Презентация

Вступление. Какие проблемы у нас есть и как мы собираемся их решать

Всем привет, я собираюсь рассказать вам про такой процесс, как автовакуум. Это объёмная тема, и ее можно разделить на три части : причины появления, жизненный цикл и внутренняя кухня самого процесса, и наконец подробности того, как работает функция, занимающаяся вакуумированием конкретно одной таблицы

В первую очередь поймем, зачем нам нужен автовакуум. Пробежимся по пяти проблемам, которые он призван решать

mvcc

Как мы знаем, postgresql поддерживает целостность данных, реализуя модель MVCC (multiversion concurrency control). Это позволяет нам снизить уровень конфликтов блокировок и повысить производительность. Однако мы платим тем, что в таблицах копятся мертвые версии строк. Что это вообще такое?

При любом действии со строкой (кроме вроде как чтения), создается ее копия (aka версия) с внесенными изменениями, разумеется. В заголовке есть место для номера транзакции вставившей и удалившей строку. Такая операция как UPDATE распадается на две : удалить старую версию и создать новую.
Таким образом, если мы создадим, а потом 10 раз обновим одну и ту же строку, у нас будут храниться 11 ее копий. Можно представить, сколько места тратится на мертвые версии при активной работе с довольно большой страницей.

Это чревато тем, что на sql запросы будет тратиться все больше и больше времени, а может и вовсе всё зависнет.

wraparound

Помимо этого есть проблема wraparound - оборачивание счетчика транзакций. Дело в том, что мы используем 32 битный счетчик для нумерации транзакций и разумеется он может быть переполнен. В таком случае счет нужно начинать сначала, но как тогда определить, какая транзакция была раньше?

Если предствить все номера транзакций на круге, то мы считаем, что для транзакции N предшествуют транзакции, располагающиеся на левой дуге круга.
Однако в таком случае, если какая то транзакция сохраняет актуальность большое количество времени, то из левой дуги она (для самой новой транзакции) перейдет в правую, и мы уже не сможем видеть ее изменения (поскольку по логике она будет в "будущем")

Чтобы избежать таких ситуаций, мы используем заморозку. Мы особым образом помечаем эту транзакцию, так что для остальных она всегда будет считаться "в прошлом"

Кстати, стоит сказать, что риск wraparound есть не только для счетчика обычных транзакций, но и для счетчика мультитранзакций. Это такой же 32-х битный счетчик, который идентифицирует группу транзакций, спящей на какой либо блокировке (в заголовке страницы нет места, чтобы перечислить все номера транзакций. Мы указываем номер мультитранзакции и выставляем бит-подсказку чтобы отличить его от обычной транзакции).

analyze

У нас есть системные таблицы pg_class и pg_statistic. pg_class хранит базовую статистику для отношения в целом. pg_statistic статистику информацию для каждого столбца отношения. К этой статистике обращается планировщик запросов - программа (часть оптимизитора запросов) которая отвечает за создание оптимального плана выполнения SQL-запросов

Чтобы планировщик работал хорошо, мы хотим поддерживать +- актуальную информацию о нашей базе данных. Конечно, мы не стремимся к 100% достоверной информации (это было бы слишком накладно), однако хотелось бы иметь механизм, который бы раз в какой то промежуток времени обновлял нам статистику

vm и fsm

Теперь вспоминаем про существование vm и fsm. Это два полезных дополнительных слоя отношения, которые делают следующее:

Карта видимости отмечает страницы, на которых все версии строк видны во всех снимках (в vm на каждую страницу отведено 2 бита. Один отвечает за all-visible, а другой за all-frozen). Она позволяет ускорить индексный доступ к данным (и на самом деле здорово помогает вакууму)
Карта свободного пространства указывает свободное место в таблицах. Очевидно, ускоряет вставку новых версий строк (отпадает необходимость сканировать некоторые страницы в поисках свободного места)

