификатор транзакции (virtualxid), а если транзакция вносит изменения, ей присваивается постоянный 32-битный идентификатор (transaction id, xid, или txid). При выполнении команд на изменение данных (вставка, обновление, удаление) в версиях строк в служебных полях `xmin` и `xmax`[1] записываются идентификаторы транзакций: в `xmin` — создавшей строку, а в `xmax` — удалившей строку. Правила видимости строк опираются на эту информацию. Например, версия строки, созданная только что зафиксированной транзакцией, не попадет в уже начавшуюся выборку благодаря тому, что `xmin` этой версии будет превышать максимально видимый в выборке номер[2].

При постоянной нагрузке СУБД выделяет все новые и новые идентификаторы транзакций. Их диапазон ограничен 32 битами и соответствует значениям от 0 до 2^{32-1} (около 4,2 миллиарда). Получается, что в какой-то момент диапазон номеров может израсходоваться и счетчик транзакций должен будет начать отсчет с нуля. Но в таком случае все строки прошлых транзакций оказались бы в будущем — ведь их идентификаторы превысили бы текущее значение счетчика. Это привело бы к внезапной потере всех данных, которые, физически оставаясь в страницах, перестали бы удовлетворять правилам видимости и стали бы недоступными. Чтобы исключить такое поведение, диапазон идентификаторов зациклен и образует круг, разделенный на две части: для любого идентификатора следующие два миллиарда идентификаторов считаются «старше» его и находятся в будущем, а предыдущие два миллиарда — «младше» и находятся в прошлом. Строка, созданная в какой-либо транзакции, для последующих двух миллиардов транзакций находится в прошлом, но, если ничего не предпринять, для следующих транзакций окажется в будущем. Поэтому еще до того, как будет достигнута граница в два миллиарда, СУБД находит такие старые строки и ставит на них признак *заморозки*. По правилам видимости замороженные (frozen) строки считаются настолько старыми, что уже нет необходимости проверять их значение `xmin`. Такие строки являются безусловно видимыми — до тех пор, пока строка не будет обновлена или удалена, что приведет к установке значения `xmax`.

FrozenTransactionId

В версиях PostgreSQL до 9.4 заморозка строк реализована заменой `xmin` строки на служебный идентификатор `FrozenTransactionId` (равный двум). В последующих версиях признак заморозки устанавливается как битовый флаг в другое служебное поле, а `xmin` строки сохраняется.

Необходимость заморозки отдельно взятой строки определяется возрастом транзакции с идентификатором `xmin`. Возраст транзакции определяется количеством транзакций, нача-

[1] postgrespro.ru/docs/postgresql/current/ddl-system-columns
[2] postgrespro.ru/education/books/internals

тых в системе после данной, до текущего момента. Выполнять заморозку строк с малым возрастом может быть невыгодно, поскольку строка может измениться и выполненная работа пропадет даром. Откладывать заморозку надолго тоже нельзя: при интенсивной рабочей нагрузке может скопиться большое количество незамороженных строк и их обработка потребует значительных ресурсов и времени. Заморозка обычно выполняется автоматически рабочими процессами очистки (но может быть выполнена и в ручном режиме командой VACUUM FREEZE), и, следовательно, для своевременной заморозки автоочистка также должна запускаться своевременно и без задержек. В зависимости от конфигурации СУБД или рабочей нагрузки могут возникать ситуации, которые откладывают работу автоочистки или даже мешают ей. Игнорирование таких ситуаций в течение продолжительного времени может привести к снижению производительности, а в худшем случае — к аварийной остановке СУБД. В качестве примеров того, что может откладывать очистку, можно привести:

- выключенную автоочистку (*autovacuum* = off) как на уровне общей конфигурации, так и на уровне отдельных таблиц с помощью параметров хранения;
- продолжительные транзакции (это наиболее частая причина, по которой автоочистка откладывает удаление устаревших версий строк);
- забытые незавершенные подготовленные (prepared) транзакции;
- длительные выгрузки данных с помощью pg_dump или команды COPY, которые также используют механизм транзакций;
- регулярные DDL-операции, которые могут отменять автоочистку, из-за чего часть работы откладывается на потом;
- неактивные слоты репликации.

При нормальной эксплуатации автоочистка всегда должна быть включена, а в рабочей нагрузке не должно возникать продолжительных или бездействующих операций.

Горизонт заморозки. Каждая таблица имеет атрибут pg_class.relfrozenxid, который содержит идентификатор транзакции, создавшей самую позднюю из незамороженных версий строк этой таблицы. Возраст транзакции relfrozenxid является так называемым *горизонтом заморозки отдельной таблицы* — все версии строк с более старшими идентификаторами уже гарантированно заморожены. С помощью таблицы pg_class можно как получить само значение relfrozenxid, так и вычислить горизонт заморозки (поле horizon):

```
# SELECT relname, relfrozenxid, age(relfrozenxid) AS horizon,
    pg_stat_get_autovacuum_count(c.oid) +
    pg_stat_get_vacuum_count(c.oid) AS maint_count,
    now() - coalesce(
      pg_stat_get_last_autovacuum_time(c.oid),
      pg_stat_get_last_vacuum_time(c.oid)
    ) AS since_last_maint
  FROM pg_class c
  WHERE c.relkind = 'r' AND c.relname ~'pgbench'
  ORDER BY age(relfrozenxid) DESC;
```

```
     relname      | relfrozenxid |  horizon  | maint_count | since_last_maint
------------------+--------------+-----------+-------------+------------------
 pgbench_accounts |      3065607 | 193389917 |           0 |
 pgbench_history  |    186844463 |   9611061 |          88 | 00:01:58.716975
 pgbench_tellers  |    194117385 |   2338139 |        1309 | 00:00:58.733513
 pgbench_branches |    195212688 |   1242836 |        1277 | 00:00:58.741392
```

Диапазон идентификаторов транзакций в пределах горизонта заморозки определяет количество тех идентификаторов, которые еще не могут быть повторно использованы для будущих транзакций из-за оставшихся незамороженными версий строк. В таблицах, для которых регулярно выполняется очистка, происходят и своевременная заморозка строк, и обслуживание горизонта, то есть продвижение вперед значения `relfrozenxid`. Вывод запроса показывает, что для трех таблиц регулярно выполняются очистка и продвижение `relfrozenxid`, а для `pgbench_accounts` очистка ни разу не выполнялась, поэтому `relfrozenxid` постепенно уходит в прошлое и горизонт заморозки растет.

Атрибут `relminmxid` и мультитранзакции

Помимо `relfrozenxid` у таблиц есть схожий атрибут `relminmxid`, который указывает на самый старый задействованный идентификатор мультитранзакции в этой таблице. Мультитранзакции представляют собой группы обычных транзакций, которым потребовалась одновременная разделяемая блокировка одной строки. Такие транзакции также имеют 32-битные идентификаторы, которые записываются в поле `xmax` версий строк, и для них также требуются аналог заморозки и продвижение `relminmxid`. Ниже при упоминании `relfrozenxid` соответствующий контекст можно распространить и на мультитранзакции, хоть это и не будет упоминаться явно.

При очистке СУБД оценивает горизонт заморозки и выбирает подходящий режим работы. В обычном режиме очистка замораживает версии строк, возраст `xmin` которых превышает значение *vacuum_freeze_min_age* (по умолчанию 50 млн). Если горизонт заморозки таблицы превышает значение *vacuum_freeze_table_age* (по умолчанию 150 млн), очистка выполняется в агрессивном режиме с полным сканированием всех страниц, чтобы заморозить версии строк и на тех страницах, которые в обычном режиме пропускаются из-за карты видимости. Если же возраст горизонта заморозки таблицы превышает значение *autovacuum_freeze_max_age* (по умолчанию 200 млн), то очистка с заморозкой запускается принудительно, даже когда таблица не требует очистки или автоочистка выключена (глобально параметром *autovacuum* или параметрами хранения для отдельных таблиц). Это в том числе гарантирует, что версии строк будут заморожены в таблицах, строки в которых не меняются, например в архивных секциях.

Как отмечено выше, в нашем примере есть одна таблица, `pgbench_accounts`, которая ни разу не очищалась и горизонт заморозки которой больше остальных. Для эксперимента можно выполнить очистку с принудительной заморозкой (`VACUUM FREEZE`), после которой значение `relfrozenxid` продвинется вперед, а горизонт уменьшится.

```
     relname      | relfrozenxid | horizon | maint_count | since_last_maint
------------------+--------------+---------+-------------+------------------
 pgbench_accounts |    196461352 |    1059 |           1 | 00:00:33.415394
 pgbench_branches |    196461367 |    1044 |        1281 | 00:01:18.82579
 pgbench_history  |    196461367 |    1044 |          91 | 00:01:18.803942
 pgbench_tellers  |    196461398 |    1013 |        1314 | 00:00:18.804065
```

Если рассматривать все таблицы в отдельной базе данных, то их значения relfrozenxid, как правило, различаются; среди всех таблиц найдется таблица с самым большим горизонтом (самым старым значением relfrozenxid). Горизонт заморозки этой таблицы определяет *горизонт заморозки базы данных* и хранится в pg_database.datfrozenxid — все версии строк с более старыми идентификаторами в любой таблице этой базы данных гарантированно заморожены. Самое старое значение pg_database.datfrozenxid в кластере баз данных определяет *горизонт заморозки всего кластера*. Продвижение вперед этого горизонта позволяет высвободить идентификаторы транзакций для их повторного использования.

```
# SELECT datname, datfrozenxid, age(datfrozenxid) AS horizon FROM pg_database;
  datname  | datfrozenxid |  horizon
-----------+--------------+-----------
 postgres  |          717 | 196462899
 pgbench   |    196461352 |      2264
 template1 |          717 | 196462899
 template0 |          717 | 196462899
```

Значение datfrozenxid продвинуто только у базы pgbench (для нее недавно была выполнена очистка с заморозкой). В остальных базах значение сохранилось еще со времен инициализации кластера: в них нет никакой активности, нет необходимости выполнять очистку, следовательно, и заморозка в них не происходит. Но, как только горизонты заморозки этих баз превысят значение *autovacuum_freeze_max_age* (по умолчанию 200 млн), в них будет запущена принудительная очистка с заморозкой. Осталось ждать не так долго. Очистка скоро выполнится, и запрос покажет продвижение datfrozenxid и уменьшение горизонта:

```
  datname  | datfrozenxid |  horizon
-----------+--------------+-----------
 postgres  |          717 | 200000467
 pgbench   |    196461352 |   3539832
 template1 |          717 | 200000467
 template0 |          717 | 200000467

  datname  | datfrozenxid |  horizon
-----------+--------------+-----------
 postgres  |    200001187 |        79
 pgbench   |    196461352 |   3539914
 template1 |    200001187 |        79
 template0 |    200001187 |        79
```

Для базы pgbench все осталось как прежде, поскольку в ней заморозка была выполнена недавно и повторная пока не требуется.

