Работа с Postgresql

настройка, масштабирование

Алексей Васильев

The Publisher

Привет 2003

Оглавление

1

Настройка производительности

1.1 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).

1.2 Настройка сервера

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

Используемая память

Общий буфер сервера: shared_buffers

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

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

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

В то же время не следует устанавливать это значение слишком большим: это НЕ вся память, которая нужна для работы PostgreSQL, это только размер разделяемой между процессами PostgreSQL памяти, которая нужна для выполнения активных операций. Она должна занимать меньшую часть оперативной памяти вашего компьютера, так как PostgreSQL полагается на то, что операционная система кэширует файлы, и не старается дублировать эту работу. Кроме того, чем больше памяти будет отдано под буфер, тем меньше останется операционной системе и другим приложениям, что может привести к своппингу.

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

На выделенных серверах полезным объемом будет значение от 8 МБ до 2 ГБ. Объем может быть выше, если у вас большие активные порции базы данных, сложные запросы, большое число одновременных соединений, длительные транзакции, вам доступен большой объем оперативной памяти или большее количество процессоров. И, конечно же, не забываем об остальных приложениях. Выделив слишком много памяти для базы данных, мы можем получить ухудшение производительности. В качестве начальных значений можете попробовать следующие:

- Начните с 4 МБ (512) для рабочей станции

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

Для тонкой настройки параметра установите для него большое значение и потестируйте базу при обычной нагрузке. Проверяйте использование разделяемой памяти при помощи ipcs или других утилит. 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/ke 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

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

лишь помечает занимаемое ими место как свободное. Для информации о свободном месте есть следующие настройки:

- **max_fsm_relations**

  Максимальное количество таблиц, для которых будет отслеживаться свободное место в общей карте свободного пространства. Эти данные собираются VACUUM. Параметр max_fsm_relations должен быть не меньше общего количества таблиц во всех базах данной установки (лучше с запасом).

- **max_fsm_pages**

  Данный параметр определяет размер реестра, в котором хранится информация о частично освобождённых страницах данных, готовых к заполнению новыми данными. Значение этого параметра нужно установить чуть больше, чем полное число страниц, которые могут быть затронуты операциями обновления или удаления между выполнением VACUUM. Чтобы определить это число, можно запустить VACUUM VERBOSE ANALYZE и выяснить общее число страниц, используемых базой данных. max_fsm_pages обычно требует немного памяти, так что на этом параметре лучше не экономить.

Если эти параметры 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.

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_hit и понижайте vacuum_cost_page_limit. Это ослабит влияние VACUUM,

увеличив время его выполнения. В тестах с параллельными транзакциями Ян Вик (Jan Wieck) получил, что при значениях delay — 200, page_hit — 6 и 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_SYNC

Не все методы доступны на определенных платформах. По умолчанию устанавливается первый, который доступен в системе.

- **full_page_writes**

  Установите данный параметр в off, если fsync=off. Иначе, когда этот параметр on, PostgreSQL записывает содержимое каждой страницы в журнал транзакций во время первой модификации таблицы после контрольной точки. Это необходимо потому что страницы могут записаться лишь частично если в ходе процесса ОС "упала". Это приволит к тому, что на диске оказаываются 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.(Потому что журнал транзакций все время начинается с контрольной точки. Единственный способ снижения объема записи 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.

Планировщик запросов

Следующие настройки помогают планировщику запросов правильно оценивать стоимости различных операций и выбирать оптимальный план выполнения запроса. Существуют 2 глобальные настройки планировщика, на которые стоит обратить внимание:

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

- **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 много больше размеров оперативной памяти, попробуйте поднять значение параметра. Можно подойти к выбору оптимального значения и со стороны производительности запросов. Если планировщик запросов чаще, чем необходимо, предпочитает последовательные просмотры (sequential scans) просмотрам с использованием индекса (index 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. Эта подсистема контролируется следующими параметрами, принимающими значения true/false:

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

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

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

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

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

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

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

1.3 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.

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.

1.4 Примеры настроек

Среднестатическая 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;

[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

- 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;

Среднестатическая 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

Среднестатическая 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

Среднестатическая 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

1.5 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/

```
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.

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

- Команды, изменяющие данные в таблице, должны изменить также и индексы. Очевидно, чем больше индексов построено для таблицы, тем медленнее это будет происходить.

- Оптимизатор перебирает возможные пути выполнения запросов. Если построено много ненужных индексов, то этот перебор будет идти дольше.

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

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.

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!

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

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.

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

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>

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

Проблема Запрос вида

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

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

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:

   ```
   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).

