Работа с Postgresql

настройка, масштабирование

Алексей Васильев

http://leopard.in.ua

При написании книги(мануала, или просто шпаргалки) использовались материалы:

- PostgreSQL: настройка производительности. Алексей Борзов (Sad Spirit) borz_off@cs.msu.su, http://www.phpclub.ru/detail/store/pdf/postgresql-performance.pdf
- Настройка репликации в PostgreSQL с помощью системы Slony-I, Eugene Kuzin eugene@kuzin.net, http://www.kuzin.net/work/sloniki-privet.html
- Установка Londiste в подробностях, Sergey Konoplev gray.ru@gmail.com, http://gray-hemp.blogspot.com/2010/04/londiste.html
- Учебное руководство по pgpool-II, Dmitry Stasyuk, http://undenied.ru/2009/03/04/uchebnoe-rukovodstvo-po-pgpool-ii/
- Горизонтальное масштабирование PostgreSQL с помощью PL/Proxy, Чиркин Дима dmitry.chirkin@gmail.com, http://habrahabr.ru/blogs/postgresql/45475/
- Hadoop, Иван Блинков wordpress@insight-it.ru, http://www.insight-it.ru/masshtabiruemost/hadoop/
- Up and Running with HadoopDB, Padraig O'Sullivan, http://posulliv.github.com/2010/05/10/hadoopdb-mysql.html
- Масштабирование PostgreSQL: готовые решения от Skype, Иван Золотухин, http://postgresmen.ru/articles/view/25

Оглавление

Оглавление **3**

1 Настройка производительности **6**
1.1 Введение 6
Не используйте настройки по умолчанию 6
Используйте актуальную версию сервера 7
Стоит ли доверять тестам производительности 8
1.2 Настройка сервера 9
Используемая память 9
Журнал транзакций и контрольные точки 13
Планировщик запросов 15
Сбор статистики 16
1.3 Диски и файловые системы 17
Перенос журнала транзакций на отдельный диск 18
1.4 Примеры настроек 18
Среднестатическая настройка для максимальной производительности 18
Среднестатическая настройка для оконного приложения (1С), 2 ГБ памяти 19
Среднестатическая настройка для Web приложения, 2 ГБ памяти 20
Среднестатическая настройка для Web приложения, 8 ГБ памяти 20
1.5 Автоматическое создание оптимальных настроек: pgtune . . 20
1.6 Оптимизация БД и приложения 21
Поддержание базы в порядке 22
Использование индексов 22
Перенос логики на сторону сервера 25
Оптимизация конкретных запросов 26
Оптимизация запросов с помощью pgFouine 28
1.7 Заключение 29

2 Репликация **30**

2.1 Введение 30
2.2 Slony-I 32
Введение 32
Установка 33
Настройка 33
Общие задачи 38
Устранение неисправностей 40
2.3 Londiste 43
Введение 43
Установка 44
Настройка 45
Общие задачи 48
Устранение неисправностей 50
2.4 Bucardo 50
Введение 50
Установка 50
Настройка 51
Общие задачи 53
2.5 RubyRep 54
Введение 54
Установка 54
Настройка 55
Устранение неисправностей 56
2.6 Заключение 57

3 Кластеризация БД **59**
3.1 Введение 59
3.2 PL/Proxy 59
Установка 60
Настройка 60
Все ли так просто? 64
3.3 HadoopDB 64
Установка и настройка 69
Заключение 78
3.4 Заключение 78

4 PgPool-II **79**
4.1 Введение 79
4.2 Давайте начнем! 80
Установка pgpool-II 80
Файлы конфигурации 81
Настройка команд PCP 81
Подготовка узлов базы данных 82
Запуск/Остановка pgpool-II 83
4.3 Ваша первая репликация 84

Настройка репликации 84
Проверка репликации 84
4.4 Ваш первый параллельный запрос 85
Настройка параллельного запроса 85
Настройка SystemDB 86
Установка правил распределения данных 89
Установка правил репликации 90
Проверка параллельного запроса 90
4.5 Master-slave режим 91
4.6 Онлайн востановление 92
4.7 Заключение 92

5 Мультиплексоры соединений 93
5.1 Введение 93
5.2 PgBouncer 93
5.3 PgPool-II vs PgBouncer 94

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.

