Оглавление

Глава 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.

Глава 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, вебсервер).

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 было внедрено метки востановления, улучшение управления буфером, CHECKPOINT и VACUUM улучшены.
- В версии 8.1 было улучшено одновременный доступ к разделяемой памяти, автоматически использование индексов для MIN() и MAX(), pg_autovacuum внедрен в сервер (автоматизирован), повышение производительности для секционированных таблиц.
- В версии 8.2 было улучшено 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 было внедрено общие табличные выражения, рекурсивные запросы, параллельное восстановление, улучшенна производительность для EXISTS/NOT EXISTS запросов.
- В версии 9.0 «репликация из коробки», VACUUM/VACUUM FULL стали быстрее, расширены хранимые процедуры.

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. пункт 3.4).

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 свопингу.

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

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 сортироки 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).

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).

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

- **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 установленны 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, растягиваяя 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_ 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 предел — 100 вляние 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[2]. При этом производительность упадёт из-за постоянного сбрасывания на диск данных из буфера.

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

Для увеличения интервала между контрольными точками нужно увеличить количество сегментов журнала транзакций (checkpoint_segments). Данный параметр определяет количество сегментов (каждый по 16 МБ) 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 и стоит ли его 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.

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 — вызывать fsync() после каждого commit игнорирую паралельные процессы
  - fsync — вызов fsync() после каждого commit
  - open_sync — запись данных методом open() с параметром O_SYN

  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 не достаточно, что бы полность восстановить данные после «падения». full_page_writes гарантирует корректное восстановление, ценой увелечения 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[3]. 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.

[3]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 ANALY (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[4].

  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

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

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 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 статьи, ограничимся уровнем файловой системы.

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

Вы легко можете получить выигрыш в производительности без побочных эффектов, если примонтируете файловую систему, содержащую базу данных, с параметром noatime[5].

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[6], 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.

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

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

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.

2.4 Примеры настроек

Среднестатистическая настройка для максимальной производительности

Возможно для конкретного случаю лучше подойдут другие настройки. Внимательно изучите данное руководство и настройте PostgreSQL опираясь на эту информацию.

RAM — размер памяти;

- shared_buffers = 1/8 RAM или больше (но не более 1/4);
- work_mem в 1/20 RAM;
- maintenance_work_mem в 1/4 RAM;
- max_fsm_relations в планируемое кол–во таблиц в базах * 1.5;
- max_fsm_pages в max_fsm_relations * 2000;
- fsync = true;
- wal_sync_method = fdatasync;
- commit_delay = от 10 до 100 ;
- commit_siblings = от 5 до 10;
- effective_cache_size = 0.9 от значения cached, которое показывает free;
- random_page_cost = 2 для быстрых cpu, 4 для медленных;
- cpu_tuple_cost = 0.001 для быстрых cpu, 0.01 для медленных;
- cpu_index_tuple_cost = 0.0005 для быстрых cpu, 0.005 для медленных;
- autovacuum = on;
- autovacuum_vacuum_threshold = 1800;
- autovacuum_analyze_threshold = 900;

Среднестатистическая настройка для оконного приложения (1С), 2 ГБ памяти

- maintenance_work_mem = 128MB
- effective_cache_size = 512MB

- work_mem = 640kB
- wal_buffers = 1536kB
- shared_buffers = 128MB
- max_connections = 500

Среднестатистическая настройка для Web приложения, 2 ГБ памяти

- maintenance_work_mem = 128MB;
- checkpoint_completion_target = 0.7
- effective_cache_size = 1536MB
- work_mem = 4MB
- wal_buffers = 4MB
- checkpoint_segments = 8
- shared_buffers = 512MB
- max_connections = 500

Среднестатистическая настройка для Web приложения, 8 ГБ памяти

- maintenance_work_mem = 512MB
- checkpoint_completion_target = 0.7
- effective_cache_size = 6GB
- work_mem = 16MB
- wal_buffers = 4MB
- checkpoint_segments = 8
- shared_buffers = 2GB
- max_connections = 500

2.5 Автоматическое создание оптимальных настроек: pgtune

Для оптимизации настроек для PostgreSQL Gregory Smith создал утилиту pgtune[7] в расчете на обеспечение максимальной производительности для заданной аппаратной конфигурации. Утилита проста в использовании и в многих Linux системах может идти в составе пакетов. Если же нет, можно просто скачать архив и распаковать. Для начала:

[7] http://pgtune.projects.postgresql.org/

Listing 2.1: Pgtune

```
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, это будет брать взависимости от типа базы данных.

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 сложность запросов, так что оптимально настроить базу данных возможно учитывая все эти параметры.

2.6 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) Граммотное проектирование базы. Освещение этого вопроса выходит далеко за рамки этой статьи.
   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 его 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 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 столк-

нувшись с медленным запросом, в первую очередь проверьте, существуют ли индексы, которые он может использовать. Если нет — постройте их. Излишек индексов, впрочем, тоже чреват проблемами:

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

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

Команда EXPLAIN [ANALYZE]

Команда EXPLAIN [запрос] показывает, каким образом PostgreSQL собирается выполнять ваш запрос. Команда EXPLAIN ANALYZE [запрос] выполняет запрос[8] и показывает как изначальный план, так и реальный процесс его выполнения.

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

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

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

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

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

```
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

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

и запрос вида

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

будет его использовать.

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

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

который будет использоваться запросом вида

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

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

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[9] 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 выбраных 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

Listing 2.2: SQL

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

```
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 следу- ющие подходы:

1. Если точное число записей не важно, а важен порядок[10], то можно использовать информацию о количестве записей в таблице, собранную при выполнении команды ANALYZE:

   Listing 2.3: SQL

   ```
   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.

Listing 2.4: DISTINCT

```
1 postgres=# select count(*) from (select distinct i from g) a;
2  count
3 -------
4  19125
5 (1 row)
```

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

```
Time: 580,553 ms


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

Time: 36,281 ms
```

Listing 2.5: 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
```

Оптимизация запросов с помощью pgFouine

pgFouine[11] — 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

[11] http://pgfouine.projects.postgresql.org/

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

Listing 2.6: pgFouine

```
1 log_destination = 'syslog'
2 redirect_stderr = off
3 silent_mode = on
```

- Для записи запросов, длящихся дольше n миллисекунд:

Listing 2.7: pgFouine

```
1 log_min_duration_statement = n
2 log_duration = off
3 log_statement = 'none'
```

Для записи каждого обработанного запроса установите log_min_duration 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:

Listing 2.8: pgFouine

```
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:

Listing 2.9: pgFouine

```
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 — http://pgfouine.projects.postg

2.7 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.

Глава 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 выпол-

няться 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 ключевые значения 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. Так же не нужно ставить никаких ограничений на таблицу, если конечно они не будут дублироватся 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 пересекатся. Например:

Listing 3.1: Пример неверного задлания значений партиций

```
1 CHECK ( outletID BETWEEN 100 AND 200 )
2 CHECK ( outletID BETWEEN 200 AND 300 )
```

неверно заданы партиции, поскольку не понятно какой партиции пренадлежит 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» таблици в соответствующую партицию.
- Убедиться, что параметр «constraint_exclusion» не отключен в postgres Если его не включить, то запросы не будут оптимизированы при работе с партиционирование.

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 таблица уже с милионами, а то, и милиардами 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 следующию таблицу:

Listing 3.2: «Мастер» таблица

```
1 CREATE TABLE my_logs (
2     id              SERIAL PRIMARY KEY,
3     user_id         INT NOT NULL,
4     logdate         TIMESTAMP NOT NULL,
5     data            TEXT,
6     some_state      INT
7 );
```

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):

Listing 3.3: «Дочерние» таблицы

```
1  CREATE TABLE my_logs2010m10 (
2      CHECK ( logdate >= DATE '2010-10-01' AND logdate < DATE
       '2010-11-01' )
3  ) INHERITS (my_logs);
4  CREATE TABLE my_logs2010m11 (
5      CHECK ( logdate >= DATE '2010-11-01' AND logdate < DATE
       '2010-12-01' )
6  ) INHERITS (my_logs);
7  CREATE TABLE my_logs2010m12 (
8      CHECK ( logdate >= DATE '2010-12-01' AND logdate < DATE
       '2011-01-01' )
9  ) INHERITS (my_logs);
10 CREATE TABLE my_logs2011m01 (
11     CHECK ( logdate >= DATE '2011-01-01' AND logdate < DATE
       '2010-02-01' )
12 ) INHERITS (my_logs);
```

Данными командами мы создаем таблицы «my_logs2010m10», «my_logs2
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 пересекатся!). Поскольку партиционирование будет работать по полю «logdate», мы создадим индекс на это поле на всех партициях:

Listing 3.4: Создание индексов

```
1  CREATE INDEX my_logs2010m10_logdate ON my_logs2010m10
       (logdate);
2  CREATE INDEX my_logs2010m11_logdate ON my_logs2010m11
       (logdate);
3  CREATE INDEX my_logs2010m12_logdate ON my_logs2010m12
       (logdate);
4  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.

