А. Ю. Васильев aka leopard

Работа с PostgreSQL

настройка и масштабирование

4-е издание

leopard.in.ua

Работа с PostgreSQL: настройка и масштабирование

А. Ю. Васильев aka leopard

Оглавление

1

Введение

> Послушайте — и Вы забудете,
> посмотрите — и Вы
> запомните, сделайте — и Вы
> поймете.
>
> Конфуций

Данная книга не дает ответы на все вопросы по работе с PostgreSQL. Главное её задание — показать возможности PostgreSQL, методики настройки и масштабируемости этой СУБД. В любом случае, выбор метода решения поставленной задачи остается за разработчиком или администратором СУБД.

2

Настройка производительности

> Теперь я знаю тысячу
> способов, как не нужно
> делать лампу накаливания.
>
> Томас Алва Эдисон

2.1 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 глава посвящена возможностям повышения производительности PostgreSQL. Глава не претендует на исчерпывающее изложение вопроса, наиболее полным и точным руководством по использованию PostgreSQL является, конечно, официальная документация и официальный FAQ. Также существует англоязычный список рассылки postgresql-performance, посвящённый именно этим вопросам. Глава 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.

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, веб-сервер).

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);
- В версии 8.0 были внедрены метки восстановления, улучшение управления буфером, CHECKPOINT и VACUUM улучшены;
- В версии 8.1 был улучшен одновременный доступ к разделяемой памяти, автоматическое использование индексов для MIN() и MAX(), pg_autovacuum внедрен в сервер (автоматизирован), повышение производительности для секционированных таблиц;
- В версии 8.2 была улучшена скорость множества SQL запросов, усовершенствован сам язык запросов;

- В версии 8.3 внедрен полнотекстовый поиск, поддержка SQL/XML стандарта, параметры конфигурации сервера могут быть установлены на основе отдельных функций;
- В версии 8.4 были внедрены общие табличные выражения, рекурсивные запросы, параллельное восстановление, улучшена производительность для EXISTS/NOT EXISTS запросов;
- В версии 9.0 «асинхронная репликация из коробки», VACUUM/VACUUM FULL стали быстрее, расширены хранимые процедуры;
- В версии 9.1 «синхронная репликация из коробки», нелогируемые таблицы (очень быстрые на запись, но при падении БД данные могут пропасть), новые типы индексов, наследование таблиц в запросах теперь может вернуться многозначительно отсортированные результаты, позволяющие оптимизации MIN/MAX;
- В версии 9.2 «каскадная репликация из коробки», сканирование по индексу, JSON тип данных, типы данных на диапазоны, сортировка в памяти улучшена на 25%, ускорена команда COPY;
- В версии 9.3 materialized view, доступные на запись внешние таблицы, переход с использования SysV shared memory на POSIX shared memory и mmap, сокращено время распространения реплик, а также значительно ускорена передача управления от запасного сервера к первичному, увеличена производительность и улучшена система блокировок для внешних ключей;

Следует также отметить, что большая часть изложенного в статье материала относится к версии сервера не ниже 8.4.

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. «2.5 Оптимизация конкретных запросов»);

2.2 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.

Используемая память

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.

В то же время не следует устанавливать это значение слишком большим: это НЕ вся память, которая нужна для работы PostgreSQL, это только размер разделяемой между процессами PostgreSQL памяти, которая нужна для выполнения активных операций. Она должна занимать меньшую часть оперативной памяти вашего компьютера, так как PostgreSQL полагается на то, что операционная система кэширует файлы, 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.

К сожалению, чтобы знать точное число shared_buffers, нужно учесть количество оперативной памяти компьютера, размер базы данных, число соединений и сложность запросов, так что лучше воспользуемся несколькими простыми правилами настройки.

На выделенных серверах полезным объемом будет значение от 8 МБ до 2 ГБ. Объем может быть выше, если у вас большие активные порции базы данных, сложные запросы, большое число одновременных соединений, длительные транзакции, вам доступен большой объем оперативной памяти или большее количество процессоров. И, конечно же, не забываем об остальных приложениях. Выделив слишком много памяти для базы данных, мы можем получить ухудшение производительности. 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(например, free или vmstat). Рекомендуемое значение параметра будет примерно в 1,2 –2 раза больше, чем максимум использованной памяти. Обратите внимание, что память под буфер выделяется 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 (для версии PostgreSQL $<$ 9.3). В руководстве администратора PostgreSQL описано, как можно изменить эти настройки: www.postgresql.org

Вот несколько примеров, полученных на личном опыте и при тестировании:

- Laptop, Celeron processor, 384 МБ RAM, база данных 25 МБ: 12 МБ;
- Athlon server, 1 ГБ RAM, база данных поддержки принятия решений 10 ГБ: 200 МБ;
- Quad PIII server, 4 ГБ RAM, 40 ГБ, 150 соединений, «тяжелые» транзакции: 1 ГБ;
- Quad Xeon server, 8 ГБ RAM, 200 ГБ, 300 соединений, «тяжелые» транзакции: 2 ГБ.

Память для сортировки результата запроса: work_mem

Ранее известное как sort_mem, было переименовано, так как сейчас определяет максимальное количество оперативной памяти, которое может выделить одна операция сортировки, агрегации и др. Это не разделяемая память, work_mem выделяется отдельно на каждую операцию (от одного до нескольких раз за один запрос). Разумное значение параметра определяется следующим образом: количество доступной оперативной памяти (после того, как из общего объема вычли память, требуемую для других

приложений, и shared_buffers) делится на максимальное число одновременных запросов умноженное на среднее число операций в запросе, которые требуют памяти.

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

Объём памяти задаётся параметром work_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. Для веб-приложений обычно устанавливают низкие значения work_mem, так как запросов обычно много, но они простые, обычно хватает от 512 до 2048 КБ. С другой стороны, приложения для поддержки принятия решений с сотнями строк в каждом запросе и десятками миллионов столбцов в таблицах фактов часто требуют work_mem порядка 500 МБ. Для баз данных, которые используются и так, и так, этот параметр можно устанавливать для каждого запроса индивидуально, используя настройки сессии. Например, при памяти 1–4 ГБ рекомендуется устанавливать 32–128 MB.

Память для работы команды VACUUM: maintenance_work_mem

Предыдущее название в PostgreSQL 7.x vacuum_mem. Этот параметр задаёт объём памяти, используемый командами VACUUM, ANALYZE, CREATE INDEX, и добавления внешних ключей. Чтобы операции выполнялись максимально быстро, нужно устанавливать этот параметр тем выше, чем больше размер таблиц в вашей базе данных. Неплохо бы устанавливать его значение от 50 до 75% размера вашей самой большой таблицы или индекса или, если точно определить невозможно, от 32 до 256 МБ. Следует устанавливать большее значение, чем для work_mem. Слишком большие значения приведут к использованию свопа. Например, при памяти 1–4 ГБ рекомендуется устанавливать 128–512 MB.

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[1] (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).

[1] 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 в общей карте свободного пространства. Эти данные собираются VACUUM. Параметр max_fsm_relations должен быть не меньше общего количества таблиц во всех базах данной установки (лучше с запасом);
- max_fsm_pages Данный параметр определяет размер реестра, в котором хранится информация о частично освобождённых страницах данных, готовых к заполнению новыми данными. Значение этого параметра нужно установить чуть больше, чем полное число страниц, которые могут быть затронуты операциями обновления или удаления между выполнением VACUUM. Чтобы определить это число, можно запустить VACUUM VERBOSE ANALYZE и выяснить общее число страниц, используемых базой данных. max_fsm_pages обычно требует немного памяти, так что на этом параметре лучше не экономить.

Если эти параметры установлены верно и 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.

ВНИМАНИЕ! Начиная с 8.4 версии fsm параметры были убраны, поскольку Free Space Map сохраняется на жесткий диск, а не в память.

Прочие настройки

- temp_buffers Буфер под временные объекты, в основном для временных таблиц. Можно установить порядка 16 МБ;
- max_prepared_transactions Количество одновременно подготавливаемых транзакций (PREPARE TRANSACTION). Можно оставить по дефолту — 5;
- vacuum_cost_delay Если у вас большие таблицы, и производится много одновременных операций записи, вам может пригодиться функция, которая уменьшает затраты на I/O для VACUUM, растягивая его по времени. Чтобы включить эту функциональность, нужно поднять значение vacuum_cost_delay выше 0. Используйте разумную

задержку от 50 до 200 мс. Для более тонкой настройки повышайте vacuum_cost_page_hit и понижайте vacuum_cost_page_limit. Это ослабит влияние VACUUM, увеличив время его выполнения. В тестах с параллельными транзакциями Ян Вик (Jan Wieck) получил, что при значениях delay — 200, page_hit — 6 и limit —100 влияние VACUUM уменьшилось более чем на 80%, но его длительность увеличилась втрое;

- max_stack_depth Специальный стек для сервера, в идеале он должен совпадать с размером стека, выставленном в ядре ОС. Установка большего значения, чем в ядре, может привести к ошибкам. Рекомендуется устанавливать 2–4 MB;
- max_files_per_process Максимальное количество файлов, открываемых процессом и его подпроцессами в один момент времени.Уменьшите данный параметр, если в процессе работы наблюдается сообщение «Too many open files»;

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.

Уменьшение количества контрольных точек: checkpoint_segments

Если в базу заносятся большие объёмы данных, то контрольные точки могут происходить слишком часто[1]. 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). Данный параметр определяет количество сегментов (каждый по 16 МБ)

[1] «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.

лога транзакций между контрольными точками. Этот параметр не имеет особого значения для базы данных, предназначенной преимущественно для чтения, но для баз данных со множеством транзакций увеличение этого параметра может оказаться жизненно необходимым. В зависимости от объема данных установите этот параметр в диапазоне от 12 до 256 сегментов и, если в логе появляются предупреждения (warning) о том, что контрольные точки происходят слишком часто, постепенно увеличивайте его. Место, требуемое на диске, вычисляется по формуле (checkpoint_segments * 2 + 1) * 16 МБ, так что убедитесь, что у вас достаточно свободного места. Например, если вы выставите значение 32, вам потребуется больше 1 ГБ дискового пространства.

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

fsync, synchronous_commit и стоит ли их трогать

Наиболее радикальное из возможных решений — выставить значение «off» 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.

Параметр synchronous_commit определяет нужно ли ждать WAL записи на диск перед возвратом успешного завершения транзакции для подключенного клиента. По умолчанию и для безопасности данный параметр установлен в «on» (включен). При выключении данного параметра («off») может существовать задержка между моментом, когда клиенту будет сообщенно об успехе транзакции и когда та самая транзакция действительно гарантированно и безопастно записана на диск (максимальная задержка - wal_writer_delay * 3). В отличие от fsync, отключение этого параметра не создает риск краха базы данных: данные могут быть потеряны (последний набор транзакций), но базу данных не придется восстанавливать после сбоя из бэкапа. Так что synchronous_commit может быть полезной альтернативой, когда производительность важнее, чем точная уверенность в согласовании данных (данный режим можно назвать «режимом MongoDB»: изначально все клиенты для MongoDB не проверяли успешность записи данных в базу и за счет этого достигалась хорошая скорость для бенчмарков).

Прочие настройки

- 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.
- wal_sync_method Метод, который используется для принудительной записи данных на диск. Если fsync=off, то этот параметр не используется. Возможные значения:

  – open_datasync — запись данных методом open() с параметром O_DSYNC;
  – fdatasync — вызов метода fdatasync() после каждого commit;
  – fsync_writethrough — вызов fsync() после каждого commit, игнорируя параллельные процессы;
  – fsync — вызов fsync() после каждого commit;
  – open_sync — запись данных методом open() с параметром O_SYNC;

  Не все эти методы доступны на разных ОС. По умолчанию устанавливается первый, который доступен для системы.
- full_page_writes Установите данный параметр в off, если fsync=off. Иначе, когда этот параметр on, PostgreSQL записывает содержимое каждой записи в журнал транзакций при первой модификации таблицы. Это необходимо, поскольку данные могут записаться лишь частично, если в ходе процесса «упала» ОС. Это приведет к тому, что на диске окажутся новые данные смешанные со старыми. Строкового уровня записи в журнал транзакций может быть недостаточно, чтобы полностью восстановить данные после «падения». full_page_writes гарантирует корректное восстановление, ценой увеличения записываемых данных в журнал транзакций (Единственный способ снижения объема записи в журнал транзакций заключается в увеличении checkpoint_interval).
- wal_buffers Количество памяти используемое в SHARED MEMORY для ведения транзакционных логов[1]. Стоит увеличить буфер до 256–512 кБ, что позволит лучше работать с большими транзакциями. Например, при доступной памяти 1–4 ГБ рекомендуется устанавливать 256–1024 КБ.

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

Планировщик запросов

Следующие настройки помогают планировщику запросов правильно оценивать стоимости различных операций и выбирать оптимальный план выполнения запроса. Существуют 3 настройки планировщика, на которые стоит обратить внимание:

- default_statistics_target

  Этот параметр задаёт объём 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.
- effective_cache_size

  Этот параметр сообщает PostgreSQL примерный объём файлового кэша операционной системы, оптимизатор использует эту оценку для построения плана запроса[1].

  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.

  На выделенном сервере имеет смысл выставлять effective_cache_size в 2/3 от всей оперативной памяти; на сервере с другими приложениями сначала нужно вычесть из всего объема RAM размер дискового кэша ОС и память, занятую остальными процессами.
- random_page_cost

  Переменная, указывающая на условную стоимость индексного доступа к страницам данных. На серверах с быстрыми дисковыми массивами имеет смысл уменьшать изначальную настройку до 3.0, 2.5 или даже до 2.0. Если же активная часть вашей базы данных намного больше размеров оперативной памяти, попробуйте поднять значение параметра. Можно подойти к выбору оптимального значения и со стороны производительности запросов. Если планировщик запросов чаще, чем необходимо, предпочитает последовательные просмотры (sequential scans) просмотрам с использованием индекса (index

[1] Указывает планировщику на размер самого большого объекта в базе данных, который теоретически может быть закеширован

scans), понижайте значение. И наоборот, если планировщик выбирает просмотр по медленному индексу, когда не должен этого делать, настройку имеет смысл увеличить. После изменения тщательно тестируйте результаты на максимально широком наборе запросов. Никогда не опускайте значение random_page_cost ниже 2.0; если вам кажется, что random_page_cost нужно еще понижать, разумнее в этом случае менять настройки статистики планировщика.

Сбор статистики

У PostgreSQL также есть специальная подсистема — сборщик статистики, — которая в реальном времени собирает данные об активности сервера. Поскольку сбор статистики создает дополнительные накладные расходы на базу данных, то система может быть настроена как на сбор, так и не сбор статистики вообще. Эта система контролируется следующими параметрами, принимающими значения true/false:

- track_counts включать ли сбор статистики. По умолчанию включён, поскольку autovacuum демону требуется сбор статистики. Отключайте, только если статистика вас совершенно не интересует (как и autovacuum);
- track_functions отслеживание использования определенных пользователем функций;
- track_activities передавать ли сборщику статистики информацию о текущей выполняемой команде и времени начала её выполнения. По умолчанию эта возможность включена. 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.

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 (а также помогут оптимальной работе autovacuum демону).

2.3 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 главы, 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[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 (несколько логических разделов на одном диске здесь, очевидно, не помогут: головка всё равно будет одна), 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 на отдельном диске магнитная головка не будет лишний раз двигаться, что позволит ускорить запись.

Порядок действий:

- Остановите сервер (!);
- Перенесите каталоги pg_clog и 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.

CLUSTER

CLUSTER table [USING index] — команда для упорядочивания записей таблицы на диске согласно индексу, что иногда за счет уменьшения доступа к диску ускоряет выполнение запроса. Возможно создать только один физический порядок в таблице, поэтому и таблица может иметь только один кластерный индекс. При таком условии нужно тщательно выбирать, какой индекс будет использоваться для кластерного индекса.

Кластеризация по индексу позволяет сократить время поиска по диску: во время поиска по индексу выборка данных может быть значительно

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

быстрее, так как последовательность данных в таком же порядке, как и индекс. Из минусов можно отметить то, что команда CLUSTER требует «ACCESS EXCLUSIVE» блокировку, что предотвращает любые другие операции с данными (чтения и записи) пока кластеризация не завершит выполнение. Также кластеризация индекса в PostgreSQL не утверждает четкий порядок следования, поэтому требуется повторно выполнять CLUSTER для поддержания таблицы в порядке.

2.4 Утилиты для тюнинга PostgreSQL

Pgtune

Для оптимизации настроек для PostgreSQL Gregory Smith создал утилиту pgtune в расчёте на обеспечение максимальной производительности для заданной аппаратной конфигурации. Утилита проста в использовании и во многих Linux системах может идти в составе пакетов. Если же нет, можно просто скачать архив и распаковать. Для начала:

Листинг 2.1 Pgtune

```
Line 1 $ pgtune -i $PGDATA/postgresql.conf -o $PGDATA/postgresql.
       conf.pgtune
```

опцией -i, --input-config указываем текущий файл postgresql.conf, а -o, --output-config указываем имя файла для нового postgresql.conf.

Есть также дополнительные опции для настройки конфига:

- -M, --memory используйте этот параметр, чтобы определить общий объем системной памяти. Если не указано, pgtune будет пытаться использовать текущий объем системной памяти;
- -T, --type указывает тип базы данных. Опции: DW, OLTP, Web, Mixed, Desktop;
- -c, --connections указывает максимальное количество соединений. Если он не указан, то будет браться в зависимости от типа базы данных.

Существует также онлайн версия pgtune.

Хочется сразу добавить, что pgtune не панацея для оптимизации настройки PostgreSQL. Многие настройки зависят не только от аппаратной конфигурации, но и от размера базы данных, числа соединений и сложности запросов, так что оптимально настроить базу данных возможно только учитывая все эти параметры.

pg_buffercache

Pg_buffercache — расширение для PostgreSQL, которое позволяет получить представление об использовании общего буфера (shared_buffer) в

базе. Расширение позволяет взглянуть какие из данных кэширует база, которые активно используются в запросах. Для начала нужно установить расширение:

Листинг 2.2 pg_buffercache

```
# CREATE EXTENSION pg_buffercache;
```

Теперь доступно pg_buffercache представление, которое содержит:

- bufferid — ID блока в общем буфере;
- relfilenode — имя папки, где данные расположены;
- reltablespace — Oid таблицы;
- reldatabase — Oid базы данных;
- relforknumber — номер ответвления;
- relblocknumber — номер страницы;
- isdirty — грязная страница?;
- usagecount — количество LRU страниц.

ID блока в общем буфере (bufferid) соответствует количеству используемого буфера таблицей, индексом, прочим. Общее количество доступных буферов определяется двумя вещами:

- Размер буферного блока. Этот размер блока определяется опцией --with-blocksize при конфигурации. Значение по умолчанию — 8 КБ, что достаточно в большинстве случаев, но его возможно увеличить или уменьшить в зависимости от ситуации. Для того чтобы изменить это значение, необходимо будет перекомпилировать PostgreSQL;
- Размер общего буфера. Определяется опцией shared_buffers в PostgreSQL конфиге.

Например, при использовании shared_buffers в 128 МБ с 8 КБ размера блока получится 16384 буферов. Представление pg_buffercache будет иметь такое же число строк — 16384. С shared_buffers в 256 МБ и размером блока в 1 КБ получим 262144 буферов.

Для примера рассмотрим простой запрос показывающий использование буферов объектами (таблицами, индексами, прочим):

Листинг 2.3 pg_buffercache

```
# SELECT c.relname, count(*) AS buffers
FROM pg_buffercache b INNER JOIN pg_class c
ON b.relfilenode = pg_relation_filenode(c.oid) AND
b.reldatabase IN (0, (SELECT oid FROM pg_database WHERE
    datname = current_database()))
GROUP BY c.relname
ORDER BY 2 DESC
LIMIT 10;
```

```
                relname                 | buffers
----------------------------------------+---------
 pgbench_accounts                       |    4082
 pgbench_history                        |      53
 pg_attribute                           |      23
 pg_proc                                |      14
 pg_operator                            |      11
 pg_proc_oid_index                      |       9
 pg_class                               |       8
 pg_attribute_relid_attnum_index        |       7
 pg_proc_proname_args_nsp_index         |       6
 pg_class_oid_index                     |       5
(10 rows)
```

Этот запрос показывает объекты (таблицы и индексы) в кэше:

Листинг 2.4 pg_buffercache

```
# SELECT c.relname, count(*) AS buffers,usagecount
 FROM pg_class c
 INNER JOIN pg_buffercache b
 ON b.relfilenode = c.relfilenode
 INNER JOIN pg_database d
 ON (b.reldatabase = d.oid AND d.datname = current_database
    ())
GROUP BY c.relname,usagecount
ORDER BY c.relname,usagecount;

                relname                 | buffers | usagecount
----------------------------------------+---------+------------
 pg_rewrite                             |       3 |          1
 pg_rewrite_rel_rulename_index          |       1 |          1
 pg_rewrite_rel_rulename_index          |       1 |          2
 pg_statistic                           |       1 |          1
 pg_statistic                           |       1 |          3
 pg_statistic                           |       2 |          5
 pg_statistic_relid_att_inh_index       |       1 |          1
 pg_statistic_relid_att_inh_index       |       3 |          5
 pgbench_accounts                       |    4082 |          2
 pgbench_accounts_pkey                  |       1 |          1
 pgbench_history                        |      53 |          1
 pgbench_tellers                        |       1 |          1
```

Это запрос показывает какой процент общего буфера используют обьекты (таблицы и индексы) и на сколько процентов объекты находятся в самом кэше (буфере):

Листинг 2.5 pg_buffercache

```
# SELECT
```

```
-   c.relname,
-   pg_size_pretty(count(*) * 8192) as buffered,
-   round(100.0 * count(*) /
5   (SELECT setting FROM pg_settings WHERE name='shared_buffers
      ')::integer,1)
-   AS buffers_percent,
-   round(100.0 * count(*) * 8192 / pg_table_size(c.oid),1)
-   AS percent_of_relation
- FROM pg_class c
10  INNER JOIN pg_buffercache b
-   ON b.relfilenode = c.relfilenode
-   INNER JOIN pg_database d
-   ON (b.reldatabase = d.oid AND d.datname = current_database
      ())
- GROUP BY c.oid,c.relname
15 ORDER BY 3 DESC
- LIMIT 20;
-
- -[ RECORD 1 ]-------+---------------------------------
-   relname            | pgbench_accounts
20  buffered           | 32 MB
-   buffers_percent    | 24.9
-   percent_of_relation | 99.9
- -[ RECORD 2 ]-------+---------------------------------
-   relname            | pgbench_history
25  buffered           | 424 kB
-   buffers_percent    | 0.3
-   percent_of_relation | 94.6
- -[ RECORD 3 ]-------+---------------------------------
-   relname            | pg_operator
30  buffered           | 88 kB
-   buffers_percent    | 0.1
-   percent_of_relation | 61.1
- -[ RECORD 4 ]-------+---------------------------------
-   relname            | pg_opclass_oid_index
35  buffered           | 16 kB
-   buffers_percent    | 0.0
-   percent_of_relation | 100.0
- -[ RECORD 5 ]-------+---------------------------------
-   relname            | pg_statistic_relid_att_inh_index
40  buffered           | 32 kB
-   buffers_percent    | 0.0
-   percent_of_relation | 100.0
```

Используя эти данные можно проанализировать для каких объектов не хватает памяти или какие из них потребляют основную часть общего буфера. На основе этих данных можно более правильно делать тюнинг shared_buffers параметра для PostgreSQL.

2.5 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:
   a) Грамотное проектирование базы. Освещение этого вопроса выходит далеко за рамки этой книги;
   b) 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.

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 оптимальную частоту.

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 с 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.

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, и поэтому файл с индексом постепенно увеличивается в размерах. Если данные в таблице часто меняются, то расти он может весьма быстро.

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.

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 внешними ключами, и поля, по которым объединяются таблицы, индексировать надо обязательно.

Команда EXPLAIN [ANALYZE]

Команда EXPLAIN запрос[] показывает, каким образом PostgreSQL собирается выполнять ваш запрос. Команда EXPLAIN ANALYZE запрос[] выполняет запрос[1] 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

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

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,

Листинг 2.6 enable_seqscan

```
Line 1 SET enable_seqscan=false;
```

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!

Использование собранной статистики

Результаты работы сборщика статистики 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.

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

Листинг 2.7 Индекс

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

и запрос вида

Листинг 2.8 Запрос

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

будет его использовать.

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

Листинг 2.9 Индекс

```
Line 1 CREATE INDEX scheta_neuplocheno ON scheta (id) WHERE NOT
       uplocheno;
```

который будет использоваться запросом вида

Листинг 2.10 Запрос

```
Line 1 SELECT * FROM scheta WHERE NOT uplocheno AND ...;
```

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

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[1] 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.

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.

SELECT count(*) FROM <огромная таблица>

Функция count() работает очень просто: сначала выбираются все записи, удовлетворяющие условию, а потом к полученному набору записей применяется агрегатная функция — считается количество выбранных строк. Информация о видимости записи для текущей транзакции (а конкурентным транзакциям может быть видимо разное количество записей в таблице!) не хранится в индексе, поэтому, даже если использовать для выполнения запроса индекс первичного ключа таблицы, всё равно потребуется чтение записей собственно из файла таблицы.

Проблема Запрос вида

Листинг 2.11 SQL

```
Line 1 SELECT count(*) FROM foo;
```

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

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

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

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

Листинг 2.12 SQL

```
Line 1 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).

Медленный DISTINCT

Текущая реализация DISTINCT для больших таблиц очень медленна. Но возможно использовать GROUP BY взамен DISTINCT. GROUP BY может использовать агрегирующий хэш, что значительно быстрее, чем DISTINCT (актуально до версии 8.4 и ниже).

Листинг 2.13 DISTINCT

```
Line 1  postgres=# select count(*) from (select distinct i from g) a
        ;
    -    count
    -   -------
    -    19125
    5   (1 row)
    -
    -   Time: 580,553 ms
    -
    -
   10   postgres=# select count(*) from (select distinct i from g) a
        ;
    -    count
    -   -------
    -    19125
    -   (1 row)
   15
    -   Time: 36,281 ms
```

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

Листинг 2.14 GROUP BY

```
postgres=# select count(*) from (select i from g group by i)
    a;
 count
-------
 19125
(1 row)

Time: 26,562 ms


postgres=# select count(*) from (select i from g group by i)
    a;
 count
-------
 19125
(1 row)

Time: 25,270 ms
```

Утилиты для оптимизации запросов

pgFouine

pgFouine — это анализатор log-файлов для PostgreSQL, используемый для генерации детальных отчетов из log-файлов PostgreSQL. pgFouine поможет определить, какие запросы следует оптимизировать в первую очередь. pgFouine написан на языке программирования PHP с использованием объектно-ориентированных технологий и легко расширяется для поддержки специализированных отчетов, является свободным программным обеспечением и распространяется на условиях GNU General Public License. Утилита спроектирована таким образом, чтобы обработка очень больших log-файлов не требовала много ресурсов.

Для работы с pgFouine сначала нужно сконфигурировать PostgreSQL для создания нужного формата log-файлов:

- Чтобы включить протоколирование в syslog

Листинг 2.15 pgFouine

```
  log_destination = 'syslog'
  redirect_stderr = off
  silent_mode = on

```

- Для записи запросов, длящихся дольше n миллисекунд:

Листинг 2.16 pgFouine

```
Line 1    log_min_duration_statement = n
    -     log_duration = off
    -     log_statement = 'none'
    -
```

Для записи каждого обработанного запроса установите log_min_duration_statement на 0. Чтобы отключить запись запросов, установите этот параметр на -1.

pgFouine — простой в использовании инструмент командной строки. Следующая команда создаёт HTML-отчёт со стандартными параметрами:

Листинг 2.17 pgFouine

```
Line 1  pgfouine.php -file your/log/file.log > your-report.html
```

С помощью этой строки можно отобразить текстовый отчёт с 10 запросами на каждый экран на стандартном выводе:

Листинг 2.18 pgFouine

```
Line 1  pgfouine.php -file your/log/file.log -top 10 -format text
```

Более подробно о возможностях, а также много полезных примеров, можно найти на официальном сайта проекта pgfouine.projects.pgfoundry.org.

pgBadger

pgBadger — аналогичная утилита, что и pgFouine, но написанная на Perl. Еще одно большое преимущество проекта в том, что он более активно сейчас разрабатывается (на момент написания этого текста последний релиз pgFouine был в 24.02.2010, а последняя версия pgBadger — 12.10.2012). Установка pgBadger проста:

Листинг 2.19 Установка pgBadger

```
Line 1  $ tar xzf pgbadger-2.x.tar.gz
    -   $ cd pgbadger-2.x/
    -   $ perl Makefile.PL
    -   $ make && sudo make install
```

Как и в случае с pgFouine нужно настроить PostgreSQL логи:

Листинг 2.20 Настройка логов PostgreSQL

```
Line 1  logging_collector = on
    -   log_min_messages = debug1
    -   log_min_error_statement = debug1
    -   log_min_duration_statement = 0
    5   log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d '
    -   log_checkpoints = on
    -   log_connections = on
```

```
log_disconnections = on
log_lock_waits = on
log_temp_files = 0
```

Парсим логи PostgreSQL через pgBadger:

Листинг 2.21 Запуск pgBadger

```
$ ./pgbadger ~/pgsql/master/pg_log/postgresql-2012-08-30_132*
[==========>] Parsed 10485768 bytes of 10485768 (100.00%)
[==========>] Parsed 10485828 bytes of 10485828 (100.00%)
[==========>] Parsed 10485851 bytes of 10485851 (100.00%)
[==========>] Parsed 10485848 bytes of 10485848 (100.00%)
[==========>] Parsed 10485839 bytes of 10485839 (100.00%)
[==========>] Parsed 982536 bytes of 982536 (100.00%)
```

В результате получится HTML файлы, которые содержат статистику по запросам к PostgreSQL. Более подробно о возможностях можно найти на официальном сайта проекта dalibo.github.com/pgbadger.

pg_stat_statements

pg_stat_statements — расширение для сбора статистики выполнения запросов в рамках всего сервера. Преимущество данного расширения в том, что ему не требуется собирать и парсить логи PostgreSQL, как это делает pgFouine и pgBadger. Для начала установим и настроим его:

Листинг 2.22 Настройка pg_stat_statements в postgresql.conf

```
shared_preload_libraries = 'pg_stat_statements'
custom_variable_classes = 'pg_stat_statements' # данная настройка нужна для PostgreSQL 9.1 и ниже

pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

После внесения этих параметров PostgreSQL потребуется перегрузить. Параметры конфигурации pg_stat_statements:

1. pg_stat_statements.max (integer)» — максимальное количество sql запросов, которые будет хранится расширением (удаляются записи с наименьшим количеством вызовов);

2. pg_stat_statements.track (enum)» — какие SQL запросы требуется записывать. Возможные параметры: top (только запросы от приложения/клиента), all (все запросы, например в функциях) и none (отключить сбор статистики);
3. pg_stat_statements.save (boolean)» — следует ли сохранять собранную статистику после остановки PostgreSQL. По умолчанию включено.

Далее активируем расширение:

Листинг 2.23 Активация pg_stat_statements

```
Line 1 # CREATE EXTENSION pg_stat_statements;
```

Пример собранной статистики:

Листинг 2.24 pg_stat_statements статистика

```
Line 1 # SELECT query, calls, total_time, rows, 100.0 *
    shared_blks_hit /
-                 nullif(shared_blks_hit + shared_blks_read, 0)
     AS hit_percent
-           FROM pg_stat_statements ORDER BY total_time DESC
    LIMIT 10;
- -[ RECORD 1 ]--
    ----------------------------------------------------------------------

5 query        | SELECT query, calls, total_time, rows, ? *
    shared_blks_hit /
-              |                 nullif(shared_blks_hit +
    shared_blks_read, ?) AS hit_percent
-              |           FROM pg_stat_statements ORDER BY
    total_time DESC LIMIT ?;
- calls        | 3
- total_time   | 0.994
10 rows         | 7
- hit_percent | 100.0000000000000000
- -[ RECORD 2 ]--
    ----------------------------------------------------------------------

- query        | insert into x (i) select generate_series(?,?);
- calls        | 2
15 total_time   | 0.591
- rows         | 110
- hit_percent | 100.0000000000000000
- -[ RECORD 3 ]--
    ----------------------------------------------------------------------

- query        | select * from x where i = ?;
20 calls        | 2
```

```
- total_time   | 0.157
- rows         | 6
- hit_percent  | 100.0000000000000000
- -[ RECORD 4 ]--
    ----------------------------------------------------------------------------

25 query        | SELECT pg_stat_statements_reset();
- calls        | 1
- total_time   | 0.102
- rows         | 1
- hit_percent  |
```

Для сброса статистики есть команда pg_stat_statements_reset:

Листинг 2.25 Сброс статистика

```
Line 1 # SELECT pg_stat_statements_reset();
- -[ RECORD 1 ]------------+-
- pg_stat_statements_reset |
-
5 # SELECT query, calls, total_time, rows, 100.0 *
    shared_blks_hit /
-                 nullif(shared_blks_hit + shared_blks_read, 0)
    AS hit_percent
-           FROM pg_stat_statements ORDER BY total_time DESC
    LIMIT 10;
- -[ RECORD 1 ]----------------------------------
- query        | SELECT pg_stat_statements_reset();
10 calls        | 1
- total_time   | 0.175
- rows         | 1
- hit_percent  |
```

Хочется сразу отметить, что расширение только с версии PostgreSQL 9.2 contrib нормализирует SQL запросы. В версиях 9.1 и ниже SQL запросы сохраняются как есть, а значит «select * from table where id = 3» и «select * from table where id = 21» буду разными записями, что почти бесполезно для сбора полезной статистики.

2.6 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.

3

Партиционирование

> Решая какую-либо проблему, всегда полезно заранее знать правильный ответ. При условии, конечно, что вы уверены в наличии самой проблемы.
>
> Народная мудрость

3.1 Введение

Партиционирование (partitioning, секционирование) — это разбиение больших структур баз данных (таблицы, индексы) на меньшие кусочки. Звучит сложно, но на практике все просто.

Скорее всего у Вас есть несколько огромных таблиц (обычно всю нагрузку обеспечивают всего несколько таблиц СУБД из всех имеющихся). Причем чтение в большинстве случаев приходится только на самую последнюю их часть (т.е. активно читаются те данные, которые недавно появились). Примером тому может служить блог — на первую страницу (это последние 5...10 постов) приходится 40...50% всей нагрузки, или новостной портал (суть одна и та же), или системы личных сообщений, впрочем понятно. Партиционирование таблицы позволяет базе данных делать интеллектуальную выборку — сначала СУБД уточнит, какой партиции соответствует Ваш запрос (если это реально) и только потом сделает этот запрос, применительно к нужной партиции (или нескольким партициям). Таким образом, в рассмотренном случае, Вы распределите нагрузку на таблицу по ее партициям. Следовательно выборка типа `SELECT * FROM articles ORDER BY id DESC LIMIT 10` будет выполняться только над последней партицией, которая значительно меньше всей таблицы.

Итак, партиционирование дает ряд преимуществ:

- На определенные виды запросов (которые, в свою очередь, создают основную нагрузку на СУБД) мы можем улучшить производительность;

- Массовое удаление может быть произведено путем удаления одной или нескольких партиций (DROP TABLE гораздо быстрее, чем массовый DELETE);
- Редко используемые данные могут быть перенесены в другое хранилище.

3.2 Теория

На текущий момент PostgreSQL поддерживает два критерия для создания партиций:

- Партиционирование по диапазону значений (range) — таблица разбивается на «диапазоны» значений по полю или набору полей в таблице, без перекрытия диапазонов значений, отнесенных к различным партициям. Например, диапазоны дат;
- Партиционирование по списку значений (list) — таблица разбивается по спискам ключевых значений для каждой партиции.

Чтобы настроить партиционирование таблицы, достаточно выполните следующие действия:

- Создается «мастер» таблица, из которой все партиции будут наследоваться. Эта таблица не будет содержать данные. Также не нужно ставить никаких ограничений на таблицу, если конечно они не будут дублироваться на партиции;
- Создайте несколько «дочерних» таблиц, которые наследуют от «мастер» таблицы;
- Добавить в «дочерние» таблицы значения, по которым они будут партициями. Стоить заметить, что значения партиций не должны пересекаться. Например:

Листинг 3.1 Пример неверного задания значений партиций

```
Line 1  CHECK ( outletID BETWEEN 100 AND 200 )
    -   CHECK ( outletID BETWEEN 200 AND 300 )
```

  неверно заданы партиции, поскольку непонятно какой партиции принадлежит значение 200;
- Для каждой партиции создать индекс по ключевому полю (или нескольким), а также указать любые другие требуемые индексы;
- При необходимости, создать триггер или правило для перенаправления данных с «мастер» таблицы в соответствующую партицию;
- Убедиться, что параметр constraint_exclusion не отключен в postgresql.conf. Если его не включить, то запросы не будут оптимизированы при работе с партиционированием.

3.3 Практика использования

Теперь начнем с практического примера. Представим, что в нашей системе есть таблица, в которую мы собираем данные о посещаемости нашего ресурса. На любой запрос пользователя наша система логирует действия в эту таблицу. И, например, в начале каждого месяца (неделю) нам нужно создавать отчет за предыдущий месяц (неделю). При этом, логи нужно хранить в течении 3 лет. Данные в такой таблице накапливаются быстро, если система активно используется. И вот, когда в таблице уже миллионы, а то, и миллиарды записей, создавать отчеты становится все сложнее (да и чистка старых записей становится не легким делом). Работа с такой таблицей создает огромную нагрузку на СУБД. Тут нам на помощь и приходит партиционирование.

Настройка

Для примера, мы имеем следующую таблицу:

Листинг 3.2 «Мастер» таблица

```
Line 1  CREATE TABLE my_logs (
        id              SERIAL PRIMARY KEY,
        user_id         INT NOT NULL,
        logdate         TIMESTAMP NOT NULL,
5       data            TEXT,
        some_state      INT
);
```

Поскольку нам нужны отчеты каждый месяц, мы будем делить партиции по месяцам. Это поможет нам быстрее создавать отчеты и чистить старые данные.

«Мастер» таблица будет «my_logs», структуру которой мы указали выше. Далее создадим «дочерние» таблицы (партиции):

Листинг 3.3 «Дочерние» таблицы

```
Line 1  CREATE TABLE my_logs2010m10 (
        CHECK ( logdate >= DATE '2010-10-01' AND logdate < DATE
        '2010-11-01' )
) INHERITS (my_logs);
CREATE TABLE my_logs2010m11 (
5       CHECK ( logdate >= DATE '2010-11-01' AND logdate < DATE
        '2010-12-01' )
) INHERITS (my_logs);
CREATE TABLE my_logs2010m12 (
        CHECK ( logdate >= DATE '2010-12-01' AND logdate < DATE
        '2011-01-01' )
) INHERITS (my_logs);
10 CREATE TABLE my_logs2011m01 (
```

```
-     CHECK ( logdate >= DATE '2011-01-01' AND logdate < DATE
      '2010-02-01' )
- ) INHERITS (my_logs);
```

Данными командами мы создаем таблицы «my_logs2010m10», «my_logs2010m11» и т.д., которые копируют структуру с «мастер» таблицы (кроме индексов). Также с помощью «CHECK» мы задаем диапазон значений, который будет попадать в эту партицию (хочу опять напомнить, что диапазоны значений партиций не должны пересекаться!). Поскольку партиционирование будет работать по полю «logdate», мы создадим индекс на это поле на всех партициях:

Листинг 3.4 Создание индексов

```
Line 1 CREATE INDEX my_logs2010m10_logdate ON my_logs2010m10 (
       logdate);
-  CREATE INDEX my_logs2010m11_logdate ON my_logs2010m11 (
       logdate);
-  CREATE INDEX my_logs2010m12_logdate ON my_logs2010m12 (
       logdate);
-  CREATE INDEX my_logs2011m01_logdate ON my_logs2011m01 (
       logdate);
```

Далее для удобства создадим функцию, которая будет перенаправлять новые данные с «мастер» таблицы в соответствующую партицию.

Листинг 3.5 Функция для перенаправления

```
Line 1  CREATE OR REPLACE FUNCTION my_logs_insert_trigger()
-   RETURNS TRIGGER AS $$
-   BEGIN
-       IF ( NEW.logdate >= DATE '2010-10-01' AND
5           NEW.logdate < DATE '2010-11-01' ) THEN
-           INSERT INTO my_logs2010m10 VALUES (NEW.*);
-       ELSIF ( NEW.logdate >= DATE '2010-11-01' AND
-               NEW.logdate < DATE '2010-12-01' ) THEN
-           INSERT INTO my_logs2010m11 VALUES (NEW.*);
10      ELSIF ( NEW.logdate >= DATE '2010-12-01' AND
-               NEW.logdate < DATE '2011-01-01' ) THEN
-           INSERT INTO my_logs2010m12 VALUES (NEW.*);
-       ELSIF ( NEW.logdate >= DATE '2011-01-01' AND
-               NEW.logdate < DATE '2011-02-01' ) THEN
15          INSERT INTO my_logs2011m01 VALUES (NEW.*);
-       ELSE
-           RAISE EXCEPTION 'Date out of range.  Fix the
        my_logs_insert_trigger() function!';
-       END IF;
-       RETURN NULL;
20  END;
-   $$
```

```
- LANGUAGE plpgsql;
```

В функции ничего особенного нет: идет проверка поля «logdate», по которой направляются данные в нужную партицию. При не нахождении требуемой партиции — вызываем ошибку. Теперь осталось создать триггер на «мастер» таблицу для автоматического вызова данной функции:

Листинг 3.6 Триггер

```
Line 1 CREATE TRIGGER insert_my_logs_trigger
-       BEFORE INSERT ON my_logs
-       FOR EACH ROW EXECUTE PROCEDURE my_logs_insert_trigger();
```

Партиционирование настроено и теперь мы готовы приступить к тестированию.

Тестирование

Для начала добавим данные в нашу таблицу «my_logs»:

Листинг 3.7 Данные

```
Line 1 INSERT INTO my_logs (user_id,logdate, data, some_state)
     VALUES(1, '2010-10-30', '30.10.2010 data', 1);
- INSERT INTO my_logs (user_id,logdate, data, some_state)
     VALUES(2, '2010-11-10', '10.11.2010 data2', 1);
- INSERT INTO my_logs (user_id,logdate, data, some_state)
     VALUES(1, '2010-12-15', '15.12.2010 data3', 1);
```

Теперь проверим где они хранятся:

Листинг 3.8 «Мастер» таблица чиста

```
Line 1 partitioning_test=# SELECT * FROM ONLY my_logs;
-  id | user_id | logdate | data | some_state
- ----+---------+---------+------+------------
- (0 rows)
```

Как видим в «мастер» таблицу данные не попали — она чиста. Теперь проверим а есть ли вообще данные:

Листинг 3.9 Проверка данных

```
Line 1 partitioning_test=# SELECT * FROM my_logs;
-  id | user_id |       logdate       |       data       |
     some_state
- --
     --+---------+---------------------+------------------+------------

-    1 |       1 | 2010-10-30 00:00:00 | 30.10.2010 data  |
           1
5    2 |       2 | 2010-11-10 00:00:00 | 10.11.2010 data2 |
           1
```

```
-   3 |       1 | 2010-12-15 00:00:00 | 15.12.2010 data3 |
          1
-  (3 rows)
```

Данные при этом выводятся без проблем. Проверим партиции, правильно ли хранятся данные:

Листинг 3.10 Проверка хранения данных

```
Line 1  partitioning_test=# Select * from my_logs2010m10;
-   id | user_id |       logdate       |       data       |
       some_state
-  --
       --+---------+---------------------+------------------+-----------

-   1 |       1 | 2010-10-30 00:00:00 | 30.10.2010 data |
          1
5  (1 row)
-
-  partitioning_test=# Select * from my_logs2010m11;
-   id | user_id |       logdate       |       data       |
       some_state
-  --
       --+---------+---------------------+------------------+-----------

10   2 |       2 | 2010-11-10 00:00:00 | 10.11.2010 data2 |
          1
-  (1 row)
```

Отлично! Данные хранятся на требуемых нам партициях. При этом запросы к таблице «my_logs» менять не нужно:

Листинг 3.11 Проверка запросов

```
Line 1  partitioning_test=# SELECT * FROM my_logs WHERE user_id = 2;
-   id | user_id |       logdate       |       data       |
       some_state
-  --
       --+---------+---------------------+------------------+-----------

-   2 |       2 | 2010-11-10 00:00:00 | 10.11.2010 data2 |
          1
5  (1 row)
-
-  partitioning_test=# SELECT * FROM my_logs WHERE data LIKE '
       %0.1%';
-   id | user_id |       logdate       |       data       |
       some_state
-  --
       --+---------+---------------------+------------------+-----------
```

```
10    1 |        1 | 2010-10-30 00:00:00 | 30.10.2010 data  |
          1
-     2 |        2 | 2010-11-10 00:00:00 | 10.11.2010 data2 |
          1
-  (2 rows)
```

Управление партициями

Обычно при работе с партиционированием старые партиции перестают получать данные и остаются неизменными. Это дает огромное преимущество над работой с данными через партиции. Например, нам нужно удалить старые логи за 2008 год, 10 месяц. Нам достаточно выполнить:

Листинг 3.12 Чистка логов

```
Line 1  DROP TABLE my_logs2008m10;
```

поскольку DROP TABLE работает гораздо быстрее, чем удаление миллионов записей индивидуально через DELETE. Другой вариант, который более предпочтителен, просто удалить партицию из партиционирования, тем самым оставив данные в СУБД, но уже не доступные через «мастер» таблицу:

Листинг 3.13 Удаляем партицию из партиционирования

```
Line 1  ALTER TABLE my_logs2008m10 NO INHERIT my_logs;
```

Это удобно, если мы хотим эти данные потом перенести в другое хранилище или просто сохранить.