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

Медленый 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.

```
DISTINCT

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

Time: 580,553 ms

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

Time: 36,281 ms


GROUP BY

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

Time: 26,562 ms

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

Time: 25,270 ms
```

1.7 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.

2 Репликация

2.1 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/UPDATE и хранимые процедуры для выполнения действий.

[1] http://www.slony.info/

- **PGCluster**[2] — синхронная Multi-Master репликация. Проект на мой взгляд мертв, поскольку уже год не обновлялся.
- **pgpool-I/II**[3] — это замечательная тулза 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;
  - разбивать запросы, для параллельного выполнения над большими объемами данных.
- **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] — написаная 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.

2.2 Slony-I

Введение

Slony это система репликации реального времени, позволяющая организовать синхронизацию нескольких серверов PostgreSQL по сети. Slony

[2] http://pgfoundry.org/projects/pgcluster/
[3] http://pgpool.projects.postgresql.org/
[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/

использует триггеры Postgre для привязки к событиям INSERT/ DELETE/UPDATE и хранимые процедуры для выполнения действий.

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

О slonik-е

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

Установка

Установка на Ubuntu производится простой командой:

```
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.

```
pgsql@customers_master$ createuser -a -d slony
pgsql@customers_master$ psql -d template1 -c "alter \
user slony with password 'slony_user_password';"
```

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

Подготовка 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:

```
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):

```
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!

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

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

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

Инициализация БД и plpgsql на slave

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

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

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

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

Продолжаем. Создаём таблицы и всё остальное на slave:

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

pg_dump -s сдампит только структуру нашей БД.

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

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

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

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

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

Инициализация кластера

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

На мастер-хосте запускаем такой скрипт:

```
#!/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' );

echo 'Adding tables to the subscription set';

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

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

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

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

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

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

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

Подписываем slave-узел на replication set

Скрипт:

```
#!/bin/sh

CLUSTER=customers_rep

DBNAME1=customers
DBNAME2=customers

HOST1=customers_master.com
HOST2=customers_slave.com

PORT1=5432
PORT2=5432
```

```
SLONY_USER=slony

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

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

EOF
```

Старт репликации

Теперь, на обоих узлах необходимо запустить демона репликации.

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

и

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

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

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

Общие задачи

Добавление ещё одного узла в работающую схему репликации

Выполнить 2.2.1 и выполнить 2.2.2.

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 2.2.2) делаем следующее:

```
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:

```
#!/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:

```
#!/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.

```
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:

```
%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
host=mainhost.com port=5432 user=slony_user
password=slony_user_pass" pa_connretry=10
CONFIG storeListen: li_origin=1 li_receiver=3
li_provider=1
CONFIG storeSet: set_id=1 set_origin=1
set_comment='CustomersDB replication set'
WARN remoteWorker_wakeup: node 1 - no worker thread
CONFIG storeSubscribe: sub_set=1 sub_provider=1 sub_forward='f'
WARN remoteWorker_wakeup: node 1 - no worker thread
CONFIG enableSubscription: sub_set=1
WARN remoteWorker_wakeup: node 1 - no worker thread
CONFIG main: configuration complete - starting threads
CONFIG enableNode: no_id=1
CONFIG enableNode: no_id=2
CONFIG enableNode: no_id=4
ERROR remoteWorkerThread_1: "begin transaction; set
transaction isolation level
serializable; lock table "_customers_rep".sl_config_lock; select
"_customers_rep".enableSubscription(1, 1, 4);
notify "_customers_rep_Event"; notify "_customers_rep_Confirm";
insert into "_customers_rep".sl_event (ev_origin, ev_seqno,
ev_timestamp, ev_minxid, ev_maxxid, ev_xip,
ev_type , ev_data1, ev_data2, ev_data3, ev_data4 ) values
('1', '219440',
'2005-05-05 18:52:42.708351', '52501283', '52501292',
''''52501283''', 'ENABLE_SUBSCRIPTION',
'1', '1', '4', 'f'); insert into "_customers_rep".
sl_confirm (con_origin, con_received,
con_seqno, con_timestamp) values (1, 3, '219440',
CURRENT_TIMESTAMP); commit transaction;"
PGRES_FATAL_ERROR ERROR: insert or update on table
"sl_subscribe" violates foreign key
```

```
constraint "sl_subscribe-sl_path-ref"
DETAIL: Key (sub_provider,sub_receiver)=(1,4)
is not present in table "sl_path".
INFO remoteListenThread_1: disconnecting from
'dbname=customers host=mainhost.com
port=5432 user=slony_user password=slony_user_pass'
%
```

