5-MAR-25

Tantor: Настройка производительности PostgreSQL 16

tantor

Подготовлено: Олег Иванов, Александр Горбачук, Эмиль Школьник

Оглавление

Глава	Tantor: Настройка производительности PostgreSQL 16	страница
1-1	**Обзор**	9
	О компании Тантор	
	СУБД Tantor	
	Тантор xData	
	Тантор PipelineDB	
	Платформа Тантор	
	О курсе	
	Общие понятия настройки производительности	
	Методология настройки производительности	
	Последовательность действий по настройке производительности	
	Пример исключения задержки на стороне сервера приложений	
1-2	**Использование утилиты pgbench**	22
	Бенчмаркинг	
	Результат бенчмаркинга	
	pgbench - утилита бенчмаркинга PostgreSQL	
	Три встроенных теста pgbench	
	Параметры запуска pgbench	
	Рекомендации по использованию pgbench	
	Пример использования pgbench	
1-3	**Использование утилит sysbench и fio**	30
	sysbench - утилита для тестирования производительности	
	Использование sysbench для тестирования процессоров памяти дисков	
	Тестирование аппаратных ресурсов	
	Тестирование ввода-вывода утилитой Flexible IO Tester (fio)	
1-4	**Тесты TPC**	36
	Тесты TPC	
	Тесты TPC-B и TPC-C	
	Тест TPC-E тестирование устойчивости к сетевым сбоям	
	Реализация теста TPC-C	
	Приложение HammerDB	
	Параметры для теста тип-C HammerDB	
	Утилита Go-TPC	
	Практика	
2-1	**Память**	44
	Оперативная память	
	Виртуальная адресация памяти	
	Размер страниц памяти	
	Размер буфера ассоциативной трансляции (TLB)	
	Огромные страницы (Huge Pages)	
	Использование Huge Pages	
	Прозрачные огромные страницы (Transparent Huge Pages)	
	Использование экземпляром Huge Pages	
	Размер резидентного набора (RSS)	
2-2	**Нехватка памяти**	57
	Out Of Memory (OOM)	
	Параметр oom_score_adj	
	Параметр vm.overcommit_memory	
	Установка значений overcommit и swap	
	Параметр vm.swappiness	
	Дедупликация страниц памяти (KSM)	
	Неравномерный доступ к памяти (NUMA)	
	Параметр enable_large_allocations	

	Выделение локальной памяти процессами экземпляра	
	ОШИБКА: invalid memory alloc request size	
2-3	**Страничный кэш**	69
	Страничный кэш linux	
	Доля изменённых ("грязных") страниц в кэше	
	Фрагментация памяти	
	Дефрагментация памяти	
	Длительность удержания грязных страниц в кэше	
	Параметр backend_flush_after	
	Практика	
3-1	**Процессоры**	77
	Simultaneous Multi-Threading (SMT) и Hyper-Threading (HT)	
	Привязка процесса к процессору (CPU affinity)	
	Просмотр списка процессов утилитой ps	
	Запись и просмотр метрик утилитой atop	
	Переключения контекста выполнения	
	Планировщик операционной системы	
	Фоновые рабочие процессы	
	Использование процессорного времени (пропорция USER/SYS)	
	Источник времени (clock source)	
	Сравнение источников времени	
	Сравнение источников времени в PostgreSQL	
	Замена источника времени	
3-2	**Сеть**	91
	Основные параметры сети	
	Алгоритмы Congestion и slow start	
	Алгоритм BBR (Bottleneck Bandwidth и Round Trip Time)	
	Параметры сетевых соединений	
	Параметры энергосбережения	
	Практика	
4	**Система хранения**	98
	Дисковая подсистема	
	HDD SSD NVMe	
	Блочные устройства	
	Планировщик ввода/вывода (I/O Scheduler)	
	Изменение I/O Scheduler	
	Физический сектор диска	
	Взаимодействие процессов экземпляра с диском	
	Синхронизация файлов данных с диском	
	Размер блока файловой системы	
	Параметр wal_sync_method	
	Гарантия записи в WAL	
	Быстрые фиксации изменений в журнале файловой системы ext4 (fast_commit)	
	Включение быстрой фиксации (fast_commit)	
	Утилита pg_test_fsync	
	Групповая фиксация транзакций	
	Параметры commit_delay и commit_siblings	
	Команды шин ввода-вывода discard/trim	
	Поддержка discard/trim	
	Рекомендации по использованию SSD	
	Параметр max_files_per_process	
	Увеличение значения max_files_per_process	
	Временная файловая система (tmpfs)	

	RAID	
	LVM	
	Практика	
5	**Начальная настройка СУБД**	127
	Конфигураторы	
	Параметры shared_buffers temp_buffers effective_cache_size	
	Параметры work_mem hash_mem_multiplier maintenance_work_mem	
	Параметр autovacuum_work_mem	
	Параметры temp_file_limit и temp_tablespaces	
	Параметры max_slot_wal_keep_size и transaction_timeout	
	Параметры max_connections и client_connection_check_interval	
	Параметр max_locks_per_transaction	
	Параметры max_worker_processes и max_parallel_workers	
	Параметр max_parallel_workers_per_gather	
	Параметры системы хранения	
	Параметры контрольных точек	
	Параметры процесса фоновой записи bgwriter	
	Практика	
6	**Структуры хранения**	144
	Таблицы	
	Служебные столбцы	
	Расширение pageinspect	
	Padding и aligning	
	Aligning (выравнивание)	
	cache line	
	Структура блока данных	
	Число строк в блоке	
	Порядок следования столбцов в таблице	
	Порядок следования столбцов и производительность	
	Практика	
7	**Индексы**	158
	Методы доступа к строкам	
	Класс операторов для индекса	
	Семейства и классы операторов	
	Поддерживающие функции для индекса	
	Индексы для ограничений целостности	
	Индекс типа btree	
	Функции расширения pageinspect для btree	
	Индексы с дедупликацией в листовых блоках	
	Проверка поддерживается ли дедупликация	
	Параметры создания индекса и их влияние на производительность	
	Частичные (partial) индексы	
	Эволюция индексов: создание удаление перестройка	
	Структура индекса типа btree	
	High key в структуре индекса	
	Изменение структуры индекса при добавлении строк	
	Пример роста индекса при вставке строк	
	Структура индекса после его перестроения	
	FILLFACTOR в индексах типа btree	
	Быстрый путь (fastpath) вставки в индексы	
	Внутристраничная очистка в индексах	
	Влияние удаления строк на индексы	
	Исключение блоков из структуры индекса	

	Число исключённых блоков из структуры индекса	
	Практика	
8-1	**TOAST**	186
	TOAST (The Oversized-Attribute Storage Technique)	
	Поля переменной длинны	
	Вытеснение полей в TOAST	
	Алгоритм вытеснения полей в TOAST	
	TOAST chunk	
	Ограничения TOAST	
	Выравнивание строк с вытесненными в TOAST полями	
	Параметры toast_tuple_target и default_toast_compression	
	Оптимизация Heap Only Tuple	
	Мониторинг HOT update	
	Влияние FILLFACTOR на HOT cleanup	
	Внутристраничная очистка в таблицах	
	Внутристраничная очистка в индексах	
8-2	**Типы данных**	201
	Типы данных наименьшего размера: boolean char smallint	
	Типы данных переменной длинны	
	Целочисленные типы данных	
	Выбор типов данных для первичного ключа	
	Параметр cache у последовательностей	
	Хранение дат, времени, их интервалов	
	Функции проверки типа данных и размера поля	
	Типы данных для вещественных чисел	
	Параметр конфигурации extra_float_digits	
	Хранение вещественных чисел	
	Разрядность результата деления numeric	
	Практика	
9-1	**Архитектура**	214
	Запуск экземпляра, процесс postgres	
	Процесс startup	
	Синхронизация PGDATA, параметр recovery_init_sync_method	
	Синхронизация бэкапа, параметр pg_basebackup --sync-method	
	Параметр restart_after_crash	
	Особенности работы экземпляра в контейнере docker	
	Что происходит при запуске серверного процесса	
	Общая память процессов экземпляра	
	Кэш таблиц системного каталога	
	Представление pg_stat_slru	
	Локальная память процесса	
	Представление pg_backend_memory_contexts	
	Функция pg_log_backend_memory_contexts(PID)	
9-2	**Блокировки**	229
	Типы блокировок	
	Параметры deadlock_timeout и log_lock_waits	
	Параметр lock_timeout	
	Подтранзакции	
	Мультитранзакции	
	Быстрый путь блокирования (fastpath)	
	Сильные и слабые блокировки таблиц	
	Справочник устанавливаемых командами блокировок	
	Секции таблицы блокировок	
	Транши блокировок (tranches)	
	Легковесные блокировки	

	Блокирование по быстрому пути и 16 блокировок	
	Индексы соединения секции и быстрый путь	
	Параметр join_collapse_limit	
	Представление pg_locks	
	Параметр track_commit_timestamp	
	Практика	
10	**Буферный кэш**	250
	Структуры памяти обслуживающие буферный кэш	
	Структуры памяти обслуживающие буферный кэш (продолжение)	
	Поиск свободного буфера	
	Алгоритм вытеснения грязных буферов	
	Стратегии замены буферов	
	Поиск блока в буферном кэше	
	Закрепление буфера (pin) и блокировка content_lock	
	Освобождение буферов при удалении файлов	
	Оптимизированное расширение файлов	
	Изменение размера файлов и буферный кэш	
	Предварительное чтение блоков (prefetch)	
	Представление pg_stat_recovery_prefetch	
	Расширение pg_prewarm	
	Процесс фоновой записи bgwriter	
	Алгоритм очистки кэша буферов процессом bgwriter	
	Представление pg_stat_bgwriter	
	Расширение pg_buffercache	
	Настройка размера кэша буферов	
	Параметр synchronize_seqscans	
	Практика	
11	**Контрольная точка**	273
	Контрольная точка	
	Шаги выполнения контрольной точки	
	Шаги выполнения контрольной точки (продолжение)	
	Параметры конфигурации процесса checkpointer	
	Статистика для настройки параметров checkpointer	
	Пример настройки параметров checkpointer	
	Пример настройки параметров checkpointer (продолжение)	
	Практика	
12	**Автовакуум**	282
	Алгоритм вакуумирования	
	Первая фаза вакуумирования	
	Расчёт памяти под TID для вакуумирования	
	Вторая и третья фазы вакуумирования	
	Четвертая и пятая фазы вакуумирования	
	Агрессивный (Aggressive) режим вакуумирования	
	Заморозка строк (FREEZE)	
	Вакуум в 17 версии PostgreSQL	
	Сравнительное тестирование вакуума 16 и 17 версий PostgreSQL	
	Контрольные суммы и WAL	
	Параметры команды VACUUM	
	Параметры команды VACUUM (продолжение)	
	Расширение pg_vsibility	
	Мониторинг автовакуума	
	Представление pg_stat_progress_vacuum	
	Параметр log_autovacuum_min_duration	

	Параметры конфигурации автовакуума	
	Настройка автовакуума	
	Параметр autovacuum_naptime	
	Выбор таблиц автовакуумом	
	Рекомендации по настройке автовакуума	
	Важность наблюдения за горизонтом баз данных	
	Мониторинг горизонта баз данных	
	Параметры автовакуума на уровне таблиц	
	Параметр default_statistics_target	
	Раздувание (bloat) таблиц и индексов	
	Практика	
13	**Использование диагностического журнала**	313
	Диагностический журнал	
	Параметры диагностики	
	Отслеживание использования временных файлов	
	Отслеживание работы автовакуума и автоанализа	
	Наблюдение за контрольными точками	
	Описание записей log_checkpoints	
	Описание записей log_checkpoints (продолжение)	
	Утилита pg_waldump и записи log_checkpoints	
	Утилита pg_waldump и записи log_checkpoints	
	Диагностика частоты соединений	
	Диагностика блокирующих ситуаций	
	Практика	
14	**Накопительная статистика**	326
	Накопительная статистика	
	Утилита pg_test_timing	
	Просмотр статистики работы процессов	
	Представление pg_stat_database	
	Прогресс выполнения команд	
	Представление pg_stat_io	
	Статистики buffers_backend_fsync и fsyncs	
	Строки представления pg_stat_io	
	Характеристики pg_stat_io	
	Статистики представления pg_stat_io	
	Представления pg_statio_all_tables и pg_statio_all_indexes	
	Представление pg_stat_all_tables	
	Представление pg_stat_all_indexes	
	Длительность удержания горизонта баз данных	
	Представление pg_stat_wal	
	Расширение pg_walinspect	
	Использование расширения pg_walinspect	
	Представление pg_stat_activity	
	Блокирующие процессы и функция pg_blocking_pids()	
	pg_cancel_backend() и pg_terminate_backend()	
	Практика	
15	**Расширения pg_stat_statements и pg_stat_kcache**	348
	Расширение pg_stat_statements	
	Конфигурация pg_stat_statements	
	Параметры конфигурации pg_stat_statements	
	Представление pg_stat_statements	
	Запросы к представлению pg_stat_statements	
	Примеры запросов к представлению pg_stat_statements	
	Метрики pg_stat_statements	
	Примеры метрик представления pg_stat_statements	
	Расширение pg_stat_kcache	
	Статистики собираемые pg_stat_kcache	
	Просмотр статистик pg_stat_kcache	

	Практика	
16	**Расширение pg_wait_sampling**	362
	Расширение pg_wait_sampling	
	История событий ожидания	
	История событий ожидания (продолжение)	
	Параметры расширения pg_wait_sampling	
	Профиль pg_wait_sampling	
	Запросы к профилю pg_wait_sampling	
	Запросы к профилю pg_wait_sampling (продолжение)	
	Сброс статистик	
	Практика	

Авторские права

Создал: Олег Иванов

Создан: **5 марта 2025 г.**
По вопросам обучения обращайтесь: **edu@tantorlabs.ru**

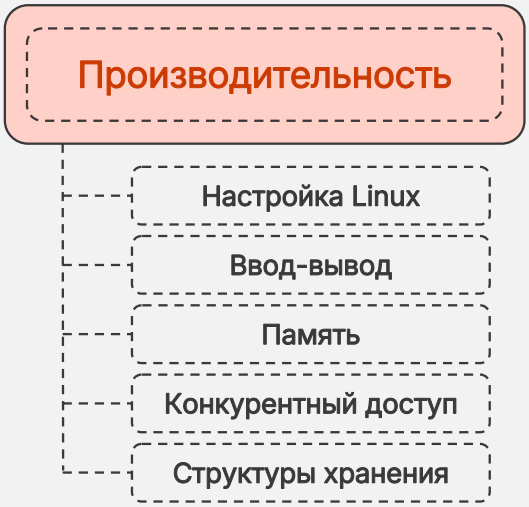
1

Введение

Настройка производительности

Обзор

- в курсе изучаются:
  - настройка операционной системы Linux
  - настройка экземпляра, обслуживающего кластер баз данных
  - оптимизация структур хранения
- в курсе не изучается:
  - настройка кода SQL



Обзор

Курс предназначен для администраторов и разработчиков, в задачи которых входит настройка производительности работы приложений с СУБД PostgreSQL.

Кластер баз данных PostgreSQL состоит из набора файлов, хранящихся в файловой системе в директории (на директорию указывает переменная окружения `PGDATA`). Клиентов (пользователей) обслуживает компьютерная программа, которая называется СУБД. Запущенная программа называется экземпляром PostgreSQL. Экземпляр, обслуживающий кластер баз данных, это набор процессов в операционной системе и оперативная память, используемая процессами. Типы аппаратных ресурсов, используемые экземпляром:

1) "ввод-вывод" (дисковая подсистема, диск, система хранения, storage)
2) "память" (оперативная память)
3) "процессор" (ядра центральных процессоров)
4) сеть (сетевые интерфейсы).

Ресурсы могут быть более нагружены (дефицитны) или один из ресурсов может стать узким местом ("бутылочным горлышком", bottleneck), которое будет определять производительность всего экземпляра.

Процессы экземпляра используют ресурсы совместно и узким местом может стать конкуренция за получение доступа к ресурсу (resource starvation). Поэтому отдельно уделяется внимание блокировкам и событиям ожидания получения доступа к ресурсу.

Экземпляр обслуживает приложения в режиме клиент-сервер: приложения передают команды SQL, процессы экземпляра выполняют команды SQL и возвращают результат.

SQL - декларативный язык. Это означает, что команды SQL описывают то, какой результат хочется получить ("что"), а не то, каким способом достичь результата ("как"). Способов выполнения команд может быть несколько. Например, полный просмотр таблицы или просмотр индекса. Способы отличаются по использованию ресурсов. Выбор способа (плана) выполнения команд SQL тоже относится к настройке производительности и называется "настройка кода SQL" (SQL tuning, query performance tuning). К этой теме стоит переходить после настройки производительности экземпляра и оптимизации структур хранения данных (таблиц и индексов), которые рассматриваются в этом курсе.

https://en.wikipedia.org/wiki/Starvation_(computer_science)

О компании Тантор

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

О компании Тантор

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

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

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

В конце 2022 года компания вошла в Группу Компаний "Астра".

СУБД Tantor

Tantor BE

Новые возможности и доработки по сравнению с PostgreSQL, техническая поддержка

Tantor SE

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

Tantor SE 1C

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

Tantor PipelineDB

Расширение, позволяющее непрерывно обрабатывать данные

В составе Tantor xData

Максимальная версия СУБД, оптимизированная для работы с 1С

СУБД Tantor

СУБД Tantor это реляционная база данных семейства PostgreSQL с повышенной производительностью и стабильностью. Выпускается в нескольких редакциях (сборках): BE, SE, SE1C, Certified. Редакция Special Edition для наиболее нагруженных OLTP-систем и хранилищ данных размерами до 100ТБ. Редакция Special Edition 1C для приложений 1С.

Для всех редакций доступна техническая поддержка, помощь в построении архитектурных решений, миграции с СУБД других производителей (импортозамещение). Программное обеспечение Тантор Лабс включено в "Единый реестр российских программ для электронных вычислительных машин и баз данных". При приобретении СУБД Тантор предоставляется лицензия на Платформу Тантор для управления приобретенными СУБД.

Tantor xData

- программно-аппаратный комплекс, с высокой производительностью, отказоустойчивостью, безопасностью
- возможности DBaaS в ЦОД предприятия
- Улучшенная автоматизация и резервное копирование
- высокая производительность и масштабируемость
- снижение затрат на инфраструктуру и администрирование
- в состав входит СУБД Tantor и Платформа Tantor

Tantor xData

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

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

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

Предназначен для создания облачных платформ.

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

Tantor PipelineDB

- расширение для СУБД Tantor и PostgreSQL с открытым исходным кодом для непрерывного выполнения SQL-запросов на потоках данных с инкрементальным сохранением результатов в обычных таблицах
- высокопроизводительное агрегирование временных рядов
- позволяет соединять потоковые данные с историческими данными для сопоставления в режиме реального времени
- может использоваться в приложениях, где нужна незамедлительная реакция
- пример непрерывного представления для вывода ежесуточного трафика, используемого топ десятью ip-адресов:

```
CREATE VIEW heavy_hitters AS
 SELECT day(arrival_timestamp), topk_agg(ip, 10, response_size)
 FROM requests_stream GROUP BY day
```


Tantor PipelineDB

Tantor PipelineDB - расширение для СУБД Tantor или PostgreSQL, выпущенное в 2024 году, с открытым исходным кодом. Позволяет непрерывно обрабатывать потоковые данные с инкрементальным сохранением результатов в таблицах. Данные обрабатываются в реальном времени, используя только запросы SQL. Имеет большое количество аналитических функций, работающих с постоянно обновляемыми данными. Позволяет соединять потоковые данные с историческими данными для сопоставления в режиме реального времени. Устраняет необходимость использовать традиционную логику ETL (Extract, Transform, Load) с CDC (Change Data Capture). Дальше описана суть расширения для знакомых с термином "CDC".

Tantor PipelineDB добавляет поддержку непрерывных представлений. Непрерывные представления - это материализованные представления с высокой скоростью обновления, инкрементально обновляемые в режиме реального времени.

Запросы к непрерывным представлениям **моментально** выдают **актуальный** результат. Это **позволяет использовать TantorPipelineDB в классе приложений, где важна незамедлительная реакция**.

Примеры создания непрерывных представлений:

Непрерывное представление для выдачи аналитических данных за последние пять минут:

```
CREATE VIEW imps WITH (action=materialize, sw = '5 minutes')
 AS SELECT count(*), avg(n), max(n) FROM imps_stream;
```

По умолчанию параметр `action=materialize`, поэтому параметр `action` можно не указывать при создании непрерывных представлений.

Непрерывное представление для вывода девяностого, девяносто пятого, девяносто девятого процентилей времени отклика:

```
CREATE VIEW latency AS
 SELECT percentile_cont(array[90, 95, 99])
 WITHIN GROUP (ORDER BY latency::integer)
 FROM latency_stream;
```

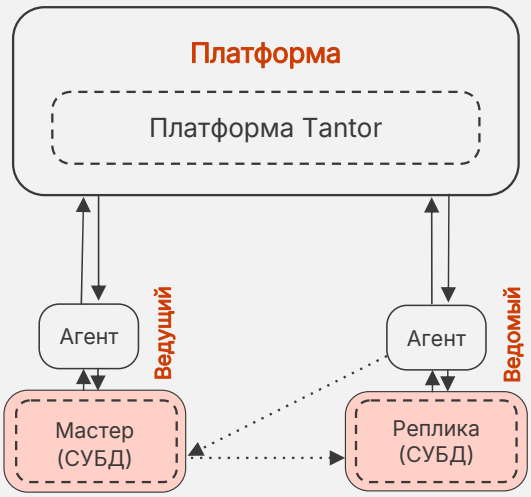
Непрерывное представление для вывода ежесуточного трафика, используемого топ десятью ip-адресов:

```
CREATE VIEW heavy_hitters AS
 SELECT day(arrival_timestamp), topk_agg(ip, 10, response_size)
 FROM requests_stream GROUP BY day;
```

https://tantorlabs.ru/products/pipelinedb

Платформа Tantor

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



Платформа Tantor

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

Преимущества использования Платформы Tantor:

1. Сбор показателей работы экземпляров PostgreSQL, хранение и обработка показателей, рекомендации по настройке производительности
2. Интуитивно понятный и функциональный графический интерфейс позволяет сосредоточиться на показателях работы экземпляров PostgreSQL
3. Автоматизирует рутинные задачи, повышая эффективность работы и снижая вероятность ошибок
4. Управляет не только СУБД Тантор, но и другими СУБД семейства PostgreSQL
5. Интеграция с почтовыми системами, службами каталогов, мессенджерами
6. Простое внедрение: развертывание и ввод СУБД под обслуживание Платформой Tantor с помощью Ansible.

Платформа Tantor DLH

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

О курсе

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

О курсе

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

Примерное расписание:

1) начало в 10:00

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

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

Курс состоит из теоретической части (глав) и практических заданий. Длительность глав примерно 20-40 минут. Точное время начала изложения глав и времени на практические задания определяет инструктор. Длительность выполнения упражнений может отличаться у разных слушателей и это не влияет на эффективность усвоения материала курса. Доделывать упражнения можно в перерывах между главами или в конце каждого дня. Порядок следования глав и упражнений на эффективность усвоения материала курса не влияет. Проверка выполнения заданий не производится. Для успешного усвоения материала курса достаточно:

1) слушать инструктора, просматривая в процессе изложения инструктора текст на слайдах и под слайдом

2) задавать инструктору вопросы, если возникает внутреннее несогласие (появляются вопросы)

3) выполнять практические задания и читать текст в практических заданиях

В материалы курса входят:

1) учебник в формате `pdf`

2) практическит задания в формате `pdf`

3) образ виртуальной машины в формате `ova`

Общие понятия настройки производительности

- цели настройки производительности:
  - при вводе в эксплуатацию, разработке, приложения, выборе оборудования
  - миграции с другой СУБД
  - в процессе эксплуатации при обнаружении снижения целевых показателей
- настройка производительности включает в себя:
  - целевые показатели (метрики), которые описывают ожидания пользователей приложений по качеству обслуживания. Показатели могут быть указаны в соглашении об уровне обслуживания
  - процедуры получения показателей (мониторинг)
  - инструменты настройки производительности: утилиты, расширения, функции, командные файлы.

Общие понятия настройки производительности

Потребность в настройке производительности возникает:
1) при вводе в эксплуатацию программы, разработке приложения, замене оборудования
2) при миграции с предыдущей версии СУБД или СУБД другого производителя
3) в процессе эксплуатации при обнаружении снижения целевых показателей. В этом случае имеется "базовый уровень показателей производительности" (baseline), когда приложение работает без нареканий пользователей этого приложения. Базовые показатели определяют значения метрик, по достижении которых можно завершить настройку производительности. Базовые показатели помогают быстрее выяснить, что поменялось в конфигурации программной системы из-за чего производительность ухудшилась.

Типы настройки производительности:
1) предупреждающий, ещё до возникновения проблем с производительностью (proactive)
2) для устранения проблем с производительностью (reactive)

Настройка производительности включает в себя:
1) Соглашение о качестве обслуживания (service level agreement, SLA): целевые показатели (метрики), которые описывают ожидания пользователей приложений по качеству обслуживания
2) Повседневное наблюдение (мониторинг) за целевыми показателями, определяющими качество обслуживания
3) Инструменты настройки производительности: утилиты, расширения, функции, командные файлы.

Источник целевых показателей:
1) стандарты качества обслуживания
2) техническое задание, на основе которого разрабатывалось приложение или технические характеристиками программно-аппаратных комплексов или технические требования к программе
3) существующий SLA, который используется при миграции приложения с СУБД другого производителя.

Формулировки целевых показателей могут быть широкими. Например: доступность 99% времени; 90% запросов должны выполняться не дольше 10 секунд. Показатели напрямую не связаны с подсистемой СУБД, проблемы с производительностью работы которой не позволяют достичь целевых показателей.

Методология настройки производительности

- не зависит от используемых инструментов
- включает в себя:
  - оценку архитектуры приложения: как приложение взаимодействует с базой данных
- ошибки на уровне архитектуры приложения создают узкие места и определяют ограничения в рамках которых возможна настройка производительности
- если в архитектуру приложения можно внести изменения, устраняющие узкие места, то достигается наибольший эффект

Методология настройки производительности

Методология (последовательность действий) не зависит от используемых инструментов. Можно совместно использовать Платформу Тантор, расширения PostgreSQL, команды SQL.

Методология включает в себя следующие шаги:

1) Оценка архитектуры приложения: как приложение взаимодействует с базой данных. Шаг выполняется один раз.

Примеры:

a) приложение работает через кэширующие решения типа ключ-значение (Valkey, Redis), активно использует временные таблицы;

b) в приложении есть таблицы, в которых часто обновляются строки. В СУБД PostgreSQL обновления (команда UPDATE) порождают устаревшие строки. Вставка строк (INSERT) не порождает устаревшие строки. Приложение, которое преимущественно вставляет, а не обновляет строки, создаёт меньшую нагрузку на СУБД.

c) приложение хранит данные в формате json, а не в скалярных типах данных;

d) приложение использует асинхронную обработку данных в СУБД, а не на промежуточном уровне (сервере приложений). СУБД PostgreSQL обслуживается только одним экземпляром, а серверов приложений может быть несколько. На уровне серверов приложений возможно относительно просто перераспределить нагрузку на несколько серверов.

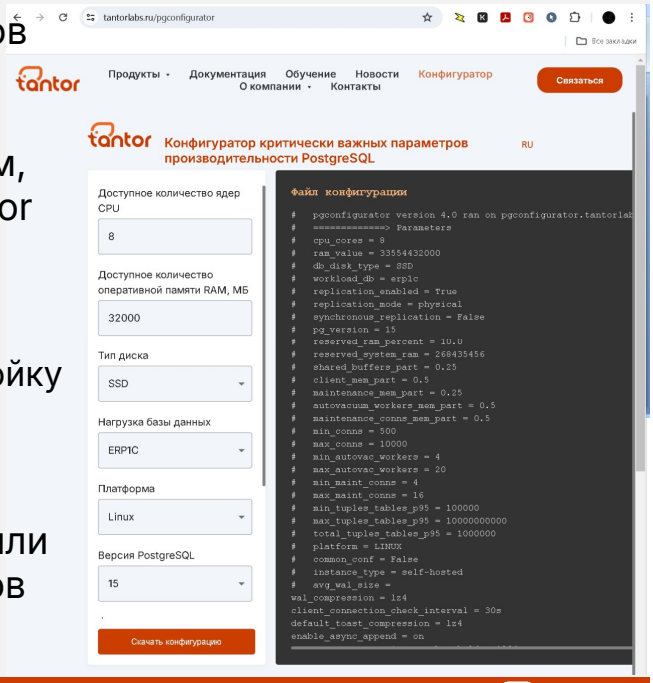
e) использование в качестве первичных ключей uuid, а не целочисленных типов. Первичные ключи типа int4 и int8 заполняются монотонно возрастающей последовательностью, а uuid генерируются случайным образом (за исключением генерируемых функцией uuidv7() расширения pg_uuidv7, имеющегося в СУБД Tantor 16.6). Вставка записей в индекс первичного ключа различается по эффективности.

f) большое количество индексов и секций таблиц, которые использовались до миграции на PostgreSQL с СУБД других производителей. В разных СУБД есть особенности, которые влияют на производительность и в разных СУБД оптимальное с точки зрения производительности число секций таблиц и количество индексов на одну таблицу может различаться.

Ошибки на уровне архитектуры приложения создают узкие места и определяют ограничения в рамках которых возможна настройка производительности. Если в архитектуру приложения можно внести изменения устраняющие узкие места, то достигается наибольший эффект. Например, перенос долгих запросов на физическую реплику существенно разгружает основную СУБД (master, primary).

Последовательность действий по настройке производительности

- настройка параметров хранения файлов кластера баз данных
- проверка параметры конфигурации экземпляра на соответствие значениям, рекомендуемым конфигуратором Tantor или Платформы Tantor
  - Конфигуратор Tantor (http://tantorlabs.ru/pgconfigurator) позволяет выполнить начальную настройку параметров конфигурации СУБД PostgreSQL
- после тестовой нагрузки или при эксплуатации приложения проверить или провести тонкую настройку параметров конфигурации экземпляра



Последовательность действий по настройке производительности

Например, в процессе миграции с СУБД других типов можно уменьшить число секций или переносить большие объекты (large objects, LOB) не в созвучный тип данных "большие объекты" (использующие для хранения одну таблицу на базу данных `pg_largeobject`), а в файлы в файловой системе или столбцы типов `bytea` или `text`.

2) Если настройка производительности выполняется впервые или после существенного изменения оборудования, то стоит проверить и настроить операционную систему. Экземпляр PostgreSQL работает с аппаратными ресурсами не на низком уровне (прямой ввод-вывод, direct i/o), а через страничный кэш операционной системы и поэтому на работу экземпляра оказывают существенное влияние настройки операционной системы. На этом шаге проверяется наличие типичных проблем, не относящиеся к СУБД: нехватка места на диске, использование неподходящих файловых систем, параметры их монтирования.

3) До настройки экземпляра стоит настроить структуры хранения. Если есть системы хранения (storage) с разными характеристиками, то разместить директории PGDATA оптимальным образом. Системы хранения обычно нечасто меняются, имеют точные характеристики (объем места, скорость чтения и записи, количество операций в секунду) и оптимизация структур хранения выполняется однократно. Например, директорию `PGDATA/pg_wal` можно разместить на отдельной файловой системе и создать табличное пространство для временных объектов

4) Выполнить первичную "настройку экземпляра": проверить установлены ли параметры конфигурации экземпляра в близкие к оптимальным значения. Параметров сотни и они связаны друг с другом: изменение значения одного параметра может сдвинуть диапазон оптимальности другого параметра. Поэтому настройка итеративна. В первой итерации проверяют, что значения установлены в рекомендуемые для планируемой нагрузки и возможностях оборудования. Значения можно получить используя конфигуратор Tantor (https://tantorlabs.ru/pgconfigurator), рекомендации из документации PostgreSQL. Пример: при большом объеме физической памяти установить размер кэша буферов примерно в четверть от объема физической памяти

5) Вторая итерация настройки параметров конфигурации экземпляра выполняется на основе результатов тестовой нагрузки или в процессе эксплуатации приложения. Во второй итерации настраивают параметры подсистем экземпляра в порядке, который уменьшает вероятность того, что после настройки какой-то подсистемы экземпляра придется возвращаться к тому, что уже настраивалось.

Последовательность действий по настройке производительности

- определение целевые показателей, по достижении которых прекращают настройку производительности
- выбор подсистемы экземпляра, усилия по настройке которой дадут наибольший эффект
  - подсистема, которая обслуживает наиболее дефицитный ресурс является узким местом
- измерение показателей производительности, относящиеся к подсистеме
- внесение изменений в параметры конфигурации, относящиеся к подсистеме
  - если показатели улучшились, то изменения значений параметров были успешны
- при достижении целевых показателей настройка завершается

Последовательность действий по настройке производительности

Следующие шаги выполняются при эксплуатации приложения.

6) Определяются цели настройки и предполагаемая трудоемкость (анализ "cost to effect"). Например, определяются целевые показатели по достижении которых стоит прекратить настройку производительности. Целевыми показателями могут быть метрики из SLA, базовые показатели. значения параметров, когда приложение работало с приемлемой производительностью. После устранения узких мест настройка производительности обычно даёт меньший эффект.

Устранение проблем (troubleshooting) и настройка производительности (performance tuning) немного отличаются. При устранении проблем, частью которых могут быть проблемы с производительностью, деградация производительности обычно резкая, а не постепенная. Обычно, причина в недавних изменениях, внесенных в систему: установка обновлений, изменение топологии (например, уменьшение числа физических реплик). При поиске проблем исключают задержки на стороне сервера приложений, так как причиной проблем может быть код приложения, а не СУБД.

7) Выбирается область (подсистема экземпляра) с наибольшим потенциалом по настройке производительности. Если есть узкое место, то его устранение даст наибольший результат от усилий по настройке. Узкое место это наиболее дефицитный ресурс. Например: центральные процессоры. **Уровень нагрузки центральных процессоров считается высокой, если загрузка процессоров больше ~90%** (граница колеблется от 85% до 95%). Узким местом могут быть блокировки, удерживаемые дольше, чем предполагается для конкретного типа блокировок. Например, легковесные блокировки должны удерживаться несколько десятков инструкций процессора. Узким местом может стать конкуренция за доступ к буферам, занимаемыми блоками таблиц, к которым часто обращается приложение ("горячий" объект).

8) Измеряются показатели, относящиеся к подсистеме ("gather evidence"). Показатели могут сравниваться с базовыми показателями, либо с показателями аналогичных эксплуатируемых СУБД.

9) Вносятся изменения в параметры подсистемы. Например, меняются параметры конфигурации автовакуума.

10) Если целевые показатели улучшились, то продолжают увеличивать или уменьшать значения параметров. Если показатели ухудшились, то прекращают вносить изменения. Если изменения незначительные, то выбирают другую подсистему. При достижении целевых показателей настройка производительности прекращается.

Пример зачем в 6 пункте указано "При поиске проблем исключают задержки на стороне сервера приложений, так как причиной проблем может быть код приложения, а не СУБД**".**

Разработчик веб-приложения проводил нагрузочное тестирование экземпляра PostgreSQL, чтобы определить максимальное число запросов, которое сможет обработать экземпляр. Запросы были простые: SELECT одной строки по индексу, аналогичный тому, который используется в утилите pgbench. При 20000 "одновременных запросов", которые посылало приложение, используя пул из 64 соединений к базе данных, длительность каждого "запроса" занимало от 4 до 10 секунд со средним значением 4,56 секунд. Одновременными запросами считались запросы, которые находятся в ожидании с момента посылки клиентским кодом запроса на получение соединения из пула, до момента получения строки и возврата соединения в пул. Для конечного пользователя или утилиты нагрузочного тестирования веб-приложения, "подключение клиента" выглядело как время от нажатия кнопки на веб-страницы до отрисовки веб-страницы (или от момента посылки запроса до получения ответа по протоколу REST). Целью тестирования было определить сколько "подключений клиентов" ("одновременных запросов") может выдержать приложение.

Тестировался пул из 8,16,32,64,72,96 соединений. "Подключения клиентов" разработчик называл "количеством пользователей, которые пытаются одновременно использовать базу данных". Разработчик обнаружил, что когда количество пользователей, которые пытаются одновременно использовать базу данных, невелико, то меньшее количество подключений (или всего одно, если пользователей всего несколько) работает намного лучше. При достижении 10 000 "одновременных запросов", большее количество подключений в пуле соединений работало лучше и приводило к лучшей производительности. Но только до определенного количества подключений (зависело от мощности хоста). Например, на макбуке разработчика с 32Гб оперативной памяти наилучшую производительность давал пул из 64 сессий. Большее или меньшее число соединений приводило к снижению производительности. Разработчик обнаружил, что изменение параметров конфигурации PostgreSQL, таких как: `shared_memory_size`, `shared_buffers`, `effective_cache_size`, `maintenance_work_mem`, `checkpoint_completion_target`, `wal_buffers`, `random_page_cost`, `work_mem`, `max_wal_size`, `max_worker_processes`, `max_parallel_workers` не влияло на длительность выполнения "запросов" и СУБД PostgreSQL работал так же, как и с настройками по умолчанию. Список параметров разработчик взял из "достоверных" источников. До использования "хинтов" не дошёл. Разработчик не был администратором и не стал задаваться риторическим вопросом "где же трассировка" и не успел подключиться отладчиком к каким-нибудь процессам и даже в LWLocki не углубился, чем сэкономил себе время. Разработчик не был из большой компании и не стал думать, что ему срочно нужна "кластеризация", greenplum, stolon, master-master. Разработчик пришел к разумному выводу, что единственное, что помогло справиться с высокой нагрузкой, это кэширование с помощью Redis и других средств для сокращения числа запросов к базе данных.

Всё знать нельзя, но нужно стремиться повышать квалификацию обучаясь и задавая вопросы. Разработчик задал вопрос в форуме и ему дали следующую рекомендацию. Исходя из представленных данных для каждого "запроса" одновременно с ним имеется 20 000/64=312 запросов. Предположим, что переключение соединений/контекстов в пулере соединений сервера приложений, посылку запроса, ожидание получения результата запроса, возврат результатов занимает 10 миллисекунд (0.01 секунды). Это означает, что в среднем, запросы ждут 0,01 * 312 = 3,12 секунды, что соответствует среднему времени выполнения запроса (не к базе данных, а полный запрос) в 4,56 секунд. Это означает, что основные задержки происходят на сервере приложений и **СУБД в его случае не является узким местом**. СУБД не требует настройки, в чём уже убедился разработчик меняя параметры конфигурации. Также разработчику рекомендовали проверить используются ли подготовленные запросы. Подготовленные запросы позволяют кэшировать план выполнения запроса в памяти серверного процесса и уменьшить время на планирование. tps=312 обычное значение для простых тестов на обычном оборудовании. При 312 запросах в секунду время на планирование соизмеримо с временем выполнения запроса и, возможно, используя подготовленные запросы удастся уменьшить задержку на планирование, но ожидаемое улучшение не слишком большое, примерно в 2 раза.

1-2

Использование утилиты pgbench

Бенчмаркинг

- бенчмаркинг - это процесс тестирования производительности оборудования, программного обеспечения или всей системы в целом
  - benchmark - критерий, ориентир
- универсальным показателем является число команд в секунду (transactions per second, tps), которые может обработать СУБД
  - если транзакция состоит из одной команды, то этот показатель называют число запросов в секунду (queries per second, qps)
  - используется для сравнения производительности до внесения изменений в конфигурацию СУБД и после внесения изменений
- для измерения показателя нужны:
  - набор команд, который будет выполняться в транзакции
  - таблицы с данными и другие объекты (индексы, ограничения целостности, последовательности), которые нужны для выполнения команд
  - число сессий, в которых параллельно будут выполняться транзакции

Бенчмаркинг

Бенчмаркинг (benchmark - критерий, ориентир) это процесс тестирования производительности оборудования, программного обеспечения или всей системы в целом. В процессе настройки производительности измеряют показатели производительности. Если настраивать производительность экземпляра в целом, какой показатель использовать?

Простым показателем является число команд в секунду (transactions per second, tps), которые может обработать СУБД. При подсчете tps могут использоваться условия. Например, учитывать только транзакции, которые были выполнены за 5 или 20 секунд. Условия могут отсутствовать, в этом случае команды посылаются с максимальной скоростью. Такой тест называется нагрузочным.

Если транзакция состоит из одной команды, то этот показатель называют число запросов в секунду (queries per second, qps). TPS используется для сравнения производительности до внесения изменений в конфигурацию СУБД и после внесения изменений. Для измерения TPS нужны:

1) набор команд, который будет выполняться в транзакции

2) таблицы с данными и другие объекты (индексы, ограничения целостности, последовательности), которые нужны для выполнения команд

3) число сессий, в которых параллельно будут выполняться транзакции.

Чтобы значения tps можно было сравнивать, вышеперечисленные характеристики должны быть одинаковы.

Для быстрой проверки тестирование должно выполняться быстро, а утилита тестирования должна быть простой в использовании. В PostgreSQL имеется утилита командной строки `pgbench`. Утилита максимально проста в использовании.

Для создания или пересоздания таблиц с данными достаточно выполнить команду:

```
pgbench -i
```

Тестирование в течение 30 секунд с выводом промежуточных результатов каждые 5 секунд запускается командой:

```
pgbench -T 30 -P 5
```

Будет выдан результат работы и основной показатель это **tps, число транзакций в секунду**:

```
latency average = 1.687 ms
latency stddev = 0.225 ms
initial connection time = 3.788 ms
tps = 590.180430 (without initial connection time)
```

Результат бенчмаркинга

- основное это tps в последней строке
- стандартное отклонение выводится для задержки (latency)
- параметр -P позволяет выдавать текущие значения tps и latency

```
pgbench -T 30 -P 5
starting vacuum...end.
progress: 5.0 s, 582.0 tps, lat 1.709 ms stddev 0.252, 0 failed
progress: 10.0 s, 597.0 tps, lat 1.667 ms stddev 0.199, 0 failed
progress: 15.0 s, 596.0 tps, lat 1.670 ms stddev 0.274, 0 failed
progress: 20.0 s, 581.8 tps, lat 1.712 ms stddev 0.186, 0 failed
progress: 25.0 s, 601.4 tps, lat 1.655 ms stddev 0.206, 0 failed
progress: 30.0 s, 582.4 tps, lat 1.710 ms stddev 0.213, 0 failed
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 17704
number of failed transactions: 0 (0.000%)
latency average = 1.687 ms
latency stddev = 0.225 ms
initial connection time = 3.788 ms
tps = 590.180430 (without initial connection time)
```


Результат бенчмаркинга

На что смотреть в результатах pgbench? Пример результата команды:

```
pgbench -T 30 -P 5
pgbench (17.0)
starting vacuum...end.
progress: 5.0 s, 582.0 tps, lat 1.709 ms stddev 0.252, 0 failed
progress: 10.0 s, 597.0 tps, lat 1.667 ms stddev 0.199, 0 failed
progress: 15.0 s, 596.0 tps, lat 1.670 ms stddev 0.274, 0 failed
progress: 20.0 s, 581.8 tps, lat 1.712 ms stddev 0.186, 0 failed
progress: 25.0 s, 601.4 tps, lat 1.655 ms stddev 0.206, 0 failed
progress: 30.0 s, 582.4 tps, lat 1.710 ms stddev 0.213, 0 failed
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 17704
number of failed transactions: 0 (0.000%)
latency average = 1.687 ms
latency stddev = 0.225 ms
initial connection time = 3.788 ms
tps = 590.180430 (without initial connection time)
```

Основное это tps в последней строке. При повторных запусках это значение будет меняться, то есть у значения есть разброс. Показатели разброса для tps не выводятся. Стандартное отклонение выводится для задержки (latency). В примере "точность" latency: 0.225/1.687*100=13.33%. Грубо говоря, такая же точность и у tps. Зачем нужна точность?

Если настраивать производительность и измерять tps до и после настройки, то если два tps различаются в пределах отклонения, то настройка не повлияла на работу СУБД.

Также удобно использовать параметр -P. Утилита pgbench будет выдавать текущие значения tps и latency. Визуально можно увидеть какой разброс в значениях tps.

Удобство в том, что можно запустить pgbench и пока он работает менять параметры экземпляра и наблюдать за tps.

Из-за чего возникает разброс? Из-за активности процессов в операционной системе. Например, завершается контрольная точка или запустился автовакуум.

pgbench - утилита бенчмаркинга PostgreSQL

- утилита командной строки, стандартно поставляемая с PostgreSQL
- используется для создания тестовой нагрузки при настройке производительности
- встроенные тесты используют четыре таблицы:
  - `pgbench_accounts` (100тыс.строк), `pgbench_tellers` (10 строк), `pgbench_branches` (1 строка), `pgbench_history` (0 строк)

```
create table pgbench_history (tid int, bid int, aid int, delta int, mtime timestamp);
create table pgbench_tellers (tid int primary key, bid int, tbalance int, filler char(84));
create table pgbench_accounts (aid int primary key, bid int, abalance int, filler char(84));
create table pgbench_branches (bid int primary key, bbalance int, filler char(88));
```

- таблицу `pgbench_accounts` можно сделать секционированной

```
pgbench -i -s 100
dropping old tables...
creating tables...
generating data (client-side)...
10000000 of 10000000 tuples (100%) done (elapsed 39.06 s, remaining 0.00 s))
vacuuming...
creating primary keys...
done in 85.48 s (drop tables 0.01 s, create tables 0.01 s, client-side generate 39.29 s,
 vacuum 10.91 s, primary keys 35.27 s).
```


pgbench - утилита бенчмаркинга PostgreSQL

Утилита используется не только для простого теста по умолчанию. pgbench - это качественный и простой инструмент для запуска произвольных транзакций.

Встроенные тесты используют таблицы, создаваемые параметром "`-i`". По умолчанию создаётся **четыре** таблицы `pgbench_accounts` (100тыс.строк), `pgbench_tellers` (10 строк), `pgbench_branches` (1 строка), `pgbench_history` (0 строк) :

```
create table pgbench_history (tid int, bid int, aid int, delta int, mtime timestamp);
create table pgbench_tellers (tid int primary key, bid int, tbalance int, filler char(84));
create table pgbench_accounts (aid int primary key, bid int, abalance int, filler char(84));
create table pgbench_branches (bid int primary key, bbalance int, filler char(88));
```

Что можно поменять в тестовых данных?

1) Параметром `-F` можно задать процент заполнения блоков (`fillfactor`) для **трёх** таблиц `pgbench_accounts, pgbench_tellers и pgbench_branches`. По умолчанию у всех таблиц `fillfactor=100%`.

2) Параметр `-s` (scale factor) задаёт во сколько раз увеличить количество строк в таблицах. Если `-s` 20000 или больше, то столбцы `aid` будут иметь тип `int8`, а не `int4`. Пример:

```
pgbench -i -s 1 -F 100
```

3) По умолчанию внешние ключи не создаются.

Можно добавить внешние ключи командой:

```
pgbench -i -I f
creating foreign keys...
```

4) Можно указать какие действия нужно выполнить при создании таблиц:

```
pgbench -i -I dtgvpf
dropping old tables...
creating tables...
generating data (server-side)...
vacuuming...
creating primary keys...
creating foreign keys...
```

5) Таблицу `pgbench_accounts` можно сделать секционированной параметрами:

`--partitions=число_секций`

`--partition-method=range` или `hash`

Полный список параметров утилиты pgbench можно посмотреть в документации:
https://docs.tantorlabs.ru/tdb/ru/16_4/se/pgbench.html

Три встроенных теста pgbench

- по умолчанию pgbench запускает тест TPC-B из 7 команд в одной транзакции:

```
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
 CURRENT_TIMESTAMP);
END;
```

- TPC-B является простым тестом и не соответствует нагрузке, создаваемой типичными приложениями, работающими с СУБД
- список встроенных тестов:

```
pgbench -b list
Available builtin scripts:
      tpcb-like: <builtin: TPC-B (sort of)>
  simple-update: <builtin: simple update>
    select-only: <builtin: select only>
```

- запуск теста select-only:

```
pgbench -b select-only -T 10 -P 3
```


Три встроенных теста pgbench

По умолчанию pgbench запускает тест примерно соответствующий TPC-B, который состоит из семи команд в одной транзакции:

```
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
 CURRENT_TIMESTAMP);
END;
```

Переменные подстановки заполняются случайными значениями.

Тест является слишком простым и не соответствует нагрузке, создаваемой типичными приложениями, работающими с СУБД. Зачем тогда он нужен, почему он не сложный? **Тест позволяет определить максимально достижимый tps.**

Функция `CURRENT_TIMESTAMP` выдаёт значение на начало транзакции, эквивалент функций `transaction_timestamp()`, `now()` и отличается от функций `clock_timestamp()`, `statement_timestamp()`. При написании своих команд это стоит учитывать, чтобы получать то время, которое нужно. Часто используют функцию `now()` из-за того, что название функции короткое.

Список встроенных в pgbench тестов:

```
pgbench -b list
Available builtin scripts:
      tpcb-like: <builtin: TPC-B (sort of)>
  simple-update: <builtin: simple update>
    select-only: <builtin: select only>
```

Тест simple-update состоит из трёх команд:

```
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid,
:delta, CURRENT_TIMESTAMP);
```

select-only состоит из одного запроса:

```
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

Тесты выбираются параметром `-b`:

```
pgbench -b select-only -T 10 -P 3
```

Параметры запуска pgbench

`-P секунд` задаёт интервал в секундах через который утилита будет выводить строку со статистикой выполнения.
`-T секунд` длительность теста
`--protocol=prepared`. В режиме `prepared` утилита pgbench в каждой сессии передает команду один раз (подготавливает) и дальше передает только новые значения параметров и команду на выполнение

```
pgbench -T 10 -P 3
        ...
        progress: 10.0 s, 530.1 tps, lat 1.883 ms stddev 1.558, 0 failed
        progress: 20.0 s, 427.8 tps, lat 2.319 ms stddev 7.186, 0 failed
        progress: 30.0 s, 174.9 tps, lat 5.753 ms stddev 25.774, 0 failed
        ...
        latency average = 2.645 ms
        latency stddev = 11.181 ms
        ...
        tps = 377.649988 (without initial connection time)
```


Параметры запуска pgbench

`-P` секунд задаёт интервал в секундах через который утилита будет выводить строку со статистикой выполнения.
`-T` секунд задаёт длительность теста:

```
pgbench -T 30 -P 10
...
progress: 10.0 s, 530.1 tps, lat 1.883 ms stddev 1.558, 0 failed
progress: 20.0 s, 427.8 tps, lat 2.319 ms stddev 7.186, 0 failed
progress: 30.0 s, 174.9 tps, lat 5.753 ms stddev 25.774, 0 failed
...
latency average = 2.645 ms
latency stddev = 11.181 ms
...
tps = 377.649988 (without initial connection time)
```

Основной результат tps число транзакций в секунду. **latency** коррелирует с **tps** и указывает время выполнения транзакции или скрипта. В отличие от TPC-C тест TPC-B не имеет искусственных задержек, которые увеличивают число клиентов (потоков), усложняют логику тестирования и толкование результатов.TPC-B выполняет команды с максимальной скоростью и определяет лимиты системы.

По умолчанию команды выполняются без использования переменных привязки, сырым текстом. Каждая команда разбирается и строится план выполнения. Но можно использовать расширенный режим и режим с подготовленными командами задав параметр `--protocol=extended` или `prepared`. В режиме `prepared` утилита pgbench в каждой сессии передает команду один раз (подготавливает) и дальше передает только новые значения параметров и команду на выполнение. При этом используется кэшированный на уровне сессии план выполнения, что обычно быстрее, так как команда повторно не разбирается.

В режиме `extended` команда (включая имена параметров `$1, $2,...`) посылается на выполнение каждый раз. Этот режим не даёт преимуществ, только накладные расходы. Переменные привязки передаются в том же вызове, но отдельным параметром, так же как в утилите psql:

```
select $1 as id, $2 as s \bind 5 'b' \g
```

или

```
insert into t values($1, $2) \bind 5 'b' \g
```

Рекомендации по использованию pgbench

- `-c N` задаёт число параллельных сессий, в которых будут выполняться транзакции или скрипты. Сессии создаёт один процесс pgbench по умолчанию одним потоком. Параметром `-j N` можно указать количество потоков (threads)
- число сессий должно быть не меньше числа потоков
- при использовании `-c N` со встроенными тестами, таблицы желательно создать с параметром `-s M`, чтобы `M` было не меньше, чем N
- параметром `-f script.sql@10` можно посылать на выполнение собственный набор команд, сохранённый в файле. В этом случае "транзакцией" в отчёте pgbench будет считаться выполнение всех команд в файле

```
pgbench -T 10 -P 5 -s 3 -c 50 -j 80
pgbench: warning: scale option ignored, using count from pgbench_branches table (100)
scaling factor: 100
number of clients: 50
number of threads: 50
```


Рекомендации по использованию pgbench

По умолчанию pgbench создаёт одну сессию с базой данных. На практике СУБД обслуживают десятки или сотни сессий. Для определения того не повлияет ли настройка параметров СУБД на ее способность обслуживать большое число сессий используется параметр `-c N`. Параметр можно использовать и в других целях. Например, определения того сможет ли СУБД обслужить заданное число активных сессий и с каким tps. Для получения tps в одной сессии нужно разделить выдаваемый утилитой tps на число сессий.

Параметр `-c N` задаёт число параллельных сессий, в которых будут выполняться транзакции или скрипты. Сессии создаёт один процесс pgbench по умолчанию одним потоком. Параметром `-j N` можно указать количество потоков (threads). При большом количестве сессий процесс pgbench может стать узким местом, так как будет использовать одно ядро. Несколько потоков могут использовать несколько ядер процессоров. Число сессий должно быть не меньше числа потоков: `c>j`.

При использовании -c N со встроенными тестами, таблицы желательно создать с параметром -s M, чтобы M было не меньше, чем N, иначе на результат будет влиять ожидание получения блокировок на уровне строк, так как высока вероятность того, что команды UPDATE в разных сессиях столкнутся на одной и той же строке.

Если вы не помните с каким `-s M` создавали таблицы, то `M` равно количеству строк в таблице `pgbench_branches`. При тестировании `-s M` задавать нет смысла (`-s` надо задавать при создании таблиц, то есть с параметром -i), о чём будет выдано предупреждение:

```
warning: scale option ignored, using count from pgbench_branches table
```

На результаты влияет выполнение контрольных точек, удержание горизонта базы данных, запуск автовакуума.

Параметром `-f` файл можно посылать на выполнение собственный набор команд, сохранённый в файле. В этом случае "транзакцией" в отчёте pgbench будет считаться выполнение всех команд в файле.

Можно указать несколько скриптов и целое число, задающее весовой коэффициент: в какой пропорции будут запускаться скрипты:

```
pgbench -f a.sql@8 -f b.sql@2
```

Скрипт `a.sql` будет запускаться в 4 раза чаще, чем `b.sql`. По умолчанию коэффициент 1.

При использовании структуры таблиц, набора индексов, команд приближенных к реальному приложению pgbench позволяет качественно протестировать изменение параметров экземпляра.

Пример использования pgbench

- задача: проверить, что быстрее `count(*)`, `count(1)`, `count(pk)`
- создание таблицы с данными:

```
drop table if exists t;
create table t(pk bigserial, c1 text default 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa');
insert into t select *, 'a' from generate_series(1, 100000);
alter table t add constraint pk primary key (pk);
```

- создание скриптов:

```
echo "select count(*) from t;" > count1.sql
echo "select count(1) from t;" > count2.sql
echo "select count(pk) from t;" > count3.sql
```

- запуск тестов:

```
pgbench -T 30 -f count1.sql 2> /dev/null | grep tps
tps = 74
pgbench -T 30 -f count2.sql 2> /dev/null | grep tps
tps = 66
pgbench -T 30 -f count3.sql 2> /dev/null | grep tps
tps = 54
```

- результат: `count(1)` быстрее `count(pk)` на ~18%, `count(*)` быстрее `count(1)` на ~10%

Пример использования pgbench

Посмотрим пример использования pgbench. Задача: проверить, что лучше использовать: `count(*)`, `count(1)`, `count(c)` при работе с PostgreSQL?

Первый шаг: создание таблицы для теста:

```
drop table if exists t;
create table t(pk bigserial, c1 text default 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa');
insert into t select *, 'a' from generate_series(1, 100000);
alter table t add constraint pk primary key (pk);
analyze t;
```

Создание файлов с запросами, которые будут сравниваться:

```
echo "select count(*) from t;" > count1.sql
echo "select count(1) from t;" > count2.sql
echo "select count(pk) from t;" > count3.sql
```

Выполнение тестов:

```
pgbench -T 300 -f count1.sql 2> /dev/null | grep tps
tps = 74
pgbench -T 300 -f count2.sql 2> /dev/null | grep tps
tps = 66
pgbench -T 300 -f count3.sql 2> /dev/null | grep tps
tps = 54
```

Результат: `count(*)` быстрее `count(1)` на ~10%, `count(1)` быстрее `count(c)` на ~18%,

Подсчет по первичному ключу `count(c)` медленнее всех.

Команда `explain (analyze) select count(1) from t;` даёт результат с большим разбросом.

Результат соответствует результатам других исследователей вопроса "`COUNT(*)` vs `COUNT(1)`": https://blog.jooq.org/whats-faster-count-or-count1/

Для проверки применялся цикл блока plpgsql. В нашем примере использовался pgbench. https://gist.github.com/lukaseder/2611212b23ba40d5f828c69b79214a0e

1-3

Использование утилит sysbench и fio

sysbench - утилита тестирования производительности

- в тестировании команд SQL функционал схож с pgbench
- тестирование выполняется с использованием простой таблицы:

```
\d sbtest1
                                Table "public.sbtest1"
 Column |      Type      | Collation | Nullable |                Default
--------+----------------+-----------+----------+---------------------------------------
 id     | integer        |           | not null | nextval('sbtest1_id_seq'::regclass)
 k      | integer        |           | not null | 0
 c      | character(120) |           | not null | ''::bpchar
 pad    | character(60)  |           | not null | ''::bpchar
Indexes:
    "sbtest1_pkey" PRIMARY KEY, btree (id)
    "k_1" btree (k)
```

- имеет несколько скриптов для тестирования:

```
ls /usr/share/sysbench
bulk_insert.lua  oltp_insert.lua        oltp_read_write.lua        oltp_write_only.lua
oltp_common.lua  oltp_point_select.lua  oltp_update_index.lua      select_random_points.lua
oltp_delete.lua  oltp_read_only.lua     oltp_update_non_index.lua  select_random_ranges.lua
sysbench --db-driver=pgsql --pgsql-port=5432 --pgsql-db=postgres --pgsql-user=postgres --
pgsql-password=postgres --table_size=100000 /usr/share/sysbench/oltp_read_only.lua prepare
```


sysbench - утилита тестирования производительности

Утилита pgbench не имеет возможности протестировать основные ресурсы, используемые СУБД: скорость процессоров (cpu), доступа к памяти (memory), файловой системы (fileio). Это полезно при сравнении оборудования или переконфигурирования linux. Для тестирования ресурсов удобна утилита sysbench. Утилита имеется в Astralinux и других дистрибутивах linux.

Sysbench разрабатывалась для нагрузочного тестирования СУБД MySql. В настоящее время имеет тесты тесты для PostgreSQL. В тестировании команд SQL функционал схож с pgbench.

Установка sysbench:

```
sudo apt install sysbench
```

Доступные скрипты для тестирования СУБД (наборы команд SQL):

```
ls /usr/share/sysbench
bulk_insert.lua  oltp_insert.lua        oltp_read_write.lua        oltp_write_only.lua
oltp_common.lua  oltp_point_select.lua  oltp_update_index.lua      select_random_points.lua
oltp_delete.lua  oltp_read_only.lua     oltp_update_non_index.lua  select_random_ranges.lua
```

Можно создавать свои тесты на языке lua.

Тестирование выполняется с использованием одной таблицы:

```
\d sbtest1
                                Table "public.sbtest1"
 Column |      Type      |Collation| Nullable |                Default
--------+----------------+---------+----------+---------------------------------------
 id     | integer        |         | not null | nextval('sbtest1_id_seq'::regclass)
 k      | integer        |         | not null | 0
 c      | character(120) |         | not null | ''::bpchar
 pad    | character(60)  |         | not null | ''::bpchar
Indexes:
    "sbtest1_pkey" PRIMARY KEY, btree (id)
    "k_1" btree (k)
```

При инициализации тестов можно указать количество таких таблиц и количество строк в таблицах:

```
sysbench --db-driver=pgsql --pgsql-port=5432 --pgsql-db=postgres --pgsql-user=postgres --pgsql-password=postgres --tables=1 --table_size=100000 /usr/share/sysbench/oltp_read_only.lua prepare
```

После создания таблиц (prepare) можно выполнять тесты:

```
sysbench --db-driver=pgsql --pgsql-port=5432 --pgsql-db=postgres --pgsql-user=postgres --pgsql-password=postgres --threads=10 --time=60 --report-interval=5 /usr/share/sysbench/oltp_read_only.lua run
```

Параметр `--time` (аналог параметра `-T` у утилиты pgbench) задает время теста в секундах; `--report-interval` (аналог параметра `-P` у утилиты pgbench) интервал выдачи строк со статистикой; `--threads` (аналог параметров `-j -c` у утилиты pgbench).

Удаление созданных объектов:

```
sysbench --db-driver=pgsql --pgsql-port=5432 --pgsql-db=postgres --pgsql-user=postgres --pgsql-password=postgres /usr/share/sysbench/oltp_read_only.lua cleanup
```

Использование sysbench для тестирования процессоров, памяти, дисков

- встроенные тесты: cpu, memory, fileio, threads, mutex
- тестирование скорости работы нескольких потоков:

```
sysbench cpu run --time=10 --threads=4 | grep sec
    events per second:  4709.35
sysbench cpu run --time=10 --threads=8 | grep sec
    events per second:  4684.77
```

- оптимальное значение --num-threads число ядер центральных процессоров
- скорость работы (чтение или запись) с памятью:

```
sysbench memory run --memory-block-size=4K --time=10 --memory-oper=read --memory-access-mode=sequential
--memory-scope=local --threads=2 | grep transf
5673.72 MiB transferred (567.27 MiB/sec)
sysbench memory run --memory-block-size=2M ...
102400.00 MiB transferred (38767.60 MiB/sec)
```


Использование sysbench для тестирования процессоров, памяти, дисков

Утилита успешно используется для тестирования целях сравнения центральных процессоров:

```
sysbench cpu run --time=10 | grep sec
    events per second:  1218.32
sysbench cpu run --time=10 --threads=4 | grep sec
    events per second:  4709.35
sysbench cpu run --time=10 --threads=8 | grep sec
    events per second:  4684.77
```

Оптимальное значение `--num-threads` число ядер центральных процессоров (в примере 4).

Скорость работы с памятью:

```
sysbench memory run --memory-block-size=4K --time=10 --memory-oper=read --memory-access-
mode=seq --memory-scope=local --threads=2 | grep transf
5673.72 MiB transferred (567.27 MiB/sec)
sysbench memory run --memory-block-size=8K ...
11331.38 MiB transferred (1132.94 MiB/sec)
sysbench memory run --memory-block-size=2M ...
102400.00 MiB transferred (38767.60 MiB/sec)
sysbench memory run --memory-block-size=1G ...
102400.00 MiB transferred (23047.27 MiB/sec)
```

Произведение `--threads` на `--memory-block-size` не должно превышать свободную физическую память; `--memory-scope=local` каждый поток работает со своей частью памяти. Потоки будут использовать MMU (memory management unit, устройство доступа к основной памяти) своего ядра. Есть зависимость производительности от числа потоков и размера куска памяти. Параметры тестирования многопоточности:

Количество потоков (`--threads`), количество блокировок (`--thread-locks`) и количество раз, которое поток должен выполнить свою рабочую нагрузку по алгоритму: lock->yield->работа->unlock (`--thread-yields`).

```
sysbench threads run --time=10 --thread-locks=1 --threads=4 | grep events:
    total number of events:              1453
sysbench threads run --time=10 --thread-locks=1 --threads=1 | grep events:
    total number of events:              2775
sysbench threads run --time=10 --thread-locks=4 --threads=4 | grep events:
    total number of events:              7697
```

Тестирование аппаратных ресурсов

- запуск потоков и их одновременная работа:

```
sysbench threads --threads=128 --time=10 run | grep events:
    total number of events:              1502
sysbench threads --threads=8 --time=10 run | grep events:
    total number of events:              3991
sysbench threads --threads=4 --time=10 run | grep events:
    total number of events:              8278
sysbench threads --threads=1 --time=10 run | grep events:
    total number of events:              2773
```

- тест работы с файлами:

```
sysbench fileio --file-total-size=128M --file-num=8 prepare
134217728 bytes written in 1.19 seconds (107.12 MiB/sec).
sysbench fileio --file-block-size=8K --file-fsync-mode=fdatasync --file-fsync-end=on --file-total-
size=128M --file-num=8 --file-test-mode=rndrw --max-time=10 run | grep /s
    reads/s:                      435.92
    writes/s:                     290.61
    fsyncs/s:                     939.62
    read, MiB/s:                  6.81
    written, MiB/s:               4.54
    events (avg/stddev):          16613.0000/0.00
    execution time (avg/stddev):  9.8619/0.00
```

- тест использования легковесных блокировок (mutex)

```
sysbench mutex run --mutex-num=9000000 | grep exec
    execution time (avg/stddev):   0.1511/0.00
```


Тестирование аппаратных ресурсов

Запуск потоков и их одновременная работа:

```
sysbench threads --threads=128 --time=10 run | grep events:
    total number of events:              1502
sysbench threads --threads=8 --time=10 run | grep events:
    total number of events:              3991
sysbench threads --threads=4 --time=10 run | grep events:
    total number of events:              8278
sysbench threads --threads=1 --time=10 run | grep events:
    total number of events:              2773
```

Тест использования легковесных блокировок (mutex). Каждый поток выполняет простое действие типа увеличение числа на единицу в цикле (`--mutex-loops`), после чего поток берёт случайный mutex (один из `--mutex-num`) увеличивает переменную на единицу и освобождает mutex. Это повторяется `--mutex-locks` число раз.

```
sysbench mutex run --mutex-num=9000000 | grep exec
    execution time (avg/stddev):   0.1511/0.00
```

Тест работы с файлами:

```
sysbench fileio --file-total-size=128M --file-num=8 prepare
134217728 bytes written in 1.19 seconds (107.12 MiB/sec).
sysbench fileio --file-block-size=8K --file-fsync-mode=fdatasync --file-fsync-end=on --
file-total-size=128M --file-num=8 --file-test-mode=rndrw --max-time=10 run | grep /s
    reads/s:                      563.16
    writes/s:                     375.44
    fsyncs/s:                     75.19
    read, MiB/s:                  4.40
    written, MiB/s:               2.93
    events (avg/stddev):          10253.0000/0.00
    execution time (avg/stddev):  9.9455/0.00
sysbench fileio cleanup
```

Значение параметра `--file-test-mode=rndrw` случайные чтение-запись. Параметрами `--file-total-size=128M --file-num=8` определяется размер файлов. PostgreSQL использует 16Мб для WAL файлов и 1Gb для файлов данных, fdatasync для синхронизации WAL и fsync для файлов данных.

Утилита не выдаёт все параметры по --help. Описание параметров нужно смотреть: man sysbench.

Тестирование ввода-вывода утилитой Flexible IO Tester (fio)

- `sysbench` использует тестовые файлы, заполненные нулями
- для тестирования ввода-вывода обычно используют утилиту `fio`
- основные параметры при тестировании оборудования для PostgreSQL:
  - `bs=8k` PostgreSQL по умолчанию читает и пишет блоками по 8Кб
  - `direct=0` PostgreSQL по умолчанию работает с файлами через страничный кэш

```
sudo apt update && apt install fio -y
sudo fio --rw=rw --rwmixread=75 --size=16m --directory=/ --fadvise_hint=0 --blocksize=8k --
direct=0 --numjobs=1 --nrfiles=1 --runtime=5s --time_based --exec_prerun="echo 3 >
/proc/sys/vm/drop_caches" --group_reporting --name=test1
...
  read: IOPS=15.2k, BW=119MiB/s (124MB/s)(594MiB/5011msec)
     lat (usec): min=3, max=51909, avg=53.39, stdev=860.06
    bw (  KiB/s): min=110946, max=147129, per=100.00%, avg=121645.70, stdev=12274.53, samples=10
    iops        : min=13868, max=18391, avg=15205.60, stdev=1534.21, samples=10
  write: IOPS=5045, BW=39.4MiB/s (41.3MB/s)(198MiB/5011msec); 0 zone resets
     lat (usec): min=4, max=137, avg= 8.57, stdev= 4.52
    bw (  KiB/s): min=37184, max=48542, per=100.00%, avg=40428.10, stdev=3900.63, samples=10
    iops        : min= 4648, max= 6067, avg=5053.30, stdev=487.36, samples=10
  cpu           : usr=23.27%, sys=5.33%, ctx=2279, majf=0, minf=14
...
```


Тестирование ввода-вывода утилитой Flexible IO Tester (fio)

При анализе производительности дисковой подсистемы `sysbench` использует тестовые файлы, заполненные нулями. Оборудование может оптимизировать работу с такими файлами. Для точного тестирования ввода-вывода используют утилиты **`fio`** (Flexible IO Tester) и **`flashbench`**. Производительность ввода-вывода может быть ограничена шиной ввода-вывода: числом линий PCIe.

Установка утилиты:

`apt install fio -y`

Параметры:

`bs=8k` PostgreSQL по умолчанию читает и пишет блоками по 8Кб

direct=0 PostgreSQL по умолчанию работает с файлами через страничный кэш. Для сравнения систем хранения данных можно использовать direct=1

`numjobs` - число потоков, которыми fio нагружает систему. Оптимально чтобы значение было равно числу ядер или hardware threads

`iodepth` - глубина очереди команд. Параметры numjobs и iodepth увеличивают для получения максимальных показателей IOPS, так как один поток вряд ли утилизирует всю пропускную способность шины PCIe. Большое значение iodepth может нагрузить ядро процессора до 100% и ядро станет узким местом.

`rwmixread=75` пропорция операций чтения/записи. Для OLTP: 80/20 или 75/25

`fadvise_hint=0` устанавливает подсказку POSIX_FADV_DONTNEED

`exec_prerun="echo 3 > /proc/sys/vm/drop_caches"` перед повторными запусками чистые страницы страничного кэша и структуры slab стоит освободить

`size=1G` максимальный размер файла данных в PostgreSQL 1Гб. В директории, где будут создаваться файлы должно иметься свободное место

`rw=randread, read, write, randwrite, randread`

`filename` путь к файлу или блочному устройству для тестирования. Не стоит указывать блочное устройство с файловой системой для тестов на запись (`readwrite, randrw, write, trimwrite`) файловая система будет испорчена (содержимое устройства затёрто). Также при указании файла, данные в нём перезаписываются.

`ioengine=libaio` наиболее быстрый (потому что неблокирующий), `psync` по умолчанию

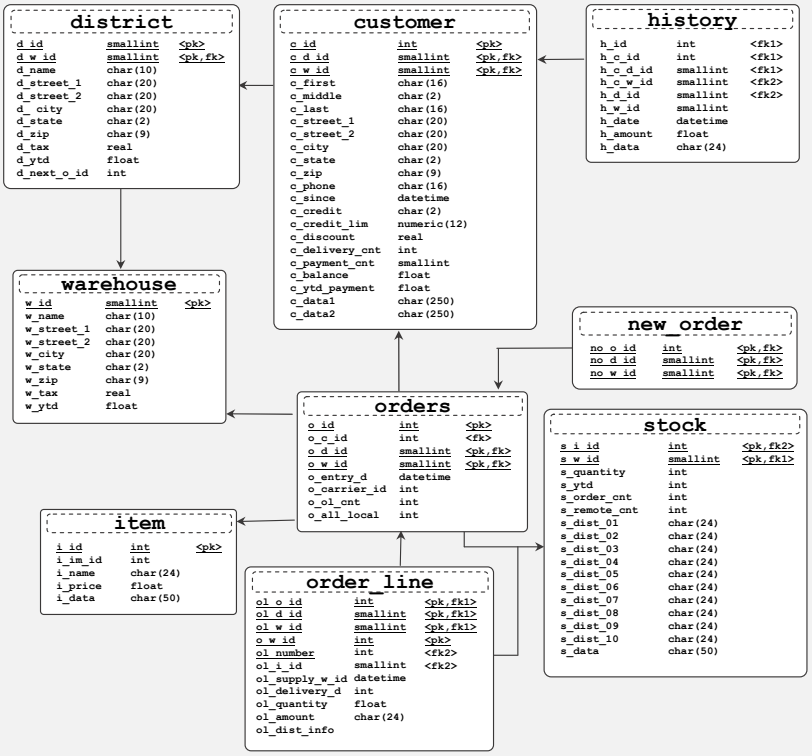
Основные метрики: iops, bw - скорость передачи данных, latency, cpu (usr, sys)

1-4

Тесты TPC

Тесты TPC-B и TPC-C

- в TPC-B команды выполняются на стороне базы данных с максимальной скоростью
- в тесте TPC-B эмулируется работа банка
- в тесте TPC-C эмулируется работа оптового склада
- время выполнения транзакций в TPC-C 5 секунд для всех транзакций кроме одной где время выполнения до 20 секунд
- основной результат это tpmC - число транзакций в минуту



Тесты TPC-B и TPC-C

В 1988 году был создан Transaction Processing Performance Council (совет по оценке производительности обработки транзакций) из 26 компаний и утверждены спецификации тестов TPC-A и TPC-B. Эти тесты отличались тем, что в первом требовалась эмуляция клиентской части ("терминалов") с задержкой времени реакции клиентской части. В TPC-B команды выполняются на стороне базы данных с максимальной скоростью. В обоих тестах эмулировалась работа банка. В тесте TPC-C эмулируется работа оптового склада. Это отражало то, что СУБД стали активно использоваться торговыми компаниями, банки уже были автоматизированы. Единицей масштабирования (scale factor) в TPC-C является склад (таблица Warehouse). На каждом складе десять районов (таблица **district**), оснащенных "терминалом" (программа-клиент СУБД). Каждый терминал предназначен для регистрации заказов (таблица **orders**). Каждый заказ (торговая транзакция) состоит из позиций (таблица **order_line**) одного продаваемого товара (таблица **stock**). В каждом округе можно сделать не более 1,2 заказа в минуту. Как только этот предел заказов в минуту (tpmC) достигается, по условиям теста добавляется еще один склад (таблица **warehouse**). Склады заранее создаются и транзакции в процессе тестирования проводятся по этим складам. Пока 90% транзакций выполняются менее чем за 5 секунд, продолжается задействование складов. Кроме заказов (45% от общей нагрузки) одновременно выполняются транзакции по приему платежа с обновлением баланса клиента (43%), проверка статуса последнего заказа одного любого клиента (4%), проверка количества товаров на складе (4%), формирование заявок на доставку (4%). В заявку включаются оформленные к моменту формирования заявки заказы. Пороговое время формирования заявки на доставку 20 секунд. Время выполнения каждой торговой транзакции (заявки, заказа), которая соответствует транзакции базы данных, состоящей из одной или нескольких команд SQL подсчитывается отдельно и распределение значений может использоваться как один из результатов теста. Основной результат это tpmC - число транзакций в минуту. Другие результаты не технические: стоимость транзакции (зависит от стоимости лицензий на СУБД или программно-аппаратный комплекс), расчетное значение сколько ватт электроэнергии потратится на тысячу транзакций в минуту (W/ktpmC). Не получили распространения тесты TPC-R для отчётов, TPC-D для OLAP, TPC-W для заказов в веб-магазине. На основе TPC-D был создан более удачный тест TPC-H для хранилищ данных и аналитических запросов ("OLAP нагрузка"). Число таблиц 8, ограничений целостности 17. В TPC-H были выделены номинации по размерам обрабатываемых данных от "до 100Гб" до 30-100Тб.

Тест TPC-E, тестирование устойчивости к сетевым сбоям

- TPC-E тест для OLTP, появился в 2006 году
- вместо оптового склада описывается работа брокерской компании, количество таблиц 33 (TPC-E), вместо 9 (TPC-C)
- первичных ключей 33 вместо 8, внешних ключей 50 вместо 9
- из-за сложности воспроизведения и реализации тестов TPC востребованы простые утилиты
- для баз данных семейства PostgreSQL используется утилита pgbench

Тест TPC-E, тестирование устойчивости к сетевым сбоям

В 2006 году появился тест TPC-E для OLTP, на смену TPC-C. Вместо оптового склада описывается работа брокерской компании, количество таблиц 33, вместо 9. Первичных ключей 33 вместо 8, внешних ключей 50 вместо 9. Добавились типы данных для столбцов boolean и lob. Метрика стала называться tpsE.

Из-за сложности воспроизведения и реализации тестов TPC востребованы простые утилиты, которые измеряют простые показатели с повторяемостью результатов и доверенными интервалами (допустимый разброс данных). Для PostgreSQL используется pgbench.

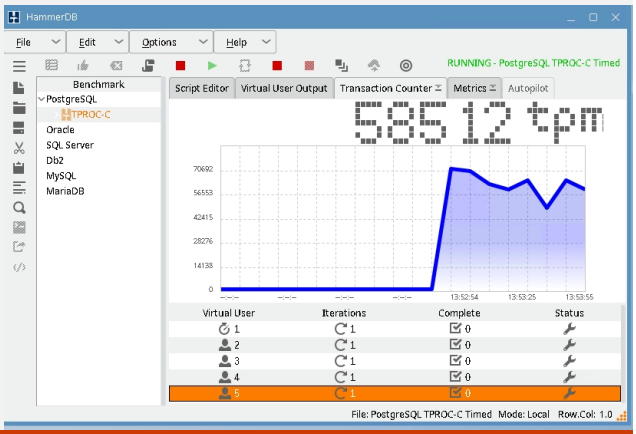
Клиенты подсоединяются к базам данных по сети. Пакеты (сообщения на каком-то из сетевых уровней) могут теряться, порядок доставки пакетов меняться, пакеты могут дублироваться. Причинами могут быть: сбой сетевой карты, ошибки в драйверах, нехватка памяти, оборудование, препятствующее прохождению сетевого трафика. Для тестирования устойчивости баз данных к таким ошибкам используется jepsen framework. Для PostgreSQL было обнаружено, что если клиент получил подтверждение о фиксации транзакции, то в 100% транзакция зафиксирована и обеспечивается durability. Если клиент получил сообщение об ошибке ввода-вывода, то транзакция может быть зафиксирована или не зафиксирована. Вероятность такого события редка. **Протокол 2PC не защищает от этого типа ошибки**, протокол E3PC предусматривает защиту от сбоев сети. Однако, эти протоколы уменьшают производительность.

В Oracle Database начиная с 12 версии есть опция Oracle Transaction Guard для защиты от такого типа сбоев. Redis и MongoDB теряли существенный процент данных (**https://www.infoq.com/articles/jepsen/**). Технологии, обещающие высокую производительность, стоит использовать с учетом отказоустойчивости. Под конкретные задачи стоит выбирать подходящие и зарекомендовавшие себя решения. Например, в задачах, где важно гарантированное время отклика (выполнения транзакции) применяются базы данных реального времени. В случаях, где потери транзакций недопустимы (финансы), применяются Oracle Database и PostgreSQL.

При измерении производительности время выполнения транзакции (может называться latency, время отклика) является самостоятельной метрикой.

Реализация теста TPC-C

- использует в качестве результата измерений tpmC число "транзакций" этого теста в минуту
- может использоваться для сравнения производительности СУБД разных производителей
- для оценки влияния изменений конфигурации экземпляра не удобен
- недостаток теста в переусложнении правил теста, что приводит к увеличению объема данных в таблицах, количества клиентов и памяти, которые они потребляют
- использует набор команд SQL на порядок сложнее, чем в TPC-B и использует 9 таблиц
- приложение HammerDB включает в себя варианты тестов TPC-C и TPC-H



Реализация теста TPC-C

Тест TPC-C использует набор команд на порядок сложнее, чем TPC-B и использует 9 таблиц: Warehouse (склад), District (район), Customer (клиент), Order (заказ), New-Order (новый заказ), History (история), Item (товар), Stock (складской запас), Order-Line (строка заказа). Использует 5 типов транзакций: New order (новый заказ), Payment (платеж), Order status (статус заказа), Delivery (доставка), Stock level (инвентаризация) выбираются случайным образом в заданной тестом пропорции.

Оба теста имитируют OLTP нагрузку: короткие транзакции с выборкой и обновлением небольшого числа строк. Недостаток теста в переусложнении правил теста, что приводит к увеличению объема данных в таблицах, количества клиентов и памяти, которые они потребляют; нет программного кода, только описание теста. TPC-C использует в качестве результата измерений tpmC число "транзакций" этого теста в минуту. Тест может использоваться для сравнения производительности СУБД разных производителей. Для оценки влияния изменений конфигурации экземпляра не удобен.

Приложений, реализующих тесты TPC немного, большая часть из них неработоспособны. Одна из работающих программ: HammerDB, она включает в себя тесты TPC-C и TPC-H. На сайте приложения публикуются результаты тестов. Результаты, выполненные в HammerDB сравнимы между собой.

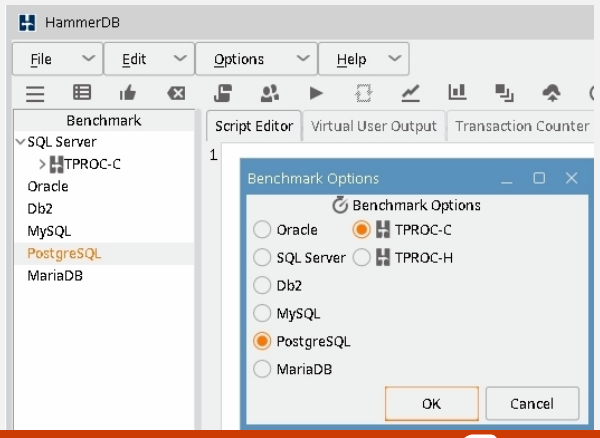
Тест TPC-H предназначен для хранилищ данных, включает в себя 22 запроса которые называют Q1 ... Q22. Тест TPC-H не меняет данные в таблицах, поэтому пригоден для повторных запусков без пересоздания таблиц.

Запросы **Q17** и **Q20** коррелированные, наиболее тяжелы для любых СУБД. Если оптимизируется работа СУБД с хранилищами данных, вероятно, стоит уделить внимание оптимизации выполнения именно этих запросов. Специализирующаяся на аналитике СУБД clickhouse на 2024 год не может выполнить эти два запроса.

https://habr.com/ru/companies/ydb/articles/801587/

Приложение HammerDB

- реализует подмножество полной спецификации TPC-C, измененное для упрощения и облегчения выполнения рабочей нагрузки
- результаты теста тип-C HammerDB не сравнимы с результатами других тестов, использующих метрику tpmC
- результаты сравнимы, если они проведены в HammerDB
- отличие от TPC-C в том, что по умолчанию HammerDB работает без задержек на ввод данных и обдумывание. Тест тип-C запускает нагрузку TPC-C без задержек
- результат теста HammerDB тип-C: TPM и NOPM



Приложение HammerDB

Приложение реализует тест на основе спецификации TPC-C, но по умолчанию не реализует полную спецификацию правил тестирования TPC-C.

Результаты HammerDB нельзя сравнивать с официально опубликованными бенчмарками TPC-C. Официальные бенчмарки TPC-C чрезвычайно дороги, трудоемки и сложны. HammerDB разработан для того, чтобы тест, соответствующий TPC-C можно было запустить с низкими затратами на любой системе, обеспечивая профессиональное, надежное и предсказуемое нагрузочное тестирование для всех сред баз данных. Результаты HammerDB не сравнимы с результатами других тестов, использующих метрику tpmC. Однако результаты, выполненные приложением сравнимы между собой.

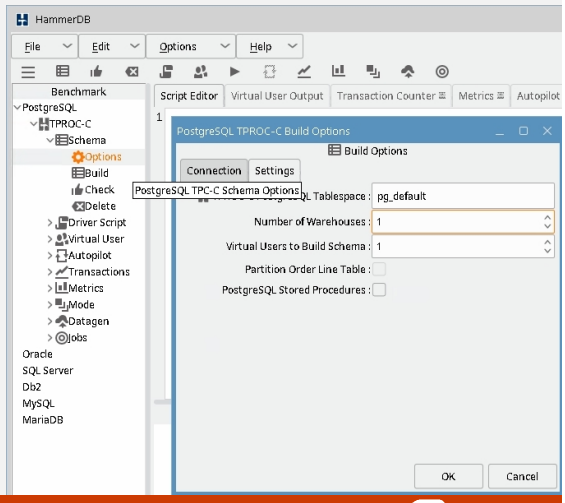
В результатах тестирования HammerDB выдаёт два показателя для сравнения с другими результатами тестирования программой HammerDB: TPM и NOPM. NOPM это количество новых заказов в минуту. NOPM можно использовать для сравнения производительности СУБД разных типов.

HammerDB можно рассматривать как подмножество полной спецификации TPC-C, измененное для упрощения и облегчения выполнения рабочей нагрузки. Основные сходства - это определение схемы хранения и сами данные, а также 5 транзакций, реализованных в виде хранимых процедур. Основное отличие заключается в том, что по умолчанию HammerDB работает без задержек на ввод данных и обдумывание. Это означает, что HammerDB TPROC-C запускает рабочую нагрузку соответствующую TPC-C без задержек, максимально нагружая СУБД. Число пользователей и требуемый объем данных, на которых достигается максимальная производительность будут намного меньше, чем в полной реализации TPC-C.

HammerDB не реализует "терминалы", как это делает полная спецификация. Благодаря этому не требуется большое число клиентов и огромные объемы данных. При этом обеспечивается надежный тест возможностей и производительности реляционных баз данных.

Параметры для теста тип-С HammerDB

- официальный тест TPC-C имеет фиксированное количество пользователей на хранилище и использует время ввода и обдумывания (think time), чтобы рабочая нагрузка, создаваемая каждым пользователем, не была интенсивной
- HammerDB по умолчанию не использует время на ввода и обдумывание, и поэтому число Virtual Users примерно равно числу ядер на хосте с СУБД
- 4-5 складов на одного Virtual User будет минимальным значением для обеспечения равномерного распределения Virtual Users по складу



Параметры для теста тип-С HammerDB

При создании таблиц для TPC-C указываются:

Number of Warehouses - число складов. Для 2000 "складов" размер таблиц ~250Гб. Число складов должно быть в 10 раз больше, чем "клиентов", так как число одновременных транзакций по одному складу ограничено по условиям теста. В терминах теста:

Virtual Users число "клиентов" - то же самое что потоков, сессий для параллельной нагрузки.

Driver Script - командный файл создается автоматически.

Rampup Time - время постепенного увеличения нагрузки.

Сколько складов создавать? Базовое число складов 250-500 на один центральный процессор. Официальный тест TPC-C имеет фиксированное количество пользователей на хранилище и использует время ввода и обдумывания (think time), чтобы рабочая нагрузка, создаваемая каждым пользователем, не была интенсивной. Это увеличивает число клиентов и требует большого числа хостов. HammerDB по умолчанию не использует время на ввода и обдумывание, и поэтому число Virtual Users примерно равно числу ядер на хосте с СУБД. При использовании Hyper Threading число Virtual Users можно увеличить на треть от числа физических ядер.

10 складов для обслуживания 100 Virtual Users будет означать, что рабочая нагрузка будет тратить значительно больше времени на конкуренцию за блокировки и tpm будет ниже. 4-5 складов на одного Virtual User будет минимальным значением для обеспечения равномерного распределения Virtual Users по складу. Для 100 Virtual Users стоит создать минимум 400–500 складов.

Для топовых конфигураций СУБД любых производителей максимальная производительность для тестов без времени на обдумывание достигается в районе 2000 складов и до 500 сессий (Virtual Users).

Утилита Go-TPC

- утилита написана на языке go, работает в командной строке, что позволяет автоматизировать ее запуск
- реализует тесты TPC-C, TPC-H, CHmark
- CHmark объединяет оба теста, использует схему таблиц TPC-C и упрощенную схему TPC-H
  - был создан для баз обслуживающих смешанную нагрузку: OLTP и одновременно OLAP
  - Для PostgreSQL рекомендуется переносить OLAP нагрузку на физические реплики

```
wget https://raw.githubusercontent.com/pingcap/go-tpc/master/install.sh
chmod +x install.sh
./install.sh
cd .g-tpc/bin
./go-tpc tpcc prepare --warehouses 1 -d postgres -U postgres -p postgres -H 127.0.0.1 -P 5432
./go-tpc tpcc run --time 30s -d postgres -U postgres -p postgres -H 127.0.0.1 -P 5432
```


Утилита Go-TPC

Утилита G0-TPC написана на языке go, работает в командной строке, что позволяет автоматизировать ее запуск и в этом её преимущество. Утилита реализует тесты TPC-C, TPC-H, CHmark. CHmark объединяет оба теста, использует схему таблиц TPC-C и упрощенную схему TPC-H. Тест был создан для баз обслуживающих смешанную нагрузку: OLTP и одновременно OLAP. Для PostgreSQL рекомендуется переносить OLAP нагрузку на физические реплики. Тест может быть интересен как использование более сложной нагрузки, чем короткие по времени транзакции TPC-C.

Установка утилиты:

```
wget https://raw.githubusercontent.com/pingcap/go-tpc/master/install.sh
chmod +x install.sh
./install.sh
/var/lib/postgresql/.bash_profile has been modified to to add go-tpc to PATH
Installed path: /var/lib/postgresql/.go-tpc/bin/go-tpc
cd .g-tpc/bin
./go-tpc tpcc prepare -d postgres -U postgres -p postgres -D tpcc -H 127.0.0.1 -P 5432
```

По умолчанию создаёт 10 warehouses. Параметром `--warehouses N` можно указать желаемое число складов. По умолчанию создаёт базу данных с названием test. Таблицы для тестов должны использовать разные базы. Если таблицы не нужны, достаточно удалить созданную базу данных.

Для создания таблиц для теста TPC-H используется команда:

```
./go-tpc tpch prepare -d postgres -U postgres -p postgres -D tpch -H 127.0.0.1 -P 5432
```

Запуск тестов:

```
./go-tpc tpcc run -d postgres -U postgres -p postgres -D tpcc -H 127.0.0.1 -P 5432
./go-tpc tpch run --sf 1 -d postgres -U postgres -p postgres -D tpch -H 127.0.0.1 -P 5432
```

Параметр `--sf N` задаёт scale factor, по умолчанию 1. По умолчанию размер таблиц небольшой: в таблице lineitem 6млн. строк, orders 1,5млн.

Параметром `--count N` можно задать число итераций. Для tpch это число запросов.

Параметром `-T` можно задать число потоков, по умолчанию 1.

Параметром `--time NhNmNs` можно задать длительность теста.

Тестирование можно прервать комбинацией клавиш <ctrl+c> и получить результат:

```
Got signal [interrupt] to exit.
Finished
tpmC: 939.7, tpmTotal: 2089.7, efficiency: 730.7%
```

Практика

- Часть 1. Стандартный тест pgbench
- Часть 2. Использование pgbench с собственным скриптом
- Часть 3. Использование утилиты sysbench
- Часть 4. Использование приложения HammerDB
- Часть 5. Использование приложения Go-TPC

Практика

Цель практики: использовать утилиты тестирования производительности.

В практике вы:

запустите тест pgbench и проверите как меняется результат теста, если удерживать горизонт базы данных;

проверите что быстрее count(*) или count(1);

во сколько раз уменьшается время выполнения команды EXPLAIN при использовании опции timing off;

посмотрите как читать отчеты утилиты sysbench.

Установите программу HammerDB, запустите тест и посмотрите как автоанализ ухудшает работу СУБД.

Установите программу Go-TPC и узнаете как небольшие запросы могут выполняться несколько часов (не все СУБД могут выполнять такие запросы).

2

2-1

Память

Оперативная память

Работа с памятью осуществляется страницами

- размер обычной страницы 4Кб задан аппаратно
- размер TLB (Translation Lookaside Buffer, буфер ассоциативной трансляции) 2-4Кб
- частота непопаданий в TLB 0.01-1%
- при попадании в TLB скорость доступа к памяти 1/2-1 такт процессора
- если ссылка отсутствует в TLB, используется медленный механизм преобразования
- непопадания обрабатываются за 10-100 тактов процессора
- размер обычной страницы памяти выдает команда:

```
astra@tantor:~$ getconf PAGE_SIZE
4096
```


Оперативная память

Оперативная память - один из основных ресурсов, который используется процессам экземпляра. Размер оперативной памяти это не единственный параметр, который влияет на эффективность работы с памятью. В этой главе рассматриваются особенности работы с памятью, которые влияют на эффективность.

MMU (memory management unit, блок управления памятью) делит виртуальное адресное пространство (одномерный массив адресов, используемых процессором) на участки одинакового размера, называемые страницами. Адрес ячейки памяти (которую обрабатывает процессор за один такт) состоит из смещения внутри страницы и номера страницы. Конкатенация номера физической страницы со смещением внутри страницы даёт физический адрес. MMU преобразует номера виртуальных страниц в номера физических страниц, используя TLB.

Операционная система работает поверх оборудования и выделяет память страницами, которые использует оборудование. Размер обычной страницы памяти 4 Килобайта (4096 байт). Размер страницы выдается командой:

```
astra@tantor:~$ getconf PAGE_SIZE
4096
```

Почему размер страницы 4КБ, можно ли его поменять и может ли он быть другим?

У всех процессоров архитектуры x86 в настоящее время размер обычной страницы 4КБ. Размер выбран опытным путем и определяется ограничениями взаимного расположения полупроводниковых элементов на кристалле кремния. Для x86 и ARM в будущем предполагается использование страниц размером 64Кб.

Для трансляции виртуальной памяти в адрес физической памяти в чипе CPU имеется кэш TLB (Translation Lookaside Buffer, буфер ассоциативной трансляции). TLB функционирует как массив ссылок на кэш основной памяти. Объем памяти, который TLB может одновременно отображать определяется числом "записей" TLB и размером "записи". Размер TLB 2Кб-4Кб "записей". Доступ к памяти, ссылка на которую есть в TLB происходит за 1 или половину такта (dual mode) CPU. Если ссылки нет, то доступ занимает 10-100 тактов CPU (miss penalty).

Если ссылка отсутствует в TLB, используется медленный механизм преобразования: аппаратный или на программных структурах (page table, PT, таблицы страниц) операционной системы. Данные в этих структурах называются page table entries (PTE). Процедура называется page walk, она просматривает PT. PT имеет древовидную структуру (radix-tree).

Виртуальная адресация памяти

- операционная система и процессы работают с виртуальным адресным пространством
- MMU (memory management unit, блок управления памятью) делит виртуальное адресное пространство на участки одинакового размера, называемые страницами
- преобразование виртуального адреса в физический должно быть максимально быстрым
- операционная система должна прозрачно сохранять (вытеснять) содержимое физической памяти на внешний носитель и читать обратно (swapping)

Виртуальная адресация памяти

PTE и TLB могут содержать дополнительную информацию: бит признака записи в страницу (dirty bit), время последнего доступа к странице (accessed bit), который используется для реализации алгоритма вытеснения страниц (least recently used, LRU), какие процессы (пользовательские, user) или системные (supervisor) могут читать или записывать данные в страницу, необходимо ли кэшировать страницу.

Практическая ("эмпирическая") частота непопаданий в кэш TLB 0,01 -1% (1:100...10000).

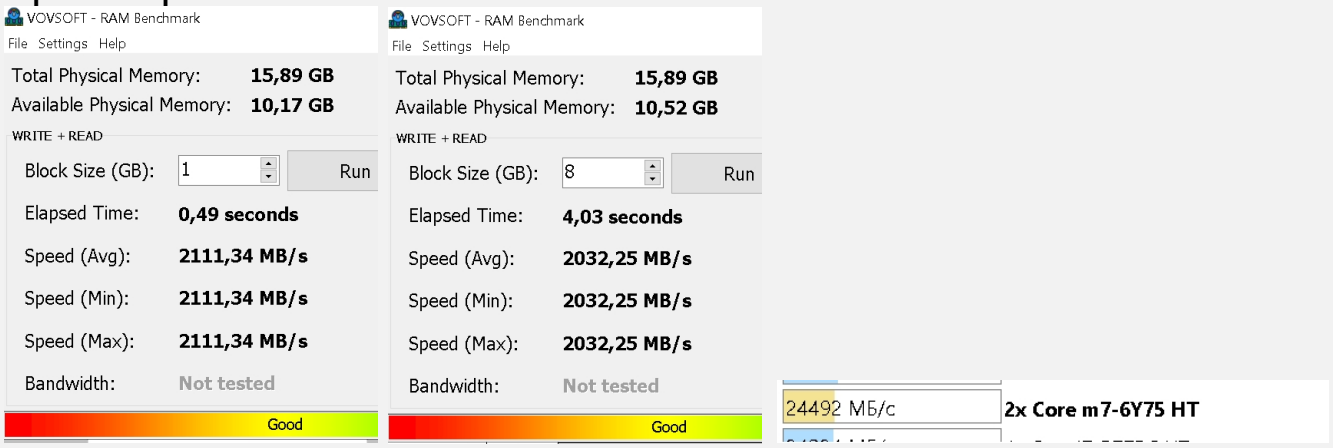
Почему? Ведь даже 4Гб виртуальной памяти соответствуют миллиону страниц. Объем TLB 4Кб (и даже делятся обычно пополам на страницы с данными dTLB и исполняемым кодом iTLB). Получается разница в миллион раз, а не в 10000 раз. Помогает то, что у программ обращение к памяти сильно неоднородное (нелинейное распределение): к части страниц памяти очень частое, к части редкое. Например, если программа выделяет память по гигабайту (пишет, потом читает), то скорость будет около 2 гигабайт в секунду, что медленнее доступа к SSD. При этом скорость доступа небольшими объемами будет на **порядок** выше (20 гигабайт в секунду).

Архитектура x86 обрабатывает непопадания в TLB на уровне железа, а не операционной системы (программные обработчики). При обработке операционной системой, код обработчика непопадания в TLB обычно длинной 10-100 инструкций (при обработке на уровне железа: "тактов") CPU может быть вытеснен из кэша инструкций CPU и обработка непопадания может длиться намного дольше, чем при обработке на уровне железа. Программные TLB встречались в архитектурах MIPS, SPARC, Alpha и PA-RISC. **Только на этих архитектурах операционная система (linux) могла использовать страницы 8Кб.**

Преобразование виртуального адреса в физический должно быть максимально быстрым. Операционная система прозрачно сохраняет (вытесняет) содержимое физической памяти на внешний носитель (файлы или разделы подкачки) и читает страницы обратно. Это называется swapping. Содержимое сохраняется страницами, swaping работает со страницами обычного размера. Huge Pages в linux не отображаются на страничный кэш, поэтому не выгружаются в swap, что обеспечивает более предсказуемую производительность.

Размер страниц памяти

- процессоры поддерживают 2 типа страниц, обычные и огромные (Huge Pages)
- На архитектуре x86-64 огромные страницы могут иметь размер 2Мб и 1Гб



Размер страниц памяти

Современные процессоры поддерживают обычные и огромные (Huge Pages) страницы.
Для процессоров архитектуры x86-64 огромные страницы могут иметь размер 2Мб и 1Гб.
Процессор поддерживает большие страницы размером 2Мб, если команды:

```
astra@tantor:~$ lscpu | grep pse
```

или

```
astra@tantor:~$ cat /proc/cpuinfo | grep pse | uniq
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2 ht syscall nx ...
```

выдают непустую строку, в выданной строке присутствует слово **pse**.
Это был пример вывода для

```
model name      : Intel(R) Core(TM) i3-8100 CPU @ 3.60GHz
```

Процессор поддерживает HugePages размером 1Гб, если команда

```
astra@tantor:~$ cat /proc/cpuinfo | grep pdpe1gb | uniq
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl ...
```

выдаст непустую строку, в выведенной строке будет присутствовать слово **pdpe1gb** .
Это был пример вывода для

```
model name : Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz.
```

В выводе для `i3-8100` этого слова не было, значит процессор i3 не поддерживает большие страницы размером 1Гб.
Для `AMD Ryzen 5 1600X Six-Core Processor`

```
flags : .. pae .. pse36 ..pdpe1gb ..
TLB size : 2560 4K pages
```

В выводе для AMD присутствует строка TLB size.
На картинке слайда показано измерение скорости записи-чтения куска памяти размером 1Гб, 8Гб ~2Гб/с и отдельных страниц ~24Гб/с на одном и том же хосте обычными страницами, что на порядок медленнее. Начальное выделение 1Гб или 8Гб не вносит сколько-нибудь существенной задержки. Основную роль играет увеличение частоты непопадания в TLB. Доступ к странице к которой недавно был доступ на порядок быстрее, чем к той, ссылка на которую была вытеснена из TLB.
https://wiki.debian.org/Hugepages#x86_64

Размер буфера ассоциативной трансляции (TLB)

- организация TLB зависит от процессора (поколения, архитектуры)
- производительность при выборе 2M или 1G страниц на разных процессорах может существенно различаться
- У процессоров Intel Sky Lake, Cofee Lake, Cascade Lake в L2 TLB хранится 1536 ссылок на страницы размером 4K и 2M и 16 ссылок на страницы 1G
- у процессоров семейства Intel Sunny Cove количество ссылок 2048, из них может использоваться до 1024 ссылок на страницы размером 2M или 1G

Размер буфера ассоциативной трансляции (TLB)

Выбор размера страниц (4Кб, 2Мб, 1Гб) зависит от модели процессора. **Производительность при выборе 2M или 1G страниц на разных процессорах может существенно различаться**.

Характеристики семейства процессоров Intel "Lake":

```
TLB consists of dedicated L1 TLB for instruction cache (ITLB) and another one for data cache (DTLB). Additionally there is a unified L2 TLB (STLB).
DTLB
4K page translations: 64 entries; 4-way set associative fixed partition
2M page translations: 32 entries; 4-way set associative fixed partition
1G page translations: 4 entries; 4-way set associative fixed partition
STLB
4K+2M page translations: 1536 entries; 12-way set associative fixed partition
1G page translations: 16 entries; 4-way set associative fixed partition
```

Характеристики приведены без перевода, так как состоят преимущественно из терминов. При переводе узкоспециализированных терминов теряется смысл.

Характеристики семейств процессоров Intel приведены в руководстве:
https://cdrdv2-public.intel.com/821613/355308-Optimization-Reference-Manual-050-Changes-Doc.pdf

Смысл терминов:

"partition" - распределение количества entries для страниц разного размера.

"fixed" - количество entries для страниц какого-то размера фиксировано.

"shared" - операционная система сможет выбирать сколько entries использовать для страниц одного размера и остаток места под entries для страниц другого размера.

Для семейств процессоров Intel "Lake" L2 TLB может хранить ссылки на 16 страниц размером 1Гб.

Указанные характеристики определяют возможности процессоров, публикуются производителями процессоров частично. Не только производители процессоров избегают прямого сравнения своих продуктов. Это оправданно тем, что для определения реальных возможностей процессоров нужно сравнивать значения в целом, а также тем, что из-за отличий в реализации характеристики у разных производителей и даже продуктов только созвучны и их нельзя напрямую сравнивать. Пример: стоимость плана выполнения (cost) сравнимы только для одного запроса, для разных запросов cost несравнимы. Второй пример: в одном процессоре может быть 256 vCPU, во втором 8 vCPU, при этом реальная производительность у второго процессора может быть больше. Производительность определяется сбалансированностью характеристик процессоров. Это не значит, что о характеристиках процессоров и знать не нужно. При определении характеристик нужно детально выяснять, что подразумевается под названием характеристики и как изменение характеристики влияет на общую производительность, так как влияние не всегда линейно.

Характеристики семейств процессоров приведены на сайте:
https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

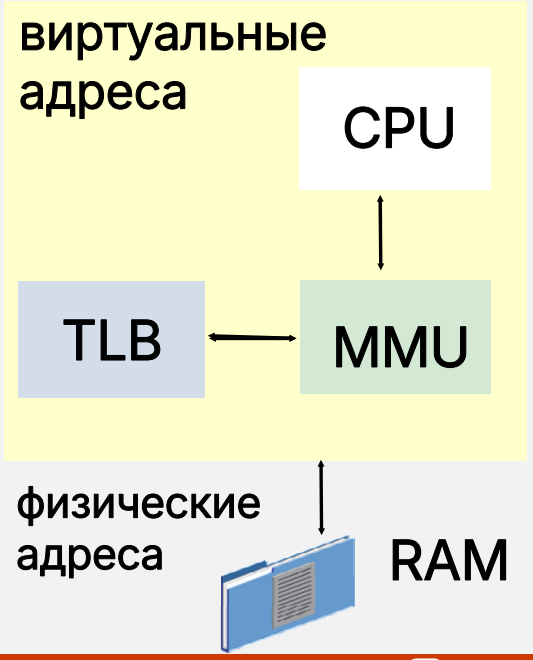
```
DTLB 4 KiB TLB competitively shared (from fixed partitioning)
Single unified TLB for all pages (from 4 KiB+2/4 MiB and separate 1 GiB)
STLB uses dynamic partitioning (from partition fixed partitioning):
4K pages can use all 2,048 entries
2M pages can use 1,024 entries (8-way sets), shared with 4K pages
1G pages can use 1,024 entries (8-way sets), shared with 4K pages
DTLB
4K page translations: 64 entries; 4-way set associative competitively shared
2M page translations: 32 entries; 4-way set associative competitively shared
1G page translations: 8 entries; 8-way set associative competitively partition
STLB
All pages: 2,048 entire; 16-way set associative
```

Про степень ассоциативности (*N*-way set associative) достаточно знать: чем больше число (*N*-ways), тем больше эффективность кэша. **4-way размером 1K эффективен примерно так же как 2-way размером 2K. 8-way размером 1K эффективен примерно так же как 4-way у которого размер немного меньше 2K.**

https://ru.m.wikipedia.org/wiki/Кэш_процессора

Огромные страницы (Huge Pages)

- количество Huge Pages нужно установить вручную в параметре операционной системы `vm.nr_hugepages`
- устанавливать можно и нужно с небольшим запасом
- изменять количество Huge Pages можно без перезагрузки
- рассматривают использование при большом объеме физической памяти (больше ~128Гб)



Огромные страницы (Huge Pages)

Перед включением использования Huge Pages (HP) нужно оценить сколько HP планируется использовать. Для получения размера памяти, выделенной процессу операционной системы можно найти PID процесса. В примере берется PID процесса postmaster, чтобы посмотреть сколько виртуальной памяти выделил процесс исходя из того, что HugePages выделяются или резервируются при запуске экземпляра:

```
astra@tantor:~$ ps -ef | grep postgres
postgres     926       1  0  00:00:02 /usr/lib/postgresql/15/bin/postgres
postgres     978     926  0  00:00:00 postgres: 15/main: logger
postgres     991     926  0  00:00:00 postgres: 15/main: checkpointer
```

Дальше найти строку в статистике командой:

```
astra@tantor:~$ cat /proc/926/status | grep VmPeak
VmPeak:   265284 kB
```

В этом примере выдано `265284kB`, что чуть меньше 260Мб или 130 страниц размером 2Мб.

Огромные страницы могут использовать разделяемый пул и параллельные процессы. В примере разделяемый пул 128Мб. 260Мб гораздо больше разделяемого пула. **При выделении HugePages можно использовать VmPeak только как ориентир.** Также можно ориентироваться на сумму значений `shared_buffers + min_dynamic_shared_memory`. Для получения более точной оценки придется остановить экземпляр и запустить postgres из командной строки с параметром `-C shared_memory_size_in_huge_pages`. Однако в форках PostgreSQL огромные страницы могут использовать и другие структуры памяти.

Пример резервирования адресного пространства в виртуальной памяти под 300 страниц:

```
root@tantor:~# sysctl -w vm.nr_hugepages=300
vm.nr_hugepages = 300
```

Для сохранения нового значения после рестарта операционной системы можно вставить в конец файла конфигурации желаемое количество больших страниц:

```
root@tantor:~# echo "vm.nr_hugepages = 300" >> /etc/sysctl.conf
```

Применить изменения, которые появились в файле:

```
root@tantor:~# sysctl -p
vm.nr_hugepages = 300
```

Насколько большим делать запас страниц HP? Это определяется формулой `CommitMemory` на слайде "Установка значений overcommit и swap", который будет чуть дальше. При отсутствии swap запас должен быть минимален.

Использование Huge Pages

Статистика использования:

- `cat /proc/meminfo | grep Huge`
  - `HugePages_Rsvd`: используется
  - `HugePages_Free`: можно использовать
  - `HugePages_Total`: доступно

```
root@tantor:~# cat /proc/meminfo | grep Huge
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:     300
HugePages_Free:      281
HugePages_Rsvd:       72
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:          614400 kB
```


Использование Huge Pages

Использование HP в PostgreSQL определяется значением параметра `huge_pages`. По умолчанию он установлен в значение `try`. Если HP доступны, то выделяются, если выделение неудачно, выделяются обычные страницы. Если установить параметр в значение "on", то при невозможности выделить HP страницу обычные страницы не будут выделяться и экземпляр может не запуститься, если нехватка возникнет при запуске экземпляра. Не стоит использовать значение "`on`" не обеспечив достаточное количество страниц HP.

Проверим что большие страницы доступны процессам:

```
root@tantor:~# cat /proc/meminfo | grep Huge
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:     300
HugePages_Free:      300
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:          614400 kB
```

Перезапустим экземпляр:

```
root@tantor:~# systemctl restart postgresql
```

Проверим, что страницы были выделены:

```
root@tantor:~# cat /proc/meminfo | grep Huge
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:     300
HugePages_Free:      281
HugePages_Rsvd:       72
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:          614400 kB
```

Выделено в виртуальном адресном пространстве 72 страницы, что равно 144Мб. Из них пул буферов (`shared_buffers=128MB`) занимает 128Мб. Выделено, но не использовано, так как доступа к этим страницам не было (пул буферов не заполнен), поэтому команда free --mega покажет, что после запуска экземпляра свободная память уменьшилась всего на 8Мб:

```
root@tantor:~# free --mega
               total        used        free      shared  buff/cache   available
Mem:            2074        1378         371          12         505         696
Mem:            2074        1386         363          14         506         688
```

Использование экземпляром Huge Pages

- Huge Pages используются для разделяемого пула буферов и памяти Dynamic Shared Memory (DSA), которую используют параллельные процессы
- память выделяется способом задаваемым параметром shared_memory_type.
- в linux выделение памяти большими страницами поддерживается только способом mmap ("анонимные" страницы).
- Huge Pages не вытесняются в своп.
- если этого объема памяти будет недостаточно, то параллельные процессы будут дополнительно выделять (а потом освобождать) память страницами 4Кб и способом, задаваемым в параметре dynamic_shared_memory_type:

```
postgres=# select name, setting, enumvals from pg_settings where name like '%memory_type';
            name            | setting |     enumvals
----------------------------+---------+------------------
 dynamic_shared_memory_type | posix   | {posix,sysv,mmap}
 shared_memory_type         | mmap    | {sysv,mmap}
```


Использование экземпляром Huge Pages

Наибольший размер в экземпляре занимает пул буферов. В нагруженных экземплярах обычно используется распараллеливание выполнения команд. Выделение HP относительно долгая операция, поэтому **Huge Pages используются только для разделяемого пула буферов и памяти, которую используют параллельные процессы (parallel workers)**. Размер пула буферов задается параметром shared_buffers. Память выделяется способом, задаваемым параметром shared_memory_type. В linux выделение памяти большими страницами поддерживается способом mmap ("анонимные" страницы). Большие страницы не вытесняются в своп.

HP могут также использоваться параллельными процессами. Объем резервируемых HP задается параметром **min_dynamic_shared_memory**. Если этого объема памяти будет недостаточно, то параллельные процессы будут дополнительно выделять (а потом освобождать) память способом, задаваемым в параметре dynamic_shared_memory_type:

```
postgres=# select name, setting, enumvals from pg_settings
           where name like '%memory_type';
            name            | setting |     enumvals
----------------------------+---------+------------------
 dynamic_shared_memory_type | posix   | {posix,sysv,mmap}
 shared_memory_type         | mmap    | {sysv,mmap}
```

Значения по умолчанию оптимальны и менять их не стоит.

posix - память выделяется страницами **обычного размера 4Кб** системным вызовом shm_open. **Не стоит использовать значение mmap для параметра dynamic_shared_memory_type**, так как при использовании mmap используется (создается, если не создана) директория на диске PGDATA/pg_dynshmem и файлы в ней будут использоваться для отображения разделяемой параллельными процессами памяти, если им не хватило памяти зарезервированной **min_dynamic_shared_memory**. При исчерпании **min_dynamic_shared_memory** параллельные процессы выделяют память обычными страницами. **Для параметра же shared_memory_type нужно использовать значение mmap.**

Возможность использования экземпляром Huge Pages задается параметром huge_pages. Значение по умолчанию **try**. Это означает, что если при запуске экземпляра удастся выделить Huge Pages, то они будут выделены и использоваться. Если не удастся, то не будут использоваться вообще.

https://www.cybertec-postgresql.com/en/huge-pages-postgresql/

Если параметр конфигурации PostgreSQL `huge_pages=on`, и страниц `HugePages` не хватит при запуске экземпляра, то экземпляр не запустится.

```
root@tantor:~# sysctl -w vm.nr_hugepages=300
vm.nr_hugepages = 300
root@tantor:~# systemctl restart postgresql
postgres@tantor:~$ psql -p 5435 -c "show shared_buffers;"
 shared_buffers
----------------
 128MB
(1 row)
root@tantor:~# cat /proc/meminfo | grep HugeP
HugePages_Total:     300
HugePages_Free:      281
HugePages_Rsvd:       72
HugePages_Surp:        0
```

Было выделено 72 двухмегабайтных больших страницы. Из них 64 страницы непосредственно под разделяемый пул. Значение `HugePages_Rsvd` может не увеличиться, но тогда на выделение HP укажет уменьшится значениея `HugePages_Free`.

```
postgres@tantor:~$ psql -p 5435 -c "alter system set min_dynamic_shared_memory='200MB';"
ALTER SYSTEM
postgres@tantor:~$ sudo systemctl restart  postgresql
postgres@tantor:~$ cat /proc/meminfo | grep HugeP
HugePages_Total:     300
HugePages_Free:      280
HugePages_Rsvd:      171
HugePages_Surp:        0
```

Было выделено на 200МБ больше для использования параллельными процессами.

Список процессов, использующих HP:

```
root@tantor:~# grep "KernelPageSize:     2048 kB" /proc/[[:digit:]]*/smaps | awk {'print $1'} | cut -d "/" -f3 | sort | uniq
62133
...
```

Размер и тип HP, используемых процессом:

```
root@tantor:~# cat /proc/62133/smaps_rollup | grep tlb
Shared_Hugetlb:    30720 kB
Private_Hugetlb:   10240 kB
```

Это процесс postgres из Astralinux PostgreSQL.

Есть возможность проверить на остановленном экземпляре, сколько страниц HP с текущими настройками параметров конфигурации может запросить экземпляр при запуске:

```
postgres@tantor:~$ /usr/lib/postgresql/15/bin/postgres -c config_file=/etc/postgresql/15/main/postgresql.conf -D /var/lib/postgresql/15/main -C shared_memory_size_in_huge_pages
91
```

Выдано число 91, при запуске было выделено 72 страницы.

Для `dynamic_shared_memory_type=posix` страницы не свопятся, при этом создаются отображения памяти в виде файлов директории, смонтированной на файловой системе типа `tmpfs`:

```
postgres@tantor:~ $ ls -al /dev/shm
total 1136
-rw-------  1 postgres postgres 1048576 PostgreSQL.1357156412
-rw-------  1 postgres postgres  108576 PostgreSQL.2756012128
-rwx------  1 astra    astra         32 sem.user_1000_label_none
```

В 17 версии появился описательный (для чтения) параметр `huge_pages_status`, который показывает используются ли HP экземпляром.

Прозрачные огромные страницы (Transparent Huge Pages)

- замедляют работу процессов СУБД
- не выделены, если
  - `AnonHugePages:  0 kB`
- проверить включено ли использование:

```
root@tantor:~# cat
/sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
root@tantor:~# cat
/sys/kernel/mm/transparent_hugepage/defrag
always defer defer+madvise [madvise] never
```


Прозрачные огромные страницы (Transparent Huge Pages)

Помимо Huge Pages в linux доступно использование прозрачных огромных страниц (Transparent Huge Pages, THP), которые появились в ядре linux 2.6.38 в 2012 году.

THP замедляет работу СУБД и в настоящее время THP использовать не стоит.

На использование THP указывает строка `AnonHugePages`:

```
root@tantor:~# cat /proc/meminfo
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
...
```

Проверить отключены ли THP можно командой:

```
root@tantor:~# cat /sys/kernel/mm/transparent_hugepage/enabled
always [madvise] never
```

Квадратными скобками выделено текущее значение. Значение **never** отключает использование THP. Значение **madvise** позволяет процессам запрашивать использование THP системным вызовом `madvise()`. PostgreSQL не использует такой системный вызов, поэтому достаточно проверить, что значение параметра не установлено в `always`.

С THP используется "дефрагментация" страниц THP. Проверить режим дефрагментации можно командой:

```
root@tantor:~# cat /sys/kernel/mm/transparent_hugepage/defrag
always defer defer+madvise [madvise] never
```

Значение `always` или `defer` приводит к синхронной дефрагментации (direct compaction), блокирующей работу процессов. По большей части неоптимизированность дефрагментации и медленное перемещение огромных ("huge") диапазонов памяти является причиной замедления работы приложений при использовании THP.

Изменить значения можно командами:

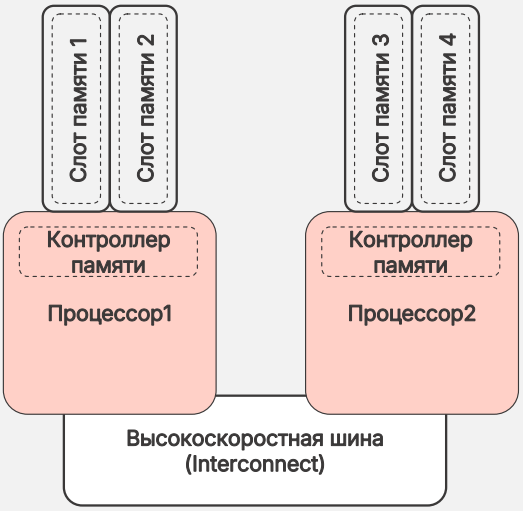
```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

Чтобы сохранить отключенное состояние после перезагрузки, а также чтобы в процессе запуска linux не использовались THP можно указать это в параметрах загрузчика linux.

