

Autovacuum - Poe

 poe.com/chat/kxmzablqr9a06hqi4k

3.6K followers

.

171+ points

History

Gemini 2.5 Pro Experimental is Google's latest most capable model with frontier performance on various key benchmarks; supports web search and 1 million tokens of input context. Powered by gemini-2.5-pro-preview-05-06 snapshot.

1. OFFICIAL

Today

Thinking...

Considering Autovacuum Parameters

Alright, let's start with the basics. I need to refresh my memory on the specifics of `autovacuum_analyze_scale_factor` and how it interacts with `autovacuum_analyze_threshold`. A quick Google search should provide the foundation for my analysis. Then I can move onto more advanced considerations.

Researching Scale Factor

Okay, first things first. I'm Googling "autovacuum_analyze_scale_factor" to solidify my understanding. I'll then seek specific calculations and examples to clarify its behavior in PostgreSQL. The goal is to grasp how that setting, specifically `0.005`, impacts analysis on a 1-million-row table.

`autovacuum_analyze_scale_factor` — это параметр конфигурации в PostgreSQL, который влияет на то, как часто будет запускаться процесс автоматического анализа таблицы. [1][2] Анализ таблицы собирает статистику о распределении данных в ней, которая затем используется планировщиком запросов для выбора наиболее эффективных планов выполнения запросов. [3][4]

Как это работает:

Процесс `ANALYZE` для конкретной таблицы запускается автоматически, когда количество измененных (вставленных, обновленных или удаленных) строк с момента последнего анализа превышает определенный порог. Этот порог рассчитывается по следующей формуле:

Порог для ANALYZE = `autovacuum_analyze_threshold` + `autovacuum_analyze_scale_factor` * количество_строк_в_таблице

- `autovacuum_analyze_threshold` — это базовое пороговое значение количества измененных строк (по умолчанию 50). [5][6]
- `autovacuum_analyze_scale_factor` — это доля от общего числа строк в таблице (по умолчанию 0.1, то есть 10%). [1][2]

- **количество_строк_в_таблице** (в терминах PostgreSQL **reltuples**) — это текущее количество строк в таблице. [6]

Что означает **autovacuum_analyze_scale_factor = 0.005** для таблицы в 1 миллион строк:

Если для таблицы с 1 миллионом строк установлен **autovacuum_analyze_scale_factor = 0.005**, то часть порога, зависящая от этого коэффициента, составит:

$0.005 * 1,000,000 \text{ строк} = 5,000 \text{ строк}$

Таким образом, автоматический анализ этой таблицы будет запускаться, когда количество измененных строк достигнет:

autovacuum_analyze_threshold (по умолчанию 50) + 5,000 строк = 5,050 строк

Значение по умолчанию и рекомендации:

Значение по умолчанию для **autovacuum_analyze_scale_factor** составляет 0.1 (10%). [1][2] Для очень больших таблиц (например, с 1 миллионом строк и более) рекомендуется уменьшать это значение, например, до 0.05 (5%) или даже 0.01 (1%), чтобы анализ запускался чаще и статистика оставалась более актуальной. [1][7] Значение 0.005 (0.5%) является еще более агрессивной настройкой, предполагающей очень частый анализ таблицы.

Важно отметить:

- Более частый анализ может привести к небольшому увеличению нагрузки на систему, но обеспечивает планировщик запросов более точной статистикой, что, в свою очередь, может значительно улучшить производительность запросов, особенно для таблиц с часто изменяющимися данными или неравномерным распределением данных. [3][8]
- Для очень больших таблиц иногда имеет смысл установить **autovacuum_analyze_scale_factor** в 0 и полагаться только на **autovacuum_analyze_threshold**, чтобы запускать анализ после фиксированного числа изменений, независимо от общего размера таблицы. [9][10]

Таким образом, установка **autovacuum_analyze_scale_factor = 0.005** для таблицы в 1 миллион строк означает, что автоматический анализ статистики для этой таблицы будет инициирован после изменения примерно 5050 строк (при условии значения **autovacuum_analyze_threshold** по умолчанию). Это настройка для более частого обновления статистики по сравнению со стандартными значениями.

Learn more:

5:06 PM

```
Какие выводы в совокупности: WITH (
FILLFACTOR = 90,
autovacuum_analyze_scale_factor = 0.005,
autovacuum_analyze_threshold = 50,
autovacuum_vacuum_scale_factor = 0.001
)
```

5:08 PM

Thinking...

Examining PostgreSQL Parameters

Okay, I need to break down these PostgreSQL table storage parameters. I'll explain their function and impact on performance, outlining trade-offs. This is basic, but the user needs a clear overview before deeper diving.

Refining Parameter Explanations

Right, first, I'll tackle `FILLFACTOR`. Then, I'll move on to the `autovacuum` settings, specifically `autovacuum_analyze_scale_factor` and `autovacuum_analyze_threshold`. Each parameter needs a clear explanation, so the user understands its impact on table storage and performance. I'll keep it concise, focusing on the essentials.

Эти параметры хранения таблицы в PostgreSQL указывают на довольно специфическую конфигурацию, нацеленную на таблицы с высокой частотой обновлений (UPDATE) и вставок (INSERT), особенно если таблица имеет значительный размер. Давайте разберем каждый параметр и затем сделаем общие выводы.

Разбор параметров:

1. `FILLFACTOR = 90`

- **Что это:** `FILLFACTOR` определяет, насколько плотно PostgreSQL будет заполнять страницы таблицы данными при первоначальной загрузке и последующих вставках. [1][2] Значение 100 (по умолчанию для таблиц) означает полное заполнение страницы. [1][2]
- **Влияние `FILLFACTOR = 90`:** Оставляя 10% свободного места на каждой странице, вы даете PostgreSQL возможность размещать обновленные версии строк на той же странице, где находилась старая версия (это называется HOT-updates - Heap Only Tuple updates). [3] Это более эффективно, так как не требует обновления индексов для этой строки. [3] Это помогает уменьшить раздувание таблицы (bloat) и фрагментацию индексов, особенно в таблицах с частыми операциями `UPDATE`. [1][3] Для таблиц с интенсивными обновлениями рекомендуются значения в диапазоне 70-90. [3]

2. `autovacuum_analyze_scale_factor = 0.005`

- **Что это:** Этот параметр, вместе с `autovacuum_analyze_threshold`, определяет, когда будет запускаться автоматический `ANALYZE` для таблицы. [4][5] `ANALYZE` собирает статистику о распределении данных, которую планировщик запросов использует для построения эффективных планов выполнения. [6][7]
- **Влияние `autovacuum_analyze_scale_factor = 0.005`:** Порог для запуска `ANALYZE` рассчитывается как: `autovacuum_analyze_threshold + (autovacuum_analyze_scale_factor * количество_строк_в_таблице)`. [4] Значение 0.005 (или 0.5%) означает, что `ANALYZE` будет запускаться, когда изменится (вставки, обновления, удаления) 0.5% строк таблицы плюс значение `autovacuum_analyze_threshold`. [4] Это более агрессивная настройка, чем значение по умолчанию (0.1 или 10%), и приводит к более частому обновлению статистики. [4][5] Это полезно для таблиц, где распределение данных быстро меняется.

3. `autovacuum_analyze_threshold = 50`

- **Что это:** Минимальное количество измененных (вставленных, обновленных или удаленных) кортежей, которое должно произойти в таблице, чтобы инициировать операцию `ANALYZE` со стороны автовакуума. [4][8]
- **Влияние `autovacuum_analyze_threshold = 50`:** Это значение по умолчанию. [4][8] В сочетании с `autovacuum_analyze_scale_factor = 0.005`, для таблицы, скажем, в 1 миллион строк, `ANALYZE` сработает после $50 + (0.005 * 1000000) = 50 + 5000 = 5050$ измененных строк. Для маленьких таблиц этот порог будет доминировать, а для больших — `scale_factor`. [9]

4. `autovacuum_vacuum_scale_factor = 0.001`

- **Что это:** Этот параметр, вместе с `autovacuum_vacuum_threshold`, определяет, когда будет запускаться автоматический `VACUUM` для таблицы. [4][10] `VACUUM` освобождает место, занятое "мертвыми" строками (удаленными или старыми версиями обновленных строк), и делает это место доступным для повторного использования. [6][11]
- **Влияние `autovacuum_vacuum_scale_factor = 0.001`:** Порог для запуска `VACUUM` рассчитывается как: `autovacuum_vacuum_threshold + (autovacuum_vacuum_scale_factor * количество_строк_в_таблице)`. [4] Значение 0.001 (или 0.1%) означает, что `VACUUM` будет запускаться, когда количество "мертвых" строк достигнет 0.1% от общего числа строк в таблице плюс значение `autovacuum_vacuum_threshold` (по умолчанию 50 строк). [4][10] Это очень агрессивная настройка по сравнению со значением по умолчанию (0.2 или 20%), что приведет к очень частому запуску `VACUUM`. [4][10] Это критически важно для таблиц с высокой частотой `UPDATE` и `DELETE` операций, чтобы предотвратить чрезмерное раздувание таблицы и поддерживать производительность. [12][13]

Совокупные выводы:

Такая комбинация настроек явно предназначена для **очень активно изменяемой (много `UPDATE`, `INSERT`, `DELETE`) и, вероятно, большой таблицы**.

- **Агрессивная очистка и анализ:** Низкие значения `autovacuum_analyze_scale_factor` (0.005) и особенно `autovacuum_vacuum_scale_factor` (0.001) указывают на стремление запускать `ANALYZE` и `VACUUM` как можно чаще. [4][12] Это помогает:
 - **Поддерживать актуальность статистики (`ANALYZE`):** Для быстро меняющихся данных точная статистика критична для производительности запросов. [6][14]
 - **Минимизировать раздувание таблицы (`VACUUM`):** Частые `VACUUM` быстро освобождают место от мертвых строк, предотвращая рост таблицы и снижение производительности сканирования. [10][12]
- **Оптимизация для `UPDATE` (`FILLFACTOR = 90`):** Уменьшенный `FILLFACTOR` помогает снизить негативные эффекты от частых `UPDATE`, позволяя выполнять больше HOT-обновлений и уменьшая фрагментацию. [1][3] Это хорошо сочетается с агрессивным `VACUUM`, так как даже при наличии свободного места для HOT-обновлений, старые версии строк все равно нужно убирать.
- **Потенциальные компромиссы:**
 - **Повышенная нагрузка от автовакуума:** Более частые запуски `VACUUM` и `ANALYZE` могут создавать дополнительную нагрузку на CPU и I/O. [15] Однако, для таблиц, где производительность запросов и контроль над раздуванием критичны, это может быть оправданным компромиссом.
 - **Небольшое увеличение размера таблицы из-за `FILLFACTOR`:** Оставление свободного места на страницах означает, что таблица изначально займет немного больше места на диске. [2] Но это компенсируется уменьшением раздувания в долгосрочной перспективе для таблиц с частыми обновлениями.

Для каких таблиц это подходит?

- Таблицы с очень высокой интенсивностью операций **UPDATE** и **DELETE**.
- Большие таблицы, где даже небольшой процент измененных строк означает значительное их количество, и стандартные настройки **autovacuum** могут приводить к слишком редким запускам.
- Системы, где производительность запросов и предотвращение раздувания таблиц имеют высокий приоритет, и некоторая дополнительная нагрузка от автовакуума приемлема.