Listing 3.5: Функция для перенаправления

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

Listing 3.6: Триггер

```
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»:

Listing 3.7: Данные

```
1 INSERT INTO my_logs (user_id,logdate, data, some_state)
    VALUES(1, '2010-10-30', '30.10.2010␣data', 1);
2 INSERT INTO my_logs (user_id,logdate, data, some_state)
    VALUES(2, '2010-11-10', '10.11.2010␣data2', 1);
3 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:

Listing 3.8: «Мастер» таблица чиста

```
1 partitioning_test=# SELECT * FROM ONLY my_logs;
2  id | user_id | logdate | data | some_state
3 ----+---------+---------+------+------------
4 (0 rows)
```

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

Listing 3.9: Проверка данных

```
1 partitioning_test=# SELECT * FROM my_logs;
2  id | user_id |       logdate       |       data       |
     some_state
3 ----+---------+---------------------+------------------+---------
4   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
6   3 |       1 | 2010-12-15 00:00:00 | 15.12.2010 data3 |
          1
7 (3 rows)
```

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

Listing 3.10: Проверка хранения данных

```
1 partitioning_test=# Select * from my_logs2010m10;
2  id | user_id |       logdate       |      data       |
     some_state
3 ----+---------+---------------------+-----------------+---------
4   1 |       1 | 2010-10-30 00:00:00 | 30.10.2010 data |
        1
5 (1 row)

7 partitioning_test=# Select * from my_logs2010m11;
```

```
 id | user_id |       logdate       |       data       |
   some_state
----+---------+---------------------+------------------+---------
  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:

Listing 3.11: Проверка запросов

```
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
(1 row)

partitioning_test=# SELECT * FROM my_logs WHERE data LIKE
   '%0.1%';
 id | user_id |       logdate       |       data       |
   some_state
----+---------+---------------------+------------------+---------
  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 выполить:

Listing 3.12: Чистка логов

```
DROP TABLE my_logs2008m10;
```

поскольку «DROP TABLE» работает гораздо быстрее, чем удаление милионов записей индивидуально через «DELETE». Другой вариант, который более предпочтителен, просто удалить партицию из партициониро-

вания, тем самым оставив данные в СУБД, но уже не доступные через «мастер» таблицу:

Listing 3.13: Удаляем партицию из партиционирования

```
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, выпоним простой запрос:

Listing 3.14: «constraint_exclusion» OFF

```
1  partitioning_test=# SET constraint_exclusion = off;
2  partitioning_test=# EXPLAIN SELECT * FROM my_logs WHERE
       logdate > '2010-12-01';
4                                                    QUERY PLAN
5  ------------------------------------------------------------------------
6   Result  (cost=6.81..104.66 rows=1650 width=52)
7     ->  Append  (cost=6.81..104.66 rows=1650 width=52)
8           ->  Bitmap Heap Scan on my_logs  (cost=6.81..20.93
       rows=330 width=52)
9                 Recheck Cond: (logdate > '2010-12-01␣
       00:00:00'::timestamp without time zone)
0                 ->  Bitmap Index Scan on my_logs_logdate
       (cost=0.00..6.73 rows=330 width=0)
1                       Index Cond: (logdate > '2010-12-01␣
       00:00:00'::timestamp without time zone)
2           ->  Bitmap Heap Scan on my_logs2010m10 my_logs
       (cost=6.81..20.93 rows=330 width=52)
3                 Recheck Cond: (logdate > '2010-12-01␣
       00:00:00'::timestamp without time zone)
4                 ->  Bitmap Index Scan on
       my_logs2010m10_logdate  (cost=0.00..6.73 rows=330 width=0)
5                       Index Cond: (logdate > '2010-12-01␣
       00:00:00'::timestamp without time zone)
6           ->  Bitmap Heap Scan on my_logs2010m11 my_logs
       (cost=6.81..20.93 rows=330 width=52)
```

```
7                 Recheck Cond: (logdate > '2010-12-01␣
    00:00:00'::timestamp without time zone)
8                 ->  Bitmap Index Scan on
    my_logs2010m11_logdate  (cost=0.00..6.73 rows=330 width=0)
9                       Index Cond: (logdate > '2010-12-01␣
    00:00:00'::timestamp without time zone)
0           ->  Bitmap Heap Scan on my_logs2010m12 my_logs
    (cost=6.81..20.93 rows=330 width=52)
1                 Recheck Cond: (logdate > '2010-12-01␣
    00:00:00'::timestamp without time zone)
2                 ->  Bitmap Index Scan on
    my_logs2010m12_logdate  (cost=0.00..6.73 rows=330 width=0)
3                       Index Cond: (logdate > '2010-12-01␣
    00:00:00'::timestamp without time zone)
4           ->  Bitmap Heap Scan on my_logs2011m01 my_logs
    (cost=6.81..20.93 rows=330 width=52)
5                 Recheck Cond: (logdate > '2010-12-01␣
    00:00:00'::timestamp without time zone)
6                 ->  Bitmap Index Scan on
    my_logs2011m01_logdate  (cost=0.00..6.73 rows=330 width=0)
7                       Index Cond: (logdate > '2010-12-01␣
    00:00:00'::timestamp without time zone)
8 (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 партицый, где подходит такое условие. А теперь включим «constraint_exclusion»:

Listing 3.15: «constraint_exclusion» ON

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

```
1                       Index Cond: (logdate > '2010-12-01␣
    00:00:00'::timestamp without time zone)
2           ->  Bitmap Heap Scan on my_logs2010m12 my_logs
    (cost=6.81..20.93 rows=330 width=52)
3                 Recheck Cond: (logdate > '2010-12-01␣
    00:00:00'::timestamp without time zone)
4                 ->  Bitmap Index Scan on
    my_logs2010m12_logdate  (cost=0.00..6.73 rows=330 width=0)
5                       Index Cond: (logdate > '2010-12-01␣
    00:00:00'::timestamp without time zone)
6   (10 rows)
```

Как мы видим, теперь запрос работает правильно, и сканирует только партиции, что подходят под условие запроса. Но включать «constraint_exclus 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.

Глава 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. Преимущество 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 рассматриваться в этой книге (ну, сами понимаете). Вот список свободных решений:

- **Slony-I**[1] — асинхронная Master-Slave репликация, поддерживает каскады(cascading) и отказоустойчивость(failover). Slony-I использует триггеры PostgreSQL для привязки к событиям INSERT/ DELETE/U и хранимые процедуры для выполнения действий.
- **PGCluster**[2] — синхронная Multi-Master репликация. Проект на мой взгляд мертв, поскольку уже год не обновлялся.
- **pgpool-I/II**[3] — 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.

[1] http://www.slony.info/
[2] http://pgfoundry.org/projects/pgcluster/
[3] http://pgpool.projects.postgresql.org/

- **Bucardo**[4] — асинхронная репликация, которая поддерживает Multi-Master и Master-Slave режимы, а также несколько видов синхронизации и обработки конфликтов.
- **Londiste**[5] — асинхронная Master-Slave репликация. Входит в состав Skytools[6]. Проще в использовании, чем Slony-I.
- **Mammoth Replicator**[7] — асинхронная Multi-Master репликация.
- **Postgres-R**[8] — асинхронная Multi-Master репликация.
- **RubyRep**[9] — 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 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/U 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 slonikом, 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.

[4] http://bucardo.org/
[5] http://skytools.projects.postgresql.org/doc/londiste.ref.html
[6] http://pgfoundry.org/projects/skytools/
[7] http://www.commandprompt.com/products/mammothreplicator/
[8] http://www.postgres-r.org/
[9] http://www.rubyrep.org/

Установка

Установка на Ubuntu производится простой командой:

Listing 4.1: Установка

```
1 sudo aptitude install slony1-bin
```

Настройка

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

Наши данные

- БД: customers
- master_host: customers_master.com
- slave_host_1: customers_slave.com
- cluster name (нужно придумать): customers_rep

Подготовка master-сервера

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

Listing 4.2: Подготовка master-сервера

```
1 pgsql@customers_master$ createuser -a -d slony
2 pgsql@customers_master$ psql -d template1 -c "alter␣\
3 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 одного slave-сервера

Здесь я рассматриваю, что серверы кластера соединены посредством сети Internet (как в моём случае), необходимо чтобы с каждого из ведомых серверов можно было установить соединение с PostgreSQL на мастер-хосте, и наоборот. То есть, команда:

Listing 4.3: Подготовка одного slave-сервера

```
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):

Listing 4.4: 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 завести его тоже!

Listing 4.5: 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_pa
Итак:

Listing 4.6: Инициализация БД и plpgsql на slave

```
1 slony@customers_master$ createdb -O customers_owner \
2 -h customers_slave.com customers
3 slony@customers_master$ createlang -d customers \
4 -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-a.

Продолжаем. Создаём таблицы и всё остальное на slave:

Listing 4.7: Инициализация БД и plpgsql на slave

```
1 slony@customers_master$ pg_dump -s customers | \
2 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_

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:

Listing 4.8: Инициализация БД и plpgsql на slave

```
1 slonik <<EOF
2 cluster name = customers_slave;
3 node Y admin conninfo = 'dbname=customers
     host=customers_master.com
4 port=5432 user=slony password=slony_user_pass';
5 uninstall node (id = Y);
6 echo 'okay';
7 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:

Listing 4.9: Инициализация кластера

```
#!/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';
init cluster ( id = 1, comment = 'Customers DB
replication cluster' );

echo 'Create set';

create set ( id = 1, origin = 1, comment = 'Customers
DB replication set' );
```

```
0  echo 'Adding tables to the subscription set';

2  echo ' Adding table public.customers_sales...';
3  set add table ( set id = 1, origin = 1, id = 4, full qualified
4  name = 'public.customers_sales', comment = 'Table
       public.customers_sales' );
5  echo ' done';

7  echo ' Adding table public.customers_something...';
8  set add table ( set id = 1, origin = 1, id = 5, full qualified
9  name = 'public.customers_something,
0  comment = 'Table public.customers_something );
1  echo ' done';

3  echo 'done adding';
4  store node ( id = 2, comment = 'Node 2, $HOST2' );
5  echo 'stored node';
6  store path ( server = 1, client = 2, conninfo =
       'dbname=$DBNAME1 host=$HOST1
7  port=$PORT1 user=slony password=slony_user_password' );
8  echo 'stored path';
9  store path ( server = 2, client = 1, conninfo =
       'dbname=$DBNAME2 host=$HOST2
0  port=$PORT2 user=slony password=slony_user_password' );

2  store listen ( origin = 1, provider = 1, receiver = 2 );
3  store listen ( origin = 2, provider = 2, receiver = 1 );
4  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:

Listing 4.10: Подписываем slave-узел на replication set

```
1  #!/bin/sh
```

```
3  CLUSTER=customers_rep

5  DBNAME1=customers
6  DBNAME2=customers

8  HOST1=customers_master.com
9  HOST2=customers_slave.com

1  PORT1=5432
2  PORT2=5432

4  SLONY_USER=slony

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

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

6  EOF
```

Старт репликации

Теперь, на обоих узлах необходимо запустить демона репликации.