Важность «constraint_exclusion» для партиционирования

Параметр constraint_exclusion отвечает за оптимизацию запросов, что повышает производительность для партиционированых таблиц. Например, выполним простой запрос:

Листинг 3.14 «constraint_exclusion» OFF

```
Line 1  partitioning_test=# SET constraint_exclusion = off;
-   partitioning_test=# EXPLAIN SELECT * FROM my_logs WHERE
        logdate > '2010-12-01';
-
-                                   QUERY PLAN
5   --
        ------------------------------------------------------------------------

-    Result  (cost=6.81..104.66 rows=1650 width=52)
-      ->  Append  (cost=6.81..104.66 rows=1650 width=52)
-            ->  Bitmap Heap Scan on my_logs  (cost=6.81..20.93
        rows=330 width=52)
```

```
-                 Recheck Cond: (logdate > '2010-12-01 00:00:00
    '::timestamp without time zone)
10                ->  Bitmap Index Scan on my_logs_logdate  (
    cost=0.00..6.73 rows=330 width=0)
-                       Index Cond: (logdate > '2010-12-01
    00:00:00'::timestamp without time zone)
-           ->  Bitmap Heap Scan on my_logs2010m10 my_logs  (
    cost=6.81..20.93 rows=330 width=52)
-                 Recheck Cond: (logdate > '2010-12-01 00:00:00
    '::timestamp without time zone)
-                 ->  Bitmap Index Scan on
    my_logs2010m10_logdate  (cost=0.00..6.73 rows=330 width
    =0)
15                      Index Cond: (logdate > '2010-12-01
    00:00:00'::timestamp without time zone)
-           ->  Bitmap Heap Scan on my_logs2010m11 my_logs  (
    cost=6.81..20.93 rows=330 width=52)
-                 Recheck Cond: (logdate > '2010-12-01 00:00:00
    '::timestamp without time zone)
-                 ->  Bitmap Index Scan on
    my_logs2010m11_logdate  (cost=0.00..6.73 rows=330 width
    =0)
-                       Index Cond: (logdate > '2010-12-01
    00:00:00'::timestamp without time zone)
20          ->  Bitmap Heap Scan on my_logs2010m12 my_logs  (
    cost=6.81..20.93 rows=330 width=52)
-                 Recheck Cond: (logdate > '2010-12-01 00:00:00
    '::timestamp without time zone)
-                 ->  Bitmap Index Scan on
    my_logs2010m12_logdate  (cost=0.00..6.73 rows=330 width
    =0)
-                       Index Cond: (logdate > '2010-12-01
    00:00:00'::timestamp without time zone)
-           ->  Bitmap Heap Scan on my_logs2011m01 my_logs  (
    cost=6.81..20.93 rows=330 width=52)
25                Recheck Cond: (logdate > '2010-12-01 00:00:00
    '::timestamp without time zone)
-                 ->  Bitmap Index Scan on
    my_logs2011m01_logdate  (cost=0.00..6.73 rows=330 width
    =0)
-                       Index Cond: (logdate > '2010-12-01
    00:00:00'::timestamp without time zone)
-   (22 rows)
```

Как видно через команду EXPLAIN, данный запрос сканирует все партиции на наличие данных в них, что не логично, поскольку данное условие «logdate > 2010-12-01» говорит о том, что данные должны браться только с партиций, где подходит такое условие. А теперь включим

constraint_exclusion:

Листинг 3.15 «constraint_exclusion» ON

```
Line 1  partitioning_test=# SET constraint_exclusion = on;
  -     SET
  -     partitioning_test=# EXPLAIN SELECT * FROM my_logs WHERE
           logdate > '2010-12-01';
  -                                                 QUERY PLAN
  5     --
           ------------------------------------------------------------------------------------------
  -      Result  (cost=6.81..41.87 rows=660 width=52)
  -        ->  Append  (cost=6.81..41.87 rows=660 width=52)
  -              ->  Bitmap Heap Scan on my_logs  (cost=6.81..20.93
           rows=330 width=52)
  -                    Recheck Cond: (logdate > '2010-12-01 00:00:00
           '::timestamp without time zone)
 10                   ->  Bitmap Index Scan on my_logs_logdate  (
           cost=0.00..6.73 rows=330 width=0)
  -                          Index Cond: (logdate > '2010-12-01
           00:00:00'::timestamp without time zone)
  -              ->  Bitmap Heap Scan on my_logs2010m12 my_logs  (
           cost=6.81..20.93 rows=330 width=52)
  -                    Recheck Cond: (logdate > '2010-12-01 00:00:00
           '::timestamp without time zone)
  -                    ->  Bitmap Index Scan on
           my_logs2010m12_logdate  (cost=0.00..6.73 rows=330 width
           =0)
 15                         Index Cond: (logdate > '2010-12-01
           00:00:00'::timestamp without time zone)
  -     (10 rows)
```

Как мы видим, теперь запрос работает правильно, и сканирует только партиции, что подходят под условие запроса. Но включать «constraint_exclusion» не желательно для баз, где нет партиционирования, поскольку команда CHECK будет проверятся на всех запросах, даже простых, а значит производительность сильно упадет. Начиная с 8.4 версии PostgreSQL constraint_exclusion может быть «on», «off» и «partition». По умолчанию (и рекомендуется) ставить constraint_exclusion не «on», и не «off», а «partition», который будет проверять «CHECK» только на партиционированых таблицах.

3.4 Заключение

Партиционирование — одна из самых простых и менее безболезненных методов уменьшения нагрузки на СУБД. Именно на этот вариант стоит

посмотреть сперва, и если он не подходит по каким либо причинам — переходить к более сложным. Но если в системе есть таблица, у которой актуальны только новые данные, но огромное количество старых (не актуальных) данных дает 50% или более нагрузки на СУБД — Вам стоит внедрить партиционирование.

4

Репликация

> Когда решаете проблему, ни о чем не беспокойтесь. Вот когда вы её решите, тогда и наступит время беспокоиться.
>
> Ричард Филлипс Фейман

4.1 Введение

Репликация (англ. replication) — механизм синхронизации содержимого нескольких копий объекта (например, содержимого базы данных). Репликация — это процесс, под которым понимается копирование данных из одного источника на множество других и наоборот. При репликации изменения, сделанные в одной копии объекта, могут быть распространены в другие копии. Репликация может быть синхронной или асинхронной.

В случае синхронной репликации, если данная реплика обновляется, все другие реплики того же фрагмента данных также должны быть обновлены в одной и той же транзакции. Логически это означает, что существует лишь одна версия данных. В большинстве продуктов синхронная репликация реализуется с помощью триггерных процедур (возможно, скрытых и управляемых системой). Но синхронная репликация имеет тот недостаток, что она создаёт дополнительную нагрузку при выполнении всех транзакций, в которых обновляются какие-либо реплики (кроме того, могут возникать проблемы, связанные с доступностью данных).

В случае асинхронной репликации обновление одной реплики распространяется на другие спустя некоторое время, а не в той же транзакции. Таким образом, при асинхронной репликации вводится задержка, или время ожидания, в течение которого отдельные реплики могут быть фактически неидентичными (то есть определение реплика оказывается не совсем подходящим, поскольку мы не имеем дело с точными и своевременно созданными копиями). В большинстве продуктов асинхронная репликация реализуется посредством чтения журнала транзакций или постоянной очереди тех обновлений, которые подлежат распространению. Пре-

имущество асинхронной репликации состоит в том, что дополнительные издержки репликации не связаны с транзакциями обновлений, которые могут иметь важное значение для функционирования всего предприятия и предъявлять высокие требования к производительности. К недостаткам этой схемы относится то, что данные могут оказаться несовместимыми (то есть несовместимыми с точки зрения пользователя). Иными словами, избыточность может проявляться на логическом уровне, а это, строго говоря, означает, что термин контролируемая избыточность в таком случае не применим.

Рассмотрим кратко проблему согласованности (или, скорее, несогласованности). Дело в том, что реплики могут становиться несовместимыми в результате ситуаций, которые трудно (или даже невозможно) избежать и последствия которых трудно исправить. В частности, конфликты могут возникать по поводу того, в каком порядке должны применяться обновления. Например, предположим, что в результате выполнения транзакции A происходит вставка строки в реплику X, после чего транзакция B удаляет эту строку, а также допустим, что Y — реплика X. Если обновления распространяются на Y, но вводятся в реплику Y в обратном порядке (например, из-за разных задержек при передаче), то транзакция B не находит в Y строку, подлежащую удалению, и не выполняет своё действие, после чего транзакция A вставляет эту строку. Суммарный эффект состоит в том, что реплика Y содержит указанную строку, а реплика X — нет.

В целом задачи устранения конфликтных ситуаций и обеспечения согласованности реплик являются весьма сложными. Следует отметить, что, по крайней мере, в сообществе пользователей коммерческих баз данных термин репликация стал означать преимущественно (или даже исключительно) асинхронную репликацию.

Основное различие между репликацией и управлением копированием заключается в следующем: если используется репликация, то обновление одной реплики в конечном счёте распространяется на все остальные автоматически. В режиме управления копированием, напротив, не существует такого автоматического распространения обновлений. Копии данных создаются и управляются с помощью пакетного или фонового процесса, который отделён во времени от транзакций обновления. Управление копированием в общем более эффективно по сравнению с репликацией, поскольку за один раз могут копироваться большие объёмы данных. К недостаткам можно отнести то, что большую часть времени копии данных не идентичны базовым данным, поэтому пользователи должны учитывать, когда именно были синхронизированы эти данные. Обычно управление копированием упрощается благодаря тому требованию, чтобы обновления применялись в соответствии со схемой первичной копии того или иного вида.

Для репликации PostgreSQL существует несколько решений, как закрытых, так и свободных. Закрытые системы репликации не будут рас-

сматриваться в этой книге. Вот список свободных решений:

- Slony-I — асинхронная Master-Slave репликация, поддерживает каскады(cascading) и отказоустойчивость(failover). Slony-I использует триггеры PostgreSQL для привязки к событиям INSERT/DELETE/UPDATE и хранимые процедуры для выполнения действий;
- PGCluster — синхронная Multi-Master репликация. Проект на мой взгляд мертв;
- Pgpool-I/II — это замечательный инструмент для PostgreSQL (лучше сразу работать с II версией). Позволяет делать:
  - репликацию (в том числе, с автоматическим переключением на резервный stand-by сервер);
  - online-бэкап;
  - pooling коннектов;
  - очередь соединений;
  - балансировку SELECT-запросов на несколько postgresql-серверов;
  - разбиение запросов для параллельного выполнения над большими объемами данных;
- Bucardo — асинхронная репликация, которая поддерживает Multi-Master и Master-Slave режимы, а также несколько видов синхронизации и обработки конфликтов;
- Londiste — асинхронная Master-Slave репликация. Входит в состав Skytools[1]. Проще в использовании, чем Slony-I;
- Mammoth Replicator — асинхронная Multi-Master репликация;
- Postgres-R — асинхронная Multi-Master репликация. Проект на мой взгляд мертв;
- RubyRep — написанная на Ruby, асинхронная Multi-Master репликация, которая поддерживает PostgreSQL и MySQL.

Это, конечно, не весь список свободных систем для репликации, но я думаю даже из этого есть что выбрать для PostgreSQL.

4.2 Потоковая репликация (Streaming Replication)

Введение

Потоковая репликация (Streaming Replication, SR) дает возможность непрерывно отправлять и применять wall xlog записи на резервные сервера для создания точной копии текущего. Данная функциональность появилась у PostgreSQL начиная с 9 версии (репликация из коробки!). Этот тип репликации простой, надежный и, вероятней всего, будет использоваться

[1] http://pgfoundry.org/projects/skytools/

в качестве стандартной репликации в большинстве высоконагруженных приложений, что используют PostgreSQL.

Отличительными особенностями решения являются:

- репликация всего инстанса PostgreSQL;
- асинхронный механизм репликации;
- простота установки;
- мастер база данных может обслуживать огромное количество слейвов из-за минимальной нагрузки.

К недостаткам можно отнести:

- невозможность реплицировать только определенную базу данных из всех на PostgreSQL инстансе.

Установка

Для начала нам потребуется PostgreSQL не ниже 9 версии. В момент написания этой главы была доступна 9.3 версия. Все работы, как полагается, будут проводится на Linux.

Настройка

Для начала обозначим мастер сервер как masterdb(192.168.0.10) и слейв как slavedb(192.168.0.20).

Предварительная настройка

Для начала позволим определенному пользователю без пароля ходить по ssh. Пусть это будет postgres юзер. Если же нет, то создаем набором команд:

Листинг 4.1 Создаем пользователя userssh

```
Line 1  $ sudo groupadd userssh
    -   $ sudo useradd -m -g userssh -d /home/userssh -s /bin/bash \
    -   -c "user ssh allow" userssh
```

Дальше выполняем команды от имени пользователя (в данном случае postgres):

Листинг 4.2 Логинимся под пользователем postgres

```
Line 1  $ su postgres
```

Генерим RSA-ключ для обеспечения аутентификации в условиях отсутствия возможности использовать пароль:

Листинг 4.3 Генерим RSA-ключ

```
$ ssh-keygen -t rsa -P ""
Generating public/private rsa key pair.
Enter file in which to save the key (/var/lib/postgresql/.
    ssh/id_rsa):
Created directory '/var/lib/postgresql/.ssh'.
Your identification has been saved in /var/lib/postgresql/.
    ssh/id_rsa.
Your public key has been saved in /var/lib/postgresql/.ssh/
    id_rsa.pub.
The key fingerprint is:
16:08:27:97:21:39:b5:7b:86:e1:46:97:bf:12:3d:76
    postgres@localhost
```

И добавляем его в список авторизованных ключей:

Листинг 4.4 Добавляем его в список авторизованных ключей

```
$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

Этого должно быть более чем достаточно. Проверить работоспособность соединения можно просто написав:

Листинг 4.5 Пробуем зайти на ssh без пароля

```
$ ssh localhost
```

Не забываем предварительно инициализировать sshd:

Листинг 4.6 Запуск sshd

```
$ $/etc/init.d/sshd start
```

После успешно проделаной операции скопируйте «$HOME/.ssh» на slavedb. Теперь мы должны иметь возможность без пароля заходить с мастера на слейв и со слейва на мастер через ssh.

Также отредактируем pg_hba.conf на мастере и слейве, разрешив им друг к другу доступ без пароля(trust) (тут добавляется роль replication):

Листинг 4.7 Мастер pg_hba.conf

```
host    replication    all    192.168.0.20/32    trust
```

Листинг 4.8 Слейв pg_hba.conf

```
host    replication    all    192.168.0.10/32    trust
```

Не забываем после этого перегрузить postgresql на обоих серверах.

Настройка мастера

Для начала настроим masterdb. Установим параметры в postgresql.conf для репликации:

Листинг 4.9 Настройка мастера

```
Line 1  # To enable read-only queries on a standby server, wal_level
          must be set to
        # "hot_standby". But you can choose "archive" if you never
          connect to the
        # server in standby mode.
        wal_level = hot_standby
      5
        # Set the maximum number of concurrent connections from the
          standby servers.
        max_wal_senders = 5
        #
        # To prevent the primary server from removing the WAL
          segments required for
     10 # the standby server before shipping them, set the minimum
          number of segments
        # retained in the pg_xlog directory. At least
          wal_keep_segments should be
        # larger than the number of segments generated between the
          beginning of
        # online-backup and the startup of streaming replication. If
           you enable WAL
        # archiving to an archive directory accessible from the
          standby, this may
     15 # not be necessary.
        wal_keep_segments = 32
        #
        # Enable WAL archiving on the primary to an archive
          directory accessible from
        # the standby. If wal_keep_segments is a high enough number
          to retain the WAL
     20 # segments required for the standby server, this may not be
          necessary.
        archive_mode    = on
        archive_command = 'cp %p /path_to/archive/%f'
```

Давайте по порядку:

- wal_level = hot_standby — сервер начнет писать в WAL логи так же как и при режиме «archive», добавляя информацию, необходимую для восстановления транзакции (можно также поставить archive, но тогда сервер не может быть слейвом при необходимости);
- max_wal_senders = 5 — максимальное количество слейвов;
- wal_keep_segments = 32 — минимальное количество файлов с WAL сегментами в pg_xlog директории;

- archive_mode = on — позволяем сохранять WAL сегменты в указанное переменной archive_command хранилище. В данном случае в директорию «/path/to/archive/».

По умолчанию репликация асинхронная. В версии 9.1 добавили параметр synchronous_standby_names, который включает синхронную репликацию. В данные параметр передается application_name, который используется на слейвах в recovery.conf:

Листинг 4.10 recovery.conf для синхронной репликации на слейве

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
          # e.g. 'cp /mnt/server/archivedir/%f %p'
standby_mode = on
primary_conninfo = 'host=masterdb port=59121 user=
    replication password=replication application_name=
    newcluster'              # e.g. 'host=localhost port=5432'
trigger_file = '/tmp/trig_f_newcluster'
```

После изменения параметров перегружаем PostgreSQL сервер. Теперь перейдем к slavedb.

Настройка слейва

Для начала нам потребуется создать на slavedb точную копию masterdb. Перенесем данные с помощью «Онлайн бекапа».

Для начала зайдем на masterdb сервер. Выполним в консоли:

Листинг 4.11 Выполняем на мастере

```
$ psql -c "SELECT pg_start_backup('label', true)"
```

Теперь нам нужно перенести данные с мастера на слейв. Выполняем на мастере:

Листинг 4.12 Выполняем на мастере

```
$ rsync -C -a --delete -e ssh --exclude postgresql.conf --
    exclude postmaster.pid \
--exclude postmaster.opts --exclude pg_log --exclude pg_xlog
    \
--exclude recovery.conf master_db_datadir/ slavedb_host:
    slave_db_datadir/
```

где

- master_db_datadir — директория с postgresql данными на masterdb
- slave_db_datadir — директория с postgresql данными на slavedb
- slavedb_host — хост slavedb(в нашем случае - 192.168.1.20)

После копирования данных с мастера на слейв, остановим онлайн бекап. Выполняем на мастере:

Листинг 4.13 Выполняем на мастере

```
Line 1 $ psql -c "SELECT pg_stop_backup()"
```

Для версии PostgreSQL 9.1+ можно воспользоватся командой pg_basebackup (копирует базу на slavedb подобным образом):

Листинг 4.14 Выполняем на слейве

```
Line 1 $ pg_basebackup -R -D /srv/pgsql/standby --host=192.168.0.10
    --port=5432
```

Устанавливаем такие же данные в конфиге postgresql.conf, что и у мастера (чтобы при падении мастера слейв мог его заменить). Так же установим дополнительный параметр:

Листинг 4.15 Конфиг слейва

```
Line 1 hot_standby = on
```

Внимание! Если на мастере поставили wal_level = archive, тогда параметр оставляем по умолчанию (hot_standby = off).

Далее на slavedb в директории с данными PostgreSQL создадим файл recovery.conf с таким содержимым:

Листинг 4.16 Конфиг recovery.conf

```
Line 1 # Specifies whether to start the server as a standby. In
    streaming replication,
  - # this parameter must to be set to on.
  - standby_mode          = 'on'
  -
  5 # Specifies a connection string which is used for the
    standby server to connect
  - # with the primary.
  - primary_conninfo      = 'host=192.168.0.10 port=5432 user=
    postgres'
  -
  - # Specifies a trigger file whose presence should cause
    streaming replication to
 10 # end (i.e., failover).
  - trigger_file = '/path_to/trigger'
  -
  - # Specifies a command to load archive segments from the WAL
    archive. If
  - # wal_keep_segments is a high enough number to retain the
    WAL segments
 15 # required for the standby server, this may not be necessary
    . But
  - # a large workload can cause segments to be recycled before
    the standby
  - # is fully synchronized, requiring you to start again from a
     new base backup.
```

```
- restore_command = 'scp masterdb_host:/path_to/archive/%f "%p
    "'
```

где

- standby_mode='on' — указываем серверу работать в режиме слейв;
- primary_conninfo — настройки соединения слейва с мастером;
- trigger_file — указываем триггер-файл, при наличии которого будет остановлена репликация;
- restore_command — команда, которой будут восстанавливаться WAL логи. В нашем случае через scp копируем с masterdb (masterdb_host - хост masterdb).

Теперь мы можем запустить PostgreSQL на slavedb.

Тестирование репликации

Теперь мы можем посмотреть отставание слейвов от мастера с помощью таких команд:

Листинг 4.17 Тестирование репликации

```
Line 1  $ psql -c "SELECT pg_current_xlog_location()" -h192.168.0.10
          (masterdb)
-  pg_current_xlog_location
-  --------------------------
-   0/2000000
5  (1 row)
-
-  $ psql -c "select pg_last_xlog_receive_location()" -h192
     .168.0.20 (slavedb)
-   pg_last_xlog_receive_location
-  -------------------------------
10   0/2000000
-  (1 row)
-
-  $ psql -c "select pg_last_xlog_replay_location()" -h192
     .168.0.20 (slavedb)
-   pg_last_xlog_replay_location
15 -------------------------------
-   0/2000000
-  (1 row)
```

Начиная с версии 9.1 добавили дополнительные view для просмотра состояния репликации. Теперь master знает все состояния slaves:

Листинг 4.18 Состояние слейвов

```
Line 1  # SELECT * from pg_stat_replication ;
```

```
-  procpid | usesysid |   usename   | application_name |
   client_addr | client_hostname | client_port |
   backend_start         |   state   | sent_location |
   write_location | flush_location | replay_location |
   sync_priority | sync_state
-  --
   -------+----------+-------------+------------------+-------------+------

-    17135 |    16671 | replication | newcluster       |
   127.0.0.1    |                 |       43745 | 2011-05-22
   18:13:04.19283+02 | streaming | 1/30008750    |
   1/30008750     | 1/30008750     | 1/30008750      |
          1 | sync
```

Также с версии 9.1 добавили view pg_stat_database_conflicts, с помощью которой на слейв базах можно просмотреть сколько запросов было отменено и по каким причинам:

Листинг 4.19 Состояние слейва

```
Line 1 # SELECT * from pg_stat_database_conflicts ;
-   datid |  datname  | confl_tablespace | confl_lock |
    confl_snapshot | confl_bufferpin | confl_deadlock
-   --
    -----+-----------+------------------+------------+----------------+-----

-       1 | template1 |                0 |          0 |
            0 |               0 |                0
5   11979 | template0 |                0 |          0 |
            0 |               0 |                0
-   11987 | postgres  |                0 |          0 |
            0 |               0 |                0
-   16384 | marc      |                0 |          0 |
            1 |               0 |                0
```

Еще проверить работу репликации можно с помощью утилиты ps:

Листинг 4.20 Тестирование репликации

```
Line 1 [masterdb] $ ps -ef | grep sender
- postgres  6879  6831  0 10:31 ?        00:00:00 postgres:
    wal sender process postgres 127.0.0.1(44663) streaming
    0/2000000
-
- [slavedb] $ ps -ef | grep receiver
5 postgres  6878  6872  1 10:31 ?        00:00:01 postgres:
    wal receiver process   streaming 0/2000000
```

Теперь проверим реприкацию. Выполним на мастере:

Листинг 4.21 Выполняем на мастере

```
$ psql test_db
test_db=# create table test3(id int not null primary key,
    name varchar(20));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit
    index "test3_pkey" for table "test3"
CREATE TABLE
test_db=# insert into test3(id, name) values('1', 'test1');
INSERT 0 1
test_db=#
```

Теперь проверим на слейве:

Листинг 4.22 Выполняем на слейве

```
$ psql test_db
test_db=# select * from test3;
 id | name
----+-------
  1 | test1
(1 row)
```

Как видим, таблица с данными успешно скопирована с мастера на слейв.

Общие задачи

Переключение на слейв при падении мастера

Достаточно создать триггер-файл (trigger_file) на слейве, который становится мастером.

Остановка репликации на слейве

Создать триггер-файл (trigger_file) на слейве. Также с версии 9.1 добавили команды pg_xlog_replay_pause() и pg_xlog_replay_resume() для остановки и возобновления репликации.

Перезапуск репликации после сбоя

Повторяем операции из раздела «Настройка слейва». Хочется заметить, что мастер при этом не нуждается в остановке при выполнении данной задачи.

Перезапуск репликации после сбоя слейва

Перезагрузить PostgreSQL на слейве после устранения сбоя.

Повторно синхронизировать репликации на слейве

Это может потребоваться, например, после длительного отключения от мастера. Для этого останавливаем PostgreSQL на слейве и повторяем операции из раздела «Настройка слейва».

4.3 PostgreSQL Bi-Directional Replication (BDR)

BDR (Bi-Directional Replication) это новая функциональность добавленая в ядро PostgreSQL которая предоставляет расширенные средства для репликации. На данный момент это реализовано в виде небольшого патча и модуля. Заявлено что полностью будет только в PostgreSQL 9.5. BDR позволяет создавать географически распределенные асинхронные мультимастер конфигурации используя для этого встроенную логическую потоковую репликацию LLSR (Logical Log Streaming Replication).

BDR не является инструментом для кластеризации, т.к. здесь нет каких-либо глобальных менеджеров блокировок или координаторов транзакций. Каждый узел не зависит от других, что было бы невозможно в случае использования менеджеров блокировки. Каждый из узлов содержит локальную копию данных идентичную данным на других узлах. Запросы также выполняются только локально. При этом каждый из узлов внутренне консистентен в любое время, целиком же группа серверов является согласованной в конечном счете (eventually consistent). Уникальность BDR заключается в том что она непохожа ни на встроенную потоковую репликацию, ни на существующие trigger-based решения (Londiste, Slony, Bucardo).

Самым заметным отличием от потоковой репликации является то, что BDR (LLSR) оперирует базами (per-database replication), а классическая PLSR реплицирует целиком инстанс (per-cluster replication), т.е. все базы внутри инстанса. Существующие ограничения и особенности:

- Все изменения данных вызываемые INSERT/DELETE/UPDATE реплицируются (TRUNCATE на момент написания статьи пока не реализован);
- Большинство операции изменения схемы (DDL) реплицируются успешно. Неподдерживаемые DDL фиксируются модулем репликации и отклоняются с выдачей ошибкой (на момент написания статьи не работал CREATE TABLE ... AS);
- Определения таблиц, типов, расширений и т.п. должны быть идентичными между upstream и downstream мастерами;
- Действия которые отражаются в WAL, но непредставляются в виде логических изменений не реплицируются на другой узел (запись полных страниц, вакуумация таблиц и т.п.). Таким образом логическая потоковая репликация (LLSR) избавлена от некоторой части накладных расходов которые присутствуют в физической потоковой

репликации PLSR (тем не менее это не означает что LLSR требуется меньшая пропускная способность сети чем для PLSR).

Небольшое примечание: временная остановка репликации осуществляется выключением downstream мастера. Однако стоит отметить что остановленная реплика приводит к тому что upstream мастер продолжит накапливать WAL журналы что в свою очередь может привести к неконтролируемому расходу пространства на диске. Поэтому крайне не рекомендуется надолго выключать реплику. Удаление реплики навсегда осуществляется через удаление конфигурации BDR на downstream сервере с последующим перезапуском downstream мастера. Затем нужно удалить соответствующий слот репликации на upstream мастере с помощью функции pg_drop_replication_slot('slotname'). Доступные слоты можно просмотреть с помощью функции pg_get_replication_slots().

На текущий момент собрать BDR можно из исходников по данному мануалу. С официальным принятием данных патчей в ядро PostgreSQL данный раздел про BDR будет расширен и дополнен.

4.4 Slony-I

Введение

Slony это система репликации реального времени, позволяющая организовать синхронизацию нескольких серверов PostgreSQL по сети. Slony использует триггеры Postgre для привязки к событиям INSERT/DELETE/UPDATE и хранимые процедуры для выполнения действий.

Система Slony с точки зрения администратора состоит из двух главных компонент: репликационного демона slony и административной консоли slonik. Администрирование системы сводится к общению со slonik-ом, демон slon только следит за собственно процессом репликации. А админ следит за тем, чтобы slon висел там, где ему положено.

О slonik-е

Все команды slonik принимает на свой stdin. До начала выполнения скрипт slonik-а проверяется на соответствие синтаксису, если обнаруживаются ошибки, скрипт не выполняется, так что можно не волноваться если slonik сообщает о syntax error, ничего страшного не произошло. И он ещё ничего не сделал. Скорее всего.

Установка

Установка на Ubuntu производится простой командой:

Листинг 4.23 Установка

```
Line 1 $ sudo aptitude install slony1-2-bin
```

Настройка

Рассмотрим теперь установку на гипотетическую базу данных customers (названия узлов, кластеров и таблиц являются вымышленными).

Исходные данные:

- customers — база данных;
- master_host — хост master базы;
- slave_host — хост slave базы;
- customers_rep — имя кластера.

Подготовка master базы

Для начала нам нужно создать пользователя в базе, под которым будет действовать Slony. По умолчанию, и отдавая должное системе, этого пользователя обычно называют slony.

Листинг 4.24 Подготовка master-сервера

```
Line 1 $ createuser -a -d slony
  - $ psql -d template1 -c "ALTER USER slony WITH PASSWORD '
    slony_user_password';"
```

Также на каждом из узлов лучше завести системного пользователя slony, чтобы запускать от его имени репликационного демона slon. В дальнейшем подразумевается, что он (и пользователь и slon) есть на каждом из узлов кластера.

Подготовка slave базы

Здесь я рассматриваю, что серверы кластера соединены посредством сети. Необходимо чтобы с каждого из серверов можно было установить соединение с PostgreSQL на master хосте, и наоборот. То есть, команда:

Листинг 4.25 Подготовка одного slave-сервера

```
Line 1 anyuser@customers_slave$ psql -d customers \
  - -h customers_master.com -U slony
```

должна подключать нас к мастер-серверу (после ввода пароля, желательно). Если что-то не так, возможно требуется поковыряться в настройках firewall-а, или файле pg_hba.conf, который лежит в $PGDATA.

Теперь устанавливаем на slave-хост сервер PostgreSQL. Следующего обычно не требуется, сразу после установки Postgres «up and ready», но в

случае каких-то ошибок можно начать «с чистого листа», выполнив следующие команды (предварительно сохранив конфигурационные файлы и остановив postmaster):

Листинг 4.26 Подготовка одного slave-сервера

```
pgsql@customers_slave$ rm -rf $PGDATA
pgsql@customers_slave$ mkdir $PGDATA
pgsql@customers_slave$ initdb -E UTF8 -D $PGDATA
pgsql@customers_slave$ createuser -a -d slony
pgsql@customers_slave$ psql -d template1 -c "alter \
user slony with password 'slony_user_password';"
```

Запускаем postmaster.

Внимание! Обычно требуется определённый владелец для реплицируемой БД. В этом случае необходимо создать его тоже!

Листинг 4.27 Подготовка одного slave-сервера

```
pgsql@customers_slave$ createuser -a -d customers_owner
pgsql@customers_slave$ psql -d template1 -c "alter \
user customers_owner with password 'customers_owner_password
    ';"
```

Эти две команды можно запускать с customers_master, к командной строке в этом случае нужно добавить -h customers_slave, чтобы все операции выполнялись на slave.

На slave, как и на master, также нужно установить Slony.

Инициализация БД и plpgsql на slave

Следующие команды выполняются от пользователя slony. Скорее всего для выполнения каждой из них потребуется ввести пароль (slony_user_password). Итак:

Листинг 4.28 Инициализация БД и plpgsql на slave

```
slony@customers_master$ createdb -O customers_owner \
-h customers_slave.com customers
slony@customers_master$ createlang -d customers \
-h customers_slave.com plpgsql
```

Внимание! Все таблицы, которые будут добавлены в replication set должны иметь primary key. Если какая-то из таблиц не удовлетворяет этому условию, задержитесь на этом шаге и дайте каждой таблице primary key командой ALTER TABLE ADD PRIMARY KEY.

Если столбца который мог бы стать primary key не находится, добавьте новый столбец типа serial (ALTER TABLE ADD COLUMN), и заполните его значениями. Настоятельно НЕ рекомендую использовать «table add key» slonik-а.

Продолжаем. Создаём таблицы и всё остальное на slave:

Листинг 4.29 Инициализация БД и plpgsql на slave

```
slony@customers_master$ pg_dump -s customers | \
psql -U slony -h customers_slave.com customers
```

pg_dump -s сдампит только структуру нашей БД.

pg_dump -s customers должен пускать без пароля, а вот для psql -U slony -h customers_slave.com customers придётся набрать пароль (slony_user_pass). Важно: я подразумеваю что сейчас на мастер-хосте ещё не установлен Slony (речь не про make install), то есть в БД нет таблиц sl_*, триггеров и прочего. Если есть, то возможно два варианта:

- добавляется узел в уже функционирующую систему репликации (читайте раздел 5);
- это ошибка :-) Тогда до переноса структуры на slave выполните следующее:

Листинг 4.30 Инициализация БД и plpgsql на slave

```
slonik <<EOF
cluster name = customers_slave;
node Y admin conninfo = 'dbname=customers host=
    customers_master.com
port=5432 user=slony password=slony_user_pass';
uninstall node (id = Y);
echo 'okay';
EOF
```

Y — число. Любое. Важно: если это действительно ошибка, cluster name может иметь какой-то другое значение, например T1 (default). Нужно его выяснить и сделать uninstall.

Если структура уже перенесена (и это действительно ошибка), сделайте uninstall с обоих узлов (с master и slave).

Инициализация кластера

Если Сейчас мы имеем два сервера PgSQL которые свободно «видят» друг друга по сети, на одном из них находится мастер-база с данными, на другом — только структура.

На мастер-хосте запускаем такой скрипт:

Листинг 4.31 Инициализация кластера

```
#!/bin/sh

CLUSTER=customers_rep

DBNAME1=customers
DBNAME2=customers
```

```
-
- HOST1=customers_master.com
- HOST2=customers_slave.com
10
- PORT1=5432
- PORT2=5432
-
- SLONY_USER=slony
15
- slonik <<EOF
- cluster name = $CLUSTER;
- node 1 admin conninfo = 'dbname=$DBNAME1 host=$HOST1 port=
    $PORT1
- user=slony password=slony_user_password';
20 node 2 admin conninfo = 'dbname=$DBNAME2 host=$HOST2
- port=$PORT2 user=slony password=slony_user_password';
- init cluster ( id = 1, comment = 'Customers DB
- replication cluster' );
-
25 echo 'Create set';
-
- create set ( id = 1, origin = 1, comment = 'Customers
- DB replication set' );
-
30 echo 'Adding tables to the subscription set';
-
- echo ' Adding table public.customers_sales...';
- set add table ( set id = 1, origin = 1, id = 4, full
    qualified
- name = 'public.customers_sales', comment = 'Table public.
    customers_sales' );
35 echo ' done';
-
- echo ' Adding table public.customers_something...';
- set add table ( set id = 1, origin = 1, id = 5, full
    qualified
- name = 'public.customers_something,
40 comment = 'Table public.customers_something );
- echo ' done';
-
- echo 'done adding';
- store node ( id = 2, comment = 'Node 2, $HOST2' );
45 echo 'stored node';
- store path ( server = 1, client = 2, conninfo = 'dbname=
    $DBNAME1 host=$HOST1
- port=$PORT1 user=slony password=slony_user_password' );
- echo 'stored path';
```

```
store path ( server = 2, client = 1, conninfo = 'dbname=
    $DBNAME2 host=$HOST2
port=$PORT2 user=slony password=slony_user_password' );

store listen ( origin = 1, provider = 1, receiver = 2 );
store listen ( origin = 2, provider = 2, receiver = 1 );
EOF
```

Здесь мы инициализируем кластер, создаём репликационный набор, включаем в него две таблицы. Важно: нужно перечислить все таблицы, которые нужно реплицировать, id таблицы в наборе должен быть уникальным, таблицы должны иметь primary key.

Важно: replication set запоминается раз и навсегда. Чтобы добавить узел в схему репликации не нужно заново инициализировать set.

Важно: если в набор добавляется или удаляется таблица нужно переподписать все узлы. То есть сделать unsubscribe и subscribe заново.

Подписываем slave-узел на replication set

Скрипт:

Листинг 4.32 Подписываем slave-узел на replication set

```
Line 1  #!/bin/sh

CLUSTER=customers_rep

DBNAME1=customers
DBNAME2=customers

HOST1=customers_master.com
HOST2=customers_slave.com

PORT1=5432
PORT2=5432

SLONY_USER=slony

slonik <<EOF
cluster name = $CLUSTER;
node 1 admin conninfo = 'dbname=$DBNAME1 host=$HOST1
port=$PORT1 user=slony password=slony_user_password';
node 2 admin conninfo = 'dbname=$DBNAME2 host=$HOST2
port=$PORT2 user=slony password=slony_user_password';

echo 'subscribing';
subscribe set ( id = 1, provider = 1, receiver = 2, forward
    = no);
```

```
25
- EOF
```

Старт репликации

Теперь, на обоих узлах необходимо запустить демона репликации.

Листинг 4.33 Старт репликации

```
Line 1 slony@customers_master$ slon customers_rep \
- "dbname=customers user=slony"
```

и

Листинг 4.34 Старт репликации

```
Line 1 slony@customers_slave$ slon customers_rep \
- "dbname=customers user=slony"
```

Сейчас слоны обменяются сообщениями и начнут передачу данных. Начальное наполнение происходит с помощью COPY, slave DB на это время полностью блокируется.

В среднем время актуализации данных на slave-системе составляет до 10-ти секунд. slon успешно обходит проблемы со связью и подключением к БД, и вообще требует к себе достаточно мало внимания.

Общие задачи

Добавление ещё одного узла в работающую схему репликации

Выполнить 4.4 и выполнить 4.4.

Новый узел имеет id = 3. Находится на хосте customers_slave3.com, «видит» мастер-сервер по сети и мастер может подключиться к его PgSQL. После дублирования структуры (п 4.4.2) делаем следующее:

Листинг 4.35 Общие задачи

```
Line 1 slonik <<EOF
- cluster name = customers_slave;
- node 3 admin conninfo = 'dbname=customers host=
    customers_slave3.com
- port=5432 user=slony password=slony_user_pass';
5 uninstall node (id = 3);
- echo 'okay';
- EOF
```

Это нужно чтобы удалить схему, триггеры и процедуры, которые были сдублированы вместе с таблицами и структурой БД.

Инициализировать кластер не надо. Вместо этого записываем информацию о новом узле в сети:

Листинг 4.36 Общие задачи

```
#!/bin/sh

CLUSTER=customers_rep

DBNAME1=customers
DBNAME3=customers

HOST1=customers_master.com
HOST3=customers_slave3.com

PORT1=5432
PORT2=5432

SLONY_USER=slony

slonik <<EOF
cluster name = $CLUSTER;
node 1 admin conninfo = 'dbname=$DBNAME1 host=$HOST1
port=$PORT1 user=slony password=slony_user_pass';
node 3 admin conninfo = 'dbname=$DBNAME3
host=$HOST3 port=$PORT2 user=slony password=slony_user_pass'
    ;

echo 'done adding';

store node ( id = 3, comment = 'Node 3, $HOST3' );
echo 'sored node';
store path ( server = 1, client = 3, conninfo = 'dbname=
   $DBNAME1
host=$HOST1 port=$PORT1 user=slony password=slony_user_pass'
     );
echo 'stored path';
store path ( server = 3, client = 1, conninfo = 'dbname=
   $DBNAME3
host=$HOST3 port=$PORT2 user=slony password=slony_user_pass'
     );

echo 'again';
store listen ( origin = 1, provider = 1, receiver = 3 );
store listen ( origin = 3, provider = 3, receiver = 1 );

EOF
```

Новый узел имеет id 3, потому что 2 уже есть и работает. Подписываем новый узел 3 на replication set:

Листинг 4.37 Общие задачи

```
#!/bin/sh

CLUSTER=customers_rep

DBNAME1=customers
DBNAME3=customers

HOST1=customers_master.com
HOST3=customers_slave3.com

PORT1=5432
PORT2=5432

SLONY_USER=slony

slonik <<EOF
cluster name = $CLUSTER;
node 1 admin conninfo = 'dbname=$DBNAME1 host=$HOST1
port=$PORT1 user=slony password=slony_user_pass';
node 3 admin conninfo = 'dbname=$DBNAME3 host=$HOST3
port=$PORT2 user=slony password=slony_user_pass';

echo'subscribing';
subscribe set ( id = 1, provider = 1, receiver = 3, forward
   = no);

EOF
```

Теперь запускаем slon на новом узле, так же как и на остальных. Перезапускать slon на мастере не надо.

Листинг 4.38 Общие задачи

```
slony@customers_slave3$ slon customers_rep \
"dbname=customers user=slony"
```

Репликация должна начаться как обычно.

Устранение неисправностей

Ошибка при добавлении узла в систему репликации

Периодически, при добавлении новой машины в кластер возникает следующая ошибка: на новой ноде всё начинает жужжать и работать, имеющиеся же отваливаются с примерно следующей диагностикой:

Листинг 4.39 Устранение неисправностей

```
%slon customers_rep "dbname=customers user=slony_user"
CONFIG main: slon version 1.0.5 starting up
```

```
- CONFIG main: local node id = 3
- CONFIG main: loading current cluster configuration
5 CONFIG storeNode: no_id=1 no_comment='CustomersDB
- replication cluster'
- CONFIG storeNode: no_id=2 no_comment='Node 2,
- node2.example.com'
- CONFIG storeNode: no_id=4 no_comment='Node 4,
10 node4.example.com'
- CONFIG storePath: pa_server=1 pa_client=3
- pa_conninfo="dbname=customers
- host=mainhost.com port=5432 user=slony_user
- password=slony_user_pass" pa_connretry=10
15 CONFIG storeListen: li_origin=1 li_receiver=3
- li_provider=1
- CONFIG storeSet: set_id=1 set_origin=1
- set_comment='CustomersDB replication set'
- WARN remoteWorker_wakeup: node 1 - no worker thread
20 CONFIG storeSubscribe: sub_set=1 sub_provider=1 sub_forward=
    'f'
- WARN remoteWorker_wakeup: node 1 - no worker thread
- CONFIG enableSubscription: sub_set=1
- WARN remoteWorker_wakeup: node 1 - no worker thread
- CONFIG main: configuration complete - starting threads
25 CONFIG enableNode: no_id=1
- CONFIG enableNode: no_id=2
- CONFIG enableNode: no_id=4
- ERROR remoteWorkerThread_1: "begin transaction; set
- transaction isolation level
30 serializable; lock table "_customers_rep".sl_config_lock;
    select
- "_customers_rep".enableSubscription(1, 1, 4);
- notify "_customers_rep_Event"; notify "
    _customers_rep_Confirm";
- insert into "_customers_rep".sl_event (ev_origin, ev_seqno,
- ev_timestamp, ev_minxid, ev_maxxid, ev_xip,
35 ev_type , ev_data1, ev_data2, ev_data3, ev_data4 ) values
- ('1', '219440',
- '2005-05-05 18:52:42.708351', '52501283', '52501292',
- '''52501283''', 'ENABLE_SUBSCRIPTION',
- '1', '1', '4', 'f'); insert into "_customers_rep".
40 sl_confirm (con_origin, con_received,
- con_seqno, con_timestamp) values (1, 3, '219440',
- CURRENT_TIMESTAMP); commit transaction;"
- PGRES_FATAL_ERROR ERROR: insert or update on table
- "sl_subscribe" violates foreign key
45 constraint "sl_subscribe-sl_path-ref"
- DETAIL: Key (sub_provider,sub_receiver)=(1,4)
```

```
- is not present in table "sl_path".
- INFO remoteListenThread_1: disconnecting from
- 'dbname=customers host=mainhost.com
50 port=5432 user=slony_user password=slony_user_pass'
- %
```

Это означает что в служебной таблице _имя< кластера>.sl_path;, например _customers_rep.sl_path на уже имеющихся узлах отсутствует информация о новом узле. В данном случае, id нового узла 4, пара (1,4) в sl_path отсутствует.

Видимо, это баг Slony. Как избежать этого и последующих ручных вмешательств пока не ясно.

Чтобы это устранить, нужно выполнить на каждом из имеющихся узлов приблизительно следующий запрос (добавить путь, в данном случае (1,4)):

Листинг 4.40 Устранение неисправностей

```
Line 1 slony_user@masterhost$ psql -d customers -h
     _every_one_of_slaves -U slony
- customers=# insert into _customers_rep.sl_path
- values ('1','4','dbname=customers host=mainhost.com
- port=5432 user=slony_user password=slony_user_password,'10')
     ;
```

Если возникают затруднения, да и вообще для расширения кругозора можно посмотреть на служебные таблицы и их содержимое. Они не видны обычно и находятся в рамках пространства имён _имя< кластера>, например _customers_rep.

Что делать если репликация со временем начинает тормозить

В процессе эксплуатации наблюдаю как со временем растёт нагрузка на master-сервере, в списке активных бекендов — постоянные SELECT-ы со слейвов. В pg_stat_activity видим примерно такие запросы:

Листинг 4.41 Устранение неисправностей

```
Line 1 select ev_origin, ev_seqno, ev_timestamp, ev_minxid,
     ev_maxxid, ev_xip,
- ev_type, ev_data1, ev_data2, ev_data3, ev_data4, ev_data5,
     ev_data6,
- ev_data7, ev_data8 from "_customers_rep".sl_event e where
- (e.ev_origin = '2' and e.ev_seqno > '336996') or
5 (e.ev_origin = '3' and e.ev_seqno > '1712871') or
- (e.ev_origin = '4' and e.ev_seqno > '721285') or
- (e.ev_origin = '5' and e.ev_seqno > '807715') or
- (e.ev_origin = '1' and e.ev_seqno > '3544763') or
- (e.ev_origin = '6' and e.ev_seqno > '2529445') or
```

```
10 (e.ev_origin = '7' and e.ev_seqno > '2512532') or
 - (e.ev_origin = '8' and e.ev_seqno > '2500418') or
 - (e.ev_origin = '10' and e.ev_seqno > '1692318')
 - order by e.ev_origin, e.ev_seqno;
```

Не забываем что _customers_rep — имя схемы из примера, у вас будет другое имя.

Таблица sl_event почему-то разрастается со временем, замедляя выполнение этих запросов до неприемлемого времени. Удаляем ненужные записи:

Листинг 4.42 Устранение неисправностей

```
Line 1 delete from _customers_rep.sl_event where
 - ev_timestamp<NOW()-'1 DAY'::interval;
```

Производительность должна вернуться к изначальным значениям. Возможно имеет смысл почистить таблицы _customers_rep.sl_log_* где вместо звёздочки подставляются натуральные числа, по-видимому по количеству репликационных сетов, так что _customers_rep.sl_log_1 точно должна существовать.

4.5 Londiste

Введение

Londiste представляет собой движок для организации репликации, написанный на языке python. Основные принципы: надежность и простота использования. Из-за этого данное решение имеет меньше функциональности, чем Slony-I. Londiste использует в качестве транспортного механизма очередь PgQ (описание этого более чем интересного проекта остается за рамками данной главы, поскольку он представляет интерес скорее для низкоуровневых программистов баз данных, чем для конечных пользователей — администраторов СУБД PostgreSQL). Отличительными особенностями решения являются:

- возможность потабличной репликации;
- начальное копирование ничего не блокирует;
- возможность двухстороннего сравнения таблиц;
- простота установки.

К недостаткам можно отнести:

- триггерная репликация, что ухудшает производительность базы.

Установка

На серверах, которые мы настраиваем рассматривается ОС Linux, а именно Ubuntu Server. Автор данной книги считает, что под другие операционные системы (кроме Windows) все мало чем будет отличаться, а держать кластера PostgreSQL под ОС Windows, по меньшей мере, неразумно.

Поскольку Londiste — это часть Skytools, то нам нужно ставить этот пакет. На таких системах, как Debian или Ubuntu skytools можно найти в репозитории пакетов и поставить одной командой:

Листинг 4.43 Установка

```
% sudo aptitude install skytools
```

Но в системных пакетах может содержатся версия 2.x, которая не поддерживает каскадную репликацию, отказоустойчивость(failover) и переключение между серверами (switchover). По этой причине я не буду её рассматривать. Скачать самую последнюю версию пакета можно с официального сайта. На момент написания главы последняя версия была 3.2. Итак, начнем:

Листинг 4.44 Установка

```
$ wget http://pgfoundry.org/frs/download.php/3622/skytools
    -3.2.tar.gz
$ tar zxvf skytools-3.2.tar.gz
$ cd skytools-3.2/
# пакеты для сборки deb
$ sudo aptitude install build-essential autoconf \
automake autotools-dev dh-make \
debhelper devscripts fakeroot xutils lintian pbuilder \
python-all-dev python-support xmlto asciidoc \
libevent-dev libpq-dev libtool
# python-psycopg нужен для работы Londiste
$ sudo aptitude install python-psycopg2 postgresql-server-
    dev-all
# данной командой собираем deb пакет
$ make deb
$ cd ../
# ставим skytools
$ dpkg -i *.deb
```

Для других систем можно собрать Skytools командами:

Листинг 4.45 Установка

```
$ ./configure
$ make
$ make install
```

Дальше проверим, что все у нас правильно установилось

Листинг 4.46 Установка

```
$ londiste3 -V
londiste3, Skytools version 3.2
$ pgqd -V
bad switch: usage: pgq-ticker [switches] config.file
Switches:
  -v        Increase verbosity
  -q        No output to console
  -d        Daemonize
  -h        Show help
  -V        Show version
 --ini      Show sample config file
  -s        Stop - send SIGINT to running process
  -k        Kill - send SIGTERM to running process
  -r        Reload - send SIGHUP to running process
```

Если у Вас похожий вывод, значит все установлено правильно и можно приступать к настройке.