Запас идентификаторов транзакций. При нормальной эксплуатации предполагается, что очистка и заморозка выполняются своевременно: идентификаторы datfrozenxid продвигаются вперед, и у СУБД есть большой запас идентификаторов для будущих транзакций. В реальности в рабочей нагрузке могут возникать аномалии, из-за которых очистка (и заморозка) может откладываться. Как правило, наиболее частой причиной пропуска строк при очистке являются долгие транзакции: они удерживают горизонт видимости других транзакций, что исключает очистку строк в пределах этого горизонта. В худшем случае, когда начнет откладываться заморозка, начнет исчерпываться и запас идентификаторов. Поэтому очень важно отслеживать продолжительность транзакций, устранять источники появления долгих транзакций и использовать тайм-ауты для их принудительного завершения.

С точки зрения мониторинга нас интересует запас доступных идентификаторов транзакций. Его можно вычислить либо на основе горизонта заморозки кластера с помощью поля pg_database.datfrozenxid, либо на основе горизонта видимости базы данных, который определяется наибольшим горизонтом видимости среди активных транзакций в этой базе — строки за горизонтом видимости уже не нужны ни одной из активных транзакций и в будущем могут быть заморожены (или изменены, что повлечет изменение xmax). Второй способ дает более оптимистичную оценку, поскольку в запас идентификаторов включается диапазон между горизонтом видимости базы данных и горизонтом заморозки кластера, а этот диапазон может быть очень большим и включать в себя несколько десятков миллионов идентификаторов. В поисках наибольшего горизонта видимости среди активных транзакций нам могут помочь следующие поля:

- pg_stat_activity.backend_xid — идентификатор транзакции, выданный сеансу;
- pg_prepared_xacts.transaction — идентификатор подготовленной транзакции;
- pg_replication_slots.xmin — идентификатор транзакции, определяющий горизонт видимости, удерживаемый данным слотом;

Первые два поля указывают на активные транзакции, выполняющиеся на текущем узле. Если используются реплики и включена обратная связь (*hot_standby_feedback* = on), оставшееся поле определяет горизонт, удерживаемый транзакциями на реплике с помощью слота репликации. Если репликация не использует слот, эта же информация будет доступна в поле pg_stat_activity.backend_xid соответствующего процесса walsender. С помощью обратной связи администратор может отслеживать долгую активность на репликах со стороны основного узла, но при этом обратная связь откладывает работу очистки, приводя ко всем сопутствующим негативным последствиям. Поэтому решение об использовании обратной связи должно приниматься взвешенно.

Давайте рассмотрим оба способа вычисления запаса идентификаторов транзакций.

Первый способ заключается в использовании горизонта заморозки кластера на основе полей datfrozenxid и datminmxid: возраст старшего из этих идентификаторов нужно вычесть из половины всего доступного диапазона идентификаторов (двух миллиардов).

```
# SELECT
    max_horizon,
    2147483647 - max_horizon AS available_xids
  FROM
    (
      SELECT greatest(
          max(age(datfrozenxid)),
          max(mxid_age(datminmxid))
      ) AS max_horizon
      FROM pg_database
    ) AS d;
 max_horizon | available_xids
-------------+----------------
     3547151 |     2143936496
```

В приведенном выводе горизонт достиг 3,5 млн и имеется очень большой запас идентификаторов (поле available_xids), превышающий два миллиарда.

Второй способ определения запаса идентификаторов опирается на вычисление горизонта видимости базы данных. В этом случае нужно проверить все источники активности, найти самый долгий, вычислить его возраст и, как и в первом способе, вычесть из половины всего доступного диапазона идентификаторов. Дополнительно можно вывести и идентификатор транзакции, которая удерживает горизонт видимости:

```
# SELECT pid, src, age AS active_horizon,
    2147483647 - coalesce(age, 0) AS available_xids
  FROM
    (
      SELECT pid, 'pg_stat_activity', age(backend_xid)
        FROM pg_stat_activity WHERE pid != pg_backend_pid()
      UNION
      SELECT NULL, 'pg_prepared_xacts', age(transaction)
        FROM pg_prepared_xacts
      UNION
      SELECT active_pid, 'pg_replication_slots',
          greatest(age(xmin), age(catalog_xmin))
        FROM pg_replication_slots
    ) AS x(pid, src, age)
  WHERE age IS NOT NULL
  ORDER BY age DESC LIMIT 1;
   pid  |       src        | active_horizon |  available_xids
--------+------------------+----------------+-----------------
  35643 | pg_stat_activity |              1 |      2147483646
```

В нормальной ситуации вывод запроса будет пустой, однако если использовать \watch с коротким интервалом (например, одна десятая секунды), то можно будет увидеть, как быстро появляются и быстро исчезают строки с очень коротким горизонтом — например, как в выводе выше. В данном примере горизонт равен единице: в тестовом окружении нет долгих транзакций, которые бы надолго удерживали горизонт видимости.

Однако можно открыть еще один сеанс, обновить в новой транзакции произвольную строку в pgbench_accounts и некоторое время просто не завершать начатую транзакцию:

```
# BEGIN;
# UPDATE pgbench_accounts SET abalance = abalance + 1 WHERE aid = 1;
```

Теперь, если повторить запрос еще несколько раз, будет видно, что картина изменилась:

```
  pid   |       src        | active_horizon |  available_xids
--------+------------------+----------------+-----------------
  35705 | pg_stat_activity |            295 |     2147483352

  pid   |       src        | active_horizon |  available_xids
--------+------------------+----------------+-----------------
  35705 | pg_stat_activity |            648 |     2147482999

  pid   |       src        | active_horizon |  available_xids
--------+------------------+----------------+-----------------
  35705 | pg_stat_activity |            990 |     2147482657
```

Из-за активности в базе данных счетчик транзакций увеличивается, а из-за открытой транзакции горизонт начал расти (идентификатор, определяющий снимок открытой транзакции, уходит в прошлое относительно номера текущей транзакции, в которой выполняется наш запрос). Количество доступных идентификаторов тоже начало уменьшаться, хотя это не так заметно. Самое главное — чем дольше транзакция во втором сеансе будет находиться в открытом состоянии, тем больше будет увеличиваться горизонт видимости, в пределах которого автоочистка не сможет удалять устаревшие версии строк (что будет приводить к раздуванию) и замораживать старые версии строк (что со временем может привести к реальному уменьшению количества доступных идентификаторов). После фиксации или отката транзакции горизонт снова уменьшится до нуля.

Независимо от того, какой способ вычисления запаса идентификаторов используется, в целях мониторинга его удобно выводить в виде графика. На рис. 8.2 показана ситуация, когда возраст datfrozenxid одной из баз превысил значение *autovacuum_freeze_max_age*, в результате чего была выполнена принудительная автоочистка с заморозкой. Для большей наглядности взят интервал в несколько дней, поскольку в тестовом окружении запас идентификаторов уменьшается относительно медленно. После выполнения заморозки горизонт уменьшился, а запас идентификаторов, соответственно, увеличился. В производственных средах с большим объемом транзакционной активности такое продвижение горизонта происходит регулярно и довольно часто.

Предотвращение зацикливания. В дополнение к механизму заморозки в СУБД встроены различные проверки, задача которых — уведомлять администратора СУБД об опасном увеличении горизонта заморозки и исключить ошибки, связанные с зацикливанием счетчика транзакций. Как только принудительная очистка не сможет нормально выполнять свою работу,

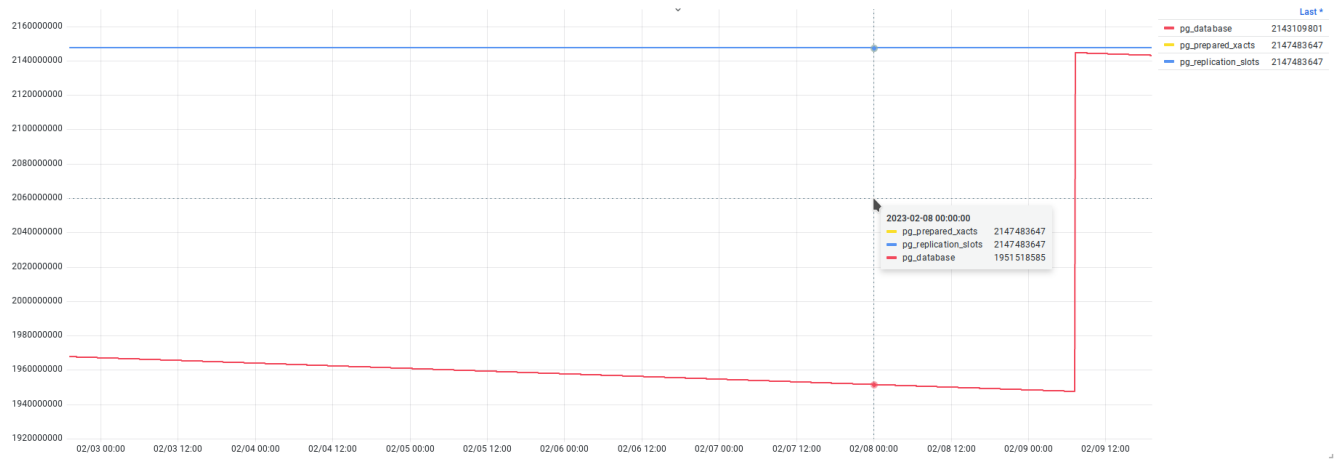


Рис. 8.2. Количество доступных идентификаторов

в журнале сообщений начнут появляться уведомления о том, что самый старый используемый идентификатор транзакции находится далеко в прошлом:

```
WARNING: oldest xmin is far in the past
HINT: Close open transactions soon to avoid wraparound problems.
      You might also need to commit or roll back old prepared transactions,
      or drop stale replication slots.
```