Обрезка страниц памяти

В процессе работы с таблицей может возникнуть ситуация, когда она разрослась на N страниц, и последние K страниц содержат только неактуальные данные. Во первых, (возвращаемся к первому пункту) мы хотели бы избавиться от этих версий строк, а во-вторых, "откусить" эти страницы от таблицы и отдать их операционной системе (если их больше, чем 1/16 всей таблицы, если не путаю)

VACUUM врукопашную

Итак, мы выделили 4 проблемы. Есть ли механизм их решения? Да, есть - это операция VACUUM.
Эту команду мы вручную вызываме из консоли, при этом можно указать ему следующие параметры:

- FULL - происходит полное перестроение таблицы, но в рамках доклада мы не будем в это углубляться, поскольку автовакуум не допустит такого
- FREEZE - испольовать агрессивную заморозку. То есть мы можем замораживать даже те версии строк, которые находятся на страницах, помеченных в карте видимости как all-visible
- ANALYZE - обновляет статистику
- DISABLE_PAGE_SKIPPING - параметр, который (внезапно) запрещает пропускать страницы и используется, когда целостность карты видимости вызывает подозрения (если верить документации)
- SKIP_LOCKED
- INDEX_CLEANUP
- PROCESS_MAIN

- PROCESS_TOAST
- TRUNCATE
- PARALLEL
- SKIP_DATABASE_STATS
- ONLY_DATABASE_STATS
- BUFFER_USAGE_LIMIT

Их много. И с каждой версией постгреса все больше. Мы не будем сейчас рассматривать эти опции. Просто скажу, что команда настраивается довольно гибко, и, забегая вперед, решает все наши вышеперечисленные проблемы. Прекрасно! Осталось только завести таймер и вызывать команду VACUUM раз в N квантов времени
Нет....

Очевидно, что с таким подходом мы можем либо лишний раз нагружать систему, работая над таблицами, в которой происходит пренебрежимо мало изменений, либо наоборот, мирно спать и дожидаться таймера, в то время как идет очень активная работа с таблицами и счетчик транзакций движется к wraparound (либо заканчивается место на диске, либо запросы работают все хуже и так далее)

Тут мы подходим к мысли, что хорошо бы иметь процесс, который динамически бы определял подходящее время и подходящие таблицы для того, чтобы прийти и запустить для них вакуум

AUTOVACUUM

Launcher и Worker

Система автоматической очистки состоит из двух типов процессов : autovacuum launcher и autovacuum worker
В системе может быть активным только один autovacuum launcher, и несколько autovacuum worker

Launcher - это always-running процесс, который запускается postmaster'ом после своей собственной инициализации. Postmaster также перезапускает Launcher, при возникновении каких то проблем (например, если администратор сам его убьет)

Сам запуск происходит при помощи функции StartAutoVacLauncher, которая инициализирует базовое окружение при помощи функции InitPostmasterChild, закрывает порты постмастера функцией ClosePostmasterPorts (у самого постмастера эти порты конечно остаются открытыми) и наконец вызывает функцию AutoVacLauncherMain

AutoVacLauncherMain - настраиваетсвя на работу со средой (выделяет свой контекст памяти, выставляет конфигурационные переменные (например, default_transaction_isolation = read committed), инициализирует обработчики ошибок), составляет список баз данных (берет его из pg_database) и запускает бесконечный цикл (не совсем бесконечный - работаем, пока не получим ShutdownRequest от постмастера), в котором мы спим пока не понадобится запустить нового worker'a,
Launcher старается запускать нового worker'a для каждой базы данных раз в **autovacuum_naptime** секунд, при этом одновременно могут выполняться до autovacuum_max_workers рабочих процессов.