Настройка

Обозначения:

- master-host — мастер база данных;
- slave1-host, slave2-host, slave3-host, slave4-host — слейв базы данных;
- l3simple - название реплицируемой базы данных;

Создаём конфигурацию репликаторов

Для начала создадим конфигурационный файл для master базы (пусть конфиг будет у нас /etc/skytools/master-londiste.ini):

Листинг 4.47 Создаём конфигурацию репликатора

```
[londiste3]
job_name = master_l3simple
db = dbname=l3simple
queue_name = replika
logfile = /var/log/skytools/master_l3simple.log
pidfile = /var/pid/skytools/master_l3simple.pid

# Задержка между проверками наличия активности
# новых( пакетов данных) в секундах
loop_delay = 0.5
```

Инициализируем Londiste для master базы:

Листинг 4.48 Инициализируем Londiste

```
Line 1  $ londiste3 /etc/skytools/master-londiste.ini create-root
        master-node "dbname=l3simple host=master-host"
    -   INFO plpgsql is installed
    -   INFO Installing pgq
    -   INFO   Reading from /usr/share/skytools3/pgq.sql
    5   INFO pgq.get_batch_cursor is installed
    -   INFO Installing pgq_ext
    -   INFO   Reading from /usr/share/skytools3/pgq_ext.sql
    -   INFO Installing pgq_node
    -   INFO   Reading from /usr/share/skytools3/pgq_node.sql
    10  INFO Installing londiste
    -   INFO   Reading from /usr/share/skytools3/londiste.sql
    -   INFO londiste.global_add_table is installed
    -   INFO Initializing node
    -   INFO Location registered
    15  INFO Node "master-node" initialized for queue "replika" with
        type "root"
    -   INFO Done
```

master-server — это имя провайдера (мастера базы).

Теперь можем запустить демон:

Листинг 4.49 Запускаем демон для master базы

```
Line 1  $ londiste3 -d /etc/skytools/master-londiste.ini worker
    -   $ tail -f /var/log/skytools/master_l3simple.log
    -   INFO {standby: 1}
    -   INFO {standby: 1}
```

Если нужно перегрузить демон (например, изменили конфиг), то можно воспользоватся параметром -r:

Листинг 4.50 Перегрузка демона

```
Line 1  $ londiste3 /etc/skytools/master-londiste.ini -r
```

Для остановки демона есть параметр -s:

Листинг 4.51 Остановка демона

```
Line 1  $ londiste3 /etc/skytools/master-londiste.ini -s
```

или если потребуется «убить» (kill -9) демон:

Листинг 4.52 Остановка демона

```
Line 1  $ londiste3 /etc/skytools/master-londiste.ini -k
```

Для автоматизации этого процесса skytools3 имеет встроенный демон, который подымает все воркеры из директории /etc/skytools/. Сама конфигурация демона находится в /etc/skytools.ini. Что бы запустить все демоны londiste достаточно выполнить:

Листинг 4.53 Демон для ticker

```
Line 1  $ /etc/init.d/skytools3 start
    -   INFO Starting master_l3simple
```

Перейдем к slave базе.

Для начала мы должны создать базу данных:

Листинг 4.54 Копирования структуры базы

```
Line 1  $ psql -h slave1-host -U postgres
    -   # CREATE DATABASE l3simple;
```

Подключение должно быть «trust» (без паролей) между master и slave базами данных.

Далее создадим конфиг для slave базы (/etc/skytools/slave1-londiste.ini):

Листинг 4.55 Создаём конфигурацию для slave

```
Line 1  [londiste3]
    -   job_name = slave1_l3simple
    -   db = dbname=l3simple
    -   queue_name = replika
    5   logfile = /var/log/skytools/slave1_l3simple.log
    -   pidfile = /var/pid/skytools/slave1_l3simple.pid
    -
    -   # Задержка между проверками наличия активности
    -   # новых( пакетов данных) в секундах
    10  loop_delay = 0.5
```

Инициализируем Londiste для slave базы:

Листинг 4.56 Инициализируем Londiste для slave

```
Line 1  $ londiste3 /etc/skytools/slave1-londiste.ini create-leaf
        slave1-node "dbname=l3simple host=slave1-host" --provider
        ="dbname=l3simple host=master-host"
```

Теперь можем запустить демон:

Листинг 4.57 Запускаем демон для slave базы

```
Line 1  $ londiste3 -d /etc/skytools/slave1-londiste.ini worker
```

Или же через главный демон:

Листинг 4.58 Запускаем демон для slave базы

```
Line 1  $ /etc/init.d/skytools3 start
    -   INFO Starting master_l3simple
    -   INFO Starting slave1_l3simple
```

Создаём конфигурацию для PgQ ticker

Londiste требуется PgQ ticker для работы с мастер базой данных, который может быть запущен и на другой машине. Но, конечно, лучше его запускать на той же, где и master база данных. Для этого мы настраиваем специальный конфиг для ticker демона (пусть конфиг будет у нас /etc/skytools/pgqd.ini):

Листинг 4.59 PgQ ticker конфиг

```
[pgqd]
logfile = /var/log/skytools/pgqd.log
pidfile = /var/pid/skytools/pgqd.pid
```

Запускаем демон:

Листинг 4.60 Запускаем PgQ ticker

```
$ pgqd -d /etc/skytools/pgqd.ini
$ tail -f /var/log/skytools/pgqd.log
LOG Starting pgqd 3.2
LOG auto-detecting dbs ...
LOG l3simple: pgq version ok: 3.2
```

Или же через глобальный демон:

Листинг 4.61 Запускаем PgQ ticker

```
$ /etc/init.d/skytools3 restart
INFO Starting master_l3simple
INFO Starting slave1_l3simple
INFO Starting pgqd
LOG Starting pgqd 3.2
```

Теперь можно увидеть статус кластера:

Листинг 4.62 Статус кластера

```
$ londiste3 /etc/skytools/master-londiste.ini status
Queue: replika   Local node: slave1-node

master-node (root)
  |                           Tables: 0/0/0
  |                           Lag: 44s, Tick: 5
  +--: slave1-node (leaf)
                              Tables: 0/0/0
                              Lag: 44s, Tick: 5

$ londiste3 /etc/skytools/master-londiste.ini members
Member info on master-node@replika:
node_name         dead              node_location
--------------    --------------
    --------------------------------
```

```
15 master-node       False          dbname=l3simple host=
       master-host
   slave1-node       False          dbname=l3simple host=
       slave1-host
```

Но репликация еще не запущенна: требуется добавить таблици в очередь, которые мы хотим реплицировать. Для этого используем команду add-table:

Листинг 4.63 Добавляем таблицы

```
Line 1 $ londiste3 /etc/skytools/master-londiste.ini add-table --
       all
   $ londiste3 /etc/skytools/slave1-londiste.ini add-table --
       all --create-full
```

В данном примере используется параметр --all, который означает все таблицы, но вместо него вы можете перечислить список конкретных таблиц, если не хотите реплицировать все. Если имена таблиц отличаются на master и slave, то можно использовать --dest-table параметр при добавлении таблиц на slave базе. Также, если вы не перенесли струкруру таблиц заранее с master на slave базы, то это можно сделать автоматически через --create параметр (или --create-full, если нужно перенести полностью всю схему таблицы).

Подобным образом добавляем последовательности (sequences) для репликации:

Листинг 4.64 Добавляем последовательности

```
Line 1 $ londiste3 /etc/skytools/master-londiste.ini add-seq --all
   $ londiste3 /etc/skytools/slave1-londiste.ini add-seq --all
```

Но последовательности должны на slave базе созданы заранее (тут не поможет --create-full для таблиц). Поэтому иногда проще перенести точную копию структуры master базы на slave:

Листинг 4.65 Клонирование структуры базы

```
Line 1 $ pg_dump -s -npublic l3simple | psql -hslave1-host l3simple
```

Далее проверяем состояние репликации:

Листинг 4.66 Статус кластера

```
Line 1 $ londiste3 /etc/skytools/master-londiste.ini status
   Queue: replika    Local node: master-node
   
   master-node (root)
 5   |                           Tables: 4/0/0
   |                           Lag: 18s, Tick: 12
   +--: slave1-node (leaf)
                               Tables: 0/4/0
                               Lag: 18s, Tick: 12
```

Как можно заметить, возле «Table» содержится три цифры (x/y/z). Каждая обозначает:

- x - количество таблиц в состоянии «ok» (replicated). На master базе указывает, что она в норме, а на slave базах - таблица синхронизирована с master базой;
- y - количество таблиц в состоянии half (initial copy, not finnished), у master должно быть 0, а у slave базах это указывает количество таблиц в процессе копирования;
- z - количество таблиц в состоянии ignored (table not replicated locally), у master должно быть 0, а у slave базах это количество таблиц, которые не добавлены для репликации с мастера (т.е. master отдает на репликацию эту таблицу, но slave их просто не забирает).

Через небольшой интервал времени все таблици должны синхронизироватся:

Листинг 4.67 Статус кластера

```
$ londiste3 /etc/skytools/master-londiste.ini status
Queue: replika   Local node: master-node

master-node (root)
  |                           Tables: 4/0/0
  |                           Lag: 31s, Tick: 20
  +--: slave1-node (leaf)
                              Tables: 4/0/0
                              Lag: 31s, Tick: 20
```

Дополнительно Londiste позволяет просмотреть состояние таблиц и последовательностей на master и slave базах:

Листинг 4.68 Статус таблиц и последовательностей

```
$ londiste3 /etc/skytools/master-londiste.ini tables
Tables on node
table_name                merge_state       table_attrs
------------------------  ---------------   ---------------
public.pgbench_accounts   ok
public.pgbench_branches   ok
public.pgbench_history    ok
public.pgbench_tellers    ok

$ londiste3 /etc/skytools/master-londiste.ini seqs
Sequences on node
seq_name                          local             last_value
--------------------------------  ---------------
    ---------------
public.pgbench_history_hid_seq    True              33345
```

Проверка

Для этого буду использовать pgbench утилиту. Запуская добавление данных в таблицу и мониторим логи одновлеменно:

Листинг 4.69 Проверка

```
$ pgbench -T 10 -c 5 l3simple
$ tail -f /var/log/skytools/slave1_l3simple.log
INFO {count: 1508, duration: 0.307, idle: 0.0026}
INFO {count: 1572, duration: 0.3085, idle: 0.002}
INFO {count: 1600, duration: 0.3086, idle: 0.0026}
INFO {count: 36, duration: 0.0157, idle: 2.0191}
```

Как видно по логам slave база успешно реплицируется с master базой.

Каскадная репликация

Каскадная репликация позволяет реплицировать данные с одного слейва на другой. Создадим конфиг для второго slave (пусть конфиг будет у нас /etc/skytools/slave2-londiste.ini):

Листинг 4.70 Конфиг для slave2

```
[londiste3]
job_name = slave2_l3simple
db = dbname=l3simple host=slave2-host
queue_name = replika
logfile = /var/log/skytools/slave2_l3simple.log
pidfile = /var/pid/skytools/slave2_l3simple.pid

# Задержка между проверками наличия активности
# новых( пакетов данных) в секундах
loop_delay = 0.5
```

Для создания slave, от которого можно реплицировать другие базы данных используется команда create-branch вместо create-leaf (root, корень - master нода, предоставляет информацию для репликации; branch, ветка - нода с копией данных, с которой можно реплицировать; leaf, лист - нода с копией данными, но реплицировать с нее уже не возможно):

Листинг 4.71 Инициализируем slave2

```
$ psql -hslave2-host -d postgres -c "CREATE DATABASE
    l3simple;"
$ pg_dump -s -npublic l3simple | psql -hslave2-host l3simple
$ londiste3 /etc/skytools/slave2-londiste.ini create-branch
    slave2-node "dbname=l3simple host=slave2-host" --provider
    ="dbname=l3simple host=master-host"
INFO plpgsql is installed
INFO Installing pgq
```

```
- INFO   Reading from /usr/share/skytools3/pgq.sql
- INFO pgq.get_batch_cursor is installed
- INFO Installing pgq_ext
- INFO   Reading from /usr/share/skytools3/pgq_ext.sql
10 INFO Installing pgq_node
- INFO   Reading from /usr/share/skytools3/pgq_node.sql
- INFO Installing londiste
- INFO   Reading from /usr/share/skytools3/londiste.sql
- INFO londiste.global_add_table is installed
15 INFO Initializing node
- INFO Location registered
- INFO Location registered
- INFO Subscriber registered: slave2-node
- INFO Location registered
20 INFO Location registered
- INFO Location registered
- INFO Node "slave2-node" initialized for queue "replika" with
    type "branch"
- INFO Done
```

Далее добавляем все таблицы и последовательности:

Листинг 4.72 Инициализируем slave2

```
Line 1 $ londiste3 /etc/skytools/slave2-londiste.ini add-table --
    all
- $ londiste3 /etc/skytools/slave2-londiste.ini add-seq --all
```

И запускаем новый демон:

Листинг 4.73 Инициализируем slave2

```
Line 1 $ /etc/init.d/skytools3 start
- INFO Starting master_l3simple
- INFO Starting slave1_l3simple
- INFO Starting slave2_l3simple
5 INFO Starting pgqd
- LOG Starting pgqd 3.2
```

Повторим вышеперечисленные операции для slave3 и slave4, только поменяем provider для них:

Листинг 4.74 Инициализируем slave3 и slave4

```
Line 1 $ londiste3 /etc/skytools/slave3-londiste.ini create-branch
    slave3-node "dbname=l3simple host=slave3-host" --provider
    ="dbname=l3simple host=slave2-host"
- $ londiste3 /etc/skytools/slave4-londiste.ini create-branch
    slave4-node "dbname=l3simple host=slave4-host" --provider
    ="dbname=l3simple host=slave3-host"
```

В результате получаем такую картину с кластером:

Листинг 4.75 Кластер с каскадной репликацией

```
$ londiste3 /etc/skytools/slave4-londiste.ini status
Queue: replika   Local node: slave4-node

master-node (root)
  |                                  Tables: 4/0/0
  |                                  Lag: 9s, Tick: 49
  +--: slave1-node (leaf)
  |                                  Tables: 4/0/0
  |                                  Lag: 9s, Tick: 49
  +--: slave2-node (branch)
     |                               Tables: 4/0/0
     |                               Lag: 9s, Tick: 49
     +--: slave3-node (branch)
        |                            Tables: 4/0/0
        |                            Lag: 9s, Tick: 49
        +--: slave4-node (branch)
                                     Tables: 4/0/0
                                     Lag: 9s, Tick: 49
```

Londiste позволяет «на лету» изменять топологию кластера. Например, изменим «provider» для slave4:

Листинг 4.76 Изменяем топологию

```
$ londiste3 /etc/skytools/slave4-londiste.ini change-
   provider --provider="dbname=l3simple host=slave2-host"
$ londiste3 /etc/skytools/slave4-londiste.ini status
Queue: replika   Local node: slave4-node

master-node (root)
  |                                  Tables: 4/0/0
  |                                  Lag: 12s, Tick: 56
  +--: slave1-node (leaf)
  |                                  Tables: 4/0/0
  |                                  Lag: 12s, Tick: 56
  +--: slave2-node (branch)
     |                               Tables: 4/0/0
     |                               Lag: 12s, Tick: 56
     +--: slave3-node (branch)
     |                               Tables: 4/0/0
     |                               Lag: 12s, Tick: 56
     +--: slave4-node (branch)
                                     Tables: 4/0/0
                                     Lag: 12s, Tick: 56
```

Также топологию можно менять с стороны репликатора через команду takeover:

Листинг 4.77 Изменяем топологию

```
Line 1  $ londiste3 /etc/skytools/slave3-londiste.ini takeover
           slave4-node
    -   $ londiste3 /etc/skytools/slave4-londiste.ini status
    -   Queue: replika   Local node: slave4-node
    -
    5   master-node (root)
    -     |                                   Tables: 4/0/0
    -     |                                   Lag: 9s, Tick: 49
    -     +--: slave1-node (leaf)
    -     |                                   Tables: 4/0/0
   10     |                                   Lag: 9s, Tick: 49
    -     +--: slave2-node (branch)
    -         |                               Tables: 4/0/0
    -         |                               Lag: 9s, Tick: 49
    -         +--: slave3-node (branch)
   15             |                           Tables: 4/0/0
    -             |                           Lag: 9s, Tick: 49
    -             +--: slave4-node (branch)
    -                                         Tables: 4/0/0
    -                                         Lag: 9s, Tick: 49
```

Через команду drop-node можно удалить slave из кластера:

Листинг 4.78 Удаляем ноду

```
Line 1  $ londiste3 /etc/skytools/slave4-londiste.ini drop-node
           slave4-node
    -   $ londiste3 /etc/skytools/slave3-londiste.ini status
    -   Queue: replika   Local node: slave3-node
    -
    5   master-node (root)
    -     |                                   Tables: 4/0/0
    -     |                                   Lag: 9s, Tick: 49
    -     +--: slave1-node (leaf)
    -     |                                   Tables: 4/0/0
   10     |                                   Lag: 9s, Tick: 49
    -     +--: slave2-node (branch)
    -         |                               Tables: 4/0/0
    -         |                               Lag: 9s, Tick: 49
    -         +--: slave3-node (branch)
   15                                         Tables: 4/0/0
    -                                         Lag: 9s, Tick: 49
```

Команда tag-dead может использоваться, что бы указать slave как не живой (прекратить на него репликацию), а через команду tag-alive его можно вернуть в кластер.

Общие задачи

Проверка состояния слейвов

Этот запрос на мастере дает некоторую информацию о каждой очереди и слейве.

Листинг 4.79 Проверка состояния слейвов

```
# SELECT queue_name, consumer_name, lag, last_seen FROM pgq.
    get_consumer_info();
 queue_name |      consumer_name       |       lag        |
    last_seen
--
    ----------+--------------------------+------------------+----------------

 replika    | .global_watermark        | 00:03:37.108259 |
    00:02:33.013915
 replika    | slave1_l3simple          | 00:00:32.631509 |
    00:00:32.533911
 replika    | .slave1-node.watermark | 00:03:37.108259 |
    00:03:05.01431
```

lag столбец показывает отставание от мастера в синхронизации, last_seen — время последней запроса от слейва. Значение этого столбца не должно быть больше, чем 60 секунд для конфигурации по умолчанию.

Удаление очереди всех событий из мастера

При работе с Londiste может потребоваться удалить все ваши настройки для того, чтобы начать все заново. Для PGQ, чтобы остановить накопление данных, используйте следующие API:

Листинг 4.80 Удаление очереди всех событий из мастера

```
SELECT pgq.unregister_consumer('queue_name', 'consumer_name'
    );
```

Добавление столбца в таблицу

Добавляем в следующей последовательности:

1. добавить поле на все слейвы;
2. BEGIN; – на мастере;
3. добавить поле на мастере;
4. COMMIT;

Удаление столбца из таблицы

1. BEGIN; – на мастере;
2. удалить поле на мастере;
3. COMMIT;
4. Проверить lag, когда londiste пройдет момент удаления поля;
5. удалить поле на всех слейвах.

Хитрость тут в том, чтобы удалить поле на слейвах только тогда, когда больше нет событий в очереди на это поле.

Устранение неисправностей

Londiste пожирает процессор и lag растет

Это происходит, например, если во время сбоя админ забыл перезапустить ticker. Или когда вы сделали большой UPDATE или DELETE в одной транзакции, но теперь что бы реализовать каждое событие в этом запросе создаются транзакции на слейвах ...

Следующий запрос позволяет подсчитать, сколько событий пришло в pgq.subscription в колонках sub_last_tick и sub_next_tick.

Листинг 4.81 Устранение неисправностей

```
Line 1  SELECT count(*)
    -     FROM pgq.event_1,
    -       (SELECT tick_snapshot
    -         FROM pgq.tick
    5         WHERE tick_id BETWEEN 5715138 AND 5715139
    -       ) as t(snapshots)
    - WHERE txid_visible_in_snapshot(ev_txid, snapshots);
```

В нашем случае, это было более чем 5 миллионов и 400 тысяч событий. Чем больше событий с базы данных требуется обработать Londiste, тем больше ему требуется памяти для этого. Мы можем сообщить Londiste не загружать все события сразу. Достаточно добавить в INI конфиг PgQ ticker следующую настройку:

Листинг 4.82 Устранение неисправностей

```
Line 1  pgq_lazy_fetch = 500
```

Теперь Londiste будет брать максимум 500 событий в один пакет запросов. Остальные попадут в следующие пакеты запросов.

4.6 Bucardo

Введение

Bucardo — асинхронная master-master или master-slave репликация PostgreSQL, которая написана на Perl. Система очень гибкая, поддерживает несколько видов синхронизации и обработки конфликтов.

Установка

Установку будет проводиться на Ubuntu Server. Сначала нам нужно установить DBIx::Safe Perl модуль.

Листинг 4.83 Установка

```
Line 1 $ apt-get install libdbix-safe-perl
```

Для других систем можно поставить из исходников:

Листинг 4.84 Установка

```
Line 1 $ tar xvfz dbix_safe.tar.gz
  - $ cd DBIx-Safe-1.2.5
  - $ perl Makefile.PL
  - $ make
  5 $ make test
  - $ sudo make install
```

Теперь ставим сам Bucardo. Скачиваем его и инсталлируем:

Листинг 4.85 Установка

```
Line 1 $ wget http://bucardo.org/downloads/Bucardo-5.0.0.tar.gz
  - $ tar xvfz Bucardo-5.0.0.tar.gz
  - $ cd Bucardo-5.0.0
  - $ perl Makefile.PL
  5 $ make
  - $ sudo make install
```

Для работы Bucardo потребуется установить поддержку pl/perl языка PostgreSQL.

Листинг 4.86 Установка

```
Line 1 $ sudo aptitude install postgresql-plperl-9.3
```

и дополнительные пакеты для Perl (DBI, DBD::Pg, Test::Simple, boolean):

Листинг 4.87 Установка

```
Line 1 $ sudo aptitude install libdbd-pg-perl libboolean-perl
```

Можем приступать к настройке.

Настройка

Инициализация Bucardo

Запускаем установку командой:

Листинг 4.88 Инициализация Bucardo

```
$ bucardo install
```

Bucardo покажет настройки подключения к PostgreSQL, которые можно будет изменить:

Листинг 4.89 Инициализация Bucardo

```
This will install the bucardo database into an existing
   Postgres cluster.
Postgres must have been compiled with Perl support,
and you must connect as a superuser

We will create a new superuser named 'bucardo',
and make it the owner of a new database named 'bucardo'

Current connection settings:
1. Host:           <none>
2. Port:           5432
3. User:           postgres
4. Database:       postgres
5. PID directory: /var/run/bucardo
```

Когда вы измените требуемые настройки и подтвердите установку, Bucardo создаст пользователя bucardo и базу данных bucardo. Данный пользователь должен иметь право логиниться через Unix socket, поэтому лучше заранее дать ему такие права в pg_hda.conf.

После успешной установки мы можем проверить конфигурацию через команду bucardo show all:

Листинг 4.90 Инициализация Bucardo

```
$ bucardo show all
autosync_ddl              = newcol
bucardo_initial_version   = 5.0.0
bucardo_vac               = 1
bucardo_version           = 5.0.0
ctl_checkonkids_time      = 10
ctl_createkid_time        = 0.5
ctl_sleep                 = 0.2
default_conflict_strategy = bucardo_latest
default_email_from        = nobody@example.com
default_email_host        = localhost
default_email_to          = nobody@example.com
...
```

Настройка баз данных

Теперь нам нужно настроить базы данных, с которыми будет работать Bucardo. Пусть у нас будет master_db и slave_db. Реплицировать будем simple_database базу. Сначала настроим мастер:

Листинг 4.91 Настройка баз данных

```
Line 1 $ bucardo add db master_db dbname=simple_database host=
       master_host
   - Added database "master_db"
```

Данной командой мы указали базу данных и дали ей имя master_db (для того, что в реальной жизни master_db и slave_db имеют одинаковое название базы simple_database и их нужно отличать в Bucardo).

Дальше добавляем slave_db:

Листинг 4.92 Настройка баз данных

```
Line 1 $ bucardo add db slave_db dbname=simple_database port=5432
       host=slave_host
```

Настройка репликации

Теперь нам нужно настроить синхронизацию между этими базами данных. Делается это командой sync:

Листинг 4.93 Настройка репликации

```
Line 1 $ bucardo add sync delta dbs=master_db:source,slave_db:
       target conflict_strategy=bucardo_latest tables=all
   - Added sync "delta"
   - Created a new relgroup named "delta"
   - Created a new dbgroup named "delta"
 5   Added table "public.pgbench_accounts"
   -   Added table "public.pgbench_branches"
   -   Added table "public.pgbench_history"
   -   Added table "public.pgbench_tellers"
```

Данной командой мы установим Bucardo триггеры в PostgreSQL для master-slave репликации. Значения параметров:

- dbs — список баз данных, которые следует синхронизировать. Значение source или target указывает, что это master или slave база данных соответственно (их может быть больше одной);
- conflict_strategy — для работы в режиме master-master нужно указать как Bucardo должен решать конфликты синхронизации. Существуют следующие стратегии:
  - bucardo_source — при конфликте мы копируем данные с source;
  - bucardo_target — при конфликте мы копируем данные с target;

  - bucardo_skip — конфликт мы просто не реплицируем. Не рекомендуется для продакшен систем;
  - bucardo_random — каждая БД имеет одинаковый шанс, что её изменение будет взято для решение конфликта;
  - bucardo_latest — запись, которая была последней изменена решает конфликт;
  - bucardo_abort — синхронизация прерывается;

- tables — таблици, которые требуется синхронизировать. Через «all» указываем все;

Для master-master репликации:

Листинг 4.94 Настройка репликации

```
Line 1 $ bucardo add sync delta dbs=master_db:source,slave_db:
       source conflict_strategy=bucardo_latest tables=all
```

Пример для создания master-master и master-slave репликации:

Листинг 4.95 Настройка репликации

```
Line 1 $ bucardo add sync delta dbs=master_db1:source,master_db2:
       source,slave_db1:target,slave_db2:target
       conflict_strategy=bucardo_latest tables=all
```

Проверка состояния репликации:

Листинг 4.96 Проверка состояния репликации

```
Line 1 $ bucardo status
   - PID of Bucardo MCP: 12122
   - Name    State    Last good    Time    Last I/D    Last bad
         Time
   - ========+========+============+=======+===========+==========+========

   5 delta | Good   | 13:28:53   | 13m 6s | 3685/7384 | none
       |
```

Запуск/Остановка репликации

Запуск репликации:

Листинг 4.97 Запуск репликации

```
Line 1 $ bucardo start
```

Остановка репликации:

Листинг 4.98 Остановка репликации

```
Line 1 $ bucardo stop
```

Общие задачи

Просмотр значений конфигурации

Листинг 4.99 Просмотр значений конфигурации

```
Line 1 $ bucardo show all
```

Изменения значений конфигурации

Листинг 4.100 Изменения значений конфигурациии

```
Line 1 $ bucardo set name=value
```

Например:

Листинг 4.101 Изменения значений конфигурации

```
Line 1 $ bucardo_ctl set syslog_facility=LOG_LOCAL3
```

Перегрузка конфигурации

Листинг 4.102 Перегрузка конфигурации

```
Line 1 $ bucardo reload_config
```

Более подробную информацию можно найти на официальном сайте.

Репликация в другие типы баз данных

Начиная с версии 5.0 Bucardo поддерживает репликацию в другие источники данных: drizzle, mongo, mysql, oracle, redis и sqlite (тип базы задается при использовании команды bucardo add db через ключ «type», который по умолчанию postgres). Давайте рассмотрим пример с redis. Для начала потребуется установить redis адаптер для Perl (для других баз устанавливаются соответствующие):

Листинг 4.103 Установка redis

```
Line 1 $ aptitude install libredis-perl
```

Далее зарегистрируем redis базу в Bucardo:

Листинг 4.104 Добавление redis базы

```
Line 1 $ bucardo add db R dbname=simple_database type=redis
   - Added database "R"
```

Создадим группу баз данных под названием pg_to_redis:

Листинг 4.105 Группа баз данных

```
Line 1 $ bucardo add dbgroup pg_to_redis master_db:source slave_db:
     source R:target
   - Created dbgroup "pg_to_redis"
   - Added database "master_db" to dbgroup "pg_to_redis" as
     source
```

```
- Added database "slave_db" to dbgroup "pg_to_redis" as source
5 Added database "R" to dbgroup "pg_to_redis" as target
```

И создадим репликацию:

Листинг 4.106 Установка sync

```
Line 1 $ bucardo add sync pg_to_redis_sync tables=all dbs=
     pg_to_redis status=active
- Added sync "pg_to_redis_sync"
-   Added table "public.pgbench_accounts"
-   Added table "public.pgbench_branches"
5   Added table "public.pgbench_history"
-   Added table "public.pgbench_tellers"
```

После перезапуска Bucardo данные с PostgreSQL таблиц начнуть реплицироватся в Redis:

Листинг 4.107 Репликация в redis

```
Line 1 $ pgbench -T 10 -c 5 simple_database
- $ redis-cli monitor
- "HMSET" "pgbench_history:6" "bid" "2" "aid" "36291" "delta"
     "3716" "mtime" "2014-07-11 14:59:38.454824" "hid" "4331"
- "HMSET" "pgbench_history:2" "bid" "1" "aid" "65179" "delta"
     "2436" "mtime" "2014-07-11 14:59:38.500896" "hid" "4332"
5 "HMSET" "pgbench_history:14" "bid" "2" "aid" "153001" "delta
     " "-264" "mtime" "2014-07-11 14:59:38.472706" "hid" "4333
     "
- "HMSET" "pgbench_history:15" "bid" "1" "aid" "195747" "delta
     " "-1671" "mtime" "2014-07-11 14:59:38.509839" "hid" "
     4334"
- "HMSET" "pgbench_history:3" "bid" "2" "aid" "147650" "delta"
      "3237" "mtime" "2014-07-11 14:59:38.489878" "hid" "4335"
- "HMSET" "pgbench_history:15" "bid" "1" "aid" "39521" "delta"
      "-2125" "mtime" "2014-07-11 14:59:38.526317" "hid" "4336
     "
- "HMSET" "pgbench_history:14" "bid" "2" "aid" "60105" "delta"
      "2555" "mtime" "2014-07-11 14:59:38.616935" "hid" "4337"
10 "HMSET" "pgbench_history:15" "bid" "2" "aid" "186655" "delta
     " "930" "mtime" "2014-07-11 14:59:38.541296" "hid" "4338"
- "HMSET" "pgbench_history:15" "bid" "1" "aid" "101406" "delta
     " "668" "mtime" "2014-07-11 14:59:38.560971" "hid" "4339"
- "HMSET" "pgbench_history:15" "bid" "2" "aid" "126329" "delta
     " "-4236" "mtime" "2014-07-11 14:59:38.5907" "hid" "4340"
- "DEL" "pgbench_tellers:20"
```

Данные в Redis хранятся в виде хешей:

Листинг 4.108 Данные в redis

```
Line 1 $ redis-cli "HGETALL" "pgbench_history:15"
```

```
 1) "bid"
 2) "2"
 3) "aid"
 4) "126329"
 5) "delta"
 6) "-4236"
 7) "mtime"
 8) "2014-07-11 14:59:38.5907"
 9) "hid"
10) "4340"
```

Также можно проверить состояние репликации:

Листинг 4.109 Установка redis

```
$ bucardo status
PID of Bucardo MCP: 4655
 Name                 State    Last good    Time     Last I/D
      Last bad    Time
-------------------+---------+------------+---------+-----------+-----------+
delta               | Good    | 14:59:39   | 8m 15s | 0/0
    | none       |
pg_to_redis_sync | Good    | 14:59:40   | 8m 14s | 646/2546
    | 14:59:39   | 8m 15s
```

4.7 RubyRep

Введение

RubyRep представляет собой движок для организации асинхронной репликации, написанный на языке ruby. Основные принципы: простота использования и не зависит от БД. Поддерживает как master-master, так и master-slave репликацию, может работать с PostgreSQL и MySQL. Отличительными особенностями решения являются:

- возможность двухстороннего сравнения и синхронизации баз данных;
- простота установки.

К недостаткам можно отнести:

- работа только с двумя базами данных для MySQL;
- медленная работа синхронизации;
- при больших объемах данных «ест» процессор и память;
- проект не развивается.

Установка

RubyRep поддерживает два типа установки: через стандартный Ruby или JRuby. Рекомендуется ставить JRuby вариант — производительность на порядок выше.

Установка JRuby версии

Предварительно должна быть установлена Java.

1. загрузите последнюю версию JRuby rubyrep с Rubyforge;
2. распакуйте;

Установка стандартной Ruby версии

1. установить Ruby, Rubygems;
2. установить драйвера базы данных;

   Для MySQL:

   Листинг 4.110 Установка

```
Line 1    $ sudo gem install mysql2
-
```

   Для PostgreSQL:

   Листинг 4.111 Установка

```
Line 1    $ sudo gem install postgres
-
```

3. Устанавливаем rubyrep:

   Листинг 4.112 Установка

```
Line 1    $ sudo gem install rubyrep
-
```

Настройка

Создание файла конфигурации

Выполним команду:

Листинг 4.113 Настройка

```
Line 1  $ rubyrep generate myrubyrep.conf
```

Команда generate создала пример конфигурации в файл myrubyrep.conf:

Листинг 4.114 Настройка

```
Line 1  RR::Initializer::run do |config|
-     config.left = {
```

```
    :adapter  => 'postgresql', # or 'mysql'
    :database => 'SCOTT',
    :username => 'scott',
    :password => 'tiger',
    :host     => '172.16.1.1'
  }

  config.right = {
    :adapter  => 'postgresql',
    :database => 'SCOTT',
    :username => 'scott',
    :password => 'tiger',
    :host     => '172.16.1.2'
  }

  config.include_tables 'dept'
  config.include_tables /^e/ # regexp matches all tables
   starting with e
  # config.include_tables /./ # regexp matches all tables
end
```

В настройках просто разобраться. Базы данных делятся на «left» и «right». Через config.include_tables мы указываем какие таблицы включать в репликацию (поддерживает RegEx).

Сканирование баз данных

Сканирование баз данных для поиска различий:

Листинг 4.115 Сканирование баз данных

```
$ rubyrep scan -c myrubyrep.conf
```

Пример вывода:

Листинг 4.116 Сканирование баз данных

```
dept 100% ..........................    0
emp 100% ..........................    1
```

Таблица dept полностью синхронизирована, а emp — имеет одну не синхронизированую запись.

Синхронизация баз данных

Выполним команду:

Листинг 4.117 Синхронизация баз данных

```
$ rubyrep sync -c myrubyrep.conf
```

Также можно указать только какие таблицы в базах данных синхронизировать:

Листинг 4.118 Синхронизация баз данных

```
Line 1 $ rubyrep sync -c myrubyrep.conf dept /^e/
```

Настройки политики синхронизации позволяют указывать как решать конфликты синхронизации. Более подробно можно почитать в документации.

Репликация

Для запуска репликации достаточно выполнить:

Листинг 4.119 Репликация

```
Line 1 $ rubyrep replicate -c myrubyrep.conf
```

Данная команда установит репликацию (если она не была установлена) на базы данных и запустит её. Чтобы остановить репликацию, достаточно просто убить процесс. Даже если репликация остановлена, все изменения будут обработаны триггерами rubyrep. После перезагрузки, все изменения будут автоматически восстановлены.

Для удаления репликации достаточно выполнить:

Листинг 4.120 Репликация

```
Line 1 $ rubyrep uninstall -c myrubyrep.conf
```

Устранение неисправностей

Ошибка при запуске репликации

При запуске rubyrep через Ruby может возникнуть подобная ошибка:

Листинг 4.121 Устранение неисправностей

```
Line 1 $ rubyrep replicate -c myrubyrep.conf
  - Verifying RubyRep tables
  - Checking for and removing rubyrep triggers from unconfigured
      tables
  - Verifying rubyrep triggers of configured tables
  5 Starting replication
  - Exception caught: Thread#join: deadlock 0xb76ee1ac - mutual
    join(0xb758cfac)
```

Это проблема с запусками потоков в Ruby. Решается запуском на JRuby.

4.8 Заключение

Репликация — одна из важнейших частей крупных приложений, которые работают на PostgreSQL. Она помогает распределять нагрузку на

базу данных, делать фоновый бэкап одной из копий без нагрузки на центральный сервер, создавать отдельный сервер для логирования и м.д.

В главе было рассмотрено несколько видов репликации PostgreSQL. Нельзя четко сказать какая лучше всех. Потоковая репликация — один из самых лучших вариантов для поддержки идентичных кластеров баз данных, но доступна только с 9.0 версии PostgreSQL. Slony-I — громоздкая и сложная в настройке система, но имеющая в своем арсенале множество функций, таких как поддержка каскадной репликации, отказоустойчивости (failover) и переключение между серверами (switchover). В тоже время Londiste имея в своем арсенале подобный функционал, может похвастатся еще компактностью и простой в установке. Bucardo — система которая может быть или master-master, или master-slave репликацией. RubyRep, как для master-master репликации, очень просто в установке и настройке, но за это ему приходится расплачиваться скоростью работы — самый медленный из всех (синхронизация больших объемов данных между таблицами).

5

Шардинг

Если ешь слона, не пытайся
запихать его в рот целиком.

Народная мудрость

5.1 Введение

Шардинг — разделение данных на уровне ресурсов. Концепция шардинга заключается в логическом разделении данных по различным ресурсам исходя из требований к нагрузке.

Рассмотрим пример. Пусть у нас есть приложение с регистрацией пользователей, которое позволяет писать друг другу личные сообщения. Допустим оно очень популярно и много людей им пользуются ежедневно. Естественно, что таблица с личными сообщениями будет намного больше всех остальных таблиц в базе (скажем, будет занимать 90% всех ресурсов). Зная это, мы можем подготовить для этой (только одной!) таблицы выделенный сервер помощнее, а остальные оставить на другом (послабее). Теперь мы можем идеально подстроить сервер для работы с одной специфической таблицей, постараться уместить ее в память, возможно, дополнительно партиционировать ее и т.д. Такое распределение называется вертикальным шардингом.

Что делать, если наша таблица с сообщениями стала настолько большой, что даже выделенный сервер под нее одну уже не спасает. Необходимо делать горизонтальный шардинг — т.е. разделение одной таблицы по разным ресурсам. Как это выглядит на практике? Все просто. На разных серверах у нас будет таблица с одинаковой структурой, но разными данными. Для нашего случая с сообщениями, мы можем хранить первые 10 миллионов сообщений на одном сервере, вторые 10 - на втором и т.д. Т.е. необходимо иметь критерий шардинга — какой-то параметр, который позволит определять, на каком именно сервере лежат те или иные данные.

Обычно, в качестве параметра шардинга выбирают ID пользователя (user_id) — это позволяет делить данные по серверам равномерно и про-

сто. Т.о. при получении личных сообщений пользователей алгоритм работы будет такой:

- Определить, на каком сервере БД лежат сообщения пользователя исходя из user_id;
- Инициализировать соединение с этим сервером;
- Выбрать сообщения.

Задачу определения конкретного сервера можно решать двумя путями:

- Хранить в одном месте хеш-таблицу с соответствиями «пользователь=сервер». Тогда, при определении сервера, нужно будет выбрать сервер из этой таблицы. В этом случае узкое место — это большая таблица соответствия, которую нужно хранить в одном месте. Для таких целей очень хорошо подходят базы данных «ключ=значение»;
- Определять имя сервера с помощью числового (буквенного) преобразования. Например, можно вычислять номер сервера, как остаток от деления на определенное число (количество серверов, между которыми Вы делите таблицу). В этом случае узкое место — это проблема добавления новых серверов — Вам придется делать перераспределение данных между новым количеством серверов.

Для шардинга не существует решения на уровне известных платформ, т.к. это весьма специфическая для отдельно взятого приложения задача.

Естественно, делая горизонтальный шардинг, Вы ограничиваете себя в возможности выборок, которые требуют пересмотра всей таблицы (например, последние посты в блогах людей будет достать невозможно, если таблица постов шардится). Такие задачи придется решать другими подходами. Например, для описанного примера, можно при появлении нового поста, заносить его ID в общий стек, размером в 100 элементом.

Горизонтальный шардинг имеет одно явное преимущество — он бесконечно масштабируем. Для создания шардинга PostgreSQL существует несколько решений:

- Postgres-XC
- PL/Proxy
- HadoopDB (Shared-nothing clustering)
- Greenplum Database

5.2 PL/Proxy

PL/Proxy представляет собой прокси-язык для удаленного вызова процедур и партицирования данных между разными базами. Основная идея

его использования заключается в том, что появляется возможность вызывать функции, расположенные в удаленных базах, а также свободно работать с кластером баз данных (например, вызвать функцию на всех узлах кластера, или на случайном узле, или на каком-то одном определенном).

Чем PL/Proxy может быть полезен? Он существенно упрощает горизонтальное масштабирование системы. Становится удобным разделять таблицу с пользователями, например, по первой латинской букве имени — на 26 узлов. При этом приложение, которое работает непосредственно с прокси-базой, ничего не будет замечать: запрос на авторизацию, например, сам будет направлен прокси-сервером на нужный узел. То есть администратор баз данных может проводить масштабирование системы практически независимо от разработчиков приложения.

PL/Proxy позволяет полностью решить проблемы масштабирования OLTP систем. В систему легко вводится резервирование с failover-ом не только по узлам, но и по самим прокси-серверам, каждый из которых работает со всеми узлами.

Недостатки и ограничения:

- все запросы и вызовы функций вызываются в autocommit-режиме на удаленных серверах;
- в теле функции разрешен только один SELECT; при необходимости нужно писать отдельную процедуру;
- при каждом вызове прокси-сервер стартует новое соединение к бакенд-серверу; в высоконагруженных системах;
- целесообразно использовать менеджер для кеширования соединений к бакенд-серверам, для этой цели идеально подходит PgBouncer;
- изменение конфигурации кластера (количества партиций, например) требует перезапуска прокси-сервера.

Установка

1. Скачать PL/Proxy и распаковать;
2. Собрать PL/Proxy командами make и make install.

Так же можно установить PL/Proxy из репозитория пакетов. Например в Ubuntu Server достаточно выполнить команду для PostgreSQL 9.4:

Листинг 5.1 Установка

```
Line 1 $ sudo aptitude install postgresql-9.4-plproxy
```

Настройка

Для примера настройки используется 3 сервера PostgreSQL. 2 сервера пусть будут node1 и node2, а главный, что будет проксировать запросы на два других — proxy. Для корректной работы pl/proxy рекомендуется

использовать количество нод равное степеням двойки. База данных будет называтся plproxytest, а таблица в ней — users. Начнем!

Для начала настроим node1 и node2. Команды написанные ниже нужно выполнять на каждой ноде.

Создадим базу данных plproxytest(если её ещё нет):

Листинг 5.2 Настройка

```
Line 1 CREATE DATABASE plproxytest
-        WITH OWNER = postgres
-          ENCODING = 'UTF8';
```

Добавляем табличку users:

Листинг 5.3 Настройка

```
Line 1 CREATE TABLE public.users
-    (
-     username character varying(255),
-     email character varying(255)
5    )
-    WITH (OIDS=FALSE);
- ALTER TABLE public.users OWNER TO postgres;
```

Теперь создадим функцию для добавления данных в таблицу users:

Листинг 5.4 Настройка

```
Line 1 CREATE OR REPLACE FUNCTION public.insert_user(i_username
       text,
- i_emailaddress    text)
- RETURNS integer AS
- $BODY$
5 INSERT INTO public.users (username, email) VALUES ($1,$2);
-      SELECT 1;
- $BODY$
-   LANGUAGE 'sql' VOLATILE;
- ALTER FUNCTION public.insert_user(text, text) OWNER TO
       postgres;
```

С настройкой нодов закончено. Приступим к серверу proxy.

