PostgreSQL: настройка производительности

Алексей Борзов (Sad Spirit)
borz_off@cs.msu.su

Содержание

1 Введение

Скорость работы, вообще говоря, не является основной причиной использования реляционных СУБД. Более того, первые реляционные базы работали *медленнее* своих предшественников. Выбор этой технологии был вызван скорее

- возможностью возложить поддержку целостности данных на СУБД;
- независимостью логической структуры данных от физической.

Эти особенности позволяют сильно упростить написание приложений, но требуют для своей реализации дополнительных ресурсов.

Таким образом, прежде, чем искать ответ на вопрос «как заставить РСУБД работать быстрее в моей задаче?» следует ответить на вопрос «нет ли более подходящего средства для решения моей задачи, чем РСУБД?» Иногда использование другого средства потребует меньше усилий, чем настройка производительности.

Данная статья посвящена возможностям повышения производительности свободной РСУБД PostgreSQL. Статья не претендует на исчерпывающее изложение вопроса, наиболее полным и точным руководством по использованию PostgreSQL является, конечно, официальная документация [1] и официальный FAQ [2]. Также существует англоязычный список рассылки postgresql-performance, посвящённый именно этим вопросам.

Статья состоит из двух разделов, первый из которых ориентирован скорее на администратора, второй — на разработчика приложений. Рекомендуется прочесть оба раздела: отнесение многих вопросов к какому-то одному из них весьма условно. Большая часть раздела, посвящённого настройке сервера, является переводом материалов [3], [4], [5]. В разделе, посвящённом оптимизации БД и приложения, использовались [6], [7], [8] и личный опыт.

1.1 Не используйте настройки по умолчанию

По умолчанию PostgreSQL сконфигурирован таким образом, чтобы он мог быть запущен практически на любом компьютере и не слишком мешал при этом работе других приложений. Это *особенно* касается используемой памяти.

Настройки по умолчанию подходят только для следующего использования: с ними вы сможете проверить, работает ли установка PostgreSQL, создать тестовую базу уровня записной книжки и потренироваться писать к ней запросы. Если вы собираетесь разрабатывать (а тем более запускать в работу) реальные приложения, то настройки придётся радикально изменить.

В дистрибутиве PostgreSQL, к сожалению, не поставляется файлов с «рекомендуемыми» настройками. Вообще говоря, такие файлы создать весьма сложно, т.к. оптимальные настройки конкретной установки PostgreSQL будут определяться:

- конфигурацией компьютера;
- объёмом и типом данных, хранящихся в базе;
- отношением числа запросов на чтение и на запись;
- тем, запущены ли другие требовательные к ресурсам процессы (например, вебсервер).

1.2 Используйте актуальную версию сервера

Если у вас стоит устаревшая версия PostgreSQL, то наибольшего ускорения работы вы сможете добиться, обновив её до текущей. Укажем лишь наиболее значительные из связанных с производительностью изменений.

- В версии 7.1 появился журнал транзакций, до того данные в таблицу сбрасывались каждый раз при успешном завершении транзакции.
- В версии 7.2 появились:
  - новая версия команды VACUUM, не требующая блокировки;
  - команда ANALYZE, строящая гистограмму распределения данных в столбцах, что позволяет выбирать более быстрые планы выполнения запросов;
  - подсистема сбора статистики.
- В версии 7.4 была ускорена работа многих сложных запросов (включая печально известные подзапросы IN/NOT IN).

Следует также отметить, что большая часть изложенного в статье материала относится к версии сервера не ниже 7.2.

1.3 Стоит ли доверять тестам производительности

Перед тем, как заниматься настройкой сервера, вполне естественно ознакомиться с опубликованными данными по производительности, в том числе в сравнении с другими СУБД. К сожалению, многие тесты служат не столько для облегчения вашего выбора, сколько для продвижения конкретных продуктов в качестве «самых быстрых».

При изучении опубликованных тестов в первую очередь обратите внимание, соответствует ли величина и тип нагрузки, объём данных и сложность запросов в тесте тому, что *вы* собираетесь делать с базой? Пусть, например, обычное использование вашего приложения подразумевает несколько одновременно работающих запросов на обновление к таблице в миллионы записей. В этом случае СУБД, которая в несколько раз быстрее всех остальных ищет запись в таблице в тысячу записей, может оказаться не лучшим выбором.