Рекомендации:

Эти настройки являются специфичными и должны применяться осознанно, после анализа нагрузки на конкретную таблицу. [13][16] Важно мониторить активность автовакуума и производительность системы после применения таких изменений. [7][17] Для некоторых очень больших таблиц может быть даже целесообразно установить **autovacuum_vacuum_scale_factor** и **autovacuum_analyze_scale_factor** в 0 и полностью полагаться на **autovacuum_vacuum_threshold** и **autovacuum_analyze_threshold** для более предсказуемого поведения. [18]

Learn more:

5:08 PM

Thinking...

Refining the Search Strategy

Okay, I'll start with a targeted Google search. I need to find specific information on "RLS Policies PostgreSQL" and "Row Level Security examples." That should give me a solid foundation to begin my analysis.

Reviewing the Initial Approach

I'll begin by searching for "RLS Policies PostgreSQL" and "Row Level Security PostgreSQL" to get a grasp of the basics and definitions. Next, I'll delve into the mechanics: permissive/restrictive policies, **USING**, and **WITH CHECK** clauses. I'll then seek out practical examples to cement my understanding before I synthesize everything to formulate my explanation.

RLS Policies, или Политики Безопасности на Уровне Строк (Row-Level Security), в PostgreSQL — это мощный механизм, который позволяет контролировать доступ к данным на уровне отдельных строк таблицы. [1][2] В отличие от стандартных привилегий SQL (GRANT/REVOKE), которые обычно определяют доступ ко всей таблице или отдельным столбцам, RLS позволяет задавать более гранулированные правила, определяющие, какие именно строки пользователь может видеть или изменять. [3][4]

Как это работает:

1. **Включение RLS для таблицы:** Прежде чем политики начнут действовать, для конкретной таблицы необходимо активировать RLS с помощью команды **ALTER TABLE ... ENABLE ROW LEVEL SECURITY**. [5][6] Если RLS включена, но для таблицы не определено ни одной политики, доступ к строкам будет запрещен для всех пользователей, кроме владельца таблицы и ролей с атрибутом **BYPASSRLS**. [1][2]

2. **Создание политик:** Политики создаются с помощью команды **CREATE POLICY**. [5][7] Каждая политика содержит:

- **Имя политики:** Имена политик должны быть уникальны в пределах одной таблицы. [5][7]
- **Таблица:** Указывается таблица, к которой применяется политика. [5]
- **Команды (необязательно):** Политика может применяться к определенным SQL-командам (**SELECT**, **INSERT**, **UPDATE**, **DELETE**) или ко всем командам (**ALL**). [5][7] По умолчанию политика применяется ко всем командам. [7]
- **Роли (необязательно):** Политика может быть назначена определенным ролям базы данных. [5] По умолчанию политика применяется ко всем ролям (**PUBLIC**). [5]
- **Выражение USING:** Это SQL-выражение, которое возвращает булево значение (ИСТИНА/ЛОЖЬ). [5][7] Оно проверяется для каждой существующей строки таблицы при выполнении операций **SELECT**, **UPDATE** или **DELETE**. [5][7] Если выражение возвращает **TRUE**, строка видима и доступна для операции. Если **FALSE** или **NULL**, строка скрывается или операция над ней не выполняется (ошибка при этом не возникает, строка просто "не существует" для данного пользователя). [5][8]
- **Выражение WITH CHECK:** Это также SQL-выражение, возвращающее булево значение. [5][7] Оно проверяется для новых или изменяемых строк при выполнении операций **INSERT** или **UPDATE**. [5][7] Если выражение возвращает **TRUE**, операция разрешена. Если **FALSE** или **NULL**, возникает ошибка, и операция не выполняется. [5][7] Выражения **WITH CHECK** применяются после BEFORE-триггеров, но до фактического изменения данных и до других ограничений. [7] Если для политики **UPDATE** или **ALL** не указано выражение **WITH CHECK**, то используется выражение **USING**. [7][9]

3. **Типы политик: PERMISSIVE vs RESTRICTIVE:**

- **PERMISSIVE (Разрешающие):** Это тип политики по умолчанию. [5][10] Если к таблице применяется несколько разрешающих политик, строка считается доступной, если **хотя бы одна** из разрешающих политик для данной команды и роли возвращает **TRUE**. [10][11] Чтобы иметь доступ к строкам таблицы с включенным RLS, должна существовать хотя бы одна разрешающая политика. [10][11]
- **RESTRICTIVE (Ограничивающие):** Ограничивающие политики применяются **после** разрешающих. [10][12] Чтобы строка была доступна, она должна удовлетворять **всем** ограничивающим политикам, которые к ней применяются (в дополнение к удовлетворению хотя бы одной разрешающей). [10][11] Если хотя бы одна ограничивающая политика возвращает **FALSE**, доступ к строке запрещается, даже если разрешающие политики его предоставили. [10]

Ключевые моменты и соображения:

- **Множественные политики:** К одной таблице можно применять несколько политик. [7][11] Их взаимодействие определяется типом (PERMISSIVE/RESTRICTIVE) и тем, к каким командам и ролям они относятся. [11][12]
- **Безопасность:** RLS предоставляет мощный инструмент для реализации детализированного контроля доступа непосредственно на уровне базы данных. [4][5] Это может упростить логику безопасности в приложениях, так как правила доступа применяются последовательно, независимо от того, как осуществляется доступ к данным. [4][10]
- **Примеры использования:**
 - Многопользовательские приложения, где каждый пользователь должен видеть только свои данные (например, свои заказы, свои документы).
 - Ограничение доступа к конфиденциальной информации на основе роли пользователя или других атрибутов.
 - Реализация иерархического доступа, когда менеджеры видят данные своих подчиненных.

Пример простой политики:

Предположим, есть таблица `documents` со столбцом `owner_user_id`. Мы хотим, чтобы пользователи могли видеть только свои документы.

sql

```
-- Включаем RLS для таблицы documents
ALTER TABLE documents ENABLE ROW LEVEL SECURITY;

-- Создаем политику, разрешающую пользователю видеть только свои документы
CREATE POLICY
user_can_see_own_documents
ON documents
FOR SELECT TO PUBLIC -- Применяется ко всем ролям
USING (owner_user_id = current_user); -- current_user -
это специальная функция PostgreSQL, возвращающая имя текущего пользователя
-- Для операций INSERT и UPDATE, чтобы пользователь мог создавать/изменять только свои документы:
CREATE POLICY user_can_manage_own_documents
ON documents
FOR ALL -- Можно указать INSERT, UPDATE отдельно
TO PUBLIC
USING (owner_user_id = current_user)
WITH CHECK (owner_user_id = current_user);
```

В этом примере выражение `USING (owner_user_id = current_user)` гарантирует, что при запросе `SELECT` пользователь увидит только те строки, где `owner_user_id` совпадает с его именем. Выражение `WITH CHECK` с тем же условием не позволит пользователю вставить новую строку или обновить существующую, если `owner_user_id` в новой/измененной строке не будет равен его имени. [5][12]

RLS Policies являются важной частью системы безопасности PostgreSQL, позволяя администраторам баз данных точно настраивать права доступа к данным. [2][15]

Learn more:

5:13 PM

Нужно найти размеры TOAST таблицы и TOAST индекса для таблицы и столбца PostgreSQL

5:18 PM

Thinking...

Examining Table and Index Names

Okay, I need to pinpoint the TOAST table and index names. My initial approach will be a targeted search to understand how these objects are identified within the system. This should be straightforward; the documentation is usually quite precise about this.

Delineating Search Parameters

I'll begin by identifying the methods for pinpointing the TOAST table and index names linked to a table and column. Next, I'll find the functions or queries needed to get table and index sizes by name. Finally, I will piece it all together.

Чтобы найти размеры TOAST-таблицы и TOAST-индекса для конкретной таблицы в PostgreSQL, вам нужно выполнить несколько шагов, используя системные каталоги и функции PostgreSQL.

Что такое TOAST?

Прежде чем перейти к запросам, кратко напомним, что TOAST (The Oversized-Attribute Storage Technique) — это механизм, который PostgreSQL использует для хранения больших значений атрибутов (колонок).

[1][2] Если значение в колонке превышает определенный порог (обычно около 2 КБ), оно может быть

сжато и/или сохранено во внешней TOAST-таблице, а в основной таблице останется только указатель. [1]
[3] Каждая TOAST-таблица имеет связанный с ней индекс для быстрого доступа к "кускам" (chunks) данных. [1][3]

Шаги для определения размеров:

1. Найти OID основной таблицы:

Вам понадобится OID (Object Identifier) вашей таблицы. OID – это уникальный внутренний идентификатор объекта в PostgreSQL.

sql

```
SELECT 'имя_вашей_схемы.имя_вашей_таблицы'::regclass::oid;
```

Или, если вы работаете в psql и знаете имя таблицы:

sql

```
SELECT c.oid
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE c.relname = 'имя_вашей_таблицы' AND n.nspname = 'имя_вашей_схемы';
```

2. Найти OID и имя TOAST-таблицы:

OID TOAST-таблицы хранится в системном каталоге `pg_class` в поле `reltoastrelid` для основной таблицы. [1][3]

sql

```
SELECT c.reltoastrelid, t.relname AS toast_table_name
FROM pg_class c
LEFTJOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.oid = (SELECT 'имя_вашей_схемы.имя_вашей_таблицы'::regclass::oid);
```

Или используя OID из предыдущего шага:

sql

```
SELECT c.reltoastrelid, t.relname AS toast_table_name
FROM pg_class c
LEFTJOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.oid = YOUR_TABLE_OID; -- Замените YOUR_TABLE_OID на OID вашей таблицы
```

Если `reltoastrelid` равен 0, это означает, что у таблицы нет TOAST-таблицы (например, если в ней нет TOAST-колонок или они еще не хранят достаточно большие значения).

3. Найти имя TOAST-индекса:

TOAST-таблица всегда имеет один индекс, обычно с именем `pg_toast_ИМЯ_TOAST_ТАБЛИЦЫ_index` или что-то подобное. Более надежный способ – найти его через системные каталоги, зная OID TOAST-таблицы. Индекс TOAST-таблицы создается по столбцам `chunk_id` и `chunk_seq`. [1][3]

sql

```
SELECT i.relname AS toast_index_name
FROM pg_class t
JOIN pg_index ix ON ix.indrelid = t.oid
JOIN pg_class i ON i.oid = ix.indexrelid
WHERE t.oid = YOUR_TOAST_TABLE_OID; -- Замените YOUR_TOAST_TABLE_OID на OID TOAST-таблицы из шага 2
```


4. Определить размер TOAST-таблицы:

Используйте функцию `pg_total_relation_size()` или `pg_table_size()` для TOAST-таблицы.

`pg_total_relation_size()` включает размер самой таблицы и всех ее индексов (в данном случае, одного TOAST-индекса). `pg_table_size()` показывает размер только данных таблицы.

С использованием имени TOAST-таблицы:

sql

```
SELECT pg_size_pretty(pg_total_relation_size('pg_toast.имя_вашей_toast_таблицы'));
SELECT pg_size_pretty(pg_table_size('pg_toast.имя_вашей_toast_таблицы'));
```

Или с использованием OID TOAST-таблицы:

sql

```
SELECT pg_size_pretty(pg_total_relation_size(YOUR_TOAST_TABLE_OID));
SELECT pg_size_pretty(pg_table_size(YOUR_TOAST_TABLE_OID));
```

Функция `pg_size_pretty()` форматирует размер в человекочитаемый вид (например, "10 MB"). [4]
[5]

5. Определить размер TOAST-индекса:

Используйте функцию `pg_relation_size()` для TOAST-индекса.