Неравномерный доступ к памяти (NUMA)

Non-Uniform Memory Access (неравномерный доступ к памяти):

- каждый процессор имеет локальную физическую память и получает к ней доступ обычным образом через свой контроллер памяти. Помимо этого каждый процессор имеет доступ к локальной памяти других процессоров через более медленную шину ввода-вывода
- PostgreSQL оптимизирован для работы с однородным доступом к физической памяти и не оптимизирован для работы с NUMA



Неравномерный доступ к памяти (NUMA)

NUMA (Non-Uniform Memory Access, неравномерный доступ к памяти) это аппаратная архитектура, в которой каждый процессор имеет локальную физическую память и получает к ней доступ обычным образом через свой контроллер памяти. Помимо этого каждый процессор имеет доступ к локальной памяти других процессоров через более медленную шину ввода-вывода.

UMA (Uniform Memory Access, однородный доступ к памяти) используется в SMP (симметричная мультипроцессорность). Процессоры используют физическую память одновременно. Скорость доступа не зависит ни от того, какой именно процессор обращается к памяти, ни от того, какой чип памяти содержит нужные данные. Каждый процессор может использовать свои локальные кэши.

PostgreSQL оптимизирован для работы с однородным доступом к физической памяти (UMA) и не оптимизирован для работы с NUMA хотя бы потому что буферный кэш однородный.

При использовании оборудования, имеющего возможность использовать NUMA, в firmware (BIOS) такого оборудования может присутствовать параметр с названием "Node Interleaving" или "Memory Nodes" (чередование узлов памяти). Этот параметр нужно включить, тогда наличие NUMA не будет представлено (represent) операционной системе и память, выделяемая в операционной системе будет автоматически распределяться между узлами памяти (memory nodes).

Если этого не сделать, то по умолчанию разделяемые структуры памяти (буферный кэш, журнальный кэш и другие) будут выделяться по возможности в локальной памяти одного процессора (на котором запустился процесс postgres) и все остальные процессоры будут иметь медленный доступ (через более медленную шину ввода-вывода) к разделяемым структурам памяти. Это будет хуже, чем если бы физическая память, в которой выделяются разделяемые структуры памяти, была равномерно распределена между локальной памятью всех процессоров (Memory Nodes).

У каждого процессора свой интерфейс доступа к памяти и свой содержимое кэшей (L1,L2,L3) и буферов (TLB). При наличии нескольких процессоров может быть оптимальным, чтобы процессы по возможности выполнялись на одном и том же процессоре, а не мигрировали между процессорами. Миграция между ядрами одного процессора не играет роли. Задача привязки процессов к процессорам называется CPU affinity. Актуальность задачи возрастает, если число активных процессов намного больше числа ядер процессоров.

2-2

Нехватка памяти

Out Of Memory (OOM)

- процессам посылается сигнал `SIGKILL`
- лучшим кандидатом считается процесс, который
  - освободит максимум памяти
  - является наименее важным для системы
  - в идеале должен остановиться один процесс
- оценка для каждого процесса заранее расчитывается и записана в `/proc/PID/oom_score`
- сообщения об остановке процессов записываются в журнал операционной системы

Out Of Memory (OOM)

Операционная система старается задействовать всю оперативную память, чтобы она "не простаивала". Память, которая не используется процессами, используется страничным кэшем. В страничном кэше часть страниц может быть "грязной" и быстро такая память не освобождается, так как страницы нужно записать на диск. Если процессы запрашивают память, то для выделения памяти освобождается часть страничного кэша. Чтобы не возникло ситуации, когда выделение выполняется долго и замедляется работа всех процессов, в linux может освобождаться память путем принудительной остановки процессов, которые потребляют много памяти. Этим занимается процесс oom killer. Он останавливает процессы передавая им сигнал `SIGKILL` .

Алгоритм oom killer, по которому выбирается процесс или процессы для остановки считается "эвристическим". Алгоритм учитывает много параметров и может меняться от версии к версии linux. Лучшим кандидатом считается процесс, который освободит максимум памяти, а также является наименее важным для системы. Также желательным является остановить меньшее количество процессов, поэтому выбираются процессы, выделившие много памяти. **Оценку для каждого процесса можно посмотреть в `/proc/PID/oom_score`. Чем больше число, тем более вероятно что именно этот процесс будет остановлен.**

Пример сообщения oom kill в журнале операционной системы:

```
Out of memory: Kill process 58302 (postmaster) score 837 or sacrifice child
Killed process 58302 (postmaster) total-vm:72328760kB, anon-rss:55470760kB,
file-rss:4753180kB
```

"`total-vm`" - размер **виртуальной** памяти, которую использует процесс. **"`-rss`" - часть виртуальной памяти, которая отображена на оперативную память (выделена и используется)**.

"`anon-rss`" - память, выделенная процессом в физических страницах оперативной памяти и не имеет отображения в файлы и устройства (нет имени, анонимная).

Например, если процесс выполнит системный вызов `malloc()` выделив 1Гб и будет в выделенный 1Гб писать или читать, то "`anon-rss`" и "`total-vm`" увеличатся на 1Гб. Если процесс не будет писать и читать, то "`total-vm`" увеличится на 1Гб, а "`anon-rss`" не изменится.

"`file-rss`" - память, выделенная в физических страницах оперативной памяти и отображаемая на файлы или устройства. "`file-rss`" будет высоким, если открыть большой файл на чтение и прочесть его содержимое.

https://www.baeldung.com/linux/processes-memory-short

Размер Резидентного Набора (RSS)

- RSS (резидентный размер)- объем памяти, выделенных процессу и в находящихся в физической памяти
- PSS (пропорциональный размер) дает полноценное представление о распределении физической памяти между процессами и разделяемыми библиотеками
- привлекательность процесса для `oom kill`:
  - `/proc/'$PID'/oom_score`

Размер резидентного набора (RSS)

Рассмотрим, что означает размер памяти "**-rss**", который отображается в сообщении OOM kill и статистиках linux.

RSS (Размер Резидентного Набора) - объем памяти, выделенных процессу операционной системой и в настоящее время находящихся в ОЗУ (физической памяти). Не учитываются страницы памяти в пространстве подкачки (размер swap). При этом учитывается физическая память, выделенная для разделяемых библиотек, которые использует процесс.

RSS завышает использование памяти.

PSS (Proportional Set Size) дает полноценное представление о распределении физической памяти между процессами и разделяемыми библиотеками.

Для определения PSS можно использовать утилиту `smem` или команду:

```
for PID in $(pgrep "postgres"); do awk '/Pss/ {PSS+=$2} END { getline cmd < "/proc/'$PID'/cmdline"; sub("\0", " ", cmd); getline oom < "/proc/'$PID'/oom_score"; printf "%.0f -%s-> %s (PID %s) \n", PSS, oom, cmd, '$PID'}' /proc/$PID/smaps; done|sort -n -r
```

Пример: если библиотека использует 100Кб, а библиотеку использует 10 процессов, то в RSS памяти каждого процесса дополнительно учитывается по 100Кб. PSS в таком примере добавит 10Кб к памяти каждого процесса, то есть разделит объем памяти, используемый библиотекой пропорционально между процессами, использующими библиотеку. Конечно, один процесс мог заставить библиотеку активно использовать память под свое обслуживание, но для цели оценки используемой памяти это не играет роли.

Пример:

PSS Без HP:

```
66573 -85-> /usr/lib/postgresql/15/bin/postgres (PID 58302)
5799 -669-> postgres: 15/main: walwriter   (PID 58308)
3726 -669-> postgres: 15/main: autovacuum launcher   (PID 58309)
3212 -668-> postgres: 15/main: logical replication launcher   (PID 58310)
2913 -668-> postgres: 15/main: background writer   (PID 58306)
1716 -668-> postgres: 15/main: checkpointer   (PID 58305)
1507 -668-> postgres: 15/main: logger   (PID 58304)
```

Сумма: 66573 (у процесса postgres)+18873 (первый столбец остальных строк)= 85446

```
root@tantor:~# free
               total        used        free      shared  buff/cache   available
Mem:         2025796      647160      846888       47436      611812     1378636
```

PSS с HP:

```
7583 -74-> /usr/lib/postgresql/15/bin/postgres (PID 58353)
2966 -668-> postgres: 15/main: autovacuum launcher   (PID 58359)
2724 -668-> postgres: 15/main: logical replication launcher   (PID 58360)
1528 -668-> postgres: 15/main: checkpointer   (PID 58355)
1508 -668-> postgres: 15/main: logger   (PID 58354)
1493 -668-> postgres: 15/main: walwriter   (PID 58358)
1469 -668-> postgres: 15/main: background writer   (PID 58356)
```

Сумма 7583 (у процесса postgres)+11688= 19271

```
root@tantor:~# free
               total        used        free      shared  buff/cache   available
Mem:         2025796     1022900      471092       12624      577048     1002896
```

Вывод команды **free** по умолчанию в килобайтах. После увеличения числа огромных страниц, в выводе команды значения free и available уменьшаются, а значение used увеличивается.

oom_score у процесса postgres немного уменьшился за счет того, что стало учитываться существенно меньше памяти: для PSS было 66573, стало 7583. Для RSS аналогичное уменьшение.

При использовании HP размер памяти процессов экземпляра, который показывает PSS, уменьшился в несколько раз (было 85446, стало 19271). Объем памяти, с которой работает экземпляр остался примерно тем же.

Оценить насколько увеличится производительность по показателям сложно. Определить механизм увеличения ещё сложнее. На небольших объемах памяти роль может играть то, что HP не вытесняются из памяти, что обеспечивает более предсказуемую работу. На больших объемах оперативной памяти быстрый доступ будет обеспечиваться к страницам, ссылки на которые находятся в TLB.

Пример: число не занятых операционной системой (под страницы ядра linux и библиотек) записей в TLB 1024. Для обычных страниц размером 4Кб быстрый доступ (через TLB) будет к 1000*4Кб=4Мб памяти. Для HP размером 2Мб быстрый доступ будет к 2Гб памяти.

В любом случае, использование HP с СУБД не приведет к деградации производительности, в отличие от THP.

https://www.percona.com/blog/why-linux-hugepages-are-super-important-for-database-servers-a-case-with-postgresql/

Параметр oom_score_adj

- значение устанавливается после запуска для каждого процесса:
  › echo -900 > /proc/58253/oom_score_adj
- изменяет /proc/PID/oom_score для этого процесса
- значение -1000 для процесса postgres устанавливается в файле службы СУБД Tantor tantor-se-server-16.service
- у остальных процессов экземпляра по умолчанию не меняется и установлено в ноль

Параметр oom_score_adj

В примере приведенном ранее выводился oom_score процессов экземпляра. У процесса postgres oom_score был 74 или 85, у остальных процессов 668-669. За счет чего скоринг был понижен? За счет того, что было изменено значение oom_score_adj и установлено -900

```
root@tantor:~# cat /proc/58253/oom_score_adj
-900
```

У остальных процессов экземпляра используется значение по умолчанию ноль:

```
root@tantor:~# cat /proc/58359/oom_score_adj
0
```

Если поменять на лету значение процессу postgres :

```
root@tantor:~# echo 0 > /proc/58253/oom_score_adj
```

то значение oom_score у postmaster станет примерно таким же как у остальных процессов экземпляра:

```
root@tantor:~# cat /proc/58253/oom_score
673
```

Процесс postgres порождает другие процессы экземпляра и значение oom_score_adj не наследуется порожденными процессами.

Остановка любого процесса сигналом SIGKILL приводит к рестарту экземпляра процессом postgres. Однако, остановка процесса postgres сигналом SIGKILL хуже. Поэтому процессу postgres устанавливается oom_score_adj. СУБД Tantor устанавливает значение "-1000" и oom_score станет равно нулю. Это делается директивой в файле службы СУБД Tantor tantor-se-server-16.service:

```
# Запрещает OOM kill убивать процесс postmaster
OOMScoreAdjust=-1000
# ... но позволяет убивать процессы, которые порождает postmaster
Environment=PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
Environment=PG_OOM_ADJUST_VALUE=0
```

Можно поменять oom_score_adj остальным процессам экземпляра заменив в файле службы 0 например на -300. **Менять oom_score_adj имеет смысл только, если на хосте работают процессы нескольких экземпляров или процессы, потребляющие много памяти.** Параметры устанавливают переменные окружения перед запуском процесса postgres.

https://docs.tantorlabs.ru/tdb/ru/15_6/se/kernel-resources.html#LINUX-MEMORY-OVERCOMMIT

Параметр vm.overcommit_memory

- имеет три значения:
  - 0 - значение по умолчанию. Выделяется столько памяти, сколько запросит процесс.
  - 1 - с СУБД не используется.
  - 2 - отказ в выделении памяти, если суммарный объем выделенной памяти превысит размер пространства подкачки плюс объем физической памяти умноженной на значение в процентах параметра vm.overcommit_ratio (по умолчанию 50%) или абсолютное значение заданное параметром vm.overcommit_kbytes
- **если раздел подкачки отключён и vm.overcommit_ratio<100, то устанавливать значение 2 не стоит, оно должно быть 0**
- значение параметра vm.overcommit_ratio не играет роли при значениях параметра vm.overcommit_memory равных 0 или 1

Параметр vm.overcommit_memory

Рассмотрим параметры, которые напрямую влияют на срабатывание OOM kill или выдачу процессу сообщения о нехватке памяти. Формула, связывающая параметры, приведена на следующем слайде. Изменение одного параметра, без учёта значений остальных может привести к срабатыванию OOM kill или отказу в выделени памяти даже в случае, если памяти достаточно.

Параметр vm.overcommit_memory может быть установлен в одно из трёх значений:

•0 - значение по умолчанию. Выделяется столько памяти, сколько запросит процесс. В swap/reserved попадают только те страницы, которые будут использоваться процессом. Системный вызов выделения памяти mmap (описание вызова можно посмотреть командой man mmap) без опции MAP_NORESERVE не проверяет сколько доступно памяти и сообщает процессу, что память выделена. При попытке использовании выделенной памяти есть вероятность срабатывания oom-kill.

•1 - проверка сколько памяти есть в наличии не производятся. Этот режим обычно используется для научных задач, работающих с большими массивами. С СУБД не используется.

•2 - отказ в выделении памяти, если суммарный объем выделенной памяти превысит размер пространства подкачки плюс объем физической памяти умноженной на значение в процентах параметра vm.overcommit_ratio (**по умолчанию 50%, значение выбрано исходя из того, что раздел подкачки равен 50% физической памяти**) или абсолютное значение, заданное параметром vm.overcommit_kbytes.

Значение параметра vm.overcommit_ratio не играет роли при значениях параметра vm.overcommit_memory. равных 0 или 1.

Выбор для хоста, обслуживающего СУБД, сводится к значениям 0 и 2.

Текущие значения объема памяти, которые можно выбрать можно посмотреть командой:

```
cat /proc/meminfo | grep Commit
```

Если раздел подкачки отключён и vm.overcommit_ratio<100, то не стоит устанавливать значение 2.

При установке значения 2 вместо срабатывания oom kill процессы будут получать ошибку выделения памяти.

https://www.kernel.org/doc/Documentation/vm/overcommit-accounting

Установка значений overcommit и swap

- при обслуживании в linux только СУБД стоит установить значения **`vm.overcommit_ratio=100`**, **`vm.overcommit_memory=2`**
  - при таких значениях срабатывание oom kill маловероятно и будет использоваться вся физическая память
  - станет возможным отключить раздел подкачки без увеличения вероятности срабатывания oom kill
  - если установить значение `vm.overcommit_ratio>100` увеличивается вероятность срабатывания oom kill
- общий размер виртуальной памяти, который может быть выделен определяется по формуле: **`CommitLimit =(total_RAM-total_huge_TLB)* vm.overcommit_ratio/100+total_swap`**
- активное использование раздела подкачки занимает пропускную способность шины ввода-вывода

Установка значений overcommit и swap

Общий размер виртуальной памяти, который может быть выделен определяется по формуле:
`CommitLimit =(total_RAM-total_huge_TLB)*vm.overcommit_ratio/100+total_swap`
Эта формула связывает размер разделов (файлов) подкачки и памяти, которая может выделяться процессам.

Использование подкачки нежелательно тем, что возникает торможение и ухудшается отзывчивость всех процессов, так как используется шина ввода-вывода. При использовании под раздел подкачки NVMe скорость ввода-вывода сравнима со скоростью доступа к физической памяти и использование подкачки может иметь смысл. Недостатком может быть то, что линии PCIe будут использоваться устройством NVMe. Если пропускная способность PCIe является узким местом, то замедлится доступ к дискам, на которых располагаются файлы кластера баз данных и в этом случае использование swap нежелательно.

В чем преимущество наличия даже небольшого раздела подкачки? Процессы запрашивают память с округлением и часть страниц выделенной памяти могут не использовать до освобождения памяти. такие страницы могут быть вытеснены в swap без влияния на производительность. Без swap такие страницы находятся в физической памяти и физическая память "недоиспользуется", а могла бы использоваться (при нормальной работе под страничный кэш).

Какие значения выставить при отсутствии подкачки и при этом уменьшить вероятность срабатывания OOM-kill? Нужно установить `vm.overcommit_ratio=100` и `vm.overcommit_memory=2`. С такими значениями по формуле `CommitLimit=total_RAM`. При таких значениях срабатывание oom kill маловероятно и будет использоваться вся физическая память. Если установить значение `vm.overcommit_ratio` больше `100`, то увеличивается вероятность срабатывания oom kill.

Если oom kill срабатывает и посылает сигнал SIGKILL процессу экземпляра PostgreSQL, то процесс postmaster останавливает все процессы экземпляра и запускает их снова. Получается эффект перезапуска экземпляра. Если postmaster не сможет перезапустить процессы, то postmaster останавливается и экземпляр становится недоступен. Также возможно, что при перезапуске процессов какой-то из фоновых процессов (checkpointer) подвиснет. В таком случае экземпляр придётся останавливать командой `pg_ctl stop -m immediate`. Команда `pg_stl stop` в таком случае подвиснет, а `systemctl stop` не остановит процессы.

Параметр `vm.swappiness`

- по умолчанию значение равно 60
- изменить значение можно без рестарта операционной системы
- оптимальное значение значение около 10
- **значение 0 не стоит использовать**
- влияет на то какие части памяти будут кандидатами на вытеснение в swap:
  - `anon_prio = swappiness;`
  - `file_prio = 200 - anon_prio;`
- Пример сообщения в журнале linux об остановке процесса:

```
Out of memory: Kill process 58302 (postmaster) score 837 or sacrifice child
Killed process 58302 (postmaster) total-vm:72328760kB, anon-rss:55470760kB, file-rss:4753180kB
```


Параметр vm.swappiness

Параметр `vm.swappiness` влияет на вероятность использования подкачки, если он используется.

По умолчанию значение равно 60:

```
root@tantor:~# sysctl -a | grep swap
vm.swappiness = 60
```

Для хоста с физической памятью больше 8-16Гб значение 60 не оптимально. Оптимальное значение значение около 10. Изменять значение стоит с небольшим шагом, следя за результатом (будут ли вытеснены какие-либо страницы в раздел подкачки, при нормальной работе они не должны вытесняться). Изменить значение можно без рестарта операционной системы, отредактировав файл `/etc/sysctl.conf` и применив изменения командой `sysctl -p`.

```
/*
* With swappiness at 100, anonymous and file have the same priority.
* This scanning priority is essentially the inverse of IO cost.
*/
anon_prio = swappiness;
file_prio = 200 - anon_prio;
```

Значение 0 указывает ядру linux избегать вытеснения страниц из памяти настолько, насколько возможно. Это значение на современных версиях linux **не стоит** использовать, так как OOM killer может работать, даже если есть свободное пространство подкачки. OOM обычно легче завершить процесс, чем освобождать память из файлового кэша. Пример сообщения в журнале linux об остановке процесса:

```
Out of memory: Kill process 58302 (postmaster) score 837 or sacrifice child
Killed process 58302 (postmaster) total-vm:72328760kB, anon-rss:55470760kB,
file-rss:4753180kB
```

Подкачка (swap) не является обязательной и её можно отключить. С использованием подкачки операционная система может запускать программы, которым требуется больше памяти, чем доступно физически. Это также помогает предотвратить сбой системы, если ей не хватает оперативной памяти. Также повышается производительность за счет увеличения файлового кэша.

https://eklitzke.org/swappiness

Дедупликация страниц памяти (KSM)

Kernel Same-page Merging (дедупликация страниц памяти):

- может объединять страницы локальной памяти anon-rss разных процессов
- не объединяет file-rss и разделяемую память
- используется в виртуализации, а не с СУБД
- память сканирует в поисках одинаковых страниц служба linux

Проверить, что KSM отключён можно командой:

```
cat /sys/kernel/mm/ksm/run
0
```


Дедупликация страниц памяти (KSM)

KSM (kernel same-page merging, дедупликация страниц памяти) позволяет ядру linux начиная с версии 2.6.32 объединять одинаковые страницы памяти `anon-rss` (но не `file-rss`), используемые различными процессами в одну страницу для совместного использования. Разделяемая память не дедуплицируется. Одинаковые страницы локальной памяти процессов объединяются в одну и помечаемую как подлежащие копированию при изменении (записи в страницу), чтобы страницы разделялись при изменении одним из процессов.

Наибольший эффект имеет при запуске виртуальных машин с похожих образов.

Параметры и статистика KSM находятся в файлах директории `/sys/kernel/mm/ksm` :

```
ls /sys/kernel/mm/ksm
full_scans         merge_across_nodes  pages_to_scan   sleep_millisecs      use_zero_pages
general_profit     pages_scanned       pages_unshared  stable_node_chains
ksm_zero_pages     pages_shared        pages_volatile  stable_node_chains_prune_millisecs
max_page_sharing   pages_sharing       run             stable_node_dups
```

Для эффективной работы требуется служба ksm, которая сканирует память в поисках дубликатов страниц.

С СУБД не используется.

По умолчанию отключён. Проверить что отключён можно командой:

```
cat /sys/kernel/mm/ksm/run
0
```

Ноль - отключено.

Проверить, что ksm не работал можно командой:

```
cat /proc/vmstat | grep ksm
ksm_swpin_copy 0
cow_ksm 0
```

Ноль означает, что активности не было.

https://docs.kernel.org/admin-guide/mm/ksm.html

Выделение локальной памяти процессами экземпляра

- в PostgreSQL реализована логика управления памяти через "контексты памяти"
- используется вызов palloc(), который выделяет память в области памяти, называемой MemoryContext, который ещё называют "Arena"
- контексты памяти образуют иерархию
- TopMemoryContext - корень иерархии контекстов памяти серверного процесса
- **при запросе на выделение памяти, выделяется размер памяти являющийся ближайшей степенью 2 в большую сторону от запрошенного размера**
- если в контексте недостаточно свободной памяти, то к нему добавляется память **в двойном размере от начального**

Выделение памяти процессам экземпляра

Для снижения вероятности утечек памяти в PostgreSQL реализована логика управления памяти через "контексты памяти". Программный код не обращается каждый раз к операционной системе вызовами malloc(). Если нужно выделить память, что используется вызов palloc(), который выделяет память в области памяти, называемой MemoryContext, который ещё называют "Arena". MemoryContext примечателен тем, что может полностью освобождать память, выделенную под него и не нужно вести подсчёт многочисленных вызовов на выделение памяти. Главное, чтобы память выделялась в контексте с подходящим сроком жизни. По истечении срока жизни контекста опасности утечки (неосвобождение) памяти отсутствует. Например, в начале транзакции выделяется TopTransactionContext, память которого будет освобождена по завершению транзакции. Объем памяти выделенный контексту может быть значительным. Второй аспект контекстов - минимизировать число вызовов malloc() и выделять память заранее большими кусками с округлением по степеням двойки.

Контексты памяти образуют иерархию. TopMemoryContext - корень иерархии контекстов памяти серверного процесса. Все остальные контексты памяти имеют родительский контекст. Когда программный код удаляет контекст памяти (завершение транзакции, закрытие курсора, портала), удаляются все дочерние контексты.

Память может выделяться по логике Slab (slab.c). Это реализация MemoryContext, когда память выделяется **одинаковыми размерами (чанками)**. Размер чанка задаётся при создании контекста этого типа. **Куски памяти разного размера называются не чанками, а блоками**. Более часто используется логика AllocSet (aset.c) и Generation (generation.c, блоки со схожим сроком жизни). Особенностью логики управления памятью этих контекстов является то, что **при запросе на выделение памяти, выделяется размер памяти являющийся ближайшей степенью 2 в большую сторону от запрошенного размера** ("round request sizes up to the next power of 2"). Например, если запрашивается 600Кб, то выделяется 1Мб. Если запрашивается ровно 1Мб, то выделяется 1Мб.

Если в контексте недостаточно свободной памяти, то к нему добавляется память (вызовом malloc()) **в двойном размере от начального**. Блок может ограничиваться размером 1Гб или 2Гб (при использовании `enable-large-allocations`), но не во всех случаях. Выделенная с запасом часть памяти может не использоваться (читаться и писаться), но она будет учитываться в операционной системе, как выделенная в виртуальном адресном пространстве. Лучше не доводить операционную систему до нехватки физической памяти: страничный кэш linux должен быть большим (сравним по размеру с буферным кэшем). Страничный кэш linux использует почти всю не испльзуемую физическую память и является резервом для обслуживания запросов на выделение памяти.

Выделение памяти в generation.c:

```
/*
 * The first such block has size initBlockSize, and we double the
 * space in each succeeding block, but not more than maxBlockSize.
 */
blksize = set->nextBlockSize;
set->nextBlockSize <<= 1;
if (set->nextBlockSize > set->maxBlockSize)
  set->nextBlockSize = set->maxBlockSize;

/* we'll need a block hdr too, so add that to the required size */
required_size += Generation_BLOCKHDRSZ;

/* round the size up to the next power of 2 */
if (blksize < required_size)
  blksize = pg_nextpower2_size_t(required_size);

block = (GenerationBlock *) malloc(blksize);

if (block == NULL)
  return NULL;

context->mem_allocated += blksize;
```

В практике к этой главе будет пример, когда при загрузке файла размером 1Гб серверный процесс при чтении файла дампа размером 1Гб выделяет виртуальную память размером **6Гб**:

```
Killed process 12518 (postgres) total-vm:6523848kB, anon-rss:3151300kB
```

Пример анализа выделения и освобождения памяти контекстами:

https://dev.to/yugabyte/postgres-memory-allocation-and-os-memory-allocation-30f1

"operating system reporting 70M and the PostgreSQL level memory dump saying it released memory down to approximately 1M!

What we see is 'Arena 0', which is roughly put the administration of memory allocations of malloc() for this process, which has allocated from 'system' 63832064 bytes (60.9M), whilst actually in use (by PostgreSQL) is 917696 bytes (1M). What malloc() tries to do, is keep memory allocated to prevent having to deallocate and allocate over and over."

Если память под контекст была выделена, она не возвращается в операционную систему. Память будет возвращена в течение срока жизни серверного процесса, когда жизненный цикл контекста подойдёт к концу и контекст будет освобождён.

В докладе https://pgconf.ru/media/2024/10/21/12/613/Александров.pdf

рассматривается случай последовательного выделения памяти в 5 контекстах `CachedPlan` под 5 частных планов и один общий размером по 32Мб каждый в родительском контексте `CacheMemoryContext`. Выделенная под контекст память размером 192Мб (общая память серверного процесса в момент освобождения памяти в контексте USS 188.7M, PSS 189.5M, RSS 195.0M) не возвращалась операционной системе.

ОШИБКА: invalid memory alloc request size

- причина ошибки: попытка выделить блок памяти, превышающий ограничение 2Гб или 1Гб-1, установленный макросом `MaxAllocSize`
- в текстовых функциях типа lpad, rereat вставлены проверки, которые выдают ошибку: requested length too large
- если размер памяти после текста ошибки больше 2Гб, это означает, что размер памяти, который должен быть выделен рассчитан неверно и может указывать на наличие повреждения записей в блоках данных
  - также расчет может быть неверно выполнен из-за ошибок в библиотеках расширений, загруженных в память процесса
- пример команд, приводящих к ошибкам:

```
postgres=# create table a as select repeat('a', 1024 * 1024 * 1024 - 5);
ERROR:  invalid memory alloc request size 1073741887
postgres=# select repeat('x', 1024 * 1024 * 1024);
ERROR:  requested length too large
```


Выделение памяти процессам экземпляра

Программный код процесса экземпляра не обращается каждый раз к операционной системе вызовами malloc(). Если контекст выделяет память одинакового размера это называется чанком, если произвольного размера это называется блоком. При нехватке памяти в контексте, в операционную систему посылается вызов malloc(). Перед выполнением вызова производится проверка не превысит ли запрашиваемый размер ограничение:

```
#define AllocSizeIsValid(size)  ((Size) (size) <= MaxAllocSize)
```

которое установлено макросом:

```
#define MaxAllocSize ((Size) 0x3fffffff) /* 1 gigabyte - 1 */
```

Ошибка с текстом "invalid memory alloc request size" вызывается проверкой на превышение этого ограничения.

Аналогичные ошибки:

array size exceeds the maximum allowed

В текстовые функциях типа lpad(..), repeat(..) вставлены проверки, которые выдают менее пугающую ошибку:

requested length too large.

Выделение памяти типа Dynamic Shared Memory (DSA) также имеет проверки. Текст ошибки при превышении ограничений: "invalid DSA memory alloc request size".

Если размер памяти после текста ошибки больше 2Гб, это означает, что размер памяти, который должен быть выделен рассчитан неверно и может указывать на наличие повреждения записей в блоках данных. Также расчет может быть неверно выполнен из-за ошибок в библиотеках расширений, загруженных в память процесса.

Контексты могут выделять блоки и чанки размером больше 1Гб, но тогда будет выдана ошибка операционной системы - либо out of memory, либо Cannot allocate memory, либо процесс будет остановлен oom kill.

Общий размер контекста может превышать 1Гб, ограничивается выделение блока или чанка, размером больше 1Гб-1. Пример описания размеров контекстов:

```
CacheMemoryContext: 59375840 total in 21 blocks; 8112520 free (13 chunks)
CachedPlan: 34199136 total in 24 blocks; 7386392 free (0 chunks)
```

`0 chunks` означает, что контекст выделает память не чанками, а кусками переменного размера, то есть блоками.

Параметр enable_large_allocations

- параметр СУБД Tantor который увеличивает размер StringBuffer с 1 гигабайта до 2 гигабайт

```
postgres=# select * from pg_settings where name like '%large%'\gx
name       | enable_large_allocations
setting    | off
category   | Resource Usage / Memory
short_desc | whether to use large memory buffer greater than 1Gb, up to 2Gb
context    | superuser
vartype    | bool
boot_val   | off
```

- может устанавливаться на уровне сессии и утилитами pg_dump, pg_dumpall

```
postgres@tantor:~$ pg_dump --help | grep alloc
  --enable-large-allocations   enable memory allocations with size up to 2Gb
```

- проблема возникает со строкой таблицы config приложений 1С:ERP, Комплексная автоматизация, Управление производственным предприятием

Параметр enable_large_allocations

параметр СУБД Tantor 16.2, который **увеличивает размер StringBuffer в локальной памяти процессов экземпляра с 1 гигабайта до 2 гигабайт**. Размер одной строки таблицы при выполнении команд SQL должен поместиться в StringBuffer. Если не поместится, то любому клиенту с которым работает серверный процесс, выдастся ошибка, в том числе утилитам `pg_dump` и `pg_dumpall`. Размер поля строки таблицы всех типов не может превышать 1Гб, но столбцов в таблице может быть несколько и размер строки может превысить и гигабайт и несколько гигабайт.

Утилита `pg_dump` может отказаться выгружать такие строки, так как она не использует опцию `WITH BINARY` команды `COPY`. Для текстовых полей непечатный символ, занимающий один байт будет заменен последовательностью печатных символов размером 2 байта (например, \n) и текстовое поле может увеличиться в размере до двух раз.

```
postgres=# select * from pg_settings where name like '%large%'\gx
name       | enable_large_allocations
setting    | off
category   | Resource Usage / Memory
short_desc | whether to use large memory buffer greater than 1Gb, up to 2Gb
context    | superuser
vartype    | bool
boot_val   | off
```

и у утилит командной строки:

```
postgres@tantor:~$ pg_dump --help | grep alloc
  --enable-large-allocations   enable memory allocations with size up to 2Gb
```

Параметр можно установить на уровне сессии. StringBuffer выделяется динамически в процессе обработки каждой строки, а не при запуске серверного процесса. Если таких строк нет, параметр не влияет на работу серверного процесса.

Такая проблема возникает со строкой таблицы config приложений 1С:ERP, Комплексная автоматизация, Управление производственным предприятием. Пример:

```
pg_dump: error: Dumping the contents of table "config" failed: PQgetResult() failed.
Error message from server: ERROR: invalid memory alloc request size 1462250959
The command was: COPY public.config
(filename, creation, modified, attributes, datasize, binarydata) TO stdout;
```

2-3

Страничный кэш

Страничный кэш linux

Страничный кэш (кэш операционной системы, файловый кэш):

- кэш 4Кбайтных страниц отображаемых на файлы в файловой системе
- Под страничный кэш linux может использовать всю свободную (не занятую процессами и ядром) физическую память, за исключением части, размер которой косвенно определяется параметром `vm.min_free_kbytes`
  - параметр не позволяет использовать под страничный кэш всю физическую память
  - значение по умолчанию не больше 66Мб

Страничный кэш linux

Страничный кэш (кэш операционной системы, файловый кэш) - это кэш 4Кбайтных страниц, отображаемых на файлы в файловой системе. Текущий размер кэша:

```
root@tantor:~# cat /proc/meminfo | grep Cached
Cached:           898924 kB
SwapCached:            0 kB
```

В вывод команды включаются также отображаемые в память страницы исполняемого кода (исполняемых файлов и библиотек).

Под страничный кэш linux может использовать всю свободную память, за исключением части размер которой косвенно определяется параметром `vm.min_free_kbytes`. Этот параметр определяет пороговые значения (границы) срабатывания процессов kernel swap daemon (`kswapdN`), по которым процессы начинают и останавливают очистку физической памяти от грязных блоков путём их записи на их исходные места на диске. Этот параметр не позволяет использовать под страничный кэш всю физическую память. Также, если какой-то процесс хочет выделить память больше чем `vm.min_free_kbytes` и доступной (`free` или `cat /proc/meminfo | grep Available`) памяти нет или она фрагментирована, то срабатывает `oom killer`.

При нормальной работе linux объем незанятой под кэш памяти оказывается примерно равным значению `vm.min_free_kbytes`.

Значение по умолчанию (если физической памяти больше 4Гб) невелико и равно 66Мб:

```
root@tantor:~# sysctl -a | grep vm.min_free
vm.min_free_kbytes = 67584
```

Процент грязных страниц в физической памяти задаётся параметрами `vm.dirty_ratio`. и `vm.dirty_background_ratio`. Процент берётся от объема незанятой процессами и ядром физической памяти (available), а не от объема всей физической памяти.

По достижению `vm.dirty_background_ratio` начинается запись грязных страниц на диск: kswapd помечает страницы, bdflush инициирует запись, pdflush записывают. По достижении `vm.dirty_ratio` (значение должно быть больше первого) процессы, которые пишут в блоки файлов (грязнят страницы в кэше) блокируются. Это позволяет избежать `oom kill`, но вносит задержки в работу процессов (отзывчивость). Рекомендаций по точным значениям нет, иначе они были бы установлены. Типичный диапазон значений этих параметров 5-10% и 10-20%.

https://hydrusnetwork.github.io/hydrus/Fixing_Hydrus_Random_Crashes_Under_Linux.html

Доля изменённых ("грязных") страниц в кэше

- процент изменённых ("грязных") страниц в физической памяти задаётся параметрами `vm.dirty_background_ratio` и `vm.dirty_ratio`
- процент берётся от объема незанятой процессами и ядром физической памяти (available), а не от объема всей физической памяти.
- также есть параметры `vm.dirty_background_bytes` и `vm.dirty_bytes`, задающие абсолютные значения, по умолчанию ноль
- текущие значения порога срабатывания начала записи грязных страниц на их исходные места (сначала в фоновом режиме без блокировки, потом с блокировкой), в количестве страниц:

```
root@tantor:~# cat /proc/vmstat | grep dirty
nr_dirty 4
nr_dirty_threshold 152308
nr_dirty_background_threshold 76061
```


Доля изменённых ("грязных") страниц в кэше

Также есть параметры `vm.dirty_background_bytes` и `vm.dirty_bytes`, задающие абсолютные значения, по умолчанию ноль. Если их устанавливать, то не меньше 1Гб и 100Мб.

Команда `echo 1 > /proc/sys/vm/drop_caches` освобождает все чистые страницы кэша. Команда может использоваться перед запуском тестов, измеряющих производительность, чтобы на повторные запуски тестов не влияло то, что после предыдущего запуска теста страницы были закэшированы. После использования команды на хосте с большим объемом физической памяти может освободиться большое количество памяти. **Процент грязных страниц резко увеличится, так как процент удерживаемых грязных страниц рассчитывается как процент от свободной физической, а не от общей физической памяти**. Состояние незаполненного кэша (большого объема свободной памяти) не характерно для обычного режима, в котором работает операционная система linux.

Посмотреть установленные значения:

```
root@tantor:~# sysctl -a | grep dirty\.*ratio
vm.dirty_background_ratio = 10
vm.dirty_ratio = 20
```

Объем доступной процессам (available) памяти:

```
root@tantor:~# free
               total        used        free      shared  buff/cache   available
Mem:         3908744      476408      428256      112560     3004080     3040044
```

Текущие значения **порога срабатывания начала записи грязных страниц на их исходные места**. Сначала в фоновом режиме без блокировки, потом с блокировкой процессов, которые грязнят (меняют содержимое) страниц. Измеряется в количестве страниц:

```
root@tantor:~# cat /proc/vmstat | grep dirty
nr_dirty 4
nr_dirty_threshold 152308
nr_dirty_background_threshold 76061
```

Грязных страниц немного, кэш занят чистыми страницами.

Порог начала срабатывания фоновой записи примерно соответствует формуле:

`nr_dirty_background_threshold (76061) ~= 3040044` (свободная память в килобайтах)`*dirty_background_ratio/4` (размер страницы в килобайтах)

https://www.yugabyte.com/blog/linux-performance-tuning-memory-disk-io/

Фрагментация памяти

- на фрагментацию виртуального адресного пространства памяти указывает то, что кусков памяти размером больше 16Кб выше 16Мб нет:

```
root@tantor:~# cat /proc/buddyinfo
Node 0,zone      DMA      1      0      0      1      2      1      1      0      1      1      3
Node 0,zone    DMA32   3173    856    529      0      0      0      0      0      0      0      0
Node 0,zone   Normal  19030   8688   7823      0      0      0      0      0      0      0      0
```

- пример, когда большая часть памяти кусками размером не меньше 4Мб:

```
root@tantor:~# cat /proc/buddyinfo
Node 0,zone      DMA      0      0      0      0      0      0      0      0      0      1      3
Node 0,zone    DMA32      1      2      4      2      3      4      2      2      2      2    863
Node 0,zone   Normal      1      0      3     91     14     10      9      5      5     23    352
```

- каждая зона делится на части адресного пространства памяти размером (4096 байт *2^n) : 4Кб, 8Кб, 16Кб, 32Кб, 64Кб, 128Кб, 256Кб, 512Кб, 1Мб, 2Мб, 4Мб.

Фрагментация памяти

На фрагментацию виртуального адресного пространства указывает то, что кусков памяти размером больше 16Кб выше 16Мб нет:

```
root@tantor:~# cat /proc/buddyinfo
Node 0,zone      DMA      1      0      0      1      2      1      1      0      1      1      3
Node 0,zone    DMA32   3173    856    529      0      0      0      0      0      0      0      0
Node 0,zone   Normal  19030   8688   7823      0      0      0      0      0      0      0      0
```

Пример, когда большая часть памяти кусками не меньше 4Мб:

```
root@tantor:~# cat /proc/buddyinfo
Node 0,zone      DMA      0      0      0      0      0      0      0      0      0      1      3
Node 0,zone    DMA32      1      2      4      2      3      4      2      2      2      2    863
Node 0,zone   Normal      1      0      3     91     14     10      9      5      5     23    352
```

Node 0 - номер физического процессора. Зоны:
1)DMA - виртуальная память со смещением от нуля до 16Мб
2)DMA32 от 16Мб до 4Гб
3)Normal - от 4Гб и до 2^48

Каждая зона делится на части адресного пространства памяти размером (4096 байт *2^n) : 4Кб, 8Кб, 16Кб, 32Кб, 64Кб, 128Кб, 256Кб, 512Кб, 1Мб, 2Мб, 4Мб.

На основе этих данных рассчитывается производная метрика "индекс":

```
root@tantor:~# cat /sys/kernel/debug/extfrag/extfrag_index
Node 0, zone      DMA -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
Node 0, zone    DMA32 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
Node 0, zone   Normal -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000  0.964  0.982  0.991  0.996
```

ближе к **-1** (часто -1.000 называют минус тысячей) всё хорошо, ближе к 1 - память фрагментирована, ближе к 0 - мало памяти (надо освобождать). Предполагается, что по нему можно определять нужно ли освободить память или дефрагментировать.

Как выглядит фрагментация с точки зрения администратора:
https://habr.com/ru/companies/odnoklassniki/articles/266005/
История изменений в ядре для работы с фрагментацией:
https://habr.com/ru/companies/ruvds/articles/673024/

Дефрагментация памяти

- значение по умолчанию для минимального размера свободной оперативной памяти может быть недостаточным:

```
root@tantor:~# sysctl -a | grep min_free
vm.min_free_kbytes = 67584
```

- дефрагментация запускается при уменьшении свободной памяти ниже `vm.min_free_kbytes`
- рекомендуется установить в 2% от размера физической памяти
- параметр `vm.watermark_scale_factor` задает вторую границу для процесса дефрагментации
- значение по умолчанию **0.1% (число 10)** от размера **свободной** физической памяти:

```
root@tantor:~# sysctl -a | grep watermark
vm.watermark_boost_factor = 15000
vm.watermark_scale_factor = 10
```

- рекомендуется установить 1% (число 100)

Дефрагментация памяти

Если свободной (она занята страничным кэшем) памяти много, при этом возникают ошибки нехватки памяти, например, запускается oom kill, то в этом случае можно эффективно устранить фрагментацию просто освободив куски памяти, занимаемые "чистыми" страницами кэша:

```
root@tantor:~# echo 1 > /proc/sys/vm/drop_caches
```

или

```
postgres@tantor:~# echo 1 | sudo tee /proc/sys/vm/drop_caches
```

Грязные страницы записываются на диск командой **sync**.

Также можно периодически во время минимальной загрузки операционной системы запускать дефрагментацию командой:

```
root@tantor:~# echo 1 > /proc/sys/vm/compact_memory
```

Также можно настроить работу дефрагментации. Значение по умолчанию для минимального размера свободной оперативной памяти может быть недостаточным:

```
root@tantor:~# sysctl -a | grep min_free
vm.min_free_kbytes = 67584
```

Страничный кэш является основным фрагментатором виртуального адресного пространства. Дефрагментация запускается при уменьшении свободной памяти ниже `vm.min_free_kbytes`. Рекомендуется установить в 2% от размера физической памяти. Рекомендуемый диапазон 1-3%.

Параметр `vm.watermark_scale_factor` задает вторую границу для процесса дефрагментации. Значение по умолчанию 0.1% (число 10) от размера **свободной** физической памяти:

```
root@tantor:~# sysctl -a | grep watermark
vm.watermark_boost_factor = 15000
vm.watermark_scale_factor = 10
```

Максимальное значение `vm.watermark_scale_factor=1000`, что обозначает 10% свободной физической памяти. Рекомендуется установить в 1%, то есть значение 100.

Страничный кэш linux не отключается и не ограничивается в размере.

https://www.alibabacloud.com/help/en/alinux/support/solutions-to-memory-fragmentation-in-linux-operating-systems

Описание `drop_caches`: https://www.kernel.org/doc/Documentation/sysctl/vm.txt

Длительность удержания грязных страниц

- `vm.dirty_expire_centisecs` - сколько буфер может быть грязным, прежде чем будет помечен для записи
- `vm.dirty_writeback_centisecs` - период ожидания между записями на диск
- `net.ipv4.tcp_timestamps` может уменьшить периодические задержки из-за генерации временных меток

Пример настроек:

```
vm.dirty_expire_centisecs = 500
vm.dirty_writeback_centisecs = 250
vm.swappiness = 10
vm.dirty_ratio = 10
vm.dirty_background_ratio = 3
net.ipv4.tcp_timestamps=0
```


Длительность удержания грязных страниц в кэше

Длительность удержания страницы с момента изменения определяется параметром: `vm.dirty_expire_centisecs` - сколько буфер может быть грязным, прежде чем будет помечен для записи. **Значение по умолчанию 3000 (30 секунд)**. Значение можно уменьшить до 500 (5 секунд).

Также можно установить параметры:

`vm.dirty_writeback_centisecs` - период ожидания между записями на диск, по умолчанию 500 (5 секунд), можно уменьшить до 250 (2,5 секунды)

`vm.swappiness = 10`

`vm.dirty_ratio = 10`

`vm.dirty_background_ratio = 3`

`net.ipv4.tcp_timestamps=0`

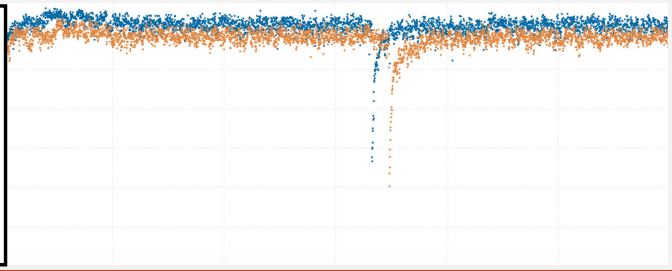
Последний параметр полезен и может уменьшить периодические задержки из-за генерации временных меток. Параметр добавляет 12 байт в заголовок каждого пакета TCP с меткой времени. Параметр описан в RFC1323. Может использоваться в реализациях алгоритма BBR, основанном на измерении сетевой задержки. Также используется в опции `tcp_tw_reuse`, позволяющей повторное использование TIME-WAIT сокетов в случаях, если это считается безопасным.

https://www.enterprisedb.com/blog/tuning-debian-ubuntu-postgresql

Параметр backend_flush_after

- количество грязных блоков, вытесняемых из буферного кэша каждым серверным процессом, по достижении которого будет послана команда на вытеснение страниц файлов, которые соответствуют этим блокам из страничного кэша
- ограничивает объем грязных страниц в страничном кэше linux и уменьшает вероятность проседания производительности при выполнении вызовов fsync по файлам данных в конце контрольной точки

```
\dconfig *flush*
List of configuration parameters
       Parameter        | Value
------------------------+-------
 backend_flush_after    | 0
 bgwriter_flush_after   | 512kB
 checkpoint_flush_after | 256kB
 wal_writer_flush_after | 1MB
```



Параметр backend_flush_after

Параметром PosgreSQL `backend_flush_after` можно задать число грязных блоков, вытесняемых из буферного кэша каждым серверным процессом, по достижении которого будет послана команда на вытеснение страниц файлов, которые соответствуют этим блокам из страничного кэша. Параметр уменьшает вероятность проседания производительности при выполнении вызовов fsync по файлам в конце контрольной точки. По умолчанию значение ноль (отключено), так как возможно снижение производительности. Это происходит, если число блоков, грязнящихся всеми процессами экземпляра между контрольными точками, больше разделяемого пула, но существенно меньше страничного кэша. Параметр можно установить, если проседание производительности в конце контрольной точки заметно и нежелательно.

В конце контрольной точки процесс checkpointer посылает системные вызовы fsync по файлам, блоки которых передавались в страничный кэш. Это выполняется вызовом функции BufferSync(flag). Только checkpointer может вызвать эту функцию. Флаги при вызове функции CHECKPOINT_FLUSH_ALL, CHECKPOINT_IS_SHUTDOWN, CHECKPOINT_END_OF_RECOVERY, CHECKPOINT_IMMEDIATE убирают задержки в передаче вызовов fsync. Идентификаторы блоков (BufferTag) передаются процессу checkpointer серверными процессами. Для хранения идентификаторов используются структуры PendingWriteback и WritebackContext. Сортировка блоков для определения диапазонов страниц выполняется в функции IssuePendingWritebacks(..). Серверные процессы при вытеснении одного блока из буферного кэша вызывают функцию ScheduleBufferTagForWriteback(..) и в неё вставлена проверка не превысил ли серверный процесс значение `backend_flush_after`.

Диапазон значений от 0 до 256 блоков (2Мб).

```
\dconfig *flush*
List of configuration parameters
       Parameter        | Value
------------------------+-------
 backend_flush_after    | 0
 bgwriter_flush_after   | 512kB
 checkpoint_flush_after | 256kB
 wal_writer_flush_after | 1MB
```

Параметры `checkpoint_flush_after` и `bgwriter_flush_after` имеют ту же цель (уменьшить проседание производительности в конце контрольной точки), только для блоков, посылаемых на запись одноимёнными процессами.

Практика

- Часть 1. Запуск экземпляра с огромными страницами
- Часть 2. Изменение значения oom_score
- Часть 3. Выгрузка длинных строк утилитой `pg_dump`
- Часть 4. Нехватка памяти
- Часть 5. Включение подкачки (swap)
- Часть 6. Страничный кэш

Практика

Основная цель всех практик: закрепить в памяти пройденный материал.

Выполнение описанные в главе команд, помогает лучше запомнить технические подробности, которые изучалось в главе. В практике вы увидите, как выглядит срабатывание процесса oom kill. Наблюдение за падением экземпляра вызывает эмоции у ответственных администраторов, а эмоции помогают запомнить изученный материал. Пример использования параметра конфигурации СУБД Tantor `enable_large_allocations` помогает запомнить, где этот параметр может пригодиться.

В практике вы запустите экземпляр с Huge Pages и посмотрите какие команды позволяют убедиться в том, что они используются экземпляром.

Вы вызовете срабатывание OOM kill. Практическое задание позволит вам разобраться как учитывается и отображается память, если процессы используют разделяемую память.

Вы научитесь вызывать OOM killer простой командой:

```
select repeat('a', 1000000000) from generate_series(1, 1000);
```

Вы посмотрите как диагностировать фрагментацию памяти linux и как её дефрагментировать.

3

3-1

Процессоры

Simultaneous Multi-Threading (SMT) и Hyper-Threading (HT)

- Hyper-Threading - название технологии Intel
- Simultaneous Multi-Threading - название технологии AMD
- одно физическое ядро процессора определяется linux как два виртуальных (логических) ядра
- реализует концепцию одновременной многопоточности
- проверка сколько имеется потоков на ядро и имеется ли поддержка SMT и активен ли SMT:

```
root@tantor:~# lscpu | grep Thread
Thread(s) per core: 1
root@tantor:~# cat /sys/devices/system/cpu/smt/active
0
root@tantor:~# cat /sys/devices/system/cpu/smt/control
notsupported
```


Simultaneous Multi-Threading (SMT) и Hyper-Threading (HT)

Одновременная многопоточность (Simultaneous Multi-Threading, SMT) название технологии. Технология реализована в процессорах AMD. У процессоров Intel технология называется Hyper-Threading (HT). Одно физическое ядро процессора определяется linux как два "логических" (как антоним слову "физическое") ядра. Физическое ядро может хранить состояние двух потоков выполнения (threads), содержит по одному набору регистров и по одному контроллеру прерываний (APIC) на каждое логическое ядро. Остальные ресурсы физического ядра общие для логических ядер. Для операционной системы это выглядит как наличие двух логических ядер.

При выполнении потока команд логического ядра, физическое ядро приостановит выполнение и начнет выполнять команды другого ядра, если:

произошёл промах при обращении к кэшу процессора;

выполнено неверное предсказание ветвления;

ожидается результат предыдущей инструкции.

Пока логическое ядро ожидает получения данных из памяти в кэш процессора, вычислительные ресурсы физического ядра будут использоваться потоком команд второго логического ядра.

Вычислительные мощности процессора обычно не является узким местом в СУБД. Использование многопоточности может привести к большой частоте переключений контекста выполнения и перезагрузке кэшей. При этом процессор использует не только код экземпляра, но и код ядра linux. Будет ли многопоточность улучшать общую производительность экземпляра лучше проверить тестированием.

Включение и выключение многопоточости выполняется в BIOS.

Пример результатов теста `pgbench` с масштабом (`-s`) 300:

clients	tps (HT)	tps (без HT)
4	517	520
8	1013	999
16	1938	1913
32	3574	3560
64	5873	5412
128	8351	7450
256	9426	7840
512	9357	7288

https://www.postgresql.org/message-id/53FD5D6C.40105%40catalyst.net.nz

https://elibsystem.ru/node/490

Привязка процесса к процессору (CPU affinity)

- увеличивает вероятность попадания данных процессов в кэши процессоров (TLB и другие)
- можно не рассматривать, если:
  - процессор один
  - нагрузка на ядра процессоров невысокая (меньше 80%)
- пример привязки работающего процесса:

```
root@tantor:~# ps -ef | grep check
postgres  2181  2179  0      00:00:01 postgres: checkpointer
root@tantor:~# taskset -p 1 2181
pid 2181's current affinity mask: f
pid 2181's new affinity mask: 1
```


Привязка процесса к процессору (CPU affinity)

Привязка процесса к процессору - указание планировщику операционной системы ограничить выполнение процесса одним или несколькими процессорами. Смысл в том, чтобы увеличить вероятность попадания данных процессов в различные кэши железа, например, TLB. Привязанный процесс не будет нагружать своими данными в кэши других процессоров.

Процесс сам может программно управлять привязкой. PostgreSQL полагается на операционную систему в доступе к ресурсам железа и разработчики PostgreSQL не стали реализовывать привязку.

Для привязки процессов к процессору можно использовать утилиту **taskset**.

Если процессор один, привязка не дает преимуществ, так как кэши процессора общие для всех ядер одного процессора. Если процессоров и экземпляров несколько, то привязка процессов разных экземпляров к разным процессорам может увеличить производительность, если нагрузка на ядра процессоров высокая (больше 80-90%). Если нагрузки ядра процессоров нет, то ядра не являются узким местом и процессы редко мигрируют между ядрами.

Пример просмотра на каком ядре (psr) работает процесс:

```
root@tantor:~# ps -eLo psr,pid,cmd --headers | grep checkpointer
  3    2181 postgres: 15/main: checkpointer
  1  126674 postgres: checkpointer
```

Пример привязки работающего процесса:

```
root@tantor:~# taskset -p -c 1 2181
pid 2181's current affinity mask: f
pid 2181's new affinity mask: 1
```

Пример вывода текущей привязки работающего процесса:

```
root@tantor:~# taskset -p 2181
pid 2181's current affinity mask: 1
```

https://www.postgresql.org/message-id/4B0ADE3F.2080703%40meteorsolutions.com

Просмотр списка процессов утилитой ps

- утилита командной строки ps выдаёт список работающих процессов
- Используя пайпы можно форматировать вывод утилиты
- Пример первых 4 процессов в выводе, включая первую строку заголовка, описывающего значение в столбцах:

```
ps -A -o pid,psr,cmd | head -4
    PID PSR CMD
      1   2 /sbin/init splash
      2   1 [kthreadd]
      3   0 [pool_workqueue_release]
```

- пример вывода процессов с именем postgres и их сортировки по столбцу pss в убывающем порядке:

```
ps -C postgres -o pcpu,vsz,rss,pss,rops,wops,cmd --sort -pss | head -4
%CPU    VSZ   RSS   PSS  ROPS  WOPS CMD
 0.0 231628 18416 14263     5  3069 postgres: checkpointer
 0.0 233712 16172  9491    87    26 postgres: postgres postgres [local] idle
 0.0 231516  6768  4301    34    98 postgres: walwriter
```


Просмотр списка процессов утилитой ps

Утилита командной строки ps выдаёт список работающих процессов и/или потоков. Утилита выдаёт статический отчёт и после его выдачи завершает работу, в отличие от интерактивных утилит top, htop, atop. Утилита ps собирает данные из директории /proc, в которой находятся сведения обо всех процессах и представляет их в удобной форме.

Используя пайпы можно форматировать вывод утилиты. Пример первых 4 процессов в выдаче, включая первую строку заголовка, описывающего значение в столбцах:

```
ps -A -o pid,psr,cmd | head -4
    PID PSR CMD
      1   2 /sbin/init splash
      2   1 [kthreadd]
      3   0 [pool_workqueue_release]
```

Пример вывода списка процессов, в названии которых есть буквосочетание postgres и ограничение вывода первыми двумя строками:

```
ps -A -o pid,psr,cmd | grep postgres | head -2
   1458   3 /usr/lib/postgresql/15/bin/postgres -D /var/lib/postgresql/15/main
             -c config_file=/etc/postgresql/15/main/postgresql.conf
   1696   3 postgres: 15/main: logger
```

Пример вывода последних 8 процессов, в имени команды (параметр cmd) есть буквосочетание postgres:

```
ps -o pid,psr,cmd -C postgres | tail -8
  20406   3 postgres: checkpointer
  20407   1 postgres: background writer
  20409   3 postgres: walwriter
  20410   2 postgres: autovacuum launcher
  20411   1 postgres: autoprewarm leader
  20412   2 postgres: pg_wait_sampling collector
  20413   2 postgres: logical replication launcher
  65139   3 postgres: postgres postgres [local] idle
```

Параметр -C задаёт имя команды, которой запущен процесс.

Пример использования параметра `-o` для перечисления столбцов, которые хочется видеть в выводе утилиты:

```
ps -e -o user,pcpu,vsz,rss,pss,cls,nlwp,psr,rops,wops,ppid,pid,tid,s,cmd --sort -wops
```

Также это пример вывода с сортировкой по столбцу `wops` в убывающем порядке

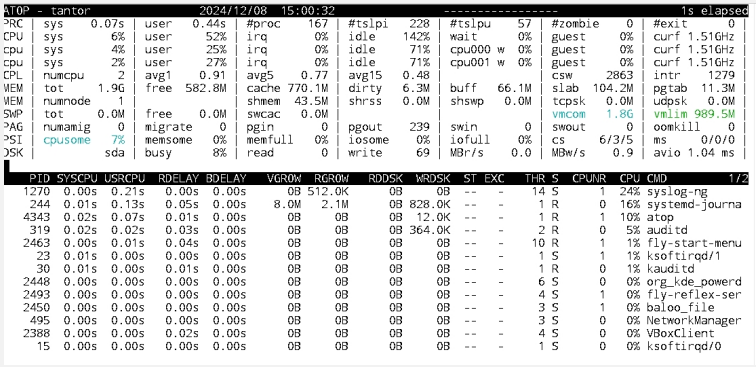
Описание столбцов:
USER имя пользователя, с правами которого запущен процесс
%CPU - загрузка ядра процессора в %
VSZ - виртуальная память в кБ
RSS - резидентный (находящийся в физической памяти, а не вытесненный в swap) размер памяти в кБ
PSS - пропорциональный размер памяти в кБ
CLS - политика планировщика процессов
NLWP число потоков процесса
PSR - номер ядра процессора (cpu), на котором выполняется процесс или поток
ROPS - число операций ввода/вывода
WOPS - число операций ввода/вывода
RBYTES - число прочитанных байт
WBYTES - число записанных байт
PPID - PID родительского процесса
PID - PID самого процесса
TID - PID потока (LWP)
S - статус процесса
CMD - команда, которой запущен процесс с параметрами

Приведены не все названия столбцов, которые может выводить утилита ps.

Запись и просмотр метрик утилитой atop

- для мониторинга процессов и текущей нагрузки удобно использовать утилиту командной строки top
- преимущество утилиты top в том, что она обычно установлена по умолчанию
- утилита atop позволяет записать в бинарный файл показатели работы операционной системы и, затем, визуализировать собранные показатели:

```
root@tantor:~# apt install atop -y
root@tantor:~# dpkg -S /usr/bin/atop
atop: /usr/bin/atop
root@tantor:~# atop -w /atop.record 1 15
root@tantor:~# atop -r /atop.record
```



Запись и просмотр метрик утилитой atop

Для мониторинга процессов и текущей нагрузки удобно использовать утилиту командной строки top. Преимущество утилиты top в том, что она обычно установлена по умолчанию. Утилита top входит в пакет procps:

```
root@tantor:~# dpkg -S /usr/bin/top
procps: /usr/bin/top
```

На хостах в промышленной эксплуатации установка дополнительного программного обеспечения обычно затруднительна. Вероятность установки простых и стандартных утилит более высока. Утилита atop имеется в расширенном репозитории Astralinux.

Преимущество утилиты atop в том, что она позволяет записать в бинарный файл показатели работы операционной системы и, затем, визуализировать собранные показатели. Визуализировать собранные метрики можно на другом хосте.

Для записи используется команда:

atop -w /путь_к_файлу интервал длительность_записи

Интервал (по умолчанию 10 секунд) и длительность записи (по умолчанию бесконечно) указываются в секундах.

Для проигрывания используется команда:

atop -r /путь_к_файлу

При проигрывании клавиша клавиатуры '**t**' переходит на показ следующего по времени интервала, клавиша в верхнем регистре '**T**' для возврата назад, к предыдущему интервалу. Клавиша '**b**' для перехода к времени, которое будет предложено ввести в формате [YYYYMMDD]hhmm. Клавиша '**r**' для возврата к началу файла.

Второе преимущество утилиты atop в том, что метрики в бинарных файлах можно визуализировать в Grafana:

https://github.com/rchakode/atop-graphite-grafana-monitoring

Помимо утилит top и atop может использоваться утилита htop. Преимущества утилиты htop в цветной псевдографике и сохраняемым настройкам.

https://wiki.astralinux.ru/tandocs/instruktsiya-dlya-podgotovki-k-nagruzochnomu-testirovaniyu-302054346.html

Переключения контекста выполнения

- добровольное переключение контекста:
  - процесс не может выполнять свой код, так как ждет выполнения операции ввода-вывода, получения блокировки
- недобровольное переключение контекста:
  - процесс превышает время (timeslice) выделенное ему планировщиком
  - в соответствии с политикой, установленной для процесса планировщик имеет право приостановить выполнение процесса (вытеснить процесс с ядра процессора)
- постоянное большое число недобровольных переключений контекста указывает на слишком большую степень параллелизма, не соответствующую числу ядер процессоров

```
root@tantor:~# grep ctxt /proc/212233/status
voluntary_ctxt_switches:        0
nonvoluntary_ctxt_switches:     62728
```


Переключения контекста выполнения

Ядер процессоров обычно меньше, чем процессов. Добровольное переключение контекста (volunary context switch, VCX) происходит когда процесс обратился системным вызовом (дисковый или сетевой ввод-вывод, ожидание получения блокировки) к коду ядра (sys) и ждёт получения результата, а до получения результата не может выполнять свой код (user). Недобровольное переключение контекста (involuntary context switch, ICX) происходит, когда планировщик приостанавливает выполнение кода процесса потому, что код процесса превышает время (timeslice), которое планировщик выделил процессу или появился процесс с более высоким приоритетом. Число ICX должно быть раз в десять меньше VCX. Обратное указывает на то, что число активных процессов (степень распараллеливания какой-то задачи: пул сессий с базой данных, число workers) слишком большое. VCX и ICX можно посмотреть в статистике конкретного процесса:

```
root@tantor:~# grep ctxt /proc/212233/status
voluntary_ctxt_switches:        0
nonvoluntary_ctxt_switches:     62728
```

или **perf stat -p PID**. По всей системе **perf stat -a sleep 1**.

perf не выдаёт ICX, что делает использование утилиты бесполезным. Остальные **утилиты (pidstat, vmstat) усредняют значения и показатели (cs, nvcswch/s) и будут вводить в заблуждение.** Например, процесс может вытесняться 100 раз в секунду, на ядре может быть 500 переключений в секунду, процесс на этом ядре мог иметь 10 переключений в секунду, а в операционной системе среднее число переключений на процесс 0,2 в секунду. Время определяется политикой планировщика, но не ниже `kernel.sched_rr_timeslice_ms`, по умолчанию 100 миллисекунд, (1/10 секунды, ноль означает значение по умолчанию) значение которого можно менять. По умолчанию используется политика `SCHED_OTHER` - Completely Fair Scheduler (CFS) до версии 6.6 ядра linux, после Earliest Eligible Virtual Deadline First (EEVDF). Время (`timeslice`) плавающее.

Недостаток CFS в том, что под нагрузкой CFS вносит задержку перед началом выполнения задачи (процесса который стал активным) и эта задержка добавляется к response time, снижая отзывчивость процесса. EEVDF устраняет эту проблему.

Версию ядра можно посмотреть командой:

```
root@tantor:~# cat /proc/version
Linux version 6.6.28-1-generic
```

https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/sched_policy_prio/start

Планировщик операционной системы

- По умолчанию CFS используется политика `SCHED_OTHER` с алгоритмом:
  - CFS - до версии ядра 6.6 linux
  - EEVDF - заменяет CFS начиная с версии 6.6 ядра linux, устраняет задержку в начале выполнения задачи
- минимальное время работы процесса до вытеснения: не ниже `kernel.sched_rr_timeslice_ms`, по умолчанию 1/10 секунды, значение можно менять
- политики могут принимать во внимание приоритет процесса или nice или игнорировать их
- политики играют роль когда процессора полностью нагружены
- политику можно менять для каждого процесса:

```
root@tantor:~# chrt -r -p 10 96878
root@tantor:~# chrt -p 96878
pid 96878's current scheduling policy: SCHED_RR
pid 96878's current scheduling priority: 10
```


Планировщик операционной системы

Политика SCHED_OTHER (и её производные SCHED_BATCH, SCHED_IDLE) не являются политиками реального времени. К политикам реального времени (для интерактивных задач, где важна отзывчивость - получать timeslice с какой-то частотой) относятся: SCHED_FIFO - вытесняется только процессами с более высоким приоритетом или политикой SCHED_DEADLINE, приводит к тому что процесс может надолго занять ядро; SCED_RR (Round Robin Scheduler) - время одинаковое и равно `kernel.sched_rr_timeslice_ms`. Процессы с SCHED_DEADLINE могут вытеснять процессы с SCHED_FIFO и SCHED_RR, то есть являются наиболее приоритетными.

Политику планировщика можно менять для процесса. Текущая политика:

```
root@tantor:~# chrt -p 96878
pid 96878's current scheduling policy: SCHED_OTHER
pid 96878's current scheduling priority: 0
```

Установить новую:

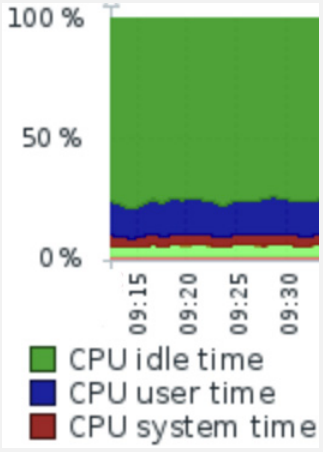
```
root@tantor:~# chrt -r -p 10 96878
root@tantor:~# chrt -p 96878
pid 96878's current scheduling policy: SCHED_RR
pid 96878's current scheduling priority: 10
root@tantor:~# chrt -f -p 10 96878
root@tantor:~# chrt -p 96878
pid 96878's current scheduling policy: SCHED_FIFO
pid 96878's current scheduling priority: 10
```

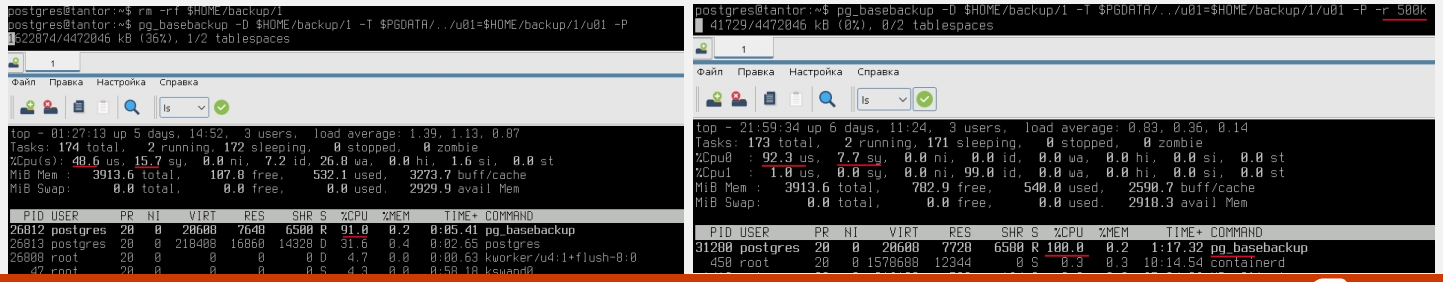
Установить процессу с pid=96878 гарантированные 5 миллисекунд времени выполнения с периодом 15 миллисекунд и дедлайном 10мс:

```
root@tantor:~# chrt -d --sched-runtime 5000000 --sched-deadline 10000000 --sched-period 15000000 -p 0 96878
root@tantor:~# chrt -p 96878
pid 96878's current scheduling policy: SCHED_DEADLINE
pid 96878's current scheduling priority: 0
pid 96878's current runtime/deadline/period parameters: 5000000/10000000/15000000
```

Использование процессорного времени кодом приложений и ядра (пропорция USER/SYS)

- отношение User CPU time/System CPU time должно быть около 60/40
- если User больше это означает неэффективный код приложения
- если System больше - linux испытывает проблемы





Использование процессорного времени кодом приложений и ядра (пропорция USER/SYS)

Метрика USER/SYS CPU Time показывает пропорцию использования процессорного времени кодом приложений и кодом ядра.

Для быстрого мониторинга можно использовать команду top.

Значения:

`us` - User CPU time, процессорное время потраченное кодом процесса приложения, которое не было niced.

`sy` - System CPU time, процессорное время потраченное кодом ядра linux.

`ni` - процессорное время, потраченное кодом процесса приложения, которое niced.

nice свойство, принимаемое во внимание некоторыми типами планировщиков.

При нормально работе приложения и всех процессов в сумме **пропорция (us+ny)/sy должна быть примерно 60/40**. Например: `us=30, sy=20, ni=0`.

Если пропорция сильно (80/20) смещается в сторону **us** это означает, что код приложения неэффективен. Если смещается в сторону **sy** это означает, что операционная система испытывает проблемы.

Пример проблемы (ошибки) в коде приложения (коде утилиты pg_basebackup 16 версии):

```
postgres@tantor:~$ time pg_basebackup -c fast -D $HOME/backup/1 -P -r 10M
51752/51752 kB (100%), 1/1 tablespace
real    0m7.647s
user    0m4.991s
sys     0m0.091s
```

При использовании троттлинга (замедления резервирования) одно из ядер процессора загружается на 100% и `us/sy` = ~100/1. Код реализующий троттлинг имеет ошибку. Утилита `mpstat` не позволяет диагностировать проблему:

```
root@tantor:~# mpstat -n
Linux 6.6.28-1-generic (tantor)  _x86_64_     (4 CPU)
NODE    %usr   %nice    %sys  %iowait  %soft  %idle
 all    2.27    0.03    0.23     0.11   0.29  97.08
```

выдает %idle=97.08% и с виду всё хорошо - нагрузки нет. Если посчитать отношение USER/SYS в утилите **mpstat** (2.27+0.03%)/0.23% что равно 10/1 (реальное ~100/1), можно заметить, что 10/1 далеко от пропорции 60/40.

Примечание: ошибка в коде `pg_basebackup`, приводящая к загрузке ядра процессора на 100% при использовании троттлинга, устранена в 17 версии PostgreSQL.

Источник времени (clock source)

- используются и ядром linux и приложениями для получения меток времени
- во время загрузки linux проверяет доступные источники времени и выбирает один для использования
- наиболее быстрый источник: Time Stamp Counter (TSC)
- ACPI Power Management Timer (ACPI_PM) медленнее в несколько раз
- доступные источники:

```
cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

- используемый источник:

```
cat /sys/devices/system/clocksource/clocksource0/current_clocksource
acpi_pm
```

- сообщения при выборе источника времени можно посмотреть в сообщениях ядра командой: **`dmesg | grep tsc`**

Источник времени (clock source)

Источник времени (clock source) используется ядром linux и программами для получения меток времени. Например, linux может маркировать каждый сетевой пакет меткой времени. Также метки используются, чтобы воспроизведение звука и видео было гладким, а не плавающим. Обращение к источникам времени может быть довольно частым. Важны детальность (частота, fine grain) временных меток, реальная точность (drift, jitter), отсутствие рассинхронизации между ядрами процессора, отсутствие выдачи разным процессам меток в обратном порядке, то есть не соответствующем ходу реального времени. Такое возникает из-за низкоуровневой оптимизации out-of-order execution. Использование обратного порядка для меток времени может привести к плохо диагностируемым ошибкам в работе программ, особенно "кластерных решений" сторонних производителей.

Во время загрузки linux проверяет доступные источники времени и выбирает один для использования. Предпочтителен `Time Stamp Counter (TSC)`. В нём для получения метки используется инструкция процессора RDTSC, по которой в регистре процессора появляется 64-битное целое число кванта (тика, jiffy) времени. Число обнуляется при сбросе питания, засыпания. Посмотреть параметры счетчика времени можно командой `cat /proc/cpuinfo`. Команда может выдать значение `rdtsp` (у процессора есть регистр выдающий время TSC) или `constant_tsc` (процессор нормализует тики, чтобы не зависели от частоты процессора а соответствовали реальному времени). Следующий по предпочтению источник интегрированный в чипсет: `High Precision Event Timer (HPET)`. Технология была создана Intel и Microsoft. Одна из проблем технологии в том, что время доступа к счетчику нивелирует его детальность. HPET запрещен в использовании с процессорами Intel Cofee Lake начиная с версии ядра linux 5.4 по причине неточности. HPET используется для калибровки TSC. Если HPET отключен в BIOS или запрещен в linux, то для калибровки используются другой способ (PMTIMER). Следующий источник: ACPI Power Management Timer (ACPI_PM, он же PMTIMER). Следующие Programmable Interval Timer (PIT) и Real Time Clock (RTC), они менее предпочтительны.

Список источников, которые linux счел возможными к использованию:

```
cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

Используемый источник:

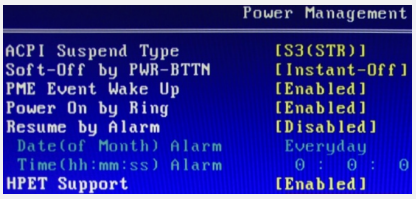
```
cat /sys/devices/system/clocksource/clocksource0/current_clocksource
acpi_pm
```

Сравнение источников времени

- для проверки скорости источника времени можно использовать программу, запрашивающую время много раз

для acpi_pm:
```
time ./clock_timing
real    0m38.889s
user    0m15.760s
sys     0m23.126s
```

Для tsc:
```
time ./clock_timing
real   0m13,967s
user   0m13,938s
sys    0m0,008s
```



- **разница в скорости между acpi_pm и tsc существена: 2,8 раза**
- `real`: время от вызова программы до завершения. real включает в себя user и sys и может быть больше их суммы, если программа вытеснялась планировщиком (involuntary context switches).
- `user`: время выполнения кода программы.
- `sys`: время выполнения кода ядра linux (работа с оборудованием, памятью, файлами, потоками, сетью)

Сравнение источников времени

Проверить скорость источника времени можно создав файл `clock_timing.c`:

```
#include <time.h>
int main()
{
 int rc;
 long i;
 struct timespec ts;
 for(i=0; i<10000000; i++) rc = clock_gettime(CLOCK_MONOTONIC, &ts);
 return 0;
}
```

скомпилировать файл командой:

`gcc clock_timing.c -o clock_timing -lrt`

и проверить какую нагрузку дает считывание времени **`10000000`** раз.

Для `acpi_pm`:

```
time ./clock_timing
real    0m38.889s
user    0m15.760s
sys     0m23.126s
```

Для `tsc`:

```
time ./clock_timing
real 0m13,967s
user 0m13,938s
sys  0m0,008s
```

Разница в 2,8 раз.

Если в BIOS есть пункт включение/отключение HPET, то HPET стоит отключить, так как обычно снижает производительность.

Можно заменить `CLOCK_MONOTONIC` на `CLOCK_MONOTONIC_COARSE` и скорость увеличится в 50 раз.

`real`: время от вызова программы до завершения. `real` включает в себя `user` плюс `sys` и может быть больше их суммы, если программа вытеснялась планировщиком (involuntary context switches).

`user`: время выполнения кода программы.

`sys`: время выполнения кода ядра linux (работа с оборудованием, памятью, файлами, потоками, сетью).

Сравнение источников времени в PostgreSQL

- источник времени активно используется при включении параметров конфигурации `track_wal_io_timing, track_io_timing, track_commit_timestamp`
- пример влияния выбора источника времени на время выполнения команды `explain analyze`:

```
postgres=# \! sudo sh -c 'echo acpi_pm > /sys/devices/system/clocksource/clocksource0/current_clocksource'
postgres=# explain analyze select count(pk) from t;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1791.00..1791.01 rows=1 width=8) (actual time=836.212..836.228 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1541.00 rows=100000 width=8) (actual time=0.019..408.416 rows=100000 loops=>
 Planning Time: 0.056 ms
 Execution Time: 836.334 ms
(4 rows)
postgres=# \! sudo sh -c 'echo tsc > /sys/devices/system/clocksource/clocksource0/current_clocksource'
postgres=# explain analyze select count(pk) from t;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1791.00..1791.01 rows=1 width=8) (actual time=308.180..308.187 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1541.00 rows=100000 width=8) (actual time=0.022..153.991 rows=100000 loops=>
 Planning Time: 0.375 ms
 Execution Time: 308.373 ms
(4 rows)
```


Сравнение источников времени в PostgreSQL

Источник времени может активно использоваться процессами экземпляра. Примером является команда explain analyze. Время активно считывается при включении параметров конфигурации `track_wal_io_timing, track_io_timing, track_commit_timestamp`.

Пример как влияет источник времени на команду explain analyze:

```
postgres=# drop table if exists t;
create table t(pk bigserial, c1 text default 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa');
insert into t select *, 'a' from generate_series(1, 100000);
DROP TABLE
CREATE TABLE
INSERT 0 100000
ALTER TABLE
postgres=# \! sudo sh -c 'echo acpi_pm > /sys/devices/system/clocksource/clocksource0/current_clocksource'
postgres=# explain analyze select count(pk) from t;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1791.00..1791.01 rows=1 width=8) (actual time=836.212..836.228 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1541.00 rows=100000 width=8) (actual time=0.019..408.416 rows=100000
 Planning Time: 0.056 ms
 Execution Time: 836.334 ms
(4 rows)
postgres=# \! sudo sh -c 'echo tsc > /sys/devices/system/clocksource/clocksource0/current_clocksource'
postgres=# explain analyze select count(pk) from t;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1791.00..1791.01 rows=1 width=8) (actual time=308.180..308.187 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1541.00 rows=100000 width=8) (actual time=0.022..153.991 rows=100000
 Planning Time: 0.375 ms
 Execution Time: 308.373 ms
(4 rows)
```

Вместе с PostgreSQL помимо утилиты тестирования методов синхронизации WAL `pg_test_fsync` поставляется утилита командной строки для проверки скорости источника времени **`pg_test_timing`**.

https://docs.tantorlabs.ru/tdb/ru/16_4/se/**pgtesttiming**.html

Замена источника времени

- при загрузке linux может выбираться источник времени acpi_pm, который медленнее tsc
- для изменения источника на tsc нужно добавить в файл `/etc/default/grub` после `quiet splash` параметры:

```
clocksource=tsc nohpet processor.max_cstate=1 intel_idle.max_cstate=0
```

- и выполнить команду `update-grub`
- после перезагрузки проверить что используется `tsc` и `max_cstate=0`:

```
cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
cat /sys/module/intel_idle/parameters/max_cstate
0
```

- проверить скорость и стабильность источника времени можно утилитой командной строки `pg_test_timing`
  - утилита кроме выдаёт распределение задержек в скорости отдачи времени

Замена источника времени

При загрузке linux случайным образом может выбираться источник времени acpi_pm, который медленнее tsc. Это может быть связано с аппаратными ошибками (https://bugzilla.kernel.org/show_bug.cgi?id=203183).

HPET не выбирается, но в последних ядрах используется для калибровки TSC. Если HPET не может использоваться, то для калибровки используется PIT или PMTIMER.

Для изменения источника на tsc нужно добавить в файл `/etc/default/grub` после `quiet splash` параметры:

```
nohpet processor.max_cstate=1 intel_idle.max_cstate=0
```

или

```
clocksource=tsc nohpet processor.max_cstate=1 intel_idle.max_cstate=0
```

и выполнить команду **`update-grub`**, которая обновит файл `/boot/grub/grub.cfg`

Помимо неверно выбранного источника времени из-за работы в виртуальной машине, ошибки может иметь и оборудование. Например, часы TSC могут останавливаться или рассинхронизироваться при переходе процессора в состояния энергосбережения (sleep) или при изменении частоты процессора. Выглядеть это может как подвисание linux при загрузке, при остановке. В таких случаях используют workaround, отключая "энергосбережение" параметрами в /grub.cfg:

```
processor.max_cstate=1 intel_idle.max_cstate=0 idle=poll
```

Проверить, какие CSTATE используются:

```
cat /sys/module/intel_idle/parameters/max_cstate
0
```

Использование `idle=poll` может привести к более высокому тепловыделению на процессорах, если они постоянно не нагружены.

Если в списке:

```
cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

есть источник, то поменять источник часов без перезагрузки можно командой:

```
echo tsc > /sys/devices/system/clocksource/clocksource0/current_clocksource
```

Проверить скорость и стабильность источника времени можно утилитой командной строки `pg_test_timing`

https://linuxreviews.org/Linux_Kernel_Disables_Coffee_Lakes_HPET_On_The_Grounds_That_It_Is_%22Unreliable%22

Также можно проверить утилитой командной строки `pg_test_timing`
Разница будет аналогичная.

acpi_pm:

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 3998.44 ns
Histogram of timing durations:
  < us   % of total      count
     1      0.00147         11
     2      0.00200         15
     4     21.37522     160377
     8     78.16336     586455
    16      0.24870       1866
    32      0.08970        673
    64      0.09663        725
   128      0.01839        138
```

tsc:

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 1388,24 ns
Histogram of timing durations:
  < us   % of total      count
     1      0,00801        173
     2     63,83282    1379438
     4     36,02110     778422
     8      0,00139         30
    16      0,07807       1687
    32      0,04262        921
    64      0,01328        287
   128      0,00199         43
```

Разные сборки linux могут работать с разными источниками времени на том же оборудовании.
Пример сообщений когда tsc не выбирается:

dmesg | grep tsc

```
tsc: Marking TSC unstable due to clocksource watchdog
TSC found unstable after boot, most likely due to broken BIOS. Use 'tsc=unstable'.
clocksource: Checking clocksource tsc synchronization from CPU 1 to CPUs 0,3.
clocksource: Override clocksource tsc is unstable and not HRT
 compatible - cannot switch while in HRT/NOHZ mode
```

Сообщения о нестабильности могут быть связаны с аппаратными ошибками.
Пример сообщений после **clocksource=tsc** без **nohpet**

dmesg | grep tsc

```
tsc: Fast TSC calibration using PIT
tsc: Detected 3600.150 MHz processor
Kernel command line: BOOT_IMAGE=/boot/vmlinuz-6.6.28-1-generic root=UUID=aca1a090-eba2-49ba-a8fc-ba12e9e2bf26 ro
quiet splash clocksource=tsc processor.max_cstate=1 intel_idle.max_cstate=0 parsec.max_ilev=0 parsec.mac=0
pcie_aspm=performance
clocksource: tsc-early: mask: 0xffffffffffffffff max_cycles: 0x33e4e0fd970, max_idle_ns: 440795362981 ns
clocksource: Switched to clocksource tsc-early
tsc: Refined TSC clocksource calibration: 3600.003 MHz
clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x33e4564530a, max_idle_ns: 440795343825 ns
clocksource: Switched to clocksource tsc
clocksource: wd-tsc-wd read-back delay of 427149ns, clock-skew test skipped!
```

Пример сообщений после **nohpet**

```
tsc: Fast TSC calibration failed
tsc: Unable to calibrate against PIT
tsc: using PMTIMER reference calibration
tsc: Detected 3599.954 MHz processor
Kernel command line: BOOT_IMAGE=/boot/vmlinuz-6.6.28-1-generic root=UUID=aca1a090-eba2-49ba-a8fc-ba12e9e2bf26 ro
quiet splash nohpet processor.max_cstate=1 intel_idle.max_cstate=0 parsec.max_ilev=0 parsec.mac=0
pcie_aspm=performance
clocksource: tsc-early: mask: 0xffffffffffffffff max_cycles: 0x33e42840770, max_idle_ns: 440795330420 ns
clocksource: Switched to clocksource tsc-early
tsc: Refined TSC clocksource calibration: 3600.002 MHz
clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x33e4559ce44, max_idle_ns: 440795364889 ns
clocksource: Switched to clocksource tsc
clocksource: wd-tsc-wd read-back delay of 159517ns, clock-skew test skipped!
```

3-2

Сеть

Основные параметры сети

- основные параметры:
  - пропускная способность
  - сетевая задержка
  - наличие и частота сетевых сбоев
- пропускная способность обычно не является узким местом
- экземпляр использует:
  - unix sockets для локальных соединений
  - TCP/IP для соединений через сетевые интерфейсы
- сетевая задержка важна при синхронной фиксации транзакций с подтверждением репликой

Основные параметры сети

Основные параметры сети: пропускная способность (bandwidth), сетевая задержка (network latency), частота сетевых сбоев (в том числе потерь пакетов). Postgresql использует unix sockets для локальных соединений и TCP/IP через сетевые интерфейсы. Обычно сеть не является узким местом для СУБД. Объем данных передаваемых клиентам при обычной работе не очень большой. Узким местом сеть может быть при использовании синхронной фиксации транзакции, где важна сетевая задержка. Типичная сетевая задержка 8-25 микросекунд. Использование RDMA (Remote Direct Memory Access) теоретически позволяет уменьшить задержку до 1-2 микросекунд, а также увеличить реальную пропускную способность за счет уменьшения нагрузки на центральные процессора. Достигается это тем, что контроллер сетевой карты удаленного хоста пишет в адресное пространство памяти произвольного процесса на удалённом хосте, что разгружает центральные процессора от операций, связанных с передачей данных через сетевые интерфейсы (маркетинговый термин zero-copy). Существуют стандарты RDMA over Converged Ethernet (RoCE) в решениях Ethernet и встроенный функционал в решениях InfiniBand. В 2019 году NVIDIA поглотила единственного производителя оборудования Infiniband компанию Mellanox. В Oracle Exadata X8M и более новых используется Ethernet 100 гигабит (например, свитчи Cisco Nexus 9336C-FX2).

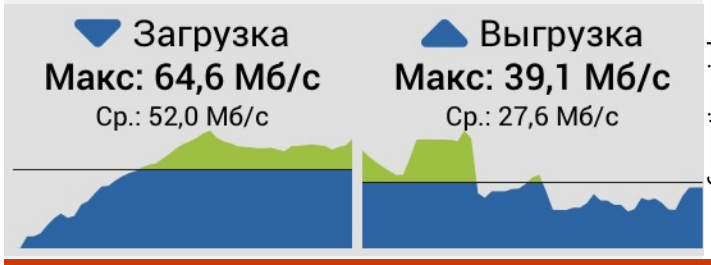
Технология RDMA использовалась в форке Postgres Pro Enterprise 10 и 11 версии (параметр `listen_rdma_addresses`), в более новых версиях форка технология не упоминается и параметр отсутствует. При использовании до 3 синхронных реплик и оборудовании Mellanox эффект был не слишком заметный.

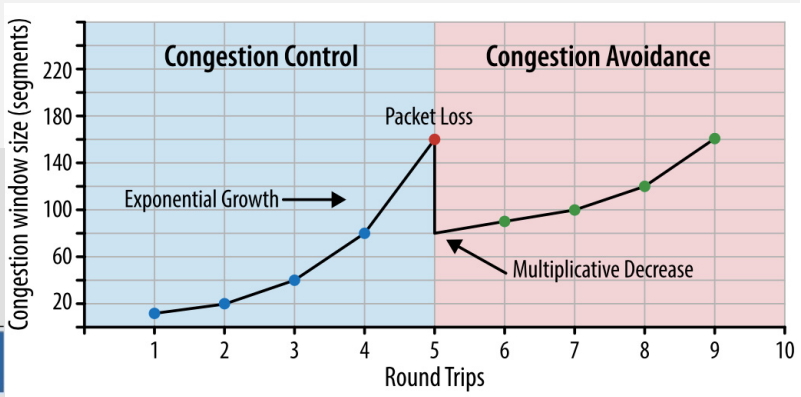
При высокой нагрузке практическая производительность определяется сбалансированностью используемого железа, а не использованием какой-либо технологии. RDMA может использоваться в программно-аппаратном комплексе Tantor xData для резервирования со скоростью 2,5 гигабайт в секунду (10 терабайт в час).

https://ibs.ru/media/superkompyuternoe-mezhsoedinenie-v-mashinakh-baz-dannykh/

Алгоритмы congestion и slow start

- `net.ipv4.tcp_available_congestion_control` определяет алгоритм выбора скорости передачи данных отдельно по каждому сетевому соединению (сокету)
- `net.ipv4.tcp_slow_start_after_idle=0` отключает медленный старт
- без подтверждения передается 3 пакета для размера пакета 1095-2190 байт





Алгоритмы congestion и slow start

В сетях с низкой задержкой (latency) при неиспользовании канала передачи на 100%, полном отсутствии потерь пакетов, передаче небольших объемов данных (3 пакета размером от 1095 до 2190 байт по формуле из RFC5681) параметры сети не влияют на производительность. Если используются балансировщики нагрузки на промежуточных хостах, то сетевая задержка и разница в характеристиках сетевых карт может играть роль. Если имеется один сетевой интерфейс и он используется для резервирования с максимальной скоростью, то соединения серверных процессов с клиентами по тому же сетевому интерфейсу могут испытывать задержки в передаче данных. Почему такое происходит? Представим себе что процесс (серверный или wal_sender) должен передать через сокет TCP поле (данные) размером 500Мб. Он будет передавать не по байту и ждать подтверждения, процесс системным вызовом укажет "передай" и ссылку на область памяти с 500Мб. Дальше допустим у операционной системы есть буфер tcp размером 10Мб и она передает в среду передачи все 10Мб потоком, а у операционной системы процесса-приемника буфер 1Мб и медленный процессор. 9Мб будут не приняты (проигнорированы) так как их некуда принять, а среда передачи при этом была занята всеми 10Мб. По tcp 9Мб будут повторно передаваться когда-то, когда обнаружится, что они не были приняты. То есть возможны ситуации, когда "сбрасываются" данные, так как другая сторона или промежуточный процесс (балансировщик) не могут накопить данные, поступающие из среды передачи. С другой стороны недозагрузка канала тоже плохо, недозагрузка проявляется если относительно высокая сетевая задержка. На уровне среды передачи (канальный и другие уровни) устанавливается MTU и другие параметры, используются jumbo frames. Мы рассматриваем уровень TCP, так как этот протокол использует PostgreSQL.

Значение окна перегрузки сбрасывается после простоя. Это может оказывать влияние на производительность долгоживущих TCP-соединений, которые могут простаивать, из-за бездействия клиента. **Лучше отключить slow start на сервере, чтобы улучшить производительность долгоживущих соединений**:

`net.ipv4.tcp_slow_start_after_idle = 0.`

На уровне tcp за выбор начального объема передаваемых данных отвечает алгоритм

`net.ipv4.tcp_congestion_control = cubic`

`net.core.default_qdisc = pfifo_fast`

https://habr.com/ru/companies/yandex/articles/533530/

https://habr.com/ru/companies/webo/articles/327050/

Алгоритм BBR (Bottleneck Bandwidth и Round Trip Time)

- по умолчанию используется алгоритм CUBIC
- CUBIC чувствителен к потерям пакетов, BBR не чувствителен
- BBR при полной нагрузке на среду передачи потребляет всю доступную полосу пропускания и вытесняет другие сокеты, которые используют cubic и другие алгоритмы
- включение:
  - `sysctl -w net.ipv4.tcp_congestion_control=bbr`
  - `sysctl -w net.core.default_qdisc=fq`
  - `net.ipv4.tcp_timestamps = 1`

Алгоритм BBR (Bottleneck Bandwidth и Round Trip Time)

На уровне tcp за выбор начального объема передаваемых данных отвечает алгоритм
Поддержка BBR доступна начиная с ядра linux 4.9. То что он не выдается не значит, что BBR не поддерживается. По умолчанию в linux используется cubic начиная с ядра 2.6.19 (этот алгоритм также используется и в Windows 10).
BBR (Bottleneck Bandwidth and Round Trip Time, метрики ширины канала и времени передачи пакета туда-обратно) один из алгоритмов, определяющий то, как пакеты уходят в сеть. Разработан Google в 2016 году. Алгоритм конфигурируется двумя связанным параметрами. Для BBR нужно установить `net.core.default_qdisc=fq` , иначе при полной нагрузке сети, что могут делать сокеты с BBR остальные сокеты не смогут передавать данные. Из-за чего это происходит? Алгоритм BBR основан **не на потере пакетов, а на ширине канала и сетевой задержке**. алгоритм не чувствителен к потерям пакетов по любой причине.
Другие алгоритмы выставляют объем передаваемых данных измеряя потери данных, что нежелательно и приводит их повторной передаче. Начинается потеря, уменьшают объемы.
Эти алгоритмы очень чувствителен к потерям пакетов по любой причине.
Сокеты, использующие алгоритмы, которые основаны на потерях пакетов при полном использовании канала не смогут передавать пакеты. Сокеты с BBR не ориентируются на потерю данных. BBR потребляет всю доступную полосу пропускания и вытесняет другие сокеты, которые используют cubic (RFC8312) и другие алгоритмы.
Чем управляется скорость отправки пакетов? Техникой `pacing`, которая реализована в планировщике FQ. Установка планировщика и не зависят от значения `net.core.default_qdisc`
На какую сторону действуют параметры? На сторону, которая является инициатором передачи. Сторона-приемник может использовать любые настройки и не поддерживать BBR.
Поэтому тестовые измерения могут показывать, что в одну сторону график передачи данных нарастает и скорость высокая, а в обратную сторону всплеск, потом провал и устанавливается на небольшой скорости. В таком случае нужно устанавливать значения на клиенте или разгружать каналы передачи от клиента к серверу.
Мониторинг:

```
ss -tin
ss --options --extended --memory --processes --info
```

https://djangocas.dev/blog/huge-improve-network-performance-by-change-tcp-congestion-control-to-bbr/

Параметры сетевых соединений

- `tcp_user_timeout` сколько переданные данные могут оставаться неподтверждёнными, прежде чем будет принято решение о принудительном закрытии TCP-соединения
- `client_connection_check_interval` (в миллисекундах) интервал между проверками в процессе выполнения команды путем опроса состояния сокета, по которому не передаются данные

```
root@tantor:~# sysctl -a | grep keepalive
net.ipv4.tcp_keepalive_intvl = 75 -> 20
net.ipv4.tcp_keepalive_probes = 9 -> 5
net.ipv4.tcp_keepalive_time = 7200 -> 240

net.ipv4.tcp_slow_start_after_idle = 1 -> 0
net.ipv4.tcp_retries2 = 15 -> 3
net.ipv4.tcp_timestamps = 1 -> 0
```


Параметры сетевых соединений

Устойчивость работы сетевых соединений особенно под нагрузкой позволяет избежать уменьшения производительности связанной с повторным выполнением действий, завершившихся неудачно из-за сетевого сбоя.

Значения следующих параметров по умолчанию установлены в ноль, что означает интервал установленный по умолчанию в linux для сокетов. Значения по умолчанию консервативны и их можно уменьшить:

```
root@tantor:~# sysctl -a | grep keepalive
net.ipv4.tcp_keepalive_intvl = 75 -> 20
net.ipv4.tcp_keepalive_probes = 9 -> 5
net.ipv4.tcp_keepalive_time = 7200 -> 240
```

`tcp_user_timeout` интервал (в миллисекундах), в течение которого переданные данные могут оставаться неподтверждёнными, прежде чем будет принято решение о принудительном закрытии TCP-соединения. Значение `tcp_user_timeout` должно быть установлено в интервале от `(tcp_keepalives_idle + tcp_keepalives_interval * (tcp_keepalives_count - 1))` до **`(tcp_keepalives_idle + tcp_keepalives_interval * tcp_keepalives_count)`**. Можно установить по верхней границе.

`tcp_keepalives_idle` (в секундах) период отсутствия трафика, по истечении интервала linux отправляет пакет TCP для сохранения соединения. `tcp_keepalives_interval` (в секундах) интервал для повторных посылок пакета сохранения соединения, если от другой стороны не был получен ответ на первый пакет. `tcp_keepalives_count` число посылок пакета сохранения соединения после которых соединение будет разорвано.

`client_connection_check_interval` (в миллисекундах) интервал между проверками в процессе выполнения команды путем опроса состояния сокета, по которому не передаются данные. По умолчанию проверка отключена. Сокет может быть закрыт другой стороной или кодом ядра операционной системы из-за неответа другой стороны на пакеты keepalive.

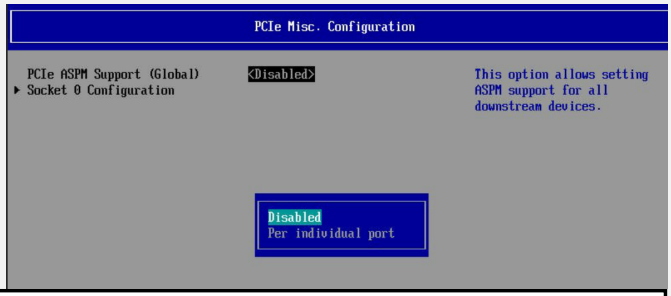
Значения не должны приводить ложному срабатыванию. Также есть параметры `idle_in_transaction_session_timeout`, `idle_session_timeout`, `transaction_timeout`, `wal_receiver_timeout`, `wal_sender_timeout`, но они обычно имеют бо́льшие значения, чем параметры у сокетов.

https://www.postgresql.org/message-id/flat/160741519849.701.13355787096244067178%40wrigleys.postgresql.org

https://blog.cloudflare.com/when-tcp-sockets-refuse-to-die/

Параметры энергосбережения

- в операционной системе не нужно включать
- из-за ошибок в реализации может снижать производительность
- часть настроек в firmware




```
root@student:~# cat /sys/module/pcie_aspm/parameters/policy
[default] performance powersave powersupersave
root@student:~# echo performance > /sys/module/pcie_aspm/parameters/policy
root@student:~# cat /sys/module/pcie_aspm/parameters/policy
default [performance] powersave powersupersave
```


Параметры энергосбережения

В BIOS могут быть настройки энергосбережения. Например, Intel TurboBoost, HyperThread, C-states, DDR Frequency. Стоит проверить значения, так же как параметры виртуализации. На оборудовании для центров обработки данных параметры энергосбережения могут быть включены по умолчанию, так как увеличенное энергопотребление приводит к повышенному нагреву. Центры обработки данных уделяют внимание PM (power management) и RM (resource scheduling). Обычно BIOS позволяет управлять энергосбережением операционной системе. Обычно в операционной системе выделяется несколько планов (политик), среди них максимальная производительность, максимальное энергосбережение. При выборе чего-то отличного от "performance" (максимальная производительность) операционная система обращается к firmware, которое снижает энергопотребление. Проблема в том, что функционал энергосбережения имеет ошибки в реализации. Например, периферийное устройство может некорректно выйти из режима энергосбережения. Например, высокоскоростные (100GbE) сетевые карты могут начать работать со скоростью в два раза меньше. Энергосбережение периферии не дает большого нагрева и смысл в энергосбережении для сервера с СУБД отсутствует. Производители могут объяснять ошибки реализации как то, что при использовании Active-State Power Management (ASPM) добавляется задержка при выводе устройства из пониженного уровня питания (https://edc.intel.com/content/www/us/en/design/products/ethernet/config-guide-e810-dpdk/active-state-power-management/).

Производители оборудования могут включать функционал, который увеличивает производительность, но так как приводит к перегреву не может работать постоянно и должен отключаться через какое-то время (TurboBoost). Такой функционал вряд ли стоит включать, так как неравномерность скорости работы оборудования может нарушить логику работы операционной системы и приложений, если в них имеется самонастройка, вряд ли она учитывает внезапные изменения производительности оборудования.

Оптимальные настройки, гарантирующие декларируемые показатели производительности, встречаются в программно-аппаратных комплексах. Например, произвольное изменение содержимого ячеек памяти при частом чтении соседних участков памяти ("rowhammer") может встречаться у чипов одних производителей и отсутствовать у других независимо от наличия контроля четности ECC (не распознаёт изменение трех битов), что косвенно (выбирают комплектующие по которым мало гарантийных обращений) учитывается производителями аппаратных комплексов.

Практика

- Часть 1. Стандартный тест pgbench
- Часть 2. Привязка процессов к ядру процессора
- Часть 3. Переключения контекстов выполнения
- Часть 4. Мониторинг нагрузки на процессор
- Часть 5. Сбор статистик в файл и его просмотр утилитой `atop`
- Часть 6. Источник времени linux
- Часть 7. Сетевые соединения
- Часть 8. Замена политики планирования и проверка работы планировщика

Практика

В практике закрепляется то, что изучалось в главе.
Вы научитесь
привязывать процессы к ядрам процессоров;
диагностировать переключения контекстов и посмотрите как утилиты linux perf, pidstat выдают бесполезные данные, которые вводят в заблуждение и какие команды выдают реальные данные;
Вы посмотрите как изменение источника времени в несколько раз ускоряет или замедляет выполнение команды EXPLAIN ANALYZE
Вы поменяете параметры планировщика linux и увидите, что утилита perf не выдаёт число принудительных переключений контекстов. Также вы посмотрите как получить реальные данные о переключениях контекстов.

4

Система хранения

Дисковая подсистема

- синонимы: ввод-вывод (I/O), система хранения данных (СХД)
- наиболее нагруженный ресурс СУБД из основных ресурсов:
  - диск
  - процессор
  - память
  - сеть
- экземпляр может использовать прямой ввод/вывод, в 16 версии не обеспечивает лучшей отказоустойчивости и производительности
- настройка дисковой подсистемы включает:
  - выбор систем хранения (например, SSD или HDD)
  - выбор i/o scheduler
- настройка параметров кластера и табличных пространств
- создание точек монтирования, выбор файловых систем, менеджеров томов, параметров монтирования
- мониторинг ввода/вывода и смежные настройки

Дисковая подсистема

Дисковая подсистема (ввод-вывод, система хранения, storage) наиболее нагруженный ресурс СУБД из основных ресурсов:
диск
процессор
память
сеть.

Кластер баз данных хранится в файлах директории PGDATA файловой системы и других директориях, на которые есть символические ссылки в директории `PGDATA/pg_tblspc`. При работе с файлами по умолчанию используются обычные системные вызовы и работа идёт через страничный кэш.

Экземпляр может использовать прямой ввод/вывод (direct i/o, O_DIRECT). Теоретически такой доступ имеет преимущества, но исторически работа с вводом/выводом была оптимизирована для обычного доступа к файлам, а не для прямого. Прямой доступ в 16 версии не обеспечивает лучшей отказоустойчивости и производительности. Открыв файл с O_DIRECT операционная система не кэширует его содержимое, но и не убеждается в записи, то есть не посылает ATA команду flush контроллеру диска и запись идет в кэш диска. Процессам для гарантии записи все равно понадобится выполнять fsync, либо запрещать в firmware диска кэширование записи. Включается прямой доступ параметром `debug_io_direct`. По умолчанию пустая строка (прямой ввода/вывода отключен). Значением могут быть слова. разделенные запятыми: `data` (прямой доступ к файлам данных), `wal` (прямой доступ к файлам WAL), `wal_init` (прямой доступ при создании файлов WAL). Настройка дисковой подсистемы включает:
1.выбор систем хранения (например, SSD или HDD)
2.выбор i/o scheduler
3.настройка параметров кластера и табличных пространств
4.создание точек монтирования для поддиректорий PGDATA, выбор файловых систем, менеджеров томов для точек монтирования, выбор параметров монтирования
5.мониторинг нагрузки на устройства ввода/вывода, подстройка параметров и перераспределения точек монтирования по устройствам
6. смежные настройки: настройка параметров кластера для обеспечения отказоустойчивой и производительной записи в WAL, мониторинг свободного места и настройка очистки от ненужных файлов для своевременного освобождения свободного места.

HDD, SSD, NVMe

- SSD (solid-state drive) - твердотельный (на микросхемах) "диск"
- NVMe, (Non-Volatile Memory express, энергонезависимая память) - интерфейс доступа к SSD, подключенным по шине PCIe
- SATA (Serial Advanced Technology Attachment) транспортный протокол, который определяет взаимодействие между контроллером и устройствами хранения
- SAS (Serial Attached SCSI) - набор команд SCSI по физическому интерфейсу, аналогичному SATA
- AHCI (Advanced Host Controller Interface) стандарт, описывающий операции с контроллерами SATA
  - на каждый порт одна очередь глубиной до 32 команд
  - поддерживается горячая замена устройств
- M.2 (NGFF, Next Generation Form Factor) - общее название форм-фактора и физического интерфейса для SSD, имеет 4 линии PCIe с общей скоростью 4 или 8 гигабайт в секунду
- Скорость SATA - 600 мегабайт в секунду

HDD, SSD, NVMe

HDD (hard disk drive) - жесткий диск, характеризуется задержкой при произвольном доступе к данным, приемлемой последовательной скоростью, небольшой стоимостью единицы хранения. Срок жизни зависит от времени работы.

SSD (solid-state drive, твердотельное устройство, то есть без движущихся частей) - устройство хранения данных на микросхемах (NAND), размещение и считывание данных выполняет чип контроллера. Дороже HDD, одинаково быстрая скорость произвольного и последовательного доступа. Срок жизни зависит от количества записанных данных. Скорость записи зависит от объема хранимых данных.

NVMe, (Non-Volatile Memory express, энергонезависимая память) - интерфейс доступа к SSD, подключенным по шине PCIe.

PCIe (Peripheral Component Interconnect Express) имеет высокую пропускную способность и невысокую задержку. Устройства NVMe могут иметь оперативную память (DRAM) размером порядка 1Гбайт, в которой кэшируют таблицу размещения данных, могут не иметь её или использовать основную память хоста размером около 32Мб (HMB, Host Memory Buffer), используя возможность прямого доступа к памяти (DMA) шины PCIe.

SATA (Serial Advanced Technology Attachment) транспортный протокол (набор команд и последовательность их использования), который определяет взаимодействие между контроллером и устройствами хранения. SATA также называют технические спецификации, которым следуют производители кабелей, разъемов.

SAS (Serial Attached SCSI) - набор команд SCSI по физическому интерфейсу, аналогичному SATA. Преимущества SAS - возможность подключения устройства одновременно по нескольким каналам, длинна кабеля до 10 метров вместо 1 метра.

AHCI (Advanced Host Controller Interface) - стандарт, разработанный Intel, описывающий операции с контроллерами SATA, которые обслуживают устройства хранения, такие как HDD, SSD, DVD. Устройства могут поддерживать NCQ (собственную очередь команд, которая позволяет устройству принимать более одной команды одновременно и самостоятельно определять какая команда из очереди будет выполнена первой. В AHCI на каждый порт (интерфейс с устройством) может иметься одна очередь глубиной до 32 команд (в устройствах SAS до 256). В стандарте AHCI заложен hot-plugging. Это означает, что операционная система, при работе с контроллерами SATA, которые поддерживают физическое подключение/отключение устройств сможет обрабатывать такие события.

Скорость SATA - 600 мегабайт в секунду. У SAS встречается максимум 2400МБ, что в несколько раз ниже скорости M.2 PCIe 4x4 (версия 4 линий 4) NVMe.

Для сравнения скорость передачи USB (универсальная последовательная шина):

3.0 (он же 3.1 Gen1, он же 3.2 Gen1) - 500 МБ/с

3.1 (он же 3.1 Gen2, он же 3.2 Gen2) - 1200МБ/с

3.2 (он же 3.2 Gen 2x2) - 2400МБ/с

USB4 - 40Гбит/с или 20Гбит/с, обратно совместим с USB 3.2 и 2.0.

Интерфейсы SATA могут использоваться потому, что оборудование не очень дорогое.

Контроллеры SAS могут поддерживать подключение устройств с интерфейсом SATA, при прямом подключении с использованием протокола SATA, через платы расширения с использованием туннелирования через протокол STP (SATA Tunneled Protocol).

M.2 (NGFF, Next Generation Form Factor) - общее название форм-фактора и физического интерфейса для SSD, адаптеров WiFi, Bluetooth модемов 4G, других устройств. Называется так потому, что считается второй версией устаревшего разъема Mini PCIe, тоже имевшего до 4 линий PCIe и одну SATA. Устройства, имеющие разъем M.2 могут использовать любую из шин разъема, поэтому знания о том, что устройство имеет разъем M.2 недостаточно. Например, 4G модем может использовать сколько-то линий PCIe или одну USB.

Устройства с разъемом M.2 могут иметь вырезы ("Key"). Обычно используются:

B.Key PCIe x2 (две линии), SATA, USB, PMC, IUM, SSIC, I2C

M.Key PCIe x4 или SATA

B/M Key PCIe x2 или SATA.

Пропускная способность одной линии PCIe:

версии 3.x в каждую сторону - чуть меньше 1 Гигабайт в секунду

PCIe версии 4 - 2ГБ/с. **При использовании 4 линий ("PCIe 4x4") - чуть меньше 8ГБ/с**

SSD форм-фактора M.2 с интерфейсом SATA имеют скорость не выше 600 МБ/с.

Производители плат для установки SSD делают разъем M.2 с ключом M и двумя интерфейсами на выбор PCIe или SATA. Бывают исключения, когда разъем M.2 на плате подключен только к шине PCIe или только к SATA контроллеру.

Технология Intel Optane (альтернатива NAND) развития не получила. Она была примечательна тем, что помимо интерфейса PCIe могла использовать разъем DDR4.

Спецификация NVMe, которая полезна для интерпретации сокращений и метрик, связанных с NVMe:

https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-Revision-2.1-2024.08.05-Ratified.pdf

Ссылка на страницу со спецификациями:

https://nvmexpress.org/specifications/

Блочные устройства

- вид специальных файлов в linux, обеспечивающих интерфейс доступа к устройству (или обычному файлу)
- чтение-запись идет блоками равного размера
- располагаются в директории `/dev` смонтированной на виртуальной файловой системе `devtmpfs`
- в `/dev` находятся файлы только тех устройств, которые в настоящий момент доступны (подключены)
- действия при обнаружении устройств задаются в файлах директории `/lib/udev/rules.d`
- список блочных устройств:

```
root@tantor:~# ls -l /dev | grep br
brw-rw----  1 root    disk      7,   0 дата время loop0
brw-rw----  1 root    disk      8,   0 дата время sda
brw-rw----  1 root    disk      8,   1 дата время sda1
brw-rw----+ 1 root    cdrom    11,   0 дата время sr0
```


Блочные устройства

Блочное устройство (block device) - вид специальных файлов в linux, обеспечивающих интерфейс доступа к устройству (или обычному файлу). Называются блочными потому, что чтение-запись идет блоками равного размера.

Доступ произвольный, указывается порядковый номер блока. Для доступа может использовать прямой ввод/вывод.

Файл блочного устройства может ссылаться на диск, раздел диска, том.

Файловые системы при доступе к носителю используют блочные устройства.

Блочные устройства располагаются в директории `/dev` смонтированной на виртуальной файловой системе `devtmpfs`:

```
root@tantor:~# mount | grep /dev
udev on /dev type devtmpfs (rw,nosuid,relatime,size=968888k,
                            nr_inodes=242222,mode=755,inode64)
hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime,pagesize=2M)
```

В /dev находятся файлы только тех устройств, которые в настоящий момент доступны (подключены). Если устройство отключается, то файл удаляется из `/dev`.

Занимается этим процесс udev, он принимает события, которые генерируются при инициализации или удалении устройства. Задаваемые в файлах директории `/lib/udev/rules.d` правила сверяются со свойствами события и совпавшие правила выполняются и могут создать файлы устройств, запустить программы и командные файлы для инициализации и конфигурирования устройств. Например, смонтировать файловые системы при подключении носителя.

Список блочных устройств:

```
root@tantor:~# ls -l /dev | grep br
brw-rw----  1 root    disk      7,   0 дата время loop0
brw-rw----  1 root    disk      8,   0 дата время sda
brw-rw----  1 root    disk      8,   1 дата время sda1
brw-rw----+ 1 root    cdrom    11,   0 дата время sr0
```

Первая буква b обозначает block device. Вместо размера файла выдаются два числа: тип и порядковый номер (или режим работы) устройства.

Также есть директория `/sys/dev/block` с символическими ссылками на устройства.

Планировщик ввода/вывода (I/O Scheduler)

- при использовании высокоскоростных устройств NVMe нет смысла использовать какой-либо планировщик, то есть стоит установить планировщик **`none`**
- контроллеры устройств памяти NVMe на шинах с минимальной задержкой PCIe справляются с потоком параллельных запросов и при этом не нагружают центральные процессоры и кэши процессоров, в отличие от кода планировщиков
- цель планировщиков: выстроить запросы от большого количества процессов в очередь, передавая контроллеру устройства хранения запросы один за другим, чтобы не перегружать контроллер
- современные планировщики: **`none mq-deadline kyber bfq`**

Планировщик ввода/вывода (I/O Scheduler)

Ранее рассматривался планировщик операционной системы, он планирует использование ресурсов центральных процессоров и не связан с планировщиком ввода /вывода.

Планировщик ввода/вывода определяет, в каком порядке операции блочного ввода-вывода передаются на устройства хранения. Цель планировщика i/o это оптимизация обработки запросов к диску для повышения производительности и пропускной способности ввода-вывода. Когда в системе запущено большое количество процессов, посылающих запросы к операционной системе на выполнение операции ввод-вывод, образуется очередь из таких запросов.

При использовании высокоскоростных устройств (NVMe) нет смысла использовать какой-либо планировщик, то есть можно установить имя none. Контроллеры устройств памяти NVMe на шинах с минимальной задержкой (PCIe) справляются с потоком параллельных запросов и при этом не нагружают центральные процессоры и кэши процессоров, в отличие от кода планировщиков (особенно у которых много очередей, mq). С высокоскоростными устройствами можно использовать планировщик kyber, если нужно снижение задержки на чтение в 2-8 раз за счет уменьшения пропускной способности (20-30%) и увеличения задержки на запись (~50%).

Планировщики, начиная с версий ядра linux 4.12: **`none mq-deadline kyber bfq`**

Логика deadline означает, что критерием является длительность нахождения запроса в очереди. Гарантируется, что каждый запрос будет обслужен планировщиком. По умолчанию приоритет отдается запросам на чтение.

После появления NVMe SSD стало ясно, что код планировщиков только снижает производительность и от элегантных программных алгоритмов придется отказаться (использовать none), был создан планировщик blk-mq.

mq-deadline это реализация deadline с использованием blk-mq.

kyber использует две очереди для запросов на запись и на чтение, kyber отдает приоритет запросам на чтение, перед запросами на запись. Алгоритм измеряет время завершения каждого запроса и корректирует фактический размер очереди для достижения установленных в настройках задержек. Kyber может использоваться с быстрыми устройствами и нацелен на снижение задержки на чтение, с приоритетом для синхронных запросов.

Рекомендация для виртуальных машин:
https://access.redhat.com/solutions/5427

Изменение I/O Scheduler

- тип планировщика можно установить отдельно для разных устройств и их типов SSD или HDD
- посмотреть планировщик для блочного устройства:

```
root@tantor:~# cat /sys/block/sda/queue/scheduler
none [mq-deadline]
```

- поменять планировщик без перезагрузки на другой:

```
root@tantor:~# echo none > /sys/block/sda/queue/scheduler
```

- для создания постоянного правила создать или отредактировать файл `/etc/udev/rules.d/70-schedulerset.rules`

```
ACTION=="add|change", KERNEL=="sd[a-z]", TEST!="queue/rotational",
                                          ATTR{queue/scheduler}="none"
```


Изменение I/O Scheduler

Тип i/o scheduler используемого при работе с блочного устройством можно посмотреть:

```
root@tantor:~# cat /sys/dev/block/8:0/queue/scheduler
```

или

```
root@tantor:~# cat /sys/block/sda/queue/scheduler
[mq-deadline] none
```

Используется `mq-deadline`.

Изменение планировщика без перезагрузки:

```
root@tantor:~# echo kyber > /sys/block/sda/queue/scheduler
root@tantor:~# cat /sys/block/sda/queue/scheduler
mq-deadline [kyber] none
root@tantor:~# echo none > /sys/dev/block/8:0/queue/scheduler
root@tantor:~# cat /sys/dev/block/8:0/queue/scheduler
[none] mq-deadline kyber bfq
```

Для создания постоянного правила можно создать (название файла можно выбрать самостоятельно) или отредактировать существующий файл `/etc/udev/rules.d/70-schedulerset.rules`

и добавить нужные строки вида:

```
ACTION=="add|change", KERNEL=="sd[a-z]", ATTR{queue/rotational}=="0",
ATTR{queue/scheduler}="none"
```

где

`ATTR{queue/scheduler}="none"` название желаемого планировщика для устройств с названием sda, sdb .. sdz.

`ATTR{queue/rotational}=="0"` если драйвер сообщит, что устройство имеет одинаковую скорость произвольного и последовательного чтения.

`ATTR{queue/rotational}=="1"` если произвольный доступ медленнее.

Если драйвер не устанавливает атрибут в 0 или 1, можно указать:

`TEST!="queue/rotational"`

В AHCI одна очередь на порт глубиной 32 команды (i/o request, RQ-SIZE команды `lsblk -td`). В NVMe обычно 256, но может быть и до 64000.

Физический сектор диска

- минимальная единица хранения, которую физическое устройство хранения данных может записывать атомарно
- обычно имеет размер 512Кбайт или 4Кбайт
- для NVMe, linux использует значение параметра Atomic Write Unit Power Fail (AWUPF), если он предоставляется оборудованием

Физический сектор диска

Физический сектор - это минимальная единица хранения, которую физическое устройство хранения может записать "атомарно", что означает либо полностью записать, либо не выполнить запись. Обычно физический сектор равен 512Кбайт или 4Кбайт.

Для NVMe, linux использует для физического сектора значение параметра оборудования **Atomic Write Unit Power Fail** (AWUPF), если оборудование его предоставляет драйверу.

Логический сектор используется для чтения и записи на устройство хранения данных на программном уровне операционной системой. Размер логического сектора может отличаться от размера физического сектора. Посмотреть размеры можно командами:

```
root@tantor:~# fdisk -l | grep size
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
root@tantor:~/flashbench-dev# lsblk -td
NAME  ALIGNMENT MIN-IO OPT-IO PHY-SEC LOG-SEC ROTA SCHED       RQ-SIZE  RA
sda           0    512      0     512     512    0 none             32 128
sr0           0    512      0     512     512    1 mq-deadline       2 128
nvme0n1       0    512      0     512     512    0 mq-deadline     256 128
```

Некоторые диски NVMe и SATA поддерживают изменение сообщаемого размера сектора с помощью стандартных команд NVMe (Format NVM из спецификации набора команд NVM) или ATA4 (SET SECTOR CONFIGURATION EXT). Для жестких дисков это изменяет размер логического сектора, чтобы он соответствовал размеру физического сектора для оптимальной производительности. Для NVMe изменяются значения как логического, так и физического сектора.

На первом уровне чип NAND состоит из нескольких target, каждый из которых содержит несколько кристаллов (die). Каждый кристалл является независимым блоком хранения, который состоит из нескольких слоев (plane). Несколько слоев совместно используют одну и ту же шину и могут быть объединены в один блок для многслойных параллельных операций. Каждый слой состоит из нескольких erase unit (блок стирания).

Размер erase unit определяет гранулярность очистки (discard/trim) на уровне прошивки (firmware)контроллера (SoC, system-on-chip) SSD . Каждый блок стирания состоит из нескольких страниц. Страница является наименьшей единицей записи, **обычно 16 Килобайт**. Размер блока стирания варьируется в широких пределах, от 1Мб до десятков Мб.

Для повышения производительности в SSD может использоваться DRAM для кэширования как данных, так и для кэширования таблиц отображения Flash Translation Layer (FTL). FTL похожа на журналируемую файловую систему. FTL записывает данные, добавляя записи в файл журнала. Используется таблица сопоставления адресов Logical Block Address -Physical Block Address. Проблемой является сборка мусора - очистка от ненужных записей.

Для повышения производительности обычно используется список, а не сложные иерархические структуры, но список имеет большой размер ~1/1000 от объема SSD. Для терабайтного SSD это 1Гб. Обычно SSD используют сектора размером 4КБ, что оптимально для linux. Для удешевления (экономии DRAM) некоторые производители используют сектора размером 16Кб.

Кроме производительности важна отказоустойчивость, когда SoC при пропадании питания должен успеть записать накопленные данные. Это достигается наличием **конденсаторов в цепях питания** у SSD для промышленного использования. Сборка мусора, выравнивание износа блоков стирания, избыточность, сохранение данных увеличивают внутренний трафик SSD.

В спецификацию NVMe 1.4 включены показатели, которые может предоставлять производитель: Preferred Write Alignment, Preferred Write Granularity, Deallocate alignment и статистика гранулярности.

При анализе производительности не стоит использовать `sysbench`, так как эта утилита использует тестовые файлы забитые нулями. Для тестирования стоит использовать утилиты fio или flashbench.

Тестирование с `fsync` после каждой записи:

```
root@tantor:~# fio --filename=/dev/nvme0n1 --name=a --blocksize=8k --rw=randrw --iodepth=32 --runtime=10 --rwmixread=90 --fsync=1
READ: bw=27.1MiB/s (28.4MB/s), 27.1MiB/s-27.1MiB/s (28.4MB/s-28.4MB/s), io=271MiB (284MB), run=10001-10001msec
WRITE: bw=3099KiB/s (3173kB/s), 3099KiB/s-3099KiB/s (3173kB/s-3173kB/s), io=30.3MiB (31.7MB), run=10001-10001msec
```

Без `fsync`:

```
root@tantor:~# fio --filename=/dev/nvme0n1 --name=b --blocksize=8k --rw=randrw --iodepth=32 --runtime=10 --rwmixread=90 --fsync=0
READ: bw=30.6MiB/s (32.1MB/s), 30.6MiB/s-30.6MiB/s (32.1MB/s-32.1MB/s), io=306MiB (321MB), run=10001-10001msec
WRITE: bw=3523KiB/s (3607kB/s), 3523KiB/s-3523KiB/s (3607kB/s-3607kB/s), io=34.4MiB (36.1MB), run=10001-10001msec
```

Определение размера блока стирания путем тестирования утилитой https://github.com/bradfa/flashbench:

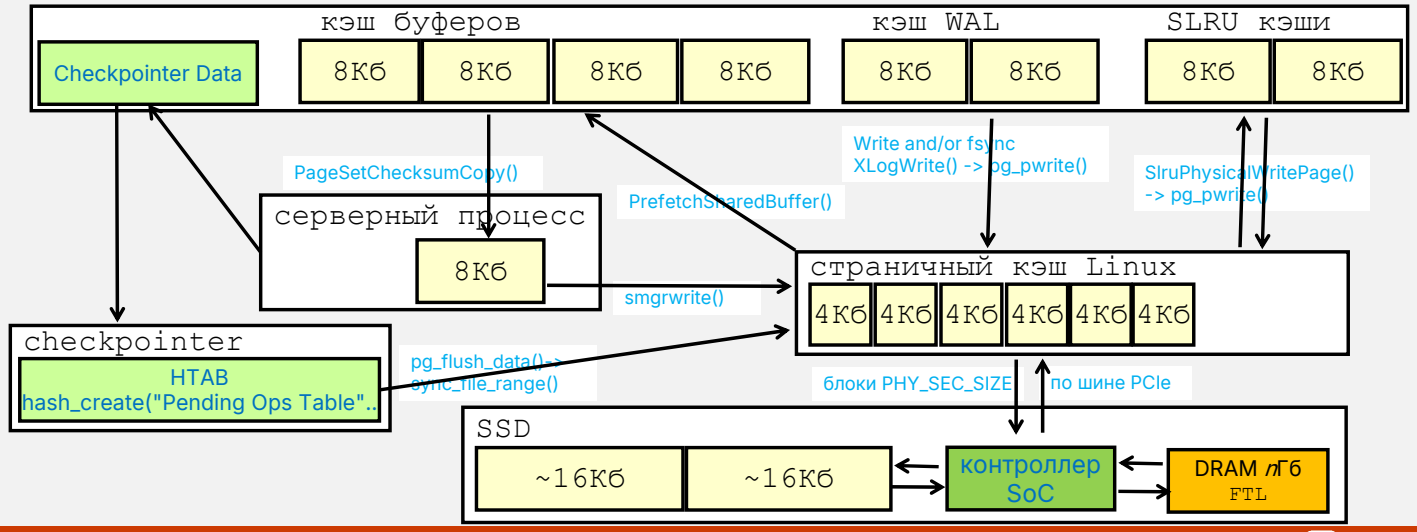
```
root@tantor:~/flashbench-dev# ./flashbench -a /dev/nvme0n1 --blocksize=1024
align 33554432  pre 213µs       on 287µs        post 224µs      diff 68µs
align 16777216  pre 199µs       on 248µs        post 210µs      diff 43.4µs
align 8388608   pre 159µs       on 230µs        post 130µs      diff 85.7µs
align 4194304   pre 228µs       on 259µs        post 257µs      diff 16µs
align 2097152   pre 197µs       on 236µs        post 217µs      diff 29.4µs
align 1048576   pre 171µs       on 208µs        post 210µs      diff 17.4µs
```

Утилита может использоваться для определения оптимального размера страйпа RAID.

Пример проверки записи по fsync при выключении питания: https://habr.com/ru/companies/selectel/articles/521168/ скриптом diskchecker.pl

Взаимодействие процессов экземпляра с диском

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



Взаимодействие процессов экземпляра с диском

При чтении в буферный кэш может использоваться рекомендация linux считать две следующие страницы по 4Кб (итого блок PostgreSQL размером BLCKSZ=8Кб):
`PrefetchSharedBuffer()->smgrprefetch()-> smgrsw[].smgrprefetch = mdprefetch -> FilePrefetch(,BLCKSZ,) -> posix_fadvise(.., BLCKSZ, POSIX_FADV_WILLNEED)`

Для синхронизации грязных буферов используется вызов `sync_file_range(fd, offset, nbytes, SYNC_FILE_RANGE_WRITE)`. Вызов `posix_fadvise(fd, offset, nbytes, POSIX_FADV_DONTNEED)` не используется по умолчанию, так как имеет побочный эффект - помимо записи изменившихся страниц, убирает из памяти страницу и изменившуюся и неизменившуюся.

По каждому файлу вызовы writeback выполняются по диапазонам блоков. Ссылки на блоки для синхронизации (в будущем) записываются хэш-таблицу на 100 штук блоков, создаваемую функцией `hash_create("Pending Ops Table" или "pending sync hash"..)` **в локальной памяти** checkpointer, сортируются функцией sort_pending_writebacks(..), чтобы упорядочить блоки для передачи диапазона блоков. **fsync() выполняется один раз по каждому файлу (где были изменения хоть в одном блоке) в конце контрольной точки.**

Для синхронизации нужно запомнить все файлы, которые были изменены с момента последней контрольной точки, чтобы выполнить синхронизацию до завершения следующей контрольной точки. Хэш-таблица (а не linked list) выбрана, чтобы исключать дублирование команд (операций) на запись одного и того же блока. В хэш-таблицах запоминаются блоки, которые нужно синхронизировать. Для команд на удаление файлов используется linked list, так как повторных команд (операций) удаления файлов не должно быть.

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

Число вызовов синхронизации, которые выполняет процесс контрольной точки до засыпания, ограничивается константами:

```
/* Intervals for calling AbsorbSyncRequests */
#define FSYNCS_PER_ABSORB    10
#define UNLINKS_PER_ABSORB   10
/* interval for calling AbsorbSyncRequests in CheckpointWriteDelay */
#define WRITES_PER_ABSORB    1000
```

Временные таблицы не синхронизируются, так как для них отказоустойчивость не нужна.
https://medium.com/@hnasr/following-a-database-read-to-the-metal-a187541333c2

Синхронизация файлов данных с диском

- выполняет преимущественно checkpointer
- выполняется по диапазонам блоков системным вызовом: `sync_file_range(fd, offset, nbytes, SYNC_FILE_RANGE_WRITE)`
- если на системный вызов выдается ошибка записи, то экземпляр аварийно останавливается и блоки будут восстановлены по журналам WAL благодаря их полным образам (параметр конфигурации `full_page_writes=on`), если `data_sync_retry=off`

```
postgres=# \dconfig *flush*
List of configuration parameters
       Parameter        | Value
------------------------+-------
 backend_flush_after    | 0
 bgwriter_flush_after   | 512kB
 checkpoint_flush_after | 256kB
 wal_writer_flush_after | 1MB
(4 rows)
```

```
postgres=# select backend_type name, sum(writes) buffers_written, round(sum(write_time)) w_time,
sum(writebacks) writebacks, sum(evictions) evictions, sum(fsyncs) fsyncs, round(sum(fsync_time))
fsync_time from pg_stat_io group by backend_type having sum(writes)> 0 or sum(writebacks)> 0 or
sum(fsyncs)>0 or sum(evictions)>0;
       name         | buffers_written | w_time | writebacks | evictions | fsyncs | fsync_time
--------------------+-----------------+--------+------------+-----------+--------+------------
 background worker |               7 |      0 |          0 |        43 |      0 |          0
 client backend    |         7451184 |  83639 |          0 |  15253670 |      0 |          0
 autovacuum worker |           61803 |    705 |          0 |     94784 |      0 |          0
 background writer |         2344704 |  26415 |    2344704 |           |      0 |          0
 checkpointer      |         5595043 |  99818 |    5598779 |           |  84627 |     843454
(5 rows)
```


Синхронизация файлов данных с диском

На Linux для выполнения fsync **по файлам данных** используется системный вызов:
`sync_file_range``(fd, offset, nbytes, SYNC_FILE_RANGE_WRITE);`
Вызов проверяет, что диапазон страниц файла сброшен операционной системой на диск.
Процесс checkpointer выполняет большую часть fsync и только часть write (копирование блоков их буферного кэша в страничный кэш linux).
Если по умолчанию checkpoint_flush_after не равен нулю, то синхронизация по диапазонам блоков файла включена, так как в исходном коде PostgreSQL:

```
 /* Default and maximum values for backend_flush_after, bgwriter_flush_after and checkpoint_flush_after; measured
in blocks.  Currently, these are enabled by default if sync_file_range() exists, ie, only on Linux.  Perhaps we
could also enable by default if we have mmap and msync(MS_ASYNC)? */
 #ifdef HAVE_SYNC_FILE_RANGE
 #define DEFAULT_BACKEND_FLUSH_AFTER 0 /* never enabled by default */
 #define DEFAULT_BGWRITER_FLUSH_AFTER 64
 #define DEFAULT_CHECKPOINT_FLUSH_AFTER 32
 #else
 #define DEFAULT_BACKEND_FLUSH_AFTER 0
 #define DEFAULT_BGWRITER_FLUSH_AFTER 0
 #define DEFAULT_CHECKPOINT_FLUSH_AFTER 0
 #endif
```

Обычно, серверные процессы не синхронизируют файлы данных, это делают checkpointer и bgwriter. Если в структуре разделяемой памяти Checkpinter Data (из неё checkpointer перемещает идентификаторы в хэш-таблицу в своей локальной памяти "Pending Ops Table"), в которую все процессы посылают идентификаторы блоков на синхронизацию, не будет места, серверный процесс начнет выполнять fsyncs. Пример будет рассматриваться в части 2 практики к главе 14 "Выполнение fsyncs при остановленном checkpointer".
Если операционная система откажется выполнять вызов с ошибкой "не реализовано", то в диагностический журнал выдается предупреждение: "`could not flush dirty data:`".
Если операционная система не сможет выполнить системный вызов синхронизации и выдаст ошибку записи, то экземпляр немедленно останавливается со статусом PANIC и блоки, по которым linux выдал отказ в синхронизации будут восстановлены по WAL с момента начала последней контрольной точки и с использованием полного образа блока (full page image), а значит гарантированно восстановлены. Можно отключить падение с PANIC установкой параметра data_sync_retry в on (по умолчанию off), но делать это не стоит, так как будут повреждены блоки данных.
https://habr.com/ru/articles/803347/

Размер блока файловой системы

- должен быть равен размеру страницы - 4Кб
- файловая система ext4 лучший выбор
- параметры создаваемой по умолчанию ext4 оптимальны
- список параметров, с которыми смонтирована файловая система: **`cat /proc/fs/ext4/sda1/options`**

Размер блока файловой системы

На дисках 4Kn (размер физического сектора 4096 байт и размер логического сектора 4096 байт) утилита `mkfs` будет использовать размер блока 4096 байт.

На дисках 512e (размер физического сектора 4096 байт, размер логического сектора 512 байт) и 512n (размер физического сектора 512 байт и размер логического сектора 512 байт) mkfs.ext4 по умолчанию использует блоки размером 1024 байта для файловых систем размером менее 512 МБ и блоки размером 4096 байт для файловых систем размером 512МБ и больше.

Linux будет монтировать только файловые системы с размером блока, меньшим или равным размеру страницы памяти, то есть 4Кб.

Для СУБД PostgreSQL оптимальна файловая система ext4.

Параметры монтирования указываются в:

```
root@tantor:~# cat /etc/fstab | grep ext4
UUID=aca1a090-eba2-49ba-a8fc-ba12e9e2bf26 / ext4 defaults 1 1
```

defaults означает, что параметры берутся из самой файловой системы (суперблока). Параметры можно посмотреть и поменять утилитой `tune2fs`:

```
root@tantor:~# tune2fs -l /dev/sda1 | grep opt
Default mount options:    user_xattr acl
```

Параметры, с которыми смонтирована файловая система:

```
root@tantor:~# mount | grep ext4
/dev/sda1 on / type ext4 (rw,relatime)
```

Полный список параметров: **`cat /proc/fs/ext4/sda1/options`**

Параметры, создаваемой файловой системы ext4 по умолчанию оптимальны, при поиске рекомендаций нужно обращать внимание не устарели ли они. Например, рекомендация монтировать с опцией `noatime` устарела, так как используется более быстрая опция `realtime`. Параметр `nobarrier` существенно не увеличивает производительность. Параметр `data=ordered` означает, что запись блоков данных производится до записи метаданных в журнал файловой системы. Параметр `data=writeback` приводит к порче содержимого файлов.

https://www.enterprisedb.com/blog/postgres-vs-file-systems-performance-comparison

Параметр wal_sync_method

- системный вызов, который используется для того, чтобы гарантированно сохранить записи блоков в WAL файл, если значение параметра `fsync=on`
- `fdatasync` - значение по умолчанию для linux

```
postgres=# show fsync;
 fsync
-------
 on
postgres=# show wal_sync_method;
 wal_sync_method
-----------------
 fdatasync
```


Параметр wal_sync_method

Параметр задаёт метод который используется для того, чтобы гарантированно сохранить записи в WAL файле, если значение параметра `fsync=on`. Возможные значения:

`fdatasync` - вызывается `fdatasync()` при каждой (с учетом групповой фиксации транзакций, которая будут рассмотрена через 6 страниц) фиксации транзакции серверным процессом и при закрытии файла WAL, если он полностью заполнен. **Для linux значение по умолчанию.** Место под файлы WAL выделяется заранее и файлы создаются заранее, поэтому синхронизация метаданных файловой системы не нужна, поэтому отказоустойчивость `fdatasync` такая же, как у `fsync`.

`open_datasync` - файл WAL открывается на запись с параметром `O_DSYNC`

`fsync` - вызывается `fsync()` при каждой фиксации транзакции серверным процессом (с учетом группировки транзакций) и при закрытии файла WAL, если файл полностью заполнен

`open_sync` - файл WAL открывается на запись с параметром `O_SYNC`.

Если значение параметра `fsync=on`, то экземпляр не убеждается в том, что запись в WAL файл была выполнена. Измененные блоки могут остаться в кэше операционной системы по умолчанию до 5 секунд (параметр commit файловой системы ext4) и при внезапной остановке операционной системы или пропадании питания будут потеряны.

Этот параметр не влияет на синхронизацию файлов данных, только на файлы WAL. При синхронизации файлов данных используются системные вызовы fsync, fdatasync с файлами данных не используется, так как файлы данных могут увеличиваться в размерах.

Гарантия записи в WAL

- системные вызовы `fsync()` или `fdatasync()`, вызываемые после записи в WAL журнальной записи о COMMIT при `wal_sync_method = fdatasync` или `fsync` гарантируют, что запись блоков данных в WAL файл будет выполнена контроллером диска до возврата результата из этих функций
- перед записью в файл WAL он создается (переименовывается) полного размера и пока в него идёт запись его метаданные не меняются и гарантия записи в журнал файловой системы для записи в WAL не имеет значения

Гарантия записи в WAL

Вызовы к операционной системе `fsync()` и `fdatasync()` гарантируют, что запись блоков данных в WAL файл будет выполнена контроллером диска: "includes writing through or flushing a disk cache if present. The call blocks until the device reports that the transfer has completed" о чем написано в описании вызова:

https://man7.org/linux/man-pages/man2/fsync.2.html

Поскольку перед записью в файл WAL он создается (переименовывается) полного размера, то в процессе записи в файл WAL его атрибуты не меняются пока процессы экземпляра пишут в файл WAL журнал JBD2 не играет роли.

Для справки рассмотрим как гарантируется отказоустойчивость записи в журналы файловой системы. Для того, чтобы блоки журналов файловой системы не задержались в кэше диска используются вызовы cache flush и опцией Force Unit Access (`O_FUA`). При использовании `journal=ordered` (другие не стоит использовать) сначала посылается команда записи данных контроллеру диска. Затем все измененные блоки журнала JBD2. Затем контроллеру диска посылается команда cache flush - записать всё что находится в кэше контроллера диска, чтобы гарантировать, что блоки данных и блоки журнала были сохранены и защищены от потери при пропадании питания. Дальше в JBD2 записывается один блок фиксации (commit block), указывающий, что транзакция была успешно завершена. Это позволяет гарантировать атомарность транзакции журнала. Блок фиксации записывается с использованием опции `O_FUA`, которая может быть установлена для записи только одного блока (а не нескольких) и является легкой операцией, так как не выполняет полную очистку кэша. Поскольку транзакция в JBD2 состоит минимум из 3 блоков, то при каждой фиксации выполняется cache flush. Если транзакция в журнале быстрой фиксации состоит из одного блока, то при записи в журнал быстрой фиксации используется `O_FUA`, а не cache flush.

Быстрые фиксации изменений в журнале файловой системы ext4 (fast_commit)

- если WAL находятся на отдельной файловой системе, то быстрая фиксация не увеличит производительность
- параметр `commit` (по умолчанию 5 секунд) задает частоту вызова sync на файловой системе
- быстрые фиксации работают в режиме `data=ordered`
- быстрая фиксация уменьшает задержку на выполнение `fsync` и `fdatasync` но только в случае если метаданные изменились
- запись в журнал быстрой фиксации выполняется процессом, вызвавшим `fsync`
- запись в обычный журнал выполняется потоком ядра
- быстрая фиксация работает не только с ext4, но и с другими журналируемыми файловыми системами

Быстрые фиксации изменений в журнале файловой системы ext4 (fast_commit)

Файловая система ext4 записывает изменения в метаданных (в каких блоках что находится) в журнал типа JBD2 отдельно по каждой смонтированной файловой системе. Запись выполняется по логике транзакций, чтобы не было повреждений при пропадании питания. Параметр commit (по умолчанию 5 секунд) задает частоту вызова sync. `fsync` записывает изменения только по тому файлу, для которого вызывается.

По `fsync` сбрасываются все грязные блоки и метаданные для всех файловых дескрипторов. Это может создать большой объем ввода-вывода, который не нужен для сохранения изменений в отдельном файле по `fsync`.

Если вынести директорию `PGDATA/pg_wal` на отдельный раздел файловой системы, где ничего кроме WAL файлов не будет храниться, для WAL это не существенно. Это играет роль когда часто создаются и удаляются файлы. Часто создаваться, удаляться и менять размер могут файлы временных таблиц, что актуально для приложений 1С.

Преимущества быстрой фиксации:

•быстрая фиксация записывает один блок 4Кб, при фиксации изменений в JBD2 сохраняется не менее 3 блоков (в среднем 6 блоков) на каждую фиксацию. Журнал быстрой фиксации содержит операции, выполненные с момента последней стандартной фиксации, которая вызывается по умолчанию раз в 5 секунд. Если быстрые фиксации не могут обработать операцию, используется запись в JBD2. Увеличение частоты commit может привести к тому, что fsync будет чаще выполняться путем записи в JBD2. Параметр commit также коррелирует с параметрами `vm.dirty_expire_centisecs` и `vm.dirty_writeback_centisecs`

•запись в JBD2 выполняется минимум двумя переключениями контекста выполнения, так как запись в JBD2 выполняется потоком ядра. Запись в журнал быстрой фиксации выполняется процессом, вызвавшим fsync. Использование планировщиков, отличных от none при большой нагрузке на процессор могут увеличить разницу.

Помимо снижения задержки выполнения fsync, задержка при использовании быстрой фиксации имеет меньший разброс по времени, чем запись в JBD2.

Быстрая фиксация работает не только с ext4, но и с файловыми системами, использующими другие типы журналов (xfs, jfs).

Описание быстрой фиксации:
https://www.usenix.org/system/files/atc24-shirwadkar.pdf

Статистика после запуска на виртуальных машинах `pg_test_fsync` два раза и `pgbench -c 3 -t 50000`. Количество клиентов 3, так как процессоров 4, при большем количестве узким местом стал бы процессор.

С включенным `fast_commit`:

```
postgres@tantor:~$ pgbench -c 3 -t 50000
number of transactions actually processed: 150000/150000
latency average = 2.147 ms
initial connection time = 12.987 ms
tps = 1397.598698 (without initial connection time)
root@tantor:~# cat /proc/fs/jbd2/sda1-8/info
895 transactions (867 requested), each up to 16320 blocks
average:
  844ms running transaction (number of milliseconds the transaction was running)
  3447us average transaction commit time
  90 handles per transaction (number of filesystem transaction handles for journal transaction)
  3 blocks per transaction (number of filesystem blocks in the transaction)
  5 logged blocks per transaction (total number of blocks written to the journal for this transaction, including journal overhead)
root@tantor:~# cat /proc/fs/ext4/sda1/fc_info
fc stats:
fc stats:
180988 commits
731 ineligible
181026 numblks
596us avg_commit_time
```

С выключенным fast_commit:

```
postgres@tantor:~$ pgbench -c 3 -t 50000
number of transactions actually processed: 150000/150000
latency average = 2.125 ms
initial connection time = 12.188 ms
tps = 1411.806469 (without initial connection time)
root@tantor:~# cat /proc/fs/jbd2/sda1-8/info
27446 transactions (27430 requested), each up to 16384 blocks
average:
  12ms running transaction
  4447us average transaction commit time
  2 handles per transaction
  1 blocks per transaction
  3 logged blocks per transaction
```

Приведенные данные по большей части относятся к работе `pg_test_fsync`.

150000 COMMIT вызвали 150000 fdatasync(), но они были обработаны 22 фиксациями в JBD2 как при включенном, так и при выключенном `fast_commit`, то есть раз в 5 секунд. В примере это не видно, для измерения нужно перед запуском pgbench и в процессе работы pgbench перечитывать info, будет видно что фиксации в JBD2 идут раз в 5 секунд.

При включенном `fast_commit` в JBD2 каждая транзакция записывала немного больше блоков, а также по каждому COMMIT записывала блоки (150000 из 181026 numblks) в журнал быстрой фиксации.

Если включить fsync вместо fdatasync, то разница в TPS будет больше: 970 (с включенным `fast_commit`) и 1300. При этом фиксаций в JBD2 будет 590 и 28600.

Это относится к записи в журналы. Записи блоков данных в файл WAL гарантированно записываются при каждом COMMIT.

Быстрые фиксации изменений в журнале файловой системы ext4 (fast_commit)

- параметр commit (по умолчанию 5 секунд) задает частоту вызова sync на файловой системе
- быстрые фиксации работают в режиме `data=ordered`
- включение быстрых фиксаций:

```
root@tantor:~# dumpe2fs /dev/sda1 | grep Fast
dumpe2fs 1.47.0 (5-Feb-2023)
Fast commit length:       0
root@tantor:~# tune2fs -O fast_commit /dev/sda1
tune2fs 1.47.0 (5-Feb-2023)
root@tantor:~# dumpe2fs /dev/sda1 | grep Fast
dumpe2fs 1.47.0 (5-Feb-2023)
Fast commit length:       256
```


Быстрые фиксации изменений в журнале файловой системы ext4 (fast_commit)

Для поддержки быстрых фиксаций нужен журнал быстрой фиксации для операций.
Быстрые фиксации работают в режиме `data=ordered`.
Проверка доступен ли журнал быстрых фиксаций:

```
root@tantor:~# dumpe2fs /dev/sda1 | grep Fast
Fast commit length:       0
```

Ноль означает, что не доступен и быстрые фиксации не используются. Включение журнала:

```
root@tantor:~# tune2fs -O +fast_commit /dev/sda1
root@tantor:~# dumpe2fs /dev/sda1 | grep Fast
Fast commit length:       256
```

Отличное от нуля значение означает, что журнал быстрых фиксаций доступен на этой файловой системе. Файловую систему нужно перемонтировать, чтобы быстрые фиксации стали использоваться. Убедиться в использовании можно посмотрев **Статистику быстрых фиксаций:**

```
cat /proc/fs/ext4/sda1/fc_info
```

Также проверить доступен ли журнал fast_commit можно командой:

```
root@tantor:~# tune2fs -l /dev/sda1 | grep fast_commit
Filesystem features:      has_journal ext_attr resize_inode dir_index
fast_commit filetype needs_recovery extent 64bit flex_bg sparse_super
large_file huge_file dir_nlink extra_isize metadata_csum
```

Отключение fast_commit:

```
root@tantor:~# tune2fs -O ^fast_commit /dev/sda1
```

Почему по умолчанию журнал быстрой фиксации не создается? Для обычного десктопного использования, где fsync не вызывается часто `fast_commit` не дает прироста производительности. Запись в JBD2 идет независимо от наличия быстрой фиксации и объем записи в JBD2 не уменьшается.

В в Astralinux 1.8.1 установлена версия 1.47.0.

Включение-выключение `fast_commit` поддерживается с версии 1.46 (2021 год):

"E2fsprogs now supports the fast_commit (COMPAT_FAST_COMMIT) feature. This feature, first available in Linux version 5.10, adds a fine-grained journaling which improves the latency of the fsync(2) system call. It should also improve the performance of ext4 file systems exported via NFS." https://e2fsprogs.sourceforge.net/e2fsprogs-release.html

Утилита `pg_test_fsync`

- утилита командной строки, которая выполняет стандартные тесты для оценки производительности при выборе значения для параметра `wal_sync_method`

```
postgres@tantor:~$ pg_test_fsync
5 seconds per test
O_DIRECT supported on this platform for open_datasync and open_sync.
Compare file sync methods using one 8kB write:
(in wal_sync_method preference order, except fdatasync is Linux's default)
        open_datasync                  3396.549 ops/sec     294 usecs/op
        fdatasync                      3459.610 ops/sec     289 usecs/op
        fsync                          3136.642 ops/sec     319 usecs/op
        fsync_writethrough                   n/a
        open_sync                      3275.082 ops/sec     305 usecs/op
Compare file sync methods using two 8kB writes:
```


Утилита `pg_test_fsync`

Утилита командной строки которая выполняет тесты для оценки производительности при выборе значения для параметра `wal_sync_method`. Утилита записывает в тестовый файл один или два 8Кбайтныйх блока, так же как это делает процесс сохраняющий в WAL-файл запись о фиксации транзакции.

Утилита выдает количество операций записи блоков в секунду, что может служить оценкой для максимального TPS (количества транзакций в секунду) для транзакций небольшого размера, если узким местом является запись в WAL-файлы.

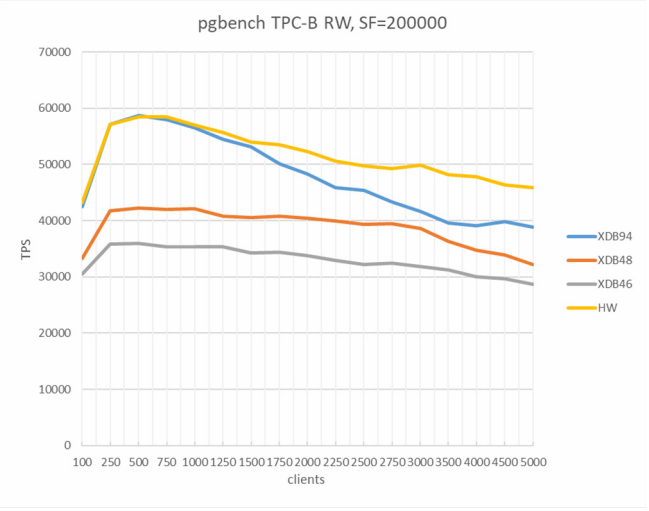
По умолчанию тестирование каждого метода длится 5 секунд.

pg_test_fsync выводит среднее время операции синхронизации с ФС для каждого метода `wal_sync_method`, что может быть полезно при выборе значения для параметра `commit_delay`, если планируется использовать `commit_siblings`. Эти два параметра могут увеличить TPS, если на экземпляре имеется большое число параллельно фиксирующихся транзакций и узким местом является запись в WAL-файлы.

Поддержка `O_DIRECT` означает, что на операционной системе поддерживается прямой ввод-вывод и можно (но не нужно) использовать параметр `debug_io_direct`.

Групповая фиксация транзакций

- транзакции могут фиксироваться группами
- обрабатывая COMMIT не обязательно что каждый серверный процесс выполнит запись в WAL-файл
- журнальная подсистема обычно не является узким местом при этом гарантирует отказоустойчивость
- алгоритм работы с WAL обеспечивает масштабируемость: при увеличении числа сессий до ~500 TPS растет




```
HW - физический сервер Tantor xData 96CPU, 1536GB RAM, 18TB SSD
shared_buffers=384GB

контейнер Tantor XDataBox
94CPU, 1120GB RAM, LVM 5TB,shared_buffers=384GB
48CPU, 560GB RAM, LVM 5TB, shared_buffers=192GB
46CPU, 560GB RAM, LVM 5TB, shared_buffers=192GB
```


Групповая фиксация транзакций

Если выполнение записи журнальных буферов на диск занимает время и в этот промежуток времени другие серверные процессы выполняют COMMIT, то нельзя сказать, что сброс буферов будет выполнен каждым из этих процессов. Наибольшую задержку обычно вызывает выполнение вызова, гарантирующего запись на диск буферов (flush cache) - по умолчанию `fdatasync`. Это достигается алгоритмом работы процессов с журнальным буфером и логикой получения блокировок.

Когда процесс желает выполнить сброс буферов он должен получить блокировку `WALWriteLock`. Если процесс ее получил немедленно, то он становится "лидером" и начинает сброс всех накопленных в буфере журнала буферов, а потом дает `fdatasync` (если не используется другой метод гарантированной записи). Другие процессы которые сформировали запись о своем `COMMIT` и готовые тоже скинуть буфера на диск встают в очередь за блокировкой `WALWriteLock`. Когда блокировка становится доступной, ни один из процессов стоящих в очереди сразу не получает ставшую доступной блокировку. Вместо этого каждый процесс еще раз проверяет, были ли сброшены буфера до и включая журнальную запись с `COMMIT` их транзакции. Часто процессы обнаруживают, что журнальные записи сброшены "лидером". В этом случае процессы отказываются от получения блокировки, записи в WAL-сегмент и подтверждают клиенту что транзакция зафиксирована. При значительных задержках выполнения `fdatasync` и большой частоте транзакций небольшая часть процессов ("лидеров") выполнят сброс буферов. Получается что фиксация транзакций обрабатывается группой (batch). Благодаря этому алгоритму журнальная подсистема обычно не является узким местом.

https://pgeoghegan.blogspot.com/2012/06/towards-14000-write-transactions-on-my.html

https://rutube.ru/video/private/e9a69100951dd2865db96ec49293423c/?p=G5067rhW71F2GeS7U3cJ7Q

Параметры commit_delay и commit_siblings

- параметр конфигурации устанавливает максимальную задержку в микросекундах, которую серверный процесс будет ждать после создания журнальной записи с COMMIT в буфере WAL перед вызовом команды записи в WAL-файл если имеется commit_siblings открытых транзакций и fsync=on
- по умолчанию значение ноль, задержки нет
- не ускоряет запись в WAL
- можно использовать для ограничения TPS чтобы не возникло узкое место в подсистеме отличной от WAL
- значение commit_delay, с которого можно начать - половина от задержки выдаваемой утилитой pg_test_fsync для используемого wal_sync_method
- изменение значений параметров commit_delay и commit_siblings не требует рестарта экземпляра

Параметры commit_delay и commit_siblings

Параметр commit_delay устанавливает максимальную задержку в микросекундах, которую серверный процесс будет ждать после создания журнальной записи с COMMIT в буфере WAL перед вызовом команды записи в WAL-файл. По умолчанию ноль - задержки нет. Серверный процесс будет ждать если: после формирования журнальной записи в экземпляре есть не меньше чем commit_siblings открытых транзакций, fsync=on. Причем, первый серверный процесс, который готов к передаче команды на запись (сбросу накопленных блоков журнального кэша на диск), ждет в течение commit_delay, следующие за ним процессы которые сформируют журнальную запись с COMMIT ждут только до тех пор, когда первый ожидающий процесс не выполнит сброс своей журнальной записи, их журнальных записей (которые были созданы в буфере пока первый процесс ждал) и всего что накопится в буфере к моменту окончания ожидания. Такая логика появилась начиная с версии 9.3 PostgreSQL, сам параметр появился в версии 7.1 когда методы сброса на диск работали неэффективно.

Поскольку fdatasync работает достаточно быстро, есть алгоритм групповой фиксации транзакций, commit_delay не ускоряет запись в WAL-файл. Этот параметр можно использовать, если большое количество транзакций в секунду нагружает экземпляр так, что не в журнальной системе, а в какой-то другой возникает узкое место, которое сложно устранить другими способами, кроме как уменьшить число транзакций в секунду (TPS), вставив небольшую задержку в выполнение команды COMMIT. Предполагается, что приложение, встретившись с уменьшением TPS, не станет увеличивать число сессий. Большая нагрузка обычно возникает не с 5 сессиями, а на прядки больше, поэтому значение commit_siblings должно быть достаточно большим. Можно провести аналогию: при ламинарном течении жидкости пропускная способность трубы высокая, при турбулентном она снижается и вызывает вибрацию трубы и нужно следить за тем, чтобы напор жидкости ограничивался когда течение станет турбулентным где-то обычно в наиболее узком месте.

Значение commit_delay, с которого можно начать - половина от задержки выдаваемой утилитой pg_test_fsync для используемого wal_sync_method (обычно fdatasync).

Типичный диапазон значений commit_delay 200-1000. Для commit_siblings ~500.

Изменение значений параметров commit_delay и commit_siblings не требует рестарта экземпляра. Изменение значения commit_delay требует привилегии на изменение или атрибут суперпользователя, commit_siblings может менять любой пользователь.

https://postgresqlco.nf/doc/en/param/commit_delay/

https://pgpedia.info/c/commit_delay.html

Команды шин ввода-вывода `discard/trim`

- важен для SSD
- `trim` - для SSD на шине SATA
- `discard` - для SSD на шине pci
- контроллер SSD не знает о том, что на нем располагаются файлы и есть файловые системы
- операционная система должна посылать команды очистки блоков (физических секторов) содержимое которых ей не нужно
- ресурс записи в режиме SLC ~100000 циклов
- ресурс записи не-SLC ~1000 циклов

Команды шин ввода-вывода `discard/trim`

При использовании SSD на любых шинах - PCIe (NVMe) или SATA операционная система должна сообщать контроллеру устройства о том, что содержимое физических блоков не нужно операционной системе. Контроллер SSD не знает о том, что на нем располагаются файлы и есть файловые системы. Операционная система должны посылать команду trim (для шины SATA) или discard (для шины pci) очистки блоков (физических секторов).

После инициализации диска и создания разделов данные о том какие блоки используются хранятся в файловой системе. Для SSD критически важно получать данные от операционной системы, что какие-то блоки не используются файловой системой и могут быть очищены.

Например, SSD была инициализирована, создан раздел, отформатирован и смонтирован. Таблица FTL устройства SSD хранит всего несколько записей о том, что несколько блоков используются операционной системой. Раздел начинает заполняться файлами. Контроллер SSD записывает их в "SLC кэш" (высокоскоростной физический слой с ресурсом около 100000 операций записи). Максимальный размер SLC зависит от чипов памяти. В настоящее время распространены TLC 3 бита (SLC максимум 1/3 за вычетом резервных блоков, практически меньше, так как зависит от алгоритма по которому контроллер использует SLC), для дешевых QLC 4 бита (SLC максимум 1/4 размера за вычетом резервных блоков).

Пока заполняется SLC контроллер (зависит от алгоритма работы контроллера) может перемещать из SLC блоки в другие слои, запись в которые в несколько раз медленнее, но самое главное, что количество записей в такие слои невелика - около 1000. Также контроллер если SLC кэш заполнен может принимаемые блоки записывать в другие слои со скоростью в несколько раз меньше. В файловой системе файлы могут удаляться и место освобождаться. Контроллер об этом не будет знать, так как пометка о том, что файловая система перестала использовать такие-то блоки хранятся только в ее метаданных. Чтобы SSD узнал, что блоки занятые удаленным файлам больше не нужны, операционная система должна при удалении в файловой системе файла передать контроллеру SSD команду discard или trim указав диапазон блоков, которые больше не нужны (обычно те, которые занимал файл, если файловая система не объединяла несколько файлов в один блок для экономии места). Будет ли посылать linux такую команду устанавливается в свойствах монтирования файловой системы.

Не стоит полагаться на то, что при инициализации SSD (создании раздела GPT/MBR) или при создании файловой системы командой mkfs.ext4 будет выполнен discard на весь диапазон блоков. Если SSD использовался, стоит отдельно инициировать discard.

Поддержка `discard/trim`

- если у контроллера не будет свободных блоков скорость записи и срок жизни SSD будут низкими
- контроллер SSD обычно очищает блоками размера `erase size`

```
root@tantor:~# tune2fs -o +discard /dev/sda1
root@tantor:~# tune2fs -l /dev/sda1 | grep discard
Default mount options:    user_xattr acl discard
root@tantor:~# fstrim -v /
fstrim: /: the discard operation is not supported
root@tantor:~# lsblk --discard
NAME    DISC-ALN DISC-GRAN DISC-MAX DISC-ZERO
sda            0        0B       0B         0
└─sda1         0        0B       0B         0
```


Поддержка `discard/trim`

Для очистки использованного ранее SSD можно использовать утилиту `blkdiscard`.

Для очистки блоков, не используемых смонтированной файловой системой можно использовать утилиту `fstrim`.

Контроллер SSD очищает блоками размера `erase size`. Если диапазон блоков не вписывается в границы `erase size`, то `discard` по физическим блокам попадающим на часть блока erase size обычно (поведение SSD полностью определяется микропрограммой контроллера) игнорируется и такие блоки не очищаются. Размер `erase size` может быть десятки мегабайт.

В описании (выдаётся командой `man`) утилиты `fstrim` написано, что для большинства десктопов и серверов ее достаточно запускать раз в неделю.

Файловая система может быть смонтирована с опцией discard и при удалении файла в файловой системе будет посылаться команда `discard/trim`. Время на выполнение контроллером команды `trim` (если только она не даётся по всему SSD) может быть пропорционально количеству блоков. Если размер файла большой, то контроллер может приостановить прием команд от операционной системы на некоторое время. PostgreSQL не использует большие файлы. В СУБД Tantor максимальный размер файлов данных и журналов 1Гб. Однако linux может обслуживать и другие приложения, для них придумали логику по которой файловая система монтируется без `discard`, при этом очистка выполняется службой `fstrim`. Лучше включить `dicsard` в параметрах монтирования `ext4`.

Свойство discard можно установить в суперблоке ext4 как параметр монтирования по умолчанию: **`tune2fs -o +discard /dev/sda1`** или в `/etc/fstab`:

```
/dev/sda1 / ext4 rw,discard 0 0
```

Для LVM в `/etc/lvm/lvm.conf`: **`devices`** `{issue_discards = 1}`

Для шифрованных разделов в `/boot/grub/grub.cfg`

Проверить посылались ли discard контроллеру SSD можно командами:

```
root@tantor:~# fstrim -v /
fstrim: /: the discard operation is not supported
root@tantor:~# lsblk --discard
NAME    DISC-ALN DISC-GRAN DISC-MAX DISC-ZERO
sda            0        0B       0B         0
└─sda1         0        0B       0B         0
```

Нули в `DISC-GRAN` и `DISC-MAX` означают, что `discard` не используется.

Рекомендации по использованию SSD

- не рекомендуется активно писать на SSD, если процент заполнения больше чем ~80% (зависит от алгоритма контроллера SSD)
- Для того чтобы часть SSD была всегда свободна можно создавать раздел меньшего размера, чем SSD
- статус службы, выполняющей discard(trim) по неиспользуемым блокам на файловых системах, использующие устройства с поддержкой discard(trim):

```
root@tantor:~# systemctl status fstrim.timer
  fstrim.timer - Discard unused blocks once a week
  Loaded: loaded (/lib/systemd/system/fstrim.timer; enabled; preset: enabled)
  Active: active (waiting) since ..; 7h ago
 Trigger: 1 day 6h left
Triggers: fstrim.service
   Docs: man:fstrim
systemd[1]: Started fstrim.timer - Discard unused blocks once a week.
```


Рекомендации по использованию SSD

При работе с SSD полезна утилита `nvme`. Она устанавливается в пакете `nvme-cli`, который зависит от пакета uuid-runtime:

```
root@tantor:~# apt install nvme-cli uuid-runtime -y
```

Количество блоков, которые могут одновременно очищаться может быть ограничено. Посмотреть ограничения можно: `/sys/block/sda/queue/discard_max_hw_bytes`

"Many experts recommend limiting SSD usage to only 80% of its total capacity":
https://www.seagate.com/nl/nl/blog/what-are-ssd-trim-and-garbage-collection/

Для того чтобы часть SSD была всегда свободна можно создавать раздел меньшего размера, чем SSD.

Рекомендуется периодически (раз в неделю) очищать неиспользуемые блоки, это делает служба fstrim.

https://wiki.archlinux.org/title/Solid_state_drive/Memory_cell_clearing

Под операционной системой Windows очистку можно выполнить вручную командой в командной строке с повышенными привилегиями:

```
powershell -command optimize-volume -driveletter C -retrim -verbose
```

Отключение журнала JBD2 в файловой системе ext без потери данных возможно командой:

```
tune2fs -O ^has_journal /dev/sda1
tune2fs 1.47.0 (5-Feb-2023)
The has_journal feature may only be cleared when the filesystem is
unmounted or mounted read-only.
```

Включение журнала JBD2:

```
tune2fs -O +has_journal /dev/sda1
```

https://wiki.astralinux.ru/pages/viewpage.action?pageId=48759308

Параметр max_files_per_process

- значение по умолчанию 1000
- по достижению значения серверный процесс будет часто закрывать-открывать файлы
- большое количество файлов может открываться если команды работают с большим количеством relations или таблицами состоящими из большого количества гигабайтных файлов

```
select setting, min_val,max_val,vartype, short_desc from pg_settings where
name = 'max_files_per_process';
 setting | min_val |  max_val   | vartype |          short_desc
---------+---------+------------+---------+------------------------------
 1000    | 64      | 2147483647 | integer | Sets the maximum number of
                       simultaneously open files for each server process.
```


Параметр max_files_per_process

При работе с большим количеством объектов, например серверный процесс может открывать много файлов. У каждой таблицы три слоя плюс TOAST-таблица и индекс. При большом объеме данных основной слой может состоять из большого количества гигабайтных файлов (до 32K). Параметр кластера `max_files_per_process` ограничивает количество файлов, которые могут быть открыты каждым серверным процессом. Разделяемые библиотеки не учитываются. Значение по умолчанию 1000. Если серверный процесс должен работать с большим количеством файлов в течение небольшого времени, то можно увеличить значение параметра, иначе серверный процесс будет вынужден часто закрывать-открывать файлы. Большое количество файлов может открываться в сессиях, обслуживающих приложения 1С.

Значение 1000 установлено исходя из того что старые версии операционных систем устанавливали ограничение на количество открытых файлов в `/etc/security/limits.conf` (параметр `nofile`) или в `/etc/systemd/*.conf` (параметры `LimitNOFILE` , `LimitNOFILESoft`, `DefaultLimitNOFILE`). **Установка в этих файлах значений, отличных от infinity заставляет ядро собирать статистику открытых файлов, что не ускоряет его работу.**

```
root@tantor:~# cat /proc/sys/fs/file-nr
6336    0       9223372036854775807
```

первое число - всего открытых файлов в linux, второе - открытые, но не используемые в данный момент файлы, третье - максимально возможное число открытых файлов.

Максимальное значение параметра `max_files_per_process`:

```
postgres=# select setting, min_val,max_val,vartype, short_desc
           from pg_settings where name = 'max_files_per_process';
 setting | min_val |  max_val   | vartype |          short_desc
---------+---------+------------+---------+------------------------------
 1000    | 64      | 2147483647 | integer | Sets the maximum number of
                       simultaneously open files for each server process.
```

Посмотреть количество открытых процессом (PID=854) файлов:

```
root@tantor:~# lsof -p 852 | wc -l
66
```

Увеличение значения max_files_per_process

- при увеличении значения этого параметра нужно убедиться, что операционная система не ограничит число открытых файлов:

```
ERROR:  could not open relation 5/16550: Too many open files in system
```

- реальные ограничения запущенных процессов:

```
for PID in $(pgrep "postgres"); do cat /proc/$PID/limits | grep files; done | uniq
Max open files   1024     524288    files
```

- ограничения процессов запускаемых не через службы:

```
sudo -u postgres bash -c 'ulimit -n'
1024
```

- в файле /etc/security/limits.conf добавить строки:

```
postgres hard nofile infinity
postgres soft nofile infinity
```

- в файлах служб /usr/lib/systemd/system/tantor* добавить после [Service]

```
LimitNOFILE=infinity
LimitNOFILESoft=infinity
```


Увеличение значения max_files_per_process

Значение по умолчанию max_files_per_process=1000. При увеличении значения этого параметра нужно убедиться, что операционная система не ограничит число открытых файлов. Пример ошибок:

```
ERROR:  could not open relation 5/16550: Too many open files in system
```

Проверить реальные ограничения уже запущенных процессов с именем postgres:

```
for PID in $(pgrep "postgres"); do cat /proc/$PID/limits | grep files; done | uniq
Max open files   1024     524288    files
```

В примере soft limit 1024, hard limit 524288. При превышении soft limit процесс будет получать предупреждение. При превышении hard limit процесс не сможет открыть новые файлы, пока не будут закрыты ранее открытые файлы.

soft limit для одного процесса пользователя postgres можно посмотреть командой:

```
sudo -u postgres bash -c 'ulimit -n'
1024
```

С параметром -n (maximum number of open file descriptors) можно использовать параметры -S (soft) или -H (hard):

```
sudo -u postgres bash -c 'ulimit -Hn'
1048576
```

Чтобы поменять лимиты для экземпляров, запускаемых вручную утилитой pg_ctl, в файле /etc/security/limits.conf нужно добавить или поменять строки:

```
postgres hard nofile infinity
postgres soft nofile infinity
```

На запускаемые через systemd экземпляры это не подействует. Нужно отредактировать файл /usr/lib/systemd/system/tantor-se-server-16.service, добавив после [Service]

```
LimitNOFILE=infinity
LimitNOFILESoft=infinity
```

Установка значения отличного от infinity заставляет ядро собирать статистику открытых файлов, что не ускоряет его работу.

после редактирования обновить конфигурацию systemd и перезапустить экземпляр:

```
systemctl daemon-reload
systemctl restart tantor-se-server-16.service
```

Альтернативно можно редактировать командой systemctl edit tantor-se-server-16.service , обратив внимание на то, какой файл будет редактироваться.

Временная файловая система (tmpfs)

- файловая система, использующая оперативную память для хранения файлов
- СУБД Tantor не использует `tmpfs` для хранения файлов
- не стоит использовать `tmpfs`

```
root@tantor:~# mount | grep tmp
udev on /dev type devtmpfs
tmpfs on /run type tmpfs
tmpfs on /dev/shm type tmpfs
tmpfs on /run/lock type tmpfs
ramfs on /run/credentials/systemd-tmpfiles-setup-dev.service type ramfs
ramfs on /run/credentials/systemd-tmpfiles-setup.service type ramfs
tmpfs on /run/user/102 type tmpfs
tmpfs on /run/user/1000 type tmpfs
```


Временная файловая система (tmpfs)

`tmpfs` (temporary file system, временная файловая система), ранее известна как `shmfs`. Существует аналог `ramfs`. Это файловые системы, использующие оперативную память для хранения файлов. После перезагрузки операционной системы файлы в них исчезают. Если в операционной системе используется swap, то блоки `tmpfs` из памяти могут вытесняться в разделы или файлы swap.

СУБД Tantor не использует `tmpfs` для хранения файлов и не нужно их хранить в ней. В СУБД PostgreSQL до 15 версии статистика собиралась в директорию задаваемую параметром `pg_stat_tmp`, по умолчанию `PGDATA/pg_stat_tmp`. Запись была настолько активной, что приходилось монтировать эту директорию в `tmpfs`. В 15 версии этот неудачный функционал поменяли, но рекомендации были написаны и только путают. В табличных пространствах могут создаваться файлы, относящиеся к временным таблицам и индексам на временные таблицы. Эти файлы создаются в поддиректориях с названием `pgsql_tmp`. Например, `PGDATA/base/pgsql_tmp` созвучно с `pg_stat_tmp` и в памяти администратора может всплыть рекомендация монтировки чего-то с использованием файловой системы `tmpfs`. Таблица временная, но файлы постоянные. Эти файлы нельзя хранить на `tmpfs` и не имеет смысла хотя бы потому, что таблица может дорасти до 32Тб и либо не хватит места (если ограничить размер файловой системы `tmpfs` при монтировке), либо не хватит оперативной памяти и будет использован раздел (или файл) swap, либо oom kill начнет останавливать процессы сигналом `SIGKILL`.

Старые версии операционных систем linux могли монтировать временные директории как tmpfs (/tmp). Сейчас это не используется и не рекомендуется использовать хотя бы потому, что скорость записи в оперативную память больших объемов без использования Huge Pages обычно медленнее, чем запись на современные NVMe (до ~8 гигабайт в секунду по шине PCI 4x4).

В современных версиях linux файловая система tmpfs используется для хранения некоторых виртуальных служебных директорий (`/run`, `/dev/shm`) с файлами небольшого размера, либо функционалом код которого был написан давно, работает и переписывать который нет смысла.

RAID

- массив "дисков" (устройств хранения)
- используется как том (единое блочное устройство)

RAID

Термин RAID появился в 1987 году как сокращение от Redundant Array of Inexpensive Disks (массив недорогих дисков с избыточностью). Предлагалось использовать набор недорогих ненадёжных дисков, вместо дисков большой ёмкости "SLED" (Single Large Expensive Drive). Позднее Inexpensive заменили на Independent, так как стали использовать недешёвые диски. Виды ("уровни"):

RAID 0 (stripe) - stripe распределены по всем дискам, отказоустойчивости нет

RAID 1 (mirror) - дублирование (зеркалирование). Высокие накладные расходы

RAID 2,3,4 на практике не используются;

RAID 5 (stripe with parity) - stripe с данными чётности распределены по всем дискам, диски равноправны. Получил распространение благодаря небольшим накладным расходам: у зеркала расходы половина дисков против одного диска. Защищает от потери одного диска. Недостатки: скорость при записи в произвольном порядке (Random Write) меньше на 10-25% при сравнении с RAID 0 (чередование страйпов без контроля четности), так как каждая операция записи заменяется на две операции чтения и две операции записи. При замене вышедшего из строя диска инициализация нового диска сопровождается сильной нагрузкой на существующие, что может приводить к выходу из строя других дисков и полной потере массива без возможности восстановления. Не рекомендуется использовать с СУБД.

RAID 6 (double parity) - устойчив к выходу из строя двух дисков. Недостатки: скорость записи до ~2 раз ниже, чем у RAID 5; большие накладные расходы. Минимум 4 диска.

RAID 10 (1+0) зеркалированные пары дисков. Накладные расходы как у зеркала - половина размера дисков. Минимум 4 диска.

RAID 50 (5+0) минимальное количество дисков 6. Страйпы с четностью дублируются. Накладные расходы: два диска.

Поддержка RAID имеется в linux на уровне ядра. Linux поддерживает программные RAID уровней 0, 1, 4, 5, 6, 1+0. С помощью менеджера томов можно комбинировать уровни и получить 5+0. Управлять RAID-устройствами в Linux можно с помощью утилиты mdadm. Программный RAID использует ресурсы центральных процессоров. Центральные процессора обычно не являются узким местом для СУБД.

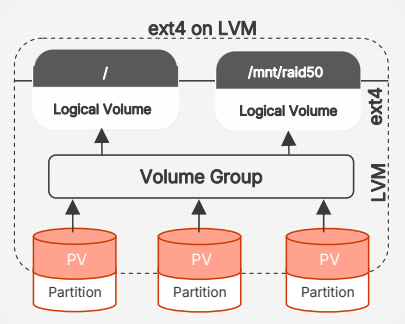
В Oracle ASM (менеджер томов и кластерная файловая система для хранения файлов Oracle Database) используется аналог программного (не на уровне ядра linux) RAID (двойное, тройное дублирование страйпов или без дублирования).

Также можно использовать аппаратное зеркалирование и RAID (Intel Matrix).

LVM

- Logical Volume Manager (менеджер логических томов)
- объединяет физические диски в группы - Volume Groups (VG), на которых можно создавать логические разделы Logical Volumes (LV)
- можно изменять размеры LV и добавлять новые диски в VG без перемещения данных и изменения файловых систем
- LVM не увеличивает производительность, добавляет дополнительный уровень абстракции, что увеличивает вероятность ошибок при переконфигурировании
- используется при создании RAID 5+0:

```
mdadm --create /dev/md0 --level=5 --raid-devices=2 /dev/sd1 /dev/sd2
mdadm --create /dev/md1 --level=5 --raid-devices=2 /dev/sd3 /dev/sd4
pvcreate /dev/md0
pvcreate /dev/md1
vgcreate vg0 /dev/md0 /dev/md1
lvcreate -L 100%FREE -n lv0 vg0
mkfs.ext4 /dev/vg0/lv0
mkdir /mnt/raid50
mount /dev/vg0/lv0 /mnt/raid50
```



LVM

Logical Volume Manager (менеджер логических томов). Позволяет на уровне операционной системы ("логически") объединять физические устройства хранения ("диски") в группы - Volume Groups (VG), на которых можно создавать логические разделы Logical Volumes (LV). Логические разделы можно использовать так же, как и обычные разделы жестких дисков. Преимущество в том, что можно изменять размеры LV и добавлять новые диски в VG без перемещения данных и изменения файловых систем.

LVM не увеличивает производительность и добавляет дополнительный уровень абстракции, что увеличивает вероятность ошибок при переконфигурировании.

LVM используются, если нет аппаратных массивов хранения и приходится использовать отдельные устройства.

Также LVM используется при создании RAID 5+0. Пример:

1) Создаются разделы на дисках, появляются блочные устройства типа `/dev/sdN`

2) Создание двух RAID 5:

```
mdadm --create /dev/md0 --level=5 --raid-devices=2 /dev/sd1 /dev/sd2
mdadm --create /dev/md1 --level=5 --raid-devices=2 /dev/sd3 /dev/sd4
```

3) Объединение RAID 5 в физические тома (PV, Physical Volume):

```
pvcreate /dev/md0
pvcreate /dev/md1
```

4) Объединение PV в VG:

```
vgcreate vg0 /dev/md0 /dev/md1
```

5) Создание логического тома (LV):

```
lvcreate -L 100%FREE -n lv0 vg0
```

6) Форматирование и монтирование логического тома:

```
mkfs.ext4 /dev/vg0/lv0
mkdir /mnt/raid50
mount /dev/vg0/lv0 /mnt/raid50
```

Для ротационных дисков (HDD), объединенных в RAID, можно установить параметр `effective_io_concurrency` в число дисков, по которым распределяются данные (страйпинг). Параметр включает или отключает (значение ноль) предварительную выборку блоков таблиц при индексном методе доступа Bitmap Heap Scan. Помимо обычных уровней, параметр можно установить в свойствах табличного пространства, как и параметр `maintenance_io_concurrency`.

Практика

- Часть 1. Параметры дисковой подсистемы
- Часть 2. Установка пакетов в Astralinux
- Часть 3. Работа с SSD и тестирование производительности диска утилитой `fio`
- Часть 4. Тестирование журнала быстрой фиксации ext4
- Часть 5. Снятие ограничения на число открытых файлов
- Часть 6. Пример командных файлов для тестирования
- Часть 7. Пример создания программ для тестирования

Практика

В практике вы сравните производительность интерфейсов SATA и NVME виртуальной машины на том же физическом устройстве и проверите, что использование интерфейса NVME быстрее на 10%, чем SATA.

Узнаете, что устройства NVME по умолчанию форматируются не в ext4, а в ext2 и узнаете даёт ли ext2 преимущества.

Вы научитесь включать журнал быстрой фиксации и какими параметрами оценивать производительность файловой системы.

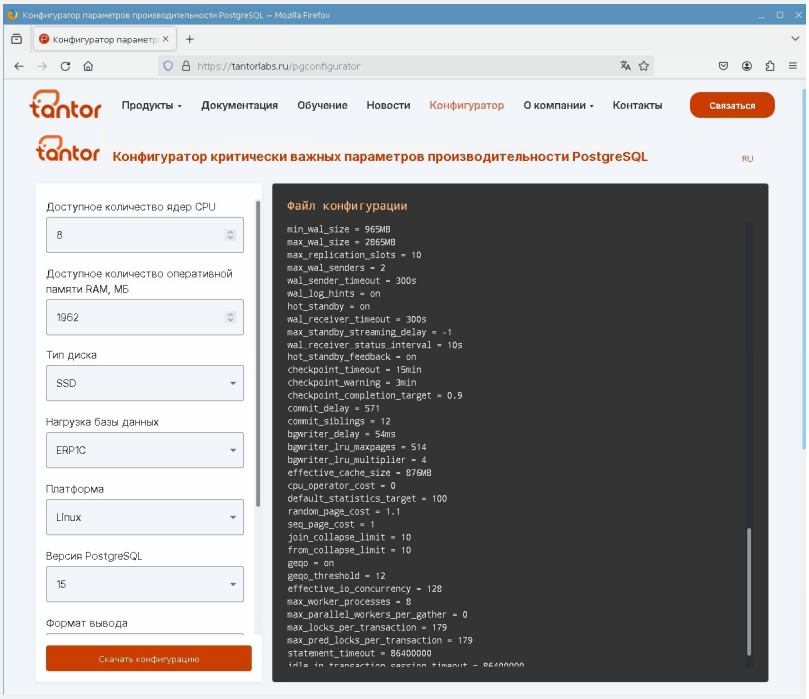
Изменения в файле limits.conf для снятия ограничений на число на число открытых файлов не действуют на службы. Вы изучите процедуру снятия ограничений и команды проверки реально действующих ограничений.

5

Начальная настройка СУБД

Конфигураторы

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



Конфигураторы

Кластер баз данных создается утилитой initdb. Утилита создает файл postgresql.conf со значениями по умолчанию. Эти значения рассчитаны на обслуживание не очень нагруженного приложения так, чтобы СУБД можно было использовать на десктопе обычному разработчику. В СУБД Tantor утилита initdb не меняет значения параметров по сравнению с initdb PostgreSQL. Предполагается, что настройка параметров для промышленного использования будет выполнена отдельно.

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

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

Утилит начальной конфигурации не очень много. Из известных:

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

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

https://tantorlabs.ru/pgconfigurator

Настройка работы PostgreSQL с продуктами 1С:

https://wiki.astralinux.ru/tandocs/nastrojka-postgresql-tantor-dlya-raboty-1s-294394904.html

Дальше рассматриваются параметры, значения которых устанавливают в первую очередь. Конфигуратор даёт начальные значения. Важно понимать смысл параметров и на что они влияют.

Параметры shared_buffers, temp_buffers, effective_cache_size

- shared_buffers по умолчанию 128MB. Можно установить в ~1/4 размера физической памяти. После изменения значения нужно перезапустить экземпляр
- effective_cache_size по умолчанию 4GB
  - дает планировщику оценку размера памяти, в которой могут поместиться блоки таблиц
  - влияет на расчет стоимости ввода-вывода, которая является часть общей стоимости плана выполнения
  - при небольшим значении параметра стоимость ввода вывода (cost) увеличивается для любых блоков - и таблиц и индексов и будут выбираться планы, в которых считывается меньше блоков
  - можно установить в 70-80% от размера физической памяти

Параметры shared_buffers, temp_buffers, effective_cache_size

При начальной установке значений параметров предполагается, что значения параметров, которые не будут упомянуты, установлены в значения по умолчанию. Если значения параметров отличаются от значений по умолчанию, то проверить не снижает ли это отказоустойчивость. Например, стоит проверить включен ли подсчет контрольных сумм. Если не включен, то включить. Проверить нет ли долго неиспользуемых инициализированных слотов репликации:

```
select slot_name, pg_current_wal_lsn()-restart_lsn from pg_replication_slots;
```

Параметры, значения которых стоит установить в зависимости от параметров хоста и нагрузки:

shared_buffers по умолчанию 128MB. Можно установить в ~1/4 размера физической памяти. После изменения значения нужно перезапустить экземпляр

effective_cache_size по умолчанию 4GB. Дает планировщику оценку размера памяти в которой могут поместиться блоки таблиц (кэш операционной системы и возможно кэш буферов если блоки в них не задублируются). Можно установить в 70-80% от размера физической памяти. Параметр не влияет на выделение памяти, только на оценки планировщика. Параметр может устанавливаться на любых уровнях (сессии, транзакции, функции и других) поэтому установленные значения могут не соответствовать размеру памяти, а использоваться для влияния на выбор планировщиком методов доступа. Стоимость (cost) плана выполнения складывается из двух частей: процессор и ввод-вывод. Параметр влияет на расчет стоимости ввода-вывода. **При небольшим значении этого параметра оценка стоимости ввода-вывода увеличивается** для любых блоков: и таблиц и индексов. **Будут выбираться планы, в которых считывается меньше блоков**.

temp_buffers задает размер локального (в памяти серверного процесса) кэша буферов, который используется при работе с временными таблицами. По умолчанию 8Мб. Значение можно поменять в сессии, но только до первого использования временных таблиц, после этого значение до конца сессии не меняется. Память выделяется постепенно по мере необходимости, а не сразу. Помимо памяти под хранение блоков временных таблиц, дополнительно в памяти серверного процесса сразу выделяется **по 64 байта на каждый буфер для хранения описателей буферов**. Если приложение активно использует временные таблицы, то значение параметра можно увеличить. Для приложений 1С увеличивают до 256Мб.

Параметры work_mem, hash_mem_multiplier, maintenance_work_mem

- work_mem по умолчанию 4MB
  - Вместе с параметром hash_mem_multiplier влияет на память, выделяемую каждым серверным и параллельным процессом
  - Начальное значение устанавливается в зависимости от оценки количества сессий, в которых выполняются команды, обрабатывающие в памяти большие наборы данных, то есть зависит от типа нагрузки OLTP (однострочные вставки и выборки) и не-OLTP
- maintenance_work_mem значение по умолчанию 64MB
  - Задает объем памяти, выделяемый каждым процессом (серверным, параллельным), участвующем в выполнении команд VACUUM, ANALYZE, CREATE INDEX, ALTER TABLE ADD FOREIGN KEY
  - Начальное значение ~0.5% от размера физической памяти

Параметры work_mem, hash_mem_multiplier, maintenance_work_mem

Параметр work_mem по умолчанию равен 4MB. Вместе с параметром hash_mem_multiplier влияет на память, выделяемую каждым серверным и параллельным процессом. Например, при соединении таблиц хэшированием (Hash Join) объем выделяемой памяти на обслуживание JOIN будет work_mem*hash_mem_multiplier*(Workers + 1). Параметр work_mem задает сколько памяти может использовать процесс для выполнения одного шага в плане выполнения, а не память на всю команду.

Помимо work_mem серверный процесс выделяет память под обработку одной строки, но обычно строки таблиц не очень длинные. Поэтому начальное значение устанавливается в зависимости от оценки количества сессий, в которых выполняются команды, обрабатывающие в памяти большие наборы данных. То есть значение зависит от типа нагрузки OLTP (однострочные вставки и выборки) и не-OLTP. Если исходить из того, что shared_buffers будет установлен в ~1/4 размера физической памяти, то ~3/4 будет занято кэшем операционной системы, чистые страницы кэша могут быть быстро освобождены, то памяти обычно достаточно.

Грубая оценка памяти, которую займёт экземпляр: shared_buffers+2*work_mem * процессов.

maintenance_work_mem значение по умолчанию 64MB. Задает объем памяти, выделяемый каждым процессом (серверным, параллельным), участвующем в выполнении команд VACUUM, ANALYZE, CREATE INDEX, ALTER TABLE ADD FOREIGN KEY.

Количество параллельных процессов ограничивается параметром max_parallel_maintenance_workers. Распараллеливается создание индексов и обычный (без FULL) вакуум. При вакуумировании только на фазе обработки индексов (другие фазы не распараллеливаются) один индекс может обрабатывать один (а не несколько) параллельный процесс. Будут ли использоваться параллельные процессы зависит от размера индексов и параметров конфигурации. Скорость и нагрузка, порождаемая командами обслуживания объектов базы ("maintenance"), сильно зависит от выделенной памяти, в таких случаях значение устанавливают перед выполнением команды на уровне сессии, но это не начальная настройка, а настройка в процессе эксплуатации. Начальное значение на уровне кластера зависит от объема доступной (чистых страниц страничного кэша) физической памяти, например, 0.5% от размера физической памяти. Выделяется сразу в начале выполнения команды. Совмещение VACUUM и ANALYZE в одной команде VACUUM (ANALYZE) не даст преимуществ, так как в текущих версиях PostgreSQL вакуум и анализ выполняются независимо друг за другом.

Параметр `autovacuum_work_mem`

- по умолчанию -1 (используется значение `maintenance_work_mem`)
- выделяется сразу каждым рабочим процессом автовакуума
- процессы автовакуума не используют параллельные процессы
- под хранение tid (tuple identifier) используется не больше 1Гб, независимо от значения параметра `maintenance_work_mem`

Параметр `autovacuum_work_mem`

Параметр `autovacuum_work_mem` по умолчанию -1 (то есть равно `maintenance_work_mem`). Выделяется **сразу каждым** рабочим процессом автовакуума. Количество рабочих процессов автовакуума ограничивается параметром `autovacuum_max_workers`, они не считаются параллельными процессами (то есть параметры `max_worker_processes`, `max_parallel_maintenance_workers` на них не влияют и они не исчерпывают пул параллельных процессов). Процессы автовакуума **не** используют параллельные процессы, в отличие от команды VACUUM (без FULL), каждую несекционированную таблицу с её индексами одновременно обрабатывает только один процесс autovacuum worker. Память используется для сохранения идентификаторов строк (tid), вышедших за горизонт видимости базы данных из блоков таблицы, которые могут содержать неактуальные версии строк, то есть отсутствуют в карте видимости (visibility map). Объем памяти под такие tid зависит не от размера таблицы, а от частоты вакуумирования (чем чаще оно выполняется, тем по вероятности меньше блоков будет сканироваться в текущем цикле автовакуума), от количества обновлений или удалений строк (зависит от того как работает приложение со строками), эффективна ли внутристраничная очистка. Если памяти `autovacuum_work_mem`, в которую записываются tid не хватит, чтобы записать все tid из блоков таблицы которые должны быть очищены, то будет несколько фаз очистки всех индексов по таблице и эффективность вакуумирования снизится: длительность возрастет, но что хуже нагрузка на ресурсы возрастут из-за повторных сканирований и очистки блоков индексов.

Значение параметра можно установить только на уровне кластера (на уровне баз данных или таблиц не устанавливается), для изменения значения достаточно перечитать конфигурацию.

Основная особенность в том, что **под хранение tid используется не больше 1Гб независимо от значений параметра**. Поэтому `autovacuum_work_mem` должно быть меньше 1Гб. Значение в которое устанавливать зависит от `autovacuum_max_workers` и свободной физической памяти. Например, если число ядер процессоров 4 а памяти 32Гб, то `autovacuum_max_workers=4` (не больше числа ядер) и `autovacuum_work_mem=1GB`.

В 17 версии под хранение tid используется не список, а адаптивное префиксное дерево (adptive radix tree) и число повторных сканирований индексов не линейно зависит от `autovacuum_work_mem`.

https://pganalyze.com/blog/5mins-postgres-17-faster-vacuum-adaptive-radix-trees

Параметры temp_file_limit и temp_tablespaces

- temp_file_limit задаёт максимальный объём дискового пространства, который сможет использовать один процесс для временных файлов
- temp_tablespaces определяет названия табличных пространств в которых будут создаваться файлы временных объектов

Параметры temp_file_limit и temp_tablespaces

temp_tablespaces устанавливает названия табличных пространств в которых будут создаваться файлы временных таблиц и индексов на временные таблицы; файлы, создаваемые при выполнении в командах SQL сортировок, соединений наборов данных, создания индексов, если не хватит памяти процесса (work_mem и maintenance_work_mem). По умолчанию список табличных пространств пуст и используется табличное пространство по умолчанию той базы данных, с которой работает процесс экземпляра.

Срок жизни временных файлов небольшой - до окончания выполнения команды, транзакции, усечения или удаления временной таблицы. В операционной системы используются обычные файлы.

Если в параметре temp_tablespaces указано несколько табличных пространств, процесс выбирает табличное пространство случайным образом при каждом создании временного объекта. Если объекты создаются в рамках транзакции, то для уменьшения задержек (на получение случайного значения) табличные пространства перебираются по кругу. Имена несуществующих табличных пространств или тех, которые не могут использоваться (нет привилегий) игнорируются и не вызывают ошибки.

Временные файлы могут достигать больших объемов, их появление может быть нежелательным в табличных пространствах с постоянно хранимыми данными. Из-за больших объемов записи во временные файлы использование систем хранения на основе магнитных дисков (HDD) возможно будет предпочтительнее массивов хранения на основе чипов памяти (SSD), так как ресурс последних определяется объемом записываемых данных.

Ограничение на размер временных файлов используемых одним процессом можно установить параметром temp_file_limit. По умолчанию ограничение не установлено.

temp_file_limit задаёт максимальный объём дискового пространства, который сможет использовать один процесс для временных файлов, например, при сортировке и хешировании или для сохранения удерживаемого курсора. При превышении ограничения команда прервётся, а транзакция, которая попытается превысить этот предел, будет отменена.

Ограничивается объём временных файлов, которые создаются неявно при выполнении команд. Этот параметр не ограничивает объем файлов временных таблиц, явно созданных командой CREATE TEMPORARY TABLE.

Параметры max_slot_wal_keep_size и transaction_timeout

- max_slot_wal_keep_size по умолчанию -1 (без ограничений) максимальный размер журнальных файлов, который может оставаться в каталоге pg_wal после выполнения контрольной точки для слотов репликации
- transaction_timeout по умолчанию ноль (таймаут отключен). Позволяет отменить любую транзакцию или одиночную команду, длительность которой превышает указанный период времени, а не только простаивающую. Защищает от удержания горизонта базы данных и раздувания файлов (bloat)

Параметры max_slot_wal_keep_size и transaction_timeout

Чтобы место неограниченно не расходовалось, стоит проверить или установить значения следующих параметров.

max_slot_wal_keep_size по умолчанию -1 (без ограничений). Максимальный размер журнальных файлов, который может оставаться в каталоге pg_wal после выполнения контрольной точки для слотов репликации. Если слот активирован и клиент не подсоединяется, то журнальные файлы удерживаются. Если этим параметром не установлено ограничение, то файлы журнала заполнят всю файловую систему и **экземпляр аварийно остановится**. Серверный процесс, который не сможет записать в журнал данные прервётся:

```
LOG:  server process (PID 6543) was terminated by signal 6: Aborted
```

Затем экземпляр попытается рестартовать:

```
LOG:  all server processes terminated; reinitializing
```

Чтобы не допустить нехватки места, стоит установить ограничение. Однако, реплике, которая не сможет получить журналы, а они будут стёрты, придётся получать файлы журналов откуда-то ещё или реплику придётся удалить и создать заново.

transaction_timeout по умолчанию ноль, таймаут отключен. Позволяет отменить не только простаивающую, но и любую транзакцию или одиночную команду, длительность которой превышает указанный период времени. Действие параметра распространяется как на явные транзакции (начатые с помощью команды BEGIN), так и на неявно начатые транзакции, соответствующие отдельному оператору. Параметр появился в СУБД Tantor версии 15.4.

Длительные транзакции и одиночные команды удерживают горизонт базы данных. Удержания горизонта базы данных не даёт вычищать старые версии строк и приводит к раздуванию (bloat) файлов объектов.

Параметры statement_timeout + idle_session_timeout не защищают от транзакций, состоящих из серии недолгих команд и коротких пауз между ними (например, длинная серия быстрых UPDATE в цикле). Для защиты от долгих команд SELECT может использоваться параметр old_snapshot_threshold. Его не стоит устанавливать на физических репликах. В 17 версии old_snapshot_threshold убран и transaction_timeout позволяет его заменить.

Параметры max_connections и client_connection_check_interval

- max_connections устанавливает максимальное число одновременных подключений к экземпляру
  - по умолчанию 100
  - Изменение значения требует перезапуск экземпляра, поэтому стоит установить в первую очередь исходя из планируемого количества сессий.
  - Смежные параметры reserved_connections и superuser_reserved_connections
  - при увеличении значения сначала увеличивать на физических репликах, иначе они перестанут обслуживать запросы
- client_connection_check_interval по умолчанию ноль, проверка отключена. Позволяет прерывать длительные запросы, результат которых клиент не сможет получить.

Параметры max_connections и client_connection_check_interval

max_connections по умолчанию 100. Максимальное число одновременных подключений к экземпляру. При увеличении значения нужно сначала увеличить на физических репликах, иначе они приостановят свою работу. Изменение значения передается через журналы. Изменение значения требует перезапуск экземпляра, поэтому стоит установить заранее исходя из планируемого количества сессий.

Смежные параметры reserved_connections и superuser_reserved_connections, но их можно не менять, так как у второго значение по умолчанию 3. Когда число одновременных подключений достигнет max_connections-superuser_reserved_connections, три последних соединения смогут выполнить только суперпользователи.

Примерные оценки количества соединений (серверных процессов): максимальные TPS на высокопроизводительном железе достигаются в районе ~500 **активных** (в которых выполняются команды), а не простаивающих сессий. **Экземпляр с настройками по умолчанию может эффективно обслуживать до ~4000 сессий. При большем количестве нужна настройка параметров экземпляра (например, размеры SLRU кэшей).**

client_connection_check_interval по умолчанию ноль, проверка отключена. Интервал между проверками (в процессе выполнения команды SQL) путем опроса состояния сокета, по которому не передаются данные. Сокет может быть закрыт другой стороной или кодом ядра операционной системы из-за неответа другой стороны на пакеты keepalive. Позволяет прерывать длительные запросы, результат которых клиент не сможет получить. Можно установить установить в ~30s или меньше. Если единица измерения не установлена, используются миллисекунды.

Число блокировок объектов и рекомендательных блокировок (advisory locks, которые автоматически не используются, их использование требует программирования) на экземпляре определяются произведением `max_locks_per_transaction * max_connections` (предполагается, что `max_prepared_transactions=0` и менять его не нужно). Произведение этих параметров определяет объем разделяемой памяти для хранения блокировок на экземпляре. Если место в структуре памяти будет исчерпано, то ни один процесс экземпляра не сможет получить блокировку на объект или advisory блокировку. Утилита `pg_dump` с использованием параллельных процессов (параметр `--jobs`) устанавливает блокировки на все выгружаемые объекты. При выгрузке базы данных - **на все объекты базы**. Если хочется, чтобы она не сбоила имеет смысл установить значение `max_locks_per_transaction` и `max_connections`, чтобы их произведение было не меньше числа объектов в каждой базе данных кластера.

Почему не установить параметр `max_connections` просто в большое значение?

Этот параметр определяет суммарный размер структур PGPROC, на которые указывает структура PROC_HDR. PROC_HDR (массив указателей) используется для поиска свободных слотов PGPROC при порождении процесса и проверки статуса.

Поля в структуре PGPROC используются всеми процессами для тех задач, которые требуют координации и поля очень часто опрашиваются. Например поля структуры PGPROC каждого процесса проверяется при формировании моментального снимка, то есть при каждом SELECT или начале транзакции. Также при синхронизации ожиданий/защелок. Поскольку PGPROC считываются часто, то указателям на страницы памяти PGPROC стоит удерживаться в кэшах оборудования (TLB). **Чем больше произведение `PGPROC*max_connections`, тем больше вероятность, что какой-нибудь еще структуре останется меньше места, скорость доступа упадет и работа процессов замедлится.**

Параметр max_connections также влияет и на размер общей структуры блокировок. Детали описаны в в `src/backend/storage/lmgr/lock.c`:

```
 /* Allocate hash table for LOCK structs.  This stores per-locked-object
information. */
  info.keysize = sizeof(LOCKTAG);
  info.entrysize = sizeof(LOCK);
```

Параметр `max_locks_per_transaction`

- число блокировок объектов и рекомендательных блокировок (advisory locks), которые могут использовать приложения, ограничено на экземпляре произведением `max_locks_per_transaction * (max_connections + max_prepared_transactions)`
- по умолчанию `max_prepared_transaction=0` и менять его не нужно если не используются распределенные транзакции
- формула устанавливает объем разделяемой памяти для хранения блокировок в памяти экземпляра
- утилита `pg_dump` с использованием параллельных процессов (параметр `--jobs`) устанавливает блокировки на все выгружаемые объекты
- для 300000 блокировок потребуется ~48Мб разделяемой памяти

Параметр `max_locks_per_transaction`

Количество блокировок объектов и рекомендательных блокировок (advisory locks), которые могут использовать приложения, ограничено на экземпляре произведением `max_locks_per_transaction * (max_connections + max_prepared_transactions)`. Формула задана в `src/backend/storage/lmgr/lock.c`:

```
#define NLOCKENTS() mul_size(max_locks_per_xact, add_size(MaxBackends, max_prepared_xacts))
```

По умолчанию `max_prepared_transactions=0` и менять не нужно, если не используются распределенные транзакции.

Формула определяет объем разделяемой памяти для хранения блокировок в памяти экземпляра. Памяти выделяется больше - до ближайшей степени двойки.

На каждую блокировку отводится ~168 байт в разделяемой памяти. Если место в структуре памяти будет исчерпано, то ни один процесс экземпляра не сможет получить блокировку на объект или advisory блокировку.

Утилита `pg_dump` с использованием параллельных процессов (параметр `--jobs`) устанавливает блокировки на все выгружаемые объекты. При выгрузке базы данных - **на все объекты базы**. Если хочется, чтобы выгрузка могла быть выполнена, имеет смысл установить значение `max_locks_per_transaction` и `max_connections`, чтобы их произведение было не меньше числа объектов для каждой базы данных кластера.

Параметры устанавливаются только при запуске экземпляра, изменения их значений передаются через WAL, на физических репликах значения должны быть не меньше (лучше равными), чем на мастере. То есть изменение этих параметров затруднительно, так как требует рестарта экземпляров мастера и реплик в правильном порядке - сначала реплик, потом мастера. Поэтому значения этих параметров стоит установить заранее.

Например, число сессий 1000, объектов в базе 300000, тогда `max_locks_per_transaction=300`, размер разделяемой памяти, выделяемый при запуске экземпляра для хранения блокировок на объекты ~48Мб. Объем выделенной памяти будет больше: ~73Мб. Изменения затрагивают память с именем `name=<anonymous>` в выводе запроса:

```
SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM pg_shmem_allocations ORDER BY size DESC;
```

Фоновые рабочие процессы

- `max_worker_processes` (по умолчанию 8) максимальное число фоновых процессов на экземпляре
  - измерение параметра требует перезапуска экземпляра
  - на физических репликах значение параметра должно быть не меньше, чем на мастере.
- `max_parallel_workers` (по умолчанию 8) максимальное число фоновых процессов для обслуживания команд
- `max_parallel_workers_per_gather` (по умолчанию 2) ограничение степени параллелизма
- `parallel_leader_participation` (по умолчанию `on`) серверный процесс будет помогать фоновым процессам

Фоновые рабочие процессы

Параметр max_worker_processes (по умолчанию 8) устанавливает максимальное число фоновых процессов на экземпляре. Измерение параметра требует перезапуска экземпляра. На физических репликах значение параметра должно быть не меньше, чем на мастере.

Фоновые процессы могут использоваться для выполнения произвольных задач: распараллеливания команд, задач посылаемых на выполнение приложением функциями расширения `pg_background`. Такие задачи могут и не нагружать ядра центральных процессоров.

Параметр `max_parallel_workers` (по умолчанию 8) `устанавливает` максимальное число фоновых процессов, которые могут использоваться для распараллеливания выполнения команд выполняемых серверными процессами.

Фоновые процессы использующиеся для распараллеливания выполнения команд выполняемых серверными процессами обычно требуют большой нагрузки на ядро процессора, поэтому количество фоновых процессов коррелирует с количеством ядер процессоров. Устанавливать больше, чем `max_worker_processes` не имеет смысла.

Параметр `max_parallel_workers_per_gather` (по умолчанию 2) `устанавливает` максимальное число фоновых процессов, которые могут использоваться серверным процессом для обслуживания одного узла `Gather` или `Gather Merge` из плана выполнения команды. Значение 0 отключает использование параллельных планов. Конвеерная обработка (набор процессов читает данные и передает их другому набору процессов) не используется, поэтому удвоения использования процессов как в Oracle Database нет.

Степень параллелизма служебных команд (`CREATE INDEX` при построении индекса типа `btree` и `VACUUM` без `FULL`) ограничивается параметром `max_parallel_maintenance_workers` (по умолчанию 2).

`Параметр parallel_leader_participation` (по умолчанию `on`) устанавливает, что серверный процесс будет выполнять ту же работу, что и фоновые процессы, а не простаивать пока работают фоновые процессы. Оптимальность зависит от плана запроса. В целом, чем больше степень параллелизма, длительность запроса, в меньшей степени объем возвращаемых запросом данных, тем значение true менее оптимально.

Параметры `max_worker_processes` и `max_parallel_workers`

- `max_worker_processes` по умолчанию 8. Максимальное число фоновых (рабочих) процессов (background workers), которые могут могут быть запущены на экземпляре
  - устанавливается только на уровне кластера
- `max_parallel_workers` по умолчанию 8. Максимальное число параллельных процессов (parallel workers)
  - Фактически ограничено значением `max_worker_processes`, но может быть установлено в большее значение (для удобства)
  - Параллельные процессы используются для обслуживания команд SQL, выполняемых серверными процессами, а также для передачи изменений в логической репликации
  - можно установить на уровне базы, роли, роли в базе
- `max_parallel_maintenance_workers` по умолчанию 2. Максимальное число рабочих процессов, которые могут запускаться одной командой CREATE INDEX или VACUUM (без FULL)

Параметры `max_worker_processes` и `max_parallel_workers`

Параллельные процессы позволяют существенно ускорить обработку большого количества строк. Оптимальное количество зависит от количества ядер центральных процессоров. Значения по умолчанию невелики и не зависят от количества ядер.

`max_worker_processes` по умолчанию 8. Максимальное число фоновых (рабочих) процессов (background workers), которые могут могут быть запущены на экземпляре. При увеличении значения сначала увеличивать на физических репликах, иначе они приостановят свою работу. Изменение значения передается через журналы. Устанавливается только на уровне кластера.

`max_parallel_workers` по умолчанию 8. Максимальное число параллельных процессов (parallel workers). Фактически ограничено значением `max_worker_processes`, но может быть установлено в большее значение (для удобства). Параллельные процессы используются для обслуживания команд SQL, выполняемых серверными процессами, а также для передачи изменений в логической репликации (параметр `max_logical_replication_workers` ограничивается вышеприведенными параметрами). Можно установить на различных уровнях, например, на уровне базы данных, роли, роли подсоединенной к конкретной базе:

```
postgres=# alter role postgres set max_
max_parallel_maintenance_workers  max_parallel_workers_per_gather
max_parallel_workers              max_stack_depth
```

`max_parallel_maintenance_workers` по умолчанию 2. Максимальное число рабочих процессов, которые могут запускаться одной командой CREATE INDEX или VACUUM (без FULL). Автовакуум не использует рабочие процессы. Количество процессов ограничивается `max_worker_processes` и `max_parallel_workers`.

Параметр `max_parallel_workers_per_gather`

- `max_parallel_workers_per_gather` (по умолчанию 2) задаёт максимальное число рабочих процессов, которые могут использоваться серверным процессом для обслуживания обычных (не maintenance) команд
- если число свободных процессов меньше расчетного, они и используются и команду обслуживает меньшее число процессов (степень параллелизма уменьшается).
- по умолчанию серверный процесс участвует в обработке данных (узлов плана ниже Gather) наравне с параллельными процессами, но обработает меньше строк, так как обслуживает последовательную часть плана и узлы Gather.
- участие серверного процесса в обработке параллельной части плана можно отключить `parallel_leader_participation=off`, но обычно не приводит к увеличению производительности

Параметр `max_parallel_workers_per_gather`

Параметр `max_parallel_workers_per_gather` по умолчанию равен 2. Максимальное число рабочих процессов, которые могут использоваться серверным процессом для обслуживания обычных (не maintenance) команд. Параллельные процессы обслуживают те операции, которые передают данные узлам Gather или Gather Merge плана выполнения команды. Операции Gather обрабатываются самим серверным процессом. Устанавливать значение больше, чем число ядер центрального процессора не стоит, так как параллельные процессы так же нагружают все ресурсы (процессор, память, диск), как и активные серверные процессы. Количество процессов рассчитывается автоматически для каждой команды SQL. Если число свободных процессов меньше расчетного, они и используются и команду обслуживает меньшее число процессов (степень параллелизма уменьшается). По умолчанию серверный процесс участвует в обработке данных (узлов плана ниже Gather) наравне с параллельными процессами, но обработает меньше строк, так как обслуживает последовательную часть плана и узлы Gather. Участие серверного процесса можно отключить `parallel_leader_participation=off` но обычно не приводит к увеличению производительности ни экземпляра, ни команды.

`min_dynamic_shared_memory` также относится к работе параллельных процессов. Рассматривать его стоит, если физической памяти много (сотни гигабайт).

Параметры системы хранения

- `random_page_cost` значение по умолчанию 4 подходит только для для одиночного HDD. Для SSD значение должно быть близко к seq_page_cost, которое по умолчанию 1. Можно установить значение 1.1
- `effective_io_concurrency` значение по умолчанию 1 подходит только для для одиночного HDD. Для нескольких HDD (RAID) устанавливается в число устройств HDD по которым выполняется распределение записи (страйпинг). Для одиночного SSD на шине SATA подходят значения от 64. Для NVMe выше 128-512
- `maintenance_io_concurrency` по умолчанию 10. Используется вакуумом и анализом. Для SSD рекомендованные значения такие же как для `effective_io_concurrency`

Параметры системы хранения

Есть набор параметров, который определяется системой хранения ("диском"). Это могут быть HDD, SSD (NVMe), их наборы устройств RAID. Характеристики для HDD и SSD существенно отличаются.

`random_page_cost` значение по умолчанию 4 подходит только для для одиночного HDD. Для SSD значение должно быть близко к `seq_page_cost`, которое по умолчанию 1. Для SSD скорость доступа к блокам последовательно и случайным образом не отличаются, с учетом физических характеристик SSD (erase size), который обычно больше 4Кб. Можно установить значение 1.1

`effective_io_concurrency` значение по умолчанию 1 подходит для одиночного HDD. Для ротационных дисков (HDD), объединенных в RAID можно установить в число дисков, по которым распределяются данные (страйпинг). Для одиночного SSD на шине SATA подходят значения 64-200. Для NVMe 500-1000. Максимальное значение 1000. Параметр учитывается планировщиком при индексном методе доступа Bitmap Heap Scan. Параметр указывает планировщику сколько операций ввода-вывода можно ожидать, что будут выполняться одновременно. Также параметр включает или отключает (значение ноль) предварительную выборку блоков. Помимо обычных уровней для параметров конфигурации, параметр можно установить в свойствах табличного пространства.

`maintenance_io_concurrency` по умолчанию 10. Используется вакуумом и анализом. Для SSD рекомендуемые значения такие же как для `effective_io_concurrency`.

Эти параметры можно установить на уровне табличных пространств:

```
alter tablespace имя set (<TAB>
EFFECTIVE_IO_CONCURRENCY    RANDOM_PAGE_COST
MAINTENANCE_IO_CONCURRENCY  SEQ_PAGE_COST
```

Параметры контрольных точек

- параметр `full_page_writes=on` (по умолчанию)
- `checkpoint_completion_target=0.9` (по умолчанию)
- `checkpoint_timeout` рекомендуется установить в значение 20-30 минут
- `max_wal_size` размер журнальных файлов (pg_wal) по достижению которого вызывается контрольная точка раньше, чем установлено параметром `checkpoint_timeout`
  - › допустимы при массовой загрузке или изменениям данных: когда блоки заполняются за секунды и позже не меняются
- статистика в столбцах `checkpoints_req` и `checkpoints_timed` представления `pg_stat_bgwriter`

Параметры контрольных точек

При настройке контрольных точек сначала нужно проверить, что `full_page_writes = on` (значение по умолчанию). Установка в значение off уменьшит объем журнальных записей и повысит производительность, но увеличит вероятность не восстановиться после сбоя экземпляра. Значение off устанавливают временно имея бэкапы (физические реплики) и возможность повторить изменения в случае сбоя экземпляра.

`checkpoint_timeout` по умолчанию `5min`. Это значение слишком маленькое и не оптимально. Рекомендуется установить в 20-30 минут.

`checkpoint_completion_target` по умолчанию 0.9, это значение оптимально.

`max_wal_size` по умолчанию 1Гб. Задает ориентировочный размер WAL-сегментов по достижению которого вызывается контрольная точка "по размеру". Выполнение контрольной точки позволяет удалять WAL-сегменты, хранящие изменения, произошедшие до начала контрольной точки. Обычно, контрольные точки выполняются "по времени" - с периодичностью, задаваемой параметром `checkpoint_timeout`. **Контрольные точки "по размеру" допустимы при массовой загрузке или изменениям данных: когда блоки заполняются за секунды и позже не меняются.**

`max_wal_size` подбирается так, чтобы контрольные точки чаще всего вызывались по времени, а не по размеру при этом вписываясь в размер файловой системы, на которую смонтирована директория pg_wal. Если в pg_wal достаточно достаточно свободного места, можно установить `max_wal_size` в большое значение. Для SSD ~1/4 (зависит от алгоритма работы контроллера SSD) размера SSD, чтобы вписаться в размер SLC-кэша SSD.

Если PGDATA и табличные пространства располагаются на медленных HDD, то ориентировочно установка `shared_buffers` может ориентировочно дать прирост производительности в ~3.5 раза по сравнению со значением по умолчанию, настройка `max_wal_size` в ~1,5 раза. Контрольные точки, выполняющиеся чаще, чем раз в ~15 минут позволяют удерживаться грязным блокам в памяти меньше времени. Из-за этого один и тот же блок будет грязниться несколько раз и несколько раз записываться на диск.

Статистика контрольных точек - столбцы `checkpoints_req` и `checkpoints_timed` представления `pg_stat_bgwriter`.

https://www.enterprisedb.com/blog/tuning-maxwalsize-postgresql

Параметры процесса фоновой записи `bgwriter`

- `bgwriter_delay` по умолчанию 200ms, уменьшают до 20-40ms
- `bgwriter_lru_multiplier` по умолчанию 2. Обычно устанавливают значение больше 4-10.
- `bgwriter_lru_maxpages` по умолчанию 100 буферов, увеличивают до 400-500
- `bgwriter_flush_after` по умолчанию 512kB
- Значения устанавливают на основе данных из представления `pg_stat_bgwriter`:

```
select * from pg_stat_bgwriter \gx
-[ RECORD 1 ]---------+---------
checkpoints_timed     | 1461
checkpoints_req       | 90
checkpoint_write_time | 3805215
checkpoint_sync_time  | 116032
buffers_checkpoint    | 130655
buffers_clean         | 250785
maxwritten_clean      | 1421
buffers_backend       | 5243535
buffers_backend_fsync | 0
buffers_alloc         | 653441
```


Параметры процесса фоновой записи `bgwriter`

`bgwriter_delay` по умолчанию 200ms. Если в разделяемом пуле не остаётся загрязнённых буферов, `bgwriter` может быть неактивен больше, чем задано. Разрешение таймера на многих системах составляет 10 мс, поэтому если задать в `bgwriter_delay` значение, не кратное 10, будет получен тот же результат, что и со следующим за ним значением кратным 10. По умолчанию установлено большое значение, которое подходит для ненагруженного кластера. При нагрузке процесс будет неэффективно работать и очисткой буферов будут заниматься `checkpointer` и серверные процессы. Небольшое значение может нагрузить одного из ядер процессоров. Если ядер много и экземпляр нагружен имеет смысл поставить в небольшое значение: 20ms.

`bgwriter_lru_multiplier` по умолчанию 2. Обычно устанавливают значение от 4-7.

`bgwriter_lru_maxpages` по умолчанию 100 буферов. За один цикл bgwriter записывает не больше, чем было записано в прошлый цикл, умноженное на `bgwriter_lru_multiplier`, но не больше чем `bgwriter_lru_maxpages`. Обычно увеличивают до 400-500.

`bgwriter_flush_after` по умолчанию 512kB. Диапазон значение от 0 (отключает flush) до 2MB.

Значения устанавливают на основе данных из представления `pg_stat_bgwriter`:

```
select * from pg_stat_bgwriter\gx
-[ RECORD 1 ]---------+------------------------------
checkpoints_timed     | 1461
checkpoints_req       | 90
checkpoint_write_time | 3805215
checkpoint_sync_time  | 116032
buffers_checkpoint    | 130655
buffers_clean         | 250785
maxwritten_clean      | 1421
buffers_backend       | 5243535
buffers_backend_fsync | 0
buffers_alloc         | 653441
```

Статистика ввода-вывода по серверным и фоновым процессам имеется в представлении `pg_stat_io`.

Практика

- Часть 1. Блокировки объектов
- Часть 2. Наблюдение за памятью серверного процесса
- Часть 3. Временные таблицы и файлы
- Часть 4. Влияние параметров конфигурации на разделяемую память
- Часть 5. Параметр `max_connections` и производительность
- Часть 6. Размер кэша буферов и освобождение буферов

Практика

Примеры практики позволяют сопоставить данные о выделенной памяти, выдаваемые утилитами операционной системы и функциями системного каталога.

Вы посмотрите:

как выделяется память под временные таблицы и в какой момент создаются файлы временных таблиц;

как изменения параметров конфигурации влияют на реальные размеры структур памяти;

насколько замедляется удаление таблиц при увеличении буферного кэша.

6

6-1

Структуры хранения

Таблицы

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

Таблицы

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

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

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

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

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

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

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

Для просмотра содержимого блока используются функции стандартного расширения pageinpect.

https://docs.tantorlabs.ru/tdb/ru/15_6/se/ddl-alter.html

Служебные столбцы

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

Служебные столбцы

При доступе к строкам таблиц в командах SQL можно использовать названия псевдостолбцов (служебных, системных, виртуальных). Их набор зависит от вида таблицы. Для обычных (heap) таблиц доступны псевдостолбцы:

`ctid` адрес физического расположения строки. Используя `ctid` планировщик может получить доступ к странице (блоку файла основного слоя) таблицы без полного сканирования всех страниц. `ctid` изменится, если строка будет физически перемещена в другой блок.

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

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

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

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

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

`xmin, cmin, xmax, cmax` хранятся в трех физических полях заголовка строки. `xmin` и `xmax` хранятся в отдельных полях. `cmin, cmax, xvac` (использовался VACUUM FULL до 9 версии PostgreSQL) в одном физическом поле. `cmin` и `cmax` интересны только в течение жизненного цикла транзакции для вставки (`cmin`) и удаления (`cmax`). `ctid` вычисляется на основе адреса строки. Физически у версии строки хранится `t_ctid` хранит адрес следующей (созданной в результате UPDATE) версии строки. Причем, это не "цепочка", связь может теряться, так как вакуум может удалить более новую версию строки раньше, чем старую (блок обработал раньше) и старая версия строки будет ссылаться на отсутствующую версию. Если версия последняя, то `t_ctid` хранит адрес этой версии. Для секционированных таблиц, если UPDATE привел к тому, что новая версия переместилась в другую секцию (значение столбца, входящего в ключ секционирования изменилось) устанавливается специальное значение. Также в процессе INSERT временно может устанавливаться "speculative insertion token" вместо адреса версии строки.

https://docs.tantorlabs.ru/tdb/ru/15_6/se/ddl-system-columns.html

Расширение pageinspect

- стандартное расширение, не требует загрузки библиотеки
- контрольная сумма:
  - рассчитывается и сохраняется в момент записи в файл грязного блока
  - проверяется при считывании блока из файла в буфер кэша буферов
  - пока блок в буфере поле контрольной суммы в заголовке блока не меняется и не проверяется

```
create extension pageinspect;
drop table if exists t;
create table t(s text);
insert into t values ('a');
select * from page_header(get_raw_page('t', 0));
    lsn   |checksum|flags|lower|upper|special|pagesize|version|prune_xid
----------+--------+-----+-----+-----+-------+--------+-------+---------
6/B6EC12C0|      0 |   0 |  28 | 8144|  8176 |   8192 |     5 |       0
vacuum full t;
select * from page_header(get_raw_page('t', 0));
    lsn   |checksum|flags|lower|upper|special|pagesize|version|prune_xid
----------+--------+-----+-----+-----+-------+--------+-------+---------
6/B6ED5458| -11261 |   0 |  28 | 8144|  8176 |   8192 |     5 |       0
```


Расширение pageinspect

Стандартное расширение. Можно устанавливать в промышленную базу данных командой:

```
create extension pageinspect;
drop table if exists t;
create table t(s text);
insert into t values ('a');
select * from page_header(get_raw_page('t', 0));
    lsn   |checksum|flags|lower|upper|special|pagesize|version|prune_xid
----------+--------+-----+-----+-----+-------+--------+-------+---------
6/B6EC12C0|      0 |   0 |  28 | 8144|  8176 |   8192 |     5 |       0
```

Почему контрольная сумма ноль?

Контрольная сумма:

1) рассчитывается и сохраняется в момент записи в файл грязного блока

2) проверяется при считывании блока из файла в буфер кэша буферов

3) пока блок в буфере поле контрольной суммы в заголовке блока не меняется и не проверяется

При создании таблицы блок в файле пустой: состоит из нулевых байтов, в том числе контрольная сумма ноль. Пока блок в буфере контрольная сумма в нем остается такой какой она была на момент считывания из файла. В памяти, если буфер не очищен и не считан из файла заново контрольная сумма прежняя - на момент считывания из файла в буфер.

После рестарта экземпляра или полного вакуумирования поле с контрольной суммой обновится:

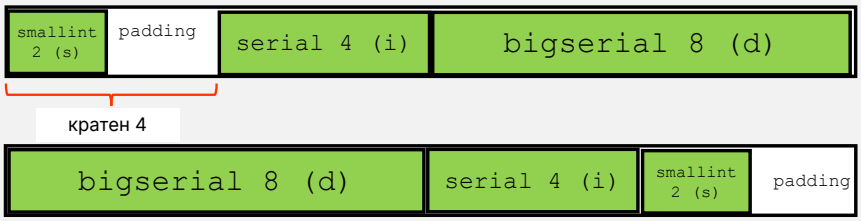
```
vacuum full t;
select * from page_header(get_raw_page('t', 0));
    lsn   |checksum|flags|lower|upper|special|pagesize|version|prune_xid
----------+--------+-----+-----+-----+-------+--------+-------+---------
6/B6ED5458| -11261 |   0 |  28 | 8144|  8176 |   8192 |     5 |       0
```

При изменениях в блоке, пока он находится в кэше буферов значение в поле контрольной суммы не будет обновляться:

```
delete from t;
   lsn   |checksum|flags|lower|upper|special|pagesize|version|prune_xid
---------+--------+-----+-----+-----+-------+--------+-------+---------
6/B6ED7008| -11261 |   0 |  28 | 8144|  8176 |   8192 |     5 |    1026
```

Padding и aligning

- выравнивание (align) - длинна поля кратна числу байт
- указано в столбцах pg_type.typalign и pg_attribute.attalign
- возможны следующие значения выравнивания:
  - s (short) 2 байта
  - i (int) 4 байта
  - d (double) 8 байт
  - c (char) без выравнивания (побайтово)
- padding - добавление неиспользуемого места, чтобы выполнить выравнивание



Padding и aligning

В столбцах `pg_type.typalign` и `pg_attribute.attalign` указано выравнивание при хранении поля для типа данных в обычных (heap) таблицах. Все типы:

```
select distinct typname, typalign from pg_type where typname not like 'pg_%' and typname not like '\_%' order by typname;
```

Все используемые в столбцах типы:

```
select distinct atttypid::regtype, attalign from pg_attribute order by attalign;
```

Выравнивание может быть:

`c (char), 1 байт`, то есть без выравнивания
`s (short), 2 байта`
`i (int), 4 байта`
`d (double) по 8 байт`
`x` для 64-битного типа `xid` в Tantor SE и SE1C.

Padding - добавление неиспользуемого места, чтобы выполнить выравнивание (aligning).

Например, создаются две таблицы с разным порядком столбцов:

```
create table t1 (c1 varchar(1), c2 bigserial , c3 date, c4 timestamp);
create table t2 (c1 bigserial , c2 timestamp, c3 date, c4 varchar(1));
```

Вставляются строки с одинаковыми значениями:

```
insert into t1 values('A', 1, now(), current_timestamp);
insert into t2 values(1, current_timestamp, now(), 'A');
```

Строки будут храниться в виде последовательности байт в HEX:

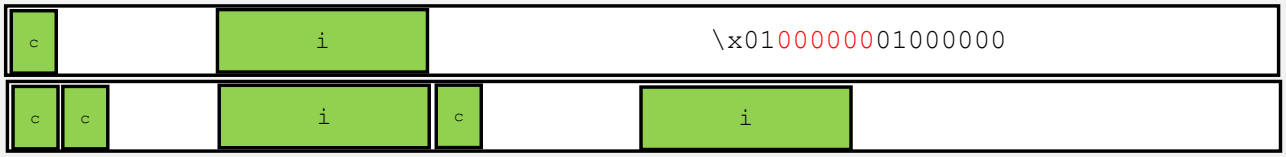
```
create extension pageinspect;
select t_data, lp_len, t_hoff from heap_page_items(get_raw_page('t1','main',0));
                          t_data                          |lp_len|t_hoff
----------------------------------------------------------+------+------
0541000000000000 0100000000000000 3623000000000000 8a31e17666c40200     56     24
select t_data, lp_len, t_hoff from heap_page_items(get_raw_page('t2','main',0));
                          t_data                          |lp_len|t_hoff
----------------------------------------------------------+------+------
0100000000000000 c93ae17666c40200 36230000  0541                      46     24
```

Разница 8 байт на версию строки. Длинна версии строки 56 и 46 байта, размер заголовка версии строки 24 байта для Tantor SE. Для PostgreSQL длинна версии строки: 64 и 54 байт, размер заголовка версии строки 32 байта (хранится в одном байте t_hoff, значение кратно 8). https://docs.tantorlabs.ru/tdb/ru/16_4/se/storage-page-layout.html

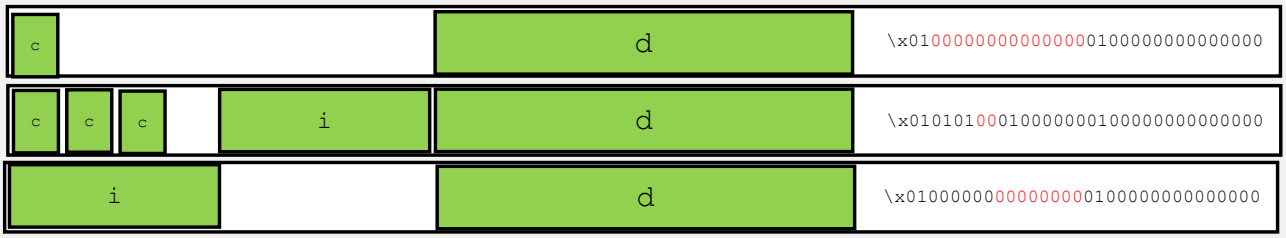
Aligning (выравнивание)

- "c" и типы переменной длинны до 127 байт не выравниваются:

- "i" - начало поля может располагаться только в 1,5,9,13... байте:



- "d" - начало поля может располагаться только в 1,9,17,25... байте:



Aligning (выравнивание)

Поля типов переменной длинны (text, numeric) длинной до 127 байт не выравниваются, от 127 байт выравниваются по pg_type.typalign (если значение "i", то по 4 байта).

У типов переменной длинны в столбце typlen таблицы pg_type значение -1:

```
select typname, typlen, typalign from pg_type where typname like '%bool%';
 typname | typlen | typalign
---------+--------+----------
 bool    |      1 | c
 _bool   |     -1 | i
(2 rows)
```

Пример для 2 картинки на слайде:

```
create table t (a boolean, b int4);
insert into t values (true, 1);
select t_data, lp_len, t_hoff from heap_page_items(get_raw_page('t','main',0));
       t_data       | lp_len | t_hoff
--------------------+--------+--------
 \x0100000001000000 |     32 |     24
```

Пример для 6 картинки на слайде:

```
drop table if exists t;
create table t (a int4, b int8);
insert into t values (1, 1);
select t_data, lp_len, t_hoff from heap_page_items(get_raw_page('t','main',0));
                t_data                | lp_len | t_hoff
--------------------------------------+--------+--------
 \x0100000000000000010000000000000000 |     40 |     24
```

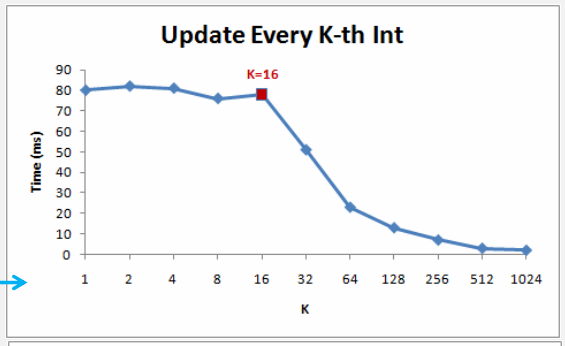
Пример для numeric:

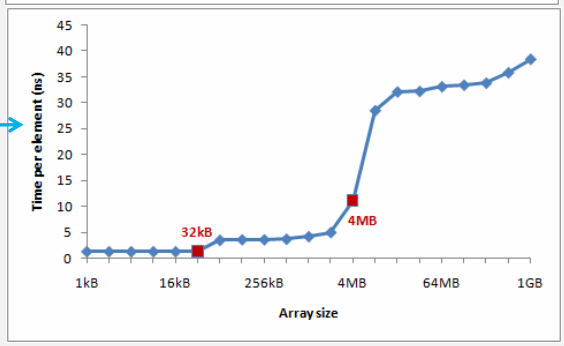
```
drop table if exists t;
create table t (a numeric, b numeric, c int4);
insert into t values (1, 1, 1);
select t_data, lp_len, t_hoff from heap_page_items(get_raw_page('t','main',0));
                t_data                | lp_len | t_hoff
--------------------------------------+--------+--------
 \x0b008001000b0080010000001000000 |     40 |     24
(1 row)
```

Значение 1 в полях numeric занимает 5 байт, но выровнено оно по 1 байту, поэтому между столбцами a и b нет padding. Поле int4 расположилось с 13 по 16 байт. Padding (неиспользуемые байты) обозначены красным цветом.

cache line

- выравнивание существенно ускоряет обработку данных, поэтому используется по возможности везде где это не приводит к сильному увеличению места хранения
- до 16*8=64 байт (K=16) время на обращение к памяти одинаково
- время на обработку данных, если они помещаются в кэше





cache line

Выравнивание существенно ускоряет обработку данных, поэтому используется по возможности везде где это не приводит к сильному увеличению места хранения.

Пример двух циклов:

```
int [] arr = new int [64 * 1024 * 1024];
for( int i = 0; i < arr.length; i += 1) arr[i] *= 3;
for( int i = 0; i < arr.length; i += 16) arr[i] *= 3;
```

В массиве целых чисел по 4 байта в первом цикле изменяется (арифметическая операция умножения на 3) все числа. Во втором цикле изменяется каждое шестнадцатое число. Время выполнения циклов будет одинаковым: второй цикл быстрее всего на 2.5%. Большая часть времени тратится на доступ к памяти. Как файлы на HDD и SDD читаются блоками (512 байт или 4Кб или другой размер), также и процессор работает с основной (не кэшами) физической памятью кусками, называемыми cache lines (линии кэша). У современных процессоров размер **cache line (LineSize)** 64 байта. Как физически процессор считывает байты или наборы байт не играет роли, важно то что время на считывание из основной памяти от 1 до 64 байта в кэш и последующая запись в основную память из кэша одинаково. Время на обработку от 1 до 64 байт находящихся в кэше в ~60 раз быстрее. В циклах 16 чисел типа int (4 байта = 32 бит) занимают 64 байта - cache line.

На графике на слайде показано время обработки массива в зависимости от шага считывания чисел int.

Если усложнить цикл и многократно возвращаться к обработке значений, то скорость обработки значений удерживающихся в кэшах будет выше до границ кэшей, а позже падать. Пример:

```
int steps = 64 * 1024 * 1024;
int lengthMod = arr.length - 1;
 for( int i = 0; i < steps; i++) arr[(i * 16) % lengthMod]++;
```

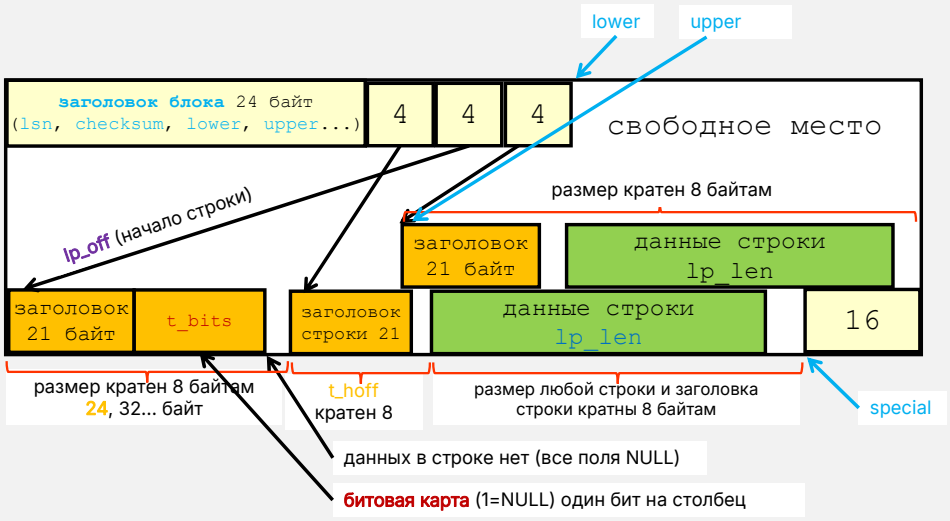
На графике на слайде пример для размеров кэшей ядра L1 = 32Кб, L2 = 4Мб.

Многие структуры PostgreSQL в разделяемой памяти выравниваются по 64 байта. У каждого ядра может быть свой кэш. Если в отображаемые кэши ядер подгружены те же адреса памяти, то при изменении значения в кэше помимо записи всей cache line в основную память, инвалидируется cache line (64 байта) в кэше всех других ядер и если ядра одновременно работают с байтами в пределах 64 байта получается замедление в ~15 раз.

https://igoro.com/archive/gallery-of-processor-cache-effects/

Cache line contention (Andres Freund): https://m.youtube.com/watch?v=dLrqQOCRFOU

Структура блока данных




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

```
select * from heap_page_items(get_raw_page('t','main',0)) limit 1;
lp|lp_off|lp_flags|lp_len|t_xmin|t_xmax|t_field3|t_ctid|t_infomask2|t_infomask|t_hoff| t_bits |t_oid
--+------+--------+------+------+------+--------+------+-----------+----------+------+--------+-----
18|   776|       1|    38| 43339|     0|     369|(0,18)|          6|      2307|    24|01111100|
```


Структура блока данных

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

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

`prune_xid` - по умолчанию 0. xmax наиболее старой неочищенной строки. Используется как подсказка (it's like a hint-bit, non-WAL-logged hint field that helps determine whether pruning will be useful. It is currently unused in index pages) процессу, который будет искать место в блоке, чтобы попытался очистить блок (First check whether there's any chance there's something to prune). Запрашивается в функции heap_page_prune_opt(..) которая вызывается функцией heapgetpage(..), которая по случаю (opt - opportunity) может очистить блок.

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

В заголовке блока есть флаги (pd_flags). Биты(флаги) могут использоваться в коде процессов как подсказки, а могут и не использоваться:

```
PD_HAS_FREE_LINES  0x0001 /* возможно есть Unused указатели в заголовке*/
PD_PAGE_FULL       0x0002 /* возможно для новых строк нет места */
PD_ALL_VISIBLE     0x0004 /* все строки видны всем транзакциям, мёртвых нет */
PD_VALID_FLAG_BITS 0x0007 /* используется как контрольная сумма для этих битов */
```

Хотя адрес (смещение до) начала строки для табличных блоков кратен 8 байт, смещение хранит адрес с точностью до байта.