Ну и наконец, вещи, которые должны сразу насторожить:

- Тестирование устаревшей версии СУБД.
- Использование настроек по умолчанию (или отсутствие информации о настройках).
- Тестирование в однопользовательском режиме (если, конечно, вы не предполагаете использовать СУБД именно так).
- Использование расширенных возможностей одной СУБД при игнорировании расширенных возможностей другой.
- Использование заведомо медленно работающих запросов (см. пункт 3.4).

2 Настройка сервера

В этом разделе описаны рекомендуемые значения параметров, влияющих на производительность СУБД. Эти параметры обычно устанавливаются в конфигурационном файле `postgresql.conf` и влияют на все базы в текущей установке.

2.1 Используемая память

2.1.1 Общий буфер сервера: `shared_buffers`

PostgreSQL не читает данные напрямую с диска и не пишет их сразу на диск. Данные загружаются в общий буфер сервера, находящийся в разделяемой памяти, серверные процессы читают и пишут блоки в этом буфере, а затем уже изменения сбрасываются на диск.

Если процессу нужен доступ к таблице, то он сначала ищет нужные блоки в общем буфере. Если блоки присутствуют, то он может продолжать работу, если нет — делается системный вызов для их загрузки. Загружаться блоки могут как из файлового кэша ОС, так и с диска, и эта операция может оказаться весьма «дорогой».

Если объём буфера недостаточен для хранения часто используемых рабочих данных, то они будут постоянно писаться и читаться из кэша ОС или с диска, что крайне отрицательно скажется на производительности.

Объём задаётся параметром `shared_buffers` в файле `postgresql.conf`. Единица измерения параметра — блоки величиной 8 кБ. По умолчанию значение параметра составляет 64[1], что соответствует 512 кБ памяти. Это весьма мало, и для полноценной работы значение параметра следует увеличить.

В то же время не следует устанавливать это значение слишком большим: PostgreSQL полагается на то, что операционная система кэширует файлы (см. пункт 2.4.1), и не старается дублировать эту работу. Кроме того, чем больше памяти будет отдано под буфер, тем меньше останется операционной системе и другим приложениям, что может привести к своппингу.

В качестве начальных значений можете попробовать следующие:

- Начните с 4 МБ (512) для рабочей станции
- Средний объём данных и 256–512 МБ доступной памяти: 16–32 МБ (2048–4096)
- Большой объём данных и 1–4 ГБ доступной памяти: 64–256 МБ (8192–32768)

Для тонкой настройки параметра установите для него большое значение и потестируйте базу при обычной нагрузке. Проверяйте использование разделяемой памяти при помощи `ipcs` или других утилит. Рекомендуемое значение параметра будет примерно в 1,2–2 раза больше, чем максимум использованной памяти.

Обратите внимание, что память под буфер выделятся при запуске сервера, и её объём при работе не изменяется. Учтите также, что настройки ядра операционной системы могут не дать вам выделить большой объём памяти. В руководстве администратора PostgreSQL описано, как можно изменить эти настройки: http://developer.postgresql.org/docs/postgres/kernel-resources.html

2.1.2 Память для сортировки результата запроса: `sort_mem`

Этот параметр определяет объём памяти, которую процесс может использовать для сортировки результата запроса. Учтите, что такой объём может быть использован *каждым* процессом для *каждой* сортировки (в сложных запросах их может быть несколько).

Если объём памяти недостаточен для сортироки некоторого результата, то серверный процесс будет использовать временные файлы. Если же объём памяти слишком велик, то это может привести к своппингу.

[1] актуально для версий до 7.4

Объём памяти задаётся параметром `sort_mem` в файле `postgresql.conf`. Единица измерения параметра — 1 кБ. Значение по умолчанию — 1024.

В качестве начального значения для параметра можете взять 2–4% доступной памяти.

Этот параметр может также быть задан для отдельного соединения. Если вы знаете, что в конкретном соединении будет выполняться запрос, требующий сортировки значительного объёма данных, то можете поднять значение `sort_mem` перед выполнением запроса.

2.1.3 Память для работы команды VACUUM: `vacuum_mem`

Этот параметр задаёт объём памяти, используемый командой VACUUM. Обычно эта команда больше нагружает диски, но увеличение `vacuum_mem` позволит ускорить процесс за счёт хранения в памяти б*о*льших объёмов информации об удалённых записях.

Объём памяти задаётся параметром `vacuum_mem` в файле `postgresql.conf`. Единица измерения параметра — 1 кБ. Значение по умолчанию — 8192.

Этот параметр может также быть задан для отдельного соединения. Можете сделать его поменьше для частых регулярных запусков VACUUM и большим для ежедневных/еженедельных запусков VACUUM FULL.