С использованием имени TOAST-индекса:

sql

```
SELECT pg_size_pretty(pg_relation_size('pg_toast.имя_вашего_toast_индекса'));
```

Или с использованием OID TOAST-индекса (который можно получить из `pg_class` по его имени, если вы нашли имя на шаге 3, или напрямую из `pg_index`):

sql

```
-- Если вы знаете OID TOAST-индекса
SELECT
pg_size_pretty(pg_relation_size(YOUR_TOAST_INDEX_OID));

-- Или, чтобы получить OID индекса по его имени (если нашли имя)
SELECT
pg_size_pretty(pg_relation_size((SELECT oid FROM pg_class WHERE relname
='имя_вашего_toast_индекса' AND relnamespace = (SELECT oid FROM pg_namespace WHERE nspname
='pg_toast'))));
```

Сводный запрос (пример):

Этот запрос попытается объединить некоторые шаги. Замените `'your_schema_name'` и `'your_table_name'` на актуальные значения.

sql

```

WITH main_table AS (
SELECT      c.oid AS table_oid,
            c.relname AS table_name,
            n.nspname AS schema_name,
            c.reltoastrelid AS toast_table_oid
FROM        pg_class c
JOIN        pg_namespace n ON n.oid = c.relnamespace
WHERE       n.nspname = 'your_schema_name' AND c.relname = 'your_table_name'),
toast_info AS (
SELECT      mt.table_oid AS main_table_oid,
            mt.table_name AS main_table_name,
            mt.schema_name AS main_schema_name,
            tt.oid AS toast_table_oid,
            tt.relname AS toast_table_name,
            ni.oid AS toast_index_oid,
            ni.relname AS toast_index_name
FROM        main_table mt
LEFTJOIN    pg_class tt ON mt.toast_table_oid = tt.oid
LEFTJOIN    pg_index ix ON ix.indrelid = tt.oid
LEFTJOIN    pg_class ni ON ni.oid = ix.indexrelid
WHERE       tt.oid IS NOT NULL -- Убедимся, что TOAST таблица существует)
SELECT      ti.main_schema_name || '.' || ti.main_table_name AS main_table,
            ti.toast_table_name,
            pg_size_pretty(pg_table_size(ti.toast_table_oid)) AS toast_table_data_size,
            ti.toast_index_name,
            pg_size_pretty(pg_relation_size(ti.toast_index_oid)) AS toast_index_size,
            pg_size_pretty(pg_total_relation_size(ti.toast_table_oid)) AS toast_table_total_size_inc_index
FROM        toast_info ti;

```

Важно:

- TOAST-таблицы находятся в схеме `pg_toast`. [6][7]
- Если у таблицы нет TOAST-данных (или они очень малы), соответствующая TOAST-таблица и индекс могут отсутствовать или иметь нулевой размер.
- Функции `pg_relation_size`, `pg_table_size`, `pg_indexes_size` и `pg_total_relation_size` являются стандартными для получения информации о размерах объектов в PostgreSQL. [8][9]

Эти запросы должны помочь вам получить необходимую информацию о размерах TOAST-структур для ваших таблиц.

Learn more:

1. [Техника TOAST \(The Oversized-Attribute Storage Technique\) в PostgreSQL - Habr](#)
2. [PostgreSQL : Документация: 9.4: TOAST - Postgres Pro](#)
3. [TOAST в кластере Arenadata Postgres](#)
4. [15 полезных команд PostgreSQL - Tproger](#)
5. [Обслуживание базы данных PostgreSQL - LabEx](#)
6. [TOAST in PostgreSQL - Блог по архитектуре и оптимизации PostgreSQL от Аристова Евгения - Аристов Евгений](#)
7. [TOAST в Postgres - Базы данных, игры и другие колониальные товары](#)
8. [Сортировка таблиц в PostgreSQL по размеру: методы вывода - Skypro](#)
9. [PostgreSQL : Документация: 10: 9.26. Функции для системного администрирования - Postgres Pro](#)

5:18 PM

Thinking...

Defining the Query

Okay, first things first: I need to make it abundantly clear *why* someone might even *think* about looking into a `pg_toast` table. This isn't regular data access, so I need to frame it correctly. I'll start by explicitly discouraging direct querying for general use. It's a diagnostic, not a data retrieval, thing.

Clarifying Initial Approach

I'm starting with a clear warning: direct queries on `pg_toast` are generally a bad idea, unless you're doing specific diagnostics. My initial framing will be focused on why *not* to use them, before I even consider *how* someone might.

Прямой запрос к TOAST-таблице, такой как `pg_toast.pg_toast_17017342`, технически возможен, но **обычно не является полезным или рекомендуемым для получения самих данных в их исходном виде**. PostgreSQL управляет TOAST-таблицами автоматически, и их внутренняя структура предназначена для служебных целей.

Почему прямой запрос к TOAST-таблице не для получения данных:

- Внутренняя структура:** TOAST-таблицы хранят большие значения атрибутов, разбитые на "куски" (chunks). Данные в этих кусках могут быть сжаты и хранятся в бинарном формате. Вы не увидите там исходное текстовое или JSON-значение в читаемом виде.
- Отсутствие смысла без основной таблицы:** Данные в TOAST-таблице имеют смысл только в контексте основной таблицы, которая на них ссылается. TOAST-таблица содержит части данных, которые PostgreSQL собирает воедино, когда вы запрашиваете соответствующее поле из основной таблицы.
- Служебное использование:** Эти таблицы предназначены для внутреннего использования системой PostgreSQL. Прямое манипулирование ими или частые запросы к ним могут быть неэффективны и не несут практической пользы для большинства задач.

Что вы увидите, если сделаете запрос:

TOAST-таблица обычно имеет следующие столбцы:

- `chunk_id`: Идентификатор TOAST-значения (соответствует OID конкретного TOAST-объекта, на который ссылается основная таблица).
- `chunk_seq`: Порядковый номер куска внутри одного TOAST-значения. Большие значения разбиваются на несколько кусков, и этот столбец определяет их порядок.
- `chunk_data`: Сами данные куска в бинарном формате (`bytea`). Это могут быть сжатые данные.

Как можно сделать запрос (в исследовательских или диагностических целях):

Если вы все же хотите посмотреть на структуру или содержимое TOAST-таблицы (например, для очень глубокой диагностики или из академического интереса), вы можете выполнить стандартный SQL-запрос. Вам, скорее всего, потребуются права суперпользователя или соответствующие разрешения на доступ к схеме `pg_toast`.

sql

```
SELECT chunk_id, chunk_seq, length(chunk_data) AS chunk_data_length, chunk_data
FROM pg_toast.pg_toast_17017342
LIMIT 10; -- Ограничьте вывод, так как таблица может быть большой
```

Что означает `pg_toast_17017342`?

- **pg_toast**: Это системная схема, где хранятся все TOAST-таблицы.
- **pg_toast_17017342**: Имя самой TOAST-таблицы. Число **17017342** — это OID (Object Identifier) **основной таблицы**, для которой была создана эта TOAST-таблица. Вы можете найти имя основной таблицы по этому OID:

sql

```
SELECT oid::regclass AS table_name
FROM pg_class
WHERE oid =17017342;
```

Когда может понадобиться информация о TOAST-таблице (не прямой запрос к данным):

- **Определение размера TOAST-данных:** Как мы обсуждали ранее, для оценки, сколько места занимают TOAST-значения конкретной таблицы.
- **Диагностика раздувания (bloat):** Иногда проблемы с производительностью могут быть связаны с раздуванием TOAST-таблиц.
- **Понимание механизмов хранения PostgreSQL:** Для глубокого изучения того, как PostgreSQL работает с большими объектами.

Итог:

- **Не запрашивайте TOAST-таблицы для получения фактических данных приложения.** Для этого всегда обращайтесь к основной таблице и ее столбцам. PostgreSQL автоматически "соберет" TOAST-значения за вас.
- Прямые запросы к TOAST-таблицам могут быть полезны для административных задач, диагностики или глубокого анализа, но не для повседневной работы с данными.
- Содержимое столбца **chunk_data** будет бинарным и, вероятно, сжатым, поэтому оно не будет напрямую читаемым без специальной обработки, которую PostgreSQL выполняет внутренне.

5:23 PM

You cleared the context

Thinking...

Investigating TOAST Tables

Okay, first things first: I need to figure out how to pinpoint tables in PostgreSQL that are using TOAST. Then, I should look at how to deal with the common issues related to them. I'll start with some basic queries.

Querying TOAST Tables

Right, let's get this rolling. I'm focusing on pinpointing tables using TOAST in PostgreSQL, looking for the best queries to expose the biggest users of this storage mechanism. Size is everything here, so I'll be prioritizing results by size.

Чтобы найти таблицы с самыми большими TOAST-данными в PostgreSQL, вы можете использовать следующий SQL-запрос. TOAST (The Oversized Attribute Storage Technique) — это механизм, который PostgreSQL использует для хранения больших значений столбцов (например, **TEXT**, **BYTEA**, **JSONB**) путем их разделения на более мелкие части и хранения в отдельной TOAST-таблице. [1][2]

SQL-запрос для поиска таблиц с самыми большими TOAST-данными:

sql

```

SELECT    c.relname AS table_name,
          t.relname AS toast_table_name,
          pg_size_pretty(pg_total_relation_size(t.oid)) AS toast_table_size_pretty,
          pg_total_relation_size(t.oid) AS toast_table_size_bytes
FROM      pg_class c
JOIN      pg_namespace n ON n.oid = c.relnamespace
LEFTJOIN  pg_class t ON c.reltoastrelid = t.oid
WHERE     c.relkind IN ('r', 'm') -- 'r' для обычных таблиц, 'm' для материализованных представленийAND
t.oid ISNOTNULL-- Убедимся, что TOAST-таблица существуетAND n.nspname NOTIN ('pg_catalog',
'information_schema') -- Исключаем системные схемыAND n.nspname NOTLIKE 'pg_toast%'-- Исключаем сами
TOAST-схемыORDERBY    toast_table_size_bytes DESC;

```

Разбор запроса:

- **pg_class c**: Системный каталог, содержащий информацию о таблицах и других объектах базы данных. [3] Мы используем его для получения имени основной таблицы (**c.relname**).
- **pg_namespace n**: Системный каталог, содержащий информацию о схемах. Мы присоединяем его, чтобы фильтровать по имени схемы (**n.nspname**).
- **pg_class t**: Еще один экземпляр **pg_class**, используемый для получения информации о TOAST-таблице. Связь между основной таблицей и ее TOAST-таблицей осуществляется через **c.reltoastrelid = t.oid**. [4]
- **c.relkind IN ('r', 'm')**: Фильтр для выбора только обычных таблиц ('r') и материализованных представлений ('m'). [3]
- **t.oid IS NOT NULL**: Условие, гарантирующее, что для таблицы существует связанная TOAST-таблица.
- **n.nspname NOT IN ('pg_catalog', 'information_schema')**: Исключает системные таблицы из результатов. [3][5]
- **n.nspname NOT LIKE 'pg_toast%'**: Исключает таблицы, находящиеся в схемах, имена которых начинаются с **pg_toast**. [3][5]
- **pg_total_relation_size(t.oid)**: Функция, возвращающая общий размер TOAST-таблицы в байтах, включая ее индексы. [6][7]
- **pg_size_pretty(...)**: Функция, форматирующая размер в байтах в человекочитаемый формат (например, KB, MB, GB). [6][7]
- **ORDER BY toast_table_size_bytes DESC**: Сортирует результаты по размеру TOAST-таблицы в байтах в порядке убывания, чтобы самые большие таблицы были наверху.