Как и на всех нодах, на главном сервере (proxy) должна присутствовать база данных:

Листинг 5.5 Настройка

```
Line 1 CREATE DATABASE plproxytest
-        WITH OWNER = postgres
-          ENCODING = 'UTF8';
```

Теперь надо указать серверу что эта база данных управляется с помощью pl/proxy:

Листинг 5.6 Настройка

```
Line 1  CREATE OR REPLACE FUNCTION public.plproxy_call_handler()
  -       RETURNS language_handler AS
  -     '$libdir/plproxy', 'plproxy_call_handler'
  -       LANGUAGE 'c' VOLATILE
  5   COST 1;
  -   ALTER FUNCTION public.plproxy_call_handler()
  -   OWNER TO postgres;
  -   -- language
  -   CREATE LANGUAGE plproxy HANDLER plproxy_call_handler;
 10   CREATE LANGUAGE plpgsql;
```

Также, для того что бы сервер знал где и какие ноды у него есть, надо создать 3 сервисные функции, которые pl/proxy будет использовать в своей работе. Первая функция — конфиг для кластера баз данных. Тут указываются параметры через key-value:

Листинг 5.7 Настройка

```
Line 1  CREATE OR REPLACE FUNCTION public.get_cluster_config
  -     (IN cluster_name text,   OUT "key" text, OUT val text)
  -       RETURNS SETOF record AS
  -     $BODY$
  5   BEGIN
  -       -- lets use same config for all clusters
  -       key := 'connection_lifetime';
  -       val := 30*60; -- 30m
  -       RETURN NEXT;
 10       RETURN;
  -   END;
  -   $BODY$
  -       LANGUAGE 'plpgsql' VOLATILE
  -       COST 100
 15       ROWS 1000;
  -   ALTER FUNCTION public.get_cluster_config(text)
  -   OWNER TO postgres;
```

Вторая важная функция, код которой надо будет подправить. В ней надо будет указать DSN нод:

Листинг 5.8 Настройка

```
Line 1  CREATE OR REPLACE FUNCTION
  -     public.get_cluster_partitions(cluster_name text)
  -       RETURNS SETOF text AS
  -     $BODY$
  5   BEGIN
  -       IF cluster_name = 'usercluster' THEN
  -         RETURN NEXT 'dbname=plproxytest host=node1 user=postgres
          ';
```

```
-       RETURN NEXT 'dbname=plproxytest host=node2 user=postgres
        ';
-       RETURN;
10    END IF;
-     RAISE EXCEPTION 'Unknown cluster';
- END;
- $BODY$
-   LANGUAGE 'plpgsql' VOLATILE
15    COST 100
-   ROWS 1000;
- ALTER FUNCTION public.get_cluster_partitions(text)
- OWNER TO postgres;
```

И последняя:

Листинг 5.9 Настройка

```
Line 1 CREATE OR REPLACE FUNCTION
-  public.get_cluster_version(cluster_name text)
-    RETURNS integer AS
- $BODY$
5  BEGIN
-    IF cluster_name = 'usercluster' THEN
-      RETURN 1;
-    END IF;
-    RAISE EXCEPTION 'Unknown cluster';
10 END;
- $BODY$
-   LANGUAGE 'plpgsql' VOLATILE
-   COST 100;
- ALTER FUNCTION public.get_cluster_version(text)
15 OWNER TO postgres;
```

Ну и собственно самая главная функция, которая будет вызываться уже непосредственно в приложении:

Листинг 5.10 Настройка

```
Line 1 CREATE OR REPLACE FUNCTION
-  public.insert_user(i_username text, i_emailaddress text)
-    RETURNS integer AS
- $BODY$
5    CLUSTER 'usercluster';
-    RUN ON hashtext(i_username);
- $BODY$
-   LANGUAGE 'plproxy' VOLATILE
-   COST 100;
10 ALTER FUNCTION public.insert_user(text, text)
- OWNER TO postgres;
```

Все готово. Подключаемся к серверу proxy и заносим данные в базу:

Листинг 5.11 Настройка

```
Line 1  SELECT insert_user('Sven','sven@somewhere.com');
    -   SELECT insert_user('Marko', 'marko@somewhere.com');
    -   SELECT insert_user('Steve','steve@somewhere.com');
```

Пробуем извлечь данные. Для этого напишем новую серверную функцию:

Листинг 5.12 Настройка

```
Line 1  CREATE OR REPLACE FUNCTION
    -   public.get_user_email(i_username text)
    -    RETURNS SETOF text AS
    -   $BODY$
    5    CLUSTER 'usercluster';
    -    RUN ON hashtext(i_username) ;
    -    SELECT email FROM public.users
    -    WHERE username = i_username;
    -   $BODY$
   10    LANGUAGE 'plproxy' VOLATILE
    -    COST 100
    -    ROWS 1000;
    -   ALTER FUNCTION public.get_user_email(text)
    -   OWNER TO postgres;
```

И попробуем её вызвать:

Листинг 5.13 Настройка

```
Line 1  SELECT plproxy.get_user_email('Steve');
```

Если потом подключиться к каждой ноде отдельно, то можно четко увидеть, что данные users разбросаны по таблицам каждой ноды.

Все ли так просто?

Как видно на тестовом примере ничего сложного в работе с pl/proxy нет. Но, я думаю все кто смог дочитать до этой строчки уже поняли что в реальной жизни все не так просто. Представьте что у вас 16 нод. Это же надо как-то синхронизировать код функций. А что если ошибка закрадётся — как её оперативно исправлять?

Этот вопрос был задан и на конференции Highload++ 2008, на что Аско Ойя ответил что соответствующие средства уже реализованы внутри самого Skype, но ещё не достаточно готовы для того что бы отдавать их на суд сообществу opensource.

Второй проблема, которая не дай бог коснётся вас при разработке такого рода системы, это проблема перераспределения данных в тот момент когда нам захочется добавить ещё нод в кластер. Планировать эту

масштабную операцию придётся очень тщательно, подготовив все сервера заранее, занеся данные и потом в один момент подменив код функции get_cluster_partitions.

5.3 Postgres-XC

Postgres-XC – система для создания мульти-мастер кластеров, работающих в синхронном режиме – все узлы всегда содержат актуальные данные. Postgres-XC поддерживает опции для увеличения масштабирования кластера как при преобладании операций записи, так и при основной нагрузке на чтение данных: поддерживается выполнение транзакций с распараллеливанием на несколько узлов, за целостностью транзакций в пределах всего кластера отвечает специальный узел GTM (Global Transaction Manager).

Измерение производительности показало, что КПД кластера Postgres-XC составляет примерно 64%, т.е. кластер из 10 серверов позволяет добиться увеличения производительности системы в целом в 6.4 раза, относительно производительности одного сервера (цифры приблизительные).

Система не использует в своей работе триггеры и представляет собой набор дополнений и патчей к PostgreSQL, дающих возможность в прозрачном режиме обеспечить работу в кластере стандартных приложений, без их дополнительной модификации и адаптации (полная совместимость с PostgreSQL API). Кластер состоит из одного управляющего узла (GTM), предоставляющего информацию о состоянии транзакций, и произвольного набора рабочих узлов, каждый из которых в свою очередь состоит из координатора и обработчика данных (обычно эти элементы реализуются на одном сервере, но могут быть и разделены).

Хоть Postgres-XC и выглядит похожим на MultiMaster, но он им не является. Все сервера кластера должны быть соединены сетью с минимальными задержками, никакое географически-распределенное решение с разумной производительностью построить на нем невозможно (это важный момент).

Архитектура

Рис. 5.1 показывает архитектуру Postgres-XC с тремя её основными компонентами:

1. Глобальный менеджер транзакций (GTM) — собирает и обрабатывает информацию о транзакциях в Postgres-XC, решает вопросы глобального идентификатора транзакции по операциям (для поддержания согласованного представления базы данных на всех узлах). Он обеспечивает поддержку других глобальных данных, таких как по-

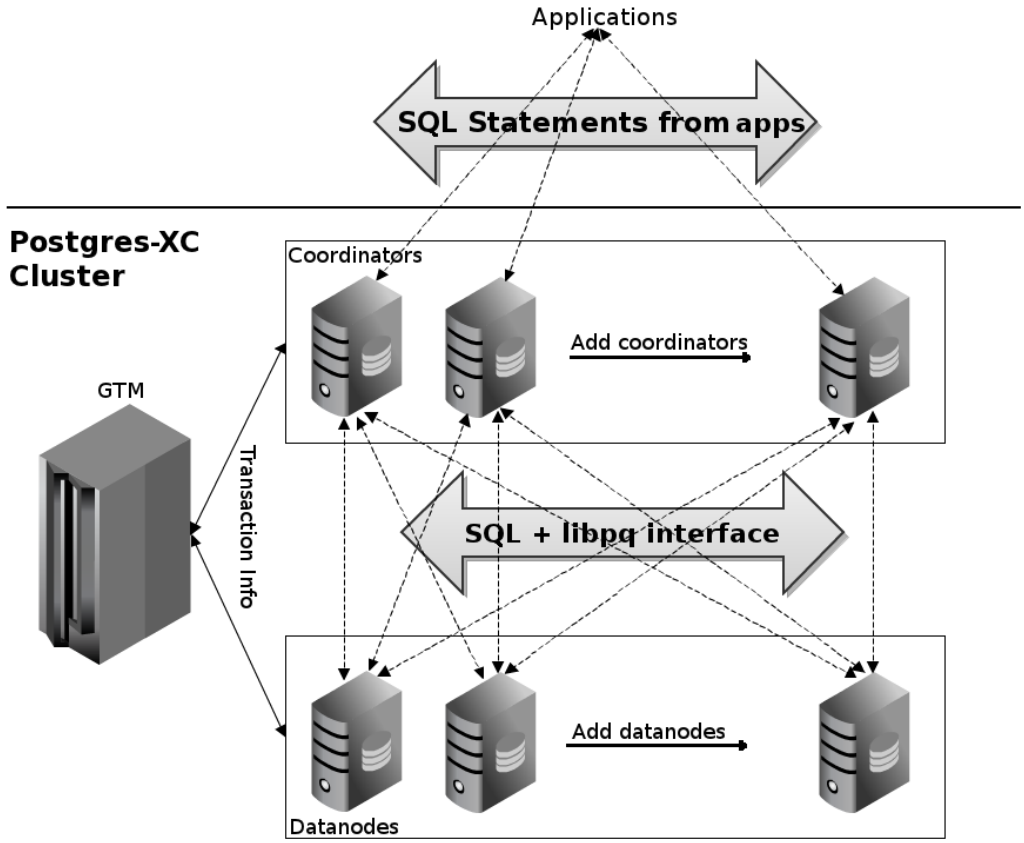

Рис. 5.1: Архитектура Postgres-XC

следовательности и временные метки. Он хранит данные пользователя, за исключением управляющей информации.

2. Координаторы (coordinators) — обеспечивают точку подключения для клиента (приложения). Они несут ответственность за разбор и выполнение запросов от клиентов и возвращение результатов (при необходимости). Они не хранят пользовательские данные, а собирают их из обработчиков данных (datanodes) с помощью запросов SQL через PostgreSQL интерфейс. Координаторы также обрабатывают данные, если требуется, и даже управляют двухфазной фиксацией. Координаторы используются также для разбора запросов, составления планов запросов, поиска данных и т.д.
3. Обработчики данных (datanodes) — обеспечивают сохранение пользовательских данных. Datanodes выполняют запросы от координаторов и возвращают им полученный результат.

Установка

Установить Postgres-XC можно из исходников или же из пакетов системы. Например в Ubuntu 14.04 можно установить postgres-xc так:

Листинг 5.14 Установка Postgres-XC

```
Line 1  $ sudo apt-get install postgres-xc postgres-xc-client
        postgres-xc-contrib postgres-xc-server-dev
```

По умолчанию он создаст один координатор и два обработчика данных.

Распределение данных и масштабируемость

Postgres-XC предусматривает два способа хранения данных в таблицах:

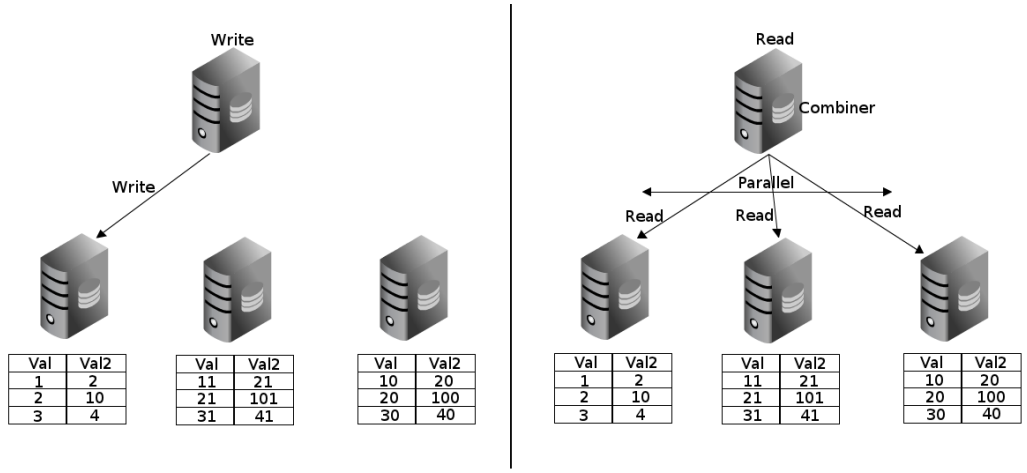

Рис. 5.2: Распределенные таблицы

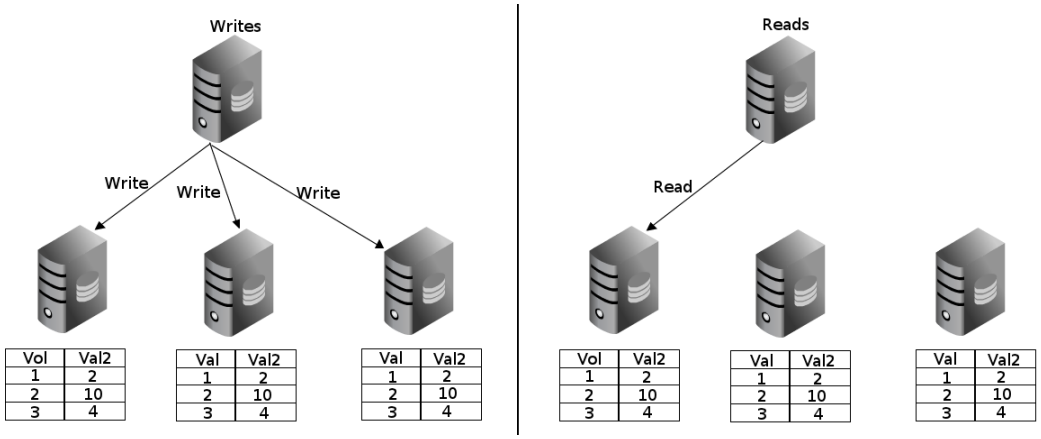

Рис. 5.3: Реплицированные таблицы

1. Распределенные таблицы (distributed tables, рис. 5.2): данные по таблице распределяются на указанный набор обработчиков данных с использованием указанной стратегии (hash, round-robin, modulo). Каждая запись в таблице находится только на одном обработчике данных. Параллельно могут быть записаны или прочитаны данные с

различных обработчиков данных. За счет этого значительно улучшена производительность на запись и чтение;

2. Реплицированные таблицы (replicated tables, рис. 5.3): данные по таблице реплицируется (клонируются) на указанный набор обработчиков данных. Каждая запись в таблице находится на всех обработчиках данных (которые были указаны) и любые изменения дублируются на все обработчики данных. Так как все данные доступны на любом обработчике данных, координатор может собрать все данные из одного узла, что позволяет направить различные запросы на различные обработчики данных. Таким образом создается балансировка нагрузки и увеличения пропускной способности на чтение.

Таблицы и запросы к ним

После установки работа с Postgres-XC ведется как с обыкновенным PostgreSQL. Подключаться для работы с данными нужно только к координаторам (по умолчанию координатор работает на порту 5432). Для начала создадим распределенные таблицы.

Листинг 5.15 Создание распределенных таблиц

```
Line 1  CREATE TABLE
  -     users_with_hash (id SERIAL, type INT, ...)
  -     DISTRIBUTE by HASH(id) TO NODE dn1, dn2;
  -
  5     CREATE TABLE
  -     users_with_modulo (id SERIAL, type INT, ...)
  -     DISTRIBUTE by MODULO(id) TO NODE dn1, dn2;
  -
  -     CREATE TABLE
 10     users_with_rrobin (id SERIAL, type INT, ...)
  -     DISTRIBUTE by ROUND ROBIN TO NODE dn1, dn2;
```

На листинге 5.15 создано 3 распределенные таблицы:

1. Таблица users_with_hash распределяется по хешу значения из указанного поля в таблице (тут указано поле id) по обработчикам данных. Вот как распределились первые 15 значений:

Листинг 5.16 Данные с координатора и обработчиков данных

```
Line 1  # координатор
  -     $ psql
  -     # SELECT id, type from users_with_hash ORDER BY id;
  -      id    | type
  5     -------+-------
  -           1 |    946
  -           2 |    153
  -           3 |    484
```

```
-        4 |    422
10       5 |    785
-        6 |    906
-        7 |    973
-        8 |    699
-        9 |    434
15      10 |    986
-       11 |    135
-       12 |   1012
-       13 |    395
-       14 |    667
20      15 |    324
-
- # первый обработчик данных
- $ psql -p15432
- # SELECT id, type from users_with_hash ORDER BY id;
25   id   | type
- ------+-------
-       1 |    946
-       2 |    153
-       5 |    785
30      6 |    906
-       8 |    699
-       9 |    434
-      12 |   1012
-      13 |    395
35    15 |    324
-
- # второй обработчик данных
- $ psql -p15433
- # SELECT id, type from users_with_hash ORDER BY id;
40  id    | type
- -------+-------
-       3 |    484
-       4 |    422
-       7 |    973
45     10 |    986
-       11 |    135
-       14 |    667
```

2. Таблица users_with_modulo распределяется по модулю значения из указанного поля в таблице (тут указано поле id) по обработчикам данных. Вот как распределились первые 15 значений:

Листинг 5.17 Данные с координатора и обработчиков данных

```
Line 1  # координатор
-       $ psql
-       # SELECT id, type from users_with_modulo ORDER BY id;
```

```
 id    | type
-------+-------
     1 |   883
     2 |   719
     3 |    29
     4 |   638
     5 |   363
     6 |   946
     7 |   440
     8 |   331
     9 |   884
    10 |   199
    11 |    78
    12 |   791
    13 |   345
    14 |   476
    15 |   860

# первый обработчик данных
$ psql -p15432
# SELECT id, type from users_with_modulo ORDER BY id;
 id    | type
-------+-------
     2 |   719
     4 |   638
     6 |   946
     8 |   331
    10 |   199
    12 |   791
    14 |   476

# второй обработчик данных
$ psql -p15433
# SELECT id, type from users_with_modulo ORDER BY id;
 id   | type
------+-------
    1 |   883
    3 |    29
    5 |   363
    7 |   440
    9 |   884
   11 |    78
   13 |   345
   15 |   860
```

3. Таблица users_with_rrobin распределяется циклическим способом(round-robin) по обработчикам данных. Вот как распределились первые 15 значений:

Листинг 5.18 Данные с координатора и обработчиков данных

```
# координатор
$ psql
# SELECT id, type from users_with_rrobin ORDER BY id;
 id  | type
-----+------
  1  |  890
  2  |  198
  3  |  815
  4  |  446
  5  |   61
  6  |  337
  7  |  948
  8  |  446
  9  |  796
 10  |  422
 11  |  242
 12  |  795
 13  |  314
 14  |  240
 15  |  733

# первый обработчик данных
$ psql -p15432
# SELECT id, type from users_with_rrobin ORDER BY id;
 id  | type
-----+------
  2  |  198
  4  |  446
  6  |  337
  8  |  446
 10  |  422
 12  |  795
 14  |  240

# второй обработчик данных
$ psql -p15433
# SELECT id, type from users_with_rrobin ORDER BY id;
 id  | type
-----+------
  1  |  890
  3  |  815
  5  |   61
  7  |  948
  9  |  796
 11  |  242
 13  |  314
```

```
-      15 |    733
```

Теперь создадим реплицированную таблицу:

Листинг 5.19 Создание реплицированной таблицы

```
Line 1  CREATE TABLE
-       users_replicated (id SERIAL, type INT, ...)
-       DISTRIBUTE by REPLICATION TO NODE dn1, dn2;
```

Естественно данные идентичны на всех обработчиках данных:

Листинг 5.20 Данные с координатора и обработчиков данных

```
Line 1  # SELECT id, type from users_replicated  ORDER BY id;
-         id    | type
-       -------+-------
-            1 |     75
5            2 |    262
-            3 |    458
-            4 |    779
-            5 |    357
-            6 |     51
10           7 |    249
-            8 |    444
-            9 |    890
-           10 |    810
-           11 |    809
15          12 |    166
-           13 |    605
-           14 |    401
-           15 |     58
```

Рассмотрим как выполняются запросы для таблиц. Выберем все записи из распределенной таблицы:

Листинг 5.21 Выборка записей из распределенной таблицы

```
Line 1  # EXPLAIN VERBOSE SELECT * from users_with_modulo ORDER BY
          id;
-                                                 QUERY PLAN
-       --
          ------------------------------------------------------------------------------
-        Sort  (cost=49.83..52.33 rows=1000 width=8)
5          Output: id, type
-          Sort Key: users_with_modulo.id
-          ->  Result  (cost=0.00..0.00 rows=1000 width=8)
-                Output: id, type
-                ->  Data Node Scan on users_with_modulo  (cost
          =0.00..0.00 rows=1000 width=8)
```

```
10                  Output: id, type
-                   Node/s: dn1, dn2
-                   Remote query: SELECT id, type FROM ONLY
    users_with_modulo WHERE true
-  (9 rows)
```

Как видно на листинге 5.21 координатор собирает данные из обработчиков данных, а потом собирает их вместе.

Подсчет суммы с группировкой по полю из распределенной таблицы:

Листинг 5.22 Выборка записей из распределенной таблицы

```
Line 1  # EXPLAIN VERBOSE SELECT sum(id) from users_with_modulo
    GROUP BY type;
-
                        QUERY PLAN
-   --
        ------------------------------------------------------------------------

-    HashAggregate  (cost=5.00..5.01 rows=1 width=8)
5      Output: pg_catalog.sum((sum(users_with_modulo.id))),
    users_with_modulo.type
-      ->  Materialize  (cost=0.00..0.00 rows=0 width=0)
-            Output: (sum(users_with_modulo.id)),
    users_with_modulo.type
-            ->  Data Node Scan on "__REMOTE_GROUP_QUERY__"  (
    cost=0.00..0.00 rows=1000 width=8)
-                  Output: sum(users_with_modulo.id),
    users_with_modulo.type
10                  Node/s: dn1, dn2
-                   Remote query: SELECT sum(group_1.id), group_1
    .type  FROM (SELECT id, type FROM ONLY users_with_modulo
    WHERE true) group_1 GROUP BY 2
-  (8 rows)
```

JOIN между и с участием реплицированных таблиц, а также JOIN между распределенными по одному и тому же полю в таблицах будет выполняются на обработчиках данных. Но JOIN с участием распределенных таблиц по другим ключам будут выполнены на координаторе и скорее всего это будет медленно (листинг 5.23).

Листинг 5.23 Выборка записей из распределенной таблицы

```
Line 1  # EXPLAIN VERBOSE SELECT * from users_with_modulo,
    users_with_hash WHERE users_with_modulo.id =
    users_with_hash.id;
-                                                   QUERY PLAN
-   --
        ------------------------------------------------------------------------
```

```
-  Nested Loop  (cost=0.00..0.01 rows=1 width=16)
5    Output: users_with_modulo.id, users_with_modulo.type,
     users_with_hash.id, users_with_hash.type
-    Join Filter: (users_with_modulo.id = users_with_hash.id)
-    ->  Data Node Scan on users_with_modulo  (cost=0.00..0.00
      rows=1000 width=8)
-          Output: users_with_modulo.id, users_with_modulo.
     type
-          Node/s: dn1, dn2
10         Remote query: SELECT id, type FROM ONLY
     users_with_modulo WHERE true
-    ->  Data Node Scan on users_with_hash  (cost=0.00..0.00
     rows=1000 width=8)
-          Output: users_with_hash.id, users_with_hash.type
-          Node/s: dn1, dn2
-          Remote query: SELECT id, type FROM ONLY
     users_with_hash WHERE true
15 (11 rows)
```

Пример выборки данных из реплицированной таблицы:

Листинг 5.24 Выборка записей из реплицированной таблицы

```
Line 1 # EXPLAIN VERBOSE SELECT * from users_replicated;
-                                  QUERY PLAN
-  --
     ------------------------------------------------------------------------------

-  Data Node Scan on "__REMOTE_FQS_QUERY__"  (cost=0.00..0.00
     rows=0 width=0)
5    Output: users_replicated.id, users_replicated.type
-    Node/s: dn1
-    Remote query: SELECT id, type FROM users_replicated
-  (4 rows)
```

Как видно из запроса для выборки данных используется один обработчик данных, а не все (что и логично).

Высокая доступность (HA)

По архитектуре у Postgres-XC всегда есть согласованность данных. По теореме CAP в такой системе тяжело обеспечить высокую доступность. Для достижения высокой доступности в распределенных системах требуется избыточность данных, резервные копии и автоматическое восстановление. В Postgres-XC избыточность данных может быть достигнута с помощью PostgreSQL потоковой (streaming) репликации с hot-standby для обработчиков данных. Каждый координатор способен записывать и читать данные независимо от другого, поэтому координаторы способны заменять друг друга. Поскольку GTM отдельный процесс и может стать

точкой отказа, лучше создать GTM-standby как резервную копию. Ну а вот для автоматического восстановления придется использовать сторонние утилиты.

Ограничения

1. Postgres-XC базируется на PostgreSQL 9.1 (9.2 в разработке);
2. Нет системы репартиционирования при добавлении или удалении нод (в разработке);
3. Нет глобальных UNIQUE на распределенных таблицах;
4. Не поддерживаются foreign keys между нодами поскольку такой ключ должен вести на данные расположенные на том же обработчике данных;
5. Не поддерживаются курсоры (в разработке);
6. Не поддерживается INSERT ... RETURNING (в разработке);
7. Невозможно удаление и добавление нод в кластер без полной реинициализации кластера (в разработке).

Заключение

Postgres-XC очень перспективное решение для создание кластера на основе PostgreSQL. И хоть это решение имеет ряд недостатков, нестабильно (очень часты случаи падения координаторов при тяжелых запросах) и еще очень молодое, со временем это решение может стать стандартом для масштабирования систем на PostgreSQL.

5.4 HadoopDB

Hadoop представляет собой платформу для построения приложений, способных обрабатывать огромные объемы данных. Система основывается на распределенном подходе к вычислениям и хранению информации, основными ее особенностями являются:

- Масштабируемость: с помощью Hadoop возможно надежное хранение и обработка огромных объемов данных, которые могут измеряться петабайтами;
- Экономичность: информация и вычисления распределяются по кластеру, построенному на самом обыкновенном оборудовании. Такой кластер может состоять из тысяч узлов;
- Эффективность: распределение данных позволяет выполнять их обработку параллельно на множестве компьютеров, что существенно ускоряет этот процесс;

- Надежность: при хранении данных возможно предоставление избыточности, благодаря хранению нескольких копий. Такой подход позволяет гарантировать отсутствие потерь информации в случае сбоев в работе системы;
- Кроссплатформенность: так как основным языком программирования, используемым в этой системе является Java, развернуть ее можно на базе любой операционной системы, имеющей JVM.

HDFS

В основе всей системы лежит распределенная файловая система под незамысловатым названием Hadoop Distributed File System. Представляет она собой вполне стандартную распределенную файловую систему, но все же она обладает рядом особенностей:

- Устойчивость к сбоям, разработчики рассматривали сбои в оборудовании скорее как норму, чем как исключение;
- Приспособленность к развертке на самом обыкновенном ненадежном оборудовании;
- Предоставление высокоскоростного потокового доступа ко всем данным;
- Настроена для работы с большими файлами и наборами файлов;
- Простая модель работы с данными: один раз записали — много раз прочли;
- Следование принципу: переместить вычисления проще, чем переместить данные.

Архитектура HDFS

- Namenode

  Этот компонент системы осуществляет всю работу с метаданными. Он должен быть запущен только на одном компьютере в кластере. Именно он управляет размещением информации и доступом ко всем данным, расположенным на ресурсах кластера. Сами данные проходят с остальных машин кластера к клиенту мимо него.
- Datanode

  На всех остальных компьютерах системы работает именно этот компонент. Он располагает сами блоки данных в локальной файловой системе для последующей передачи или обработки их по запросу клиента. Группы узлов данных принято называть Rack, они используются, например, в схемах репликации данных.
- Клиент

  Просто приложение или пользователь, работающий с файловой системой. В его роли может выступать практически что угодно.

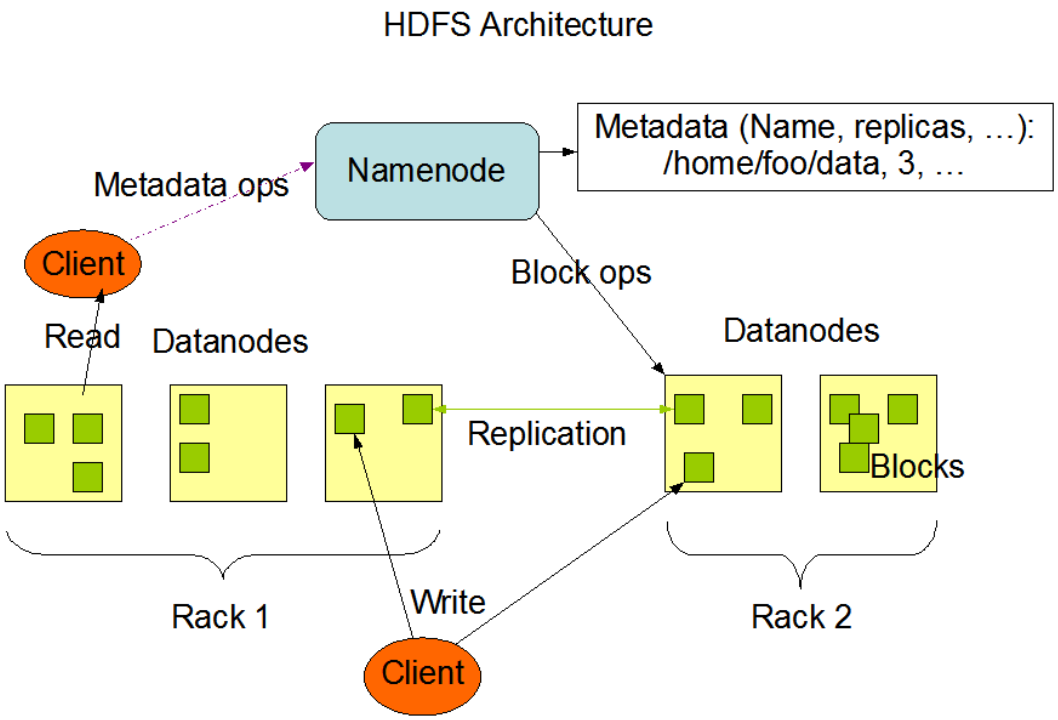

Рис. 5.4: Архитектура HDFS

Пространство имен HDFS имеет классическую иерархическую структуру: пользователи и приложения имеют возможность создавать директории и файлы. Файлы хранятся в виде блоков данных произвольной (но одинаковой, за исключением последнего; по умолчанию 64 mb) длины, размещенных на Datanode'ах. Для обеспечения отказоустойчивости блоки хранятся в нескольких экземплярах на разных узлах, имеется возможность настройки количества копий и алгоритма их распределения по системе. Удаление файлов происходит не сразу, а через какое-то время после соответствующего запроса, так как после получения запроса файл перемещается в директорию /trash и хранится там определенный период времени на случай если пользователь или приложение передумают о своем решении. В этом случае информацию можно будет восстановить, в противном случае — физически удалить.

Для обнаружения возникновения каких-либо неисправностей, Datanode периодически отправляют Namenode'у сигналы о своей работоспособности. При прекращении получения таких сигналов от одного из узлов Namenode помечает его как «мертвый», и прекращает какое-либо с ним взаимодействие до возвращения его работоспособности. Данные, хранившиеся на «умершем» узле реплицируются дополнительный раз из оставшихся «в живых» копий и система продолжает свое функционирование как ни в чем не бывало.

Все коммуникации между компонентами файловой системы проходят по специальным протоколам, основывающимся на стандартном TCP/IP. Клиенты работают с Namenode с помощью так называемого ClientProtocol, а передача данных происходит по DatanodeProtocol, оба они обернуты в Remote Procedure Call (RPC).

Система предоставляет несколько интерфейсов, среди которых командная оболочка DFSShell, набор ПО для администрирования DFSAdmin, а также простой, но эффективный веб-интерфейс. Помимо этого существуют несколько API для языков программирования: Java API, C pipeline, WebDAV и так далее.

MapReduce

Помимо файловой системы, Hadoop включает в себя framework для проведения масштабных вычислений, обрабатывающих огромные объемы данных. Каждое такое вычисление называется Job (задание) и состоит оно, как видно из названия, из двух этапов:

- Map

  Целью этого этапа является представление произвольных данных (на практике чаще всего просто пары ключ-значение) в виде промежуточных пар ключ-значение. Результаты сортируются и групируются по ключу и передаются на следующий этап.
- Reduce

  Полученные после map значения используются для финального вычисления требуемых данных. Практические любые данные могут быть получены таким образом, все зависит от требований и функционала приложения.

Задания выполняются, подобно файловой системе, на всех машинах в кластере (чаще всего одних и тех же). Одна из них выполняет роль управления работой остальных — JobTracker, остальные же ее бесприкословно слушаются — TaskTracker. В задачи JobTracker'а входит составление расписания выполняемых работ, наблюдение за ходом выполнения, и перераспределение в случае возникновения сбоев.

В общем случае каждое приложение, работающее с этим framework'ом, предоставляет методы для осуществления этапов map и reduce, а также указывает расположения входных и выходных данных. После получения этих данных JobTracker распределяет задание между остальными машинами и предоставляет клиенту полную информацию о ходе работ.

Помимо основных вычислений могут выполняться вспомогательные процессы, такие как составление отчетов о ходе работы, кэширование, сортировка и так далее.

HBase

В рамках Hadoop доступна еще и система хранения данных, которую правда сложно назвать СУБД в традиционном смысле этого слова. Чаще

проводят аналогии с проприетарной системой этого же плана от Google — BigTable.

HBase представляет собой распределенную систему хранения больших объемов данных. Подобно реляционным СУБД данные хранятся в виде таблиц, состоящих из строк и столбцов. И даже для доступа к ним предоставляется язык запросов HQL (как ни странно — Hadoop Query Language), отдаленно напоминающий более распространенный SQL. Помимо этого предоставляется итерирующий интерфейс для сканирования наборов строк.

Одной из основных особенностей хранения данных в HBase является возможность наличия нескольких значений, соответствующих одной комбинации таблица-строка-столбец, для их различения используется информация о времени добавления записи. На концептуальном уровне таблицы обычно представляют как набор строк, но физически же они хранятся по столбцам. Это достаточно важный факт, который стоит учитывать при разработки схемы хранения данных. Пустые ячейки не отображаются каким-либо образом физически в хранимых данных, они просто отсутствуют. Существуют конечно и другие нюансы, но я постарался упомянуть лишь основные.

HQL очень прост по своей сути, если Вы уже знаете SQL, то для изучения его Вам понадобится лишь просмотреть по диагонали коротенький вывод команды «help;», занимающий всего пару экранов в консоли. Все те же SELECT, INSERT, UPDATE, DROP и так далее, лишь со слегка измененным синтаксисом.

Помимо обычно командной оболочки HBase Shell, для работы с HBase также предоставлено несколько API для различных языков программирования: Java, Jython, REST и Thrift.

HadoopDB

В проекте HadoopDB специалисты из университетов Yale и Brown предпринимают попытку создать гибридную систему управления данными, сочетающую преимущества технологий и MapReduce, и параллельных СУБД. В их подходе MapReduce обеспечивает коммуникационную инфраструктуру, объединяющую произвольное число узлов, в которых выполняются экземпляры традиционной СУБД. Запросы формулируются на языке SQL, транслируются в среду MapReduce, и значительная часть работы передается в экземпляры СУБД. Наличие MapReduce обеспечивается масштабируемость и отказоустойчивость, а использование в узлах кластера СУБД позволяет добиться высокой производительности.

Установка и настройка

Вся настройка ведется в операционной системе Ubuntu Server.

Установка Hadoop

Перед тем, как приступить собственно говоря к установке Hadoop, необходимо выполнить два элементарных действия, необходимых для правильного функционирования системы:

- открыть доступ одному из пользователей по ssh к этому же компьютеру без пароля, можно например создать отдельного пользователя для этого [hadoop]:

Листинг 5.25 Создаем пользователя с правами

```
$ sudo groupadd hadoop
$ sudo useradd -m -g hadoop -d /home/hadoop -s /bin/
 bash \
-c "Hadoop software owner" hadoop
```

Далее действия выполняем от его имени:

Листинг 5.26 Логинимся под пользователем hadoop

```
$ su hadoop
```

Генерим RSA-ключ для обеспечения аутентификации в условиях отсутствия возможности использовать пароль:

Листинг 5.27 Генерим RSA-ключ

```
$ ssh-keygen -t rsa -P ""
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hadoop/.ssh
 /id_rsa):
Your identification has been saved in /home/hadoop/.
 ssh/id_rsa.
Your public key has been saved in /home/hadoop/.ssh/
 id_rsa.pub.
The key fingerprint is:
7b:5c:cf:79:6b:93:d6:d6:8d:41:e3:a6:9d:04:f9:85
 hadoop@localhost
```

И добавляем его в список авторизованных ключей:

Листинг 5.28 Добавляем его в список авторизованных ключей

```
$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/
 authorized_keys
```

Этого должно быть более чем достаточно. Проверить работоспособность соединения можно просто написав:

Листинг 5.29 Пробуем зайти на ssh без пароля

```
$ ssh localhost
```

Не забываем предварительно инициализировать sshd:

Листинг 5.30 Запуск sshd

```
$ /etc/init.d/sshd start
```

- Помимо этого необходимо убедиться в наличии установленной JVM версии 1.5.0 или выше.

Листинг 5.31 Устанавливаем JVM

```
$ sudo aptitude install openjdk-6-jdk
```

Дальше скачиваем и устанавливаем Hadoop:

Листинг 5.32 Устанавливаем Hadoop

```
$ cd /opt
$ sudo wget http://www.gtlib.gatech.edu/pub/apache/hadoop/
    core/hadoop-0.20.2/hadoop-0.20.2.tar.gz
$ tar zxvf hadoop-0.20.2.tar.gz
$ ln -s /opt/hadoop-0.20.2 /opt/hadoop
$ chown -R hadoop:hadoop /opt/hadoop /opt/hadoop-0.20.2
$ mkdir -p /opt/hadoop-data/tmp-base
$ chown -R hadoop:hadoop /opt/hadoop-data/
```

Далее переходим в /opt/hadoop/conf/hadoop-env.sh и добавляем в начало файла:

Листинг 5.33 Указываем переменные окружения

```
$ export JAVA_HOME=/usr/lib/jvm/java-6-openjdk
$ export HADOOP_HOME=/opt/hadoop
$ export HADOOP_CONF=$HADOOP_HOME/conf
$ export HADOOP_PATH=$HADOOP_HOME/bin
$ export HIVE_HOME=/opt/hive
$ export HIVE_PATH=$HIVE_HOME/bin
$ export PATH=$HIVE_PATH:$HADOOP_PATH:$PATH
```

Далее добавим в /opt/hadoop/conf/hadoop-site.xml:

Листинг 5.34 Настройки hadoop

```
<configuration>
<property>
  <name>hadoop.tmp.dir</name>
  <value>/opt/hadoop-data/tmp-base</value>
```

```
    <description>A base for other temporary directories</
     description>
  </property>

  <property>
    <name>fs.default.name</name>
    <value>localhost:54311</value>
    <description>
      The name of the default file system.
    </description>
  </property>

  <property>
    <name>hadoopdb.config.file</name>
    <value>HadoopDB.xml</value>
    <description>The name of the HadoopDB
    cluster configuration file</description>
  </property>
  </configuration>
```

В /opt/hadoop/conf/mapred-site.xml:

Листинг 5.35 Настройки mapreduce

```
Line 1 <configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:54310</value>
    <description>
      The host and port that the
      MapReduce job tracker runs at.
    </description>
  </property>
  </configuration>
```

В /opt/hadoop/conf/hdfs-site.xml:

Листинг 5.36 Настройки hdfs

```
Line 1 <configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
    <description>
      Default block replication.
    </description>
  </property>
  </configuration>
```

Теперь необходимо отформатировать Namenode:

Листинг 5.37 Форматирование Namenode

```
$ hadoop namenode -format
10/05/07 14:24:12 INFO namenode.NameNode: STARTUP_MSG:
/*
************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = hadoop1/127.0.1.1
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 0.20.2
STARTUP_MSG:   build = https://svn.apache.org/repos
/asf/hadoop/common/branches/branch-0.20 -r
911707; compiled by 'chrisdo' on Fri Feb 19 08:07:34 UTC
    2010
************************************************************
    */
10/05/07 14:24:12 INFO namenode.FSNamesystem:
fsOwner=hadoop,hadoop
10/05/07 14:24:12 INFO namenode.FSNamesystem:
supergroup=supergroup
10/05/07 14:24:12 INFO namenode.FSNamesystem:
isPermissionEnabled=true
10/05/07 14:24:12 INFO common.Storage:
Image file of size 96 saved in 0 seconds.
10/05/07 14:24:12 INFO common.Storage:
Storage directory /opt/hadoop-data/tmp-base/dfs/name has
    been
successfully formatted.
10/05/07 14:24:12 INFO namenode.NameNode:
SHUTDOWN_MSG:
/*
************************************************************
SHUTDOWN_MSG: Shutting down NameNode at hadoop1/127.0.1.1
************************************************************
    */
```

Готово. Теперь мы можем запустить Hadoop:

Листинг 5.38 Запуск Hadoop

```
$ start-all.sh
starting namenode, logging to /opt/hadoop/bin/..
/logs/hadoop-hadoop-namenode-hadoop1.out
localhost: starting datanode, logging to
/opt/hadoop/bin/../logs/hadoop-hadoop-datanode-hadoop1.out
localhost: starting secondarynamenode, logging to
/opt/hadoop/bin/../logs/hadoop-hadoop-secondarynamenode-
    hadoop1.out
```

```
- starting jobtracker, logging to
- /opt/hadoop/bin/../logs/hadoop-hadoop-jobtracker-hadoop1.out
localhost: starting tasktracker, logging to
- /opt/hadoop/bin/../logs/hadoop-hadoop-tasktracker-hadoop1.
    out
```

Остановка Hadoop производится скриптом stop-all.sh.

Установка HadoopDB и Hive

Теперь скачаем HaddopDB и распакуем hadoopdb.jar в $HADOOP_HOME/lib:

Листинг 5.39 Установка HadoopDB

```
Line 1 $ cp hadoopdb.jar $HADOOP_HOME/lib
```

Также нам потребуется PostgreSQL JDBC библиотека. Скачайте её и распакуйте в директорию $HADOOP_HOME/lib.

Hive используется HadoopDB как SQL интерфейс. Подготовим директорию в HDFS для Hive:

Листинг 5.40 Установка HadoopDB

```
Line 1 $ hadoop fs -mkdir /tmp
- $ hadoop fs -mkdir /user/hive/warehouse
- $ hadoop fs -chmod g+w /tmp
- $ hadoop fs -chmod g+w /user/hive/warehouse
```

В архиве HadoopDB также есть архив SMS_dist. Распакуем его:

Листинг 5.41 Установка HadoopDB

```
Line 1 $ tar zxvf SMS_dist.tar.gz
- $ mv dist /opt/hive
- $ chown -R hadoop:hadoop hive
```

Поскольку мы успешно запустили Hadoop, то и проблем с запуском Hive не должно быть:

Листинг 5.42 Установка HadoopDB

```
Line 1 $ hive
- Hive history file=/tmp/hadoop/
- hive_job_log_hadoop_201005081717_1990651345.txt
- hive>
5
- hive> quit;
```

Тестирование

Теперь проведем тестирование. Для этого скачаем бенчмарк:

Листинг 5.43 Тестирование

```
$ svn co http://graffiti.cs.brown.edu/svn/benchmarks/
$ cd benchmarks/datagen/teragen
```

Изменим скрипт benchmarks/datagen/teragen/teragen.pl к виду:

Листинг 5.44 Тестирование

```
use strict;
use warnings;

my $CUR_HOSTNAME = `hostname -s`;
chomp($CUR_HOSTNAME);

my $NUM_OF_RECORDS_1TB     = 10000000000;
my $NUM_OF_RECORDS_535MB   = 100;
my $BASE_OUTPUT_DIR        = "/data";
my $PATTERN_STRING         = "XYZ";
my $PATTERN_FREQUENCY      = 108299;
my $TERAGEN_JAR            = "teragen.jar";
my $HADOOP_COMMAND         = $ENV{'HADOOP_HOME'}."/bin/hadoop";

my %files = ( "535MB" => 1,
);
system("$HADOOP_COMMAND fs -rmr $BASE_OUTPUT_DIR");
foreach my $target (keys %files) {
my $output_dir = $BASE_OUTPUT_DIR."/SortGrep$target";
my $num_of_maps = $files{$target};
my $num_of_records = ($target eq "535MB" ?
$NUM_OF_RECORDS_535MB : $NUM_OF_RECORDS_1TB);
print "Generating $num_of_maps files in '$output_dir'\n";

##
## EXEC: hadoop jar teragen.jar 10000000000
## /data/SortGrep/ XYZ 108299 100
##
my @args = ( $num_of_records,
             $output_dir,
             $PATTERN_STRING,
             $PATTERN_FREQUENCY,
             $num_of_maps );
my $cmd = "$HADOOP_COMMAND jar $TERAGEN_JAR ".join(" ",
    @args);
print "$cmd\n";
system($cmd) == 0 || die("ERROR: $!");
} # FOR
exit(0);
```

При запуске данного Perl скрипта сгенерится данные, которые будут сохранены на HDFS. Поскольку мы настроили систему как единственный кластер, то все данные будут загружены на один HDFS. При работе с большим количеством кластеров данные были бы распределены по кластерам. Создадим базу данных, таблицу и загрузим данные, которые мы сохранили на HDFS, в нее:

Листинг 5.45 Тестирование

```
Line 1  $ hadoop fs -get /data/SortGrep535MB/part-00000 my_file
  -     $ psql
  -     psql> CREATE DATABASE grep0;
  -     psql> USE grep0;
  5     psql> CREATE TABLE grep (
  -         ->    key1 character varying(255),
  -         ->    field character varying(255)
  -         -> );
  -     COPY grep FROM 'my_file' WITH DELIMITER '|';
```

Теперь настроим HadoopDB. В архиве HadoopDB можно найти пример файла Catalog.properties. Распакуйте его и настройте:

Листинг 5.46 Тестирование

```
Line 1  #Properties for Catalog Generation
  -     ######################################
  -     nodes_file=machines.txt
  -     relations_unchunked=grep, EntireRankings
  5     relations_chunked=Rankings, UserVisits
  -     catalog_file=HadoopDB.xml
  -     ##
  -     #DB Connection Parameters
  -     ##
 10     port=5432
  -     username=postgres
  -     password=password
  -     driver=com.postgresql.Driver
  -     url_prefix=jdbc\:postgresql\://
 15     ##
  -     #Chunking properties
  -     ##
  -     chunks_per_node=0
  -     unchunked_db_prefix=grep
 20     chunked_db_prefix=cdb
  -     ##
  -     #Replication Properties
  -     ##
  -     dump_script_prefix=/root/dump_
 25     replication_script_prefix=/root/load_replica_
```

```
- dump_file_u_prefix=/mnt/dump_udb
- dump_file_c_prefix=/mnt/dump_cdb
- ##
- #Cluster Connection
30 ##
- ssh_key=id_rsa
```

Создайте файл machines.txt и добавьте туда строчку «localhost» (без кавычек). Тепер создадим конфиг HadoopDB и скопируем его в HDFS:

Листинг 5.47 Тестирование

```
Line 1 $ java -cp $HADOOP_HOME/lib/hadoopdb.jar \
- > edu.yale.cs.hadoopdb.catalog.SimpleCatalogGenerator \
- > Catalog.properties
- hadoop dfs -put HadoopDB.xml HadoopDB.xml
```

Также возможно создать конфиг для создания репликации командой:

Листинг 5.48 Репликация

```
Line 1 $ java -cp hadoopdb.jar edu.yale.cs.hadoopdb.catalog.
    SimpleRandomReplicationFactorTwo Catalog.properties
```

Инструмент генерирует новый файл HadoopDB.xml, в котором случайные порции данных реплицируются на все узлы. После этого не забываем обновить конфиг на HDFS:

Листинг 5.49 Обновляем конфиг

```
Line 1 $ hadoop dfs -rmr HadoopDB.xml
- $ hadoop dfs -put HadoopDB.xml HadoopDB.xml
```

и поставить «true» для «hadoopdb.config.replication» в HADOOP_HOME/conf/hadoop-site.xml.

