

А. Ю. Васильев aka leopard



Работа с
POSTGRESQL
настройка и масштабирование

5-Е ИЗДАНИЕ

Работа с PostgreSQL: настройка и масштабирование

А. Ю. Васильев aka leopard

Creative Commons Attribution-Noncommercial 4.0 International
2017

Оглавление

Оглавление	1
1 Введение	4
1.1 Что такое PostgreSQL?	4
2 Настройка производительности	6
2.1 Введение	6
2.2 Настройка сервера	9
2.3 Диски и файловые системы	19
2.4 Утилиты для тюнинга PostgreSQL	20
2.5 Оптимизация БД и приложения	24
2.6 Заключение	34
3 Индексы	35
3.1 Типы индексов	36
3.2 Возможности индексов	43
4 Партиционирование	45
4.1 Введение	45
4.2 Теория	46
4.3 Практика использования	47
4.4 Pg_partman	53
4.5 Pgslice	58
4.6 Заключение	62
5 Репликация	63
5.1 Введение	63
5.2 Поточковая репликация (Streaming Replication)	65
5.3 PostgreSQL Bi-Directional Replication (BDR)	79
5.4 Pglogical	80
5.5 Slony-I	83

5.6	Londiste	93
5.7	Bucardo	106
5.8	Заключение	113
6	Шардинг	114
6.1	Введение	114
6.2	PL/Proху	115
6.3	Postgres-X2	121
6.4	Postgres-XL	131
6.5	Citus	133
6.6	Greenplum Database	140
6.7	Заключение	153
7	PgPool-II	154
7.1	Введение	154
7.2	Установка и настройка	155
7.3	Настройка репликации	157
7.4	Параллельное выполнение запросов	158
7.5	Master-slave режим	164
7.6	Онлайн восстановление	166
7.7	Заключение	168
8	Мультиплексоры соединений	169
8.1	Введение	169
8.2	PgBouncer	169
8.3	PgPool-II vs PgBouncer	171
9	Кэширование в PostgreSQL	172
9.1	Введение	172
9.2	Pgmemcache	173
9.3	Заключение	178
10	Расширения	179
10.1	Введение	179
10.2	PostGIS	179
10.3	pgSphere	182
10.4	HStore	184
10.5	PLV8	186
10.6	Pg_repack	192
10.7	Pg_prewarm	200
10.8	Smlar	201
10.9	Multicorn	207
10.10	Pgaudit	215
10.11	Ltree	216
10.12	PostPic	221

10.13	Fuzzystrmatch	221
10.14	Pg_trgm	223
10.15	Cstore_fdw	226
10.16	Postgresql-hll	229
10.17	Tsearch2	233
10.18	PL/Proxy	235
10.19	Texcaller	235
10.20	Pgmemcache	235
10.21	Prefix	236
10.22	Dblink	238
10.23	Postgres_fdw	241
10.24	Pg_cron	243
10.25	PGStrom	245
10.26	ZomboDB	246
10.27	Заключение	248
11	Бэкап и восстановление PostgreSQL	249
11.1	Введение	249
11.2	SQL бэкап	250
11.3	Бэкап уровня файловой системы	252
11.4	Непрерывное резервное копирование	253
11.5	Утилиты для непрерывного резервного копирования	254
11.6	Заключение	268
12	Стратегии масштабирования для PostgreSQL	269
12.1	Введение	269
12.2	Проблема чтения данных	270
12.3	Проблема записи данных	271
12.4	Заключение	271
13	Утилиты для PostgreSQL	272
13.1	Введение	272
13.2	Pgcli	272
13.3	Pgloader	273
13.4	Postgres.app	273
13.5	pgAdmin	273
13.6	PostgREST	273
13.7	Ngx_postgres	273
13.8	Заключение	274
14	Полезные мелочи	275
14.1	Введение	275
14.2	Мелочи	275
	Литература	285

Введение

Послушайте — и Вы забудете,
посмотрите — и Вы
запомните, сделайте — и Вы
поймете

Конфуций

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

1.1 Что такое PostgreSQL?

PostgreSQL (произносится «Пост-Грес-Кью-Эль») — свободная объектно-реляционная система управления базами данных (СУБД).

PostgreSQL ведёт свою «родословную» от некоммерческой СУБД Postgres, разработанной, как и многие open-source проекты, в Калифорнийском университете в Беркли. К разработке Postgres, начавшейся в 1986 году, имел непосредственное отношение Майкл Стоунбрейкер, руководитель более раннего проекта Ingres, на тот момент уже приобретённого компанией Computer Associates. Само название «Postgres» расшифровывалось как «Post Ingres», соответственно, при создании Postgres были применены многие уже ранее сделанные наработки.

Стоунбрейкер и его студенты разрабатывали новую СУБД в течение восьми лет с 1986 по 1994 год. За этот период в синтаксис были введены процедуры, правила, пользовательские типы и многие другие компоненты. Работа не прошла даром — в 1995 году разработка снова разделилась: Стоунбрейкер использовал полученный опыт в создании коммерческой СУБД

1.1. Что такое PostgreSQL?

Шлустра, продвигаемой его собственной одноимённой компанией (приобретённой впоследствии компанией Informix), а его студенты разработали новую версию Postgres — Postgres95, в которой язык запросов POSTQUEL — наследие Ingres — был заменен на SQL.

В этот момент разработка Postgres95 была выведена за пределы университета и передана команде энтузиастов. С этого момента СУБД получила имя, под которым она известна и развивается в текущий момент — PostgreSQL.

На данный момент, в PostgreSQL имеются следующие ограничения:

Максимальный размер базы данных	Нет ограничений
Максимальный размер таблицы	32 Тбайт
Максимальный размер записи	1,6 Тбайт
Максимальный размер поля	1 Гбайт
Максимум записей в таблице	Нет ограничений
Максимум полей в записи	250—1600, в зависимости от типов полей
Максимум индексов в таблице	Нет ограничений

Согласно [результатам](#) автоматизированного исследования различного ПО на предмет ошибок, в исходном коде PostgreSQL было найдено 20 проблемных мест на 775000 строк исходного кода (в среднем, одна ошибка на 39000 строк кода). Для сравнения: MySQL — 97 проблем, одна ошибка на 4000 строк кода; FreeBSD (целиком) — 306 проблем, одна ошибка на 4000 строк кода; Linux (только ядро) — 950 проблем, одна ошибка на 10 000 строк кода.

Настройка производительности

Теперь я знаю тысячу
способов, как не нужно
делать лампу накаливания

Томас Алва Эдисон

2.1 Введение

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

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

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

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

Данная глава посвящена возможностям повышения производительности PostgreSQL. Глава не претендует на исчерпывающее изложение вопроса, наиболее полным и точным руководством по использованию PostgreSQL является, конечно, [официальная документация](#) и [официальный FAQ](#). Также существует англоязычный список рассылки `postgresql-performance`, посвящённый именно этим вопросам. Глава состоит из двух разделов, первый из которых ориентирован скорее на администратора,

второй — на разработчика приложений. Рекомендуется прочесть оба раздела: отнесение многих вопросов к какому-то одному из них весьма условно.

Не используйте настройки по умолчанию

По умолчанию PostgreSQL сконфигурирован таким образом, чтобы он мог быть запущен практически на любом компьютере и не слишком мешал при этом работе других приложений. Это особенно касается используемой памяти. Настройки по умолчанию подходят только для следующего использования: с ними вы сможете проверить, работает ли установка PostgreSQL, создать тестовую базу уровня записной книжки и потренироваться писать к ней запросы. Если вы собираетесь разрабатывать (а тем более запускать в работу) реальные приложения, то настройки придётся радикально изменить. В дистрибутиве PostgreSQL, к сожалению, не поставляются файлы с «рекомендуемыми» настройками. Вообще говоря, такие файлы создать весьма сложно, т.к. оптимальные настройки конкретной установки PostgreSQL будут определяться:

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

Используйте актуальную версию сервера

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

- В версии 7.4 была ускорена работа многих сложных запросов (включая печально известные подзапросы IN/NOT IN);
- В версии 8.0 были внедрены метки восстановления, улучшение управления буфером, CHECKPOINT и VACUUM улучшены;
- В версии 8.1 был улучшен одновременный доступ к разделяемой памяти, автоматическое использование индексов для MIN() и MAX(), pg_autovacuum внедрен в сервер (автоматизирован), повышение производительности для секционированных таблиц;
- В версии 8.2 была улучшена скорость множества SQL запросов, усовершенствован сам язык запросов;
- В версии 8.3 внедрен полнотекстовый поиск, поддержка SQL/XML стандарта, параметры конфигурации сервера могут быть установлены на основе отдельных функций;

2.1. Введение

- В версии 8.4 были внедрены общие табличные выражения, рекурсивные запросы, параллельное восстановление, улучшена производительность для EXISTS/NOT EXISTS запросов;
- В версии 9.0 «асинхронная репликация из коробки», VACUUM/VACUUM FULL стали быстрее, расширены хранимые процедуры;
- В версии 9.1 «синхронная репликация из коробки», нелогируемые таблицы (очень быстрые на запись, но при падении БД данные могут пропасть), новые типы индексов, наследование таблиц в запросах теперь может вернуться многозначительно отсортированные результаты, позволяющие оптимизации MIN/MAX;
- В версии 9.2 «каскадная репликация из коробки», сканирование по индексу, JSON тип данных, типы данных на диапазоны, сортировка в памяти улучшена на 25%, ускорена команда COPY;
- В версии 9.3 materialized view, доступные на запись внешние таблицы, переход с использования SysV shared memory на POSIX shared memory и mmap, сокращено время распространения реплик, а также значительно ускорена передача управления от запасного сервера к первичному, увеличена производительность и улучшена система блокировок для внешних ключей;
- В версии 9.4 появился новый тип поля JSONB (бинарный JSON с поддержкой индексов), логическое декодирование для репликации, GIN индекс в 2 раза меньше по размеру и в 3 раза быстрее, неблокирующие обновление materialized view, поддержка Linux Huge Pages;
- В версии 9.5 добавлена поддержка UPSERT, Row Level Security, CUBE, ROLLUP, GROUPING SETS функции, TABLESAMPLE, BRIN индекс, ускорена скорость работы индексов для текстовых и цифровых полей;
- В версии 9.6 добавлена поддержка параллелизации некоторых запросов, что позволяет использовать несколько ядер (CPU core) на сервере, чтобы возвращать результаты запроса быстрее, полнотекстовый поиск поддерживает фразы, новая опция «remote_apply» для синхронной репликации, которая позволяет дожидаться, пока запрос завершится на слейве;

Следует также отметить, что большая часть изложенного в статье материала относится к версии сервера не ниже 9.0.