```
select * from page_header(get_raw_page('t','main',0));
 lsn |checksum|flags| lower | upper | special |pagesize| version | prune_xid
-----+--------+-----+-------+-------+---------+--------+---------+----------
 0/50|       0|    0|   928 |   944 |    8176 |    8192|       5 |         0
```

В запросе можно не указывать название слоя 'main':

```
select * from page_header(get_raw_page('t', 0));
  lsn  |checksum|flags|lower| upper |special|pagesize|version|prune_xid
-------+--------+-----+-----+-------+-------+--------+-------+---------
 6/C0D8|       0 |   2 |  40 |  1264 |  8176 |   8192 |     5 |      966
```

В этом примере выставлен флаг `PD_PAGE_FULL (flags=2)` означающий, что UPDATE не нашел места в блоке и был вынужден новую версию строки вставить в другой блок. При обращении к блоку с таким флагом (даже SELECT) процесс попытается очистить место в блоке (выполнить HOT cleanup). В блоке также установлена подсказка `prune_xid=966`.

```
select ctid,* from heap_page('t',0);
 ctid  | lp_off | ctid  | state  | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------+------+------+-----+-----+--------+------
 (0,1) | 6448   | (0,1) | normal | 963c | 966c | t   |     | (0,3)  | f
 (0,2) | 4720   | (0,2) | normal | 964c | 967c | t   |     | (0,4)  | f
 (0,3) | 2992   | (0,3) | normal | 966c | 969  |     | t   | (1,1)  | f
 (0,4) | 1264   | (0,4) | normal | 967c | 968c |     | t   | (0,4)  | f
select ctid from t;
 ctid
-------
 (1,1)
```

Процесс выполнивший предыдущий select очистил блок:

```
 ctid  | lp_off| ctid  | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+-------+-------+-------+------+------+-----+-----+--------+------
 (0,1) | 0     | (0,1) | dead  |      |      |     |     |        | f
 (0,2) | 0     | (0,2) | dead  |      |      |     |     |        | f
  lsn  |checksum|flags|lower|upper|special|pagesize| version | prune_xid
-------+--------+-----+-----+-----+-------+--------+---------+----------
 6/DE30|       0|    0|   32|8176 |  8176 |   8192 |       5 |       966
```

Флаг `PD_PAGE_FULL` был снят, но prune_xid остался.

```
vacuum t;
```

Вакуум очистил ссылки на строки в заголовке блока и установил флаги `PD_HAS_FREE_LINES` и `PD_ALL_VISIBLE` (**flag=5**):

```
 ctid  | lp_off | ctid  | state  | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------+------+------+-----+-----+--------+------
 (0,1) | 0      | (0,1) | unused |      |      |     |     |        | f
  lsn  |checksum|flags|lower|upper| special | pagesize | version | prune_xid
-------+--------+-----+-----+-----+---------+----------+---------+----------
 6/0648|       0|    5|   28| 8176|    8176 |     8192 |       5 |         0
```

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

```
select * from heap_page_items(get_raw_page('t','main',0)) limit 1;
lp|lp_off|lp_flags|lp_len|t_xmin|t_xmax|t_field3|t_ctid|t_infomask2|t_infomask|t_hoff| t_bits |t_oid
--+------+--------+------+------+------+--------+------+-----------+----------+------+--------+-----
18|   776|       1|    38| 43339|    0 |    369 |(0,18)|         6 |     2307 |   24 |01111100|
```

Число строк в блоке

- не больше 291 для строк длинной ноль (все поля пустые)
- максимальное число непустых строк, которые смогут поместиться в блок: **226, 185, 156 (157), 135 (136), 119(120), 107, 97, 88, 81, 75, 70, 65,** ... так как длинна строки кратна 8 байтам
- на каждую строку к заголовку блока добавляется 4 байта
- в начале блока 24 байта заняты служебными данными
- в Tantor SE, SE1C в конце блока таблиц используется 16 байт под служебные данные
- если столбцов в таблице не больше 8, то заголовок строки 24 байта
- заголовок строки кратен 8 байт может иметь размер 24, 32,... байт
- в заголовке строки выделяется место под битовую карту пустых значений в полях строки, если столбцы могут содержать NULL
- пустые поля не занимают места в области данных
- NULL не занимают места в области данных, использование NULL эффективно с точки зрения компактности хранения
- FILLFACTOR (процент заполнения) для таблиц по умолчанию 100%

Число строк в блоке

Чтобы представить себе распределение места в странице таблицы рассмотрим пример. Таблица **с двумя столбцами uuid и boolean**. Вставим строки заполнив uuid значением `uuid_generate_v1()`, а boolean значением true. У этих типов данных возможный способ хранения PLAIN. **FILLFACTOR для таблиц PostgreSQL по умолчанию 100%**, для индексов btree 90%. Для такой таблицы в странице Tantor SE поместится 156 строк. В странице postgresql из Astralinux 1.8.1 поместится 136 строк. Если boolean оставить пустым (null), в страницу поместится 185 и 157 строк. Если оба поля NULL, то 291 и 226 строк.

Заголовок страницы размером 648 байт и 568 байт (`page_header.lower`). **Заголовок блока состоит из 24 байт плюс 4 байта на каждую строку в блоке.**

В блоке Tantor SE в конце блока используется 16 байт под хранение служебных данных. В postgresql область данных доходит до конца блока, специальной области в конце блоков в таблиц нет.

Длинна строки у Tantor SE 41 байт (lp_len), однако строка занимает 48 байт (lp_off), так как: выравниваются все поля; заголовок по 8 байт; заголовок+данные по 8 байт.

У postgresql Astralinux полная длинна строки 56 байт и lp_len=49. Заголовок строки Tantor SE - 24 байта и postgresql Astralinux - 32 байта (хранит t_infomaskpgac, t_hasmac, t_maclabel).

Пример таблицы со столбцами:

```
create table t1 (c1 varchar(1), c2 bigserial , c3 date, c4 timestamp);
create table t2 (c1 bigserial , c2 timestamp, c3 date, c4 varchar(1));
```

Число строк в блоке: Tantor SE 135 (неоптимальный порядок столбцов) и 156, в postgresql Astralinux: 120 и 136. Перестановка столбцов уменьшила накладные расходы на ~10%.

Размер заголовка блока в Tantor SE 564 байт и 648 байт, в postgresql Astralinux: 504 и 568.

Накладные расходы на хранение ~13% из них заголовок блока занимает ~7%, заголовки строк ~6%.

Максимальное число строк в 8-килобайтном блоке PostgreSQL: 291. Однако, полностью пустые строки (во всех полях NULL) почти не встречаются. Поэтому можно ориентироваться на то, что **максимальное число непустых строк, которые смогут поместиться в блок: 226, потом 185, потом 156** (у PostgreSQL на 1 строку больше из-за 16 байт в конце блока у Tantor SE)**, потом 135, потом 120.** Например, для таблиц:

`t1(c bigserial); t2(c serial); t3(c smallint);` число строк, помещающихся в блок будет одинаковым: 226. Почему? Заголовок строки 24 байта, строка выравнивается по 8 байт и область данных может иметь размер 0,8,16,24...байт.

Максимальное число строк в блоке Tantor SE может быть:

```
select string_agg(a::text,',') rows from (SELECT distinct
trunc(2038/(2*generate_series(0, 1015)+7)) a order by a desc) a;
291,226,185,156,135,119,107,97,88,81,75,70,65,61,58,55,52,49,47,45,43,41,39,38,37
,35,34,33,32,31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,
8,7,6,5,4,3,2,1
```

Число строк при максимальном размере данных в строке:

```
SELECT trunc(2038/(2*generate_series+7)) rows , max(generate_series*8) size FROM
generate_series(0, 1015) group by rows order by 1 desc;
 rows | size
------+-----
  291 |    0
  226 |    8
  185 |   16
  156 |   24
  135 |   32
  119 |   40 (полный размер строки 40+24=64 равен cache line)
  107 |   48
   97 |   56
   88 |   64
   81 |   72
   75 |   80
   70 |   88
   65 |   96
   61 |  104 (полный размер строки 128 байт кратен cache line)
 ...
   10 |  784
    9 |  872 (строка ещё поместится в 10% от размера блока для HOT cleanup)
    8 |  984 (размер превышает 10% размера блока)
    7 | 1136
    6 | 1328
    5 | 1600
    4 | 2008
    3 | 2688
    2 | 4048
    1 | 8120
(60 rows)
```

По умолчанию FILLFACTOR=100, но HOT cleanup срабатывает если осталось меньше 10% свободного места в блоке. 90% много или мало? Если в блоке помещается больше 9 строк, то освободившегося места достаточно, чтобы в него поместилась строка. Точную статистику случаев, когда при UPDATE не было найдено место в блоке и новая версия строки была вставлена в другой блок показывает `pg_stat_all_tables.n_tup_newpage_upd`.

наиболее оптимальный полный размер строки кратен 64 байтам (размер cache line). Число строк и размер области данных строки:

```
SELECT distinct trunc(2038/(16*generate_series+4)) rows, max(generate_series*64-
24) size FROM generate_series(1, 64) group by rows order by 1 desc;
rows | size
-----+-----
 101 |   40
  56 |  104
  39 |  168
  29 |  232
  24 |  296
  20 |  360
  17 |  424
  15 |  488
  13 |  552
 ...
```

Если столбцов в таблице не больше 8, то заголовок строки 24 байта. Если столбцов больше 8, то заголовок строки 32 байта. Начиная с 73 столбцов заголовок строки станет 40 байт.

Порядок следования столбцов в таблице

- первыми столбцами стоит делать d (выравниваются по 8 байт), затем i, затем s (smallint), затем c (boolean, char, uuid)
- типы переменной длинны (text, bytea, numeric и много других) имеют выравнивание i или d. Однако, **поля таких типов будут выравниваться только, если в них хранится больше 126 байт**
- у типов переменной длинны в столбце typlen таблицы pg_type отрицательное число: -1 обозначает тип varlena
- столбцы с большим количеством значений NULL стоит помещать после полей фиксированной ширины и до varlena (если в них будет храниться больше 126 байт)

Порядок следования столбцов в таблице

Первыми столбцами стоит делать d (выравниваются по 8 байт), затем i, затем s (smallint), затем c (boolean, char, uuid). Можно ли первыми поставить i, а за ними d? Можно. Например, "iid" - потерь места не будет. "id" - будет потеря места в 4 байта, так как d должен располагаться начиная с 8 байта.

Типы переменной длинны (text, bytea, numeric и много других) имеют выравнивание i или d. Однако, **поля таких типов будут выравниваться только, если в них хранится больше 126 байт**. У типов переменной длинны в столбце typlen таблицы pg_type отрицательное число: -1 обозначает тип varlena. Также есть значение "-2" (cstring и unknown), это псевдотипы (`pg_type.typtype='p'`). Псевдотипы нельзя использовать как тип данных столбца, они используются как параметры подпрограмм для передачи значений.

Столбцы с большим количеством значений NULL стоит помещать последними. NULL передается напрямую из битовой карты, расположенной перед первым полем, поэтому расположение поля со значением NULL не влияет на стоимость доступа, но размерность типа данных этих столбцов влияет как на выравнивание, так и на доступ к данным столбцов, которые располагаются после них.

```
create table t1 (c1 serial, c2 timestamp);
create table t2 (c2 timestamp , c1 serial);
insert into t1 (c2) values(current_timestamp);
insert into t2 (c2) values(current_timestamp);
select t_data, lp_len, t_hoff from heap_page_items(get_raw_page('t1','main',0)) union
select t_data, lp_len, t_hoff from heap_page_items(get_raw_page('t2','main',0));
                t_data                | lp_len | t_hoff
--------------------------------------+--------+--------
 \x0100000000000000b51ff02b6fc40200 |     40 |     24
 \x8523f02b6fc4020001000000         |     36 |     24
```

На lp_len нельзя ориентироваться, так как помимо padding выравнивается общий размер строки, находящейся в блоке. Это означает, что если lp_len не делится на 8, то в конец строки добавятся 1-7 байт (со значением ноль), чтобы размер стал кратным 8 байтам. lp_len хранится в заголовке блока и используется, чтобы определить конец последнего поля.

Числа 36 и 40 кратны 8, поэтому в данном примере будет экономия в 4 байта на строку.

Порядок следования столбцов и производительность

- столбцы, входящие в PRIMARY KEY должны идти первыми как не имеющие пустых значений и наиболее часто используемые
- дальше поля с типами данных фиксированной ширины, в которых нет пустых значений
- дальше столбцы с типами данных фиксированной ширины, в которых редко встречаются пустые значения
- столбцы, реже встречающиеся в запросах должны идти после столбцов, которые встречаются чаще
- столбцы с полями большего размера лучше делать последними
- учитывать алгоритм выноса полей в TOAST
- **если в таблице не больше 8 столбцов, то заголовок строки 24 байта**
- Если столбцов больше 8, то NULL в любом из полей строки увеличивает заголовок строки на 8 байт, но с другой стороны область данных уменьшается на размер поля

Порядок следования столбцов и производительность

Пустые поля занимают ноль байт в области данных. При большом количестве столбцов в таблице и появлении NULL хотя бы в одном поле битовая карта t_bits может скачкообразно (на 8 байт) увеличить заголовок строки с 24 до 32 байта. В битовой карте 1 столбец - 1 бит, ее размер кратен байту.

Заголовок строки без битовой карты занимает 21 байт. Физически заголовок выровнен до 24, 32, 40... байт. Чтобы заголовок не увеличился с 24 до 32 байт, битовая карта должна вписаться в 3 байта, то есть **если в таблице не больше 8 столбцов, то заголовок** 24 байта.

При превышении размера строки ~2000 байт (после сжатия полей переменной ширины, у которых стратегия позволяет их сжимать) в TOAST выносятся поля переменной ширины, начиная с самого длинного в строке, пока строка не поместится в ~2000 байт. Если не останется выносимых полей, строка превысит эту границу. Поля фиксированной ширины не могут храниться в TOAST и не могут сжиматься.

Пример: таблица с 25 столбцами типа int4. Если все поля непустые, то заголовок строки 24 байта. Если хоть одно поле пусто, заголовок строки 32 байта. При этом в обоих случаях блоке поместится 61 строка из-за того, что длинна строки без NULL 24+25*4=124 дополняется 4 байтами, чтобы быть кратным 8. Длинна строки с NULL: 32+**24***4=128 (на один столбец меньше, так как NULL занимает ноль байт в области данных)

```
 lp | lp_off | lp_len | t_ctid | t_hoff
 61 |    368 |    124 | (0,61) |     24
 61 |    368 |    128 | (0,61) |     32
```

Если первый столбец сделать int8, а второй или последующий столбец NULL, то t_hoff=32, lp_len=132 в блоке поместится 58 строк, **на 5%** меньше:

```
 58 |    288 |    132 | (0,58) |     32
```

БОльшая часть столбцов может быть в TOAST и разница будет больше.

Суммарный эффект оптимизации от неудачного порядка до лучшего (перед столбцами фиксированной ширины нет столбцов переменной ширины), если строка помещается в блок ~27%.

Причина использования выравнивания (aligning) в лучшей производительности при обработке набора байт, иначе бы padding не использовался.

https://gitlab.com/dhyannataraj/tuple-internals-presentation

Практика

- Часть 1. Карта свободного пространства
- Часть 2. Изменение порядка следования столбцов
- Часть 3. Содержимое блоков таблицы
- Часть 4. Выравнивание полей в строках таблиц
- Часть 5. Выравнивание строк в блоках таблиц
- Часть 6. Хранение пустых (NULL) значений в строках таблиц
- Часть 7. Число строк в блоке таблицы

Практика

В практике вы посмотрите:
как влияет порядок следования столбцов на место, занимаемое таблицей;
что наличие индекса замедляет вставку в таблицу на порядок;
насколько увеличивается заголовок строки, если в любом поле строки присутствует пустое значение;
как просматривать содержимое блоков таблицы.

7

Индексы

Методы доступа к строкам

```
\dA
List of access methods
   Name    | Type
-----------+-------
 bloom     | Index
 brin      | Index
 btree     | Index
 columnar  | Table
 gin       | Index
 gist      | Index
 hash      | Index
 heap      | Table
 spgist    | Index
(9 rows)
```

- два типа "методов" (способов) доступа к строкам таблиц: табличный и индексный
- Методы доступа можно добавлять расширениями:
  - `create extension pg_columnar;`
  - `create extension bloom;`
- Табличные методы доступа определяют способ хранения данных в таблицах и обычно считывают все строки
- Индексные методы обычно считывают часть блоков таблицы
- Для индексных методов (способов) нужно создать вспомогательный объект, называемый индексом
- Индексы создаются на один или несколько столбцов таблицы:

```
create table t (id int8, d date, s text);
create index t_idx on  t using btree (id int8_ops, d date_ops);
create index t_idx1 on t using btree (s text_ops);
create index if not exists t_idx2 on t (id, d);
```


Методы доступа к строкам

Есть два типа "методов" (способов) доступа к строкам таблиц: табличный и индексный.
Список доступных методов доступа: **\dA** или запрос:

```
SELECT * FROM pg_am;
```

```
   oid   |  amname  |          amhandler          | amtype
---------+----------+-----------------------------+--------
       2 | heap     | heap_tableam_handler        | t
     403 | btree    | bthandler                   | i
     405 | hash     | hashhandler                 | i
     783 | gist     | gisthandler                 | i
    2742 | gin      | ginhandler                  | i
    4000 | spgist   | spghandler                  | i
    3580 | brin     | brinhandler                 | i
```

Методы доступа можно добавлять расширениями:

```
create extension pg_columnar;
create extension bloom;
```

Расширения добавят в таблицу `pg_am` методы доступа:

```
 2425358 | columnar | columnar.columnar_handler | t
 2425512 | bloom    | blhandler                 | i
```

Табличные методы доступа определяют способ хранения данных в таблицах. Чтобы планировщик использовал индексный метод (способ) доступа, нужно создать вспомогательный объект, называемый индексом. **"Тип индекса" и "индексный метод доступа" синонимы.**

Индексы создаются на один или несколько столбцов таблицы:

```
create table t (id int8, s text);
create index t_idx on t using btree (id int8_ops) include (s) with (fillfactor = 90, deduplicate_items = off);
```

При создании индекса указывается название таблицы и столбец или столбцы ("составной индекс"), значения которых будут индексироваться. Опция INCLUDE позволяет сохранить в структуре индекса значения столбцов, выражения нельзя использовать. Для типов данных таких столбцов не нужны классы операторов. Смысл включения столбцов: чтобы планировщик использовал Index Only Scan.

Можно создать несколько одинаковых индексов, но с разными названиями.

Названия класса операторов обычно не указывается, так как есть класс по умолчанию для типа столбца. По умолчанию используется тип индекса btree.

https://docs.tantorlabs.ru/tdb/ru/16_4/se/sql-createindex.html

Класс операторов для индекса

- при создании индекса можно указать класс операторов отдельно для каждого столбца индекса

```
create table t (id int8, s text);
create index t_idx on t using btree (id int8_ops, s text_pattern_ops);
```

- если не указать класс операторов, то используется класс по умолчанию для типа столбца:

```
\dAc+ btree integer
                                  List of operator classes
  AM   | Input type | Storage type | Operator class | Default? | Operator family |  Owner
-------+------------+--------------+----------------+----------+-----------------+----------
 btree | integer    |              | int4_ops       | yes      | integer_ops     | postgres
```

- список функций, используемых семействами операторов:

```
\dAp+ btree integer_ops
List of support functions of operator families
  AM   | Operator family | Registered left type | Registered right type | Number |  Function
-------+-----------------+----------------------+-----------------------+--------+-----------------
btree  | integer_ops     | bigint               | bigint                |      1 | btint8cmp(bigint,bi..
btree  | integer_ops     | bigint               | bigint                |      2 | btint8sortsupport(in..
```


Класс операторов для индекса

При создании индекса можно указать класс операторов для каждого столбца индекса. В класс входят операторы, которые будут использоваться индексом для сравнений, сортировок, упорядочивания значений столбца. Если не указать класс операторов, то используется класс операторов по умолчанию для типа столбца и метода доступа. Несколько классов для одного и того же типа данных позволяют по-разному выполнять вычисления и упорядочивание данных.

Классы операторов для схожих типов (`int8, int4`) входят в семейство операторов. Операторы это способ записать выражение более компактно, чем с использованием функций. При создании оператора указывается название функции. Например "+" эквивалент функции `sum(a,b)`. Функции - это алгоритмы обработки данных. Алгоритмы могут быть универсальными: работать с разными типами данных. Чтобы не создавать большое число функций, создаются универсальные функции. На основе этих функций создаются "универсальные" операторы, работающие с многими типами данных. Операторы объединяются в семейства операторов. Семейства операторов позволяют создавать планы выполнения с выражениями разных типов без использования явного приведения типов.

Список функций, используемых семействами операторов: **\dAp+ btree integer_ops**

```
                          List of support functions of operator families
  AM   | Operator family | Registered left type | Registered right type | Number |  Function
-------+-----------------+----------------------+-----------------------+--------+-----------------
btree  | integer_ops     | bigint               | bigint                |      1 | btint8cmp(bigint,bi..
btree  | integer_ops     | bigint               | bigint                |      2 | btint8sortsupport(in..
btree  | integer_ops     | bigint               | bigint                |      3 | in_range(bigint,..
btree  | integer_ops     | bigint               | bigint                |      4 | btequalimage(oid)
```

Список классов по умолчанию: **\dAc+ btree integer**

```
                                 List of operator classes
  AM   | Input type | Storage type | Operator class | Default? | Operator family |  Owner
-------+------------+--------------+----------------+----------+-----------------+----------
 btree | integer    |              | int4_ops       | yes      | integer_ops     | postgres
...
```

Запрос, аналогичный команде **\dAc+** :

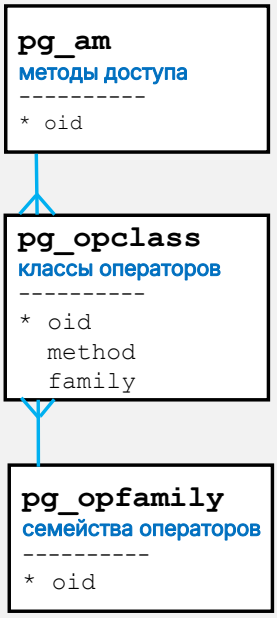
```
SELECT am.amname "AM", format_type(c.opcintype, NULL) "type",
 c.opcname "op_class", c.opcdefault "d", of.opfname "op_family"
FROM pg_opclass c JOIN pg_am am on am.oid = c.opcmethod JOIN pg_opfamily of ON of.oid = c.opcfamily
ORDER BY 1, 2, 4;
```

В класс `text_pattern_ops` включены другие другие операторы, реализующие поиск по шаблону и регулярным выражениям.

Семейства и классы операторов

- класс операторов связывает операторы, которые будут играть роли (стратегии) в методах которые использует логика индекса для упорядочивания (сравнения, сортировки, измерения расстояний, ассоциаций) данных. В классе операторов указаны названия "опорных" (supporting) функций, которые будут задействовать метод индекса при поиске или сортировке данных
- список операторов:

```
\dAo btree
  AM   | Operator family |     Operator       | Strategy | Purpose
-------+-----------------+--------------------+----------+---------
 btree | integer_ops     | <(bigint,bigint)   |        1 | search
 btree | integer_ops     | <=(bigint,bigint)  |        2 | search
 btree | integer_ops     | =(bigint,bigint)   |        3 | search
 btree | integer_ops     | >=(bigint,bigint)  |        4 | search
 btree | integer_ops     | >(bigint,bigint)   |        5 | search
...
```



Семейства и классы операторов

Класс операторов создается для метода доступа и типа данных; имеет функции, реализующие операторы, которые поддерживает метод. Класс операторов связан с типом данных и типом индекса, опционально может включаться в семейство операторов. Если не указать семейство, то класс помещается в семейство с именем класса. **Если семейства с таким именем нет, то оно создаётся**. Может возникнуть впечатление, что типы индексов (методы доступа) связаны (зависят от наличия) с семействами:

Список операторов: **`\dAo btree`**

```
  AM   | Operator family |     Operator       | Strategy | Purpose
-------+-----------------+--------------------+----------+---------
 btree | integer_ops     | <(bigint,bigint)   |        1 | search
 btree | integer_ops     | <=(bigint,bigint)  |        2 | search
 btree | integer_ops     | =(bigint,bigint)   |        3 | search
 ...
```

наоборот: **семейство операторов связано (зависит от наличия) с методом доступа. Методы доступа связаны только с классами операторов**.

Семейства операторов позволяют создавать планы выполнения с выражениями разных типов без использования явного приведения типов.

Класс операторов связывает операторы, которые будут играть роли (стратегии) в методах которые использует логика индекса для упорядочивания (сравнения, сортировки, измерения расстояний, ассоциаций и т.п.) данных. В классе операторов указаны названия "опорных" (supporting, поддерживающих) функций, которые будут задействовать метод индекса при поиске или сортировке данных.

Операторы и опорные функции задаются в классе операторов при его создании и набор не меняется (сильная связь). Если создан индекс, который использует класс операторов, то удалить класс без удаления индекса нельзя. Операторы и функции, которые требуются для индекса включаются не в семейство, а в класс операторов. В семейство операторов операторы и функции могут добавляться и удаляться из семейства (слабая связь). Когда операторы и опорные функции добавляются в семейство с помощью ALTER OPERATOR FAMILY, они не становятся частью какого-либо класса операторов в семействе, а считаются "слабосвязанными" с семейством. Эти операторы и функции совместимы с семейством, но не требуются для корректной работы индексов.

https://docs.tantorlabs.ru/tdb/ru/16_4/se/sql-createopclass.html

Списки стратегий: https://docs.tantorlabs.ru/tdb/ru/16_4/se/xindex.html

Поддерживающие функции для индекса

- Для индексации методом btree:
  - достаточно чтобы в классе операторов имелась функция (BTORDER_PROC) сравнения (compare) двух значений
  - для эффективной сортировки (ORDER BY) желательно наличие второй функции быстрой сортировки значений (BTSORTSUPPORT_PROC)
  - для возможности использования планировщиком индекса в выражениях "RANGE" оконных функций нужна третья функция (BTINRANGE_PROC)
  - для поддержки дедупликации нужна четвертая функция (BTEQUALIMAGE_PROC).
- Список функций:

```
\dAp+ btree integer_ops
List of support functions of operator families
  AM   | Operator family | Registered left type | Registered right type | Number |  Function
-------+-----------------+----------------------+-----------------------+--------+-----------------
btree  | integer_ops     | bigint               | bigint                |      1 | btint8cmp(bigint,bi..
btree  | integer_ops     | bigint               | bigint                |      2 | btint8sortsupport(in..
btree  | integer_ops     | bigint               | bigint                |      3 | in_range(bigint,..
btree  | integer_ops     | bigint               | bigint                |      4 | btequalimage(oid)
```


Поддерживающие функции для индекса

btree - наиболее часто используемый тип индекса в СУБД. Для индексации методом btree достаточно, чтобы тип данных был сравним. Для этого в классе операторов имелась поддерживающая ("опорная") функция (BTORDER_PROC), которая могла бы сравнить два значения типа данных для которых создан класс операторов. Результат функции: отрицательное, положительное значение или ноль если значения равны.

Для эффективной сортировки (ORDER BY) желательно чтобы класс операторов имел вторую (Number 2) функцию быстрой сортировки значений (BTSORTSUPPORT_PROC).

Для возможности использования планировщиком индекса в выражениях "RANGE" оконных функций нужна третья (Number 3) функция (BTINRANGE_PROC).

Для поддержки дедупликации нужна четвертая (Number 4) функция (BTEQUALIMAGE_PROC).

Список функций, используемых семействами и одноимёнными классами операторов:

```
\dAp+ btree integer_ops
                        List of support functions of operator families
  AM   |Operator family |Registered|Registered|Number|  Function
       |                | left type|right type|
-------+----------------+----------+----------+------+-----------------
btree | integer_ops     | bigint   | bigint   |    1 | btint8cmp(bigint,bigint)
btree | integer_ops     | bigint   | bigint   |    2 | btint8sortsupport(internal)
btree | integer_ops     | bigint   | bigint   |    3 | in_range(bigint,bigint,bigint,boolean,boolean)
btree | integer_ops     | bigint   | bigint   |    4 | btequalimage(oid)
...
btree | integer_ops     | smallint | integer  |    1 | btint24cmp(smallint,smallint)
btree | integer_ops     | smallint | integer  |    3 | in_range(smallint,smallint,smallint,smallint)
...
btree | text_ops        | text     | text     |    1 | btnametextcmp(text,text)
...
btree |text_pattern_ops| text     | text     |    4 | btequalimage(oid)
...
(22 rows)
```

При создании класса операторов он включается в одноимённое семейство. Если семейства с таким именем нет, оно создаётся. Поэтому **названия семейств и классов совпадают**.

Индексы для ограничений целостности

- для ограничений целостности PRIMARY KEY и UNIQUE необходим **уникальный** индекс **типа btree** со столбцами, входящими в ограничение целостности.
- остальные индексы могут использоваться для ускорения запросов ("аналитические индексы"), полнотекстового поиска
- индексы ускоряют поиск строк и замедляют добавление, изменение, удаление строк
- индексы используют место на диске, размер сопоставим с размером таблицы
- пример замены индекса другим индексом:

```
create table t3 (n int4 primary key, m int4);
Indexes:
    "t3_pkey" PRIMARY KEY, btree (n)
create unique index concurrently t3_pkey1 on t3 (m,n);
ALTER TABLE t3 DROP CONSTRAINT t3_pkey, ADD CONSTRAINT t3_pkey PRIMARY KEY USING INDEX t3_pkey1;
NOTICE:  ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index "t3_pkey1" to "t3_pkey"
Indexes:
    "t3_pkey" PRIMARY KEY, btree (m, n)
```


Индексы для ограничений целостности

Если в команде CREATE INDEX не указать тип индекса, то создаётся индекс типа btree. btree наиболее распространенный тип индекса в реляционных базах данных, работающий со многими типами данных.

Для ограничений целостности PRIMARY KEY (PK) и UNIQUE (UK) необходимы индексы типа btree. Для других ограничений целостности необязательны и создаются если: ускоряют запросы, существенно не замедляют изменение данных, используемое индексами место не критично.

При создании ограничений целостности PRIMARY KEY (PK) и UNIQUE (UK) создаются уникальные индексы типа btree. Правила использования индексов с ограничениями целостности отличаются от Oracle Database.

Например, в PostgreSQL без уникального индекса ограничения PK и UK не могут существовать:

```
ERROR:  PRIMARY KEY constraints cannot be marked NOT VALID
```

и не могут использовать неуникальные индексы:

```
alter table t3 drop constraint t3_pkey, add constraint t3_pkey primary key using index t3_pkey1;
ERROR:  "t3_pkey1" is not a unique index;
```

В Oracle Database есть включённое и отключённое состояние ограничений целостности, индекс создаётся при включении ограничения целостности, можно использовать неуникальные индексы. Такие отличия не дают преимуществ или недостатков, об отличиях полезно знать при эксплуатации и поддержке таблиц, если вы имеете опыт работы с СУБД, отличных от PostgreSQL.

В PostgreSQL только индекс типа btree поддерживает свойство UNIQUE (может быть уникальным):

```
select amname, pg_indexam_has_property(a.oid, 'can_unique') as p from pg_am a where amtype = 'i' and pg_indexam_has_property(a.oid, 'can_unique') = true order by 1;
 amname | p
--------+---
 btree  | t
```

Индексы могут использоваться для ускорения запросов ("аналитические индексы"), полнотекстового поиска. Индексы ускоряют поиск строк и замедляют добавление, изменение, удаление строк. Размер индексов сопоставим с размером таблицы.

Можно создать новый индекс и назначить его вместо прежнего, но он должен быть уникальным:

```
create table t3 (n int4 primary key, m int4 );
\d t3
                Table "public.t3"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 n      | integer |           | not null |
 m      | integer |           |          |
Indexes:
    "t3_pkey" PRIMARY KEY, btree (n)
create unique index concurrently t3_pkey1 on t3 (m,n);
ALTER TABLE t3 DROP CONSTRAINT t3_pkey, ADD CONSTRAINT t3_pkey PRIMARY KEY USING INDEX t3_pkey1;
NOTICE:  ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index "t3_pkey1" to "t3_pkey"
\d t3
                Table "public.t3"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 n      | integer |           | not null |
 m      | integer |           | not null |
Indexes:
    "t3_pkey" PRIMARY KEY, btree (m, n)
```

Прежний индекс был удалён. Столбцы ограничения целостности определяются столбцами индекса. Удалить индекс без удаления ограничения целостности PRIMARY KEY нельзя.

PRIMARY KEY отличается от UNIQUE тем, что он один на таблице и добавляет ограничения целостности NOT NULL на все столбцы. В примере на оба столбца появились not null.

При удалении PRIMARY KEY ограничения NOT NULL не удаляются, так как они могли быть добавлены отдельно.

Ограничение целостности FOREIGN KEY можно создать только на столбцы с PRIMARY KEY или UNIQUE иначе выдастся ошибка:

```
ERROR:  there is no unique constraint matching given keys for referenced table "t3"
```

При добавлении ограничения целостности:

```
alter table t3 add constraint fk foreign key (m) references t3(n) not valid;
\d t3
                Table "public.t3"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 n      | integer |           | not null |
 m      | integer |           |          |
Indexes:
    "t3_pkey" PRIMARY KEY, btree (n)
Foreign-key constraints:
    "fk" FOREIGN KEY (m) REFERENCES t3(n) NOT VALID
Referenced by:
    TABLE "t3" CONSTRAINT "fk" FOREIGN KEY (m) REFERENCES t3(n) NOT VALID
alter table t3 validate constraint fk;
```

При валидации строк запрашивается блокировка ShareRowExclusive на обе таблицы (с FOREIGN KEY и ключём на который он ссылается). В примере таблица одна (self-referencing key, самоссылающийся ключ). При добавлении других типов ограничений целостности обычно запрашивается AccessExclusive.

Нужно ли создавать индекс на столбец FOREIGN KEY? Если в таблице с FK мало строк и полное сканирование быстрее чем индексное, то не нужно. Индекс на FK создают если:

1) в мастер-таблице (там, где PK) часто обновляется значение столбца PK или удаляются строки. Эти действия нежелательны и при проектировании приложений их избегают.

2) когда используются соединения таблиц связанных PK-FK. Это используется очень часто, та как для этого FK и нужен: определяет **связь (join)** между таблицами. Например:

```
select master.m, detail.n from t3 master join t3 detail on (master.n=detail.m);
```

https://www.cybertec-postgresql.com/en/index-your-foreign-key/

Индекс типа btree

- Для числового столбца (int2, int4, int8) во внутреннем (промежуточном) блоке при 70% заполнения помещается 286 ссылок, при 100% поместится 407 ссылок
- Если в индексе указать больше одного столбца:
  - порядок индексируемых столбцов важен
  - можно представить, что индексируется результат конкатенации значений столбцов
  - первый столбце называется ведущим
  - индекс называется составным
- Индексная запись должна поместиться в 1/3 блока (2704 байта):

```
create table t (id int8, s text storage plain);
create index t_idx on t (s text_ops) include (id);
insert into t values (1, repeat('a',2700));
ERROR:  index row size 2720 exceeds btree version 4 maximum 2704 for index "t_idx"
DETAIL:  Index row references tuple (0,2) in relation "t".
HINT:  Values larger than 1/3 of a buffer page cannot be indexed.
Consider a function index of an MD5 hash of the value, or use full text indexing.
```


Индекс типа btree

Для числового столбца (int2, int4, int8) в промежуточном (внутреннем) блоке при 70% заполнения помещается 286 ссылок, при 100% поместится 407 ссылок. Число уровней индекса (level начинающийся с нуля) на численный столбец при монотонном заполнении последовательностью не превысит 5: 286^5=1913507486176 ссылок на строки. Если размер строки 18 байт, то это 32Тб, что является максимальным размером таблицы. **Для 4 уровней число строк 6690585616**.

Для поиска строки в индексе процесс считывает не меньше чем level+1 блок и возможно один или несколько листовых блоков правее. Если не используется Index Only Scan или в блоке могут содержаться (проверяется по карте видимости vm) неактуальные строки, то ещё придется считывать блок таблицы. Каждое считывание блока это поиск по Buffer Mapping Table, закрепление блока, увеличение usagecount. Поэтому добавление нового уровня (level) в индексе снижает производительность. Если перестройка индекса уменьшит число уровней, то производительность поиска по индексу (индексного доступа) увеличивается.

В промежуточных блоках индекса хранятся значение как минимум первого проиндексированного столбца. Если значения длинные, то число записей в промежуточных блоках может быть большим, число уровней будет большим и размер индекса тоже будет большим. Поэтому **нежелательно индексировать первые столбцы с длинными значениями**. Оптимизация suffix truncation может исключать значения столбцов из нелистовых блоков. Значения INCLUDE-столбцов присутствуют только в листовых блоках.

Индексная запись должна поместиться в 1/3 блока (2704 байта):

```
create table t (id int8, s text storage plain);
create index t_idx on t (s text_ops) include (id);
insert into t values (repeat('a',2700));
ERROR:  index row size 2720 exceeds btree version 4 maximum 2704 for index "t_idx"
DETAIL:  Index row references tuple (2,1) in relation "t".
HINT:  Values larger than 1/3 of a buffer page cannot be indexed.
Consider a function index of an MD5 hash of the value, or use full text indexing.
```

Функции расширения pageinspect для btree

- `bt_metap(relname)` выдаёт информацию из блока метаданных индекса. Это всегда первый блок первого файла индекса (блок номер ноль)
- `bt_page_stats(..)` и `bt_multi_page_stats(..)` номера соседних блоков слева (btpo_prev) и справа (btpo_next) на том же уровне
- `bt_page_items(..)` содержимое блока индекса

```
CREATE TABLE t(s text storage plain) with (autovacuum_enabled=off, fillfactor=40);
create index t_idx on t (s);
INSERT INTO t VALUES (repeat('a',2500));INSERT INTO t VALUES (repeat('b',2500));
INSERT INTO t VALUES (repeat('c',2500));INSERT INTO t VALUES (repeat('d',2500));
select * from bt_multi_page_stats('t_idx',1,-1);
blkno|type|live_items|dead_items|avg_item_size|pagesize|freesize|btpo_prev|btpo_next|btpo_level|btpo_flags
-----+----+----------+----------+-------------+--------+--------+---------+---------+----------+----------
   1 | l  |        2 |        0 |        2512 |   8192 |   3116 |       0 |       2 |        0 |        1
   2 | l  |        3 |        0 |        2512 |   8192 |    600 |       1 |       0 |        0 |        1
   3 | r  |        2 |        0 |        1260 |   8192 |   5620 |       0 |       0 |        1 |        2
select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data from bt_page_items('t_idx',1);
 itemoffset | ctid  | itemlen | nulls | vars | dead | htid  | tids |          substring
------------+-------+---------+-------+------+------+-------+------+---------------------------
          1 | (1,1) |    2512 | f     | t    |      |       |      | 20 27 00 00 62 62 62 62
          2 | (0,1) |    2512 | f     | t    | f    | (0,1) |      | 20 27 00 00 61 61 61 61
```


Функции расширения pageinspect для btree

Для просмотра данных об индексах используется стандартное расширение pageinspect:
`create extension if not exists pageinspect;`
Список функций расширения можно получить командой: **`\dx+ pageinspect`**
`bt_metap(relname)` выдаёт информацию из блока метаданных индекса. Это всегда первый блок первого файла индекса (блок номер ноль).
`bt_page_stats(relname, blkno)` принимает имя индекса и номер блока, выдаёт одну строку:
1) type тип блока индекса: `l` leaf **листовой**, `i` internal **внутренний** (в промежутке между root и leaf), `r` root **корневой**, d deleted leaf (свободный), D deleted internal (свободный), e ignored
2) `avg_item_size` - средний размер записи в блоке
3) `free_size` сколько байт свободно в блоке
4) `btpo_prev btpo_next` номера блоков слева и справа (siblings) на том же уровне. Ноль означает, что блок крайний на своём уровне. В листовых блоках используются для навигации при поиске по диапазону, сортировке. `btpo_next` запрашивается серверным процессом чтобы найти правильный листовой блок в случае, если блок расщепился пока процесс спускался к нему с верхнего уровня. Для проверки наличия такой ситуации используется первая запись листового или промежуточного блока (кроме самого правого на своём уровне), которая всегда хранит значение ключевых столбцов больше или равного которому в этом блоке нет (называется High key).
По `btpo_prev`, `btpo_next`, `bt_page_items().ctid` можно выполнять навигацию по дереву индекса и даже нарисовать дерево.
5) `dead_items` - число "killed" (флаг LP_DEAD у индексной записи) записей в результате быстрой очистки (simple deletion) блока индекса
6) `live_items` - число используемых записей в индексе
`bt_multi_page_stats(relname, blkno, blk_count bigint)` выдает то же самое, что и предыдущая функция, только по диапазону блоков индекса. Отрицательное число в `blk_count` означает последнюю страницу. `blkno` - номер блока, с которого выдавать данные.
`bt_page_items(relname, blkno)` выдаёт содержимое записей блока индекса в удобном для чтении виде.
https://docs.tantorlabs.ru/tdb/ru/16_4/se/pageinspect.html#PAGEINSPECT-B-TREE-FUNCS

Дедупликация появилась в PostgreSQL 13 версии и использует для хранения информации поля, которые без дедупликации предназначены для других целей. Это позволило радикально не менять структуру индекса и не требовать перестройки индексов при миграции на новую версию PostgreSQL.

Если блок листовой и tids не пусто, то значит используется дедупликация и tids хранят в отсортированном (по блоку и слоту) формате ссылки ctid на строки в таблице. В htid сохраняется первый tid из tids. ctid при этом хранит не ссылки на блоки, а служебные данные о tids. Например, ctid промежуточных блоков будет хранть номер блока в индексе, ссылающийся на нижестоящий уровень, а вторая часть ctid - число элементов в tids.

Если tids пуст, то дедупликация в этой записи индекса не используется. В ctid листового блока хранится ссылка на проиндексированную строку таблицы. Поле htid (heap tuple id) хранит то же самое значение, что и ctid. Почему дублируется значение в ctid и htid? Алгоритм работы btree оптимизирован для работы в условиях минимума блокировок и для минимизации расщеплений блоков. В процессе чтения блоков индекса другие процессы могут менять его структуру. При навигации по блокам нет единой картины ("целостности по чтению") и дополнительные поля используются для выявления противоречий.

В промежуточных блоках в `ctid` хранится ссылка на блок индекса, а второе число единица или ноль. **Если единица, то в поле data хранится минимальное значение, которое присутствует в дочернем листовом блоке.** Если ноль `ctid=(N,0)`, то поле `data` пусто (трактуется как "минус бесконечность", то есть граница неизвестна) и эта ссылка ведёт на самый левый дочерний блок.

В индексе используются оптимизация suffix truncation и усечение проиндексированных столбцов в поле data её следствие. Из-за этой оптимизации индекс btree используемый в PostgreSQL можно называть "Simple Prefix B-Tree". Простое (Simple) потому, что усекаются целые поля(whole "attribute" truncation). Для индекса по одному столбцу остаётся пустота, трактуемая как минус бесконечность.

Поле data текущей и следующей (itemoffset+1) записи задаёт диапазон, в который должно попасть значение, по которому выполняется поиск в индексе:

```
select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data from bt_page_items('t_idx',1);
 itemoffset | ctid  | itemlen | nulls | vars | dead | htid  | tids |        substring
------------+-------+---------+-------+------+------+-------+------+-------------------------
          1 | (1,1) |    2512 | f     | t    |      |       |      | 20 27 00 00 62 62 62 62
          2 | (0,1) |    2512 | f     | t    | f    | (0,1) |      | 20 27 00 00 61 61 61 61
```

Значение data текущей записи включено в диапазон и является первым значением в том блоке, на который указывает запись. Значение data следующей записи не включено в диапазон. Следующая запись является первой записью уже своего диапазона.

Размеры индексов можно получить командой **`\di+ *имя_индекса*`**.

Для более детальной выборки можно использовать как шаблон запрос:

```
select i.relname "table", indexrelname "index",
pg_INDEXES_size(relid) "indexes_size",
pg_RELATION_size(relid) "table_size ",
pg_TOTAL_RELATION_size(relid) "total",
pg_RELATION_size(indexrelid) "index_size",
reltuples::bigint "rows",
ii.indexdef ddl
from pg_stat_all_indexes i join pg_class c on (i.relid = c.oid)
join pg_indexes ii on (i.indexrelname = ii.indexname)
where i.schemaname not like 'pg_%'  -- не выводить служебные объекты
order by pg_INDEXES_size(relid) desc, pg_RELATION_size(indexrelid) desc;
```

```
table| index     |indexes_size|table_size |   total   |index_size|rows     |  ddl
-----+-----------+------------+-----------+-----------+----------+---------+----------------------
test |test_id_idx|  6876692480|11097309184|17977090048|6876692480|299916064|CREATE INDEX test_id_idx
ON public.test USING btree (id)
t4   |t4_pkey    |      679936|    1187840|    1892352|    679936|       -1|CREATE INDEX t4_pkey
ON public.t4 USING btree (id) WITH (fillfactor='100')
```

В столбце indexdef представления pg_indexes хранится команда создания индекса.

Если rows=-1 это означает, что на таблице не собрана статистика. Можно собрать статистику командой ANALYZE.

https://raw.githubusercontent.com/postgres/postgres/refs/heads/master/src/backend/access/nbtree/README

Индексы с дедупликацией в листовых блоках

- если индекс не уникальный, то при большом количестве дубликатов они хранятся компактно за счет дедупликации в листовых блоках
- не все типы данных поддерживают дедупликацию
  - не поддерживают: `numeric, jsonb, float4, float8`
  - не поддерживают: массивы, составные, диапазонные типы
  - индексы INCLUDE не поддерживают дедупликацию
- значения проиндексированных столбцов хранятся в записи индекса, а ссылки на строки таблицы хранится в d в столбце tid (tuple id, идентификатор строк таблицы) в виде отсортированного массива ctid
- пример двух записей с дедупликацией и одной без дедупликации:

```
select itemoffset, ctid, itemlen, nulls, vars, dead, htid, data, tids[0:3] from bt_page_items('td_idx',1);
 itemoffset|  ctid   |itemlen|nulls|vars|dead|htid   |          data           |         tids
-----------+---------+-------+-----+----+----+-------+-------------------------+-----------------------
         1 |(16,8414)|  1352 | f   | f  | f  |(0,1)  |01 00 00 00 00 00 00 00| {"(0,1)","(0,2)","(0,3)"
         2 |(16,8377)|  1128 | f   | f  | f  |(10,13)|01 00 00 00 00 00 00 00| {"(10,13)","(10,14)","(1
         3 |(19,9)   |    16 | f   | f  | f  |(19,9) |01 00 00 00 00 00 00 00|
```


Индексы с дедупликацией в листовых блоках

Если индекс не уникальный, то при большом количестве дубликатов они хранятся компактно за счет дедупликации в листовых блоках. Не все типы данных поддерживают дедупликацию. Для поддержки дедупликации в классе операторов для типа данных, который индексируется, должна быть определена функция номер 4 (BTEQUALIMAGE_PROC).

Проверить поддерживается ли дедупликация в индексе можно запросом:

```
select * from bt_metap('t_idx');
magic|version|root|level|fastroot|fastlevel|last_cleanup_num_delpages|last_cleanup_num_tuples|allequalimage
-----+-------+----+-----+--------+---------+------------------------+-----------------------+------------
340322|     4 |  3 |   1 |      3 |       1 |                      0 |                    -1 | t
```

Если allequalimage=t то поддерживается. Дедупликация появилась в 13 версии PostgreSQL.

Этот же запрос показывает число уровней индекса: level=1. Нумерация уровней начинается с нуля. Поля magic и version используются для проверки того, что объект является индексом btree поддерживаемой версии. Начиная с PostgreSQL 12 версии используется 4 версия индексов, минимально поддерживаемая в PostgreSQL 17 версии version=2. Если версия меньше 4, то индекс может использоваться, но новшества не поддерживаются. Чтобы новшества поддерживались достаточно перестроить индекс (REINDEX). "Магическое" число 340322 (0x0531162). Значение числа выбрано случайно, должно находиться в нулевом блоке по "правильному" смещению.

Значения проиндексированных столбцов хранятся в записи индекса, а ссылки на строки таблицы хранится в столбце tids (tuple ids, идентификаторы строк таблицы) в виде отсортированного массива значений типа ctid.

```
drop table td; create table td(id serial) with (autovacuum_enabled=off); create index td_idx on td (id);
insert into td select 1 from generate_series(1, 408);
select itemoffset, ctid, itemlen, nulls, vars, dead, htid, data, tids[0:3] from bt_page_items('td_idx',1);
select * from bt_multi_page_stats('td_idx',1,-1);
itemoffset|  ctid   |itemlen|nulls|vars|dead|htid   |          data           |         tids
----------+---------+-------+-----+----+----+-------+-------------------------+-----------------------
        1 |(16,8414)|  1352 | f   | f  | f  |(0,1)  |01 00 00 00 00 00 00 00| {"(0,1)","(0,2)","(0,3)"
        2 |(16,8377)|  1128 | f   | f  | f  |(10,13)|01 00 00 00 00 00 00 00| {"(10,13)","(10,14)","(1
        3 |(19,9)   |    16 | f   | f  | f  |(19,9) |01 00 00 00 00 00 00 00|
```

В строке itemoffset=3 нет дедупликации (tids пуст) и на строку таблицы указывает ctid=(19,9).

В остальных двух строках на строки таблицы указывает tids.

Дедупликация выполняется когда в блоке нет места или превышается fillfactor.

https://docs.tantorlabs.ru/tdb/ru/16_4/se/btree-implementation.html#BTREE-DEDUPLICATION

Проверка поддерживается ли дедупликация

- дедупликация не поддерживается с типами данных: `numeric, jsonb, float4, float8`, массивами, составными, диапазонными типами
- индексы со столбцами INCLUDE не поддерживают дедупликацию
- примеры запроса для проверки поддерживается ли дедупликация:

```
create table td(n timestamp, n1 date, n2 integer, n3 char, n4 text, n5 varchar);
create index td_idx on td (n,n1,n2,n3,n4,n5);
select allequalimage from bt_metap('td_idx');
 allequalimage
---------------
 t
create index td1_idx on td (n) include (n1);
select allequalimage from bt_metap('td1_idx');
 allequalimage
---------------
 f
create table td(id int8[]);
create index td_idx on td (id);
select allequalimage from bt_metap('td_idx');
 allequalimage
---------------
 f
```


Проверка поддерживается ли дедупликация

Проверить поддерживается ли дедупликация в индексе можно запросом:

```
create table td(n timestamp, n1 date, n2 integer, n3 char, n4 text, n5 varchar);
create index td_idx on td (n,n1,n2,n3,n4,n5);
select allequalimage from bt_metap('td_idx');
 allequalimage
---------------
 t
```

Если `allequalimage=t` то дедупликация поддерживается. Составной индекс поддерживает дедупликацию, если типы ключевые данных ее поддерживают.

```
create index td1_idx on td (n) include (n1);
select allequalimage from bt_metap('td1_idx');
 allequalimage
---------------
 f
```

Индексы со столбцами INCLUDE не поддерживают дедупликацию даже если типы данных поддерживают.

```
create table td(id int8[]);
create index td_idx on td (id);
select allequalimage from bt_metap('td_idx');
 allequalimage
---------------
 f
```

Параметры создания индекса и их влияние на производительность

- по умолчанию индекс строиться в возрастающем порядке
- при создании индекса можно указать обратный порядок: DESC
- правые блоки оптимизированы для вставок
- по умолчанию пустые значения сохраняются в правых блоках
- вставки строк с NULL замедлятся при использовании NULLS FIRST
- вакуум читает все блоки индекса в физическом порядке от первого до последнего блока
- включение (include) в индекс значений столбцов не включая их в ключевые значения увеличивает размер индекса

```
create unique index concurrently if not exists t1_idx1 ON t1 using btree
(c2 desc nulls first, upper(c1) ) include (c3,c4) with (fillfactor=100,
deduplicate_items=off) WHERE c2>0;
```


Параметры создания индекса и их влияние на производительность

Листья дерева повторяют значения строк оригинальной таблицы. Они представлены в порядке возрастания или убывания.

По умолчанию индекс строится в возрастающем порядке, то есть слева меньшие значения, справа большие. Листовые блоки и блоки каждого уровня имеют указатели на соседние блоки в обе стороны.

При создании индекса можно указать обратный порядок: DESC. Не стоит это делать для индексов заполняемых возрастающей последовательностью. Свойство ASC и DESC при создании индекса не влияет на эффективность использования индекса планировщиком (например, ORDER BY ASC или DESC). Это свойство влияет на заполнение индекса: **правые блоки в индексе отличаются от остальных тем, что оптимизированы для вставок**. Желательно, чтобы вставки выполнялись преимущественно в правый листовой блок индекса. "Правые блоки" всегда остаются "правыми" в структуре индекса независимо от ASC или DESC.

По умолчанию пустые значения сохраняются справа, в "правых блоках". Это можно переопределить указав NULLS FIRST. При переопределении обычно исходят из того что выдавать по умолчанию первым при сортировке. Однако, использование NULLS FIRST может повлиять на производительность: если при вставке строк в таблицу в индекс вставляется NULL (при вставке строки в таблицу значение индексированного столбца не задаётся, а обновляется позже и обновления распределены по времени, а не массовые), то **оптимизация fastpath перестает работать, так как NULL будут в самом левом листовом блоке, а fastpath работает только с правым**. Вставки строк с NULL замедлятся при использовании NULLS FIRST.

"fastpath" оптимизация созвучна "fastroot", но это разные оптимизации.

Можно индексировать не только столбцы, но и выражения. Это полезная опция, но не стоит создавать большое количество индексов.

Можно включать в индекс значения столбцов, не включая их в ключевые значения. Выражения включать нельзя, только столбцы. Опция include увеличивает размер индекса. Столбцы включают, чтобы сделать индекс "покрывающим" запросы - чтобы использовался Index Only Scan, а включить столбец в ключевые нельзя, так как тип данных столбца не поддерживает класс операторов индекса.

```
create unique index concurrently if not exists t1_idx1 ON t1 using btree (c2
desc nulls first, upper(c1) ) include (c3,c4) with (fillfactor=100,
deduplicate_items=off) WHERE c2>0;
```

Частичные (partial) индексы

- создаются по части строк таблицы
- предикат WHERE указывается при создании индекса и определяет индексируемые строки
- полезны тем, что позволяют избежать индексирования наиболее часто встречающихся значений
- частичный индекс может быть уникальным
- размер частичного индекса обычно меньше
- пример создания частичного индекса:

```
create unique index t1_idx1 ON t1 (c2 desc nulls first, upper(c1))
include (c3,c4) WHERE c2>0;
```


Частичные (partial) индексы

Частичные (partial) индексы создаются по части строк таблицы. Часть строк определяется предикатом WHERE, который указывается при создании индекса и делает индекс частичным.

Размер индекса может быть существенно уменьшен и вакуумирование будет проходить быстрее, так как вакуумирование сканирует все блоки индекса. Можно создавать частичные (partial) индексы. Это полезно, если приложение не работает с непроиндексированными строками. При создании индекса можно указать условие WHERE. Размер индекса может быть существенно уменьшен и вакуумирование будет проходить быстрее, так как вакуумирование сканирует все блоки индекса.

Частичные индексы полезны тем, что позволяют избежать индексирования наиболее часто встречающихся значений. Наиболее часто встречающееся значение - это значение, которое содержится в значительном проценте всех строк таблицы. При поиске наиболее часто встречающихся значений индекс всё равно не будет использоваться, так как более эффективным будет сканирование всех строк таблицы. Индексировать строки с наиболее часто встречающимися значениями нет смысла. Исключив такие строки из индекса, можно уменьшить размер индекса, что ускорит вакуумирование таблицы. Также ускоряется внесение изменений в строки таблицы, если индекс не затрагивается.

Вторая причина, по которой используется частичный индекс это когда отсутствуют обращения к части строк таблицы, а если обращения и присутствуют, то используется не индексный доступ, а полное сканирование таблицы.

Частичный индекс может быть уникальным.

Создавать большое число частичных индексов, которые индексируют разные строки не стоит. Чем больше индексов на таблице, тем ниже производительность команд, изменяющих данные; автовакуума; вероятность использования быстрого пути блокировок уменьшается.

https://docs.tantorlabs.ru/tdb/ru/16_6/se/indexes-partial.html

Эволюция индексов: создание, удаление, перестройка

- команды `create/drop/reindex index имя_индекса` устанавливают блокировку SHARE, не совместимую с внесением изменений в строки таблицы
- эти команды могут выполняться одновременно, они совместимы сами с собой, при этом с concurrently не совместимы
- автовакуум не совместим ни с `concurrently`, ни без
- для временных индексов на временные таблицы не надо использовать concurrently, так как блокировок на временные объекты нет
- `create/reindex concurrently` сканирует таблицу два раза, без `concurrently` один раз
- `concurrently` позволяет выполняться командам `SELECT, WITH, INSERT, UPDATE, DELETE, MERGE` и позволяет использовать быстрый путь (fastpath) блокирования объектов (таблиц, индексов, секций)

Эволюция индексов: создание, удаление, перестройка

Создание, удаление, перестройка индекса без указания CONCURRENTLY:

```
create index название..;
drop index имя_индекса;
reindex index имя_индекса;
```

устанавливают блокировку SHARE, не совместимую с внесением изменений в строки таблицы. Блокировка SHARE позволяет работать только командам:

1) SELECT и любому запросу, который только читает таблицу (то есть устанавливает блокировку ACCESS SHARE)

2) SELECT FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE, FOR KEY SHARE (устанавливают блокировку ROW SHARE)

3) CREATE/DROP/REINDEX INDEX (**без** CONCURRENTLY). **Можно одновременно создавать, удалять, перестраивать несколько индексов на одной таблице**, так как блокировка SHARE совместима с самой собой. CONCURRENTLY не совместим с SHARE.

"Не совместим" означает, что либо команда будет ждать, либо сразу выдаст ошибку, либо выдаст ошибку после таймаута, заданного параметром `lock_timeout`.

Для временных индексов на временные таблицы не надо использовать CONCURRENTLY, так как блокировок на временные объекты нет, к ним имеет доступ только один процесс, даже параллельные процессы не имеют доступа.

`create index` **`concurrently`** `название..;` устанавливает блокировку SHARE UPDATE EXCLUSIVE, которая позволяет выполняться командам SELECT, WITH, INSERT, UPDATE, DELETE, MERGE и позволяет использовать быстрый путь блокирования объектов процессами (fastpath).

Блокировку SHARE UPDATE EXCLUSIVE устанавливают также команды DROP INDEX CONCURRENTLY, REINDEX CONCURRENTLY, а также VACUUM (без FULL), ANALYZE, CREATE STATISTICS, COMMENT ON, некоторые виды ALTER INDEX и ALTER TABLE, автовакуум и автоанализ. Эти команды не могут одновременно работать с одной таблицей. Автовакуум попускает таблицы, если не может немедленно получить блокировку. **Автовакуум несовместим с созданием, удалением, пересозданием индексов.**

CONCURRENTLY имеет существенный недостаток. Без CONCURRENTLY таблица сканируется один раз, с CONCURRENTLY таблица сканируется два раза и используются три транзакции.

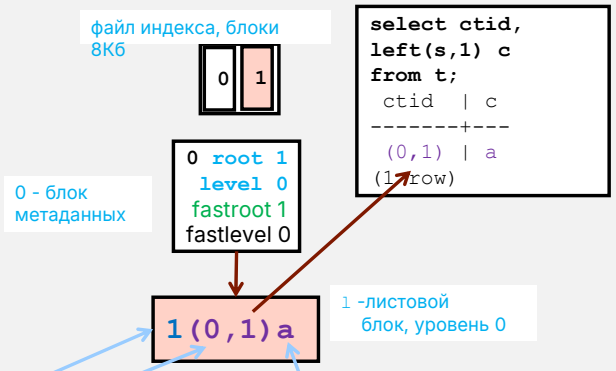
CONCURRENTLY дожидается завершения всех существующих транзакций, которые потенциально могут модифицировать и использовать индекс. В первой транзакции создаётся определение индекса, оно добавляется в системный каталог как нерабочий (invalid). Дальше процесс ждёт завершения всех транзакций, которые модифицировали таблицу во время первой транзакции. Начинается вторая транзакция, в ней таблица сканируется и строится структура индекса. Дальше процесс ждёт завершения всех транзакций, которые модифицировали таблицу во время второй транзакции. Начинается третья транзакция, в ней таблица сканируется ещё раз и структура индекса обновляется. Процесс дожидается завершения всех транзакций, получивших снимок перед второй транзакцией.

Если при сканировании таблицы возникает проблема, например взаимоблокировка или нарушение уникальности в уникальном индексе, команда CREATE INDEX CONCURRENTLY завершится с ошибкой и оставит после себя индекс в состоянии invalid.

Как устранить последствия неудачно выполнившейся команды? Либо выполнить команду REINDEX INDEX CONCURRENTLY, либо удалить индекс командой DROP INDEX или DROP INDEX CONCURRENTLY. После удаления индекса можно повторить команду или создать индекс без CONCURRENTLY.

Структура индекса типа btree

- типы блоков: метаданных, корневой, внутренние, листовые
- путь от корня до листовых блоков одинаковой глубины (балансирован)
- в листовых блоках ссылки на строки таблицы (блок, строка в блоке)
- в не-листовых блоках ссылки на блоки индекса (блок, строка в блоке)




```
create table t(s text storage plain) with (fillfactor=10);
create index t_idx on t (s) with (fillfactor=10);
insert into t values (repeat('a',2500));
select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, left(data,24) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',1);
 itemoffset| ctid   | itemlen | nulls | vars | dead | htid   | tids |              data          | c
 ----------+-------+---------+-------+------+------+-------+------+--------------------------+--
         1 | (0,1) |    2512 | f     | t    | f    | (0,1) |      | 20 27 00 00 61 61 61 61 | a
```


Структура индекса типа btree

Чтобы представить себе структуру индекса типа btree создадим таблицу и индекс:

```
drop table t;
create table t(s text storage plain) with (autovacuum_enabled=off, fillfactor=10);
create index t_idx on t (s) with (fillfactor=10, deduplicate_items = off);
```

Был создан файл индекса с одним блоком. Первый (0) блок индекса содержит метаданные:

```
select * from bt_metap('t_idx');
magic|version|root|level|fastroot|fastlevel|last_cleanup_num_delpages|last_cleanup_num_tuples|allequalimage
-----+-------+----+-----+--------+---------+------------------------+----------------------+------------
340322|    4 |  0 |   0 |      0 |       0 |                      0 |                   -1 | t
```

1) level - число уровней в дереве индекса. Уровни нумеруются с нуля и с листовых блоков потому, что индекс растёт снизу вверх и новый уровень добавляется выше корневого блока

2) root - номер корневого блока. Корневой блок может меняться при увеличении числа уровней индекса

При вставке строки в индекс добавляется блок номер 1. Он формально листовой (type=1), ни корневого ни внутренних блоков нет:

```
insert into t values (repeat('a',2500));
select * from bt_page_stats('t_idx',1);
blkno|type|live_items|dead_items|avg_item_size|pagesize|freesize|btpo_prev|btpo_next|btpo_level|btpo_flags
-----+----+----------+----------+-------------+--------+--------+---------+---------+----------+----------
    1 | l  |        1 |        0 |        2512 |   8192 |   5632 |       0 |       0 |        0 |        3
select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, left(data,24) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',1);
 itemoffset | ctid   | itemlen | nulls | vars | dead | htid   | tids |              data          | c
 -----------+-------+---------+-------+------+------+-------+------+--------------------------+--
          1 | (0,1) |    2512 | f     | t    | f    | (0,1) |      | 20 27 00 00 61 61 61 61 | a
```

У буквы "a" шестнадцатеричный код "61".

Ссылка на блок номер 1 вставляется в поля root и fastroot блока метаданных:

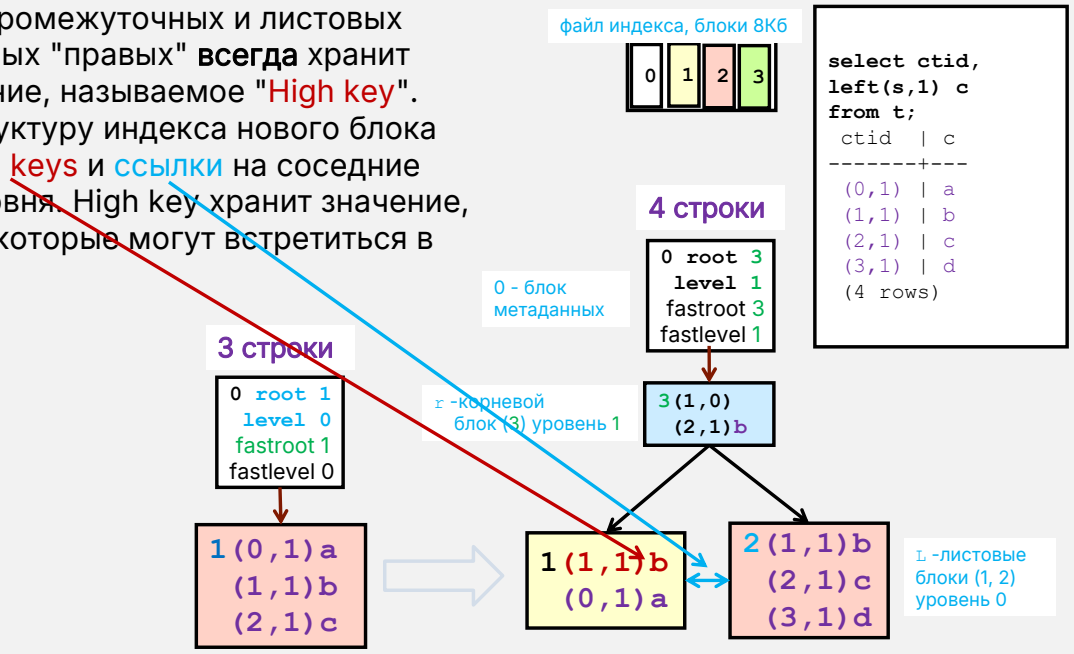
```
magic|version|root|level|fastroot|fastlevel|last_cleanup_num_delpages|last_cleanup_num_tuples|allequalimage
-----+-------+----+-----+--------+---------+------------------------+----------------------+------------
340322|    4 |  1 |   0 |      1 |       0 |                      0 |                   -1 | t
```

Число уровней пока 0. Длинна строки `itemlen=2512` байт.

Используя itemlen можно оценить стоит ли индексировать этот столбец. В примере длинна значительная: строка размером 2500 байт и индекс будет больше чем таблица.

High Key в структуре индекса

- первая строка в промежуточных и листовых блоках кроме самых "правых" **всегда** хранит служебное значение, называемое "High key".
- при вставке в структуру индекса нового блока обновляются High keys и ссылки на соседние блоки того же уровня. High key хранит значение, большее, чем те, которые могут встретиться в этом блоке.



High key в структуре индекса

Вторая и третья строки поместятся в тот же блок:

```
 itemoffset | ctid  | itemlen | nulls | vars | dead | htid  | tids |             data
------------+-------+---------+-------+------+------+-------+------+---------------------------
          1 | (0,1) |    2512 | f     | t    | f    | (0,1) |      | 20 27 00 00 61 61 61 61
          2 | (1,1) |    2512 | f     | t    | f    | (1,1) |      | 20 27 00 00 62 62 62 62
          3 | (2,1) |    2512 | f     | t    | f    | (2,1) |      | 20 27 00 00 63 63 63 63
```

В поле ctid индексной записи указываются ссылки на адрес проиндексированной строки в файлах таблицы: (0,1) (1,1) (2,1) - строки со значениями "a..", "b..", "c.." в 0, 1 и 2 (номер блока в слое main таблицы) блоках таблицы. В блоках **таблицы** размещено по одной строке длинна строки в блоках **таблицы** 2504+24 (заголовок строки в таблице)=2528 байта:

```
select pg_column_size(s) from t limit 1;
2504
```

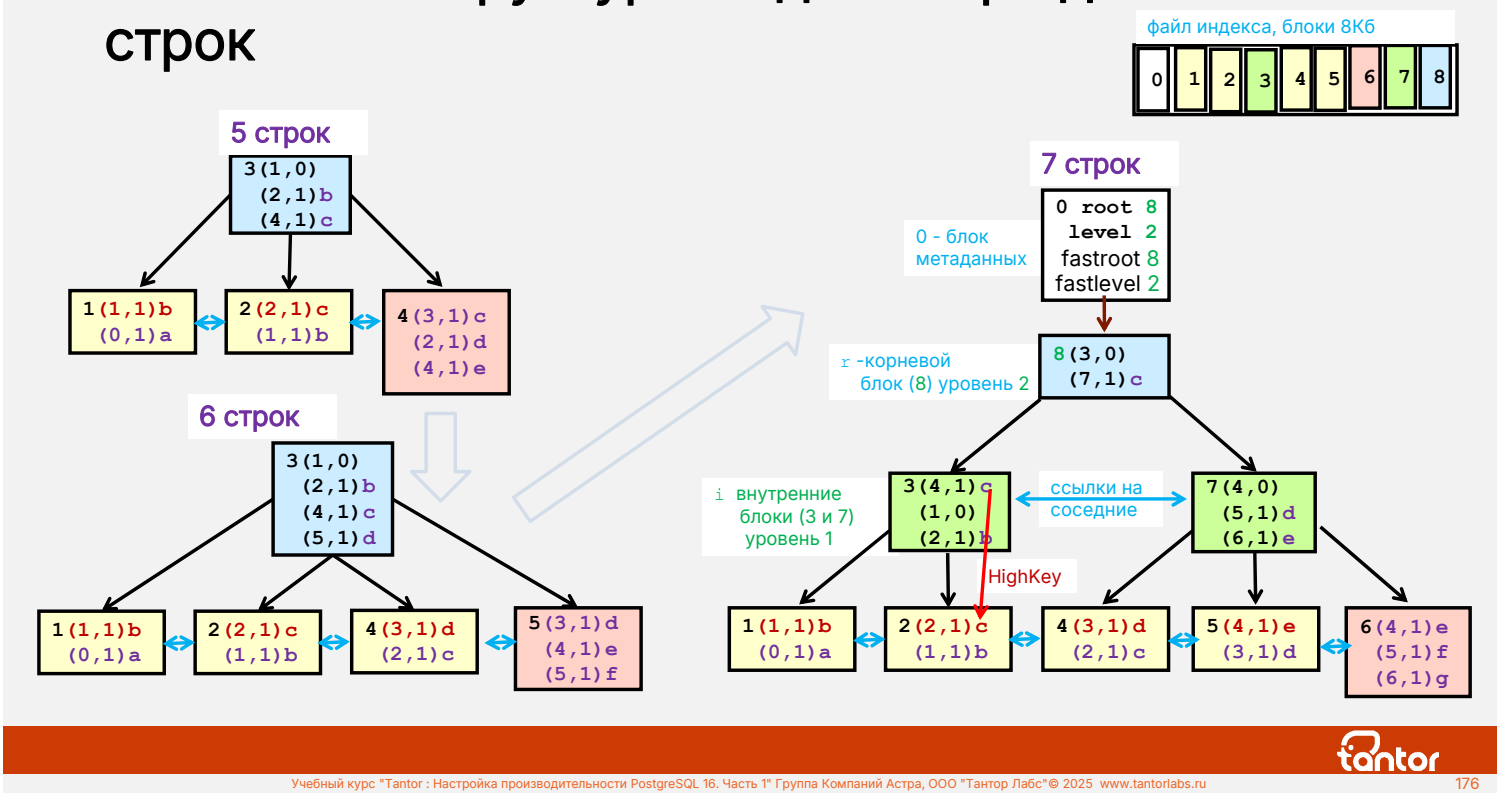
После вставки четвёртой строки в дереве индекса будет три блока (плюс блок метаданных): 1,2,3. Блоки 1 и 2 будут листовыми, а 3 корневым. Ссылки на строки перераспределятся: в первом листовом блоке останутся ссылки на a:

```
 itemoffset | ctid  | itemlen | nulls | vars | dead | htid  | tids |             data
------------+-------+---------+-------+------+------+-------+------+---------------------------
          1 | (1,1) |    2512 | f     | t    |      |       |      | 20 27 00 00 62 62 62 62
          2 | (0,1) |    2512 | f     | t    | f    | (0,1) |      | 20 27 00 00 61 61 61 61
```

во втором листовом блоке ссылки на строки b, c, d:

```
 itemoffset | ctid  | itemlen | nulls | vars | dead | htid  | tids |             data
------------+-------+---------+-------+------+------+-------+------+---------------------------
          1 | (1,1) |    2512 | f     | t    | f    | (1,1) |      | 20 27 00 00 62 62 62 62
          2 | (2,1) |    2512 | f     | t    | f    | (2,1) |      | 20 27 00 00 63 63 63 63
          3 | (3,1) |    2512 | f     | t    | f    | (3,1) |      | 20 27 00 00 64 64 64 64
```

Первая строка (itemoffset=1) в блоках **кроме самых "правых" всегда** хранит служебное значение, называемое "High key". Корневой является "правым". При вставке в структуру индекса нового блока обновляются High keys и ссылки на соседние блоки того же уровня. High key хранит значение, большее, чем те, которые могут встретиться в этом блоке. High key всегда проверяется при поиске по индексу. Зачем? В процессе спуска с предыдущего уровня до листового другой процесс мог уже расщепить блок, на который спускаются и перераспределить ссылки на строки таблиц, а это значит, что искомое значение находится в блоке (или даже блоках если было несколько расщеплений) правее того блока, на который спустились. Если значение High key блока, на который перешли по самой правой ссылке **отличается** от значения HighKey блока откуда спустились, то нужно двигаться вправо по листовому уровню и проверять на всякий случай нет ли там искомых значений.



Изменение структуры индекса при добавлении строк

Индекс растёт снизу вверх. В листовом блоке не хватает места, он "делится": добавляется пустой блок, строки перераспределяются, ссылка на новый блок вставляется в вышестоящий блок. Если в вышестоящем блоке нет места, то он "делится". Структура индекса выдаётся функцией расширения pageinspect:

```
select * from bt_multi_page_stats('t_idx',1,-1);
```

blkno	type	live_items	dead_items	avg_item_size	page_size	free_size	btpo_prev	btpo_next	btpo_level	btpo_flags
1	l	2	0	2512	8192	3116	0	2	0	1
2	l	2	0	2512	8192	3116	1	4	0	1
3	i	3	0	1677	8192	3104	0	7	1	0
4	l	2	0	2512	8192	3116	2	5	0	1
5	l	2	0	2512	8192	3116	4	6	0	1
6	l	3	0	2512	8192	600	5	0	0	1
7	i	3	0	1677	8192	3104	3	0	1	0
8	r	2	0	1260	8192	5620	0	0	2	2

`btpo_prev` хранит номер блока слева от текущего (blkno) на том же уровне. Значение 0 означает, что слева блока нет, этот блок самый левый.

`btpo_next` номер блока справа от текущего. Значение 0: текущий блок самый правый.

`type` тип блока: r - корневой (root); i - внутренний (internal); l - листовой (list), e - (ignored), d - удалённый листовой (deleted leaf), D - удалённый внутренний (deleted internal)

btpo_flags битовая карта:

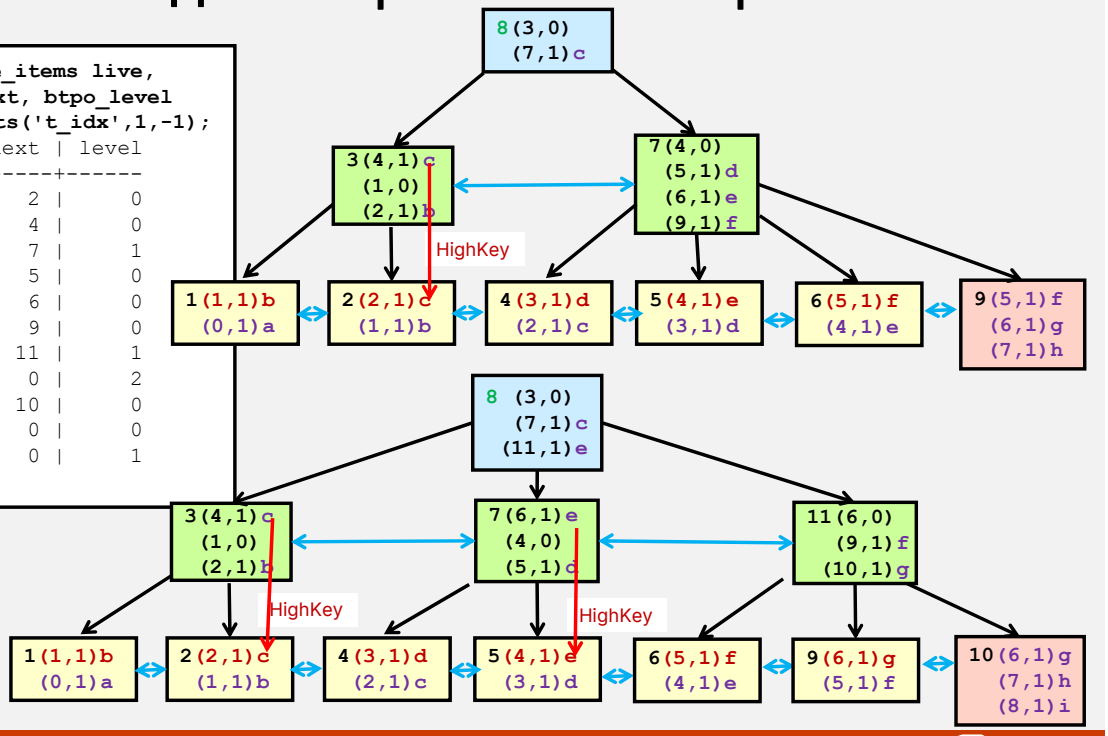
1 - листовой блок, 2 - корневой блок, 4 - свободный блок, был удалён из структуры индекса, 8 - блок метаданных, 16 - пустой блок, но в структуре дерева (half-dead). Остальные биты используются при вакуумировании для отслеживания изменения в структуре индекса: 256 признак удаленного блока (BTDeletedPageData).

Можно заметить, что номера блоков соответствуют последовательности добавления блока в структуру индекса. Корневой блок меняется при увеличении числа уровней (level).

Индекс растёт снизу вверх. В листовом блоке не хватает места, он "делится": добавляется пустой блок, строки перераспределяются, ссылка на новый блок вставляется в вышестоящий блок. Если в вышестоящем блоке нет места, то он тоже "делится". Почему используется слово "делится"? Потому что строки в блоке перераспределяются между ним и добавляемым в структуру индекса.

Пример роста индекса при вставке строк

```
select blkno blk, type, live_items live,
btpo_prev prev, btpo_next next, btpo_level
level from bt_multi_page_stats('t_idx',1,-1);
 blk | type | live | prev | next | level
-----+------+------+------+------+------
   1 | l    |    2 |    0 |    2 |     0
   2 | l    |    2 |    1 |    4 |     0
   3 | i    |    3 |    0 |    7 |     1
   4 | l    |    2 |    2 |    5 |     0
   5 | l    |    2 |    4 |    6 |     0
   6 | l    |    2 |    5 |    9 |     0
   7 | i    |    3 |    3 |   11 |     1
   8 | r    |    3 |    0 |    0 |     2
   9 | l    |    2 |    6 |   10 |     0
  10 | l    |    3 |    9 |    0 |     0
  11 | i    |    3 |    7 |    0 |     1
(11 rows)
```



Пример роста индекса при вставке строк

Содержимое блоков индекса показывает запрос, пример приведён для 9 строк:

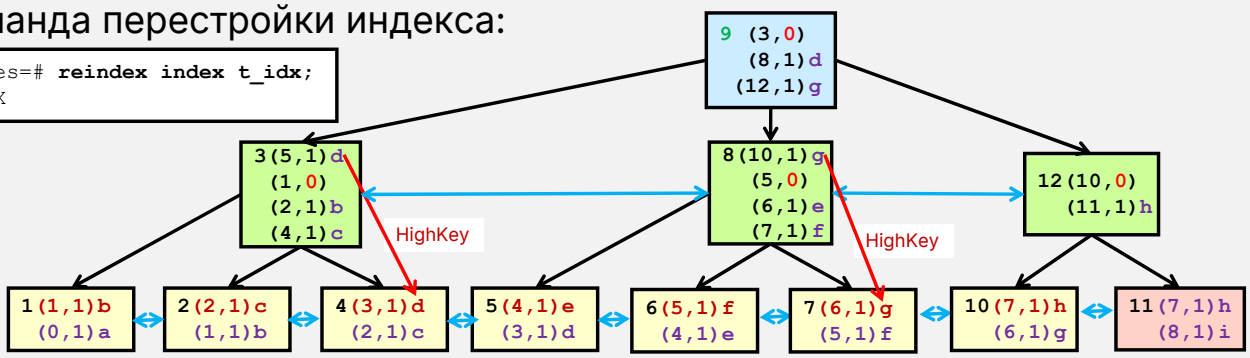
```
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',номер_блока);
```

```
 o | ctid   | itemlen | htid   |        data        | c
---+-------+---------+-------+--------------------+---
 1 | (1,1) |    2512 |       | 20 27 00 00 62 62  | b
 2 | (0,1) |    2512 | (0,1) | 20 27 00 00 61 61  | a
---+-------+---------+-------+--------------------+---
 1 | (2,1) |    2512 |       | 20 27 00 00 63 63  | c
 2 | (1,1) |    2512 | (1,1) | 20 27 00 00 62 62  | b
---+-------+---------+-------+--------------------+---
 1 | (4,1) |    2512 |       | 20 27 00 00 63 63  | c
 2 | (1,0) |       8 |       |                    |
 3 | (2,1) |    2512 |       | 20 27 00 00 62 62  | b
---+-------+---------+-------+--------------------+---
 1 | (3,1) |    2512 |       | 20 27 00 00 64 64  | d
 2 | (2,1) |    2512 | (2,1) | 20 27 00 00 63 63  | c
---+-------+---------+-------+--------------------+---
 1 | (4,1) |    2512 |       | 20 27 00 00 65 65  | e
 2 | (3,1) |    2512 | (3,1) | 20 27 00 00 64 64  | d
---+-------+---------+-------+--------------------+---
 1 | (5,1) |    2512 |       | 20 27 00 00 66 66  | f
 2 | (4,1) |    2512 | (4,1) | 20 27 00 00 65 65  | e
---+-------+---------+-------+--------------------+---
 1 | (6,1) |    2512 |       | 20 27 00 00 65 65  | e
 2 | (4,0) |       8 |       |                    |
 3 | (5,1) |    2512 |       | 20 27 00 00 64 64  | d
---+-------+---------+-------+--------------------+---
 1 | (3,0) |       8 |       |                    |
 2 | (7,1) |    2512 |       | 20 27 00 00 63 63  | c
 3 | (11,1)|    2512 |       | 20 27 00 00 65 65  | e
---+-------+---------+-------+--------------------+---
 1 | (6,1) |    2512 |       | 20 27 00 00 67 67  | g
 2 | (5,1) |    2512 | (5,1) | 20 27 00 00 66 66  | f
---+-------+---------+-------+--------------------+---
 1 | (6,1) |    2512 | (6,1) | 20 27 00 00 67 67  | g
 2 | (7,1) |    2512 | (7,1) | 20 27 00 00 68 68  | h
 3 | (8,1) |    2512 | (8,1) | 20 27 00 00 69 69  | i
---+-------+---------+-------+--------------------+---
 1 | (6,0) |       8 |       |                    |
 2 | (9,1) |    2512 |       | 20 27 00 00 66 66  | f
 3 | (10,1)|    2512 |       | 20 27 00 00 67 67  | g
```

Структура индекса после его перестроения

- HighKey вышестоящего блока должно присутствовать в самом правом нижестоящем блоке, на который есть ссылка в вышестоящем блоке. Если значение отличается, то процесс должен прочесть блоки нижнего уровня правее самого правого, так как скорее всего он уже не является правым
- в самых правых блоках на своём уровне HighKey не хранятся, так как правее их блоков нет и проверка не нужна
- команда перестройки индекса:

```
postgres=# reindex index t_idx;
REINDEX
```



Структура индекса после его перестроения

Содержимое блоков индекса после перестройки командой:
`reindex index t_idx;`
Пример на слайде приведён для 10 строк со значениями: `a,b,c,d,e,f,g,h,i`.

Стрелками красного цвета показана связь HighKey. Значение HighKey вышестоящего блока должно присутствовать в самом правом нижестоящем блоке, на который есть ссылка в вышестоящем блоке. Если значение отличается, то процесс должен прочесть блоки нижнего уровня правее самого правого, так как скорее всего он уже не является правым.

В самых правых блоках на своём уровне HighKey не хранятся, так как правее их блоков нет и проверка не нужна.

Файлы индекса могут включать в себя блоки, которые исключены из структуры индекса. При вакуумировании читаются все блоки файлов индекса по порядку от начала файла до конца каждого файла. Уменьшение размера файлов индекса ускоряет работу автовакуума и уменьшает число пустых блоков индекса, которые вынужден подгружать в буферный кэш автовакуум.

В промежуточных блоках в `ctid` хранится ссылка на блок индекса, а второе число единица или ноль. **Если единица, то в поле data хранится минимальное значение, которое присутствует в дочернем листовом блоке.** Если ноль `ctid=(N,0)`, то поле `data` пусто (трактуется как "минус бесконечность", то есть граница неизвестна) и эта ссылка ведёт на самый левый дочерний блок.

В примере на слайде второе число ноль в `ctid: (3,0) (1,0) (5,0) (10,0)` и у них поле `data` пусто.

FILLFACTOR в индексах типа btree

- для индексов на столбец, заполняемый возрастающей последовательностью стоит устанавливать fillfactor=100
- в случае установки `fillfactor=100` левый блок при делении самого правого листового блока будет максимально заполнен, данные будут храниться более компактно, деление правого блока будет происходить реже
- в btree индексах fillfactor установлен в значения:
  - 90% для листовых блоков
  - 70% для нелистовых блоков и это не меняется
  - 96% при разделении любого листового блока, который полностью заполнен дубликатами (одним и тем же значением)
- для листовых блоков fillfactor применяется:
  - во время построения индекса
  - при разделении крайней правой страницы как листового, так и промежуточных уровней

FILLFACTOR в индексах типа btree

Если при вставке строки в таблицу делится самая правая страница на уровне (последняя страница уровня), то **деления поровну не происходит, левая страница заполняется до fillfactor**, а правая остается почти свободной. Это полезно для индексов на автоинкрементальные столбцы или заполняемые возрастающей последовательностью, так как вставки будут всегда идти в самый правый листовой блок и он будет делиться. Для таких индексов стоит устанавливать fillfactor=100, иначе индексы будут иметь больший размер, место в листовых блоках будет расходоваться впустую. В случае установки `fillfactor=100` левый блок при делении самого правого листового блока будет максимально заполнен, данные будут храниться более компактно, деление правого блока будет происходить реже. Промежуточные блоки таких индексов будут заполняться на 70% независимо от значения fillfactor. Однако промежуточные блоки вносят небольшой вклад в размер индекса. **Для числового столбца (int2, int4, int8) в промежуточном блоке при 70% заполнения помещается 286 ссылок, при 100% поместится 407 ссылок.**

В примерах это не показано, даже наоборот: в правых блоках строк оказывается больше. Это из-за размера строки и fillfactor, который превышается даже одной строкой.

Если бы в блоках помещалось больше строк, то при делении правого блока (блок делится если в нем нет места независимо от fillfactor) в левый уходили бы строки до fillfactor, а остальные оставались в правом блоке.

В индексах типа btree fillfactor установлен в значения:

1) 90% для листовых блоков (`BTREE_DEFAULT_FILLFACTOR=90`)

2) 70% для нелистовых блоков и это не меняется (`BTREE_NONLEAF_FILLFACTOR=70`).

Для листовых блоков fillfactor применяется:

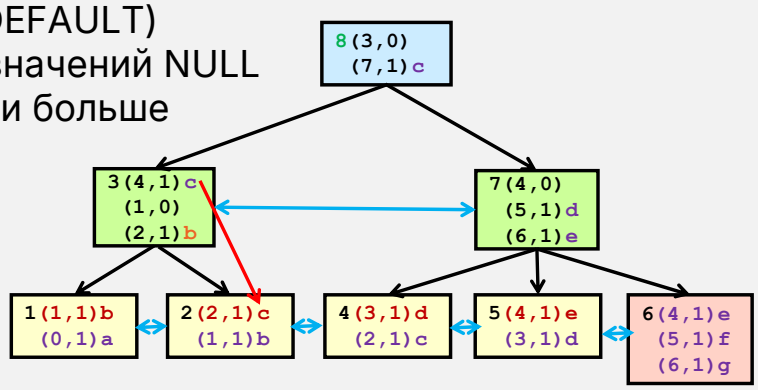
1) во время построения индекса

2) при разделении крайней правой страницы как листового, так и промежуточных уровней.

При разделении не самых правых страниц данные перераспределяются поровну. В родительский блок вставляется ссылка на второй блок. Если в родительском блоке нет места, то он делится и так до корневого блока. Если в корневом блоке нет места, то добавляется новый уровень. При разделении любого (в том числе правого) листового блока, который полностью заполнен одним и тем же значением (дубликатами) fillfactor=96 (`BTREE_SINGLEVAL_FILLFACTOR=96`).

Быстрый путь (fastpath) вставки в индексы

- процесс, который выполнил вставку в правый листовой блок запоминает ссылку на него и при последующей вставке, если новое значение больше предыдущего (или пусто) и не проходит путь от корня до листового блока
- ускоряет вставки:
- в целочисленные столбцы, заполняемые последовательностью
- датавременные (заполняемые DEFAULT)
- при вставке преимущественно значений NULL
- используется при числе level=2 и больше



Быстрый путь (fastpath) вставки в индексы

Столбец с первичным ключом обычно делают автоинкрементальным, то есть используется монотонно возрастающая последовательность целых чисел.

В таком случае, при вставке (INSERT) новой строки она всегда будет вставляться в самый правый листовой блок и при вставке нет смысла проходить блоки начиная с корня. Также при вставке в датавременной столбец, заполняемый по DEFAULT временем вставки или заполняемые другим способом. Также при вставке пустого значения, то есть если последовательно вставляются строки, а в проиндексированный столбец вставляется NULL.

Оптимизация вставки строки называется fastpath. Процесс, который выполнил вставку в правый листовой блок, запоминает ссылку на него и при последующей вставке, если новое значение больше предыдущего (или пусто) и не проходит путь от корня до листового блока.

Процесс забывает адрес блока и снова начинает поиск с корня, если по какой-либо причине выполнил вставку (в индекс записи только вставляются, они не меняются, а удаляются только вакуумом) не в самый правый блок.

Fastpath используется при числе уровней в индексе начиная с 2 (`BTREE_FASTPATH_MIN_LEVEL`). Оптимизация хорошо работает, если один процесс вставляет строки в таблицу. Если процесс столкнется на блокировке правого листового блока с

другим процессом (проверяется что буфер не закреплен другими процессами), этот процесс перестает использовать fastpath. Другой процесс может продолжать,

так как он не сталкивался. Получается, что процесс "отправляется на штрафной круг": прочтет несколько блоков от корня до правого листового блока, и на за это время не будет сталкиваться с другими процессами.

Не стоит распараллеливать в таблицу вставки строк, предполагая что это всегда быстрее: процессы могут встретиться с закрепленным буфером.

Значение в третьем внутреннем блоке b указывает на самое "левое" (в порядке сортировки, с которой создан индекс) значение, которое равно b во втором листовом блоке. В листовых блоках, кроме самого правого листового блока, хранится одно значение и HighKey.

Внутристраничная очистка в индексах

- выполняется при индексном сканировании
- если строка в таблице удалена, индексная запись на строку или цепочку версий помечается флагом LP_DEAD
- пометка может ставиться командой SELECT при индексном сканировании
- журнальная запись не создается, но блок грязнится
- помеченная индексная запись будет очищена при выполнении команд, которые меняют данные в таблице

```
create table t (id int primary key, c text) with (autovacuum_enabled = off);
insert into t SELECT i, 'simple delete ' || i from generate_series(1, 10000) as i;
delete from t where id between 100 and 9000;
analyze t;
explain (analyze, settings, buffers, costs off) select * from t where id between 1 and 9000;
select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, left(data,8) from
bt_page_items('t_pkey',20) where dead=true limit 2;
 itemoffset |  ctid   | itemlen | nulls | vars | dead |  htid   | tids | substring
------------+---------+---------+-------+------+------+---------+------+-----------
          2 | (42,37) |      16 | f     | f    | t    | (42,37) |      | bd 19 00
          3 | (42,38) |      16 | f     | f    | t    | (42,38) |      | be 19 00
```


Внутристраничная очистка в индексах

Если при индексном сканировании (Index Scan) серверный процесс обнаружит, что строка (или цепочка строк, на которую ссылается индексная запись) удалена и вышла за горизонт базы, то в индексной записи листового блока (leaf page) в lp_flags устанавливается бит-подсказка LP_DEAD (**называют known dead, killed tuple**). Бит можно посмотреть в столбце `dead`, **выдаваемый функцией `bt_page_items('t_idx',блок)`.**

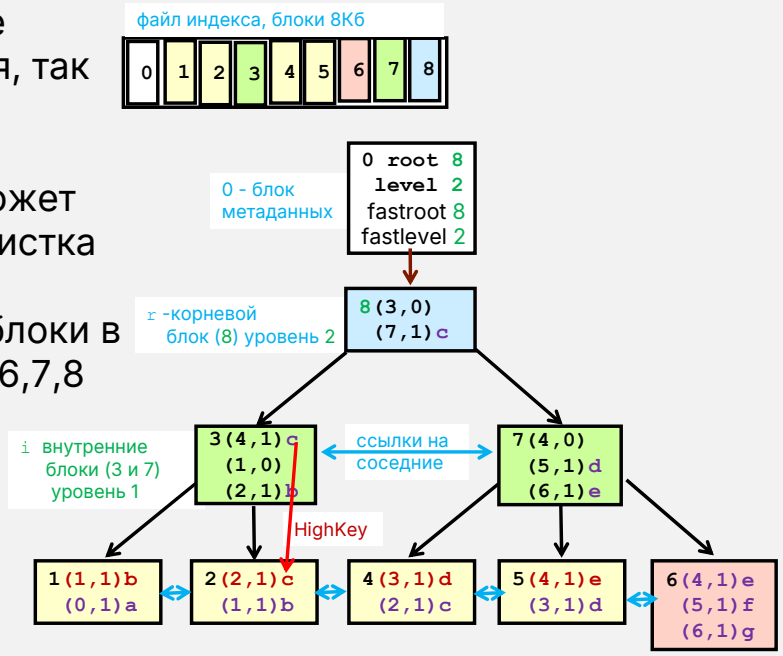
При Bitmap Index Scan и Seq Scan (неиндексном доступе) не устанавливается. Помеченная таким флагом строка будет удалена позже при выполнении команды, которая вносит изменения в блок индекса.

Возврат в блок и установка в нем флага добавляет накладные расходы и увеличивает время выполнения команды, но делается однократно. Зато последующие команды смогут игнорировать индексную запись и не будут обращаться к блоку таблицы.

```
drop table t;
create table t (id int primary key, c text) with (autovacuum_enabled = off);
insert into t SELECT i, 'simple delete ' || i from generate_series(1, 10000) as i;
delete from t where id between 100 and 9000;
analyze t;
select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, left(data,8) from bt_page_items('t_pkey',20)
limit 2;
 itemoffset |  ctid   | itemlen | nulls | vars | dead |  htid   | tids | substring
------------+---------+---------+-------+------+------+---------+------+-----------
          1 | (44,1)  |      16 | f     | f    |      |         |      | 2b 1b 00
          2 | (42,37) |      16 | f     | f    | f    | (42,37) |      | bd 19 00
explain (analyze, settings, buffers, costs off) select * from t where id between 1 and 9000;
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using t_pkey on t (actual time=0.013..1.480 rows=99 loops=1)
   Index Cond: ((id >= 1) AND (id <= 9000))
select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, left(data,8) from bt_page_items('t_pkey',20)
where dead=true limit 2;
 itemoffset |  ctid   | itemlen | nulls | vars | dead |  htid   | tids | substring
------------+---------+---------+-------+------+------+---------+------+-----------
          2 | (42,37) |      16 | f     | f    | t    | (42,37) |      | bd 19 00
          3 | (42,38) |      16 | f     | f    | t    | (42,38) |      | be 19 00
```

Влияние удаления строк на индексы

- при удалении строки в таблице индексная запись не удаляется, так как строка в табличном блоке остаётся до вакуумирования
- удалять из индексов записи может вакуум и внутристраничная очистка в индексах
- вакуум сканирует индексные блоки в физическом порядке: 1,2,3,4,5,6,7,8



Влияние удаления строк на индексы

При удалении строки в таблице индексная запись не удаляется, так как строка в табличном блоке остаётся до вакуумирования. Удалять из индексов записи может вакуум и внутристраничная очистка в индексах.

При удалении записи (в листовом блоке) в вышестоящий блок не вносятся изменения, так как в родительском блоке хранится ссылка на первую запись дочернего блока, которая хранит High key.

В самом правом листовом блоке High key нет, так как правее него нет блоков. В промежуточных (внутренних) и корневом блоке строк с High key нет, используются все строки. В примере на слайде в 3 промежуточном блоке первая строка "`3 (2,1)b`" ссылается на 2 листовой блок. В 7 промежуточном блоке первая строка "(4,0)" ссылается на 4 листовой блок. В промежуточных блоках в строке хранится проиндексированное **значение** и ссылка на нижестоящий блок, который содержит это значение или больше (High key листовых блоков не учитываются). В одной из строк промежуточного блока значение пусто, это означает, что эта ссылка "самая левая".

Если в блоке удалены все записи, то он помечается флагом `btpo_flags+=16` (означает пустой блок), но блок пока находится в структуре дерева (half-dead), только убираются ссылки на блок с соседних блоков на том же уровне.

Позже блок удаляется из структуры индекса (на него убирается ссылка из вышестоящего блока), блок помечается флагом `btpo_flags+=4` как свободный блок (удалён из структуры индекса). При этом блок остаётся в файле индекса. Блок не будет использоваться для вставок новых строк, пока его не обработает вакуум. Блок сможет вернуться в структуру индекса, так как вакуумом он помечается в карте свободного пространства fsm как свободный. Серверные процессы ищут свободные блоки в карте видимости и только потом расширяют файл индекса. Разбиение на два шага нужно из-за того, что другие процессы могут иметь блокировки на блок. Ждать снятия или проверять блокировки менее производительно, чем разбить процесс удаления блока из структуры индекса на два шага.

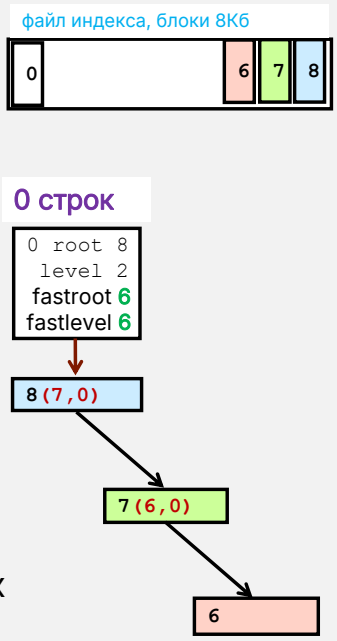
Как вывод - обработка индекса вакуумом (автовакуумом) важна.

Вакуум сканирует индексные блоки не в логическом порядке (листовые по ссылкам на соседние), а физическом порядке: 0,1,2,3,4,5,6,7,8.

Именно поэтому вакуум может найти неиспользуемые (убранные из структуры индекса) блоки, даже если файлы fsm отсутствуют. Вакуум обновляет или создает fsm файлы индекса. vm файла у индекса btree нет.

Исключение блоков из структуры индекса

- после удаления строк в таблице вакуум может исключать блоки из структуры индекса
  - блоки остаются в файлах индекса
  - помечаются как свободные в карте свободного пространства (fsm) индекса
  - могут использоваться повторно встраиваясь в структуру индекса и в другом месте дерева
- число уровней не уменьшается вакуумом
- "правый" блок каждого уровня индекса не может быть удалён из дерева
- если у корневого и внутренних (промежуточных) блоков один наследник, то он становится "fastroot" - быстрым корнем
  - его адрес и его уровень сохраняется в блоке метаданных в полях fastroot и fastlevel
  - поиск по индексу начинается с fastroot,а не root



Исключение блоков из структуры индекса

Первый (0) блок индекса содержит метаданные:

3) fastroot и fastlevel. После удаления всех или большого количества строк в таблице вакуум может исключать блоки из структуры индекса. Такие блоки остаются в файлах индекса, в карте свободного пространства (fsm) индекса помечаются как свободные и могут использоваться повторно встраиваясь в структуру индекса и в другом месте дерева. Но даже при удалении всех строк число уровней до которого дорос индекс не уменьшается. Это объясняют тем, что самый "правый" блок каждого уровня индекса не может быть удалён из дерева. В примере на слайде неудаляемые блоки: 8, 7, 6. Если такое (вставили в таблицу много строк, число level стало большим, потом удалили все строки и level не уменьшился) произошло, то используется оптимизация "fastroot". Если у корневого и промежуточных (внутренних) блоков один наследник, то он становится "fastroot" - быстрым корнем. Его адрес и его уровень сохраняется в блоке метаданных в полях fastroot и fastlevel. И поиск по индексу начинается с fastroot,а не root. Это существенно ускоряет поиск.

```
select * from bt_multi_page_stats('t_idx',1,-1);
blkno|type|live_|dead_|avg_item|page_|free_|btpo_|btpo_|btpo_|btpo_
     |    |items|items|   _size| size| size| prev| next|level|flags
-----+----+-----+-----+--------+-----+-----+-----+-----+-----+----
   1 | d  |   0 |  0  |   2512 |8192 |8140 |   0 |   2 |   0 |261
   2 | d  |   0 |  0  |   2512 |8192 |8140 |   0 |   4 |   0 |261
   3 | D  |   0 |  0  |   1677 |8192 |8140 |   0 |   7 |   1 |260
   4 | d  |   0 |  0  |   2512 |8192 |8140 |   0 |   5 |   0 |261
   5 | d  |   0 |  0  |   2512 |8192 |8140 |   0 |   6 |   0 |261
   6 | l  |   0 |  0  |   2512 |8192 |8148 |   0 |   0 |   0 |  1
   7 | i  |   3 |  0  |      8 |8192 |8136 |   0 |   0 |   1 |  0
   8 | r  |   2 |  0  |      8 |8192 |8136 |   0 |   0 |   2 |  2
```

261=1+4+256, что значит листовой, удалённый, признак удалённого блока.

В аргументах функции:, 1 с первого блока -1 до последнего блока.