Как это работает:

1. Запрос выбирает основные таблицы (**c**) и их схемы (**n**).
2. Затем он присоединяет соответствующие TOAST-таблицы (**t**), используя поле **reltoastrelid** из **pg_class**, которое хранит OID TOAST-таблицы. [4]
3. Фильтрует результаты, чтобы включить только пользовательские таблицы и материализованные представления, у которых есть TOAST-таблицы.
4. Рассчитывает общий размер каждой TOAST-таблицы (включая ее собственный индекс). [4][6]
5. Отображает имя основной таблицы, имя TOAST-таблицы и ее размер в удобном для чтения формате и в байтах.
6. Сортирует результаты так, чтобы таблицы с самыми большими TOAST-данными отображались первыми.

Выполнение этого запроса в вашей базе данных PostgreSQL предоставит вам список таблиц, отсортированных по размеру их TOAST-хранилищ, что поможет определить, какие из них занимают больше всего места из-за больших значений столбцов. [2]

Learn more:

1. [Unlocking the Secrets of TOAST: How To Optimize Large Column Storage in PostgreSQL for Top Performance and Scalability - Percona](#)
2. [Postgres TOAST: The Greatest Thing Since Sliced Bread? | Crunchy Data Blog](#)

3. [Calculating Table Size in PostgreSQL - Christian Emmer](#)
4. [Documentation: 17: 65.2. TOAST - PostgreSQL](#)
5. [PostgreSQL – Get the Top 20 Biggest Tables - SQL Conjuror](#)
6. [How to Analyze PostgreSQL Table Size - SimpleBackups](#)
7. [postgresql list and order tables by size - Stack Overflow](#)

5:28 PM

Thinking...

Examining Field Sizes

Okay, I'm starting to consider the sheer scale of the data. The user's request means I need to pinpoint the widest fields, which are likely the most memory-intensive. This requires a good strategy for quickly scanning tables and identifying the largest columns.

Pinpointing TOAST Columns

Right, the user's focus is on *columns* and their TOAST usage now. That changes the approach. I have to query for individual columns, determine their TOAST eligibility, and then figure out how to estimate the storage used by each. Direct measurement looks like a non-starter.

Assessing Approaches for Columns

Now, I'm evaluating the best path forward. The user needs column-specific TOAST size insights. Direct measurement is out. I'll focus on `octet_length()` and similar size estimations. This should help pinpoint the biggest, most TOAST-dependent columns for each table.

Refining the Strategy Again

Right, now I'm revisiting the core objective: the *largest fields*, inclusive of TOAST. The user's request boils down to identifying those columns that, when TOASTed, consume the most storage. My current approach of using `octet_length()` and related methods is appropriate, but I have to ensure the results reflect the impact of TOAST itself.

Evaluating Impact of TOAST

Okay, the user's request is about finding the *actual stored size* including TOAST data per column, a more complex problem. I'll need to identify TOAST-eligible columns, then estimate their TOAST contribution. Direct measurement is effectively impossible, given the design of TOAST.

Refining Measurement Approach

The user's need for the *actual stored size* including TOAST presents a tricky problem. While direct measurement is out, I can provide a valuable estimate. I'll combine the identification of potentially TOASTed columns with their average/maximum uncompressed sizes, providing insight into TOAST contribution.

Revisiting TOAST Estimation

The user specifically wants *actual stored size*, including TOAST data per column. This is a significant challenge. While precise measurement is out, I can identify TOAST-eligible columns and use `octet_length()` to estimate their uncompressed

size, thereby estimating the impact of TOAST.

Refining Data Analysis Strategy

Now, I'm refining my approach. The user wants to identify the largest fields, including TOASTed size, for each table, which is more complex than simply identifying the table's total TOAST size. The need for a query that includes actual stored size requires careful consideration of column types, uncompressed sizes using `octet_length()`, and potentially TOAST contributions, given the constraints of directly measuring TOASTed sizes.

Estimating with Uncompressed Size

My approach is evolving to focus on providing an *estimate* of TOAST impact. The user needs to find the largest fields including TOAST, so I'll identify TOAST-eligible columns. Then, I can estimate the uncompressed size to understand potential TOAST contribution.

Constructing Dynamic Queries

Now, I'm tackling the problem of actually retrieving the data for the `octet_length()` and `pg_column_size()`. The best approach seems to be a dynamic query that can process individual columns. The user needs a template, I guess.

Reassessing the Approach

The user wants *actual* stored size, including TOAST data per column. This remains challenging. I'm aiming to provide an estimate by focusing on potentially TOASTed columns and using uncompressed size estimations. I need to present a query and some guidance.

Summarizing and Simplifying

I'm focusing on simplifying the user's request. They need to pinpoint *large fields*, including TOAST data per column. Since direct measurement is difficult, I'll provide a two-step approach: first identifying tables with large TOAST segments, then providing an analysis of individual columns.

Defining TOAST Contribution

My focus remains on the user's objective: determining the largest fields, including the TOASTed part. Directly measuring TOAST per column remains infeasible. I must define a practical estimate.

Summarizing the Strategy

Okay, I'm pivoting the approach. The user needs to pinpoint the *largest* fields, accounting for TOAST. Since direct size measurement is impossible, I'll highlight columns likely to contribute to TOAST and then guide the user on how to approximate their sizes and therefore their TOAST contribution.

Finalizing the Approach

Okay, I'm settling on a final strategy. The user's need for the *largest fields including TOAST* size is challenging. I'll provide an explanation of the limitations. I'll then offer a query to identify tables with a lot of TOASTed data. Finally, I'll offer a PL/pgSQL script to estimate uncompressed column sizes.

Clarifying the Goal

My focus now is refining the explanation. The user wants to identify large fields and the *impact of TOAST*. Direct TOAST size per column is hard. I must emphasize the approach: uncompressed size is the key indicator.

Defining Problem and Strategy

My focus is now on defining the user's request: finding the *largest fields*, including TOASTed data. Since per-column TOAST measurement is complex, I'll show how to identify TOAST-eligible columns and then estimate their impact by focusing on uncompressed size.

Defining the Core Approach

Okay, here's my final strategy. The user requests to pinpoint the largest fields including TOAST data. While I cannot directly measure the TOAST per column, I can provide a solid estimate based on uncompressed size, and then guide the user.

Summarizing the Solution

My focus is on providing the user with the right tools. I'll give an overview of the challenges, then a query to identify potentially TOASTed columns. Finally, I'll present a PL/pgSQL script to estimate uncompressed sizes, a good indicator of TOAST impact, enabling them to pinpoint large fields.

text

Finalizing the Solution

I'm aiming for a complete solution. The user wants to know about the **largest fields**, and that includes TOAST. I can't get a direct TOAST-size, so I'll offer a query to find potentially TOASTed columns and a PL/pgSQL script that provides average/max uncompressed sizes.

Providing a Complete Solution

The user is after the **largest fields** with TOAST in mind. Providing an exact TOAST size per column isn't possible, so I've worked up a full solution. It includes identifying potentially TOASTed columns and a PL/pgSQL script that calculates **uncompressed** sizes.

```text

### ### Providing the Solution

My focus is now on providing the right tools to address the user's needs. I'll describe the inherent challenges, and then offer a query to locate potentially TOASTed columns, alongside a PL/pgSQL script to estimate uncompressed column sizes, offering insight into TOAST impact, and thus help the user pinpoint those large fields.

My approach is now complete. The user is requesting the largest fields, accounting for TOAST. Direct measurement is unavailable. I'll provide a complete solution: a table-level TOAST size query, and a PL/pgSQL script to estimate the uncompressed size of columns, helping the user to determine the *largest fields*.

Here's my summary of the thought process:

text

### ### Delivering the Final Answer

I'm now ready to give the user a solid solution. They need to identify the *\*largest fields\**, inclusive of the impact of TOAST. While I can't directly measure per-column TOAST size, I'll provide the right queries and explain my approach.

Понять, какие конкретно поля (столбцы) в таблицах занимают больше всего места, *включая их TOAST-хранилище*, — задача нетривиальная. PostgreSQL не отслеживает напрямую, какая часть TOAST-таблицы принадлежит какому конкретно столбцу основной таблицы. TOAST-таблица хранит данные для всех TOAST-активных столбцов таблицы в сжатом и разделенном на части виде.

Однако мы можем определить, какие столбцы содержат наибольший объем *необработанных (несжатых)* данных, что напрямую влияет на размер используемого TOAST-хранилища. Столбцы с большим объемом таких данных будут основными "виновниками" большого размера TOAST-сегмента таблицы.

Вот двухэтапный подход для этого:

**Шаг 1: Определите таблицы с наибольшим общим размером TOAST-хранилища**

Сначала используйте запрос из предыдущего ответа (или приведенный ниже), чтобы найти таблицы, у которых TOAST-сегменты в целом самые большие. Это поможет сузить круг поиска.

sql

```
SELECT c.relname AS table_name,
 n.nspname AS schema_name,
 t.relname AS toast_table_name,
 pg_size_pretty(pg_total_relation_size(t.oid)) AS toast_table_size_pretty,
 pg_total_relation_size(t.oid) AS toast_table_size_bytes
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
LEFTJOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm') -- 'r' для обычных таблиц, 'm' для материализованных представлений AND
t.oid ISNOTNULL -- Убедимся, что TOAST-таблица существует AND n.nspname NOTIN ('pg_catalog',
'information_schema') -- Исключаем системные схемы AND n.nspname NOTLIKE 'pg_toast%' -- Исключаем сами
TOAST-схемы ORDERBY toast_table_size_bytes DESC;
```

## Шаг 2: Проанализируйте столбцы в выбранных таблицах на предмет размера их данных

После того как вы определили таблицы с большими TOAST-сегментами, вы можете использовать следующий PL/pgSQL скрипт для анализа столбцов *внутри конкретной таблицы*. Этот скрипт рассчитает максимальный и средний *несжатый* размер данных для каждого потенциально TOAST-активного столбца.

**PL/pgSQL скрипт для анализа столбцов таблицы:**

plpgsql

```

DO $$
DECLARE
 r RECORD;
 query TEXT;
 max_len_pretty TEXT;
 avg_len_pretty TEXT;
 max_len_bytes BIGINT;
 avg_len_bytes NUMERIC;
 -- =====
 -- >> ПАРАМЕТРЫ: Укажите схему и имя таблицы для анализа <<
 table_schema_param TEXT := 'public'; -- ЗАМЕНИТЕ НА ИМЯ ВАШЕЙ СХЕМЫ
 table_name_param TEXT := 'your_table_name'; -- ЗАМЕНИТЕ НА ИМЯ ВАШЕЙ ТАБЛИЦЫ
 -- =====
BEGIN
 RAISE NOTICE 'Анализ таблицы: %.', quote_ident(table_schema_param), quote_ident(table_name_param);
 RAISE NOTICE '-----';
';
 RAISE NOTICE 'Имя столбца | Тип | Макс. несжатый размер | Сред. несжатый размер';
 RAISE NOTICE '-----|-----|-----|-----';