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

2.2. Настройка сервера

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 (раздел «2.5 Оптимизация конкретных запросов»);

2.2 Настройка сервера

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

Используемая память

Общий буфер сервера: `shared_buffers`

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

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

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

2.2. Настройка сервера

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

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

На выделенных серверах полезным объемом для `shared_buffers` будет значение 1/4 памяти в системе. 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, то можно подымать это значение и мониторить результат, чтобы не привести к «деградации» (падению) производительности. Выделив слишком много памяти для базы данных, мы можем получить ухудшение производительности, поскольку PostgreSQL также использует кэш операционной системы (увеличение данного параметра более 40% оперативной памяти может давать «нулевой» прирост производительности).

Для тонкой настройки параметра установите для него большое значение и потестируйте базу при обычной нагрузке. Проверяйте использование разделяемой памяти при помощи `ipcs` или других утилит (например, `free` или `vmstat`). Рекомендуемое значение параметра будет примерно в 1,2–2 раза больше, чем максимум использованной памяти. Обратите внимание, что память под буфер выделяется при запуске сервера, и её объём при работе не изменяется. Учтите также, что настройки ядра операционной системы могут не дать вам выделить большой объём памяти (для версии PostgreSQL < 9.3). В [руководстве администратора PostgreSQL](#) описано, как можно изменить эти настройки.

Также следует помнить, что на 32 битной системе (Linux) каждый процесс лимитирован в 4 ГБ адресного пространства, где хотя бы 1 ГБ зарезервирован ядром. Это означает, что независимо, сколько на машине памяти, каждый PostgreSQL инстанс сможет обратиться максимум к 3 ГБ памяти. А значит максимум для `shared_buffers` в такой системе — 2–2.5 ГБ.

Хочу обратить внимание, что на Windows, большие значения для `shared_buffers` не столь эффективны, как на Linux системах, и в результате лучшие результаты можно будет получить, если держать это значение относительно небольшое (от 64 МБ до 512 МБ) и использовать кэш системы вместо него.

2.2. Настройка сервера

Память для сортировки результата запроса: `work_mem`

`work_mem` параметр определяет максимальное количество оперативной памяти, которое может выделить одна операция сортировки, агрегации и др. Это не разделяемая память, `work_mem` выделяется отдельно на каждую операцию (от одного до нескольких раз за один запрос). Разумное значение параметра определяется следующим образом: количество доступной оперативной памяти (после того, как из общего объема вычли память, требуемую для других приложений, и `shared_buffers`) делится на максимальное число одновременных запросов умноженное на среднее число операций в запросе, которые требуют памяти.

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

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 столбцов в каждом запросе и десятками миллионов строк 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 МБ.

Максимальное количество клиентов: `max_connections`

Параметр `max_connections` устанавливает максимальное количество клиентов, которые могут подключиться к PostgreSQL. Поскольку для каждого клиента требуется выделять память (`work_mem`), то этот параметр предполагает максимально возможное использование памяти для всех клиентов. Как правило, PostgreSQL может поддерживать несколько сотен подключений, но создание нового является дорогостоящей операцией. Поэтому, если требуются тысячи подключений, то лучше использовать пул подключений (отдельная программа или библиотека для продукта, что использует базу).

Память для работы команды VACUUM: `maintenance_work_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 дан-

2.2. Настройка сервера

ных. 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 МБ.

Большие страницы: `huge_pages`

В PostgreSQL, начиная с версии 9.4, появилась поддержка больших страниц. В ОС Linux работа с памятью основывается на обращении к страницам размер которых равен 4кВ (на самом деле зависит от платформы, проверить можно через `getconf PAGE_SIZE`). Так вот, когда объем памяти переваливает за несколько десятков, а то и сотни гигабайт, управлять ею становится сложнее, увеличиваются накладные расходы на адресацию памяти и поддержание страничных таблиц. Для облегчения жизни и были придуманы большие страницы, размер которых может быть 2МВ, а то и 1GB. За счет использования больших страниц можно получить ощутимый прирост скорости работы и увеличение отзывчивости в приложениях которые активно работают с памятью.

Вообще запустить PostgreSQL с поддержкой больших страниц можно было и раньше, с помощью `libhugetlbfs`. Однако теперь есть встроенная поддержка. Итак, ниже описание процесса как настроить и запустить PostgreSQL с поддержкой больших страниц.

Для начала следует убедиться, что ядро поддерживает большие страницы. Проверяем конфиг ядра на предмет наличия опций `CONFIG_HUGETLBFS` и `CONFIG_HUGETLB_PAGE`.

Листинг 2.1 Проверка конфига ядра на поддержку huge pages

```
Line 1 $ grep HUGETLB /boot/config -$(uname -r)
- CONFIG_CGROUP_HUGETLB=y
- CONFIG_HUGETLBFS=y
- CONFIG_HUGETLB_PAGE=y
```

В случае отсутствия этих опций, ничего не заработает и ядро следует пересобрать.

Очевидно, что нам понадобится PostgreSQL версии не ниже 9.4. За поддержку больших страниц отвечает параметр `huge_page`, который может принимать три значения: `off` — не использовать большие страницы, `on` — использовать большие страницы, `try` — попытаться использовать большие страницы и в случае недоступности откатиться на использование обычных страниц. Значение `try` используется по умолчанию и является безопасным вариантом. В случае `on`, PostgreSQL не запустится, если большие страницы не определены в системе (или их недостаточно).

После перезапуска базы с параметром `try` потребуется включить поддержку больших страниц в системе (по умолчанию они не задействованы).

2.2. Настройка сервера

Расчет страниц приблизительный и здесь следует опираться на то, сколько памяти вы готовы выделить под нужды СУБД. Отмечу, что значение измеряется в страницах размером 2Mb, если вы хотите выделить 16GB, то это будет 8000 страниц.

Официальная документация предлагает опираться на значение `VmPeak` из `status` файла, который размещен в `/proc/PID/` директории, соответствующей номеру процесса `postmaster`. `VmPeak` как следует из названия это пиковое значение использования виртуальной памяти. Этот вариант позволяет определить минимальную планку, от которой следует отталкиваться, но на мой взгляд такой способ определения тоже носит случайный характер.

Листинг 2.2 Включаем поддержку huge pages в системе

```
Line 1 $ head -1 /var/lib/pgsql/9.5/data/postmaster.pid
- 3076
- $ grep ^VmPeak /proc/3076/status
- VmPeak: 4742563 kB
5 $ echo $((4742563 / 2048 + 1))
- 2316
- $ echo 'vm.nr_hugepages = 2316' >> /etc/sysctl.d/30-
  postgresql.conf
- $ sysctl -p --system
```

В ОС Linux есть также система по менеджменту памяти под названием «Transparent HugePages», которая включена по умолчанию. Она может вызывать проблему при работе с huge pages для PostgreSQL, поэтому рекомендуется выключать этот механизм:

Листинг 2.3 Отключаем Transparent HugePages

```
Line 1 $ echo never > /sys/kernel/mm/transparent_hugepage/defrag
- $ echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

После этого перезапускаем PostgreSQL и смотрим использование больших страниц:

Листинг 2.4 Проверяем использование huge pages