Теперь мы готовы проверить работу HadoopDB. Теперь можем протестировать поиск по данным, загруженым ранее в БД и HDFS:

Листинг 5.50 Тестирование

```
Line 1 $ java -cp $CLASSPATH:hadoopdb.jar \
- > edu.yale.cs.hadoopdb.benchmark.GrepTaskDB \
- > -pattern %wo% -output padraig -hadoop.config.file HadoopDB
    .xml
```

Приблизительный результат:

Листинг 5.51 Тестирование

```
Line 1 $ java -cp $CLASSPATH:hadoopdb.jar edu.yale.cs.hadoopdb.
    benchmark.GrepTaskDB \
- > -pattern %wo% -output padraig -hadoop.config.file HadoopDB
    .xml
- 14.08.2010 19:08:48 edu.yale.cs.hadoopdb.exec.DBJobBase
    initConf
```

```
- INFO: SELECT key1, field FROM grep WHERE field LIKE '%%wo%%'
    ;
5 14.08.2010 19:08:48 org.apache.hadoop.metrics.jvm.JvmMetrics
    init
- INFO: Initializing JVM Metrics with processName=JobTracker,
    sessionId=
- 14.08.2010 19:08:48 org.apache.hadoop.mapred.JobClient
    configureCommandLineOptions
- WARNING: Use GenericOptionsParser for parsing the arguments.
- Applications should implement Tool for the same.
10 14.08.2010 19:08:48 org.apache.hadoop.mapred.JobClient
    monitorAndPrintJob
- INFO: Running job: job_local_0001
- 14.08.2010 19:08:48 edu.yale.cs.hadoopdb.connector.
    AbstractDBRecordReader getConnection
- INFO: Data locality failed for leo-pgsql
- 14.08.2010 19:08:48 edu.yale.cs.hadoopdb.connector.
    AbstractDBRecordReader getConnection
15 INFO: Task from leo-pgsql is connecting to chunk 0 on host
    localhost with
- db url jdbc:postgresql://localhost:5434/grep0
- 14.08.2010 19:08:48 org.apache.hadoop.mapred.MapTask
    runOldMapper
- INFO: numReduceTasks: 0
- 14.08.2010 19:08:48 edu.yale.cs.hadoopdb.connector.
    AbstractDBRecordReader close
20 INFO: DB times (ms): connection = 104, query execution = 20,
    row retrieval = 79
- 14.08.2010 19:08:48 edu.yale.cs.hadoopdb.connector.
    AbstractDBRecordReader close
- INFO: Rows retrieved = 3
- 14.08.2010 19:08:48 org.apache.hadoop.mapred.Task done
- INFO: Task:attempt_local_0001_m_000000_0 is done. And is in
    the process of commiting
25 14.08.2010 19:08:48 org.apache.hadoop.mapred.
    LocalJobRunner$Job statusUpdate
- INFO:
- 14.08.2010 19:08:48 org.apache.hadoop.mapred.Task commit
- INFO: Task attempt_local_0001_m_000000_0 is allowed to
    commit now
- 14.08.2010 19:08:48 org.apache.hadoop.mapred.
    FileOutputCommitter commitTask
30 INFO: Saved output of task 'attempt_local_0001_m_000000_0'
    to file:/home/leo/padraig
- 14.08.2010 19:08:48 org.apache.hadoop.mapred.
    LocalJobRunner$Job statusUpdate
- INFO:
```

```
- 14.08.2010 19:08:48 org.apache.hadoop.mapred.Task sendDone
- INFO: Task 'attempt_local_0001_m_000000_0' done.
35 14.08.2010 19:08:49 org.apache.hadoop.mapred.JobClient
     monitorAndPrintJob
- INFO:  map 100% reduce 0%
- 14.08.2010 19:08:49 org.apache.hadoop.mapred.JobClient
     monitorAndPrintJob
- INFO: Job complete: job_local_0001
- 14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
40 INFO: Counters: 6
- 14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
- INFO:   FileSystemCounters
- 14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
- INFO:     FILE_BYTES_READ=141370
45 14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
- INFO:     FILE_BYTES_WRITTEN=153336
- 14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
- INFO:   Map-Reduce Framework
- 14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
50 INFO:     Map input records=3
- 14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
- INFO:     Spilled Records=0
- 14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
- INFO:     Map input bytes=3
55 14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
- INFO:     Map output records=3
- 14.08.2010 19:08:49 edu.yale.cs.hadoopdb.exec.DBJobBase run
- INFO:
- JOB TIME : 1828 ms.
```

Результат сохранен в HDFS, в папке padraig:

Листинг 5.52 Тестирование

```
Line 1 $ cd padraig
- $ cat part-00000
- some data
```

Проверим данные в PostgreSQL:

Листинг 5.53 Тестирование

```
Line 1 psql> select * from grep where field like '%wo%';
- +---------------------------------+------------------+
- | key1                            | field
- |
5 +---------------------------------+------------------+
- some data
-
- 1 rows in set (0.00 sec)
```

```
-
10 psql>
```

Значения идентичны. Все работает как требуется.
Проведем еще один тест. Добавим данные в PostgreSQL:

Листинг 5.54 Тестирование

```
Line 1 psql> INSERT into grep(key1, field) VALUES('I am live!', '
    Maybe');
- psql> INSERT into grep(key1, field) VALUES('I am live!', '
    Maybewqe');
- psql> INSERT into grep(key1, field) VALUES('I am live!', '
    Maybewqesad');
- psql> INSERT into grep(key1, field) VALUES(':)', 'May cool
    string!');
```

Теперь проверим через HadoopDB:

Листинг 5.55 Тестирование

```
Line 1 $ java -cp $CLASSPATH:hadoopdb.jar edu.yale.cs.hadoopdb.
    benchmark.GrepTaskDB -pattern %May% -output padraig -
    hadoopdb.config.file /opt/hadoop/conf/HadoopDB.xml
- padraig
- 01.11.2010 23:14:45 edu.yale.cs.hadoopdb.exec.DBJobBase
    initConf
- INFO: SELECT key1, field FROM grep WHERE field LIKE '%%May%%
    ';
5 01.11.2010 23:14:46 org.apache.hadoop.metrics.jvm.JvmMetrics
     init
- INFO: Initializing JVM Metrics with processName=JobTracker,
    sessionId=
- 01.11.2010 23:14:46 org.apache.hadoop.mapred.JobClient
    configureCommandLineOptions
- WARNING: Use GenericOptionsParser for parsing the arguments.
     Applications should implement Tool for the same.
- 01.11.2010 23:14:46 org.apache.hadoop.mapred.JobClient
    monitorAndPrintJob
10 INFO: Running job: job_local_0001
- 01.11.2010 23:14:46 edu.yale.cs.hadoopdb.connector.
    AbstractDBRecordReader getConnection
- INFO: Data locality failed for leo-pgsql
- 01.11.2010 23:14:46 edu.yale.cs.hadoopdb.connector.
    AbstractDBRecordReader getConnection
- INFO: Task from leo-pgsql is connecting to chunk 0 on host
    localhost with db url jdbc:postgresql://localhost:5434/
    grep0
15 01.11.2010 23:14:47 org.apache.hadoop.mapred.MapTask
    runOldMapper
```

```
- INFO: numReduceTasks: 0
- 01.11.2010 23:14:47 edu.yale.cs.hadoopdb.connector.
    AbstractDBRecordReader close
- INFO: DB times (ms): connection = 181, query execution = 22,
     row retrieval  = 96
- 01.11.2010 23:14:47 edu.yale.cs.hadoopdb.connector.
    AbstractDBRecordReader close
20 INFO: Rows retrieved = 4
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.Task done
- INFO: Task:attempt_local_0001_m_000000_0 is done. And is in
    the process of commiting
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.
    LocalJobRunner$Job statusUpdate
- INFO:
25 01.11.2010 23:14:47 org.apache.hadoop.mapred.Task commit
- INFO: Task attempt_local_0001_m_000000_0 is allowed to
    commit now
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.
    FileOutputCommitter commitTask
- INFO: Saved output of task 'attempt_local_0001_m_000000_0'
    to file:/home/hadoop/padraig
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.
    LocalJobRunner$Job statusUpdate
30 INFO:
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.Task sendDone
- INFO: Task 'attempt_local_0001_m_000000_0' done.
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.JobClient
    monitorAndPrintJob
- INFO:  map 100% reduce 0%
35 01.11.2010 23:14:47 org.apache.hadoop.mapred.JobClient
    monitorAndPrintJob
- INFO: Job complete: job_local_0001
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.Counters log
- INFO: Counters: 6
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.Counters log
40 INFO:   FileSystemCounters
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.Counters log
- INFO:     FILE_BYTES_READ=141345
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.Counters log
- INFO:     FILE_BYTES_WRITTEN=153291
45 01.11.2010 23:14:47 org.apache.hadoop.mapred.Counters log
- INFO:   Map-Reduce Framework
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.Counters log
- INFO:     Map input records=4
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.Counters log
50 INFO:     Spilled Records=0
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.Counters log
```

```
- INFO:     Map input bytes=4
- 01.11.2010 23:14:47 org.apache.hadoop.mapred.Counters log
- INFO:     Map output records=4
55 01.11.2010 23:14:47 edu.yale.cs.hadoopdb.exec.DBJobBase run
- INFO:
- JOB TIME : 2332 ms.
```

Как паттерн поиска я задал «May». В логах можно увидеть как производится поиск. На выходе получаем:

Листинг 5.56 Тестирование

```
Line 1 $ cd padraig
- $ cat part-00000
- I am live!     Maybe
- I am live!     Maybewqe
5 I am live!     Maybewqesad
- :)     May cool string!
```

В упрощенной системе с одним кластером PostgreSQL не понятно ради чего такие сложности. Но если к HadoopDB подключить более одного кластера PostgreSQL, то данной методикой возможно работать с данными PostgreSQL, объединенных в shared-nothing кластер.

Более подробно по HadoopDB можно почитать по данной ссылке hadoopdb.sourceforge.net.

Заключение

В данной статье не показывается, как работать с Hive, как более проще работать с HadoopDB. Эта книга не сможет учесть все, что требуется для работы с Hadoop. Назначение этой главы — дать основу для работы с Hadoop и HaddopDB.

HadoopDB не заменяет Hadoop. Эти системы сосуществуют, позволяя аналитику выбирать соответствующие средства в зависимости от имеющихся данных и задач.

HadoopDB может приблизиться в отношении производительности к параллельным системам баз данных, обеспечивая при этом отказоустойчивость и возможность использования в неоднородной среде при тех же правилах лицензирования, что и Hadoop. Хотя производительность HadoopDB, вообще говоря, ниже производительности параллельных систем баз данных, во многом это объясняется тем, что в PostgreSQL таблицы хранятся не по столбцам, и тем, что в PostgreSQL не использовалось сжатие данных. Кроме того, Hadoop и Hive — это сравнительно молодые проекты с открытыми кодами.

В HadoopDB применяется некоторый гибрид подходов параллельных СУБД и Hadoop к анализу данных, позволяющий достичь производительности и эффективности параллельных систем баз данных, обеспечивая

при этом масштабируемость, отказоустойчивость и гибкость систем, основанных на MapReduce. Способность HadoopDB к прямому включению Hadoop и программного обеспечения СУБД с открытыми исходными текстами (без изменения кода) делает HadoopDB особенно пригодной для выполнения крупномасштабного анализа данных в будущих рабочих нагрузках.

5.5 Заключение

В данной главе рассмотрены лишь базовые настройки кластеров БД. Про кластеры PostgreSQL потребуется написать отдельную книгу, чтобы рассмотреть все шаги с установкой, настройкой и работой кластеров. Надеюсь, что несмотря на это, информация будет полезна многим читателям.

6

PgPool-II

Имеется способ сделать
лучше — найди его.

Томас Алва Эдисон

6.1 Введение

pgpool-II — это прослойка, работающая между серверами PostgreSQL и клиентами СУБД PostgreSQL. Она предоставляет следующие функции:

- Объединение соединений

  Pgpool-II сохраняет соединения с серверами PostgreSQL и использует их повторно в случае если новое соединение устанавливается с теми же параметрами (т.е. имя пользователя, база данных, версия протокола). Это уменьшает накладные расходы на соединения и увеличивает производительность системы в целом;
- Репликация

  Pgpool-II может управлять множеством серверов PostgreSQL. Использование функции репликации данных позволяет создание резервной копии данных в реальном времени на 2 или более физических дисков, так что сервис может продолжать работать без остановки серверов в случае выхода из строя диска;
- Балансировка нагрузки

  Если база данных реплицируется, то выполнение запроса SELECT на любом из серверов вернет одинаковый результат. pgpool-II использует преимущество функции репликации для уменьшения нагрузки на каждый из серверов PostgreSQL распределяя запросы SELECT на несколько серверов, тем самым увеличивая производительность системы вцелом. В лучшем случае производительность возрастает пропорционально числу серверов PostgreSQL. Балансировка нагрузки лучше всего работает в случае когда много пользователей выполняют много запросов в одно и то же время.

- Ограничение лишних соединений

  Существует ограничение максимального числа одновременных соединений с PostgreSQL, при превышении которого новые соединения отклоняются. Установка максимального числа соединений, в то же время, увеличивает потребление ресурсов и снижает производительность системы. pgpool-II также имеет ограничение на максимальное число соединений, но «лишние» соединения будут поставлены в очередь вместо немедленного возврата ошибки.
- Параллельные запросы

  Используя функцию параллельных запросов можно разнести данные на множество серверов, благодаря чему запрос может быть выполнен на всех серверах одновременно для уменьшения общего времени выполнения. Параллельные запросы работают лучше всего при поиске в больших объемах данных.

Pgpool-II общается по протоколу бэкенда и фронтенда PostgreSQL и располагается между ними. Таким образом, приложение базы данных (фронтенд) считает что pgpool-II — настоящий сервер PostgreSQL, а сервер (бэкенд) видит pgpool-II как одного из своих клиентов. Поскольку pgpool-II прозрачен как для сервера, так и для клиента, существующие приложения, работающие с базой данных, могут использоваться с pgpool-II практически без изменений в исходном коде.

Оригинал руководства доступен по адресу pgpool.projects.pgfoundry.org.

6.2 Давайте начнем!

Перед тем как использовать репликацию или параллельные запросы мы должны научиться устанавливать и настраивать pgpool-II и узлы базы данных.

Установка pgpool-II

Установка pgpool-II очень проста. В каталоге, в который вы распаковали архив с исходными текстами, выполните следующие команды.

Листинг 6.1 Установка pgpool-II

```
Line 1  $ ./configure
    -   $ make
    -   $ make install
```

Скрипт configure собирает информацию о вашей системе и использует ее в процедуре компиляции. Вы можете указать параметры в командной строке скрипта configure чтобы изменить его поведение по умолчанию,

такие, например, как каталог установки. pgpool-II по умолчанию будет установлен в каталог /usr/local.

Команда make скомпилирует исходный код, а make install установит исполняемые файлы. У вас должно быть право на запись в каталог установки.

Обратите внимание: для работы pgpool-II необходима библиотека libpq для PostgreSQL 7.4 или более поздней версии (3 версия протокола). Если скрипт configure выдает следующее сообщение об ошибке, возможно не установлена библиотека libpq или она не 3 версии.

Листинг 6.2 Установка pgpool-II

```
Line 1  configure: error: libpq is not installed or libpq is old
```

Если библиотека 3 версии, но указанное выше сообщение все-таки выдается, ваша библиотека libpq, вероятно, не распознается скриптом configure.

Скрипт configure ищет библиотеку libpq начиная от каталога /usr/local/pgsql. Если вы установили PostgreSQL в каталог отличный от /usr/local/pgsql используйте параметры командной строки --with-pgsql или --with-pgsql-includedir вместе с параметром --with-pgsql-libdir при запуске скрипта configure.

Во многих Linux системах pgpool-II может находиться в репозитории пакетов. Для Ubuntu Linux, например, достаточно будет выполнить:

Листинг 6.3 Установка pgpool-II

```
Line 1  $ sudo aptitude install pgpool2
```

Файлы конфигурации

Параметры конфигурации pgpool-II хранятся в файле pgpool.conf. Формат файла: одна пара «параметр = значение» в строке. При установке pgpool-II автоматически создается файл pgpool.conf.sample. Мы рекомендуем скопировать его в файл pgpool.conf, а затем отредактировать по вашему желанию.

Листинг 6.4 Файлы конфигурации

```
Line 1  $ cp /usr/local/etc/pgpool.conf.sample /usr/local/etc/pgpool
        .conf
```

pgpool-II принимает соединения только с localhost на порт 9999. Если вы хотите принимать соединения с других хостов, установите для параметра listen_addresses значение «*».

Листинг 6.5 Файлы конфигурации

```
Line 1  listen_addresses = 'localhost'
    -   port = 9999
```

Мы будем использовать параметры по умолчанию в этом руководстве. В Ubuntu Linux конфиг находится /etc/pgpool.conf.

Настройка команд PCP

У pgpool-II есть интерфейс для административных целей — получить информацию об узлах базы данных, остановить pgpool-II и т.д. — по сети. Чтобы использовать команды PCP, необходима идентификация пользователя. Эта идентификация отличается от идентификации пользователей в PostgreSQL. Имя пользователя и пароль нужно указывать в файле pcp.conf. В этом файле имя пользователя и пароль указываются как пара значений, разделенных двоеточием (:). Одна пара в строке. Пароли зашифрованы в формате хэша md5.

Листинг 6.6 Настройка команд PCP

```
Line 1  postgres:e8a48653851e28c69d0506508fb27fc5
```

При установке pgpool-II автоматически создается файл pcp.conf.sample. Я рекомендуем скопировать его в файл pcp.conf и отредактировать.

Листинг 6.7 Настройка команд PCP

```
Line 1  $ cp /usr/local/etc/pcp.conf.sample /usr/local/etc/pcp.conf
```

В Ubuntu Linux файл находится /etc/pcp.conf.

Для того чтобы зашифровать ваш пароль в формате хэша md5 используете команду pg_md5, которая устанавливается как один из исполняемых файлов pgpool-II. pg_md5 принимает текст в параметре командной строки и отображает текст его md5 хэша.

Например, укажите «postgres» в качестве параметра командной строки и pg_md5 выведет текст хэша md5 в стандартный поток вывода.

Листинг 6.8 Настройка команд PCP

```
Line 1  $ /usr/bin/pg_md5 postgres
    -   e8a48653851e28c69d0506508fb27fc5
```

Команды PCP выполняются по сети, так что в файле pgpool.conf должен быть указан номер порта в параметре pcp_port.

Мы будем использовать значение по умолчанию для параметра pcp_port 9898 в этом руководстве.

Листинг 6.9 Настройка команд PCP

```
Line 1  pcp_port = 9898
```

Подготовка узлов баз данных

Теперь нам нужно настроить серверы бэкендов PostgreSQL для pgpool-II. Эти серверы могут быть размещены на одном хосте с pgpool-II или на отдельных машинах. Если вы решите разместить серверы на том же хосте, для всех серверов должны быть установлены разные номера портов. Если серверы размещены на отдельных машинах, они должны быть настроены так чтобы могли принимать сетевые соединения от pgpool-II.

В этом руководстве мы разместили три сервера в рамках одного хоста вместе с pgpool-II и присвоили им номера портов 5432, 5433, 5434 соответственно. Для настройки pgpool-II отредактируйте файл pgpool.conf как показано ниже.

Листинг 6.10 Подготовка узлов баз данных

```
backend_hostname0 = 'localhost'
backend_port0 = 5432
backend_weight0 = 1
backend_hostname1 = 'localhost'
backend_port1 = 5433
backend_weight1 = 1
backend_hostname2 = 'localhost'
backend_port2 = 5434
backend_weight2 = 1
```

В параметрах backend_hostname, backend_port, backend_weight укажите имя хоста узла базы данных, номер порта и коэффициент для балансировки нагрузки. В конце имени каждого параметра должен быть указан идентификатор узла путем добавления положительного целого числа начиная с 0 (т.е. 0, 1, 2).

Параметры backend_weight все равны 1, что означает что запросы SELECT равномерно распределены по трем серверам.

Запуск/Остановка pgpool-II

Для старта pgpool-II выполните в терминале следующую команду.

Листинг 6.11 Запуск

```
$ pgpool
```

Указанная выше команда, однако, не печатает протокол своей работы потому что pgpool отсоединяется от терминала. Если вы хотите показать протокол работы pgpool, укажите параметр -n в командной строке при запуске pgpool. Pgpool-II будет запущен как процесс не-демон и терминал не будет отсоединен.

Листинг 6.12 Запуск

```
$ pgpool -n &
```

Протокол работы будет печататься на терминал, так что рекомендуемые для использования параметры командной строки, например, такие.

Листинг 6.13 Запуск

```
Line 1  $ pgpool -n -d > /tmp/pgpool.log 2>&1 &
```

Параметр -d включает генерацию отладочных сообщений.

Указанная выше команда постоянно добавляет выводимый протокол работы в файл /tmp/pgpool.log. Если вам нужно ротировать файлы протоколов, передавайте протоколы внешней команде, у которой есть функция ротации протоколов. Вам поможет, например, cronolog.

Листинг 6.14 Запуск

```
Line 1  $ pgpool -n 2>&1 | /usr/sbin/cronolog \
-       --hardlink=/var/log/pgsql/pgpool.log \
-       '/var/log/pgsql/%Y-%m-%d-pgpool.log' &
```

Чтобы остановить процесс pgpool-II, выполните следующую команду.

Листинг 6.15 Остановка

```
Line 1  $ pgpool stop
```

Если какие-то из клиентов все еще присоединены, pgpool-II ждет пока они не отсоединятся и потом завершает свою работу. Если вы хотите завершить pgpool-II насильно, используйте вместо этой следующую команду.

Листинг 6.16 Остановка

```
Line 1  $ pgpool -m fast stop
```

6.3 Ваша первая репликация

Репликация включает копирование одних и тех же данных на множество узлов базы данных.

В этом разделе мы будем использовать три узла базы данных, которые мы уже установили в разделе «6.2 Давайте начнем!», и проведем вас шаг за шагом к созданию системы репликации базы данных. Пример данных для репликации будет сгенерирован программой для тестирования pgbench.

Настройка репликации

Чтобы включить функцию репликации базы данных установите значение true для параметра replication_mode в файле pgpool.conf.

Листинг 6.17 Настройка репликации

```
Line 1  replication_mode = true
```

Если параметр replication_mode равен true, pgpool-II будет отправлять копию принятого запроса на все узлы базы данных.

Если параметр load_balance_mode равен true, pgpool-II будет распределять запросы SELECT между узлами базы данных.

Листинг 6.18 Настройка репликации

```
Line 1  load_balance_mode = true
```

В этом разделе мы включили оба параметра replication_mode и load_balance_mode.

Проверка репликации

Для отражения изменений, сделанных в файле pgpool.conf, pgpool-II должен быть перезапущен. Пожалуйста обращайтесь к разделу «6.2 Запуск/Остановка pgpool-II».

После настройки pgpool.conf и перезапуска pgpool-II, давайте проверим репликацию в действии и посмотрим все ли работает хорошо.

Сначала нам нужно создать базу данных, которую будем реплицировать. Назовем ее bench_replication. Эту базу данных нужно создать на всех узлах. Используйте команду createdb через pgpool-II и база данных будет создана на всех узлах.

Листинг 6.19 Проверка репликации

```
Line 1  $ createdb -p 9999 bench_replication
```

Затем мы запустим pgbench с параметром -i. Параметр -i инициализирует базу данных предопределенными таблицами и данными в них.

Листинг 6.20 Проверка репликации

```
Line 1  $ pgbench -i -p 9999 bench_replication
```

Указанная ниже таблица содержит сводную информацию о таблицах и данных, которые будут созданы при помощи pgbench -i. Если на всех узлах базы данных перечисленные таблицы и данные были созданы, репликация работает корректно.

Имя таблицы	Число строк
branches	1
tellers	10
accounts	100000
history	0

Для проверки указанной выше информации на всех узлах используем простой скрипт на shell. Приведенный ниже скрипт покажет число строк в таблицах branches, tellers, accounts и history на всех узлах базы данных (5432, 5433, 5434).

Листинг 6.21 Проверка репликации

```
for port in 5432 5433 5434; do
    echo $port
    for table_name in branches tellers accounts history;
  do
        echo $table_name
        psql -c "SELECT count(*) FROM $table_name" -p \
        $port bench_replication
    done
done
```

6.4 Ваш первый параллельный запрос

Данные из разных диапазонов сохраняются на двух или более узлах базы данных параллельным запросом. Это называется распределением (часто используется без перевода термин partitioning прим. переводчика). Более того, одни и те же данные на двух и более узлах базы данных могут быть воспроизведены с использованием распределения.

Чтобы включить параллельные запросы в pgpool-II вы должны установить еще одну базу данных, называемую «Системной базой данных» («System Database») (далее будем называть ее SystemDB).

SystemDB хранит определяемые пользователем правила, определяющие какие данные будут сохраняться на каких узлах базы данных. Также SystemDB используется чтобы объединить результаты возвращенные узлами базы данных посредством dblink.

В этом разделе мы будем использовать три узла базы данных, которые мы установили в разделе «6.2 Давайте начнем!», и проведем вас шаг за шагом к созданию системы баз данных с параллельными запросами. Для создания примера данных мы снова будем использовать pgbench.

Настройка параллельного запроса

Чтобы включить функцию выполнения параллельных запросов установите для параметра parallel_mode значение true в файле pgpool.conf.

Листинг 6.22 Настройка параллельного запроса

```
parallel_mode = true
```

Установка параметра parallel_mode равным true не запустит параллельные запросы автоматически. Для этого pgpool-II нужна SystemDB и правила определяющие как распределять данные по узлам базы данных.

Также SystemDB использует dblink для создания соединений с pgpool-II. Таким образом, нужно установить значение параметра listen_addresses таким образом чтобы pgpool-II принимал эти соединения.

Листинг 6.23 Настройка параллельного запроса

```
Line 1  listen_addresses = '*'
```

Внимание: Репликация не реализована для таблиц, которые распределяются посредством параллельных запросов и, в то же время, репликация может быть успешно осуществлена. Вместе с тем, из-за того что набор хранимых данных отличается при параллельных запросах и при репликации, база данных «bench_replication», созданная в разделе «6.3 Ваша первая репликация» не может быть повторно использована.

Листинг 6.24 Настройка параллельного запроса

```
Line 1  replication_mode = true
    -   load_balance_mode = false
```

или

Листинг 6.25 Настройка параллельного запроса

```
Line 1  replication_mode = false
    -   load_balance_mode = true
```

В этом разделе мы установим параметры parallel_mode и load_balance_mode равными true, listen_addresses равным «*», replication_mode равным false.

Настройка SystemDB

В основном, нет отличий между простой и системной базами данных. Однако, в системной базе данных определяется функция dblink и присутствует таблица, в которой хранятся правила распределения данных. Таблицу dist_def необходимо определять. Более того, один из узлов базы данных может хранить системную базу данных, а pgpool-II может использоваться для распределения нагрузки каскадным подключеним.

В этом разделе мы создадим SystemDB на узле с портом 5432. Далее приведен список параметров конфигурации для SystemDB

Листинг 6.26 Настройка SystemDB

```
Line 1  system_db_hostname = 'localhost'
    -   system_db_port = 5432
    -   system_db_dbname = 'pgpool'
    -   system_db_schema = 'pgpool_catalog'
    5   system_db_user = 'pgpool'
    -   system_db_password = ''
```

На самом деле, указанные выше параметры являются параметрами по умолчанию в файле pgpool.conf. Теперь мы должны создать пользователя с именем «pgpool» и базу данных с именем «pgpool» и владельцем «pgpool».

Листинг 6.27 Настройка SystemDB

```
Line 1  $ createuser -p 5432 pgpool
    -   $ createdb -p 5432 -O pgpool pgpool
```

Установка dblink

Далее мы должны установить dblink в базу данных «pgpool». dblink — один из инструментов включенных в каталог contrib исходного кода PostgreSQL.

Для установки dblink на вашей системе выполните следующие команды.

Листинг 6.28 Установка dblink

```
Line 1  $ USE_PGXS=1 make -C contrib/dblink
    -   $ USE_PGXS=1 make -C contrib/dblink install
```

После того как dblink был установлен в вашей системе мы добавим функции dblink в базу данных «pgpool». Если PostgreSQL установлен в каталог /usr/local/pgsql, dblink.sql (файл с определениями функций) должен быть установлен в каталог /usr/local/pgsql/share/contrib. Теперь выполним следующую команду для добавления функций dblink.

Листинг 6.29 Установка dblink

```
Line 1  $ psql -f /usr/local/pgsql/share/contrib/dblink.sql -p 5432
        pgpool
```

Создание таблицы dist_def

Следующим шагом мы создадим таблицу с именем «dist_def», в которой будут храниться правила распределения данных. Поскольку pgpool-II уже был установлен, файл с именем system_db.sql должен быть установлен в /usr/local/share/system_db.sql (имейте в виду, что это учебное руководство и мы использовали для установки каталог по умолчанию — /usr/local). Файл system_db.sql содержит директивы для создания специальных таблиц, включая и таблицу «dist_def». Выполните следующую команду для создания таблицы «dist_def».

Листинг 6.30 Создание таблицы dist_def

```
Line 1  $ psql -f /usr/local/share/system_db.sql -p 5432 -U pgpool
        pgpool
```

Все таблицы в файле system_db.sql, включая «dist_def», создаются в схеме «pgpool_catalog». Если вы установили параметр system_db_schema на использование другой схемы, вам нужно, соответственно, отредактировать файл system_db.sql.

Описание таблицы «dist_def» выглядит так как показано ниже. Имя таблицы не должно измениться.

Листинг 6.31 Создание таблицы dist_def

```
Line 1  CREATE TABLE pgpool_catalog.dist_def (
        dbname text, -- имя базы данных
        schema_name text, -- имя схемы
        table_name text, -- имя таблицы
5       col_name text NOT NULL CHECK (col_name = ANY (col_list))
      ,
        -- столбец ключ для распределения данных
        col_list text[] NOT NULL, -- список имен столбцов
        type_list text[] NOT NULL, -- список типов столбцов
        dist_def_func text NOT NULL,
10      -- имя функции распределения данных
        PRIMARY KEY (dbname, schema_name, table_name)
    );
```

Записи, хранимые в таблице «dist_def», могут быть двух типов:

- Правило Распределения Данных (col_name, dist_def_func);
- Мета-информация о таблицах (dbname, schema_name, table_name, col_list, type_list).

Правило распределения данных определяет как будут распределены данные на конкретный узел базы данных. Данные будут распределены в зависимости от значения столбца col_name. dist_def_func — это функция, которая принимает значение col_name в качестве агрумента и возвращает целое число, которое соответствует идентификатору узла базы данных, на котором должны быть сохранены данные.

Мета-информация используется для того чтобы переписывать запросы. Параллельный запрос должен переписывать исходные запросы так чтобы результаты, возвращаемые узлами-бэкендами, могли быть объединены в единый результат.

Создание таблицы replicate_def

В случае если указана таблица, для которой производится репликация в выражение SQL, использующее зарегистрированную в «dist_def» таблицу путем объединения таблиц, информация о таблице, для которой необходимо производить репликацию, предварительно регистрируется в таблице с именем «replicate_def». Таблица «replicate_def» уже была создана при обработке файла system_db.sql во время создания таблицы «dist_def». Таблица «replicate_def» описана так как показано ниже.

Листинг 6.32 Создание таблицы replicate_def

```
Line 1  CREATE TABLE pgpool_catalog.replicate_def (
        dbname text, -- имя базы данных
        schema_name text, -- имя схемы
```

```
    table_name text, -- имя таблицы
    col_list text[] NOT NULL, -- список имен столбцов
    type_list text[] NOT NULL, -- список типов столбцов
    PRIMARY KEY (dbname, schema_name, table_name)
);
```

Установка правил распределения данных

В этом учебном руководстве мы определим правила распределения данных, созданных программой pgbench, на три узла базы данных. Тестовые данные будут созданы командой pgbench -i -s 3 (т.е. масштабный коэффициент равен 3). Для этого раздела мы создадим новую базу данных с именем «bench_parallel».

В каталоге sample исходного кода pgpool-II вы можете найти файл dist_def_pgbench.sql. Мы будем использовать этот файл с примером для создания правил распределения для pgbench. Выполните следующую команду в каталоге с распакованным исходным кодом pgpool-II.

Листинг 6.33 Установка правил распределения данных

```
Line 1 $ psql -f sample/dist_def_pgbench.sql -p 5432 pgpool
```

Ниже представлено описание файла dist_def_pgbench.sql.

В файле dist_def_pgbench.sql мы добавляем одну строку в таблицу «dist_def». Это функция распределения данных для таблицы accounts. В качестве столбца-ключа указан столбец aid.

Листинг 6.34 Установка правил распределения данных

```
Line 1 INSERT INTO pgpool_catalog.dist_def VALUES (
    'bench_parallel',
    'public',
    'accounts',
    'aid',
    ARRAY['aid', 'bid', 'abalance', 'filler'],
    ARRAY['integer', 'integer', 'integer',
    'character(84)'],
    'pgpool_catalog.dist_def_accounts'
);
```

Теперь мы должны создать функцию распределения данных для таблицы accounts. Заметим, что вы можете использовать одну и ту же функцию для разных таблиц. Также вы можете создавать функции с использованием языков отличных от SQL (например, PL/pgSQL, PL/Tcl, и т.д.).

Таблица accounts в момент инициализации данных хранит значение масштабного коэффициента равное 3, значения столбца aid от 1 до 300000. Функция создана таким образом что данные равномерно распределяются по трем узлам базы данных.

Следующая SQL-функция будет возвращать число узлов базы данных.

Листинг 6.35 Установка правил распределения данных

```
CREATE OR REPLACE FUNCTION
pgpool_catalog.dist_def_branches(anyelement)
RETURNS integer AS $$
    SELECT CASE WHEN $1 > 0 AND $1 <= 1 THEN 0
        WHEN $1 > 1 AND $1 <= 2 THEN 1
        ELSE 2
    END;
$$ LANGUAGE sql;
```

Установка правил репликации

Правило репликации — это то что определяет какие таблицы должны быть использованы для выполнения репликации.

Здесь это сделано при помощи pgbench с зарегистрированными таблицами «branches» и «tellers».

Как результат, стало возможно создание таблицы «accounts» и выполнение запросов, использующих таблицы «branches» и «tellers».

Листинг 6.36 Установка правил репликации

```
INSERT INTO pgpool_catalog.replicate_def VALUES (
    'bench_parallel',
    'public',
    'branches',
    ARRAY['bid', 'bbalance', 'filler'],
    ARRAY['integer', 'integer', 'character(88)']
);

INSERT INTO pgpool_catalog.replicate_def VALUES (
    'bench_parallel',
    'public',
    'tellers',
    ARRAY['tid', 'bid', 'tbalance', 'filler'],
    ARRAY['integer', 'integer', 'integer', 'character(84)']
);
```

Подготовленный файл Replicate_def_pgbench.sql находится в каталоге sample. Команда psql запускается с указанием пути к исходному коду, определяющему правила репликации, например, как показано ниже.

Листинг 6.37 Установка правил репликации

```
$ psql -f sample/replicate_def_pgbench.sql -p 5432 pgpool
```

Проверка параллельного запроса

Для отражения изменений, сделанных в файле pgpool.conf, pgpool-II должен быть перезапущен. Пожалуйста, обращайтесь к разделу «6.2 Запуск/Остановка pgpool-II».

После настройки pgpool.conf и перезапуска pgpool-II давайте проверим хорошо ли работают параллельные запросы.

Сначала нам нужно создать базу данных, которая будет распределена. Мы назовем ее «bench_parallel». Эту базу данных нужно создать на всех узлах. Используйте команду createdb посредством pgpool-II и база данных будет создана на всех узлах.

Листинг 6.38 Проверка параллельного запроса

```
Line 1 $ createdb -p 9999 bench_parallel
```

Затем запустим pgbench с параметрами -i -s 3. Параметр -i инициализирует базу данных предопределенными таблицами и данными. Параметр -s указывает масштабный коэффициент для инициализации.

Листинг 6.39 Проверка параллельного запроса

```
Line 1 $ pgbench -i -s 3 -p 9999 bench_parallel
```

Созданные таблицы и данные в них показаны в разделе «6.4 Установка правил распределения данных».

Один из способов проверить корректно ли были распределены данные — выполнить запрос SELECT посредством pgpool-II и напрямую на бэкендах и сравнить результаты. Если все настроено правильно база данных «bench_parallel» должна быть распределена как показано ниже.

Имя таблицы	Число строк
branches	3
tellers	30
accounts	300000
history	0

Для проверки указанной выше информации на всех узлах и посредством pgpool-II используем простой скрипт на shell. Приведенный ниже скрипт покажет минимальное и максимальное значение в таблице accounts используя для соединения порты 5432, 5433, 5434 и 9999.

Листинг 6.40 Проверка параллельного запроса

```
Line 1 for port in 5432 5433 5434i 9999; do
- >     echo $port
- >     psql -c "SELECT min(aid), max(aid) FROM accounts" \
- >     -p $port bench_parallel
5 > done
```

6.5 Master-slave режим

Этот режим предназначен для использования pgpool-II с другой репликацией (например Slony-I, Londiste). Информация про БД указывается как для репликации. master_slave_mode и load_balance_mode устанавливается в true. pgpool-II будет посылать запросы INSERT/UPDATE/DELETE на Master DB (1 в списке), а SELECT — использовать балансировку нагрузки, если это возможно.

При этом, DDL и DML для временной таблицы может быть выполнен только на мастере. Если нужен SELECT только на мастере, то для этого нужно использовать комментарий /*NO LOAD BALANCE*/ перед SELECT.

В Master/Slave режиме replication_mode должен быть установлен false, а master_slave_mode — true.

Streaming Replication (Потоковая репликация)

В master-slave режиме с потоковой репликацией, если мастер или слейв упал, возможно использовать отказоустоичивый функционал внутри pgpool-II. Автоматически отключив упавший нод PostgreSQL, pgpool-II переключится на следующий слейв как на новый мастер (при падении мастера), или останется работать на мастере (при падении слейва). В потоковой репликации, когда слейв становится мастером, требуется создать триггер файл (который указан в recovery.conf, параметр trigger_file), чтобы PostgreSQL перешел из режима восстановления в нормальный. Для этого можно создать небольшой скрипт:

Листинг 6.41 Скрипт выполняется при падении нода PostgreSQL

```
Line 1  #! /bin/sh
  -     # Failover command for streming replication.
  -     # This script assumes that DB node 0 is primary, and 1 is
            standby.
  -     #
  5     # If standby goes down, does nothing. If primary goes down,
            create a
  -     # trigger file so that standby take over primary node.
  -     #
  -     # Arguments: $1: failed node id. $2: new master hostname. $3
            : path to
  -     # trigger file.
 10
  -     failed_node=$1
  -     new_master=$2
  -     trigger_file=$3
  -
 15     # Do nothing if standby goes down.
  -     if [ $failed_node = 1 ]; then
```

```
-         exit 0;
- fi
-
20 # Create trigger file.
- /usr/bin/ssh -T $new_master /bin/touch $trigger_file
-
- exit 0;
```

Работает он просто: если падает слейв — скрипт ничего не выполняет, при падении мастера — создает триггер файл на новом мастере. Сохраним этот файл под именем «failover_stream.sh» и в pgpool.conf добавим:

Листинг 6.42 Что выполнять при падении нода

```
Line 1 failover_command = '/path_to_script/failover_stream.sh %d %H
       /tmp/trigger_file'
```

где /tmp/trigger_file — триггер файл, указаный в конфиге recovery.conf.

Теперь, если мастер СУБД упадет, слейв будет переключен из режима восстановления в обычный и сможет принимать запросы на запись.

6.6 Онлайн восстановление

pgpool-II в режиме репликации может синхронизировать базы данных и добавлять их как ноды к pgpool. Называется это «онлайн восстановление». Этот метод также может быть использован когда нужно вернуть в репликацию упавший нод базы данных.

Вся процедура выполняется в два задания. Несколько секунд или минут клиент может ждать подключения к pgpool, в то время как восстанавливается узел базы данных. Онлайн восстановление состоит из следующих шагов:

- CHECKPOINT;
- Первый этап восстановления;
- Ждем, пока все клиенты не отключатся;
- CHECKPOINT;
- Второй этап восстановления;
- Запуск postmaster (выполнить pgpool_remote_start);
- Восстанавливаем нод СУБД.

Для работы онлайн восстановления потребуется указать следующие параметры:

- backend_data_directory Каталог данных определенного PostgreSQL кластера;
- recovery_user Имя пользователя PostgreSQL;
- recovery_password Пароль пользователя PostgreSQL;

- recovery_1st_stage_command Параметр указывает команду для первого этапа онлайн восстановления. Файл с командами должен быть помещен в каталог данных СУБД кластера из соображений безопасности. Например, если recovery_1st_stage_command = 'some_script', то pgpool-II выполнит $PGDATA/some_script. Обратите внимание, что pgpool-II принимает подключения и запросы в то время как выполняется recovery_1st_stage;
- recovery_2nd_stage_command Параметр указывает команду для второго этапа онлайн восстановления. Файл с командами должен быть помещен в каталог данных СУБД кластера из-за проблем безопасности. Например, если recovery_2st_stage_command = 'some_script', то pgpool-II выполнит $PGDATA/some_script. Обратите внимание, что pgpool-II НЕ принимает подключения и запросы в то время как выполняется recovery_2st_stage. Таким образом, pgpool-II будет ждать пока все клиенты не закроют подключения.

Streaming Replication (Потоковая репликация)

В master-slave режиме с потоковой репликацией, онлайн восстановление — отличное средство вернуть назад упавший нод PostgreSQL. Вернуть возможно только слейв ноды, таким методом не восстановить упавший мастер. Для восстановления мастера потребуется остановить все PostgreSQL ноды и pgpool-II (для восстановления из резервной копии мастера).

Для настройки онлайн восстановления нам потребуется:

- Установить recovery_user. Обычно это «postgres».

Листинг 6.43 recovery_user

```
Line 1  recovery_user = 'postgres'
```

- Установить recovery_password для recovery_user для подключения к СУБД.

Листинг 6.44 recovery_password

```
Line 1  recovery_password = 'some_password'
```

- Настроить recovery_1st_stage_command. Для этого создадим скрипт basebackup.sh и положим его в папку с данными мастера ($PGDATA), установив ему права на выполнение. Содержание скрипта:

Листинг 6.45 basebackup.sh

```
Line 1  #! /bin/sh
  -     # Recovery script for streaming replication.
  -     # This script assumes that DB node 0 is primary, and 1
            is standby.
  -     #
  5     datadir=$1
```

```
- desthost=$2
- destdir=$3
-
- psql -c "SELECT pg_start_backup('Streaming Replication',
      true)" postgres
10
- rsync -C -a --delete -e ssh --exclude postgresql.conf --
    exclude postmaster.pid \
- --exclude postmaster.opts --exclude pg_log --exclude
    pg_xlog \
- --exclude recovery.conf $datadir/ $desthost:$destdir/
-
15 ssh -T localhost mv $destdir/recovery.done $destdir/
    recovery.conf
-
- psql -c "SELECT pg_stop_backup()" postgres
```

При восстановления слейва, скрипт запускает бэкап мастера и через rsync передает данные с мастера на слейв. Для этого необходимо настроить SSH так, чтобы recovery_user мог заходить с мастера на слейв без пароля.

Далее добавим скрипт на выполнение для первого этапа онлайн востановления:

Листинг 6.46 recovery_1st_stage_command

```
Line 1 recovery_1st_stage_command = 'basebackup.sh'
```

- Оставляем recovery_2nd_stage_command пустым. После успешного выполнения первого этапа онлайн восстановления, разницу в данных, что успели записатся во время работы скрипта basebackup.sh, слейв заберет через WAL файлы с мастера.
- Устанавливаем C и SQL функции для работы онлайн востановления на каждый нод СУБД.

Листинг 6.47 Устанавливаем C и SQL функции

```
Line 1 $ cd pgpool-II-x.x.x/sql/pgpool-recovery
- $ make
- $ make install
- $ psql -f pgpool-recovery.sql template1
```

Вот и все. Теперь возможно использовать pcp_recovery_node для онлайн восстановления упавших слейвов.

6.7 Заключение

PgPool-II — прекрасное средство, которое нужно применять при масштабировании PostgreSQL.

7

Мультиплексоры соединений

> Если сразу успеха не добились, пытайтесь снова и снова. Затем оставьте эти попытки. Какой смысл глупо упорствовать?
>
> Уильям Клод Филдс

7.1 Введение

Мультиплексоры соединений (программы для создания пула соединений) позволяют уменьшить накладные расходы на базу данных, в случае, когда огромное количество физических соединений ведет к падению производительности PostgreSQL. Это особенно важно на Windows, когда система ограничивает большое количество соединений. Это также важно для веб-приложений, где количество соединений может быть очень большим.

Вот список программ, которые создают пулы соединений:

- PgBouncer
- Pgpool

7.2 PgBouncer

Это мультиплексор соединений для PostgreSQL от компании Skype. Существуют три режима управления.

- Session Pooling — наиболее «вежливый» режим. При начале сессии клиенту выделяется соединение с сервером; оно приписано ему в течение всей сессии и возвращается в пул только после отсоединения клиента;

- Transaction Pooling — клиент владеет соединением с бакендом только в течение транзакции. Когда PgBouncer замечает, что транзакция завершилась, он возвращает соединение назад в пул;
- Statement Pooling — наиболее агрессивный режим. Соединение с бакендом возвращается назад в пул сразу после завершения запроса. Транзакции с несколькими запросами в этом режиме не разрешены, так как они гарантировано будут отменены. Также не работают подготовленные выражения (prepared statements) в этом режиме.

К достоинствам PgBouncer относится:

- малое потребление памяти (менее 2 КБ на соединение);
- отсутствие привязки к одному серверу баз данных;
- реконфигурация настроек без рестарта.

Базовая утилита запускается так:

Листинг 7.1 PgBouncer

```
Line 1 $ pgbouncer [-d][-R][-v][-u user] <pgbouncer.ini>
```

Простой пример для конфига:

Листинг 7.2 PgBouncer

```
Line 1 [databases]
- template1 = host=127.0.0.1 port=5432 dbname=template1
- [pgbouncer]
- listen_port = 6543
5 listen_addr = 127.0.0.1
- auth_type = md5
- auth_file = userlist.txt
- logfile = pgbouncer.log
- pidfile = pgbouncer.pid
10 admin_users = someuser
```

Нужно создать файл пользователей userlist.txt примерно такого содержания: "someuser" "same_password_as_in_server"

Административный доступ из консоли к базе данных pgbouncer:

Листинг 7.3 PgBouncer

```
Line 1 $ psql -h 127.0.0.1 -p 6543 pgbouncer
```

Здесь можно получить различную статистическую информацию с помощью команды SHOW.

7.3 PgPool-II vs PgBouncer

Все очень просто. PgBouncer намного лучше работает с пулами соединений, чем PgPool-II. Если вам не нужны остальные возможности, которыми владеет PgPool-II (ведь пулы коннектов это мелочи к его функционалу), то конечно лучше использовать PgBouncer.

- PgBouncer потребляет меньше памяти, чем PgPool-II;
- у PgBouncer возможно настроить очередь соединений;
- в PgBouncer можно настраивать псевдо базы данных (на сервере они могут называтся по-другому).

Хотя некоторые используют PgBouncer и PgPool-II совместно.