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_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 брается 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
```

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 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

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

- Для записи запросов, длящихся дольше n миллисекунд:

```
log_min_duration_statement = n
log_duration = off
log_statement = 'none'
```

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

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

```
$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:

```
$ 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.postgresql.org.

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

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 замечательный инструмент для PostgreSQL (лучше сразу работать с II версией). Позволяет делать:
  - репликацию (в том числе, с автоматическим переключением на резервный stand-by сервер);
  - online-бэкап;
  - pooling коннектов;
  - очередь соединений;
  - балансировку SELECT-запросов на несколько postgresql-серверов;
  - разбиение запросов для параллельного выполнения над большими объемами данных.
- **Bucardo**[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.

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|
```

2.6 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. 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 расплачивается скоростью работы — самый медленный из всех (синхронизация больших обьемов данных между таблицами).

3

Кластеризация БД

3.1 Введение

Кластер (в информационных технологиях) — группа серверов (программных или аппаратных), объединённых логически, способных обрабатывать идентичные запросы и использующихся как единый ресурс. Для PostgreSQL это означает, что несколько серверов баз данных ведут себя как одна база данных. В большинстве случаев, кластеры серверов функционируют на раздельных компьютерах. Это позволяет повышать производительность за счёт распределения нагрузки на аппаратные ресурсы и обеспечивает отказоустойчивость на аппаратном уровне.

Для создания кластера PostgreSQL существует несколько решений:

- **Greenplum Database**[1]
- **GridSQL for EnterpriseDB Advanced Server**[2]
- **PL/Proxy**[3]
- **HadoopDB**[4]

3.2 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

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

таблицу с пользователями, например, по первой латинской букве имени — на 26 узлов. При этом приложение, которое работает непосредственно с прокси-базой, ничего не будет замечать: запрос на авторизацию, например, сам будет направлен прокси-сервером на нужный узел. То есть администратор баз данных может проводить масштабирование системы практически независимо от разработчиков приложения.

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

Недостатки и ограничения:

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

Установка

1. Скачать PL/Proxy[5] 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:

```
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!

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

Для начала настроим node1 и node2. Команды написаные нижу нужно выполнять на каждом ноде.

Создадим базу данных plproxytest(если её ещё нет):

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

Добавляем табличку users:

```
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:

```
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:

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

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

```
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:

```
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:

```
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
  COST 100
  ROWS 1000;
```

```
ALTER FUNCTION public.get_cluster_partitions(text)
OWNER TO postgres;
```

И последняя:

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

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

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

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

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

Пробуем извлечь данные. Для этого напишем новую серверную функцию:

```
CREATE OR REPLACE FUNCTION
public.get_user_email(i_username text)
```

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

И попробуем её вызвать:

```
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 сообществе 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.

3.3 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

- **Namenode**

  Этот компонент системы осуществляет всю работу с метаданными. Он должен быть запущен только на одном компьютере в кластере.