```
Line 1 $ grep ^HugePages /proc/meminfo
- HugePages_Total: 2316
- HugePages_Free: 2301
- HugePages_Rsvd: 128
5 HugePages_Surp: 0
```

Прочие настройки

- `temp_buffers` — буфер под временные объекты, в основном для временных таблиц. Можно установить порядка 16 МБ;

2.2. Настройка сервера

- `max_prepared_transactions` — количество одновременно подготавливаемых транзакций (PREPARE TRANSACTION). Можно оставить по умолчанию — 5;
- `vacuum_cost_delay` — если у вас большие таблицы, и производится много одновременных операций записи, вам может пригодиться функция, которая уменьшает затраты на I/O для VACUUM, растягивая его по времени. Чтобы включить эту функциональность, нужно поднять значение `vacuum_cost_delay` выше 0. Используйте разумную задержку от 50 до 200 мс. Для более тонкой настройки повышайте `vacuum_cost_page_hit` и понижайте `vacuum_cost_limit`. Это ослабит влияние VACUUM, увеличив время его выполнения. В тестах с параллельными транзакциями Ян Вiek (Jan Wieck) получил, что при значениях `delay` — 200, `page_hit` — 6 и `limit` — 100 влияние VACUUM уменьшилось более чем на 80%, но его длительность увеличилась втрое;
- `max_stack_depth` — специальный стек для сервера, который в идеале должен совпадать с размером стека, выставленном в ядре ОС. Установка большего значения, чем в ядре, может привести к ошибкам. Рекомендуется устанавливать 2–4 МВ;
- `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

Для обеспечения отказоустойчивости СУБД PostgreSQL, как и многие базы данных, использует специальный журнал, в котором ведет историю изменения данных. Перед тем как записать данные в файлы БД, сервер PostgreSQL аккумулирует изменения в оперативной памяти и записывает в последовательный файл журнала, чтобы не потерять их из-за непредвиденного отключения питания.

Данные в журнал пишутся до того как пользователь базы данных получит сообщение об успешном применении изменений. Этот журнал называется журналом упреждающей записи (Write-Ahead Log или просто WAL), а файлы журнала хранятся в каталоге `pg_xlog`. Также периодически PostgreSQL сбрасывает измененные аккумулярованные данные из оперативной памяти на диск. Этот процесс согласования данных называется контрольной точкой (`checkpoint`). Контрольная точка выполняется также при каждом штатном выключении PostgreSQL.

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

2.2. Настройка сервера

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 («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). При этом производительность упадёт из-за постоянного сбрасывания на диск данных из буфера.

Для увеличения интервала между контрольными точками нужно увеличить количество сегментов журнала транзакций через параметр `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 объёма 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 $(\text{checkpoint_segments} * (2 + \text{checkpoint_completion_target}) + 1) * 16$ 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.

Начиная с версии 9.5 `checkpoint_segments` был заменен на параметры `min_wal_size` и `max_wal_size`. Теперь система может автоматически сама решать сколько `checkpoint_segments` требуется хранить (вычислять по ранее приведенной формуле от указанного размера). Преимуществом этого является то, что вы можете установить `max_wal_size` очень большим, но система не будет на самом деле потреблять указанное количество места на жестком диске, если в этом нет никакой необходимости. `min_wal_size` устанавливает минимальный размер места, который будет использоваться сегментами (можно отключить такую автонастройку, установив для `min_wal_size` и `max_wal_size` одинаковое значение).

2.2. Настройка сервера

`fsync`, `synchronous_commit` и стоит ли их трогать

Для увеличения производительности наиболее радикальное из возможных решений — выставить значение «off» параметру `fsync`. При этом записи в журнале транзакций не будут принудительно сбрасываться на диск, что даст большой прирост скорости записи. Учтите: вы жертвуете надёжностью, в случае сбоя целостность базы будет нарушена, и её придётся восстанавливать из резервной копии! Использовать этот параметр рекомендуется лишь в том случае, если вы всецело доверяете своему «железу» и своему источнику бесперебойного питания. Ну или если данные в базе не представляют для вас особой ценности.

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

Прочие настройки

- `commit_delay` (в микросекундах, 0 по умолчанию) и `commit_siblings` (5 по умолчанию) определяют задержку между попаданием записи в буфер журнала транзакций и сбросом её на диск. Если при успешном завершении транзакции активно не менее `commit_siblings` транзакций, то запись будет задержана на время `commit_delay`. Если за это время завершится другая транзакция, то их изменения будут сброшены на диск вместе, при помощи одного системного вызова. Эти параметры позволяют 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`;
 - `fdasync` — вызов метода `fdasync()` после каждого `commit`;

2.2. Настройка сервера

- `fsync_writethrough` — вызов `fsync()` после каждого `commit`, игнорируя параллельные процессы;
- `fsync` — вызов `fsync()` после каждого `commit`;
- `open_sync` — запись данных методом `open()` с параметром `O_SYNC`;

Не все эти методы доступны на разных ОС. По умолчанию устанавливается первый, который доступен для системы;

- `full_page_writes` Установите данный параметр в «off», если `fsync=off`. Иначе, когда этот параметр «on», PostgreSQL записывает содержимое каждой записи в журнал транзакций при первой модификации таблицы. Это необходимо, поскольку данные могут записаться лишь частично, если в ходе процесса «упала» ОС. Это приведет к тому, что на диске окажутся новые данные смешанные со старыми. Строкового уровня записи в журнал транзакций может быть недостаточно, чтобы полностью восстановить данные после «падения». `full_page_writes` гарантирует корректное восстановление, ценой увеличения записываемых данных в журнал транзакций (Единственный способ снижения объема записи в журнал транзакций заключается в увеличении `checkpoint_interval`);
- `wal_buffers` Количество памяти, используемое в Shared Memory для ведения транзакционных логов (буфер находится в разделяемой памяти и является общим для всех процессов). 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 3 настройки планировщика, на которые стоит обратить внимание:

- `default_statistics_target` — этот параметр задаёт объём статистики, собираемой командой `ANALYZE`. Увеличение параметра заставит эту команду работать дольше, но может позволить оптимизатору строить более быстрые планы, используя полученные дополнительные данные. Объём статистики для конкретного поля может быть задан командой `ALTER TABLE ... SET STATISTICS`;
- `effective_cache_size` — этот параметр сообщает PostgreSQL примерный объём файлового кэша операционной системы, оптимизатор использует эту оценку для построения плана запроса (указывает планировщику на размер самого большого объекта в базе данных, который теоретически может быть закеширован). Пусть в вашем компьютере

2.2. Настройка сервера

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 выставить `effective_cache_size` в 2/3 от всей оперативной памяти; на сервере с другими приложениями сначала нужно вычесть из всего объема RAM размер дискового кэша ОС и память, занятую остальными процессами;

- `random_page_cost` — переменная, указывающая на условную стоимость индексного доступа к страницам данных. На серверах с быстрыми дисковыми массивами имеет смысл уменьшать изначальную настройку до 3.0, 2.5 или даже до 2.0. Если же активная часть вашей базы данных намного больше размеров оперативной памяти, попробуйте поднять значение параметра. Можно подойти к выбору оптимального значения и со стороны производительности запросов. Если планировщик запросов чаще, чем необходимо, предпочитает последовательные просмотры (`sequential scans`) просмотрам с использованием индекса (`index scans`), понижайте значение. И наоборот, если планировщик выбирает просмотр по медленному индексу, когда не должен этого делать, настройку имеет смысл увеличить. После изменения тщательно тестируйте результаты на максимально широком наборе запросов. Никогда не опускайте значение `random_page_cost` ниже 2.0; если вам кажется, что `random_page_cost` нужно еще понижать, разумнее в этом случае менять настройки статистики планировщика.

Сбор статистики

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

- `track_counts` — включать ли сбор статистики. По умолчанию включён, поскольку `autovacuum` демону требуется сбор статистики. Отключайте, только если статистика вас совершенно не интересует (как и `autovacuum`);
- `track_functions` — отслеживание использования определенных пользователем функций;

2.3. Диски и файловые системы

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

Данные, полученные сборщиком статистики, доступны через специальные системные представления. При установках по умолчанию собирается очень мало информации, рекомендуется включить все возможности: дополнительная нагрузка будет невелика, в то время как полученные данные позволят оптимизировать использование индексов (а также помогут оптимальной работе `autovacuum` демону).

2.3 Диски и файловые системы

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

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

Вы легко можете получить выигрыш в производительности без побочных эффектов, если примонтируете файловую систему, содержащую базу данных, с параметром `noatime` (но при этом не будет отслеживаться время последнего доступа к файлу).

Перенос журнала транзакций на отдельный диск

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

Если в вашем сервере есть несколько физических дисков (несколько логических разделов на одном диске здесь, очевидно, не помогут: головка всё равно будет одна), то вы можете разнести файлы базы данных и журнал транзакций по разным дискам. Данные в сегменты журнала пишутся последовательно, более того, записи в журнале транзакций сразу сбрасываются на диск, поэтому в случае нахождения его на отдельном диске магнитная головка не будет лишней раз двигаться, что позволит ускорить запись.

Порядок действий:

2.4. Утилиты для тюнинга PostgreSQL

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

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

CLUSTER

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

Кластеризация по индексу позволяет сократить время поиска по диску: во время поиска по индексу выборка данных может быть значительно быстрее, так как последовательность данных в таком же порядке, как и индекс. Из минусов можно отметить то, что команда `CLUSTER` требует «ACCESS EXCLUSIVE» блокировку, что предотвращает любые другие операции с данными (чтения и записи) пока кластеризация не завершит выполнение. Также кластеризация индекса в PostgreSQL не утверждает четкий порядок следования, поэтому требуется повторно выполнять `CLUSTER` для поддержания таблицы в порядке.

2.4 Утилиты для тюнинга PostgreSQL

Pgtune

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

Листинг 2.5 Pgtune

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

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

Есть также дополнительные опции для настройки конфига:

2.4. Утилиты для тюнинга PostgreSQL

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

Существует также [онлайн версия pgtune](#).

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

`pg_buffercache`

`pg_buffercache` — расширение для PostgreSQL, которое позволяет получить представление об использовании общего буфера (`shared_buffer`) в базе. Расширение позволяет взглянуть какие из данных кэширует база, которые активно используются в запросах. Для начала нужно установить расширение:

Листинг 2.6 `pg_buffercache`

```
Line 1 # CREATE EXTENSION pg_buffercache;
```

Теперь доступно `pg_buffercache` представление, которое содержит:

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

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

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

2.4. Утилиты для тюнинга PostgreSQL

- Размер общего буфера. Определяется опцией `shared_buffers` в PostgreSQL конфиге.

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

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

Листинг 2.7 pg_bufferscache

```
Line 1 # SELECT c.relname, count(*) AS buffers
- FROM pg_bufferscache b INNER JOIN pg_class c
- ON b.relfilenode = pg_relation_filenode(c.oid) AND
- b.reldatabase IN (0, (SELECT oid FROM pg_database WHERE
  datname = current_database()))
5 GROUP BY c.relname
- ORDER BY 2 DESC
- LIMIT 10;
-
-          relname          | buffers
10 -----+-----
-  pgbench_accounts        |    4082
-  pgbench_history         |      53
-  pg_attribute            |      23
-  pg_proc                 |      14
15  pg_operator            |      11
-  pg_proc_oid_index       |       9
-  pg_class                |       8
-  pg_attribute_relid_attnum_index |       7
-  pg_proc_proname_args_nsp_index |       6
20  pg_class_oid_index     |       5
- (10 rows)
```

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

Листинг 2.8 pg_bufferscache

```
Line 1 # SELECT c.relname, count(*) AS buffers, usagecount
- FROM pg_class c
- INNER JOIN pg_bufferscache b
- ON b.relfilenode = c.relfilenode
5 INNER JOIN pg_database d
- ON (b.reldatabase = d.oid AND d.datname = current_database
  ())
- GROUP BY c.relname, usagecount
- ORDER BY c.relname, usagecount;
-
```


2.4. Утилиты для тюнинга PostgreSQL

	relname	buffers	usagecount
-	pg_rewrite	3	1
-	pg_rewrite_rel_rulename_index	1	1
-	pg_rewrite_rel_rulename_index	1	2
15	pg_statistic	1	1
-	pg_statistic	1	3
-	pg_statistic	2	5
-	pg_statistic_relid_att_inh_index	1	1
-	pg_statistic_relid_att_inh_index	3	5
20	pgbench_accounts	4082	2
-	pgbench_accounts_pkey	1	1
-	pgbench_history	53	1
-	pgbench_tellers	1	1

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

Листинг 2.9 pg_buffercache

```

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

```

2.5. Оптимизация БД и приложения