Listing 4.11: Старт репликации

```
1  slony@customers_master$ slon customers_rep \
2  "dbname=customers␣user=slony"
```

и

Listing 4.12: Старт репликации

```
1  slony@customers_slave$ slon customers_rep \
2  "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 4.2.1 и выполнить 4.2.2.

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 4.2.2) делаем следующее:

Listing 4.13: Общие задачи

```
slonik <<EOF
cluster name = customers_slave;
node 3 admin conninfo = 'dbname=customers
    host=customers_slave3.com
port=5432 user=slony password=slony_user_pass';
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:

Listing 4.14: 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:

Listing 4.15: 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.

Listing 4.16: 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:

Listing 4.17: 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
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,
node4.example.com'
CONFIG storePath: pa_server=1 pa_client=3
pa_conninfo="dbname=customers
```

```
3  host=mainhost.com␣port=5432␣user=slony_user
4  password=slony_user_pass" pa_connretry=10
5  CONFIG storeListen: li_origin=1 li_receiver=3
6  li_provider=1
7  CONFIG storeSet: set_id=1 set_origin=1
8  set_comment='CustomersDB replication set'
9  WARN remoteWorker_wakeup: node 1 - no worker thread
0  CONFIG storeSubscribe: sub_set=1 sub_provider=1
      sub_forward='f'
1  WARN remoteWorker_wakeup: node 1 - no worker thread
2  CONFIG enableSubscription: sub_set=1
3  WARN remoteWorker_wakeup: node 1 - no worker thread
4  CONFIG main: configuration complete - starting threads
5  CONFIG enableNode: no_id=1
6  CONFIG enableNode: no_id=2
7  CONFIG enableNode: no_id=4
8  ERROR remoteWorkerThread_1: "begin␣transaction;␣set
9  transaction␣isolation␣level
0  serializable;␣lock␣table␣"_customers_rep".sl_config_lock;␣
      select
1  "_customers_rep".enableSubscription(1,␣1,␣4);
2  notify␣"_customers_rep_Event";␣notify␣
      "_customers_rep_Confirm";
3  insert␣into␣"_customers_rep".sl_event␣(ev_origin,␣ev_seqno,
4  ev_timestamp,␣ev_minxid,␣ev_maxxid,␣ev_xip,
5  ev_type␣,␣ev_data1,␣ev_data2,␣ev_data3,␣ev_data4␣)␣values
6  ('1',␣'219440',
7  '2005-05-05␣18:52:42.708351',␣'52501283',␣'52501292',
8  ''''52501283''',␣'ENABLE_SUBSCRIPTION',
9  '1',␣'1',␣'4',␣'f');␣insert␣into␣"_customers_rep".
0  sl_confirm␣(con_origin,␣con_received,
1  con_seqno,␣con_timestamp)␣values␣(1,␣3,␣'219440',
2  CURRENT_TIMESTAMP);␣commit␣transaction;"
3  PGRES_FATAL_ERROR ERROR: insert or update on table
4  "sl_subscribe" violates foreign key
5  constraint "sl_subscribe-sl_path-ref"
6  DETAIL: Key (sub_provider,sub_receiver)=(1,4)
7  is not present in table "sl_path".
8  INFO remoteListenThread_1: disconnecting from
9  'dbname=customers host=mainhost.com
0  port=5432 user=slony_user password=slony_user_pass'
1  %
```

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)):

Listing 4.18: Устранение неисправностей

```
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:

Listing 4.19: Устранение неисправностей

```
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
(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
(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:

Listing 4.20: Устранение неисправностей

```
1 delete from _customers_rep.sl_event where
2 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.

4.3 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:

- отсутствие поддержки каскадной репликации, отказоустойчивости(failo и переключение между серверами (switchover) (все это обещают к 3 версии реализовать [10])

[10] http://skytools.projects.postgresql.org/skytools-3.0/doc/skytools3.html

Установка

На серверах, которые мы настраиваем расматривается 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:

Listing 4.21: Установка

```
sudo aptitude install skytools
```

Но все же лучше скачать самую последнюю версию пакета с официального сайта — http://pgfoundry.org/projects/skytools. На момент написания статьи последняя версия была 2.1.11. Итак, начнем:

Listing 4.22: Установка

```
$wget http://pgfoundry.org/frs/download.php/2561/
skytools-2.1.11.tar.gz
$tar zxvf skytools-2.1.11.tar.gz
$cd skytools-2.1.11/
# это для сборки deb пакета
$sudo aptitude install build-essential autoconf \
automake autotools-dev dh-make \
debhelper devscripts fakeroot xutils lintian pbuilder \
python-dev yada
# ставим пакет исходников для postgresql 8.4.x
$sudo aptitude install postgresql-server-dev-8.4
# python-psycopg нужен для работы Londiste
$sudo aptitude install python-psycopg2
# данной командой я собираю deb пакет для
# postgresql 8.4.x для( 8.3.x например будет "make deb83")
$sudo make deb84
$cd ../
# ставим skytools
$dpkg -i skytools-modules-8.4_2.1.11_i386.deb
skytools_2.1.11_i386.deb
```

Для других систем можно собрать Skytools командами

Listing 4.23: Установка

```
./configure
```

```
2 make
3 make install
```

Дальше проверим, что все у нас правильно установилось

Listing 4.24: Установка

```
1 $londiste.py -V
2 Skytools version 2.1.11
3 $pgqadm.py -V
4 Skytools version 2.1.11
```

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

Настройка

Обозначения:

- host1 — мастер;
- host2 — слейв;

Настройка ticker-а

Londiste требуется 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 мастер база данных. Для этого мы настраиваем специальный конфиг для ticker-а (пусть конфиг будет у нас /etc/skytools/db1-ticker.ini):

Listing 4.25: Настройка ticker-а

```
1 [pgqadm]
2 # название
3 job_name = db1-ticker

5 # мастер база данных
6 db = dbname=P host=host1

8 # Задержка между запусками обслуживания
9 # ротация( очередей и тп..) в секундах
0 maint_delay = 600

2 # Задержка между проверками наличия активности
3 # новых( пакетов данных) в секундах
```

```
4  loop_delay = 0.1

6  # log и pid демона
7  logfile = /var/log/%(job_name)s.log
8  pidfile = /var/pid/%(job_name)s.pid
```

Теперь необходимо инсталлировать служебный код (SQL) и запустить ticker как демона для базы данных. Делается это с помощью утилиты pgqadm.py следующими командами:

Listing 4.26: Настройка ticker-а

```
1  pgqadm.py /etc/skytools/db1-ticker.ini install
2  pgqadm.py /etc/skytools/db1-ticker.ini ticker -d
```

Проверим, что в логах (/var/log/skytools/db1-tickers.log) всё нормально. На данном этапе там должны быть редкие записи (раз в минуту).

Если нам потребуется остановить ticker, мы можем воспользоватся этой командой:

Listing 4.27: Настройка ticker-а

```
1  pgqadm.py /etc/skytools/db1-ticker.ini ticker -s
```

или если потребуется «убить» ticker:

Listing 4.28: Настройка ticker-а

```
1  pgqadm.py /etc/skytools/db1-ticker.ini ticker -k
```

Востанавливаем схему базы

Londiste не умеет переносить изменения структуры базы данных. Поэтому на всех slave базах данных перед репликацией должна быть создана такая же структура БД, что и на мастере.

Создаём конфигурацию репликатора

Для каждой из реплицируемых баз создадим конфигурационные файлы (пусть конфиг будет у нас /etc/skytools/db1-londiste.ini):

Listing 4.29: Создаём конфигурацию репликатора

```
1  [londiste]
2  # название
```

```
3  job_name = db1-londiste

5  # мастер база данных
6  provider_db = dbname=db1 port=5432 host=host1
7  # слейв база данных
8  subscriber_db = dbname=db1 host=host2

0  # Это будет использоваться в качестве
1  # SQLидентификатора-, тч.. не используйте
2  # точки и пробелы.
3  # ВАЖНО! Если есть живая репликация на другой слейв,
4  # именуем очередь также-
5  pgq_queue_name = db1-londiste-queue

7  # log и pid демона
8  logfile = /var/log/%(job_name)s.log
9  pidfile = /var/run/%(job_name)s.pid

1  # рзмер лога
2  log_size = 5242880
3  log_count = 3
```

Устанавливаем Londiste в базы на мастере и слейве

Теперь необходимо установить служебный SQL для каждой из созданных в предыдущем пункте конфигураций.

Устанавливаем код на стороне мастера:

Listing 4.30: Londiste

```
1  londiste.py /etc/skytools/db1-londiste.ini provider install
```

и подобным образом на стороне слейва:

Listing 4.31: Londiste

```
1  londiste.py /etc/skytools/db1-londiste.ini subscriber install
```

После этого пункта на мастере будут созданы очереди для репликации.

Запускаем процессы Londiste

Для каждой реплицируемой базы делаем:

Listing 4.32: Запускаем

```
1 londiste.py /etc/skytools/db1-londiste.ini replay -d
```

Таким образом запустятся слушатели очередей репликации, но, т.к. мы ещё не указывали какие таблицы хотим реплицировать, они пока будут работать в холостую.

Убедимся что в логах нет ошибок (/var/log/db1-londistes.log).

Добавляем реплицируемые таблицы

Для каждой конфигурации указываем что будем реплицировать с мастера:

Listing 4.33: Добавляем реплицируемые таблицы

```
1 londiste.py /etc/skytools/db1-londiste.ini provider add --all
```

и что со слейва:

Listing 4.34: Добавляем реплицируемые таблицы

```
1 londiste.py /etc/skytools/db1-londiste.ini subscriber add
    --all
```

В данном примере я использую спец-параметр «–all», который означает все таблицы, но вместо него вы можете перечислить список конкретных таблиц, если не хотите реплицировать все.

Добавляем реплицируемые последовательности (sequence)

Так же для всех конфигураций. Для мастера:

Listing 4.35: Добавляем последовательности

```
1 londiste.py /etc/skytools/db1-londiste.ini provider add-seq
    --all
```

Для слейва:

Listing 4.36: Добавляем реплицируемые таблицы

```
1 londiste.py /etc/skytools/db1-londiste.ini subscriber add-seq
    --all
```

Точно также как и с таблицами можно указать конкретные последовательности вместо «–all».