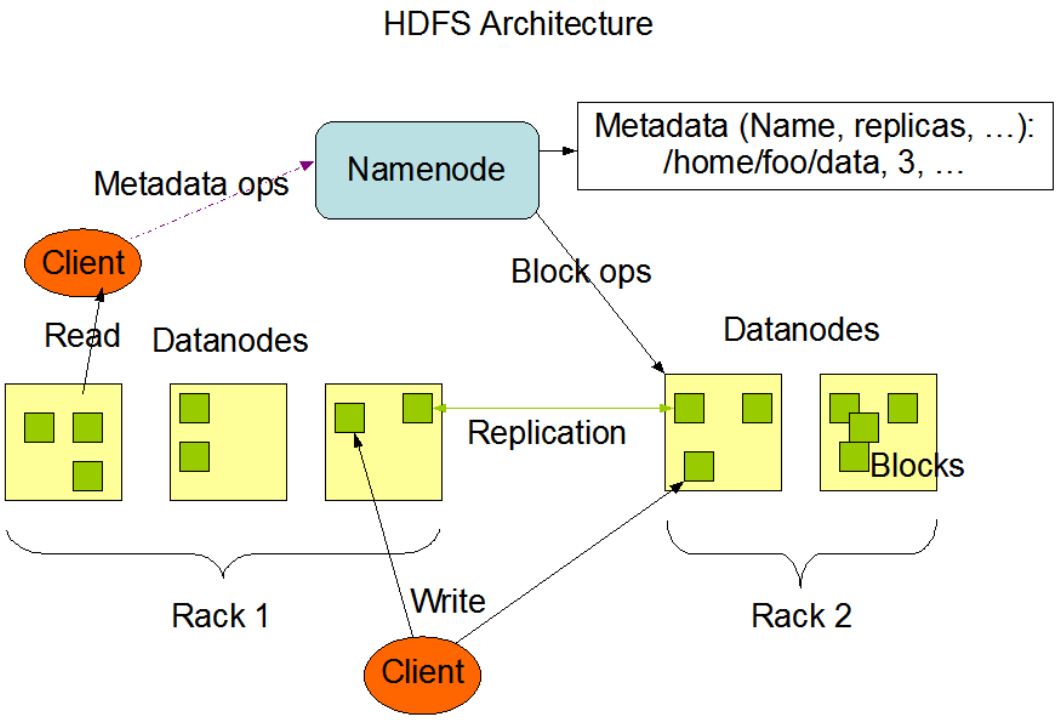

Рис. 3.1: Архитектура HDFS

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 групируются по ключу и передаются на следующий этап.

- **Reduce**

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

Задания выполняются, подобно файловой системе, на всех машинах в кластере (чаще всего одних и тех же). Одна из них выполняет роль

управления работой остальных — JobTracker, остальные же ее бесприкословно слушаются — TaskTracker. В задачи JobTracker'а входит составление расписания выполняемых работ, наблюдение за ходом выполнения, и перераспределение в случае возникновения сбоев.

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

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

HBase

В рамках Hadoop доступна еще и система хранения данных, которую правда сложно назвать СУБД в традиционном смысле этого слова. Чаще проводят аналогии с проприетарной системой этого же плана от Google — BigTable.

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

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

  Далее действия выполняем от его имени:

  ```
  su hadoop
  ```

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

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

  И добавляем его в список авторизованных ключей:

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

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

```
ssh localhost
```

Не забываем предварительно инициализировать sshd:

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

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

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

Дальше скачиваем и устанавливаем Hadoop:

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

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

```
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:

```
<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:

```
<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:

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

Теперь необходимо отформатировать 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:

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

Остановка Hadoop производится скриптом stop-all.sh.

Установка HadoopDB и Hive

Теперь скачаем HaddopDB[6] и распакуем hadoopdb.jar в $HADOOP_HOME/lib:

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

Также нам потребуется PostgreSQL JDBC библиотека. Скачайте её[7] и распакуйте в директорию $HADOOP_HOME/lib.

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

```
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:

```
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:

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

hive> quit;
$
```

Тестирование

Теперь проведем тестирование. Скачаем бенчмарк:

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

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

```
use strict;
use warnings;

my $CUR_HOSTNAME = ‘hostname -s‘;
```

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

```
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:

```
$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:

```
#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:

```
java -cp $HADOOP_HOME/lib/hadoopdb.jar \
```

```
> edu.yale.cs.hadoopdb.catalog.SimpleCatalogGenerator \
> Catalog.properties
hadoop dfs -put HadoopDB.xml HadoopDB.xml
```

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

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

Приблизительный результат:

```
$java -cp $CLASSPATH:hadoopdb.jar edu.yale.cs.hadoopdb.benchmark.GrepTaskDB \
> -pattern %wo% -output padraig -hadoop.config.file HadoopDB.xml
14.08.2010 19:08:48 edu.yale.cs.hadoopdb.exec.DBJobBase initConf
INFO: SELECT key1, field FROM grep WHERE field LIKE '%%wo%%';
14.08.2010 19:08:48 org.apache.hadoop.metrics.jvm.JvmMetrics init
INFO: Initializing JVM Metrics with processName=JobTracker, sessionId=
14.08.2010 19:08:48 org.apache.hadoop.mapred.JobClient configureCommandLineOptic
WARNING: Use GenericOptionsParser for parsing the arguments.
Applications should implement Tool for the same.
14.08.2010 19:08:48 org.apache.hadoop.mapred.JobClient monitorAndPrintJob
INFO: Running job: job_local_0001
14.08.2010 19:08:48 edu.yale.cs.hadoopdb.connector.AbstractDBRecordReader getCon
INFO: Data locality failed for leo-pgsql
14.08.2010 19:08:48 edu.yale.cs.hadoopdb.connector.AbstractDBRecordReader getCon
INFO: Task from leo-pgsql is connecting to chunk 0 on host localhost with
db url jdbc:postgresql://localhost:5434/grep0
14.08.2010 19:08:48 org.apache.hadoop.mapred.MapTask runOldMapper
INFO: numReduceTasks: 0
14.08.2010 19:08:48 edu.yale.cs.hadoopdb.connector.AbstractDBRecordReader close
INFO: DB times (ms): connection = 104, query execution = 20, row retrieval  = 79
14.08.2010 19:08:48 edu.yale.cs.hadoopdb.connector.AbstractDBRecordReader close
INFO: Rows retrieved = 3
14.08.2010 19:08:48 org.apache.hadoop.mapred.Task done
INFO: Task:attempt_local_0001_m_000000_0 is done. And is in the process of commi
14.08.2010 19:08:48 org.apache.hadoop.mapred.LocalJobRunner$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/pad
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
INFO:   FileSystemCounters
14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
INFO:     FILE_BYTES_READ=141370
14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
INFO:     FILE_BYTES_WRITTEN=153336
14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
INFO:   Map-Reduce Framework
14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
INFO:     Map input records=3
14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
INFO:     Spilled Records=0
14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
INFO:     Map input bytes=3
14.08.2010 19:08:49 org.apache.hadoop.mapred.Counters log
INFO:     Map output records=3
14.08.2010 19:08:49 edu.yale.cs.hadoopdb.exec.DBJobBase run
INFO:
JOB TIME : 1828 ms.


$
```

Результат сохранен в HDFS, в папке padraig:

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

Проверим данные в PostgreSQL:

```
psql> select * from grep where field like '%wo%';
+--------------------------------+-------------------+
| key1                           | field
|
+--------------------------------+-------------------+
some data
```

```
1 rows in set (0.00 sec)

psql>
```

Значения совадают. Все работает как требуется.

Заключение

В данной статье не показывается, как настроить Hive для работы 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.

3.4 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.

4

PgPool-II

4.1 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 использует преимущество функции репликации для уменьшения нагрузки на каждый из серверов PostgreSQL распределяя запросы SELECT на несколько серверов, тем самым увеличивая производительность системы вцелом. В лучшем случае производительность возрастает пропорционально числу серверов PostgreSQL. Балансировка нагрузки лучше всего работает в случае когда много пользователей выполняют много запросов в одно и 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.postgresql.org/pgpool-II/doc/tutorial-en.html.

4.2 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.

```
$ ./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.

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

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

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

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

```
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.