```
- percent_of_relation | 94.6
- -[ RECORD 3 ] -----+-----
- relname              | pg_operator
30 buffered            | 88 kB
- buffers_percent      | 0.1
- percent_of_relation | 61.1
- -[ RECORD 4 ] -----+-----
- relname              | pg_opclass_oid_index
35 buffered            | 16 kB
- buffers_percent      | 0.0
- percent_of_relation | 100.0
- -[ RECORD 5 ] -----+-----
- relname              | pg_statistic_relid_att_inh_index
40 buffered            | 32 kB
- buffers_percent      | 0.0
- percent_of_relation | 100.0
```

Используя эти данные можно проанализировать для каких объектов не хватает памяти или какие из них потребляют основную часть общего буфера. На основе этого можно более правильно настраивать `shared_buffers` параметр для PostgreSQL.

2.5 Оптимизация БД и приложения

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

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

Поддержание базы в порядке

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

Команда ANALYZE

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

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

Команда REINDEX

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

- порчи индекса;
- постоянного увеличения его размера;

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

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

Использование индексов

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

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

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

2.5. Оптимизация БД и приложения

Команда EXPLAIN [ANALYZE]

Команда `EXPLAIN запрос[]` показывает, каким образом PostgreSQL собирается выполнять ваш запрос. Команда `EXPLAIN ANALYZE запрос[]` выполняет запрос (и поэтому `EXPLAIN ANALYZE DELETE ...` — не слишком хорошая идея) и показывает как изначальный план, так и реальный процесс его выполнения.

Чтение вывода этих команд — искусство, которое приходит с опытом. Для начала обращайтесь 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, уместающихся 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,

Листинг 2.10 `enable_seqscan`

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

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

Использование собранной статистики

Результаты работы сборщика статистики доступны через специальные системные представления. Наиболее интересны для наших целей следующие:

- `pg_stat_user_tables` содержит — для каждой пользовательской таблицы в текущей базе данных — общее количество полных просмотров

2.5. Оптимизация БД и приложения

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.

Перенос логики на сторону сервера

Этот пункт очевиден для опытных пользователей PostgreSQL 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, правила¹ 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;

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

2.5. Оптимизация БД и приложения

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

Оптимизация конкретных запросов

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

```
SELECT count(*) FROM <огромная таблица>
```

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

Проблема Запрос вида

Листинг 2.11 SQL

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

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

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

1. Если точное число записей не важно, а важен порядок¹, то можно использовать информацию о количестве записей в таблице, собранную при выполнении команды `ANALYZE`:

Листинг 2.12 SQL

```
Line 1 SELECT reltuples FROM pg_class WHERE relname = 'foo';
```

2. Если подобные выборки выполняются часто, а изменения в таблице достаточно редки, то можно завести вспомогательную таблицу, хранящую число записей в основной. На основную же таблицу повесить триггер, который будет уменьшать это число в случае удаления

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

2.5. Оптимизация БД и приложения

записи и увеличивать в случае вставки. Таким образом, для получения количества записей потребуется лишь выбрать одну запись из вспомогательной таблицы;

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

Медленный DISTINCT

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

Листинг 2.13 DISTINCT

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

Листинг 2.14 GROUP BY

```
Line 1 postgres=# select count(*) from (select i from g group by i)
      a;
-   count
-   -----
-   19125
5 (1 row)
-
- Time: 26,562 ms
-
-
10 postgres=# select count(*) from (select i from g group by i)
      a;
-   count
```

2.5. Оптимизация БД и приложения

```
- -----  
- 19125  
- (1 row)  
15  
- Time: 25,270 ms
```

Утилиты для оптимизации запросов

pgFouine

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

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

- Чтобы включить протоколирование в syslog

Листинг 2.15 pgFouine

```
Line 1 log_destination = 'syslog'  
- redirect_stderr = off  
- silent_mode = on  
-
```

- Для записи запросов, длящихся дольше n миллисекунд:

Листинг 2.16 pgFouine

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

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

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

Листинг 2.17 pgFouine

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


2.5. Оптимизация БД и приложения

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

Листинг 2.18 pgFouine

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

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

pgBadger

pgBadger — аналогичная утилита, что и **pgFouine**, но написанная на Perl. Еще одно большое преимущество проекта в том, что он более активно сейчас разрабатывается (на момент написания этого текста последний релиз **pgFouine** был в 24.02.2010, а последняя версия **pgBadger** — 24.01.2017). Установка **pgBadger** проста:

Листинг 2.19 Установка pgBadger

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

Как и в случае с **pgFouine** нужно настроить PostgreSQL логи:

Листинг 2.20 Настройка логов PostgreSQL

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

Парсим логи PostgreSQL через **pgBadger**:

Листинг 2.21 Запуск pgBadger

```
Line 1 $ ./pgbadger ~/pgsql/master/pg_log/postgresql-2012-08-30_132
*
- [=====>] Parsed 10485768 bytes of
  10485768 (100.00%)
- [=====>] Parsed 10485828 bytes of
  10485828 (100.00%)
- [=====>] Parsed 10485851 bytes of
  10485851 (100.00%)
```

2.5. Оптимизация БД и приложения

```
5 |=====|>| Parsed 10485848 bytes of
   | 10485848 (100.00%)
- |=====|>| Parsed 10485839 bytes of
   | 10485839 (100.00%)
- |=====|>| Parsed 982536 bytes of 982536
   | (100.00%)
```

В результате получится HTML файлы, которые содержат статистику по запросам к PostgreSQL. Более подробно о возможностях можно найти на официальном сайте проекта <http://dalibo.github.io/pgbadger/>.

pg_stat_statements

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

Листинг 2.22 Настройка pg_stat_statements в postgresql.conf

```
Line 1 shared_preload_libraries = 'pg_stat_statements'
- custom_variable_classes = 'pg_stat_statements' # данная
  настройка нужна для PostgreSQL 9.1 и ниже
-
- pg_stat_statements.max = 10000
5 pg_stat_statements.track = all
```

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

1. `pg_stat_statements.max (integer)`» — максимальное количество sql запросов, которое будет храниться расширением (удаляются записи с наименьшим количеством вызовов);
2. `pg_stat_statements.track (enum)`» — какие SQL запросы требуется записывать. Возможные параметры: top (только запросы от приложения/клиента), all (все запросы, например в функциях) и none (отключить сбор статистики);
3. `pg_stat_statements.save (boolean)`» — следует ли сохранять собранную статистику после остановки PostgreSQL. По умолчанию включено;

Далее активируем расширение:

Листинг 2.23 Активация pg_stat_statements

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

Пример собранной статистики:

Листинг 2.24 pg_stat_statements статистика

2.5. Оптимизация БД и приложения

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

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

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

```
-----
- query      | select * from x where i = ?;
20 calls     | 2
- total_time | 0.157
- rows       | 6
- hit_percent | 100.0000000000000000
- -[ RECORD 4 ]--
```

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

Для сброса статистики есть команда `pg_stat_statements_reset`:

Листинг 2.25 Сброс статистики