 FOR r IN
 SELECT
 a.attname AS column_name,
 t.typname AS column_type
 FROM
 pg_class c
 JOIN
 pg_namespace n ON n.oid = c.relnamespace
 JOIN
 pg_attribute a ON a.attrelid = c.oid
 JOIN
 pg_type t ON t.oid = a.atttypid
 WHERE
 c.relkind IN ('r', 'm') -- Обычные таблицы и материализованные представления
 AND n.nspname = table_schema_param
 AND c.relname = table_name_param
 AND a.attnum > 0 -- Исключаем системные столбцы
 AND NOT a.attisdropped -- Исключаем удаленные столбцы
 -- Выбираем типы, которые обычно подвергаются TOAST или могут быть большими
 AND (t.typname IN ('text', 'varchar', 'bpchar', 'bytea', 'json', 'jsonb', 'xml')
 OR t.typtype = 'A' -- Любой тип массива
 OR t.typtype IN ('S', 'X', 'J', 'V', 'U') -- Строки, XML, JSON, битовые строки,
пользовательские типы (часто текстовые)
)
 -- Проверяем, есть ли в столбце вероятно ненулевые данные (для оптимизации)
 AND EXISTS (
 SELECT 1
 FROM pg_catalog.pg_stats ps
 WHERE ps.schemaname = table_schema_param
 AND ps.tablename = table_name_param
 AND ps.attname = a.attname
 AND ps.null_frac < 1.0 -- Доля NULL меньше 100%
)
 AND c.reltuples > 0 -- Только если в таблице есть строки
 LOOP
 -- Для типа bytea используем octet_length напрямую
 IF r.column_type = 'bytea' THEN
 query := format(
 'SELECT pg_size_pretty(MAX(octet_length(%I))), ' ||
 ' pg_size_pretty(AVG(octet_length(%I))), ' ||
 ' MAX(octet_length(%I)), ' ||
 ' AVG(octet_length(%I)) ' ||
 'FROM %I.%I WHERE %I IS NOT NULL',
 r.column_name, r.column_name, r.column_name, r.column_name,
 table_schema_param, table_name_param, r.column_name
);
 ELSE
 -- Для других типов (text, json, jsonb, массивы и т.д.) приводим к text для получения длины
 -- сериализованного представления.
 -- Это аппроксимация для сложных типов, но дает хорошее представление о размере.

```

```

 query := format(
 'SELECT pg_size_pretty(MAX(octet_length(%I::text))), ' ||
 ' pg_size_pretty(AVG(octet_length(%I::text))), ' ||
 ' MAX(octet_length(%I::text)), ' ||
 ' AVG(octet_length(%I::text)) ' ||
 'FROM %I.%I WHERE %I IS NOT NULL',
 r.column_name, r.column_name, r.column_name, r.column_name,
 table_schema_param, table_name_param, r.column_name
);
 END IF;

BEGIN
 EXECUTE query INTO max_len_pretty, avg_len_pretty, max_len_bytes, avg_len_bytes;

 IF max_len_bytes IS NOT NULL THEN
 RAISE NOTICE '%|%|%',
 rpad(r.column_name, 18),
 rpad(r.column_type, 10),
 rpad(COALESCE(max_len_pretty, 'N/A'), 22),
 COALESCE(avg_len_pretty, 'N/A');
 ELSE
 RAISE NOTICE '%|%' (Нет данных > NULL) | (Нет данных > NULL)',
 rpad(r.column_name, 18),
 rpad(r.column_type, 10);
 END IF;
EXCEPTION
 WHEN undefined_function THEN -- Ошибка, если octet_length неприменим (маловероятно с
::text)
 RAISE WARNING 'Не удалось применить octet_length к столбцу % (%.) типа %. Пропуск.',
 r.column_name, table_schema_param, table_name_param, r.column_type;
 WHEN others THEN -- Другие возможные ошибки при выполнении динамического SQL
 RAISE WARNING 'Ошибка при обработке столбца % (%.): %', r.column_name,
 table_schema_param, table_name_param, SQLERRM;
 END;
END LOOP;
RAISE NOTICE '-----';
';
RAISE NOTICE 'Примечание: Размеры указаны для несжатых данных. Большие несжатые размеры ведут к
использованию TOAST.';
RAISE NOTICE 'Используйте первый запрос (Шаг 1) для просмотра общего размера TOAST-таблиц.';
END $$;

```

## Как использовать скрипт (Шаг 2):

1. **Определите таблицу для анализа:** По результатам Шага 1 выберите таблицу (например, `public.my_large_table`).

2. **Измените параметры в скрипте:** В начале PL/pgSQL блока найдите строки:

```
plpgsql
```

```

table_schema_param TEXT := 'public'; -- ЗАМЕНИТЕ НА ИМЯ ВАШЕЙ СХЕМЫ
table_name_param TEXT := 'your_table_name'; -- ЗАМЕНИТЕ НА ИМЯ ВАШЕЙ ТАБЛИЦЫ

```

Замените `'public'` и `'your_table_name'` на актуальные имя схемы и имя таблицы, которую вы хотите проанализировать.

3. **Выполните скрипт:** Запустите измененный DO-блок в вашем SQL-клиенте (например, psql, DBeaver, pgAdmin).

4. **Интерпретируйте результаты:** Скрипт выведет список столбцов указанной таблицы, их типы, а также максимальный и средний размер их *несжатых* данных. Столбцы с наибольшими значениями **Макс. несжатый размер** и **Сред. несжатый размер** являются основными кандидатами, вносящими наибольший вклад в TOAST-хранилище этой таблицы.

## Важно понимать:

- Скрипт измеряет `octet_length` (длину в байтах) данных столбца. Для текстовых типов это количество байт в текущей кодировке базы данных. Для `json`, `jsonb`, `xml` и массивов данные сначала преобразуются в их текстовое представление, а затем измеряется их длина, что дает хорошую оценку "сырого" объема данных.
- Эти размеры представляют данные до сжатия и разделения на части механизмом TOAST. Однако именно эти большие "сырые" данные и являются причиной активного использования TOAST.
- Выполнение этого скрипта для таблиц с очень большим количеством строк и очень широкими столбцами может быть ресурсоемким, так как он будет сканировать данные столбцов. Рекомендуется запускать его в периоды низкой нагрузки.

Этот подход позволит вам выявить "самые большие поля таблицы, включая toast" в том смысле, что вы найдете поля, которые из-за своего большого исходного размера наиболее активно используют механизм TOAST и вносят наибольший вклад в размер соответствующей TOAST-таблицы.

5:35 PM

*Thinking...*

Consider: Text Conversion in Postgres

Okay, converting text to lowercase in PostgreSQL. Seems straightforward. I'll focus on the most efficient methods, probably utilizing built-in functions. The core concept will involve the `LOWER()` function, but I should consider potential edge cases and performance implications.

## Analyzing Text Conversion

Okay, the core function is `LOWER()`. Simple enough, though I'll need to consider the nuances of character sets and collation to ensure it functions correctly across various data scenarios.

В PostgreSQL для преобразования строк в нижний регистр используется функция `LOWER()`.

### Синтаксис:

sql

`LOWER(string)`

Где `string` — это текстовое значение или столбец, который вы хотите преобразовать.

### Примеры:

#### 1. Преобразование строкового литерала:

sql

```
SELECT LOWER('Привет МИР!');
```

Результат:

привет мир!



## 2. Преобразование данных из столбца таблицы:

Предположим, у вас есть таблица `users` со столбцом `username`:

sql

```
CREATETABLE users (
 id SERIAL PRIMARY KEY,
 username VARCHAR(50)
);

INSERTINTO users (username) VALUES ('JohnDoe'),
 ('JaneDOE'),
 ('PETERPAN');
```

Чтобы выбрать имена пользователей в нижнем регистре:

sql

```
SELECT username, LOWER(username) AS username_lowercase
FROM users;
```

Результат:

asciidoc

```
username | username_lowercase
-----+----- JohnDoe | johndoe JaneDOE | janedoe PETERPAN | peterpan(3
rows)
```

## 3. Использование в условии `WHERE` (для регистронезависимого сравнения):

Если вы хотите найти пользователей, не обращая внимания на регистр символов в имени:

sql