Кто такой worker? Worker - это процесс, который непосредственно занимается вакуумированием. Запуск worker'a осуществляется в функции do_start_worker. Она подбирает наиболее подходящую базу данных для вакуумирования, складывает ее имя и еще некоторую служебную информацию в общую память и шлет сигнал postmaster'y, чтобы он запустил worker'a

Как подбирается наиболее подходящая база данных? Мы пробегаем по всем базам данных и сравниваем их друг с другом по трем критериям. Приоритет критериев следующий:

1. База данных, для которой есть риск wraparound'a обычных транзакций (то есть **datfrozenxid** (All transaction IDs before this one have been replaced with a permanent (“frozen”) transaction ID in this database) перешагнул за порог, который определяется как разница текущей транзакции и параметра **autovacuum_freeze_max_age**)
    1. Если таких несколько, то выбираем базу с самым старым **datfrozenxid**
    2. **datfrozenxid** - номер транзакции такой, что все предшествующие ему были заменены постоянным (“замороженным”) идентификатором транзакции в этой базе данных
2. База данных, для которой есть риск wraparound'a мультитранзакций

    1. Если их несколько, то логика аналогичная (только смотрим на параметры **datminmxid** и **autovacuum_multixact_freeze_max_age**)
3. База данных, которая не была автовакуумирована дольше всех (при том условии, что упоминание о ней есть в pg_stat)
   1. Логика наличия упоминания в pg_stat в том, что если никто не присоединялся к базе данных со времен инициализации pg_stat, то и обрабатывать ее не надо
   Еще раз - база без упоминаний в pg_stat будет обрабатываться только в том случае, если есть риск wraparound'a
   Также пропускаем базу данных, если для нее был запущен worker меньше, чем **autovacuum_naptime** секунд назад

Итак, когда база выбрана мы записываем имя базы данных в общем для автовакуума контексте памяти, шлем сигнал в postmaster, и он уже запускает worker'a аналогично launcher'y (при помощи fork).

StartAutoVacWorker почти идентична StartAutoVacLauncher, а AutoVacWorkerMain почти идентична AutoVacLauncherMain, с тем отличиеем, что в ней мы забираем из общей памяти инфу о базе данных, которую будем обрабатывать и вызываем функцию do_autovacuum. Одну и ту же базу данных в один момент могут обрабатывать несколько воркеров. Это произойдет, если первый запущенный воркер не успел справиться за autovacuum_naptime секунд и Launcher запускает в базу еще один процесс. Процессы могут работать параллельно на уровне таблиц : работать с одной и той же таблицей одновременно они не могут (также как и не могут обрабатывать индексы одной таблицы параллельно)

Наконец-то, работа

do_autovacuum

Итак, у нас есть рабочий процесс и есть база данных, которую нужно вакуумировать
Функция do_autovacuum является последней "оберткой" перед вызовом главной низкоуровневой рабочей лошадки

Если раньше мы выбирали базу данных для вакуума, то теперь нам нужно отобрать таблицы из этой базы данных. Мы копируем в свой локальный контекст памяти системную таблицу pg_class, которая содержит информацию обо всех таблицах в базе. Далее мы сканируем pg_class в два

прохода: на первом мы собираем список обычных таблиц и материализованных представлений, а на втором - собираем TOAST-таблицы. Если таблица оказалось временной (предположительно, какого то другого бэкенда), мы пропускаем ее, поскольку не можем безопасно обработать

Далее проходим по всем собранным таблицам и для каждой:

- Проверяем, нуждается ли таблица в вакууме или анализе (вызываем функцию relation_needs_vacanalyze)
  Таблица требует вакуумирования, если количество мертвых кортежей превышает порог. Этот порог рассчитывается следующим образом: threshold = vac_base_thresh + vac_scale_factor * reltuples. Также вакуумирование будет проведено принудительно, если значение relfrozenxid (самая старая незамороженная транзакция) таблицы больше чем freeze_max_age транзакций назад, и если relminmxid больше чем multixact_freeze_max_age мультитранзакций назад.
- Таблица требует анализа, если количество вставленных, удаленных и обновленных кортежей превышает пороговое значение, рассчитанное таким же образом, как указано выше
- TOAST-таблицы мы только вакуумируем (никакого анализа)

Теперь, когда у нас уже есть набор таблиц и мы знаем, что для них делать, запускаем цикл по всем таблицам, в котором:

1. Если какой то worker уже работает над таблицей - пропускаем ее, иначе - заявляем о том, что мы работаем над этой таблицей
2. Обновляем параметры балансировки : **autovacuum_cost_limit**, **autovacuum_cost_delay**, **vacuum_cost_page_hit**, **vacuum_cost_page_miss**, **vacuum_cost_page_dirty**
3. Наконец, вызываем функцию vacuum

Summary

Перед тем, как перейти к алгоритму вакуумирования одной таблицы, давайте еще раз повторим, как мы оказались в этой точке
Excalidraw/Summary

vacuum()

Функция vacuum - точка входа как для автовакуума, так и для консольной команды VACUUM. Если вызываем команду вакуум из консоли, то вызывается оберточная функция ExecVacuum(), которая парсит параметры, проверяет их корректность и также вызывает vacuum. Получается то же самое, но никакого динамического анализа состояния системы
Vacuum Call Scheme

В vacuum присутствует цикл по всем переданным отношениям (список сформировали ранее), в котором для каждого сначала вызывается vacuum_rel, а потом, при необходимости, analyze_rel. Что значит при необходимости? Я вроде упоминал, что мы определяем для таблицы два параметра - необходимость вакуума и анализа, так что не обязательно они должны выполняться вместе. К тому же в процессе vacuum_rel мы можем выяснить что больше не являемся владельцами / таблица удалена / произойдет какая то ошибка

vacuum_rel

- Обработка каждой таблицы происходит в отдельной транзакции (иначе потребовались бы блокровки на всю базу данных).
- Мы открываем таблицу, используя блокировку ShareUpdateExclusiveLock (эта блокировка допускает одновременное чтение или добавление данных другим процессом). Если не получилось открыть, то делаем commit транзакции и переходим к следующей таблице
- Еще раз проверяем (на случай, если уже что то успело измениться), что мы являемся владелцем таблицы
- Создаем массив dead_items (Если что, это уже происходит в другой вложенной функции, но я буду заострять внимание только на названиях самых важных из них)

dead_items

Основным потребителем памяти при выполнении операции VACUUM является хранение массива TID (идентификаторов кортежей), которые нужно удалить из индексов. Мы хотим обеспечить возможность выполнять VACUUM даже для самых больших отношений с ограниченным использованием памяти. Для этого мы устанавливаем верхние ограничения на количество TID, которые мы можем отслеживать одновременно.

Размер данного массива определяет параметр maintenance_work_mem, который изначально равен 64 Мб. Память под этот массив выделяется сразу одним куском (не по частям)

Если массиву угрожает переполнению, мы должны вызвать lazy_vacuum для вакуумирования индексов (и вакуумирования страниц, которые мы удалили). Это освобождает пространство памяти, выделенное для хранения dead TID.

Алгоритм очистки

Теперь, когда все приготовления сделаны, можно рассказать про сам алгоритм прохода по таблице

Мы хотим просканировать таблицу в поисках мертвых кортежей. Мертвыми будут считаться те, которых удалила транзакция, ушедшая за горизонт баы данных. Их TID мы как раз будем записывать в dead_items. Однако мы не можем сразу их вычистить, поскольку на них все еще ссылаются индексы. Удаление на этом этапе привело бы к ошибкам при индексном доступе (к тому же есть строгое правило, что не должно быть ссылок на невалидные данные из индексов). Так что после прохода по таблице (либо переполнения dead_items) мы идем по всем индексам таблцы (это можно делать параллельно) и вычищаем оттуда указатели, которые ссылаются на строки из dead_items. После этого удаляем dead_items уже из основной таблицы

Когда TOAST?