```
Line 1 # SELECT pg_stat_statements_reset();
```

2.6. Заключение

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

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

2.6 Заключение

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

Индексы

«Ну у вас и запросики»
сказала база данных и зависла
интернет

Что такое таблица в реляционной СУБД? Это такой список из кортежей (tuple). Каждый кортеж состоит из ячеек (row). Количество ячеек в кортеже и их тип совпадают со схемой колонки, нескольких колонок. Этот список имеет сквозную нумерацию RowId — порядковый номер. Таким образом, таблицы можно осознавать как список пар (RowId, Кортеж).

Индексы — это обратные отношения (Кортеж, RowId). Кортеж обязан содержать хотя бы одну ячейку (т. е. быть построенным минимум по одной колонке). Для индексов, которые индексируют более одной колонки — они ещё называются составными, и участвуют в отношениях вида «многие-ко-многим» — всё написанное верно в равной степени. Очевидно, если кортеж — не уникален (в колонке существует два одинаковых кортежа), то эти отношения выглядят как (Кортеж, Список RowId) — т. е. кортежу сопоставляется список RowId.

Индексы могут использоваться для таких операций в базе данных:

- Поиск данных — абсолютно все индексы поддерживают поиск значений по равенству. А B-Tree — по произвольным диапазонам;
- Like — B-Tree и Bitmap индексы можно использовать для ускорения префиксных Like-предикатов (вида abc%);
- Оптимизатор — B-Tree и R-Tree индексы представляют из себя гистограмму произвольной точности;
- Join — индексы могут быть использованы для Merge, Index алгоритмов;
- Relation — индексы могут быть использованы для операций except/intersect;
- Aggregations — индексы позволяют эффективно вычислять некоторые агрегатные функции — COUNT, MIN, MAX, а также их DISTINCT версии;

3.1. Типы индексов

- Grouping — индексы позволяют эффективно вычислять группировки и произвольные агрегатные функции (sort-group алгоритм);

3.1 Типы индексов

В зависимости от структуры, используемой в реализации индексов, существенно различаются поддерживаемые операции, их стоимости, а также свойства читаемых данных. Давайте рассмотрим какие существуют типы индексов в PostgreSQL.

B-Tree

B-Tree (Boeing/Bayer/Balanced/Broad/Bushy-Tree) называют упорядоченное блочное дерево. Узлы в дереве представляют из себя блоки фиксированного размера. У каждого узла фиксированное число детей. Структура B-Tree представлена на рисунке 3.1.

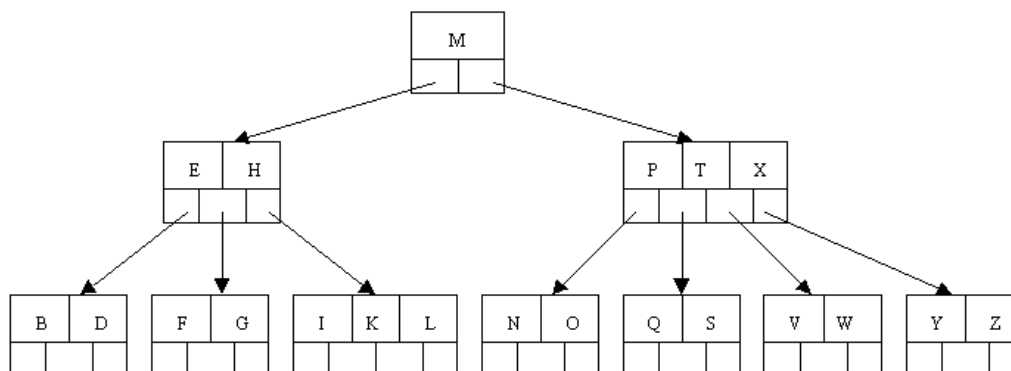


Рис. 3.1: B-Tree индекс

B-Tree для индексов отличается от представленной на Википедии — есть дублированные данные в промежуточных блоках. Для i -ой записи в блоке сохраняется не значение, которое больше максимума i -го поддерева, и меньше минимума $(i+1)$ поддерева, а максимум i -го поддерева. Различия проистекают из того, что википедия приводит пример B-Tree для множества, а нам нужен ассоциативный массив.

В индексном B-Tree значения и RowId размещаются совместно на нижнем слое дерева. Каждый узел дерева представляет из себя одну страницу (page) в некотором формате. В начале страницы всегда идёт некоторый заголовок. Для корневого и промежуточного узла в страницах хранятся пары (Значение, Номер страницы). Для листовых — пары (Значение, RowId) либо (Значение, Список RowId) (в зависимости от свойств значения — уникально или нет). B-Tree деревья имеют крайне маленькую высоту — по-

3.1. Типы индексов

рядка $H = \log_m N$, где m — количество записей в блоке, N — количество элементов. В-Tree деревья являются упорядоченными — все элементы в любой странице (блоке) дерева лежат последовательно. Предыдущие два свойства позволяют крайне эффективно производить поиск — начиная с первой страницы, половинным делением (binary search) выделяются дети, в которых лежат границы поиска. Таким образом, прочитав всего H , $2H$ страниц мы находим искомый диапазон. Важным нюансом является также факт, что страницы в листьях связаны в односвязный либо двусвязный список — это означает, что, выполнив поиск, мы можем дальше просто последовательно читать страницы, и эффективность чтения большего объёма данных (длинного диапазона) сравнима с эффективностью чтения данных из таблицы.

Сильные стороны В-Tree индексов:

- сохраняют сортированность данных;
- поддерживают поиск по унарным и бинарным предикатам ($<a; = b ; >c \text{ and } <d; <e \text{ and } >f$) за $O(\log_m N)$, где m — количество записей в блоке, N — количество элементов;
- позволяют не сканируя последовательность данных целиком оценить cardinality (количество записей) для всего индекса (а следовательно таблицы), диапазона, причём с произвольной точностью. Посмотрели корневую страницу — получили одну точность. Посмотрели следующий уровень дерева — получили точность получше. Просмотрели дерево до корня — получили точное число записей;
- самобалансируемый, для внесения изменения не требуется полного перестроения, происходит не более $O(\log_m N)$ действий, где m — количество записей в блоке, N — количество элементов;

Слабые стороны В-Tree индексов:

- занимают много места на диске. Индекс по уникальным Integer-ам, к примеру, весит в два раза больше аналогичной колонки (т.к. хранятся ещё и RowId);
- при постоянной записи дерево начинает хранить данные разреженно (сразу после построения они могут лежать очень плотно), и время доступа увеличивается за счёт увеличения объёма дисковой информации. Поэтому В-Tree индексы требуют присмотра и периодического перепостроения (REBUILD);

R-Tree

R-Tree (Rectangle-Tree) предназначен для хранения пар (X, Y) значений числового типа (например, координат). По способу организации R-Tree очень похоже на В-Tree. Единственное отличие — это информация, записываемая в промежуточные страницы в дереве. Для i -го значения в

3.1. Типы индексов

узле в B-Tree мы пишем максимум из i -го поддерева, а в R-Tree — минимальный прямоугольник, покрывающий все прямоугольники из ребёнка. Подробнее можно увидеть на рисунке 3.2.

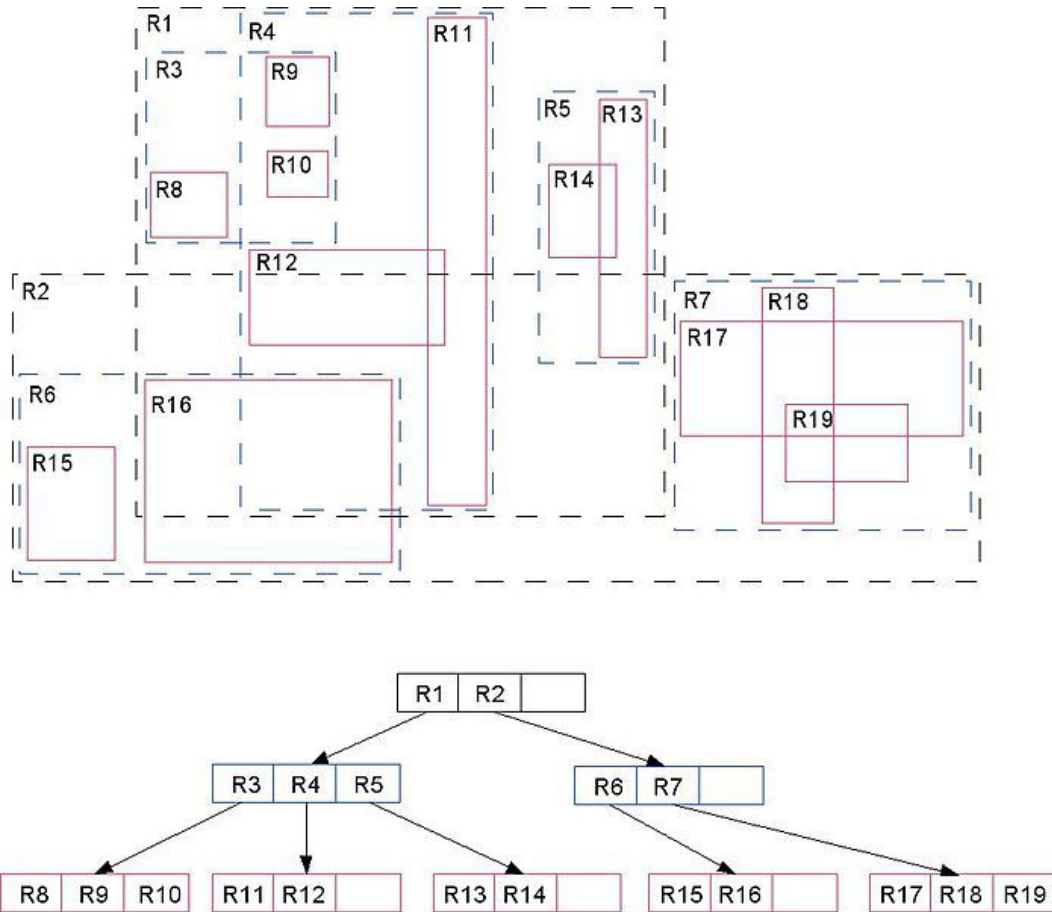


Рис. 3.2: R-Tree индекс

Сильные стороны:

- поиск произвольных регионов, точек за $O(\log_m N)$, где m — количество записей в блоке, N — количество элементов;
- позволяет оценить количество точек в некотором регионе без полного сканирования данных;

Слабые стороны:

- существенная избыточность в хранении данных;
- медленное обновление данных;

В целом, плюсы-минусы очень напоминают B-Tree.

Hash индекс

Hash индекс по сути является ассоциативным хеш-контейнером. Хеш-контейнер — это массив из разряженных значений. Адресуются отдельные элементы этого массива некоторой хеш-функцией которая отображает каждое значение в некоторое целое число. Т. е. результат хеш-функции является порядковым номером элемента в массиве. Элементы массива в хеш-контейнере называются бакетами (bucket). Обычно один бакет — одна страница. Хеш-функция отображает более мощное множество в менее мощное, возникают так называемые коллизии — ситуация, когда одному значению хеш-функции соответствует несколько разных значений. В бакете хранятся значения, образующие коллизию. Разрешение коллизий происходит посредством поиска среди значений, сохранённых в бакете.

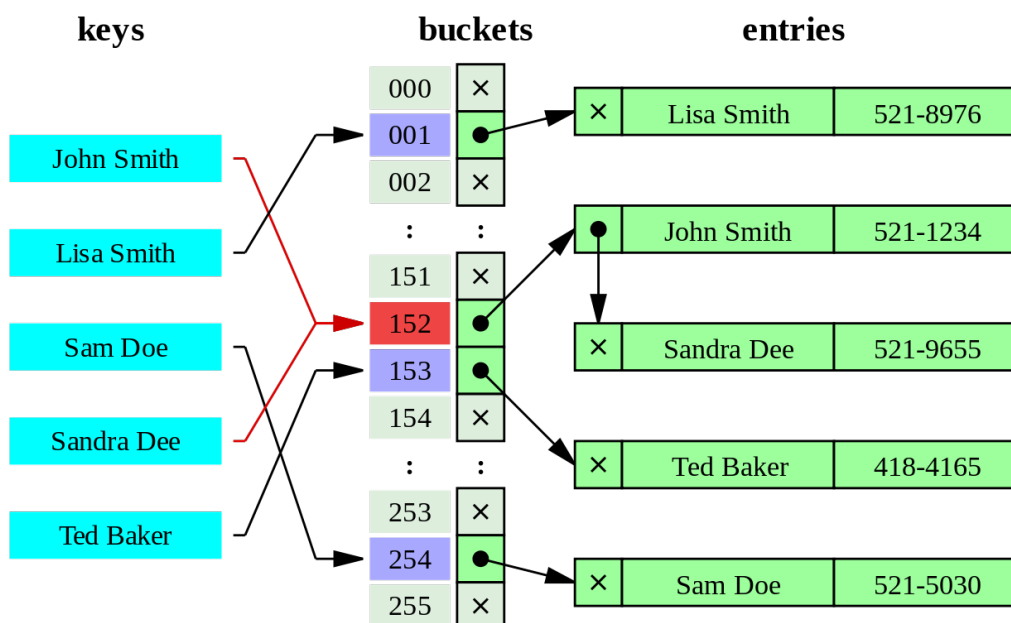


Рис. 3.3: Hash индекс

Сильные стороны:

- очень быстрый поиск $O(1)$;
- стабильность — индекс не нужно перестраивать;

Слабые стороны:

- хеш очень чувствителен к коллизиям хеш-функции. В случае «плохого» распределения данных, большинство записей будет сосредоточено в нескольких бакетах, и фактически поиск будет происходить путем разрешения коллизий;

3.1. Типы индексов

- из-за нелинейности хэш-функций данный индекс нельзя сортировать по значению, что приводит к невозможности использования в сравнениях больше/меньше и «IS NULL»;
- данный индекс в PostgreSQL транзакционно небезопасен, нужно перестраивать после краха и не реплицируется через потоковую (streaming) репликацию (разработчики обещают это исправить к 10 версии);

Битовый индекс (bitmap index)

Битовый индекс (bitmap index) — метод битовых индексов заключается в создании отдельных битовых карт (последовательность 0 и 1) для каждого возможного значения столбца, где каждому биту соответствует строка с индексируемым значением, а его значение равное 1 означает, что запись, соответствующая позиции бита содержит индексируемое значение для данного столбца или свойства ([алгоритм Хаффмана](#)).

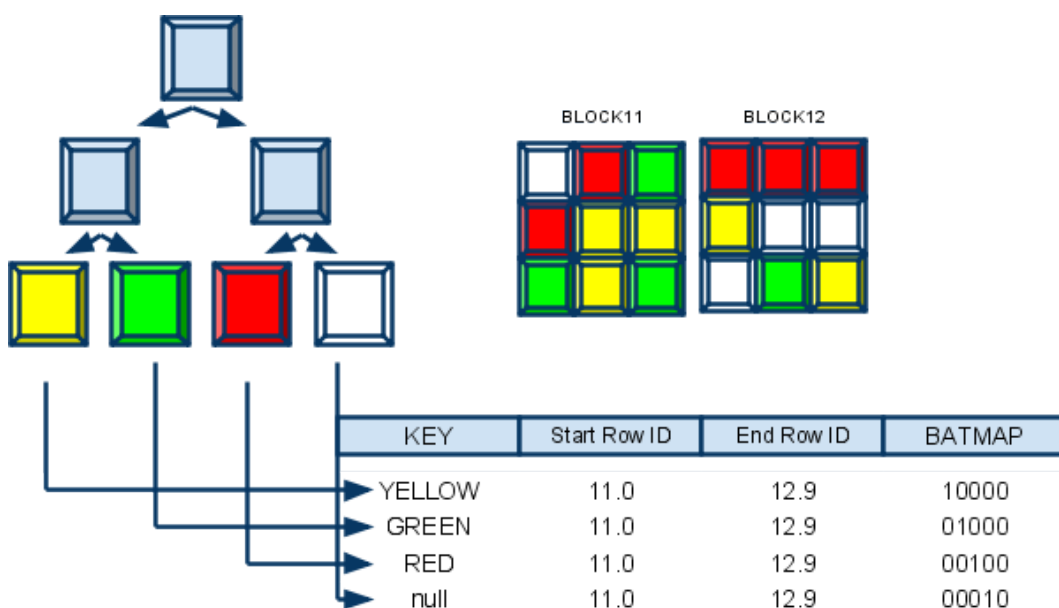


Рис. 3.4: Битовый индекс

Сильные стороны:

- компактность представления (занимает мало места);
- быстрое чтение и поиск по предикату «равно»;

Слабые стороны:

- невозможность изменить способ кодирования значений в процессе обновления данных;

3.1. Типы индексов