8

Кэширование в PostgreSQL

> Чтобы что-то узнать, нужно уже что-то знать.
>
> Станислав Лем

8.1 Введение

Кэш или кеш — промежуточный буфер с быстрым доступом, содержащий информацию, которая может быть запрошена с наибольшей вероятностью. Кэширование SELECT запросов позволяет повысить производительность приложений и снизить нагрузку на PostgreSQL. Преимущества кэширования особенно заметны в случае с относительно маленькими таблицами, имеющими статические данные, например, справочными таблицами.

Многие СУБД могут кэшировать SQL запросы, и данная возможность идет у них, в основном, «из коробки». PostgreSQL не обладает подобным функционалом. Почему? Во-первых, мы теряем транзакционную чистоту происходящего в базе. Что это значит? Управление конкурентным доступом с помощью многоверсионности (MVCC — MultiVersion Concurrency Control) — один из механизмов обеспечения одновременного конкурентного доступа к БД, заключающийся в предоставлении каждому пользователю «снимка» БД, обладающего тем свойством, что вносимые данным пользователем изменения в БД невидимы другим пользователям до момента фиксации транзакции. Этот способ управления позволяет добиться того, что пишущие транзакции не блокируют читающих, и читающие транзакции не блокируют пишущих. При использовании кэширования, которому нет дела к транзакциям СУБД, «снимки» БД могут быть с неверными данными. Во-вторых, кеширование результатов запросов, в основном, должно происходить на стороне приложения, а не СУБД. В таком случае управление кэшированием может работать более гибко (включать и отключать его где потребуется для приложения), а СУБД будет заниматься своей непосредственной целью — хранением и обеспечение целостности данных.

Для организации кэширования существует два инструмента для PostgreSQL:

- Pgmemcache (с memcached)
- Pgpool-II (query cache)

8.2 Pgmemcache

Memcached — компьютерная программа, реализующая сервис кэширования данных в оперативной памяти на основе парадигмы распределенной хеш-таблицы. С помощью клиентской библиотеки позволяет кэшировать данные в оперативной памяти одного или нескольких из множества доступных серверов. Распределение реализуется путем сегментирования данных по значению хэша ключа по аналогии с сокетами хэш-таблицы. Клиентская библиотека используя ключ данных вычисляет хэш и использует его для выбора соответствующего сервера. Ситуация сбоя сервера трактуется как промах кэша, что позволяет повышать отказоустойчивость комплекса за счет наращивания количества memcached серверов и возможности производить их горячую замену.

Pgmemcache — это PostgreSQL API библиотека на основе libmemcached для взаимодействия с memcached. С помощью данной библиотеки PostgreSQL может записывать, считывать, искать и удалять данные из memcached. Посмотрим, что из себя представляет данный тип кэширования.

Установка

Во время написания этой главы была доступна 2.0.6 версия pgmemcache. Pgmemcache будет устанавливаться и настраиваться на PostgreSQL версии 9.2, операционная система — Ubuntu Server 12.04. Поскольку Pgmemcache идет как модуль, то потребуется PostgreSQL с PGXS (если уже не установлен, поскольку в сборках для Linux присутствует PGXS). Также потребуется memcached и libmemcached библиотека версии не ниже 0.38.

После скачивания и распаковки исходников, существует два варианта установки Pgmemcache:

- Установка из исходников

  Для этого достаточно выполнить в консоли:

  Листинг 8.1 Установка из исходников

```
Line 1  $ make
-       $ sudo make install
```

- Создание и установка deb пакета (для Debian, Ubuntu)

  Если у Вас на серверах стоит Debian или Ubuntu, то удобнее создать deb пакет нужной программы и распостранать его через собственный репозиторий на все сервера с PostgreSQL:

  Листинг 8.2 Создание и установка deb пакета

```
Line 1  $ sudo apt-get install libmemcached-dev postgresql-
          server-dev-9.2 libpq-dev devscripts yada flex bison
    -   $ make deb92
    -   # устанавливаем deb пакет
    -   $ sudo dpkg -i ../postgresql-pgmemcache-9.2*.deb
```

  Для версии 2.0.4 утилита yada выдавала ошибку при создании deb пакета со следующим текстом:

  Листинг 8.3 Создание и установка deb пакета

```
Line 1  Cannot recognize source name in 'debian/changelog' at /
          usr/bin/yada line 145, <CHANGELOG> line 1.
    -   make: *** [deb92] Ошибка 9
```

  Для устранения этой ошибки потребуется удалить первую строчку текста в «debian/changelog» в каталоге, котором происходит сборка:

  Листинг 8.4 Создание и установка deb пакета

```
Line 1  $ PostgreSQL: pgmemcache/debian/changelog,v 1.2
          2010/05/05 19:56:50 ormod Exp $ <---- удалить
    -   pgmemcache (2.0.4) unstable; urgency=low
    -
    -     * v2.0.4
```

  После устранения данной ошибки сборка deb-пакета должна пройти без проблем.

Настройка

После успешной установки Pgmemcache потребуется добавить во все базы данных (на которых вы хотите использовать Pgmemcache) функции для работы с этой библиотекой:

Листинг 8.5 Настройка

```
Line 1  % psql [mydbname] [pguser]
    -   [mydbname]=# CREATE EXTENSION pgmemcache;
    -   # или
    -   # BEGIN;
    5   # \i /usr/share/postgresql/9.2/contrib/pgmemcache.sql
    -   # COMMIT;
```

Теперь можно добавлять сервера memcached через memcache_server_add и работать с кэшем. Но есть одно но. Все сервера memcached придется задавать при каждом новом подключении к PostgreSQL. Это ограничение можно обойти, если настроить параметры в postgresql.conf файле:

- Добавить «pgmemcache» в shared_preload_libraries (автозагрузка библиотеки pgmemcache во время старта PostgreSQL);
- Добавить «pgmemcache» в custom_variable_classes (устанавливаем переменную для pgmemcache);
- Создаем pgmemcache.default_servers, указав в формате «host:port» (port - опционально) через запятую. Например:

Листинг 8.6 Настройка default_servers

```
Line 1  pgmemcache.default_servers = '127.0.0.1,
            192.168.0.20:11211' # подключили два сервера memcached
```

- Также можем настроить работу самой библиотеки pgmemcache через pgmemcache.default_behavior. Настройки соответствуют настрокам libmemcached. Например:

Листинг 8.7 Настройка pgmemcache

```
Line 1  pgmemcache.default_behavior='BINARY_PROTOCOL:1'
```

Теперь не требуется при подключении к PostgreSQL указывать сервера memcached.

Проверка

После успешной установки и настройки pgmemcache, становится доступен список команд для работы с memcached серверами.

Посмотрим работу в СУБД данных функций. Для начала получим информацию по memcached серверах:

Листинг 8.8 Проверка memcache_stats

```
Line 1  pgmemcache=# SELECT memcache_stats();
    -           memcache_stats
    -  ---------------------------
    -
    5   Server: 127.0.0.1 (11211)
    -   pid: 1116
    -   uptime: 70
    -   time: 1289598098
    -   version: 1.4.5
   10   pointer_size: 32
    -   rusage_user: 0.0
```

Таблица 8.1: Список команд pgmemcache

Команда	Описание
memcache_server_add('hostname:port'::TEXT) memcache_server_add('hostname'::TEXT)	Добавляет memcached сервер в список доступных серверов. Если порт не указан, по умолчанию используется 11211.
memcache_add(key::TEXT, value::TEXT, expire::TIMESTAMPTZ) memcache_add(key::TEXT, value::TEXT, expire::INTERVAL) memcache_add(key::TEXT, value::TEXT)	Добавляет ключ в кэш, если ключ не существует.
newval = memcache_decr(key::TEXT, decrement::INT4) newval = memcache_decr(key::TEXT)	Если ключ существует и является целым числом, происходит уменьшение его значения на указаное число (по умолчанию на единицу). Возвращает целое число после уменьшения.
memcache_delete(key::TEXT, hold_timer::INTERVAL) memcache_delete(key::TEXT)	Удаляет указанный ключ. Если указать таймер, то ключ с таким же названием может быть добавлен только после окончания таймера.
memcache_flush_all()	Очищает все данные на всех memcached серверах.
value = memcache_get(key::TEXT)	Выбирает ключ из кэша. Возвращает NULL, если ключ не существует, иначе — текстовую строку.
memcache_get_multi(keys::TEXT[]) memcache_get_multi(keys::BYTEA[])	Получает массив ключей из кэша. Возвращает список найденных записей в виде «ключ=значение».
newval = memcache_incr(key::TEXT, increment::INT4) newval = memcache_incr(key::TEXT)	Если ключ существует и является целым числом, происходит увеличение его значения на указаное число (по умолчанию на единицу). Возвращает целое число после увеличения.
memcache_replace(key::TEXT, value::TEXT, expire::TIMESTAMPTZ) memcache_replace(key::TEXT, value::TEXT, expire::INTERVAL) memcache_replace(key::TEXT, value::TEXT)	Заменяет значение для существующего ключа.
memcache_set(key::TEXT, value::TEXT, expire::TIMESTAMPTZ) memcache_set(key::TEXT, value::TEXT, expire::INTERVAL) memcache_set(key::TEXT, value::TEXT)	Создает ключ со значение. Если такой ключ существует — заменяет в нем значение на указаное.
stats = memcache_stats()	Возвращает статистику по всем серверам memcached.

```
- rusage_system: 0.24001
- curr_items: 0
- total_items: 0
15 bytes: 0
- curr_connections: 5
- total_connections: 7
- connection_structures: 6
- cmd_get: 0
```

```
 cmd_set: 0
 get_hits: 0
 get_misses: 0
 evictions: 0
 bytes_read: 20
 bytes_written: 782
 limit_maxbytes: 67108864
 threads: 4

(1 row)
```

Теперь сохраним данные в memcached и попробуем их забрать:

Листинг 8.9 Проверка

```
pgmemcache=# SELECT memcache_add('some_key', 'test_value');
 memcache_add
--------------
 t
(1 row)

pgmemcache=# SELECT memcache_get('some_key');
 memcache_get
--------------
 test_value
(1 row)
```

Можно также проверить работу счетчиков в memcached (данный функционал может пригодиться для создания последовательностей):

Листинг 8.10 Проверка

```
pgmemcache=# SELECT memcache_add('some_seq', '10');
 memcache_add
--------------
 t
(1 row)

pgmemcache=# SELECT memcache_incr('some_seq');
 memcache_incr
---------------
            11
(1 row)

pgmemcache=# SELECT memcache_incr('some_seq');
 memcache_incr
---------------
            12
(1 row)

```

```
pgmemcache=# SELECT memcache_incr('some_seq', 10);
 memcache_incr
---------------
            22
(1 row)

pgmemcache=# SELECT memcache_decr('some_seq');
 memcache_decr
---------------
            21
(1 row)

pgmemcache=# SELECT memcache_decr('some_seq');
 memcache_decr
---------------
            20
(1 row)

pgmemcache=# SELECT memcache_decr('some_seq', 6);
 memcache_decr
---------------
            14
(1 row)
```

Для работы с pgmemcache лучше создать функции и, если требуется, активировать эти функции через триггеры.

Например, наше приложение кэширует зашифрованые пароли пользователей в memcached (для более быстрого доступа), и нам требуется обновлять информацию в кэше, если она изменяется в СУБД. Создаем функцию:

Листинг 8.11 Функция для обновления данных в кэше

```
CREATE OR REPLACE FUNCTION auth_passwd_upd() RETURNS TRIGGER
    AS $$
        BEGIN
        IF OLD.passwd != NEW.passwd THEN
                PERFORM memcache_set('user_id_' || NEW.
    user_id || '_password', NEW.passwd);
        END IF;
        RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';
```

Активируем триггер для обновления таблицы пользователей:

Листинг 8.12 Триггер

```
CREATE TRIGGER auth_passwd_upd_trg AFTER UPDATE ON passwd
    FOR EACH ROW EXECUTE PROCEDURE auth_passwd_upd();
```

Но(!!!) данный пример транзакционно не безопасен — при отмене транзакции кэш не вернется на старое значение. Поэтому лучше очищать старые данные:

Листинг 8.13 Очистка ключа в кэше

```
Line 1 CREATE OR REPLACE FUNCTION auth_passwd_upd() RETURNS TRIGGER
       AS $$
  -  BEGIN
  -          IF OLD.passwd != NEW.passwd THEN
  -                  PERFORM memcache_delete('user_id_' || NEW.
       user_id || '_password');
  5          END IF;
  -          RETURN NEW;
  -  END;$$ LANGUAGE 'plpgsql';
```

Также нужен триггер на чистку кэша при удалении записи из СУБД:

Листинг 8.14 Триггер

```
Line 1 CREATE TRIGGER auth_passwd_del_trg AFTER DELETE ON passwd
       FOR EACH ROW EXECUTE PROCEDURE auth_passwd_upd();
```

Замечу от себя, что создавать кэш в memcached на кешированый пароль нового пользователя (или обновленного) лучше через приложение.

Заключение

PostgreSQL с помощью Pgmemcache библиотеки позволяет работать с memcached серверами, что может потребоваться в определенных случаях для кэширования данных напрямую с СУБД. Удобство данной библиотеки заключается в полном доступе к функциям memcached, но вот готовой реализации кэширование SQL запросов тут нет, и её придется дорабатывать вручную через функции и триггеры PostgreSQL.

8.3 Заключение

Кэширование в PostgreSQL может быть реализованно с помощью различных утилит. Это показывает отличную гибкость PostgreSQL, но, как я думаю, оптимальным решением является, чтобы кешированием занималось другое решение (ваше приложение, Varnish, другое).

9

Расширения

Гибкость ума может заменить красоту.

Стендаль

9.1 Введение

Один из главных плюсов PostgreSQL это возможность расширения его функционала с помощью расширений. В данной статье я затрону только самые интересные и популярные из существующих расширений.

9.2 PostGIS

Лицензия: Open Source
Ссылка: www.postgis.org
PostGIS добавляет поддержку для географических объектов в PostgreSQL. По сути PostGIS позволяет использовать PostgreSQL в качестве бэкэнда пространственной базы данных для геоинформационных систем (ГИС), так же, как ESRI SDE или пространственного расширения Oracle. PostGIS соответствует OpenGIS «Простые особенности. Спецификация для SQL» и был сертифицирован.

9.3 pgSphere

Лицензия: Open Source
Ссылка: pgsphere.projects.postgresql.org
pgSphere обеспечивает PostgreSQL сферическими типами данных, а также функциями и операторами для работы с ними. Используется для работы с географическими (может использоваться вместо PostGIS) или астрономическими типами данных.

9.4 HStore

Лицензия: Open Source

HStore – расширение, которое реализует тип данных для хранения ключ/значение в пределах одного значения в PostgreSQL (например, в одном текстовом поле). Это может быть полезно в различных ситуациях, таких как строки с многими атрибутами, которые редко вибираются, или полу-структурированные данные. Ключи и значения являются простыми текстовыми строками.

Пример использования

Для начала активируем расширение:

Листинг 9.1 Активация hstore

```
Line 1  # CREATE EXTENSION hstore;
```

Проверим работу расширения:

Листинг 9.2 Проверка hstore

```
Line 1  # SELECT 'a=>1,a=>2'::hstore;
    -     hstore
    -  ----------
    -   "a"=>"1"
    5  (1 row)
```

Как видно на листинге 9.2 ключи в hstore уникальны. Создадим таблицу и заполним её данными:

Листинг 9.3 Проверка hstore

```
Line 1  CREATE TABLE products (
    -     id serial PRIMARY KEY,
    -     name varchar,
    -     attributes hstore
    5  );
    -  INSERT INTO products (name, attributes)
    -  VALUES (
    -     'Geek Love: A Novel',
    -     'author     => "Katherine Dunn",
   10     pages      => 368,
    -     category   => fiction'
    -  ),
    -  (
    -    'Leica M9',
   15    'manufacturer  => Leica,
    -     type           => camera,
    -     megapixels     => 18,
    -     sensor         => "full-frame 35mm"'
```

```
),
( 'MacBook Air 11',
 'manufacturer  => Apple,
  type          => computer,
  ram           => 4GB,
  storage       => 256GB,
  processor     => "1.8 ghz Intel i7 duel core",
  weight        => 2.38lbs'
);
```

Теперь можно производить поиск по ключу:

Листинг 9.4 Поиск по ключу

```
# SELECT name, attributes->'pages' as page FROM products
   WHERE attributes ? 'pages';
        name         | page
--------------------+------
 Geek Love: A Novel | 368
(1 row)
```

Или по значению ключа:

Листинг 9.5 Поиск по значению ключа

```
# SELECT name, attributes->'manufacturer' as manufacturer
   FROM products WHERE attributes->'type' = 'computer';
       name       | manufacturer
----------------+--------------
 MacBook Air 11 | Apple
(1 row)
```

Создание индексов:

Листинг 9.6 Индексы

```
# CREATE INDEX products_hstore_index ON products USING GIST
   (attributes);
# CREATE INDEX products_hstore_index ON products USING GIN (
   attributes);
```

Можно также создавать индекс на ключ:

Листинг 9.7 Индекс на ключ

```
# CREATE INDEX product_manufacturer ON products ((products.
   attributes->'manufacturer'));
```

Заключение

HStore — расширение для удобного и индексируемого хранения слабоструктурированых данных в PostgreSQL.

9.5 PLV8

Лицензия: Open Source
Ссылка: code.google.com/p/plv8js

PLV8 является расширением, которое предоставляет PostgreSQL процедурный язык с движком V8 JavaScript. С помощью этого расширения можно писать в PostgreSQL JavaScript функции, которые можно вызывать из SQL.

Скорость работы

V8 компилирует JavaScript код непосредственно в машинный код и с помощью этого достигается высокая скорость работы. Для примера расмотрим расчет числа Фибоначчи. Вот функция написана на plpgsql:

Листинг 9.8 Фибоначчи на plpgsql

```
Line 1  CREATE OR REPLACE FUNCTION
    -   psqlfib(n int) RETURNS int AS $$
    -    BEGIN
    -        IF n < 2 THEN
    5            RETURN n;
    -        END IF;
    -        RETURN psqlfib(n-1) + psqlfib(n-2);
    -    END;
    -   $$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

Замерим скорость её работы:

Листинг 9.9 Скорость расчета числа Фибоначчи на plpgsql

```
Line 1  SELECT n, psqlfib(n) FROM generate_series(0,30,5) as n;
    -    n   | psqlfib
    -   -----+---------
    -      0 |       0
    5      5 |       5
    -     10 |      55
    -     15 |     610
    -     20 |    6765
    -     25 |   75025
    10    30 |  832040
    -   (7 rows)
    -
    -   Time: 16003,257 ms
```

Теперь сделаем то же самое, но с использованием PLV8:

Листинг 9.10 Фибоначчи на plv8

```
Line 1  CREATE OR REPLACE FUNCTION
    -   fib(n int) RETURNS int as $$
```

```
-
-   function fib(n) {
5     return n<2 ? n : fib(n-1) + fib(n-2)
-   }
-   return fib(n)
-
- $$ LANGUAGE plv8 IMMUTABLE STRICT;
```

Замерим скорость работы:

Листинг 9.11 Скорость расчета числа Фибоначчи на plv8

```
Line 1 SELECT n, fib(n) FROM generate_series(0,30,5) as n;
-  n  |  fib
- ----+--------
-   0 |      0
5   5 |      5
-  10 |     55
-  15 |    610
-  20 |   6765
-  25 |  75025
10  30 | 832040
- (7 rows)
-
- Time: 59,254 ms
```

Как видим PLV8 приблизительно в 270 (16003.257/59.254) раз быстрее plpgsql. Можно ускорить работу расчета чисел Фибоначи на PLV8 за счет кеширования:

Листинг 9.12 Фибоначчи на plv8

```
Line 1 CREATE OR REPLACE FUNCTION
- fib1(n int) RETURNS int as $$
-   var memo = {0: 0, 1: 1};
-   function fib(n) {
5     if(!(n in memo))
-       memo[n] = fib(n-1) + fib(n-2)
-     return memo[n]
-   }
-   return fib(n);
10 $$ LANGUAGE plv8 IMMUTABLE STRICT;
```

Замерим скорость работы:

Листинг 9.13 Скорость расчета числа Фибоначчи на plv8

```
Line 1 SELECT n, fib1(n) FROM generate_series(0,30,5) as n;
-  n  |  fib1
- ----+--------
-   0 |      0
5   5 |      5
```

```
 10 |      55
 15 |     610
 20 |    6765
 25 |   75025
 30 | 832040
(7 rows)

Time: 0,766 ms
```

Теперь расчет на PLV8 приблизительно в несколько раз быстрее, чем на plpgsql.

Использование

Одним из полезных применений PLV8 может быть создание на базе PostgreSQL документоориентированного хранилища. Для хранения неструктурированных данных можно использовать hstore, но у него есть свои недостатки:

- нет вложенности
- все данные (ключ и значение по ключу) это строка

Для хранения данных многие документоориентированные базы данных используют JSON (MongoDB, CouchDB, Couchbase и т.д.). Для этого, начиная с PostgreSQL 9.2, добавлен тип данных JSON. Такой тип можно добавить для PostgreSQL 9.1 и ниже используя PLV8 и DOMAIN:

Листинг 9.14 Создание типа JSON

```
CREATE OR REPLACE FUNCTION
valid_json(json text)
RETURNS BOOLEAN AS $$
  try {
    JSON.parse(json); return true;
  } catch(e) {
    return false;
  }
$$ LANGUAGE plv8 IMMUTABLE STRICT;

CREATE DOMAIN json AS TEXT
CHECK(valid_json(VALUE));
```

Функция «valid_json» используется для проверки JSON данных. Пример использования:

Листинг 9.15 Проверка JSON

```
$ INSERT INTO members
VALUES('not good json');
ERROR:  value for domain json
```

```
violates check constraint "json_check"
$ INSERT INTO members
VALUES('{"good": "json", "is": true}');
INSERT 0 1
$ select * from members;
             profile
------------------------------
  {"good": "json", "is": true}
(1 row)
```

Рассмотрим пример использования JSON для хранения данных и PLV8 для их поиска. Для начала создадим таблицу и заполним её данными:

Листинг 9.16 Таблица с JSON полем

```
$ CREATE TABLE members ( id SERIAL, profile json );
$ SELECT count(*) FROM members;
  count
---------
 1000000
(1 row)

Time: 201.109 ms
```

В «profile» поле мы записали приблизительно такую структуру JSON:

Листинг 9.17 JSON структура

```
{                                   +
  "name": "Litzy Satterfield",      +
  "age": 24,                        +
  "siblings": 2,                    +
  "faculty": false,                 +
  "numbers": [                      +
    {                               +
      "type":   "work",             +
      "number": "684.573.3783 x368"+
    },                              +
    {                               +
      "type":   "home",             +
      "number": "625.112.6081"      +
    }                               +
  ]                                 +
}
```

Теперь создадим функцию для вывода значения по ключу из JSON (в данном случае ожидаем цифру):

Листинг 9.18 Функция для JSON

```
CREATE OR REPLACE FUNCTION get_numeric(json_raw json, key
    text)
```

```
- RETURNS numeric AS $$
-   var o = JSON.parse(json_raw);
-   return o[key];
5 $$ LANGUAGE plv8 IMMUTABLE STRICT;
```

Теперь мы можем произвести поиск по таблице, фильтруя по значениям ключей «age», «siblings» или другим числовым полям:

Листинг 9.19 Поиск по данным JSON

```
Line 1 $ SELECT * FROM members WHERE get_numeric(profile, 'age') =
     36;
- Time: 9340.142 ms
- $ SELECT * FROM members WHERE get_numeric(profile, 'siblings
     ') = 1;
- Time: 14320.032 ms
```

Поиск работает, но скорость очень маленькая. Чтобы увеличить скорость, нужно создать функциональные индексы:

Листинг 9.20 Создание индексов

```
Line 1 $ CREATE INDEX member_age ON members (get_numeric(profile, '
     age'));
- $ CREATE INDEX member_siblings ON members (get_numeric(
     profile, 'siblings'));
```

С индексами скорость поиска по JSON станет достаточно высокая:

Листинг 9.21 Поиск по данным JSON с индексами

```
Line 1 $ SELECT * FROM members WHERE get_numeric(profile, 'age') =
     36;
- Time: 57.429 ms
- $ SELECT * FROM members WHERE get_numeric(profile, 'siblings
     ') = 1;
- Time: 65.136 ms
5 $ SELECT count(*) from members where  get_numeric(profile, '
     age') = 26 and get_numeric(profile, 'siblings') = 1;
- Time: 106.492 ms
```

Получилось отличное документоориентированное хранилище из PostgreSQL.

PLV8 позволяет использовать некоторые JavaScript библиотеки внутри PostgreSQL. Вот пример рендера Mustache темплейтов:

Листинг 9.22 Функция для рендера Mustache темплейтов

```
Line 1 CREATE OR REPLACE FUNCTION mustache(template text, view json
     )
- RETURNS text as $$
-   // ...400 lines of mustache....js
-   return Mustache.render(template, JSON.parse(view))
5 $$ LANGUAGE plv8 IMMUTABLE STRICT;
```

Листинг 9.23 Рендер темплейтов

```
$ SELECT mustache(
    'hello {{#things}}{{.}} {{/things}}:) {{#data}}{{key}}{{/
     data}}',
    '{"things": ["world", "from", "postgresql"], "data": {"key
     ": "and me"}}'
);
                  mustache
--------------------------------------
  hello world from postgresql :) and me
(1 row)

Time: 0.837 ms
```

Этот пример показывает как можно использовать PLV8. В действительности рендерить Mustache в PostgreSQL не лучшая идея.

Заключение

PLV8 расширение предоставляет PostgreSQL процедурный язык с движком V8 JavaScript, с помощью которого можно работать с JavaScript билиотеками, индексировать JSON данные и использовать его как более быстрый язык.

9.6 Pg_repack

Лицензия: Open Source
Ссылка: reorg.github.io/pg_repack/

Таблицы в PostgreSQL представлены в виде страниц, размером 8 КБ, в которых размещены записи. Когда одна страница полностью заполняется записями, к таблице добавляется новая страница. При удалалени записей с помощью DELETE или изменении с помощью UPDATE, место где были старые записи не может быть повторно использовано сразу же. Для этого процесс очистки autovacuum, или команда VACUUM, пробегает по изменённым страницам и помечает такое место как свободное, после чего новые записи могут спокойно записываться в это место. Если autovacuum не справляется, например в результате активного изменения большего количества данных или просто из-за плохих настроек, то к таблице будут излишне добавляться новые страницы по мере поступления новых записей. И даже после того как очистка дойдёт до наших удалённых записей, новые страницы останутся. Получается что таблица становится более разряженной в плане плотности записей. Это и называется эффектом раздувания таблиц, table bloat.

Процедура очистки, autovacuum или VACUUM, может уменьшить размер таблицы убрав полностью пустые страницы, но только при условии

что они находятся в самом конце таблицы. Чтобы максимально уменьшить таблицу в PostgreSQL есть VACUUM FULL или CLUSTER, но оба эти способа требуют «exclusively locks» на таблицу (то есть в это время с таблицы нельзя ни читать, ни писать), что далеко не всегда является подходящим решением.

Для решение подобных проблем существует расширение pg_repack. Это расширение позволяет сделать VACUUM FULL или CLUSTER команды без блокировки таблицы. Для чистки таблицы pg_repack создает точную её копию в «repack» схеме базы данных (ваша база по умолчанию работает в «public» схеме) и сортирует строки в этой таблице. После переноса данных и чиски мусора, утилита меняет схему у таблиц. Для чистки индексов утилита создает новые индексы с другими именами, а по выполнению работы меняет их на первоначальные. Для выполнения всех этих работ потребуется дополнительное место на диске (например, если у вас 100ГБ данных, и из них 40 ГБ - распухание таблиц или индексов, то вам потребуется 100 ГБ + (100 ГБ - 40 ГБ) = 160 ГБ на диске минимум). Для проверки «распухания» таблиц и индексов в вашей базе можно воспользоватся советом из раздела «12.2 Размер распухания (bloat) таблиц и индексов в базе данных».

Существует ряд ограничений в работе pg_repack:

- Не может очистить временные таблицы;
- Не может очистить таблицы с использованием GIST индексов;
- Нельзя выполнять DDL (Data Definition Language) на таблице во время работы.

Примеры

Выполнить команду CLUSTER всех кластерных таблиц и VACUUM FULL для всех не кластерных таблиц в test базе данных:

Скачать Листинг

```
Line 1 $ pg_repack test
```

Выполните команду VACUUM FULL на foo и bar таблицах в test базе данных (кластеризация таблиц игнорируется):

Скачать Листинг

```
Line 1 $ pg_repack --no-order --table foo --table bar test
```

Переместить все индексы таблицы foo в неймспейс tbs:

Скачать Листинг

```
Line 1 $ pg_repack -d test --table foo --only-indexes --tablespace
    tbs
```

Заключение

Pg_repack — расширение, которое может помочь в больбе с «table bloat» в PostgreSQL «на лету».

9.7 Pg_prewarm

Лицензия: Open Source

Модуль pg_prewarm обеспечивает удобный способ загрузки данных обьектов (таблиц, индексов, прочего) в буферный кэш PostgreSQL или операционной системы. Данный модуль добавлен в contrib начиная с PostgreSQL 9.4.

Для начала нужно установить модуль:

Скачать Листинг

```
Line 1  $ CREATE EXTENSION pg_prewarm;
```

После уставновки доступна функция pg_prewarm:

Скачать Листинг

```
Line 1  $ SELECT pg_prewarm('pgbench_accounts');
    -    pg_prewarm
    -   ------------
    -          4082
    5   (1 row)
```

Первый аргумент — обьект, который требуется предварительно загружать в память. Второй аргумент — «режим» загрузки в память, который может содержать такие варианты:

- prefetch — чтение файла ядром системы в асинхронном режиме (в фоновом режиме). Это позволяет положить содержимое файла в кэше ядра системы. Но этот режим не работает на всех платформах;
- read — результат похож на prefetch, но делается синхронно (а значит медленнее). Работает на всех платформах;
- buffer — в этом режиме данные будут грузится в PostgreSQL shared_buffers. Этот режим используется по умолчанию.

Третий аргумент называется «fork». Про него не нужно беспокоиться. Возможные значения: «main» (используется по умолчанию), «fsm», «vm».

Четвертый и пятый аргументы указывают диапазон страниц для загрузки данных. По умолчанию загружается весь обьект в память, но можно решить, например, загрузить только последние 1000 страниц:

Скачать Листинг

```
Line 1  $ SELECT pg_prewarm(
    -       'pgbench_accounts',
```

```
-     first_block := (
-         SELECT pg_relation_size('pgbench_accounts') /
    current_setting('block_size')::int4 - 1000
5     )
- );
```

Заключение

Pg_prewarm — расширение, которое позволяет предварительно загрузить («подогреть») данные в буферной кэш PostgreSQL или операционной системы.

9.8 Smlar

Лицензия: Open Source
Ссылка: sigaev.ru

Поиск похожести в больших базах данных является важным вопросом в настоящее время для таких систем как блоги (похожие статьи), интернет-магазины (похожие продукты), хостинг изображений (похожие изображения, поиск дубликатов изображений) и т.д. PostgreSQL позволяет сделать такой поиск более легким. Прежде всего, необходимо понять, как мы будем вычислять сходство двух объектов.

Похожесть

Любой объект может быть описан как список характеристик. Например, статья в блоге может быть описана тегами, продукт в интернет-магазине может быть описан размером, весом, цветом и т.д. Это означает, что для каждого объекта можно создать цифровую подпись — массив чисел, описывающих объект (отпечатки пальцев, n-grams). То есть нужно создать массив из цифр для описания каждого объекта. Что делать дальше?

Расчет похожести

Есть несколько методов вычисления похожести сигнатур объектов. Прежде всего, легенда для расчетов:

- N_a, N_b — количество уникальных элементов в массивах;
- N_u — количество уникальных элементов при объединении массивов;
- N_i — количество уникальных элементов при пересечение массивов.

Один из простейших расчетов похожести двух объектов - количество уникальных элементов при пересечение массивов делить на количество уникальных элементов в двух массивах:

$$S(A,B) = \frac{N_i}{(N_a + N_b)} \tag{9.1}$$

или проще

$$S(A,B) = \frac{N_i}{N_u} \tag{9.2}$$

Преимущества:

- Легко понять;
- Скорость расчета: $N * \log N$;
- Хорошо работает на похожих и больших N_a и N_b.

Также похожесть можно рассчитана по формуле косинусов:

$$S(A,B) = \frac{N_i}{\sqrt{N_a * N_b}} \tag{9.3}$$

Преимущества:

- Скорость расчета: $N * \log N$;
- Отлично работает на больших N.

Но у обоих этих методов есть общие проблемы:

- Если элементов мало, то разброс похожести не велик;
- Глобальная статистика: частые элементы ведут к тому, что вес ниже;
- Спамеры и недобросовестные пользователи. Один «залетевший дятел» разрушит цивилизацию - алгоритм перестанет работать на Вас.

Для избежания этих проблем можно воспользоваться TF/IDF метрикой:

$$S(A,B) = \frac{\sum_{i<N_a, j<N_b, A_i=B_j} TF_i * TF_j}{\sqrt{\sum_{i<N_a} TF_i^2 * \sum_{j<N_b} TF_j^2}} \tag{9.4}$$

где инвертированный вес элемента в коллекции:

$$IDF_{element} = \log\left(\frac{N_{objects}}{N_{objects\ with\ element}} + 1\right) \tag{9.5}$$

и вес элемента в массиве:

$$TF_{element} = IDF_{element} * N_{occurrences} \tag{9.6}$$

Не пугайтесь! Все эти алгоритмы встроены в smlar расширение, учить (или даже глубоко понимать) их не нужно. Главное понимать, что для TF/IDF метрики требуются вспомогательная таблица для хранения данных, по сравнению с другими простыми метриками.

Smlar

Перейдем к практике. Олег Бартунов и Теодор Сигаев разработали PostgreSQL расширение smlar, которое предоставляет несколько методов для расчета похожести массивов (все встроенные типы данных поддерживаются) и оператор для расчета похожести с поддержкой индекса на базе GIST и GIN. Для начала установим это расширение (PostgreSQL уже должен быть установлен):

Листинг 9.24 Установка smlar

```
$ git clone git://sigaev.ru/smlar
$ cd smlar
$ USE_PGXS=1 make && make install
```

В PostgreSQL 9.2 и выше это расширение должно встать без проблем, для PostgreSQL 9.1 и ниже вам нужно сделать небольшое исправление в исходниках. В файле «smlar_guc.c» в строке 214 сделайте изменение с:

Листинг 9.25 Фикс для 9.1 и ниже

```
set_config_option("smlar.threshold", buf, PGC_USERSET,
    PGC_S_SESSION ,GUC_ACTION_SET, true, 0);
```

на (нужно убрать последний аргумент):

Листинг 9.26 Фикс для 9.1 и ниже

```
set_config_option("smlar.threshold", buf, PGC_USERSET,
    PGC_S_SESSION ,GUC_ACTION_SET, true);
```

Теперь проверим расширение:

Листинг 9.27 Проверка smlar

```
$ psql
psql (9.2.1)
Type "help" for help.

test=# CREATE EXTENSION smlar;
CREATE EXTENSION

test=# SELECT smlar('{1,4,6}'::int[], '{5,4,6}'::int[]);
  smlar
----------
 0.666667
(1 row)

test=# SELECT smlar('{1,4,6}'::int[], '{5,4,6}'::int[], 'N.i
    / sqrt(N.a * N.b)' );
  smlar
----------
 0.666667
(1 row)
```

Расширение установлено успешно, если у Вас такой же вывод в консоли. Методы, которые предоставляет это расширение:

- float4 smlar(anyarray, anyarray) — вычисляет похожесть двух массивов. Массивы должны быть одного типа;
- float4 smlar(anyarray, anyarray, bool useIntersect) — вычисляет похожесть двух массивы составных типов. Составной тип выглядит следующим образом:

Листинг 9.28 Составной тип

```
Line 1 CREATE TYPE type_name AS (element_name anytype,
       weight_name float4);
```

useIntersect параметр для использования пересекающихся элементов в знаменателе;

- float4 smlar(anyarray a, anyarray b, text formula) — вычисляет похожесть двух массивов по данной формуле, массивы должны быть того же типа. Доступные переменные в формуле:
  - N.i — количество общих элементов в обоих массивов (пересечение);
  - N.a — количество уникальных элементов первого массива;
  - N.b — количество уникальных элементов второго массива;
- anyarray % anyarray — возвращает истину, если похожесть массивов больше, чем указанный предел. Предел указывается в конфиге PostgreSQL:

Листинг 9.29 Smlar предел

```
Line 1 custom_variable_classes = 'smlar'
  - smlar.threshold = 0.8 # предел от 0 до 1
```

Также в конфиге можно указать дополнительные настройки для smlar:

Листинг 9.30 Smlar настройки

```
Line 1 custom_variable_classes = 'smlar'
  - smlar.threshold = 0.8 # предел от 0 до 1
  - smlar.type = 'cosine' # по какой формуле производить расчет
      похожести: cosine, tfidf, overlap
  - smlar.stattable = 'stat' # Имя таблицы для хранения
      статистики при работе по формуле tfidf
```

Более подробно можно прочитать в README этого расширения.

GiST и GIN индексы поддерживаются для оператора %.

Пример: поиск дубликатов картинок

Рассмотрим простой пример поиска дубликатов картинок. Алгоритм помогает найти похожие изображения, которые, например, незначительно отличаются (изображение обесцветили, добавили водяные знаки, пропустили через фильтры). Но, поскольку точность мала, то у алгоритма есть и позитивная сторона — скорость работы. Как можно определить, что картинки похожи? Самый простой метод — сравнивать попиксельно два изображения. Но скорость такой работы будет не велика на больших разрешениях. Тем более, такой метод не учитывает, что могли изменять уровень света, насыщенность и прочие характеристики изображения. Нам нужно создать сигнатуру для картинок в виде массива цифр:

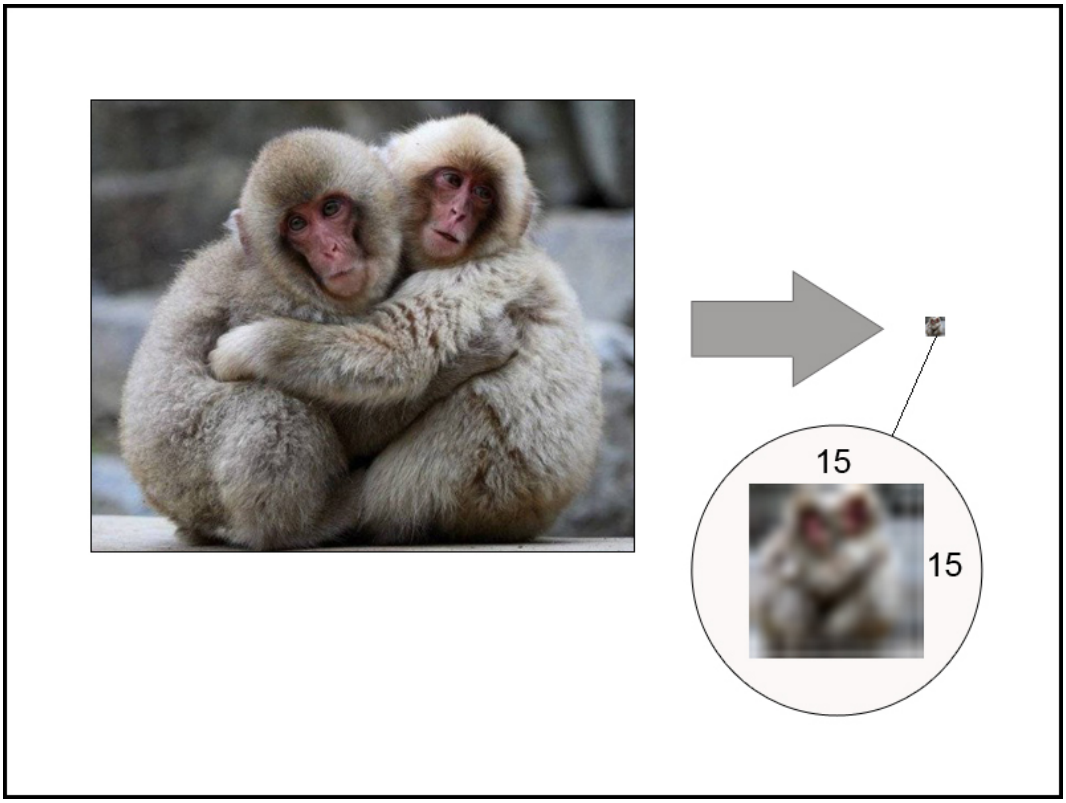

Рис. 9.1: Пиксельная матрица

- Создаем пиксельную матрицу к изображению (изменения размера изображения к требуемоему размеру пиксельной матрице), например 15X15 пикселей(Рис. 9.1);
- Рассчитаем интенсивность каждого пикселя (интенсивность вычисляется по формуле $0.299 * \text{красный} + 0.587 * \text{зеленый} + 0.114 * \text{синий}$). Интенсивность поможет нам находить похожие изображения, не обращая внимание на используемые цвета в них;

- Узнаем отношение интенсивности каждого пикселя к среднему значению интенсивности по всей матрице(Рис. 9.2);
- Генерируем уникальное число для каждой ячейки (отношение интенсивности + координаты ячейки);
- Сигнатура для картинки готова.

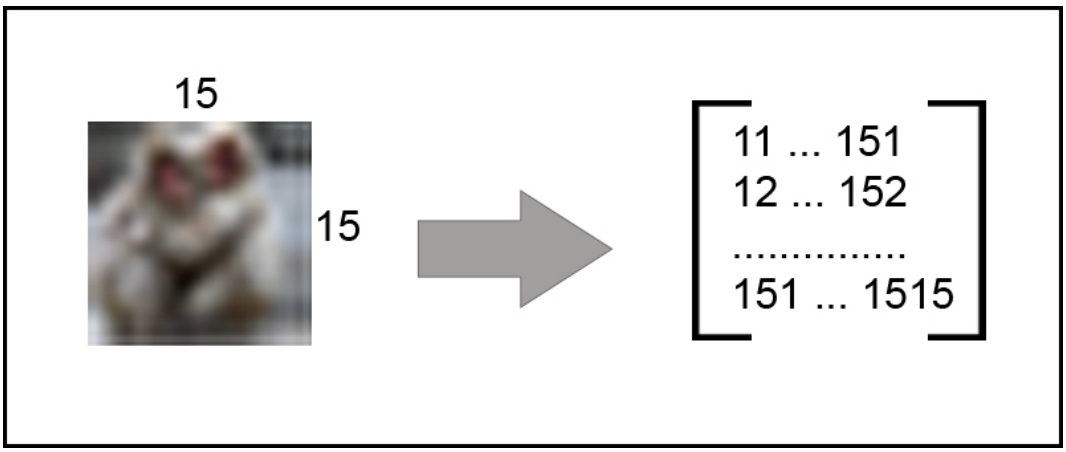

Рис. 9.2: Пиксельная матрица

Создаем таблицу, где будем хранить имя картинки, путь к ней и её сигнатуру:

Листинг 9.31 Таблица для изображений

```
Line 1 CREATE TABLE images (
   -  id serial PRIMARY KEY,
   -  name varchar(50),
   -  img_path varchar(250),
   5  image_array integer[]
   -  );
```

Создадим GIN или GIST индекс:

Листинг 9.32 Создание GIN или GIST индекса

```
Line 1 CREATE INDEX image_array_gin ON images USING GIN(image_array
       _int4_sml_ops);
   -  CREATE INDEX image_array_gist ON images USING GIST(
       image_array _int4_sml_ops);
```

Теперь можно произвести поиск дубликатов:

Листинг 9.33 Поиск дубликатов

```
Line 1 test=# SELECT count(*) from images;
   -    count
   -  ---------
   -   1000000
```

```
(1 row)

test=# EXPLAIN ANALYZE SELECT count(*) FROM images WHERE
    images.image_array % '
    {1010259,1011253,...,2423253,2424252}'::int[];

 Bitmap Heap Scan on images  (cost=286.64..3969.45 rows=986
    width=4) (actual time=504.312..2047.533 rows=200000 loops
    =1)
   Recheck Cond: (image_array % '
    {1010259,1011253,...,2423253,2424252}'::integer[])
   ->  Bitmap Index Scan on image_array_gist  (cost
    =0.00..286.39 rows=986 width=0) (actual time
    =446.109..446.109 rows=200000 loops=1)
         Index Cond: (image_array % '
    {1010259,1011253,...,2423253,2424252}'::integer[])
 Total runtime: 2152.411 ms
(5 rows)
```

где '{1010259,...,2424252}'::int[] — сигнатура изображения, для которой пытаемся найти похожие изображения. С помощью smlar.threshold управляем % похожести картинок (при каком проценте они будут попадать в выборку).

Дополнительно можем добавить сортировку по самым похожим изображениям:

Листинг 9.34 Добавляем сортировку по сходству картинок

```
Line 1 test=# EXPLAIN ANALYZE SELECT smlar(images.image_array, '
    {1010259,...,2424252}'::int[]) as similarity FROM images
    WHERE images.image_array % '{1010259,1011253,
    ...,2423253,2424252}'::int[] ORDER BY similarity DESC;


 Sort  (cost=4020.94..4023.41 rows=986 width=924) (actual
    time=2888.472..2901.977 rows=200000 loops=1)
   Sort Key: (smlar(image_array, '{...,2424252}'::integer[])
    )
   Sort Method: quicksort  Memory: 15520kB
   ->  Bitmap Heap Scan on images  (cost=286.64..3971.91
    rows=986 width=924) (actual time=474.436..2729.638 rows
    =200000 loops=1)
         Recheck Cond: (image_array % '{...,2424252}'::
    integer[])
         ->  Bitmap Index Scan on image_array_gist  (cost
    =0.00..286.39 rows=986 width=0) (actual time
    =421.140..421.140 rows=200000 loops=1)
               Index Cond: (image_array % '{...,2424252}'::
    integer[])
```

```
- Total runtime: 2912.207 ms
- (8 rows)
```

Достаточно эффективно для 1 милиона записей(P.S. Мои данные не помещались в память и PostgreSQL читал их с диска, поэтому скорость будет лучше, если у Вас эта таблица будет в памяти или будут быстрые диски).

Заключение

Smlar расширение может быть использовано в системах, где нам нужно искать похожие объекты, такие как: тексты, темы, блоги, товары, изображения, видео, отпечатки пальцев и прочее.

9.9 Multicorn

Лицензия: Open Source
Ссылка: multicorn.org
Multicorn — расширение для PostgreSQL версии 9.1 или выше, которое позволяет создавать собственные FDW (Foreign Data Wrapper) используя язык программирования Python. Foreign Data Wrapper позволяют подключится к другим источникам данных (другая база, файловая система, REST API, прочее) в PostgreSQL и были представленны с версии 9.1.

Пример

Установка будет проводится на Ubuntu Linux. Для начала нужно установить требуемые зависимости:

Листинг 9.35 Multicorn

```
Line 1 $ sudo aptitude install build-essential postgresql-server-
       dev-9.3 python-dev python-setuptools
```

Следующим шагом установим расширение:

Листинг 9.36 Multicorn