```
select * from bt_metap('t_idx');
magic|version|root|level|fastroot|fastlevel|last_cleanup_num_delpages|last_cleanup_num_tuples|allequalimag
-----+-------+----+-----+--------+---------+-------------------------+-----------------------+------------
340322|    4 |  8 |   2 |      6 |       0 |                       5 |                    -1 | t
select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, left(data,24) data from
bt_page_items('t_idx',номер_блока);
 блок | itemoffset | ctid  | itemlen | nulls | vars | dead | htid  | tids | data
------------------+-------+---------+-------+------+------+-------+------+------
   7 |          1 | (6,0) |       8 | f     | t    |      |       |      |
   8 |          1 | (7,0) |       8 | f     | t    |      |       |      |
```

Число исключённых блоков из структуры индекса

- запрос для получения числа исключённых из структуры индекса блоков:

```
select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
 type | count
------+-------
 D    |     6
 r    |     1
 l    |   552
 i    |     4
 d    |  2181
```

- type: d - удалёный листовой (deleted leaf) блок, D - удалённый внутренний (deleted internal) блок
- второй столбец в запросе показывает число блоков
- изменения LP_DEAD не передаются на реплику, на реплике биты LP_DEAD при сканировании по индексу не выставляются
- при вакуумировании читаются все блоки в файлах индекса
  - чем меньше блоков в файлах индекса, тем быстрее будет работать автовакуум

Число исключённых блоков из структуры индекса

Запрос для получения числа исключённых из структуры индекса блоков:

```
select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
 type | count
------+-------
 D    |     6
 r    |     1
 l    |   552
 i    |     4
 d    |  2181
(5 rows)
```

В первом столбце показываются типы блоков: d - удалёный листовой (deleted leaf) блок, D - удалённый внутренний (deleted internal) блок.

Второй столбец в запросе показывает число блоков.

Блоки, в которых все записи были помечены как удалённые (при вакуумировании или раньше), исключаются из структуры индекса вакуумом. Исключённые блоки не сканируются запросами, но они считываются при любом вакуумировании, так как у индексов нет карты видимости и считываются все блоки. Вакуум считывает все блоки индекса последовательно и в физическом порядке. Исключённые из структуры индекса блоки возвращаются в структуру индекса при расщеплении блоков индекса (то есть росте структуры индекса).

На физической реплике индексные записи не помечаются флагом LP_DEAD, так как реплика не может менять содержимое блоков локальных объектов баз данных.

Изменения в битах-подсказках LP_DEAD не журналируются и не передаются на реплику.

https://www.postgresql.org/message-id/flat/7067.1529246768@sss.pgh.pa.us#d9e2e570ba34fc96c4300a362cbe8c38

Практика

- Часть 1. Методы доступа
- Часть 2. Использование индексов ограничениями целостности
- Часть 3. Характеристики btree индексов
- Часть 4. Навигация по структуре btree индексов
- Часть 5. Дедупликация в btree индексах
- Часть 6. Индексы в убывающем порядке
- Часть 7. Покрывающие индексы и Index Only Scan
- Часть 8. Частичные (partial) индексы
- Часть 9. Изучение структуры индекса типа btree
- Часть 10. Очистка блоков индекса при его сканировании
- Часть 11. Медленное выполнение запросов на реплике из-за отсутствия очистки блоков индекса
- Часть 12. Определение числа удаленных строк
- Часть 13. Поиск по структуре индекса btree

Практика

В практике рассматриваются все детали использования индексов типа b-tree.

В 11 части практики вы создадите физическую реплику и посмотрите как очистка блоков от старых версий строк может влиять на запросы, выполняющиеся на реплике.

8-1

TOAST

TOAST (The Oversized-Attribute Storage Technique)

- TOAST это техника хранения "атрибутов" (полей) большого размера
- поля большого размера - это поля не помещающиеся в блок
- позволяет хранить поля размером до 1Гб
- если в таблице есть заведомо большое поле или вставляется строка с большим полем, то создается toast-таблица и индекс на toast-таблицу
- в TOAST выносятся отдельные поля строк
- вынесенные поля делятся на части (chunk) по 1996 байт:

```
postgres@tantor:~$ pg_controldata | grep TOAST
Maximum size of a TOAST chunk:        1996
postgres@tantor:~$ /usr/lib/postgresql/15/bin/pg_controldata
-D /var/lib/postgresql/15/main| grep TOAST
Maximum size of a TOAST chunk:        1988
postgres@vanilla-x32:~$ pg_controldata | grep TOAST
Maximum size of a TOAST chunk:        2000
```


TOAST (The Oversized-Attribute Storage Technique)

TOAST (The Oversized-Attribute Storage Technique, техника хранения атрибутов большого размера) используется не только для хранения отдельных полей в TOAST-таблице. Код ядра PostgreSQL используется при обработке длинных значений в памяти. Не все встроенные типы данных поддерживают технику TOAST. Не поддерживают типы данных фиксированной длинны, так как их длинна небольшая и для любых значений одинакова ("фиксирована"), например, 1,2,4,8 байт.

Размер типа, поддерживающего TOAST ограничен 1 Гигабайт. Это ограничение следует из того, что под длину в начале поля в блоке отводится 30 бит (2^30=1Гб) из 1 или 4 байт (32 бита). Два бита в этих байтах используются для обозначения: 00 - значение короткое, не TOAST, оставшиеся биты задают длину поля вместе с этим байтом; 01 - длинна поля хранится в одном байте, оставшиеся биты задают длину поля в байтах и эти 6 бит могут хранить длину от 1 до 126 байт (2^6=64, но это для диапазона от нуля); 10 - значение сжато, оставшиеся биты задают длину поля в сжатом виде. Значения с одним байтом заголовка поля не выравниваются. Значения четырьмя байтами заголовка поля выравниваются по границе pg_type.typealign.

Вынесенные в TOAST поля делятся на части - "чанки" (после сжатия, если оно применялось) размером 1996 байт (значение задано константой TOAST_MAX_CHUNK_SIZE), которые располагаются в строках TOAST-таблицы размером 2032 байта (значение задано константой TOAST_TUPLE_THRESHOLD). **Значения выбраны так, чтобы в блок таблицы TOAST поместилось четыре строки.** Так как размер поля таблицы не кратен 1996 байт, то последний чанк поля может быть меньшего размера.

Значение TOAST_MAX_CHUNK_SIZE хранится в управляющем файле кластера, его можно посмотреть утилитой **pg_controldata**.

В таблице TOAST есть три столбца: chunk_id (тип OID, уникальный для поля вынесенного в TOAST размер 4 байта), chunk_seq (порядковый номер чанка, размер 4 байта), chunk_data (данные поля, тип bytea, размер сырых данных плюс 1 или 4 байта на хранение размера). Для быстрого доступа к чанкам на TOAST-таблицу создается составной уникальный индекс по chunk_id и chunk_seq. В блоке таблицы остаётся указатель на первый чанк поля и другие данные. **Общий размер остающейся в таблице части поля всегда 18 байт.**

В 32-разрядном PostgreSQL размер чанка на 4 байта больше: 2000 байт.

В AstraLinux PostgreSQL размер чанка на 8 байт меньше: 1988 байт.

https://docs.tantorlabs.ru/tdb/ru/15_6/se/storage-toast.html

Поля переменной длинны

- строка должна поместиться в один блок
- поля varlena, которые не помещаются в блок выносятся в таблицу TOAST
- типы данных фиксированной ширины не сжимаются и не выносятся в TOAST
- стратегию хранения можно установить командой `ALTER TABLE имя ALTER COLUMN имя SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT }`.
- сжатие поддерживается для MAIN и EXTENDED
- **1 байт в начале поля переменной ширины хранит длину поля длиной до 127 байт**
- **4 байта в начале поля переменной ширины хранит длину поля для полей длиннее 126 байт**

Поля переменной длинны

Строка (запись) таблицы должна поместиться в один блок размером 8Кб и не может находиться в нескольких блоках файлов таблицы. Однако строки могут иметь размер больше 8Кб. Для их хранения применяется TOAST.

Индексная запись индекса btree не может превышать примерно треть блока (после сжатия проиндексированных столбцов, если оно применялось в таблице).

TOAST поддерживают типы данных `varlena (pg_type.typlen=-1)`. Поля фиксированной длины не могут храниться вне блока таблицы, так как для этих типов данных не написан код, реализующий хранение вне блока таблицы (в TOAST-таблице). При этом строка должна поместиться в один блок и фактическое число столбцов в таблице будет меньше, чем лимит в 1600 столбцов (`MaxHeapAttributeNumber` в `htup_details.h`).

Чтобы поддерживать TOAST, **в поле типа varlena первый байт или первые 4 байта всегда (даже если размер поля небольшой и не вытеснен в TOAST) содержат общую длину поля** в байтах (включая эти 4 байта). Причем, эти байты могут (но не всегда) быть сжаты вместе с данными, то есть храниться в сжатом виде. Один байт используется, если длинна поля не превышает 126 байт. Поэтому, при хранении данных поля размером до 127 байт "экономится" три байта на каждой версии строки, а также отсутствует выравнивание, на чем можно сэкономить до 3 (typealign='i') или до 7 байт (typealign='d').

Другими словами, проектировщику схем хранения **лучше задать char(126) и меньше, чем char(127) и больше**.

Поля varlena с одним байтом длинны **не выравниваются**, а **поля с 4 байтами длинны выравниваются до pg_type.typealign.** Для большинства типов переменной длинны выравнивание до 4 байт (`pg_type.typalign=i`). Отсутствие выравнивания даёт выигрыш в объёме хранения, что ощутимо для коротких значений. Но всегда нужно помнить о выравнивании **всей строки до 8 байт, которое выполняется всегда**.

Сжатие поддерживается только для типов данных переменной длины. Сжатие производится только, если режим хранения столбца установлен в MAIN или EXTENDED. Если поле хранится в TOAST и команда UPDATE не затрагивает это поле, то поле не будет специально сжиматься-разжиматься.

Для большинства типов переменной длинны по умолчанию используется режим EXTENDED, кроме типов:

```
 select distinct typname, typalign, typstorage, typcategory, typlen from pg_type where typtype='b' and typlen<0 and typstorage<>'x' order by typname;
   typname   | typalign | typstorage | typcategory | typlen
-------------+----------+------------+-------------+--------
 cidr        | i        | m          | I           |     -1
 gtsvector   | i        | p          | U           |     -1
 inet        | i        | m          | I           |     -1
 int2vector  | i        | p          | A           |     -1
 numeric     | i        | m          | N           |     -1
 oidvector   | i        | p          | A           |     -1
 tsquery     | i        | p          | U           |     -1
(7 rows)
```

Для каждого столбца помимо режима можно еще установить алгоритм сжатия (CREATE или ALTER TABLE). Если не устанавливать, то используется алгоритм из параметра `default_toast_compression`, который по умолчанию установлен в `pglz`.

Режим (стратегию) хранения можно установить командой **`ALTER TABLE имя ALTER COLUMN имя SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT }`**.

EXTERNAL похож на EXTENDED, только без сжатия и по умолчанию не установлено на стандартных типах. Если алгоритм `pglz` не может сжать первый килобайт данных, он прекращает попытку сжатия.

Вытеснение полей в TOAST

- в СУБД Tantor и PostgreSQL выносятся поля строк длинной больше 2032 байт
  - при этом поля режутся на части по 1996 байт
- алгоритм (очередность) вытеснения зависит от порядка следования столбцов
- при доступе к каждому вытесненному полю дополнительно читается 2-3 блока TOAST-индекса

```
select reltoastrelid, reltoastrelid::regclass
from pg_class where relname='t';
 reltoastrelid |      reltoastrelid
---------------+-------------------------
         74295 | pg_toast.pg_toast_74292
\d+ pg_toast.pg_toast_74292
TOAST table "pg_toast.pg_toast_74292"
   Column    |  Type   | Storage
-------------+---------+---------
 chunk_id    | oid     | plain
 chunk_seq   | integer | plain
 chunk_data  | bytea   | plain
Owning table: "public.t"
Indexes:
    "pg_toast_74292_index" PRIMARY KEY,
            btree (chunk_id, chunk_seq)
Access method: heap
select chunk_id, chunk_seq, length(chunk_data)
from pg_toast.pg_toast_74292;
 chunk_id | chunk_seq | length
----------+-----------+--------
    74297 |         0 |   1996
    74297 |         1 |      9
```


Вытеснение полей в TOAST

Способ хранения для обычных таблиц (heap tables) допускает сжатие значений отдельных полей. На данных небольшого размера алгоритмы сжатия менее эффективны. Доступ к отдельным столбцам не очень эффективен из-за того, что серверному процессу нужно найти блок в котором хранится часть строки, помещающаяся в блок, затем по каждой строке отдельно выяснить нужно ли обращаться к строкам TOAST-таблицы, читать её блоки и склеивать части полей (chunk), которые в ней хранятся в виде строк этой таблицы.

У одной таблицы может быть только одна связанная с ней таблица TOAST и один TOAST-индекс (уникальный btree индекс по столбцам `chunk_id` и `chunk_seq`). OID TOAST-таблицы хранится в поле `pg_class.reltoastrelid`.

При доступе к каждому вытесненному полю дополнительно читается 2-3 блока TOAST-индекса, что снижает производительность даже, если блоки в буферном кэше. Основное замедление на получение блокировки для чтения каждого лишнего блока. Любые разделяемые ресурсы (то, что не в локальной памяти процесса) требуют получения блокировки даже для чтения ресурса.

Поля после сжатия (если оно есть) делятся на части (chunk) по **1996 байт:**

```
postgres@tantor:~$ pg_controldata | grep TOAST
Maximum size of a TOAST chunk:        1996
```

В СУБД Tantor и PostgreSQL строка рассматривается на предмет помещения части ее полей в TOAST, если размер строки больше 2032 байт. Поля будут сжиматься и рассматриваться на предмет хранения в TOAST пока строка не поместится в 2032 байта или `toast_tuple_target` байт, если значение было установлено командой:

```
ALTER TABLE t SET (toast_tuple_target = 2032);
```

Оставшаяся часть строки в любом случае должна поместиться в один блок (8Кб).

Для postgresql из Astrallnux 1.8.1:

```
Maximum size of a TOAST chunk: 1988
```

поэтому будут выноситься поля длиннее, чем 1988+8=1996 байт, а не 2004. При этом поле длинной 1997 байт так же породит 2 чанка, второй чанк размером 9 байт, первый размером 1988 байт.

В 32-битном PostgreSQL 9.6 - 2009 байт (а максимальный размер чанка 2000).

Алгоритм вытеснения полей в TOAST

- при вставке строки в таблицу она полностью размещается в памяти серверного процесса в строковом буфере размера 1Гб (или 2Гб
- при обновлении строки обработка выполняется по затрагиваемым командой полям в пределах строкового буфера. Поля, не затрагиваемые командой, представлены в буфере заголовком длинной 18 байт.
- после обработки (сжатия или выноса) каждого поля проверяется размер строки. Если размер не превышает `toast_tuple_target` (по умолчанию 2032 байт) строка сохраняется в буфер и обработка строки заканчивается
- обработка начинается с поля EXTENDED и EXTERNAL от наибольшего размера к меньшему
- сжатие и вынос полей MAIN выполняется только после выноса всех полей EXTENDED и EXTERNAL

Алгоритм вытеснения полей в TOAST

При вставке строки в таблицу она полностью размещается в памяти серверного процесса в строковом буфере размера 1Гб (или 2Гб для сессий у которых установлен параметр конфигурации `enable_large_allocations=on`).

Алгоритм вытеснения в четыре прохода:

1) По очереди выбираются поля EXTENDED и EXTERNAL в порядке от наибольшего размера до меньшего. После обработки каждого поля проверяется размер строки и если размер меньше или равен `toast_tuple_target` (по умолчанию 2032 байт), то вытеснение останавливается и строка сохраняется в блоке таблицы.

Берется поле EXTENDED или EXTERNAL. EXTENDED сжимается. Если размер строки с полем в сжатом виде превышает 2032, поле вытесняется в TOAST. Поле EXTERNAL вытесняется не сжимаясь.

2) Если размер строки всё ещё превышает 2032, во второй проход вытесняются оставшиеся уже сжатые EXTENDED и EXTERNAL по очереди, пока размер строки не станет меньше 2032.

3) Если размер строки не стал меньше 2032, по очереди в порядке размера сжимаются поля MAIN. После сжатия каждого поля проверяется размер строки.

4) Если размер строки не стал меньше 2032, по очереди вытесняются сжатые на 3 проходе MAIN.

5) Если размер строки не помещается в блок, выдаётся ошибка:

```
row is too big: size ..., maximum size ...
```

При обновлении строки обработка выполняется по затрагиваемым командой полям в пределах строкового буфера. Поля не затрагиваемые командой, представлены в буфере заголовком 18 байт.

Toast chunk

- при использовании EXTERNAL поля размером от 1997 байт создают второй чанк небольшого размера из-за которого в блок TOAST помещается только 3 чанка большого размера
- для полей EXTERNAL с размером от 1997 до ~2300 байт есть вероятность менее плотного хранения

```
select reltoastrelid, reltoastrelid::regclass
from pg_class where relname='t';
 reltoastrelid |      reltoastrelid
---------------+-------------------------
         74295 | pg_toast.pg_toast_74292
\d+ pg_toast.pg_toast_74292
TOAST table "pg_toast.pg_toast_74292"
   Column    |  Type   | Storage
-------------+---------+---------
 chunk_id    | oid     | plain
 chunk_seq   | integer | plain
 chunk_data  | bytea   | plain
Owning table: "public.t"
Indexes:
    "pg_toast_74292_index" PRIMARY KEY,
            btree (chunk_id, chunk_seq)
Access method: heap
select chunk_id, chunk_seq, length(chunk_data)
from pg_toast.pg_toast_74292;
 chunk_id | chunk_seq | length
----------+-----------+--------
    74297 |         0 |   1996
    74297 |         1 |      9
```


TOAST chunk

Поле вытесняется в TOAST, если размер строки больше, чем 2032 байта, а резаться поле будет на части по 1996 байт. Из-за этого для поля больше 1996 байт появися чанк небольшого размера, который будет вставлен серверным процессом в блок с чанком большого размера. Например, в таблицу вставить 4 строки:

```
drop table if exists t;
create table t (c text);
alter table t alter column c set storage external;
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
```

в блок TOAST поместится 3 длинных чанка:

```
SELECT lp,lp_off,lp_len,t_ctid,t_hoff FROM heap_page_items(get_raw_page( (SELECT
reltoastrelid::regclass::text FROM pg_class WHERE relname='t'),'main',0));
 lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+-------
  1 |   6152 |   2032 | (0,1)  |     24
  2 |   6104 |     45 | (0,2)  |     24
  3 |   4072 |   2032 | (0,3)  |     24
  4 |   4024 |     45 | (0,4)  |     24
  5 |   1992 |   2032 | (0,5)  |     24
  6 |   1944 |     45 | (0,6)  |     24
```

Полный размер строки с длинным чанком **2032** байт (6104-4072).

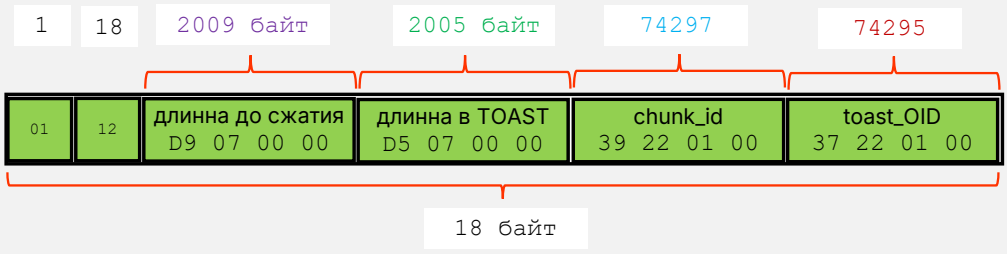
```
select lower, upper, special, pagesize from page_header(get_raw_page( (SELECT
reltoastrelid::regclass::text FROM pg_class WHERE relname='t'),'main',0));
 lower| upper | special | pagesize
------+-------+---------+---------
   48 |  1944 |    8184 |     8192
```

Пример как рассчитывать место в блоке для 4 строк размера 2032 байт (с 4 чанками):
24 (заголовок)+ 4*4 (заголовок) + 2032*4 + 8 (pagesize-special)=8176. Не используется 16 байт, но они бы и не могли использоваться, так строки выравниваются по 8 байт, а их 4.

Ограничения TOAST

- в любой таблице в TOAST может быть вынесено не больше 2^32 полей (~4млрд.)
- для вынесенного в TOAST поля в блоке таблицы хранится 18 байт, они не выравниваются, но вся строка выравнивается
- значение, остающееся в блоке таблицы, после выноса поля в TOAST:

```
select lp_len, t_data from heap_page_items(get_raw_page('t', 0));
 lp_len |                  t_data
--------+-----------------------------------------
     42 | \x0112d9070000d5070000392201003722 0100
```



Ограничения TOAST

В PostgreSQL служебной области special в конце блоков таблиц нет:

```
  48 |  1952 |    8192 |      8192
```

В 32-разрядном PostgreSQL:

```
Maximum size of a TOAST chunk: 2000
```

При использовании EXTENDED скорее всего поле будет сжато и маленького чанка не будет.
https://eax.me/postgresql-toast/

Каждое поле хранится в TOAST таблице в виде набора строк (chunk) хранится в виде одной строки в TOAST-таблице.

В поле основной таблицы хранится указатель на первый chunk **размером 18 байт** (независимо от размера поля). В этих 18 байтах хранится структура `varatt_external`, описанная в `varatt.h`:

первый байт имеет значение 0x01, это признак того, что поле вынесено в TOAST;
второй байт - длинна этой записи (значение 0x12 = 18 байт);
4 байта длинна поля с заголовком поля до сжатия;
4 байта длинна того, что вынесено в TOAST;
4 байта - указатель на первый чанк в TOAST (столбец chunk_id таблицы TOAST);
4 байта - oid toast-таблицы (`pg_class.reltoastrelid`)

В столбце chunk_id (тип oid 4 байта) может быть 4млрд. (2 в степени 32) значений. Это значит, что в одной таблице в TOAST может быть вытеснено только 4млрд. **полей** (даже не строк). Это существенно ограничивает количество строк в исходной таблице и, вероятно, желателен мониторинг. Обойти ограничение можно секционированием.

Режим MAIN применяется для хранения внутри блока в сжатом виде, EXTERNAL - для хранения в TOAST в несжатом виде, EXTENDED для хранения в TOAST в сжатом виде. Если значения плохо сжимаются или планируется обрабатывать значения полей (например, текстовые поля функциями substr, upper), то эффективным будет использование режима EXTERNAL. Для типов фиксированной ширины установлен режим PLAIN, который поменять командой ALTER TABLE нельзя, будет выдана ошибка "`ERROR: column data type тип can only have storage PLAIN`".

https://habr.com/ru/companies/postgrespro/articles/710104/

Выравнивание строк с вытесненными в TOAST полями

- выравниваются как отдельные поля, так и **вся строка**
- каждая строка выравнивается по 8 байт

Пример:

Две таблицы:
```
create table t3 (c1 text);
create table t4 (c1 serial , c2 smallint, c3 text);
```
вставить строки с полем большой длинны, так чтобы все поля были вытеснены в TOAST:
```
DO $$ BEGIN
FOR i IN 1 .. 200 LOOP
insert into t3 VALUES (repeat('a',1000000)); insert into t4 VALUES (default, 1, repeat('a',1000000));
END LOOP; END; $$ LANGUAGE plpgsql;
```
то в блок обоих таблиц помещается одинаковое число строк:
```
select count(*) from heap_page_items(get_raw_page('t3','main',0)) where (t_ctid::text::point)[0]=0 union
select count(*) from heap_page_items(get_raw_page('t4','main',0)) where (t_ctid::text::point)[0]=0;
 count
-------
   156
(1 row)
```
Место, занимаемое таблицами тоже одинаковое

- в таблице t4 можно сохранить два столбца за счет места, которое в первой таблице не могло использоваться

Выравнивание строк с вытесненными в TOAST полями

В блоке таблицы для поля, вынесенного в TOAST-таблицу хранится указатель размером всегда 18 байт.

Про выравнивание отдельных полей известно, но о выравнивании всей строки часто забывают. Выравнивается строка с заголовком или область данных? И то и другое. Заголовок строки всегда выравнивается по 8 байт и может иметь размер 24, 32, 40.. байт. Если сказать что выравнивается область данных, то автоматически будет выровнена область данных с заголовком (вся строка). Если сказать, что выравнивается вся строка и заголовок, то из этого автоматически следует, что будет выровнена область данных. **Все** они выравниваются по 8 байт, так как СУБД Tantor работает только на 64-битных операционных системах (x86-64 и ARM64). Если создать две таблицы:

```
create table t3 (c1 text);
create table t4 (c1 serial , c2 smallint, c3 text);
```

вставить 200 строк с полем большой длинны, так чтобы значение длинного поля было вытеснено в TOAST:

```
DO $$ BEGIN FOR i IN 1 .. 200 LOOP
insert into t3 VALUES (repeat('a',1000000));
insert into t4 VALUES (default, 1, repeat('a',1000000));
END LOOP; END; $$ LANGUAGE plpgsql;
```

то в блок обоих таблиц помещается одинаковое число строк:

```
select count(*) from heap_page_items(get_raw_page('t3','main',0)) where
(t_ctid::text::point)[0]=0 union all select count(*) from
heap_page_items(get_raw_page('t4','main',0)) where (t_ctid::text::point)[0]=0;
    156
    156
```

Во второй таблице можно хранить два столбца в том же месте, которое в первой таблице не могло использоваться (6 байт) из-за padding всей строки.

Размер файлов таблиц также одинаков:

```
select pg_total_relation_size('t3'), pg_total_relation_size('t4');
          5070848                |          5070848
```

Если заменить serial и smallint на bigserial, то размер будет разный (156 и 135 строк в каждом блоке). bigserial бесполезен, так как в таблицах сможет сохраниться не больше 4млрд (2^32) строк с длинным полем.

Параметры `toast_tuple_target` и `default_toast_compression`

- на вытеснение влияют два макроса: TOAST_TUPLE_THRESHOLD и TOAST_TUPLE_TARGET
  - по умолчанию равны **2032**
- если размер строки больше TOAST_TUPLE_THRESHOLD, то начинается сжатие и/или вытеснение полей строки
- поля будут сжиматься и рассматриваться на предмет хранения в TOAST, пока оставшиеся поля не поместятся в TOAST_TUPLE_TARGET
- TOAST_TUPLE_TARGET можно переопределить только **на уровне таблицы**:
- `ALTER TABLE t SET (toast_tuple_target = 2032);`
- **TOAST_TUPLE_THRESHOLD не переопределяется**
- параметр конфигурации `default_toast_compression` устанавливает алгоритм сжатия:
- `ALTER SYSTEM SET default_toast_compression = pglz;`

Параметры `toast_tuple_target` и `default_toast_compression`

На вытеснение влияют два макроса, установленные в исходном коде (`heaptoast.h`):
TOAST_TUPLE_THRESHOLD и TOAST_TUPLE_TARGET, которые имеют одинаковые значения. Если размер строки больше TOAST_TUPLE_THRESHOLD, то начинается сжатие и/или вытеснение полей строки.

Поля будут сжиматься и рассматриваться на предмет хранения в TOAST, пока оставшаяся часть строки (полная: с заголовком строки) не поместится в TOAST_TUPLE_TARGET. Значение можно переопределить на уровне таблицы:

```
ALTER TABLE t SET (toast_tuple_target = 2032);
```

TOAST_TUPLE_THRESHOLD не переопределяется.

Также есть параметр, устанавливающий алгоритм сжатия pglz или lz4:

```
default_toast_compression
```

Константы определены в исходном коде:

```
#define MaximumBytesPerTuple(tuplesPerPage) MAXALIGN_DOWN((BLCKSZ - MAXALIGN(SizeOfPageHeaderData + (tuplesPerPage) * sizeof(ItemIdData)))/(tuplesPerPage))
#define TOAST_TUPLES_PER_PAGE 4
#define TOAST_TUPLE_THRESHOLD MaximumBytesPerTuple(TOAST_TUPLES_PER_PAGE)
#define TOAST_TUPLE_TARGET TOAST_TUPLE_THRESHOLD
```

Параметры заголовка блока:

```
ItemIdData = 4 байта
SizeOfPageHeaderData = 24 байта
```

Если подставить значения, то получится:

TOAST_TUPLE_TARGET=TOAST_TUPLE_THRESHOLD=MAXALIGN_DOWN((BLCKSZ - MAXALIGN(24 + (4) * sizeof(4)))/(4))=MAXALIGN_DOWN((BLCKSZ - MAXALIGN(24 + 4*4))/4)=MAXALIGN_DOWN((8192 - MAXALIGN(40))/4)=MAXALIGN_DOWN((8192 - 40)/4)=MAXALIGN_DOWN(2038)=**2032**.

TOAST_TUPLE_TARGET также определяет максимальный размер строк TOAST таблиц. Заголовок строки обычной и TOAST таблицы 24 байта. Размер области данных строки TOAST-таблицы 2032-24=2008 байт. В строке три поля: oid (4 байта), int4 (4 байта), bytea. В bytea первый байт в начале поля переменной ширины хранит длину поля длиной до 127 байт, первые 4 байта в начале поля переменной ширины хранят длину поля для bytea длиннее 126 байт. Выравнивание по 4 байта. 2008-4-4-4=1996.

Оптимизация Heap Only Tuple

- при обновлении (`UPDATE`) строки в индексы могут не вноситься изменения
- изменения не выходят за пределы блока таблицы (heap only)
- условия:
  - изменяются только поля, не входящие ни в один из индексов (кроме индексов типа brin) на таблицу
  - новая версия строки размещается в том же блоке, что и прежняя версия
- если новая версия строки размещается в блоке, отличном от того в котором находится прежняя версия строки, то:
  - прежняя версия станет последней в цепочке HOT-версий
  - во всех индексах на таблице будут созданы новые записи, указывающие на новую версию строки

Оптимизация Heap Only Tuple

При обновлении строки создается новая строка внутри блока таблицы. Если поля, вынесенные в TOAST не изменялись, то содержимое полей, ссылающихся на TOAST будут скопированы без изменений и в TOAST изменений не будет.

Если на какие-то столбцы таблицы созданы индексы, то индексные записи указывают на ctid прежней версии строки. Индексные записи указывают на поле в заголовке блока.

Если **меняются только поля, не упомянутые ни в одном из индексов (кроме индексов типа brin)**, то в индексы не вносятся изменения. Частичный индекс:

```
create index t5_idx on t5 (c1) where c1 is not null;
```

не дает выполнять HOT, если в команде UPDATE упоминается столбец c1 даже, если в UPDATE стоит условие WHERE c1 is null.

Аналогично частичный покрывающий индекс:

```
create index t5_idx1 on t5 (c1) include (c2) where c1 is not null;
```

не дает выполнять HOT, если в команде UPDATE упоминаются столбцы c1 и c2.

При HOT поле t_ctid ссылается на новую версию строки. В заголовке старой строки в t_infomask2 устанавливается бит HEAP_HOT_UPDATED, а у новой версии строки устанавливается бит HEAP_ONLY_TUPLE.

Из индекса серверный процесс попадает на старую версию строки, видит бит HEAP_HOT_UPDATED, переходит по полю t_ctid на новую версию строки (с учетом правил видимости, если ему видна эта версия, то серверный процесс на ней и останавливается), проверяет тот же бит, если он установлен, то переходит дальше на более новую версию строки. Такие версии строки называются HOT-цепочкой версий (HOT chain). С учетом правил видимости серверный процесс может дойти до самой свежей версии строки, на которой установлен бит HEAP_ONLY_TUPLE и остановиться на ней.

Если новая версия строки размещается в блоке, отличном от того в котором находится прежняя версия строки, то HOT не применяется. Поле t_ctid прежней версии будет ссылаться на более новую версию в другом блоке, но бит HEAP_HOT_UPDATED не будет установлен. Прежняя версия станет последней в цепочке HOT-версий. Во всех индексах на таблице будут созданы новые записи, указывающие на новую версию строки.

https://www.cybertec-postgresql.com/en/hot-updates-in-postgresql-for-better-performance/

Мониторинг HOT update

- статистика HOT доступна в представлениях `pg_stat_all_tables` и `pg_stat_user_tables`
- **счетчик HOT обновлений собирается по каждой таблице и отражается в столбце: `n_tup_hot_upd`**
- все обновления отражаются в столбце `n_tup_upd`
- **случаи, если при обновлении не нашлось места для новой версии строки и цепочка HOT была оборвана, а новая версия была вставлена в другой блок показывает `n_tup_newpage_upd`**
- статистика по базе обнуляется вызовом функции `pg_stat_reset();`
  - после вызова функции рекомендуется выполнить ANALYZE по всей базе

```
 select relname, n_tup_upd, n_tup_hot_upd, n_tup_newpage_upd,
round(n_tup_hot_upd*100/n_tup_upd,2) as hot_ratio from pg_stat_all_tables where n_tup_upd<>0
order by 5;
     relname     | n_tup_upd | n_tup_hot_upd | n_tup_newpage_upd | hot_ratio
-----------------+-----------+---------------+-------------------+-----------
  pg_rewrite     |        14 |             9 |                 5 |     64.00
  pg_proc        |        33 |            23 |                10 |     69.00
  pg_class       |     71645 |         63148 |              8351 |     88.00
```


Мониторинг HOT update

Статистика HOT доступна в двух представлениях **`pg_stat_all_tables` и `pg_stat_user_tables`**:

```
 select relname, n_tup_upd, n_tup_hot_upd, n_tup_newpage_upd,
round(n_tup_hot_upd*100/n_tup_upd,2) as hot_ratio
 from pg_stat_all_tables where n_tup_upd<>0 order by 5;
     relname     | n_tup_upd | n_tup_hot_upd | n_tup_newpage_upd | hot_ratio
-----------------+-----------+---------------+-------------------+-----------
  pg_rewrite     |        14 |             9 |                 5 |     64.00
  pg_proc        |        33 |            23 |                10 |     69.00
  pg_class       |     71645 |         63148 |              8351 |     88.00
  pg_attribute   |       270 |           267 |                 3 |     98.00
```

Статистика накапливается с момента последнего вызова функции `pg_stat_reset()`.

`pg_stat_reset()` обнуляет счетчики накопительной статистики по текущей базе, но не обнуляет счетчики уровня кластера. **Обнуление счетчиков обнуляет счетчики по которым автовакуум решает когда нужно запустить вакуумирование и анализ. После вызова функции рекомендуется выполнить ANALYZE по всей базе.** Статистики уровня кластера, накопленные в представлениях `pg_stat_*` обнуляют ("сбрасывают") вызовы функции:

```
select pg_stat_reset_shared('recovery_prefetch');
select pg_stat_reset_shared('bgwriter');
select pg_stat_reset_shared('archiver');
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('wal');
```

Начиная с 17 версии `pg_stat_reset_shared(null)` сбрасывает все эти кэши, в 16 версии ничего не делает.

Как выполнять мониторинг? Например, создали дополнительный индекс или увеличили количество секций секционированной таблицы, стоит проверить как поменялся процент HOT обновлений. n_tup_hot_upd - счетчик обновлений HOT, n_tup_upd - все обновления.

Приблизительная оценка числа мертвых строк:

```
 select relname, n_live_tup, n_dead_tup from pg_stat_all_tables where
n_dead_tup<>0 order by 3 desc;
```

Код очистки реализован в heapam.c и pruneheap.c: "We prune when a previous UPDATE failed to find enough space on the page for a new tuple version, or when free space falls below the relation's fill-factor target (but not less than 10%)".

https://postgres.ai/blog/20211029-how-partial-and-covering-indexes-affect-update-performance-in-postgresql

Влияние FILLFACTOR на HOT cleanup

- чтобы HOT cleanup выполнился, предыдущий UPDATE строки должен превысить границу `min(90%, FILLFACTOR)`
- если размер строк в таблице такой, что в блок помещается меньше 9 строк, то восьмая строка не превысит границу 90%, а девятая строка будет больше 11% размера блока и она не поместится в блок
- установка FILLFACTOR в значение, отличное от значения по умолчанию (FILLFACTOR=100) в большинстве случаев бесполезно и только увеличивает размер файлов таблицы
- наиболее эффективно при проектировании схем хранения данных делать размер строк небольшими. Разумный размер строки не больше 600 байт

Влияние FILLFACTOR на HOT cleanup

Быстрая очистка (HOT cleanup) важна и во многих случаях активно работает. Если условия HOT соблюдены, то при обновлении строк в блоке новая версия ищет место в блоке и создается цепочка версий. Если вставляемая новая версия строки поместится в блок и при этом процент заполнения превысит границу min(90%, FILLFACTOR), то в заголовке блока будет выставлен признак, что блок можно очистить. Следующее обновление строки блока выполнит HOT cleanup - очистит блок от строк в цепочке версий, вышедших за горизонт базы и новая версия строки скорее всего поместится в блок.

Но если процент заполнения не превысил границу min(90%, FILLFACTOR), а новая версия не помещается в оставшееся место в блоке, то быстрая очистка не выполняется, версия строки вставляется в другой блок, цепочка HOT прерывается, в заголовок блока вставляется флаг, что в блоке нет места. Такое будет происходить, если в блоке меньше 9 строк и FILLFACTOR=100% (значение по умолчанию). В таком случае, возможно, стоит установить FILLFACTOR в значение, при котором новая версия строки помещалась в блоке и при этом переходила границу FILLFACTOR. Не стоит проектировать таблицы, чтобы размер строк был настолько большим, что в блок помещалось меньше 6 строк.

```
create table t(s text storage plain) with (autovacuum_enabled=off);
insert into t values (repeat('a',2010));
update t set s=(repeat('c',2010)) where ctid::text = '(0,1)';
update t set s=(repeat('c',2010)) where ctid::text = '(0,2)';
update t set s=(repeat('c',2010)) where ctid::text = '(0,3)';
select ctid,* from heap_page('t',0);
```

```
 ctid  | lp_off | ctid  | state  | xmin  | xmax  | hhu | hot | t_ctid | multi
-------+--------+-------+--------+-------+-------+-----+-----+--------+-------
 (0,1) | 6136   | (0,1) | normal | 1001c | 1002c | t   |     | (0,2)  | f
 (0,2) | 4096   | (0,2) | normal | 1002c | 1003c | t   | t   | (0,3)  | f
 (0,3) | 2056   | (0,3) | normal | 1003c | 1004  |     | t   | (1,1)  | f
(3 rows)
```

```
select ctid from t;
 ctid
-------
 (1,1)
```

Четвертая версия строки была вставлена во второй блок.

График на слайде будет рассмотрен в практике к этой главе.

Внутристраничная очистка в таблицах

- выполняя SELECT и UPDATE, серверный процесс может удалить dead tuples (версии строк, вышедшие за горизонт видимости базы данных, xmin horizon), выполнив реорганизацию версий строк внутри блока
- внутристраничная очистка совместима с HOT и может предварительно освобождать место, которое будет использоваться новыми версиями строк, появляющимися в результате UPDATE
- выполняется, если:
  - блок заполнен более чем на 90% или FILLFACTOR (по умолчанию 100%)
  - ранее выполнявшийся UPDATE не смог разместить новую версию строки в этом блоке
- приблизительная оценка по версиям строк, которые могут быть очищены: `pg_stat_all_tables.n_dead_tup`

Внутристраничная очистка в таблицах

Серверный процесс выполняя SELECT и другие команды может удалить dead tuples (версии строк, вышедшие за горизонт видимости базы данных, xmin horizon), выполнив реорганизацию версий строк внутри блока. Это называется внутристраничной очисткой.

HOT cleanup/pruning выполняется, если выполняется одно из условий:

блок заполнен более чем на 90% или FILLFACTOR (по умолчанию 100%):

```
minfree = Max(minfree, BLCKSZ / 10);
```

Ранее выполнявшийся UPDATE не нашел места (то есть установил в заголовке блока подсказку `PD_PAGE_FULL`).

Внутристраничная очистка работает в пределах одной табличной страницы, не очищает индексные страницы (в индексных страницах есть аналогичный алгоритм), не обновляет карту свободного пространства и карту видимости.

Внутристраничная очистка не является основным способом очистки и была создана для того, чтобы хоть как-то очищать страницы в случае, если автовакуум не справлялся или не мог работать (In fact, page pruning was designed specifically for cases where autovacuum wasn't running or couldn't keep up)

Указатели (4 байта) в заголовке блока не освобождаются, они обновляются чтобы указывать на актуальную версию строки. Указатели освободить нельзя, так как на них могут существовать ссылки из индексов, это проверить серверный процесс не может. Только вакуум сможет освободить указатели (сделать указатели unused), чтобы указатель мог снова использоваться. В области данных версии место dead tuples очищается и остальные строки сдвигаются.

Не стоит в одной транзакции несколько раз обновлять одну и ту же строку, так как место, занимаемое порождаемыми версиями не сможет очиститься. Такое может произойти при использовании триггеров AFTER.. FOR EACH ROW. Стоит проверять код приложения на присутствие фраз FOR EACH ROW и DEFERRABLE INITIALLY DEFERRED.

https://www.cybertec-postgresql.com/en/reasons-why-vacuum-wont-remove-dead-rows/

Внутристраничная очистка в индексах (повторение)

- выполняется при индексном сканировании
- если строка в таблице удалена, индексная запись на строку или цепочку версий может быть помечена флагом LP_DEAD
- пометка может ставиться командой SELECT
- журнальная запись не создается, но блок грязнится
- помеченная индексная запись игнорируется на мастере, но не на реплике
- помеченная индексная запись будет очищена при выполнении команд, которые меняют данные в таблице

```
create table t (id int primary key, c text) with (autovacuum_enabled = off);
insert into t SELECT i, 'simple delete ' || i from generate_series(1, 1000000) as i;
delete from t where id between 100 and 900000;
analyze t;
explain (analyze, buffers, costs off) select * from t where id between 1 and 900000;
 Index Scan using t_pkey on t (actual time=0.010..218.477 rows=99 loops=1)
   Buffers: shared hit=11489
 Execution Time: 218.600 ms
```


Внутристраничная очистка в индексах (повторение)

Если при индексном сканировании (Index Scan) серверный процесс обнаружит, что строка (или цепочка строк, на которую ссылается индексная запись) удалена и вышла за горизонт базы, то в индексной записи листового блока (leaf page) в lp_flags устанавливается бит-подсказка LP_DEAD (называют known dead, killed tuple). Бит можно посмотреть в столбце `dead`, **выдаваемый функцией `bt_page_items('t_idx',блок)`**. При Bitmap Index Scan и Seq Scan не устанавливается. Помеченная таким флагом строка будет удалена позже при выполнении команды, которая вносит изменения в блок индекса. Почему место в индексе не освобождается сразу? Индексное сканирование выполняется SELECT, который устанавливает разделяемые блокировки на объект и страницы. Биты-подсказки как в индексных блоках (flags), так и в блоках таблиц (infomask и infomask2) могут меняться с такими блокировками. Для остальных изменений в блоке нужна эксклюзивная блокировка на блок и другая блокировка на сам объект. Устанавливать их SELECT не будет. Из-за этого пометка записи и освобождение места разнесены во времени.

Возврат в блок и установка в нем флага добавляет накладные расходы и увеличивает время выполнения команды, но делается однократно. Зато последующие команды смогут игнорировать индексную запись и не будут обращаться к блоку таблицы.

На репликах в блок не могут вноситься никакие изменения и SELECT на репликах биты-подсказки не устанавливает. Более того на репликах игнорируется LP_DEAD ("ignore_killed_tuples"), установленный на мастере. Изменение бита LP_DEAD не журналируется, но блок грязнится и передаётся по full_page_writes. Из-за этой особенности **запросы на реплике могут выполняться на порядок медленнее, чем на мастере**. После отработки автовакуума на мастере и применении журнальных записей, сгенерированных автовакуумом на реплике, разницы в скорости не будет.

Пример SELECT с установкой битов на 899900 удаленных строк в 7308 блоках таблицы:

`Buffers: shared hit=11489` читаются блоки индекса и таблицы

`Execution Time: 218.600 ms`

Тот же SELECT повторно на еще не очищенных блоках:

`Buffers: shared hit=2463` читались блоки индекса и несколько блоков таблицы

`Execution Time: 8.607 ms`

После REINDEX или вакуумирования таблицы (результат примерно одинаков):

`Buffers: shared hit=6` читались несколько блоков индекса и таблицы

`Execution Time: 0.373 ms`

8-2

Типы данных

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

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

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

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

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

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

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

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

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

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

Типы данных переменной длинны

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

Типы данных переменной длинны

Следующими по компактности идут типы данных переменной длинны.

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

Занимаемое место:

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

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

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

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

Пример:

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

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

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

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

Целочисленные типы данных

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

Целочисленные типы данных

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

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

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

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

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

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

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

Выбор типов данных для первичного ключа

- при меньшем числе строк размер индекса по столбцу типа uuid существенно больше, чем по столбцу типа bigint
- дополнительные функции устанавливается расширением:

```
create extension uuid-ossp;
ERROR:  syntax error at or near "-"
create extension "uuid-ossp";
CREATE EXTENSION
```

- размер индекса по столбцу первичного ключа и пример теста:

```
psql -c "create table tt1 (id bigint generated by default as identity (cache 60) primary key, data bigint);"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
tps = 2693
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from tt1;"
 count | pg_indexes_size | pg_table_size
-------+-----------------+---------------
 80760 |         1884160 |       3612672
psql -c "create table tt1 (id uuid default gen_random_uuid() primary key, data bigint);"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
tps = 2338
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from tt1;"
 count | pg_indexes_size | pg_table_size
-------+-----------------+---------------
 70115 |         2777088 |       3760128
```


Выбор типов данных для первичного ключа

При использовании в качестве уникального ключа типа uuid (за исключением генерируемых функцией `uuidv7()` расширения `pg_uuidv7`, имеющегося в СУБД Tantor начина с версии 16.6) нужно помнить, что размер поля для этого типа 16 байт и не будет работать оптимизация: "The fastpath optimization avoids most of the work of searching the tree repeatedly when a single backend inserts successive new tuples on the rightmost leaf page of an index" (`nbtinsert.c`). Также вставка случайных значений, а не возрастающих, в листовые блоки индекса приводит к увеличению объема full page writes в журналах.

Пример сравнения `bigint` и `uuid`:

```
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id bigint generated by default as identity (cache 60) primary key, data bigint);"
echo "insert into tt1(data) values(1);" >txn.sql
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from tt1;"
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id uuid default gen_random_uuid() primary key, data bigint);"
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from tt1;"
tps = 2693
 count | pg_indexes_size | pg_table_size
 80760 |         1884160 |       3612672
tps = 2338
 count | pg_indexes_size | pg_table_size
 70115 |         2777088 |       3760128
```

При меньшем числе строк размер индекса по столбцу типа uuid **существенно** больше, чем по столбцу типа bigint. Второй столбец заполняется константой, а не функцией, чтобы не измерять скорость выполнения функции.

Вместо `gen_random_uuid()` можно использовать `uuid_generate_v4()`.

Установка расширения с дополнительными функциями:

```
create extension uuid-ossp;
ERROR:  syntax error at or near "-"
create extension "uuid-ossp";
CREATE EXTENSION
```

`uuidv7()` возможно появится в PostgreSQL https://commitfest.postgresql.org/47/4388/

Пример тестов: https://ardentperf.com/2024/02/03/uuid-benchmark-war/

Параметр cache у последовательностей

- параметр последовательностей и identity столбцов
- по умолчанию `cache=1`

```
create table tt1 (id bigint generated by default as identity (cache 1) primary key)
```

- не стоит увеличивать значение
- выборка значений из последовательности не является узким местом
- установка в большее значение приводит к неэффективным вставкам в индекс, созданным на столбец в который вставляются сгенерированные значения
- **увеличение `cache` приводит к замедлению вставок в индекс**

Параметр cache у последовательностей

Параметр последовательностей и identity столбцов. По умолчанию cache=1. Устанавливается у последовательностей и identity столбцов. Определяет сколько значений будет кэшировать серверный процесс в своей памяти для будущих вставок. Увеличение значения больше 1 ухудшает производительность. Причина в том, что если несколько сессий кэшируют диапазоны значений, велика вероятность, что вставки пойдут не в самый правый листовой блок индекса. Если вставки идут не в самый правый блок индекса, то структура индекса становится неэффективной, индекс становится большего размера, чем мог бы быть. Выборка значений из последовательности не является узким местом и быстрая операция. **Смысла использовать `cache>1` нет.**

В части 8 практики к этой главе вы измерите скорость вставок при разных значениях cache.

Использование параметра cache было бы оправдано, если бы кластер баз данных обслуживало несколько экземпляров на разных узлах и между экземплярами надо было бы синхронизировать доступ к последовательности.

Хранение дат, времени, их интервалов

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

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


Хранение дат, времени, их интервалов

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

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

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

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

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

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

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

Функции проверки типа данных и размера поля

- для проверки типа значения используется функция `pg_typeof(значение)`
- функция получения размера поля и полного размера строки `pg_column_size(строка или поле)`
- при хранении строки в блоке ее размер должен быть кратен 8 байт, если он меньше, то будет добавлено от 1 до 7 байт
- для датавременных типов есть тип date (4 байта, с точностью до дня), остальные типы `timestamp, timestamptz, time` имеют точность до микросекунды, размер 8 байт. Типы `timetz` и `interval` имеют длину 12 и 16 байт и из-за длинны не стоит их использовать.

```
pg_typeof(now()) -> timestamp with time zone
pg_typeof(now()::date) -> date
pg_typeof(current_date) -> date
pg_typeof(1.1) -> numeric
pg_column_size(1.1) -> 10
pg_column_size(1.1::float) -> 8
интервальный тип: pg_column_size(interval '1s') -> 16
размер строки из трёх полей  в удачном порядке: pg_column_size(row(true::boolean, 1::int4, 1::int8)) -> 40
неудачная перестановка столбцов: pg_column_size(row(true::boolean, 1::int8, 1::int4)) -> 44
размер строки с ещё одним порядком: select pg_column_size(row(1::int4, 1::int8, true::boolean))  -> 41;
```


Функции проверки типа данных и размера поля

Для проверки типа значения используется функция `pg_typeof(значение)`. Она полезна для того, чтобы увидеть будет ли неявное приведение типа.

Для того, чтобы быстро получить размер поля или строки используется функция `pg_column_size(строка или поле)`. При хранении строки в блоке она может занять больший размер, так чтобы занимаемое место было кратно 8 байт. Выдает размер в байтах.

Если `pg_column_size` применяется к хранящемуся полю таблицы и поле было сжато, то выдаст размер в сжатом виде:

```
select pg_column_size(relname) from pg_class limit 1;
 pg_column_size
----------------
             64
```

Для датавременных типов есть тип date (4 байта, с точностью до дня), остальные типы `timestamp, timestamptz, time` имеют точность до микросекунды, размер 8 байт. Типы `timetz` и `interval` имеют длину 12 и 16 байт и из-за длинны не стоит их использовать:

```
select pg_column_size(interval '1year');
 pg_column_size
----------------
             16
```

Функция `now()` выдает timestamptz начала транзакции. Для получения date (например, в DEFAULT или ограничениях целостности CHECK) начала транзакции также можно использовать функцию `current_date`, описанную в стандарте SQL как вызываемую без круглых скобок:

```
create table t (c date constraint c check(c<=now()), c1 date constraint c1
check(c1<=current_date), c2 date constraint c2 check(c2<=current_timestamp));
```

Для получения в виде `timestamptz` момента **начала команды** нужно использовать функцию `statement_timestamp();`

Для получения **текущего** (на момент вызова) timestamptz нужно использовать функцию `clock_timestamp();`

Знание момента, на который выдается время важно при вычислениях показателей производительности.

Типы данных для вещественных чисел

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

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

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

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

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

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


Типы данных для вещественных чисел

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

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

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

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

```
select (1234567890123456789.1234567890123456789::float8 +
123456789::float8)::numeric;
        12345678901234600000000000000
```

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

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

При округлении float8 учитывается шестнадцатый разряд:

```
select 1234567890123449999::float8::numeric, 1234567890123444999::float8::numeric;
       12345678901234500     |              12345678901234400
select 0.12345678901234499::float8::numeric, 0.12345678901234499::float8::numeric;
       0.123456789012345     |               0.123456789012344
```

При округлении float4 учитывается седьмой разряд:

```
select 123449::float4::numeric, 123444::float4::numeric;
       1234500 |                123450
select 0.123449::float4::numeric, 0.123444::float4::numeric;
       0.12345 |                  0.12345
```

Параметр конфигурации `extra_float_digits`

- параметром `extra_float_digits` можно уменьшить число цифр в текстовом представлении чисел float8, float4 и геометрических типов
- диапазон значений от -15 до 3 включительно
- значения 1,2,3 эквивалентны
- параметр влияет только на отображение, на вычисления и приведения к типу numeric не влияет

```
show extra_float_digits;
 1
select 1234567890.123456789::float8, 1.123456789::float4;
       1234567890.1234567     |      1.1234568
set extra_float_digits = 0;
select 1234567890.123456789::float8, 1.123456789::float4;
       1234567890.12346       |      1.12346
set extra_float_digits = -5;
select 1234567890.123456789::float8, 1.123456789::float4;
       1234567890             |      1
reset extra_float_digits;
select 234567890.199999989::float8::numeric, 1.19999999123::float4::numeric;
       234567890.2            |              1.2
```


Параметр конфигурации `extra_float_digits`

Влияет на текстовое представление `float8, float4` и геометрических типов. Значение по умолчанию 1. Значения параметра `extra_float_digits` 1,2,3 эквивалентны:

```
show extra_float_digits;
1
select 1234567890.123456789::float8, 1.123456789::float4;
       1234567890.1234567     |      1.1234568
set extra_float_digits = 3;
select 1234567890.123456789::float8, 1.123456789::float4;
       1234567890.1234567     |      1.1234568
```

Значение ноль и отрицательные значения убирают из вывода разряды с округлением:

```
set extra_float_digits = 0;
select 1234567890.123456789::float8, 1.123456789::float4;
       1234567890.12346       |      1.12346
set extra_float_digits = -1;
select 1234567890.123456789::float8, 1.123456789::float4;
       1234567890.1235        |      1.1235
set extra_float_digits = -2;
select 1234567890.123456789::float8, 1.123456789::float4;
       1234567890.123         |      1.123
set extra_float_digits = -5;
select 1234567890.123456789::float8, 1.123456789::float4;
       1234567890             |      1
```

Параметр влияет только на представление (отображение, вывод). На вычисления и приведения к типу numeric не влияет:

```
select 1234567890.123456789::float8::numeric, 1.123456789::float4::numeric;
       1234567890.12346       |               1.12346
```

Округление может убрать много разрядов:

```
reset extra_float_digits;
select 234567890.199999989::float8::numeric, 1.19999999123::float4::numeric;
       234567890.2            |              1.2
```

Хранение вещественных чисел

- обычно в столбцах хранятся хранятся числа небольшой размерности
- тип numeric для небольших целых чисел хранит данные компактнее, чем float8 (точность 15 цифр после точки)
- типы float8 и float4 теряют точность целочисленных разрядов, numeric не теряет
- точности float4 может быть недостаточно: 6 десятичных разрядов
- точность float8 15 десятичных разрядов

```
create table t5( c1 double precision, c2 real, c3 numeric);
insert into t5 values (1,1,1),(1.0/3, 1.0/3, 1.0/3),(1111,1111,1111),(1111.11,1111.11, 1111.11);
select lp_off, lp_len, t_hoff, t_data from heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
lp_off |lp_len |t_hoff |                    t_data
-------+-------+-------+-----------------------------------------------
 7984 |    43 |    24 |\x3d0ad7a3705c914085e38a440f008157044c04
 8032 |    41 |    24 |\x00000000005c914000e08a440b00805704
 8080 |    49 |    24 |\x555555555555d53fabaaaa3e1b7f8a050d050d050d050d050d
 8136 |    36 |    24 |\x000000000000f03f0000803f0b00800100
```


Хранение вещественных чисел

При выборе типа данных для хранения вещественных чисел стоит учесть, что тип numeric имеет переменную длину и для небольших чисел хранит данные компактнее, чем float8 : точность 15 "десятичных разрядов", цифр в десятичном виде, то есть цифр до и после точки в десятичном виде, то есть если разрядов не хватает убираются десятичные и потом целочисленные цифры и заменяются нулями.

Точности float4 (real) может быть недостаточно: 6 цифр.

```
drop table if exists t5;
create table t5( c1 double precision, c2 real, c3 numeric);
insert into t5 values
  (1,1,1),
  (1.0/3, 1.0/3, 1.0/3),
  (1111,1111,1111),
  (1111.11,1111.11, 1111.11);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
 lp_off|lp_len|t_hoff|                    t_data
 ------+------+------+-----------------------------------------------
  8136 |    36 |    24 |\x000000000000f03f0000803f0b00800100
  8080 |    49 |    24 |\x555555555555d53fabaaaa3e1b7f8a050d050d050d050d050d
  8032 |    41 |    24 |\x00000000005c914000e08a440b00805704
  7984 |    43 |    24 |\x3d0ad7a3705c914085e38a440f008157044c04
```

Цветом выделены поля трёх типов.

Функция проверки размера поля более наглядно показывает занимаемое полями место:

```
select pg_column_size(c1), pg_column_size(c2), pg_column_size(c3) from t5;
 pg_column_size | pg_column_size | pg_column_size
----------------+----------------+----------------
              8 |              4 |              5
              8 |              4 |             13
              8 |              4 |              5
              8 |              4 |              7
```

Разрядность результата деления numeric

```
* For inherently inexact calculations such as division and square root,
* we try to get at least this many significant digits; the idea is to
* deliver a result no worse than float8 would.
#define NUMERIC_MIN_SIG_DIGITS          16
```

- не менее 16 значащих цифр, то есть не хуже, чем float8
- не меньше, чем разрядность любого из входных параметров

```
insert into t5 values (1,1, 1.0000000000000000000000000000000001/3);
select lp, lp_off, lp_len, t_hoff, t_data from heap_page_items(get_raw_page('t5','main',0)) order by
lp desc limit 1;
 lp | lp_off | lp_len | t_hoff |                                                                                t_data
----+--------+--------+--------+-------------------------------------------------------------------------------------
  5 |   7912 |     57 |     24 | \x000000000000f03f0000803f2b7f92050d050d050d050d050d050d050d060d
(5 rows)
select * from t5 order by ctid desc limit 1;
 c1 | c2 |                  c3
----+----+-----------------------------------------
  1 |  1 | 0.33333333333333333333333333333333333334
```

- Посмотреть описание операторов можно командой psql:

```
\doS+ /                                    List of operators
   Schema   | Name |  Left arg type |  Right arg type |   Result type |   Function  | Description
------------+------+----------------+-----------------+---------------+-------------+-------
 pg_catalog | /    | bigint         | bigint          | bigint        | int8div     | divide
 pg_catalog | /    | numeric        | numeric         | numeric       | numeric_div | divide
...
```


Разрядность результата деления numeric

Разрядность результата деления двух чисел типа numeric:
1) не менее 16 значащих цифр, то есть не хуже, чем float8
2) не меньше, чем разрядность любого из входных параметров.

Для вычисления квадратного корня и других операций с потерей точности действует аналогичное правило. Для операторов сложения, вычитания, умножения и других потери точности нет.

Пример:

```
insert into t5 values (1,1, 1.0000000000000000000000000000000001/3);
select lp, lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0))
order by lp desc limit 1;
 lp | lp_off | lp_len | t_hoff |                    t_data
----+--------+--------+--------+-----------------------------------
  5 |   7912 |     57 |     24 |
\x000000000000f03f0000803f2b7f92050d050d050d050d050d050d050d060d
(5 rows)
select * from t5 order by ctid desc limit 1;
 c1 | c2 |                  c3
----+----+-----------------------------------------
  1 |  1 | 0.33333333333333333333333333333333333334
```

Посмотреть описание операторов можно командой psql:

```
\doS+ /
                                        List of operators
   Schema   | Name |  Left arg type |  Right arg type |   Result type |   Function  | Descri
------------+------+----------------+-----------------+---------------+-------------+-------
 pg_catalog | /    | bigint         | bigint          | bigint        | int8div     | divide
 pg_catalog | /    | numeric        | numeric         | numeric       | numeric_div | divide
...
```

Комментарий в файле `numeric.h` исходного кода PostgreSQL:

```
/*
 * For inherently inexact calculations such as division and square root,
 * we try to get at least this many significant digits; the idea is to
 * deliver a result no worse than float8 would.
 */
#define NUMERIC_MIN_SIG_DIGITS          16
```

Практика

- Часть 1. TOAST
- Часть 2. Структура таблиц TOAST
- Часть 3. Эффективность UPDATE в сравнении с INSERT
- Часть 4. HOT cleanup
- Часть 5. Мониторинг HOT cleanup
- Часть 6. Типы данных небольшого размера
- Часть 7. Хранение типов данных переменной длинны
- Часть 8. Тип данных для столбца первичного ключа
- Часть 9. Типы данных для хранения дат и времени
- Часть 10. Типы данных float и real

Практика

В практике вы посмотрите случаи замедления работы команд при наличии триггеров и оцените эффекты использования разных типов триггеров. Замена FOR EACH ROW на FOR EACH STATEMENT может дать прирост производительности в несколько раз.

Вы посмотрите пример, когда замена логики приложения с UPDATE на INSERT и DELETE дает прирост производительности в сотню раз.

Вы посмотрите механизм деградации производительности при увеличении числа кэшированных значений последовательности (параметр CACHE).

Практика полезна для того, чтобы понять как PostgreSQL хранит данные и работает с ними. Это полезно для тех, кто проектирует структуры хранения (ER-моделирование).

9

9-1

Архитектура

Запуск экземпляра, процесс postgres

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

Запуск экземпляра, процесс postgres

Основные шаги при запуске экземпляра:

1. Запускается процесс postgres ("postmaster"). Устанавливаются параметры LC_MONETARY="C", LC_NUMERIC="C", LC_TIME="C", убирается unserenv LC_ALL, переменные LC_COLLATE, LC_CTYPE, LC_MESSAGES устанавливаются в одноименные переменные окружения. LC_MESSAGES (сообщения в stdout, stderr или лог) позже будет выставлена по параметру конфигурации lc_messages
2. Считываются файлы параметров конфигурации, параметры комбинируются с параметрами командной строки, переданными postmaster и переменными окружения.
3. Проверяются права на директорию PGDATA, они должны быть 0700 или 0750
4. Проверяется наличие файла pg_control, текущей директорией для процесса устанавливается PGDATA, в ней создается файл postmastr.pid, инициализируется TLS, загружаются разделяемые библиотеки по параметру shared_preload_libraries, регистрируется обработчик на случай исчезновения процесса для корректной остановки дочерних процессов, инициализируется (по параметрам конфигурации) менеджер памяти, регистрируется обработчик закрытия сетевых сокетов

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

5. регистрируются сокеты по всем адресам (параметр конфигурации listen_addresses). Создается файл UNIX-сокета. LockFile добавляется хост первого успешно открывшегося сокета (TCP или UNIX). Для сокетов будут установлены параметры TCP_NODELAY и TCP_KEEPALIVE
6. Считывается файл pg_hba.conf
7. Запускается процесс startup и ему выставляется статус STARTUP_RUNNING, а процесс postgres выставляет себе статус PM_STARTUP. Начинается запуск фоновых процессов.

Postmaster хранит PID восьми основных фоновых процессов:

```
/* PIDs of special child processes; 0 when not running */
static pid_t StartupPID = 0, BgWriterPID = 0, CheckpointerPID = 0, WalWriterPID
= 0,WalReceiverPID = 0, AutoVacPID = 0, PgArchPID = 0, SysLoggerPID = 0;
```

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

Процесс startup

- процесс startup определяет состояние кластера по управляющему файлу. Возможные состояния:
  - "shut down" после корректной (с контрольной точкой) остановки экземпляра
  - "shut down in recovery" экземпляр реплики сбойнул, когда находился в режиме восстановления
  - "in production" экземпляр был остановлен без контрольной точки
  - "shutting down" экземпляр сбойнул в процессе остановки
  - "in crash recovery" экземпляр мастера сбойнул при восстановлении
  - "in archive recovery" экземпляр сбойнул в процессе управляемого восстановления
  - "unrecognized status code" состояние неизвестно
- процесс startup выполняет синхронизацию файлов PGDATA и их восстановление по файлам WAL

```
postgres@tantor:~$ pg_controldata | grep state
Database cluster state:               shut down
```


Запуск процесса startup

Процесс startup выполняет восстановление кластера. Действия запрограммированы в функции `StartupXLOG()`. Если процесс работает дольше, чем указано в параметре конфигурации `log_startup_progress_interval=10s` , то с частотой заданной этим же параметром процесс startup будет записывать в а лог кластера информацию о том, что он делает. Состояние экземпляра можно посмотреть в управляющем файле:

```
pg_controldata | grep state
Database cluster state:               in production
```

1) В лог записывается состояние экземпляра, определённое по управляющему файлу. Список возможных состояний по управляющему файлу (`pg_control`) и записями в логе:

a) `"shut down"` запись в логе: `database system was shut down at` после корректной (с контрольной точкой) остановки экземпляра

b) `"shut down in recovery" database system was shut down in recovery at` экземпляр реплики сбойнул, когда находился в режиме восстановления

c) `"in production" database system was interrupted; last known up at` экземпляр был остановлен в режиме immediate или сбойнул (например, выключилось питание)

d) `"shutting down" database system shutdown was interrupted; last known up at` экземпляр сбойнул в процессе остановки

e) `"in crash recovery" database system was interrupted while in recovery at время. HINT: This probably means that some data is corrupted and you will have to use the last backup for recovery.` экземпляр мастера сбойнул при восстановлении ("crash recovery")

f) `"in archive recovery" database system was interrupted while in recovery at log time время. HINT: If this has occurred more than once some data might be corrupted and you might need to choose an earlier recovery target.` экземпляр сбойнул в процессе управляемого восстановления

g) `"unrecognized status code" control file contains invalid database cluster state` - состояние неизвестно

2) Для всех состояний кроме "a" и "b" из `pg_wal` удаляются недосозданные WAL-файлы, чтобы в директории не было мусора. Сбой мог быть в процессе создания WAL-файла

3) Для всех состояний кроме "a" и "b" запускается функция `SyncDataDirectory()`, которая скидывает страницы кэша linux по файлам в PGDATA и директориям табличных пространств

Синхронизация PGDATA, параметр `recovery_init_sync_method`

- процесс startup выполняет выполняет синхронизацию PGDATA способом, указанным в параметре конфигурации `recovery_init_sync_method`
- по умолчанию `recovery_init_sync_method=`**`fsync`**, что означает, что процесс startup будет открывать и посылать fsync по ВСЕМ файлам в PGDATA:

```
LOG:  database system was interrupted; last known up at 03:17:41 MSK
LOG:  syncing data directory (fsync), elapsed time: 10.00 s, current path: ./base/5/4066825
LOG:  syncing data directory (fsync), elapsed time: 20.00 s, current path: ./base/5/1903193
...
LOG:  syncing data directory (fsync), elapsed time: 70.00 s, current path: ./base/5/4081550
LOG:  database system was not properly shut down; automatic recovery in progress
```

- можно установить значение параметра конфигурации `recovery_init_sync_method=`**`syncfs`**. При этом значении выполняется sync в целом на смонтированных файловых системах, на которых расположены PGDATA

Синхронизация PGDATA, параметр recovery_init_sync_method

Для всех состояний кроме "a" и "b" SyncDataDirectory() выполняет fsync по всей PGDATA способом, указанным в параметре конфигурации recovery_init_sync_method.

По умолчанию `recovery_init_sync_method=fsync`, что означает, что процесс startup будет открывать и посылать fsync по ВСЕМ файлам в PGDATA и файлам директорий, на которые указывает символическая ссылка PGDATA/pg_wal (если есть) и символические ссылки в директории PGDATA/pg_tblspc (директории табличных пространств). По другим символическим ссылкам синхронизации не будет. Это делается для того, чтобы убедиться, что все файлы в PGDATA надёжно сохранены на диске перед накатом WAL. Гарантировать, что в результате некорректной остановки экземпляра блоки файлов PGDATA физически были записаны на диск нельзя. fsync по файлам данных посылается в конце контрольной точки.

Для всех состояний кроме "a" и "b" означает, что SyncDataDirectory() выполняется также и для восстановления из бэкапов, созданных утилитой pg_basebackup. Время синхронизации увеличивает время простоя обслуживания пользователей СУБД при восстановлении.

Если в кластере сотни тысяч файлов, то синхронизация по каждому файлу займёт долгое время. Время не зависит от того есть ли в страничном кэше linux грязные страницы или нет, все файлы будут открываться.

Значение по умолчанию установлено исходя из того, что в кластере мало файлов, а файловой системе, где находится PGDATA, операционная система активно работает с файлами других приложений, например хост используется под файловый сервер. Для промышленных СУБД это не так: на хосте с СУБД другие приложения обычно не работают.

Рекомендуется установить значение параметра конфигурации recovery_init_sync_method=syncfs. При этом значении выполняется sync в целом на смонтированных файловых системах, на которых расположены PGDATA, табличные пространства и директория `PGDATA/pg_wal`. **При этом значении синхронизация выполнится гораздо быстрее, чем открывать каждый файл и давать fsync по нему.**

После синхронизации PGDATA процесс startup начинает накат журналов функцией InitWalRecovery(..) и другие действия. Процедура наката с момента начала контрольной точки, указанной в файле bacup_label или управляющем файле может занять время. Накат выполняется одним процессом startup последовательно, но при этом используется предварительное чтение блоков журналов (prefetching), поэтому startup работает быстро и думать, что его надо "распараллеливать" не нужно.

Синхронизация бэкапа, параметр `pg_basebackup --sync-method`

- в 17 версии у утилиты `pg_basebackup` появился параметр `--sync-method` с двумя значениями fsync и syncfs
  - значение по умолчанию fsync
- до 17 версии можно использовать параметр `--no-sync` (`-N`) и затем команду операционной системы `sync -f`.
- при большом числе файлов в кластере, а значит и в создаваемых бэкапах использование syncfs или sync позволяют уменьшить время на резервирование с той же отказоустойчивостью. Пример:

```
rm -rf /var/lib/postgresql/backup/1
time pg_basebackup -c fast -D $HOME/backup/1 -P
3783151/3783151 kB (100%), 1/1 tablespace
real    2m50.010s
user    0m52.035s
sys     0m3.515s
rm -rf /var/lib/postgresql/backup/1
time pg_basebackup --sync-method=syncfs -c fast -D $HOME/backup/1 -P
real    0m40.807s
user    0m36.967s
sys     0m2.612s
```


Синхронизация бэкапа, параметр `pg_basebackup --sync-method`

Число файлов может быть большим не только из-за числа объектов, но и размера объектов. Максимальный размер файла 1Гб.

В 17 версии у утилиты `pg_basebackup` появился параметр `--sync-method` с двумя значениями fsync и syncfs. Значение по умолчанию fsync.

По окончании резервирования утилита либо выполняет синхронизацию на уровне файловой системы куда выполнялось резервирование, либо fsync по всем файлам, которые положила в директорию бэкапа. При использовании fsync по очереди открываются все файлы во всех директориях бэкапа. Оба способа гарантируют сохранение данных на диске.

До 17 версии можно использовать параметр `--no-sync` (`-N`) и затем команду операционной системы **`sync -f`** для синхронизации файловых систем.

При большом числе файлов в кластере, а значит и в создаваемых бэкапах использование syncfs или sync позволяют уменьшить время на резервирование с той же отказоустойчивостью.

Пример:

```
rm -rf /var/lib/postgresql/backup/1
time pg_basebackup -c fast -D $HOME/backup/1 -P
3783151/3783151 kB (100%), 1/1 tablespace
real    2m50.010s
user    0m52.035s
sys     0m3.515s
rm -rf /var/lib/postgresql/backup/1
time pg_basebackup --no-sync -c fast -D $HOME/backup/1 -P
real    0m40.807s
user    0m36.967s
sys     0m2.612s
time sync
real    0m0.015s
user    0m0.001s
sys     0m0.000s
```

Параметр restart_after_crash

- по умолчанию значение `on` и это повышает доступность экземпляра, так как при отключении экземпляр будет просто принудительно остановлен
- определяет будут ли перезапущены процессы экземпляра при сбое серверного процесса:

```
postgres=# select name, setting, context, max_val, min_val from pg_settings where name ~ 'restart';
        name         | setting | context | max_val | min_val
---------------------+---------+---------+---------+---------
 restart_after_crash | on      | sighup  |         |
(1 row)
```

- при изменении значения достаточно перечитать конфигурацию
- параметр может быть установлен в значение `off`, если экземпляр управляется кластерным программным обеспечением и оно решает запустить экземпляр или нет

Параметр restart_after_crash

При сбое серверного процесса поведение процесса postgres определяется параметром конфигурации restart_after_crash:

```
postgres=# select name, setting, context, max_val, min_val from pg_settings
where name ~ 'restart';
         name          | setting | context | max_val | min_val
-----------------------+---------+---------+---------+---------
 restart_after_crash | on      | sighup  |         |
(1 row)
```

По умолчанию параметр включён и это повышает доступность экземпляра, так как при отключении экземпляр будет просто принудительно остановлен.

По умолчанию параметр включён и после сбоя серверного процесса процесс postgres аварийно остановит все дочерние процессы (эквивалент остановки в режиме `immediate` без контрольной точки) и запустит процессы заново (эквивалент "crash recovery" - запуска экземпляра после сбоя). Если отключить параметр, то все процессы экземпляра будут принудительно остановлены, в том числе процесс postgres. Состояние кластера после остановки будет таким же как после принудительной остановки утилитой `pg_ctl stop -m immediate`:

```
postgres@tantor:~$ pg_controldata | grep state
Database cluster state:               in production
```

Параметр обычно отключают, когда экземпляр управляется кластерным программным обеспечением, для которого автозапуск экземпляра нежелателен.

Независимо от значения параметра при запуске процессов выполняется процедура crash recovery: синхронизации и наката журналов.

Серверный процесс может сбойнуть по причине передачи сигнала SIGKILL от OOM kill при нехватке памяти.

Особенности работы экземпляра в контейнере docker

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

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


Особенности работы экземпляра в контейнере docker

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

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

В контейнере Docker процесс 1 это процесс, который запускается для запуска контейнера. Процесс postgres **не должен иметь PID=1**:

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

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

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

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

Работа экземпляра в контейнере не добавляет высокой доступности.

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

https://www.cybertec-postgresql.com/en/docker-sudden-death-for-postgresql/

Что происходит при запуске серверного процесса

- для каждой сессии создается серверный процесс
- получает структуру (часть памяти) PGPROC из списка свободных и устанавливает поля в начальные значения
- списки свободных PGPROC для серверных процессов хранятся в поле FreeProcs структуры PROC_HDR, все они находятся в разделяемой памяти
- Инициализируются (выделяются и заполняются) три кэша в локальной памяти серверного процесса:
  - Кэш для быстрого доступа к таблицам (RelationCache)
  - Кэш таблиц системного каталога (CatalogCache)
  - Кэш планов выполнения команд (PlanCache)
- выделяется память под менеджер "порталов" TopPortalContext
- клиент аутентифицируется
- догружаются разделяемые библиотеки, указанные в параметрах `session_preload_libraries` и `local_preload_libraries`
- выделяется память MessageContext для команд и процесс готов к приему команд

Что происходит при запуске серверного процесса

Основные шаги при запуске серверного процесса:

1. При запуске процесс получает структуру (часть памяти) PGPROC из списка свободных и устанавливает поля в начальные значения. Списки свободных PGPROC для серверных процессов хранятся в поле FreeProcs структуры PROC_HDR. Структуры находятся в разделяемой памяти. Доступ на запись к PROC_HDR использует вызовы SpinLockAcquire, SpinLockRelease. Доступ к полям этих структур (один PGPROC ~880 байт, размер кратен 16 байт) идет часто от каждого процесса. Хранятся они в страницах по 4Кб, ссылки на которые займут часть записей в TLB процессоров, а поля к которым часто идет доступ займет место в кэшах процессоров различных уровней. Дальше каждый процесс меняет поля в своей структуре PGPROC на осмысленные значения. Дальше процесс получает порядковый номер процесса (id) в массиве shmInvalBuffer.

2. Процесс регистрирует таймауты:

```
/* Identifiers for timeout reasons. Note that in case multiple timeouts trigger at the same time, they are serviced in the order of this enum. */
typedef enum TimeoutId
{ /* Predefined timeout reasons */
STARTUP_PACKET_TIMEOUT,
DEADLOCK_TIMEOUT,
LOCK_TIMEOUT,
STATEMENT_TIMEOUT,
STANDBY_DEADLOCK_TIMEOUT, STANDBY_TIMEOUT,
STANDBY_LOCK_TIMEOUT,
IDLE_IN_TRANSACTION_SESSION_TIMEOUT,
IDLE_SESSION_TIMEOUT,
CLIENT_CONNECTION_CHECK_TIMEOUT,
USER_TIMEOUT, /* First user-definable timeout reason */
MAX_TIMEOUTS = USER_TIMEOUT + 10 /* Maximum number of timeout reasons */
} TimeoutId;
```

3. Инициализируются три кэша в локальной памяти серверного процесса:
Кэш для быстрого доступа к таблицам (RelationCache)
Кэш таблиц системного каталога (CatalogCache)
Кэш планов выполнения команд (PlanCache)

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

5. Выполняется задержка по параметру `pre_auth_delay`. Инициализируется структура PgBackendStatus. Выполняется аутентификация

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

7. Обновляется структура PgBackendStatus.

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

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

10. Выделяется память под обработку сообщений от клиента (MessageContext), row_description_context - описание строк при ответе клиенту (название столбца, тип данных столбца (object ID) или модификаторы типа), input_message - хранение строки входящего запроса.

11. Клиенту отправляется сообщение ReadyForQuery. Если от клиента придет команда, то обнулятся таймеры IDLE_IN_TRANSACTION_SESSION_TIMEOUT и IDLE_SESSION_TIMEOUT.

Клиент может передать сообщения:

```
 case 'Q': /* simple query */
 case 'P': /* parse */
 case 'B': /* bind */
 case 'E': /* execute */
 case 'F': /* fastpath function call начало транзакции*/
 case 'C': /* close  закрыть портал или именованный запрос*/
 case 'D': /* describe */
 case 'H': /* flush сброс буфера отправки */
 case 'S': /* sync фиксация транзакции*/
 case 'X': /* 'X' means that the frontend is closing down the socket. */
 case EOF: /* EOF means unexpected loss of frontend connection. Either way,
perform normal shutdown. */
 case 'd': /* copy data игнорировать пакеты, могут посылаться командой copy */
 case 'c': /* copy done */
 case 'f': /* copy fail */
```

Примеры структур в разделяемой (общей) памяти экземпляра:

```
 Proc Array, PROC, PROCLOCK, Lock Hashes, LOCK, Multi-XACT Buffers, Two-Phase
Structs, Subtrans Buffers, CLOG Buffers, XLOG Buffers, Shared Invalidation,
Lightweight Locks, Auto Vacuum, Btree Vacuum, Buffer Descriptors, Shared Buffers,
Background Writer Synchronized Scan, Semaphores, Statistics
```

Примеры структур в локальной памяти серверного процесса:

```
 RelationCache, CatalogCache, PlanCache, work_mem, maintenans_work_mem,
StringBuffer, temp_buffers
```

PGPROC используют не только серверные процессы, но и автовакуум launcher и workers, фоновые рабочие процессы (bgworkers), walsender и другие вспомогательные процессы, процессы обслуживающие распределенные транзакции.

Общая память процессов экземпляра

- может существовать больше 72 структур
- размеры большинства структур можно посмотреть в представлении `pg_shmem_allocations`
- размер некоторых структур меняется параметрами конфигурации
- размеры всех хэш-структур выдаются не верно

```
select * from (select *, lead(off) over(order by off) - off as true_size from
pg_shmem_allocations) as a order by 1;
                name          |    off    |   size    | allocated_size | true_size
------------------------------+-----------+-----------+----------------+-----------
 <anonymous>                  |           |   4946048 |        4946048 |
 Archiver Data                | 147726208 |         8 |            128 |       128
...
 XLOG Recovery Ctl            |   4377728 |       104 |            128 |       128
                              | 148145024 |   2849920 |        2849920 |
(60 rows)
select * from (select *, lead(off) over(order by off) - off as true_size from
pg_shmem_allocations) as a where a.true<>a.allocated_size order by 1;
 LOCK hash                    | 142583808 | 2896      |           2944 |    695168
 PREDICATELOCK hash           | 144423680 | 2896      |           2944 |   1260416
...
```


Общая память процессов экземпляра

Могут существовать больше 72 структур в разделяемой памяти. К этим структурам имеют доступ процессы экземпляра. Для получения доступа как на чтение, так и на изменение используются блокировки типа LWLock (легковесные). Расширения могут создавать собственные структуры. Список структур и их размеров:

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

Строка с пустым (NULL) именем отражает неиспользуемую память. Строка с именем "<anonymous>" отражает суммарный размер структур, память под которые выделялась вызовом функции ShmemAlloc(), а не функциями ShmemInitStruct("имя..) и ShmemInitHash("имя..).

По названиям можно искать в исходном коде места где выделяется память. Например:

```
procs = (PGPROC *) ShmemAlloc(TotalProcs * sizeof(PGPROC));
```

Размеры всех хэш-структур выдаются не верные, но их размер можно оценить.

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

Параметром конфигурации `min_dynamic_shared_memory` можно выделить страницы типа HugePages, которые будут использоваться процессами, выделяющими динамически память. Если памяти заданной этим параметром окажется недостаточно, она будет выделяться обычным способом: страницами по 4Кб, если не используется Transparent HugePages.

Только `shared_buffers` и `min_dynamic_shared_memory` могут использовать HugePages.

Кэш таблиц системного каталога

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

```
 select * from (select *, lead(off) over(order by off) - off as true_size
from pg_shmem_allocations) as a where name='shmInvalBuffer' order by 1;
       name       |    off    | size  | allocated_size | true_size
------------------+-----------+-------+----------------+-----------
 shmInvalBuffer | 146865024 | 68128 |          68224 |     68224
(1 row)
```


Кэш таблиц системного каталога

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

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

```
 select * from (select *, lead(off) over(order by off) - off as true_size from
pg_shmem_allocations) as a where name='shmInvalBuffer' order by 1;
       name       |    off    | size  | allocated_size | true_size
------------------+-----------+-------+----------------+-----------
 shmInvalBuffer | 146865024 | 68128 |          68224 |     68224
```

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

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

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

Представление pg_stat_slru

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

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


Представление pg_stat_slru

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

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

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

Сброс статистики: `select pg_stat_reset_slru(null);`

В качестве аргумента можно передать название кэша или NULL, если нужно сбросить статистику для всех кэшей.

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

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

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

https://docs.tantorlabs.ru/tdb/ru/16_4/se/monitoring-stats.html

Локальная память процесса

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

Локальная память процесса

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

Параметры, наиболее сильно влияющие на выделение локальной памяти процесса:

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

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

enable_large_allocations по умолчанию false. Размер памяти, которую постепенно выделяет процесс под обработку одной строки выборки ограничен 1Гб. На обработку строки процесс может выделить до 2Гб памяти в дополнение к всей остальной памяти, которую он использует. Этот параметр позволяет увеличить размер буфера строки (StringBuffer) до 2Гб.

Представление pg_backend_memory_contexts

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

```
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from pg_backend_memory_contexts;
   sum    |   sum    |  sum
----------+----------+--------
 2114816 | 1380760 | 734056
```


Представление pg_backend_memory_contexts

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

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

```
 with recursive dep as
 (select name, total_bytes as total, ident, parent, 1 as level, name as path from
pg_backend_memory_contexts where parent is null
 union all
 select c.name, c.total_bytes, c.ident, c.parent, p.level + 1, p.path || '->' ||
c.name
 from dep p, pg_backend_memory_contexts c
 where c.parent = p.name)
 select * from dep limit 3;
        name         |total|ident|      parent      |level|              path
---------------------+-----+-----+------------------+-----+--------------------------------
 TopMemoryContext     |97664|     |                  |   1 |TopMemoryContext
 TopTransactionContext| 8192|     | TopMemoryContext |   2 |TopMemoryContext->TopTransactionContext
 PLpgSQL cast cache   | 8192|     | TopMemoryContext |   2 |TopMemoryContext->PLpgSQL cast cache
 (3 rows)
```

Память, выделенная текущему серверному процессу:

```
 select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
   sum    |   sum    |  sum
 ---------+----------+--------
  2114816 | 1380760 | 734056
```

В будущих версиях представление будут добавлены столбцы id , parent_id, path.

Функция pg_log_backend_memory_contexts(PID)

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

```
postgres=# SELECT pg_log_backend_memory_contexts(111);
 pg_log_backend_memory_contexts
--------------------------------
 t
(1 row)
LOG: statement: SELECT pg_log_backend_memory_contexts(111);
...
LOG: logging memory contexts of PID 111
LOG: level: 0; TopMemoryContext: 60528 total in 5 blocks; 16224 free (6 chunks); 44304 used
LOG: level: 1; TopTransactionContext: 8192 total in 1 blocks; 6728 free (0 chunks); 1464 used
...
LOG: level: 2; AV dblist: 8192 total in 1 blocks; 7840 free (0 chunks); 352 used
LOG: Grand total: 658848 bytes in 38 blocks; 270616 free (32 chunks); 388232 used
```


Функция pg_log_backend_memory_contexts(PID)

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

```
LOG: statement: SELECT pg_log_backend_memory_contexts(111);
...
LOG: logging memory contexts of PID 111
LOG: level: 0; TopMemoryContext: 60528 total in 5 blocks; 16224 free (6 chunks); 44304 used
LOG: level: 1; TopTransactionContext: 8192 total in 1 blocks; 6728 free (0 chunks); 1464 used
...
LOG: level: 2; AV dblist: 8192 total in 1 blocks; 7840 free (0 chunks); 352 used
LOG: Grand total: 658848 bytes in 38 blocks; 270616 free (32 chunks); 388232 used
```

Функция появилась в 14 версии PostgreSQL и была доступна только пользователям с атрибутом SUPERUSER. В 15 версии появилась возможность дать привилегию на выполнение функции непривилегированному пользователю:

```
postgres=# create role alice;
postgres=# grant execute on function pg_log_backend_memory_contexts to alice;
postgres=# drop role alice;
ERROR:  role "alice" cannot be dropped because some objects depend on it
DETAIL:  privileges for function pg_log_backend_memory_contexts(integer)
postgres=# revoke all on function pg_log_backend_memory_contexts from alice;
postgres=# drop role alice;
```

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

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

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

Память под TID store при вакуумировании учитывается в строках:

```
level: 1; TopTransactionContext: 33570864 total in 3 blocks; 11056 free (405 chunks); 33559808 used
level: 2; _bt_pagedel: 8192 total in 1 blocks; 7928 free (0 chunks); 264 used
Grand total: 35510408 bytes in 234 blocks; 736144 free (626 chunks); 34774264 used
```

Выделяется память размером `maintenance_work_mem` в контексте (памяти для) транзакции. После выполнения транзакции в процессе вакуумирования, память контекста транзакции освобождается.

9-2

Блокировки

Типы блокировок

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

Типы блокировок

Экземпляр использует блокировки для взаимодействия между процессами:

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

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

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

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

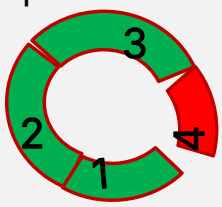
Параллельные процессы объединяются в группу с их серверным процессом (лидером группы). В 16 версии процессы в группе не конфликтуют, что реализуется алгоритмом их работы. Параллелизм развивается и логика блокирования может развиваться.

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

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

Параметры deadlock_timeout и log_lock_waits

- время ожидания получения блокировки, через которое ожидающий процесс начнет проверку на наличие взаимоблокировок
- в одном цикле ожидания одна проверка
- по умолчанию 1 секунда, значение стоит увеличить хотя бы до длительности выполнения типичной для приложения транзакции
- рекомендуется установить `log_lock_waits = true`, чтобы получать сообщения в диагностический журнал что какой-либо процесс ждет дольше, чем `deadlock_timeout`
- наличие взаимоблокировки это ошибка в архитектуре приложения
- взаимно заблокироваться могут два или более процесса ("кольцо" взаимоблокировки)



Параметры deadlock_timeout и log_lock_waits

Если процесс не может получить блокировку, он засыпает и устанавливает себе таймер, чтобы проснуться через время, заданное параметром конфигурации `deadlock_timeout` (по умолчанию 1 секунда). Он проснется до истечения таймаута, если блокировка ему будет предоставлена. Если же таймаут истечет, то будет вызвана процедура обнаружения взаимоблокировки. Если взаимоблокировки нет, то процесс снова заснет больше не будет проверять есть ли взаимоблокировка в этом цикле ожидания.

Процесс, проверяющий наличие взаимоблокировки, монопольно блокирует (легковесными блокировками) доступ ко всем разделам таблицы блокировок до конца проверки, так что все процессы экземпляра, которые захотят получить даже разделяемую блокировку (например для выполнения SELECT) будут ждать пока проверка не завершится:

```
CheckDeadLock(void)
{
 int i;
/* Acquire exclusive lock on the entire shared lock data structures. Must  grab LWLocks in partition-number order to avoid LWLock deadlock.
 Note that the deadlock check interrupt had better not be enabled anywhere that this process itself holds lock partition locks, else this will wait forever. Also note that LWLockAcquire creates a critical section, so that this routine cannot be interrupted by cancel/die interrupts. */
for (i = 0; i < NUM_LOCK_PARTITIONS; i++)
 LWLockAcquire(LockHashPartitionLockByIndex(i), LW_EXCLUSIVE);
```

Наличие взаимоблокировки - всегда ошибка в архитектуре приложения. Большинство приложений спроектированы правильно и не допускают взаимоблокировок. В какое значение установить параметр?

Стоит установить значение параметра `log_lock_waits = true` и настраивать значение `deadlock_timeout` так, чтобы сообщения об ожиданиях получения блокировки возникали нечасто. В идеале значение должно превышать типичное время транзакций, чтобы повысить шансы на то, что блокировка будет освобождена, прежде чем ожидающая транзакция решит запустить проверку на взаимоблокировку. Нужно помнить, что ожидания будут долгими, если какой-то процесс запросит монопольную (AccessExclusive) блокировку на объект и будет ждать ее получения. https://www.postgresql.org/message-id/flat/4530a101c4d17174582b07875ead600d%40oss.nttdata.com

Параметр lock_timeout

- по умолчанию не действует (значение ноль)
- не стоит устанавливать на уровне кластера
- действует на явные и неявные попытки получить блокировку

```
set lock_timeout = '1s';
ALTER TABLE test ADD COLUMN ts timestamp DEFAULT clock_timestamp();
ERROR:  canceling statement due to lock timeout
```

- granted=f означает, что блокировка не получена (ShareLock другого процесса препятствует получению):

```
select pid, relation::regclass::text, mode, granted, fastpath from pg_locks
where locktype='relation' and database = (select oid from pg_database where
datname=current_database()) order by pid, fastpath desc;
  pid   | relation |        mode         | granted | fastpath
--------+----------+---------------------+---------+----------
 33210  | test     | AccessExclusiveLock | f       | f
 43344  | test     | ShareLock           | t       | f
```


Параметр lock_timeout

Если выполняется команда, которая может столкнуться с длительным ожиданием получения блокировки, можно использовать опцию NOWAIT или SKIP LOCKED команды, если у нее такая опция есть. Недостаток NOWAIT в том, что даже если осталось ждать долю секунды, команда выдаст ошибку.

Кроме этих опций можно установить параметр lock_timeout. Преимущество в том, что он действует на любые команды и на то, что можно установить в разумное для конкретного момента время ожидания получения блокировки. Устанавливать на уровне кластера не рекомендуется, так как действует на все команды, что может быть нежелательно.

Рекомендуется устанавливать на ровне сессии перед выполнением команд, требующих монопольной блокировки, так как они пока такая команда ждет получения блокировки никакой другой процесс (даже выполняющий SELECT) не сможет получить блокировку и будет ждать пока команда не получит блокировку, не выполнится и не снимет монопольную блокировку. Часто длительность ожидания получения монопольной блокировки дольше, чем длительность выполнения команды.

При выполнении команд, изменяющих объекты стоит ознакомиться какой тип блокировки на объект использует команда; устанавливаются ли блокировки на другие объекты; меняются ли блоки данных и требуется ли монопольная блокировка на определение буфера (легковесная блокировка BufMappingLock) для каждого блока данных.

Например, добавление столбца к таблице со значением по умолчанию, выдаваемое функцией, а не константу приводит не только к установке блокировки AccessExclusive на таблицу, но и к изменению всех блоков таблицы. Удаление столбца устанавливает такую же блокировку, но не меняет блоки и после получения блокировки выполняется моментально.

Создание индексов с опцией CONCURRENTLY устанавливает блокировку на таблицу, которая позволяет работать командам меняющим строки в таблице. Создание индекса без опции CONCURRENTLY не позволяет. Недостаток опции в том, что индекс может создаться с ошибкой - команда не откатывается. В таком случае индекс нужно удалить командой REINDEX INDEX CONCURRENTLY. Также можно удалять индексы с меньшим уровнем блокирования командой DROP INDEX CONCURRENTLY.

Подтранзакции

- подтранзакции это точки сохранения
  - используются чтобы откатиться, а не переводить транзакцию в состояние сбоя
- создаются
  - командой SAVEPOINT
  - секцией EXCEPTION в блоке на языке pl/pgsql
- в структуре PGPROC сохраняется до 64 номеров подтранзакций
- подтранзакциям которые только читают данные присваивается виртуальный номер
- если встречается команда изменения данных, то подтранзакциям вплоть до основной транзакции присваиваются реальные номера
- рекомендация: не писать часто исполняемый код, который порождает больше 64 подтранзакций, его выполнение станет узким местом (падение TPS на ~25%)
- В СУБД Tantor есть параметр `subtransaction_buffers`

Подтранзакции

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

Подтранзакции создаются:

1) командой SAVEPOINT;

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

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

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

Рекомендация: не писать часто исполняемый код, который порождает больше 64 подтранзакций, его выполнение станет узким местом (**падение TPS ~25%**).

`PGPROC_MAX_CACHED_SUBXIDS` константа (макрос), так как компиляция с константами создает более эффективный и компактный код, чем с изменяемыми параметрами.

В СУБД Tantor есть параметр конфигурации `subtransaction_buffers` которым можно настроить размер буфера, хранящего данные подтранзакций. По умолчанию значение 256Кб. Размер буфера можно посмотреть:

```
SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM
pg_shmem_allocations where name like '%btrans%';
       name       | allocated_size | pg_size_pretty
------------------+----------------+----------------
 subtransaction |         267520 | 261 kB
```

увеличение буфера решает проблему №4 описанную в:

https://postgres.ai/blog/20210831-postgresql-subtransactions-considered-harmful

https://gitlab.com/postgres-ai/postgresql-consulting/tests-and-benchmarks/-/issues/20

https://www.cybertec-postgresql.com/en/subtransactions-and-performance-in-postgresql/

Мультитранзакции

- блокировка FOR NO KEY UPDATE устанавливается командой UPDATE, которая не вносит изменения в ключевые столбцы
- блокировка FOR KEY SHARE устанавливается командой DELETE и UPDATE которая обновляет значения ключевых столбцов
- также разделяемые блокировки устанавливаются командами SELECT .. FOR SHARE, FOR NO KEY UPDATE, FOR KEY SHARE
- разделяемые блокировки позволяют одновременно работать со строкой нескольким транзакциям
- одновременная работа реализуется "мультитранзакциями", которые имеют свой счетчик `xid`, свои файлы и кэши
- соответствие между `xid` транзакций и мультитранзакций хранятся в директории `PGDATA/pg_multixact`
- advisory locks не являются заменой блокировок строк, так как их количество ограничено

Мультитранзакции

Команды SELECT .. FOR SHARE, FOR NO KEY UPDATE, FOR KEY SHARE позволяют одновременно работать со строкой нескольким транзакциям. Блокировка FOR NO KEY UPDATE устанавливается командой UPDATE, которая не вносит изменения в ключевые столбцы. Блокировка FOR KEY SHARE устанавливается командой DELETE и UPDATE которая обновляет значения ключевых столбцов. Более детальные формулировки есть в документации. Важно то, что обычные команды DELETE и UPDATE могут устанавливать на строки разделяемые блокировки. При появлении второй транзакции, пока работает первая, второй серверный процесс создаст мультитранзакцию. Большинство приложений, которые преимущественно создают строки, не испытывают проблем, так как вставляемая строка другим сессиям не видна и они не могут ее заблокировать. Конфликт может возникнуть при вставке записи в уникальный индекс и тогда вторая транзакция будет ждать (мультитранзакций не будет). Да и это маловероятно, так как правильно спроектированные приложения используют автоинкрементальные столбцы. Обновление строк трудоемкая операция во всех реляционных СУБД, а в PostgreSQL особенно из-за того, что PostgreSQL сохраняет старые версии строк в блоках данных. Если архитектор (дизайнер) приложения активно использует UPDATE, то помимо уменьшения доли HOT cleanup, возможно, что часть транзакций "столкнется" на части строк и второй серверный процесс создаст мультитранзакцию. Последующие транзакции могут присоединяться к мультитранзакции, то есть транзакций может быть две и больше. Причем, создается новая мультитранзакция, куда включаются прежние транзакции. Это неоптимально, но вероятность того, что строку захотят менять не две, а три и более транзакции обычно не высока.

Если в приложении возникают взаимоблокировки это прямо указывает на ошибки в архитектуре приложения. Если вместо изменения логики работы с данными использовать разделяемые блокировки, то взаимоблокировки могут прекратиться, но производительность не улучшится.

В `pg_stat_activity wait_event IN ('MultiXactMemberControlLock', 'MultiXactOffsetControlLock', 'multixact_member', 'multixact_offset')`, они **относятся к типу `wait_event_type=LWLock`.**

LWLock (легковесные блокировки) - тип блокировок (алгоритм), которые наравне со spinlock используются для арбитража доступа к структурам разделяемой памяти. **Их общее количество в экземпляре ограничено константой: `MAX_SIMUL_LWLOCKS=200`.**

https://gitlab.com/postgres-ai/postgresql-consulting/tests-and-benchmarks/-/issues/24

Вспомогательная функция для выдачи информационных битов в удобном виде:

```
create or replace function heap_page(relname text, pageno integer) returns
 table(lp_off text, ctid tid,state text,xmin text,xmax text,hhu text,hot
text,t_ctid tid, multi text)
as $$
select lp_off, (pageno,lp)::text::tid as ctid,
 case lp_flags
  when 0 then 'unused'
  when 1 then 'normal'
  when 2 then 'redirect to '|| lp_off
  when 3 then 'dead'
 end as state,
 t_xmin || case
  when (t_infomask & 256) > 0 then 'c'
  when (t_infomask & 512) > 0 then 'a'
  else ''
 end as xmin,
 t_xmax || case
  when (t_infomask & 1024) >0 then 'c'
  when (t_infomask & 2048) >0 then 'a'
  else ''
 end as xmax,
  case when (t_infomask2 & 16384) >0 then 't' end as hhu,
  case when (t_infomask2 & 32768) >0 then 't' end as hot,
 t_ctid,
 case when (t_infomask&4096) >0 then 't' else 'f'
 end as multi
 from heap_page_items (get_raw_page(relname, pageno))
 order by lp;
$$ language sql;
select * from heap_page('t',0);
```

Мультитранзакция может порождаться и такими командами:

```
drop table if exists t; create table t (c int); insert into t values(1);
begin;
select c from t where c=1 for update;
savepoint s1;
update t set c=1 where c=1;
commit;
select lp, lp_off, lp_len, t_ctid, t_xmin, t_xmax, t_ctid, t_infomask,
(t_infomask&4096)!=0 as m from heap_page_items(get_raw_page('t', 0));
 lp  | lp_off | lp_len | t_ctid | t_xmin | t_xmax | t_ctid | t_infomask | m
-----+--------+--------+--------+--------+--------+--------+------------+---
   1 |   8144 |     28 | (0,2)  |  46891 |     11 | (0,2)  |       4416 | t
   2 |   8112 |     28 | (0,2)  |  46893 |  46892 | (0,2)  |       8336 | f
select * from heap_page('t',0);
 ctid  | state  |  xmin   | xmax  | hhu | hot | t_ctid | multi
-------+--------+---------+-------+-----+-----+--------+-------
 (0,1) | normal | 46891c | 11    | t   |     | (0,2)  | t
 (0,2) | normal | 46893   | 46892 |     | t   | (0,2)  | f
```

Это одна из причин избегать использования точек сохранения и анализировать блоки подпрограмм на pl/pgsql с секциями EXCEPTION.

Была создана цепочка HOT, так как на таблице нет индексов.

hhu - подсказка процессам, что надо идти по цепочке ctid.

hot - на данную версию строки нет ссылок из индексов.

HOT cleanup не был выполнен потому, что не было условий для его срабатывания - в блоке больше 10% свободного места.

При обращении к строке:

```
select * from t;
```

серверный процесс обратится к статусу мультитранзакции и обновит биты в маске о том, что транзакция была зафиксирована:

```
 select * from heap_page('t',0);
 ctid  | state  |  xmin  | xmax  | hhu | hot | t_ctid | multi
-------+--------+--------+-------+-----+-----+--------+-------
 (0,1) | normal | 46891c | 11    | t   |     | (0,2)  | t
 (0,2) | normal | 46893c | 46892 |     | t   | (0,2)  | f
(2 rows)
```

После вакуума версия строки и следы мультитранзакции очистятся:

```
vacuum t;
 select * from heap_page('t',0);
 ctid  |     state     |  xmin  |  xmax  | hhu | hot | t_ctid | multi
-------+---------------+--------+--------+-----+-----+--------+-------
 (0,1) | redirect to 2 |        |        |     |     |        | f
 (0,2) | normal        | 46893c | 46892a |     | t   | (0,2)  | f
```

Если горизонт удерживался бы было бы так:

```
 ctid  | state  |  xmin  | xmax  | hhu | hot | t_ctid | multi
-------+--------+--------+-------+-----+-----+--------+-------
 (0,1) | normal | 2c     | 46893 | t   |     | (0,2)  | f
 (0,2) | normal | 46893c | 0a    |     | t   | (0,2)  | f
```

Биты подсказки `c` `a` являются признаком заморозки версии строки.

Есть параметры:

```
\dconfig multixact*
 List of configuration parameters
        Parameter         | Value
--------------------------+-------
 multixact_member_buffers | 256kB
 multixact_offset_buffers | 128kB
```

которыми решается частная проблема: при появлении мультитранзакций, процессы должны проверять статус транзакций и для этого обращаться в структурам multixact. Эффект от изменения размера ~4%.

Число блокировок объектов и рекомендательных блокировок (advisory locks) ограничено на экземпляре произведением `max_locks_per_transaction * (max_connections + max_prepared_transactions)`. Увеличение значений параметров приводит к увеличению структур разделяемой памяти при том, что **число разделов (на которые устанавливается блокировка LWlock) этими параметрами не меняется**. Массовое использование advisory locks в качестве замены блокировок строк может привести к исчерпанию общего количества блокировок на экземпляре.

https://www.postgresql.org/message-id/flat/2BEC2B3F-9B61-4C1D-9FB5-5FAB0F05EF86@yandex-team.ru

Быстрый путь блокирования

- предназначен для уменьшения накладных расходов на получение и освобождение блокировок на relations (секции таблиц, обычные таблицы, индексы), которые часто запрашиваются, но редко конфликтуют
- используются для "слабых" блокировок: AccessShare, RowSharek, RowExclusive которые устанавливают команды SELECT, INSERT, UPDATE, DELETE, MERGE
- наличие "сильной" (Share, ShareRowExclusive, Exclusive, AccessExclusive) блокировки на relation, попадающий в 1/1024 часть таблицы блокировок, не дает использовать быстрый путь, поэтому:
  - Если установить сильную блокировку на таблицу, то команды, работающие с ней, не смогут использовать быстрый путь
  - не стоит выполнять большое число команд, устанавливающих сильные блокировки во время большой нагрузки на экземпляр

Быстрый путь блокирования

предназначен для уменьшения накладных расходов на получение и освобождение блокировок, которые часто запрашиваются, но редко конфликтуют. Это блокировки связанных друг с другом объектов типа relations. Команды SELECT, INSERT, UPDATE, DELETE, MERGE должны заблокировать все relations (таблицы, индексы, секции таблиц) на время формирования плана и оставить блокировки на тех relations, которые используются в плане выполнения. С одной таблицей может одновременно выполняться множество этих команд и они обычно не конфликтуют.

Общая таблица блокировок, даже разделенная на разделы, стала бы узким местом. Даже на двух ядрах процессора процессы сталкиваются на попытке установить блокировку на раздел таблицы. С увеличением количества ядер, а еще хуже - процессоров, ожидания стали бы узким местом. Для устранения узкого места используется структура PROC. В ней для каждого процесса выделено место для хранения блокировок на максимум 16 relations.

Как можно убедиться в отсутствии конфликтующих блокировок на relation не обращаясь к таблице блокировок (иначе придется получать shared LWLock на раздел таблицы блокировок, а это не сильно быстро) и просмотрев только структуры PGPROC? Блокировки на один и тот же relation всегда попадают в один и тот же раздел. **Хэш для relations рассчитывается только на основе dboid и reloid**. Используется массив в разделяемой памяти из 1024 частей (разделов) с целыми числами (`FAST_PATH_STRONG_LOCK_HASH_PARTITIONS`). Каждое число отражает количество "сильных" блокировок (`Share`, `ShareRowExclusive`, `Exclusive`, `AccessExclusive`), которые способны конфликтовать с теми, которые можно установить по быстрому пути в 1/1024 части таблицы блокировок. Вычисление какой объект в какую часть попал происходит быстро - по хэшу. Если число сильных блокировок на части равно нулю, то используется быстрый путь.

Таблица блокировок разделена на `NUM_LOCK_PARTITIONS=16` разделов.

Если число сильных блокировок на разделе больше нуля, то быстрый путь не используется. **Если установить сильную блокировку на таблицу, то команды, работающие с ней, не смогут использовать быстрый путь.** То же самое, если установить блокировку на объект, хэш от которого попадет в ту же часть 1/1024 таблицы блокировок. Примерная оценка вероятности: если число сильных блокировок на объекты 10, быстрый путь не сможет использоваться в ~1%. Для процессов, которые **не** смогут использовать fastpath доступно всего 16 траншей на общую таблицу блокировок всего кластера. То есть **одновременно создать или снять блокировку на таблицы и индексы по обычному пути могут максимум 16 процессов**.

Сильные и слабые блокировки таблиц

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

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

Сильные и слабые блокировки таблиц

Слабые блокировки могут быть получены по быстрому пути:

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

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

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

Не слабая и не сильная блокировка:

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

Автовакуум и автоанализ не мешают использовать быстрый путь.

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

Share - CREATE INDEX (без CONCURRENTLY)

ShareRowExclusive - устанавливает CREATE TRIGGER и некоторыми видами ALTER TABLE

Exclusive - устанавливает REFRESH MATERIALIZED VIEW CONCURRENTLY

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

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

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

Автовакуум в следующем цикле попробует снова обработать таблицу и ее индексы. Такие ошибки не должны быть постоянными по одной и той же таблице. Если автовакуум не сможет обработать таблицу долгое время, то это приведет к раздуванию ее файлов данных.

https://github.com/postgres/postgres/blob/master/src/backend/storage/lmgr/README

Справочник устанавливаемых командами блокировок

ShareUpdateExclusive устанавливают команды:

```
VACUUM
REINDEX CONCURRENTLY
DROP INDEX CONCURRENTLY
CREATE STATISTICS
CREATE INDEX CONCURRENTLY
COMMENT ON
ANALYZE
ALTER TABLE VALIDATE CONSTRAINT
ALTER TABLE SET WITHOUT CLUSTER
ALTER TABLE SET TOAST
ALTER TABLE SET STATISTICS
ALTER TABLE SET N_DISTINCT
ALTER TABLE SET FILLFACTOR
ALTER TABLE SET AUTOVACUUUM
ALTER TABLE DETACH PARTITION CONCURRENTLY (PARENT)
ALTER TABLE CLUSTER ON
ALTER TABLE ATTACH PARTITION (PARENT)
ALTER INDEX (RENAME)
```

AccessExclusive устанавливают команды:

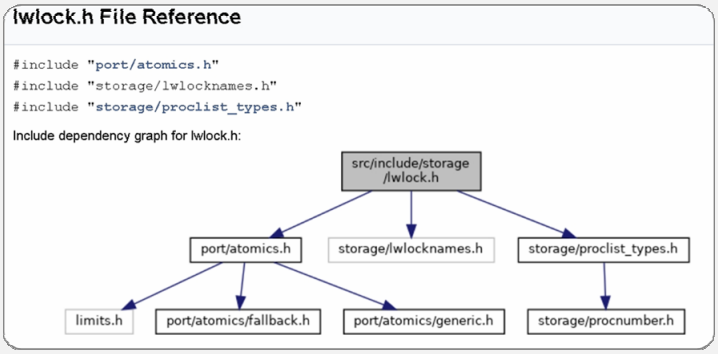
```
VACUUM FULL
TRUNCATE
REINDEX
REFRESH MATERIALIZED VIEW
DROP TABLE
DROP INDEX
CLUSTER
ALTER TABLE SET/DROP DEFAULT
ALTER TABLE SET TABLESPACE
ALTER TABLE SET STORAGE
ALTER TABLE SET SEQUENCE
ALTER TABLE SET DATA TYPE
ALTER TABLE SET COMPRESSION
ALTER TABLE RESET STORAGE
ALTER TABLE RENAME
ALTER TABLE INHERIT PARENT
ALTER TABLE ENABLE/DISABLE RULE
ALTER TABLE ENABLE/DISABLE ROW LEVEL SECURITY
ALTER TABLE DROP EXPRESSION
ALTER TABLE DROP CONSTRAINT
ALTER TABLE DROP COLUMN
ALTER TABLE DETACH PARTITION (PARENT)
ALTER TABLE DETACH PARTITION (TARGET/DEFAULT)
ALTER TABLE DETACH PARTITION CONCURRENTLY (TARGET/DEFAULT)
ALTER TABLE ATTACH PARTITION (TARGET/DEFAULT)
ALTER TABLE ALTER CONSTRAINT
ALTER TABLE ADD COLUMN
ALTER TABLE ADD CONSTRAINT
ALTER TABLE SET (LOGGED | UNLOGGED)
ALTER INDEX SET TABLESPACE
ALTER INDEX SET FILLFACTOR
ALTER INDEX ATTACH PARTITION
```

https://pglocks.org/

Команда `ALTER TABLE SET (LOGGED | UNLOGGED)` полностью перестраивает таблицу и индексы, монопольно блокируя их и зависимые объекты (последовательности).

Секции таблицы блокировок

- таблица блокировок разделена на 16 разделов (partitions, секций, частей)
- в PGPROC каждого процесса сохраняется список частей, которые он блокирует
- число легковесных блокировок, которые используются для защиты структур памяти таких как lock partition, ограничено 200 (MAX_SIMUL_LWLOCKS)



Секции таблицы блокировок

Таблица блокировок разделена на 16 частей (2^4). Степень двойки установлена макросом:

```
/* Number of partitions the shared lock tables are divided into */
#define LOG2_NUM_LOCK_PARTITIONS  4
#define NUM_LOCK_PARTITIONS  (1 << LOG2_NUM_LOCK_PARTITIONS)
```

В PGPROC каждого процесса сохраняется список частей, которые он блокирует:

```
  /* All PROCLOCK objects for locks held or awaited by this backend are linked
into one of these lists, according to the partition number of  their lock. */
 dlist_head  myProcLocks[NUM_LOCK_PARTITIONS];
```

В PostgreSQL и СУБД Tantor LOG2_NUM_LOCK_PARTITIONS - константа. **В форках с параметром конфигурации `log2_num_lock_partitions` увеличение его значения увеличивает структуру PGPROC.**

Общее количество легковесных блокировок, которые используются для защиты структур памяти таких как lock partition, ограничено макросом MAX_SIMUL_LWLOCKS=200.

Доступ к структурам PGPROC идет очень часто, как к PROC_HDR, BufferDesc. Если "горячие" структуры или их части перестанут помещаться в кэши процессоров, производительность начнет уменьшаться. Из-за этого в форках, где используется **`log2_num_lock_partitions`** вынуждены менять логику LWLocks утяжеляя их (параметр lwlock_shared_limit).

Компиляция с константами создает более эффективный и компактный код, чем с изменяемыми параметрами типа **`log2_num_lock_partitions`**: "As for turning the parameter into a GUC, that has a cost too. Either direct - a compiler can do far more optimizations with compile-time constants than with values that may change during execution, for example. Or indirect - if we can't give users any guidance how/when to tune the GUC, it can easily lead to misconfiguration (I can't even count how many times I had to deal with systems where the values were "tuned" following the logic that more is always better)." (Tomas Vondra, EnterpriseDB)

https://www.postgresql.org/message-id/flat/779f2bd6-00f3-4aac-a792-b81f47e41abd%40enterprisedb.com

Транши блокировок (tranches)

- число легковесных блокировок 200 это общее ограничение, количество блокировок для каждого подтипа LWLock (которых больше 73) имеет меньшее ограничение и называется траншем
- для работы с таблицей тяжеловесных блокировок одновременно можно использовать 16 траншей (NUM_LOCK_PARTITIONS=16)
  - по обычному пути только 16 процессов одновременно может находиться в процессе получения блокировки на объект
- для работы с описателем буферов 128 траншей (`NUM_BUFFER_PARTITIONS=128`)
- названия легковесных блокировок - значение столбца `wait_event` для `wait_event_type='LWLock'` представления `pg_stat_activity`

Транши блокировок (tranches)

Максимальное количество LWLocks всех типов ограничено 200. При этом для каждого подтипа LWLocks имеется меньшее ограничение, ограничивающее количество блокировок подтипа.

Транши выделяются функцией LWLockInitialize(..). Три подтипа LWLocks, конкуренция за которые становится узким местом чаще остальных: BufferMapping, LockManager, ProcArray.

Для BufMappingLock 128 траншей, то есть для блокировок на доступ к описателям буферов кэша буферов используется 128 траншей:

```
/* Initialize buffer mapping LWLocks in main array */
lock = MainLWLockArray + BUFFER_MAPPING_LWLOCK_OFFSET;
for (id = 0; id < NUM_BUFFER_PARTITIONS; id++, lock++)
  LWLockInitialize(&lock->lock, LWTRANCHE_BUFFER_MAPPING);
```

Для таблицы блокировок инициализируется 16 траншей:

```
/* Initialize lmgrs' LWLocks in main array */
lock = MainLWLockArray + LOCK_MANAGER_LWLOCK_OFFSET;
for (id = 0; id < NUM_LOCK_PARTITIONS; id++, lock++)
  LWLockInitialize(&lock->lock, LWTRANCHE_LOCK_MANAGER);
```

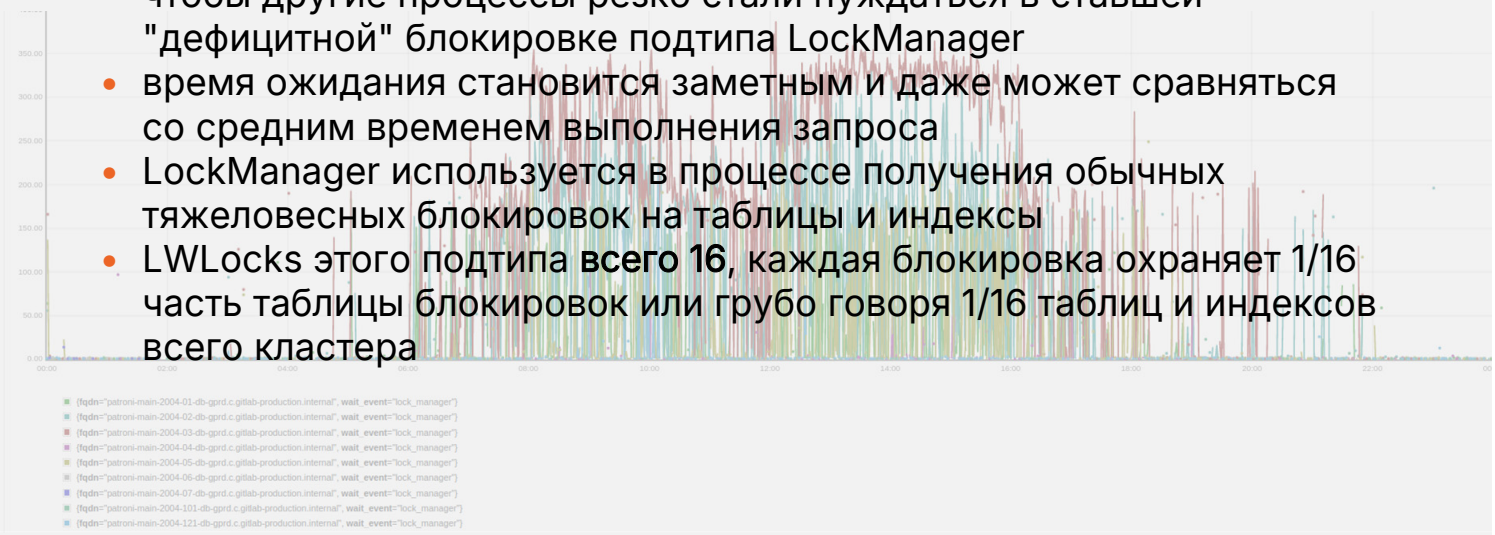
Это означает, что по обычному пути только 16 процессов одновременно может находиться в процессе получения блокировки на объект. С точки зрения мониторинга, то есть поиска: какой же подтип легковесной блокировки ожидается процессами, а значит являлся узким местом, можно использовать представление `pg_stat_activity`. Названия легковесных блокировок - значение столбца wait_event для wait_event_type='LWLock' таблицы pg_stat_activity:

https://docs.tantorlabs.ru/tdb/ru/16_4/se/monitoring-stats.html#WAIT-EVENT-LWLOCK-TABLE

Названия легковесных блокировок, использующих транши, есть в массивах BuiltinTrancheNames[] и IndividualLWLockNames[] из lwlocknames.c Расширения могут создавать и регистрировать транши для своих блокировок.

Легковесные блокировки

- LWLocks должны удерживаться микросекунды
- задержка в сотни или тысячи микросекунд - этого достаточно, чтобы другие процессы резко стали нуждаться в ставшей "дефицитной" блокировке подтипа LockManager
- время ожидания становится заметным и даже может сравняться со средним временем выполнения запроса
- LockManager используется в процессе получения обычных тяжеловесных блокировок на таблицы и индексы
- LWLocks этого подтипа **всего 16**, каждая блокировка охраняет 1/16 часть таблицы блокировок или грубо говоря 1/16 таблиц и индексов всего кластера



Легковесные блокировки

Обычно, LWLocks снимаются быстро - **в течение микросекунд**, но в моменты большой нагрузки могут длиться долго и процессы, ожидающие получения легковесной блокировки на 1/16 часть таблицы блокировок могут ожидать относительно долго, проблема нарастает как снежный ком. Транш (подтип) LockManager используется в процессе получения обычных тяжеловесных блокировок на таблицы и индексы. Например, просто чтобы создать план выполнения. Каждый запрос должен получить эти блокировки. LWLocks этого подтипа **всего 16**, каждая блокировка охраняет 1/16 часть таблицы блокировок или грубо говоря 1/16 таблиц и индексов всего кластера.

Пока серверный процесс удерживает блокировку LockManager, если он тормознёт по какой-либо причине (например, его вытеснит планировщик операционной системы), это может быстро перерасти в блокирование других процессов. Пулы соединений серверов приложений станут переполняться из-за увеличения длительности запросов, генерируемых кодом приложений. Задержки в получении LWLocks увеличиваются при неравномерном доступе к разделяемым структурам памяти: если используется несколько процессоров.

Если держатель LockManager задержится на несколько микросекунд, это нормально. Но задержка в сотни или тысячи микросекунд - этого достаточно, чтобы другие процессы резко стали нуждаться в ставшей "дефицитной" блокировке подтипа LockManager. Так как этот подтип используется в монопольном режиме (меняется, а не читается запись в структуре блокировок), все процессы ждут , пока текущий держатель блокировки освободит её. Получается "бутылочное горлышко" . И довольно быстро время ожидания становится заметным и даже может сравняться со средним временем выполнения запроса.

С такой проблемой столкнулся гитлаб, обслуживающий большое количество запросов (lock_manager прежнее название LockManager):

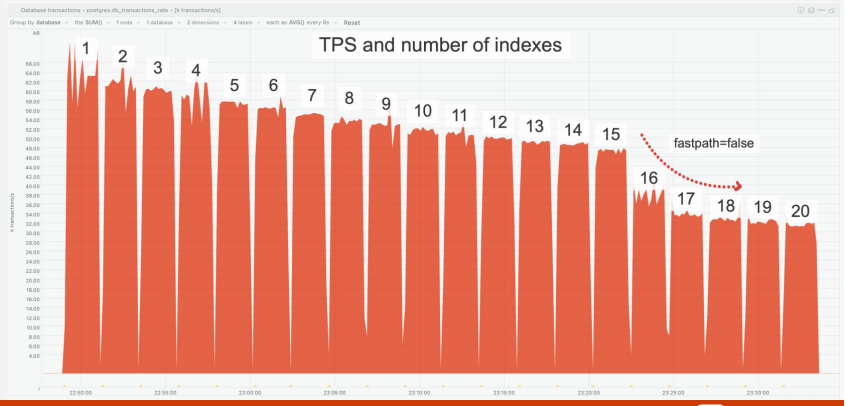
https://gitlab.com/gitlab-com/gl-infra/scalability/-/issues/2301

Конкуренция более часто возникает за блокировки LockManager, BufferMapping, ProcArrayLock. Узкие места постепенно устраняются в новых версиях PostgreSQL. Вероятность проявления узких мест возникает при увеличении структур памяти и ядер процессоров больше обычного. В таких случаях убрать узкое место могут: ограничение процессов экземпляра одним процессором; отключение Hyper Threading (число "ядер" становится меньше в 2 раза); уменьшение shared_buffers с сотен гигабайт до десятка гигабайт; max_connections с тысяч до сотен. Пример проблем на старых версиях PostgreSQL:

https://pgconf.ru/media//2020/02/06/Korotkov_pgconfru_2020_Bottlenecks_2.pdf

Блокирование по быстрому пути и 16 блокировок

- первые 16 блокировок типа AccessShareLock, RowShareLock, RowExclusiveLock процесс получает по быстрому пути
- для остальных используется более медленный метод
- при большом числе активных процессов и большом числе `fastpath=false` блокировки `wait_event='LockManager'` в `pg_stat_activity` выходят в топ и становятся узким местом



Блокирование по быстрому пути и 16 блокировок

Макрос в proc.h FP_LOCK_SLOTS_PER_BACKEND устанавливает, что **16 блокировок типа AccessShare, RowShare, RowExclusive будут получены по быстрому пути**, что уменьшает конкуренцию за доступ у структурам памяти и уменьшает событие ожидания (pg_stat_activity) типа LockManager. Только для 16 блокировок каждого процесса на объекты типа relation в столбце `pg_locks.fastpath=true`. Для остальных `fastpath=false`.

При fastpath=false, менеджер блокировок использует более медленный метод для получения блокировок. При большом числе активных процессов и большом числе fastpath=false блокировки `wait_event='LockManager'` в `pg_stat_activity` выходят в топ и становятся узким местом. Причем, блокировки берутся как на этапе создания плана, так и на этапе выполнения, в том числе для SELECT, а это значит, что проблема проявится и на репликах. Деградация производительности проявляется под высокой нагрузкой.

Такая проблема была обнаружена когда реплики стали консолидировать: уменьшили количество реплик и перенесли запросы на одну. Проблема может возникнуть при большом количестве секций с запросами, не позволяющими исключать секции из сканирования (partition pruning) или индексов.

Увеличение FP_LOCK_SLOTS_PER_BACKEND с 16 до 64 приведет к увеличению структуры PGPROC в разделяемой памяти. PGPROC хранит состояние процесса. Она занимает 880 байт, что равно 14 cache lines (блоков данных), добавление 48 xid увеличит ее на 192 байта (3 cache lines). "Линии кэша" - блоки размером 64 байт, которыми передаются данные между кэшем процессора и памятью, содержит копию данных из основной памяти.

В 16 версии был оптимизирован способ получения блокировки обычным способом. При каждом выполнении SELECT или начале транзакции (получении моментального снимка) идет обращение к PGPROC, чтобы составить список активных транзакций. Также обращение идёт при проверке наличия взаимоблокировок. **Количество структур PGPROC задается параметром max_connections.** Увеличение и того и другого снижает производительность.

Влияние на производительность начинает ощущаться при большом числе активных запросов и более чем одним процессором.

https://ardentperf.com/2024/03/03/postgres-indexes-partitioning-and-lwlocklockmanager-scalability/
https://gitlab.com/postgres-ai/postgresql-consulting/tests-and-benchmarks/-/issues/41

Воспользуемся стандартными таблицами pgbench. Создадим таблицу с 8 секциями и выполним к ней запрос, в котором нет предикатов, позволяющих исключить секции из сканирования: в запросе нет столбца, входящего в ключ секционирования, используется столбец bid. Пример:

pgbench -i -s 1 --partitions=8

первая сессия:

```
postgres=# begin transaction;
postgres=*# explain (analyze, costs false) select abalance from pgbench_accounts where bid=1 limit 1;
                              QUERY PLAN
-----------------------------------------------------------------------
Limit (actual time=0.030..0.054 rows=1 loops=1)
->  Append (actual time=0.022..0.034 rows=1 loops=1)
   ->  Seq Scan on pgbench_accounts_1 (actual time=0.013..0.017 rows=1 loops=1)
Filter: (bid = 1)
      ->  Seq Scan on pgbench_accounts_2 (never executed)
      ->  Seq Scan on pgbench_accounts_3 (never executed)
      ->  Seq Scan on pgbench_accounts_4 (never executed)
      ->  Seq Scan on pgbench_accounts_5 (never executed)
      ->  Seq Scan on pgbench_accounts_6 (never executed)
      ->  Seq Scan on pgbench_accounts_7 (never executed)
      ->  Seq Scan on pgbench_accounts_8 (never executed)
 Planning Time: 0.170 ms
 Execution Time: 0.095 ms
```

вторая сессия:

```
postgres=# select pid, relation::regclass::text, mode, granted, fastpath from pg_locks where locktype='relation' and database = (select oid from pg_database where datname=current_database()) order by pid, fastpath desc;
  pid  |        relation         |      mode       | granted | fastpath
-------+-------------------------+-----------------+---------+----------
 93229 | pg_locks                | AccessShareLock | t       | t
 94341 | pgbench_accounts_7_pkey | AccessShareLock | t       | t
 94341 | pgbench_accounts_7      | AccessShareLock | t       | t
 94341 | pgbench_accounts_6_pkey | AccessShareLock | t       | t
 94341 | pgbench_accounts_6      | AccessShareLock | t       | t
 94341 | pgbench_accounts_5_pkey | AccessShareLock | t       | t
 94341 | pgbench_accounts_5      | AccessShareLock | t       | t
 94341 | pgbench_accounts_4_pkey | AccessShareLock | t       | t
 94341 | pgbench_accounts_4      | AccessShareLock | t       | t
 94341 | pgbench_accounts_3_pkey | AccessShareLock | t       | t
 94341 | pgbench_accounts_3      | AccessShareLock | t       | t
 94341 | pgbench_accounts_2_pkey | AccessShareLock | t       | t
 94341 | pgbench_accounts_2      | AccessShareLock | t       | t
 94341 | pgbench_accounts_1_pkey | AccessShareLock | t       | t
 94341 | pgbench_accounts_1      | AccessShareLock | t       | t
 94341 | pgbench_accounts_pkey   | AccessShareLock | t       | t
 94341 | pgbench_accounts        | AccessShareLock | t       | t
 94341 | pgbench_accounts_8      | AccessShareLock | t       | f
 94341 | pgbench_accounts_8_pkey | AccessShareLock | t       | f
(19 rows)
```

7 секций и индексов заблокированы быстрым способом (fastpath=t).

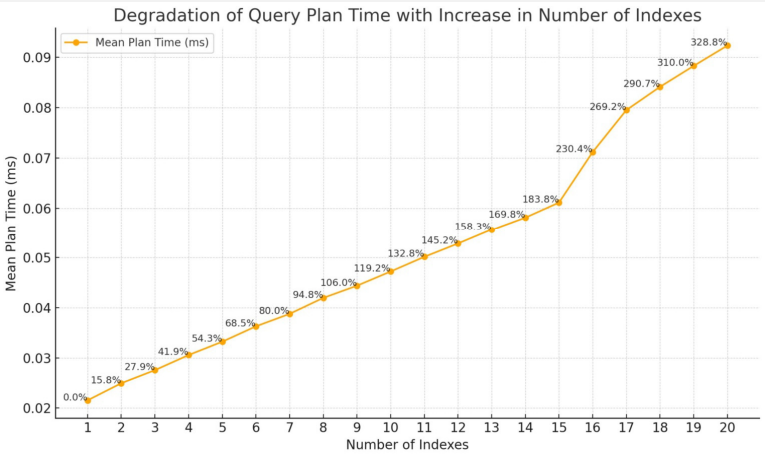
Восьмая секция и индекс на нее заблокированы обычным способом (fastpath=f)

Индексы, соединения, секции и быстрый путь

- на этапе планирования на все таблицы, индексы, секции таблиц, которые рассматриваются планировщиком устанавливаются блокировки типа AccessShareLock
- после создания плана блокировки с тех relations, которые **отсутствуют в плане** снимаются

Рекомендации:

- использовать подготовленные запросы
- использовать меньше индексов
- использовать меньше соединений
- использовать меньше секций



Индексы, соединения, секции и быстрый путь

Блокировки типа AccessShareLock устанавливаются на этапе планирования - на все таблицы, индексы, секции таблиц, которые рассматриваются планировщиком. После создания плана блокировки с тех relations, которые **отсутствуют в плане** снимаются. Первые 15 блокировок получаются быстро и не зависят от нагрузки на экземпляр, а последующие блокировки получают дольше, из-за чего время планирования увеличивается. Замедление зависит от уровня конкуренции процессов за ресурсы LockManager (количества сессий обращающихся за блокировками).

Увеличение времени планирования может быть заметно на тех запросах, где время планирования больше времени выполнения, при этом запрос выполняется часто. Неподготовленные запросы не кэшируются и каждый раз планируются заново.

Рекомендации для фазы планирования:

1. Использование подготовленных запросов (в JDBC Prepared Statements, в libpq расширенный режим) позволяет использовать generic plan и устранить проблему конкуренции но только на фазе планирования.

2. Использовать как можно меньше индексов в таблицах, которые используются в часто выполняющихся запросах. Если индексов на таблице больше 15 (при выполнении запроса имеются fast_path=false) стоит изменить код, чтобы этот запрос не выполнялся слишком часто (например, чаще 100 раз в секунду). Возможно результат можно закэшировать.

Рекомендации для фазы выполнения:

1. Использовать как можно **меньше индексов** в таблицах, которые используются в часто выполняющихся запросах. Дополнительный индекс может не дать возможности использовать HOT, что приедет к деградации производительности.

2. Использовать **меньше соединений** таблиц. Каждое соединение это блокировка на присоединяемую таблицу и её индекс (если используется при соединении)

3. При использовании секционированных таблиц нужно понимать что большое количество секций само по себе не увеличивает производительность, а упрощает администрирование и оправдано только на больших объемах данных. Например, не нужно делать секции посуточно, если можно обойтись недельными секциями (в 7 раз меньше).

Стоит обращать внимание эффективно ли исключаются секции из сканирования в планах выполнения (partition pruning).

4. На высоконагруженных системах можно рассмотреть увеличение количества реплик или кластеров с подписками и перераспределить запросы на дополнительные экземпляры.

Параметр `join_collapse_limit`

- при соединении больше 8 таблиц эффективность снижается
- `from_collapse_limit` (по умолчанию 8) задаёт максимальное число "элементов" в списке FROM, **до** которого планировщик будет объединять вложенные запросы с внешним запросом
- `join_collapse_limit` (по умолчанию равен `from_collapse_limit`) задаёт максимальное число "элементов" с фразой JOIN, **до** достижения которого планировщик будет полностью перебирать последовательность соединений
- `geqo_threshold` (по умолчанию 12) минимальное число "элементов", при котором для планирования запроса станет использоваться генетический алгоритм

```
drop table if exists t cascade;
create table t (id numeric, c text);
insert into t values (1, 'a');
create or replace view t1 as select * from t;
select * from t1 as ta join (select * from t) as tb on (ta.id=tb.id) join
generate_series(1, 3) as tc(id) on (tb.id=tc.id);
```


Параметр `join_collapse_limit`

Первые 16 блокировок получаются быстро и не зависят от нагрузки на экземпляр и что каждое соединение это блокировка на присоединяемую таблицу и её индекс (если используется при соединении). Блокировки на 8 соединяемых таблиц (с 8 индексами) будут браться по быстрому пути. Стоит проектировать схемы хранения так, чтобы по возможности число соединяемых таблиц было не больше 8. Если объект уже заблокирован, то он до конца транзакции или снятия блокировки повторно не блокируется. Поэтому параметр Tantor SE1C `enable_self_join_removal` на блокирование не влияет.

Параметр `join_collapse_limit` (по умолчанию равен `from_collapse_limit`) устанавливает сколько "элементов" (отношений, подзапросов, табличных функций), **соединяемых фразой JOIN**, может быть в запросе, чтобы планировщик полностью перебирал последовательность соединений (соединяется всегда пара "элементов"). При небольших значениях сокращается время планирования, **при увеличении больше 6 время планирования резко возрастает**. Если число "элементов" больше, чем `join_collapse_limit`, то отношения разбиваются на группы с числом "элементов" не больше `join_collapse_limit` и перебор порядка соединения идёт внутри групп, а затем соединяются наборы строк, которые возвращаются из групп. Элементы, соединяемые FULL OUTER JOIN не входят в группы и учитываются как один "элемент", так как не участвует в переборе (переставлять таблицы в другие JOIN нельзя).

При значении равном 1, отношения будут соединяться в том порядке, в котором они указаны в JOIN и планировщик сможет выбирать разве что "элемент", по которому будет строиться хэш-таблица при соединении хэшированием. Порядок "элементов", заданный в запросе определит фактический порядок соединения. Можно задать параметру значение 1 и перечислить "элементы" в желаемом порядке.

`from_collapse_limit` (по умолчанию 8) задаёт максимальное число "элементов", перечисленных в списке FROM, **до** которого планировщик будет объединять вложенные запросы с основным запросом. **Рекомендуется, чтобы значение было меньше, чем `geqo_threshold`.**

`geqo_threshold` (по умолчанию 12) минимальное число "элементов", при котором для планирования запроса будет использоваться генетический алгоритм. "`x FULL OUTER JOIN y`" учитывается как один "элемент".

Если в коде приложения соединяется большое число "элементов"(1С), можно устанавливать значения обоих параметров `*_collapse_limit` в 20 и больше.

Представление pg_locks

- выдает все тяжеловесные блокировки на экземпляре (по всем базам данных)
- Число тяжеловесных блокировок ограничено на экземпляре произведением `max_locks_per_transaction * (max_connections + max_prepared_transactions)`
- `max_prepared_transactions` относится не к подготовленным командам, а распределенным транзакциям, по умолчанию значение ноль
- `max_locks_per_transaction` по умолчанию 64
- `max_connections` по умолчанию 100
- в столбце `locktype` представления `pg_locks` указан один из 12 типов блокировок: **`relation`**, `extend`, `frozenid`, `page`, `tuple`, `transactionid`, `virtualxid`, `spectoken`, `object`, `userlock`, **`advisory`**, `applytransaction`

Представление pg_locks

Размер структуры памяти, в которой хранятся тяжеловесные блокировки определяется произведением: `max_locks_per_transaction * (max_connections + max_prepared_transactions)`.

Параметр `max_prepared_transactions` относится не к подготовленным командам, а распределенным транзакциям, по умолчанию значение ноль.

Значение `max_locks_per_transaction` по умолчанию 64.

Значение `max_connections` по умолчанию 100.

Количество блокировок, которые может использовать одна транзакция не ограничивается параметром `max_locks_per_transaction`.

Значения всех трех параметров помимо файлов параметров конфигурации сохраняются в файле pg_control.

Поскольку память под структуру выделяется при запуске экземпляра, изменение параметров требует перезапуск экземпляра. Изменения их значений передаются через WAL, на физических репликах значения должны быть не меньше (лучше равными), чем на мастере.

Размер структуры блокировок учитывается в строке:

```
SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM pg_shmem_allocations where name='<anonymous>';
     name      | allocated_size | pg_size_pretty
---------------+----------------+----------------
 <anonymous>   |        5003392 | 4886 kB
(1 row)
```

В данном примере структура блокировок занимает ~1Мб из 4,8Мб.

В столбце locktype указан один из **12 типов блокировки: relation, extend, frozenid, page, tuple, transactionid, virtualxid, spectoken, object, userlock, advisory, applytransaction**.

Представление можно соединить с pg_stat_activity, чтобы получить данные о сессии, удерживающей или ожидающей блокировку:

```
select * from pg_locks pl left join pg_stat_activity psa ON pl.pid = psa.pid;
```

При получении имени объекта из pg_class нужно помнить, что pg_class хранит данные только по локальным объектам базы данных:

```
select relation::regclass::text from pg_locks where locktype='relation' and database = (select oid from pg_database where datname=current_database());
```

Параметр `track_commit_timestamp`

- значение по умолчанию false и включать не стоит, так как экземпляр эти данные не использует
- в директории `PGDATA/pg_commit_ts` будет сохраняться метка времени фиксации транзакции
- данные о времени фиксации транзакции могут использоваться расширением pglogical для процедур разрешения конфликтов
- временные метки можно получить функциями:
  - `pg_last_committed_xact()`
  - `pg_xact_commit_timestamp()`
  - `pg_xact_commit_timestamp_origin()`
- значение параметра сохраняется в управляющем файле
- изменение значения требует рестарт экземпляра

Параметр `track_commit_timestamp`

Значение по умолчанию false. Если `track_commit_timestamp=true`, то **при фиксации транзакции** помимо сохранения статуса транзакции в директории `PGDATA/pg_xact`, в директории `PGDATA/pg_commit_ts` будет сохраняться метка времени фиксации транзакции. **Это не ускоряет работу экземпляра и сохраненная информация не используется экземпляром**. Причина появления функционал а : данные о времени фиксации используется расширением `pglogical` для процедур разрешения конфликтов. Если это расширение или аналогичные не используются, то устанавливать `track_commit_timestamp=true` не стоит.

Временные метки можно получить функциями:

```
pg_last_committed_xact()
pg_xact_commit_timestamp()
pg_xact_commit_timestamp_origin()
```

Например:

```
select pg_xact_commit_timestamp(xmin), xmin, * from t;
select * from pg_last_committed_xact();
```

Также утилита `pg_waldump` покажет временную метку фиксации транзакции. Пример:

```
rmgr: Transaction len (rec/tot):     34/    34, tx:        108, lsn: 0/021588,
prev 0/021550, desc: COMMIT 2025-01-01 11:11:00.000011 CET
```

Значение параметра сохраняется в управляющем файле (`pg_control`):

```
pg_controldata | grep timestamp
track_commit_timestamp setting:       off
```

Фиксацию транзакции также замедляет использование курсоров `WITH HOLD`.

Практика

- Часть 1. Запуск экземпляра в контейнере docker
- Часть 2. Разделяемая память экземпляра
- Часть 3. Локальная память серверного процесса
- Часть 4. Логирование памяти процесса в диагностический журнал
- Часть 5. Взаимоблокировки при проведении тестов
- Часть 6. Мультитранзакции
- Часть 7. Пример теста

Практика

Вы создадите контейнер с PostgreSQL 17 версии и посмотрите как падает экземпляр при выполнении простых команд.

Вы посмотрите распределение локальной и разделяемой памяти, как выделяется и освобождается память.

10

Буферный кэш

Структуры памяти, обслуживающие буферный кэш

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

Структуры памяти, обслуживающие буферный кэш

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

Дальше приводятся названия структур в разделяемой памяти экземпляра, относящиеся к буферному кэшу и формулы расчета их размера в байтах. Названия приведены как в представлении `pg_shmem_allocations`. Названия типов, макросов приведены, чтобы было удобно искать текст в исходном коде PostgreSQL, если захочется детально изучить алгоритмы.

Buffer Blocks - сам буферный кэш. Размер каждого буфера равен размеру блока. Точный размер выделенной памяти: `NBuffers * (BLKSZ=8196) + (PG_IO_ALIGN_SIZE=4096)`. `NBuffers` - количество разделяемых буферов задается параметром конфигурации `shared_buffers` (по умолчанию 16384, максимум 1073741823=30 бит).

Buffer Descriptors - описатели (заголовки) буферов. Структура описателя называется `BufferDesc`. Располагается в отдельной части памяти, один описатель для каждого буфера. Размер: `NBuffers * (BufferDescPadded = 64)` - описатели выровнены по cache line, который у современных процессоров обычно 64 байта. В этих 64 байтах находятся:

1) структура `BufferTag`, в которой указан прямой (самодостаточный, то есть хранящий всё чтобы найти файл и в нём блок) адрес блока на диске:

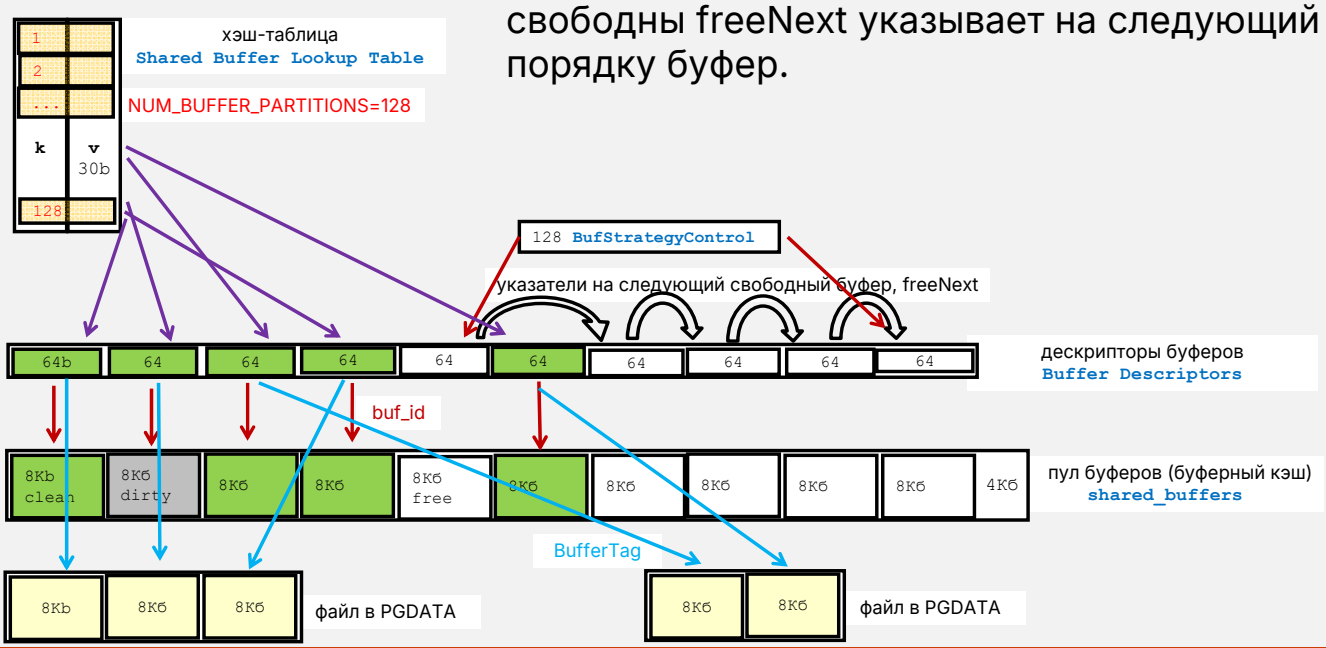
```
typedef struct buftag
{
 Oid  spcOid;  oid табличного пространства (название симлинка в PGDATA/pg_tblspc)
 Oid  dbOid;  oid базы данных (поддиректория)
 RelFileNumber relNumber;  название файла, представляет собой число
 ForkNumber  forkNum;  номер форка (enum на 5 возможных значений: -1 invalid, 0 main, 1 fsm, 2 vm, 3 init)
 BlockNumber blockNum;  номер блока относительно 0 блока 0 файла, размер 4 байта, максимум задается макросом MaxBlockNumber
} BufferTag;
```

Размер `BufferTag` 17 байт. Размер с выравниванием 20 байт.

2) `int buf_id` - порядковый номер буфера в кэше буферов начиная с нуля.

Структуры памяти, обслуживающие буферный кэш

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



Структуры памяти, обслуживающие буферный кэш (продолжение)

3) 32 бита, которые содержат: 18 бит `refcount`, 4 бита `usage count` (от 0 до BM_MAX_USAGE_COUNT=5, всего 6 градаций), 10 бит флагов, в которых отражается:

1 - BM_LOCKED стоит блокировка на заголовок буфера

2 - BM_DIRTY **грязный**

3 - BM_VALID блок не поврежден

4 - BM_TAG_VALID блок существует в файле на диске

5 - BM_IO_IN_PROGRESS буфер в процессе заполнения образом с диска или записи на диск

6 - BM_IO_ERROR предыдущая операция ввода-вывода сбойнула

7 - BM_JUST_DIRTIED в процессе записи на диск загрязнился

8 - BM_PIN_COUNT_WAITER **ждет снятия закреплений другими процессами чтобы заблокировать буфер для изменения**

9 - BM_CHECKPOINT_NEEDED **помечен процессом контрольной точки для записи на диск**

10 - BM_PERMANENT относится к журналируемому объекту.

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

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

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

Закрепление препятствует замене в буфере блока другим блоком.

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

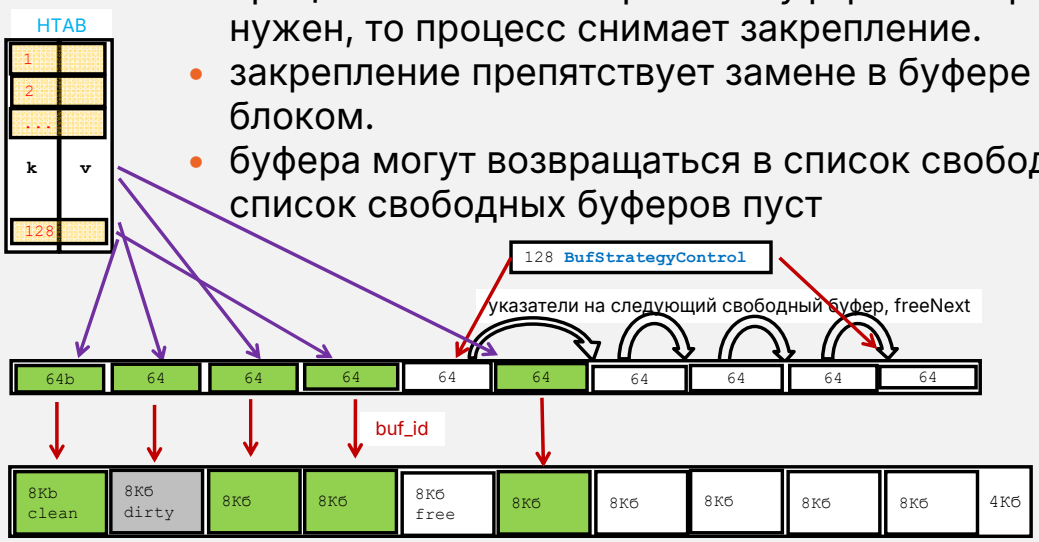
5) `int freeNext` - ссылка на номер следующего свободного блока. **После запуска экземпляра, пока все буфера свободны указывает на следующий по порядку буфер**. Используется логика связанного списка ("Linked List").

6) `LWLock content_lock` - легковесная блокировка на содержимое буфера

Легковесная блокировка buffer content_lock устанавливаются процессами на короткое время. Два вида: Exclusive и Shared. Exclusive препятствует установке блокировки другими процессами, Shared могут установить несколько процессов.

Поиск свободного буфера

- если процесс хочет работать с блоком, то он его ищет в буферном кэше. Если находит, то закрепляет. Множество процессов может закрепить буфер. Если процессу буфер не нужен, то процесс снимает закрепление.
- закрепление препятствует замене в буфере блока другим блоком.
- буфера могут возвращаться в список свободных. Обычно список свободных буферов пуст



Поиск свободного буфера

Buffer Strategy Status размер: `BufferStrategyControl = 128`. Хранит данные для поиска свободных блоков:

`slock_t buffer_strategy_lock;` `Spinlock` - для доступа к этой структуре

`pg_atomic_uint32 nextVictimBuffer;` счетчик поиска свободных буферов, указатель на следующий свободный буфер получается остатком от деления на NBuffers

`int firstFreeBuffer;` первый неиспользовавшийся (после рестарта экземпляра) буфер, после того как все буфера начнут использоваться примет значение -1.

`int lastFreeBuffer;`

`uint32 completePasses;` используется для статистики

`pg_atomic_uint32 numBufferAllocs;` используется для статистики

`int bgwprocno;` номер процесса bgwriter для уведомления

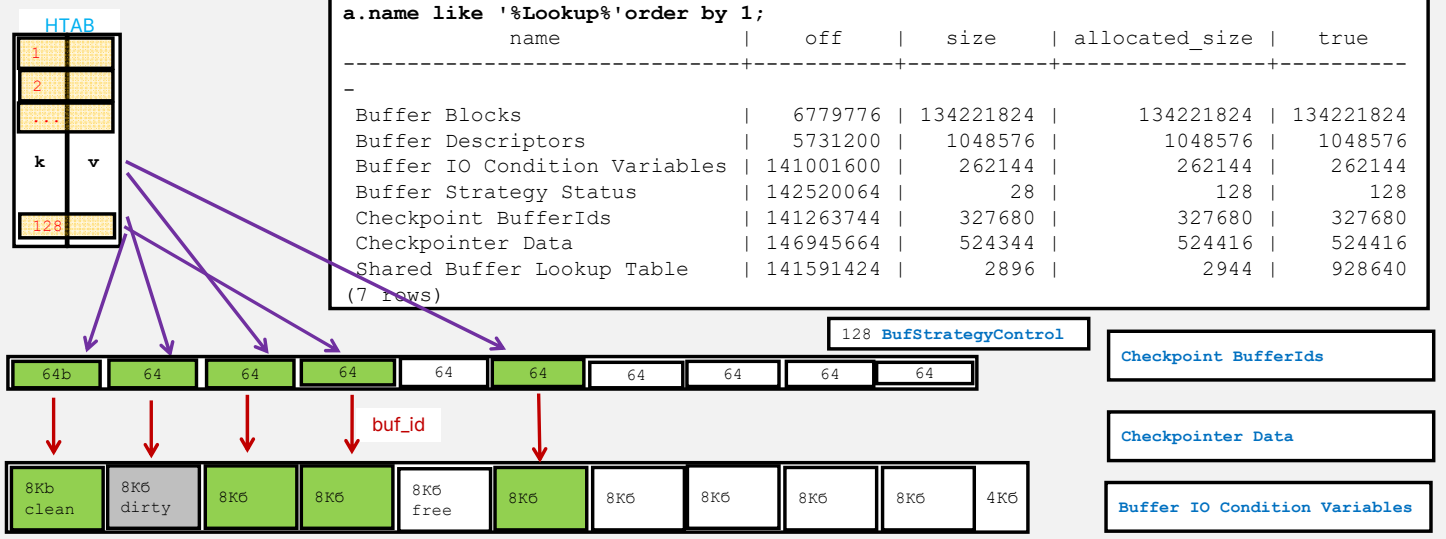
Слово "стратегия" используется в смысле "метод" из фразы "buffer cache replacement strategy".

Буфера могут возвращаться в список свободных функцией **StrategyFreeBuffer(**BufferDesc) и цепочка свободных буферов может дополняться и и даже создаваться заново после того как `firstFreeBuffer` стал равен `-1`. Такое происходит после удаления объекта, когда буфера, содержащие его блоки инвалидируются функией InvalidateBuffer(BufferDesc), которая вызывается из функции DropRelationsAllBuffers(..) или при удалении отдельного файла объекта функцией DropRelationBuffers(..) или усечении файла (в том числе вакуумом) функцией RelationTruncate() вызывающей smgrtruncate(..) или при удалении базы данных функцией DropDatabaseBuffers(dbid). При расширении файлов объекта функцией ExtendBufferedRelShared(...), когда расширения происходили одновременно и другой процесс уже подгрузил блок в буфер, пока первый процесс очищал буфер. Очищенный буфер будет возвращен первым процессом в список свободных. Или процесс намеревается использовать буфер функцией BufferAlloc(...) для загрузки блока, но пока процесс расчищал буфер, другой процесс уже загрузил блок в другой буфер. Первый процесс в этом случае возвратит очищенный блок в список свободных. Почему такое происходит? Потому что доступ к структурам построен на легковесных блокировках, которые не оптимально долго удерживать и массово устанавливать. Процесс разбивает большую задачу на подзадачи и устанавливает легковесные блокировки в подзадачах только там где они нужны. Процесс снимает блокировку, выполняет другие подзадачи, возвращается к первой и снова получает блокировку, а объект за это время был изменен.

Алгоритм вытеснения грязных буферов

- процесс checkpointer перед посылкой грязных блоков на запись сортирует их отдельно по каждому файлу

```
select * from (select *, lead(off) over(order by off) - off as true from
pg_shmem_allocations) as a where a.name like 'Buf%' or a.name like 'Check%' or
a.name like '%Lookup%'order by 1;
             name                |    off    |   size    | allocated_size |   true
---------------------------------+-----------+-----------+----------------+----------
-
 Buffer Blocks                   |   6779776 | 134221824 |      134221824 | 134221824
 Buffer Descriptors              |   5731200 |   1048576 |        1048576 |   1048576
 Buffer IO Condition Variables   | 141001600 |    262144 |         262144 |    262144
 Buffer Strategy Status          | 142520064 |        28 |            128 |       128
 Checkpoint BufferIds            | 141263744 |    327680 |         327680 |    327680
 Checkpointer Data               | 146945664 |    524344 |         524416 |    524416
 Shared Buffer Lookup Table      | 141591424 |      2896 |           2944 |    928640
(7 rows)
```



Алгоритм вытеснения грязных буферов

Checkpoint BufferIds размер: `NBuffers*(CkptSortItem=20)` 320Кб если буферный пул 128Мб. Память выделяемая с с запасом для сортировки грязных буферов, которые будут записаны на диск по контрольной точке. **Процесс checkpointer перед посылкой грязных блоков на запись сортирует их отдельно по каждому файлу**. Если файл на HDD, это уменьшает перемещения головок HDD.

Команда на запись на диск посылается через `checkpoint_flush_after` блоков (от 0 до 2Мб, по умолчанию 256Кб), `bgwriter_flush_after`=512Кб, `wal_writer_flush_after`=1Мб

Checkpointer Data размер: `CheckpointerShmemSize() = 524416`

Buffer IO Condition Variables NBuffers * (ConditionVariableMinimallyPadded = 16) флаги (переменные), которые переводят процесс в сон и пробуждают его если флаг изменится. Реализуют метод ожидания.

Shared Buffer Lookup Table, прежнее название `Buffer Mapping Table`.Третья по важности и вторая по размеру структура. В отличие от других структур является хэш-таблицей. Не самая оптимальная (нет поиска по диапазону), но приемлемая.

Размеры девяти структур, в том числе `Shared Buffer Lookup (Mapping)Table` в `pg_shmem_allocations` выдаются не верные. Оценить размеры структур можно по столбцу off (offset) от соседней структуры:

```
select * from (select *, lead(off) over(order by off) - off as true from
pg_shmem_allocations) as a where a.true<>a.allocated_size order by 1;
             name              |    off    | size | allocated_size |   true
-------------------------------+-----------+------+----------------+---------
 LOCK hash                     | 142635392 | 4944 |           4992 | 7210752
 PREDICATELOCK hash            | 159427456 | 4944 |           4992 | 5940096
 PREDICATELOCKTARGET hash      | 157270016 | 2896 |           2944 | 2157440
 PredXactList                  | 165367552 |   88 |            128 |  890624
 Proc Header                   | 167902336 |  112 |            128 |  475136
 PROCLOCK hash                 | 149846144 | 9040 |           9088 | 7419648
 RWConflictPool                | 166496256 |   24 |            128 | 1272192
 SERIALIZABLEXID hash          | 166258176 | 2896 |           2944 |  238080
 Shared Buffer Lookup Table    | 141706624 | 2896 |           2944 |  928640
(9 rows)
```

Размер 928640 байт для буферного кэша размера 128Мб.

Стратегии замены буферов

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

Стратегии замены буферов

Число записей в BufTable это сумма NBuffers и NUM_BUFFER_PARTITIONS из-за особенностей инициализации описателей блоков. Для эффективности функция BufferAlloc() при подгрузке блока в буфер, который занимал другой блок, первым делом вставляет в BufTable запись со ссылкой на описатель нового блока и только потом освобождает запись в BufTable со ссылкой на описатель старого блока. Чтобы не случилось ситуации, что в BufTable нет места для вставки ссылки на описатель нового блока (обработать которую сложно) в таблицу сразу добавляется дополнительное место. Поскольку параллельная работа с содержимым BufTable ограничена количеством партиций, то место под запасные записи выделяется в количестве партиций, число которых установлено макросом NUM_BUFFER_PARTITIONS и равно 128.

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

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

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

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

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

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

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

Поиск блока в буферном кэше

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

Поиск блока в буферном кэше

Процессу требуется работать с блоком. Процесс:
1) создает в своей локальной памяти экземпляр структуры BufferTag.
2) вычисляет хэш функцией `uint32 newHash = BufTableHashCode(BufferTag)`
3) по значению хэша определяет номер партиции `newPartitionLock = BufMappingPartitionLock(newHash)`
4) запрашивает легковесную (`LWLock`) блокировку типа `BufMappingLock` партиции хэш-таблицы в которую попал хэш: `LWLockAcquire(newPartitionLock, LW_SHARED);`
3) вызывает функцию `BufTableLookup(BufferTag, uint32 hashcode)` которая возвращает порядковый номер блока в кэше буферов типа int или -1 если блока нет в кэше.

Размер записи (hash bucket slot) в **Shared Buffer Lookup Table** 8 байт, она состоит из хэша (тип uint32, беззнаковое целое размером 4 байта) и порядкового номера буфера (его заголовка) типа int.

Число блоков может быть больше числа буферов и хэш от разных блоков может совпасть. В этом случае в таблицу вставляются записи с тем же значением ключа, но со ссылками на разные буфера (cache chains).

Таблица разделена на NUM_BUFFER_PARTITIONS=128 частей. Один процесс может получить блокировки на несколько частей, даже на все части. Блокировка удерживается недолго: по номеру буфера в записи без блокировок считывается заголовок буфера (`Buffer Descriptors`), атомарной операцией (`pg_atomic_read_u32(&buf->state)`) увеличивается refcount (он же ref_count, 18 бит), usage_count (4 бита) которые хранятся с флагами (10 бит) в 4 байтах. LWLock:BufMappingLock тут же снимается и только потом в заголовке буфера устанавливается LWLock:content_lock которая обеспечивает доступ к буферу и остальному содержимому заголовка.

Стоило бы увеличить NUM_BUFFER_PARTITIONS? Значение не зависит от размера буферного кэша, оно зависит от min(ядер центральных процессоров, активных процессов), но больше от длительности удержания LWLock. Если NUM_BUFFER_PARTITIONS было бы увеличено до 1024, то пришлось бы увеличить MAX_SIMUL_LWLOCKS=200, оно должно быть не меньше. Простое увеличение не дает эффекта и в некоторых форках PostgreSQL меняют алгоритм работы легковесных блокировок вводя очередь параметром конфигурации `lwlock_shared_limit`. Сложные алгоритмы не всегда оправданы из-за накладных расходов. Например, в linux на NVMe оптимален планировщик "none". https://www.postgresql.org/message-id/8c4a5f06-7476-4646-bb8a-6581a26b0650%40enterprisedb.com

Закрепление буфера (pin) и блокировка content_lock

- закрепление (pin) используется для того, чтобы блок в буфере не был заменен на другой
- для чтения или изменения содержимого блока в буфере нужна легковесная блокировка content_lock, ссылка на которую сохраняется в описателе блока `Buffer Descriptors`
- блокировка должна удерживаться короткое время, в отличие от pin
- для удаления места занимаемого строкой (HOT cleanup или vacuum) после pin и Exclusive процесс дожидается, чтобы у других процессов не стало pin (то есть pincount=1)
- для добавления в блок новой строки или изменения xmin, xmax существующих строк процесс должен получить блокировку content_lock типа Exclusive
- если процесс имеет pin и Shared content_lock, то он может менять некоторые биты в t_infomask, в частности статус фиксации/отката

Закрепление буфера (pin) и блокировка content_lock

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

1. Для доступа к строкам и их заголовкам в блоке устанавливаются: pin и content_lock (Exclusive или Shared в зависимости от намерений процесса).

2. После нахождения нужных строк content_lock может быть снят, но pin не сниматься и в таком режиме процесс сможет читать строки блока, которые увидел пока у процесса был content_lock.

3. Для добавления в блок новой строки или изменения xmin, xmax существующих строк процесс должен получить content_lock типа Exclusive. При наличии Exclusive никто не может иметь Shared content_lock и соответственно видеть новые строки, которые в процессе изменения. Старые строки могут продолжать читать, так как они все равно не меняются: очиститься и заморозиться не могут из-за удержания горизонта событий.

4. Если процесс имеет pin и Shared content_lock, то он может менять некоторые биты в t_infomask, в частности статус фиксации/отката. Эти биты могут даже потеряться, в таком случае процесс просто перепроверит статус транзакции. Менять биты и xmin которые относятся к заморозке при этом нельзя, для этого нужнен Exclusive content_lock и такие изменения логируются. А что с контрольными суммами? При изменении любых битов контрольная сумма станет другой, но контрольная сумма меняется перед записью блока на диск.

5. Для удаления места занимаемого строкой (HOT cleanup или vacuum) после pin и Exclusive процесс дожидается, чтобы у других процессов не стало pin (то есть pincount=1). После достижения pincount=1 (и получения Exclusive если был снят) можно освобождать место. Интересно что другие процессы при этом могут увеличивать pincount (закреплять блок, показывая намерение работать с его содержимым, ведь подгрузить блок в другой буфер не могут), так как из-за Exclusive не смогут установить Shared которая нужна чтобы заглянуть в блок.

Если pincount>1, то (авто)вакуум вписывает себя в поле описателя блока waiting for pincount 1, снимает Exclusive и ждёт. HOT cleanup не ждёт. Ожидающий может быть только один, но это норма, так как таблицу может очищать только один процесс вакуума.

Освобождение буферов при удалении файлов

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

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

CreateAdndDrop.sql:

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


Освобождение буферов при удалении файлов

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

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

```
#define BUF_DROP_FULL_SCAN_THRESHOLD    (uint64) (NBuffers / 32)
```

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

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

Скорость создания и удаления небольшой таблицы командами:

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

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

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

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

Оптимизированное расширение файлов

- для расширения размера файлов используются две функции `mdzeroextend(SMgrRelation reln, ForkNumber forknum, BlockNumber blocknum, int nblocks, bool skipFsync)` и `mdextend(..)`
- функция `mdzeroextend(..)` появилась в 16 версии и может расширять файл сразу несколькими блоками (8Кб) за один вызов к операционной системе
- при расширении файлов более, чем на 8 блоков (64Кб) используется системный вызов `posix_fallocate()`. При этом:
  - команда на забивание блоков нулями на диск не передаётся, но будет считаться, что блоки забиты нулями
  - для добавляемых блоков не выделяется место под страницы (по 4Кб) в страничном кэше linux
  - файловые системы могут не выделять реальное место на диске (delayed allocation) и помечать в метаданных, что диапазоны блоков содержат нулевые значения (sparse file)

Оптимизированное расширение файлов

Для расширения размера файлов используются функции `mdzeroextend(SMgrRelation reln, ForkNumber forknum, BlockNumber blocknum, int nblocks, bool skipFsync)` или `mdextend(SMgrRelation reln, ForkNumber forknum, BlockNumber blocknum, const void *buffer, bool skipFsync)`.`mdextend(..)` расширяет файл по одному блоку (8Кб). Функция `mdzeroextend(..)` появилась в 16 версии и может расширять файл сразу несколькими блоками (8Кб) за один вызов к операционной системе. При расширении файлов более, чем на 8 блоков (64Кб) используется системный вызов posix_fallocate(). При этом:

1) команда на забивание блоков нулями через шину ввода-вывода на диск не передаётся, но файловая система и система хранения (если в системе хранения реализовано) будет считать, что блоки забиты нулями

2) для добавляемых блоков не выделяется место под страницы (по 4Кб) в страничном кэше linux

3) файловые системы могут не выделять реальное место на диске (delayed allocation) и помечать в метаданных, что диапазоны блоков содержат нулевые значения (sparse file). При записи в такие блоки файловая система будет выделять место. В файловых системах, которые не рекомендуется использовать и которые обычно реализованы с ошибками (xfs) может выдаваться ложная ошибка: `could not extend file "..." with FileFallocate()` из-за повреждения метаданных на таких файловых системах.

Для справки: если число блоков меньше 8, то используется цепочка вызовов FileZero(..)->pg_pwrite_zeros(..)->pg_pwritev_with_retry(..); для записи в WAL используется pg_pwrite_zeros(..) если установлен параметр конфигурации wal_init_zero= true, если не установлен, то используется pg_write(..).

Реализация записи в 16 версии:

https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=4d330a61bb1969df31f2cebfe1ba9d1d004346d8

Изменение размера файлов и буферный кэш

- если в файле нет места, то файлы relations расширяются блоками от 1 до 64
- автовакуум и вакуум могут на последнем этапе обработки таблицы усечь последний файл форка main или даже удалить файлы форка, если не меньше чем в **тысяче блоков** с конца форка нет строк
  - Если монопольная блокировка на таблицу не будет получена за **5 секунд**, то усечение не выполняется
  - Если монопольная блокировка будет получена, то **через хэш-таблицу** ищутся буфера с блоками, которые будут усечены
  - Если не будут найдены, то полностью сканируются дескрипторы буферов

Изменение размера файлов и буферный кэш

Если в файле нет места, то файлы relations расширяются блоками от 1 до `MAX_BUFFERS_TO_EXTEND_BY=64` блока. Для команд, обрабатывающих большое число строк используется признак BulkInsertStateData bistate. Если файл расширялся набором блоков, то, скорее всего, будет и дальше расширяться. Поэтому последующие расширения будут выполняться наборами блоков того же размера. Это улучшает производительность тем, что предотвращает переключение вызова функции mdzeroextend() между разными системными вызовами: posix_fallocate() и write(), что снижало бы производительность. Граница переключения с расширения по одному блоку на несколько, если команде потребуется не меньше чем 8 блоков (64Кб).

Автовакуум и вакуум могут на последнем этапе обработки таблицы усечь последний файл форка main или даже удалить файлы форка, если не меньше чем в тысяче блоков с конца форка нет строк. Точная формула:

```
min((REL_TRUNCATE_MINIMUM=1000) or (relsize/16))
```

Усечение выполняется функцией lazy_truncate_heap(..), которая пытается получить монопольную блокировку на таблицу, но если не сможет получить блокировку за VACUUM_TRUNCATE_LOCK_TIMEOUT=5 секунд, то усечение не выполняется. Если сможет получить блокировку, то выполняет RelationTruncate(..) которая вызывает smgrtruncate(..), которая вызывает DropRelationBuffers(..) которой надо освободить буфера, которые хранят усекаемые блоки. Если хоть один буфер не освободится, то bgwriter или checkpointer аварийно завершат работу и экземпляр перезапустится. Функция сначала пытается найти буфера через хэш-таблицу. Если не найдет хоть один буфер, то полностью сканирует дескрипторы буферов. Можно отключить эту фазу работу автовакуума параметрами на уровне таблицы и даже для TOAST-таблицы отдельно. Примеры команд:

```
CREATE TEMP TABLE reloptions_test (i INT NOT NULL, j text)
  WITH (vacuum_truncate=false, toast.vacuum_truncate=false);
SELECT reloptions FROM pg_class WHERE oid = 'reloptions_test'::regclass;
               reloptions
-----------------------------------------------
 {vacuum_truncate=false,autovacuum_enabled=false}
ALTER TABLE reloptions_test RESET (vacuum_truncate);
VACUUM (TRUNCATE FALSE, FULL TRUE) reloptions_test;
```

Для мониторинга есть **событие ожидания VacuumTruncate**.

Предварительное чтение блоков (prefetch)

- предварительное чтение блоков данных в страничный кэш linux
- реализуются вызовом к операционной системе вызовом `posix_fadvise(VfdCache[file].fd, offset, amount, POSIX_FADV_WILLNEED)`
- для мониторинга нет ли задержек связанных с этим вызовом есть **событие ожидания DataFilePrefetch**
- индексный метод доступа Bitmap Heap Scan использует prefetch на фазе сканирования блоков таблицы
- вакуум и анализ также использует prefetch. Ограничение на количество буферов устанавливается параметром `maintenance_io_concurrency`
- в 17 версии появился параметр `io_combine_limit`
  - По умолчанию 128Кб
  - Диапазон значений от 8Кб до 256Кб
  - Устанавливает сколько блоков можно объединить в один вызов к linux

Предварительное чтение блоков (prefetch)

Prefetch - предварительное чтение блоков данных в страничный кэш linux. Для этого используется системный вызов с параметром `POSIX_FADV_WILLNEED`, который указывает операционной системе прочесть блок (размером `BLCKSZ=8Кб`). Вызов называют "асинхронным" потому, что возвращает результат без задержки, до реального чтения блоков из файла.

Для предварительного чтения используется функция PrefetchBuffer(Relation reln, ForkNumber forkNum, BlockNumber blockNum), которая вызывает PrefetchLocalBuffer(..) для временных объектов или PrefetchSharedBuffer(..) для объектов использующих буферный кэш. Функции в конечном итоге обращаются к linux вызовом `posix_fadvise(VfdCache[file].fd, offset, amount, POSIX_FADV_WILLNEED)`.

Для мониторинга нет ли задержек связанных с этим вызовом есть **событие ожидания DataFilePrefetch**.

Индексный метод доступа Bitmap Heap Scan использует prefetch на фазе сканирования блоков таблицы.

Вакуум и анализ также использует prefetch. Ограничение на число буферов устанавливается параметром `maintenance_io_concurrency`.

Расширение `pg_prewarm` также может использовать prefetch.

В 17 версии появился параметр `io_combine_limit`, по умолчанию 128Кб. Диапазон значений от 8Кб до 256Кб. Устанавливает сколько блоков можно объединить в один вызов к операционной системе, чтобы снизить число системных вызовов. До 17 версии объединения нет.

Сколько данных может заранее (read-ahead) прочесть контроллер диска с конкретного блочного устройства:

```
postgres@tantor:~$ sudo blockdev --getra /dev/sda
256
```

Измеряется в 512 байт, по умолчанию 256 * 512 байт = 128Кб

Для набора дисков в RAID можно установить большие значения:

```
postgres@tantor:~$ blockdev --setra 4096 /dev/sda
```

Представление pg_stat_recovery_prefetch

- предварительное чтение используется процессом startup
- параметр recovery_prefetch
  - установлен по умолчанию в значение try
  - используется процессом startup при запуске экземпляра и на репликах
- параметр wal_decode_buffer_size
  - устанавливает размер журнальных записей, которые процесс startup заранее читает, чтобы определить блоки для предварительной загрузки в кэш буферов
  - значение по умолчанию 512Кб

```
select * from pg_stat_recovery_prefetch \gx
stats_reset     | 2044-01-01 11:11:11.000000+03
prefetch        | 2242   число prefetched блоков
hit             | 52138  не загружались, уже находились в кэше буферов
skip_init       | 139    не загружались, так как инициализированлись нулями
skip_new        | 6      не загружались, так как блоки не существовали
skip_fpw        | 1868   не загружались, так как в WAL был полный образ блоков
skip_rep        | 52272  команда prefetched уже была послана
wal_distance    | 0      количество блоков WAL, прочитанных заранее для prefetch
block_distance  | 0      число блоков в процессе предвыборки
io_depth        | 0      число необработанных вызовов prefetch
```


Представление pg_stat_recovery_prefetch

Предварительное чтение используется процессом startup. Восстановление выполняет один процесс и предварительная выборка ускоряет его работу. При работе физической реплики процесс walreceiver сохраняет блоки WAL и они будут в страничном кэше, если нет задержки в накате изменений из-за параметров max_standby_streaming_delay (и долгого запроса на реплике) и recovery_min_apply_delay.

Параметр recovery_prefetch установлен по умолчанию в значение try. Менять значение не нужно. Используется процессом startup. Этот процесс работает после запуска экземпляра и постоянно на репликах. Процесс читает WAL-файлы. В журнальных записях есть ссылки на блоки, которые меняются журнальными записями и образы блоков. Процесс startup вызывает функцию PrefetchBuffer(..) по этим блокам, которая устанавливает рекомендацию операционной системе прочесть блоки в страничный кэш.

Параметр wal_decode_buffer_size устанавливает размер журнальных записей, которые процесс startup заранее читает, чтобы определить блоки для предварительной загрузки в кэш буферов, если recovery_prefetch не отключён. Если значение указано без единиц измерения, оно считается заданным в байтах. Значение по умолчанию 512Кб.

В представлении отражается статистика предварительной выборки. Если все значения нули, то при запуске экземпляра восстановления не было (экземпляр был остановлен корректно).

В столбцах wal_distance, block_distance и io_depth текущие значения, в других столбцах накопленная с момента запуска экземпляра статистика, которую можно сбросить функцией:

```
 select pg_stat_reset_shared('recovery_prefetch');
select * from pg_stat_recovery_prefetch\gx
-[ RECORD 1 ]--+-----------------------------
stats_reset    | 2044-01-01 11:11:11.000000+03
prefetch       | 2242  число prefetched блоков
hit            | 52138 не загружались, уже находились в кэше буферов
skip_init      | 139   не загружались, так как инициализированлись нулями
skip_new       | 6     не загружались, так как блоки не существовали
skip_fpw       | 1868  не загружались, так как в WAL был полный образ блоков
skip_rep       | 52272 команда на предварительное чтение уже была послана
wal_distance   | 0     количество блоков WAL, прочитанных заранее для prefetch
block_distance | 0     число блоков в процессе предвыборки
io_depth       | 0     число необработанных вызовов prefetch
```

Расширение pg_prewarm

- стандартное расширение
- устанавливается добавлением библиотеки в параметр:
  `alter system set shared_preload_libraries='pg_prewarm';`
- сохраняет в текстовый файл в корне директории PGDATA адрес каждого блока, который загрузит в кэш буферов после перезапуска экземпляра

```
postgres=# select autoprewarm_dump_now();
 autoprewarm_dump_now
----------------------
                  264
(1 row)
postgres@tantor:~$ head $PGDATA/autoprewarm.blocks
<<264>>
0,1664,1262,0,0
0,1664,6100,0,0
5,1663,1259,0,0
...
```

```
postgres=# \dconfig *prewarm*
    List of configuration parameters
            Parameter             | Value
----------------------------------+-------
 pg_prewarm.autoprewarm           | on
 pg_prewarm.autoprewarm_interval  | 5min
(2 rows)
postgres=# select pg_prewarm('pg_class');
 pg_prewarm
------------
         19
```


Расширение pg_prewarm

Стандартное расширение, реализующее периодическое (или при остановке экземпляра) сохранение адресов блоков, находящихся в буферном кэше в текстовый файл и подгрузку этих блоков в кэш после рестарта экземпляра. Название файла `autoprewarm.blocks` и не меняется. Файл создается в директории `PGDATA`. В первой строке файла число блоков, данные о которых сохранены в файле.

В каждой строке 5 полей с адресом одного блока: `db, tbs, file, fork, block`.

Расширение устанавливается добавлением библиотеки в параметр:

`alter system set shared_preload_libraries='pg_prewarm';`

Функции (если они нужны) устанавливаются командой **`create extension pg_prewarm;`**

После рестарта экземпляра запускается фоновый процесс `autoprewarm leader`, который пока работает экземпляр или при остановке экземпляра сохраняет в фоновом режиме данные в файл. При рестарте экземпляра процесс `leader` сортирует список блоков из файла и запускает `bgworker`, который последовательно подключается к базам и загружает блоки объектов баз в кэш буферов. У расширения два параметра:

```
\dconfig *prewarm*
 pg_prewarm.autoprewarm           | on
 pg_prewarm.autoprewarm_interval  | 5min
```

Параметром `pg_prewarm.autoprewarm` можно отключить запуск процесса `leader`. Изменение этого параметра требует перезапуск экземпляра. Запустить или остановить фоновый процесс пока экземпляр работает нельзя, функция `autoprewarm_start_worker()` бесполезна. Частота сохранения данных устанавливается параметром `pg_prewarm.autoprewarm_interval`. По умолчанию 5 минут. Если установить ноль, то файл со списком блоков не будет обновляться.

Функция `autoprewarm_dump_now()` немедленно обновляет или создает файл `autoprewarm.blocks`. Это может быть полезно, если `leader` не запущен, но нужно, чтобы после перезапуска экземпляра блоки были подгружены в буферный кэш. Функция возвращает количество блоков, имеющихся в файле `autoprewarm.blocks`.

Можно загрузить блоки одного relation в буферный кэш (`mode=buffer`) или только в страничный кэш синхронным вызовом (`mode=read`) или `posix_fadvise(.., POSIX_FADV_WILLNEDED))` (`mode=prefetch`) функцией у которой 5 параметров, 4 из которых имеют значения по умолчанию. При вызове функции достаточно указать имя объекта: **`select pg_prewarm('pg_class', 'prefetch');`** Функция вернет число загруженных блоков.

Процесс фоновой записи bgwriter

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

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


Процесс фоновой записи bgwriter

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

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

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

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

Что если в предыдущих итерациях серверные процессы не использовали новые буфера? Чтобы не было "медленного старта", в итерации будет сканироваться не меньше, чем:
`NBuffers/120000*bgwriter_delay+reusable_buffers_est` блоков.

Для размера кэша буферов 128Мб и 200 миллисекундами задержки получится **27**`+reusable_buffers_est` блоков.

Алгоритм очистки кэша буферов процессом bgwriter

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

Алгоритм очистки кэша буферов процессом bgwriter

Запись блока выполняется функцией SyncOneBuffer(..). Сначала берется спин-блокировка на дескриптор блока и устанавливается бит BM_LOCKED. Проверяются значения: **refcount=0 (блок не нужен процессам), usage_count=0 (попадает в градацию давно не использовавшихся), бит BM_DIRTY=1 (грязный), BM_VALID=1** и если значения не такие как приведены, то спин-блокировка снимается и блок не сбрасывается на диск. Иначе буфер закрепляется, берется легковесная разделяемая блокировка, вызывается функция передачи буфера в страничный кэш linux, снимаются блокировка и закрепление.

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

Считывается LSN из блока в буфере и выполняется функция XLogFlush(XLogRecPtr record), сбрасывающая содержимое WAL-буфера вплоть до этого LSN. Этим гарантируется логика Write Ahead - журнал с изменениями в блоке должен быть записан раньше самого блока.

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

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

Проверяется набор флагов (BM_JUST_DIRTIED, BM_IO_IN_PROGRESS, BM_CHECKPOINT_NEEDED, BM_IO_ERROR) которые используются для отслеживания изменений в блоке в процессе записи на диск. Если флаги показывают что "всё чисто" (другие процессы не меняли содержимое блока), то в дескрипторе буфера снимается флаг BM_DIRTY и буфер становится "чистым" и снимается блокировка дескриптора блока.

Так как вытесняются давно неиспользовавшиеся (usage_count=0 и refcount=0) буфера, то вероятность того, что блок понадобится другому процессу мала; что будут ожидания получения блокировок; что потребуется запись в WAL. Функция XLogFlush(XLogRecPtr record) сначала проверяет, что LSN меньше чем тот, который уже записан в WAL.

Буфер не попадает в список свободных, буфер становится чистым.

Представление pg_stat_bgwriter

- отражает статистику эффективности работы bgwriter по всему экземпляру
- обнуление статистики: `pg_stat_reset_shared('bgwriter');`
- общее количество записанных на диск буферов: `buffers_backend +buffers_clean +buffers_checkpoint`
- buffers_backend - сколько буферов **очистили** (послали на запись) серверные процессы
- buffers_clean - сколько буферов очистил bgwriter
- **buffers_backend*100/buffers_alloc** в каком проценте случаев серверные процессы сталкивались с грязным буфером, должно быть меньше 1%

```
select * from pg_stat_bgwriter\gx
-[ RECORD 1 ]---------+-------------
checkpoints_timed     | 1567
checkpoints_req       | 97
checkpoint_write_time | 1463587
checkpoint_sync_time  | 59994
buffers_checkpoint    | 86752
buffers_clean         | 4234
maxwritten_clean      | 25
buffers_backend       | 1570184
buffers_backend_fsync | 0
buffers_alloc         | 450550
stats_reset           | 2025-01-01 09:13:
```


Представление pg_stat_bgwriter

Отражает статистику эффективности работы bgwriter по всему экземпляру. Представление содержит одну строку. В столбце `maxwritten_clean` выдаётся сколько раз bgwriter приостанавливал работу из-за того, что достиг `bgwriter_lru_maxpages` или расчетного значения количества блоков (с учетом `bgwriter_lru_multiplier`), если `bgwriter_lru_maxpages` уже установлен в максимальное значение.

Обнулить (синонимы: сбросить, reset) данные в представлении можно функцией `pg_stat_reset_shared('bgwriter');` Время обнуления указывается в столбце `stats_reset`. Вызов функции без параметров `pg_stat_reset()` не обнуляет статистику ни в этом представлении, ни в представлении `pg_stat_io`.

Общее количество записанных на диск буферов: `buffers_backend +buffers_clean +buffers_checkpoint`.

`buffers_clean` - сколько буферов **очистил** (послал на запись) bgwriter.

`buffers_backend` - сколько буферов **очистили** (послали на запись) серверные процессы и автовакуум. В процессе освобождения буферов чтобы использовать их под новые блоки серверные процессы наталкивались на грязные буфера и освобождали их посылая на запись, что не желательно, так как увеличивает время выполнения команд. Если `buffers_backend*2>buffers_clean` или `buffers_backend*100%/ (buffers_backend +buffers_clean +buffers_checkpoint)>~50%`, то работу bgwriter нужно сделать более агрессивной.

`buffers_alloc` - сколько блоков подгрузили ("allocated") с диска в буферный кэш серверные процессы. При обработке данных серверные процессы по хэш-таблице проверяют , есть ли нужные блоки буферном кэше. Если блоков нет в буферном кэше, то "выделяют" (allocate) буферы и загружают в них блоки. Что значит "выделили"? Если есть свободные буфера, то использовали их, а если нет (а их обычно нет, так как появиться они могут только в результате DROP, TRUNCATE), то пытались освободить буфера по алгоритму clock sweep. Напрямую значение ни о чем не говорит. Может быть буферный кэш слишком маленький относительно типичного объема блоков с которыми работали процессы с момента обнуления статистики. Может быть экземпляр только что запустился и серверные процессы наполняли кэш буферов.

buffers_backend*100/buffers_alloc - в каком проценте случаев серверные процессы сталкивались с необходимостью сбросить грязный буфер на диск. Это значение должно быть меньше 1%.

Для мониторинга обычно используют запросы, выводящие проценты:

```
 select to_char(100*checkpoints_timed::numeric / nullif((checkpoints_timed+checkpoints_req),0),'990D9')
  || ' %' "ckpt by time",
 to_char(100*checkpoints_req::numeric / nullif((checkpoints_timed+checkpoints_req),0),'990D9')
  || ' %' AS "ckpt by size",
 to_char(100*buffers_checkpoint::numeric / nullif((buffers_checkpoint+buffers_clean +
buffers_backend),0),'990D9')
  || ' %' "checkpointer",
 to_char(100*buffers_backend::numeric / nullif((buffers_checkpoint+buffers_clean + buffers_backend),0),'990D9')
  || ' %' "backend",
 to_char(100*buffers_backend_fsync::numeric / nullif((buffers_checkpoint+buffers_clean +
buffers_backend),0),'990D9')
  || ' %' "backend_fsync",
 to_char(100*buffers_clean::numeric / nullif((buffers_checkpoint+buffers_clean + buffers_backend),0),'990D9')
  || ' %' "bgwriter",
 pg_size_pretty((buffers_checkpoint+buffers_clean+buffers_backend)*8192/ (extract (epoch from current_timestamp -
stats_reset))::bigint) || ' / s' "speed"
 FROM pg_stat_bgwriter;
```

```
ckpt by time | ckpt by size | checkpointer | backend  | backend_fsync | bgwriter |    speed
-------------+--------------+--------------+----------+---------------+----------+-------------
 100.0 %     |    0.0 %     |  100.0 %     |    0.0 % |    0.0 %      |    0.0 % | 32 bytes / s
(1 row)
```

```
 SELECT
  clock_timestamp()-pg_postmaster_start_time()                    "Uptime",
  clock_timestamp()-stats_reset                                   "Since stats reset",
  round(100.0*checkpoints_req/total_checkpoints,1)                "Forced checkpoint ratio (%)",
  round(np.min_since_reset/total_checkpoints,2)                   "Minutes between checkpoints",
  round(checkpoint_write_time::numeric/(total_checkpoints*1000),2) "Average write time per checkpoint (s)",
  round(checkpoint_sync_time::numeric/(total_checkpoints*1000),2)  "Average sync time per checkpoint (s)",
  round(total_buffers/np.mp,1)                                    "Total MB written",
  round(buffers_checkpoint/(np.mp*total_checkpoints),2)           "MB per checkpoint",
  round(buffers_checkpoint/(np.mp*np.min_since_reset*60),2)       "Checkpoint MBps",
  round(buffers_clean/(np.mp*np.min_since_reset*60),2)            "Bgwriter MBps",
  round(buffers_backend/(np.mp*np.min_since_reset*60),2)          "Backend MBps",
  round(total_buffers/(np.mp*np.min_since_reset*60),2)            "Total MBps",
  round(1.0*buffers_alloc/total_buffers,3)                        "New buffer allocation ratio",
  round(100.0*buffers_checkpoint/total_buffers,1)                 "Clean by checkpoints (%)",
  round(100.0*buffers_clean/total_buffers,1)                      "Clean by bgwriter (%)",
  round(100.0*buffers_backend/total_buffers,1)                    "Clean by backends (%)",
  round(100.0*maxwritten_clean/(np.min_since_reset*60000/np.bgwr_delay),2) "Bgwriter halt-only length (buffers)",
  coalesce(round(100.0*maxwritten_clean/(nullif(buffers_clean,0)/np.bgwr_maxp),2),0) "Bgwriter halt ratio (%)",
  '------------------'  "--------------------------------",
  bgstats.*
   FROM (
     SELECT bg.*,
  checkpoints_timed + checkpoints_req total_checkpoints,
  buffers_checkpoint + buffers_clean + buffers_backend total_buffers,
  pg_postmaster_start_time() startup,
  current_setting('checkpoint_timeout') checkpoint_timeout,
  current_setting('max_wal_size') max_wal_size,
  current_setting('checkpoint_completion_target') checkpoint_completion_target,
  current_setting('bgwriter_delay') bgwriter_delay,
  current_setting('bgwriter_lru_maxpages') bgwriter_lru_maxpages,
  current_setting('bgwriter_lru_multiplier') bgwriter_lru_multiplier
     FROM pg_stat_bgwriter bg
  ) bgstats,
  (
     SELECT
  round(extract('epoch' from clock_timestamp() - stats_reset)/60)::numeric min_since_reset,
  (1024 * 1024 / block.setting::numeric) mp,
  delay.setting::numeric bgwr_delay,
  lru.setting::numeric bgwr_maxp
     FROM pg_stat_bgwriter bg
     JOIN pg_settings lru   ON lru.name = 'bgwriter_lru_maxpages'
     JOIN pg_settings delay ON delay.name = 'bgwriter_delay'
     JOIN pg_settings block ON block.name = 'block_size'
  ) np\gx
```

https://dataegret.com/2017/03/deep-dive-into-postgres-stats-pg_stat_bgwriter-reports/

Расширение pg_buffercache

- стандартное расширение
- создает представление pg_buffercache, являющееся обёрткой для функции pg_buffercache_pages() и две функции без параметров pg_buffercache_summary() и pg_buffercache_usage_counts()
- bgwriter не уменьшает usage_count, его уменьшают серверные процессы
- распределение буферов по шести "корзинкам" usage_count не должно иметь перекоса в одну или другую стороны

```
select * from pg_buffercache_usage_counts();
 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
           0 |   16000 |     0 |      0
           1 |      17 |     0 |      0
           2 |      23 |     1 |      0
           3 |      43 |     0 |      0
           4 |      24 |     2 |      0
           5 |     277 |     1 |      0
(6 rows)
```


Расширение pg_buffercache

Стандартное расширение. Создает представление pg_buffercache, являющееся обёрткой для функции pg_buffercache_pages() и две функции без параметров pg_buffercache_summary() и pg_buffercache_usage_counts(). Эти две функции не устанавливают блокировки на структуры памяти буферного кэша, в отличие от первой функции и работают быстро.

Для настройки работы процесса bgwriter особый интерес представляет:

```
select * from pg_buffercache_usage_counts();
```

bgwriter не уменьшает usage_count, его уменьшают серверные процессы. **Чтобы bgwriter очистил буфер нужно, чтобы usage_count=0 и pinned=0**. Если таких буферов нет, то bgwriter будет бессмысленно накручивать циклы, нагружая ядро процессора и не освобождая буфера. Если количество dirty при этом большое или серверные процессы часто сталкиваются с грязными буферами - например, buffers_backend*100/buffers_alloc>1%, то в этом случае нужно чтобы checkpointer работал активнее: стоит увеличить частоту контрольных точек. Процесс checkpointer сбрасывает **все** грязные буфера на диск, на usage_count внимание не обращает и не меняет его.

В 17 версии появилась функция, которая убирает один блок из кэша pg_buffercache_**evict**(буфер) и часть столбцов из pg_stat_bgwriter перенесли в новое представление pg_stat_bgwriter, часть (по серверным процессам) перенесли в pg_stat_io:

```
\d pg_stat_bgwriter
        View "pg_catalog.pg_stat_bgwriter"
      Column      |           Type           |
------------------+--------------------------+
 buffers_clean    | bigint                   |
 maxwritten_clean | bigint                   |
 buffers_alloc    | bigint                   |
 stats_reset      | timestamp with time zone |
postgres=# \d pg_stat_checkpointer
        View "pg_catalog.pg_stat_checkpointer"
     Column      |           Type           |
-----------------+--------------------------+
 num_timed       | bigint                   |
 num_requested   | bigint                   |
 write_time      | double precision         |
 sync_time       | double precision         |
 buffers_written | bigint                   |
 stats_reset     | timestamp with time zone |
```

https://www.postgresql.org/message-id/flat/20230210004604.mcszbscsqs3bc5nx%40awork3.anarazel.de

Настройка размера кэша буферов

- размер устанавливается параметром `shared_buffers`
- при изменении значения нужно перезапустить экземпляр
- очистить кэш нельзя, только рестартом экземпляра
- страничный кэш linux можно очистить
- распределение буферов по шести значениям `usage_count` не должно иметь явного перекоса в одну или другую стороны иначе алгоритм вытеснения неэффективен
- для оценки можно использовать столбец `usagecount_avg` значение которого должно быть примерно посередине интервала от 0 до 6 (2.5):

```
select * from pg_buffercache_summary();
buffers_used|buffers_unused|buffers_dirty|buffers_pinned|usagecount_avg
------------+--------------+-------------+--------------+--------------
         384 |        16000 |           0 |            0 | 4.37760416666
```


Настройка размера кэша буферов

Размер устанавливается параметром `shared_buffers`. При изменении значения нужно перезапустить экземпляр.

Столбцы `usage_count` и `buffers` помогают определить оптимален ли размер буферного кэша при текущей активности и настройках экземпляра. Если в столбце buffers есть перекос: в сторону `usage_count=5` это означает, что размер буферного кэша недостаточен. Если большая часть буферов в строке с `usage_count=1` размер буферного кэша можно уменьшить. Если 0, то либо экземпляр только что перегрузился и большая часть буферов свободна, либо размер буферного кэша можно уменьшить.

Распределение буферов по шести "корзинкам" `usage_count` не должно иметь явного перекоса в одну или другую стороны. При перекосе алгоритм вытеснения теряет эффективность. Для оценки можно использовать столбец `usagecount_avg` значение которого должно быть примерно посередине интервала от 0 до 6:

```
select * from pg_buffercache_summary();
buffers_used|buffers_unused|buffers_dirty|buffers_pinned|usagecount_avg
------------+--------------+-------------+--------------+--------------
         384 |        16000 |           0 |            0 | 4.37760416666
```

Стоит смотреть на само распределение - "фигуру" двумерного графика `usage_count` по горизонтали, buffers по вертикали выдаваемых табличной функцией **`pg_buffercache_usage_counts()`**.

Частный случай настройки кэша буферов когда он больше размера кластера. В этом случае вытеснения блоков из кэша нет и это наиболее оптимально, так как серверным процессам не нужно подгружать блоки и искать свободные блоки.

Пример вывода объектов, занимающих большую часть кэша:

```
create or replace view buffercache as
select relname, buffers, relpages pages, (buffers*8)/1024 as size_mb,
 usage, dirty, pins,
 round(((100*dirty::float8)/nullif(buffers,0)::float8)::numeric,2) as "dirty%",
 round(((100*buffers::float8)/nullif(relpages,0)::float8)::numeric,2) as
"cached%"
from
 (select c.relname, count(*) as buffers, sum(usagecount) as usage,
   count(*) filter (where isdirty) as dirty, sum(pinning_backends) as pins,
   max(c.relpages) as relpages
  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()))
group by c.relname)
order by buffers desc;

select * from buffercache limit 20;
```

relname	buffers	pages	size_mb	usage	dirty	pins	dirty%	cached%
t1	4492	0	35	22456	4490	0	99.96	
t	3850	0	30	19246	3848	0	99.95	
pg_class	**2802**	**2798**	21	14006	3	1	0.11	**100.14**
pgbench_accounts	**1698**	**1695**	13	8490	0	0	0.00	**100.18**
pgbench_history	**664**	**637**	5	3320	0	0	0.00	**104.24**
pgbench_accounts_pkey	276	276	2	1378	0	0	0.00	100.00
pg_class_oid_index	64	1879	0	210	1	0	1.56	3.41
pg_class_relname_nsp_index	63	4237	0	123	1	0	1.59	1.49
pg_toast_2618	46	84	0	106	0	0	0.00	54.76
pg_attribute	42	4421	0	165	4	0	9.52	0.95
pg_proc	35	103	0	89	0	0	0.00	33.98
pg_statistic	29	182	0	71	0	0	0.00	15.93
pg_attribute_relid_attnum_index	16	11663	0	78	1	0	6.25	0.14
pg_operator	14	14	0	66	0	0	0.00	100.00
pg_proc_proname_args_nsp_index	12	34	0	30	0	0	0.00	35.29
pg_depend_reference_index	12	6991	0	56	3	0	25.00	0.17
pg_type_typname_nsp_index	12	505	0	51	2	0	16.67	2.38
pg_proc_oid_index	11	12	0	40	0	0	0.00	91.67
pg_type	10	487	0	33	1	0	10.00	2.05
pg_depend_depender_index	10	9076	0	46	2	0	20.00	0.11

(20 rows)

Запрос определяет размер relation по статистике, которая может отсутствовать. В этом случае в столбце `cached%` будет выдано пустое значение, а в `pages` ноль.

`buffers > pages` так как кэшируются блоки слоёв fsm и vm, а также в случаях неактуальной статистики.

Параметр `synchronize_seqscans`

- при чтении таблицы размером больше 1/4 буферного кэша табличным методом Seq Scan
  - используется буферное кольцо
  - новый процесс "синхронизируется" с тем процессом, который уже сканирует блоки таблицы для совместного чтения блоков из кольца
  - блоки в буферном кэше, читаются работающими процессами примерно в одно время, а не подгружаются в буферный кэш несколько раз
- порядок строк, возвращаемых запросами, в которых отсутствует предложение ORDER BY, может быть произвольным
- по умолчанию включёно:

```
show synchronize_seqscans;
 synchronize_seqscans
----------------------
 on
```


Параметр `synchronize_seqscans`

Параметр по умолчанию включён. При чтении таблицы размером больше 1/4 буферного кэша табличным методом Seq Scan помимо того, что используется буферное кольцо, новый процесс "синхронизируется" с тем процессом, который уже сканирует блоки таблицы.

Читающий процесс считается лидером (leader), а процессы начавшие читать таблицу позже него followers. Followers начинают читать таблицу с последней позиции, которую читал лидер, а не с первого блока. Получается, что блоки в буферном кэше, читаются работающими процессами примерно в одно время, а не подгружаются в буферный кэш несколько раз. По достижении конца таблицы лидером один из не завершивших выборку процессов становится лидером. Новый лидер переходит к чтению таблицы с начала и читает те блоки, которые он не читал. Порядок строк, возвращаемых запросами, в которых отсутствует предложение ORDER BY, может быть произвольным.

Побочный эффект: лидер может закончить читать таблицу на произвольном блоке, адрес этого блока сохраняется (последняя позиция, обновленная лидером) и другие процессы начнут читать блоки этой таблицы с этой позиции.

Практика

- Часть 1. Расширение pg_buffercache
- Часть 2. Буферные кольца
- Часть 3. Расширение pg_prewarm
- Часть 4. Процесс фоновой записи bgwriter

Практика

В практике вы посмотрите использование расширений для диагностики и оптимизации наполнения буферного кэша.

11

Контрольная точка

Контрольная точка

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

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


Контрольная точка

Выполняется процессом checkpointer. Контрольные точки выполняются: периодически, в конце процедуры остановки и запуска экземпляра экземпляра, продвижении реплики, резервировании, команде checkpoint, создании базы. На реплике контрольные точки не инициируются, но выполняются restart points. В случае падения экземпляра и последующего перезапуска алгоритм контрольной точки должен гарантировать, что журнальные данные начиная с LSN начала успешно завершившейся, то есть записанной в pg_control (на последней фазе выполнения) контрольной точки будут достаточны для восстановления кластера. Контрольные точки позволяют не хранить WAL-сегменты, которые не нужны для восстановления.

Свойства контрольных точек, которые отражаются в логе кластера:

IS_SHUTDOWN (shutdown) остановка экземпляра в режиме fast или smart

END_OF_RECOVERY (end-of-recovery) вызывается процессом startup в конце восстановления

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

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

WAIT (wait) вернуть управление только после завершения контрольной точки

CAUSE_XLOG (wal) по параметру max_wal_size при переключении WAL сегмента

CAUSE_TIME (time) по времени заданным параметром checkpoint_timeout

FLUSH_ALL (flush-all) сохраняет блоки нежурналируемых объектов, устанавливается при создании базы данных методом FILE_COPY

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

Свойства описаны в функции RequestCheckpoint(int flags) файла checkpointer.c

Шаги выполнения контрольной точки

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

Шаги выполнения контрольной точки

При выполнении контрольной точки выполняются следующие действия.

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

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

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

Алгоритм выполнения действий, связанных с записью грязных блоков буферного кэша описан в функции `BufferSync(int flags)`:

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

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

Шаги выполнения контрольной точки

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

Шаги выполнения контрольной точки (продолжение)

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

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

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

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

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

Формируется журнальная запись, содержащая LSN журнальной записи, которая была сформирована на момент начала контрольной точки. Сформированная журнальная запись с посылается в WAL системным вызовом fdatasync (или другим методом). В pg_control сохраняется LSN сформированной записи об окончании контрольной точки. Контрольная точка завершена.

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

Параметры конфигурации процесса checkpointer

- `log_checkpoints` по умолчанию `on` начиная с 15 версии
- уменьшать значение `checkpoint_completion_target` не рекомендуется
- `checkpoint_timeout` - основной параметр, влияющий на производительность. Оптимальное значение около 20 минут
- `max_wal_size` второй влияющий на производительность при настройке контрольной точки

```
select name, setting, unit, min_val, max_val, context from pg_settings where name
like '%checkpoint%' or name='max_wal_size';
             name              | setting | unit | min_val |  max_val   | context
-------------------------------+---------+------+---------+------------+---------
 checkpoint_completion_target  | 0.9     |      | 0       | 1          | sighup
 checkpoint_flush_after        | 32      | 8kB  | 0       | 256        | sighup
 checkpoint_timeout            | 300     | s    | 30      | 86400      | sighup
 checkpoint_warning            | 30      | s    | 0       | 2147483647 | sighup
 log_checkpoints               | on      |      |         |            | sighup
 max_wal_size                  | 1024    | MB   | 2       | 2147483647 | sighup
(6 rows)
```


Параметры конфигурации процесса checkpointer

Параметры: **\dconfig *checkpoint***

```
           Parameter            | Value
--------------------------------+-------
 checkpoint_completion_target   | 0.9
 checkpoint_flush_after         | 256kB
 checkpoint_timeout             | 5min
 checkpoint_warning             | 30s
 log_checkpoints                | on
 max_wal_size                   | 1GB
```

`log_checkpoints` стал включённым по умолчанию начиная с 15 версии. При выполнении контрольной точки или на репликах точки рестарта в лог кластера выводится запись с деталями выполнения контрольной точки.

Если промежуток времени между контрольными точками будет меньше, чем задано в `checkpoint_warning`, то в лог кластера выводится сообщение:

```
LOG:  checkpoints are occurring too frequently (25 seconds apart)
HINT:  Consider increasing the configuration parameter "max_wal_size".
```

Если сообщения редки, то это означает что в базе выполнялись команды, которые генерировали больше, чем `max_wal_size` журнальных записей за `checkpoint_warning` секунд. Если такие сообщения идут постоянно, то нужно увеличить `max_wal_size`.

Параметр `checkpoint_flush_after` диапазон от 0 (вызов flush отключается) до 2Мб. Ограничивает объём грязных блоков в страничном кэше linux и уменьшает вероятность торможения при выполнении fsync в конце контрольной точки или при фиксации транзакций.

Уменьшать значение `checkpoint_completion_target` не рекомендуется. При увеличении `checkpoint_timeout` стоит пропорционально увеличить, чтобы простой оставался 30 секундным. Например, для 20 минут установить 0.97.

`checkpoint_timeout` - основной параметр, которым настраивается контрольная точка. Оптимальное значение около 20 минут определён опытным ("эмпирически") путем производителями СУБД - это время через которое в новый блок данных перестают вноситься изменения из-за того, что блок заполняется полностью и активность корректировки вставленной строки спадает. Отражает поведение человека: за 20 минут почти всем надоедает корректировать данные в приложении или снимать деньги в банкомате.

Статистика для настройки параметров checkpointer

- серверные процессы при поиске буфера для замены блока на свой не пропускают грязные буфера, они ориентируются только на pin и usagecount
- Кроме checkpointer и серверных процессов сброс грязных блоков выполняет bgwriter
- bgwriter сбрасывает буфера с usagecount=0 и **только у незакрепленных буферов**
- после настройки `checkpoint_timeout` можно настраивать bgwriter: проверять сколько буферов он очистил `pg_stat_bgwriter.buffers_clean` и мог ли очистить:

```
select dirty from pg_buffercache_usage_counts() where pinned=0 and usage_count=0;
 dirty
-------
     0
(1 row)
```


Статистика для настройки параметров checkpointer

Использование значения 20 минут - хорошее приближение, но как оценить значение более точно? Слишком большое значение может увеличить время открытия кластера при некорректной остановке. Это может быть не важным, если в случае падения экземпляра идёт переключение на реплику. Размер директории `PGDATA/pg_wal` может играть роль, но места может быть много.

Контрольная точка сбрасывает грязные блоки, что ускоряет выполнение команд серверными процессами. Если сильно увеличивать интервал контрольных точек, то процент грязных блоков в буферном кэше может увеличиться и серверные процессы по вероятности будут чаще наталкиваться на грязные буфера, разыскивая место под блоки. Серверные процессы при поиске буферов для замены блоков не пропускают грязные буфера, они ориентируются только на pin и usagecount. Частный случай: кластер целиком помещается в буферный кэш. В этом случае серверные процессы не будут искать место в буферном кэше. Но обычно размер кластера больше, чем буферного кэша. Кроме checkpointer и серверных процессов сброс грязных блоков выполняет bgwriter. Если bgwriter сбросил буфер, то этот буфер не сбрасывается по контрольной точке, так как оба процесса сбрасывают только грязные буфера. Однако, bgwriter сбрасывает буфера с usagecount=0, а usagecount уменьшается на единицу (`-= BUF_USAGECOUNT_ONE;`) только процессами, которые ищут свободные блоки (серверные, автовакуум) вызовом функции `StrategyGetBuffer(..)` и **только у незакрепленных буферов**. То есть bgwriter не достаточен.

Из-за `full_page_writes=on` (отключать не стоит) изменение блока после контрольной точки вызывает однократное сохранение образа блока, за исключением неиспользуемого в блоке места (pd_upper минус pd_lower). Блок сохраняется не сразу после изменения, а в процессе обхода буферов процессом checkpointer. Можно предположить, что более частые контрольные точки более часто будут писать целиком блоки. Однако, если блок закончил грязниться, в том числе автовакуумом (до автовакуума возможно SELECT, который установит биты infomask о фиксации транзакции), который очистит блок от старых версий строк, то блок вряд ли будет меняться и потребует записи. Можно оценить сколько времени удерживается горизонт баз данных плюс длительность цикла автовакуума и ориентироваться на это значение для установки `checkpoint_timeout`.

После настройки `checkpoint_timeout` можно настраивать bgwriter: проверять сколько буферов он очистил `pg_stat_bgwriter.buffers_clean` и мог ли очистить `select dirty from pg_buffercache_usage_counts() where pinned=0 and usage_count=0;`

Пример настройки параметров checkpointer

- `select pg_sleep(:t3);` удерживает горизонт базы, что более приближено к реальной нагрузке
- "треугольник" в dirty:

```
select * from pg_buffercache_usage_counts();
 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
           0 |    6957 |     0 |      0
           1 |      88 |    13 |      0
           2 |      48 |    23 |      0
           3 |      62 |    24 |      0
           4 |      81 |    61 |      0
           5 |    9148 |  7895 |      0
```

```
alter system set checkpoint_timeout = :t3;
select pg_reload_conf();
select pg_current_wal_lsn() AS t1 \gset
select pg_sleep(:t3);
select pg_current_wal_lsn() AS t2 \gset
select pg_size_pretty((:'t2'::pg_lsn - :'t1'::pg_lsn)/:'t3');
```

```
#!/bin/bash
for ((i=300;i<=7200;i+=300))
do
export t3="$i"
psql -f ckpt.sql
done
60  494kB
120 360kB
180 254kB
240 199kB
300 165kB
360 143kB
420 128kB
540 133kB
600  81kB
660  98kB
780  85kB
1200 60kB
1500 53kB
1680 56kB
1800 51kB
2400 45kB
```


Пример настройки параметров checkpointer

Командный файл для измерения объема, записываемого в WAL при изменении параметра `checkpoint_timeout`:

```
#!/bin/bash
for (( i=60; i <= 2400; i+=60 ))
do
export t3="$i"
psql -f ckpt.sql
done
```

Скрипт с названием **`ckpt.sql`**:

```
\pset tuples_only
\getenv t3 t3
\echo :t3
\o ckpt.tmp
alter system set checkpoint_timeout = :t3;
select pg_reload_conf();
select pg_current_wal_lsn() AS t1 \gset
select pg_sleep(:t3);
select pg_current_wal_lsn() AS t2 \gset
\o
select pg_size_pretty((:'t2'::pg_lsn - :'t1'::pg_lsn)/:'t3');
```

если нагрузки на кластер нет, можно создать тестовую нагрузку запустив в отдельном окне генерацию нагрузки:

```
pgbench -T 60000 -P 10
pgbench (16.2)
starting vacuum...end.
progress: 10.0 s, 529.6 tps, lat 1.879 ms stddev 1.984, 0 failed
...
```

Объем WAL будет зависеть от TPS и объема полных блоков, записываемых в WAL после контрольной точки. Если объем останется одинаков при изменении `checkpoint_timeout`, это означает, что один и тот же блок после завершения в нем изменений будет записан один раз. Измерения нужно выполнять после настройки автовакуума. Нужно ли измерять объем записи в файлы данных в PGDATA? Не обязательно, так как объем, записываемый в WAL коррелирует с записью в файлы данных.

`select pg_sleep(:t3);` удерживает горизонт базы, что более приближено к реальной нагрузке. В тесте pgbench по умолчанию транзакции короткие, отрабатывает HOT cleanup.

Пример настройки параметров checkpointer

- `\! sleep $t3` не удерживает горизонт, TPS увеличивается с ~44 до ~620

```
progress:    600.0 s, 618.6 tps, lat 1.609 ms stddev 1.036
progress:    900.0 s, 629.0 tps, lat 1.582 ms stddev 0.914
progress:   1800.0 s, 627.7 tps, lat 1.585 ms stddev 0.858
progress:   3300.0 s, 615.7 tps, lat 1.616 ms stddev 0.912
progress: 10500.0 s, 618.4 tps, lat 1.609 ms stddev 0.882
progress: 12900.0 s, 618.5 tps, lat 1.609 ms stddev 0.842
```

```
alter system set checkpoint_timeout = :t3;
select pg_reload_conf();
select pg_current_wal_lsn() AS t1 \gset
\! sleep $t3
select pg_current_wal_lsn() AS t2 \gset
select pg_size_pretty((:'t2'::pg_lsn - :'t1'::pg_lsn)/:'t3');
```

```
#!/bin/bash
for ((i=30;i<=600;i+=30))
do
export t3="$i"
psql -f ckpt.sql
done
30  689kB
60  567kB
90  452kB
120 393kB
150 368kB
180 340kB
210 332kB
240 408kB
270 332kB
300 310kB
330 307kB
360 300kB
390 300kB
420 295kB
450 292kB
480 291kB
```


Пример настройки параметров checkpointer (продолжение)

Если использовать `\! sleep $t3`, то TPS 40 увеличивается до 600 и частота контрольных точек может быть любой. Если не использовать pgbench или создать для него реальный тест, то стоит использовать `\! sleep $t3`

Значения записанных в WAL килобайтов в секунду зависят от объема изменяемых данных (TPS), для разной нагрузки значения несравнимы.

Почему значение по умолчанию для `checkpoint_timeout=300` (5 минут)? Это значение подходит для нагрузки, создаваемой тестом pgbench по умолчанию.

Тест по умолчанию ("TPC-B") состоит из трех UPDATE, одного INSERT и одного SELECT в одной транзакции. В документации к pgbench написано: "It is very easy to use pgbench to produce completely meaningless numbers".

Практика

- Часть 1. Настройка частоты контрольных точек
- Часть 2. Задержка при запуске экземпляра. Параметр `recovery_init_sync_method`
- Часть 3. Длительность контрольной точки
- Часть 4. Длительность финальной контрольной при остановке экземпляра
- Часть 5. Длительность контрольной точки после падения экземпляра
- Часть 6. Контрольная точка по запросу

Практика

Вы рассмотрите пример как при наличии 30тыс. файлов в кластере синхронизация файлов занимает больше минуты.

Вы научитесь

читать сообщения о контрольных точках в диагностическом журнале кластера;

рассмотрите как выполняются разные типы контрольных точек.

12

Автовакуум

Алгоритм вакуумирования

- каждая таблица вакуумируется в отдельной транзакции
- до конца транзакции удерживается моментальный снимок
- если блокировка на таблицу не может быть установлена, то таблица пропускается
- по TOAST-таблицам статистика не собирается
- индексы одной таблицы могут очищаться параллельными рабочими процессами, они могут использовать HugePages для списка TID
- временные объекты очищаются последовательно
- пять фаз вакуумирования каждой таблицы:
  - `SCAN_HEAP, VACUUM_INDEX, VACUUM_HEAP, INDEX_CLEANUP, VACUUM TRUNCATE`

Алгоритм вакуумирования

Сначала строится список таблиц, которые будут очищаться. Эта подготовительная фаза цикла автовакуума называется инициализацией (`initializing`).

Перед обработкой **каждой** таблицы (в том числе TOAST) открывается транзакция и создается моментальный снимок. Это позволяет быстрее снимать блокировки и сдвигаться горизонту базы данных.

На таблицу устанавливается блокировка ShareUpdateExclusive. Если блокировка не может быть установлена, то транзакция завершается и освобождаются установленные блокировки, иначе блокировка отдельным вызовом распространяется на сессию вакуума (LockRelationIdForSession(..)), чтобы вакуумировать TOAST-таблицу в отдельной транзакции и не ждать блокировку.

TOAST таблицы не будут анализироваться, так как доступ к строкам TOAST всегда идёт по TOAST-индексу и статистика на TOAST бесполезна. Сессия переключается на работу под владельцем таблицы, чтобы функции в индексе выполнялись из под владельца таблицы, так как параметры конфигурации могут устанавливаться на роль-владельца и влиять на результат выполнения функций.

Очистка индексов **может использовать параллельные процессы**, но один индекс вакуумирует только **один** рабочий процесс. Таблица (секции таблицы) всегда сканируется одним процессом, в котором выполняется вакуум. Если планируется (по каждой таблице отдельно) параллельная очистка индексов, то под хранение идентификаторов строк (TID, каждый по 6 байт) которые уже помечены в блоках битом-подсказкой LP_DEAD (это делает HOT) и будут помечены вакуумом на первой фазе его выполнения, выделяется разделяемая память (dynamic shared memory). Если не планируется, то локальная память процесса, который вакуумирует таблицу. Использование разделяемой памяти не влияет на производительность, так как блокировки не нужны: в нее пишет только основной вакуумирующий процесс. Разделяемая область памяти **может использовать HugePages.**

Временные объекты не обрабатываются автовакуумом, но могут обрабатываться серверным процессом по командам ANALYZE И VACUUM. Временные таблицы (и индексы на них) очищаются последовательно и не могут очищаться параллельно, так как доступ к временным таблицам имеет только серверный процесс.

Есть пять фаз вакуумирования каждой таблицы, mwiew, toast и индексов на них:
SCAN_HEAP, VACUUM_INDEX, VACUUM_HEAP, INDEX_CLEANUP, VACUUM TRUNCATE.
Помимо них есть подготовительная фаза инициализации и завершающая фаза.

Первая фаза вакуумирования

- под накопление TID мертвых строк (LP_DEAD) выделяется память в размере `autovacuum_work_mem`
- в 17 версии если на таблице нет ни одного индекса место в блоках очищается за один проход, а не за два прохода, каждый блок меняется один раз, а не два раза и генерируется меньше журнальных записей
- на первой фазе читаются блоки, в них ставятся биты LP_DEAD и TID всех строк с этим флагом сохраняются в памяти
- до 17 версии память под TID ограничена 1Гб, с 17 версии может использоваться вся `autovacuum_work_mem` и памяти используется в ~20 раз меньше
- если блоков с LP_DEAD будет меньше 2% от всех блоков таблицы и памяти под TID будет использовано меньше 32Мб вакуум заканчивает обработку таблицы и индексы не вакуумируются

Первая фаза вакуумирования

Первая фаза: под накопление TID мертвых строк (LP_DEAD) выделяется память в размере `autovacuum_work_mem` (если он установлен в -1, то `maintenance_work_mem`).

Сканируются блоки таблицы в которых могут быть мертвые строки (блоки менялись с предыдущего вакуума, проверяется по карте видимости), если только не установлена опция VACUUM (DISABLE_PAGE_SKIPPING). У автовакуума нет такого параметра. Строки, вышедшие за горизонт базы помечаются как мертвые и их TID (вместе с уже помеченными в блоке ранее HOT) сохраняются в памяти.

Если памяти не хватит, первая фаза приостанавливается, идет переход на вторую фазу, потом возврат на первую фазу и на первой фазе продолжают сканироваться блоки дальше.

При этом вторая фаза будет выполняться полностью - сканировать все индексы ещё раз. Поэтому **стоит устанавливать объем памяти для вакуума или настраивать частоту вакуумирования так, чтобы вакуум выполнялся в один проход**. До 17 версии если `maintenance_work_mem` или `autovacuum_work_mem` были больше 1Гб, то под хранение TID использовался 1Гб, TID хранились в виде списка и поиск был неэффективным. Начиная с 17 версии используется префиксное дерево (radix tree) с компактным хранением (path compression). Объем памяти для хранения TID уменьшился в ~20 раз.

По умолчанию вакуумирование использует опцию INDEX_CLEANUP AUTO. Это означает, что если блоков, в которых есть хоть одна dead строка наберется меньше 2% (`BYPASS_THRESHOLD_PAGES=0.02`) от всех блоков в таблице и одновременно память под TID (6 байт) мертвых строк будет меньше, чем 32Мб (2^25/6=5592400 строк **до 17** версии), то остальные фазы не выполняются и индексы не сканируются. Второе условие нужно для больших таблиц. Условие в 17 версии:

```
TidStoreMemoryUsage(vacrel->dead_items) < (32L * 1024L * 1024L))
```

Без этого достаточно бы было в таблице обновить или удалить строку и сканировались бы все индексы на таблице, что долго и трудоемко, а результат был бы никакой. Карты видимости (и заморозки) у индексов нет, индексные записи не замораживаются и не имеют xmin, xmax. Небольшое число мертвых строк получается, когда активно работает HOT cleanup (нужно стремиться, чтобы в основном он и освобождал место). Если **INDEX_CLEANUP OFF или на уровне таблицы установлен параметр VACUUM_INDEX_CLEANUP OFF, то индексы не сканируются.** Это имеет смысл, если нужно как можно быстрее просканировать блоки таблицы и пометить строки как LP_DEAD, чтобы избежать переполнения счетчика транзакций.Отключается: VACUUM (INDEX_CLEANUP ON).

Расчёт памяти под TID для вакуумирования

- под накопление TID мертвых строк (LP_DEAD) выделяется память в размере
  - `autovacuum_work_mem` для автовакуума
  - `maintenance_work_mem` для команды VACUUM
  - по умолчанию 64Гб
- память под хранение идентификаторов строк до 17 версии расчитывается по формуле: `maintenance_work_mem = n_dead_tup * 6`. Но не больше 1 гигабайта.
- Пример: 1000000*6/1Мб=5.74 = 6 проходов

```
select schemaname, relname, n_dead_tup, n_live_tup from pg_stat_user_tables where
relname = 'test';
 schemaname | relname | n_dead_tup | n_live_tup
------------+---------+------------+------------
 public     | test    |    1000000 |   10000000
set maintenance_work_mem='1MB';
vacuum verbose test;
INFO:  vacuuming "postgres.public.test"
INFO:  finished vacuuming "postgres.public.test": index scans: 6
```


Расчёт памяти под TID для вакуумирования

Первая фаза: под накопление TID мертвых строк (LP_DEAD) выделяется память в размере `autovacuum_work_mem` (если он установлен в -1, то `maintenance_work_mem`).

Число мертвых строк определяет верхнюю границу, то есть максимум сколько может быть удалено строк при вакуумировании. Это число заполняется всеми процессами обновлении строк таблицы, не требует анализа и актуально:

```
 select schemaname, relname, n_dead_tup, n_live_tup from pg_stat_user_tables
where relname = 'test';
  schemaname | relname | n_dead_tup | n_live_tup
------------+---------+------------+------------
  public     | test    |    1000000 |   10000000
```

Память под хранение идентификаторов строк до 17 версии расчитывается по формуле: `maintenance_work_mem = n_dead_tup * 6`. Но не больше 1 гигабайта.

В примере: maintenance_work_mem = 1000000*6=6000000 байт.

Если установить:

```
set maintenance_work_mem='1MB';
```

то будет 6000000/(1024*1024)=5.74 что даёт 6 проходов. Округление в большую сторону, так как проходов целое число. Пример:

```
vacuum verbose test;
INFO:  vacuuming "postgres.public.test"
INFO:  finished vacuuming "postgres.public.test": index scans: 6
pages: 0 removed, 48712 remain, 48712 scanned (100.00% of total)
tuples: 1000000 removed, 10000000 remain, 0 are dead but not yet removable, oldest xmin: 22113886
removable cutoff: 22113886, which was 0 XIDs old when operation ended
frozen: 4427 pages from table (9.09% of total) had 1000002 tuples frozen
index scan needed: 4425 pages from table (9.08% of total) had 1000000 dead item identifiers removed
index "test_id_idx": pages: 30163 in total, 0 newly deleted, 0 currently deleted, 0 reusable
I/O timings: read: 1185.969 ms, write: 317.146 ms
avg read rate: 79.722 MB/s, avg write rate: 20.861 MB/s
buffer usage: 96868 hits, 186041 misses, 48681 dirtied
WAL usage: 76277 records, 54147 full page images, 429319587 bytes
system usage: CPU: user: 3.41 s, system: 3.18 s, elapsed: 18.23 s
```

В 17 версии память кореллирует с n_dead_tup, но точной зависимости нет. Требуемый объем памяти меньше в ~20 раз.

Вторая и третья фазы вакуумирования

- на второй фазе в индексах очищаются ссылки на TID, сохраненные в памяти на первой фазе
- на третьей фазе в блоках таблиц очищаются TID с LP_DEAD, собранные в памяти на первой фазе: слоты в заголовке блоков становятся пригодными для использования
- замораживаются строки, которые можно заморозить
- обновляются карты свободного места, видимости, заморозки
- для очистки описатель буфера блока блокируется в режиме Exclusive
  - Если получить Exclusive блокировку на описатель буфера не удастся, то блок TID с LP_DEAD не очищается
  - процесс будет пытаться получить блокировку и блок будет очищен, если команда VACUUM использует опции SKIP_LOCKED false или FREEZE или DISABLE_PAGE_SKIPPING, а также если цикл автовакуума запустился в агрессивном режиме (для заморозки строк)

Вторая и третья фазы вакуумирования

Начиная с 17 версии PostgreSQL, **если на таблицу нет индексов** опции игнорируются, второй фазы нет, первая фаза совмещается с третьей: вместо установки LP_DEAD место которое занимает строка (вместе с маской где ставится бит LP_DEAD) освобождается, а указатель в заголовке блока становится UNUSED, благодаря этому **объем записи в WAL может уменьшиться** (https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=c120550ed).

На второй фазе в индексах очищаются ссылки на TID, сохраненные в памяти на первой фазе.

На третьей фазе в блоках таблиц очищаются TID с LP_DEAD, собранные в памяти на первой фазе: слоты в заголовке блоков становятся пригодными для использования (UNUSED). Если очистятся слоты в конце заголовка блока, то место может стать свободным и заголовок блока уменьшиться (truncate the line pointer array).

Для очистки описатель буфера блока блокируется в режиме Exclusive. Блоки в которых на первой фазе не было обнаружено LP_DEAD не читаются.

Замораживаются строки, которые можно заморозить, fsm и vm обновляются или создаются если их не было. В комментариях к функции lazy_scan_heap() файла vacuumlazy.c написано: prunes each page in the heap, and considers the need to freeze remaining tuples with storage (not including pages that can be skipped using the visibility map). Also performs related maintenance of the FSM and visibility map.

Если получить Exclusive блокировку на описатель буфера не удастся, то по умолчанию устанавливается блокировка Share, блок не очищается, а читается.

Если вакуум был запущен в агрессивном режиме ("aggressive mode"), то вакуум будет ждать получения Exclusive на описатель буфера.

Агрессивным режимом команды VACUUM называется использование параметра SKIP_LOCKED FALSE или FREEZE или DISABLE_PAGE_SKIPPING.

Цель агрессивного режима - обработать все блоки с целью заморозки. SKIP_LOCKED FALSE предназначен для случаев когда картам видимости и заморозки нельзя доверять, есть подозрение на их повреждение, он включает в себя FREEZE только по всем блокам.

Смысл пропуска блоков в том, чтобы ускорить вакуумирование таблицы, иначе автовакуум может не успеть обработать все запланированные таблицы. Также если описатель блока заблокирован, то есть вероятность того, что содержимое блока всё ещё меняется.

Четвертая и пятая фазы вакуумирования

- фаза усечения файла (файлов) таблицы вызывается если объем пустых блоков в конце последнего файла основного слоя больше 8Мб
  - запрашивается монопольная блокировка на таблицу
  - если блокировка не получена в течение 5 секунд, фаза пропускается
  - фазу можно отключить на уровне таблицы и уровне ее TOAST-таблицы
- анализ выполняется отдельно от вакуума

Четвертая и пятая фазы вакуумирования

На четвертой фазе вызывается документированная (https://docs.tantorlabs.ru/tdb/ru/16_4/se/index-functions.html) функция индексного метода доступа amvacuumcleanup(..), которая может выполнить какие-то полезные действия в зависимости от типа индекса. Например, освободить пустые страницы индекса. Выходное значение функции - статистика по индексу, которая используется для вывода VERBOSE или в диагностический лог. Эта функция также вызывается в конце фазы ANALYZE.

Пятая фаза вызывается только, если объем пустых блоков в конце последнего файла основного слоя таблицы **больше 8Мб** (макрос REL_TRUNCATE_MINIMUM). Выполняется функцией `lazy_truncate_heap(..)`, которая пытается получить монопольную блокировку на таблицу, но **если не сможет получить блокировку за VACUUM_TRUNCATE_LOCK_TIMEOUT=5 секунд, то усечение не выполняется.** Пятая фаза может отключаться на уровне таблицы и отдельно для TOAST-таблицы параметрами `VACUUM_TRUNCATE` и `TOAST.VACUUM_TRUNCATE`, а также для команды `VACUUM (TRUNCATE false)`. Установка параметра конфигурации `old_snapshot_threshold` (убран в 17 версии) отключает пятую фазу.

Финальные действия: параметры сессии, если менялись в теле функций индексов то они восстанавливаются. Сессия возвращается в исходного пользователя. Транзакция фиксируется. С основной таблицы или материализованного представления снимается блокировка.
Если обрабатывалась TOAST-таблица, то блокировка с основной таблицы не снимается и выполняется обработка TOAST с открытием и фиксацией транзакции, после чего снимается блокировка с основной таблицы.
В финальной фазе обновляется `pg_database.datfrozenxid` и по возможности усекается pg_xact. Это можно отключить параметром SKIP_DATABASE_STATS false.**Анализ выполняется отдельно от вакуума**, даже если совместить в одной команде VACUUM (ANALYZE) анализ выполнится после вакуума.

Агрессивный (Aggressive) режим вакуумирования

- если обычный автовакуум сможет продвигать `pg_class.relfrozenxid` и `pg_class.relminmxid`, то агрессивный автовакуум не понадобится
- агрессивный режим автовакуума запускается, если `age(pg_class.relfrozenxid) > vacuum_freeze_table_age - vacuum_freeze_min_age` или `mxid_age(pg_class.relminmxid) > vacuum_multixact_freeze_table_age - vacuum_freeze_min_age`
- `age(pg_database.datfrozenxid) > autovacuum_freeze_max_age` или `mxid_age(pg_database.datminmxid) > autovacuum_multixact_freeze_max_age`

```
\dconfig *freeze*
               Parameter                |    Value
----------------------------------------+-------------
 autovacuum_freeze_max_age              | 10000000000
 autovacuum_multixact_freeze_max_age    | 20000000000
 vacuum_freeze_min_age                  |    50000000
 vacuum_freeze_table_age                |   150000000
 vacuum_multixact_freeze_min_age        |     5000000
 vacuum_multixact_freeze_table_age      |   150000000
```

```
SELECT datname, age(datfrozenxid),
mxid_age(datminmxid) FROM
pg_database;
   datname  |    age     | mxid_age
------------+------------+---------
 postgres   | 121346629 |      19
 template1  | 121346629 |      19
 template0  | 121346629 |      19
```


Агрессивный (Aggressive) режим вакуумирования

Если обычный автовакуум сможет продвигать `pg_class.relfrozenxid` и `pg_class.relminmxid`, то агрессивный автовакуум не понадобится. Продвигать сможет, если сможет получить блокировки на описатели блоков, которые захочет очистить.

Агрессивный режим вызывается опциями SKIP_LOCKED FALSE, FREEZE, DISABLE_PAGE_SKIPPING команды VACUUM. Все три режима замораживают строки. У автовакуума опций нет, но тоже есть агрессивный режим.

Агрессивный режим автовакуума запускается, если `age(pg_class.relfrozenxid) > vacuum_freeze_table_age - vacuum_freeze_min_age` **или если** `mxid_age(pg_class.relminmxid) > vacuum_multixact_freeze_table_age - vacuum_freeze_min_age`.

Слишком большое значение параметра `vacuum_freeze_min_age` устанавливать не стоит, так как запуск автовакуума для заморозки будет частым.

Эти параметры с созвучными названиями **можно устанавливать на уровне таблиц.**

Агрессивный режим запускается по базе данных даже если автовакуум отключен на кластере, если `age(pg_database.datfrozenxid) > autovacuum_freeze_max_age` или `mxid_age(pg_database.datminmxid) > autovacuum_multixact_freeze_max_age`.

Мониторинг:

```
SELECT datname, age(datfrozenxid), mxid_age(datminmxid) FROM pg_database;
  datname  |    age     | mxid_age
-----------+------------+---------
 postgres  | 121346629 |      19
 template1 | 121346629 |      19
 template0 | 121346629 |      19
```

Выдается насколько далеко xid транзакций и мультитранзакций находится в прошлом относительно текущей транзакции. **На 32битных XID по умолчанию `autovacuum_freeze_max_age=200000000`, значит результат не должен приближаться к 200млн. Если приближается к 2млрд., то нужно искать таблицы с большими `age(pg_class.relfrozenxid)` и замораживать их.**

Если значения в `pg_class` не смогли измениться, то при `age(..)>vacuum_failsafe_age` автовакуум запускается в специальном режиме: задержки **`autovacuum_vacuum_cost_delay`** не применяются, **автовакуум не ограничен буферным кольцом и использует все блоки буферного кэша**, индексы не обрабатываются (аналог INDEX_CLEANUP off).

VACUUM FULL автоматически замораживает все строки.

Заморозка строк (FREEZE)

- можно и вакуумировать и замораживать TOAST отдельно:

```
select relname, relfrozenxid, age(relfrozenxid), relminmxid, mxid_age(relminmxid)
from pg_class  where relfrozenxid<>0 order by 3 desc;

vacuum (freeze, verbose) pg_toast.pg_toast_1262;
INFO:  finished vacuuming "postgres.pg_toast.pg_toast_1262": index scans: 0
tuples: 0 removed, 0 remain, 0 are dead but not yet removable, oldest xmin: 105637454
removable cutoff: 105637454, which was 3 XIDs old when operation ended
new relfrozenxid: 105637454, which is 105636725 XIDs ahead of previous value
new relminmxid: 37, which is 36 MXIDs ahead of previous value
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
```

- Обычный вакуум по возможности (если обработает все блоки которые не обрабатывал раньше и в сумме окажутся замороженными все строки во всех блоках) выполняет заморозку:

```
relname | relfrozenxid | relminmxid
---------+--------------+------------
 t3      |        67551 |         37
vacuum (verbose) t3;
 relname | relfrozenxid | relminmxid
---------+--------------+------------
 t3      |    105738379 |         37
```


Заморозка строк (FREEZE)

Отключать автовакуум во время большой нагрузки и запускать вакуум во время пониженной нагрузки не стоит. Выполнять VACUUM (FREEZE) вручную можно на кластерах с 32-битным счетчиком транзакций при приближении момента вызова автовакуума в этом режиме, чтобы избежать выполнения во время большой нагрузки. В этом режиме вакуумирующий процесс ждет получения монопольной блокировки не только на блоки, которые давно не замораживались, но и на активно используемые блоки, так как в таких блоках может оказаться строка со старым XID.

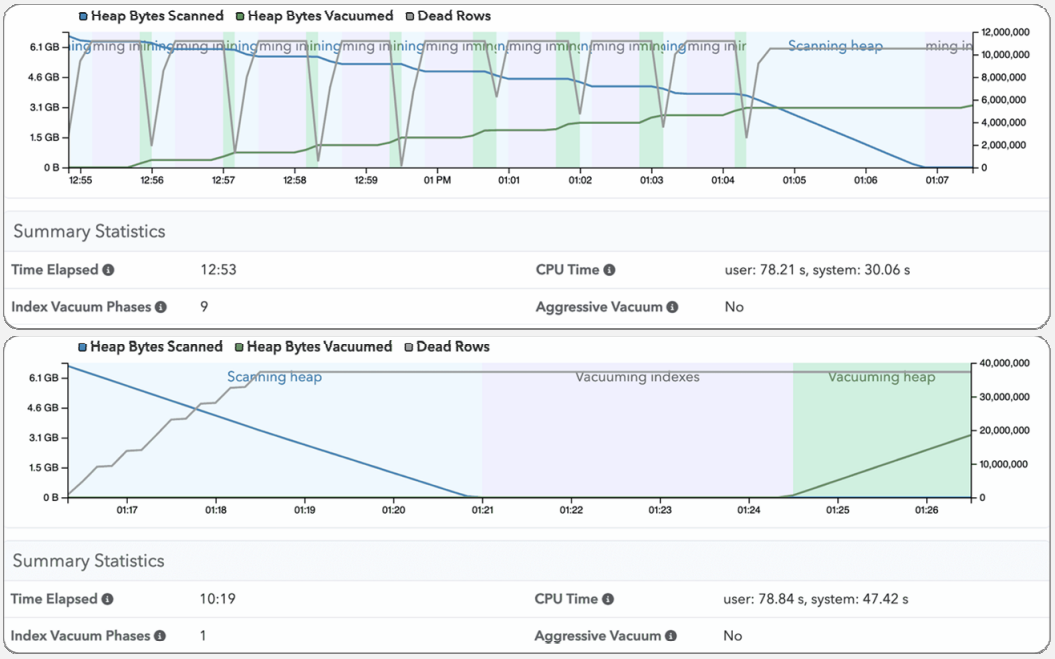
При загрузке строк в таблицу командой COPY **имеет смысл использовать опцию FREEZE**. Если ее не использовать, то первое вакуумирование скорее пошлет в WAL полные образы блоков, если между COPY и вакуумом начнется контрольная точка. На оговорку в документации про "нарушения правил видимости" и "потенциальные проблемы" можно не обращать внимание. Строки в TOAST у COPY WITH FREEZE в текущей версии PostgreSQL не замораживаются.

Пример вакуумирования с заморозкой TOAST-таблицы:

```
 select relname, relfrozenxid, age(relfrozenxid), relminmxid,
mxid_age(relminmxid) from pg_class where relfrozenxid<>0 and
relname='pg_toast_3394';
     relname     | relfrozenxid |    age    | relminmxid | mxid_age
-----------------+--------------+-----------+------------+---------
 pg_toast_3394 |          723 | 140062561 |          1 |      19
 vacuum (freeze, verbose) pg_toast.pg_toast_3394;
 INFO:  aggressively vacuuming "postgres.pg_toast.pg_toast_3394"
 tuples: 0 removed, 0 remain, 0 are dead but not yet removable, oldest xmin: 140063284
 removable cutoff: 140063284, which was 0 XIDs old when operation ended
 new relfrozenxid: 140063284, which is 140062561 XIDs ahead of previous value
 new relminmxid: 37, which is 36 MXIDs ahead of previous value
 frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
 buffer usage: 22 hits, 1 misses, 1 dirtied
 WAL usage: 1 records, 1 full page images, 8179 bytes
 system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
 select relname, relfrozenxid, age(relfrozenxid), relminmxid,
mxid_age(relminmxid) from pg_class  where relfrozenxid<>0 and
relname='pg_toast_3394';
     relname     | relfrozenxid | age | relminmxid | mxid_age
-----------------+--------------+-----+------------+----------
 pg_toast_3394 |    140063284 |   0 |         37 |        0
```

Вакуум в 17 версии PostgreSQL

- использует меньше памяти



Вакуум в 17 версии PostgreSQL

В 17 версии TID строк LP_DEAD более эффективное как по использованию памяти, так и при поиске TID при сканировании индексов на второй фазе. Если памяти не хватит, то используется несколько проходов, а это резкая деградация производительности. При очистке блоков тестовой таблицы размером 3.4Гб и настройках экземпляра по умолчанию maintenance_work_mem=64MB:

```
CREATE TABLE test AS SELECT * FROM generate_series(1, 100000000) x(id);
CREATE INDEX ON test(id);
UPDATE test SET id = id - 1;
```

вакуум до 17 версии выполнит очистку в 9 проходов (по 64Мб памяти) и за 770 секунд:

```
LOG:  automatic vacuum of table "postgres.public.test": index scans: 9
WAL usage: 2316372 records, 1189127 full page images, 2689898432 bytes
system usage: CPU: user: 78.21 s, system: 30.06 s, elapsed: 773.23 s
```

в 17 версии за 1 проход, 37Мб и 620 секунд (на 20% быстрее):

```
WAL usage: 2316363 records, 1431435 full page images, 2586981769 bytes
system usage: CPU: user: 78.84 s, system: 47.42 s, elapsed: 619.04 s
```

Время выполнения вакуума - это время удержания горизонта базы данных.

При увеличении памяти в 16 версии, можно было бы получить 1 проход и меньшее время. После "настройки контрольных точек", то есть увеличения `checkpoint_timeout` и `max_wal_size` количество full page images станет одинаковым. **Анализировать журнальную активность вакуумирования стоит только после настройки частоты контрольных точек.**

Количество проходов указывается в столбце **index_vacuum_count** представления pg_stat_progress_vacuum:

```
select * from pg_stat_progress_vacuum;
-[ RECORD 1 ]------+--------------
pid                | 559
datid              | 5
datname            | postgres
relid              | 16422
phase              | vacuuming heap
heap_blks_total    | 884956
heap_blks_scanned  | 884956
heap_blks_vacuumed | 441819
index_vacuum_count | 9
max_dead_tuples    | 11184809
num_dead_tuples    | 10522080
```

https://pganalyze.com/blog/5mins-postgres-17-faster-vacuum-adaptive-radix-trees

Сравнительное тестирование вакуума 16 и 17 версий PostgreSQL

- использует меньше памяти в ~20 раз
- память на создание radix tree ограничивается `autovacuum_work_mem` или `maintenance_work_mem`
- использование WAL одинаково в обеих версиях.
- использование процессора на построение и сканирование radix tree (вместо массива) и длительность вакуумирования в 17 версии больше на ~20%
- повышения нагрузки на ввод-вывод нет, только на процессор
  - на avg read/write rate в статистике вакуума влияет CPU usage
- при включенных `(data_checksums=on OR wal_log_hints=on) AND (full_page_writes=on)` объем журнальных записей увеличивается по причине того, что если блок меняется, то один раз после контрольной точки в журнал записывается образ этого блока (неиспользуемое место не записывается)
  - чем реже контрольные точки, тем меньше вероятность многократной записи полного образа блока

Сравнительное тестирование вакуума 16 и 17 версий PostgreSQL

Пример теста:

```
alter system set max_wal_size = '8GB';
alter system set checkpoint_timeout='30min';
select pg_reload_conf();
drop table test;
CREATE TABLE test with (autovacuum_enabled=off) AS SELECT * FROM
generate_series(1, 10000000) x(id);
CREATE INDEX ON test(id);
UPDATE test SET id = id - 1;
checkpoint;
vacuum verbose test;
```

Сравнение 16 и 17 версии при 1 проходе по индексам:

17 версия:

```
INFO:  finished vacuuming "postgres.public.test": index scans: 1
pages: 0 removed, 88535 remain, 88535 scanned (100.00% of total)
tuples: 10000000 removed, 10000000 remain, 0 are dead but not yet removable
removable cutoff: 752, which was 0 XIDs old when operation ended
new relfrozenxid: 752, which is 3 XIDs ahead of previous value
frozen: 44249 pages from table (49.98% of total) had 10000000 tuples frozen
index scan needed: 44248 pages from table (49.98% of total) had 10000000 dead item identifiers removed
index "test_id_idx": pages: 54840 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 74.961 MB/s, avg write rate: 81.952 MB/s
buffer usage: 106583 hits, 169651 misses, 185473 dirtied
WAL usage: 364397 records, 143159 full page images, 1005717614 bytes
system usage: CPU: user: 10.84 s, system: 1.00 s, elapsed: 17.68 s
```

16 версия:

```
avg read rate: 90.693 MB/s, avg write rate: 99.367 MB/s
WAL usage: 364398 records, 143159 full page images, 1005540671 bytes
system usage: CPU: user: 10.77 s, system: 1.78 s, elapsed: 14.57 s
```

Сравнение 16 и 17 версии при 1 проходе по индексам без контрольных сумм и без `wal_log_hints`:

```
buffer usage: 107081 hits, 169152 misses, 185287 dirtied
WAL usage: 231640 records, 98897 full page images, 277373985 bytes
```

16 версия: `system usage: CPU: user: 10.01 s, system: 1.03 s, elapsed: 12.29 s`

17 версия: `system usage: CPU: user: 10.18 s, system: 0.70 s, elapsed: 14.72 s`

Без контрольных сумм объем журнальных записей **ровно** в 3 раза меньше.

с контрольными суммами с чтением таблицы командой перед checkpoint: **`explain (analyze) select * from test;`**

```
buffer usage: 121140 hits, 155093 misses, 143159 dirtied
WAL usage: 187405 records, 143159 full page images, 576514722 bytes
```

16 версия:

```
avg read rate: 108.505 MB/s, avg write rate: 100.156 MB/s
system usage: CPU: user: 8.81 s, system: 1.03 s, elapsed: 11.16 s
```

17 версия:

```
avg read rate: 81.625 MB/s, avg write rate: 75.001 MB/s
system usage: CPU: user: 10.50 s, system: 0.66 s, elapsed: 14.91 s
```

без контрольных суммам с чтением таблицы (для чтения используется explain) перед checkpoint:

```
buffer usage: 120474 hits, 155760 misses, 143159 dirtied
WAL usage: 187391 records, 98897 full page images, 215117884 bytes
```

16 версия:

```
avg read rate: 118.284 MB/s, avg write rate: 109.183 MB/s
system usage: CPU: user: 8.62 s, system: 0.74 s, elapsed: 10.24 s
```

17 версия:

```
avg read rate: 93.486 MB/s, avg write rate: 85.923 MB/s
system usage: CPU: user: 10.24 s, system: 0.50 s, elapsed: 13.01 s
```

Без контрольных сумм объем журнальных записей **ровно** в 2 раза меньше.

Выводы:

1) Использование WAL одинаково в обоих версиях.

2) Использование процессора на построение и сканирование radix tree вместо массива и длительность вакуумирования в 17 версии увеличилось на **~20%**. `avg read/write rate` кореллирует с `CPU usage`.

3) При включенных `(data_checksums=on OR wal_log_hints=on) AND (full_page_writes=on)`

объем журнальных записей увеличивается за счет того, что при изменении бита-подсказки или контрольной суммы (если меняется бит, то меняется и контрольная сумма) меняется блок. Если блок меняется, то **один раз** после контрольной точки в журнал записывается образ этого блока (кроме неиспользуемого места). Чем реже контрольные точки, тем меньше вероятность многократной записи полного образа блока.

В приведенном примере таблица создавалась, менялась и вакуумировалась. При реальной работе блоки перестают меняться через какое-то время и контрольные суммы с full page writes влияют на объем журналов не так сильно, как в искусственных тестах. Параметры `data_checksums=on` и `full_page_writes=on` не стоит отключать.

При включенных контрольных суммах и full page writes **повторные** изменения блоков после контрольной точки генерируют **100 мегабайт**, а не 1 гигабайт и **31** полный образ, а не `143159 full page images`:

```
checkpoint; vacuum verbose test;
WAL usage: 364398 records, 143159 full page images, 1005540671 bytes
delete from test; vacuum verbose test;
WAL usage: 296887 records, 204 full page images, 80286553 bytes
insert into test SELECT * FROM generate_series(1, 10000000) x(id);
UPDATE test SET id = id - 1;
vacuum verbose test;
avg read rate: 126.019 MB/s, avg write rate: 96.746 MB/s
buffer usage: 161648 hits, 224141 misses, 172076 dirtied
WAL usage: 231668 records, 31 full page images, 100307466 bytes
system usage: CPU: user: 10.54 s, system: 1.91 s, elapsed: 13.89 s
```

При full_page_writes неиспользуемое место в блоке (`lp_upper - lp_lower`) в журнал не записывается. Если блок заполнен полностью , то неиспользуемого места в нем нет. В искусственных тестах блоки заполняются полностью в соответствии с `fillfactor=100%`.

Контрольные суммы и WAL

- трудоемкость расчета и проверки контрольных сумм несущественна
- включение контрольных сумм "снижает" производительность тем, что:
  - если выражение `(data_checksums=on OR wal_log_hints=on) AND (full_page_writes=on)` истинно, то при любом изменении содержимого блока между контрольными точками блок запишется в WAL полностью (8Кб минус незанятое место), но только один раз
  - при включенных контрольных суммах блок из буфера копируется в локальную память процесса и оттуда в страничный кэш linux; при копируется из кэша буферов и расходов на копирование блока в локальную память нет
- по умолчанию `wal_log_hints=off`, `full_page_writes=on` и менять значения не стоит
- производительность хоста "снижает" сама работа экземпляра, но это не значит что экземпляр нужно останавливать

Контрольные суммы и WAL

Неверно считать, что контрольные суммы снижают производительность тем, что на их расчет и проверку тратятся ресурсы процессора. Накладные расходы пренебрежимо малы. Например, для блоков WAL подсчет контрольных сумм всегда включён. Включение контрольных сумм на блоках с данными снижает производительность тем, что:

1) изменение битов-подсказок и вообще любого бита в блоке меняет контрольную сумму блока. Образ блока должен быть записан один раз после контрольной точки в WAL в соответствии с `full_page_writes=on`. Увеличение объема WAL значительно в сравнении с отключенными контрольными суммами, но при работе реальных приложений увеличение не так высоко, в сравнении с искусственными тестами.

2) Блок находится в разделяемой памяти - буфере кэша буферов экземпляра. При включенных контрольных суммах блок из буфера копируется в локальную память процесса, который вычисляет и вставляет контрольную сумму в заголовок образа блока в локальной памяти. Из локальной памяти образ посылается на запись в операционную систему - в её страничный (по 4Кб) кэш. При отключенных контрольных суммах образ блока посылается из буфера кэша буферов экземпляра без копирования в локальную память процесса.

Параметр конфигурации `wal_log_hints` примечателен тем, что менять его значение не нужно, он полезен текстом его описания в документации:

1) Параметр может показать насколько увеличится объем WAL при включении контрольных сумм. Включение контрольных сумм относительно долгая операция (чтение-запись каждого блока кластера), а изменение параметра это только рестарт экземпляра

2) При включении параметра в WAL записывается всё содержимое каждого блока **при первом** изменении блока после контрольной точки, даже при изменениях битов-подсказок. При последующих **любых** изменениях полный образ блока не записывается, в журнал записываются только изменения в соответствии с размером этих изменений. Записываются ли изменения битов-подсказок? Как отдельная журнальная запись "поменять бит в блоке" не записываются, потому что при наложении журнальных записей на блок биты-подсказки рассчитываются. Дублирование породило бы неатомарность и вероятность рассогласования. Неатомарность правда всё равно присутствует для части битов, которые - производные от изменений в блоках файлов PGDATA/pg_xact, PGDATA/pg_multixact. А несогласованность? Программный код PostgreSQL устраняет её порядком наложения изменений. Попытки успешны, если нет повреждений. Например, накладывается запись WAL с образом блока, а файла нет (удалили, не зарезервировали), то файл может создаться пустым.

Параметры команды VACUUM

- `DISABLE_PAGE_SKIPPING` обрабатывает все блоки таблиц без исключения
- `SKIP_LOCKED false` - не дает пропускать заблокированные объекты, секции таблиц, блоки
- `INDEX_CLEANUP auto/on/off` указывает нужно ли обрабатывать индексы. OFF используется если нужно быстрее убрать мертвые строки из блоков таблиц
- `PROCESS_TOAST false` - отключает обработку таблиц TOAST
- `TRUNCATE false` - отключает пятую фазу
- `PARALLEL n`. Число *n* ограничивает число фоновых процессов
- `FULL` полная очистка, использует монопольные блокировки, последовательно устанавливаемые на каждую обрабатываемую таблицу

Параметры команды VACUUM

Вакуум можно вызвать вручную, он будет выполняться серверным процессом. Алгоритм выполнения такой же как и у автовакуума и программный код тот же самый, только команде можно передать опции выполнения. Команду VACUUM имеет смысл выполнять после создания таблиц или загрузки данных. Параметры:

`DISABLE_PAGE_SKIPPING` обрабатывает все блоки таблиц без исключения. Если блоки заблокированы, ждет получения блокировки. Включает в себя опцию FREEZE.

`SKIP_LOCKED false` - не дает пропускать заблокированные объекты, секции таблиц, блоки

`INDEX_CLEANUP auto/on/off` указывает нужно ли обрабатывать индексы. OFF используется при приближении к wrap around, когда нужно быстрее убрать мертвые строки из блоков таблиц.

`PROCESS_TOAST false` - отключает обработку таблиц TOAST

`PROCESS_MAIN false` - отключает обработку таблиц и обрабатывает TOAST

`TRUNCATE false` - отключает пятую фазу. На этой фазе устанавливается монопольная блокировка. При ожидании дольше 5 секунд по каждой таблице фаза пропускается. При постановке в очередь монопольная блокировка заставляет ждать все команды, желающие работать с таблицей. Можно установить параметр vacuum_truncate off на уровне таблицы.

`PARALLEL n`. Число *n* ограничивает число фоновых процессов. Их также ограничивает значение параметра max_parallel_maintenance_workers. Параллельные процессы используются, если размер индекса превышает min_parallel_index_scan_size и таких индексов больше одного. На анализ не влияет, только на фазу обработки индексов.

`FULL` полная очистка, использует монопольные блокировки, последовательно устанавливаемые на каждую обрабатываемую таблицу. Требует дополнительное место на диске, так как создаются новые файлы и старые файлы не удаляются до конца транзакции. Возможно, стоит использовать команду `CLUSTER`, так как она выполняется так же, но при этом упорядочивает строки.

Параметры команды VACUUM

- SKIP_DATABASE_STATS отключает обновление pg_database.datfrozenxid
  - позволяет не выполнять полное сканирование таблицы pg_class
- VERBOSE - выводит статистику выполнения команды
- FREEZE - выполняет заморозку строк во всех блоках, кроме тех в которых все строки актуальны и заморожены
- BUFFER_USAGE_LIMIT размер буферного кольца вместо vacuum_buffer_usage_limit
  - в отличие от параметра конфигурации, BUFFER_USAGE_LIMIT можно установить в значение ноль и буферное кольцо не будет использоваться

Параметры команды VACUUM (продолжение)

SKIP_DATABASE_STATS отключает обновление числа pg_database.datfrozenxid - самый старый не замороженный XID в объектах базы данных. Для получения значения выполняется запрос relfrozenxid, relminmxid from pg_class путем полного сканирования (нет индекса по этим столбцам и он не нужен). Если размер pg_class большой, то запрос тратит ресурсы. Можно отключать это и оставлять для любого VACUUM по любой таблице, например, раз в сутки, либо использовать:

VACUUM (ONLY_DATABASE_STATS VERBOSE) который ничего очищать не будет, а только обновит значение pg_database.datfrozenxid.

VERBOSE - выводит статистику выполнения команды. Дополнительной нагрузки не даёт, рекомендуется использовать.

ANALYZE - обновляет статистику. Обновление выполняется отдельно. Совмещение вакуумирования и анализа в одной команде не даёт преимуществ по производительности.

FREEZE - выполняет заморозку строк во всех блоках, кроме тех в которых все строки актуальны и заморожены. Называется "aggressive" режимом. Добавление указания FREEZE равносильно выполнению команды VACUUM с параметрами vacuum_freeze_min_age=0 и vacuum_freeze_table_age=0. В режиме FULL использование FREEZE избыточно, так как FULL тоже замораживает строки.

BUFFER_USAGE_LIMIT размер буферного кольца вместо параметра конфигурации vacuum_buffer_usage_limit (диапазон от 128Кб до 16Мб, по умолчанию 256Мб). В отличие от параметра конфигурации, BUFFER_USAGE_LIMIT можно установить в значение ноль. В этом случае, буферное кольцо не используется и блоки всех обрабатываемых командой объектов как при очистке, так и при анализе могут занять все буфера. Это ускорит выполнение вакуумирования и если кэш буферов большой, загрузит обрабатываемые блоки в него. Пример команды:

```
VACUUM(ANALYZE, BUFFER_USAGE_LIMIT 0);
```

Если автовакуум запускается для защиты от переполнения счетчика транзакций, то буферное кольцо не используется и автоочистка выполняется в агрессивном режиме.

Расширение `pg_visibility`

- стандартное расширение
- для каждого блока таблицы в карте видимости (слой `_vm`) хранится два бита:
  - `all_visible` все строки в блоке актуальны и видны всем транзакциям
  - `all_frozen` все строки в блоке заморожены
- примеры функций:

```
select * from pg_visibility_map_summary('pg_class');
 all_visible | all_frozen
-------------+------------
          14 |         12
select * from pg_visibility('pg_class',0);
 all_visible | all_frozen | pd_all_visible
-------------+------------+----------------
 f           | f          | f
select * from pg_visibility_map('pg_class');
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | f           | f
     1 | f           | f
...
```


Расширение `pg_vsibility`

Стандартное расширение. Позволяет просматривать карту видимости и пересоздавать её. Для каждого блока таблицы в карте видимости (слой `_vm`) хранится два бита (BITS_PER_HEAPBLOCK=2). Один блок карты видимости хранит данные о 32672 блоках таблицы.

1) all_visible все строки в блоке актуальны и видны всем транзакциям и текущим и будущим. **В заголовке блока есть бит-подсказка PD_ALL_VISIBLE, он согласован с битом в карте видимости.** При восстановлении по WAL может случиться, что в карте видимости бит не установлен, а в блоке установлен.

2) all_frozen все строки в блоке заморожены. Пока в блок не будет добавлена, измена, удалена или заблокирована (изменен xmax) строка, вакууму обрабатывать блок не нужно

Функция показывает суммарную информацию, полезна чтобы определить сколько блоков будет читать вакуум:

```
select * from pg_visibility_map_summary('pg_class');
 all_visible | all_frozen
-------------+------------
          14 |         12
```

Также можно проверять эффект автовакуума - были ли заморожены или полностью очищены строки в блоках.

Информация по первому блоку:

```
select * from pg_visibility('pg_class',0);
 all_visible | all_frozen | pd_all_visible
-------------+------------+----------------
 f           | f          | f
```

По всем блокам:

```
select * from pg_visibility_map('pg_class');
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | f           | f
     1 | f           | f
...
```

Пересоздание карты (обнуление) видимости:

```
select pg_truncate_visibility_map('pg_class');
```

Карта заполнится при первом вакуумировании.

Функции `pg_check_visible('pg_class')` и `pg_check_frozen('pg_class')` проверяют не повреждена ли карта видимости. Если повреждений нет, то ничего не выдают.

Мониторинг автовакуума

- на то, что автовакуум с текущими настройками не справляется могут указать:
  - таблицы с мертвыми строками, существенно превышающими порог срабатывания автовакуума;
  - долгая очистка каких-то таблиц;
  - предупреждения в логе
- для мониторинга используются **pg_stat_progress_vacuum, pg_stat_activity, pg_stat_all_tables, pg_class**

```
relation |dead(%)|reltuples|n_dead_tup| effective_settings|  last_vacuumed   | status |..
---------+-------+---------+----------+-------------------+------------------+--------+--
pgbench_ | 19400 |       1 |      194 | vt: 50, vsf: 0.2, | 2024-10-07 12:0 | queued |
branches |       |         |          | DISABLED          | 2:33 (auto)      |        |
pgbench_ |  1160 |      10 |      116 | vt: 50, vsf: 0.2, | 2024-10-07 12:0 | queued |
tellers  |       |         |          | DISABLED          | 2:33 (auto)      |        |

pid | duration | waiting | mode | database | table | phase | table_size | total_size | scanned | vacuumed
----+----------+---------+------+----------+-------+-------+------------+------------+---------+--------
123  01:00:00.0        f  regular  postgres  test   vacuuming indexes 1 GB       2 GB    500 MB   400 MB
```


Мониторинг автовакуума

Автовакуум освобождает место строк, которые не видны ни в одном моментальном снимке (то же самое что вышли за горизонт базы данных) и не удерживаются по обратной связи физическими репликами. Автовакуум обрабатывает таблицы, материализованные представления, секции таблиц, таблицы системного каталога. Автовакуум не обрабатывает временные таблицы, но их могут вакуумировать и анализировать команды VACUUM, ANALYZE в сессии, где создана временная таблица.

Как узнать требуется ли настройка автовакуума? На то, что автовакуум с текущими настройками не справляется могут указать: таблицы с мертвыми строками, существенно превышающими порог срабатывания автовакуума; долгая очистка каких-то таблиц; предупреждения в логе.

Пример запроса:

```
SELECT p.pid, clock_timestamp() - a.xact_start AS duration,
coalesce(wait_event_type ||'.'|| wait_event, 'f') AS waiting,
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,
pg_size_pretty(p.heap_blks_vacuumed * current_setting('block_size')::int) AS vacuumed,
round(100.0 * p.heap_blks_scanned / p.heap_blks_total, 1) AS scanned_pct,
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 dead_pct
FROM pg_stat_progress_vacuum p
JOIN pg_stat_activity a using (pid)
ORDER BY clock_timestamp()-a.xact_start desc;
```

```
pid | duration | waiting | mode | database | table | phase | table_size | total_size | scanned | vacuumed
----+----------+---------+------+----------+-------+-------+------------+------------+---------+--------
123  01:00:00.0        f  regular  postgres  test   vacuuming indexes 1 GB       2 GB    500 MB   400 MB
```

В примере обычный (regular) автовакуум работает 1 час, находится в фазе очистки индексов, не заблокирован (`waiting=f`).

https://dataegret.de/2017/10/deep-dive-into-postgres-stats-pg_stat_progress_vacuum/

Относительно простой запрос, показывающий нуждается ли таблица в вакуумировании:

```
WITH s AS (
 SELECT
  current_setting('autovacuum_analyze_scale_factor')::float8 AS analyze_factor,
  current_setting('autovacuum_analyze_threshold')::float8    AS analyze_threshold,
  current_setting('autovacuum_vacuum_scale_factor')::float8  AS vacuum_factor,
  current_setting('autovacuum_vacuum_threshold')::float8     AS vacuum_threshold
)
SELECT nspname, relname, n_dead_tup, v_threshold, n_mod_since_analyze, a_threshold,
  CASE WHEN n_dead_tup > v_threshold THEN 'yes' ELSE 'no' END AS do_vacuum,
  CASE WHEN n_mod_since_analyze > a_threshold THEN 'yes' ELSE 'no' END AS do_analyze,
  pg_relation_size(relid) AS relsize,
  pg_total_relation_size(relid) AS total
FROM (SELECT n.nspname, c.relname, c.oid AS relid, t.n_dead_tup, t.n_mod_since_analyze,
  trunc(c.reltuples * s.vacuum_factor + s.vacuum_threshold) AS v_threshold,
  trunc(c.reltuples * s.analyze_factor + s.analyze_threshold) AS a_threshold
 FROM s, pg_class c
  JOIN pg_namespace n ON c.relnamespace = n.oid
  JOIN pg_stat_all_tables t ON c.oid = t.relid
 WHERE c.relkind = 'r')
WHERE n_dead_tup > v_threshold OR n_mod_since_analyze > a_threshold
ORDER BY nspname, relname limit 5;
```

```
 nspname  |    relname     |n_dea| v_thre|n_mod_ |a_thre|do_va|do_an|relsize|total
          |                |d_tup| shold |since_ |shold |cuum |alyze|       |
          |                |     |       |analyze|      |     |     |       |
----------+----------------+-----+-------+-------+------+-----+-----+-------+------
pg_catalog|pg_statistic    |  84 | 140.8 |150927 | 95.4 | no  | yes |352256 |589824
public    |pgbench_branches| 210 |  50.2 |   615 | 50.1 | yes | yes | 16384 |106496
public    |pgbench_tellers | 291 |    52 |   615 |   51 | yes | yes | 73728 |466944
```

Запрос не учитывает настройки на уровне таблиц, только параметры уровня кластера, поэтому список таблиц, которые будут вакуумированы или анализированы неточный.
Запрос показывает, что у трёх таблиц v_threshold выше, чем порог срабатывания автоввакуума 20%, установленный по умолчанию

Запрос, учитывающий настройки на уровне таблиц приведен на следующей странице и запрос довольно длинный. Этот запрос выдает приближенный к реальности список таблиц, которые будут вакуумированы в следующем цикле автовакуума. Пример результата этого запроса:

```
relation |dead(%)|reltuples|n_dead_tup| effective_settings|  last_vacuumed  | status |..
---------+-------+---------+----------+-------------------+-----------------+--------+--
pgbench_ | 19400 |       1 |      194 | vt: 50, vsf: 0.2, | 2024-10-07 12:0 | queued |
branches |       |         |          | DISABLED          | 2:33 (auto)     |        |
pgbench_ |  1160 |      10 |      116 | vt: 50, vsf: 0.2, | 2024-10-07 12:0 | queued |
tellers  |       |         |          | DISABLED          | 2:33 (auto)     |        |
```

dead(%) = 19400% указывает на то, что оценка числа мертвых строк, которые могут быть очищены вакуумом существенно превышает порог срабатывания автоввакуума 20%, установленный по умолчанию. Время последнего вакуумирования `last_vacuumed` показывает активно ли менялась таблица (если недавно вакуумировалась) или проблемы в том, что автовакуум не справляется: не мог очистить таблицу (из-за постоянных блокировок на нее) или не может обработать все таблицы.
https://gitlab.com/-/snippets/1889668

```
with table_opts as (
 select pg_class.oid, relname, nspname, array_to_string(reloptions, '') as relopts
 from pg_class join pg_namespace ns on relnamespace = ns.oid
), vacuum_settings as (
   select oid, relname, nspname,
    case
      when relopts like '%autovacuum_vacuum_threshold%' then
        regexp_replace(relopts, '.*autovacuum_vacuum_threshold=([0-9.]+).*', e'\\1')::int8
      else current_setting('autovacuum_vacuum_threshold')::int8
    end as autovacuum_vacuum_threshold,
    case
      when relopts like '%autovacuum_vacuum_scale_factor%'
       then regexp_replace(relopts, '.*autovacuum_vacuum_scale_factor=([0-9.]+).*', e'\\1')::numeric
      else current_setting('autovacuum_vacuum_scale_factor')::numeric
    end as autovacuum_vacuum_scale_factor,
    case
      when relopts ~ 'autovacuum_enabled=(false|off)' then false else true
    end as autovacuum_enabled
   from table_opts
), p as (select * from pg_stat_progress_vacuum)
select
  coalesce(
coalesce(nullif(vacuum_settings.nspname, 'public') || '.', '') || vacuum_settings.relname, --this DB
format('[something in "%I"]', p.datname) --another DB
  ) as relation,
  round((100 * psat.n_dead_tup::numeric / nullif(pg_class.reltuples, 0))::numeric, 0) as "dead(%)",
  pg_class.reltuples::numeric, psat.n_dead_tup,
  format (
    'vt: %s, vsf: %s, %s', -- 'vt' - vacuum_threshold, 'vsf' - vacuum_scale_factor
    vacuum_settings.autovacuum_vacuum_threshold,
    vacuum_settings.autovacuum_vacuum_scale_factor,
    (case when autovacuum_enabled then 'DISABLED' else 'enabled' end)
  ) as effective_settings,
  case
when last_autovacuum > coalesce(last_vacuum, '0001-01-01')
then left(last_autovacuum::text, 19) || ' (auto)'
when last_vacuum is not null then left(last_vacuum::text, 19) || ' (manual)'
    else null
  end as last_vacuumed,
  coalesce(p.phase, 'queued') as status,
  p.pid as pid,
  case
    when a.query ~ '^autovacuum.*to prevent wraparound' then 'wraparound'
    when a.query ~ '^vacuum' then 'user'
    when a.pid is null then null
    else 'regular'
  end as mode,
  case
    when a.pid is null then null
    else coalesce(wait_event_type || '.' || wait_event, 'f')
  end as waiting,
  round(100.0 * p.heap_blks_scanned / nullif(p.heap_blks_total, 0), 1) as scanned_pct,
  round(100.0 * p.heap_blks_vacuumed / nullif(p.heap_blks_total, 0), 1) as vacuumed_pct,
  p.index_vacuum_count,
  case
    when psat.relid is not null and p.relid is not null then
      (select count(*) from pg_index where indrelid = psat.relid)
    else null
  end as index_count
from pg_stat_all_tables psat
join pg_class on psat.relid = pg_class.oid
left join vacuum_settings on pg_class.oid = vacuum_settings.oid
full outer join p on p.relid = psat.relid and p.datname = current_database()
left join pg_stat_activity a using (pid)
where
  psat.relid is null
  or p.phase is not null
  or (autovacuum_vacuum_threshold + (autovacuum_vacuum_scale_factor::numeric * pg_class.reltuples)
    < psat.n_dead_tup
  )
order by status, relation;
```

Представление `pg_stat_progress_vacuum`

- содержит по одной строке для каждого серверного процесса, выполняющего команду VACUUM и каждого autovacuum worker выполняющих вакуумирование в момент обращения к представлению
- в столбце phase отражается текущая фаза вакуума: `initializing` (подготовительная, проходит быстро), `scanning heap`, `vacuuming indexes`, `vacuuming heap`, `cleaning up indexes`, `truncating heap`, `performing final cleanup` (финальная)
- по столбцам `heap_blks_total` , `heap_blks_scanned` , `heap_blks_vacuumed` оценить ход выполнения очистки
- `num_dead_tuples` - число TID, которые сейчас помещены в структуру памяти. Если достигнет `max_dead_tuples` увеличится значение в **`index_vacuum_count`**
- `VACUUM` **`FULL`** отслеживается через `pg_stat_progress_`**`cluster`**
- `ANALYZE` отслеживается через `pg_stat_progress_analyze`

Представление `pg_stat_progress_vacuum`

Представление **pg_stat_progress_vacuum** содержит по одной строке для каждого серверного процесса, выполняющего команду VACUUM и каждого autovacuum worker выполняющих вакуумирование в момент обращения к представлению.

Выполнения VACUUM FULL отслеживается через представление pg_stat_progress_cluster. VACUUM FULL является частным случаем команды CLUSTER и выполняется тем же кодом. Вместо VACUUM FULL оптимально использовать CLUSTER, так как она создает файлы данных со строками в упорядоченном виде.

Команда ANALYZE отслеживается через представление pg_stat_progress_analyze.

В столбце phase отражается текущая фаза вакуума: `initializing` (подготовительная, проходит быстро), `scanning heap`, `vacuuming indexes`, `vacuuming heap`, `cleaning up indexes`, `truncating heap`, `performing final cleanup` (финальная).

Столбцы heap_blks_total , heap_blks_scanned , heap_blks_vacuumed выдают значения в блоках. По значениям можно оценить размер таблицы и сколько блоков уже обработано (оценить ход выполнения очистки).

max_dead_tuples - оценка максимального количества идентификаторов строк (TID), которые поместятся в память, ограниченную параметром autovacuum_work_mem или maintenance_work_mem, действующим для процесса, к которому относится строка представления.

num_dead_tuples - число TID, которые сейчас помещены в структуру памяти. Если число достигнет значения, при котором память будет исчерпана (max_dead_tuples) начнется фаза очистки индексов и увеличится значение в поле index_vacuum_count.

Одновременно можно использовать представление pg_stat_activity, в котором также отображаются действия серверных процессов и рабочих процессов автовакуума (autovacuum workers). Это представление полезно тем, что показывает не ожидает ли чего-то процесс.

При переходу к вакуумированию таблицы считываются параметры, с которыми будет работать процесс и до завершения вакуумирования таблицы поменять значения нельзя.

Параметр `log_autovacuum_min_duration`

- по умолчанию 10 минут
- если автовакуум превысит это время при обработке (вакуум или анализ) таблицы, то в лог кластера запишется сообщение
- при возникновении таких сообщений стоит выяснять причину долгого вакуумирования таблицы

```
LOG:  automatic vacuum of table "postgres.public.pgbench_tellers": index scans: 0
pages: 0 removed, 9 remain, 9 scanned (100.00% of total)
tuples: 125 removed, 10 remain, 0 are dead but not yet removable, oldest xmin: 87738094
removable cutoff: 87738094, which was 2 XIDs old when operation ended
new relfrozenxid: 87738066, which is 609 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 42 hits, 0 misses, 0 dirtied
WAL usage: 4 records, 0 full page images, 722 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
LOG:  automatic analyze of table "postgres.public.pgbench_tellers"
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 52 hits, 0 misses, 0 dirtied
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
```


Параметр `log_autovacuum_min_duration`

Анализировать работу автовакуума можно с помощью журнала. Для этого нужно включить логирование обработки таблиц долгое время. Долгое время указывает либо на то, что таблица долго не вакуумировалась по какой-то причине. Если автовакуум не был отключен, таблица не была долгое время заблокирована (например, безостановочным выполнением команд ANALYZE, режим блокирования которых несовместим с вакуумом), высоким порогом срабатывания автовакуума, то могло случиться что автовакуум был занят вакуумированием других таблиц и до этой таблицы не мог добраться. **Отключать автовакуум нельзя, чем дольше время которое он отключен, тем дольше будет обрабатывать накопившиеся мертвые строки**, когда будет запущен.

Для наблюдения за возникновением долгого времени вакуумирования объектов можно использовать параметр:

`log_autovacuum_min_duration`, по умолчанию установлен в 10 минут. Если автовакуум превысит это время при обработке таблицы, то в лог кластера запишется сообщение. При возникновении таких сообщений стоит выяснять причину долгого вакуумирования таблицы.

Включение коллектора, если сообщения выводятся в журнал linux или другое неудобное место:

```
alter system set logging_collector = on;
sudo systemctl restart tantor-se-server-16.service
```

Установка порога длительности обработки таблицы, дольше которого сообщения должны выводиться в лог. Ноль выводит все сообщения по всем таблицам, при большом количестве таблиц соощений будет много.

```
alter system set log_autovacuum_min_duration = 0;
select pg_reload_conf();
```

Сообщения автовакуума:

```
LOG:  automatic vacuum of table "postgres.public.pgbench_tellers": index scans: 0
pages: 0 removed, 9 remain, 9 scanned (100.00% of total)
...
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
LOG:  automatic analyze of table "postgres.public.pgbench_tellers"
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 52 hits, 0 misses, 0 dirtied
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
```

Параметры конфигурации автовакуума

- Список параметров:

```
select name, setting||coalesce(unit,'') unit, context, min_val, max_val, short_desc from
pg_settings where category='Autovacuum'  or name like '%autovacuum%' order by 2 desc;
                 name                 |    unit    |  context |min_val| max_val
--------------------------------------+------------+----------+-------+----------------
autovacuum                            | on         | sighup   |       |
autovacuum_naptime                    | 60s        | sighup   | 1     | 2147483
log_autovacuum_min_duration           | 600000ms   | sighup   | -1    | 2147483647
autovacuum_analyze_threshold          | 50         | sighup   | 0     | 2147483647
autovacuum_vacuum_threshold           | 50         | sighup   | 0     | 2147483647
autovacuum_max_workers                | 3          |postmaster| 1     | 262143
autovacuum_vacuum_cost_delay          | 2ms        | sighup   | -1    | 100
autovacuum_multixact_freeze_max_age   |20000000000|postmaster| 10000 | 9223372036854775807
autovacuum_work_mem                   | -1kB       | sighup   | -1    | 2147483647
autovacuum_freeze_max_age             |10000000000|postmaster| 100000| 9223372036854775807
autovacuum_vacuum_insert_threshold    | 1000       | sighup   | -1    | 2147483647
autovacuum_vacuum_cost_limit          | -1         | sighup   | -1    | 10000
autovacuum_vacuum_insert_scale_factor| 0.2        | sighup   | 0     | 100
autovacuum_vacuum_scale_factor        | 0.2        | sighup   | 0     | 100
autovacuum_analyze_scale_factor       | 0.1        | sighup   | 0     | 100
(15 rows)
```


Параметры конфигурации автовакуума

Список параметров: **`select name, setting||coalesce(unit,'') unit, context, min_val, max_val, short_desc from pg_settings where category='Autovacuum'  or name like '%autovacuum%' order by 2 desc;`**

```
                 name                 |    unit    |  context |min_val| max_val
--------------------------------------+------------+----------+-------+----------------
autovacuum                            | on         | sighup   |       |
autovacuum_naptime                    | 60s        | sighup   | 1     | 2147483
log_autovacuum_min_duration           | 600000ms   | sighup   | -1    | 2147483647
autovacuum_analyze_threshold          | 50         | sighup   | 0     | 2147483647
autovacuum_vacuum_threshold           | 50         | sighup   | 0     | 2147483647
autovacuum_max_workers                | 3          |postmaster| 1     | 262143
autovacuum_vacuum_cost_delay          | 2ms        | sighup   | -1    | 100
autovacuum_multixact_freeze_max_age   |20000000000|postmaster| 10000 | 9223372036854775807
autovacuum_work_mem                   | -1kB       | sighup   | -1    | 2147483647
autovacuum_freeze_max_age             |10000000000|postmaster| 100000| 9223372036854775807
autovacuum_vacuum_insert_threshold    | 1000       | sighup   | -1    | 2147483647
autovacuum_vacuum_cost_limit          | -1         | sighup   | -1    | 10000
autovacuum_vacuum_insert_scale_factor| 0.2        | sighup   | 0     | 100
autovacuum_vacuum_scale_factor        | 0.2        | sighup   | 0     | 100
autovacuum_analyze_scale_factor       | 0.1        | sighup   | 0     | 100
(15 rows)
```

Значения параметров конфигурации `autovacuum = on` и `track_counts=on` менять не нужно, иначе автовакуум отключится.

`autovacuum_max_workers` определяет максимальное количество фоновых рабочих процессов автовакуума, которые могут выполняться одновременно на экземпляре. Так как процессы автовакуума по умолчанию не ожидают снятия блокировок (не ждут или ждут до 5 секунд), то они работают достаточно активно и их число не должно превышать число ядер процессоров, которые вы хотите выделить для работы процессов автовакуума. При этом, если все процессы автовакуума надолго задержатся на таблицах, то другие таблицы кластера не будут очищаться до тех пор, пока не появится свободный рабочий процесс.

Длительность работы автовакуума на одной таблице зависит от числа листовых блоков всех индексов (для индексов типа btree) на таблице; числа блоков таблицы, в которые вносились изменения с предыдущего вакуумирования (как следствие, частоты вакуумирования этой таблицы); index_vacuum_count (числа итераций сканирования индексов). Для больших таблиц порог в 2% блоков с хотя бы одной мертвой строкой, при недостижении которого индексы не очищаются, уменьшает проблему длительности задержки автовакуума на большой таблице.

https://docs.tantorlabs.ru/tdb/ru/16_4/se/routine-vacuuming.html

Настройка автовакуума

- для минимизации простоя автовакуума `autovacuum_naptime` **стоит установить в минимальное значение: 1 секунду**
- `log_autovacuum_min_duration` стоит подобрать значение (по аналогии с `checkpoint_warnings`), при котором сообщения в диагностический лог кластера будут выводиться не часто
- `autovacuum_vacuum_cost_limit` по умочанию -1, что означает что он равен vacuum_cost_limit, который по умолчанию равен 200
  - делится на всех autovacuum workers
  - по достижению лимита каждый рабочий процесс автовакуума засыпает на время в диапазоне: от `autovacuum_cost_delay` до его четырёхкратного значения
  - параметр можно установить на уровне таблицы
- **`autovacuum_vacuum_cost_delay` стоит установить в ноль на уровне кластера и не устанавливать на уровне таблиц**
  - для команды VACUUM `vacuum_cost_delay` по умолчанию ноль

Настройка автовакуума

Если автовакуум работает на нагруженном экземпляре, то для минимизации простоя: **autovacuum_naptime стоит установить в минимальное значение 1 секунду**. Параметр устанавливает минимальную задержку между двумя запусками автоочистки для отдельной базы данных. Если в базе не было активности, автовакуум не будет на ней работать.

`log_autovacuum_min_duration` по умолчанию 10 минут. Если автовакуум превысит порог, в лог кластера запишется сообщение. По аналогии с `checkpoint_warnings` стоит подобрать значение, при котором сообщения будут возникать не часто.

`autovacuum_vacuum_cost_limit` по умолчанию -1, что означает что он равен `vacuum_cost_limit`, который по умолчанию равен 200 и делится на всех autovacuum workers. При увеличении `autovacuum_max_workers` имеет смысл увеличить значение параметра. По достижению лимита каждый рабочий процесс автовакуума **засыпает на время в диапазоне от `autovacuum_cost_delay` до его четырёхкратного значения**. Параметр можно установить на уровне таблицы. До 12 версии значение по умолчанию было в 10 раз больше.

Для команды VACUUM vacuum_cost_delay=0 и задержки нет потому, что если запущена команда VACUUM, то наверняка хочется чтобы она была выполнена как можно быстрее. Для автовакуума задержка по умолчанию есть. Задержка вакуума вредна тем, что дольше длится транзакция автовакуума. Это значит, что блокировка таблицы и горизонт базы удерживаются дольше. О монопольных блокировках в документации написано: для "certain operations that hold critical locks" задержка не выполняется.

Задержка выполняется функцией vacuum_delay_point(void) из vacuum.c которая вызывается перед сканированием каждого блока. Помимо засыпания функция vacuum_delay_point(void) выполняет расчет и обновление cost, статуса ожидания процесса, на что тратятся ресурсы процессора. Небольшая задержка эквивалентна недозагрузке ядер процессоров, что тоже нежелательно. Расчет не выполняется, если `autovacuum_vacuum_cost_delay=0` на уровне кластера, а на уровне таблиц и TOAST не был установлен в значение больше нуля. Обнуление задержки не приводит к безостановочной работе автовакуума, так как минимальное значение `autovacuum_naptime` 1 секунда. **Имеет смысл установить `autovacuum_vacuum_cost_delay=0` на уровне кластера и не устанавливать на уровне таблиц.**

Изменение значений параметров `autovacuum_vacuum_cost_delay` и `autovacuum_vacuum_cost_limit` действуют немедленно даже на процессы, уже вакуумирующие таблицы.

Параметр `autovacuum_naptime`

- для минимизации простоя автовакуума `autovacuum_naptime` стоит установить в минимальное значение `'1s'`
- если автовакууму не удалось заблокировать таблицу, он ее пропускает. Следующая попытка будет в новом цикле автовакуума
- при **`autovacuum_naptime = '1s'`** попытки через в секунду после окончания предыдущего цикла вакуумирования:

```
22:29:07.466 [195773] LOG:  statement: CREATE INDEX ON test(id);
22:29:07.652 [195796] LOG:  skipping vacuum of "test" --- lock not available
22:29:08.653 [195797] LOG:  skipping vacuum of "test" --- lock not available
22:29:09.652 [195798] LOG:  skipping vacuum of "test" --- lock not available
22:29:10.653 [195799] LOG:  skipping vacuum of "test" --- lock not available
```

- При **`autovacuum_naptime = '3s'`** попытки раз в три секунды:

```
22:33:47.021 [195773] LOG:  statement: CREATE INDEX ON test(id);
22:33:49.366 [195978] LOG:  skipping vacuum of "test" --- lock not available
22:33:52.373 [195981] LOG:  skipping vacuum of "test" --- lock not available
22:33:55.373 [195985] LOG:  skipping vacuum of "test" --- lock not available
22:33:58.378 [195991] LOG:  skipping vacuum of "test" --- lock not available
```


Параметр `autovacuum_naptime`

Если рабочий процесс автовакуума сталкивается с блокировкой объекта, то в этом цикле объект начинать обрабатываться не будет. Блоки на которые автовакуум не смог получить монопольную блокировку в целях освобождения места не будут очищаться до следующего цикла. **Следующий цикл начинается через `autovacuum_naptime`.** Значение по умолчанию 1 минута.

Если рабочий процесс автовакуума не успел выполнить цикл (а объектов может быть много) за `autovacuum_naptime`, процесс `autovacuum launcher` пошлет в ту же базу данных еще один рабочий процесс, если есть свободный рабочий процесс. Второй и следующие рабочие процессы построят свои списки таблиц для обработки и будут работать по своим спискам. В список попадают таблицы, для которых превышены пороговые значения. Рабочие процессы не координируют списки. Если таблица заблокирована другим рабочим процессом. то она просто пропускается.

Настроек, которых у автовакуума нет:

`SKIP_LOCKED false` - не дает команде VACUUM пропускать заблокированные объекты, секции таблиц, блоки

`DISABLE_PAGE_SKIPPING` указывает команде VACUUM обрабатывать все блоки таблиц без исключения. Если блоки заблокированы, команда ждет получения блокировки.

Для автовакуума ожидание получения блокировок есть в агрессивном режиме. Агрессивный режим запускается нечасто. Условия запуска:

если `age(pg_class.relfrozenxid) > vacuum_freeze_table_age - vacuum_freeze_min_age` или если `mxid_age(pg_class.relminmxid) > vacuum_multixact_freeze_table_age - vacuum_freeze_min_age`.

Таким образом, уменьшение `autovacuum_naptime` до минимума (одна секунда) и отключение задержек установкой `autovacuum_vacuum_cost_delay` в нулевое значение повышает вероятность своевременной обработки таблиц автовакуумом и автоанализом. Увеличивать значение больше секунды может быть оптимальным в случае, если `autovacuum_max_workers` установлено в большое значение.

С идентификаторами мультитранзакций нужно использовать функцию mxid_age(..), использование функции age(..) с мультитранзакциями даст неверный результат.

Выбор таблиц автовакуумом

- таблица вакуумируется, если:
  - доля мертвых строк больше scale-фактора для вакуума: `pg_stat_all_tables.n_dead_tup/pg_class.reltuples > autovacuum_vacuum_scale_factor`
  - доля вствленных с последнего вакуумирования трок больше scale-фактора для вставок
- статистика по таблице обновится автовакуумом, если:
  - доля изменившихся строк с последнего анализа больше, чем scale-фактор для анализа:
  - `pg_stat_all_tables.n_mod_since_analyze/pg_class.reltuples > autovacuum_analyze_scale_factor`
- значения параметров `autovacuum_vacuum_threshold`, `autovacuum_vacuum_insert_threshold`, `autovacuum_analyze_scale_factor` можно не учитывать, так как таблицы с небольшим числом строк редко требуют внимания

Выбор таблиц автовакуумом

Формулы, определяющая будет ли автовакуум вакуумировать таблицу.

Условие изменившихся строк:

`pg_stat_all_tables.n_dead_tup > autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor * pg_class.reltuples`

Условие вставленных строк:

`(pg_stat_all_tables.n_tup_ins > 0) && (pg_stat_all_tables.n_ins_since_vacuum > autovacuum_vacuum_insert_threshold + autovacuum_vacuum_insert_scale_factor * pg_class.reltuples)`

Будет ли анализироваться таблица в цикле автоанализа:

`pg_stat_all_tables.n_mod_since_analyze > autovacuum_analyze_threshold + autovacuum_analyze_scale_factor * pg_class.reltuples`

`autovacuum_vacuum_threshold` и `autovacuum_analyze_threshold` по умолчанию 50 строк для подстановки в формулу. Если строк в таблице много и _scale_factor не равен нулю, то слабо влияют на результат.

`autovacuum_vacuum_insert_threshold` = 1000, количество вставленных строк.

`autovacuum_vacuum_insert_scale_factor` по умолчанию 0.2, то есть **20% строк таблицы**. Вакуумирование выполняет заморозку. Параметр был добавлен в 13 версии PostgreSQL исходя из того, что в какие-то таблицы строки только вставлялись и автовакуум их не обрабатывал. и когда на 32-битном счетчике XID подходило время заморозки строк, автовакууму приходилось обрабатывать большое количество строк, что было долго и из-за блокировок не все блоки могли быть быстро обработаны. Поэтому заморозка строк в необработанных заморозкой блоках также должна периодически выполняться.

`autovacuum_analyze_scale_factor` по умолчанию 0.1 (10%)

Команда ANALYZE устанавливает блокировку несовместимую с вакуумом и безостановочно анализировтаь таблицу не стоит. Автоанализ не мешает автовакууму, так как выполняется после автоанализа.

`autovacuum_vacuum_scale_factor` по умолчанию 0.2 (**20% строк таблицы**) Это основной параметр, определяющий будет ли таблица вакуумирована. Если пренебречь абсолютным числом строк (`autovacuum_vacuum_threshold`), то формула сводится к условию:

таблица вакуумируется, если доля мертвых строк больше scale-фактора

(**`pg_stat_all_tables.n_dead_tup`**/pg_class.reltuples>autovacuum_vacuum_scale_factor)

Рекомендации по настройке автовакуума

- возможность очистить старые версии строк в блоках определяется горизонтом базы данных
- если горизонт не сдвигается долгое время, то ни HOT cleanup, ни автовакуум не смогут очистить место в блоках от старых версий строк
- лучше уменьшать задержки в циклах процессов автовакуума, чем увеличивать число рабочих процессов автовакуума

Рекомендации по настройке автовакуума

Возможность очистить старые версии строк в блоках определяется горизонтом базы данных. Если горизонт не сдвигается долгое время, то ни HOT cleanup, ни автовакуум не смогут очистить место в блоках от старых версий строк. Вакуумирование работает в пределах буферного кольца и сканируя блоки таблиц может создавать нагрузку на ввод-вывод. Но делать вывод о том, что автовакууму лучше работать реже нельзя потому, что раз блоки содержат старые версии строк, значит блоки относительно недавно менялись и находятся в буферном кэше. На нагруженных экземплярах объем кэша буферов большой. Буферное кольцо вакуума роли не играет - блок не может находиться в двух буферах одновременно. Если автовакуум будет вакуумировать (хоть и без особых результатов) таблицу реже, то может случиться что часть блоков таблицы вытеснится из буферного кэша автовакуум их загрузит с диска увеличив ввод-вывод. Если автовакуум без задержки пробегает по блокам таблиц, он повышает вероятность их удержания в буферном кэше до следующего сканирования.

А если таблица большая? Вакуум будет пробегать по блокам, которые имеют неактуальные строки. Что это за блоки? В которых были изменения строк и/или потом эти блоки удерживались читателями. Набор блоков в буферном кэше, которые меняются или запрашиваются называют "активным набором" блоков. Автовакуум сканирует активный набор. В "неактивный набор" блоков автовакуум зайдет в целях заморозки, но их количество будет небольшим, так как часто при установке бита all_visible удаётся установить и бит all_frozen. Точное число можно получить запросом к карте видимости и заморозки:

```
select * from pg_visibility_map_summary('таблица');
 all_visible | all_frozen
-------------+------------
         113 |        113
```

Если размер буферного кэша больше "активного набора", лишние операции ввода-вывода, которые могут появиться из-за работы автовакуума вряд ли появятся.

Более оптимально уменьшать задержки в циклах процессов автовакуума, чем увеличивать число рабочих процессов автовакуума. Процессы автовакуума могут быть назначены планировщиком операционной системы на ядра процессоров и через шины памяти, не мешая другим ядрам читать блоки в кэше буферов.

Важность наблюдения за горизонтом баз данных

- наблюдать за горизонтом баз данных нужно для поиска причин, по которым он удерживается
- горизонт удерживают запросы на репликах, если включена обратная связь (`hot_standby_feedback=on`)
- автовакуум стоит настраивать после выбора `checkpoint_timeout` и `shared_buffres`
- время обработки базы данных автовакуумом желательно чтобы было не длиннее времени до горизонта
- автовакуум обрабатывает одну таблицу в одной транзакции, она не должна удерживать горизонт (быть самой длинной по времени)
  - на таких таблицах стоит уменьшить `autovacuum_vacuum_scale_factor`
- уменьшать время обработки баз данных можно уменьшая этот параметр на уровне кластера
- если уменьшить не удается, то увеличивать `autovacuum_max_workers`

Горизонт баз данных

Наблюдать за горизонтом баз данных нужно для поиска причин, по которым он удерживается или не сдвигается долгое время.

Горизонт баз данных кластера в количестве номеров транзакций, отстоящих от текущей:

```
select datname, greatest(max(age(backend_xmin)), max(age(backend_xid))) from pg_stat_activity where backend_xmin is not null or backend_xid is not null group by datname order by datname;
```

Длительность самого долгого запроса или транзакции, который и удерживает горизонт:

```
select datname, extract(epoch from max(clock_timestamp()-xact_start)) from pg_stat_activity where backend_xmin is not null or backend_xid is not null group by datname order by datname;
```

Удержание горизонта (удерживают на всех базах) физическими слотами репликации, если включена обратная связь (`hot_standby_feedback=on`):

```
select max(age(xmin)) from pg_replication_slots;
select backend_xmin, application_name from pg_stat_replication order by age(backend_xmin) desc;
```

В самих репликах искать процессы, выполняющие команды, удерживающие горизонт можно так же, как и на мастере - запросом к `pg_stat_activity`:

```
select backend_xmin, backend_xid, pid, datname, state from pg_stat_activity where backend_xmin IS NOT NULL OR backend_xid IS NOT NULL order by greatest(age(backend_xmin), age(backend_xid)) desc;
```

Автовакуум стоит настраивать после выбора `checkpoint_timeout` и `shared_buffres`. Время обработки базы данных автовакуумом должно быть не очень длинным (не сильно длиннее времени до горизонта). Эти интервалы можно оценить по времени последней обработки таблицы или по упоминанию таблицы в сообщениях автовакуума в логе кластера. Второе, на что можно обратить внимание - таблицы, которые обрабатываются в одном цикле автовакуума дольше всего. На время обработки одной таблицы удерживается горизонт базы. Именно поэтому не стоит использовать `autovacuum_vacuum_cost_delay`. Если именно автовакуум удерживает горизонт, а не серверные процессы, то на таких таблицах стоит уменьшить `autovacuum_vacuum_scale_factor`. На уровне кластера этот параметр стоит уменьшать (увеличивать не стоит), чтобы уменьшить время обработки базы данных до времени удержания горизонта серверными процессами. Если не удаётся, то увеличивать число `autovacuum_max_workers`.

Мониторинг горизонта баз данных

- горизонт баз данных кластера в количестве номеров транзакций, отстоящих от текущей:

```
select datname, greatest(max(age(backend_xmin)), max(age(backend_xid))) from pg_stat_activity
where backend_xmin is not null or backend_xid is not null group by datname order by datname;
```

- длительность самого долгого запроса или транзакции, который и удерживает горизонт:

```
select datname, extract(epoch from max(clock_timestamp()-xact_start)) from pg_stat_activity
where backend_xmin is not null or backend_xid is not null group by datname order by datname;
```

- удержание горизонта (удерживают на всех базах) физическими слотами репликации, если включена обратная связь (hot_standby_feedback=on

```
select max(age(xmin)) from pg_replication_slots;
select backend_xmin, application_name from pg_stat_replication order by  age(backend_xmin) desc;
```

- в самих репликах искать процессы, выполняющие команды, удерживающие горизонт можно так же, как и на мастере - запросом к `pg_stat_activity`

```
select age(backend_xmin), extract(epoch from (clock_timestamp()-xact_start)) secs, pid, datname database,
state from pg_stat_activity where backend_xmin IS NOT NULL OR backend_xid IS NOT NULL order by
greatest(age(backend_xmin), age(backend_xid)) desc;
```


Мониторинг горизонта баз данных

Наблюдать за горизонтом баз данных нужно для поиска причин, по которым он удерживается или не сдвигается долгое время.

Горизонт баз данных кластера в количестве номеров транзакций, отстоящих от текущей:

```
select datname, greatest(max(age(backend_xmin)), max(age(backend_xid))) from
pg_stat_activity where backend_xmin is not null or backend_xid is not null group by
datname order by datname;
```

Длительность самого долгого запроса или транзакции, который и удерживает горизонт:

```
select datname, extract(epoch from max(clock_timestamp()-xact_start)) from
pg_stat_activity where backend_xmin is not null or backend_xid is not null group by
datname order by datname;
```

Удержание горизонта (удерживают на всех базах) физическими слотами репликации, если включена обратная связь (`hot_standby_feedback=on`):

```
select max(age(xmin)) from pg_replication_slots;
select backend_xmin, application_name from pg_stat_replication order by
age(backend_xmin) desc;
```

В самих репликах искать процессы, выполняющие команды, удерживающие горизонт можно так же, как и на мастере - запросом к `pg_stat_activity`:

```
select age(backend_xmin), extract(epoch from (clock_timestamp()-xact_start)) secs, pid,
datname database, state from pg_stat_activity where backend_xmin IS NOT NULL OR
backend_xid IS NOT NULL order by greatest(age(backend_xmin), age(backend_xid)) desc;
```

Автовакуум стоит настраивать после выбора `checkpoint_timeout` и `shared_buffres`. Время обработки базы данных автовакуумом должно быть не очень длинным (не сильно длиннее времени до горизонта). Эти интервалы можно оценить по времени последней обработки таблицы или по упоминанию таблицы в сообщениях автовакуума в логе кластера. Второе, на что можно обратить внимание - таблицы, которые обрабатываются в одном цикле автовакуума дольше всего. На время обработки одной таблицы удерживается горизонт базы. Именно поэтому не стоит использовать `autovacuum_vacuum_cost_delay`. Если именно автовакуум удерживает горизонт, а не серверные процессы, то на таких таблицах стоит уменьшить `autovacuum_vacuum_scale_factor`. На уровне кластера этот параметр стоит уменьшать (увеличивать не стоит), чтобы уменьшить время обработки базы данных до времени удержания горизонта серверными процессами. Если не удаётся, то увеличивать число `autovacuum_max_workers`.

Параметры автовакуума на уровне таблиц

- определить, какие параметры установлены, можно запросом:

```
select nspname, relname, reloptions from pg_class join pg_namespace ns on relnamespace
= ns.oid where reloptions is not null and relkind in ('r','t','m','p') order by 2;
     nspname     |     relname     |                    reloptions
-----------------+-----------------+-------------------------------------------------
 pg_toast_temp_0 | pg_toast_16394  | {vacuum_truncate=false}
 pg_toast        | pg_toast_16399  | {autovacuum_vacuum_cost_delay=23}
 pg_temp_0       | reloptions_test | {vacuum_truncate=false,autovacuum_enabled=false}
 public          | test            | {autovacuum_enabled=off}
```

- вернуть значения по умолчанию:

```
alter table test reset (autovacuum_enabled , fillfactor, ..| all );
```

- установить значения:

```
alter table test set (fillfactor=100,..)
```


Параметры автовакуума на уровне таблиц

На уровне таблиц можно устанавливать параметры автовакуума для таблицы и для ее TOAST-таблицы. Определить, какие параметры установлены, можно запросом:

```
select nspname, relname, reloptions from pg_class join pg_namespace ns on
relnamespace = ns.oid where reloptions is not null and relkind in
('r','t','m','p') order by 2;
     nspname     |     relname     |                    reloptions
-----------------+-----------------+-------------------------------------------------
 pg_toast_temp_0 | pg_toast_16394  | {vacuum_truncate=false}
 pg_toast        | pg_toast_16399  | {autovacuum_vacuum_cost_delay=23}
 pg_temp_0       | reloptions_test | {vacuum_truncate=false,autovacuum_enabled=false}
 public          | test            | {autovacuum_enabled=off}
```

Установить значения можно командой:

```
alter table test set (
AUTOVACUUM_ENABLED
AUTOVACUUM_ANALYZE_SCALE_FACTOR
AUTOVACUUM_VACUUM_INSERT_SCALE_FACTOR
AUTOVACUUM_VACUUM_SCALE_FACTOR
AUTOVACUUM_VACUUM_THRESHOLD
...
FILLFACTOR=N диапазон от 10 до 100
LOG_AUTOVACUUM_MIN_DURATION
PARALLEL_WORKERS
TOAST.AUTOVACUUM* ...,
TOAST_TUPLE_TARGET
TOAST.VACUUM_INDEX_CLEANUP
TOAST.VACUUM_TRUNCATE
VACUUM_INDEX_CLEANUP
VACUUM_TRUNCATE
```

Вернуть значения по умолчанию: `alter table test reset (autovacuum_enabled , fillfactor, ..| all );`

Примеры:
https://github.com/postgres/postgres/blob/master/src/test/regress/sql/reloptions.sql

Параметр `default_statistics_target`

- устанавливает
  - число наиболее часто встречающихся значений в столбцах таблиц (`pg_stats.most_common_vals`)
  - количество корзин в гистрограммах распределения значений в столбцах (`pg_stats.histogram_bounds`)
  - число строк (`default_statistics_target * 300`) случайной выборки по которым собирается статистика
- по умолчанию 100
- максимальное значение 10000
- можно установить для конкретного столбца таблицы или индекса по выражению:

```
alter table test alter column id set statistics 10000;
alter index test alter column  1 set statistics 10000;
```


Параметр `default_statistics_target`

Для сбора статистики используется случайная выборка строк в количестве `300*default_statistics_target`. Значение по умолчанию 100. Максимальное значение 10000. Значение по умолчанию достаточно для репрезентативной выборки и достаточной точности. Помимо этого параметр устанавливает число наиболее часто встречающихся значений в столбцах таблиц (`pg_stats.most_common_vals`) и количество корзин в гистограммах распределения значений в столбцах (`pg_stats.histogram_bounds`). Если в таблице много строк, распределение значений неравномерное, то можно увеличить значение для столбца таблицы командой:

```
alter table test alter column id set statistics 10000;
```

и планировщик будет более точно вычислять стоимость.

Чем больше значение, тем больше времени потребуется для автоанализа и объем статистики будет больше.

Значение -1 возвращает к применению параметра `default_statistics_target`. Команда запрашивает на таблицу блокировку `SHARE UPDATE EXCLUSIVE`.

Для индексов, где индексируются выражения (индекс основанный на функции) можно установить значение командой:

```
alter index test alter column 1 set statistics 10000;
```

Так как у выражений нет уникальных имён, указывается порядковый номер столбца в индексе. Диапазон значений: 0..10000; Значение -1 возвращает к применению параметра `default_statistics_target`.

Значение параметра в диапазоне от 100 до 10000 не влияет на длительность выполнения цикла автоанализа.

Тестирование команды ANALYZE по временной таблице:

https://wiki.astralinux.ru/tandocs/vliyanie-nastrojki-default_statistics_target-na-skorost-vypolneniya-komandy-analyze-294400566.html

Раздувание (bloat) таблиц и индексов

- автовакуум может не обработать таблицу из-за того, что:
  - горизонт базы данных долго не сдвигался
  - в момент обращения автовакуума к таблице на ней была установлена несовместимая с автовакуумом блокировка
- после того как автовакуум отработает, размеры файлов вряд ли уменьшатся
- блоки будут использоваться в будущем под новые версии строк
- слишком часто заниматься мониторингом не нужно, более актуальна проверка свободного места на дисках
- пример, как может выглядеть отчёт:

```
Is N/A |      Table      | Size | Extra        |Bloat estimate|  Live  |   Last Vaccuum    |Fillfactor
-------+-----------------+------+--------------+--------------+--------+-------------------+----------
       | pgbench_history |2832kB|~16 kB (0.56%)|~16 kB (0.56%)|~2816 kB|   11:11:11 (auto)|    100
```

```
create extension pgstattuple;
\dx+ pgstattuple
select relname, b.* from pg_class, pgstattuple_approx(oid) b WHERE relkind='r';
select relname, b.* from pg_class, pgstatindex(oid) b WHERE relkind='i' order by 10;
```


Раздувание (bloat) таблиц и индексов

Старые версии строк хранятся в блоках таблиц. В индексах сохраняются ссылки на версии строк, в том числе старые версии. Автовакуум может не обработать таблицу из-за того, что горизонт базы данных долго не сдвигался или в момент обращения к таблице на ней была установлена несовместимая с автовакуумом блокировка. Во втором случае автовакуум пропускает обработку таблицы. Это приводит к увеличению размера файлов таблиц и индексов. После того как автовакуум отработает, размеры файлов вряд ли уменьшатся. Блоки будут использоваться в будущем под новые версии строк. Раздуванием (bloat) таблиц и индексов можно считать увеличение размера так, что свободное место не будет использовано в ближайшее время. Если размер объекта большой, то неиспользуемое место может быть ощутимо для администратора. Найти таблицы, в которых имеется неиспользуемое место и запустить задачи обслуживания можно с помощью Платформы Тантор.

Можно оценить неиспользуемое место по базовой статистике, собираемой автоанализом. Объекты вряд ли раздуваются быстро, поэтому не нужно часто заниматься мониторингом. Мониторинг свободного места на дисках более актуален. Точность оценки можно верифицировать (сопоставить с реальностью), выполнив полное вакуумирование (`CLUSTER` или `VACUUM FULL`) и сравнить результат с оценкой.

Пример запросов:

https://raw.githubusercontent.com/NikolayS/postgres_dba/refs/heads/master/sql/b1_table_estimation.sql
https://raw.githubusercontent.com/NikolayS/postgres_dba/refs/heads/master/sql/b2_btree_estimation.sql

Пример результата запроса:

```
Is N/A  |      Table      | Size | Extra        |Bloat estimate|  Live  |   Last Vaccuum    |Fillfactor
--------+-----------------+------+--------------+--------------+--------+-------------------+----------
        | pgbench_history |2832kB|~16 kB (0.56%)|~16 kB (0.56%)|~2816 kB|   11:11:11 (auto)|    100
```

Можно использовать функции стандартного расширения pgstattuple:

```
create extension pgstattuple;
\dx+ pgstattuple
select relname, b.* from pg_class, pgstattuple_approx(oid) b WHERE relkind='r' order by 9 desc;
select relname, b.* from pg_class, pgstatindex(oid) b WHERE relkind='i' order by 10;
```

Оценивать можно по `dead_tuple_percent` для таблиц и `avg_leaf_density` для индексов.

Практика

- Часть 1. Параметры команды `vacuum`
- Часть 2. Наблюдение за вакуумом
- Часть 3. Расширение для просмотра карты видимости и заморозки `pg_visibility`
- Часть 4. Интервал между циклами автовакуума, параметр `autovacuum_naptime`
- Часть 5. Сравнение вакуума 17 версии с предыдущими версиями
- Часть 6. Число сканирований индексов при вакуумировании
- Часть 7. Логирование автовакуума
- Часть 8. Расширение `pgststtuple`
- Часть 9: Условие обработки индексов: 2% строк

Практика

В практике детально рассматривается работа вакуума и его диагностика.
Вы сравните работу вакуума в 16 и 17 версиях.

13

Использование диагностического журнала

Диагностический журнал

- `logging_collector` (по умолчанию `off`) рекомендуется установить значение в `on`
- запустится фоновый процесс logger, который собирает (collect) сообщения, отправленные в stderr и записывает их в файлы лога
- `log_min_messages=WARNING` по умолчанию, что означает логирование сообщений с уровнями `ERROR, LOG, FATAL, PANIC`
- `log_min_error_statement`=ERROR по умолчанию. Минимальный уровень важности для команд SQL, которые завершились с ошибкой
- `log_directory=log` (PGDATA/log) по умолчанию. Задаёт путь к директории файлов лога, можно поменять на точку монтирования, скорость и объем которых достаточны для приема логов. Есть параметры, настраивающие ротацию файлов лога

```
psql -c "alter system set logging_collector = on;"
sudo systemctl restart tantor-se-server-16
ps -ef | grep logger
postgres   21861   21860  0 09:37  00:00:00 postgres: logger
```


Диагностический журнал

В коде PostgreSQL вставлены вызовы функции вида:
`ereport(WARNING, (errcode(КОД_СООБЩЕНИЯ), errmsg("текст сообщения")));`
Первый параметр - уровень важности (error level codes). В elog.h определено 15 уровней.
Включение процесса-сборщика логов:
`logging_collector=on` (по умолчанию off). Рекомендуется установить значение в on. По умолчанию сообщения передаются syslog и записываются в его формате, что неудобно для анализа. При большом количестве сообщений, с которыми нельзя справиться (скорость записи в файл ниже, чем скорость генерации) syslog не записывает часть сообщений (и правильно делает), logger же не очищает буфер errlog и процессы экземпляра, генерирующие сообщения блокируются до тех пор пока logger не запишет всё что накопилось (что тоже правильно). Другими словами logger не теряет сообщения, что может быть важным для диагностики. Такая ситуация может возникнуть из-за сбоя записи в файлы лога или включения высокого уровня логирования.
Если `logging_collector=on`, запускается фоновый процесс logger, который собирает (collect) сообщения, отправленные в stderr и записывает их в файлы лога.
Уровень сообщений, записываемых в журнал кластера задаётся параметрами:
`log_min_messages`, по умолчанию WARNING, что означает логирование сообщений с уровнями `ERROR, LOG, FATAL, PANIC`.
`log_min_error_statement`, по умолчанию ERROR. Задает минимальный уровень важности для команд SQL, которые завершились с ошибкой
`log_destination=stderr` менять не нужно
`log_directory=log` (PGDATA/log) по умолчанию. Задаёт путь к директории файлов лога. Можно задать абсолютный путь (/u01/log) или относительно PGDATA (../log).
Название текущего файла (или файлов) лога указано в текстовом файле `PGDATA/current_logfiles`
Уровни важности от большей детальности к меньшей:
`DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1` для отладки
`INFO` сообщения, обычно запрашиваемые опцией команды (VERBOSE)
`NOTICE` Полезные для клиентов сообщения
`WARNING` Предупреждения о возможных проблемах
`ERROR` ошибка, из-за которой прервана текущая команда
`LOG` сообщения, полезные для администраторов
`FATAL` ошибка из-за которой серверный процесс был остановлен (завершена сессия)
`PANIC` остановка серверных процессов основным процессом

Параметры диагностики

- параметров конфигурации логирования больше 35
- также есть больше 8 параметров для отладки команд SQL
- расширения могут иметь параметры для логирования
- основные параметры:

```
alter system set logging_collector=on;
alter system set log_min_duration_statement='8s';
alter system set log_statement=ddl;
alter system set log_min_error_statement=ERROR;
alter system set log_temp_files='1MB';
alter system set cluster_name='main';
alter system set log_autovacuum_min_duration='10s';
alter system set log_disconnections=on;
alter system set log_connections=on;
alter system set log_lock_waits=true;
alter system set deadlock_timeout='60s';
alter system set log_recovery_conflict_waits=on;
select pg_reload_conf();
```

```
\dconfig *debug*
debug_assertions                        | off
debug_discard_caches                    | 0
debug_io_direct                         |
debug_logical_replication_streaming     | buffered
debug_parallel_query                    | off
debug_pretty_print                      | on
debug_print_parse                       | off
debug_print_plan                        | off
debug_print_rewritten                   | off
debug_raw_expression_coverage_test      | off
(10 rows)
\dconfig log*
log_autovacuum_min_duration             | 10min
log_checkpoints                         | on
log_connections                         | off
log_disconnections                      | off
log_duration                            | off
log_error_verbosity                     | default
log_executor_stats                      | off
logging_collector                       | off
log_lock_waits                          | off
log_file_mode                           | 0600
log_filename                            | postgresql-
%Y-%m-%d_%H%M%S.log
logging_collector                       | on
log_hostname                            | off
logical_decoding_work_mem               | 64MB
log_line_prefix                         | %m [%p]
log_lock_waits                          | off
log_min_duration_sample                 | -1
log_min_duration_statement              | -1
```


Параметры диагностики

Какие параметры можно использовать для мониторинга возможных проблем с производительностью?

`log_min_duration_statement='8s'` в лог будут записаны все команды, выполняющиеся заданное значение и дольше. При нулевом значении записывается продолжительность выполнения всех команд. По умолчанию -1 ничего не записывается. Рекомендуется установить, чтобы выявлять длительно выполняющиеся команды (удерживают горизонт базы данных); случаи снижения производительности из-за которого длительность выполнения команд возрастает; возникновение проблем с командами: например, перестал использоваться индекс и время выполнения команд резко возросло. Пример:

```
LOG:  duration: 21585.110 ms
STATEMENT:  CREATE INDEX ON test(id);
```

Выдана длительность и команда.

`log_duration=off` регистрирует продолжительность всех команд после их выполнения. Недостаток: логируются все команды (без текста), одна строка на команду. Включать на уровне кластера не стоит. Преимущество - не логируется текст команд. Параметр можно использовать для сбора статистики по всем командам, но для этого нужно будет какой-то программой обработать файл журнала для анализа собранных данных. Необязательно включать на всём кластере, параметр можно включить на любом уровне. Пример:

```
LOG:  duration: 21585.110 ms
```

`log_statement=ddl` какие типы команд SQL будут логироваться. Значения: `none` (отключено), `ddl`, `mod` (то что ddl плюс команды dml), `all` (все команды).По умолчанию none. Рекомендуется установить в значение ddl. Команды ddl обычно устанавливают более высокий уровень блокирования, что увеличивает конкуренцию (contention). С помощью параметра можно выявить или исключить выполнение команды ddl как причину снижения производительности. Команды с синтаксическими ошибками по умолчанию не записываются. Если нужно логировать команды с ошибками синтаксиса, то нужно установить `log_min_error_statement=ERROR` (или детальнее). Нужно ли логировать команды с синтаксическими ошибками? Команды не нагружают серверный процесс, но могут сильно увеличить сетевой трафик. Причина ошибок может быть в коде приложения, которое безостановочно в цикле повторяет команду. Можно **периодически** включать логирование ошибочных команд. Пример записи при установленном `log_statement=ddl`:

```
LOG:  statement: drop table test;
```

Отслеживание использования временных файлов

- `cluster_name = 'main'` По умолчанию пусто. Рекомендуется установить. Значение добавляется к названию процессов экземпляра, что упрощает их идентификацию. На реплике по умолчанию используется для идентификации `wal_receiver`
- `log_temp_files='1MB'` (по умолчанию отключено) логирует имена и размеры создаваемых временных файлов **в момент их удаления**
- при нулевом значении логируются файлы любого размера
- временные файлы создаются в директории табличных пространств, указанных в параметре `temp_tablespaces`
  - можно ограничить параметром `temp_file_limit`
  - рекомендуется установить `log_temp_files` и `temp_file_limit`
- пример сообщения о том, что временный файл дорос до 97Мб:

```
STATEMENT:  CREATE INDEX ON test(id);
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp137894.0.fileset/0.0",
size 101810176
```


Отслеживание использования временных файлов

При большом количестве команд и замусоривании журнала можно использовать параметры `log_min_duration_sample` и `log_statement_sample_rate`. Параметр `log_transaction_sample_rate` имеет большие накладные расходы, так как обрабатываются все транзакции.

`cluster_name = 'main'` По умолчанию пусто. Рекомендуется установить. Значение добавляется к названию процессов экземпляра, что упрощает их идентификацию. На реплике по умолчанию используется для идентификации `wal_receiver`.

`log_temp_files='1MB'` логирует имена и размеры создаваемых временных файлов в момент их удаления. Почему в момент удаления? Потому что файлы растут в размерах и размер до которого доросли известен только в момент удаления файла. Как предотвратить рост файлов? Размер временных файлов (**в том числе файлов временных таблиц**) можно ограничить параметром `temp_file_limit`. При превышении размера команды будут выдавать ошибку. Пример:

```
insert into temp1 select * from generate_series(1, 1000000);
ERROR:  temporary file size exceeds temp_file_limit (1024kB)
```

Установка `temp_file_limit` поможет выявить ошибки из-за которых план выполнения становится неоптимальным. Например, невозможность использования индекса и вместо него выполнение сортировки огромных объемов строк.

При нулевом значении логируются файлы любого размера, а при положительном - файлы, размер которых не меньше заданного значения. Значение по умолчанию -1, логирование отключено. Рекомендуется установить `log_temp_files` в относительно большое значение, чтобы обнаружить появление команд, которые нагружают дисковую систему. Дисковая система является наиболее нагруженным ресурсом в СУБД.

```
LOG: temporary file: path "base/pgsql_tmp/pgsql_tmp36951.0", size 71835648
STATEMENT:  explain (analyze) select p1.*, p2.* from pg_class p1, pg_class p2
order by random();
```

Временные файлы создаются в директории табличных пространств, указанных в параметре `temp_tablespaces`.

https://docs.tantorlabs.ru/tdb/en/16_4/se/runtime-config-logging.html

Отслеживание работы автовакуума и автоанализа

- log_autovacuum_min_duration, по умолчанию установлен в 10 минут. Если автовакуум превысит это время при обработке таблицы, то в лог кластера запишется сообщение. При возникновении таких сообщений стоит выяснять причину долгого вакуумирования таблицы

```
LOG:  automatic vacuum of table "postgres.public.test": index scans: 37
pages: 0 removed, 88496 remain, 88496 scanned (100.00% of total)
tuples: 10000000 removed, 10000000 remain, 0 are dead but not yet removable
removable cutoff: 799, which was 0 XIDs old when operation ended
new relfrozenxid: 798, which is 2 XIDs ahead of previous value
frozen: 1 pages from table (0.00% of total) had 82 tuples frozen
index scan needed: 44249 pages from table (50.00% of total) had 10000000 dead item identifiers removed
index "test_id_idx": pages: 54840 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 518.021 MB/s, avg write rate: 47.473 MB/s
buffer usage: 385747 hits, 1864788 misses, 170895 dirtied
WAL usage: 231678 records, 83224 full page images, 248448578 bytes
system usage: CPU: user: 25.72 s, system: 0.38 s, elapsed: 28.12 s

LOG:  automatic analyze of table "postgres.public.test"
avg read rate: 498.808 MB/s, avg write rate: 0.018 MB/s
buffer usage: 2906 hits, 27199 misses, 1 dirtied
system usage: CPU: user: 0.42 s, system: 0.00 s, elapsed: 0.42 s
```


Отслеживание работы автовакуума и автоанализа

log_autovacuum_min_duration, по умолчанию установлен в 10 минут. Если автовакуум превысит это время при обработке таблицы, то в лог кластера запишется сообщение. При возникновении таких сообщений стоит выяснять причину долгого вакуумирования таблицы.

Сообщение записывается в лог после завершения обработки таблицы и ее индексов.

Сообщение записывается если "**elapsed:**" > log_autovacuum_min_duration

Полная длительность обработки таблицы и ее индексов указывается в "**elapsed:**". Значение будет больше, чем user + system. user и system это время использования процессора. При вакуумировании процессы могут посылать блоки на запись и ждать выполнения операции ввода-вывода, процессор при этом не нагружается.

В первую очередь стоит смотреть на "**elapsed:**" - это длительность транзакции автовакуума, то есть удержания горизонта. Для TOAST будет отдельная запись о вакуумировании (в том числе агрессивно) в логе со своими показателями, как у обычной таблицы. Записи об автоанализе для TOAST не будет, так как **TOAST не анализируются**:

```
analyze pg_toast.pg_toast_25267;
WARNING:  skipping "pg_toast_25267" --- cannot analyze non-tables or special system tables
```

Во вторую очередь стоит обратить внимание на число проходов по индексам "index scans:". Значение больше 1 указывает на то, что памяти для построения списка TID не хватило. В этом случае стоит увеличить значение параметра:

```
alter system set autovacuum_work_mem='1000MB'; select pg_reload_conf();
```

В третью очередь показатели эффективности цикла автовакуума "tuples:" и "frozen:". "scanned" будет меньше 100%, если блоки были очищены в предыдущем цикле вакуума, это нормально.

Значение "full page images" (и "bytes" пропорциональное ему), к эффективности вакуума не относятся и определяются случайностью: как давно была контрольная точка, либо нужно увеличить checkpoint_timeout. Даже наоборот, если значение "full page images" большие, то это может объяснять долгий цикл (значение в "**elapsed:**"). Большие значения "full page images" и "bytes" вместе с "tuples: число removed" означают эффективность цикла работы автовакуума или то, что он давно не обрабатывал таблицу (например, не мог заблокировать).

"avg read rate" и "avg write rate" ввод-вывод нельзя оценивать, так как он может не быть узким местом.

Наблюдение за контрольными точками

- первая запись передаётся в лог, когда начинается контрольная точка
- total = 09:31:35.070 - 09:27:05.095
  - примерно соответствует 270 секундам, которые получаются перемножением checkpoint_completion_target * checkpoint_timeout (0.9*300=270)
- total=write+sync время записи в WAL-файлы
- sync= время затраченное на вызовы fdatasync по WAL-файлам

```
09:27:05.095 LOG:  checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%); 0 WAL file(s) added, 0 removed,
6 recycled; write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s;
distance=109699 kB, estimate=109699 kB; lsn=8/1164B2E8, redo lsn=8/BC98978
```

- sync files=15 (синхронизировано файлов) - число файлов данных (в табличных пространствах, чьи блоки располагаются в буферном кэше) в которые велась запись и по ним в конце контрольной точке посылался вызов fsync
- longest=0.003 s (самая_долгая синхр.) - наибольшая длительность обработки одного файла

```
LOG:  checkpoint complete: wrote 8596 buffers (52.5%); 0 WAL file(s) added, 0 removed, 33 recycled;
write=25.057 s, sync=9.212 s, total=35.266 s; sync files=4, longest=9.181 s, average=2.303 s;
distance=540552 kB, estimate=550280 kB;  lsn=9/16BC03F0, redo lsn=8/F82504B0
```


Наблюдение за контрольными точками

log_checkpoints **по умолчанию on начиная с 15 версии**. Отключать не стоит, так как позволяет отследить частоту контрольных точек. Более частые контрольные точки приводят к временному повышению нагрузки на журнальную систему (WAL).

log_checkpoints создает записи в логе такого вида:

09:27:05.095 LOG: checkpoint starting: time

09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%);

0 WAL file(s) added, 0 removed, 6 recycled; write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s; distance=109699 kB, estimate=109699 kB; lsn=8/1164B2E8, redo lsn=8/BC98978

Как читать записи:

1) Первая запись передаётся в лог, когда начинается контрольная точка. Между этой записью и записью об окончании контрольной точки может быть много записей. Значение total = 09:31:35.070 - 09:27:05.095 что примерно соответствует 270 секундам, которые получаются перемножением checkpoint_completion_target * checkpoint_timeout (0.9*300=270). Количество блоков, которые должен послать на запись checkpointer расчитывается довольно часто, но ближе к концу интервала может внезапно увеличиться нагрузка на ввод-вывод и checkpointer может не успеть за заданный интервал. Чтобы минимизировать вероятность не вписаться в интервал между контрольными точками (checkpoint_timeout) для checkpoint_completion_target по умолчанию выбрано значение 0.9, которое оставляет зазор в 10% (0.1).

2) total=write+sync. sync - это время затраченное на вызовы fsync. Большое время sync указывает на повышенную нагрузку на ввод-вывод. Эти показатели относятся к файлам данных.

LOG: checkpoint complete: wrote 8596 buffers (52.5%); 0 WAL file(s) added, 0 removed, 33 recycled; write=25.057 s, sync=9.212 s, total=35.266 s; sync files=4, longest=9.181 s, average=2.303 s; distance=540552 kB, estimate=550280 kB;..

3) sync files=15 (синхронизировано файлов) - число обработанных файлов, чьи блоки располагаются в буферном кэше (relations). Контрольная точка в начале записывает блоки буферов slru кэшей, но их размеры невелики. longest=0.003 s (самая_долгая синхр.) - наибольшая длительность обработки одного файла. average=0.001 s - среднее время обработки одного файла. Эти показатели относятся к файлам табличных пространств.

Описание записей log_checkpoints

- wrote 4315 buffers число грязных буферов, которые записаны по контрольной точке. Одновременно с checkpointer грязные блоки могут записывать серверные процессы и bgwriter
- (26.3%) процент от общего количества буферов буферного кэша, задаваемых параметром shared_buffers
- file(s) added, 0 removed, 6 recycled число созданных, удалённых, повторно использованных WAL сегментов
- distance=109699 kB (расстояние) - объем записей WAL между **началом предыдущей** контрольной точки и началом текущей

```
09:22:05.087 LOG: checkpoint starting: time
09:26:35.066 LOG: checkpoint complete: wrote 3019 buffers (18.4%); 0 WAL file(s) added, 0 removed, 6 recycled;
write=269.951 s, sync=0.009 s, total=269.980 s; sync files=14, longest=0.004 s, average=0.001 s;
distance=99467 kB, estimate=108859 kB; lsn=8/AA004C8, redo lsn=8/5177990
09:27:05.095 LOG: checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%); 0 WAL file(s) added, 0 removed, 6 recycled;
write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s;
distance=109699 kB, estimate=109699 kB; lsn=8/1164B2E8, redo lsn=8/BC98978
```


Описание записей log_checkpoints

log_checkpoints создает записи в логе такого вида:

09:22:05.087 LOG: checkpoint starting: time
09:26:35.066 LOG: checkpoint complete: wrote 3019 buffers (18.4%);
0 WAL file(s) added, 0 removed, 6 recycled; write=269.951 s, sync=0.009 s,
total=269.980 s; sync files=14, longest=0.004 s, average=0.001 s;
distance=99467 kB, **estimate**=108859 kB; lsn=8/AA004C8, redo lsn=8/5177990
09:27:05.095 LOG: checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%);
0 WAL file(s) added, 0 removed, 6 recycled; write=269.938 s, sync=0.009 s,
total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s; **distance**=109699
kB, **estimate**=109699 kB; lsn=8/1164B2E8, redo lsn=8/BC98978

Как читать записи (продолжение):

4) wrote 4315 buffers количество грязных блоков, которые записаны по контрольной точке. Одновременно с checkpointer грязные блоки могут записывать серверные процессы и bgwriter. (26.3%) процент от общего количества буферов буферного кэша, задаваемых параметром shared_buffers. В примере 4315/16384*100%=26.3366699%

5) file(s) added, 0 removed, 6 recycled число созданных удалённых, повторно использованных WAL сегментов (по умолчанию, размер каждого сегмента 16Мб).

6) distance=109699 kB (расстояние) - объем записей WAL между началом предыдущей контрольной точки и началом завершенной контрольной точки

select '8/BC98978'::pg_lsn-'8/5177990'::pg_lsn; = 112332776 = 109699kB

Описание записей log_checkpoints

- estimate=109699 kB (расстояние которое ожидалось) рассчитывается, чтобы оценить сколько WAL сегментов будет использовано в следующей контрольной точке
- если нули в "0 WAL file(s) added, 0 removed", то оценка estimate верная. Сколько файлов удалить определяется параметрами min_wal_size, max_wal_size, wal_keep_size, max_slot_wal_keep_size, wal_init_zero=on, wal_recycle=on

```
09:22:05.087 LOG: checkpoint starting: time
09:26:35.066 LOG: checkpoint complete: wrote 3019 buffers (18.4%); 0 WAL file(s) added, 0 removed, 6 recycled;
write=269.951 s, sync=0.009 s, total=269.980 s; sync files=14, longest=0.004 s, average=0.001 s;
distance=99467 kB, estimate=108859 kB; lsn=8/AA004C8, redo lsn=8/5177990
09:27:05.095 LOG: checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%); 0 WAL file(s) added, 0 removed, 6 recycled;
write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s;
distance=109699 kB, estimate=109699 kB; lsn=8/1164B2E8, redo lsn=8/BC98978
```


Описание записей log_checkpoints (продолжение)

7) После checkpoint starting: указываются свойства контрольной точки. time означает, что контрольная точка была вызвана "по времени" по истечении checkpoint_timeout . Если размер WAL превышает max_wal_size будет сообщение:

LOG: checkpoint starting: wal

Если контрольная точка по wal начнется раньше, чем checkpoint_warning, то выдастся сообщение:

LOG: checkpoints are occurring too frequently (23 seconds apart)
HINT: Consider increasing the configuration parameter "max_wal_size".

23 секунды меньше, чем установлено checkpoint_warning= '30s'

Для контрольных точек после рестарта экземпляра:

LOG: checkpoint starting: end-of-recovery immediate wait

8) estimate=109699 kB (расстояние которое ожидалось) - обновляется по формуле:

```
if (estimate < distance) estimate = distance
 else estimate=0.90*estimate+0.10*distance;
```

(числа фиксированы в коде PostgreSQL)

Показатель estimate рассчитывается кодом контрольной точки, чтобы оценить сколько WAL сегментов будет использовано в следующей контрольной точке. Исходя из estimate в конце контрольной точки определяется сколько файлов переименовать с целью повторного использования, а оставшиеся удалить. Сколько файлов удалить определяется параметрами min_wal_size, max_wal_size, wal_keep_size, max_slot_wal_keep_size, wal_init_zero=on, wal_recycle=on. Повторное использование файлов не стоит отключать, оно оптимально для файловой системы ext4. Другие файловые системы (zfs, xfs, btrfs) не стоит использовать. Если нули в "0 WAL file(s) added, 0 removed", то оценка estimate верная. Такие значения должны быть большую часть контрольных точек. Цель отображения значения estimate в этом. Объем журнальных записей между контрольными точками это **distance**.

9) Между контрольными точками прошло 09:27:05.095 - 09:22:05.087 = 300.008 секунд, что с высокой точностью равно checkpoint_timeout=300s

Утилита pg_waldump и записи log_checkpoints

- для просмотра записей в WAL-файлах используется утилита pg_waldump. По умолчанию утилита ищет WAL-файлы в "." (текущей директории откуда она запущена), дальше в ./pg_wal, $PGDATA/pg_wal
- в логе и выводе pg_controldata в LSN ведущие нули после "/" не печатаются
- в выводе pg_waldump в lsn и prev ноль печатается, а в redo не печатается

```
pg_controldata | grep check | head -n 3
Latest checkpoint location:           8/1164B2E8
Latest checkpoint's REDO location:    8/0BC98978
Latest checkpoint's REDO WAL file:    000000010000000080000000B
```

```
pg_waldump -s 8/0B000000 | grep CHECKPOINT
rmgr: XLOG len (rec/tot): 148/148, tx: 0,
lsn: 8/1164B2E8, prev 8/1164B298, desc: CHECKPOINT_ONLINE redo 8/0BC98978;
tli 1; prev tli 1; fpw true; xid 8064948; oid 33402; multi 1; offset 0; oldest xid 723 in DB 1;
oldest multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0; oldest running xid 8064947; online
pg_waldump: error: error in WAL record at 8/1361C488: invalid record length at 8/1361C4B0:
expected at least 26, got 0
```

```
09:27:05.095 LOG:  checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%); 0 WAL file(s) added, 0 removed,
6 recycled; write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s;
distance=109699 kB, estimate=109699 kB; lsn=8/1164B2E8, redo lsn=8/0BC98978
```


Утилита pg_waldump и записи log_checkpoints

Данные о последней контрольной точке записываются в управляющий файл. Для просмотра содержимого управляющего файла используется утилита **pg_controldata**:

```
pg_controldata | grep check | head -n 3
Latest checkpoint location:           8/1164B2E8
Latest checkpoint's REDO location:    8/0BC98978
Latest checkpoint's REDO WAL file:    000000010000000080000000B
```

Ноль после слэша ("/") не печатается, в примерах на слайде и под слайдом нули добавлены вручную.

Данные соответствуют записи о последней контрольной точке в логе.

Для просмотра записей в WAL-файлах используется утилита pg_waldump. По умолчанию утилита ищет WAL-файлы в текущей директории откуда она запущена, потом в директориях ./pg_wal, $PGDATA/pg_wal. Пример просмотра записи в журнале об окончании контрольной точки:

```
pg_waldump -s 8/0B000000 | grep CHECKPOINT
```

или

```
pg_waldump -s 8/BC98978 | grep CHECKPOINT
rmgr: XLOG len (rec/tot): 148/148, tx: 0,
lsn: 8/1164B2E8, prev 8/1164B298, desc: CHECKPOINT_ONLINE redo 8/0BC98978;
tli 1; prev tli 1; fpw true; xid 8064948; oid 33402; multi 1; offset 0; oldest xid 723 in DB 1; oldest multi 1
in DB 5; oldest/newest commit timestamp xid: 0/0; oldest running xid 8064947; online
```

Утилите не указан LSN до которого сканировать журнал (параметр -e), поэтому дойдя до самой последней журнальной записи, которая была записана в журнал, утилита выводит сообщение, что следующая запись пустая:

```
pg_waldump: error: error in WAL record at 8/1361C488: invalid record length at
8/1361C4B0: expected at least 26, got 0
```

В логе, выводе утилиты pg_controldata в LSN ведущие нули после "/" не печатаются

в выводе pg_waldump в lsn и prev ноль печатается, а в redo не печатается. Перед числом 8 нули тоже незримо присутствуют, но их отсутствие не создаёт путаницу. Можно запомнить, что после слэша должно идти восемь HEX-символов.

Утилита pg_waldump и записи log_checkpoints

- lsn 8/1164B2E8 запись о конце контрольной точки
- redo 8/0BC98978 запись о начале контрольной точки, с которой начнется восстановление в случае сбоя экземпляра
- prev 8/1164B298 адрес начала предыдущей записи в журнале
- distance объем журнала от начала предыдущей до начала завершенной контрольной точки '8/0BC98978'::pg_lsn-'8/05177990'::pg_lsn

```
pg_controldata | grep check | head -n 3
Latest checkpoint location:           8/1164B2E8
Latest checkpoint's REDO location:    8/BC98978
Latest checkpoint's REDO WAL file:    000000010000000080000000B
```

```
pg_waldump -s 8/0B000000 | grep CHECKPOINT
rmgr: XLOG len (rec/tot): 148/148, tx: 0,
lsn: 8/1164B2E8, prev 8/1164B298, desc: CHECKPOINT_ONLINE redo 8/BC98978;...
```

```
09:26:35.066 LOG: checkpoint complete: wrote 3019 buffers (18.4%); 0 WAL file(s) added, 0 removed, 6 recycled;
write=269.951 s, sync=0.009 s, total=269.980 s; sync files=14, longest=0.004 s, average=0.001 s;
distance=99467 kB, estimate=108859 kB; lsn=8/AA004C8, redo lsn=8/5177990
09:27:05.095 LOG:  checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%); 0 WAL file(s) added, 0 removed,
6 recycled; write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s;
distance=109699 kB, estimate=109699 kB; lsn=8/1164B2E8, redo lsn=8/BC98978
```


Утилита pg_waldump и записи log_checkpoints (продолжение)

lsn 8/1164B2E8, запись о конце контрольной точки.

redo 8/0BC98978 запись о начале контрольной точки, с которой начнется восстановление в случае сбоя экземпляра. Из записи выбирается адрес записи, которая формировалась на момент начала контрольной точки (redo), читается эта запись. Все записи от redo до lsn должны быть прочитаны и наложены на файлы кластера. После наложения lsn файлы кластера считаются согласованными.

prev 8/1164B298 адрес начала предыдущей записи в журнале. По журналу можно скользить "назад". При этом в журнальных записях отсутствуют LSN следующей журнальной записи. Почему? Адрес следующей журнальной записи можно рассчитать по полю len (rec/tot): 148/148 которое хранит длинну журнальной записи. Минимальная длинна журнальной записи 26 байт (expected at least 26). При этом реальная длинна журнальной записи дополняется (padding) до 8 байт. Реальная длинна записи в примере будет **152** байта, а не 148. Пример:

```
pg_waldump -s 8/1164B298 -e 8/1164B3E8
 rmgr: Standby     len (rec/tot):     76/    76, tx:        0, lsn: 8/1164B298, prev 8/1164B240, desc:
RUNNING_XACTS nextXid 8232887 latestCompletedXid 8232885 oldestRunningXid 8232886; 1 xacts: 8232886
 rmgr: XLOG        len (rec/tot):    148/   148, tx:        0, lsn: 8/1164B2E8, prev 8/1164B298, desc:
CHECKPOINT_ONLINE redo 8/BC98978; ...
 rmgr: Heap        len (rec/tot):     86/    86, tx: 8232886, lsn: 8/1164B380, prev 8/1164B2E8, desc: HOT_UPDATE
...
```

lsn+len + padding до 8 байт = LSN начала следующей записи

По записи в логе или управляющем файле можно узнать размер журнальных записей. От него зависит длительность восстановления.

Объем записанных WAL в контрольной точке вычисляется по этим полям:

`select pg_wal_lsn_diff('8/1164B2E8','8/BC98978');` = 94054768 = 91850kB.

Объем WAL от начала до конца контрольной точки 91850kB.

Объем от начала предыдущей контрольной точки до начала завершенной, то есть расстояние (distance) между контрольными точками:

`select '8/BC98978'::pg_lsn - '8/5177990'::pg_lsn;` = 112332776 = 109699kB

Для вычислений можно использовать функцию pg_wal_lsn_diff или оператор "-", результаты одинаковы. Для использования оператора нужно привести строку к типу pg_lsn.

Диагностика частоты соединений с базой данных

- `log_disconnections=on` записывает в лог событие завершения сессии. Записывается та же информация, что `log_connections` плюс длительность сессии.
  - преимущество в том, что выводится одна строка, что не замусоривает лог
  - позволяет идентифицировать короткие по времени сессии
  - пример сообщения о длительности сессии сессии 4 секунды:

```
LOG: disconnection: session time: 0:00:04.056 user=oleg database=db1 host=[vm1]
```

- `log_connections=on` записывает в лог попытки установить сессию
  - недостаток в том, что неудачные попытки отличаются только дополнительной строкой:

```
LOG: connection received: host=[local]
LOG: connection authorized: user=postgres database=db2 application_name=psql
FATAL: database "db2" does not exist
LOG: connection received: host=[local]
LOG: connection authorized: user=alice database=alice application_name=psql
FATAL: role "alice" does not exist
```


Диагностика частоты соединений с базой данных

`log_disconnections=on` записывает в лог событие завершения сессии. Записывается та же информация, что `log_connections` плюс **длительность сессии**. Преимущество в том, что выводится одна строка, что не замусоривает лог. Позволяет идентифицировать короткие по времени сессии. Короткие сессии приводят к частому порождению серверных процессов, что увеличивает нагрузку и снижает производительность:

```
LOG: disconnection: session time: 0:00:04.056 user=oleg database=db1 host=[vm1]
```

В примере длительность сессии 4 секунды.

`log_connections=on` записывает в лог попытки установить сессию. Недостаток в том, что для многих типов клиентов в журнал выводится две строки: первая строка об определении способа аутентификации (без пароля, с паролем), вторая строка - аутентификация. Если не используется балансировщик соединений (pgbouncer), то до аутентификации порождается серверный процесс, это трудоемкая операция. Параметр полезен для идентификации проблем, когда клиент безостановочно пытается подсоединиться с неверным паролем или к несуществующей базе или с несуществующей ролью. Недостаток в том, что неудачные попытки отличаются только дополнительной строкой:

```
LOG: connection received: host=[local]
LOG: connection authorized: user=postgres database=db2 application_name=psql
FATAL: database "db2" does not exist
LOG: connection received: host=[local]
LOG: connection authorized: user=alice database=alice application_name=psql
FATAL: role "alice" does not exist
```

`log_hostname=off`. Включать не стоит, так как вносит существенные задержки при логировании создания сессии.

Диагностика блокирующих ситуаций

- `log_lock_waits=true`. По умолчанию отключен. Рекомендуется включить, чтобы получать сообщения в диагностический журнал что какой-либо процесс ждет дольше, чем: `deadlock_timeout`
- `deadlock_timeout='60s'`. По умолчанию 1 секунда, что слишком мало и на нагруженных экземплярах создает значительные издержки
- `log_startup_progress_interval='10s'` не стоит отключать
- `log_recovery_conflict_waits=on`. По умолчанию `off`. Процесс `startup` запишет сообщение в лог реплики, если не сможет применить WAL к реплике дольше, чем `deadlock_timeout`

```
LOG: recovery still waiting after 60.555 ms: recovery conflict on lock
DETAIL: Conflicting process: 5555.
```

- Наличие конфликтов можно увидеть в представлении (мало деталей):

```
select * from pg_stat_database_conflicts where datname='postgres';
datid|datname |tblspc|confl_lock|confl_snapshot|confl_bufferpin|deadlock
-----+--------+------+----------+--------------+---------------+--------
13842|postgres|    0 |        0 |            1 |             1 |       0
```


Диагностика блокирующих ситуаций

`log_lock_waits=true`. По умолчанию отключен. Рекомендуется включить, чтобы получать сообщения в диагностический журнал что какой-либо процесс ждет дольше, чем: `deadlock_timeout='60s'`. По умолчанию 1 секунда, что слишком мало и на нагруженных экземплярах создает значительные издержки. Рекомендуется настраивать значение `deadlock_timeout` так, чтобы сообщения об ожиданиях получения блокировки возникали редко. Как первое приближение можно ориентироваться на длительность типичной транзакции (для реплики - самый долгий запрос).

В 15 версии появился параметр `log_startup_progress_interval='10s'` который **не стоит отключать** (устанавливать ноль). Если процесс `startup` (выполняет восстановление), столкнется с длительной операцией, то в лог будет записано сообщение об этой операции. Сообщения позволят выявить либо проблемы с файловой системой, либо высокую нагрузку на дисковую систему. Пример сообщений процесса startup при восстановлении:

```
LOG:  syncing data directory (fsync), elapsed time: 10.07 s, current path: ./base/4/2658
LOG:  syncing data directory (fsync), elapsed time: 20.16 s, current path: ./base/4/2680
LOG:  syncing data directory (fsync), elapsed time: 30.01 s, current path: ./base/4/PG_VERSION
```

`log_recovery_conflict_waits=on`. По умолчанию `off`. **Параметр появился в 14 версии.** Процесс `startup` запишет сообщение в лог реплики, если не сможет применить WAL к реплике дольше, чем `deadlock_timeout`. Задержка может произойти из-за того, что серверный процесс на реплике выполняет команду или транзакцию (для повторяемости по чтению) и блокирует применение WAL из-за параметра `max_standby_streaming_delay` (по умолчанию 30s). Позволяет идентифицировать случаи отставания реплики. Действует на реплике, на мастере можно установить заранее. Рекомендуется установить в значение `on`.

```
LOG: recovery still waiting after 60.555 ms: recovery conflict on lock
DETAIL: Conflicting process: 5555.
CONTEXT: WAL redo at 0/3044D08 for Heap2/PRUNE: latestRemovedXid 744 nredirected
0 ndead 1; blkref #0: rel 1663/13842/16385, blk 0
```

Наличие конфликтов можно увидеть в представлении, но в нем мало деталей:

```
select * from pg_stat_database_conflicts where datname='postgres';
datid|datname |tblspc|confl_lock|confl_snapshot|confl_bufferpin|deadlock
-----+--------+------+----------+--------------+---------------+--------
13842|postgres|    0 |        0 |            1 |             1 |       0
```

Практика

- Часть 1. Чтение сообщений вакуума и автовакуума
- Часть 2. Чтение сообщений о контрольной точке
- Часть 3. Чтение сообщений о контрольной точке pg_waldump
- Часть 4. Размер директории PGDATA/pg_wal

Практика

В практике рассматривается часть тем, изученных в этой главе.

14

Накопительная статистика

Накопительная статистика

- накапливает статистику о работе экземпляра. Сбор статистики создает накладные расходы, поэтому есть параметры, которыми можно включить сбор дополнительной статистики
- параметры которые включены не стоит отключать, так как их результат используется процессами экземпляра

```
select name, setting, unit, context, min_val, max_val from pg_settings where
name like 'track%';
             name              | setting | unit |  context   | min_val | max_val
-------------------------------+---------+------+------------+---------+---------
 track_activities              | on      |      | superuser  |         |
 track_activity_query_size     | 1024    | B    | postmaster | 100     | 1048576
 track_commit_timestamp        | off     |      | postmaster |         |
 track_counts                  | on      |      | superuser  |         |
 track_functions               | none    |      | superuser  |         |
 track_io_timing               | off     |      | superuser  |         |
 track_wal_io_timing           | off     |      | superuser  |         |
(7 rows)
```


Накопительная статистика

накапливает статистику о работе экземпляра. Сбор статистики создает накладные расходы, поэтому есть параметры, которыми можно включить сбор дополнительной статистики:

```
select name, setting, unit, context, min_val, max_val from pg_settings where name like 'track%';
             name              | setting | unit |  context   | min_val | max_val
-------------------------------+---------+------+------------+---------+---------
 track_activities              | on      |      | superuser  |         |
 track_activity_query_size     | 1024    | B    | postmaster | 100     | 1048576
 track_commit_timestamp        | off     |      | postmaster |         |
 track_counts                  | on      |      | superuser  |         |
 track_functions               | none    |      | superuser  |         |
 track_io_timing               | off     |      | superuser  |         |
 track_wal_io_timing           | off     |      | superuser  |         |
(7 rows)
```

Статистика использования таблиц и индексов (`track_counts=on`) используется автовакуумом и отключать ее сбор не нужно. То что отключено экземпляром не используется. Статистику `track_commit_timestamp` можно использовать в логике разрешения конфликтов при логической репликации.

`track_functions` включает подсчёт вызовов функций и времени их выполнения. Значение pl включает отслеживание функций на языке `plpgsql`, а `all` также функций на языках SQL и C. Данные можно при оптимизации кода подпрограмм.

`track_io_timing` накапливает не только события ожидания, но и длительность событий. Этот параметр полезен для мониторинга и в процессе настройки производительности. По умолчанию отключён, так как так как потребуется часто обращаться к счетчику времени, что замедляет работу. Статистику можно посмотреть в представлениях pg_stat_database, pg_stat_io.

`track_wal_io_timing` включает замер длительности записи в журналы. Статистику можно посмотреть в представлении pg_stat_wal. По умолчанию отключён, так как потребуется часто обращаться к счетчику времени, что замедляет работу. Скорость обращения к счетчику можно измерить утилитой командной строки `pg_test_timing`. Если не использовать `pg_stat_wal` и не анализировать скорость работы журналов, то включать не стоит.

`stats_fetch_consistency` по умолчанию cache, менять не нужно. Если открыть транзакцию, то при первом обращении к статистике по объекту, она не обновится до конца транзакции. Обновить можно вызвав `pg_stat_clear_snapshot()`. Значение none не будет кэшировать статистику. Значение snapshot кэширует всю статистику накопленную по всем объектам базы данных, на что тратятся ресурсы. Без транзакции различий нет.

Утилита pg_test_timing

- позволяет оценить издержки на получение времени
- любая статистика с временем - следствие обращения к счетчику времени
- пример накладных расходов команды EXPLAIN ANALYZE, которая делает 2 замера времени на каждую строку:

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timingcv
SELECT COUNT(*) FROM t;
Time: 15.526 ms
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
Time: 310.682 ms
```

```
pg_test_timing
Testing timing overhead for 3 seconds.
Per loop time including overhead: 1424.21 ns
Histogram of timing durations:
  < us   % of total      count
     1      0.03043        641
     2     64.30326    1354500
     4     35.46601     747065
     8      0.01234        260
    16      0.09889       2083
```

- задержка зависит от используемого счетчика времени
- задержку измеряет утилита командной строки pg_test_timing
- наиболее быстрый счетчик tsc:

```
cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```


Утилита pg_test_timing

Утилита позволяет оценить издержки на получение времени. Любая статистика с временем - следствие обращения к счетчику времени. Насколько обращения влияют на время выполнения команд?

Оценить накладные расходы можно сравнив скорость выполнения команды и команды explain analyze:

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
Time: 15.526 ms
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
Time: 310.682 ms
```

295 миллисекунды это накладные расходы на 2 измерения на каждую строку, которые выполняет команда explain analyze.

Результат запуска утилиты **pg_test_timing**:

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 1424.21 ns
Histogram of timing durations:
  < us   % of total      count
     1      0.03043        641
     2     64.30326    1354500
     4     35.46601     747065
     8      0.01234        260
    16      0.09889       2083
```

1424.21 наносекунды (1.4 микросекунды) это задержка (накладные расходы) на операцию получения времени. 1.424 мкс * 100000 строк * 2 измерения= 284,8 мс, что соответствует 295 мс.

Если счетчик времени быстрый, то гистограмма распределения утилиты должна дать overhead в пределах 100 ns и более 90% запросов должны укладываться в первую строку (1 us мкс микросекунду) в гистограмме распределения.

Скорость счетчика времени зависит от его типа. Тип можно посмотреть командой:

```
cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

Наиболее быстрый счетчик tsc. acpi_pm более медленный.

Просмотр статистики работы процессов

- статистика передается серверными процессами в общую память перед простоем, то есть после выполнения команды и не чаще, чем раз в секунду
- суперпользователи и роли pg_read_all_stats, pg_monitor видят данные обо всех сессиях. Остальные пользователи видят по своей сессии, по остальным сессиям в полях null
- для просмотра накопительной статистики используется более 30 представлений и более 100 функций с названием pg_stat*
- представление pg_stat_database содержит по одной строке для каждой базы данных:

```
select datname, numbackends, sessions, round(100.0*blks_hit/NULLIF(blks_read+blks_hit,0),2) hitratio,
temp_files, temp_bytes, round((100.0*active_time/NULLIF(session_time+active_time, 0))::numeric,2) activeratio,
idle_in_transaction_time idleintrans, blk_read_time, blk_write_time from pg_stat_database;
```

datname	numbackends	sessions	hitratio	temp_files	temp_bytes	activeratio	idleintrans	blk_read_time	blk_write_time
	0	0	99.75	0	0		0	0	0
postgres	1	13	97.96	1	1400000	0.07	36177.453	8194.412	9.947
my_test_db	0	0		0	0		0	0	0
template1	0	0		0	0		0	0	0
template0	0	0		0	0		0	0	0

Просмотр статистики работы процессов

Статистика передается серверными процессами в общую память перед простоем, то есть после выполнения команды и не чаще, чем раз в секунду. Исключения:

1) представление `pg_stat_activity`, в котором отображается информация о текущем запросе (если `track_activities=on`).

2) в представлениях `pg_stat_xact_all_tables`, `pg_stat_xact_sys_tables`, `pg_stat_xact_user_tables`, `pg_stat_xact_user_functions` транзакции доступна её собственная статистика, ещё не переданная в общую память

3) столбец `pg_stat_database.numbackends`

Суперпользователи и роли `pg_read_all_stats`, `pg_monitor` видят данные обо всех сессиях. Остальные пользователи видят по своей сессии, по остальным сессиям в полях null. Для просмотра накопительной статистики используется более 30 представлений и более 100 функций с названием `pg_stat*`.

Представление **`pg_stat_database`** содержит по одной строке для каждой базы данных плюс строку с `datid=0` со статистикой по глобальным объектам.

Основные столбцы:

`stats_reset` время и дата последнего сброса статистики

`numbackends` число сессий к базе данных в текущий момент

`sessions` число сессий, которые были созданы со времени последнего сброса статистики

`blks_read` - блоки отсутствовали в буферном кэше, `blks_hit` - блоки присутствовали в буферном кэше. Не нужно переоценивать эту статистику, так как есть страничный кэш linux. В результатах команды `explain (analyze, buffers)` аналогичные данные выдаются в строках:

`Buffers: shared hit=4073 read=56839`

`temp_files, temp_bytes` число созданных временных файлов и сколько в них записано

`blk_read_time, blk_write_time` заполняются, если `track_io_timing=on` время в миллисекундах затраченное серверными процессами на ввод-вывод (чтение или запись блоков файлов данных)

`active_time/session_time` доля активности сессий

`idle_in_transaction_time` время простоя если серверный процесс имеет открытую транзакцию. Чем меньше значение, тем лучше.

Представление pg_stat_database

- представление pg_stat_database содержит по одной строке для каждой базы данных
- если растет число xact_rollback, то это может указывать на ошибки в коде приложения
- если имеются deadlocks, то это указывает на ошибки в логике приложения

```
select datname, round(100*xact_commit::numeric/nullif(xact_commit+xact_rollback,0),2) as cratio,
xact_rollback rollbacks, deadlocks, conflicts, temp_files, pg_size_pretty(temp_bytes) as tempsize
from pg_stat_database;
  datname | cratio | rollbacks | deadlocks | conflicts | temp_files | tempsize
---------+--------+-----------+-----------+-----------+------------+----------
          |        |         0 |         0 |         0 |          0 | 0 bytes
 postgres | 100.00 |        10 |         0 |         0 |          0 | 0 bytes
```

- Процент попадания должен быть не меньше 90%

```
select datname, numbackends, sessions, round(100.0*blks_hit/NULLIF(blks_read+blks_hit,0),2) hitratio,
temp_files, temp_bytes, round((100.0*active_time/NULLIF(session_time+active_time, 0))::numeric,2)
activeratio, idle_in_transaction_time idleintrans, blk_read_time, blk_write_time from pg_stat_database;
   datname   | numbackends | sessions | hitratio | temp_files | temp_bytes | activeratio | idleintrans | blk_read_time | blk_write_time
-------------+-------------+----------+----------+------------+------------+-------------+-------------+---------------+----------------
             |           0 |        0 |    99.75 |          0 |          0 |             |           0 |             0 |              0
 postgres    |           1 |       13 |    97.96 |          1 |    1400000 |        0.07 |   36177.453 |      8194.412 |          9.947
```


Представление pg_stat_database

Пример запроса:

```
select datname, numbackends, sessions, round(100.0*blks_hit/NULLIF(blks_read +
blks_hit,0),2) hitratio, temp_files, temp_bytes,
round((100.0*active_time/NULLIF(session_time+active_time, 0))::numeric,2) activeratio,
idle_in_transaction_time idleintrans, blk_read_time, blk_write_time from
pg_stat_database;
```

Процент попадания в кэш буферов:

```
round(100.0*blks_hit/NULLIF(blks_read+blks_hit,0),2) hitratio
```

Метрика дает оценку какая часть блоков берется из буферного кэша, а какая с диска или из страничного кэша. Процент попадания должен быть не меньше 90%. Меньшее значение указывает на то, что "активный набор" - данные, с которыми работают запросы не помещается в буферном кэше. Из-за этого блоки скорее всего вытесняются и подгружаются заново. В этом случае стоит увеличить значение `shared_buffers` и убедиться, что физической памяти достаточно.

Также можно проверить нет ли большого числа откатов транзакций:

```
select datname, round(100*xact_commit::numeric/nullif(xact_commit+xact_rollback,0),2) as
cratio, xact_rollback rollbacks, deadlocks, conflicts, temp_files,
pg_size_pretty(temp_bytes) as tempsize from pg_stat_database;
  datname | cratio | rollbacks | deadlocks | conflicts | temp_files | tempsize
---------+--------+-----------+-----------+-----------+------------+----------
          |        |         0 |         0 |         0 |          0 | 0 bytes
 postgres | 100.00 |        10 |         0 |         0 |          0 | 0 bytes
```

Если растет число **xact_rollback**, то это может указывать на ошибки в коде приложения.

Если имеются `deadlocks`, то это указывает на ошибки в логике приложения.

`conflicts` связаны с репликацией.

При написании запросов можно использовать функцию `NULLIF()`, чтобы избежать деления на ноль. Для вычисления десятичных значений использовать `100*integer::numeric` или `100.0*integer`.

Прогресс выполнения команд

- Шесть представлений содержат данные о выполняющихся командах:
- pg_stat_progress_analyze - `analyze` и автоанализа
- pg_stat_progress_basebackup - утилиты `pg_basebackup`
- pg_stat_progress_cluster - `cluster` и `vacuum full`
- pg_stat_progress_copy - `copy`
- pg_stat_progress_create_index - `create index` и `reindex`
- pg_stat_progress_vacuum
- можно оценить через какое время команда выполнится и наблюдать за фазами выполнения команд

```
pg_basebackup -D $HOME/backup/1 -P --max-rate=1000
28130/4564784 kB (0%), 0/1 tablespace
select * from pg_stat_progress_basebackup \watch 5
pid|          phase          |backup_total|backup_streamed|tablespaces_total|tablespaces_streamed
---+-------------------------+------------+---------------+-----------------+--------------------
414|streaming database files| 4674334720 |      6502912  |               1 |                  0
```


Прогресс выполнения команд

Шесть представлений `pg_stat_progress_*` содержат данные о выполняющихся командах: `analyze` и автоанализа, `create index` и `reindex`, `vacuum` и автовакуум, `cluster` и `vacuum full`, `copy`, а также утилиты командной строки `pg_basebackup`:

```
\dv pg_stat_progress_*
                    List of relations
   Schema   |              Name               | Type |  Owner
------------+---------------------------------+------+----------
 pg_catalog | pg_stat_progress_analyze        | view | postgres
 pg_catalog | pg_stat_progress_basebackup     | view | postgres
 pg_catalog | pg_stat_progress_cluster        | view | postgres
 pg_catalog | pg_stat_progress_copy           | view | postgres
 pg_catalog | pg_stat_progress_create_index   | view | postgres
 pg_catalog | pg_stat_progress_vacuum         | view | postgres
(6 rows)
```

По этим представлениям можно оценить через какое время команда выполнится и наблюдать за фазами выполнения команд.

Пример наблюдения за резервированием:

```
pg_basebackup -D $HOME/backup/1 -P --max-rate=1000
28130/4564784 kB (0%), 0/1 tablespace
select * from pg_stat_progress_basebackup \watch 5
pid|          phase          |backup_total|backup_streamed|tablespaces_total|tablespaces_streamed
---+-------------------------+------------+---------------+-----------------+--------------------
414|streaming database files| 4674334720 |      6502912  |               1 |                  0
```

До 17 версии опция `--max-rate` (скорость резервирования) нагружала ядро процессора на 100%. В 17 версии ошибка исправлена и нагрузки на процессор нет.

Если в psql нужно выполнять команду через несколько секунд, то вместо ";" в конце команды можно использовать `\watch секунд` и команда будет выполняться с заданным интервалом.

Пример запросов к представлениям:

https://dev.to/bolajiwahab/progress-reporting-in-postgresql-1i0d

https://docs.tantorlabs.ru/tdb/ru/16_4/se/progress-reporting.html

Представление pg_stat_io

- статистика ввода-вывода
- статистика по каждому типу процесса
- в столбце object могут быть значения:
  - relation (объект постоянного хранения)
  - temp relation (временный объект)
- в столбце context могут быть значения:
  - normal
  - bulkread, bulkwrite, vacuum (буферные кольца)
- по этим столбцам удобно группировать:

```
select * from pg_stat_bgwriter\gx
-[ RECORD 1 ]---------+--------
checkpoints_timed     | 151
checkpoints_req       | 2
checkpoint_write_time | 5074064
checkpoint_sync_time  | 250
buffers_checkpoint    | 93237
buffers_clean         | 622
maxwritten_clean      | 0
buffers_backend       | 18716
buffers_backend_fsync | 0
buffers_alloc         | 21328
```

```
select backend_type name, sum(writes) buffers_written, sum(write_time) write_time, sum(writebacks) writebacks,
sum(writeback_time) writeback_time, sum(evictions) evictions, sum(fsyncs) fsyncs, sum(fsync_time) fsync_time from
pg_stat_io group by backend_type having sum(writes)> 0 or sum(writebacks)> 0 or sum(fsyncs)>0 or sum(evictions)>0;
       name        | buffers_written | write_time | writebacks | writeback_time | evictions | fsyncs | fsync_time
-------------------+-----------------+------------+------------+----------------+-----------+--------+-----------
client backend     |               0 |          0 |          0 |              0 |       370 |      0 |          0
autovacuum worker  |               0 |          0 |          0 |              0 |        57 |      0 |          0
background writer  |             622 |      6.401 |        576 |         83.795 |           |      0 |          0
checkpointer       |           93035 |  1386.3310 |      93028 |        323.601 |           |    304 |    215.698
```


Представление pg_stat_io

В представлении есть статистика по каждому процессу. Строки дублируются из-за столбцов object и context. В столбце object могут быть значения: relation (объект постоянного хранения) и temp relation (временный объект). В столбце context могут быть значения: normal, bulkread, bulkwrite, vacuum. Последние три это буферные кольца. Для уменьшения числа строк в результате удобно использовать группировку:

```
select backend_type name, sum(writes) buffers_written, sum(write_time)
write_time, sum(writebacks) writebacks, sum(writeback_time) writeback_time,
sum(evictions) evictions, sum(fsyncs) fsyncs, sum(fsync_time) fsync_time from
pg_stat_io group by backend_type having sum(writes)> 0 or sum(writebacks)> 0 or
sum(fsyncs)>0 or sum(evictions)>0;
```

```
     name        |buf_written|write_time|writebacks|writeback_time|evictions|fsyncs|fsync_time
-----------------+-----------+----------+----------+--------------+---------+------+----------
client backend   |          0|         0|         0|             0|      370|     0|         0
autovacuum worker|          0|         0|         0|             0|       57|     0|         0
background writer|        622|     6.401|       576|        83.795|         |     0|         0
checkpointer     |      93035|  1386.331|     93028|       323.601|         |   304|   215.698
```

Чтобы не усложнять запрос в нем не фильтруются пустые значения и не исключаются `object='temp relation'`.

```
select * from pg_stat_bgwriter\gx
-[ RECORD 1 ]----------------------------+--------
checkpoints_timed     (num_timed)        | 151
checkpoints_req       (num_requested)    | 2
checkpoint_write_time (write_time)       | 5074064
checkpoint_sync_time  (sync_time)        | 250
buffers_checkpoint    (buffers_written)  | 93237
buffers_clean                            | 622
maxwritten_clean                         | 0
buffers_backend                          | 18716
buffers_backend_fsync                    | 0
buffers_alloc                            | 21328
```

В 17 версии статистику по серверным процессам `buffers_backend` и `buffers_backend_fsync` из `pg_stat_bgwriter` удалили, считая, что ее можно посмотреть в `pg_stat_io`. Другие столбцы переименовали и перенесли в появившееся в 17 версии представление `pg_stat_checkpointer`.

https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=74604a37f

Статистики `buffers_backend_fsync` и `fsyncs`

- статистики `pg_stat_bgwriter.buffers_backend_fsync` и `pg_stat_io.fsyncs, pg_stat_io.fsync_time` по `backend_type='client backend'` показывает сколько вызовов fsync выполняли серверные процессы
- если статистики больше нуля
  - стоит проверить статистики и параметры процесса bgwriter
  - производительность ввода-вывода может быть низкой и являться узким местом
  - если `pg_stat_bgwriter.maxwritten_clean` имеет большое значение, то это указывает на то, что стоит увеличивать `bgwriter_lru_maxpages`

Статистики `buffers_backend_fsync` и `fsyncs`

Статистики `pg_stat_bgwriter.buffers_backend_fsync` и `pg_stat_io.fsyncs, pg_stat_io.fsync_time` по `backend_type='client backend'` показывает сколько вызовов fsync выполняли серверные процессы.

При нормальной работе процессы (кроме checkpointer) не выполняют fsync по файлам данных (файлы данных находятся в табличных пространствах), а посылают идентификаторы блоков в разделяемую структуру памяти процесса checkpoiner и checkpointer сортируя блоки по каждому файлу передает операционной системе вызовы writeback по диапазонам страниц, соответствующим блокам и fsyncs по файлам данных. **Если checkpointer не справляется (структура памяти заполнена), то процесс выполняет запись сам**. Процессы инициируют запись блока на диск, если им нужно подгрузить блок в буфер и для этого они вытесняют (evict) грязный буфер из кэша буферов. Такое происходит при поиске свободного буфера серверными процессами, параллельными процессами, рабочими процессами автовакуума. Желательно чтобы процессы не сталкивались с задержками, особенно если `context=normal, vacuum`. Значения столбца `context=bulkwrite, bulkread` указывают, что использовался кольцевой буфер.

Если статистики больше нуля, то стоит проверить статистики и параметры процесса bgwriter, возможно, он должен работать более активно. Также производительность ввода-вывода может быть низкой и являться узким местом.

bgwriter сканирует заголовки буферов разделяемого пула и грязные страницы с `usagecount=0` и `pincount=0` посылает на запись. Процесс обеспечивает потребность в чистых страницах в буферах, если они потребуются в большом количестве. Если `pg_stat_bgwriter.maxwritten_clean` имеет большое значение, то это указывает на то, что bgwriter медленно работает и стоит увеличивать `bgwriter_lru_maxpages`, чтобы bgwriter за один цикл смог очистить больше блоков. Если значение относительно большое, то уменьшить `bgwriter_delay`.