Проверка

Итак, всё что надо сделано. Теперь Londiste запустит так называемый bulk copy процесс, который массово (с помощью COPY) зальёт присутствующие на момент добавления таблиц данные на слейв, а затем перейдёт в состояние обычной репликации.

Мониторим логи на предмет ошибок:

Listing 4.37: Проверка

```
less /var/log/db1-londiste.log
```

Если всё хорошо, смотрим состояние репликации. Данные уже синхронизированы для тех таблиц, где статус отображается как "ok".

Listing 4.38: Проверка

```
londiste.py /etc/skytools/db1-londiste.ini subscriber tables

    Table State
    public.table1 ok
    public.table2 ok
    public.table3 in-copy
    public.table4 -
    public.table5 -
    public.table6 -
    ...
```

Для удобства представляю следующий трюк с уведомление в почту об окончании первоначального копирования (мыло поменять на своё):

Listing 4.39: Проверка

```
(
while [ $(
python londiste.py /etc/skytools/db1-londiste.ini subscriber
    tables |
tail -n+2 | awk '{print $2}' | grep -v ok | wc -l) -ne 0 ];
do sleep 60; done; echo '' | mail -s 'Replication done EOM'
    user@domain.com
) &
```

Общие задачи

Добавление всех таблиц мастера слейву

Просто используя эту команду:

Listing 4.40: Добавление всех таблиц мастера слейву

```
1 londiste.py <ini> provider tables | xargs londiste.py <ini>
    subscriber add
```

Проверка состояния слейвов

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

Listing 4.41: Проверка состояния слейвов

```
1 SELECT queue_name, consumer_name, lag, last_seen
2   FROM pgq.get_consumer_info();
```

«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 потребоватся удалить все ваши настройки для того, чтобы начать все заново. Для PGQ, чтобы остановить накопление данных, используйте следующие API:

Listing 4.42: Удаление очереди всех событий из мастера

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

Или воспользуйтесь pgqadm.py:

Listing 4.43: Удаление очереди всех событий из мастера

```
1 pgqadm.py <ticker.ini> unregister 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. SELECT londiste.provider_refresh_trigger('queue_name', 'tablename');
5. COMMIT;

Удаление столбца из таблицу

1. BEGIN; – на мастере
2. удалить поле на мастере
3. SELECT londiste.provider_refresh_trigger('queue_name', 'tablename');
4. COMMIT;
5. Проверить «lag», когда londiste пройдет момент удаления поля
6. 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 собитие 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.

Listing 4.44: Устранение неисправностей

```
1 SELECT count(*)
2   FROM pgq.event_1,
3     (SELECT tick_snapshot
4       FROM pgq.tick
5       WHERE tick_id BETWEEN 5715138 AND 5715139
6     ) as t(snapshots)
7 WHERE txid_visible_in_snapshot(ev_txid, snapshots);
```

В нашем случае, это было более чем 5 миллионов и 400 тысяч событий. Многовато. 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 ticker-а следующую настройку:

Listing 4.45: Устранение неисправностей

```
1 pgq_lazy_fetch = 500
```

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

4.4 Streaming Replication (Потоковая репликация)

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 в качестве стандартной репликации в большинстве высоконагруженых 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
- асинхронный механизм — слейв отстает от мастера (но в отличие от других методов репликации, это отставание очень короткое, и может составлять всего лишь одну транзакцию, в зависимости от скорости сети, нагруженности БД и настроек «Hot Standby»)

Установка

Для начала нам потребуется PostgreSQL не ниже 9 версии. В момент написания этой главы была доступна 9.0.1 версия. Все работы, как полагается, будут проводится на ОС 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:

Listing 4.46: Создаем пользователя userssh

```
1 $sudo groupadd userssh
2 $sudo useradd -m -g userssh -d /home/userssh -s /bin/bash \
3 -c "user␣ssh␣allow" userssh
```

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

Listing 4.47: Логинимся под пользователем postgres

```
1 su postgres
```

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

Listing 4.48: Генерим RSA-ключ

```
1 postgres@localhost ~ $ ssh-keygen -t rsa -P ""
2 Generating public/private rsa key pair.
3 Enter file in which to save the key
    (/var/lib/postgresql/.ssh/id_rsa):
4 Created directory '/var/lib/postgresql/.ssh'.
5 Your identification has been saved in
    /var/lib/postgresql/.ssh/id_rsa.
6 Your public key has been saved in
    /var/lib/postgresql/.ssh/id_rsa.pub.
7 The key fingerprint is:
```

```
8 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:

Listing 4.49: Добавляем его в список авторизованных ключей

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

Этого должно быть более чем достаточно, проверить работоспособность соединения можно просто написав:

Listing 4.50: Пробуем зайти на ssh без пароля

```
1 ssh localhost
```

Не забываем предварительно инициализировать sshd:

Listing 4.51: Запуск sshd

```
1 /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):

Listing 4.52: Мастер pg_hba.conf

```
1 host  all  all  192.168.0.20/32  trust
```

Listing 4.53: Слейв pg_hba.conf

```
1 host  all  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:

Listing 4.54: Настройка мастера

```
# 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

# 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
# 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
# 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
# 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 востановления 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 указаное переменной «archive_command» хранилище. В данном случае в директорию «/path_to/archive/».

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:

Listing 4.55: Выполняем на мастере

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

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

Listing 4.56: Выполняем на мастере

```
1 rsync -C -a --delete -e ssh --exclude postgresql.conf
    --exclude postmaster.pid \
2 --exclude postmaster.opts --exclude pg_log --exclude pg_xlog \
3 --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:

Listing 4.57: Выполняем на мастере

```
1 psql -c "SELECT␣pg_stop_backup()"
```

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:

Listing 4.58: Конфиг слейва

```
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:

Listing 4.59: Конфиг recovery.conf

```
1  # Specifies whether to start the server as a standby. In
       streaming replication,
2  # this parameter must to be set to on.
3  standby_mode          = 'on'
4
5  # Specifies a connection string which is used for the standby
       server to connect
6  # with the primary.
7  primary_conninfo      = 'host=192.168.0.10 port=5432
       user=postgres'
8
9  # Specifies a trigger file whose presence should cause
       streaming replication to
10 # end (i.e., failover).
11 trigger_file = '/path_to/trigger'
12
13 # Specifies a command to load archive segments from the WAL
       archive. If
14 # 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
16 # a large workload can cause segments to be recycled before
       the standby
17 # is fully synchronized, requiring you to start again from a
       new base backup.
18 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 будет востанавливатся 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:

Listing 4.60: Тестирование репликации

```
$ psql -c "SELECT␣pg_current_xlog_location()" -h192.168.0.10
    (masterdb)
 pg_current_xlog_location
--------------------------
 0/2000000
(1 row)

$ psql -c "select␣pg_last_xlog_receive_location()"
    -h192.168.0.20 (slavedb)
 pg_last_xlog_receive_location
-------------------------------
 0/2000000
(1 row)

$ psql -c "select␣pg_last_xlog_replay_location()"
    -h192.168.0.20 (slavedb)
 pg_last_xlog_replay_location
------------------------------
 0/2000000
(1 row)
```

Еще проверить работу репликации можно с помощью утилиты ps:

Listing 4.61: Тестирование репликации

```
[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
postgres  6878  6872  1 10:31 ?        00:00:01 postgres: wal
    receiver process   streaming 0/2000000
```

Теперь проверим реприкацию. Выполним на мастере:

Listing 4.62: 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:

Listing 4.63: 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.

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

4.5 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 будем проводить на Ubuntu Server. Сначала нам нужно установить DBIx::Safe Perl модуль.

Listing 4.64: Установка

```
1 sudo aptitude install libdbix-safe-perl
```

Для других систем можно поставить из исходников[11]:

Listing 4.65: Установка

```
1 tar xvfz DBIx-Safe-1.2.5.tar.gz
2 cd DBIx-Safe-1.2.5
3 perl Makefile.PL
4 make && make test && sudo make install
```

Теперь ставим сам Bucardo. Скачиваем[12] его и инсталируем:

Listing 4.66: Установка

```
1 tar xvfz Bucardo-4.4.0.tar.gz
```

[11] http://search.cpan.org/CPAN/authors/id/T/TU/TURNSTEP/

[12] http://bucardo.org/wiki/Bucardo#Obtaining_Bucardo

```
2 cd Bucardo-4.4.0
3 perl Makefile.PL
4 make
5 sudo make install
```

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

Listing 4.67: Установка

```
1 sudo aptitude install postgresql-plperl-8.4
```

Можем приступать к настройке.

Настройка

Инициализация Bucardo

Запускаем установку командой:

Listing 4.68: Инициализация Bucardo

```
1 bucardo_ctl install
```

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

Listing 4.69: Инициализация Bucardo

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

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

8 Current connection settings:
9 1. Host:          <none>
0 2. Port:          5432
1 3. User:          postgres
2 4. Database:      postgres
3 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 логинится через Unix socket, поэтому лучше заранее дать ему такие права в pg_hda.conf.

Настройка баз данных

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

Listing 4.70: Настройка баз данных

```
1 bucardo_ctl add db master_db name=master
2 bucardo_ctl add all tables herd=all_tables
3 bucardo_ctl add all sequences herd=all_tables
```

Первой командой мы указали базу данных и дали ей имя master (для того, что в реальной жизни master_db и slave_db имеют одинаковое название и их нужно Bucardo отличать). Второй и третей командой мы указали реплицыровать все таблицы и последовательности, обьеденив их в групу all_tables.

Дальше добавляем slave_db:

Listing 4.71: Настройка баз данных

```
1 bucardo_ctl add db slave_db name=replica port=6543
    host=slave_host
```

Мы назвали replica базу данных в Bucardo.