(Если dead_items переполнился, а таблица не просканирована полностью, то мы вычищаем указатели из индексов, очищаем dead_items и продолжаем сканирования с точки, на которой остановились)

lazy_scan_heap

За все это безобразие отвечает функция **lazy_scan_heap** (вызывается внутри **vacuum_rel**). При сканировании каждой страницы таблицы используется функция lazy_scan_noprune (если не получилось захватить блокировку для очистки буфера - нужно для обрезки hot цепочек) или lazy_scan_prune, если захватить блокировку удалось

lazy_scan_noprune собирает элементы в dead_items и если оказывается, что на странице таких нет, то обновляет карту видимости (тут кстати происходит

запись в WAL)
lazy_scan_prune делает то же самое, но еще обрезает hot цепочки (элементы удаленные таким путем не относятся к dead_items)

lazy_vacuum

Функция lazy_vacuum очищает индексы функцией **lazy_vacuum_all_indexes**, но только если число мертвых кортежей достаточно велико (нет какого то определенного порога - там какой то макрос MAXDEADITEMS(32L *1024L* 1024L)). Иначе просто пропускаем очистку.

Теперь, когда из индексов удалены все ссылки на мертвые строки, удаляем их из таблицы функцией **lazy_vacuum_heap_rel** (она вычищает dead строки таблицы, указатель на них отмечает как неиспользуемый - LP_UNUSED, убирает фрагментацию, помечает страницу как грязную и делает запись в WAL). Это все делается в критической секции, так что если постгрес сломается в процессе, все изменения будут отменены. И кстати да, когда мы меняем карту видимости или свободного пространства, мы также заносим эти изменения в WAL **СЮДА ПО ХОРОШЕМУ ВСТАВИТЬ СКРИНЫ ИЗ СВОИХ ТЕСТОВ PG_WALDUMP**

Обновляем карту свободного пространства
Вызываем cleanup для индексов. Функция тут зависит от конкретного типа индекса. Я честно до конца так и не понял, что именно делает cleanup. Но в btree он точно обновляет карту свободного пространства и что то пишет в статистику

Что осталось?

На этом основная часть работы по вакуумированию выполнена. Нам осталось только очистить массив dead_items, закрыть индексы (мы их открывали, да), обновить статистику вакуумированного отношения и его индексов для системной таблицы pg_class.reltuples
Завершаем транзакцию, начатую в самом начале вакуума (надеюсь, не забыл об этом сказать)
Еще мы флашим (flush) IO статистику куда то, хы

ANALYZE

На этом операцию вакуума можно считать полностью завершенной и мы переходим к анализу.

Для каждой таблицы проверяем, существует ли она, потом мы вновь открываем отношение с ShareUpdateExclusiveLock блокировкой, потом проверяем, являемся ли мы ее владельцем (вроде как уже третий раз за все время)

Далее для таблицы и индексов выбираем столбцы для анализа (пропускаем системные атрибуты) и количество строк для анализа (параметр **default_statistics_target** *300. Дефолтное значение 100).*
Строки выбираются из случайных ***default_statistics_target*** 300 страниц. Поскольку в достаточно больших таблицах статистика собирается не по всем строкам, оценки могут немного расходиться с реальностью, но это нормально

Весь анализ проходит в отдельной транзакции. Главная функция - vac_update_relstats. Она обновляет всю статистику, связанную с таблицей и ее индексами, указанными в pg_class (это для всей таблицы) и pg_stats (для каждого столбца таблицы). Все изменения заносятся в WAL

Пора заканчивать

В конце выполнения функция vacuum `начнет новую транзакцию (которая соответствует CommitTransaction, ожидающей нас в PostgresMain() )` и попытается обновить три таблицы - pg_database, pg_xact и pg_multixact (две таблицы, хранящие метаинфомацию о текущих активных транзакциях / мультитранзакциях в системе).

pg_database мы уже знаем. В ней мы хотим поменять поля **datfrozenxid** (номер транзакции такой, что все предшествующие ему были заменены постоянным (“замороженным”) идентификатором транзакции в этой базе данных) и **datminmxid**. Если это удается сделать (то есть мы продвинули эти значения), то в таблицах pg_xact и pg_multixact удаляем все записи с транзакциями старше вышеупомянутых параметров

Рассказать про cost параметры

Пронумеровать слайды