```
Line 1 $ git clone git@github.com:Kozea/Multicorn.git
- $ cd Multicorn
- $ make && sudo make install
```

Для завершения установки активируем расширение для базы данных:

Листинг 9.37 Multicorn

```
Line 1 # CREATE EXTENSION multicorn;
- CREATE EXTENSION
```

Рассмотрим какие FDW может предоставить Multicorn.

Реляционная СУБД

Для подключения к другой реляционной СУБД Multicorn использует SQLAlchemy библиотеку. Данная библиотека поддерживает SQLite, PostgreSQL, MySQL, Oracle, MS-SQL, Firebird, Sybase, и другие базы данных. Для примера настроим подключение к MySQL. Для начала нам потребуется установить зависимости:

Листинг 9.38 Multicorn

```
Line 1 $ sudo aptitude install python-sqlalchemy python-mysqldb
```

В MySQL базе данных «testing» у нас есть таблица «companies»:

Листинг 9.39 Multicorn

```
Line 1 $ mysql -u root -p testing
-
- mysql> SELECT * FROM companies;
- +----+---------------------+---------------------+
5 | id | created_at          | updated_at          |
- +----+---------------------+---------------------+
- |  1 | 2013-07-16 14:06:09 | 2013-07-16 14:06:09 |
- |  2 | 2013-07-16 14:30:00 | 2013-07-16 14:30:00 |
- |  3 | 2013-07-16 14:33:41 | 2013-07-16 14:33:41 |
10 |  4 | 2013-07-16 14:38:42 | 2013-07-16 14:38:42 |
- |  5 | 2013-07-19 14:38:29 | 2013-07-19 14:38:29 |
- +----+---------------------+---------------------+
- 5 rows in set (0.00 sec)
```

В PostgreSQL мы должны создать сервер для Multicorn:

Листинг 9.40 Multicorn

```
Line 1 # CREATE SERVER alchemy_srv foreign data wrapper multicorn
    options (
-     wrapper 'multicorn.sqlalchemyfdw.SqlAlchemyFdw'
- );
- CREATE SERVER
```

Теперь мы можем создать таблицу, которая будет содержать данные из MySQL таблицы «companies»:

Листинг 9.41 Multicorn

```
Line 1 # CREATE FOREIGN TABLE mysql_companies (
-   id integer,
-   created_at timestamp without time zone,
-   updated_at timestamp without time zone
5 ) server alchemy_srv options (
-   tablename 'companies',
-   db_url 'mysql://root:password@127.0.0.1/testing'
- );
- CREATE FOREIGN TABLE
```

Основные опции:

- db_url (string) — SQLAlchemy настройки подключения к базе данных (примеры: mysql://<user>:<password>@<host>/<dbname>, mssql: mssql://<user>:<password>@<dsname>). Подробнее можно узнать из SQLAlchemy документации;
- tablename (string) — имя таблицы в подключенной базе данных.

Теперь можем проверить, что все работает:

Листинг 9.42 Multicorn

```
# SELECT * FROM mysql_companies;
 id |      created_at       |      updated_at
----+-----------------------+---------------------
  1 | 2013-07-16 14:06:09 | 2013-07-16 14:06:09
  2 | 2013-07-16 14:30:00 | 2013-07-16 14:30:00
  3 | 2013-07-16 14:33:41 | 2013-07-16 14:33:41
  4 | 2013-07-16 14:38:42 | 2013-07-16 14:38:42
  5 | 2013-07-19 14:38:29 | 2013-07-19 14:38:29
(5 rows)
```

IMAP сервер

Multicorn может использовается для получение писем по IMAP протоколу. Для начала установим зависимости:

Листинг 9.43 Multicorn

```
$ sudo aptitude install python-pip
$ sudo pip install imapclient
```

Следующим шагом мы должны создать сервер и таблицу, которая будет подключена к IMAP серверу:

Листинг 9.44 Multicorn

```
# CREATE SERVER multicorn_imap FOREIGN DATA WRAPPER
    multicorn options ( wrapper 'multicorn.imapfdw.ImapFdw' )
    ;
CREATE SERVER
# CREATE FOREIGN TABLE my_inbox (
    "Message-ID" character varying,
    "From" character varying,
    "Subject" character varying,
    "payload" character varying,
    "flags" character varying[],
    "To" character varying) server multicorn_imap options (
        host 'imap.gmail.com',
        port '993',
        payload_column 'payload',
```

```
-           flags_column 'flags',
-           ssl 'True',
15          login 'example@gmail.com',
-           password 'supersecretpassword'
- );
- CREATE FOREIGN TABLE
```

Основные опции:

- host (string) — IMAP хост;
- port (string) — IMAP порт;
- login (string) — IMAP логин;
- password (string) — IMAP пароль;
- payload_column (string) — имя поля, которое будет содержать текст письма;
- flags_column (string) — имя поля, которое будет содержать IMAP флаги письма (массив);
- ssl (boolean) — использовать SSL для подключения;
- imap_server_charset (string) — кодировка для IMAP команд. По умолчанию UTF8.

Теперь можно получить письма через таблицу «my_inbox»:

Листинг 9.45 Multicorn

```
Line 1  # SELECT flags, "Subject", payload FROM my_inbox LIMIT 10;
-                   flags                 |      Subject      |
            payload
-  --
       ------------------------------------+-------------------+--------------

-   {$MailFlagBit1,"\\Flagged","\\Seen"} | Test email        |
      Test email\r       +
5                                        |                   |
-   {"\\Seen"}                           | Test second email |
      Test second email\r+
-                                        |                   |
-   (2 rows)
```

RSS

Multicorn может использовать RSS как источник данных. Для начала установим зависимости:

Листинг 9.46 Multicorn

```
Line 1  $ sudo aptitude install python-lxml
```

Как и в прошлые разы, создаем сервер и таблицу для RSS ресурса:

Листинг 9.47 Multicorn

```
# CREATE SERVER rss_srv foreign data wrapper multicorn
    options (
     wrapper 'multicorn.rssfdw.RssFdw'
);
CREATE SERVER
# CREATE FOREIGN TABLE my_rss (
     "pubDate" timestamp,
     description character varying,
     title character varying,
     link character varying
) server rss_srv options (
     url      'http://news.yahoo.com/rss/entertainment'
);
CREATE FOREIGN TABLE
```

Основные опции:

- url (string) — URL RSS ленты.

Кроме того, вы должны быть уверены, что PostgreSQL база данных использовать UTF-8 кодировку (в другой кодировке вы можете получить ошибки). Результат таблицы «my_rss»:

Листинг 9.48 Multicorn

```
# SELECT "pubDate", title, link from my_rss ORDER BY "
    pubDate" DESC LIMIT 10;
        pubDate        |                          title
                       |
    link
--
    ------------------+-------------------------------------------------------

 2013-09-28 14:11:58 | Royal Mint coins to mark Prince
   George christening | http://news.yahoo.com/royal-mint-
   coins-mark-prince-george-christening-115906242.html
 2013-09-28 11:47:03 | Miss Philippines wins Miss World in
   Indonesia       | http://news.yahoo.com/miss-philippines-
   wins-miss-world-indonesia-144544381.html
 2013-09-28 10:59:15 | Billionaire's daughter in NJ court in
    will dispute | http://news.yahoo.com/billionaires-
   daughter-nj-court-dispute-144432331.html
 2013-09-28 08:40:42 | Security tight at Miss World final in
    Indonesia     | http://news.yahoo.com/security-tight-miss
   -world-final-indonesia-123714041.html
 2013-09-28 08:17:52 | Guest lineups for the Sunday news
   shows             | http://news.yahoo.com/guest-lineups-
   sunday-news-shows-183815643.html
```

```
-  2013-09-28 07:37:02 | Security tight at Miss World crowning
     in Indonesia | http://news.yahoo.com/security-tight-miss
   -world-crowning-indonesia-113634310.html
10 2013-09-27 20:49:32 | Simons stamps his natural mark on
   Dior                | http://news.yahoo.com/simons-stamps-
   natural-mark-dior-223848528.html
-  2013-09-27 19:50:30 | Jackson jury ends deliberations until
     Tuesday           | http://news.yahoo.com/jackson-jury-ends-
   deliberations-until-tuesday-235030969.html
-  2013-09-27 19:23:40 | Eric Clapton-owned Richter painting
   to sell in NYC | http://news.yahoo.com/eric-clapton-owned
   -richter-painting-sell-nyc-201447252.html
-  2013-09-27 19:14:15 | Report: Hollywood is less gay-
   friendly off-screen  | http://news.yahoo.com/report-
   hollywood-less-gay-friendly-off-screen-231415235.html
-  (10 rows)
```

CSV

Multicorn может использовать CSV файл как источник данных. Как и в прошлые разы, создаем сервер и таблицу для CSV ресурса:

Листинг 9.49 Multicorn

```
Line 1 # CREATE SERVER csv_srv foreign data wrapper multicorn
     options (
-     wrapper 'multicorn.csvfdw.CsvFdw'
- );
- CREATE SERVER
5 # CREATE FOREIGN TABLE csvtest (
-         sort_order numeric,
-         common_name character varying,
-         formal_name character varying,
-         main_type character varying,
10        sub_type character varying,
-         sovereignty character varying,
-         capital character varying
- ) server csv_srv options (
-         filename '/var/data/countrylist.csv',
15        skip_header '1',
-         delimiter ',');
- CREATE FOREIGN TABLE
```

Основные опции:

- filename (string) — полный путь к CSV файлу;
- delimiter (character) — разделитель в CSV файле (по умолчанию «,»);
- quotechar (character) — кавычки в CSV файле;

- skip_header (integer) — число строк, которые необходимо пропустить (по умолчанию 0).

Результат таблицы «csvtest»:

Листинг 9.50 Multicorn

```
# SELECT * FROM csvtest LIMIT 10;
 sort_order |      common_name      |                formal_name
                    |      main_type      | sub_type |
    sovereignty |      capital
 --
    ----------+-----------------------+-----------------------------------------

          1 | Afghanistan           | Islamic State of
    Afghanistan              | Independent State |          |
                | Kabul
          2 | Albania               | Republic of Albania
                    | Independent State |          |
       | Tirana
          3 | Algeria               | People's Democratic
    Republic of Algeria | Independent State |          |
           | Algiers
          4 | Andorra               | Principality of Andorra
                    | Independent State |          |
       | Andorra la Vella
          5 | Angola                | Republic of Angola
                    | Independent State |          |
       | Luanda
          6 | Antigua and Barbuda   |
                    | Independent State |          |
       | Saint John's
          7 | Argentina             | Argentine Republic
                    | Independent State |          |
       | Buenos Aires
          8 | Armenia               | Republic of Armenia
                    | Independent State |          |
       | Yerevan
          9 | Australia             | Commonwealth of Australia
                    | Independent State |          |
       | Canberra
         10 | Austria               | Republic of Austria
                    | Independent State |          |
       | Vienna
 (10 rows)
```

Другие FDW

Multicorn также содержать FDW для LDAP и файловой системы. LDAP FDW может использоваться для доступа к серверам по LDAP протоколу. FDW для файловой системы может быть использован для доступа к данным, хранящимся в различных файлах в файловой системе.

Собственный FDW

Multicorn предоставляет простой интерфейс для написания собственных FDW. Более подробную информацию вы можете найти по этой ссылке.

PostgreSQL 9.3+

В PostgreSQL 9.1 и 9.2 была представленна реализация FDW только на чтение, и начиная с версии 9.3 FDW может писать в внешнии источники данных. Сейчас Multicorn поддерживает запись данных в другие источники начиная с версии 1.0.0.

Заключение

Multicorn — расширение для PostgreSQL, которое позволяет использовать встроенные FDW или создавать собственные на Python.

9.10 Pgaudit

Лицензия: Open Source
Ссылка: github.com/2ndQuadrant/pgaudit

Pgaudit — расширение для PostgreSQL, которое позволяет собирать события из различных источников внутри PostgreSQL и записывает их в формате CSV с временной меткой, информацией о пользователе, информацию про обьект, который был затронут командой (если такое произошло) и полный текст команды. Поддерживает все DDL, DML (включая SELECT) и прочие команды. Данное расширение работает в PostgreSQL 9.3 и выше.

После установки расширения нужно добавит в конфиг PostgreSQL настройки расширения:

Листинг 9.51 Pgaudit

```
Line 1 shared_preload_libraries = 'pgaudit'
-
- pgaudit.log = 'read, write, user'
```

Далее перегрузить базу данных и установить расширение для базы:

Листинг 9.52 Pgaudit

```
# CREATE EXTENSION pgaudit;
```

После этого в логах можно увидеть подобный результат от pgaudit:

Листинг 9.53 Pgaudit

```
LOG:  [AUDIT],2014-04-30 17:13:55.202854+09,auditdb,ianb,
    ianb,DEFINITION,CREATE TABLE,TABLE,public.x,CREATE  TABLE
     public.x (a pg_catalog.int4    , b pg_catalog.int4    )
     WITH (oids=OFF)
- LOG:  [AUDIT],2014-04-30 17:14:06.548923+09,auditdb,ianb,
    ianb,WRITE,INSERT,TABLE,public.x,INSERT INTO x VALUES
    (1,1);
- LOG:  [AUDIT],2014-04-30 17:14:21.221879+09,auditdb,ianb,
    ianb,READ,SELECT,TABLE,public.x,SELECT * FROM x;
- LOG:  [AUDIT],2014-04-30 17:15:25.620213+09,auditdb,ianb,
    ianb,READ,SELECT,VIEW,public.v_x,SELECT * from v_x;
5 LOG:  [AUDIT],2014-04-30 17:15:25.620262+09,auditdb,ianb,
    ianb,READ,SELECT,TABLE,public.x,SELECT * from v_x;
- LOG:  [AUDIT],2014-04-30 17:16:00.849868+09,auditdb,ianb,
    ianb,WRITE,UPDATE,TABLE,public.x,UPDATE x SET a=a+1;
- LOG:  [AUDIT],2014-04-30 17:16:18.291452+09,auditdb,ianb,
    ianb,ADMIN,VACUUM,,,VACUUM x;
- LOG:  [AUDIT],2014-04-30 17:18:01.08291+09,auditdb,ianb,ianb
    ,DEFINITION,CREATE FUNCTION,FUNCTION,public.func_x(),
    CREATE FUNCTION public.func_x() RETURNS pg_catalog.int4
     LANGUAGE sql  VOLATILE  CALLED ON NULL INPUT SECURITY
    INVOKER COST 100.000000  AS $dprs_$SELECT a FROM x LIMIT
     1;$dprs_$
```

Более подробную информацию про настройку расширения можно найти в официальном README.

9.11 Ltree

Лицензия: Open Source

Ltree – расширение, которое позволяет хранить древовидные структуры в виде меток, а также предоставляет широкие возможности поиска по ним.

Почему Ltree?

- Реализация алгоритма Materialized Path (достаточно быстрый как на запись, так и на чтение);
- Как правило данное решение будет быстрее, чем использовании CTE (Common Table Expressions) или рекурсивный функции (постоянно будут пересчитываться ветвления);

- Встроены механизмы поиска по дереву;
- Индексы (!!!).

Пример

Для начала активируем расширение для базы данных:

Листинг 9.54 Ltree

```
# CREATE EXTENSION ltree;
```

Далее создадим таблицу коментариев, которые будут хранится как дерево.

Листинг 9.55 Ltree

```
CREATE TABLE comments (user_id integer, description text,
    path ltree);
INSERT INTO comments (user_id, description, path) VALUES (
    1, md5(random()::text), '0001');
INSERT INTO comments (user_id, description, path) VALUES (
    2, md5(random()::text), '0001.0001.0001');
INSERT INTO comments (user_id, description, path) VALUES (
    2, md5(random()::text), '0001.0001.0001.0001');
INSERT INTO comments (user_id, description, path) VALUES (
    1, md5(random()::text), '0001.0001.0001.0002');
INSERT INTO comments (user_id, description, path) VALUES (
    5, md5(random()::text), '0001.0001.0001.0003');
INSERT INTO comments (user_id, description, path) VALUES (
    6, md5(random()::text), '0001.0002');
INSERT INTO comments (user_id, description, path) VALUES (
    6, md5(random()::text), '0001.0002.0001');
INSERT INTO comments (user_id, description, path) VALUES (
    6, md5(random()::text), '0001.0003');
INSERT INTO comments (user_id, description, path) VALUES (
    8, md5(random()::text), '0001.0003.0001');
INSERT INTO comments (user_id, description, path) VALUES (
    9, md5(random()::text), '0001.0003.0002');
INSERT INTO comments (user_id, description, path) VALUES (
    11, md5(random()::text), '0001.0003.0002.0001');
INSERT INTO comments (user_id, description, path) VALUES (
    2, md5(random()::text), '0001.0003.0002.0002');
INSERT INTO comments (user_id, description, path) VALUES (
    5, md5(random()::text), '0001.0003.0002.0003');
INSERT INTO comments (user_id, description, path) VALUES (
    7, md5(random()::text), '0001.0003.0002.0002.0001');
INSERT INTO comments (user_id, description, path) VALUES (
    20, md5(random()::text), '0001.0003.0002.0002.0002');
INSERT INTO comments (user_id, description, path) VALUES (
    31, md5(random()::text), '0001.0003.0002.0002.0003');
```

```
- INSERT INTO comments (user_id, description, path) VALUES (
    22, md5(random()::text), '0001.0003.0002.0002.0004');
- INSERT INTO comments (user_id, description, path) VALUES (
    34, md5(random()::text), '0001.0003.0002.0002.0005');
20 INSERT INTO comments (user_id, description, path) VALUES (
    22, md5(random()::text), '0001.0003.0002.0002.0006');
```

Не забываем добавить индексы:

Листинг 9.56 Ltree

```
Line 1 # CREATE INDEX path_gist_comments_idx ON comments USING GIST
    (path);
- # CREATE INDEX path_comments_idx ON comments USING btree(
    path);
```

В данном примере я создаю таблицу comments с полем path, которые и будет содержать полный путь к этому коментарию в дереве (я использую 4 цифры и точку для делителя узлов дерева).

Для начала найдем все коментарии, у который путь начинается с «0001.0003»:

Листинг 9.57 Ltree

```
Line 1 # SELECT user_id, path FROM comments WHERE path <@ '
    0001.0003';
-  user_id |           path
- ---------+--------------------------
-        6 | 0001.0003
5        8 | 0001.0003.0001
-        9 | 0001.0003.0002
-       11 | 0001.0003.0002.0001
-        2 | 0001.0003.0002.0002
-        5 | 0001.0003.0002.0003
10       7 | 0001.0003.0002.0002.0001
-       20 | 0001.0003.0002.0002.0002
-       31 | 0001.0003.0002.0002.0003
-       22 | 0001.0003.0002.0002.0004
-       34 | 0001.0003.0002.0002.0005
15      22 | 0001.0003.0002.0002.0006
- (12 rows)
```

И проверим как работают индексы:

Листинг 9.58 Ltree

```
Line 1 # SET enable_seqscan=false;
- SET
- # EXPLAIN ANALYZE SELECT user_id, path FROM comments WHERE
    path <@ '0001.0003';
-
    QUERY PLAN
```

```
5  --
       ------------------------------------------------------------------------

-   Index Scan using path_gist_comments_idx on comments  (cost
      =0.00..8.29 rows=2 width=38) (actual time=0.023..0.034
      rows=12 loops=1)
-     Index Cond: (path <@ '0001.0003'::ltree)
-   Total runtime: 0.076 ms
-  (3 rows)
```

Данную выборку можно сделать другим запросом:

Листинг 9.59 Ltree

```
Line 1  # SELECT user_id, path FROM comments WHERE path ~ '
          0001.0003.*';
-   user_id |            path
-   ---------+--------------------------
-         6 | 0001.0003
5         8 | 0001.0003.0001
-         9 | 0001.0003.0002
-        11 | 0001.0003.0002.0001
-         2 | 0001.0003.0002.0002
-         5 | 0001.0003.0002.0003
10        7 | 0001.0003.0002.0002.0001
-        20 | 0001.0003.0002.0002.0002
-        31 | 0001.0003.0002.0002.0003
-        22 | 0001.0003.0002.0002.0004
-        34 | 0001.0003.0002.0002.0005
15       22 | 0001.0003.0002.0002.0006
-   (12 rows)
```

Не забываем про сортировку дерева:

Листинг 9.60 Ltree

```
Line 1  # INSERT INTO comments (user_id, description, path) VALUES (
            9, md5(random()::text), '0001.0003.0001.0001');
-   # INSERT INTO comments (user_id, description, path) VALUES (
            9, md5(random()::text), '0001.0003.0001.0002');
-   # INSERT INTO comments (user_id, description, path) VALUES (
            9, md5(random()::text), '0001.0003.0001.0003');
-   # SELECT user_id, path FROM comments WHERE path ~ '
          0001.0003.*';
5   user_id |            path
-   ---------+--------------------------
-         6 | 0001.0003
-         8 | 0001.0003.0001
-         9 | 0001.0003.0002
10       11 | 0001.0003.0002.0001
-         2 | 0001.0003.0002.0002
```

```
        5 | 0001.0003.0002.0003
        7 | 0001.0003.0002.0002.0001
       20 | 0001.0003.0002.0002.0002
       31 | 0001.0003.0002.0002.0003
       22 | 0001.0003.0002.0002.0004
       34 | 0001.0003.0002.0002.0005
       22 | 0001.0003.0002.0002.0006
        9 | 0001.0003.0001.0001
        9 | 0001.0003.0001.0002
        9 | 0001.0003.0001.0003
(15 rows)
# SELECT user_id, path FROM comments WHERE path ~ '
    0001.0003.*' ORDER by path;
 user_id |            path
---------+----------------------------
        6 | 0001.0003
        8 | 0001.0003.0001
        9 | 0001.0003.0001.0001
        9 | 0001.0003.0001.0002
        9 | 0001.0003.0001.0003
        9 | 0001.0003.0002
       11 | 0001.0003.0002.0001
        2 | 0001.0003.0002.0002
        7 | 0001.0003.0002.0002.0001
       20 | 0001.0003.0002.0002.0002
       31 | 0001.0003.0002.0002.0003
       22 | 0001.0003.0002.0002.0004
       34 | 0001.0003.0002.0002.0005
       22 | 0001.0003.0002.0002.0006
        5 | 0001.0003.0002.0003
(15 rows)
```

Для поиска можно использовать разные модификаторы. Пример использования «или» (|):

Листинг 9.61 Ltree

```
# SELECT user_id, path FROM comments WHERE path ~ '
    0001.*{1,2}.0001|0002.*' ORDER by path;
 user_id |            path
---------+----------------------------
        2 | 0001.0001.0001
        2 | 0001.0001.0001.0001
        1 | 0001.0001.0001.0002
        5 | 0001.0001.0001.0003
        6 | 0001.0002.0001
        8 | 0001.0003.0001
        9 | 0001.0003.0001.0001
        9 | 0001.0003.0001.0002
```

```
-         9 | 0001.0003.0001.0003
-         9 | 0001.0003.0002
-        11 | 0001.0003.0002.0001
15        2 | 0001.0003.0002.0002
-         7 | 0001.0003.0002.0002.0001
-        20 | 0001.0003.0002.0002.0002
-        31 | 0001.0003.0002.0002.0003
-        22 | 0001.0003.0002.0002.0004
20       34 | 0001.0003.0002.0002.0005
-        22 | 0001.0003.0002.0002.0006
-         5 | 0001.0003.0002.0003
-(19 rows)
```

Например, найдем прямых потомков от «0001.0003»:

Листинг 9.62 Ltree

```
Line 1 # SELECT user_id, path FROM comments WHERE path ~ '
       0001.0003.*{1}' ORDER by path;
-  user_id |       path
- ---------+-----------------
-         8 | 0001.0003.0001
5        9 | 0001.0003.0002
- (2 rows)
```

Можно также найти родителя для потомка «0001.0003.0002.0002.0005»:

Листинг 9.63 Ltree

```
Line 1 # SELECT user_id, path FROM comments WHERE path = subpath('
       0001.0003.0002.0002.0005', 0, -1) ORDER by path;
-  user_id |         path
- ---------+---------------------
-         2 | 0001.0003.0002.0002
5 (1 row)
```

Заключение

Ltree — расширение, которое позволяет хранить и удобно управлять Materialized Path в PostgreSQL.

9.12 PostPic

Лицензия: Open Source
Ссылка: github.com/drotiro/postpic

PostPic расширение для СУБД PostgreSQL, которое позволяет обрабатывать изображения в базе данных, как PostGIS делает это с пространственными данными. Он добавляет новый типа поля «image», а также несколько функций для обработки изображений (обрезка краев, создание

миниатюр, поворот и т.д.) и извлечений его атрибутов (размер, тип, разрешение).

9.13 Fuzzystrmatch

Лицензия: Open Source

Fuzzystrmatch предоставляет несколько функций для определения сходства и расстояния между строками. Функция soundex используется для согласования сходно звучащих имен путем преобразования их в одинаковый код. Функция difference преобразует две строки в soundex код, а затем сообщает количество совпадающих позиций кода. В soundex код состоит из четырех символов, поэтому результат будет от нуля до четырех: 0 — не совпадают, 4 — точное совпадение (таким образом, функция названа неверно — как название лучше подходит similarity):

Листинг 9.64 soundex

```
Line 1  # CREATE EXTENSION fuzzystrmatch;
  -     CREATE EXTENSION
  -     # SELECT soundex('hello world!');
  -      soundex
  5     ---------
  -      H464
  -     (1 row)
  -
  -     # SELECT soundex('Anne'), soundex('Ann'), difference('Anne',
            'Ann');
 10      soundex | soundex | difference
  -     ---------+---------+------------
  -      A500    | A500    |          4
  -     (1 row)
  -
 15     # SELECT soundex('Anne'), soundex('Andrew'), difference('
           Anne', 'Andrew');
  -      soundex | soundex | difference
  -     ---------+---------+------------
  -      A500    | A536    |          2
  -     (1 row)
 20
  -     # SELECT soundex('Anne'), soundex('Margaret'), difference('
           Anne', 'Margaret');
  -      soundex | soundex | difference
  -     ---------+---------+------------
  -      A500    | M626    |          0
 25     (1 row)
  -
  -     # CREATE TABLE s (nm text);
```

```
- CREATE TABLE
- # INSERT INTO s VALUES ('john'), ('joan'), ('wobbly'), ('
    jack');
30 INSERT 0 4
- # SELECT * FROM s WHERE soundex(nm) = soundex('john');
-   nm
- ------
-  john
35  joan
- (2 rows)
-
- # SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
-   nm
40 ------
-  john
-  joan
-  jack
- (3 rows)
```

Функция levenshtein вычисляет расстояние Левенштейна между двумя строками. levenshtein_less_equal ускоряется функцию levenshtein для маленьких значений расстояния:

Листинг 9.65 levenshtein

```
Line 1 # SELECT levenshtein('GUMBO', 'GAMBOL');
-  levenshtein
- -------------
-            2
5 (1 row)
-
- # SELECT levenshtein('GUMBO', 'GAMBOL', 2, 1, 1);
-  levenshtein
- -------------
10           3
- (1 row)
-
- # SELECT levenshtein_less_equal('extensive', 'exhaustive',
    2);
-  levenshtein_less_equal
15 ------------------------
-                        3
- (1 row)
-
- test=# SELECT levenshtein_less_equal('extensive', '
    exhaustive', 4);
20  levenshtein_less_equal
- ------------------------
-                        4
```

```
(1 row)
```

Функция metaphone, как и soundex, построена на идее создания кода для строки: две строки, которые будут считатся похожими, будут иметь одинаковые коды. Последним параметром указывается максимальная длина metaphone кода. Функция dmetaphone вычисляет два «как звучит» кода для строки — «первичный» и «альтернативный»:

Листинг 9.66 metaphone

```
# SELECT metaphone('GUMBO', 4);
 metaphone
-----------
 KM
(1 row)
# SELECT dmetaphone('postgresql');
 dmetaphone
------------
 PSTK
(1 row)

# SELECT dmetaphone_alt('postgresql');
 dmetaphone_alt
----------------
 PSTK
(1 row)
```

9.14 Tsearch2

Лицензия: Open Source

Tsearch2 – расширение для полнотекстового поиска. Встроен в PostgreSQL начиная с версии 8.3.

9.15 OpenFTS

Лицензия: Open Source

Ссылка: openfts.sourceforge.net

OpenFTS (Open Source Full Text Search engine) является продвинутой PostgreSQL поисковой системой, которая обеспечивает онлайн индексирования данных и актуальность данных для поиска по базе. Тесная интеграция с базой данных позволяет использовать метаданные, чтобы ограничить результаты поиска.

9.16 PL/Proxy

Лицензия: Open Source
Ссылка: pgfoundry.org/projects/plproxy
PL/Proxy представляет собой прокси-язык для удаленного вызова процедур и партицирования данных между разными базами. Подробнее можно почитать в §5.2 главе.

9.17 Texcaller

Лицензия: Open Source
Ссылка: www.profv.de/texcaller
Texcaller — это удобный интерфейс для командной строки TeX, который обрабатывает все виды ошибок. Он написан в простом C, довольно портативный, и не имеет внешних зависимостей, кроме TeX. Неверный TeX документ обрабатывается путем простого возвращения NULL, а не прерывается с ошибкой. В случае неудачи, а также в случае успеха, дополнительная обработка информации осуществляется через NOTICEs.

9.18 Pgmemcache

Лицензия: Open Source
Ссылка: pgfoundry.org/projects/pgmemcache
Pgmemcache — это PostgreSQL API библиотека на основе libmemcached для взаимодействия с memcached. С помощью данной библиотеки PostgreSQL может записывать, считывать, искать и удалять данные из memcached. Подробнее можно почитать в «8.2 Pgmemcache» главе.

9.19 Prefix

Лицензия: Open Source
Ссылка: pgfoundry.org/projects/prefix
Prefix реализует поиск текста по префиксу (prefix @> text). Prefix используется в приложениях телефонии, где маршрутизация вызовов и расходы зависят от вызывающего/вызываемого префикса телефонного номера оператора.

9.20 Dblink

Лицензия: Open Source

Dblink – расширение, которое позволяет выполнять запросы к удаленным базам данных непосредственно из SQL, не прибегая к помощи внешних скриптов.

9.21 Заключение

Расширения помогают улучшить работу PostgreSQL в решении специфических проблем. Расширяемость PostgreSQL позволяет создавать собственные расширения, или же наоборот, не нагружать СУБД лишним, не требуемым функционалом.

10

Бэкап и восстановление PostgreSQL

Есть два типа администраторов — те, кто не делает бэкапы, и те, кто уже делает

Народная мудрость

Если какая-нибудь неприятность может произойти, она случается.

Закон Мэрфи

10.1 Введение

Любой хороший сисадмин знает — бэкапы нужны всегда.Насколько бы надежной ни казалась Ваша система, всегда может произойти случай, который был не учтен, и из-за которого могут быть потеряны данные.

Тоже самое касается и PostgreSQL баз данных. Бекапы должны быть! Посыпавшийся винчестер на сервере, ошибка в файловой системе, ошибка в другой программе, которая перетерла весь каталог PostgreSQL и многое другое приведет только к плачевному результату. И даже если у Вас репликация с множеством слейвов, это не означает, что система в безопасности — неверный запрос на мастер (DELETE, DROP), и у слейвов такая же порция данных (точнее их отсутствие).

Существуют три принципиально различных подхода к резервному копированию данных PostgreSQL:

- SQL бэкап;
- Бекап уровня файловой системы;
- Непрерывное резервное копирование;

Каждый из этих подходов имеет свои сильные и слабые стороны.

10.2 SQL бэкап

Идея этого подхода в создании текстового файла с командами SQL. Такой файл можно передать обратно на сервер и воссоздать базу данных в том же состоянии, в котором она была во время бэкапа. У PostgreSQL для этого есть специальная утилита — pg_dump. Пример использования pg_dump:

Листинг 10.1 Создаем бэкап с помощью pg_dump

```
Line 1  $ pg_dump dbname > outfile
```

Для восстановления такого бэкапа достаточно выполнить:

Листинг 10.2 Восстанавливаем бэкап

```
Line 1  $ psql dbname < infile
```

При этом базу данных «dbname» потребуется создать перед восстановлением. Также потребуется создать пользователей, которые имеют доступ к данным, которые восстанавливаются (это можно и не делать, но тогда просто в выводе восстановления будут ошибки). Если нам требуется, чтобы восстановление прекратилось при возникновении ошибки, тогда потребуется восстанавливать бэкап таким способом:

Листинг 10.3 Восстанавливаем бэкап

```
Line 1  $ psql --set ON_ERROR_STOP=on dbname < infile
```

Также, можно делать бэкап и сразу восстанавливать его в другую базу:

Листинг 10.4 Бекап в другую БД

```
Line 1  $ pg_dump -h host1 dbname | psql -h host2 dbname
```

После восстановления бэкапа желательно запустить ANALYZE, чтобы оптимизатор запросов обновил статистику.

А что, если нужно сделать бэкап не одной базы данных, а всех, да и еще получить в бэкапе информацию про роли и таблицы? В таком случае у PostgreSQL есть утилита pg_dumpall. pg_dumpall используется для создания бэкапа данных всего кластера PostgreSQL:

Листинг 10.5 Бекап кластера PostgreSQL

```
Line 1  $ pg_dumpall > outfile
```

Для восстановления такого бэкапа достаточно выполнить от суперпользователя:

Листинг 10.6 Восстановления бэкапа PostgreSQL

```
Line 1  $ psql -f infile postgres
```

SQL бэкап больших баз данных

Некоторые операционные системы имеют ограничения на максимальный размер файла, что может вызывать проблемы при создании больших бэкапов через pg_dump. К счастью, pg_dump можете бэкапить в стандартный вывод. Так что можно использовать стандартные инструменты Unix, чтобы обойти эту проблему. Есть несколько возможных способов:

- Использовать сжатие для бэкапа.

  Можно использовать программу сжатия данных, например GZIP:

  Листинг 10.7 Сжатие бэкапа PostgreSQL

```
Line 1  $ pg_dump dbname | gzip > filename.gz
```

  Восстановление:

  Листинг 10.8 Восстановление бэкапа PostgreSQL

```
Line 1  $ gunzip -c filename.gz | psql dbname
```

  или

  Листинг 10.9 Восстановление бэкапа PostgreSQL

```
Line 1  cat filename.gz | gunzip | psql dbname
```

- Использовать команду split.

  Команда split позволяет разделить вывод в файлы меньшего размера, которые являются подходящими по размеру для файловой системы. Например, бэкап делится на куски по 1 мегабайту:

  Листинг 10.10 Создание бэкапа PostgreSQL

```
Line 1  $ pg_dump dbname | split -b 1m - filename
```

  Восстановление:

  Листинг 10.11 Восстановление бэкапа PostgreSQL

```
Line 1  $ cat filename* | psql dbname
```

- Использовать пользовательский формат дампа pg_dump

  PostgreSQL построен на системе с библиотекой сжатия Zlib, поэтому пользовательский формат бэкапа будет в сжатом виде. Это похоже на метод с импользованием GZIP, но он имеет дополнительное преимущество — таблицы могут быть восстановлены выборочно:

  Листинг 10.12 Создание бэкапа PostgreSQL

```
Line 1  $ pg_dump -Fc dbname > filename
```

  Через psql такой бэкап не восстановить, но для этого есть утилита pg_restore:

Листинг 10.13 Восстановление бэкапа PostgreSQL

```
Line 1  $ pg_restore -d dbname filename
```

При слишком большой базе данных, вариант с командой split нужно комбинировать со сжатием данных.

10.3 Бекап уровня файловой системы

Альтернативный метод резервного копирования заключается в непосредственном копировании файлов, которые PostgreSQL использует для хранения данных в базе данных. Например:

Листинг 10.14 Бэкап PostgreSQL файлов

```
Line 1  $ tar -cf backup.tar /usr/local/pgsql/data
```

Но есть два ограничения, которые делает этот метод нецелесообразным, или, по крайней мере, уступающим SQL бэкапу:

- PostgreSQL база данных должна быть остановлена, для того, чтобы получить актуальный бэкап (PostgreSQL держит множество объектов в памяти, буферизация файловой системы). Излишне говорить, что во время восстановления такого бэкапа потребуется также остановить PostgreSQL;
- Не получится востановить только определенные данные с такого бэкапа;

Как альтернатива, можно делать снимки (snapshot) файлов системы (папки с файлами PostgreSQL). В таком случае останавливать PostgreSQL не требуется. Однако, резервная копия, созданная таким образом, сохраняет файлы базы данных в состоянии, как если бы сервер базы данных был неправильно остановлен. Поэтому при запуске PostgreSQL из резервной копии, он будет думать, что предыдущий экземпляр сервера вышел из строя и повторит журнала WAL. Это не проблема, просто надо знать про это (и не забыть включить WAL файлы в резервную копию). Также, если файловая система PostgreSQL распределена по разным файловым системам, то такой метод бэкапа будет очень ненадежным — снимки файлов системы должны быть сделаны одновременно(!!!). Почитайте документацию файловой системы очень внимательно, прежде чем доверять снимкам файлов системы в таких ситуациях.

Также возможен вариант с использованием rsync. Первым запуском rsync мы копируем основные файлы с директории PostgreSQL (PostgreSQL при этом продолжает работу). После этого мы останавливаем PostgreSQL и запускаем повторно rsync. Второй запуск rsync пройдет гораздо быстрее, чем первый, потому что будет передавать относительно небольшой

размер данных, и конечный результат будет соответствовать остановленной СУБД. Этот метод позволяет делать бекап уровня файловой системы с минимальным временем простоя.

10.4 Непрерывное резервное копирование

PostgreSQL поддерживает упреждающую запись логов (Write Ahead Log, WAL) в pg_xlog директорию, которая находится в директории данных СУБД. В логи пишутся все изменения сделанные с данными в СУБД. Этот журнал существует прежде всего для безопасности во время краха PostgreSQL: если происходят сбои в системе, базы данных могут быть восстановлены с помощью «перезапуска» этого журнала. Тем не менее, существование журнала делает возможным использование третьей стратегии для резервного копирования баз данных: мы можем объединить бекап уровня файловой системы с резервной копией WAL файлов. Если требуется восстановить такой бэкап, то мы восстанавливаем файлы резервной копии файловой системы, а затем «перезапускаем» с резервной копии файлов WAL для приведения системы к актуальному состоянию. Этот подход является более сложным для администрирования, чем любой из предыдущих подходов, но он имеет некоторые преимущества:

- Не нужно согласовывать файлы резервной копии системы. Любая внутренняя противоречивость в резервной копии будет исправлена путем преобразования журнала (не отличается от того, что происходит во время восстановления после сбоя);
- Восстановление состояния сервера для определенного момента времени;
- Если мы постоянно будем «скармливать» файлы WAL на другую машину, которая была загружена с тех же файлов резервной базы, то у нас будет находящийся всегда в актуальном состоянии резервный сервер PostgreSQL (создание сервера горячего резерва).

Как и бэкап файловой системы, этот метод может поддерживать только восстановление всей базы данных кластера. Кроме того, он требует много места для хранения WAL файлов.

Настройка

Первый шаг — активировать архивирование. Эта процедура будет копировать WAL файлы в архивный каталог из стандартного каталога pg_xlog. Это делается в файле postgresql.conf:

Листинг 10.15 Настройка архивирования

```
Line 1  archive_mode = on # enable archiving
```

```
- archive_command = 'cp -v %p /data/pgsql/archives/%f'
- archive_timeout = 300 # timeout to close buffers
```

После этого необходимо перенести файлы (в порядке их появления) в архивный каталог. Для этого можно использовать функцию rsync. Можно поставить функцию в cron и, таким образом, файлы могут автоматически перемещаться между хостами каждые несколько минут:

Листинг 10.16 Копирование WAL файлов на другой хост

```
Line 1 $ rsync -avz --delete prod1:/data/pgsql/archives/ \
- /data/pgsql/archives/ > /dev/null
```

В конце, необходимо скопировать файлы в каталог pg_xlog на сервере PostgreSQL (он должен быть в режиме восстановления). Для этого необходимо в каталоге данных PostgreSQL создать файл recovery.conf с заданной командой копирования файлов из архива в нужную директорию:

Листинг 10.17 recovery.conf

```
Line 1 restore_command = 'cp /data/pgsql/archives/%f "%p"'
```

Документация PostgreSQL предлагает хорошее описание настройки непрерывного копирования, поэтому я не углублялся в детали (например, как перенести директорию СУБД с одного сервера на другой, какие могут быть проблемы). Более подробно вы можете почитать по этой ссылке www.postgresql.org/docs/current/static/continuous-archiving.html.

10.5 Утилиты для непрерывного резервного копирования

Непрерывное резервное копирования один из лучших спрособ для создания бэкапов и восстановления их. Нередко бэкапы сохраняются на той же файловой системе, на которой расположена база данных. Это не очень безопасно, т.к. при выходе дисковой системы сервера из строя вы можете потерять все данные (и базу, и бэкапы), или попросту столкнуться с тем, что на жестком диске закончится свободное место. Поэтому лучше, когда бэкапы складываются на отдельный сервер или в «облачное хранилище» (например AWS S3). Чтобы не писать свой «велосипед» для автоматизации этого процесса на сегодняшний день существует набор программ, которые облегчает процесс настройки и поддержки процесса создания бэкапов на основе непрерывного резервного копирования.

WAL-E

Ссылка: github.com/wal-e/wal-e

WAL-E предназначенная для непрерывной архивации PostgreSQL WAL-logs в Amazon S3 или Windows Azure (начиная с версии 0.7) и управления использованием pg_start_backup и pg_stop_backup. Утилита написана на Python и разработана в компании Heroku, где её активно используют.

Установка

У WAL-E есть пару зависимостей: lzop, psql, pv (в старых версиях используется mbuffer), python 2.6+ и несколько python библиотек (gevent >= 0.13, boto >= 2.0, azure). Также для удобства настроек переменных среды устанавливается daemontools. На Ubuntu это можно все поставить одной командой:

Листинг 10.18 Установка зависимостей для WAL-E

```
Line 1  # PostgreSQL уже установлен
    -   $ aptitude install git-core python-dev python-setuptools
          python-pip build-essential libevent-dev lzop pv
          daemontools daemontools-run
```

Теперь установим WAL-E:

Листинг 10.19 Установка WAL-E

```
Line 1  $ pip install https://github.com/wal-e/wal-e/archive/v0.7a1.
          tar.gz
```

После успешной установки можно начать работать с WAL-E.

Настройка и работа

Как уже писалось, WAL-E сливает все данные в AWS S3, поэтому нам потребуются «Access Key ID» и «Secret Access Key» (эти данные можно найти в акаунте Amazon AWS). Команда для загрузки бэкапа всей базы данных в S3:

Листинг 10.20 Загрузка бэкапа всей базы данных в S3

```
Line 1  AWS_SECRET_ACCESS_KEY=... wal-e                          \
    -     -k AWS_ACCESS_KEY_ID                                  \
    -     --s3-prefix=s3://some-bucket/directory/or/whatever    \
    -     backup-push /var/lib/postgresql/9.2/main
```

Где s3-prefix — URL, который содержит имя S3 бакета (bucket) и путь к папке, куда следует складывать резервные копии. Команда для загрузки WAL-логов на S3:

Листинг 10.21 Загрузка WAL-логов на S3

```
Line 1  AWS_SECRET_ACCESS_KEY=... wal-e                          \
    -     -k AWS_ACCESS_KEY_ID                                  \
    -     --s3-prefix=s3://some-bucket/directory/or/whatever    \
```

```
-   wal-push /var/lib/postgresql/9.2/main/pg_xlog/
    WAL_SEGMENT_LONG_HEX
```

Для управления этими переменными окружения можно использовать команду envdir (идет в поставке с daemontools). Для этого создадим envdir каталог:

Листинг 10.22 WAL-E с envdir

```
Line 1  $ mkdir -p /etc/wal-e.d/env
-   $ echo "secret-key" > /etc/wal-e.d/env/AWS_SECRET_ACCESS_KEY
-   $ echo "access-key" > /etc/wal-e.d/env/AWS_ACCESS_KEY_ID
-   $ echo 's3://some-bucket/directory/or/whatever' > /etc/wal-e
    .d/env/WALE_S3_PREFIX
5   $ chown -R root:postgres /etc/wal-e.d
```

После создания данного каталога появляется возможность запускать WAL-E команды гораздо проще и с меньшим риском случайного использования некорректных значений:

Листинг 10.23 WAL-E с envdir

```
Line 1  $ envdir /etc/wal-e.d/env wal-e backup-push ...
-   $ envdir /etc/wal-e.d/env wal-e wal-push ...
```

Теперь настроим PostgreSQL для сбрасывания WAL-логов в S3 с помощью WAL-E. Отредактируем postgresql.conf:

Листинг 10.24 Настройка PostgreSQL

```
Line 1  wal_level = hot_standby # или archive, если PostgreSQL < 9.0
-   archive_mode = on
-   archive_command = 'envdir /etc/wal-e.d/env /usr/local/bin/
    wal-e wal-push %p'
-   archive_timeout = 60
```

Лучше указать полный путь к WAL-E (можно узнать командой which wal-e), поскольку PostgreSQL может его не найти. После этого нужно перегрузить PostgreSQL. В логах базы вы должны увидеть что-то подобное:

Листинг 10.25 Логи PostgreSQL

```
Line 1  2012-11-07 14:52:19 UTC LOG:  database system was shut down
    at 2012-11-07 14:51:40 UTC
-   2012-11-07 14:52:19 UTC LOG:  database system is ready to
    accept connections
-   2012-11-07 14:52:19 UTC LOG:  autovacuum launcher started
-   2012-11-07T14:52:19.784+00 pid=7653 wal_e.worker.s3_worker
    INFO     MSG: begin archiving a file
5         DETAIL: Uploading "pg_xlog/000000010000000000000001"
     to "s3://cleverdb-pg-backups/pg/wal_005
    /000000010000000000000001.lzo".
-   2012-11-07 14:52:19 UTC LOG:  incomplete startup packet
```

```
- 2012-11-07T14:52:28.234+00 pid=7653 wal_e.worker.s3_worker
    INFO     MSG: completed archiving to a file
-         DETAIL: Archiving to "s3://cleverdb-pg-backups/pg/
    wal_005/000000010000000000000001.lzo" complete at 21583.3
    KiB/s.
- 2012-11-07T14:52:28.341+00 pid=7697 wal_e.worker.s3_worker
    INFO     MSG: begin archiving a file
10        DETAIL: Uploading "pg_xlog
    /000000010000000000000002.00000020.backup" to "s3://
    cleverdb-pg-backups/pg/wal_005
    /000000010000000000000002.00000020.backup.lzo".
- 2012-11-07T14:52:34.027+00 pid=7697 wal_e.worker.s3_worker
    INFO     MSG: completed archiving to a file
-         DETAIL: Archiving to "s3://cleverdb-pg-backups/pg/
    wal_005/000000010000000000000002.00000020.backup.lzo"
    complete at 00KiB/s.
- 2012-11-07T14:52:34.187+00 pid=7711 wal_e.worker.s3_worker
    INFO     MSG: begin archiving a file
-         DETAIL: Uploading "pg_xlog/000000010000000000000002"
     to "s3://cleverdb-pg-backups/pg/wal_005
    /000000010000000000000002.lzo".
15 2012-11-07T14:52:40.232+00 pid=7711 wal_e.worker.s3_worker
    INFO     MSG: completed archiving to a file
-         DETAIL: Archiving to "s3://cleverdb-pg-backups/pg/
    wal_005/000000010000000000000002.lzo" complete at 2466.67
    KiB/s.
```

Если ничего похожего в логах не видно, тогда нужно смотреть что за ошибка появляется и исправлять её.

Для того, чтобы бэкапить всю базу достаточно выполнить данную команду:

Листинг 10.26 Загрузка бэкапа всей базы данных в S3