Настройка синхронизации

Теперь нам нужно настроить синхронизацию между этими базами данных. Делается это командой (master-slave):

Listing 4.72: Настройка синхронизации

```
1 bucardo_ctl add sync delta type=pushdelta source=all_tables
    targetdb=replica
```

Данной командой мы установим Bucardo тригеры в PostgreSQL. А теперь по параметрам:

- **type**

  Это тип синхронизации. Существует 3 типа:

- **Fullcopy**. Полное копирование.
- **Pushdelta**. Master-slave репликация.
- **Swap**. Master-master репликация. Для работы в таком режиме потребуется указать как Bucardo должен решать конфликты синхронизации. Для этого в таблице «goat» (в которой находятся таблицы и последовательности) нужно в «standard_conflict» поле поставить значение (это значение может быть разным для разных таблиц и последовательностей):
  - source — при конфликте мы копируем данные с source (master_ в нашем случае).
  - target — при конфликте мы копируем данные с target (slave_db в нашем случае).
  - skip — конфликт мы просто не реплицируем. Не рекомендуется.
  - random — каждая БД имеет одинаковый шанс, что её изменение будет взято для решение конфликта.
  - latest — запись, которая была последней изменена решает конфликт.
  - abort — синхронизация прерывается.

- **source**

  Источник синхронизации.
- **targetdb**

  БД, в которум производим репликацию.

Для master-master:

Listing 4.73: Настройка синхронизации

```
1 bucardo_ctl add sync delta type=swap source=all_tables
    targetdb=replica
```

Запуск/Остановка репликации

Запуск репликации:

Listing 4.74: Запуск репликации

```
1 bucardo_ctl start
```

Остановка репликации:

Listing 4.75: Остановка репликации

```
1 bucardo_ctl stop
```

Общие задачи

Просмотр значений конфигурации

Просто используя эту команду:

Listing 4.76: Просмотр значений конфигурации

```
1 bucardo_ctl show all
```

Изменения значений конфигурации

Listing 4.77: Изменения значений конфигурациии

```
1 bucardo_ctl set name=value
```

Например:

Listing 4.78: Изменения значений конфигурации

```
1 bucardo_ctl set syslog_facility=LOG_LOCAL3
```

Перегрузка конфигурации

Listing 4.79: Перегрузка конфигурации

```
1 bucardo_ctl reload_config
```

Более полный список команд — http://bucardo.org/wiki/Bucardo_ctl

4.6 RubyRep

Введение

RubyRep представляет собой движок для организации асинхронной репликации, написанный на языке ruby. Основные принципы: простота использования и не зависить 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 обьемах данных «ест» процессор и память

Установка

RubyRep поддерживает два типа установки: через стандартный Ruby или JRuby. Рекомендую ставить JRuby вариант — производительность будет выше.

Установка JRuby версии

Предварительно должна быть установлена Java (версия 1.6).

1. Загрузите последнюю версию JRuby rubyrep с Rubyforge[13].
2. Распакуйте
3. Готово

Установка стандартной Ruby версии

1. Установить Ruby, Rubygems.
2. Установить драйвера базы данных.

   Для Mysql:

   Listing 4.80: Установка

```
sudo gem install mysql
```

   Для PostgreSQL:

   Listing 4.81: Установка

```
sudo gem install postgres
```

3. Устанавливаем rubyrep:

   Listing 4.82: Установка

```
sudo gem install rubyrep
```

[13]http://rubyforge.org/frs/?group_id=7932, выберите ZIP

Настройка

Создание файла конфигурации

Выполним команду:

Listing 4.83: Настройка

```
rubyrep generate myrubyrep.conf
```

Команда generate создала пример конфигурации в файл myrubyrep.conf:

Listing 4.84: Настройка

```
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:

Listing 4.85: Сканирование баз данных

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

Пример вывода:

Listing 4.86: Сканирование баз данных

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

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

Синхронизация баз данных

Выполним команду:

Listing 4.87: Синхронизация баз данных

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

Также можно указать только какие таблицы в базах данных синхронизировать:

Listing 4.88: Синхронизация баз данных

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

Настройки политики синхронизации позволяют указывать как решать конфликты синхронизации. Более подробно можно почитать в документации http://www.rubyrep.org/configuration.html.

Репликация

Для запуска репликации достаточно выполнить:

Listing 4.89: Репликация

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

Данная команда установить 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 востановлены.

Для удаления репликации достаточно выполнить:

Listing 4.90: Репликация

```
rubyrep uninstall -c myrubyrep.conf
```

Устранение неисправностей

Ошибка при запуске репликации

При зепуске rubyrep через Ruby может возникнуть подобная ошибка:

Listing 4.91: Устранение неисправностей

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

Это проблема с запусками потоков в Ruby. Решается двумя способами:

1. Запускать rubyrep через JRuby (тут с потоками не будет проблем)
2. Пофиксить rubyrep патчем:

Listing 4.92: Устранение неисправностей

```
---
    /Library/Ruby/Gems/1.8/gems/rubyrep-1.1.2/lib/rubyrep/
replication_runner.rb 2010-07-16 15:17:16.000000000 -0400
+++ ./replication_runner.rb 2010-07-16
    17:38:03.000000000 -0400
@@ -2,6 +2,12 @@

 require 'optparse'
 require 'thread'
+require 'monitor'
+
+class Monitor
+ alias lock mon_enter
+ alias unlock mon_exit
+end

 module RR
```

```
16      # This class implements the functionality of the
        'replicate' command.
17  @@ -94,7 +100,7 @@
18        # Initializes the waiter thread used for
        replication pauses
19        # and processing
20        # the process TERM signal.
21        def init_waiter
22  -   @termination_mutex = Mutex.new
23  +   @termination_mutex = Monitor.new
24          @termination_mutex.lock
25          @waiter_thread ||= Thread.new
        {@termination_mutex.lock;
26            self.termination_requested = true}
27          %w(TERM INT).each do |signal|
```

4.7 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 — одна 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 не обладает подобным функционалом, но компактный и прост в установке. Bucardo — система которая может быть или master-master, или master-slave репликацией, но не может обработать огромные обьекты, нет отказоустойчивости(failover) и переключение между серверами (switchover). RubyRep, как для master-master репликации, очень просто в установке и настройке, но за это ему приходится расплачиватся скоростью работы — самый медленный из всех (синхронизация больших обьемов данных между таблицами).

Глава 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 просто. 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 соответсвия, 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:

- **Greenplum Database**[1]
- **GridSQL for EnterpriseDB Advanced Server**[2]
- **Sequoia**[3]
- **PL/Proxy**[4]
- **HadoopDB**[5] (Shared-nothing clustering)

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

[1] http://www.greenplum.com/index.php?page=greenplum-database
[2] http://www.enterprisedb.com/products/gridsql.do
[3] http://www.continuent.com/community/lab-projects/sequoia
[4] http://plproxy.projects.postgresql.org/doc/tutorial.html
[5] http://db.cs.yale.edu/hadoopdb/hadoopdb.html

- 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[6] 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 8.4:

Listing 5.1: Установка

```
1 sudo aptitude install postgresql-8.4-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 написаные нижу нужно выполнять на каждом ноде.

Создадим базу данных plproxytest(если её ещё нет):

Listing 5.2: Настройка

```
1 CREATE DATABASE plproxytest
2      WITH OWNER = postgres
3        ENCODING = 'UTF8';
```

Добавляем табличку users:

[6] http://pgfoundry.org/projects/plproxy

Listing 5.3: Настройка

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

Теперь создадим функцию для добавления данных в таблицу users:

Listing 5.4: Настройка

```
CREATE OR REPLACE FUNCTION public.insert_user(i_username text,
i_emailaddress   text)
RETURNS integer AS
$BODY$
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:

Listing 5.5: Настройка

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

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

Listing 5.6: Настройка

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

```
CREATE LANGUAGE plproxy HANDLER plproxy_call_handler;
CREATE LANGUAGE plpgsql;
```

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

Listing 5.7: Настройка

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

Listing 5.8: Настройка

```
CREATE OR REPLACE FUNCTION
public.get_cluster_partitions(cluster_name text)
  RETURNS SETOF text AS
$BODY$
BEGIN
  IF cluster_name = 'usercluster' THEN
    RETURN NEXT 'dbname=plproxytest␣host=node1␣user=postgres';
    RETURN NEXT 'dbname=plproxytest␣host=node2␣user=postgres';
    RETURN;
  END IF;
  RAISE EXCEPTION 'Unknown␣cluster';
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE
```

```
5    COST 100
6    ROWS 1000;
7  ALTER FUNCTION public.get_cluster_partitions(text)
8  OWNER TO postgres;
```

И последняя:

Listing 5.9: Настройка

```
1  CREATE OR REPLACE FUNCTION
2  public.get_cluster_version(cluster_name text)
3    RETURNS integer AS
4  $BODY$
5  BEGIN
6    IF cluster_name = 'usercluster' THEN
7      RETURN 1;
8    END IF;
9    RAISE EXCEPTION 'Unknown␣cluster';
10 END;
11 $BODY$
12   LANGUAGE 'plpgsql' VOLATILE
13   COST 100;
14 ALTER FUNCTION public.get_cluster_version(text)
15 OWNER TO postgres;
```

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

Listing 5.10: Настройка

```
1  CREATE OR REPLACE FUNCTION
2  public.insert_user(i_username text, i_emailaddress text)
3    RETURNS integer AS
4  $BODY$
5    CLUSTER 'usercluster';
6    RUN ON hashtext(i_username);
7  $BODY$
8    LANGUAGE 'plproxy' VOLATILE
9    COST 100;
10 ALTER FUNCTION public.insert_user(text, text)
11 OWNER TO postgres;
```

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

Listing 5.11: Настройка

```
1  SELECT insert_user('Sven','sven@somewhere.com');
2  SELECT insert_user('Marko', 'marko@somewhere.com');
```

```
3  SELECT insert_user('Steve','steve@somewhere.com');
```

Пробуем извлечь данные. Для этого напишем новую серверную функцию:

Listing 5.12: Настройка

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

И попробуем её вызвать:

Listing 5.13: Настройка