В качестве подсказки СУБД предлагает закрыть открытые транзакции, подтвердить или откатить старые подготовленные транзакции или удалить неактивные слоты репликации. При дальнейшем ухудшении ситуации очистка начинает выполняться в особом режиме (failsafe), при котором для ускорения заморозки пропускается обработка индексов. Этот режим включается при превышении возрастом `relfrozenxid` значения *vacuum_failsafe_age*:

```
WARNING: bypassing nonessential maintenance of table "postgres.pg_catalog.
      pg_statistic" as a failsafe after 0 index scans
DETAIL: The table's relfrozenxid or relminmxid is too far in the past.
HINT: Consider increasing configuration parameter "maintenance_work_mem" or
      "autovacuum_work_mem". You might also need to consider other ways for
      VACUUM to keep up with the allocation of transaction IDs.
```

Если продолжать игнорировать сообщения, СУБД будет более прямолинейна и начнет сообщать о том, что вполне конкретные базы данных должны быть обработаны очисткой:

```
WARNING: database "postgres" must be vacuumed within 5715164 transactions
HINT: To avoid a database shutdown, execute a database-wide VACUUM in that
      database. You might also need to commit or roll back old prepared
      transactions, or drop stale replication slots.
```

Примерно за три миллиона транзакций до исчерпания идентификаторов СУБД переходит в режим только чтения. Любая попытка выполнить запись или изменение данных завершится ошибкой.

```
ERROR: database is not accepting commands to avoid wraparound data loss in
       database "postgres"
HINT: Stop the postmaster and vacuum that database in single-user mode.
       You might also need to commit or roll back old prepared transactions, or
       drop stale replication slots.
```

Такая ситуация расценивается как опасная и переход СУБД в аварийный режим исключает возможность продолжить работу и повредить данные. Для восстановления и продолжения нормальной работы администратору придется перезапустить СУБД в особом, однопользовательском режиме (single-user mode), выполнить очистку в тех БД, где это необходимо, после чего перезапустить СУБД в обычном режиме.

8.4. Раздувание таблиц и индексов

Устаревшие версии строк появляются в результате операций обновления или удаления. При нормальной работе занимаемое такими версиями место оперативно освобождается очисткой и используется для вставки новых версий. Однако если устаревших версий накопилось слишком много и автоочистка долгое время не имела возможности вычищать их, то после очистки в файлах таблицы и индексов образуются большие свободные участки, которые, возможно, не израсходуются в дальнейшем и будут просто занимать место на диске. В результате дисковое пространство используется неэффективно. Стоит отдельно отметить, что это относится к «пустотам», находящимся в начале или середине файла: если очистка освобождает место в конце файла, она может попытаться сделать усечение файла.

При продолжительной эксплуатации СУБД таких пустот в таблицах и индексах может образовываться все больше и больше, и происходит так называемое раздувание. Если сложить все эти пустоты вместе, может получиться внушительная цифра. С точки зрения эксплуатации необходимо регулярно проверять таблицы и индексы на предмет раздувания, выполнять их обслуживание и высвобождать неиспользуемое пространство.

Для поиска раздутых объектов наиболее эффективным и точным средством оценки является расширение `pgstattuple`[1]. Однако ценой точности являются большие накладные расходы: расширение читает каждую страницу проверяемого объекта, что требует соответствующего ввода-вывода и нагружает дисковую подсистему. Учитывая, что СУБД не имеет встроенных механизмов ограничения пропускной способности, такая «проверка» может значительно

[1] postgrespro.ru/docs/postgresql/current/pgstattuple

загрузить подсистему хранения и существенно повлиять на производительность конкурентно выполняющихся запросов. Поэтому регулярное и частое использование pgstattuple, например в агентах сбора телеметрии, может быть нежелательно; рекомендуется использовать pgstattuple только при необходимости.

Есть и другие, эвристические способы оценки с использованием данных из системного каталога, учитывающие общее количество строк, их ширину и другие особенности хранения, но, как правило, такие способы менее точны и могут давать значительные отклонения. Выбор инструмента всегда остается за администратором и должен основываться на условиях эксплуатации. Примеры таких запросов можно найти в соответствующем разделе PostgreSQL Wiki[1].

В целях ознакомления расширение pgstattuple установлено в тестовом окружении и с его помощью мы оценим раздувание двух таблиц: pgbench_accounts и маленькой pgbench_branches.

```
# SELECT * FROM pgstattuple('pgbench_accounts');
-[ RECORD 1 ]------+---------------------------
table_len          | 294264832
tuple_count        | 2000000
tuple_len          | 242000000
tuple_percent      | 82.24
dead_tuple_count   | 80131
dead_tuple_len     | 9695851
dead_tuple_percent | 3.29
free_space         | 10538908
free_percent       | 3.58

# SELECT * FROM pgstattuple('pgbench_branches');
-[ RECORD 1 ]------+---------------------------
table_len          | 311296
tuple_count        | 21
tuple_len          | 672
tuple_percent      | 0.22
dead_tuple_count   | 959
dead_tuple_len     | 30688
dead_tuple_percent | 9.86
free_space         | 274084
free_percent       | 88.05
```

Время выполнения запроса зависит от размера таблицы, так как необходимо прочитать каждый ее блок, поэтому оценка pgbench_accounts выполнялась дольше, чем оценка pgbench_branches. В результате анализа таблиц доступна следующая информация, по которой можно оценить раздувание:

- table_len — размер объекта (таблицы или индекса) в байтах;
- tuple_count — общее количество живых версий строк;

[1] wiki.postgresql.org/wiki/Show_database_bloat

- tuple_len — общий размер живых версий строк в байтах;
- tuple_percent — процент живых версий строк от общего числа всех версий;
- dead_tuple_count — общее количество мертвых версий строк;
- dead_tuple_len — общий размер мертвых версий строк в байтах;
- dead_tuple_percent — процент мертвых версий строк от общего числа всех версий;
- free_space — общий размер в байтах свободного пространства внутри файлов таблицы, за счет которого раздулась таблица. Это пространство появляется как результат очистки от мертвых версий строк и доступно для повторного использования только в этой таблице;
- free_percent — доля свободного пространства от общего размера объекта — то же самое раздутие, выраженное в процентах.

Наиболее интересными полями являются free_space и free_percent, поскольку прямо указывают объемы пустого пространства. Если сравнить статистику обеих таблиц, можно отметить, что таблица pgbench_accounts подвержена раздуванию гораздо меньше, чем pgbench_branches, 3,5 % против 88 %, но таблица pgbench_branches занимает всего 311 КБ. При высвобождении свободного места выигрыш может быть небольшим, однако если с этой таблицей связан значительный ввод-вывод, то можно существенно сократить накладные расходы.

В продолжение нашего примера выполним команду полной очистки VACUUM FULL для таблицы pgbench_branches. Это хоть и блокирующая операция, но выполнится она мгновенно, поскольку целевая таблица небольшая и содержит всего 20 строк. После операции полной очистки снова посмотрим статистику pgstattuple:

```
# VACUUM FULL pgbench_branches;
# SELECT * FROM pgstattuple('pgbench_branches');
-[ RECORD 1 ]------+---------------------------
table_len          | 8192
tuple_count        | 20
tuple_len          | 640
tuple_percent      | 7.81
dead_tuple_count   | 88
dead_tuple_len     | 2816
dead_tuple_percent | 34.38
free_space         | 4276
free_percent       | 52.2
```

Высвободилось практически все свободное место, а таблица стала умещаться в одну страницу размером 8 КБ. Однако это всего лишь небольшая таблица в тестовом окружении. В настоящих производственных окружениях значения и результаты будут совершенно иными.

На практике для высвобождения пространства используются различные инструменты. Для индексов подходит использование команды REINDEX CONCURRENTLY, которая в неблокирующем режиме создает новый индекс и затем подменяет им старый. Для обработки таблиц СУБД

предлагает использовать команду VACUUM FULL, однако ее использование полностью блокирует доступ к таблице даже на чтение, что недопустимо в производственных окружениях. В качестве альтернативы можно использовать сторонние инструменты, предлагаемые сообществом, например pg_repack[1], pgcompacttable[2] и pg_squeeze[3]. Все они имеют различные реализацию и принципы работы, но общую цель — устранить раздувание и освободить пространство.

8.5. Отслеживание активных процессов очистки

Продолжительность выполнения очистки может зависеть от разных причин: от параметров конфигурации, размеров таблицы, количества индексов и количества мертвых строк и т. п. При обработке таблиц и индексов очистка создает дополнительную нагрузку на подсистему хранения и может конкурировать за ресурсы ввода-вывода с другими процессами СУБД и влиять на их производительность. Администратору важно не только знать о ходе выполнения операций очистки, но и оценивать их влияние на систему и представлять, как используются ресурсы.

Для отслеживания активных процессов очистки в СУБД есть два инструмента:

- pg_stat_activity — основное представление для отслеживания всей активности в СУБД, которое показывает также и процессы очистки;
- pg_stat_progress_vacuum — представление, отображающее состояние и ход выполнения операций очистки (как ручных, так и автоматических).

Поскольку pg_stat_activity является представлением общего назначения и показывает активность всех процессов СУБД в некотором обобщенном виде, из него нельзя узнать особенности, характерные для конкретных процессов. Для отслеживания процессов очистки следует использовать представление pg_stat_progress_vacuum: оно содержит больше информации о работе именно очистки. Более того, оба представления можно объединить и получить максимум возможной информации.

Представление pg_stat_activity

Представление pg_stat_activity должно быть уже хорошо знакомо по предыдущим главам, поэтому не будем подробно рассматривать его, лишь перечислим интересующие нас поля:

- pid — идентификатор процесса, в рамках которого выполняется операция очистки;

[1] github.com/reorg/pg_repack
[2] github.com/dataegret/pgcompacttable
[3] github.com/cybertec-postgresql/pg_squeeze

- backend_type — по типу процесса можно отфильтровать только те строки, которые относятся к процессам автоочистки. Однако следует быть внимательным: по этому полю нельзя получить очистку, запущенную в обычном сеансе с помощью команды VACUUM: у таких процессов значение backend_type будет равно client backend, как и у всех прочих клиентских процессов;
- datname — имя базы данных, к которой выполнено подключение и в которой выполняется очистка;
- xact_start — время начала транзакции позволяет определить продолжительность выполнения очистки. Здесь тоже есть нюанс: очистка даже одной таблицы включает в себя выполнение нескольких транзакций, следовательно, отметка времени может меняться. Можно было бы использовать backend_start, но рабочий процесс автоочистки за время своего существования обрабатывает множество таблиц, и выделить время, затраченное на обработку одной конкретной, становится невозможным;
- state — состояние процесса;
- wait_event, wait_event_type — события ожидания могут быть полезны на случай, если очистка приостанавливается по какой-либо причине;
- query — текст запроса является определяющим и позволяет взять только те строки, которые относятся к процессам, выполняющим очистку, неважно, автоматическую или запущенную администратором в отдельном сеансе.