У PostgreSQL нет возможности создать постоянный битовый индекс, но база может на лету создавать данные индексы для объединения разных индексов. Чтобы объединить несколько индексов, база сканирует каждый необходимый индекс и готовит битовую карту в памяти с расположением строк таблицы. Битовые карты затем обрабатываются AND/OR операцией по мере требования запроса и после этого выбираются колонки с данными.

GiST индекс

GiST (Generalized Search Tree) — обобщение B-Tree, R-Tree дерево поиска по произвольному предикату. Структура дерева не меняется, по-прежнему в каждом нелистовом узле хранятся пары (Значения, Номер страницы), а количество детей совпадает с количеством пар в узле. Существенное отличие состоит в организации ключа. B-Tree деревья заточены под поиск диапазонов и хранят максимумы поддеревя-ребёнка. R-Tree — региона на координатной плоскости. GiST предлагает в качестве значений в нелистовых узлах хранить ту информацию, которую мы считаем существенной, и которая позволит определить, есть ли интересующие нас значения (удовлетворяющие предикату) в поддереве-ребёнке. Конкретный вид хранимой информации зависит от вида поиска, который мы желаем проводить. Таким образом параметризовав R-Tree и B-Tree дерево предикатами и значениями мы автоматически получаем специализированный под задачу индекс (PostGiST, pg_trgm, hstore, ltree, прочее).

Сильные стороны:

- эффективный поиск;

Слабые стороны:

- большая избыточность;
- необходимость специализированной реализации под каждую группу запросов;

Остальные плюсы-минусы совпадают с B-Tree и R-Tree индексами.

GIN индекс

GIN (Generalized Inverted Index) — обратный индекс, используемым полнотекстовым поиском PostgreSQL. Это означает, что в структуре индексов с каждой лексемой сопоставляется отсортированный список номеров документов, в которых она встречается. Очевидно, что поиск по такой структуре намного эффективнее, чем при использовании GiST, однако процесс добавления нового документа достаточно длителен.

Cluster индекс

Не является индексом, поскольку производит кластеризацию таблицы по заданному индексу. Более подробно можно почитать в разделе «2.3 CLUSTER».

BRIN индекс

Версия PostgreSQL 9.5 привнесла с собой новый вид индексов — BRIN (Block Range Index, или индекс блоковых зон).

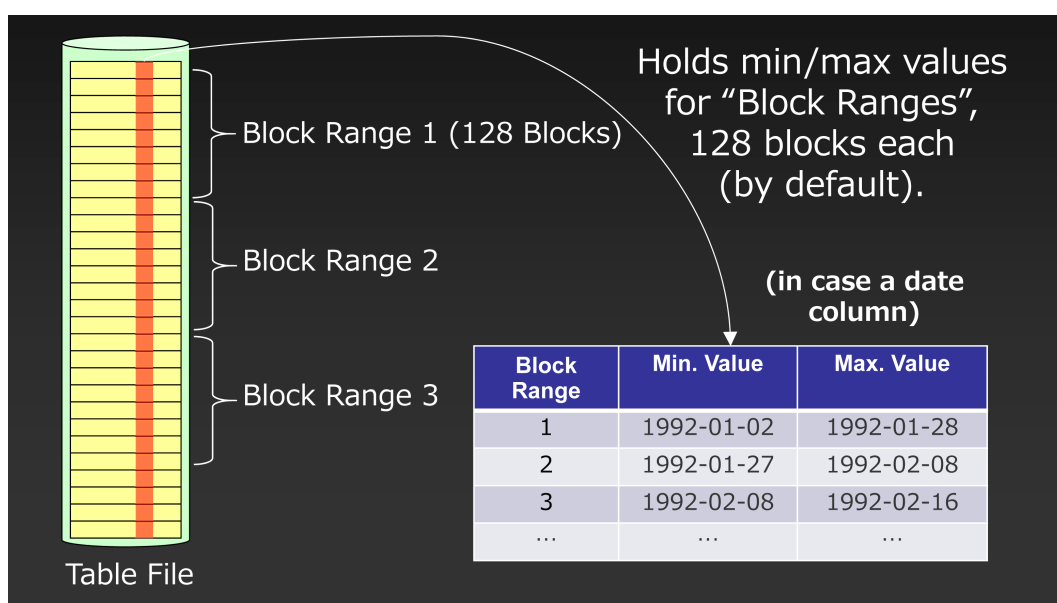


Рис. 3.5: BRIN индекс

В отличие от привычного B-Tree, этот индекс намного эффективнее для очень больших таблиц, и в некоторых ситуациях позволяет заменить собой партиционирование (подробно можно почитать в разделе «4 Партиционирование»). BRIN-индекс имеет смысл применять для таблиц, в которых часть данных уже по своей природе как-то отсортирована. Например, это характерно для логов или для истории заказов магазина, которые пишутся последовательно, а потому уже на физическом уровне упорядочены по дате/номеру, и в то же время таблицы с такими данными обычно разрастаются до гигантских размеров.

Под блоковой зоной (Block Range) подразумевается набор страниц, физически расположенных по соседству в таблице. Для каждой такой зоны создается некий идентификатор, отвечающий за «место» этой зоны в таблице. Для лога это может быть дата создания записи. Поиск по такому индексу осуществляется с потерями информации, то есть выбираются все записи, входящие в блоковые зоны с идентификаторами, соответствующими запросу, но среди записей в этих зонах могут попадаться такие, которые

3.2. Возможности индексов

на следующем этапе надо будет отфильтровать. Размер индекса при этом очень маленький, и он почти не нагружает базу. Размер индекса обратно пропорционален параметру `pages_per_range`, отвечающему за количество страниц на зону. В то же время, чем меньше размер зоны, тем меньше «лишних» данных попадёт в результат поиска (надо подходить к этому параметру с умом).

Индексы BRIN могут иметь один из нескольких встроенных классов операторов, по которым будет осуществляться разбивка на зоны и присвоение идентификаторов. Например, `int8_minmax_ops` применяется для операций сравнения целых чисел, а `date_minmax_ops` для сравнения дат.

3.2 Возможности индексов

Функциональный индекс (functional index)

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 из поля `foo_name` в любом регистре». Вы можете создать индекс

Листинг 3.1 Индекс

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

и запрос вида

Листинг 3.2 Запрос

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

будет его использовать.

Частичный индекс (partial index)

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

Листинг 3.3 Индекс

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

который будет использоваться запросом вида

3.2. Возможности индексов

Листинг 3.4 Запрос

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

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

Уникальный индекс (unique index)

Уникальный индекс гарантирует, что таблица не будет иметь более чем одну строку с тем же значением. Это удобно по двум причинам: целостность данных и производительность. Поиск данных с использованием уникального индекса, как правило, очень быстрый.

Индекс нескольких столбцов (multi-column index)

В PostgreSQL возможно создавать индексы на несколько столбцов, но нам главное нужно понять когда имеет смысл создавать такой индекс, поскольку планировщик запросов PostgreSQL может комбинировать и использовать несколько индексов в запросе путем создания битового индекса («3.1 Битовый индекс (bitmap index)»). Можно, конечно, создать индексы, которые охватят все возможные запросы, но за это придется платить производительностью (индексы нужно перестраивать при запросах на модификацию данных). Нужно также помнить, что индексы на несколько столбцов могут использоваться только запросами, которые ссылаются на эти столбцы в индексе в том же порядке. Индекс по столбцам (a, b) может быть использован в запросах, которые содержат `a = x and b = y` или `a = x`, но не будет использоваться в запросе вида `b = y`. Если это подходит под запросы вашего приложения, то данный индекс может быть полезен. В таком случае создание индекса на поле a было бы излишним. Индекс нескольких столбцов с указанием уникальности (unique) может быть также полезен для сохранения целостности данных (т. е. когда набор данных в этих столбцах должен быть уникальным).

Партиционирование

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

Народная мудрость

4.1 Введение

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

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

Итак, партиционирование дает ряд преимуществ:

4.2. Теория

- На определенные виды запросов (которые, в свою очередь, создают основную нагрузку на СУБД) мы можем улучшить производительность;
- Массовое удаление может быть произведено путем удаления одной или нескольких партиций (**DROP TABLE** гораздо быстрее, чем массовый **DELETE**);
- Редко используемые данные могут быть перенесены в другое хранилище;

4.2 Теория

На текущий момент PostgreSQL поддерживает два критерия для создания партиций:

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

Чтобы настроить партиционирование таблицы, достаточно выполнить следующие действия:

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

Листинг 4.1 Пример неверного задания значений партиций

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

неверно заданы партиции, поскольку непонятно какой партиции принадлежит значение 200;

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

4.3 Практика использования

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

Настройка

Для примера, мы имеем следующую таблицу:

Листинг 4.2 «Мастер» таблица

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

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

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

Листинг 4.3 «Дочерние» таблицы

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

4.3. Практика использования

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

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

Листинг 4.4 Создание индексов

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

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

Листинг 4.5 Функция для перенаправления

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

4.3. Практика использования

- `LANGUAGE plpgsql;`

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

Листинг 4.6 Триггер

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

Партиционирование настроено и теперь мы готовы приступить к тестированию.

Тестирование

Для начала добавим данные в нашу таблицу `my_logs`:

Листинг 4.7 Данные

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

Теперь проверим где они хранятся:

Листинг 4.8 «Мастер» таблица чиста

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

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

Листинг 4.9 Проверка данных