```
SELECT*FROM users
WHERELOWER(username) = 'johndoe';
```

Этот запрос найдет и 'JohnDoe', и 'johndoe', и 'JOHNDOE' и т.д.

### Важные моменты:

- Функция `LOWER()` работает с различными текстовыми типами данных, такими как `TEXT`, `VARCHAR`, `CHAR`.
- Преобразование зависит от правил локали (locale) вашей базы данных, особенно для символов, имеющих разные формы в нижнем регистре в разных языках. Однако для большинства стандартных случаев (например, латиница, кириллица) она работает ожидаемо.
- Если вам нужно регистронезависимое сравнение, использование `LOWER()` с обеих сторон сравнения (`LOWER(column) = LOWER(value)`) — это распространенный подход. Альтернативой для некоторых типов данных и операций может быть использование оператора `ILIKE` (для `LIKE`-подобных сравнений) или регистронезависимых типов данных (citext).

Функция `LOWER()` является стандартной SQL-функцией и широко используется для приведения строк к нижнему регистру.

5:43 PM

"max\_parallel\_workers\_per\_gather" "2" "Resource Usage / Asynchronous Behavior" "Sets the maximum number of parallel processes per executor node."  
детально

6:02 PM  
*Thinking...*

## Analyzing Parallel Execution Parameters

I begin by breaking down the request. The key is `max_parallel_workers_per_gather`, its default, category, and a short description. The user demands detail, so my task involves delving into its purpose, mechanics, implications, and related parameters.

### Defining The Core Concepts

I'm mapping out the core components of parallel query execution: leader process, worker processes, Gather node, and the parallel plan itself. This is crucial for understanding the *per-node* limitation that `max_parallel_workers_per_gather` enforces. I have to ensure this context is crystal clear for the detailed explanation.

### Processing Detailed Breakdown

Okay, I'm refining the "How it works?" section. I need to explicitly mention the role of the `Gather Merge` node, and clarify the differences between parallel workers initiated by a Gather node, and the global `max_worker_processes` limit. The user requires clarity on the nuanced interactions of these components.

### Revisiting The Structure

I'm now solidifying the structure. My focus is on ensuring a logical and complete flow. I'll expand each section, integrating the brainstorming, and mapping to ensure the user's detailed requirement is met.

### Checking & Consolidating Details

I'm now cross-referencing. I'll verify that each parameter and concept is precisely defined, ensuring seamless flow and clarity. I need to double-check the implications section. The user needs all the nuances.

### Analyzing the Scope and Details

I'm now diving deep into the user's requirements, beginning with the detailed explanation of `max_parallel_workers_per_gather`. This involves unpacking all the implications and cross-referencing the setting with relevant parameters.

### Examining Parameter Relationships

I'm now exploring the interdependencies between `max_parallel_workers_per_gather` and related parameters like `max_worker_processes`. The user's need for a detailed explanation requires this, to clarify potential bottlenecks.

### Processing: Identifying the Structure

I'm building the framework now, using the plan outlined. Each section, starting with "What it is," is essential. I need to ensure clarity and flow for the user. I'm focusing on the core elements, and the user's requirements.

Хорошо, давайте детально разберем параметр `max_parallel_workers_per_gather` в PostgreSQL.

**Название параметра:** `max_parallel_workers_per_gather`

**Значение по умолчанию (типичное):** "2" (как в вашем примере, но может отличаться в зависимости от версии PostgreSQL и конфигурации)

**Категория:** "Resource Usage / Asynchronous Behavior" (Использование ресурсов / Асинхронное поведение)

**Краткое описание:** "Sets the maximum number of parallel processes per executor node." (Устанавливает максимальное количество параллельных процессов на один узел исполнителя (Gather)).

**Детальное объяснение:**

### 1. Что такое параллельные запросы в PostgreSQL?

PostgreSQL может выполнять некоторые части SQL-запроса параллельно, используя несколько процессорных ядер одновременно. Это может значительно ускорить выполнение сложных запросов, особенно тех, которые обрабатывают большие объемы данных (например, сканирование больших таблиц, агрегации, соединения).

Когда планировщик PostgreSQL решает, что запрос может выиграть от параллельного выполнения, он создает план запроса, который включает специальные узлы, называемые **Gather** или **Gather Merge**.

- **Лидер (Leader):** Основной процесс (backend), который инициировал запрос, становится "лидером".
- **Рабочие процессы (Workers):** Лидер может запрашивать у системы запуск одного или нескольких дополнительных "рабочих" процессов. Эти рабочие процессы выполняют часть работы параллельно.
- **Узел Gather/Gather Merge:** Это точка в плане запроса, где результаты, полученные от параллельных рабочих процессов, собираются вместе лидером (или одним из рабочих процессов, который затем передает их лидеру).

### 2. Роль `max_parallel_workers_per_gather`:

Параметр `max_parallel_workers_per_gather` контролирует, **сколько максимально параллельных рабочих процессов может быть запрошено и использовано одним конкретным узлом Gather (или Gather Merge)** в плане выполнения запроса.

- **Не глобальный лимит:** Важно понимать, что это **не** общее максимальное количество параллельных рабочих процессов для всего запроса или для всей системы. Один запрос может иметь несколько узлов Gather (хотя это менее распространено для простых запросов), и каждый из них будет ограничен этим параметром.
- **Ограничение на узел:** Если планировщик решает, что для определенной операции (например, параллельного сканирования таблицы) было бы полезно использовать, скажем, 4 рабочих процесса, но `max_parallel_workers_per_gather` установлен в 2, то для этого конкретного узла Gather будет запрошено и запущено не более 2 рабочих процессов.

### 3. Как это работает на практике?

1. **Планирование:** Планировщик запросов анализирует запрос и решает, можно ли его распараллелить. Если да, он оценивает, сколько рабочих процессов было бы оптимально для различных частей запроса.
2. **Запрос воркеров:** Когда выполнение доходит до узла Gather, лидер пытается запустить параллельные рабочие процессы. Количество запрашиваемых воркеров будет наименьшим из:
  - Количества, которое планировщик считает оптимальным для данной операции.
  - Значения `max_parallel_workers_per_gather`.
  - Количества доступных в данный момент параллельных рабочих процессов в общем пуле (ограниченном `max_worker_processes` и другими факторами).
3. **Выполнение:** Запущенные рабочие процессы выполняют свою часть задачи.
4. **Сбор результатов:** Узел Gather собирает результаты от всех успешно запущенных рабочих процессов и передает их дальше по плану выполнения.

#### 4. Цель и назначение параметра:

- **Контроль ресурсов:** Позволяет администраторам баз данных ограничивать количество ресурсов (в данном случае, процессов и, следовательно, CPU/памяти), которые может потреблять одна операция параллельного выполнения внутри запроса. Это помогает предотвратить ситуацию, когда один очень "жадный" узел Gather в сложном запросе пытается захватить слишком много рабочих процессов, потенциально замедляя другие операции или запросы.
- **Предотвращение избыточных накладных расходов:** Запуск и координация большого количества параллельных процессов сами по себе создают накладные расходы. Для некоторых операций слишком большое количество воркеров может привести к уменьшению общей производительности из-за этих накладных расходов. Этот параметр помогает ограничить это.
- **Тонкая настройка производительности:** Дает возможность более гранулярно управлять параллелизмом.

#### 5. Влияние значения:

- **Увеличение `max_parallel_workers_per_gather`:**
  - **Потенциал:** Может позволить более полное использование доступных процессорных ядер для отдельных операций в запросе, что может привести к их более быстрому выполнению.
  - **Риски:**
    - Увеличение потребления CPU и, возможно, памяти.
    - Если общее количество рабочих процессов (`max_worker_processes`) невелико, один узел Gather может "съесть" всех доступных воркеров, не оставив их для других параллельных операций в том же или других запросах.
    - Увеличение накладных расходов на координацию, если количество воркеров становится слишком большим для объема работы.
- **Уменьшение `max_parallel_workers_per_gather`:**
  - **Потенциал:**
    - Снижает пиковую нагрузку на CPU от отдельных параллельных операций.
    - Оставляет больше доступных рабочих процессов для других операций или запросов.
  - **Риски:** Может ограничить степень параллелизма для операций, которые могли бы эффективно использовать больше воркеров, тем самым замедляя их.

#### 6. Когда изменять значение?

Значение по умолчанию (обычно 2) является разумным компромиссом для многих рабочих нагрузок. Изменять его стоит обдуманно:

- **Увеличивать:**
  - Если у вас есть многоядерный сервер, и вы видите, что во время выполнения тяжелых запросов с параллельными планами CPU недогружен, а узлы Gather используют меньше воркеров, чем доступно ядер.
  - Если вы уверены, что конкретные запросы (например, аналитические запросы на больших таблицах) выиграют от большего числа воркеров для отдельных сканирований или агрегаций, и у вас достаточно `max_worker_processes`.
  - После тщательного тестирования и мониторинга.
- **Уменьшать:**
  - Если параллельные запросы вызывают чрезмерную нагрузку на CPU, и вы хотите ограничить "агрессивность" отдельных параллельных операций.
  - Если накладные расходы на запуск большого количества воркеров для конкретных операций кажутся выше, чем выгода от параллелизма.

## 7. Связанные параметры:

- `max_worker_processes`: Общее максимальное количество фоновых процессов, которые может запустить PostgreSQL, включая параллельные рабочие процессы, процессы автовакуума и т.д. Общее количество используемых параллельных воркеров (по всем Gather узлам всех запросов) не может превысить это значение (минус другие фоновые процессы).
- `max_parallel_maintenance_workers`: Максимальное количество параллельных рабочих процессов, которые могут быть запущены одной служебной командой (например, `CREATE INDEX`).
- `parallel_setup_cost`: Оценка планировщиком стоимости запуска параллельных процессов.
- `parallel_tuple_cost`: Оценка планировщиком стоимости передачи одной строки от рабочего процесса к лидеру.
- `min_parallel_table_scan_size` / `min_parallel_index_scan_size`: Минимальный размер таблицы/индекса, для которого планировщик будет рассматривать параллельное сканирование.
- `force_parallel_mode`: (Для отладки) Может заставить планировщик использовать параллельные планы, даже если он считает их невыгодными.

## 8. Как проверить и установить:

- **Проверить текущее значение:**

```
sql
```

```
SHOW max_parallel_workers_per_gather;
```

- **Установить значение (требуется прав суперпользователя):**

- **Для текущей сессии:**

```
sql
```

```
SET max_parallel_workers_per_gather =4;
```

- **Глобально для сервера (изменение в `postgresql.conf`):**

Найдите строку `max_parallel_workers_per_gather` в файле `postgresql.conf`, измените значение и перезагрузите конфигурацию сервера (`SELECT pg_reload_conf();` или перезапустите сервер).

- **Глобально через SQL (изменяет `postgresql.auto.conf`):**

```
sql
```

```
ALTERSYSTEMSET max_parallel_workers_per_gather =4;
SELECT pg_reload_conf(); -- Применить изменения
```

## Заключение:

`max_parallel_workers_per_gather` — это важный параметр для тонкой настройки производительности параллельных запросов в PostgreSQL. Он позволяет контролировать, насколько "широко" может распараллеливаться отдельная операция внутри запроса. Изменение этого параметра требует понимания вашей рабочей нагрузки, аппаратных ресурсов и тщательного тестирования, чтобы найти оптимальное значение для вашей системы.

6:02 PM

добавить настройки с комментариями для 48 ядерного процессора

```
BEGIN;
SET parallel_setup_cost = 0;
SET parallel_tuple_cost = 0;
SET max_parallel_workers_per_gather = 6;
SET maintenance_work_mem = '256MB';
SET min_parallel_table_scan_size = 1;
```

pgsql

```
WITH deleted_rows AS (DELETEFROMpublic.messages WHERE created_at < 1739998800RETURNING0) SELECT
count(*) FROM deleted_rows;
```

```
COMMIT;
```

6:12 PM

*Thinking...*



## Processing PostgreSQL Configurations

Right, time to delve into the specifics. I need to optimize PostgreSQL for this 48-core beast. Initial thoughts: shared buffers, effective cache size – those are always the starting points. Then we'll tweak `work_mem` and `autovacuum`. Gotta find the perfect balance to avoid contention.

### Examining Parallel Setup Costs

---

Okay, focusing on the `parallel_setup_cost`. I'm assessing if aggressively lowering it is the right move for this specific delete operation on a 48-core system. The potential benefits, vs. the risk of over-parallelization, are what I'm weighing up.

#### Analyzing the Parameters

Okay, I'm now dissecting the provided parameters. Initial impressions: setting `parallel_setup_cost` and `parallel_tuple_cost` to 0 is a bold move. While the 48 cores offer ample processing power, the overhead of parallelization is crucial. I need to see if the aggressive settings will pay off.

### Deconstructing the Configuration

---

Okay, let's break this down. First, `parallel_setup_cost = 0` and `parallel_tuple_cost = 0` are bold moves, aiming for aggressive parallelism. My initial thought is to test these settings thoroughly, since the delete operation might not benefit as much from parallelism compared to a `SELECT`.

### Reviewing Parameter Choices

---

Okay, now I'm circling back to the whole picture, thinking about what's been set so far. I have to remember the `DELETE` doesn't always benefit from parallelism like a `SELECT` does. The goal is to find the ideal balance, which means more testing is needed.

### Reviewing the Implications

---

Okay, time to synthesize all this. Aggressive parallel settings, but the `DELETE` itself might not benefit as much. The `WHERE` clause *could*, though. I'll focus testing and measuring the impact. I also need to consider the downstream effects on `VACUUM`.

### Refining the Assessment

---

Okay, I'm now making sure I cover all the bases. The `DELETE` is the focus, and the parameters here are meant to influence the `WHERE` clause evaluation. I'll test how well the parallel settings perform. I'll note downstream effects like `VACUUM`.

#### Examine and Evaluate

I'm going to analyze each parameter, especially considering the aggressive settings for a 48-core system. The **DELETE** operation itself is less likely to benefit, but the **WHERE** clause *might*. I will remember that the goal is to minimize overhead and to ensure my choices support quick deletion and efficient vacuuming.

#### Analyzing and Planning Configuration

I'm now assessing the parameters provided. My immediate thought is the aggressive **parallel** settings warrant close scrutiny. I must understand how they affect the deletion operation and any potential impact on the **WHERE** clause evaluation.

#### Examining Scan Phase Optimization

I'm zeroing in on the scan phase of the **DELETE**, where the parallelism will be relevant. I'll now focus on the **WHERE** clause to optimize finding rows for deletion on the table and on the implications for the VACUUM following it.

#### Analyzing Parameter Choices and Impact

Okay, the parameters are designed for aggressive parallelism. I'm considering the trade-offs, especially how the **DELETE** operation and its **WHERE** clause will respond. I need to focus testing.

#### Existing Summary:

#### Analyzing and Planning Configuration

I'm now assessing the parameters provided. My immediate thought is the aggressive **parallel** settings warrant close scrutiny. I must understand how they affect the deletion operation and any potential impact on the **WHERE** clause evaluation.

#### Examining the Overall Strategy

The provided parameters are now completely understood. My next task is to assess the overall strategy for optimizing this **DELETE**. The aim is to use the scan phase parallelism to improve deletion efficiency on the large table.

#### Examine the Configuration Details

I'm now carefully analyzing the parameters to use for the delete operation. This will help me understand what the intended use for the settings will be. The goal is to allow for the operation to run smoothly on a 48-core system.

#### Analyzing Scan Phase Impact

Okay, I'm now considering how the aggressive settings will affect the scan phase in this delete operation. The goal is to minimize overhead and ensure efficient deletion, and how that effects the vacuum process. I must remember that the **WHERE** clause will be the key part to be optimized.

## Reviewing Scan Phase Optimization

---

Right, let's hone in on the scan phase again. The objective remains: aggressive settings should yield an optimized WHERE clause. The goal is quick deletion and efficient vacuuming.

### Examine and Refine

I'm going to incorporate the feedback. Specifically, I will change the recommendations for the settings, especially `max_parallel_workers_per_gather` and also the `maintenance_work_mem`, keeping in mind the large table and the deletion, and then review and note the final parameter selections.

### Examine the Scan Phase

I'm now focusing on the scan phase for the `DELETE` operation, which is where the parameters will have their primary effect. I'm reviewing the scan strategy, including how aggressive settings influence the process.

Конечно, давайте рассмотрим предложенные вами настройки в контексте 48-ядерного процессора и добавим комментарии. Эти настройки устанавливаются на уровне сессии внутри транзакции и будут влиять на выполнение запроса `DELETE` (точнее, на фазу поиска строк для удаления).

sql