При необходимости можно добавить и другие поля. Используя фильтры на основе перечисленных полей, можно исключить все прочие процессы и выбрать только те, что выполняют очистку. С точки зрения постоянного мониторинга нас могут интересовать ответы на следующие вопросы:

- Сколько процессов очистки выполняется в данный момент? Это позволяет понять, не достигнуто ли ограничение количества рабочих процессов (заданное параметром *autovacuum_max_workers*). Если ограничение достигнуто и при этом есть необходимость в запуске дополнительных процессов автоматической очистки, это приведет к тому, что таблицы будут вынуждены ждать очереди на обработку, а мертвые строки будут продолжать накапливаться, увеличивая объем работы для очистки. Рост очереди сдвигает обработку и других таблиц, что постепенно приводит к раздуванию и увеличению размеров таблиц и индексов.
- Сколько времени длится очистка? В идеале, если позволяют ресурсы, очистка должна выполняться быстро. Наличие свободных рабочих процессов в распоряжении СУБД позволяет избегать появления очередей таблиц на обработку. Особенно опасной может считаться ситуация, характерная для «больших» баз данных со смешанной нагрузкой (HTAP, Hybrid Transactional/Analytical Processing). В таких базах могут встречаться как таблицы больших размеров, так и таблицы с большим объемом записи. Первые таблицы могут требовать заморозки, при которой таблица должна быть прочитана полностью, что может занять много времени, а изменения вторых будут постоянно сдвигать счетчик транзакций вперед.

При таком стечении обстоятельств (продолжительная очистка больших таблиц, активная запись и недостаток рабочих процессов очистки) СУБД не будет успевать выполнять обработку и сдвигать счетчик транзакций, что может привести к аварийному переходу в режим только чтения.

Для получения ответов на эти вопросы достаточно использования pg_stat_activity. В тестовом окружении довольно сложно поймать продолжительную очистку, поэтому для удобства демонстрации перепишем таблицу pgbench_accounts:

```
# UPDATE pgbench_accounts SET abalance = abalance + 1;
```

В соседнем сеансе с помощью \watch 1 запустим выполнение запроса к pg_stat_activity. После выполнения запроса на обновление pgbench_accounts в течение минуты (поскольку *autovacuum_naptime* = 1min) запустится рабочий процесс автоочистки, который будет «пойман» запросом к pg_stat_activity.

```
# SELECT
    pid,
    datname,
    now() - xact_start as duration,
    state,
    wait_event_type,
    wait_event,
    query
  FROM pg_stat_activity
  WHERE query ~'^autovacuum' OR query ~*'^vacuum';
-[ RECORD 1 ]------+---------------------------
pid                | 2364758
datname            | pgbench
duration           | 00:00:10.04677
state              | active
wait_event_type    | Timeout
wait_event         | VacuumDelay
query              | autovacuum: VACUUM ANALYZE public.pgbench_accounts
```

В запросе используется условие на текст запроса, который должен начинаться с шаблонов, характерных для рабочих процессов автоочистки и очистки, запущенной в отдельном сеансе. Такой способ более удобен, чем условие на поле backend_type, не позволяющее выделить очистку в обычных сеансах. Одна строка в выводе запроса указывает на то, что запущен всего один рабочий процесс. Подсчитав количество строк функцией count, можно узнать, сколько рабочих процессов очистки запущено в конкретный момент. Это позволяет ответить на первый вопрос — о количестве выполняющихся процессов. Поле duration отображает продолжительность выполнения очистки, что дает ответ на второй вопрос — о длительности. Приведенная в запросе информация о состоянии и событиях ожиданий в мониторинг не собирается и чаще всего нужна только в случаях поиска и устранения проблем.

Полученные ответы можно отобразить в виде графиков. На рис. 8.3 показан график количества процессов очистки. Возникает резонный вопрос: откуда такая разница между рисунком и реальным количеством очисток, выполняемых в таблицах pgbench_branches и pgbench_tellers? Ответ заключается в специфике представления pg_stat_activity, которое показывает текущий снимок и не носит накопительного характера: все то, что происходит между снимками, остается неизвестным. Для понимания количества очисток предпочтителен график на рис. 8.1.

Рис. 8.3. Количество процессов, выполняющих очистку

График на рис. 8.4 показывает те моменты, когда удалось поймать процессы очистки в pg_stat_activity и зафиксировать их продолжительность. Значений снова не так уж и много, но, к сожалению, в СУБД нет других представлений, откуда можно было бы извлечь такую информацию. Более полную и точную информацию можно получить из журналов активности. При установке параметра *log_autovacuum_min_duration* статистика работы каждого процесса автоочистки, занявшего более указанного времени, будет записана в журнал сообщений.

Рис. 8.4. Продолжительность выполнения операций очистки

```
LOG: automatic vacuum of table "pgbench.public.pgbench_accounts": index scans: 1
pages: 0 removed, 66788 remain, 66788 scanned (100.00% of total)
tuples: 1793208 removed, 2000413 remain, 411 are dead but not yet removable
removable cutoff: 215988728, which was 1945 XIDs old when operation ended
new relfrozenxid: 215985452, which is 104702 XIDs ahead of previous value
index scan needed: 33412 pages from table (50.03% of total) had 1887056 dead item identifiers removed
index "pgbench_accounts_pkey": pages: 10970 in total, 0 newly deleted, 0 currently deleted, 0 reusable
I/O timings: read: 2668.125 ms, write: 935.198 ms
avg read rate: 13.761 MB/s, avg write rate: 14.561 MB/s
buffer usage: 83992 hits, 94033 misses, 99499 dirtied
WAL usage: 173015 records, 77548 full page images, 328276128 bytes
system usage: CPU: user: 1.90 s, system: 1.58 s, elapsed: 53.38 s
```

Статистика о работе очистки содержит массу информации, и, если проанализировать статистику за продолжительный период, наверняка можно сделать интересные наблюдения и выводы. В приведенном примере нас интересует последняя строка, где отмечено, что выполнение очистки заняло 53,38 секунды. Таким образом, анализируя поток журнала, можно отследить все очистки, выполнявшиеся дольше *log_autovacuum_min_duration*, в отличие от обращений к pg_stat_activity, где приходится полагаться на удачу. Однако не стоит злоупотреблять и слишком сильно уменьшать значение *log_autovacuum_min_duration*, поскольку в системах с большой активностью и большим количеством таблиц может выполняться огромное количество очисток, что приведет к существенному объему записи в журнал сообщений.

Представление pg_stat_progress_vacuum

Представление pg_stat_progress_vacuum содержит подробную информацию о работе процессов очистки и позволяет оценить ход их выполнения. Каждая строка в представлении описывает отдельный процесс, который идет в данный момент. Если выполняющихся процессов очистки нет, то и в представлении не будет ни одной строки.

- pid — идентификатор процесса в операционной системе;
- datid, datname — идентификатор и имя базы данных, с которой установлено соединение и в которой в данный момент выполняется очистка таблиц и индексов;
- relid — идентификатор таблицы, для которой в данный момент выполняется очистка;
- phase — фаза выполнения очистки:
  - initializing — инициализация и подготовка к работе;
  - scanning heap — сканирование таблицы, может включать в себя очистку, дефрагментацию страниц и заморозку строк;
  - vacuuming indexes — обработка индексов, может выполняться несколько раз (в случае нехватки *autovacuum_work_mem* или *maintenance_work_mem*);
  - vacuuming heap — обработка таблиц, отличается от сканирования тем, что выполняется каждый раз после обработки индексов (сначала в индексах удаляются указатели

на строки, затем в таблице удаляются непосредственно сами версии строк, отмеченные указателями);

- `cleaning up indexes` — окончательная обработка индексов, выполняется после завершения полного сканирования таблицы и очистки как таблицы, так и ее индексов;
- `truncating heap` — усечение файла таблицы и высвобождение свободных страниц с целью вернуть место операционной системе;
- `performing final cleanup` — окончательная очистка, обновление карты свободного пространства, обновление статистики в `pg_class` и т. п., завершение работы;

- `heap_blks_total` — общее количество блоков в таблице на момент начала работы; блоки, добавленные в процессе очистки, не учитываются;
- `heap_blks_scanned` — общее количество просканированных блоков. Для оптимизации сканирования используется карта видимости, и часть блоков может быть пропущена без проверки, но пропущенные блоки также учитываются в этом поле и в конечном счете `heap_blks_scanned` сравняется с `heap_blks_total`;
- `heap_blks_vacuumed` — общее количество блоков, которые были очищены от устаревших версий строк. Если в таблице нет индексов, этот счетчик увеличивается только в фазе `vacuuming heap`. Блоки, не содержащие мертвых строк, пропускаются, так что счетчик может увеличиваться большими рывками;
- `index_vacuum_count` — общее количество завершенных циклов очистки индексов;
- `max_dead_tuples` — количество указателей на устаревшие версии строк, которые можно поместить в рабочую память (*autovacuum_work_mem*, *maintenance_work_mem*). Когда рабочая память заканчивается, выполняется цикл очистки индексов;
- `num_dead_tuples` — количество указателей на устаревшие версии строк, собранных в рабочей памяти с момента завершения последнего цикла очистки индексов.