```
Line 1 partitioning_test=# SELECT * FROM my_logs;
- id | user_id | logdate | data | some_state
- --
- --+-----+-----+-----+-----
- 1 | 1 | 2010-10-30 00:00:00 | 30.10.2010 data |
- 1
- 5 | 2 | 2010-11-10 00:00:00 | 10.11.2010 data2 |
- 1
```

4.3. Практика использования

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

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

Листинг 4.10 Проверка хранения данных

```
Line 1 partitioning_test=# Select * from my_logs2010m10;
- id | user_id |          logdate          |          data          |
-   some_state
- --
-   --+-----+-----+-----+-----+-----
- 1 |          1 | 2010-10-30 00:00:00 | 30.10.2010 data |
-                                     1
5 (1 row)
-
- partitioning_test=# Select * from my_logs2010m11;
- id | user_id |          logdate          |          data          |
-   some_state
- --
-   --+-----+-----+-----+-----+-----
10 2 |          2 | 2010-11-10 00:00:00 | 10.11.2010 data2 |
-                                     1
- (1 row)
```

Данные хранятся на требуемых нам партициях. При этом запросы к таблице `my_logs` менять не требуется:

Листинг 4.11 Проверка запросов

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

4.3. Практика использования

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

Управление партициями

Обычно при работе с партиционированием старые партиции перестают получать данные и остаются неизменными. Это дает огромное преимущество над работой с данными через партиции. Например, нам нужно удалить старые логи за 2014 год, 10 месяц. Нам достаточно выполнить:

Листинг 4.12 Чистка логов

```
Line 1 DROP TABLE my_logs2014m10;
```

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

Листинг 4.13 Удаляем партицию из партиционирования

```
Line 1 ALTER TABLE my_logs2014m10 NO INHERIT my_logs;
```

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

Важность «constraint_exclusion» для партиционирования

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

Листинг 4.14 «constraint_exclusion» OFF

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

```
-
```

```
-
```

```
5 --
```

QUERY PLAN

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

4.3. Практика использования

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

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

4.4. Pg_partman

constraint_exclusion:

Листинг 4.15 «constraint_exclusion» ON

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

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

4.4 Pg_partman

Поскольку реализация партиционирования реализована неполноценно в PostgreSQL (для управления партициями и данными в них приходится писать функции, триггеры и правила), то существует расширение, кото-

рое автоматизирует полностью данный процесс. **PG Partition Manager**, он же `pg_partman`, это расширение для создания и управления партициями и партициями партиций (sub-partitioning) в PostgreSQL. Поддерживает партиционирование по времени (time-based) или по последовательности (serial-based). Для партиционирования по диапазону значений (range) существует отдельное расширение **Range Partitioning** (`range_partitioning`).

Текущая реализация поддерживает только INSERT операции, которые перенаправляют данные в нужную партицию. UPDATE операции, которые будут перемещать данные из одной партиции в другую, не поддерживаются. При попытке вставить данные, на которые нет партиции, `pg_partman` перемещает их в «мастер» (родительскую) таблицу. Данный вариант предпочтительнее, чем создавать автоматически новые партиции, поскольку это может привести к созданию десятков или сотен ненужных дочерних таблиц из-за ошибки в самих данных. Функция `check_parent` позволяет проверить попадание подобных данных в родительскую таблицу и решить, что с ними требуется делать (удалить или использовать `partition_data_time/partition_data_id` для создания и переноса этих данных в партиции).

Данное расширение использует большинство атрибутов родительской таблицы для создания партиций: индексы, внешние ключи (опционально), tablespace, constraints, privileges и ownership. Под такое условие попадают OID и UNLOGGED таблицы.

Партициями партиций (sub-partitioning) поддерживаются разных уровней: time->time, id->id, time->id и id->time. Нет лимитов на создание таких партиций, но стоит помнить, что большое число партиций влияет на производительность родительской таблицы. Если размер партиций станет слишком большим, то придется увеличивать `max_locks_per_transaction` параметр для базы данных (64 по умолчанию).

В PostgreSQL 9.4 появилась возможность создания пользовательских фоновых воркеров и динамически загружать их во время работы базы. Благодаря этому в `pg_partman` есть собственный фоновый воркер, задача которого запускать `run_maintenance` функцию каждый заданный промежуток времени. Если у Вас версия PostgreSQL ниже 9.4, то придется воспользоваться внешним планировщиком для выполнения данной функции (например cron). Задача данной функции - проверять и автоматически создавать партиции и опционально чистить старые.

Пример использования

Для начала установим данное расширение:

Листинг 4.16 Установка

```
Line 1 $ git clone https://github.com/keithf4/pg_partman.git
- $ cd pg_partman/
- $ make
```


4.4. Pg_partman

```
- $ sudo make install
```

Если не требуется использовать фоновый воркер, то можно собрать без него:

Листинг 4.17 Установка

```
Line 1 $ sudo make NO_BGW=1 install
```

Для работы фонового воркера нужно загружать его на старте PostgreSQL. Для этого потребуется добавить настройки в `postgresql.conf`:

Листинг 4.18 Настройки воркера

```
Line 1 shared_preload_libraries = 'pg_partman_bgw'      # (change
      requires restart)
- pg_partman_bgw.interval = 3600
- pg_partman_bgw.role = 'myrole'
- pg_partman_bgw.dbname = 'mydatabase'
```

где:

- `pg_partman_bgw.dbname` — база данных, в которой будет выполняться `run_maintenance` функция. Если нужно указать больше одной базы, то они указываются через запятую. Без этого параметра воркер не будет работать;
- `pg_partman_bgw.interval` — количество секунд между вызовами `run_maintenance` функции. По умолчанию 3600 (1 час);
- `pg_partman_bgw.role` — роль для запуска `run_maintenance` функции. По умолчанию `postgres`. Разрешена только одна роль;
- `pg_partman_bgw.analyze` — запускать или нет `ANALYZE` после создания партиций на родительскую таблицу. По умолчанию включено;
- `pg_partman_bgw.jobmon` — разрешить или нет использовать `pg_jobmon` расширение для мониторинга, что партиционирование работает без проблем. По умолчанию включено;

Далее подключаемся к базе данных и активируем расширение:

Листинг 4.19 Настройка расширения

```
Line 1 # CREATE SCHEMA partman;
- CREATE SCHEMA
- # CREATE EXTENSION pg_partman SCHEMA partman;
- CREATE EXTENSION
```

Теперь можно приступить к использованию расширения. Создадим и заполним таблицу тестовыми данными:

Листинг 4.20 Данные

```
Line 1 # CREATE TABLE users (
-     id                serial primary key,
```

4.4. Pg_partman

```
-      username      text not null unique ,
-      password      text ,
5      created_on    timestamptz not null ,
-      last_logged_on timestamptz not null
- );
-
- # INSERT INTO users (username, password, created_on ,
-   last_logged_on)
10  SELECT
-      md5(random() :: text) ,
-      md5(random() :: text) ,
-      now() - '1 years' :: interval * random() ,
-      now() - '1 years' :: interval * random()
15  FROM
-      generate_series(1, 10000);
```

Далее активируем расширение для поля `created_on` с партицией на каждый год:

Листинг 4.21 Партицирование

```
Line 1 # SELECT partman.create_parent('public.users', 'created_on',
-      'time', 'yearly');
-      create_parent
-      -----
-      t
5      (1 row)
```

Указывание схемы в имени таблицы обязательно, даже если она «public» (первый аргумент функции).

Поскольку родительская таблица уже была заполнена данными, перенесем данные из нее в партиции через `partition_data_time` функцию:

Листинг 4.22 Перенос данных в партиции

```
Line 1 # SELECT partman.check_parent();
-      check_parent
-      -----
-      (public.users,10000)
5      (1 row)
-
- # SELECT partman.partition_data_time('public.users', 1000);
-      partition_data_time
-      -----
10      10000
-      (1 row)
-
- # SELECT partman.check_parent();
-      check_parent
15  -----
```

4.4. Pg_partman