```
1  select plproxy.get_user_email('Steve');
```

Если потом подключится 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 прийдётся очень тщательно, подготовив все сервера заранее, занеся данные и потом в один момент подменив код функции get_cluster_partitions.

5.3 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

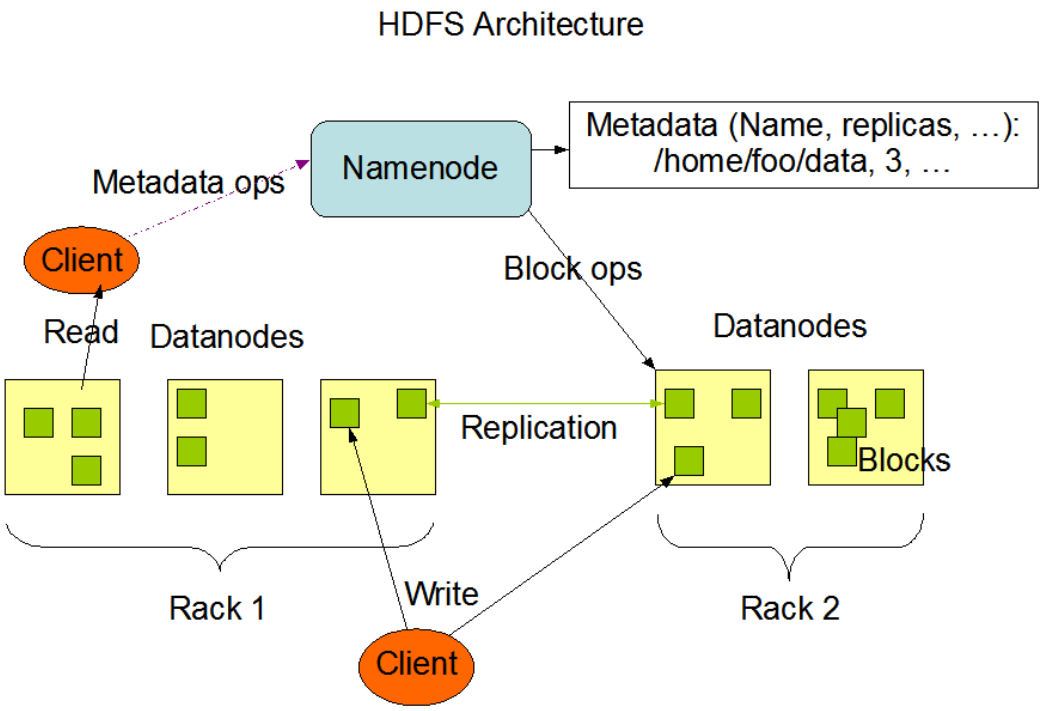

Рис. 5.1: Архитектура HDFS

- **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.

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 какой-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 группируются 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 беспрекословно 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 итерирующмй 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, 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 на Ubuntu Server операционной системе.

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]:

  Listing 5.14: Создаем пользователя с правами

```
1 $sudo groupadd hadoop
2 $sudo useradd -m -g hadoop -d /home/hadoop -s /bin/bash \
3 -c "Hadoop␣software␣owner" hadoop
```

  Далее действия выполняем от его имени:

  Listing 5.15: Логинимся под пользователем hadoop

```
1 su hadoop
```

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

  Listing 5.16: Генерим RSA-ключ

```
1 hadoop@localhost ~ $ ssh-keygen -t rsa -P ""
2 Generating public/private rsa key pair.
3 Enter file in which to save the key
     (/home/hadoop/.ssh/id_rsa):
4 Your identification has been saved in
     /home/hadoop/.ssh/id_rsa.
5 Your public key has been saved in
     /home/hadoop/.ssh/id_rsa.pub.
6 The key fingerprint is:
7 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:

Listing 5.17: Добавляем его в список авторизованных ключей

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

Этого должно быть более чем достаточно, проверить работоспособность соединения можно просто написав:

Listing 5.18: Пробуем зайти на ssh без пароля

```
1 ssh localhost
```

Не забываем предварительно инициализировать sshd:

Listing 5.19: Запуск sshd

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

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

Listing 5.20: Устанавливаем JVM

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

Дальше скачиваем и устанавливаем Hadoop:

Listing 5.21: Устанавливаем Hadoop

```
1 cd /opt
2 sudo wget http://www.gtlib.gatech.edu/pub/apache/hadoop
3 /core/hadoop-0.20.2/hadoop-0.20.2.tar.gz
4 sudo tar zxvf hadoop-0.20.2.tar.gz
5 sudo ln -s /opt/hadoop-0.20.2 /opt/hadoop
6 sudo chown -R hadoop:hadoop /opt/hadoop /opt/hadoop-0.20.2
7 sudo mkdir -p /opt/hadoop-data/tmp-base
8 sudo chown -R hadoop:hadoop /opt/hadoop-data/
```

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

Listing 5.22: 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:

Listing 5.23: 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:

Listing 5.24: Настройки mapreduce

```
<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:

Listing 5.25: Настройки hdfs

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

Теперь необходимо отформатировать Namenode:

Listing 5.26: 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:

Listing 5.27: 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.o
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[7] и распакуем hadoopdb.jar в $HADOOP_HOM

Listing 5.28: Установка HadoopDB

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

Также нам потребуется PostgreSQL JDBC библиотека. Скачайте её[8] и распакуйте в директорию $HADOOP_HOME/lib.

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

Listing 5.29: Установка HadoopDB

```
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:

[7] http://sourceforge.net/projects/hadoopdb/files/
[8] http://jdbc.postgresql.org/download.html

Listing 5.30: Установка HadoopDB

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

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

Listing 5.31: Установка HadoopDB

```
$ hive
Hive history file=/tmp/hadoop/
hive_job_log_hadoop_201005081717_1990651345.txt
hive>

hive> quit;
```

Тестирование

Теперь проведем тестирование. Для этого скачаем бенчмарк:

Listing 5.32: Тестирование

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

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

Listing 5.33: 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, что мы сохранили на HDFS, в нее:

Listing 5.34: Тестирование

```
$hadoop fs -get /data/SortGrep535MB/part-00000 my_file
$psql
psql> CREATE DATABASE grep0;
psql> USE grep0;
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. Распакуйт его и настройте:

Listing 5.35: Тестирование

```
#Properties for Catalog Generation
##################################
nodes_file=machines.txt
relations_unchunked=grep, EntireRankings
relations_chunked=Rankings, UserVisits
catalog_file=HadoopDB.xml
##
#DB Connection Parameters
##
port=5432
username=postgres
password=password
driver=com.postgresql.Driver
url_prefix=jdbc\:postgresql\://
##
#Chunking properties
##
chunks_per_node=0
unchunked_db_prefix=grep
chunked_db_prefix=cdb
##
#Replication Properties
##
dump_script_prefix=/root/dump_
replication_script_prefix=/root/load_replica_
dump_file_u_prefix=/mnt/dump_udb
dump_file_c_prefix=/mnt/dump_cdb
##
#Cluster Connection
##
ssh_key=id_rsa
```

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

Listing 5.36: Тестирование

```
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:

Listing 5.37: Репликация

```
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:

Listing 5.38: Обновляем конфиг

```
1 hadoop dfs -rmr HadoopDB.xml
2 hadoop dfs -put HadoopDB.xml HadoopDB.xml
```

и поставить «true» для «hadoopdb.config.replication» в HADOOP_HOME/co
site.xml.

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

Listing 5.39: Тестирование

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

Приблизительный результат:

Listing 5.40: Тестирование

```
1 $java -cp $CLASSPATH:hadoopdb.jar
    edu.yale.cs.hadoopdb.benchmark.GrepTaskDB \
2 > -pattern %wo% -output padraig -hadoop.config.file
    HadoopDB.xml
3 14.08.2010 19:08:48 edu.yale.cs.hadoopdb.exec.DBJobBase
    initConf
4 INFO: SELECT key1, field FROM grep WHERE field LIKE '%%wo%%';
5 14.08.2010 19:08:48 org.apache.hadoop.metrics.jvm.JvmMetrics
    init
6 INFO: Initializing JVM Metrics with processName=JobTracker,
    sessionId=
7 14.08.2010 19:08:48 org.apache.hadoop.mapred.JobClient
    configureCommandLineOptions
8 WARNING: Use GenericOptionsParser for parsing the arguments.
9 Applications should implement Tool for the same.
0 14.08.2010 19:08:48 org.apache.hadoop.mapred.JobClient
    monitorAndPrintJob
1 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
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
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
14.08.2010 19:08:48
    org.apache.hadoop.mapred.LocalJobRunner$Job statusUpdate
INFO:
14.08.2010 19:08:48 org.apache.hadoop.mapred.Task commit
INFO: Task attempt_local_0001_m_000000_0 is allowed to commit
    now
14.08.2010 19:08:48
    org.apache.hadoop.mapred.FileOutputCommitter commitTask
INFO: Saved output of task 'attempt_local_0001_m_000000_0' to
    file:/home/leo/padraig
14.08.2010 19:08:48
    org.apache.hadoop.mapred.LocalJobRunner$Job statusUpdate
INFO:
14.08.2010 19:08:48 org.apache.hadoop.mapred.Task sendDone
INFO: Task 'attempt_local_0001_m_000000_0' done.
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
INFO: Counters: 6
14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
```

```
2  INFO:   FileSystemCounters
3  14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
4  INFO:     FILE_BYTES_READ=141370
5  14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
6  INFO:     FILE_BYTES_WRITTEN=153336
7  14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
8  INFO:   Map-Reduce Framework
9  14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
0  INFO:     Map input records=3
1  14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
2  INFO:     Spilled Records=0
3  14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
4  INFO:     Map input bytes=3
5  14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
6  INFO:     Map output records=3
7  14.08.2010 19:08:49 edu.yale.cs.hadoopdb.exec.DBJobBase run
8  INFO:
9  JOB TIME : 1828 ms.
```

Результат сохранен в HDFS, в папке padraig:

Listing 5.41: Тестирование

```
1  $ cd padraig
2  $ cat part-00000
3  some data
```

Проверим данные в PostgreSQL:

Listing 5.42: Тестирование

```
1  psql> select * from grep where field like '%wo%';
2  +---------------------------------+-------------------+
3  | key1                            | field
4  |
5  +---------------------------------+-------------------+
6  some data

8  1 rows in set (0.00 sec)

0  psql>
```

Значения совадают. Все работает как требуется.

Проведем еще один тест. Добавим данные в PostgreSQL:

Listing 5.43: Тестирование

```
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:

Listing 5.44: Тестирование

```
$ 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%%';
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
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
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
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$Job statusUpdate
INFO:
01.11.2010 23:14:47 org.apache.hadoop.mapred.Task commit
INFO: Task attempt_local_0001_m_000000_0 is allowed to commit
    now
01.11.2010 23:14:47
    org.apache.hadoop.mapred.FileOutputCommitter commitTask
INFO: Saved output of task 'attempt_local_0001_m_000000_0' to
    file:/home/hadoop/padraig
01.11.2010 23:14:47
    org.apache.hadoop.mapred.LocalJobRunner$Job statusUpdate
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%
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
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
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
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
```

```
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:

Listing 5.45: Тестирование

```
$ cd padraig
$ cat part-00000
I am live!  Maybe
I am live!  Maybewqe
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 http://hadoopdb.sourceforge.net/guide/quick_start_guide.html.