Строки представления pg_stat_io

```
select backend_type, left(object,4) obj, context, writes w, round(write_time::numeric,2) wt, writebacks wb,
round(writeback_time::numeric,2) wbt, extends ex, round(extend_time::numeric,2) et, evictions ev, reuses ru,
fsyncs fs, round(fsync_time::numeric,2) fst from pg_stat_io;
```

backend_type	object	context	w	wt	wb	wbt	ex	et	ev	ru	fs	fst
autovacuum launcher	relation	bulkread	0	0.00	0	0.00			0	0		
autovacuum launcher	relation	normal	0	0.00	0	0.00			0		0	0.00
autovacuum worker	relation	bulkread	0	0.00	0	0.00			0	0		
autovacuum worker	relation	normal	0	0.00	0	0.00	3	0.085	0		0	0.00
autovacuum worker	relation	vacuum	0	0.00	0	0.00	0	0	0	0		
client backend	relation	bulkread	0	0.00	0	0.00			0	0		
client backend	relation	bulkwrite	0	0.00	0	0.00	6	0.136	0	0		
client backend	relation	normal	0	0.00	0	0.00	165	1.967	0		0	0.00
client backend	relation	vacuum	24	0.24	0	0.00	0	0	0	24		
client backend	temp relation	normal	50	0.39			151	1.567	51			
background worker	relation	bulkread	0	0.00	0	0.00			0	0		
background worker	relation	bulkwrite	0	0.00	0	0.00	0	0	0	0		
background worker	relation	normal	0	0.00	0	0.00	0	0	0		0	0.00
background worker	relation	vacuum	0	0.00	0	0.00	0	0	0	0		
background worker	temp relation	normal	0	0.00			0	0	0			
background writer	relation	normal	0	0.00	0	0.00					0	0.00
checkpointer	relation	normal	94	4.76	94	2.70					62	160.88
standalone backend	relation	bulkread	0	0.00	0	0.00			0	0		
standalone backend	relation	bulkwrite	0	0.00	0	0.00	0	0	0	0		
standalone backend	relation	normal	0	0.00	0	0.00	0	0	0		0	0.00
standalone backend	relation	vacuum	0	0.00	0	0.00	0	0	0	0		
startup	relation	bulkread	0	0.00	0	0.00			0	0		
startup	relation	bulkwrite	0	0.00	0	0.00	0	0	0	0		
startup	relation	normal	0	0.00	0	0.00	0	0	0		0	0.00
startup	relation	vacuum	0	0.00	0	0.00	0	0	0	0		
walsender	relation	bulkread	0	0.00	0	0.00			0	0		
walsender	relation	bulkwrite	0	0.00	0	0.00	0	0	0	0		
walsender	relation	normal	0	0.00	0	0.00	0	0	0		0	0.00
walsender	relation	vacuum	0	0.00	0	0.00	0	0	0	0		
walsender	temp relation	normal	0	0.00			0	0	0			

(30 rows)

Строки представления pg_stat_io

Строки представления относятся к типам процессов, по которым собирается статистика. Названия процессов дублируются потому, что отдельно собирается статистика по двум типам объектов - постоянным и временным и по контекстам. Контекст normal - обычная работа с блоком в разделяемом буферном кэше и локальном кэше (для временных объектов). Локальный буфер не использует буферные кольца, поэтому в строках с object='temp_relation' есть только context='normal'.

В представлении есть столбец op_bytes, его значение всегда равно размеру блока данных (8192), поэтом выводить его в запросах нет смысла.

evictions заполняется с любым контекстом - и кольцами и normal. evictions означает, что в буфере был заменён блок.

reused заполняется только для колец.

В чем отличия evictions от reused при работе с кольцами?

Замена блока в буфере учитывается в статистике evictions соответствующего типа (context) кольца. Если же блок находится в буфере относящимся к кольцу и идёт замена блока на другой блок в буфере того же кольца, то замена блока учитывается в статистике reused этого типа (context) кольца.

Пример запроса к представлению:

```
select backend_type, context, writes, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric,2) et, hits, evictions ev, reuses, fsyncs fs,
round(fsync_time::numeric,2) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
```

backend_type	context	writes	write_time	wb	wbt	ex	et	hits	ev	reuses	fsyncs	fst
autovacuum worker	normal	4	83.46	4	0.06	48	0.59	77932	6		4	73.67
autovacuum worker	vacuum	3	0.07	0	0.00	0	0.00	26747	0	3		
client backend	bulkread	0	0.00	0	0.00			0	320	723283		
client backend	normal	8024	126050.50	8024	203.95	7537	57465.56	4019507	29556		12761	179428.24
background worker	bulkread	0	0.00	0	0.00			0	640	356537		
background worker	normal	203	6.51	203	110.04	0	0.00	9191	1737		0	0.00
background writer	normal	17675	250172.10	17675	4937.16						13779	228832.52
checkpointer	normal	12250	235.43	12250	1853.66						33033	272578.26

(8 rows)

Характеристики `pg_stat_io`

- основной интерес представляют столбцы writebacks, writeback_time fsyncs, fsync_time так как именно они приводят к передаче системных вызовов в операционную систему
- writebacks выполняются:
  - серверными процессами при расширении файлов (extends) и поиске буфера на вытеснение
  - процессами bgwriter и checkpointer, которые работают только в контексте normal
- fsyncs
  - передаются процессу checkpointer для выполнения в конце контрольной точки один раз по каждому файлу
  - если не могут быть переданы (нет места в структуре памяти, которую очищает checkpointer), то выполняются самим процессом
  - у колец fsyncs не считаются
- у bgwriter и checkpointer в столбцах reads, hits, evictions всегда нули
- у bgwriter и autovacuum launcher в столбце extends нули

Характеристики `pg_stat_io`

Столбцы writeback_time и fsync_time заполняются только, если параметр конфигурации track_io_timing = on.

С временными объектами writebacks и fsyncs не выполняются, так как гарантия записи временным таблицам не нужна.

reuses непустые только у колец, у normal отсутствуют.

У колец fsyncs не считаются.

У bulkread отсутствуют extends, так как при чтении файлы не меняют размер.

Процессы autovacuum launcher и autovacuum worker не работают с кольцами типа bulkwrite. Процесс autovacuum launcher не работает с кольцом типа vacuum.

Процессы bgwriter и checkpointer с буферными кольцами не работают.

У bgwriter и checkpointer в столбцах reads, hits, evictions всегда нули.

У bgwriter и autovacuum launcher в столбце extends нули, так как они не увеличивают размеры файлов.

По процессу loggingg collector статистика не собирается, так как процесс loggingg collector не имеет доступа к разделяемой памяти.

По процессу архиватора статистика не собирается, так как archiver запускает команды или программы, которые и выполняют операции ввода-вывода.

Процессы WAL Receiver и WAL Writer отсутствуют в pg_stat_io (не реализовано).

Основной интерес представляют столбцы writebacks и fsyncs, так как именно они приводят к передаче системных вызовов в операционную систему.

writebacks выполняются:

a) серверными процессами при расширении файлов (extends) и поиске буфера на вытеснение (вызовом функции GetVictimBuffer(..))

b) процессами bgwriter (функцией BgBufferSync()) и checkpointer (функцией BufferSync()) в контексте normal. Обе функции вызывают SyncOneBuffer(..) и использует очередь процесса checkpointer путем вызова функции ScheduleBufferTagForWriteback(..).

Правило выполнения fsyncs процессами: If there is a local pending-ops table, just make an entry in it for ProcessSyncRequests to process later. Otherwise, try to pass off the fsync request to the checkpointer process. If that fails (нет места в структуре памяти, которую очищает checkpointer), just do the fsync locally before returning (we hope this will not happen often enough to be a performance problem).

Статистики представления pg_stat_io

- evictions сколько раз процесс очистил буфер для нового блока
  - был ли буфер грязным или чистым не отражается
- context=bulkwrite,bulkread означает, что использовался кольцевой буфер
- extends, extend_time - время, затраченное на увеличение файлов relations
- buffers_alloc сколько раз процессы загружали блоки в буфер, если при этом не использовалось буферное кольцо
- если свободных буферов в кэше буферов нет, то buffers_alloc соответствуют evictions

```
select backend_type, context, reads, trunc(read_time) r_time, writes, trunc(write_time) w_time, extends, extend_time,
hits, evictions, reuses from pg_stat_io where reads<>0 or writes<>0 or extends<>0 or hits<>0 or evictions<>0;
    backend_type     | context | reads | r_time | writes | w_time | extends | extend_time |    hits     | evictions | reuses
---------------------+---------+-------+--------+--------+--------+---------+-------------+-------------+-----------+-------
 autovacuum launcher | normal  |     1 |      6 |      0 |      0 |         |             |         590 |         0 |
 autovacuum worker   | normal  |   217 |    310 |      0 |      0 |       2 |       0.038 |      799993 |       101 |
 autovacuum worker   | vacuum  | 47889 |    305 |     13 |      0 |       0 |           0 |     1636676 |       702 | 47171
 client backend      | normal  |  2420 |   2573 |      0 |      0 |   30914 |     764.359 |   459710941 |     12675 |
 client backend      | vacuum  |     1 |      0 |      0 |      0 |       0 |           0 |           3 |         0 |     0
 background writer   | normal  |       |        |   2751 |     21 |         |             |             |           |
 checkpointer        | normal  |       |        | 169856 |   2338 |         |             |             |           |
```


Статистики представления pg_stat_io

В примере статистика по серверным процессам в pg_stat_io нулевая, кроме evictions, в pg_stat_bgwriter.buffers_backend ненулевая. Процессы сами не пишут, они передают блоки на запись процессу checkpointer, за исключением использования буферного кольца.

Если процесс (например, серверный или рабочий процесс автовакуума) очистил буфер для нового блока, то увеличивается evictions. Был ли буфер грязным или чистым не отражается.

context=bulkwrite,bulkread означает, что использовался кольцевой буфер.

extends, extend_time - время, затраченное на увеличение файлов relations.

в столбце object значения relation или temp relation и можно сгруппировать результат получив данные по временным таблицам и индексам.

Производительность снижает расширение файлов. В примере на расширение файлов потрачено 764 миллисекунды, а на чтение блоков 2571 миллисекунды.

pg_stat_bgwriter.buffers_alloc не относится к bgwriter, но эту статистику оставили в представлении и даже не удалили. bgwriter не анализирует эту статистику, хотя цель появления этой статистики была в: "count buffer allocation requests so that the bgwriter can estimate the rate of buffer consumption".

Статистика numBufferAllocs, соответствующая buffers_alloc увеличивается при вызове функции StrategyGetBuffer(..) если не используется буферное кольцо. Функция в этом случае вернет буфер, в который может быть загружен блок. **Если свободных буферов в кэше буферов нет, то buffers_alloc соответствуют evictions**. Если свободные буфера есть, то они берутся из списка свободных статистика увеличивается, а evictions не увеличиваются:

```
select backend_type, context, evictions from pg_stat_io where  evictions<>0;
    backend_type     | context | evictions
---------------------+---------+-----------
 autovacuum worker   | normal  |     60758
 autovacuum worker   | vacuum  |     23958
 client backend      | normal  |   3259501
select buffers_alloc from pg_stat_bgwriter;
   3365169
select * from pg_buffercache_summary();
 buffers_used | buffers_unused | buffers_dirty | buffers_pinned | usagecount_avg
--------------+----------------+---------------+----------------+----------------
        16384 |              0 |           642 |              2 |   4.1064453125
```

Представления pg_statio_all_tables и pg_statio_all_indexes

- в pg_statio_all_tables статистика чтения блоков таблиц, всех индексов на эту таблицу, TOAST таблицы и её индекса
- в pg_statio_all_indexes статистика по конкретному индексу
  - "hit" означает, что блок находился в буферном кэше
  - "read" означает, что блок отсутствовал в буферном кэше, процессу приходилось очищать буфер (если список свободных блоков пуст), загружать в очищенный буфер блок и только потом читать его содержимое

```
 select schemaname||'.'||relname name, heap_blks_read tabread, heap_blks_hit tabhit, idx_blks_read idxread, idx_blks_hit
idxhit, toast_blks_read toastread, toast_blks_hit toasthit from pg_statio_all_tables order by 2 desc limit 3;
          name           | tabread  |  tabhit  | idxread |  idxhit  | toastread | toasthit
-------------------------+----------+----------+---------+----------+-----------+----------
 public.pgbench_accounts | 22702740 | 82951068 |  479143 | 36142117 |           |
 pg_catalog.pg_statistic |   193579 |  2042126 |    2792 |    41643 |         5 |        1
 public.pgbench_history  |   148559 |  5882517 |         |          |           |
 select schemaname||'.'||relname table, indexrelname index, idx_blks_read idxread, idx_blks_hit idxhit from
pg_statio_all_indexes order by 4 desc limit 3;
          table           |         index          | idxread |  idxhit
-------------------------+------------------------+---------+----------
 public.pgbench_accounts | pgbench_accounts_pkey |  483328 | 36178084
 public.pgbench_tellers  | pgbench_tellers_pkey  |    1128 | 13472177
 public.pgbench_branches | pgbench_branches_pkey |    1129 |  9042373
```


Представления pg_statio_all_tables и pg_statio_all_indexes

В этих представлениях статистика чтения блоков таблиц, всех индексов на эту таблицу, TOAST таблицы и её индекса (к TOAST доступ идёт всегда через TOAST-индекс и поэтому данные по TOAST-индексу пропорциональны данным по TOAST-таблице) с подгрузкой с диска (столбцы *_blks_read) и из кэша буферов (столбцы *_blks_hit).

В представлении pg_statio_all_tables статистика по всем индексам на таблицу. Статистику (чтение с подгрузкой с диска и из кэша буферов) по конкретному индексу можно посмотреть в представлении pg_statio_all_indexes.

"**hit**" означает, что блок находился в буферном кэше.

"**read**" означает, что блок отсутствовал в буферном кэше. Процессу приходилось очищать буфер (если список свободных блоков пуст), загружать в очищенный буфер блок и только потом читать его содержимое.

Представление pg_stat_all_tables

- содержит детальную статистику по таблицам и индексам:

```
 select schemaname||'.'||relname name, seq_scan, idx_scan, idx_tup_fetch, autovacuum_count, autoanalyze_count from
pg_stat_all_tables where idx_scan is not null order by 3 desc limit 1;
          name          | seq_scan | idx_scan | idx_tup_fetch | autovacuum_count | autoanalyze_count
------------------------+----------+----------+---------------+------------------+-------------------
 public.pgbench_accounts |        0 | 11183162 |      11183162 |             1512 |               266
  select relname name, n_tup_ins ins, n_tup_upd upd, n_tup_del del, n_tup_hot_upd hot_upd, n_tup_newpage_upd
newblock, n_live_tup live, n_dead_tup dead, n_ins_since_vacuum sv, n_mod_since_analyze sa from pg_stat_all_tables
where idx_scan is not null order by 3 desc limit 1;
       name       | ins |   upd   | del | hot_upd | newblock |  live  |  dead   | sv |  sa
------------------+-----+---------+-----+---------+----------+--------+---------+----+------
 pgbench_tellers  |   0 | 5598056 |   0 | 5497197 |   100859 |     10 | 1456051 |  0 |  165
```

- пример теста pgbench если горизонт удерживался и перестал удерживаться:

```
progress: 92100.0 s,   9.4 tps, lat 105.960 ms stddev 4.277, 0 failed
progress: 92200.0 s, 556.1 tps, lat   1.796 ms stddev 2.213, 0 failed
       name       | ins |   upd   | del | hot_upd | newblock |  live  | dead | iv |  ma
------------------+-----+---------+-----+---------+----------+--------+------+----+------
 pgbench_tellers  |   0 | 5762566 |   0 | 5656271 |   106295 |     10 |  124 |  0 | 1201
```

- представление pg_stat_xact_all_tables имеет те же столбцы, что и pg_stat_all_tables, но показывает только действия, выполненные в текущей транзакции и ещё не попавшие в pg_stat_all_*. Столбцы для **n_live_tup, n_dead_tup** и относящиеся к вакуумированию и анализу отсутствуют

Представление pg_stat_all_tables

Детальная статистика по таблицам:

```
 select schemaname||'.'||relname name, seq_scan, idx_scan, idx_tup_fetch, autovacuum_count, autoanalyze_count
from pg_stat_all_tables where idx_scan is not null order by 3 desc limit 3;
          name           | seq_scan | idx_scan | idx_tup_fetch | autovacuum_count | autoanalyze_count
-------------------------+----------+----------+---------------+------------------+-------------------
 public.pgbench_accounts |        0 | 11183162 |      11183162 |             1512 |               266
 public.pgbench_tellers  |   906731 |  4684850 |       4684850 |             1524 |              1536
 public.pgbench_branches |   907256 |  4684327 |       4684327 |             1527 |              1536
 select relname name, n_tup_ins ins, n_tup_upd upd, n_tup_del del, n_tup_hot_upd hot_upd, n_tup_newpage_upd
newblock, n_live_tup live, n_dead_tup dead, n_ins_since_vacuum sv, n_mod_since_analyze sa from pg_stat_all_tables
where idx_scan is not null order by 3 desc limit 3;
       name       | ins |   upd   | del | hot_upd | newblock |  live  |  dead   | sv |  sa
------------------+-----+---------+-----+---------+----------+--------+---------+----+------
 pgbench_tellers  |   0 | 5598056 |   0 | 5497197 |   100859 |     10 | 1456051 |  0 |  165
 pgbench_branches |   0 | 5598056 |   0 | 5589787 |     8269 |      1 | 1456044 |  0 |  175
 pgbench_accounts |   0 | 5598056 |   0 | 3923068 |  1674988 | 100001 | 1456032 |  0 | 7619
```

Пример обычного теста pgbench если удерживать горизонт, а потом сдвинуть:

после отработки вакуума число n_dead_tup уменьшится с 1456051 до минимума, а TPS возрастет с 9 до 556:

```
progress: 92100.0 s,   9.4 tps, lat 105.960 ms stddev 4.277, 0 failed
progress: 92200.0 s, 556.1 tps, lat   1.796 ms stddev 2.213, 0 failed
       name       | ins |   upd   | del | hot_upd | newblock |  live  | dead | iv |  ma
------------------+-----+---------+-----+---------+----------+--------+------+----+------
 pgbench_tellers  |   0 | 5762566 |   0 | 5656271 |   106295 |     10 |  124 |  0 | 1201
 pgbench_branches |   0 | 5762566 |   0 | 5753853 |     8713 |      1 |  185 |  0 | 1201
 pgbench_accounts |   0 | 5762566 |   0 | 4055493 |  1707073 | 100000 | 5687 |  0 | 1740
```

Статистика **n_tup_hot_upd** вакуумом не обновляется. За этим и другими накапливаемыми статистиками нужно наблюдать в динамике, например, с помощью графиков. После освобождения горизонта доля HOT update возросла с 14% до 54%.

Представление pg_stat_xact_all_tables имеет те же столбцы, что и pg_stat_all_tables, но показывает только действия, выполненные в текущей транзакции к настоящему моменту и ещё не попавшие в pg_stat_all_*. Столбцы для **n_live_tup, n_dead_tup** и относящиеся к вакуумированию и анализу в этих представлениях отсутствуют:

```
 select schemaname||'.'||relname name, seq_scan, idx_scan, idx_tup_fetch, n_tup_ins ins, n_tup_upd upd, n_tup_del
del, n_tup_hot_upd hot_upd, n_tup_newpage_upd newblock from pg_stat_xact_all_tables where idx_scan is not null
order by 3 desc limit 3;
          name           | seq_scan | idx_scan | idx_tup_fetch | ins | upd | del | hot_upd | newblock
-------------------------+----------+----------+---------------+-----+-----+-----+---------+----------
 pg_catalog.pg_namespace |        1 |        2 |             0 |   0 |   0 |   0 |       0 |        0
```

Представление pg_stat_all_indexes

- содержат детальную статистику по таблицам и индексам:

```
select schemaname||'.'||relname name, indexrelname, idx_scan, round(extract('epoch' from clock_timestamp() -
 last_idx_scan)) scan_sec, idx_tup_read, idx_tup_fetch from pg_stat_all_indexes order by 3 desc limit 3;
          name           |     indexrelname      | idx_scan | scan_sec | idx_tup_read | idx_tup_fetch
-------------------------+-----------------------+----------+----------+--------------+---------------
 public.pgbench_accounts | pgbench_accounts_pkey |  7080442 |    19489 |     20471311 |       7080442
 public.pgbench_tellers  | pgbench_tellers_pkey  |  2799175 |    19489 |   3740278460 |        124998
 public.pgbench_branches | pgbench_branches_pkey |  2798720 |    19489 |   3049729576 |       2798720
```

- idx_scan число сканирований этого индекса
- last_idx_scan время последнего сканирования этого индекса
- idx_tup_read число индексных записей, возвращённых при сканированиях этого индекса
- idx_tup_fetch актуальных строк таблицы, выбранных с помощью при простом сканировании этого индекса (Index Scan)

Представление pg_stat_all_indexes

Детальная статистика по индексам:

```
select schemaname||'.'||relname name, indexrelname, idx_scan, round(extract('epoch' from clock_timestamp() -
last_idx_scan)) scan_sec, idx_tup_read, idx_tup_fetch from pg_stat_all_indexes order by 3 desc limit 3;
          name           |     indexrelname      | idx_scan | scan_sec | idx_tup_read | idx_tup_fetch
-------------------------+-----------------------+----------+----------+--------------+---------------
 public.pgbench_accounts | pgbench_accounts_pkey |  7080442 |    19489 |     20471311 |       7080442
 public.pgbench_tellers  | pgbench_tellers_pkey  |  2799175 |    19489 |   3740278460 |        124998
 public.pgbench_branches | pgbench_branches_pkey |  2798720 |    19489 |   3049729576 |       2798720
```

idx_scan число сканирований этого индекса. Помогает определить, какие индексы используются часто, а какие редко или никогда не используются.

last_idx_scan время последнего сканирования этого индекса. Столбец появился в 16 версии PostgreSQL. Отображаем время окончания транзакции, в которой использовался индекс, а не точный момент использования индекса (который может значительно отличаться, если это длительная транзакция). Периодически запрашивая время последнего использования и сравнивая с idx_scan можно определить редко используемые индексы, чтобы их удалить.

Неиспользуемые индексы занимают место, сравнимое с размером таблицы; замедляют операции DML; мешают HOT update.

idx_tup_read число индексных записей, возвращённых при сканированиях этого индекса. Значение увеличивается всякий раз, когда считывается индексная запись.

idx_tup_fetch число актуальных строк таблицы, выбранных с помощью индексной записи при простом сканировании этого индекса (Index Scan).

Индексы могут использоваться при Index Scan, при Bitmap Index Scan и планировщиком. Битовые карты, построенные по нескольким индексам могут объединяться операциями AND и OR. При Bitmap Index Scan увеличиваются счётчики pg_stat_all_indexes.idx_tup_read используемых индексов и счётчик pg_stat_all_tables.idx_tup_fetch для каждой таблицы, а pg_stat_all_indexes.idx_tup_fetch не меняется.

Значения счётчиков idx_tup_read и idx_tup_fetch могут различаться, даже если сканирование с использованием битовой карты не используется, поскольку idx_tup_read подсчитывает полученные из индекса элементы, а idx_tup_fetch — количество «живых» строк, выбранных из таблицы. Различие будет меньше, если dead или ещё не зафиксированные строки будут извлекаться с использованием индекса или если для получения строк таблицы будет использоваться Index Only Scan.

Длительность удержания горизонта баз данных

- длительность самого долгого запроса или транзакции всех баз данных кластера:

```
select datname database, max(now()-xact_start) duration, greatest(max(age(backend_xmin)),
max(age(backend_xid))) age from pg_stat_activity where backend_xmin is not null or backend_xid is not null
group by datname order by datname;
  database |    duration     |  age
----------+-----------------+-------
 postgres | 00:09:41.946193 | 79348
```

- процессы, удерживающие горизонт:

```
select now()-xact_start duration, age(backend_xmin) age_xmin, age(backend_xid) age_xid, pid, datname,
usename, state, wait_event, left(query,20) query from pg_stat_activity where backend_xmin is not null or
backend_xid is not null order by greatest(age(backend_xmin), age(backend_xid)) desc;
 duration  | age_xmin | age_xid | pid  | datname  | usename  |        state        | wait_event | query
----------+----------+---------+------+----------+----------+---------------------+------------+-------
 00:02:52 |    46732 |         | 8806 | postgres | postgres | idle in transaction | ClientRead | select
 00:00:00 |          |       1 | 8825 | postgres | postgres | active              | WALSync    | END;
 00:00:00 |        1 |         | 9049 | postgres | postgres | active              |            |
```

- можно идентифицировать запросы или транзакции, которые удерживают горизонт и оптимизировать их или перенести запросы на реплики

Длительность удержания горизонта баз данных

Горизонт критичен для производительности, так как определяет возможность очистить старые версии строк автовакуумом и HOT cleanup. Можно идентифицировать запросы или транзакции, которые удерживают горизонт и оптимизировать их или перенести запросы на реплику.

Длительность самого долгого запроса или транзакции всех баз данных кластера:

```
select datname database, max(now()-xact_start) duration,
greatest(max(age(backend_xmin)), max(age(backend_xid))) age from pg_stat_activity
where backend_xmin is not null or backend_xid is not null group by datname order
by datname;
  database |    duration     |  age
----------+-----------------+-------
 postgres | 00:09:41.946193 | 79348
```

age - количество номеров транзакций, отстоящих от текущей. Процессы, удерживающие горизонт:

```
select now()-xact_start duration, age(backend_xmin) age_xmin,
age(backend_xid) age_xid, pid, datname, usename, state, wait_event, left(query,
20) from pg_stat_activity where backend_xmin is not null or backend_xid is not
null order by greatest(age(backend_xmin), age(backend_xid)) desc;
 duration  | age_xmin | age_xid | pid  | datname  | usename  |        state        | wait_event | query
----------+----------+---------+------+----------+----------+---------------------+------------+-------
 00:02:52 |    46732 |         | 8806 | postgres | postgres | idle in transaction | ClientRead | select
 00:00:00 |          |       1 | 8825 | postgres | postgres | active              | WALSync    | END;
 00:00:00 |        1 |         | 9049 | postgres | postgres | active              |            |
```

Если запрос выполняется на мастере, то выдаются данные по процессам мастера. Если запрос выполняется на реплике, то по процессам реплики.

Представление `pg_replication_slots` содержит состояние всех слотов репликации. Столбец `xmin` содержит идентификатор старейшей транзакции, для которой должен удерживаться горизонт. Пример запроса:

```
select max(age(xmin)) from pg_replication_slots;
```

Представление `pg_stat_replication` на мастере содержит по одной строке для каждого `walsender`. Столбец `backend_xmin` содержит идентификатор старейшей транзакции ("xmin") реплики, если включена обратная связь (`hot_standby_feedback=on`). Пример запроса:

```
select age(backend_xmin) age_xmin, application_name from pg_stat_replication order by
age(backend_xmin) desc;
```

Представление pg_stat_wal

- в представлении одна строка
- wal_records сколько журнальных записей записано в WAL-файлы
- wal_fpi сколько записано полных образов страниц (full page images)
- wal_buffers_full сколько раз вызывалась запись в журналы из-за заполнения журнального буфера
- журнальные буфера вытесняются в страничный кэш linux функцией XLogWrite
- если wal_sync_method=fdatasync, fsync или fsync_writethrough, то функция XLogWrite вызывает функцию issue_xlog_fsync для сброса блоков, относящихся к файлам журнала, из страничного кэша на диск
- Столбцы wal_write_time и wal_sync_time имеют значения отличные от нуля только, если параметр track_wal_io_timing=on

```
select * from pg_stat_wal;
 wal_records | wal_fpi |  wal_bytes  | wal_buffers_full | wal_write | wal_sync |...
-------------+---------+-------------+------------------+-----------+----------+
    33373424 | 1158848 | 10892483009 |                0 |   5736124 |  5730955 |
```


Представление pg_stat_wal

В представлении одна строка.
wal_records сколько журнальных записей записано в WAL-файлы.
wal_fpi сколько полных образов страниц (full page images) записано в WAL-файлы.
wal_buffers_full сколько раз вызывалась запись в файлы журнала из-за заполнения журнального буфера.
wal_write сколько раз блоки из журнального буфера были вытеснены в страничный кэш функцией XLogWrite
wal_sync (и wal_sync_time - затраченное время) сколько раз журнальные записи сбрасывались на диск функцией issue_xlog_fsync, если параметр fsync=on и wal_sync_method имеет значение fdatasync, fsync (и fsync_writethrough, который не используется в linux). Если параметр wal_sync_method=open_datasync или open_sync, то issue_xlog_fsync не используется и в столбце wal_sync значение ноль.
Журнальные буфера вытесняются в страничный кэш linux функцией XLogWrite, которую вызывают:
1) функция XLogInsertRecord когда в буферах WAL нет места для новых записей
2) функция XLogFlush
3) процесс walwriter
Если wal_sync_method=fdatasync, fsync или fsync_writethrough, функция XLogWrite вызывает функцию issue_xlog_fsync для сброса блоков, относящихся к файлам журнала, из страничного кэша на диск.
Столбцы wal_write_time и wal_sync_time имеют значения отличные от нуля только, если параметр track_wal_io_timing=on.

Расширение pg_walinspect

- альтернатива утилите командной строки pg_waldump
- pg_get_wal_stats(lsn1, lsn2, true) выдаёт статистику по журнальным записям
  - Если третий параметр true, то выдаётся статистика ещё и по record_type, а не суммарная по каждому resource_manager
- в примере полные образы блоков (fpi_size) занимают: 83% для таблиц и 76% для индексов. По тесту pgbench tps=14.

```
 select "resource_manager/record_type" type, count, round(count_percentage) "count%", record_size,
round(record_size_percentage) "size%", fpi_size, round(fpi_size_percentage(tantor)) "fpi%", combined_size,
round(combined_size_percentage) "total%" from pg_get_wal_stats((select '0/0'::pg_lsn +
(pg_split_walfile_name(name(oleg))).segment_number * size from pg_ls_waldir() oleg order by name limit 1),
'FFFFFFFF/FFFFFFFF', true) tantor where count<>0 order by 8 desc;
          type          | count | count% | record_size | size% | fpi_size | fpi% | combined_size | total%
------------------------+-------+--------+-------------+-------+----------+------+---------------+--------
 Heap/LOCK              | 16223 |     15 |     1003405 |    11 | 48582684 |   81 |      49586089 |     71
 Btree/INSERT_LEAF      | 16224 |     15 |     1049301 |    11 | 10692280 |   18 |      11741581 |     17
 Heap/UPDATE            | 15822 |     14 |     2865744 |    31 |    12620 |    0 |       2878364 |      4
 Heap/HOT_UPDATE        | 29779 |     27 |     2560740 |    27 |     5432 |    0 |       2566172 |      4
 Heap/INSERT            | 15215 |     14 |     1232365 |    13 |     7620 |    0 |       1239985 |      2
 XLOG/FPI_FOR_HINT      |    89 |      0 |        4539 |     0 |   719100 |    1 |        723639 |      1
 Transaction/COMMIT     | 15331 |     14 |      553376 |     6 |        0 |    0 |        553376 |      1
...
```


Расширение pg_walinspect

Стандартное расширение для просмотра журнальных записей. Появилось в 15 версии PostgreSQL. Работает с запущенным экземпляром. Выдаёт данные по текущей линии времени. Функционал похож на утилиту командной строки pg_waldump, но работает через SQL. Доступно суперпользователям и роли pg_read_server_files. Установка расширения:

```
create extension pg_walinspect;
```

Расширение состоит из четырёх функций. Их список можно посмотреть командой:

```
\dx+ pg_walinspect
function pg_get_wal_block_info(pg_lsn,pg_lsn,boolean)
function pg_get_wal_record_info(pg_lsn)
function pg_get_wal_records_info(pg_lsn,pg_lsn)
function pg_get_wal_stats(pg_lsn,pg_lsn,boolean)
```

Функция pg_get_wal_stats(lsn1,lsn2, true) показывает статистику по журнальным записям между двумя LSN.

Если третий параметр true, то выдаётся статистика ещё и по record_type, а не суммарная по каждому resource_manager. Второй LSN можно указать максимальным. Пример теста pgbench с удержанием горизонта:

```
select "resource_manager/record_type" type, count, round(count_percentage)
"count%", record_size, round(record_size_percentage) "size%", fpi_size,
round(fpi_size_percentage(tantor)) "fpi%", combined_size,
round(combined_size_percentage) "total%" from pg_get_wal_stats((select
'0/0'::pg_lsn + (pg_split_walfile_name(name(oleg))).segment_number * size from
pg_ls_waldir() oleg order by name limit 1), 'FFFFFFFF/FFFFFFFF', true) tantor
where count<>0 order by 8 desc;
        type       | count | count% | record_size | size% | fpi_size | fpi% | combined_size | total%
-------------------+-------+--------+-------------+-------+----------+------+---------------+--------
 Heap/LOCK         | 16223 |     15 |     1003405 |    11 | 48582684 |   81 |      49586089 |     71
 Btree/INSERT_LEAF | 16224 |     15 |     1049301 |    11 | 10692280 |   18 |      11741581 |     17
 Heap/UPDATE       | 15822 |     14 |     2865744 |    31 |    12620 |    0 |       2878364 |      4
...
```

Полные образы блоков (fpi_size) занимают существенное место: 69668516/83897623= 83% для таблиц и 9634496/12734954=76% для индексов.

Список resource_manager можно посмотреть запросом:

```
select * from pg_get_wal_resource_managers();
```

Использование расширения `pg_walinspect`

- если горизонт не удерживается, то доля HOT становится больше:

```
          type          |   count | count% | record_size | size% | fpi_size | fpi% | combined_size | total%
------------------------+---------+--------+-------------+-------+----------+------+---------------+--------
 Heap/HOT_UPDATE        | 1125325 |     54 |    96690453 |    62 | 28860128 |   47 |     125550581 |     58
 Heap/INSERT            |  372681 |     18 |    30187111 |    19 |     8764 |    0 |      30195875 |     14
 Heap2/PRUNE            |  131095 |      6 |    11757645 |     8 |  4885592 |    8 |      16643237 |      8
 Heap2/VISIBLE          |   58699 |      3 |     3822780 |     2 | 11975568 |   20 |      15798348 |      7
 XLOG/FPI_FOR_HINT      |    1929 |      0 |       98379 |     0 | 15082820 |   25 |      15181199 |      7
 Transaction/COMMIT     |  375127 |     18 |    13509524 |     9 |        0 |    0 |      13509524 |      6
...
```

- TPS увеличиваются с 14 до 660
- чем ближе горизонт к текущему моменту и чем меньше FILLFACTOR тем больше вероятность того что доля HOT увеличится
- даёт ли уменьшение этих значений эффект можно запросом обратив внимание на долю Heap/HOT_UPDATE
- HOT возможен если:
  - изменения не затрагивают проиндексированных столбцов
  - новая версия строки должна поместиться в тот же блок, что и обновляемая версия строки

Использование расширения `pg_walinspect`

В предыдущем примере TPS небольшой:

```
progress: 36590.0 s, 13.9 tps, lat 71.833 ms stddev 5.460, 0 failed
```

Если горизонт не удерживается, то доля HOT становится больше:

```
          type          |   count | count% | record_size | size% | fpi_size | fpi% | combined_size | total%
------------------------+---------+--------+-------------+-------+----------+------+---------------+--------
 Heap/HOT_UPDATE        | 1125325 |     54 |    96690453 |    62 | 28860128 |   47 |     125550581 |     58
 Heap/INSERT            |  372681 |     18 |    30187111 |    19 |     8764 |    0 |      30195875 |     14
 Heap2/PRUNE            |  131095 |      6 |    11757645 |     8 |  4885592 |    8 |      16643237 |      8
 Heap2/VISIBLE          |   58699 |      3 |     3822780 |     2 | 11975568 |   20 |      15798348 |      7
 XLOG/FPI_FOR_HINT      |    1929 |      0 |       98379 |     0 | 15082820 |   25 |      15181199 |      7
 Transaction/COMMIT     |  375127 |     18 |    13509524 |     9 |        0 |    0 |      13509524 |      6
...
```

TPS увеличиваются с 14 до 660:

```
progress: 38320.0 s, 660.6 tps, lat 1.511 ms stddev 0.360, 0 failed
```

HOT возможен если:

Индексированные столбцы не обновляются, включая столбцы в частичных индексах.

Новая версия строки должна поместиться в тот же блок, что и обновляемая версия строки. По умолчанию FILLFACTOR=100%, быстрая очистка сработает если предыдущее обновление любой строки в блоке поместилось в блок и в блоке осталось меньше 10%. В эти 10% должна поместиться новая версия строки. Если предыдущая версия любой строки в блоке вышла за горизонт, то ее место будет очищено и занято новой версией.

Если размер строк в таблице меньше 5-10% размера блока, то уменьшение FILLFACTOR не увеличит процент HOT. Если размер строки больше 10% размера блока,

то уменьшение FILLFACTOR может увеличить долю HOT.

Вывод: чем ближе горизонт к текущему моменту и чем меньше FILLFACTOR тем больше вероятность того что доля HOT увеличится. Даёт ли уменьшение этих значений эффект можно запросом обратив внимание на долю Heap/HOT_UPDATE.

Обновление индексов при обновлении строк таблицы называют "index write amplification". Он не связан с "WAL write amplification".

https://www.adyen.com/knowledge-hub/postgresql-hot-updates

Представление `pg_stat_activity`

- по одной строке для каждого процесса экземпляра
- строки отражают чем занят процесс в момент обращения к строке
- `backend_type` - название процесса
- state текущее состояние процесса
- `wait_type='Activity'`, это нормальное состояние процесса, он не заблокирован, простаивает и ждет пробуждения в своём основном (Main) цикле
- `wait_type='Lock'`, то процесс ожидает получения блокировки

```
select left(backend_type, 14) type, left(state,6) state, wait_event_type wait_type, wait_event,
age(backend_xid) age_xid, age(backend_xmin) age_xmin, round(extract('epoch' from clock_timestamp()
xact_start)) sec, substring(query,0, 20) query from pg_stat_activity;
      type      | state  | wait_type |      wait_event      | age_xid | age_xmin | sec  |        query
----------------+--------+-----------+----------------------+---------+----------+------+--------------------
autovacuum lau |        | Activity  | AutoVacuumMain       |         |          |      |
pg_wait_sampli |        | Extension | Extension            |         |          |      |
client backend | idle i | Client    | ClientRead           |   92700 |          | 1010 | lock table t;
client backend | active | Lock      | relation             |         |    92700 |  967 | select * from t lim
client backend | active |           |                      |       1 |    92700 |    0 | UPDATE pgbench_bran
...
```


Представление `pg_stat_activity`

В представлении pg_stat_activity для каждого процесса экземпляра будет присутствовать по одной строке с информацией о том, что делает процесс в момент обращения к представлению. Возможно стоит отфильтровать серверные процессы, которые подключены к не текущей базе данных: `datname = current_database()`

Столбец backend_type содержит название процесса. Пример: client backend, autovacuum worker, checkpointer, background writer, startup, walreceiver, walsender.

Столбцы client_addr, client_hostname, client_port, application_name, usename (имя пользователя в сессии), usesysid (OID пользователя) позволяют идентифицировать клиента, чья сессия обслуживается серверным процессом.

state текущее состояние процесса. Значения:

active: серверный процесс выполняет команду. Пиковые значения (spike) в какие-то времена указывает на синхронную логику в приложении. Если wait_event не пусто, это значит что процесс чем-то заблокирован и ожидает снятия блокировки.

idle: серверный процесс не в транзакции и ожидает команду. Уменьшение по сравнению с обычным значением может указывать на нехватку соединений в пуле и длинному ожиданию запуска новых серверных процессов для расширения пула соединений. Монотонное увеличение может указывать на утечку соединений в пуле соединений клиента

idle in transaction: серверный процесс открыл транзакцию и простаивает. Долгих простоев в открытых транзакциях в логике приложения не должно быть.

idle in transaction (aborted): транзакция в состоянии сбоя.

fastpath function call: серверный процесс вызвал функцию через интерфейс "быстрого пути". Это устаревший способ вызова функций клиентами, использующими библиотеку libpq. Вызов процедур, групповых, оконных функций этим способом не поддерживается. Вместо этого способа вызова рекомендуется использовать подготовленные команды.

disabled: параметр `track_activities=off`

Если wait_type=Activity, это нормальное состояние процесса, он не заблокирован, простаивает и ждет пробуждения в своём основном (Main) цикле

Если wait_type=Lock, то процесс ожидает получения блокировки.

Пример запроса:

```
select left(backend_type, 14) type, left(state,6) state, wait_event_type wait_type, wait_event, age(backend_xid)
age_xid, age(backend_xmin) age_xmin, round(extract('epoch' from clock_timestamp() - xact_start)) sec,
left(query,20) query from pg_stat_activity;
```

Блокирующие процессы и функция `pg_blocking_pids()`

- если pg_stat_activity.wait_type='Lock' или pg_locks.granted='f', то процесс заблокирован, то есть ожидает получения блокировки
- `pg_blocking_pids(pid)` показывает процессы-блокировщики
  - функция на короткое время получает легковесную блокировку на все секции структуры блокировок, поэтому не стоит ее вызывать часто

```
 select a.pid blocked, bl.pid blocker, a.wait_event_type wait, a.wait_event event, age(a.backend_xmin) age,
round(extract('epoch' from clock_timestamp() - a.query_start)) waitsec, bl.wait_event_type bl_type,
bl.wait_event bl_wait, round(extract('epoch' from clock_timestamp() - bl.query_start)) bl_sec,
left(a.query,10) blocked_q, left(bl.query,10) blocker_q from pg_stat_activity a join pg_stat_activity bl on
bl.pid = ANY(pg_blocking_pids(a.pid)) where a.wait_event_type='Lock' and
cardinality(pg_blocking_pids(a.pid))>0;
 blocked|blocker|wait| event   |age|waitsec|bl_type| bl_wait   |bl_sec|    blocked_q   | blocker_q
--------+-------+----+---------+---+-------+-------+-----------+------+---------------+----------
  755637| 754580|Lock|relation|  1|    815| Client|ClientRead|  1126|select count(*)| select 1;
```

- для мониторинга можно использовать представление `pg_locks`:

```
 select pl.pid blocked, pl.mode, a.wait_event_type wait, a.wait_event event, age(a.backend_xmin) age_xmin,
round(extract('epoch' from clock_timestamp() - a.query_start)) waitsec, left(a.query,40) blocked_query from
pg_locks pl left join pg_stat_activity a ON pl.pid = a.pid where pl.granted='f' and a.datname =
current_database();
 blocked|      mode       | wait |  event   | age_xmin | waitsec |      blocked_query
--------+-----------------+------+----------+----------+---------+--------------------------
   1234 | AccessShareLock | Lock | relation |   447502 |   11077 | select * from t limit 1;
```


Блокирующие процессы и функция `pg_blocking_pids()`

Если `pg_stat_activity.wait_type='Lock'`, то процесс ожидает получения блокировки. LWLock не должен появляться в `pg_stat_activity`, так как это короткоживущие (лёгкие) блокировки.

Для поиска процессов-блокировщиков используется функция `pg_blocking_pids(PID)`. Функция выдаёт массив с PID блокирующих процессов, либо пустой массив.

Процесс блокирует другой процесс, если:

1) процесс-блокировщик удерживает блокировку, конфликтующую с блокировкой, которую хочет получить блокируемый процесс

2) находится в очереди ожидания перед блокируемым.

При использовании параллельных процессов функция выдаёт PID родительского серверного процесса, а не рабочих процессов. Из-за этого в массиве могут выдаваться повторяющиеся PID. Функция на короткое время получает легковесную блокировку все секции структуры блокировок, поэтому не стоит её вызывать часто. Другими словами не заниматься "мониторингом". Для минимизации вызовов в запрос добавлено условие: `where a.wait_event_type = 'Lock' and cardinality(pg_blocking_pids(a.pid))>0`. Добавив условие `a.datname = current_database()` можно ограничить результат сессиями текущей базы. Командой `explain analyze` можно сравнить время выполнения и стоимость выполнения команды с условием и без условия.

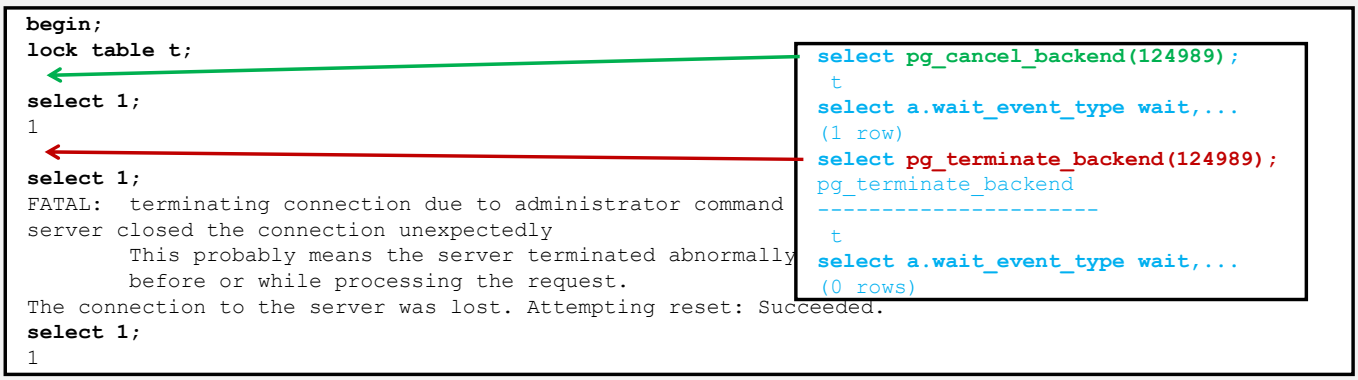
```
 select a.pid blocked, bl.pid blocker, a.wait_event_type wait, a.wait_event event,
age(a.backend_xmin) age, round(extract('epoch' from clock_timestamp() - a.query_start)) waitsec,
bl.wait_event_type bl_type, bl.wait_event bl_wait, round(extract('epoch' from clock_timestamp() -
bl.query_start)) bl_sec, left(a.query,10) blocked_q, left(bl.query,10) blocker_q from
pg_stat_activity a join pg_stat_activity bl on bl.pid = ANY(pg_blocking_pids(a.pid)) where
a.wait_event_type='Lock' and cardinality(pg_blocking_pids(a.pid))>0;
 blocked|blocker|wait| event   |age|waitsec|bl_type| bl_wait   |bl_sec|    blocked_q   | blocker_q
--------+-------+----+---------+---+-------+-------+-----------+------+---------------+----------
  755637| 754580|Lock|relation|  1|    815| Client|ClientRead|  1126|select count(*)| select 1;
```

Для мониторинга можно использовать представление `pg_locks`:

```
 select pl.pid blocked, pl.mode, a.wait_event_type wait, a.wait_event event, age(a.backend_xmin)
age_xmin, round(extract('epoch' from clock_timestamp() - a.query_start)) waitsec, left(a.query,40)
blocked_query from pg_locks pl left join pg_stat_activity a ON pl.pid = a.pid where pl.granted='f'
and a.datname = current_database();
 blocked|      mode       | wait |  event   | age_xmin | waitsec |      blocked_query
--------+-----------------+------+----------+----------+---------+--------------------------
   1234 | AccessShareLock | Lock | relation |   447502 |   11077 | select * from t limit 1;
```

pg_cancel_backend() и pg_terminate_backend()

- отменить команду в блокирующей сессии можно функцией `pg_cancel_backend(pid)`
- если сессия простаивает (`wait_event='ClientRead'`), то отменять нечего и функция бесполезна
- сессию и транзакцию в сессии можно завершить функцией `pg_terminate_backend(pid, timeout миллисекунд DEFAULT 0)`



pg_cancel_backend() и pg_terminate_backend()

Отменить команду в блокирующей (или произвольной) сессии можно функцией `pg_cancel_backend(pid)`. Отменить можно команду, которая работает. Если сессия простаивает (`wait_event='ClientRead'`), то отменять нечего и функция вернет true, но ничего не сделает.

Обычно блокировки снимаются по окончании транзакции. Независимо от того открыта транзакция или нет можно завершить сессию функцией `pg_terminate_backend(pid, timeout миллисекунд DEFAULT 0)`. Транзакция завершаемой сессии будет прервана и не зафиксируется. Если параметр timeout не задан или равен нулю, функция без задержки вернёт true, если процесс с таким PID существует. Если timeout больше нуля, функция вернёт true сразу как завершится процесс. В случае тайм-аута выдаётся предупреждение и возвращается false.

Для вызова обеих функций достаточно прав пользователя из-под которого создана сессия с командой. Также функции доступны пользователям с ролью pg_signal_backend. Применять функции к сессиям суперпользователей могут только суперпользователи.

PID процесса и свойства сессии получают из столбцов pg_stat_activity. PID соответствует номеру процесса в операционной системе.

Список функций для администрирования можно посмотреть в документации:
https://docs.tantorlabs.ru/tdb/ru/16_4/se/functions-admin.html

Практика

- Часть 1. Статистика ввода-вывода в представлении pg_stat_io
- Часть 2. Выполнение fsyncs при остановленном checkpointer
- Часть 3. Тестирование производительности ввода-вывода
- Часть 4. Выбор размера temp_buffers при работе с временными таблицами с помощью pg_stat_io
- Часть 5. Пример анализа статистик работы команды vacuum с кольцом
- Часть 6. Работа bgwriter и сопоставление статистик в представлениях pg_stat_bgwriter и pg_stat_io
- Часть 7. Работа bgwriter на буферном кэше 128Мб и 1Гб
- Часть 8. Использование расширения pg_walinspect
- Часть 9. Наблюдение за блокировками

Практика

Детально рассматриваются примеры мониторинга ввода-вывода, анализ статистик представления pg_stat_io.

В практике будет приостановлена работа процессов checkpointer и bgwriter и вы посмотрите как это отразится на работе экземпляра.

Вы сравните работу экземпляра с разным размером кэша буферов.

Будет дан пример использования расширения pg_walinspect и сравнение с утилитой pg_waldump.

15

Расширения pg_stat_statements и pg_stat_kcache

Расширение pg_stat_statements

- детальная статистика работы экземпляра с точностью до команд SQL
- для установки нужно загрузить библиотеку и установить расширение:

```
alter system set shared_preload_libraries = pg_stat_statements;
create extension pg_stat_statements;
```

- в расширение входят 3 функции и 2 представления:

```
\dx+ pg_stat_statements
  function pg_stat_statements(boolean)
  function pg_stat_statements_info()
  function pg_stat_statements_reset(oid,oid,bigint,boolean)
  view pg_stat_statements
  view pg_stat_statements_info
```

- команды объединяются в одну строку в `pg_stat_statements`, когда они выполняются одним и тем же пользователем и имеют идентичные структуры запросов, то есть семантически равнозначны, за исключением литералов и переменных подстановки (literal constants)
- Например: `select * from t where id = 'a'` и `select * from t where id = 'b'` объединятся в `select * from t where id = $1`

Расширение pg_stat_statements

Стандартное расширение. Выдает детальную статистику работы экземпляра с точностью до команд SQL. Для установки нужно загрузить библиотеку и установить расширение:

```
alter system set shared_preload_libraries = pg_stat_statements;
create extension pg_stat_statements;
```

В расширение входят 3 функции и 2 представления:

```
\dx+ pg_stat_statements
  function pg_stat_statements(boolean)
  function pg_stat_statements_info()
  function pg_stat_statements_reset(oid,oid,bigint,boolean)
  view pg_stat_statements
  view pg_stat_statements_info
```

Расширение собирает статистики выполнения команд, сгруппированные по командам.

Для группировки команд используется функционал, устанавливаемый параметром конфигурации `compute_query_id`. Значение параметра должно быть в auto (значение по умолчанию) или on.

Команды объединяются в одну команду в `pg_stat_statements`, когда они выполняются одним и тем же пользователем и имеют идентичную структуру, то есть семантически равнозначны, за исключением литералов и переменных подстановки (literal constants). Например запросы: `select * from t where id = 'a'` и `select * from t where id = 'b'` объединятся в запрос: `select * from t where id = $1`. Запросы с визуально различными текстами могут быть объединены если они семантически равнозначны. Из-за коллизии хеша могут объединиться разные команды, но вероятность этого невелика. И обратно: команды с одинаковым текстом могут считаться разными, если они получили разное дерево разбора, например, из-за разного `search_path`.

Для того, чтобы статистики были актуальны, стоит регулярно сбрасывать статистику, например раз в сутки. Можно сохранять данные за предыдущие сутки. Статистика сбрасывается вызовом функции `pg_stat_statements_reset()`.

Конфигурация pg_stat_statements

- в представлении pg_stat_statements_info два поля:
  - dealloc сколько раз в pg_stat_statements отбрасывались записи о редко выполняемых командах. Число отслеживаемых команд задаётся параметром pg_stat_statements.max, по умолчанию 5000.
  - stats_reset момент последнего сброса всей статистики расширения вызовом функции pg_stat_statements_reset() без параметров
- функцией pg_stat_statements_reset(userid, dbid, queryid, minmax_only) можно сбросить не всю, а только часть статистики:
  - по командам выполняемым пользователем, выполняемым в базе данных, по отдельному запросу

```
select pg_stat_statements_reset();
   pg_stat_statements_reset
-------------------------------
 2025-01-01 11:11:11.111111+03
(1 row)
select * from pg_stat_statements_info;
 dealloc |         stats_reset
---------+-------------------------------
       0 | 2025-01-01 11:11:11.111111+03
(1 row)
```


Конфигурация pg_stat_statements

В представлении pg_stat_statements_info одна строка и два столбца:

1) dealloc сколько раз в pg_stat_statements отбрасывались записи о редко выполняемых командах, поскольку на обработку поступало больше различных команд, чем было задано задано в параметре конфигурации расширения pg_stat_statements.max (по умолчанию 5000). Параметр pg_stat_statements.max задаёт максимальное число команд, отслеживаемых расширением, то есть, максимальное число строк в представлении pg_stat_statements.

2) stats_reset момент последнего сброса всей статистики расширения вызовом функции pg_stat_statements_reset() без задания параметров:

```
select pg_stat_statements_reset();
 2025-01-01 11:11:11.111111+03
select * from pg_stat_statements_info;
 dealloc |         stats_reset
---------+-------------------------------
       0 | 2025-01-01 11:11:11.111111+03
```

Функция возвращает время сброса в формате timestamp with time zone.

После полного сброса в представлении **pg_stat_statements_info** обновляется момент сброса.

Функцией можно сбросить не всю, а только часть статистики:
pg_stat_statements_reset(userid, dbid, queryid, minmax_only DEFAULT false)
по командам выполняемым пользователем, выполняемым в базе данных, по отдельному запросу. Для этого нужно передать OID пользователя и/или базы данных кластера и/или идентификатор запроса.

Если minmax_only=true, то сбрасываются только значения минимального и максимального времени планирования и выполнения запроса (значения столбцов min_plan_time, max_plan_time, min_exec_time, max_exec_time). По умолчанию minmax_only=false. Время последнего сброса минимальных/максимальных значений отображается в столбце minmax_stats_since представления pg_stat_statements.

Параметры конфигурации `pg_stat_statements`

- `pg_stat_statements.max` задаёт максимальное число команд, отслеживаемых расширением, то есть, максимальное число строк в представлении `pg_stat_statements`
- `pg_stat_statements.save` определяет, должна ли статистика сохраняться после перезагрузки сервера
- `pg_stat_statements.track` определяет, какие команды будут отслеживаться: top отслеживаются только команды верхнего уровня
- pg_stat_statements.track_planning устанавливает будут ли отслеживаться операции планирования и длительность фазы планирования
- `pg_stat_statements.track_utility` будут ли отслеживаться команды, отличные от `SELECT, INSERT, UPDATE, DELETE, MERGE`

```
select name, setting, context, min_val, max_val from pg_settings where name like 'pg_stat_statements%';
               name                | setting |  context   | min_val |  max_val
-----------------------------------+---------+------------+---------+------------
 pg_stat_statements.max            | 5000    | postmaster | 100     | 1073741823
 pg_stat_statements.save           | on      | sighup     |         |
 pg_stat_statements.track          | top     | superuser  |         |
 pg_stat_statements.track_planning | on      | superuser  |         |
 pg_stat_statements.track_utility  | on      | superuser  |         |
```


Параметры конфигурации `pg_stat_statements`

```
select name, setting, context, min_val, max_val from pg_settings where name like 'pg_stat_statements%';
               name                | setting |  context   | min_val |  max_val
-----------------------------------+---------+------------+---------+------------
 pg_stat_statements.max            | 5000    | postmaster | 100     | 1073741823
 pg_stat_statements.save           | on      | sighup     |         |
 pg_stat_statements.track          | top     | superuser  |         |
 pg_stat_statements.track_planning | on      | superuser  |         |
 pg_stat_statements.track_utility  | on      | superuser  |         |
```

Параметры конфигурации расширения:

`pg_stat_statements.max` задаёт максимальное число команд, отслеживаемых расширением, то есть, максимальное число строк в представлении `pg_stat_statements`. Статистика о редко выполняющихся командах обычно не нужна и увеличивать значение не нужно, так как это увеличивает объем разделяемой памяти, выделяемой расширением. Значение по умолчанию 5000.

`pg_stat_statements.save` определяет, должна ли статистика сохраняться после перезагрузки сервера. Если значение `off`, то статистика при остановке экземпляра не сохраняется. Значение по умолчанию - `on`, что означает сохранение статистики при остановке или перезапуске экземпляра.

`pg_stat_statements.track` определяет, какие команды будут отслеживаться. Принимает значения:

1) `top` (значение по умолчанию) отслеживаются только команды верхнего уровня (передаваемые клиентами в сессии)

2) `all` - в дополнение к командам верхнего уровня отслеживаются команды внутри вызываемых функций

3) `none` - сбор статистики отключён.

`pg_stat_statements.track_planning` устанавливает будут ли отслеживаться операции планирования и длительность фазы планирования. Значение `on` может привести к заметному снижению производительности, особенно когда в нескольких сессиях в одно время выполняются команды с одинаковой структурой запросов, в результате чего эти сессии пытаются одновременно изменить одни и те же строки в `pg_stat_statements`. Значение по умолчанию `off`.

`pg_stat_statements.track_utility` определяет, будет ли расширение отслеживать служебные команды. Служебными командами считаются команды, отличные от `SELECT, INSERT, UPDATE, DELETE, MERGE`. Значение по умолчанию `on`.

Представление `pg_stat_statements`

- представление заполняется функцией pg_stat_statements(true)
- в представлении **49 столбцов**, из них 10 относятся к `jit`
- 5 столбцов с временем выполнения: total_exec_time, min_exec_time, max_exec_time, mean_exec_time, stddev_exec_time
- calls - сколько раз выполнялась команда
- rows - число прочитанных или измененных строк
- 3 столбца относятся к wal: `wal_records, wal_fpi, wal_bytes`
- 6 столбцов к планам выполнения
- 6 столбцов к кэшу буферов
- 10 столбцов к временным файлам и объектам
- 4 столбца с характеристиками команды

```
select calls, round(total_exec_time) time, round(stddev_exec_time) delta, rows, plans, round(total_plan_time)
ptime, shared_blks_hit hit, shared_blks_read read, shared_blks_dirtied dirty, shared_blks_written write, wal_fpi,
wal_bytes, left(query,20) query from pg_stat_statements order by 2 desc limit 10;
 calls  | time  | delta |  rows  | plans  | ptime |   hit   | read | dirty | write | wal_fpi | wal_bytes |  query
--------+-------+-------+--------+--------+-------+---------+------+-------+-------+---------+-----------+--------
 529554 | 13478 |     0 | 529554 | 529554 | 16851 | 3302629 | 6963 | 34309 |     0 |   38447 | 117746959 | UPDATE
 529554 | 12370 |     0 | 529554 | 529554 | 14759 | 2118690 |    5 |    15 |     0 |       9 |  46780041 | UPDATE
 529554 | 12246 |     0 | 529554 | 529554 | 15316 | 3046701 |    3 |    38 |     0 |      31 |  46927982 | UPDATE
```


Представление `pg_stat_statements`

Для просмотра собранных расширением данных используется одноимённое представление. Представление заполняется одноимённой функцией pg_stat_statements(true). Если вызвать табличную функцию с аргументом `pg_stat_statements(false)`, то функция в столбце query вернет null, а не текст команды. Это позволяет уменьшить объем возвращаемых данных и может быть удобно программам мониторинга, которые подсоединяются по сети. Почему бы системе мониторинга не обратиться к представлению убрав из выборки столбец? При обращении к представлению вызывается pg_stat_statements(true), функция выбирает данные для столбца query и нагрузка на ввод-вывод выше, чем вызов pg_stat_statements(false).

В представлении **49 столбцов**, из них 10 относятся к `jit`.

2 столбца `stats_since, minmax_stats_since` момент последнего сброса статистик.

3 столбца относятся к wal: `wal_records, wal_fpi, wal_bytes` число журнальных записей, образов страниц, число байт, сгенерированных при выполнении команды.

10 столбцов к временным файлам и объектам: `local_blks_hit, local_blks_read, local_blks_dirtied, local_blks_written, local_blk_read_time, local_blk_write_time, temp_blks_read, temp_blks_written, temp_blk_read_time, temp_blk_write_time`.

6 столбцов к кэшу буферов: `shared_blks_hit, shared_blks_read, shared_blks_dirtied, shared_blks_written, shared_blk_read_time, shared_blk_write_time`.

6 столбцов к планам выполнения: `plans, total_plan_time, min_plan_time, max_plan_time, mean_plan_time, stddev_plan_time`. **Они заполняются, если `pg_stat_statements.track_planning=on`.**

5 столбцов с временем выполнения: `total_exec_time, min_exec_time, max_exec_time, mean_exec_time, stddev_exec_time`.

`calls` - сколько раз выполнялась команда

`rows` - число прочитанных или измененных строк

4 столбца с характеристиками команды: `userid` - OID пользователя, выполнявшего команду; `dbid` - OID базы в которой выполнялась команда; `toplevel=true` (если параметр `pg_stat_statements.track=top` то всегда `true`) или false; `queryid` -идентификатор команды, целое число типа bigint.

```
select calls, round(total_exec_time) time, round(stddev_exec_time) delta, rows, plans, round(total_plan_ time)
ptime, shared_blks_hit hit, shared_blks_read read, shared_blks_dirtied dirty, shared_blks_written write, wal_fpi,
wal_bytes, left(query,20) query from pg_stat_statements order by 2 desc limit 10;
```

Запросы к представлению `pg_stat_statements`

- `order by total_exec_time desc limit 10` выдаёт "Top SQL" - наиболее нагружающие экземпляр команды
- `where plans<>0` убирает вспомогательные команды типа: `BEGIN, END`
- `where rows/calls>1000 and toplevel=true` запросы, возвращающие клиенту большое число строк
- `temp_blks_written>0` может указывать на недостаток work_mem
- `100*(blk_read_time+blk_write_time)/total_exec_time` доля ввода-вывода в общем времени выполнения запроса
- `total_plan_time>total_exec_time` время на создание плана больше времени выполнения

```
 select round(total_exec_time::numeric, 2) time, calls, round(mean_exec_time::numeric, 2) mean, round((100 *
total_exec_time / sum(total_exec_time::numeric) OVER ())::numeric, 2) "%cpu", left(query,70) query from
pg_stat_statements order by total_exec_time desc limit 10;
     time    | calls  | mean  | %cpu  |                  query
-------------+--------+-------+-------+-----------------------------------------
  2773963.94 | 227303 | 12.20 | 82.86 | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2
   556235.76 | 227303 |  2.45 | 16.62 | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2
     7922.75 | 227303 |  0.03 |  0.24 | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
     3735.72 | 227302 |  0.02 |  0.11 | INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES ($1,
     3692.31 | 227303 |  0.02 |  0.11 | SELECT abalance FROM pgbench_accounts WHERE aid = $1
```


Запросы к представлению `pg_stat_statements`

Примеры как можно выбирать данные из представления.

`order by total_exec_time desc limit 10` выдаёт "Top SQL" - наиболее нагружающие экземпляр команды. Пример:

```
 select round(total_exec_time::numeric, 2) time, calls,
round(mean_exec_time::numeric, 2) mean, round((100 * total_exec_time /
sum(total_exec_time::numeric) OVER ())::numeric, 2) "%cpu", left(query,40) query
from pg_stat_statements order by total_exec_time desc limit 10;
     time    | calls  | mean  | %cpu  |                  query
-------------+--------+-------+-------+-----------------------------------------
  2690386.86 | 222903 | 12.07 | 82.87 | UPDATE pgbench_branches SET bbalance = b
   539192.95 | 222904 |  2.42 | 16.61 | UPDATE pgbench_tellers SET tbalance = tb
     7739.33 | 222904 |  0.03 |  0.24 | UPDATE pgbench_accounts SET abalance = a
     3656.78 | 222903 |  0.02 |  0.11 | INSERT INTO pgbench_history (tid, bid, a
     3617.65 | 222904 |  0.02 |  0.11 | SELECT abalance FROM pgbench_accounts WH
```

`where plans<>0` убирает вспомогательные команды типа: `BEGIN, END`

`order by stddev_exec_time desc` большой разброс по времени выполнения, возможно неоптимальные планы

`where rows/calls>1000 and toplevel=true` запросы, возвращающие клиенту большое число строк

`temp_blks_written>0` может указывать на недостаток work_mem. **Временные файлы используются при соединении хэшированием Hash Join (кроме `work_mem` влияет параметр `hash_mem_multiplier`); сортировке externl merge; материализации Materialize.**

`local_blks_*` относится к локальному кэшу, используемому для работы с временными таблицами.

В планах выполнения `explain (analyze, buffers)` статистика соответствует строке после узла плана в котором использовались временные файлы:

`Buffers: shared hit= read=, temp read= written=`

`100*(blk_read_time+blk_write_time)/total_exec_time` доля ввода-вывода в общем времени выполнения запроса

`total_plan_time>total_exec_time` время на создание плана больше времени выполнения. Нужно использовать подготовленные команды.

Пример запроса к представлению:
https://habr.com/ru/articles/805813/

Примеры запросов к представлению pg_stat_statements

- blk_read_time+blk_write_time время, затраченное на ввод-вывод
- total_exec_time-blk_read_time-blk_write_time не на ввод-вывод
- (100*(blk_read_time+blk_write_time)/total_exec_time) процент времени, затраченного на ввод-вывод
- (total_exec_time::numeric/calls) среднее время выполнения команды
- ((total_exec_time-blk_read_time-blk_write_time)::numeric/calls) среднее время не на ввод-вывод
- ((blk_read_time+blk_write_time)::numeric/calls) среднее время на ввод-вывод

```
 select (blk_read_time+blk_write_time)::numeric(20,2) io_time, (total_exec_time-blk_read_time-
blk_write_time)::numeric(20,2) non_io_time, (100*blk_read_time+blk_write_time/total_exec_time)::numeric(20,2)
"io_time%", (total_exec_time::numeric/calls)::numeric(20,2) avg_time, ((total_exec_time-blk_read_time-
blk_write_time)::numeric/calls)::numeric(20, 2) avg_non_io_time, ((blk_read_time+blk_write_time)::numeric/
calls)::numeric(20, 2) avg_io_time, left(query,20) query from pg_stat_statements where calls<>0 order by
io_time desc limit 10;
  io_time | non_io_time | io_time_percent | avg_time | avg_non_io_time | avg_io_time |        query
 ---------+-------------+-----------------+----------+-----------------+-------------+----------------------
   273.58 |       58.60 |        27343.66 |   332.19 |           58.60 |      273.58 | create extension pg_
    42.74 |      -42.64 |         4274.07 |     0.10 |          -42.64 |       42.74 | SELECT c.relname, $1
```


Примеры запросов к представлению pg_stat_statements

Примеры метрик:

(blk_read_time+blk_write_time)::numeric(20,2) время, затраченное на ввод-вывод

(total_exec_time-blk_read_time-blk_write_time)::numeric(20,2) non_io_time не на ввод-вывод

(100*blk_read_time+blk_write_time/total_exec_time)::numeric(20,2) "io_time%" процент времени, затраченного на ввод-вывод

((total_exec_time+total_plan_time)/calls)::numeric(20,2) среднее время выполнения команды, ещё называют "время отклика"

((total_exec_time-blk_read_time-blk_write_time)::numeric/calls)::numeric(20,2) avg_non_io_time

((blk_read_time+blk_write_time)::numeric/calls)::numeric(20,2) avg_io_time

Статистика total_exec_time не является полным временем выполнения команды, только фазы выполнения (execution). В полное время входит фаза планирования. Обычно фаза планирования меньше, чем выполнения. Статистики *blk_*_time включают в себя ввод-вывод как на фазе выполнения, так и на фазе планирования. В примере у запроса non_io_time отрицательное потому, что запрос планировался 38мс, выполнялся 0.5мс и время при планировании было потрачено на 37мс чтения блоков:

```
select * from pg_stat_statements where queryid=5827178464283574856\gx
-[ RECORD 1 ]---+-----------------------------------------------------
..
query           |SELECT a.attname, .. FROM pg_catalog.pg_attribute a WHERE a.attrelid = $2 AND a.attnum >
                |$3 AND NOT a.attisdropped ORDER BY a.attnum
plans           |3
total_plan_time |38.33984
min_plan_time   |0.19539
max_plan_time   |37.909368
mean_plan_time  |12.779946666666667
stddev_plan_time|17.769191620608545
calls           |3
total_exec_time |0.500417
min_exec_time   |0.12470800000000001
max_exec_time   |0.18976
mean_exec_time  |0.16680566666666669
stddev_exec_time|0.0290881765553607
rows            |147
shared_blks_hit |447
shared_blks_read|12
..
blk_read_time   |37.228767
```

Можно добавить в формулы total_plan_time, но размер запроса увеличится и результаты с отрицательными значениями более наглядны.

Примеры запросов к представлению:

```
select
  sum(calls) as calls,
  sum(rows) as rows,
  round(sum(total_exec_time)::numeric, 0) as ex_t,
  round((sum(mean_exec_time * calls) / sum(calls))::numeric, 2) as mean_ex_t,
  format('%s-%s', round(min(min_exec_time)::numeric, 2),
round(max(max_exec_time)::numeric, 2)) as minmax_ex_t,
  round(sum(total_plan_time)::numeric, 2) as pl_t,
  round((sum(mean_plan_time * calls) / sum(calls))::numeric, 2) as mean_pl_t,
  format('%s-%s', round(min(min_plan_time)::numeric,2),
round(max(max_plan_time)::numeric, 2)) as minmax_pl_t,
  left(query,10) query
from pg_stat_statements
where calls<>0 and (select datname from pg_database where oid = dbid)=current_database()
group by userid, dbid, query
order by sum(total_exec_time) desc
limit 10;
```

```
 calls   |  rows   |   ex_t   | mean_ex_t | minmax_ex_t |   pl_t   | mean_pl_t | minmax_pl_t |    left
---------+---------+----------+-----------+-------------+----------+-----------+-------------+------------
 2842766 | 2842766 | 15188975 |      5.34 | 0.00-156.50 | 87728.93 |      0.03 | 0.00-19.69  | UPDATE pgb
 2842767 | 2842767 |  3116249 |      1.10 | 0.00-68.69  | 83591.63 |      0.03 | 0.00-22.82  | UPDATE pgb
  109428 |  109428 |   486901 |      4.45 | 0.42-38.17  |  3607.42 |      0.03 | 0.00-8.80   | select cou
```

```
select
  sum(shared_blks_hit) as shared_hit,
  sum(shared_blks_read) as sh_r,
  sum(shared_blks_dirtied) as sh_d,
  sum(shared_blks_written) as sh_w,
  round(sum(blk_read_time)::numeric,2) as r_t,
  round(sum(blk_write_time)::numeric,2) as w_t,
  sum(local_blks_hit) as l_hit,
  sum(local_blks_read) as l_r,
  sum(local_blks_dirtied) as l_d,
  sum(local_blks_written) as l_w,
  sum(temp_blks_read) as t_r,
  sum(temp_blks_written) as t_w,
  (select usename from pg_user where usesysid = userid) as user,
  left(query,15) query
from pg_stat_statements
where calls<>0 and (select datname from pg_database where oid = dbid)=current_database()
group by userid, dbid, query
order by sum(total_exec_time) desc
limit 10;
```

```
shared_hit|sh_r| sh_d|sh_w|r_t | w_t |l_hit|l_r|l_d|l_w|t_r|t_w|  user  |  query
----------+----+-----+----+----+-----+-----+---+---+---+---+---+--------+--------------
1123724917| 65 |2930 |2750|0.84|99.20|   0 |  0| 0 | 0 | 0 | 0 |postgres|UPDATE pgbench_
1431243053| 176|4270 |1314|3.22|47.02|   0 |  0| 0 | 0 | 0 | 0 |postgres|UPDATE pgbench_
  15558221|  0 |   0 |  0 |0.00| 0.00|   0 |  0| 0 | 0 | 0 | 0 |postgres|select count(c)
```

Группировки **group by userid, dbid, query** используются, если дублируются команды с одинаковым текстом (разное дерево разбора или артефакты конкурентного доступа к разделяемому буферу расширения) и хочется объединить команды для упрощения анализа.

Метрики pg_stat_statements

- можно оптимизировать производительность отдельных команд используя команду **explain (analyze, verbose, settings, buffers, wal)**
- для наблюдения за выполнением команд в целом используется pg_stat_statements, события ожидания pg_stat_activity, диагностический журнал кластера
- ограничения pg_stat_activity:
  - не выдает информации о выполняющихся командах
  - не отслеживаются неудачно завершившиеся команды, например, прекратившие выполняться по statement_timeout
- можно вычислить три метрики для каждой кумулятивной статистики:
  - вычислить разницу между значениями кумулятивных статистик и разделить на интервал времени: ΔM/Δt, то есть вычислить производную по времени.
  - делить не на интервал времени, а на разницу в столбце calls (сколько раз выполнялась команда): ΔM/Δc.
  - Долю (процент), которую занимает показатель относительно всех вызовов M/sum(M)=%M

Метрики pg_stat_statements

Можно оптимизировать производительность отдельных команд (микрооптимизация) используя команду explain (analyze, verbose, settings, buffers, wal).

Время выполнения, измеренное командой EXPLAIN ANALYZE, может значительно отличаться от времени выполнения того же запроса в обычном режиме. Результаты EXPLAIN не следует распространять на ситуации, значительно отличающиеся от тех, в которых вы проводите тестирование. Оценки стоимости нелинейны и планировщик может выбирать разные планы в зависимости от размера таблицы.

Команда explain (generic_plan) позволяет использовать в команде символы $1, которые присутствуют в столбце pg_stat_statements.query.

Для наблюдения за выполнением команд в целом используется pg_stat_statements, события ожидания pg_stat_activity, диагностический журнал кластера.

Ограничения pg_stat_activity: не выдает информации о выполняющихся командах. Также не отслеживаются неудачно завершившиеся команды, например, прекратившие выполняться по statement_timeout. До истечения таймаута команды активно используют ресурсы хоста. Такие команды отслеживаются в диагностическом журнале. Если в команде есть комментарии (их может использовать расширение pg_hint_plan), то они игнорируются при классификации, в query сохраняется только один из комментариев.

Часть показателей (статистик) в представлении кумулятивные (накопительные), то есть значение возрастает со временем или с числом выполненных команд. Например, total_exec_time кумулятивная, а min_exec_time некумулятивная статистика. Абсолютные значения таких показателей не имеют смысла, их нужно сравнивать ещё с каким-то показателем (нормировать). С чем сравнивать (как нормировать)?

Нужно сделать две выборки из представления с каким-то интервалом. Например, через 10 минут или час. Метрикой называется производная от статистики. Получение производной называется "дифференциальный анализ".

Можно вычислить три метрики для каждой кумулятивной статистики:

1) Вычислить разницу между значениями кумулятивных статистик и разделить на интервал времени: ΔM/Δt, то есть вычислить производную по времени.

2) Поделить не на интервал времени, а на разницу в столбце calls (сколько раз выполнялась команда): ΔM/Δc.

3) Процент (долю) которую занимает показатель относительно всех вызовов M/sum(M)=%M.

Примеры метрик представления pg_stat_statements

- Δcalls/Δt - это метрика "QPS" - число вызовов в секунду
- Δ(total_plan_time+total_exec_time)/Δcalls - скользящее среднее время выполнения запроса
- Δrows/Δt - количество строк, переданных клиентам
- Δ(shared_blks_hit+shared_blks_read)*8192/Δt - объем обрабатываемых данных, нагружающих шину процессор-память
- Δ(shared_blks_hit+shared_blks_read)/Δcalls - большие значения указывают на то, что запрос имеет потенциал для оптимизации
- Δwal_bytes/Δt - команды генерирующие много WAL
- Δwal_bytes/Δcalls - объем WAL, сгенерированных командой
- %time - доля запроса среди всех:

```
select calls, round(total_exec_time::numeric, 2) total_time, round(mean_exec_time::numeric, 2) mean_time,
round((100 * total_exec_time / sum(total_exec_time) OVER ())::numeric, 2) "%time", left(query,40) query
from pg_stat_statements order by total_exec_time desc limit 10;
   calls  | total_time  | mean_time | %time |                  query
----------+-------------+-----------+-------+-----------------------------------------
  2842766 | 15188975.46 |      5.34 | 75.80 | UPDATE pgbench_branches SET bbalance = b
  2842767 |  3116248.96 |      1.10 | 15.55 | UPDATE pgbench_tellers SET tbalance = tb
```


Примеры метрик представления pg_stat_statements

Каждая метрика: ΔM/Δt, ΔM/Δc, %M обладает собственной ценностью. Если периодически делать выборки из представления, то можно построить график изменения каждой метрики.

Метрики можно рассчитывать не для отдельных query (обычно с наибольшим total_exec_time), а по группам query. Например, запросы от одного пользователя (роли из под которой работает приложение).

Примеры метрик:

1) Δcalls/Δ(t) - это метрика "QPS" - число вызовов в секунду. Δ(t) - интервал времени в скундах между двумя запросами к представлению.

2) Δ(total_plan_time + total_exec_time)/Δt - доля времени, затраченная на обслуживание запросов

Δ(total_plan_time + total_exec_time)/Δcalls - скользящее среднее время выполнения запроса. Аналог mean_exec_time, которое является усреднением за весь период от сброса статистики. Появление существенного (на порядок) отклонение метрики от mean_exec_time указывает на появление узкого места в этот период времени.

3) Δrows/Δt - количество строк, переданных клиентам. Возврат слишком большого числа строк может потребовать значительных ресурсов на стороне клиента. Большое значение может указывать на то, что узким местом является клиент. Всплески (spikes) в графике метрики может указать на то, что клиент посылает запросы синхронно и стоит вставить задержки чтобы разнести запросы во времени ("фронтэнд флудит").

4) Δ(shared_blks_hit+shared_blks_read)*8192/Δt - объем обрабатываемых данных, нагружающих шину процессор-память. Пропускная способность шины памяти ограничена диапазоном 2-20Гб/с. При больших значениях метрики стоит оптимизировать запросы, по которым рассчитывалась метрика или перенести их на реплики.

Δ(shared_blks_hit+shared_blks_read)/Δcalls - большие значения указывают на то, что запрос имеет потенциал для оптимизации

5) Δwal_bytes/Δt - команды (query) генерирующие много журнальных записей. Большой объем журналов создает нагрузку на физическую и логическую репликацию и может увеличить отставание реплик (replication lag).

6) Δwal_bytesΔcalls - объем журнальных записей, сгенерированных командой.

7) round((100*total_exec_time/sum(total_exec_time) over ())::numeric,2) - доля запроса среди всех отслеживаемых (до 5000 по умолчанию) запросов.

Расширение pg_stat_kcache

- дополняет pg_stat_statements и зависит от него
- собирает статистику linux, выполняя системный вызов getrusage после выполнении каждой команды
- в отличие от утилит операционной системы, расширение собирает статистики с детальностью до команды
- позволяет различать читался блок с диска или из страничного кэша
- использует два буфера в разделяемой памяти

```
 select * from (select *,lead(off) over(order by off)-off as diff from pg_shmem_allocations) as a where name
like 'pg_%';
           name            |    off    | size  | allocated_size |  diff
---------------------------+-----------+-------+----------------+---------
 pg_stat_statements        | 148162816 |    64 |            128 |     128
 pg_stat_statements hash   | 148162944 |  2896 |           2944 | 2188544
 pg_stat_kcache            | 150351488 |   992 |           1024 |    1024
 pg_stat_kcache hash       | 150352512 |  2896 |           2944 | 1373056
select name, setting, context, min_val, max_val from pg_settings where name like '%kcache%';
              name               | setting |  context   | min_val |  max_val
---------------------------------+---------+------------+---------+------------
 pg_stat_kcache.linux_hz         | 333333  | user       | -1      | 2147483647
 pg_stat_kcache.track            | top     | superuser |         |
 pg_stat_kcache.track_planning | off     | superuser |         |
```


Расширение pg_stat_kcache

Расширение дополняет pg_stat_statements и зависит от него. Отсутствует в стандартной поставке. Расширение работает стабильно и накладные расходы незначительны. Статистика shared_blks_read не различает находились ли страницы (размер 4Кб) из которых состоит блок (размер 8Кб) в страничном кэше linux или читались с диска. Расширение позволяет это различать, оно собирает статистику linux, выполняя системный вызов getrusage после выполнении каждой команды. Статистики, собираемые расширением могут быть полезны для определения эффективности кэширования и возможных узких мест. Данные, собираемые системным вызовом записываются в разделяемую память.

Вызов getrusage также используется параметром конфигурации log_executor_stats=on (по умолчанию отключён). Этот параметр конфигурации сохраняет собранную статистику операционной системы в диагностический лог кластера, что менее удобно для просмотра и необходимости следить за размером лога.

В отличие от утилит операционной системы, расширение собирает статистики с детальностью до команды. Число команд, по которым собирается статистика и размер структур разделяемой памяти определяется параметром pg_stat_statements.max (по умолчанию 5000), так как это расширение зависит от расширения pg_stat_statements.

Расширение использует два буфера в разделяемой памяти:

select * from (select *,lead(off) over(order by off)-off as diff from pg_shmem_allocations) as a where name like 'pg_%';

name	off	size	allocated_size	diff
pg_stat_statements	148162816	64	128	128
pg_stat_statements hash	148162944	2896	2944	2188544
pg_stat_kcache	150351488	992	1024	**1024**
pg_stat_kcache hash	150352512	2896	2944	**1373056**

Расширение имеет параметры:

\dconfig *kcache*

pg_stat_kcache.linux_hz (по умолчанию -1) устанавливается автоматически в значение параметра linux CONFIG_HZ и используется для компенсации ошибок сэмплирования. Менять не нужно.

pg_stat_kcache.track=top параметр - аналог pg_stat_statements.track

pg_stat_kcache.track_planning=off аналог pg_stat_statements.track_planning

Статистики, собираемые pg_stat_kcache

Статистики в представлениях pg_stat_kcache и pg_stat_kcache_detail:

- reads_blks reads, in 8K-blocks
- writes_blks writes, in 8K-blocks
- user_time user CPU time used
- system_time system CPU time used
- minflts page reclaims (soft page faults)
- majflts page faults (hard page faults)
- nswaps swaps
- msgsnds IPC messages sent
- msgrcvs IPC messages received
- nsignals signals received
- nvcsws voluntary context switches
- nivcsws involuntary context switches

```
alter system set shared_preload_libraries = pg_stat_statements, pg_wait_sampling, pg_stat_kcache;
create extension pg_stat_kcache;
\dx+ pg_stat_kcache
 function pg_stat_kcache()
 function pg_stat_kcache_reset()
 view pg_stat_kcache
 view pg_stat_kcache_detail
```


Статистики, собираемые pg_stat_kcache

Команды для установки расширения:

```
apt install clang-13
wget https://github.com/powa-team/pg_stat_kcache/archive/REL2_3_0.tar.gz
tar xzf ./REL2_3_0.tar.gz
cd pg_stat_kcache-REL2_3_0
make
make install
alter system set shared_preload_libraries = pg_stat_statements, pg_wait_sampling, pg_stat_kcache;
sudo systemctl restart  tantor-se-server-16.service
create extension pg_stat_kcache;
```

Расширение состоит из двух представлений и двух функций:

```
\dx+ pg_stat_kcache
 function pg_stat_kcache()
 function pg_stat_kcache_reset()
 view pg_stat_kcache
 view pg_stat_kcache_detail
```

Представление pg_stat_kcache_detail имеет столбцы: query, top, rolname и выдает данные с точностью до команды. Статистики выдаются из 14 столбцов для планирования и 14 столбцов для выполнения команд.

Представление pg_stat_kcache содержит суммарные статистики из pg_stat_kcache_detail, сгруппированные по базам данных:

```
CREATE VIEW pg_stat_kcache AS SELECT datname, SUM(столбцы) FROM pg_stat_kcache_detail WHERE top IS TRUE GROUP BY datname;
```

Статистики в обоих представлениях:

```
exec_reads       reads, in bytes
exec_writes      writes, in bytes
exec_reads_blks  reads, in 8K-blocks
exec_writes_blks writes, in 8K-blocks
exec_user_time   user CPU time used
exec_system_time system CPU time used
exec_minflts     page reclaims (soft page faults)
exec_majflts     page faults (hard page faults)
exec_nswaps      swaps
exec_msgsnds     IPC messages sent
exec_msgrcvs     IPC messages received
exec_nsignals    signals received
exec_nvcsws      voluntary context switches
exec_nivcsws     involuntary context switches
```

Просмотр статистик pg_stat_kcache

- статистика по базам данных в представлении pg_stat_kcache:

```
 select datname database, pg_size_pretty(exec_minflts*4096) reclaims, pg_size_pretty(exec_majflts*4096) faults,
pg_size_pretty(exec_reads) reads, pg_size_pretty(exec_writes) writes, round(exec_system_time::numeric,0) sys,
round(exec_user_time::numeric,0) usr,  exec_nvcsws vsw, exec_nivcsws isw from pg_stat_kcache;
  database | reclaims | faults  | reads  | writes | sys | usr | vsw  | isw
----------+----------+---------+--------+--------+-----+-----+------+------
  postgres |   183 MB | 0 bytes | 226 MB | 13 GB  |  39 | 115 | 2717 | 2831
```

- Для соединения с pg_stat_statements удобно использовать функцию **pg_stat_kcache()**:

```
 select d.datname database, round(s.total_exec_time::numeric, 0) time, s.calls, pg_size_pretty(exec_minflts*4096)
reclaims, pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads,
pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user, round(k.exec_system_time::numeric,
2) sys, k.exec_nvcsws vsw, k.exec_nivcsws isw, left(s.query, 18) query from pg_stat_statements s join
pg_stat_kcache() k using (userid, dbid, queryid) join pg_database d on s.dbid = d.oid order by 2 desc;
 database|time   |calls | reclaims|  faults | reads | writes  | user  | sys  | vsw | isw |       query
---------+-------+------+---------+---------+-------+---------+-------+------+-----+-----+-------------------
 postgres| 25407 |    1 | 3752 kB | 0 bytes | 33 MB | 1277 MB | 11.42 | 4.00 | 306 | 362 | insert into test.f
 postgres| 24817 |    1 | 3752 kB | 0 bytes | 15 MB | 1297 MB | 11.36 | 3.92 | 294 | 167 | insert into test.f
 postgres| 24187 |    1 | 3748 kB | 0 bytes | 35 MB | 1321 MB | 11.70 | 3.71 | 277 | 598 | insert into test.f
```


Просмотр статистик pg_stat_kcache

Если exec_majflts незначительно по сравнению с exec_minflts, это означает, что оперативной памяти достаточно.

Пропорция exec_user_time и exec_system_time позволяет определить нет ли перекоса в сторону кода ядра или кода PostgreSQL и функций расширений и приложения.

Наличие exec_nivcsws (принудительных переключений контекста выполнения планировщиком операционной системы) указывает на то, что запрос активно нагружал процессор и процессор был узким местом.

Для соединения с pg_stat_statements удобнее использовать функцию **pg_stat_kcache()**, а не представления:

```
 select d.datname database, round(s.total_exec_time::numeric, 0) time, s.calls,
pg_size_pretty(exec_minflts*4096) reclaims, pg_size_pretty(exec_majflts*4096)
faults, pg_size_pretty(k.exec_reads) reads, pg_size_pretty(k.exec_writes) writes,
round(k.exec_user_time::numeric, 2) user, round(k.exec_system_time::numeric, 2)
sys, k.exec_nvcsws vsw, k.exec_nivcsws isw, left(s.query, 18) query from
pg_stat_statements s join pg_stat_kcache() k using (userid, dbid, queryid) join
pg_database d on s.dbid = d.oid order by 2 desc;
 database|time  |calls| reclaims|  faults| reads | writes  | user  | sys  | vsw | isw |       query
---------+------+-----+---------+--------+-------+---------+-------+------+-----+-----+-------------------
 postgres| 25407|    1| 3752 kB | 0 bytes| 33 MB | 1277 MB | 11.42 | 4.00 | 306 | 362 | insert into test.f
 postgres| 24817|    1| 3752 kB | 0 bytes| 15 MB | 1297 MB | 11.36 | 3.92 | 294 | 167 | insert into test.f
 postgres| 24187|    1| 3748 kB | 0 bytes| 35 MB | 1321 MB | 11.70 | 3.71 | 277 | 598 | insert into test.f
```

Разница в значениях столбцов s.shared_blks_hit, s.shared_blks_read, k.exec_reads/8192 exec_reads_blk, k.exec_minflts cache,k.exec_majflts disk если выбирать эти столбцы, может быть из-за разного момента сброса статистики. Поэтому использовать столбцы из разных расширений в вычислениях надо убедившись, что статистики были обнулены одновременно. Статистики разных расширений обнуляются разными функциями.

Практика

- Часть 1. Установка расширения pg_stat_kcache
- Часть 2. Использование расширения pg_stat_kcache
- Часть 3. Производительность при использовании direct i/o

Практика

Рассматривается что показывают статистики расширения `pg_stat_kcache` и какие статистики имеют смысл.

Также вы сравните производительность работы экземпляра в обычном режиме работы с вводом-выводом и с прямым режимом.

16

Расширение `pg_wait_sampling`

Расширение pg_wait_sampling

- входит во все сборки СУБД Tantor
- выдает статистику по событиям ожиданий всех процессов экземпляра
- для установки нужно загрузить библиотеку и установить расширение:

```
alter system set shared_preload_libraries = pg_stat_statements, pg_stat_kcache, pg_wait_sampling;
create extension if not exists pg_wait_sampling;
```

- библиотека pg_wait_sampling должна быть указана позже pg_stat_statements, чтобы расширение не перезаписало идентификаторы запросов (queryid)
- расширение использует фоновый процесс pg_wait_sampling collector
- процесс опрашивает состояние всех процессов экземпляра
- в расширение входят 4 функции и 3 представления:

```
\dx+ pg_wait_sampling
 function pg_wait_sampling_get_current(integer)
 function pg_wait_sampling_get_history()
 function pg_wait_sampling_get_profile()
 function pg_wait_sampling_reset_profile()
 view pg_wait_sampling_current
 view pg_wait_sampling_history
 view pg_wait_sampling_profile
```


Расширение pg_wait_sampling

Расширение входит во все сборки СУБД Tantor. Выдает статистику по событиям ожиданий всех процессов экземпляра. Для установки нужно загрузить библиотеку и установить расширение:

```
alter system set shared_preload_libraries = pg_stat_statements, pg_stat_kcache, pg_wait_sampling, pg_qualstats, pg_store_plans;
create extension if not exists pg_wait_sampling;
```

Библиотека pg_wait_sampling должна быть указана позже pg_stat_statements, чтобы pg_wait_sampling не перезаписывала идентификаторы запросов (queryid), которые используются pg_wait_sampling.

В расширение входят 4 функции и 3 представления:

```
\dx+ pg_wait_sampling
 function pg_wait_sampling_get_current(integer)
 function pg_wait_sampling_get_history()
 function pg_wait_sampling_get_profile()
 function pg_wait_sampling_reset_profile()
 view pg_wait_sampling_current
 view pg_wait_sampling_history
 view pg_wait_sampling_profile
```

Текущие события ожидания отображаются в представлении pg_stat_activity. Многие события ожидания длятся недолго и "поймать" их маловероятно. Расширение использует фоновый процесс pg_wait_sampling collector, который с частотой заданной параметром pg_wait_sampling.history_period или pg_wait_sampling.profile_period (по умолчанию 10 миллисекунд) опрашивает состояние всех процессов экземпляра, сохраняет pg_wait_sampling.history_size (по умолчанию 5000, максимальное значение определяется типом int4) событий в истории и группируется в "профиле" событий, доступном через представление pg_wait_sampling_profile.

История используется в кольцевом режиме: старые значения перезаписываются по кругу. Приложения могут сохранять собранную историю, запрашивая историю из представления:

```
select count(*) from pg_wait_sampling_history;
 count
 -----
 5000
```

История событий ожидания

- расширение использует "сэмплрование" с частотой от 1 миллисекунды (по умолчанию 10 миллисекунд)
- история доступна через представление:

```
select count(*) from pg_wait_sampling_history;
count
-----
5000
```

- по умолчанию сохраняется история 5000 последних событий ожидания
- расширение также использует разделяемую память под хранение своих трёх структур:
  - очередь (MessageQueue) фиксированного размера 16Кб
  - память под список PID
  - память под идентификаторы команд (`queryid`), выполняющихся процессами

```
 select * from (select *, lead(off) over(order by off)-off as diff from pg_shmem_allocations) as a
where name like '%wait%';
        name        |    off    | size  | allocated_size | diff
--------------------+-----------+-------+----------------+-------
 pg_wait_sampling | 148145920 | 17536 |          17536 | 17536
```


История событий ожидания

Историю событий ожидания можно посмотреть через представление:

`\sv pg_wait_sampling_history`

```
CREATE OR REPLACE VIEW public.pg_wait_sampling_history AS SELECT pid, ts,
event_type, event, queryid FROM pg_wait_sampling_get_history()
pg_wait_sampling_get_history(pid, ts, event_type, event, queryid)
```

Функция `pg_wait_sampling_get_history()` выдает те же самые данные и не имеет входных параметров.

Получение данных о том, что в настоящее время выполняет процесс, с помощью опроса его состояния с какой-то частотой используется в Oracle Database ASH (Active Session History), являющейся частью AWR (Automatic Workload Repository).

На экземпляре с множеством активных сессий история на 5000 событий может перезаписываться за доли секунды. В истории сохраняются события ожиданий всех процессов. Если серверные процессы не сталкиваются с блокировками, то 99.98% событий ожидания будет заполнено фоновыми процессами и не связано с запросами. Например, при работе стандартного теста: **`pgbench -T 100`** среди 5000 событий в истории можно будет иногда увидеть одну строку:

```
select * from pg_wait_sampling_history where queryid<>0;
  pid  |             ts                | event_type |        event        |       queryid
-------+-------------------------------+------------+---------------------+---------------------
 53517 | 2035-11-11 11:18:19.676412+03 | IPC        | MessageQueueReceive | 6530354471556151986
```

Расширение также использует разделяемую память под хранение своих трёх структур:

```
select * from (select *, lead(off) over(order by off)-off as diff from
pg_shmem_allocations) as a where name like '%wait%';
        name        |    off    | size  | allocated_size | diff
--------------------+-----------+-------+----------------+-------
 pg_wait_sampling | 148145920 | 17536 |          17536 | 17536
```

Большая часть занята очередью (MessageQueue) фиксированного размера 16Кб, памятью под список PID, памятью под идентификаторы команд (`queryid`), выполняющихся процессами. Размер структуры под хранение списка PID процессов определяется максимальным числом процессов экземпляра. Число определяется параметрами конфигурации и примерно равно: `max_connections, autovacuum_max_workers+1` (launcher), `max_worker_processes, max_wal_senders+5` (основных фоновых процессов). Память под queryid равна максимальному числу PID, умноженному на 8 байт (размер типа `bigint`, используемым `queryid`).

История событий ожидания

- в истории сохраняются события ожиданий всех процессов
- если серверные процессы не сталкиваются с блокировками, то 99.98% событий ожидания будет заполнено ожиданиями фоновых процессов, не связанных с запросами

```
select * from pg_wait_sampling_history where queryid<>0;
 pid  |              ts               | event_type |        event        |       queryid
------+-------------------------------+------------+---------------------+---------------------
53718 | 2035-11-11 11:41:55.629229+03 | IO         | BufFileWrite        | 6530354471556151986
53718 | 2035-11-11 11:41:59.513407+03 | IO         | BufFileRead         | 6530354471556151986
```

- Программы мониторинга могут запрашивать содержимое истории и сохранять его в своём хранилище. Определение максимальной частоты опроса истории:

```
select max(ts), min(ts), max(ts)-min(ts) duration from pg_wait_sampling_history;
              max               |              min               |    duration
--------------------------------+--------------------------------+-----------------
2035-11-11 11:58:19.691753+03 | 2035-11-11 11:55:41.153914+03 | 00:02:38.537839
```

- запрос может использоваться для определения значения параметров `pg_wait_sampling.history_period` и `pg_wait_sampling.history_size`

История событий ожидания (продолжение)

Разделяемая память, используется расширением, вероятно, потому что предполагалось иметь множество процессов-обработчиков, но они не понадобились.

При увеличении числа записей:

```
alter system set pg_wait_sampling.history_size = 100000;
```

история будет вытесняться реже и удержится в несколько раз больше строк с **queryid<>0**:

```
select * from pg_wait_sampling_history where queryid<>0;
 pid  |              ts               | event_type |        event        |       queryid
------+-------------------------------+------------+---------------------+---------------------
53718 | 2035-11-11 11:41:55.629229+03 | IO         | BufFileWrite        | 6530354471556151986
...
53718 | 2035-11-11 11:41:59.513407+03 | IO         | BufFileRead         | 6530354471556151986
53718 | 2035-11-11 11:42:54.618133+03 | IPC        | MessageQueueReceive | 6530354471556151986
(14 rows)
```

Программы мониторинга могут запрашивать содержимое истории и сохранять его в своём хранилище. Определить максимальную частоту для опроса истории ожиданий можно запросом:

```
select max(ts), min(ts), max(ts)-min(ts) duration from pg_wait_sampling_history;
              max               |              min               |    duration
--------------------------------+--------------------------------+-----------------
2035-11-11 11:58:19.691753+03 | 2035-11-11 11:55:41.153914+03 | 00:02:38.537839
```

В примере 2 минуты 38 секунд. Если запрашивать реже, будет потеряна часть строк.

Запрос может использоваться для определения значения параметров `pg_wait_sampling.history_period` и `pg_wait_sampling.history_size`.

Со значениями по умолчанию строки удерживаются в истории несколько секунд.

Параметр `pg_wait_sampling.history_period` задаёт частоту с которой опрашивается состояние процессов. По умолчанию 10 миллисекунд. Минимальное значение 1 миллисекунда. Если уменьшить частоту в 10 раз (до 1мс), то история `pg_wait_sampling_history` будет перезаписываться в 10 раз чаще и стоит увеличить в 10 раз значение `pg_wait_sampling.history_size`, чтобы время удержания событий осталось тем же.

Параметры расширения `pg_wait_sampling`

- `profile_period` и `history_period` интервал в миллисекундах для выборки из истории событий и для добавления в профиль
  - по умолчанию 10 миллисекунд
  - вероятность поимки события определяется только этими параметрами
  - **данные для истории и профиля выбираются и сохраняются независимо друг от друга**
- `pg_wait_sampling.history_size` (по умолчанию 5000, максимальное значение определяется типом `int4`) событий в истории
- `profile_pid` и `profile_queries` разбивают в профиле события по процессам и запросам

```
select name, setting, context, min_val, max_val, boot_val, pending_restart from pg_settings where name
like '%pg_wait_sampling%';
              name                | setting |  context   | min_val |  max_val   | boot_val| pending_restart
----------------------------------+---------+------------+---------+------------+---------+----------------
pg_wait_sampling.history_period   | 100000  | superuser | 1       | 2147483647 | 10      | f
pg_wait_sampling.history_size     | 100     | superuser | 100     | 2147483647 | 5000    | f
pg_wait_sampling.profile_period   | 10      | superuser | 1       | 2147483647 | 10      | f
pg_wait_sampling.profile_pid      | on      | superuser |         |            | on      | f
pg_wait_sampling.profile_queries | on      | superuser |         |            | on      | f
```


Параметры расширения `pg_wait_sampling`

Параметр `pg_wait_sampling.profile_period` устанавливает интервал в миллисекундах для выборки из истории событий для группировки и добавления в профиль. По умолчанию 10 миллисекунд. **Данные для истории и профиля выбираются и сохраняются независимо друг от друга.** В профиле могут быть пойманы события ожиданий, а в истории нет и наоборот. Вероятность поимки события определяется только двумя параметрами: `pg_wait_sampling.profile_period` и `pg_wait_sampling.history_period`. Чем меньше значение, тем выше вероятность поймать редкое событие ожидания. Профиль не забирает данные из истории, а параметр `pg_wait_sampling.profile_period` не зависит от `pg_wait_sampling.history_period` и `pg_wait_sampling.history_size`.

`pg_wait_sampling.history_size` (по умолчанию 5000, максимальное значение определяется типом `int4`) событий, сохраняемых в истории. Историю можно посмотреть через представление `pg_wait_sampling_history`. Значение параметра определяет число строк в представлении. История используется для запросов администраторами нечасто, так как не хранит число событий ожидания, в отличие от профиля. Если история не используется приложениями мониторинга, то стоит увеличить интервал сбора `profile_period`.

`pg_wait_sampling.profile_pid` и `pg_wait_sampling.profile_queries` ранжируют в профиле события по процессам и запросам из-за чего объем локальной памяти процесса-коллектора увеличивается. `profile_pid=on` оказывает большее влияние на память, так как профиль хранит данные по завершенным процессам.

Изменение значений всех параметров не требует перезапуска экземпляра, только перечитывания конфигурации select `pg_reload_conf();`

Изменение числа строк в истории возможно как в сторону увеличения, так и уменьшения. Вызов функции **`pg_wait_sampling_reset_profile()`** не очищает историю. История только перезатирается. Вызов функции очищает данные, накопленные в профиле событий ожидания.

`pg_wait_sampling.sample_cpu` по умолчанию `on`. Процессы, которые ничего не ждут тоже будут опрашиваться, значения в столбцах `event_type` и `event` будут пустыми.

Расширение появилось в 2016 году, не является стандартным расширением PostgreSQL. Идея расширения интересна, но примеров, где бы расширение помогло решить практические задачи нет. В отличие от Oracle Database часто выполняющиеся запросы в PostgreSQL не требуют сэмлирования и отражаются в `pg_stat_statements`. Внедрение идей, используемых в СУБД других семейств, часто бессмысленно в применении к PostgreSQL.
https://akorotkov.github.io/blog/2016/03/25/wait_monitoring_9_6/

Профиль pg_wait_sampling

- для просмотра числа событий используется представление:

```
select * from pg_wait_sampling_profile where queryid<>0 order by count desc;
  pid  | event_type |     event      |       queryid       | count
-------+------------+----------------+---------------------+-------
 55206 | IO         | DataFileRead   | -650897274631253140 |  1290
 55206 | IO         | DataFileExtend | -877373571429139692 |     6
```

- данные сохраняются по всем процессам, в том числе по серверным процессам завершившихся сессий, которые уже отсутствуют в операционной системе
- если pg_wait_sampling.profile_pid=true, то число строк будет неуклонно расти
- **расширение не очищает память, используемую профилем**
- **нужно периодически освобождать память вызовом функции pg_wait_sampling_reset_profile()**
  - функция очищает данные, накопленные в профиле событий ожидания
  - функция не очищает данные в истории

Профиль pg_wait_sampling

Фоновый процесс аккумулирует (группирует, агрегирует) события ожиданий и подсчитывает их количество (count). Это называется "профиль ожиданий" - число событий ожиданий, сгруппированных по:

1) типам и видам событий (event_type, event)
2) процессам (pid) если pg_wait_sampling.profile_pid=true
3) типам команд (queryid), если pg_wait_sampling.profile_queries=true.

Для просмотра числа событий используется представление pg_wait_sampling_profile:

```
select * from pg_wait_sampling_profile where queryid<>0 order by count desc;
  pid  | event_type |     event      |       queryid       | count
-------+------------+----------------+---------------------+-------
 55206 | IO         | DataFileRead   | -650897274631253140 |  1290
 55206 | IO         | DataFileExtend | -877373571429139692 |     6
```

Функция pg_wait_sampling_profile() выдает те же самые данные и не имеет входных параметров.

Данные сохраняются по всем процессам, в том числе по серверным процессам завершившихся сессий, которые уже отсутствуют в операционной системе. Если pg_wait_sampling.profile_pid=true, то число строк в представлении только увеличивается. Строки находятся в структуре памяти и если число строк будет большим, памяти может быть занято много. **Расширение не очищает память, используемую этой структурой. Поэтому либо нужно периодически освобождать память вызовом функции pg_wait_sampling_reset_profile()**, либо установить pg_wait_sampling.profile_pid=false. Если кластер обслуживает произвольные запросы, а не набор предопределенных, то тогда ещё придется установить pg_wait_sampling.profile_queries=false. Проще периодически сбрасывать собранную статистику функцией:

```
select pg_wait_sampling_reset_profile();
```

Пример того, что история и профиль независимы:

```
\dconfig *_period
    List of configuration parameters
           Parameter            | Value
--------------------------------+-------
 pg_wait_sampling.history_period | 10000
 pg_wait_sampling.profile_period | 1
(2 rows)
```

Установлены различные интервалы опроса процессов ("сэмилинга").

```
postgres=# select * from pg_wait_sampling_history where queryid<>0;
 pid  |              ts               | event_type |   event    |    queryid
------+-------------------------------+------------+------------+-----------------
59651 | 2035-11-11 11:11:37.770929+03 | Client     | ClientRead | 193615527688292385
(1 row)
```

В истории присутствует только одно событие.

```
postgres=# select * from pg_wait_sampling_profile where queryid<>0;
  pid  | event_type |        event         |       queryid        | count
-------+------------+----------------------+----------------------+-------
 59651 | IO         | DataFileRead         |   193615527688292385 |    27
 59651 | IPC        | MessageQueueInternal |  3255387186388375512 |     1
 59651 | IPC        | MessageQueueInternal |   193615527688292385 |     1
 59651 | IPC        | MessageQueueInternal | -6238149883089971617 |     1
 59651 | Client     | ClientRead           |   193615527688292385 |  3841
 59651 | IPC        | MessageQueueInternal |  6530354471556151986 |     2
(6 rows)
```

В профиле же "поймано" шесть событий ожидания, которые происходили при выполнении команд (**queryid<>0**).

Буфер истории полностью не заполнен, значит события из буфера не вытеснялись:

```
postgres=# select count(*) from pg_wait_sampling_history;
 count
-------
   126
(1 row)
```

Запросы к профилю pg_wait_sampling

- профиль показывает события ожидания, если за время от сброса ожидание было поймано хотя бы один раз:

```
select * from pg_wait_sampling_profile where pid=55549;
 pid   | event_type |        event         |       queryid       | count
-------+------------+----------------------+---------------------+--------
 55549 | Client     | ClientRead           |                   0 | 615444
 55549 | IPC        | MessageQueueInternal | 3255387186388375512 |      1
 55549 | IO         | DataFileRead         |                   0 |      6
```

- столбец pid позволяет связать строки с представлением pg_stat_activity:

```
select p.pid, left(a.backend_type, 14) process_type, a.application_name app, p.event_type, p.event,
 p.count from pg_wait_sampling_profile p join pg_stat_activity a on p.pid = a.pid
 where event_type<>'Activity';
 pid   |  process_type   |   app    | event_type |         event          | count
-------+-----------------+----------+------------+------------------------+--------
 60577 | client backend  | psql     | Client     | ClientRead             | 398415
 60683 | client backend  | pgbench  | IO         | WALSync                | 246986
 60569 | checkpointer    |          | Timeout    | CheckpointWriteDelay   | 171242
 60573 | walwriter       |          | LWLock     | WALWrite               |  20021
 60683 | client backend  | pgbench  | Timeout    | SpinDelay              |      2
```

- категория event_type='Activity' используется, если фоновому процессу нечего делать
- событие ClientRead нормально, если оно не в открытой транзакции

Запросы к профилю pg_wait_sampling

В расширении есть функция **pg_wait_sampling_get_current(PID)**. Функцией можно получить текущее событие ожидания и идентификатор команды. Пример:

```
 select * from pg_wait_sampling_get_current(55549);
  pid  | event_type | event |      queryid
-------+------------+-------+--------------------
 55549 |            |       | 193615527688292385
```

Короткие события ожидания поймать этой функцией нереально (в столбцах ожиданий пусто), поэтому функция непрактична. Запрос к профилю покажет события ожидания, если за время от сброса ожидание было поймано хотя бы один раз:

```
 select * from pg_wait_sampling_profile where pid=55549;
 pid   | event_type |        event         |       queryid       | count
-------+------------+----------------------+---------------------+--------
 55549 | Client     | ClientRead           |                   0 | 615444
 55549 | IPC        | MessageQueueInternal | 3255387186388375512 |      1
 55549 | IO         | DataFileRead         |                   0 |      6
```

Столбец pid позволяет связать строки с представлением pg_stat_activity:

```
 select p.pid, left(a.backend_type, 14) process_type, a.application_name app,
p.event_type, p.event, p.count from pg_wait_sampling_profile p join
pg_stat_activity a on p.pid = a.pid where event_type<>'Activity';
  pid   |  process_type   |   app    | event_type |         event          | count
 -------+-----------------+----------+------------+------------------------+--------
  60577 | client backend  | psql     | Client     | ClientRead             | 398415
  60683 | client backend  | pgbench  | IO         | WALSync                | 246986
  60569 | checkpointer    |          | Timeout    | CheckpointWriteDelay   | 171242
  60573 | walwriter       |          | LWLock     | WALWrite               |  20021
  60683 | client backend  | pgbench  | Timeout    | SpinDelay              |      2
```

Категория event_type='Activity' используется, если фоновому процессу нечего делать (описание категории в файле wait_event.h), это обычное состояние фоновых процессов.

В категорию 'Timeout' входят события ожидания, которые обычно являются нормой, но не все (WAIT_EVENT_SPIN_DELAY). Не стоит исключать события этой категории из выборки.

Событие ClientRead нормально, если оно присутствует не в открытой транзакции. Это ожидание команды от клиента и встречается чаще всего.

Представление **pg_stat_activity** содержит строки только по работающим процессам. Представление **pg_wait_sampling_profile** содержит строки по всем процессам, в том числе тех, которые завершили работать. Если использовать LEFT JOIN, то будут выведены строки по таким процессам, но столбцы, относящиеся к **pg_stat_activity** будут пустыми.

Запросы к профилю pg_wait_sampling

- пример группировки для получения данных о том, каких событий больше всего ждали процессы экземпляра:

```
 select event_type, event, sum(count) count from pg_wait_sampling_profile where event_type<>'Activity'
group by event_type, event order by count desc;
  event_type |         event          | count
 ------------+------------------------+--------
  IO         | WALSync                |  84388
  LWLock     | WALWrite               |   4549
```

- пример соединения с представлением pg_stat_statements:

```
 select calls, round(total_exec_time+total_plan_time) time, rows, shared_blks_hit hit, shared_blks_read
read, shared_blks_dirtied dirty, p.event_type ev_t, p.event, p.count, left(query,20) query from
pg_stat_statements join pg_wait_sampling_profile p using (queryid) order by p.count desc limit 7;
 calls | time   |  rows  |   hit   | read  | dirty |  ev_t  |     event      |count |     query
------+-------+--------+---------+-------+-------+--------+----------------+------+----------------
415384|188566 | 415384 | 2572231 | 10815 | 27047 | IO     | DataFileRead   |15657 | UPDATE pgbench_accou
```

- пример соединения с представлением pg_locks:

```
 select p.pid, left(l.relation::regclass::text,19) relation, l.locktype, replace(l.mode,'Lock','') mode,
l.granted g, l.fastpath f, p.event_type type, p.event, p.count from pg_wait_sampling_profile p join
pg_locks l on p.pid = l.pid where event_type<>'Activity' and locktype<>'virtualxid' order by p.count
desc, p.pid, l.relation desc limit 30;
   pid  |      relation       |   locktype    |     mode      | g | f |  type  |   event    |  count
-------+---------------------+---------------+---------------+---+---+--------+------------+---------
 11739 | pg_locks            | relation      | AccessShare   | t | t | Client | ClientRead | 6622131
 11731 |                     | transactionid | Exclusive     | t | f | IO     | WALSync    | 1906966
```


Запросы к профилю pg_wait_sampling (продолжение)

Пример группировки для получения данных о том, каких событий больше всего ждали процессы экземпляра:

```
 select event_type, event, sum(count) count from pg_wait_sampling_profile where
event_type<>'Activity' group by event_type, event order by count desc;
  event_type |         event          | count
 ------------+------------------------+--------
  Client     | ClientRead             | 224012
  Timeout    | CheckpointWriteDelay   | 137624
  IO         | WALSync                |  84388
  IO         | DataFileRead           |  19406
  LWLock     | WALWrite               |   4549
```

Длительность событий ожидания расширение не собирает. Сбор длительности привел бы к большим накладным расходам. Можно только оценить длительность исходя из того, что если событие было поймано при опросе с интервалом раз в pg_wait_sampling.profile_period миллисекунд, то наиболее вероятно, что длительность события не сильно меньше интервала. Если длительность больше, то растет вероятность поймать событие несколько раз.

Пример соединения с представлением pg_stat_statements:

```
 select calls, round(total_exec_time+total_plan_time) time, rows, shared_blks_hit hit, shared_blks_read read,
shared_blks_dirtied dirty, p.event_type ev_t, p.event, p.count, left(query,20) query from pg_stat_statements join
pg_wait_sampling_profile p using (queryid) order by p.count desc limit 7;
 calls | time   |  rows  |   hit   | read  | dirty |  ev_t  |     event      |count |     query
------+-------+--------+---------+-------+-------+--------+----------------+------+----------------
415384|188566 | 415384 | 2572231 | 10815 | 27047 | IO     | DataFileRead   |15657 | UPDATE pgbench_accou
     1|   812 |      1 |      49 |    11 |     2 | IO     | DataFileRead   |   75 | select count(*) from
415384|  9239 | 415384 |  420880 |     0 |  2667 | IO     | DataFileExtend |    4 | INSERT INTO pgbench_
415384| 21376 | 415384 | 1404998 |     1 |     9 | LWLock | BufferContent  |    2 | UPDATE pgbench_branc
```

События ожидания фоновых процессов не выводятся, так как они не выполняют команды (queryid=0).

Пример соединения с представлением pg_locks:

```
 select p.pid, left(l.relation::regclass::text,19) relation, l.locktype, replace(l.mode,'Lock','') mode,
l.granted g, l.fastpath f, p.event_type type, p.event, p.count from pg_wait_sampling_profile p join pg_locks l on
p.pid = l.pid where event_type<>'Activity' and locktype<>'virtualxid' order by p.count desc, p.pid, l.relation
desc limit 30;
   pid  |      relation       |   locktype    |     mode      | g | f |  type  |   event    |  count
-------+---------------------+---------------+---------------+---+---+--------+------------+---------
 11739 | pg_wait_sampling_pr | relation      | AccessShare   | t | t | Client | ClientRead | 6622131
 11739 | pg_locks            | relation      | AccessShare   | t | t | Client | ClientRead | 6622131
 11731 |                     | transactionid | Exclusive     | t | f | IO     | WALSync    | 1906966
 11731 | pgbench_accounts_pk | relation      | RowExclusive | t | t | IO     | WALSync    | 1906966
```

Сброс статистик

- перед выполнением сравнительных тестов бывает удобно сбросить статистики, чтобы накопленные с предыдущего теста данные не попадали в результат. Для этого используются функции сброса статистик:

```
select pg_stat_reset();
select pg_stat_reset_shared(null);
select pg_stat_reset_shared('bgwriter');
select pg_stat_reset_shared('archiver');
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('wal');
select pg_stat_reset_shared('recovery_prefetch');
select pg_stat_reset_slru(null);
select pg_stat_statements_reset();
select pg_stat_kcache_reset();
select pg_wait_sampling_reset_profile();
select pg_qualstats_reset();
```

- для очистки кэша буферов остановить экземпляр: `pg_ctl stop -m fast`
- очистка pg_prewarm: `rm -f $PGDATA/autoprewarm.blocks`
- сброс на диск грязных страниц: `sync`
- очистка страничного кэша: `echo 3 > /proc/sys/vm/drop_caches`
- запуск экземпляра: `sudo systemctl restart tantor-se-server-16`

Сброс статистик

Загрузка библиотек расширений:

```
alter system set shared_preload_libraries = pg_stat_statements, pg_wait_sampling, pg_qualstats, pg_store_plans, pg_prewarm, pg_stat_kcache;
```

Перед выполнением сравнительных тестов бывает удобно сбросить статистики, чтобы накопленные с предыдущего теста данные не попадали в результат. Для этого используются функции сброса статистик:

```
select pg_stat_reset();
select pg_stat_reset_shared(null);
select pg_stat_reset_shared('bgwriter');
select pg_stat_reset_shared('archiver');
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('wal');
select pg_stat_reset_shared('recovery_prefetch');
select pg_stat_reset_slru(null);
select pg_stat_statements_reset();
select pg_stat_kcache_reset();
select pg_wait_sampling_reset_profile();
select pg_qualstats_reset();
```

Для очистки кэша буферов остановить экземпляр:

```
pg_ctl stop -m fast
```

Удаление файла расширения `pg_prewarm`:

```
rm -f $PGDATA/autoprewarm.blocks
```

Сброс на диск грязных страниц из кэша операционной системы:

```
sync
```

Очистка чистых страниц страничного кэша и структур памяти slab операционной системы:

```
root@tantor:~# echo 3 > /proc/sys/vm/drop_caches
```

или

```
postgres@tantor:~# echo 3 | sudo tee /proc/sys/vm/drop_caches
```

Запуск экземпляра:

```
sudo systemctl restart tantor-se-server-16
```

Практика

- Использование расширения pg_wait_sampling

Практика

В практике устанавливается расширение и рассматриваются примеры его использования.