Необработанная статистика может показаться недостаточно понятной и не отвечающей на возникающие у администратора вопросы: какая таблица сейчас обрабатывается, как долго и с какой скоростью выполняется очистка и когда она закончится?

```
# SELECT * FROM pg_stat_progress_vacuum;
-[ RECORD 1 ]------+-------------------------------------------------
pid                | 2688928
datid              | 16437
datname            | pgbench
relid              | 16462
phase              | vacuuming heap
heap_blks_total    | 66788
heap_blks_scanned  | 66788
heap_blks_vacuumed | 30540
index_vacuum_count | 1
max_dead_tuples    | 11184809
num_dead_tuples    | 1921611
```

В выводе присутствуют идентификаторы и значения в блоках, менее привычные, чем обычные байты. Для удобства к запросу можно добавить информацию из pg_stat_activity, а поля pg_stat_progress_vacuum привести к более привычному для восприятия виду:

```
# SELECT
    p.pid,
    now() - a.xact_start AS duration,
    wait_event_type ||'.'|| wait_event AS wait_event,
    CASE
        WHEN a.query ~ '^autovacuum.*to prevent wraparound' THEN 'wraparound'
        WHEN a.query ~ '^vacuum' THEN 'user'
        ELSE 'regular'
    END AS mode,
    p.datname AS database,
    p.relid::regclass AS table,
    p.phase,
    pg_size_pretty(p.heap_blks_total * current_setting('block_size')::int) AS table_size,
    pg_size_pretty(pg_total_relation_size(relid)) AS total_size,
    pg_size_pretty(p.heap_blks_scanned * current_setting('block_size')::int) AS scanned,
    round(100.0 * p.heap_blks_scanned / p.heap_blks_total, 1) AS scanned_pct,
    pg_size_pretty(p.heap_blks_vacuumed * current_setting('block_size')::int) AS vacuumed,
    round(100.0 * p.heap_blks_vacuumed / p.heap_blks_total, 1) AS vacuumed_pct,
    p.index_vacuum_count,
    round(100.0 * p.num_dead_tuples / p.max_dead_tuples,1) AS work_mem_usage
  FROM pg_stat_progress_vacuum p
  JOIN pg_stat_activity a ON a.pid = p.pid
  ORDER BY now() - a.xact_start DESC;
-[ RECORD 1 ]------+-------------------------------------------------
pid               | 2688928
duration          | 00:00:52.050535
wait_event        | Timeout.VacuumDelay
mode              | regular
database          | pgbench
table             | pgbench_accounts
phase             | vacuuming heap
table_size        | 522 MB
total_size        | 608 MB
scanned           | 522 MB
scanned_pct       | 100.0
vacuumed          | 239 MB
vacuumed_pct      | 45.7
index_vacuum_count | 1
work_mem_usage    | 17.2
```

Теперь в полученном результате есть четкие ответы:

- выполняется обработка таблицы pgbench_accounts;
- обработка текущей таблицы длится 52 секунды;
- в данный момент выполняется штатная пауза между обработкой блоков (wait_event = Timeout.VacuumDelay);

- mode = regular указывает на то, что это обычная автоочистка (не связанная с обслуживанием счетчика транзакций и не вызванная пользователем);
- размер таблицы 522 МБ, вместе с индексами — 608 МБ;
- значение scanned_pct = 100.0% говорит о том, что таблица полностью просканирована на предмет мертвых строк и выполняется непосредственно очистка (phase = vacuuming heap);
- vacuumed_pct = 45.7% указывает на то, что очистка выполнена почти наполовину;
- рабочая память заполнена на 17.2 % (work_mem_usage) и уже был выполнен один цикл обработки индексов (index_vacuum_count = 1).

Если снова провести полное обновление таблицы pgbench_accounts, а в соседнем сеансе запустить запрос с помощью \watch 1, можно будет наглядно наблюдать за тем, как выполняется очистка. Однако такой запрос и выводимая статистика больше подходят для текущей оценки происходящего, собирать такую информацию в мониторинг не имеет особого смысла.

Резюме

- Механизм конкурентного доступа допускает существование нескольких версий одной и той же строки.
- Жизненный цикл строк подразумевает существование живых и мертвых версий.
- Мертвые версии строки необходимо регулярно вычищать.
- Очистка выполняется как для таблиц, так и для индексов.
- Автоочистка выполняется, когда количество мертвых версий строк превышает определенный порог.
- Очистка создает дополнительный ввод-вывод и может влиять на производительность.
- Администратору важно отслеживать работу очистки в СУБД.
- Автоочистка должна запускаться без задержек.
- Автоочистка обслуживает счетчик транзакций и обеспечивает постоянный запас свободных идентификаторов транзакций.
- Администратору важно отслеживать запас доступных идентификаторов транзакций.
- При неблагоприятном стечении обстоятельств запас идентификаторов может оказаться исчерпанным, что приведет к остановке нормальной работы СУБД.
- Неэффективная работа очистки может приводить к эффекту раздувания.
- Следствием раздувания являются неэффективное использование дискового пространства и снижение производительности.
- Для оценки степени раздувания используется расширение pgstattuple.
- Для отслеживания активных процессов очистки могут использоваться представления pg_stat_activity и pg_stat_progress_vacuum.

Глава 9

Ход выполнения операций

В этой главе мы рассмотрим:

- инструменты для отслеживания выполнения команд;
- отслеживание сбора статистики для планировщика;
- отслеживание хода выполнения резервного копирования;
- отслеживание выполнения команд CLUSTER и VACUUM FULL;
- отслеживание выполнения команд CREATE INDEX и REINDEX;
- отслеживание выполнения команд COPY.

В этой главе мы рассмотрим отдельный набор системных представлений, которые служат индикаторами выполнения различных команд, работа которых может занимать продолжительное время. Выполнение таких команд может требовать значительных ресурсов, создавать нагрузку (например, на подсистему ввода-вывода), конкурировать с соседними процессами и даже блокировать их. Предполагая подобные риски, администратор должен заранее планировать выполнение таких операций и иметь инструменты, которые позволяют отслеживать ход их выполнения и давать представление о том, когда они будут завершены. Все инструменты, рассматриваемые в этой главе, являются представлениями и показывают состояние на момент обращения аналогично pg_stat_activity и pg_locks. Эта статистика требуется в основном в особых случаях, возникающих при эксплуатации, и может выводиться в пользовательских интерфейсах средств управления PostgreSQL. В меньшей степени она подходит для накопления и отображения в системах мониторинга.

Несмотря на большое количество потенциально долгих команд, СУБД представляет инструменты отслеживания только небольшой части из них:

- pg_stat_progress_vacuum — выполнение операций очистки и автоочистки;
- pg_stat_progress_analyze — выполнение сбора статистики для планировщика;
- pg_stat_progress_basebackup — выполнение операций резервного копирования;
- pg_stat_progress_cluster — выполнение команд CLUSTER и VACUUM FULL;
- pg_stat_progress_create_index — выполнение команд CREATE INDEX и REINDEX;
- pg_stat_progress_copy — выполнение команд COPY.

Представление pg_stat_progress_vacuum подробно рассмотрено в главе 8, посвященной очистке, поэтому здесь мы не будем рассматривать его снова.

Все представления показывают выполняющиеся команды, которые обычно запускаются при необходимости вручную (за исключением автоочистки). Также вероятно, что в последующих версиях СУБД будет расширяться поддержка команд и будут появляться новые похожие представления.

9.1. Представление pg_stat_progress_analyze

При исполнении запроса СУБД следует наиболее оптимальному с точки зрения использования ресурсов плану выполнения, выбранному планировщиком из множества возможных планов. Для оценки планов используется статистика, описывающая данные, хранящиеся в таблицах, и от ее точности зависит качество выбранного плана. Собираемая статистика отражается в системном представлении `pg_stats`[1], основанном на таблице `pg_statistic`[2]. В процессе эксплуатации СУБД количественные и качественные характеристики данных могут и будут меняться, и статистика планировщика в `pg_statistics` будет неизбежно устаревать. В условиях изменяющихся данных очень важно регулярно обновлять статистику: в противном случае выбираемые на основе устаревшей информации планы будут неэффективными, что приведет к избыточному использованию ресурсов и увеличению времени выполнения.

Статистика обычно собирается рабочими процессами автоочистки, но у администратора есть возможность запустить сбор принудительно (с помощью команды `ANALYZE`[3] или `VACUUM`) и либо дождаться его окончания, либо оценить, сколько времени он займет.

Это может понадобиться в разных ситуациях:

- после восстановления из логической резервной копии (сделанной с помощью pg_dump) или после загрузки данных с помощью команды `COPY` нужно собрать начальную статистику, поскольку ее не существовало вообще;
- после изменения параметра *default_statistics_target*[4], который ограничивает объем анализируемых и собираемых данных;
- прежде чем начинать глубже исследовать проблему производительности отдельного запроса с помощью `EXPLAIN`[5], можно предварительно убедиться, что статистика актуальна;
- после обновления версии СУБД с помощью утилиты pg_upgrade[6], поскольку при такой процедуре статистика планировщика не переносится.

Возможны и другие сценарии, но в любом случае процесс сбора статистики может занимать продолжительное время и администратору следует иметь представление о том, как долго он

[1] www.postgresql.org/docs/current/view-pg-stats.html
[2] www.postgresql.org/docs/current/catalog-pg-statistic.html
[3] www.postgresql.org/docs/current/sql-analyze.html
[4] www.postgresql.org/docs/current/runtime-config-query.html#GUC-DEFAULT-STATISTICS-TARGET
[5] www.postgresql.org/docs/current/sql-explain.html
[6] www.postgresql.org/docs/current/pgupgrade.html

будет продолжаться. Для отслеживания операций сбора статистики используется представление pg_stat_progress_analyze. Каждая строка в представлении соответствует отдельному процессу, собирающему статистику, и содержит следующие поля:

- pid — идентификатор процесса в операционной системе;
- datid, datname — идентификатор и имя базы данных, к которой выполнено подключение;
- relid — идентификатор таблицы, в которой выполняется сбор статистики;
- phase — фаза сбора статистики:
  - initializing — подготовка к сканированию таблицы;
  - acquiring sample rows — получение случайной выборки строк для анализа;
  - acquiring inherited sample rows — получение выборки строк из дочерних таблиц;
  - computing statistics — вычисление статистики на основе выбранного набора строк;
  - computing extended statistics — вычисление расширенной статистики;
  - finalizing analyze — завершение работы, обновление pg_class;
- sample_blks_total — общее количество блоков в случайной выборке, необходимой для вычисления статистики;
- sample_blks_scanned — общее количество уже просканированных блоков;
- ext_stats_total — количество объектов расширенной статистики;
- ext_stats_computed — количество вычисленных объектов расширенной статистики, увеличивается только в фазе computing extended statistics;
- child_tables_total — общее количество дочерних таблиц, в случае сбора статистики в секционированных таблицах;
- child_tables_done — количество просканированных дочерних таблиц, увеличивается только в фазе acquiring inherited sample rows;
- current_child_table_relid — идентификатор дочерней таблицы, сканируемой в данный момент; содержит актуальное значение только в фазе acquiring inherited sample rows.

Исходные данные из представления могут быть не очень понятны, поэтому для большей информативности имеет смысл провести некоторую обработку: транслировать идентификаторы в имена, перевести блоки в байты и взять дополнительную информацию из pg_stat_activity, соединив представления по полю pid.