2.2 Журнал транзакций и контрольные точки

Журнал транзакций PostgreSQL работает следующим образом: все изменения в файлах данных (в которых находятся таблицы и индексы) производятся только после того, как они были занесены в журнал транзакций, при этом записи в журнале должны быть гарантированно записаны на диск.

В этом случае нет необходимости сбрасывать на диск изменения данных при каждом успешном завершении транзакции: в случае сбоя БД может быть восстановлена по записям в журнале. Таким образом, данные из буферов сбрасываются на диск при проходе контрольной точки: либо при заполнении нескольких (параметр `checkpoint_segments`, по умолчанию 3) сегментов журнала транзакций, либо через определённый интервал времени (параметр `checkpoint_timeout`, измеряется в секундах, по умолчанию 300).

Изменение этих параметров прямо не повлияет на скорость чтения, но может принести большую пользу, если данные в базе активно изменяются.

2.2.1 `fsync` и стоит ли его трогать

Наиболее радикальное из возможных решений — выставить значение `No` параметру `fsync`. При этом записи в журнале транзакций *не будут* принудительно сбрасываться на диск, что даст большой прирост скорости записи. Учтите: вы жертвуете надёжностью, в случае сбоя целостность базы будет нарушена, и её придётся восстанавливать из резервной копии!

Использовать этот параметр рекомендуется лишь в том случае, если вы всецело доверяете своему «железу» и своему источнику бесперебойного питания. Ну или если данные в базе не представляют для вас особой ценности...

В пункте 2.5.1 описано менее радикальное решение, позволяющее, тем не менее, добиться хорошего прироста производительности.

2.2.2 Уменьшение количества контрольных точек

Если в базу заносятся большие объёмы данных, то контрольные точки могут происходить слишком часто[2]. При этом производительность упадёт из-за постоянного сбрасывания на диск данных из буфера.

Для увеличения интервала между контрольными точками нужно увеличить количество сегментов журнала транзакций (`checkpoint_segments`). Каждый сегмент занимает 16 МБ, так что на диске будет занято дополнительное место. Обычно на диске будет не менее одного и не более `2*checkpoint_segments+1` сегментов журнала.

Следует также отметить, что чем больше интервал между контрольными точками, тем дольше будут восстанавливаться данные по журналу транзакций после сбоя.

2.2.3 Прочие параметры

wal_sync_method определяет метод, при помощи которого записи в журнале транзакций принудительно сбрасываются на диск. Значение по умолчанию зависит от платформы. Возможно, изменение этого параметра позволит увеличить производительность (а возможно, и не позволит).

wal_buffers (в блоках по 8 кБ, 8 по умолчанию) определяет размер буфера журнала транзакций[3], в котором накапливаются записи перед сбросом их на диск. Стоит увеличить буфер до 256–512 кБ, что позволит лучше работать с большими транзакциями.

commit_delay (в микросекундах, 0 по умолчанию) и

commit_siblings (5 по умолчанию) определяют задержку между попаданием записи в буфер журнала транзакций и сбросом её на диск. Если при успешном завершении транзакции активно не менее `commit_siblings` транзакций, то запись будет задержана на время `commit_delay`. Если за это время завершится другая транзакция, то их изменения будут сброшены на диск вместе, при помощи одного системного вызова. Эти параметры позволят ускорить работу, если параллельно выполняется много «мелких» транзакций.

2.3 Free Space Map: как избавиться от VACUUM FULL

Особенностями версионных движков БД (к которым относится и используемый в PostgreSQL) является следующее:

- Транзакции, изменяющие данные в таблице, не блокируют транзакции, читающие из неё данные, и наоборот (это хорошо);
- При изменении данных в таблице (командами UPDATE или DELETE) накапливается мусор[4] (а это плохо).

В каждой СУБД сборка мусора реализована особым образом, в PostgreSQL для этой цели применяется команда VACUUM (описана в пункте 3.1.1).

[2] «слишком часто» можно определить как «чаще раза в минуту». Вы также можете задать параметр `checkpoint_warning` (в секундах): в журнал сервера будут писаться предупреждения, если контрольные точки происходят чаще заданного.

[3] буфер находится в разделяемой памяти и является общим для всех процессов

[4] под которым понимаются старые версии изменённых/удалённых записей

До версии 7.2 команда VACUUM полностью блокировала таблицу. Начиная с версии 7.2, команда VACUUM накладывает более слабую блокировку, позволяющую параллельно выполнять команды SELECT, INSERT, UPDATE и DELETE над обрабатываемой таблицей. Старый вариант команды называется теперь VACUUM FULL.