```

BEGIN;
-- Комментарий: Начало транзакции. Все последующие команды SET будут действовать-- только в рамках этой
транзакции и этой сессии.
-- ===== Настройки для
стимулирования параллельного выполнения запросов--
===== Цель этих настроек
- сделать так, чтобы планировщик PostgreSQL с большей-- вероятностью выбирал параллельный план
выполнения для операций, которые-- могут быть распараллелены (например, сканирование больших таблиц).--
Для 48-ядерного процессора это может быть особенно полезно при обработке-- больших объемов данных.
-- Параметр: parallel_setup_cost-- Значение: 0-- Комментарий: Этот параметр оценивает "стоимость"
запуска параллельных рабочих-- процессов. Установка в 0 (или очень низкое значение) говорит
планировщику,-- что запуск параллельных воркеров практически ничего не стоит.-- Это агрессивная
настройка, которая сильно стимулирует использование-- параллельных планов. На 48 ядрах, где потенциал
для параллелизма велик,-- это может быть оправдано, если накладные расходы на запуск воркеров--
действительно малы по сравнению с объемом работы в запросе.-- Однако, это может привести к
распараллеливанию даже небольших запросов,-- где это неэффективно. Рекомендуется тестирование.SET
parallel_setup_cost =0;

-- Параметр: parallel_tuple_cost-- Значение: 0-- Комментарий: Этот параметр оценивает "стоимость"
передачи одной строки (кортежа)-- от параллельного рабочего процесса к ведущему процессу (Gather
node).-- Установка в 0, аналогично parallel_setup_cost, делает параллельные планы-- более
привлекательными для планировщика, так как передача данных считается-- "бесплатной". Это также
агрессивная настройка.SET parallel_tuple_cost =0;

-- Параметр: max_parallel_workers_per_gather-- Значение: 6-- Комментарий: Устанавливает максимальное
количество параллельных рабочих процессов,-- которые могут быть использованы *одним* узлом Gather (или
Gather Merge)-- в плане запроса.-- Для 48-ядерного процессора значение 6 является достаточно
консервативным.-- Если ваша система ввода-вывода справляется и нет других узких мест,-- вы могли бы
рассмотреть увеличение этого значения, например, до 8, 12, 16,-- или даже 24 (например, половина
физических ядер), чтобы лучше утилизировать CPU.-- Это значение не должно превышать глобальный параметр
`max_parallel_workers`-- (который, в свою очередь, не превышает `max_worker_processes`).-- Пример для
большей утилизации: SET max_parallel_workers_per_gather = 16;SET max_parallel_workers_per_gather =6; --
Можно увеличить для 48 ядер
-- Параметр: maintenance_work_mem-- Значение: '256MB'-- Комментарий: Этот параметр определяет объем
памяти, используемый для операций-- обслуживания, таких как VACUUM, CREATE INDEX, ALTER TABLE ADD
FOREIGN KEY.-- Он НЕ влияет напрямую на параллельное выполнение самого запроса DELETE,-- но очень важен
для последующей операции VACUUM, которая будет необходима-- после массового удаления строк для
освобождения места и предотвращения-- раздувания таблицы.-- Для сервера с 48 ядрами, который, вероятно,
имеет значительный объем ОЗУ,-- 256MB - это довольно скромное значение. Рекомендуется увеличить его--
(например, до 1GB - 4GB или даже выше, в зависимости от доступной ОЗУ-- и количества одновременных
операций обслуживания), чтобы ускорить-- последующий VACUUM.-- Пример: SET maintenance_work_mem =
'2GB';SET maintenance_work_mem = '256MB'; -- Рекомендуется значительно увеличить для сервера такого
класса
-- Параметр: min_parallel_table_scan_size-- Значение: 1 (вероятно, имеется в виду 1 блок, т.е. 8KB, или
0 для отключения порога)-- Комментарий: Минимальный размер таблицы (или части таблицы, которая будет
отсканирована),-- для которого планировщик будет рассматривать параллельное сканирование.-- Установка
очень низкого значения (например, 0 или 1 блок) означает, что-- параллельное сканирование будет
рассматриваться практически для всех таблиц,-- даже очень маленьких. Это может быть неэффективно из-за
накладных-- расходов на запуск параллельных воркеров для небольших объемов данных.-- Стандартное
значение (например, 8MB) часто является более сбалансированным.-- Однако, если `public.messages` -
очень большая таблица, и вы хотите-- гарантированно попытаться распараллелить сканирование для поиска
строк-- по условию `created_at`, эта настройка может помочь.SET min_parallel_table_scan_size =1; --
Агрессивно, рассмотрит параллелизм даже для малых сканирований
-- ===== Выполнение
операции DELETE-- =====
Важно: Сам процесс физического удаления строк (модификации данных) в PostgreSQL-- обычно не
распараллеливается на несколько воркеров, работающих с одной таблицей.-- Однако, ФАЗА ПОИСКА строк,
которые подлежат удалению (т.е. выполнение-- условия WHERE created_at < 1739998800), МОЖЕТ быть
распараллелена, если-- планировщик сочтет это выгодным (например, через параллельное сканирование
таблицы).-- Приведенные выше настройки как раз и нацелены на то, чтобы сделать параллельный-- план для
этой фазы поиска более вероятным.
-- CTE `deleted_rows` используется для подсчета удаленных строк.-- `RETURNING 1` (или `RETURNING id`
если есть такой столбец и он нужен)-- эффективнее для подсчета, чем `RETURNING 0`.WITH deleted_rows AS
(
DELETEFROM public.messages
WHERE created_at <1739998800 RETURNING 1-- Возвращаем 1 для каждой удаленной строки для
корректного подсчета)
SELECTcount(*) AS number_of_deleted_rows FROM deleted_rows;