```
$ 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 «*».

```
listen_addresses = 'localhost'
port = 9999
```

Мы будем использовать параметры по-умолчанию в этом руководстве.

В Ubuntu Linux конфиг находится /etc/pgpool.conf.

Настройка команд PCP

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

```
postgres:e8a48653851e28c69d0506508fb27fc5
```

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

```
$ 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.

```
$ /usr/bin/pg_md5 postgres
e8a48653851e28c69d0506508fb27fc5
```

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

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

```
pcp_port = 9898
```

Подготовка узлов базы 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.

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

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

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

Запуск/Остановка pgpool-II

Для старта pgpool-II выполните в терминале следующую команду.

```
$ 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.

```
$ pgpool -n &
```

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

```
$ 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.

```
$ 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.

```
$ 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.

```
$ pgpool -m fast stop
```

4.3 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 «4.2. 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.

```
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.

```
load_balance_mode = true
```

В этом разделе мы включили оба параметра replication_mode и load_balance_mode.

Проверка репликации

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

```
$ createdb -p 9999 bench_replication
```

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

```
$ 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).

```
$ 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
```

4.4 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 «4.2. 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.

```
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.

```
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 «4.3. Ваша первая репликация» не может быть повторно использована.

```
replication_mode = true
load_balance_mode = false
```

ИЛИ

```
replication_mode = false
load_balance_mode = true
```

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

Настройка SystemDB

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

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

```
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».

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

Установка dblink

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

Для установки dblink на вашей системе выполните следующие команды.

```
$ 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.

```
$ 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».

```
$ 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.

```
CREATE TABLE pgpool_catalog.dist_def (
    dbname text, -- имя базы данных
```

```
    schema_name text, -- имя схемы
    table_name text, -- имя таблицы
    col_name text NOT NULL CHECK (col_name = ANY (col_list)),
    -- столбец-ключ для распределения данных
    col_list text[] NOT NULL, -- список имен столбцов
    type_list text[] NOT NULL, -- список типов столбцов
    dist_def_func text NOT NULL,
    -- имя функции распределения данных
    PRIMARY KEY (dbname, schema_name, table_name)
);
```

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

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

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

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

Создание таблицы replicate_def

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

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

Установка правил распределения данных

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

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

```
$ 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.

```
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.

```
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.

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

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

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

```
$ 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.

```
$ 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.

```
$ 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.

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

4.5 Master-slave режим

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

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

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

4.6 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.

В данной статье не будет рассматривается, как настроить онлайн восстановление. Данную информацию можно подчеркнуть из
http://pgpool.projects.postgresql.org/pgpool-II/doc/pgpool-en.html#online-recovery
или
http://pgpool.projects.postgresql.org/contrib_docs/pgpool-II_for_beginners.pdf

4.7 Заключение

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

5

Мультиплексоры соединений

5.1 Введение

Мультиплексоры соединений(программы для создания пула коннектов) позволяют уменьшить накладные расходы на базу данных, когда огромное количество физических соединений тянет производительность PostgreSQL вниз. Это особенно важно на Windows, когда система ограничивает большое количество соединений. Это также важно для веб-приложений, где количество соединений может быть очень большим.

Программы, которые создают пулы соединений:

- PgBouncer
- Pgpool

Также некоторые администраторы PostgreSQL с успехом используют Memcached для уменьшения работы БД за счет кэширования данных.

5.2 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.

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:

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

Простой пример для конфига:

```
[databases]
template1 = host=127.0.0.1 port=5432 dbname=template1
[pgbouncer]
listen_port = 6543
listen_addr = 127.0.0.1
auth_type = md5
auth_file = userlist.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = someuser
```

Нужно создать файл пользователей userlist.txt примерного содержания:"someusersame_password_as_in_server"

Админский доступ из консоли к базе данных pgbouncer:

```
$psql -h 127.0.0.1 -p 6543 pgbouncer
```

Здесь можно получить различную статистическую информацию с помощью команды SHOW.

5.3 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.