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 масштабируемсть, 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.

5.4 Заключение

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

Глава 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 вцелом.
- **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 ис-

пользует 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 тоже 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 http://pgpool.projects.postgres
II/doc/tutorial-en.html.

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.

Listing 6.1: Установка pgpool-II

```
./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.

Listing 6.2: Установка pgpool-II

```
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/pg
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 находится в репозитории пакетов. Для Ubuntu Linux, например, достаточно будет выполнить:

Listing 6.3: Установка pgpool-II

```
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.

Listing 6.4: Файлы конфигурации

```
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 «*».

Listing 6.5: Файлы конфигурации

```
1 listen_addresses = 'localhost'
2 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.

Listing 6.6: Настройка команд PCP

```
1 postgres:e8a48653851e28c69d0506508fb27fc5
```

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

Listing 6.7: Настройка команд PCP

```
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.

Listing 6.8: Настройка команд PCP

```
1 $ /usr/bin/pg_md5 postgres
2 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.

Listing 6.9: Настройка команд PCP

```
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.

Listing 6.10: Подготовка узлов баз данных

```
1 backend_hostname0 = 'localhost'
2 backend_port0 = 5432
3 backend_weight0 = 1
4 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.

Listing 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.

Listing 6.12: Запуск

```
pgpool -n &
```

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

Listing 6.13: Запуск

```
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.

Listing 6.14: Запуск

```
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.

Listing 6.15: Остановка

```
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.

Listing 6.16: Остановка

```
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.

Listing 6.17: Настройка репликации

```
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.

Listing 6.18: Настройка репликации

```
1 load_balance_mode = true
```

В этом разделе мы включили оба параметра replication_mode и load_bala

Проверка репликации

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

Listing 6.19: Проверка репликации

```
1 createdb -p 9999 bench_replication
```

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

Listing 6.20: Проверка репликации

```
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).

Listing 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 бызы 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.

Listing 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.

Listing 6.23: Настройка параллельного запроса

```
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.

Listing 6.24: Настройка параллельного запроса

```
1 replication_mode = true
2 load_balance_mode = false
```

ИЛИ

Listing 6.25: Настройка параллельного запроса

```
1 replication_mode = false
2 load_balance_mode = true
```

В этом разделе мы установим параметры parallel_mode и load_balance_m 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 подключением.

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

Listing 6.26: Настройка SystemDB

```
system_db_hostname = 'localhost'
system_db_port = 5432
system_db_dbname = 'pgpool'
system_db_schema = 'pgpool_catalog'
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».

Listing 6.27: Настройка SystemDB

```
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 PostgreS

Для установки dblink на вашей системе выполните следующие команды.

Listing 6.28: Установка dblink

```
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.

Listing 6.29: Установка dblink

```
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 буть 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».

Listing 6.30: Создание таблицы dist_def

```
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.

Listing 6.31: Создание таблицы dist_def

```
1  CREATE TABLE pgpool_catalog.dist_def (
2      dbname text, -- имя базы данных
3      schema_name text, -- имя схемы
4      table_name text, -- имя таблицы
5      col_name text NOT NULL CHECK (col_name = ANY (col_list)),
6      -- столбец ключ для распределения данных
7      col_list text[] NOT NULL, -- список имен столбцов
8      type_list text[] NOT NULL, -- список типов столбцов
9      dist_def_func text NOT NULL,
10     -- имя функции распределения данных
11     PRIMARY KEY (dbname, schema_name, table_name)
12 );
```

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.

Listing 6.32: Создание таблицы replicate_def

```
1 CREATE TABLE pgpool_catalog.replicate_def (
2     dbname text, -- имя базы данных
3     schema_name text, -- имя схемы
4     table_name text, -- имя таблицы
5     col_list text[] NOT NULL, -- список имен столбцов
6     type_list text[] NOT NULL, -- список типов столбцов
7     PRIMARY KEY (dbname, schema_name, table_name)
8 );
```

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.

Listing 6.33: Установка правил распределения данных

```
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.

Listing 6.34: Установка правил распределения данных

```
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.

Listing 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.

Listing 6.36: Установка правил репликации

```
1  INSERT INTO pgpool_catalog.replicate_def VALUES (
2      'bench_parallel',
3      'public',
4      'branches',
5      ARRAY['bid', 'bbalance', 'filler'],
6      ARRAY['integer', 'integer', 'character(88)']
7  );

9  INSERT INTO pgpool_catalog.replicate_def VALUES (
0      'bench_parallel',
1      'public',
2      'tellers',
3      ARRAY['tid', 'bid', 'tbalance', 'filler'],
4      ARRAY['integer', 'integer', 'integer', 'character(84)']
5  );
```

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.

Listing 6.37: Установка правил репликации

```
1  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 «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.

Listing 6.38: Проверка параллельного запроса

```
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.

Listing 6.39: Проверка параллельного запроса

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

Созданные таблицы и данные в них показаны в разделе «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.

Listing 6.40: Проверка параллельного запроса

```
1 for port in 5432 5433 5434i 9999; do
2 >     echo $port
3 >     psql -c "SELECT␣min(aid),␣max(aid)␣FROM␣accounts" \
4 >     -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 указывает-

ся как для репликации. master_slave_mode и load_balance_mode устанавливается в true. pgpool-II будет посылать запросы INSERT/UPDATE/DE 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:

Listing 6.41: Скрипт выполняется при падении нода PostgreSQL

```
1  #! /bin/sh
2  # Failover command for streming replication.
3  # This script assumes that DB node 0 is primary, and 1 is
       standby.
4  #
5  # If standby goes down, does nothing. If primary goes down,
       create a
6  # trigger file so that standby take over primary node.
7  #
8  # Arguments: $1: failed node id. $2: new master hostname. $3:
       path to
9  # trigger file.

11 failed_node=$1
12 new_master=$2
13 trigger_file=$3

15 # Do nothing if standby goes down.
```

```
6  if [ $failed_node = 1 ]; then
7    exit 0;
8  fi

0  # Create trigger file.
1  /usr/bin/ssh -T $new_master /bin/touch $trigger_file

3  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:

Listing 6.42: Что выполнять при падении нода

```
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 с режима восстановления в обычный и сможет принимать запросы на запись.

6.6 Онлайн востановление

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 клиента может ждать подключения к pgpool, в то время как восстановливается узел базы данных. Онлайн востановление 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-за проблем 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_1
- **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».

Listing 6.43: recovery_user

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

- Установить «recovery_password» для «recovery_user» для подключения к СУБД.

Listing 6.44: recovery_password

```
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:

Listing 6.45: basebackup.sh

```
#! /bin/sh
# Recovery script for streaming replication.
# This script assumes that DB node 0 is primary, and 1 is standby.
#
datadir=$1
desthost=$2
destdir=$3

psql -c "SELECT pg_start_backup('Streaming Replication', true)" postgres

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/

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 логинится с мастера на слейв без пароля.

Далее добавим скрипт на выполнение для первого этапа онлайн востановления:

Listing 6.46: recovery_1st_stage_command

```
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.

Listing 6.47: Устанавливаем C и SQL функции

```
1 $ cd pgpool-II-x.x.x/sql/pgpool-recovery
2 $ make
3 $ make install
4 $ psql -f pgpool-recovery.sql template1
```

Вот и все. Теперь возможно использовать «pcp_recovery_node» для онлайн восстановления упарших слейвов.

6.7 Заключение

PgPool-II — прекрасное средство, которое нужно применять при масштабировании PostgreSQL.

Глава 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 коннектов) 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:

Listing 7.1: PgBouncer

```
1 pgbouncer [-d][-R][-v][-u user] <pgbouncer.ini>
```

Простой пример для конфига:

Listing 7.2: PgBouncer

```
1  [databases]
2  template1 = host=127.0.0.1 port=5432 dbname=template1
3  [pgbouncer]
4  listen_port = 6543
5  listen_addr = 127.0.0.1
6  auth_type = md5
7  auth_file = userlist.txt
8  logfile = pgbouncer.log
9  pidfile = pgbouncer.pid
10 admin_users = someuser
```

Нужно создать файл пользователей userlist.txt примерного содержания:”someuser” ”same_password_as_in_server”

Административный доступ из консоли к базе данных pgbouncer:

Listing 7.3: PgBouncer

```
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 фичи, 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.

Глава 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 кэши-

рования, которому нет дела к транзакциям СУБД, «снимки» БД могут быть с неверними данними. 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 заниматся своей непосредственной целью — хранение и предоставление целосности данных.

Но, несмотря на все эти минуси, многим разработчикам требуется кэширование на уровне базы данных. Для организации кэширования существует два инструмента для PostgreSQL:

- Pgmemcache (с memcached)
- Pgpool-II (query cache)

8.2 Pgmemcache

Memcached[1] — 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[2] — 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. Попробуем, что из себя представляет данный тип кэширования.