Новый вариант команды не пытается удалить все старые версии записей и, соответственно, уменьшить размер файла, содержащего таблицу, а лишь помечает занимаемое ими место как свободное. Для информации о свободном месте есть следующие настройки:

max_fsm_relations максимальное количество таблиц, для которых будет отслеживаться свободное место.

max_fsm_pages количество блоков, для которых будет хранится информация о свободном месте. Информация хранится в разделяемой памяти, для каждой записи требуется по 6 байт.

Если информация обо всех изменениях помещается в FSM, то команды VACUUM будет достаточно для сборки мусора, если нет — понадобится VACUUM FULL, во время работы которой нормальное использование БД сильно затруднено.

Параметр `max_fsm_relations` должен быть не меньше общего количества таблиц во всех базах данной установки. В качестве начального приближения для `max_fsm_pages` можно взять половину от среднего количества записей, изменяемых (UPDATE или DELETE) между запусками команды VACUUM.

2.4 Прочие настройки

2.4.1 Оценка объёма кэша ОС: `effective_cache_size`

Этот параметр сообщает PostgreSQL примерный объём файлового кэша операционной системы, оптимизатор использует эту оценку для построения плана запроса.

Объём задаётся параметром `effective_cache_size` в `postgresql.conf`. Единица измерения — блоки величиной 8 кБ. По умолчанию значение параметра составляет 1000.

Пусть в вашем компьютере 1,5 ГБ памяти, параметр `shared_buffers` установлен в 32 МБ, а параметр `effective_cache_size` в 800 МБ. Если запросу нужно 700 МБ данных, то PostgreSQL оценит, что все нужные данные уже есть в памяти и выберет более агрессивный план с использованием индексов и merge joins. Но если `effective_cache_size` будет всего 200 МБ, то оптимизатор вполне может выбрать более эффективный для дисковой системы план, включающий полный просмотр таблицы.

В качестве начального значения можете использовать 25–50% доступной[5] памяти.

2.4.2 Сбор статистики

default_statistics_target задаёт объём по умолчанию статистики, собираемой командой ANALYZE (см. пункт 3.1.2). Увеличение параметра заставит эту команду работать дольше, но может позволить оптимизатору строить более быстрые планы, используя полученные дополнительные данные. Объём статистики для конкретного поля может быть задан командой ALTER TABLE ... SET STATISTICS.

[5] т.е. не занятой операционной системой и приложениями

У PostgreSQL также есть специальная подсистема — сборщик статистики, — которая в реальном времени собирает данные об активности сервера. Эта подсистема контролируется следующими параметрами, принимающими значения true/false:

stats_start_collector включать ли сбор статистики. По умолчанию включён, отключайте, только если статистика вас совершенно не интересует.

stats_reset_on_server_start обнулять ли статистику при перезапуске сервера. По умолчанию — обнулять.

stats_command_string передавать ли сборщику статистики информацию о текущей выполняемой команде и времени начала её выполнения. По умолчанию эта возможность отключена. Следует отметить, что эта информация будет доступна только привилегированным пользователям и пользователям, от лица которых запущены команды, так что проблем с безопасностью быть не должно.

stats_row_level, stats_block_level собирать ли информацию об активности на уровне записей и блоков соответственно. По умолчанию сбор отключён.

Данные, полученные сборщиком статистики, доступны через специальные системные представления. При установках по умолчанию собирается очень мало информации, рекомендуется включить все возможности: дополнительная нагрузка будет невелика, в то время как полученные данные позволят оптимизировать использование индексов (см. пункт 3.2.2).

2.5 Диски и файловые системы

Очевидно, что от качественной дисковой подсистемы в сервере БД зависит немалая часть производительности. Вопросы выбора и тонкой настройки «железа», впрочем, не являются темой данной статьи, ограничимся уровнем файловой системы.

Единого мнения насчёт наиболее подходящей для PostgreSQL файловой системы нет, поэтому рекомендуется использовать ту, которая лучше всего поддерживается вашей операционной системой. При этом учтите, что современные журналирующие файловые системы не намного медленнее нежурналирующих, а выигрыш — быстрое восстановление после сбоев — от их использования велик.

Вы легко можете получить выигрыш в производительности без побочных эффектов, если примонтируете файловую систему, содержащую базу данных, с параметром `noatime`[6].

2.5.1 Перенос журнала транзакций на отдельный диск

При доступе к диску изрядное время занимает не только собственно чтение данных, но и перемещение магнитной головки.