COMMIT;

```

-- Комментарий: Завершение транзакции. Все изменения (удаленные строки)-- фиксируются в базе данных. Настройки сессии сбрасываются.

### Дополнительные глобальные параметры (в `postgresql.conf`), важные для 48-ядерной системы:

Эти параметры устанавливаются не на уровне сессии, а в конфигурационном файле `postgresql.conf` и требуют перезагрузки конфигурации (`pg_reload_conf()`) или перезапуска сервера.

- `max_worker_processes`: Общее максимальное количество фоновых процессов, которые может запустить PostgreSQL. Это включает параллельные рабочие процессы, процессы автовакуума, логической репликации и т.д. Для 48-ядерной системы это значение должно быть достаточно большим, например, 48 или даже больше (например, `max_worker_processes = 64`), чтобы обеспечить достаточное количество воркеров для параллельных запросов и других нужд.
- `max_parallel_workers`: Максимальное количество рабочих процессов, которые система может выделить именно для параллельных запросов. Это значение не может превышать `max_worker_processes`. Для 48 ядер можно установить значение, равное количеству ядер или его части (например, `max_parallel_workers = 32` или `max_parallel_workers = 48`). Оно должно быть не меньше, чем `max_parallel_workers_per_gather`.
- `shared_buffers`: Рекомендуется установить значительный объем, обычно 25-40% от общего объема ОЗУ сервера.
- `work_mem`: Память, используемая для сортировок, хеширования внутри отдельных операций запроса (не для параллелизма сканирования напрямую, но важна для общей производительности). Может быть увеличена на уровне сессии для особо тяжелых запросов.

### Рекомендации после массового `DELETE`:

1. **VACUUM**: После удаления большого количества строк в таблице `public.messages` останется много "мертвого" пространства. Выполните `VACUUM public.messages`; чтобы его очистить и сделать доступным для повторного использования. Для ускорения этой операции важен высокий `maintenance_work_mem`.
2. **ANALYZE**: Обновите статистику для таблицы: `ANALYZE public.messages`; . Это поможет планировщику строить оптимальные планы для будущих запросов к этой таблице.
3. **VACUUM FULL (с осторожностью)**: Если нужно вернуть дисковое пространство операционной системе и сильно уменьшить физический размер файла таблицы, можно использовать `VACUUM FULL public.messages`; . Однако эта операция требует эксклюзивной блокировки таблицы на все время своего выполнения и создает полную копию таблицы, что может быть очень длительным и ресурсоемким процессом.

Перед применением агрессивных настроек на продуктивной системе всегда рекомендуется проводить тестирование на репрезентативной тестовой среде, чтобы убедиться в их положительном влиянии и отсутствии побочных эффектов.

6:12 PM

Thinking...

## Examining PostgreSQL Configurations

Okay, I need to dig into the optimal PostgreSQL settings for this beast of a 48-core processor. I'll focus on the key areas, like shared buffers, work memory, and connection limits. I'll have to consider the workload profile too, to tune them properly.

### Refining Memory Allocations

I'm focusing on the memory-related parameters now. I'll highlight `shared_buffers` again, obviously. Then, `effective_cache_size`, which is often overlooked but essential for the query planner's efficiency. I'll follow up with sensible defaults for `work_mem` and `maintenance_work_mem`.

### Thinking Through High-Core Settings

Right, let's look at some specific settings for this 48-core machine, especially considering the possibility of heavy workloads after a `DELETE` operation. I'll start with the big ones: memory, parallelism, checkpointing, autovacuum, and asynchronous I/O. It's a lot to consider!

### Categorizing Configuration Options

Okay, I'm breaking down these settings into categories. Memory, parallelism, checkpointing, autovacuum, and I/O are key. For a 48-core system, each deserves special attention. I need to present these with explanations and context, highlighting which settings are global and which might be set at the session level.

### Focusing on Settings Refinement

I'm expanding the configuration recommendations, aiming for a comprehensive set of global parameters. I'll build upon previous advice, particularly the `DELETE` operation context. Highlighting the interaction between session-level and global settings, especially regarding parallelism, is also important. Finally, I'll prioritize clear categorization for easier comprehension.

### Categorizing Server Configurations

Okay, I'm consolidating the global settings I need to present, clarifying their importance on this powerful machine. I'm emphasizing the interaction with session-level settings, particularly for parallelism during large operations. My plan is to categorize them and provide context for each, as well as some examples.

Отлично! Раз у нас такая мощная 48-ядерная система, мы можем настроить PostgreSQL для еще лучшей производительности, особенно если ожидаются высокие нагрузки или обработка больших объемов данных.

Предыдущие настройки были в основном сессионными, направленными на оптимизацию конкретной операции **DELETE**. Теперь давайте рассмотрим более **глобальные настройки (в `postgresql.conf`)**, которые помогут серверу в целом лучше утилизировать ресурсы 48-ядерного процессора и большого объема ОЗУ, который обычно сопутствует таким системам.

**Важно:**

- Эти настройки изменяются в файле `postgresql.conf`.
- После изменения `postgresql.conf` необходимо перезагрузить конфигурацию сервера (`SELECT pg_reload_conf();`) или, для некоторых параметров, перезапустить сам сервер PostgreSQL.
- **Всегда тестируйте изменения на тестовой среде перед применением на продуктивной!**
- Оптимальные значения сильно зависят от общего объема ОЗУ, скорости дисковой подсистемы и характера вашей рабочей нагрузки.

## 1. Настройки памяти (Memory Configuration)

- **`shared_buffers`**
  - **Описание:** Основной буферный кеш PostgreSQL в оперативной памяти. Чем больше, тем больше данных и блоков индекса может храниться в ОЗУ, уменьшая дисковый ввод-вывод.
  - **Рекомендация для 48 ядер (и много ОЗУ):** Обычно 25-40% от общего объема ОЗУ сервера. Например, для сервера со 128GB ОЗУ можно установить **32GB - 50GB**. Не стоит делать его слишком большим (например, >50% ОЗУ), так как ОС также нуждается в памяти для своего кеша.
  - **Пример:** `shared_buffers = 40GB`
  - **Требуется перезапуск сервера.**
- **`effective_cache_size`**
  - **Описание:** Оценка для планировщика PostgreSQL общего объема памяти, доступного для кэширования данных (`shared_buffers` + кеш ОС). Помогает планировщику принимать более обоснованные решения о стоимости использования индексов.
  - **Рекомендация:** Обычно 50-75% от общего объема ОЗУ.
  - **Пример:** `effective_cache_size = 80GB` (для сервера со 128GB ОЗУ)
- **`work_mem` (глобальный по умолчанию)**
  - **Описание:** Объем памяти, используемый для внутренних операций сортировки, хеширования, слияния битовых карт перед записью на диск. Устанавливается *на одну операцию* внутри запроса. Параллельные запросы могут использовать несколько таких блоков.
  - **Рекомендация:** Для 48-ядерной системы, где может выполняться много параллельных операций, важно найти баланс. Слишком высокое значение может быстро исчерпать память. Начните с умеренного значения (например, **64MB - 256MB**) и увеличивайте для конкретных сессий/запросов при необходимости (`SET LOCAL work_mem = '512MB';`).
  - **Пример:** `work_mem = 128MB`
- **`maintenance_work_mem` (глобальный по умолчанию)**
  - **Описание:** Объем памяти для операций обслуживания (`VACUUM`, `CREATE INDEX`, `ALTER TABLE ADD FOREIGN KEY`).
  - **Рекомендация:** Для 48-ядерной системы с большим объемом ОЗУ установите значительно больше, чем `work_mem`, чтобы ускорить эти важные операции. Например, 1-10% от общего ОЗУ.
  - **Пример:** `maintenance_work_mem = 4GB` (для сервера со 128GB ОЗУ)

## 2. Настройки параллелизма (Parallelism Configuration)



- **max\_worker\_processes**

- **Описание:** Общее максимальное количество фоновых процессов, которые может запустить PostgreSQL (включая параллельные воркеры, автовакуум, логическую репликацию и т.д.).
- **Рекомендация:** Для 48 ядер должно быть как минимум 48, плюс запас для других фоновых задач.
- **Пример:** `max_worker_processes = 64` (или даже 96, если много параллельных задач и других воркеров)
- **Требуется перезапуск сервера.**

- **max\_parallel\_workers**

- **Описание:** Максимальное количество рабочих процессов, которые система может выделить *именно для параллельных запросов*. Не может превышать `max_worker_processes`.
- **Рекомендация:** Для 48 ядер можно установить равным количеству ядер или чуть меньше, чтобы оставить ресурсы для других задач.
- **Пример:** `max_parallel_workers = 40` (или `max_parallel_workers = 48`)
- **Требуется перезапуск сервера.**

- **max\_parallel\_maintenance\_workers**

- **Описание:** Максимальное количество параллельных рабочих процессов, которые могут быть запущены *одной* служебной командой (например, `CREATE INDEX` на партиционированной таблице или `VACUUM` с опцией `PARALLEL`).
- **Рекомендация:** Для 48 ядер, если у вас есть большие таблицы/индексы, которые могут выиграть от параллельного обслуживания.
- **Пример:** `max_parallel_maintenance_workers = 8` (или 16, в зависимости от `maintenance_work_mem` на каждого воркера)

- **parallel\_leader\_participation**

- **Описание:** Позволяет ли ведущему процессу (который инициировал запрос) также выполнять часть параллельной работы, а не только координировать воркеры.
- **Рекомендация:** `on` (по умолчанию). Обычно это хорошо, так как ведущий процесс не простаивает.

### 3. Настройки WAL (Write-Ahead Log) и Checkpoints

- **wal\_buffers**

- **Описание:** Размер буфера в разделяемой памяти для записей WAL перед их записью на диск.
- **Рекомендация:** Значение по умолчанию `-1` (1/32 от `shared_buffers`, но не более 16MB) часто подходит. Можно увеличить до `16MB` или `32MB` при очень высокой интенсивности записи.
- **Пример:** `wal_buffers = 16MB` (или `-1`)

- **min\_wal\_size / max\_wal\_size**

- **Описание:** Управляют общим объемом дискового пространства, используемого для файлов WAL. Чекпоинты происходят, когда объем WAL приближается к `max_wal_size`.
- **Рекомендация:** Для систем с высокой нагрузкой на запись увеличьте эти значения, чтобы чекпоинты происходили реже, сглаживая пики I/O.
- **Пример:** `min_wal_size = 2GB`, `max_wal_size = 8GB` (или выше, например, `4GB` и `16GB`)

- **checkpoint\_completion\_target**
  - **Описание:** Доля времени между чекпоинтами, в течение которой запись "грязных" буферов на диск должна быть завершена. Сглаживает I/O нагрузку от чекпоинтов.
  - **Рекомендация:** **0.9** (по умолчанию **0.5** до PostgreSQL 14, **0.9** с 14-й версии). Значение **0.9** лучше распределяет нагрузку.
  - **Пример:** `checkpoint_completion_target = 0.9`

#### 4. Настройки автовакуума (Autovacuum Configuration)

После массовых удалений, как в вашем примере, автовакуум становится критически важным.

- **autovacuum\_max\_workers**
  - **Описание:** Максимальное количество процессов автовакуума, которые могут работать одновременно.
  - **Рекомендация:** Для 48-ядерной системы с потенциально большим количеством таблиц, нуждающихся в очистке.
  - **Пример:** `autovacuum_max_workers = 6` (или 8-10, если много баз данных и таблиц)
- **autovacuum\_naptime**
  - **Описание:** Как часто главный процесс автовакуума проверяет, не нужно ли запустить рабочие процессы для какой-либо таблицы.
  - **Рекомендация:** Можно уменьшить для более частого запуска на очень активных системах (например, **15s** или **30s**), но значение по умолчанию **1min** часто является хорошим компромиссом.
  - **Пример:** `autovacuum_naptime = 30s`
- **autovacuum\_vacuum\_scale\_factor / autovacuum\_analyze\_scale\_factor**
  - **Описание:** Доля "мертвых" строк (для vacuum) или измененных строк (для analyze) от общего числа строк в таблице, при которой запускается автовакуум/автоанализ.
  - **Рекомендация:** Для очень больших таблиц стандартные 20% (vacuum) и 10% (analyze) могут означать слишком редкий запуск. Можно уменьшить эти значения глобально или настроить их для конкретных таблиц через `ALTER TABLE`.
  - **Пример (глобально):** `autovacuum_vacuum_scale_factor = 0.10,`  
`autovacuum_analyze_scale_factor = 0.05`
- **autovacuum\_vacuum\_cost\_delay / autovacuum\_vacuum\_cost\_limit**
  - **Описание:** Управляют троттлингом I/O для автовакуума, чтобы он не слишком сильно мешал пользовательским запросам.
  - **Рекомендация:** На мощных системах с быстрыми дисками можно уменьшить `autovacuum_vacuum_cost_delay` (например, до **2ms** или **10ms** вместо **20ms** в старых версиях, в новых **2ms** по умолчанию) или увеличить `autovacuum_vacuum_cost_limit`, чтобы автовакуум работал агрессивнее.

#### 5. Настройки ввода-вывода (I/O Configuration)

- **effective\_io\_concurrency**
  - **Описание:** Оценка количества одновременных дисковых операций, которые система может эффективно выполнять. Актуально для SSD-массивов.
  - **Рекомендация:** Для хороших SSD-массивов можно установить значение от нескольких десятков до нескольких сотен (например, **100-300**). Для одного SSD **1-2**.
  - **Пример (для RAID из SSD):** `effective_io_concurrency = 200` (Только для Linux, если `synchronize_seqscans` включен, иначе влияет на bitmap heap scans)

- **random\_page\_cost**
  - **Описание:** Оценка стоимости чтения случайной страницы с диска относительно стоимости последовательного чтения (**seq\_page\_cost**, обычно **1.0**).
  - **Рекомендация:** Для SSD-дисков случайное чтение почти так же быстро, как последовательное. Уменьшите это значение (например, **1.1 - 1.5**) по сравнению со значением по умолчанию **4.0** (для HDD). Это сделает использование индексов более привлекательным для планировщика.
  - **Пример (для SSD):** **random\_page\_cost = 1.1**

## 6. JIT (Just-In-Time) компиляция

- **jit**
  - **Описание:** Включает JIT-компиляцию для ускорения выполнения выражений и некоторых частей запросов.
  - **Рекомендация:** **on** (по умолчанию в современных версиях). Для 48-ядерной системы накладные расходы на JIT-компиляцию могут быть оправданы для сложных аналитических запросов.
  - **Пример:** **jit = on**
- **jit\_above\_cost, jit\_inline\_above\_cost, jit\_optimize\_above\_cost**
  - **Описание:** Пороговые значения "стоимости" запроса/операции, выше которых будет применяться JIT-компиляция.
  - **Рекомендация:** Значения по умолчанию обычно являются хорошей отправной точкой. Тонкая настройка требует глубокого анализа планов запросов и профилирования.

### Возвращаясь к вашему **DELETE** и сессионным настройкам:

Глобальные настройки, такие как **max\_parallel\_workers**, создают "пул" доступных воркеров. Ваши сессионные **SET max\_parallel\_workers\_per\_gather = 6;** (или более высокое значение, которое вы выберете) будут запрашивать воркеров из этого общего пула для конкретного узла Gather в плане вашего **DELETE**. Агрессивные **parallel\_setup\_cost = 0;** и **parallel\_tuple\_cost = 0;** будут сильно мотивировать планировщик использовать эти воркеры для сканирования таблицы **public.messages** при поиске строк по **created\_at < 1739998800**.

### Не забывайте:

- **Мониторинг:** Используйте **EXPLAIN (ANALYZE, BUFFERS)** для анализа планов запросов. Следите за загрузкой CPU, I/O, использованием памяти.
- **Итеративный подход:** Вносите изменения постепенно и наблюдайте за эффектом.

Эти настройки должны дать хороший старт для оптимизации вашего PostgreSQL сервера под 48-ядерный процессор!

6:15 PM