В тестовом окружении довольно сложно поймать автоматически сбор статистики, поэтому запрос можно выполнить с помощью \watch 1 и в соседнем сеансе запустить команду ANALYZE. В первом сеансе появится подобный вывод:

```
# SELECT
    a.pid, now() - a.xact_start AS xact_age,
    p.datname, p.relid::regclass AS relation,
    a.state, a.wait_event_type ||'.'|| a.wait_event AS wait_event,
    p.phase,
    pg_size_pretty(p.sample_blks_total * (
        SELECT current_setting('block_size')::int )) AS sample_size,
    round(100 * p.sample_blks_scanned /
        greatest(p.sample_blks_total,1), 2) AS "scanned,%",
    p.ext_stats_total ||'/'|| p.ext_stats_computed AS "ext_total/done",
    p.child_tables_total ||'/'|| p.child_tables_done AS "child_total/done",
    current_child_table_relid::regclass AS child_in_progress,
    a.query
  FROM pg_stat_progress_analyze p
  INNER JOIN pg_stat_activity a ON p.pid = a.pid
  ORDER BY now() - a.xact_start DESC;
-[ RECORD 1 ]-----+---------------------
pid               | 536666
xact_age          | 00:00:07.660563
datname           | pgbench
relation          | pgbench_accounts
state             | active
wait_event        | IO.DataFileRead
phase             | acquiring sample rows
sample_size       | 522 MB
scanned,%         | 85.00
ext_total/done    | 0/0
child_total/done  | 0/0
child_in_progress | -
query             | ANALYZE;
```

Команда ANALYZE выполняется семь секунд, для подсчета статистики нужно получить выборку размером 522 МБ, и на данный момент просканировано уже 85 %. В данном примере через пару секунд сканирование будет закончено и начнется фаза подсчета статистики. Для более сложных случаев с секционированными таблицами или расширенной статистикой есть дополнительные поля с уточняющей информацией.

9.2. Представление pg_stat_progress_basebackup

Резервное копирование является одной из регулярных задач при эксплуатации производственных СУБД и в зависимости от размеров БД и характеристик оборудования может занимать значительное время. При использовании утилиты pg_basebackup[1] выполняется полное копирование файлов данных и необходимых WAL-журналов (для их передачи может запускаться дополнительный процесс walsender). Это создает значительную нагрузку на дисковую

[1] www.postgresql.org/docs/current/app-pgbasebackup.html

подсистему и может замедлить выполнение конкурентных запросов. В случае нехватки ресурсов и негативного влияния на производительность администратору важно представлять, как долго выполняется и как скоро закончится резервное копирование. Отслеживание встроено в саму утилиту pg_basebackup (параметры `-P`, `--progress`), но следить за ходом выполнения можно и со стороны СУБД с помощью представления `pg_stat_progress_basebackup`.

Каждая строка представления соответствует отдельному процессу резервного копирования и содержит следующую информацию:

- `pid` — идентификатор процесса в операционной системе;
- `phase` — фаза резервного копирования:
  - `initializing` — подготовка к резервному копированию;
  - `waiting for checkpoint to finish` — walsender ждет завершения контрольной точки;
  - `estimating backup size` — walsender подсчитывает размер файлов БД для передачи;
  - `streaming database files` — `walsender` передает данные;
  - `waiting for wal archiving to finish` — walsender ожидает завершения архивирования сегментов WAL-журнала;
  - `transferring wal files` — walsender передает сегменты WAL-журнала, созданные непосредственно в процессе резервного копирования;
- `backup_total` — общий объем данных, который будет передан. Значение вычисляется перед резервным копированием и является приблизительной оценкой, поскольку содержимое каталогов данных может измениться в процессе копирования. Если передаваемый объем данных начинает превышать рассчитанное значение, поле устанавливается в `backup_streamed`. Значение будет отсутствовать (`NULL`), если расчет оценки в pg_basebackup отключен (параметр `--no-estimate-size`);
- `backup_streamed` — объем переданных данных, увеличивается только в фазах `streaming database files` и `transferring wal files`;
- `tablespaces_total` — общее количество табличных пространств, которые будут переданы;
- `tablespaces_streamed` — количество уже переданных табличных пространств, увеличивается только в фазе `streaming database files`.

Чтобы увидеть какие-либо результаты, стоит выполнить приведенный ниже запрос с помощью `\watch 1`, запустив в соседнем терминале резервное копирование. Поскольку полученная резервная копия нам не понадобится, ее запись можно направить в /dev/null:

```
# pg_basebackup -P -R -X none -c fast -Ft -h 127.0.0.1 -U postgres -D - > /dev/null
```

Пока резервное копирование выполняется, в сеансе с запросом к `pg_stat_progress_basebackup` будет выводиться ход его выполнения:

```
# SELECT
    a.pid,
    host(a.client_addr) AS started_from,
    to_char(backend_start, 'YYYY-MM-DD HH24:MI:SS') AS started_at,
    now() - backend_start AS duration,
    a.state, a.wait_event_type ||'.'|| a.wait_event AS waiting,
    p.phase,
    pg_size_pretty(p.backup_total) AS size_total,
    pg_size_pretty(p.backup_streamed) AS sent,
    round(100 * p.backup_streamed / greatest(p.backup_total,1), 2) AS "sent,%",
    p.tablespaces_total ||'/'|| p.tablespaces_streamed AS "ts_total/streamed"
  FROM pg_stat_progress_basebackup p
  INNER JOIN pg_stat_activity a ON p.pid = a.pid
  ORDER BY now() - backend_start DESC;
-[ RECORD 1 ]--------------+------------------------
pid                        | 545355
started_from               | 127.0.0.1
started_at                 | 2023-03-06 04:43:41
duration                   | 00:00:36.700719
state                      | active
waiting                    | IO.BaseBackupRead
phase                      | streaming database files
size_total                 | 713 MB
sent                       | 623 MB
sent,%                     | 87.00
ts_total/streamed          | 1/0
```

Из приведенного фрагмента видно, что резервное копирование длится чуть больше 30 секунд, клиенту отправлена бóльшая часть содержимого БД, копирование выполнено на 87 % и уже подходит к завершению. Из представления `pg_stat_activity` взяты поле `state` и маркер ожидания, который в большинстве случаев будет показывать ожидания ввода-вывода, поскольку процесс копирует файлы данных и не использует блокировки.

9.3. Представление pg_stat_progress_cluster

Команды `CLUSTER`[1] и `VACUUM FULL`[2] используются для пересоздания таблиц. Они потребляют много ресурсов и устанавливают исключительную блокировку, запрещающую доступ к обрабатываемым таблицам. При неаккуратном использовании такие команды могут легко заблокировать работу приложений вплоть до своего завершения, поэтому администратору требуется инструмент, позволяющий отслеживать ход их выполнения. Для этого в СУБД есть представление `pg_stat_progress_cluster`, где каждая строка содержит информацию об отдельной операции со следующим набором полей:

[1] www.postgresql.org/docs/current/sql-cluster.html
[2] www.postgresql.org/docs/current/sql-vacuum.html

- pid — идентификатор процесса в операционной системе;
- datid, datname — идентификатор и имя базы данных, в которой выполняется операция;
- relid — идентификатор пересоздаваемой таблицы;
- command — выполняемая команда, CLUSTER или VACUUM FULL;
- phase — фаза операции:
  - initializing — подготовка к сканированию таблицы;
  - seq scanning heap — последовательное сканирование таблицы;
  - index scanning heap — команда CLUSTER выполняет индексное сканирование таблицы;
  - sorting tuples — команда CLUSTER выполняет сортировку строк;
  - writing new heap — выполняется запись данных в новый файл таблицы;
  - swapping relation files — выполняется подмена старых файлов новыми;
  - rebuilding index — перестроение индексов;
  - performing final cleanup — очистка и завершение работы;
- cluster_index_relid — идентификатор индекса для команды CLUSTER;
- heap_tuples_scanned — общее количество просканированных строк в таблице, изменяется только на стадиях последовательного или индексного сканирования и записи новой таблицы;
- heap_tuples_written — общее количество записанных строк в таблице, изменяется только на стадиях последовательного или индексного сканирования и записи новой таблицы;
- heap_blks_total — общее количество блоков в таблице на начало последовательного сканирования таблицы;
- heap_blks_scanned — общее количество просканированных блоков, изменяется только на этапе последовательного сканирования таблицы;
- index_rebuild_count — общее количество перестроенных индексов, изменяется только на этапе перестроения индексов.

На основе перечисленных полей легко понять, на каком этапе находится выполнение команды, и оценить ход выполнения. Для большей информативности также имеет смысл соединить представление с pg_stat_activity и провести некоторую обработку.

Для получения осмысленного результата запроса в соседнем терминале нужно запустить команду CLUSTER или VACUUM FULL:

```
# VACUUM FULL pgbench_accounts;
```

Тогда в сеансе с запросом к pg_stat_progress_cluster можно будет увидеть следующий вывод:

```
# SELECT
    a.pid,
    now() - a.xact_start AS xact_age,
    p.datname,
    p.relid::regclass AS relation,
    p.cluster_index_relid::regclass AS index,
    a.state,
    (SELECT count(distinct l.pid) FROM pg_locks l
      WHERE l.relation = p.relid AND NOT l.granted) AS blocked_total,
    a.wait_event_type ||'.'|| a.wait_event AS wait_event,
    p.phase,
    pg_size_pretty(p.heap_blks_total * (
        SELECT current_setting('block_size')::int)) AS size_total,
    round(100 * p.heap_blks_scanned /
        greatest(p.heap_blks_total,1), 2) AS "scanned,%",
    coalesce(p.heap_tuples_scanned, 0) AS tuples_scanned,
    coalesce(p.heap_tuples_written, 0) AS tuples_written,
    a.query
  FROM pg_stat_progress_cluster p
  INNER JOIN pg_stat_activity a ON p.pid = a.pid
  ORDER BY now() - a.xact_start DESC;
-[ RECORD 1 ]--+------------------------------
pid            | 550142
xact_age       | 00:00:06.867676
datname        | pgbench
relation       | pgbench_accounts
index          | -
state          | active
blocked_total  | 33
wait_event     | IO.DataFileExtend
phase          | seq scanning heap
size_total     | 257 MB
scanned,%      | 70.00
tuples_scanned | 1404122
tuples_written | 1404122
query          | VACUUM FULL pgbench_accounts;
```

В приведенном примере команда VACUUM FULL длится уже шесть секунд, она выполнена на 70 %, однако ее завершения ждут еще 33 сеанса, в которых выполняются запросы к перестраиваемой таблице. Соответственно, приложения, которые выполняют эти запросы, также заблокированы и вынуждены ждать завершения команды.