Если в вашем сервере есть несколько физических дисков[7], то вы можете разнести файлы базы данных и журнал транзакций по разным дискам. Данные в сегменты журнала пишутся последовательно, более того, записи в журнале транзакций сразу сбрасываются на диск, поэтому в случае нахождения его

[6] при этом не будет отслеживаться время последнего доступа к файлу

[7] несколько логических разделов на одном диске здесь, очевидно, не помогут: головка всё равно будет одна

на отдельном диске магнитная головка не будет лишний раз двигаться, что позволит ускорить запись.

Порядок действий:

- Остановите сервер (!).
- Перенесите каталог `pg_xlog`, находящийся в каталоге с базами данных, на другой диск.
- Создайте на старом месте символическую ссылку.
- Запустите сервер.

Примерно таким же образом можно перенести и часть файлов, содержащих таблицы и индексы, на другой диск, но здесь потребуется больше кропотливой ручной работы, а при внесении изменений в схему базы процедуру, возможно, придётся повторить.

3 Оптимизация БД и приложения

Для быстрой работы каждого запроса в вашей базе в основном требуется следующее:

1. Отсутствие в базе мусора, мешающего добраться до актуальных данных. Можно сформулировать две подзадачи:
   - Грамотное проектирование базы. Освещение этого вопроса выходит далеко за рамки этой статьи.
   - Сборка мусора, возникающего при работе СУБД.
2. Наличие быстрых путей доступа к данным — индексов.
3. Возможность использования оптимизатором этих быстрых путей.
4. Обход известных проблем.

3.1 Поддержание базы в порядке

В данном разделе описаны действия, которые должны периодически выполняться для каждой базы. От разработчика требуется только настроить их автоматическое выполнение (при помощи `cron`) и опытным путём подобрать его оптимальную частоту.

3.1.1 Команда VACUUM

Используется для «сборки мусора» в базе данных. Начиная с версии 7.2, существует в двух вариантах:

- VACUUM FULL (VACUUM до 7.2) пытается удалить все старые версии записей и, соответственно, уменьшить размер файла, содержащего таблицу. Этот вариант команды полностью блокирует обрабатываемую таблицу.
- VACUUM (начиная с 7.2) помечает место, занимаемое старыми версиями записей, как свободное (см. также пункт 2.3). Использование этого варианта команды, как правило, не уменьшает размер файла, содержащего таблицу, но позволяет не дать ему бесконтрольно расти, зафиксировав на некотором приемлемом уровне. При работе VACUUM возможен параллельный доступ к обрабатываемой таблице.

При использовании в форме VACUUM [FULL] ANALYZE, после сборки мусора будет обновлена статистика по данной таблице, используемая оптимизатором. В абсолютном большинстве случаев имеет смысл использовать именно эту форму.

Рекомендуется достаточно частое — в [7] и [8], например, раз в несколько минут (!) — выполнение VACUUM ANALYZE для часто обновляемых баз (или отдельных таблиц). В обыкновенных случаях достаточно ежедневного[8] выполнения этой команды. При этом обратите внимание: если «бутылочное горлышко» вашего сервера находится в районе дисковой подсистемы, то выполнение VACUUM параллельно с обычной работой может крайне отрицательно сказаться на производительности.

Команду VACUUM FULL стоит запускать достаточно редко, не чаще раза в неделю. Её также имеет смысл запускать вручную для конкретной таблицы после удаления или обновления большой части записей в ней.

3.1.2 Команда ANALYZE

Служит для обновления информации о распределении данных в таблице. Эта информация используется оптимизатором для выбора наиболее быстрого плана выполнения запроса.

Обычно команда используется в связке VACUUM ANALYZE. Если в базе есть таблицы, данные в которых не изменяются и не удаляются, а лишь добавляются, то для таких таблиц можно использовать отдельную команду ANALYZE. Также стоит использовать эту команду для отдельной таблицы после добавления в неё большого количества записей.

3.1.3 `pg_autovacuum`

Начиная с версии 7.4, в дистрибутиве PostgreSQL поставляется программа `pg_autovacuum`, которая отслеживает изменения в таблицах и автоматически запускает команды VACUUM и/или ANALYZE для этих таблиц по достижении определённого предела.

Использование этой программы позволяет отказаться от настройки периодического выполнения команд VACUUM и ANALYZE. Более того, в случае использования `pg_autovacuum` ресурсы не тратятся впустую на обработку таблиц, которые практически не подвергались изменениям.

Для работы `pg_autovacuum` должен быть включён сборщик статистики (см. пункт 2.4.2) и включён параметр `stats_row_level`.