Установка

Во время написания этой главы была доступна 2.0.4 версия pgmemcache[3]. Pgmemcache будет устанавливатся и настраиватся на PostgreSQL версии

[1] http://memcached.org/

[2] http://pgfoundry.org/projects/pgmemcache/

[3] http://pgfoundry.org/frs/download.php/2672/pgmemcache_2.0.4.tar.bz2

8.4 (для версии 9.0 все аналогично), операционная система — Ubuntu Server 10.10. 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:

  Listing 8.1: Установка из исходников

```
1 $ make
2 $ sudo make install
```

- **Создание и установка deb пакета (для Debian, Ubuntu)**

  Иногда, если у Вас на серверах Debian или Ubuntu, удобнее создать deb пакет нужной программы и распостранать его через собственный репозиторий на все сервера с PostgreSQL:

  Listing 8.2: Создание и установка deb пакета

```
1 $ sudo apt-get install libmemcached-dev
     postgresql-server-dev-8.4 libpq-dev devscripts yada
     flex bison
2 $ make deb84
3 # устанавливаем deb пакет
4 $ sudo dpkg -i ../postgresql-pgmemcache-8.4*.deb
```

  Для версии 2.0.4 утилита yada выдавала ошибку при создании deb пакета со следующим текстом:

  Listing 8.3: Создание и установка deb пакета

```
1 Cannot recognize source name in 'debian/changelog' at
     /usr/bin/yada line 145, <CHANGELOG> line 1.
2 make: *** [deb84] Ошибка 9
```

  Для устранения этой ошибки потребуется удалить первую строчку текста в «debian/changelog» в каталоге, котором происходит сборка:

Listing 8.4: Создание и установка deb пакета

```
1 $PostgreSQL: pgmemcache/debian/changelog,v 1.2
     2010/05/05 19:56:50 ormod Exp $ <---- удалить
2 pgmemcache (2.0.4) unstable; urgency=low

4   * v2.0.4
```

Устранив эту проблему, сборка deb пакета не должна составить никаких проблем.

Настройка

После успешной установки Pgmemcache потребуется добавит во все базы данных (на которых вы хотите использовать Pgmemcache) функции для работы с этой библиотекой:

Listing 8.5: Настройка

```
1 % psql [mydbname] [pguser]
2 [mydbname]=# BEGIN;
3 [mydbname]=# \i
     /usr/local/postgresql/share/contrib/pgmemcache.sql
4 # для Debian: \i
     /usr/share/postgresql/8.4/contrib/pgmemcache.sql
5 [mydbname]=# 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:

Listing 8.6: Настройка default_servers

```
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:

Listing 8.7: Настройка pgmemcache

```
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 узнаем информацию по memcached серверах:

Listing 8.8: Проверка memcache_stats

```
1  pgmemcache=# SELECT memcache_stats();
2        memcache_stats
3  ---------------------------

5   Server: 127.0.0.1 (11211)
6   pid: 1116
7   uptime: 70
8   time: 1289598098
9   version: 1.4.5
0   pointer_size: 32
1   rusage_user: 0.0
2   rusage_system: 0.24001
3   curr_items: 0
4   total_items: 0
5   bytes: 0
6   curr_connections: 5
7   total_connections: 7
8   connection_structures: 6
9   cmd_get: 0
0   cmd_set: 0
1   get_hits: 0
2   get_misses: 0
3   evictions: 0
4   bytes_read: 20
```

```
 bytes_written: 782
 limit_maxbytes: 67108864
 threads: 4

(1 row)
```

Теперь сохраним данные в memcached и попробуем их забрать:

Listing 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 пригодится для создания последовательностей):

Listing 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:

Listing 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:

Listing 8.12: Триггер

```
1 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:

Listing 8.13: Очистка ключа в кэше

```
1 CREATE OR REPLACE FUNCTION auth_passwd_upd() RETURNS TRIGGER
    AS $$
2 BEGIN
3   IF OLD.passwd != NEW.passwd THEN
4     PERFORM memcache_delete('user_id_' || NEW.user_id ||
    '_password');
5   END IF;
6   RETURN NEW;
7 END;$$ LANGUAGE 'plpgsql';
```

Также нужен триггер на чистку кэша при удалении записи из СУБД:

Listing 8.14: Триггер

```
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 потребовается 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.

Таблица 8.1: Список команд pgmemcache

Команда	Описание
memcache_server_add('hostname:port'::TEXT) memcache_server_add('hostname'::TEXT)	Добавляет memcached сервер в список досту ных серверов. Если порт не указан, по умолч нию 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 существ ет.
newval = memcache_decr(key::TEXT, decrement::INT4) newval = memcache_decr(key::TEXT)	Если ключ существует и является целым чи лом, происходит уменьшение его значения указаное число (по умолчанию на единиц Возвращает целое число после уменьшения.
memcache_delete(key::TEXT, hold_timer::INTERVAL) memcache_delete(key::TEXT)	Удаляет указанный ключ. Если указать та мер, то ключ с таким же названием может бы добавлен только после окончания таймера.
memcache_flush_all()	Очищает все данные на всех memcached сер рах.
value = memcache_get(key::TEXT)	Выбирает ключ из кэша. Возвращает NUL если ключ не существует, иначе — текстову строку.
memcache_get_multi(keys::TEXT[]) memcache_get_multi(keys::BYTEA[])	Получает массив ключей из кэша. Во вращает список найденных записей в ви «ключ=значение».
newval = memcache_incr(key::TEXT, increment::INT4) newval = memcache_incr(key::TEXT)	Если ключ существует и является целым чи лом, происходит увеличение его значения указаное число (по умолчанию на единиц 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 ключ
memcache_set(key::TEXT, value::TEXT, expire::TIMESTAMPTZ) memcache_set(key::TEXT, value::TEXT, expire::INTERVAL) memcache_set(key::TEXT, value::TEXT)	Создаем ключ со значение. Если такой кл существует — заменяем в нем значение на ук заное.
stats = memcache_stats()	Возвращает статистику по всем сервер memcached.

Глава 9

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

9.1 Введение

Любой хороший сисадмин знает — бэкапы нужны всегда. На сколько бы надежна не 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 фаловой 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.

9.2 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:

Listing 9.1: Создаем бэкап с помощью pg_dump

```
1 pg_dump dbname > outfile
```

Для восстановления такого бэкапа достаточно выполнить:

Listing 9.2: Восстанавливаем бэкап

```
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:

Listing 9.3: Восстанавливаем бэкап

```
1 psql --set ON_ERROR_STOP=on dbname < infile
```

Также, можно делать бэкап и сразу восстанавливать его на другую базу:

Listing 9.4: Бекап в другую БД

```
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:

Listing 9.5: Бекап кластера PostgreSQL

```
1 pg_dumpall > outfile
```

Для восстановления такого бэкапа достаточно выполнить от суперпользователя:

Listing 9.6: Восстановления бэкапа PostgreSQL

```
1 psql -f infile postgres
```

SQL бэкап больших баз данных

Некоторые операционные системы имеют ограничения на максимальный размер файла, что может вызывают 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:

  Listing 9.7: Сжатие бэкапа PostgreSQL

  ```
  1 pg_dump dbname | gzip > filename.gz
  ```

  Восстановление:

  Listing 9.8: Восстановление бэкапа PostgreSQL

  ```
  1 gunzip -c filename.gz | psql dbname
  ```

  или

  Listing 9.9: Восстановление бэкапа PostgreSQL

  ```
  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:

  Listing 9.10: Создание бэкапа PostgreSQL

```
1 pg_dump dbname | split -b 1m - filename
```

  Восстановление:

  Listing 9.11: Восстановление бэкапа PostgreSQL

```
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:

  Listing 9.12: Создание бэкапа PostgreSQL

```
1 pg_dump -Fc dbname > filename
```

  Через psql такой бэкап не восстановить, но для этого есть утилита pg_restore:

  Listing 9.13: Восстановление бэкапа PostgreSQL

```
1 pg_restore -d dbname filename
```

При слишком большой базе данных, вариант с командой split нужно комбинировать с сжатием данных.

9.3 Бекап уровня файловой системы

Альтернативный метод резервного копирования заключается в непосредственном копировании файлов, которые PostgreSQL использует для хранения данных в базе данных. Например:

Listing 9.14: Бэкап PostgreSQL файлов

```
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 остановленна, для того, чтобы получить актуальный бэкап (PostgreSQL держит множество обьектов 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 система, то такой метод бэкапа будет очень не надежным — 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.

9.4 Непрерывное резервное копирование

PostgreSQL поддерживает упреждаюшию запись логов (Write Ahead Log, WAL) в pg_xlog директорию, которая находится в директории дан-

ных СУБД. В логи пишутся все изменения сделаные 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 третью 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 восстановливаем 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 резервный сервер PostgreSQL всегда в актуальном состоянии (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:

Listing 9.15: Настройка архивирования

```
1 archive_mode = on # enable archiving
2 archive_command = 'cp -v %p /data/pgsql/archives/%f'
3 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 список задач крона и, таким образом, файлы могут автоматически перемещаться между хостми каждые несколько минут.

Listing 9.16: Копирование WAL файлов на другой хост

```
1 rsync -avz --delete prod1:/data/pgsql/archives/ \
2 /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 создается в каталоге данных PostgreSQL создать файл recovery.conf с заданной командой копирования файлов из архива в нужную директорию:

Listing 9.17: recovery.conf

```
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 http://www.postgresql.org/docs/9.0/static/continuous-archiving.html.

9.5 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).

Глава 10

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

10.1 Введение

Многие разработчики крупных проектов сталкиваются с проблемой, когда один единственный сервер базы данных никак не может справится 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 иметь характеристики 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 Твиттер или Фейсбук пишете). Если вдруг такое происходит — возможно система неверно спроектирована, и её реализацию следует пересмотреть.

10.2 Проблема чтения данных

Обычно, первой стучится в дверь проблема с чтением данных, когда СУБД не в состоянии обеспечить то количество выборок, которое требуется.

Симптомы

10.3 Проблема записи данных