9.4. Представление pg_stat_progress_create_index

При активной разработке приложений и при появлении в приложении новых типов запросов создание индексов может быть довольно частой операцией. Создание индексов, особенно для

больших таблиц, может занимать продолжительное время, требовать значительных ресурсов и влиять на производительность конкурентных запросов.

Для отслеживания используется представление `pg_stat_progress_create_index`, где каждая строка описывает отдельный процесс, в котором выполняется построение индекса:

- `pid` — идентификатор процесса в операционной системе;
- `datid`, `datname` — идентификатор и имя базы данных, в которой происходит операция;
- `relid` — идентификатор таблицы, которой принадлежит индекс;
- `index_relid` — идентификатор индекса, над которым выполняется операция;
- `command` — выполняемая команда: `CREATE INDEX`, `CREATE INDEX CONCURRENTLY`, `REINDEX` или `REINDEX CONCURRENTLY`;
- `phase` — фаза операции:
  - `initializing` — подготовка к работе;
  - `waiting for writers before build` — команды `CREATE INDEX CONCURRENTLY` или `REINDEX CONCURRENTLY` ожидают завершения транзакций, которые удерживают блокировки на запись и могут читать таблицу. Фаза пропускается при выполнении операции в блокирующем режиме. Детали выполнения фазы можно отслеживать в `lockers_total`, `lockers_done` и `current_locker_pid`;
  - `building index` — методы доступа, поддерживающие отслеживание, передают статистику о своем состоянии. Детали построения индекса можно отслеживать в полях `blocks_total` и `blocks_done`, но также могут меняться и значения `tuples_total` и `tuples_done`;
  - `waiting for writers before validation` — команды `CREATE INDEX CONCURRENTLY` или `REINDEX CONCURRENTLY` ожидают завершения транзакций, которые удерживают блокировки на запись и могут записывать в таблицу. Эта фаза пропускается при выполнении операции в блокирующем режиме. Детали выполнения фазы можно отслеживать в `lockers_total`, `lockers_done` и `current_locker_pid`;
  - `index validation: scanning index` — команда `CREATE INDEX CONCURRENTLY` сканирует индекс на предмет строк, требующих проверки. Фаза пропускается при выполнении операции в блокирующем режиме. Детали выполнения фазы отражаются в полях `blocks_total` и `blocks_done`;
  - `index validation: sorting tuples` — команда `CREATE INDEX CONCURRENTLY` сортирует строки, найденные в фазе сканирования;
  - `index validation: scanning table` — команда `CREATE INDEX CONCURRENTLY` сканирует таблицу, чтобы проверить строки индекса, собранные в предыдущих двух фазах. Фаза пропускается при выполнении операции в блокирующем режиме. Детали выполнение фазы отражаются в `blocks_total` и `blocks_done`;

  - waiting for old snapshots — команды CREATE INDEX CONCURRENTLY или REINDEX CONCURRENTLY ожидают освобождения снимков теми транзакциями, которые могут видеть содержимое таблицы. Фаза пропускается при выполнении операции в блокирующем режиме. Детали выполнения отражаются в lockers_total, lockers_done и current_locker_pid;
  - waiting for readers before marking dead — перед тем как пометить старый индекс как нерабочий, команда REINDEX CONCURRENTLY ожидает завершения транзакций, удерживающих блокировки чтения. Фаза пропускается при выполнении операции в блокирующем режиме. Детали выполнения отражаются в lockers_total, lockers_done и current_locker_pid;
  - waiting for readers before dropping — прежде чем удалить старый индекс, команда REINDEX CONCURRENTLY ожидает завершения транзакций, которые удерживают блокировки чтения. Фаза пропускается при выполнении операции в блокирующем режиме. Детали выполнения отражаются в lockers_total, lockers_done и current_locker_pid;
- lockers_total — количество процессов, которых приходится ждать;
- lockers_done — количество процессов, ожидание которых завершено;
- current_locker_pid — идентификатор процесса, который удерживает блокировку в данный момент;
- blocks_total (tuples_total) — общее количество блоков (строк), которое требуется обработать в данной фазе;
- blocks_done (tuples_done) — общее количество блоков (строк), уже обработанных в данной фазе;
- partitions_total — общее число секций, для которых должны быть созданы индексы (в случае работы с секционированной таблицей);
- partitions_done — общее число секций, для которых уже выполнено создание индексов (в случае работы с секционированной таблицей).

Представление довольно подробно показывает ход построения индекса. Особенно стоит отметить, что значения, указывающие на блоки и строки, относятся к отдельным фазам, а не ко всему процессу целиком.

Для получения результатов запроса в соседнем сеансе нужно запустить перестроение индекса:

```
# REINDEX CONCURRENTLY pgbench_accounts_pkey;
```

В сеансе с запросом к pg_stat_progress_create_index можно будет увидеть подобный вывод, из которого следует, что перестроение индекса находится в фазе сканирования таблицы, которая завершена на 75 %. После этого будет начата фаза загрузки строк, ход которой можно будет отследить в поле tuples_done,%.

```
# SELECT
    a.pid,
    now() - a.xact_start AS xact_age,
    p.datname,
    p.relid::regclass AS relation,
    p.index_relid::regclass AS index,
    a.state, a.wait_event_type ||'.'|| a.wait_event AS wait_event,
    p.phase,
    p.current_locker_pid AS locker_pid,
    p.lockers_total ||'/'|| p.lockers_done AS lockers,
    pg_size_pretty(p.blocks_total * (
        SELECT current_setting('block_size')::int)) AS size_total,
    round(100 * p.blocks_done /
        greatest(p.blocks_total, 1), 2) AS "size_done,%",
    p.tuples_total,
    round(100 * p.tuples_done /
        greatest(p.tuples_total, 1), 2) AS "tuples_done,%",
    p.partitions_total ||'/'|| round(100 * p.partitions_done /
        greatest(p.partitions_total, 1), 2) AS "parts_total/done,%",
    a.query
  FROM pg_stat_progress_create_index p
  INNER JOIN pg_stat_activity a ON p.pid = a.pid
  ORDER BY now() - a.xact_start DESC;
-[ RECORD 1 ]------+-----------------------------------------------
pid               | 550142
xact_age          | 00:00:02.523236
datname           | pgbench
relation          | pgbench_accounts
index             | pgbench_accounts_pkey_ccnew
state             | active
wait_event        | LWLock.WALWrite
phase             | building index: scanning table
locker_pid        | 0
lockers           | 0/0
size_total        | 261 MB
size_done,%       | 75.00
tuples_total      | 0
tuples_done,%     | 0.00
parts_total/done,% | 0/0.00
query             | REINDEX INDEX CONCURRENTLY pgbench_accounts_pkey;
```

9.5. Представление pg_stat_progress_copy

Представление `pg_stat_progress_copy` используется для отслеживания команд `COPY`[1], которые нужны для загрузки и выгрузки данных, в том числе и с помощью утилит pg_dump и pg_restore.

[1] www.postgresql.org/docs/current/sql-copy.html

На больших объемах данных и при недостаточных ресурсах команда может выполняться продолжительное время и влиять на производительность конкурентных запросов.

Каждая строка в представлении описывает выполнение отдельной команды и содержит следующую информацию:

- pid — идентификатор процесса в операционной системе;
- datid, datname — идентификатор и имя базы данных, к которой выполнено подключение;
- relid — идентификатор таблицы, с которой происходит работа;
- command — выполняемая команда, COPY TO или COPY FROM;
- type — метод ввод-вывода, который используется при работе с данными: FILE, PROGRAM, PIPE (для COPY FROM STDIN и COPY TO STDOUT) или CALLBACK в случае начальной синхронизации таблицы при логической репликации;
- bytes_processed — общее число байтов, обработанных командой;
- bytes_total — размер исходного файла в байтах в случае выполнения COPY FROM. Значение может быть равно нулю, когда размер определить нельзя;
- tuples_processed — общее число строк, обработанных командой COPY;
- tuples_excluded — общее количество строк, отброшенных указанным условием WHERE.

Для большей информативности следует соединить представление с pg_stat_activity, добавить размер таблицы с помощью вызова pg_relation_size и применить прием оценки прогресса, характерный только для команд COPY.

```
# SELECT
    a.pid,
    a.state, a.wait_event_type ||'.'|| a.wait_event AS wait_event,
    now() - a.xact_start AS xact_age,
    p.datname, p.relid::regclass AS relation,
    pg_size_pretty(pg_relation_size(p.relid)) AS table_size_total,
    c.reltuples::bigint AS table_tuples_total,
    p.tuples_processed,
    p.tuples_excluded,
    pg_size_pretty(p.bytes_total) AS source_file_total,
    pg_size_pretty(p.bytes_processed) AS processed,
    CASE WHEN p.command = 'COPY FROM'
      THEN round(100 * p.bytes_processed / greatest(p.bytes_total, 1), 2)
      ELSE round(100 * p.tuples_processed / greatest(c.reltuples::bigint, p.tuples_processed), 2)
    END AS "done,%",
    p.command || ' ' || p.type AS command,
    a.query
  FROM pg_stat_progress_copy p
  LEFT JOIN pg_class c ON p.relid = c.oid
  INNER JOIN pg_stat_activity a ON p.pid = a.pid
  ORDER BY now() - a.xact_start DESC;
```

Команда COPY имеет два режима работы — загрузка (COPY FROM) и выгрузка (COPY TO); в выводе для удобства режим объединен с указанием источника данных. Оценка выполненной работы (поле done,%) также учитывает оба режима.

Для оценки загрузки можно обойтись полями самого представления: bytes_total показывает размер исходного файла, из которого выполняется загрузка, а bytes_processed — количество байтов, прочитанных командой COPY. Следовательно, процент выполненной работы определяется отношением прочитанного объема к общему.

С оценкой выгрузки дело обстоит чуть сложнее: представление данных в СУБД отличается от формата выгрузки (не говоря уже о том, что и в СУБД данные могут по-разному сжиматься, и форматов выгрузки существует несколько), поэтому итоговый файл будет иметь совершенно другой размер, чем исходная таблица. В показанном запросе оценка вычисляется как отношение количества обработанных строк tuples_processed к общему количеству строк таблицы из поля pg_class.reltuples. Однако в этом способе тоже есть недостаток: pg_class.reltuples обновляется после очистки, и, если она выполнялась достаточно давно, значение не будет соответствовать действительности и оценка окажется неточной. Но опыт показывает, что такая оценка все равно получается более точной, чем оценка на основе bytes_processed и размера таблицы.