3.1.4 Команда REINDEX

Команда REINDEX используется для перестройки существующих индексов. Использовать её имеет смысл в случае

- порчи индекса;
- постоянного увеличения его размера.

Второй случай требует пояснений. Индекс, как и таблица, содержит блоки со старыми версиями записей. PostgreSQL не всегда может заново использовать эти блоки[9], и поэтому файл с индексом постепенно увеличивается в размерах. Если данные в таблице часто меняются, то расти он может весьма быстро.

[8] точнее еженощного, т.к. по ночам нагрузка на базу меньше

[9] в версии 7.4 были сделаны существенные улучшения

Если вы заметили подобное поведение какого-то индекса, то стоит настроить для него периодическое выполнение команды REINDEX. Учтите: команда REINDEX, как и VACUUM FULL, полностью блокирует таблицу, поэтому выполнять её надо тогда, когда загрузка сервера минимальна.

3.2 Использование индексов

Опыт показывает, что наиболее значительные проблемы с производительностью вызываются отсутствием нужных индексов. Поэтому столкнувшись с медленным запросом, в первую очередь проверьте, существуют ли индексы, которые он может использовать. Если нет — постройте их.

Излишек индексов, впрочем, тоже чреват проблемами:

- Команды, изменяющие данные в таблице, должны изменить также и индексы. Очевидно, чем больше индексов построено для таблицы, тем медленнее это будет происходить.
- Оптимизатор перебирает возможные пути выполнения запросов. Если построено много ненужных индексов, то этот перебор будет идти дольше.

Единственное, что можно сказать с большой степенью определённости — поля, являющиеся внешими ключами, и поля, по которым объединяются таблицы, индексировать надо обязательно.

3.2.1 Команда EXPLAIN [ANALYZE]

Команда EXPLAIN [запрос] показывает, каким образом PostgreSQL собирается выполнять ваш запрос. Команда EXPLAIN ANALYZE [запрос] выполняет запрос[10] и показывает как изначальный план, так и реальный процесс его выполнения.

Чтение вывода этих команд — искусство, которое приходит с опытом. Для начала обращайте внимание на следующее:

- Использование полного просмотра таблицы (seq scan).
- Использование наиболее примитивного способа объединения таблиц (nested loop).
- Для EXPLAIN ANALYZE: нет ли больших отличий в предполагаемом количестве записей и реально выбранном? Если оптимизатор использует устаревшую статистику, то он может выбирать не самый быстрый план выполнения запроса.

Следует отметить, что полный просмотр таблицы далеко не всегда медленнее просмотра по индексу. Если, например, в таблице-справочнике несколько сотен записей, умещающихся в одном-двух блоках на диске, то использование индекса приведёт лишь к тому, что придётся читать ещё и пару лишних блоков индекса. Если в запросе придётся выбрать 80% записей из большой таблицы, то полный просмотр опять же получится быстрее.

При тестировании запросов с использованием EXPLAIN ANALYZE можно воспользоваться настройками, запрещающими оптимизатору использовать определённые планы выполнения. Например,

```
SET enable_seqscan=false;
```

[10] и поэтому EXPLAIN ANALYZE DELETE ... — не слишком хорошая идея

запретит использование полного просмотра таблицы, и вы сможете выяснить, прав ли был оптимизатор, отказываясь от использования индекса. Ни в коем случае *не следует* прописывать подобные команды в `postgresql.conf`! Это может ускорить выполнение нескольких запросов, но сильно замедлит все остальные!

3.2.2 Использование собранной статистики

Результаты работы сборщика статистики (см. пункт 2.4.2) доступны через специальные системные представления. Наиболее интересны для наших целей следующие:

pg_stat_user_tables содержит — для каждой пользовательской таблицы в текущей базе данных — общее количество полных просмотров и просмотров с использованием индексов, общие количества записей, которые были возвращены в результате обоих типов просмотра, а также общие количества вставленных, изменённых и удалённых записей.

pg_stat_user_indexes содержит — для каждого пользовательского индекса в текущей базе данных — общее количество просмотров, использовавших этот индекс, количество прочитанных записей, количество успешно прочитанных записей в таблице (может быть меньше предыдущего значения, если в индексе есть записи, указывающие на устаревшие записи в таблице).

pg_statio_user_tables содержит — для каждой пользовательской таблицы в текущей базе данных — общее количество блоков, прочитанных из таблицы, количество блоков, оказавшихся при этом в буфере (см. пункт 2.1.1), а также аналогичную статистику для всех индексов по таблице и, возможно, по связанной с ней таблицей TOAST.