```
Line 1 $ envdir /etc/wal-e.d/env wal-e backup-push /var/lib/
    postgresql/9.2/main
- 2012-11-07T14:49:26.174+00 pid=7493 wal_e.operator.
    s3_operator INFO     MSG: start upload postgres version
    metadata
-         DETAIL: Uploading to s3://cleverdb-pg-backups/pg/
    basebackups_005/base_000000010000000000000006_00000032/
    extended_version.txt.
- 2012-11-07T14:49:32.783+00 pid=7493 wal_e.operator.
    s3_operator INFO     MSG: postgres version metadata
    upload complete
5 2012-11-07T14:49:32.859+00 pid=7493 wal_e.worker.s3_worker
    INFO     MSG: beginning volume compression
-         DETAIL: Building volume 0.
- ...
```

```
- HINT:  Check that your archive_command is executing properly
    .  pg_stop_backup can be canceled safely, but the
    database backup will not be usable without all the WAL
    segments.
- NOTICE:  pg_stop_backup complete, all required WAL segments
    have been archived
```

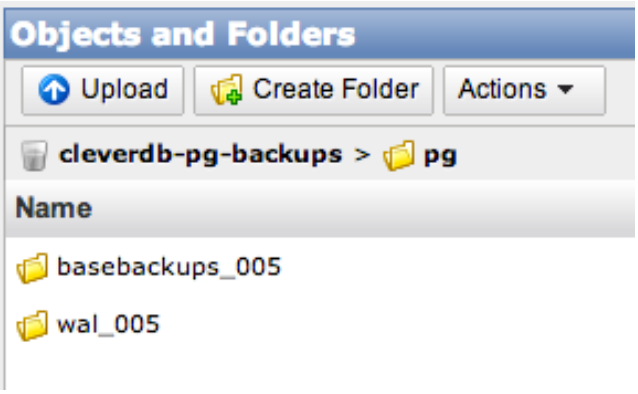

Рис. 10.1: Папка бэкапов на S3

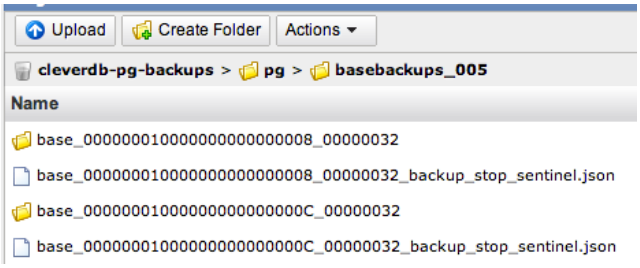

Рис. 10.2: Папка бэкапов базы на S3

Данный бэкап лучше делать раз в сутки (например, добавить в crontab). На рис 10.1-10.3 видно как хранятся бэкапы на S3. Все бэкапы сжаты через lzop. Данный алгоритм сжимает хуже чем gzip, но скорость сжатия намного быстрее (приблизительно 25Мб/сек используя 5% ЦПУ). Чтобы уменьшить нагрузку на чтение с жесткого диска бэкапы отправляются через pv утилиту (опцией cluster-read-rate-limit можно ограничить скорость чтения, если это требуется).

Теперь перейдем к восстановлению данных. Для восстановления базы из резервной копии используется backup-fetch команда:

Листинг 10.27 Восстановление бэкапа базы из S3

```
Line 1  $ sudo -u postgres bash -c "envdir /etc/wal-e.d/env wal-e
        --s3-prefix=s3://some-bucket/directory/or/whatever backup
        -fetch /var/lib/postgresql/9.2/main LATEST"
```

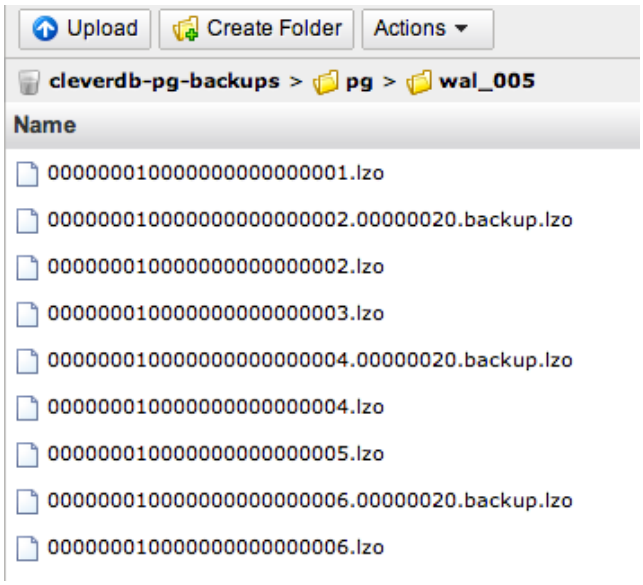

Рис. 10.3: Папка WAL-логов на S3

Где LATEST означает восстановится из последнего актуального бэкапа (PostgreSQL в это время должен быть остановлен). Для восстановления из более поздней резервной копии:

Листинг 10.28 Восстановление из поздней резервной копии

```
Line 1 $ sudo -u postgres bash -c "envdir /etc/wal-e.d/env wal-e
       --s3-prefix=s3://some-bucket/directory/or/whatever backup
       -fetch /var/lib/postgresql/9.2/main
       base_LONGWALNUMBER_POSITION_NUMBER"
```

Для получения списка доступных резервных копий есть команда backup -list:

Листинг 10.29 Список резервных копий

```
Line 1 $ envdir /etc/wal-e.d/env wal-e backup-list
  - name    last_modified    expanded_size_bytes
       wal_segment_backup_start
       wal_segment_offset_backup_start wal_segment_backup_stop
       wal_segment_offset_backup_stop
  - base_000000010000000000000008_00000032  2012-11-07T14
       :00:07.000Z                 000000010000000000000008
                00000032
  - base_00000001000000000000000C_00000032  2012-11-08T15
       :00:08.000Z                 00000001000000000000000C
                00000032
```

После завершения работы с основной резервной копией для полного восстановления нужно считать WAL-логи (чтобы данные обновились до последнего состояния). Для этого используется recovery.conf:

Листинг 10.30 recovery.conf

```
restore_command = 'envdir /etc/wal-e.d/env /usr/local/bin/
    wal-e wal-fetch "%f" "%p"'
```

После создания этого файла нужно запустить PostgreSQL. Через небольшой интервал времени база станет полностью восстановленной.

Для удаления старых резервных копий (или вообще всех) используется команда delete:

Листинг 10.31 Удаление резервных копий

```
# удаления старых бэкапов старше
    base_00000004000002DF000000A6_03626144
- $ envdir /etc/wal-e.d/env wal-e delete --confirm before
    base_00000004000002DF000000A6_03626144
- # удаления всех бэкапов
- $ envdir /etc/wal-e.d/env wal-e delete --confirm everything
# удалить все старше последних 20 бэкапов
- $ envdir /etc/wal-e.d/env wal-e delete --confirm retain 20
```

Без опции --confirm команды будут запускаться и показывать, что будет удаляться, но фактического удаления не будет производиться (dry run).

Заключение

WAL-E помогает автоматизировать сбор резервных копий с PostgreSQL и хранить их в достаточно дешевом и надежном хранилище — Amazon S3.

Barman

Ссылка: www.pgbarman.org

Barman, как и WAL-E, позволяет создать систему для бэкапа и восстановления PostgreSQL на основе непрерывного резервного копирования. Barman использует для хранения бэкапов отдельный сервер, который может собирать бэкапы как с одного, так и с нескольких PostgreSQL баз данных.

Установка и настройка

Рассмотрим простом случай с одним экземпляром PostgreSQL (один сервер) и пусть его хост будет pghost. Наша задача — автоматизировать сбор и хранение бэкапов этой базы на другом сервере (его хост будет brhost). Для взаимодействия эти два сервера должны быть полностью открыты по SSH (доступ без пароля, по ключам). Для этого можно использовать authorized_keys файл.

Листинг 10.32 Проверка подключения по SSH

```
# Проверка подключения с сервера PostgreSQL (pghost)
```

```
- $ ssh barman@brhost
- # Проверка подключения с сервера бэкапов (brhost)
- $ ssh postgres@pghost
```

Далее нужно установить на сервере для бэкапов barman. Сам barman написан на python и имеет пару зависимостей: python 2.6+, rsync >= 3.0.4 и python библиотеки (argh, psycopg2, python-dateutil < 2.0 (для python 3.0 не нужен), distribute). На Ubuntu все зависимости можно поставить одной командой:

Листинг 10.33 Установка зависимостей barman

```
Line 1 $ aptitude install python-dev python-argh python-psycopg2
       python-dateutil rsync python-setuptools
```

Далее нужно установить barman:

Листинг 10.34 Установка barman

```
Line 1 $ tar -xzf barman-1.3.2.tar.gz
- $ cd barman-1.3.2/
- $ ./setup.py build
- $ sudo ./setup.py install
```

Или используя PostgreSQL Community APT репозиторий:

Листинг 10.35 Установка barman

```
Line 1 $ apt-get install barman
```

Теперь перейдем к серверу с PostgreSQL. Для того, чтобы barman мог подключаться к базе данных без проблем, нам нужно выставить настройки доступа в конфигах PostgreSQL:

Листинг 10.36 Отредактировать в postgresql.conf

```
Line 1 listen_adress = '*'
```

Листинг 10.37 Добавить в pg_hba.conf

```
Line 1 host    all    all    brhost/32    trust
```

После этих изменений нужно перегрузить PostgreSQL. Теперь можем проверить с сервера бэкапов подключение к PostgreSQL:

Листинг 10.38 Проверка подключения к базе

```
Line 1 $ psql -c 'SELECT version()' -U postgres -h pghost
-                                                  version
- ------------------------------------------------------------------------------

-  PostgreSQL 9.3.1 on x86_64-unknown-linux-gnu, compiled by
     gcc (Ubuntu/Linaro 4.7.2-2ubuntu1) 4.7.2, 64-bit
5 (1 row)
```

Далее создадим папку на сервере с бэкапами для хранения этих самых бэкапов:

Листинг 10.39 Папка для хранения бэкапов

```
$ sudo mkdir -p /srv/barman
$ sudo chown barman:barman /srv/barman
```

Для настройки barman создадим /etc/barman.conf:

Листинг 10.40 barman.conf

```
[barman]
; Main directory
barman_home = /srv/barman

; Log location
log_file = /var/log/barman/barman.log

; Default compression level: possible values are None (
    default), bzip2, gzip or custom
compression = gzip

; 'main' PostgreSQL Server configuration
[main]
; Human readable description
description =  "Main PostgreSQL Database"

; SSH options
ssh_command = ssh postgres@pghost

; PostgreSQL connection string
conninfo = host=pghost user=postgres
```

Секция «main» (так мы назвали для barman наш PostgreSQL сервер) содержит настроки для подключения к PostgreSQL серверу и базе. Проверим настройки:

Листинг 10.41 Проверка barman настроек

```
$ barman show-server main
Server main:
        active: true
        description: Main PostgreSQL Database
        ssh_command: ssh postgres@pghost
        conninfo: host=pghost user=postgres
        backup_directory: /srv/barman/main
        basebackups_directory: /srv/barman/main/base
        wals_directory: /srv/barman/main/wals
        incoming_wals_directory: /srv/barman/main/incoming
        lock_file: /srv/barman/main/main.lock
```

```
-         compression: gzip
-         custom_compression_filter: None
-         custom_decompression_filter: None
15        retention_policy: None
-         wal_retention_policy: None
-         pre_backup_script: None
-         post_backup_script: None
-         current_xlog: None
20        last_shipped_wal: None
-         archive_command: None
-         server_txt_version: 9.3.1
-         data_directory: /var/lib/postgresql/9.3/main
-         archive_mode: off
25        config_file: /etc/postgresql/9.3/main/postgresql.
    conf
-         hba_file: /etc/postgresql/9.3/main/pg_hba.conf
-         ident_file: /etc/postgresql/9.3/main/pg_ident.conf
-
- # barman check main
30 Server main:
-         ssh: OK
-         PostgreSQL: OK
-         archive_mode: FAILED (please set it to 'on')
-         archive_command: FAILED (please set it accordingly
    to documentation)
35        directories: OK
-         compression settings: OK
```

Все хорошо, вот только PostgreSQL не настроен. Для этого на сервере с PostgreSQL отредактируем конфиг базы:

Листинг 10.42 Настройка PostgreSQL

```
Line 1 wal_level = hot_standby # archive для PostgreSQL < 9.0
-  archive_mode = on
-  archive_command = 'rsync -a %p barman@brhost:
       INCOMING_WALS_DIRECTORY/%f'
```

где INCOMING_WALS_DIRECTORY — директория для складывания WAL-логов. Её можно узнать из вывода команды barman show-server main (листинг 10.41, указано /srv/barman/main/incoming). После изменения настроек нужно перегрузить PostgreSQL. Теперь проверим статус на сервере бэкапов:

Листинг 10.43 Проверка

```
Line 1 $ barman check main
-  Server main:
-          ssh: OK
-          PostgreSQL: OK
```

```
5           archive_mode: OK
-           archive_command: OK
-           directories: OK
-           compression settings: OK
```

Все готово. Для добавления нового сервера процедуру потребуется повторить, а в barman.conf добавить новый сервер.

Создание бэкапов

Получение списка серверов:

Листинг 10.44 Список серверов

```
Line 1 $ barman list-server
-   main - Main PostgreSQL Database
```

Запуск создания резервной копии PostgreSQL (сервер указывается последним параметром):

Листинг 10.45 Создание бэкапа

```
Line 1 $ barman backup main
-   Starting backup for server main in /srv/barman/main/base
        /20121109T090806
-   Backup start at xlog location: 0/3000020
        (000000010000000000000003, 00000020)
-   Copying files.
5   Copy done.
-   Asking PostgreSQL server to finalize the backup.
-   Backup end at xlog location: 0/30000D8
        (000000010000000000000003, 000000D8)
-   Backup completed
```

Такую задачу лучше выполнять раз в сутки (добавить в cron).
Посмотреть список бэкапов для указаной базы:

Листинг 10.46 Список бэкапов

```
Line 1 $ barman list-backup main
-   main 20121110T091608 - Fri Nov 10 09:20:58 2012 - Size: 1.0
        GiB - WAL Size: 446.0 KiB
-   main 20121109T090806 - Fri Nov  9 09:08:10 2012 - Size: 23.0
        MiB - WAL Size: 477.0 MiB
```

Более подробная информация о выбраной резервной копии:

Листинг 10.47 Информация о выбраной резервной копии

```
Line 1 $ barman show-backup main 20121110T091608
-   Backup 20121109T091608:
-     Server Name        : main
-     Status:            : DONE
```

```
  PostgreSQL Version: 90201
  PGDATA directory  : /var/lib/postgresql/9.3/main

  Base backup information:
    Disk usage      : 1.0 GiB
    Timeline        : 1
    Begin WAL       : 000000010000000000000008C
    End WAL         : 000000010000000000000092
    WAL number      : 7
    Begin time      : 2012-11-10 09:16:08.856884
    End time        : 2012-11-10 09:20:58.478531
    Begin Offset    : 32
    End Offset      : 3576096
    Begin XLOG      : 0/8C000020
    End XLOG        : 0/92369120

  WAL information:
    No of files     : 1
    Disk usage      : 446.0 KiB
    Last available  : 000000010000000000000093

  Catalog information:
    Previous Backup : 20121109T090806
    Next Backup     : - (this is the latest base backup)
```

Также можно сжимать WAL-логи, которые накапливаются в каталогах командой «cron»:

Листинг 10.48 Архивирование WAL-логов

```
$ barman cron
Processing xlog segments for main
        000000010000000000000001
        000000010000000000000002
        000000010000000000000003
        000000010000000000000003.00000020.backup
        000000010000000000000004
        000000010000000000000005
        000000010000000000000006
```

Эту команду требуется добавлять в cron. Частота выполнения данной команды зависит от того, как много WAL-логов накапливается (чем больше файлов - тем дольше она выполняется). Barman может сжимать WAL-логи через gzip, bzip2 или другой компрессор данных (команды для сжатия и распаковки задаются через custom_compression_filter и custom_decompression_filter соответственно). Также можно активировать компрессию данных при передачи по сети через опцию network_compression (по умолчанию отключена). Через опции bandwidth_limit (по умолчанию

0, ограничений нет) и tablespace_bandwidth_limit возможно ограничить использования сетевого канала.

Для восстановления базы из бэкапа используется команда recover:

Листинг 10.49 Восстановление базы

```
Line 1 $ barman recover --remote-ssh-command "ssh postgres@pghost"
     main 20121109T090806 /var/lib/postgresql/9.3/main
   - Starting remote restore for server main using backup
     20121109T090806
   - Destination directory: /var/lib/postgresql/9.3/main
   - Copying the base backup.
 5 Copying required wal segments.
   - The archive_command was set to 'false' to prevent data
     losses.
   -
   - Your PostgreSQL server has been successfully prepared for
     recovery!
   -
10 Please review network and archive related settings in the
     PostgreSQL
   - configuration file before starting the just recovered
     instance.
   -
   - WARNING: Before starting up the recovered PostgreSQL server,
   - please review also the settings of the following
     configuration
15 options as they might interfere with your current recovery
     attempt:
   -
   -     data_directory = '/var/lib/postgresql/9.3/main'
               # use data in another directory
   -     external_pid_file = '/var/run/postgresql/9.3-main.pid'
                   # write an extra PID file
   -     hba_file = '/etc/postgresql/9.3/main/pg_hba.conf'    #
     host-based authentication file
20     ident_file = '/etc/postgresql/9.3/main/pg_ident.conf'
             # ident configuration file
```

Barman может восстановить базу из резервной копии на удаленном сервере через SSH (для этого есть опция remote-ssh-command). Также barman может восстановить базу используя PITR: для этого используются опции target-time (указывается время) или target-xid (id транзакции).

Заключение

Barman помогает автоматизировать сбор и хранение резервных копий PostgreSQL данных на отдельном сервере. Утилита проста, позволяет хранить и удобно управлять бэкапами нескольких PostgreSQL серверов.

10.6 Заключение

В любом случае, усилия и время, затраченные на создание оптимальной системы создания бэкапов, будут оправданы. Невозможно предугадать когда произойдут проблемы с базой данных, поэтому бэкапы должны быть настроены для PostgreSQL (особенно, если это продакшн система).

11

Стратегии масштабирования для PostgreSQL

> То, что мы называем замыслом (стратегией), означает избежать бедствия и получить выгоду.
>
> У-цзы

> В конце концов, все решают люди, не стратегии.
>
> Ларри Боссиди

11.1 Введение

Многие разработчики крупных проектов сталкиваются с проблемой, когда один-единственный сервер базы данных никак не может справиться с нагрузками. Очень часто такие проблемы происходят из-за неверного проектирования приложения(плохая структура БД для приложения, отсутствие кеширования). Но в данном случае пусть у нас есть «идеальное» приложение, для которого оптимизированы все SQL запросы, используется кеширование, PostgreSQL настроен, но все равно не справляется с нагрузкой. Такая проблема может возникнуть как на этапе проектирования, так и на этапе роста приложения. И тут возникает вопрос: какую стратегию выбрать при возникновении подобной ситуации?

Если Ваш заказчик готов купить супер сервер за несколько тысяч долларов (а по мере роста — десятков тысяч и т.д.), чтобы сэкономить время разработчиков, но сделать все быстро, можете дальше эту главу не читать. Но такой заказчик — мифическое существо и, в основном, такая проблема ложится на плечи разработчиков.

Суть проблемы

Для того, что-бы сделать какой-то выбор, необходимо знать суть проблемы. Существуют два предела, в которые могут уткнуться сервера баз данных:

- Ограничение пропускной способности чтения данных;
- Ограничение пропускной способности записи данных.

Практически никогда не возникает одновременно две проблемы, по крайне мере, это маловероятно (если вы конечно не Twitter или Facebook пишете). Если вдруг такое происходит — возможно система неверно спроектирована, и её реализацию следует пересмотреть.

11.2 Проблема чтения данных

Проблема с чтением данных обычно начинается, когда СУБД не в состоянии обеспечить то количество выборок, которое требуется. В основном такое происходит в блогах, новостных лентах и т.д. Хочу сразу отметить, что подобную проблему лучше решать внедрением кеширования, а потом уже думать как масштабировать СУБД.

Методы решения

- PgPool-II v.3 + PostgreSQL v.9 с Streaming Replication — отличное решение для масштабирования на чтение, более подробно можно ознакомится по ссылке. Основные преимущества:
  - Низкая задержка репликации между мастером и слейвом;
  - Производительность записи падает незначительно;
  - Отказоустойчивость (failover);
  - Пулы соединений;
  - Интеллектуальная балансировка нагрузки – проверка задержки репликации между мастером и слейвом (сам проверяет pg_current_xlog_location и pg_last_xlog_receive_location);
  - Добавление слейвов СУБД без остановки pgpool-II;
  - Простота в настройке и обслуживании.
- PgPool-II v.3 + PostgreSQL с Slony/Londiste/Bucardo — аналогично предыдущему решению, но с использованием Slony/Londiste/Bucardo. Основные преимущества:
  - Отказоустойчивость (failover);
  - Пулы соединений;
  - Интеллектуальная балансировка нагрузки – проверка задержки репликации между мастером и слейвом;
  - Добавление слейв СУБД без остановки pgpool-II;
  - Можно использовать Postgresql ниже 9 версии.
- Postgres-XC — подробнее можно прочитать в «5.3 Postgres-XC» главе.

11.3 Проблема записи данных

Обычно такая проблема возникает в системах, которые производят анализ больших объемов данных (например ваш аналог Google Analytics). Данные активно пишутся и мало читаются (или читается только суммарный вариант собранных данных).

Методы решения

Один из самых популярных методов решение проблем — размазать нагрузку по времени с помощью систем очередей.

- PgQ — это система очередей, разработанная на базе PostgreSQL. Разработчики — компания Skype. Используется в Londiste (подробнее «4.5 Londiste»). Особенности:
  - Высокая производительность благодаря особенностям PostgreSQL;
  - Общая очередь, с поддержкой нескольких обработчиков и нескольких генераторов событий;
  - PgQ гарантирует, что каждый обработчик увидит каждое событие как минимум один раз;
  - События достаются из очереди «пачками» (batches);
  - Чистое API на SQL функциях;
  - Удобный мониторинг.

  Также можно воспользоваться еще одной утилитой — RabbitMQ. RabbitMQ — платформа, реализующая систему обмена сообщениями между компонентами программной системы (Message Oriented Middleware) на основе стандарта AMQP (Advanced Message Queuing Protocol). RabbitMQ выпускается под Mozilla Public License. RabbitMQ создан на основе испытанной Open Telecom Platform, обеспечивающий высокую надёжность и производительность промышленного уровня и написан на языке Erlang.
- Postgres-XC — подробнее можно прочитать в «5.3 Postgres-XC» главе.

11.4 Заключение

В данной главе показаны только несколько возможных вариантов решения задач масштабирования PostgreSQL. Таких стратегий существует огромное количество и каждая из них имеет как сильные, так и слабые стороны. Самое важное то, что выбор оптимальной стратегии масштабирования для решения поставленных задач остается на плечах разработчиков и/или администраторов СУБД.

12

Советы по разным вопросам (Performance Snippets)

Быстро найти правильный ответ на трудный вопрос — ни с чем не сравнимое удовольствие.

Макс Фрай. Обжора-Хохотун

– Вопрос риторический.
– Нет, но он таким кажется, потому что у тебя нет ответа.

Доктор Хаус (House M.D.), сезон 1 серия 1

12.1 Введение

Иногда возникают очень интересные проблемы по работе с PostgreSQL, которые при нахождении ответа поражают своей лаконичностью, красотой и простым исполнением (а может и не простым). В данной главе я решил собрать интересные методы решения разных проблем, с которыми сталкиваются люди при работе с PostgreSQL. Я не являюсь огромным специалистом по данной теме, поэтому многие решения мне помогали находить люди из PostgreSQL комьюнити, а иногда хватало и поиска в интернете.

12.2 Советы

Размер объектов в базе данных

Данный запрос показывает размер объектов в базе данных (например, таблиц и индексов).

Скачать snippets/biggest_relations.sql

```
SELECT nspname || '.' || relname AS "relation",
    pg_size_pretty(pg_relation_size(C.oid)) AS "size"
  FROM pg_class C
  LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
  WHERE nspname NOT IN ('pg_catalog', 'information_schema')
  ORDER BY pg_relation_size(C.oid) DESC
  LIMIT 20;
```

Пример вывода:

Листинг 12.1 Поиск самых больших объектов в БД. Пример вывода

```
        relation         |    size
-------------------------+------------
 public.accounts         | 326 MB
 public.accounts_pkey    | 44 MB
 public.history          | 592 kB
 public.tellers_pkey     | 16 kB
 public.branches_pkey    | 16 kB
 public.tellers          | 16 kB
 public.branches         | 8192 bytes
```

Размер самых больших таблиц

Данный запрос показывает размер самых больших таблиц в базе данных.

Скачать snippets/biggest_tables.sql

```
SELECT nspname || '.' || relname AS "relation",
    pg_size_pretty(pg_total_relation_size(C.oid)) AS "
   total_size"
  FROM pg_class C
  LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
  WHERE nspname NOT IN ('pg_catalog', 'information_schema')
    AND C.relkind <> 'i'
    AND nspname !~ '^pg_toast'
  ORDER BY pg_total_relation_size(C.oid) DESC
  LIMIT 20;
```

Пример вывода:

Листинг 12.2 Размер самых больших таблиц. Пример вывода

```
             relation              | total_size
-----------------------------------+------------
 public.actions                    | 4249 MB
 public.product_history_records    | 197 MB
 public.product_updates            | 52 MB
```

```
-  public.import_products          | 34 MB
-  public.products                 | 29 MB
-  public.visits                   | 25 MB
```

«Средний» count

Данный метод позволяет узнать приблизительное количество записей в таблице. Для огромных таблиц этот метод работает быстрее, чем обыкновенный count.

Скачать snippets/count_estimate.sql

```
Line 1  CREATE FUNCTION count_estimate(query text) RETURNS integer
        AS $$
-  DECLARE
-      rec   record;
-      rows  integer;
5  BEGIN
-      FOR rec IN EXECUTE 'EXPLAIN ' || query LOOP
-          rows := substring(rec."QUERY PLAN" FROM ' rows=([[:
       digit:]]+)');
-          EXIT WHEN rows IS NOT NULL;
-      END LOOP;
10
-      RETURN rows;
-  END;
-  $$ LANGUAGE plpgsql VOLATILE STRICT;
```

Пример:

Листинг 12.3 «Средний» count. Пример

```
Line 1  CREATE TABLE foo (r double precision);
-  INSERT INTO foo SELECT random() FROM generate_series(1,
       1000);
-  ANALYZE foo;
-
5  # SELECT count(*) FROM foo WHERE r < 0.1;
-   count
-  -------
-      92
-  (1 row)
10
-  # SELECT count_estimate('SELECT * FROM foo WHERE r < 0.1');
-   count_estimate
-  ----------------
-                94
15 (1 row)
```

Узнать значение по умолчанию у поля в таблице

Данный метод позволяет узнать значение по умолчанию у поля в таблице (заданное через DEFAULT).

Скачать snippets/default_value.sql

```
Line 1  CREATE FUNCTION ret_def(text,text,text) RETURNS text AS $$
  -     SELECT
  -       COLUMNS.column_default::text
  -     FROM
  5       information_schema.COLUMNS
  -       WHERE table_name = $2
  -       AND table_schema = $1
  -       AND column_name = $3
  -     $$ LANGUAGE sql IMUTABLE;
```

Пример:

Листинг 12.4 Узнать значение по умолчанию у поля в таблице. Пример

```
Line 1  # SELECT ret_def('schema','table','column');
  -
  -     SELECT ret_def('public','image_files','id');
  -                      ret_def
  5     ----------------------------------------
  -      nextval('image_files_id_seq'::regclass)
  -     (1 row)
  -
  -     SELECT ret_def('public','schema_migrations','version');
  10     ret_def
  -     ---------
  -
  -     (1 row)
```

Случайное число из диапазона

Данный метод позволяет взять случайное число из указаного диапазона (целое или с плавающей запятой).

Скачать snippets/random_from_range.sql

```
Line 1  CREATE OR REPLACE FUNCTION random(numeric, numeric)
  -     RETURNS numeric AS
  -     $$
  -       SELECT ($1 + ($2 - $1) * random())::numeric;
  5     $$ LANGUAGE 'sql' VOLATILE;
```

Пример:

Листинг 12.5 Случайное число из диапазона. Пример

```
Line 1  SELECT random(1,10)::int, random(1,10);
-  random |       random
-  --------+------------------
-        6 | 5.11675184825435
5  (1 row)
-
-  SELECT random(1,10)::int, random(1,10);
-  random |       random
-  --------+------------------
10       7 | 1.37060070643201
-  (1 row)
```

Алгоритм Луна

Алгоритм Луна или формула Луна — алгоритм вычисления контрольной цифры, получивший широкую популярность. Он используется, в частности, при первичной проверке номеров банковских пластиковых карт, номеров социального страхования в США и Канаде. Алгоритм был разработан сотрудником компании «IBM» Хансом Петером Луном и запатентован в 1960 году.

Контрольные цифры вообще и алгоритм Луна в частности предназначены для защиты от случайных ошибок, а не преднамеренных искажений данных.

Алгоритм Луна реализован на чистом SQL. Обратите внимание, что эта реализация является чисто арифметической.

Скачать snippets/luhn_algorithm.sql

```
Line 1  CREATE OR REPLACE FUNCTION luhn_verify(int8) RETURNS BOOLEAN
        AS $$
-  -- Take the sum of the
-  -- doubled digits and the even-numbered undoubled digits,
       and see if
-  -- the sum is evenly divisible by zero.
5  SELECT
-          -- Doubled digits might in turn be two digits. In
       that case,
-          -- we must add each digit individually rather than
       adding the
-          -- doubled digit value to the sum. Ie if the
       original digit was
-          -- '6' the doubled result was '12' and we must add
       '1+2' to the
10         -- sum rather than '12'.
-          MOD(SUM(doubled_digit / INT8 '10' + doubled_digit %
        INT8 '10'), 10) = 0
-  FROM
```

```
-- Double odd-numbered digits (counting left with
-- least significant as zero). If the doubled digits end up
-- having values
-- > 10 (ie they're two digits), add their digits together.
(SELECT
        -- Extract digit 'n' counting left from least
    significant
        -- as zero
        MOD( ( $1::int8 / (10^n)::int8 ), 10::int8)
        -- Double odd-numbered digits
        * (MOD(n,2) + 1)
        AS doubled_digit
        FROM generate_series(0, CEIL(LOG( $1 ))::INTEGER -
    1) AS n
) AS doubled_digits;

$$ LANGUAGE 'SQL'
IMMUTABLE
STRICT;

COMMENT ON FUNCTION luhn_verify(int8) IS 'Return true iff
    the last digit of the
input is a correct check digit for the rest of the input
    according to Luhn''s
algorithm.';
CREATE OR REPLACE FUNCTION luhn_generate_checkdigit(int8)
    RETURNS int8 AS $$
SELECT
    -- Add the digits, doubling even-numbered digits (
    counting left
    -- with least-significant as zero). Subtract the
    remainder of
    -- dividing the sum by 10 from 10, and take the
    remainder
    -- of dividing that by 10 in turn.
    ((INT8 '10' - SUM(doubled_digit / INT8 '10' +
    doubled_digit % INT8 '10') %
                    INT8 '10') % INT8 '10')::INT8
FROM (SELECT
        -- Extract digit 'n' counting left from least
    significant\
        -- as zero
        MOD( ($1::int8 / (10^n)::int8), 10::int8 )
        -- double even-numbered digits
        * (2 - MOD(n,2))
        AS doubled_digit
        FROM generate_series(0, CEIL(LOG($1))::INTEGER - 1)
```

```
    AS n
) AS doubled_digits;

$$ LANGUAGE 'SQL'
IMMUTABLE
STRICT;

COMMENT ON FUNCTION luhn_generate_checkdigit(int8) IS 'For the input
value, generate a check digit according to Luhn''s algorithm';
CREATE OR REPLACE FUNCTION luhn_generate(int8) RETURNS int8 AS $$
SELECT 10 * $1 + luhn_generate_checkdigit($1);
$$ LANGUAGE 'SQL'
IMMUTABLE
STRICT;

COMMENT ON FUNCTION luhn_generate(int8) IS 'Append a check digit generated
according to Luhn''s algorithm to the input value. The input value must be no
greater than (maxbigint/10).';
CREATE OR REPLACE FUNCTION luhn_strip(int8) RETURNS int8 AS $$
SELECT $1 / 10;
$$ LANGUAGE 'SQL'
IMMUTABLE
STRICT;

COMMENT ON FUNCTION luhn_strip(int8) IS 'Strip the least significant digit from
the input value. Intended for use when stripping the check digit from a number
including a Luhn''s algorithm check digit.';
```

Пример:

Листинг 12.6 Алгоритм Луна. Пример

```
Line 1  Select luhn_verify(49927398716);
        luhn_verify
        -------------
        t
        (1 row)

        Select luhn_verify(49927398714);
        luhn_verify
        -------------
```

```
10   f
- (1 row)
```

Выборка и сортировка по данному набору данных

Выбор данных по определенному набору данных можно сделать с помощью обыкновенного IN. Но как сделать подобную выборку и отсортировать данные в том же порядке, в котором передан набор данных. Например:

Дан набор: (2,6,4,10,25,7,9). Нужно получить найденные данные в таком же порядке т.е. 2 2 2 6 6 4 4

Скачать snippets/order_like_in.sql

```
Line 1 SELECT foo.* FROM foo
- JOIN (SELECT id.val, row_number() over() FROM (VALUES(3),(2)
      ,(6),(1),(4)) AS
- id(val)) AS id
- ON (foo.catalog_id = id.val) ORDER BY row_number;
```

где

VALUES(3),(2),(6),(1),(4) — наш набор данных

foo – таблица, из которой идет выборка

foo.catalog_id — поле, по которому ищем набор данных (замена foo.catalog_id IN(3,2,6,1,4))

Quine — запрос который выводит сам себя

Куайн, квайн (англ. quine) — компьютерная программа (частный случай метапрограммирования), которая выдаёт на выходе точную копию своего исходного текста.

Скачать snippets/quine.sql

```
Line 1 select a || ' from (select ' || quote_literal(a) || b || ',
      ' || quote_literal(b) || '::text as b) as quine' from
- (select 'select a || '' from (select '' || quote_literal(a)
      || b || '', '' || quote_literal(b) || ''::text as b) as
- quine'''::text as a, '::text as a'::text as b) as quine;
```

Ускоряем LIKE

Автокомплит — очень популярная фишка в web системах. Реализуется это простым LIKE 'some\%', где «some» — то, что пользователь успел ввести. Проблема в том, что и в огромной таблице (например таблица тегов) такой запрос будет работать очень медленно.

Для ускорения запроса типа «LIKE 'bla%'» можно использовать text_pattern_ops (или varchar_pattern_ops если у поле varchar).

Скачать snippets/speed_like.sql

```
Line 1  prefix_test=# create table tags (
  -     prefix_test(#   tag     text primary key,
  -     prefix_test(#   name        text not null,
  -     prefix_test(#   shortname text,
  5     prefix_test(#   status      char default 'S',
  -     prefix_test(#
  -     prefix_test(#   check( status in ('S', 'R') )
  -     prefix_test(# );
  -     NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit
        index "tags_pkey" for table "tags"
 10     CREATE TABLE
  -     prefix_test=# CREATE INDEX i_tag ON tags USING btree(lower(
        tag)  text_pattern_ops);
  -     CREATE INDEX
  -
  -     prefix_test=# create table invalid_tags (
 15     prefix_test(#   tag     text primary key,
  -     prefix_test(#   name        text not null,
  -     prefix_test(#   shortname text,
  -     prefix_test(#   status      char default 'S',
  -     prefix_test(#
 20     prefix_test(#   check( status in ('S', 'R') )
  -     prefix_test(# );
  -     NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit
        index "invalid_tags_pkey" for table "invalid_tags"
  -     CREATE TABLE
  -
 25
  -
  -     prefix_test=# select count(*) from tags;
  -      count
  -     -------
 30      11966
  -     (1 row)
  -
  -     prefix_test=# select count(*) from invalid_tags;
  -      count
 35     -------
  -      11966
  -     (1 row)
  -
  -     # EXPLAIN ANALYZE select * from invalid_tags where lower(tag
        ) LIKE lower('0146%');
 40                                                          QUERY PLAN
  -     --
        ------------------------------------------------------------------------
```

```
-  Seq Scan on invalid_tags  (cost=0.00..265.49 rows=60 width
     =26) (actual time=0.359..20.695 rows=1 loops=1)
-    Filter: (lower(tag) ~~ '0146%'::text)
-  Total runtime: 20.803 ms
45 (3 rows)
-
- # EXPLAIN ANALYZE select * from invalid_tags where lower(tag
     ) LIKE lower('0146%');
-                                                      QUERY PLAN
- --
     ------------------------------------------------------------------------

50  Seq Scan on invalid_tags  (cost=0.00..265.49 rows=60 width
     =26) (actual time=0.549..19.503 rows=1 loops=1)
-    Filter: (lower(tag) ~~ '0146%'::text)
-  Total runtime: 19.550 ms
- (3 rows)
-
55 # EXPLAIN ANALYZE select * from tags where lower(tag) LIKE
     lower('0146%');
-                                                        QUERY
     PLAN
- --
     ------------------------------------------------------------------------

-  Bitmap Heap Scan on tags  (cost=5.49..97.75 rows=121 width
     =26) (actual time=0.054..0.057 rows=1 loops=1)
-    Filter: (lower(tag) ~~ '0146%'::text)
60   ->  Bitmap Index Scan on i_tag (cost=0.00..5.46 rows=120
     width=0) (actual time=0.032..0.032 rows=1 loops=1)
-          Index Cond: ((lower(tag) ~>=~ '0146'::text) AND (
     lower(tag) ~<~ '0147'::text))
-  Total runtime: 0.119 ms
- (5 rows)
-
65 # EXPLAIN ANALYZE select * from tags where lower(tag) LIKE
     lower('0146%');
-                                                        QUERY
     PLAN
- --
     ------------------------------------------------------------------------

-  Bitmap Heap Scan on tags  (cost=5.49..97.75 rows=121 width
     =26) (actual time=0.025..0.025 rows=1 loops=1)
-    Filter: (lower(tag) ~~ '0146%'::text)
70   ->  Bitmap Index Scan on i_tag (cost=0.00..5.46 rows=120
```

```
    width=0) (actual time=0.016..0.016 rows=1 loops=1)
-         Index Cond: ((lower(tag) ~>=~ '0146'::text) AND (
    lower(tag) ~<~ '0147'::text))
-  Total runtime: 0.050 ms
- (5 rows)
```

Поиск дубликатов индексов

Запрос находит индексы, созданные на одинаковый набор столбцов (такие индексы эквивалентны, а значит бесполезны).

Скачать snippets/duplicate_indexes.sql

```
Line 1  SELECT pg_size_pretty(sum(pg_relation_size(idx))::bigint) AS
        size,
-           (array_agg(idx))[1] AS idx1, (array_agg(idx))[2] AS
    idx2,
-           (array_agg(idx))[3] AS idx3, (array_agg(idx))[4] AS
    idx4
- FROM (
5       SELECT indexrelid::regclass AS idx, (indrelid::text ||E'
    \n'|| indclass::text ||E'\n'|| indkey::text ||E'\n'||
-                                                   coalesce(indexprs::
    text,'')||E'\n' || coalesce(indpred::text,'')) AS KEY
-       FROM pg_index) sub
- GROUP BY KEY HAVING count(*)>1
- ORDER BY sum(pg_relation_size(idx)) DESC;
```

Размер и статистика использования индексов

Скачать snippets/indexes_statustic.sql

```
Line 1  SELECT
-       t.tablename,
-       indexname,
-       c.reltuples AS num_rows,
5       pg_size_pretty(pg_relation_size(quote_ident(t.tablename)
        ::text)) AS table_size,
-       pg_size_pretty(pg_relation_size(quote_ident(indexrelname
        )::text)) AS index_size,
-       CASE WHEN x.is_unique = 1  THEN 'Y'
-           ELSE 'N'
-       END AS UNIQUE,
10      idx_scan AS number_of_scans,
-       idx_tup_read AS tuples_read,
-       idx_tup_fetch AS tuples_fetched
- FROM pg_tables t
```

```
- LEFT OUTER JOIN pg_class c ON t.tablename=c.relname
15 LEFT OUTER JOIN
-         (SELECT indrelid,
-             max(CAST(indisunique AS integer)) AS is_unique
-         FROM pg_index
-         GROUP BY indrelid) x
20         ON c.oid = x.indrelid
- LEFT OUTER JOIN
-      ( SELECT c.relname AS ctablename, ipg.relname AS
    indexname, x.indnatts AS number_of_columns, idx_scan,
    idx_tup_read, idx_tup_fetch,indexrelname FROM pg_index x
-             JOIN pg_class c ON c.oid = x.indrelid
-             JOIN pg_class ipg ON ipg.oid = x.indexrelid
25             JOIN pg_stat_all_indexes psai ON x.indexrelid =
    psai.indexrelid )
-      AS foo
-      ON t.tablename = foo.ctablename
- WHERE t.schemaname='public'
- ORDER BY 1,2;
```

Размер распухания (bloat) таблиц и индексов в базе данных

Запрос, который показывает «приблизительный» bloat (раздутие) таблиц и индексов в базе:

Скачать snippets/bloating.sql

```
Line 1 WITH constants AS (
-   SELECT current_setting('block_size')::numeric AS bs, 23 AS
      hdr, 4 AS ma
- ), bloat_info AS (
-   SELECT
5     ma,bs,schemaname,tablename,
-     (datawidth+(hdr+ma-(case when hdr%ma=0 THEN ma ELSE hdr%
    ma END)))::numeric AS datahdr,
-     (maxfracsum*(nullhdr+ma-(case when nullhdr%ma=0 THEN ma
    ELSE nullhdr%ma END))) AS nullhdr2
-   FROM (
-     SELECT
10       schemaname, tablename, hdr, ma, bs,
-       SUM((1-null_frac)*avg_width) AS datawidth,
-       MAX(null_frac) AS maxfracsum,
-       hdr+(
-         SELECT 1+count(*)/8
15         FROM pg_stats s2
-         WHERE null_frac<>0 AND s2.schemaname = s.schemaname
    AND s2.tablename = s.tablename
-       ) AS nullhdr
-     FROM pg_stats s, constants
```

```
-       GROUP BY 1,2,3,4,5
20    ) AS foo
- ), table_bloat AS (
-   SELECT
-     schemaname, tablename, cc.relpages, bs,
-     CEIL((cc.reltuples*((datahdr+ma-
25        (CASE WHEN datahdr%ma=0 THEN ma ELSE datahdr%ma END))+
      nullhdr2+4))/(bs-20::float)) AS otta
-   FROM bloat_info
-   JOIN pg_class cc ON cc.relname = bloat_info.tablename
-   JOIN pg_namespace nn ON cc.relnamespace = nn.oid AND nn.
      nspname = bloat_info.schemaname AND nn.nspname <> '
      information_schema'
- ), index_bloat AS (
30   SELECT
-     schemaname, tablename, bs,
-     COALESCE(c2.relname,'?') AS iname, COALESCE(c2.reltuples
      ,0) AS ituples, COALESCE(c2.relpages,0) AS ipages,
-     COALESCE(CEIL((c2.reltuples*(datahdr-12))/(bs-20::float)
      ),0) AS iotta -- very rough approximation, assumes all
      cols
-   FROM bloat_info
35   JOIN pg_class cc ON cc.relname = bloat_info.tablename
-   JOIN pg_namespace nn ON cc.relnamespace = nn.oid AND nn.
      nspname = bloat_info.schemaname AND nn.nspname <> '
      information_schema'
-   JOIN pg_index i ON indrelid = cc.oid
-   JOIN pg_class c2 ON c2.oid = i.indexrelid
- )
40 SELECT
-   type, schemaname, object_name, bloat, pg_size_pretty(
     raw_waste) as waste
- FROM
- (SELECT
-   'table' as type,
45   schemaname,
-   tablename as object_name,
-   ROUND(CASE WHEN otta=0 THEN 0.0 ELSE table_bloat.relpages/
     otta::numeric END,1) AS bloat,
-   CASE WHEN relpages < otta THEN '0' ELSE (bs*(table_bloat.
     relpages-otta)::bigint)::bigint END AS raw_waste
- FROM
50   table_bloat
-     UNION
- SELECT
-   'index' as type,
-   schemaname,
```

```
55    tablename || '::' || iname as object_name,
 -    ROUND(CASE WHEN iotta=0 OR ipages=0 THEN 0.0 ELSE ipages/
       iotta::numeric END,1) AS bloat,
 -    CASE WHEN ipages < iotta THEN '0' ELSE (bs*(ipages-iotta))
       ::bigint END AS raw_waste
 - FROM
 -    index_bloat) bloat_summary
60 ORDER BY raw_waste DESC, bloat DESC
```

Литература

[1] Алексей Борзов (Sad Spirit) borz_off@cs.msu.su PostgreSQL: настройка производительности http://www.phpclub.ru/detail/store/pdf/postgresql-performance.pdf

[2] Eugene Kuzin eugene@kuzin.net Настройка репликации в PostgreSQL с помощью системы Slony-I http://www.kuzin.net/work/sloniki-privet.html

[3] Sergey Konoplev gray.ru@gmail.com Установка Londiste в подробностях http://gray-hemp.blogspot.com/2010/04/londiste.html

[4] Dmitry Stasyuk Учебное руководство по pgpool-II http://undenied.ru/2009/03/04/uchebnoe-rukovodstvo-po-pgpool-ii/

[5] Чиркин Дима dmitry.chirkin@gmail.com Горизонтальное масштабирование PostgreSQL с помощью PL/Proxy http://habrahabr.ru/blogs/postgresql/45475/

[6] Иван Блинков wordpress@insight-it.ru Hadoop http://www.insight-it.ru/masshtabiruemost/hadoop/

[7] Padraig O'Sullivan Up and Running with HadoopDB http://posulliv.github.com/2010/05/10/hadoopdb-mysql.html

[8] Иван Золотухин Масштабирование PostgreSQL: готовые решения от Skype http://postgresmen.ru/articles/view/25

[9] Streaming Replication. http://wiki.postgresql.org/wiki/Streaming_Replication

[10] Den Golotyuk Шардинг, партиционирование, репликация - зачем и когда? http://highload.com.ua/index.php/2009/05/06/шардинг-партиционирование-репликац/

[11] Postgres-XC — A PostgreSQL Clustering Solution http://www.linuxforu.com/2012/01/postgres-xc-database-clustering-solution/

[12] Введение в PostgreSQL BDR http://habrahabr.ru/post/227959/