Запрос к pg_stat_progress_copy следует выполнить в одном сеансе с помощью метакоманды \watch 1, а в другом — запустить выполнение команды COPY или pg_dump:

```
# pg_dump -U postgres -t pgbench_accounts pgbench >/dev/null
```

В сеансе с запросом к pg_stat_progress_copy можно увидеть примерно следующее:

```
-[ RECORD 1 ]------+---------------------------------------------------------------------
pid                | 223457
state              | active
wait_event         | IO.DataFileWrite
xact_age           | 00:00:18.393571
datname            | pgbench
relation           | pgbench_accounts
table_size_total   | 277 MB
table_tuples_total | 1999223
tuples_processed   | 1755368
tuples_excluded    | 0
source_file_total  | 0 bytes
processed          | 169 MB
done,%             | 87.00
command            | COPY TO PIPE
query              | COPY public.pgbench_accounts (aid, bid, abalance, filler) TO stdout;
```

В приведенном примере таблица pgbench_accounts копируется командой COPY TO через конвейер на стандартный вывод (stdout). Выгрузка длится уже 18 секунд и завершена на 87 %. Кроме того, приведена дополнительная информация о состоянии процесса.

Резюме

- СУБД располагает ограниченным набором представлений для отслеживания выполнения продолжительных команд.
- Представления отслеживания хода выполнения не накапливают статистику и показывают текущее состояние процессов.
- `pg_stat_progress_vacuum` используется для отслеживания операций очистки.
- `pg_stat_progress_analyze` используется для отслеживания сбора статистики, необходимой планировщику.
- `pg_stat_progress_basebackup` используется для отслеживания резервного копирования.
- `pg_stat_progress_cluster` используется для отслеживания команд `CLUSTER` и `VACUUM FULL`.
- `pg_stat_progress_create_index` используется для отслеживания команд `CREATE INDEX` и `REINDEX`.
- `pg_stat_progress_copy` используется для отслеживания команд `COPY`.
- Соединение представлений с `pg_stat_activity` и `pg_locks` повышает информативность.

Приложение
Тестовое окружение

В этом приложении содержится руководство по запуску и использованию тестового окружения. Это окружение необязательно для понимания книги, но может быть полезно в процессе прочтения. В окружении можно воспроизвести все приводимые в книге примеры и поставить любые эксперименты, связанные с мониторингом.

Общая информация

Окружение представляет собой установку на основе docker-контейнеров. В состав установки входят:

- Кластер потоковой репликации PostgreSQL из двух узлов. Кластер необходим для демонстрации средств и способов мониторинга репликации.
- Экспортер метрик pgSCV. Это агент мониторинга, собирающий статистику с узлов PostgreSQL и предоставляющий ее в формате Prometheus. Агент мониторинга является экспериментальным продуктом и не рекомендуется для производственного применения.
- Система мониторинга на основе Victoriametrics и Vmagent. Система мониторинга собирает метрики с агента pgSCV и хранит их в течение двух дней.
- Приложения на основе утилиты pgbench, создающие рабочую нагрузку. В установке запущено несколько экземпляров pgbench, задача которых — создавать постоянную динамическую нагрузку на кластер PostgreSQL.
- Система визуализации Grafana. Используется как фронтенд для системы мониторинга и позволяет создавать самые разные графики с богатой настраиваемостью.

Зависимости

Убедитесь, что в системе, где будет запускаться тестовое окружение, установлены следующие инструменты:

- Docker-compose версии $\geqslant$ 1.29.2;
- Docker Engine версии $\geqslant$ 20.10.12;
- GNU Make версии $\geqslant$ 4.2.1.

Работа тестового окружения с более старыми версиями не гарантируется.

Установку docker-compose можно выполнить по официальной инструкции, находящейся по адресу docs.docker.com/compose/install/. Docker Engine также устанавливается по официальной инструкции: docs.docker.com/engine/install/#server. Пакет GNU Make может быть установлен пакетным менеджером из официального репозитория пакетов для вашей операционной системы.

Когда все зависимости установлены, можно перейти к запуску окружения.

Запуск окружения

Работа с базовыми командами доступна через Makefile, но пользователь может выполнять и другие команды, напрямую вызывая утилиты docker и docker-compose.

Выполните следующие команды, чтобы перейти в каталог тестового окружения и затем посмотреть справку по доступным целям make:

```
$ git clone https://github.com/lesovsky/postgresql-monitoring-book.git
$ cd postgresql-monitoring-book/playground
$ make help
Makefile available targets:
* help         Display this help screen
* up           Start playground environment
* ps           Show current status of playground services
* down         Stop playground environment (without cleanup)
* destroy      Stop playground environment and clean up related resources
* %/logs       Show service logs; % - service name, e.g. primary or standby
* %/logs/tail  Tail service logs; % - service name, e.g. primary or standby
* %/shell      Start shell session; % - service name, e.g. primary or standby
* %/psql       Start psql session; % - service name, e.g. primary or standby
* %/pgcenter   Start pgcenter session; % - service name, e.g. primary or standby
```

Цели make предоставляют короткий способ запуска и остановки тестового окружения и подключения к службам в контейнерах. Это будет необходимо для дальнейшей работы. При выполнении целей в самой первой строке будет выводиться команда, которую можно выполнить без использования Makefile.

Запустите тестовое приложение:

```
$ make up
```

При первом запуске выполняются сборка контейнеров, инициализация служб и загрузка данных в тестовую базу (~300 МБ). Процедура может занять некоторое время, оно зависит от производительности устройства.

Общая проверка работоспособности

После инициализации следует проверить готовность окружения к работе. Первым делом необходимо выяснить общее состояние контейнеров и сервисов:

```
$ make ps
docker-compose ps
NAME                          COMMAND                  SERVICE          STATUS    PORTS
playground-app1-1             "docker-entrypoint.s..."  app1             running   5432/tcp
playground-app2-1             "docker-entrypoint.s..."  app2             running   5432/tcp
playground-app3-1             "docker-entrypoint.s..."  app3             running   5432/tcp
playground-app4-1             "docker-entrypoint.s..."  app4             running   5432/tcp
playground-grafana-1          "/run.sh"                 grafana          running   0.0.0.0:3000->3000/tcp
playground-pgscv-1            "pgscv"                   pgscv            running   0.0.0.0:9890->9890/tcp
playground-primary-1          "docker-entrypoint.s..."  primary          running   5432/tcp
playground-standby-1          "/docker-entrypoint...."  standby          running   5432/tcp
playground-victoriametrics-1 "/victoria-metrics-p..."  victoriametrics running   8428/tcp
playground-vmagent-1          "/vmagent-prod -prom..."  vmagent          running   8429/tcp
```

Все контейнеры должны быть в состоянии `running` (поле `STATUS`). Обе службы БД должны показывать готовность принимать клиентские подключения. Репликация между узлами также должна работать. Проверить журнал сообщений основного узла можно так:

```
$ make primary/logs
...
primary_1 | 2022-03-03 08:57:44 UTC [1] LOG: listening on IPv4 address "0.0.0.0", port 5432
primary_1 | 2022-03-03 08:57:44 UTC [1] LOG: listening on IPv6 address "::", port 5432
primary_1 | 2022-03-03 08:57:44 UTC [1] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
primary_1 | 2022-03-03 08:57:44 UTC [63] LOG: database system was shut down at 2022-03-03 08:57:44 UTC
primary_1 | 2022-03-03 08:57:44 UTC [1] LOG: database system is ready to accept connections
```

Аналогичный способ для резервного узла:

```
$ make standby/logs
...
standby_1 | 2022-03-03 08:57:50 UTC [24] LOG: consistent recovery state reached at 0/11000100
standby_1 | 2022-03-03 08:57:50 UTC [1] LOG: database system is ready to accept read-only connections
standby_1 | 2022-03-03 08:57:50 UTC [28] LOG: started streaming WAL from primary at 0/12000000 on timeline 1
```

Теперь в качестве проверки работоспособности начнем сеанс с основным узлом кластера:

```
$ make primary/psql
psql (15.0)
Type "help" for help.

postgres=#
```

В открывшемся сеансе мы можем отправлять SQL-запросы к серверу:

```
postgres=# SELECT * FROM pg_stat_activity;
...
postgres=# SELECT * FROM pg_stat_replication;
...
```

Ответы на запросы будут выводиться в табличном представлении. До прочтения книги можно не вникать в смысл запросов и их результат, но именно так будет происходить работа со статистикой — нужно будет написать SQL-запрос к нужному представлению, отправить его серверу и получить результат в виде таблицы, содержащий несколько строк (а иногда — ни одной).

В книге все запросы выполняются на основном узле, если это специально не оговорено.

Grafana

Фронтенд Grafana доступен по адресу 127.0.0.1:3000. Логин и пароль для входа не требуются. Для построения графиков можно использовать режим Explore или создать отдельный набор панелей и добавлять панели с графиками в него.

Остановка и удаление

Для остановки тестового окружения выполните следующую команду:

```
$ make down
```

Если тестовое окружение больше не нужно, можно остановить его и удалить все связанные с ним компоненты:

```
$ make destroy
```

Предметный указатель

Производственное-практическое издание

Лесовский Алексей Викторович

Мониторинг PostgreSQL

При поддержке Postgres Professional
postgrespro.ru

Ответственный редактор *Ю. В. Семенова*
Редактор *Е. В. Рогов*
Корректор *О. Е. Шишмаренкова*
Дизайн обложки *М. М. Гранько*

Подписано в печать 25.12.2023.
Формат 84×108 1/16. Бумага офсетная. Печать офсетная.
Гарнитуры ПТ (Паратайп) и Iosevka (Renzhi Li).
Объем 15,5 печ. л. Тираж 1000 экз.

ООО «Бумба»
108811, г. Москва, вн.тер.г. поселение Московский,
22-й км Киевского ш., двлд. 4, стр. 5
Тел.: +7 (977) 586-38-56
email: info@boombabook.ru
www.boombabook.ru

Отпечатано в филиале «Чеховский Печатный Двор» АО «Первая Образцовая Типография».
142300, Московская обл., г. Чехов, ул. Полиграфистов д. 1
Тел.: +7 (495) 107-02-68
www.chpd.ru
Заказ №

Знак информационной продукции 12+
(Федеральный закон № 436-ФЗ от 29.12.2010 г.)

Для заметок