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

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

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

```
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:

```
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:

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

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

2.3 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(failover) 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:

```
$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:

```
$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

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

Дальше проверим, что все у нас правильно установилось

```
$londiste.py -V
Skytools version 2.1.11
$pgqadm.py -V
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):

```
[pgqadm]
# название
job_name = db1-ticker

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

# Задержка между запусками обслуживания
# (ротация очередей и т.п.) в секундах
maint_delay = 600

# Задержка между проверками наличия активности
# (новых пакетов данных) в секундах
loop_delay = 0.1

# log и pid демона
logfile = /var/log/%(job_name)s.log
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:

```
pgqadm.py /etc/skytools/db1-ticker.ini install
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 воспользовать этой командой:

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

или если потребуется «убить» ticker:

```
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):

```
[londiste]
# название
job_name = db1-londiste

# мастер база данных
provider_db = dbname=db1 port=5432 host=host1
# слейв база данных
subscriber_db = dbname=db1 host=host2

# Это будет использоваться в качестве
# SQL-идентификатора, т.ч. не используйте
# точки и пробелы.
# ВАЖНО! Если есть живая репликация на другой слейв,
# именуем очередь так-же
pgq_queue_name = db1-londiste-queue

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

# рзмер лога
log_size = 5242880
```

```
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:

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

и подобным образом на стороне слейва:

```
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:

```
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:

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

и что со слейва:

```
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:

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

Для слейва:

```
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:

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

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

```
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):

```
(
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:

```
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.

```
SELECT queue_name, consumer_name, lag, last_seen
  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 потребовается удалить все ваши настройки для того, чтобы начать все заново. Для PGQ, чтобы остановить накопление данных, используйте следующие API:

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

Или воспользуйтесь pgqadm.py:

```
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.

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

В нашем случае, это было более чем 5 миллионов и 400 тысяч событий. 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:

```
pgq_lazy_fetch = 500
```

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

2.4 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.

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

Для других систем можно поставить из исходников[11]:

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

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

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

```
tar xvfz Bucardo-4.4.0.tar.gz
cd Bucardo-4.4.0
perl Makefile.PL
make
sudo make install
```

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

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

Можем приступать к настройке.

Настройка

Инициализация Bucardo

Запускаем установку командой:

```
bucardo_ctl install
```

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

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

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

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

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

```
bucardo_ctl add db master_db name=master
bucardo_ctl add all tables herd=all_tables
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:

```
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):

```
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_db 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:

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

Запуск репликации

Запуск репликации:

```
bucardo_ctl start
```

Остановка репликации:

```
bucardo_ctl stop
```

Общие задачи

Просмотр значений конфигурации

Просто используя эту команду:

```
bucardo_ctl show all
```

Изменения значений конфигурации

```
bucardo_ctl set name=value
```

Например:

```
bucardo_ctl set syslog_facility=LOG_LOCAL3
```

Перегрузка конфигурации

```
bucardo_ctl reload_config
```

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

2.5 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:

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

```
sudo gem install mysql
```

Для PostgreSQL:

```
sudo gem install postgres
```

3. Устанавливаем rubyrep:

```
sudo gem install rubyrep
```

Настройка

Создание файла конфигурации

Выполним команду:

```
rubyrep generate myrubyrep.conf
```

Команда generate создала пример конфигурации в файл myrubyrep.conf:

```
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:

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

Пример вывода:

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

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

Синхронизация баз данных

Выполним команду:

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

Также можно указать только какие таблицы в базах данных синхронизировать:

```
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:

```
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:

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

Устранение неисправностей

Ошибка при запуске репликации

При зепуске rubyrep через Ruby может возникнуть подобная ошибка:

```
$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:

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