Из этих представлений можно узнать, в частности

- Для каких таблиц стоит создать новые индексы (индикатором служит большое количество полных просмотров и большое количество прочитанных *блоков*).
- Какие индексы вообще не используются в запросах. Их имеет смысл удалить, если, конечно, речь не идёт об индексах, обеспечивающих выполнение ограничений PRIMARY KEY и UNIQUE.
- Достаточен ли объём буфера сервера.

Также возможен «дедуктивный» подход, при котором сначала создаётся большое количество индексов, а затем неиспользуемые индексы удаляются.

3.2.3 Возможности индексов в PostgreSQL

Функциональные индексы Вы можете построить индекс не только по полю/нескольким полям таблицы, но и по выражению, зависящему от полей. Пусть, например, в вашей таблице foo есть поле foo_name, и выборки часто делаются по условию «первая буква foo_name = 'буква', в любом регистре». Вы можете создать индекс

```
CREATE INDEX foo_name_first_idx
ON foo ((lower(substr(foo_name, 1, 1))));
```

и запрос вида

```
SELECT * FROM foo
WHERE lower(substr(foo_name, 1, 1)) = 'ы';
```

будет его использовать.

Следует отметить, что возможности задания подобных индексов были значительно расширены в версии 7.4, и приведённый пример может потребовать доработки, чтобы быть запущенным на более старой версии.

Частичные индексы (partial indexes) Под частичным индексом понимается индекс с предикатом WHERE. Пусть, например, у вас есть в базе таблица scheta с параметром uplocheno типа boolean. Записей, где uplocheno = false меньше, чем записей с uplocheno = true, а запросы по ним выполняются значительно чаще. Вы можете создать индекс

```
CREATE INDEX scheta_neuplocheno ON scheta (id)
WHERE NOT uplocheno;
```

который будет использоваться запросом вида

```
SELECT * FROM scheta WHERE NOT uplocheno AND ...;
```

Достоинство подхода в том, что записи, не удовлетворяющие условию WHERE, просто не попадут в индекс.

Полнотекстовый поиск Обычные индексы не могут быть использованы в запросах, ищущих, например, вхождение подстроки в строку. Для этого требуются специальные средства полнотекстового поиска.

Наиболее продвинутым из имеющихся средств является tsearch2 http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/ Он поставляется в дистрибутиве PostgreSQL версии 7.4 в каталоге `contrib/tsearch2`, вариант для версии 7.3 можно скачать на указанном сайте.

За полным описанием возможностей tsearch2 обратитесь к поставляемой с ним документации.

3.3 Перенос логики на сторону сервера

Этот пункт очевиден для опытных пользователей PostrgeSQL и предназначен для тех, кто использует или переносит на PostgreSQL приложения, написанные изначально для более примитивных СУБД.

Реализация части логики на стороне сервера через хранимые процедуры, триггеры, правила[11] часто позволяет ускорить работу приложения. Действительно, если несколько запросов объединены в процедуру, то не требуется

- пересылка промежуточных запросов на сервер;
- получение промежуточных результатов на клиент и их обработка.

Кроме того, хранимые процедуры упрощают процесс разработки и поддержки: изменения надо вносить только на стороне сервера, а не менять запросы во всех приложениях.

[11] RULE — реализованное в PostgreSQL расширение стандарта SQL, позволяющее, в частности, создавать обновляемые представления

3.4 Оптимизация конкретных запросов

В этом разделе описываются запросы, для которых по разным причинам *нельзя* заставить оптимизатор использовать индексы, и которые будут всегда вызывать полный просмотр таблицы. Таким образом, если вам требуется использовать эти запросы в требовательном к быстродействию приложении, то придётся их изменить.

3.4.1 SELECT max(...)/min(...) FROM <огромная таблица>

Все агрегатные функции в PostgreSQL реализованы одинаково: сначала выбираются все записи, удовлетворяющие условию, а потом к полученному набору записей применяется агрегатная функция. У такого подхода есть достоинства — вы можете легко написать собственную агрегатную функцию — но есть и недостаток, который заключается в том, что для работы функций типа min() / max() весь набор записей совершенно не нужен.

Для их работы рациональней было бы воспользоваться индексом по полю, для которого ищется максимум (минимум), но для этого придётся сделать реализацию этих агрегатных функций отличной ото всех остальных.

Проблема Запрос вида

```
SELECT max(field) FROM foo;
```

не будет использовать существущий индекс по полю field, а будет делать полный просмотр таблицы. Если записей в таблице много, то это может занять изрядное время.

Решение Запрос вида

```
SELECT field FROM foo ORDER BY field DESC LIMIT 1;
```

вернёт то же самое значение[12], но при этом сможет использовать индекс по field, если таковой существует.

3.4.2 SELECT count(*) FROM <огромная таблица>

К функции count() относится всё вышесказанное по поводу реализации агрегатных функций в PostgreSQL. Кроме того, информация о видимости записи для текущей транзакции (а конкурентным транзакциям может быть видимо *разное* количество записей в таблице!) не хранится в индексе. Таким образом, даже если использовать для выполнения запроса индекс первичного ключа таблицы, всё равно потребуется чтение записей собственно из файла таблицы.

Проблема Запрос вида

```
SELECT count(*) FROM foo;
```

осуществляет полный просмотр таблицы foo, что весьма долго для таблиц с большим количеством записей.

[12] на самом деле почти то же самое: отличие будет в случае, если в таблице нет записей

Решение Простого решения проблемы, к сожалению, нет. Возможны следующие подходы:

1. Если точное число записей не важно, а важен порядок[13], то можно использовать информацию о количестве записей в таблице, собранную при выполнении команды ANALYZE:

```
SELECT reltuples FROM pg_class WHERE relname = 'foo';
```

2. Если подобные выборки выполняются часто, а изменения в таблице достаточно редки, то можно завести вспомогательную таблицу, хранящую число записей в основной. На основную же таблицу повесить триггер, который будет уменьшать это число в случае удаления записи и увеличивать в случае вставки. Таким образом, для получения количества записей потребуется лишь выбрать одну запись из вспомогательной таблицы.

3. Вариант предыдущего подхода, но данные во вспомогательной таблице обновляются через определённые промежутки времени (`cron`).

3.4.3 SELECT ... WHERE ... IN (SELECT ...)

Сразу отметим, что в версии 7.4 в обработку подзапросов с IN / NOT IN были внесены изменения, и теперь они работают (как минимум) не медленнее, чем подзапросы с EXISTS / NOT EXISTS. Если вы по каким-то причинам не можете обновить версию сервера до 7.4, то читайте дальше.

Проблема При использовании подзапроса вида

```
SELECT ...
FROM foo
WHERE foo_field IN (
    SELECT bar_field
    FROM bar
    ...
);
```

оптимизатор не может использовать индекс по таблице bar, и поэтому запрос отрабатывает крайне медленно.

Решение Перепишите подзапрос с использованием конструкции EXISTS:

```
SELECT ...
FROM foo
WHERE EXISTS (
    SELECT bar_field
    FROM bar
    WHERE bar.bar_field = foo.foo_field
    ...
);
```

Аналогично можно переписать подзапрос с NOT IN, используя конструкцию NOT EXISTS.

[13] «на нашем форуме более 10000 зарегистрированных пользователей, оставивших более 50000 сообщений!»

4 Заключение

К счастью, PostgreSQL не требует особо сложной настройки. В большинстве случаев вполне достаточно будет увеличить объём выделенной памяти, настроить периодическое поддержание базы в порядке и проверить наличие необходимых индексов. Более сложные вопросы можно обсудить в специализированном списке рассылки.

Благодарю разработчиков PostgreSQL за создание и поддержку замечательной свободной РСУБД, разработчиков редактора LyX, в котором была написана эта статья, Александра Смирнова (PHPClub) за побуждение к её написанию, Елену Теслю за редакторскую работу.

Текущая версия статьи доступна в интернете по адресу [вставить адрес]. Замечания и исправления просьба направлять на email автору.

Список литературы

[1] PostgreSQL documentation, http://www.postgresql.org/docs/

[2] Momjian B., et. al. PostgreSQL FAQ, http://www.postgresql.org/docs/faqs/FAQ.html

[3] Momjian B. PostgreSQL Hardware Performance Tuning, http://www.ca.postgresql.org/docs/momjian/hw_performance/

[4] Berkus J., Daithankar S. Tuning PostgreSQL for performance, http://www.varlena.com/varlena/GeneralBits/Tidbits/perf.html

[5] Berkus J. Annotated `postgresql.conf` and Global User Configuration (GUC) Guide, http://www.varlena.com/varlena/GeneralBits/Tidbits/annotated_conf_e.html

[6] Berkus J. Adventures in PostgreSQL Episode 2: The Joy of Index, http://techdocs.postgresql.org/techdocs/pgsqladventuresep2.php

[7] Trout J. PostgreSQL: Features, Installation, Administration and Optimization, http://postgres.jefftrout.com/

[8] Argudo J.-P. PostgreSQL Database Performance Tuning, http://www.argudo.org/postgresql/soft-tuning.html