```
- (0 rows)
-
- # SELECT * FROM ONLY users;
- id | username | password | created_on | last_logged_on
20 -----+-----+-----+-----+-----
- (0 rows)
-
- # \d+ users
-
-      Table "public.users"
25      Column          |          Type          |          |
-      Modifiers        |          Storage      | Stats
-      target | Description
- --
-      -----+-----+-----+-----+-----
- id                | integer                | not null
-      default nextval('users_id_seq'::regclass) | plain                |
-      username      | text                   | not null
-                                     | extended            |
-      password      | text                   |
-                                     | extended            |
-      created_on    | timestamp with time zone | not null
-                                     | plain                |
30      last_logged_on | timestamp with time zone | not null
-                                     | plain                |
-
- Indexes:
-      "users_pkey" PRIMARY KEY, btree (id)
-      "users_username_key" UNIQUE CONSTRAINT, btree (username)
35 Triggers:
-      users_part_trig BEFORE INSERT ON users FOR EACH ROW
-      EXECUTE PROCEDURE users_part_trig_func()
- Child tables: users_p2012,
-                users_p2013,
-                users_p2014,
40      users_p2015,
-                users_p2016,
-                users_p2017,
-                users_p2018,
-                users_p2019,
45      users_p2020
```

В результате данные в таблице `users` содержатся в партициях благодаря

4.5. Pgslice

`pg_partman`. Более подробно по функционалу расширения, его настройках и ограничениях доступно в [официальной документации](#).

4.5 Pgslice

Pgslice — утилита для создания и управления партициями в PostgreSQL. Утилита разбивает на «куски» как новую, так и существующую таблицу с данными с нулевым временем простоя («zero downtime»).

Утилита написана на **Ruby**, поэтому потребуется сначала установить его. После этого устанавливаем `pgslice` через `rubygems` (многие `ruby` разработчики используют **bundler** для лучшего управления зависимостями, но в этой главе это не рассматривается):

Листинг 4.23 Установка

```
Line 1 $ gem install pgslice
```

Создадим и заполним таблицу тестовыми данными:

Листинг 4.24 Данные

```
Line 1 # CREATE TABLE users (  
-     id                serial primary key,  
-     username          text not null unique,  
-     password          text,  
5     created_on        timestampz not null,  
-     last_logged_on    timestampz not null  
- );  
-  
- # INSERT INTO users (username, password, created_on,  
-     last_logged_on)  
10 SELECT  
-     md5(random()::text),  
-     md5(random()::text),  
-     now() + '1 month'::interval * random(),  
-     now() + '1 month'::interval * random()  
15 FROM  
-     generate_series(1, 10000);
```

Настройки подключения к базе задаются через `PGSLICE_URL` переменную окружения:

Листинг 4.25 PGSLICE_URL

```
Line 1 $ export PGSLICE_URL=postgres://username:password@localhost/  
-     mydatabase
```

Через команду `pgslice prep <table> <column> <period>` создадим таблицу `<table>_intermediate` (`users_intermediate` в примере) с соответствующим триггером для разбиения данных, где `<table>` - это название таблицы (`users` в примере), `<column>` - поле, по которому будут создаваться

4.5. Pgslice

партиции, а `<period>` - период данных в партициях (может быть `day` или `month`).

Листинг 4.26 Pgslice prep

```
Line 1 $ pgslice prep users created_on month
- BEGIN;
-
- CREATE TABLE users_intermediate (LIKE users INCLUDING ALL);
5
- CREATE FUNCTION users_insert_trigger()
- RETURNS trigger AS $$
- BEGIN
-     RAISE EXCEPTION 'Create partitions first.';
10 END;
- $$ LANGUAGE plpgsql;
-
- CREATE TRIGGER users_insert_trigger
- BEFORE INSERT ON users_intermediate
15 FOR EACH ROW EXECUTE PROCEDURE users_insert_trigger();
-
- COMMENT ON TRIGGER users_insert_trigger ON
- users_intermediate IS 'column:created_on, period:month,
- cast:timestampz';
-
- COMMIT;
```

Теперь можно добавить партиции:

Листинг 4.27 Pgslice add_partitions

```
Line 1 $ pgslice add_partitions users --intermediate --past 3 --
- future 3
- BEGIN;
-
- CREATE TABLE users_201611
5 (CHECK (created_on >= '2016-11-01 00:00:00 UTC'::
- timestampz AND created_on < '2016-12-01 00:00:00 UTC'::
- timestampz))
- INHERITS (users_intermediate);
-
- ALTER TABLE users_201611 ADD PRIMARY KEY (id);
-
10 ...
-
- CREATE OR REPLACE FUNCTION users_insert_trigger()
- RETURNS trigger AS $$
- BEGIN
15 IF (NEW.created_on >= '2017-02-01 00:00:00 UTC'::
- timestampz AND NEW.created_on < '2017-03-01 00:00:00 UTC
```

4.5. Pgslice

```
'::timestampz) THEN
-         INSERT INTO users_201702 VALUES (NEW.*);
-         ELSIF (NEW.created_on >= '2017-03-01 00:00:00 UTC'::
timestampz AND NEW.created_on < '2017-04-01 00:00:00 UTC
'::timestampz) THEN
-         INSERT INTO users_201703 VALUES (NEW.*);
-         ELSIF (NEW.created_on >= '2017-04-01 00:00:00 UTC'::
timestampz AND NEW.created_on < '2017-05-01 00:00:00 UTC
'::timestampz) THEN
20         INSERT INTO users_201704 VALUES (NEW.*);
-         ELSIF (NEW.created_on >= '2017-05-01 00:00:00 UTC'::
timestampz AND NEW.created_on < '2017-06-01 00:00:00 UTC
'::timestampz) THEN
-         INSERT INTO users_201705 VALUES (NEW.*);
-         ELSIF (NEW.created_on >= '2017-01-01 00:00:00 UTC'::
timestampz AND NEW.created_on < '2017-02-01 00:00:00 UTC
'::timestampz) THEN
-         INSERT INTO users_201701 VALUES (NEW.*);
25         ELSIF (NEW.created_on >= '2016-12-01 00:00:00 UTC'::
timestampz AND NEW.created_on < '2017-01-01 00:00:00 UTC
'::timestampz) THEN
-         INSERT INTO users_201612 VALUES (NEW.*);
-         ELSIF (NEW.created_on >= '2016-11-01 00:00:00 UTC'::
timestampz AND NEW.created_on < '2016-12-01 00:00:00 UTC
'::timestampz) THEN
-         INSERT INTO users_201611 VALUES (NEW.*);
-         ELSE
30         RAISE EXCEPTION 'Date out of range. Ensure
partitions are created.';
-         END IF;
-         RETURN NULL;
-     END;
-     $$ LANGUAGE plpgsql;
35
- COMMIT;
```

Через `--past` и `--future` опции указывается количество партиций. Далее можно переместить данные в партиции:

Листинг 4.28 Pgslice fill

```
Line 1 $ pgslice fill users
- /* 1 of 1 */
- INSERT INTO users_intermediate ("id", "username", "password"
, "created_on", "last_logged_on")
- SELECT "id", "username", "password", "created_on", "
last_logged_on" FROM users
5 WHERE id > 0 AND id <= 10000 AND created_on >= '
2016-11-01 00:00:00 UTC'::timestampz AND created_on < '
```

4.5. Pgslice

```
2017-06-01 00:00:00 UTC'::timestampz
```

Через `--batch-size` и `--sleep` опции можно управлять скоростью переноса данных.

После этого можно переключиться на новую таблицу с партициями:

Листинг 4.29 Pgslice swap

```
Line 1 $ pgslice swap users
- BEGIN;
-
- SET LOCAL lock_timeout = '5s';
5
- ALTER TABLE users RENAME TO users_retired;
-
- ALTER TABLE users_intermediate RENAME TO users;
-
10 ALTER SEQUENCE users_id_seq OWNED BY users.id;
-
- COMMIT;
```

Если требуется, то можно перенести часть данных, что накопилась между переключением таблиц:

Листинг 4.30 Pgslice fill

```
Line 1 $ pgslice fill users --swapped
```

В результате таблица `users` будет работать через партиции:

Листинг 4.31 Результат

```
Line 1 $ psql -c "EXPLAIN SELECT * FROM users"
-
- QUERY PLAN
-
- -----
-
- Append (cost=0.00..330.00 rows=13601 width=86)
5  -> Seq Scan on users (cost=0.00..0.00 rows=1 width=84)
-  -> Seq Scan on users_201611 (cost=0.00..17.20 rows=720
-    width=84)
-  -> Seq Scan on users_201612 (cost=0.00..17.20 rows=720
-    width=84)
-  -> Seq Scan on users_201701 (cost=0.00..17.20 rows=720
-    width=84)
-  -> Seq Scan on users_201702 (cost=0.00..166.48 rows
-    =6848 width=86)
10 -> Seq Scan on users_201703 (cost=0.00..77.52 rows=3152
-    width=86)
-  -> Seq Scan on users_201704 (cost=0.00..17.20 rows=720
-    width=84)
-  -> Seq Scan on users_201705 (cost=0.00..17.20 rows=720
-    width=84)
```

4.6. Заключение

- (9 rows)

Старая таблица теперь будет называться `<table>_retired` (`users_retired` в примере). Её можно оставить или удалить из базы.

Листинг 4.32 Удаление старой таблицы

```
Line 1 $ pg_dump -c -Fc -t users_retired $PGSLICE_URL >
      users_retired.dump
- $ psql -c "DROP users_retired" $PGSLICE_URL
```

Далее только требуется следить за количеством партиций. Для этого команду `pgslice add_partitions` можно добавить в cron:

Листинг 4.33 Cron

```
Line 1 # day
- 0 0 * * * pgslice add_partitions <table> --future 3 --url
      ...
-
- # month
5 0 0 1 * * pgslice add_partitions <table> --future 3 --url
      ...
```

4.6 Заключение

Партиционирование — одна из самых простых и менее безболезненных методов уменьшения нагрузки на СУБД. Именно на этот вариант стоит посмотреть сперва, и если он не подходит по каким либо причинам — переходить к более сложным.

Репликация

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

Ричард Филлипс Фейнман

5.1 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 А происходит вставка строки в реплику X, после чего транзакция В 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 В не находит в Y строку, подлежащую удалению, и не выполняет своё действие, после чего транзакция А 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

5.2. Поточковая репликация (Streaming Replication)

вида.

Для репликации PostgreSQL существует несколько решений, как закрытых, так и свободных. Закрытые системы репликации не будут рассматриваться в этой книге. Вот список свободных решений:

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

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

5.2 Поточковая репликация (Streaming Replication)

Поточковая репликация (Streaming Replication, SR) дает возможность непрерывно отправлять и применять WAL (Write-Ahead Log) записи на резервные сервера для создания точной копии текущего. Данная функциональность появилась у PostgreSQL начиная с 9 версии. Этот тип репликации простой, надежный и, вероятней всего, будет использоваться в качестве стандартной репликации в большинстве высоконагруженных приложений, что используют PostgreSQL.

Отличительными особенностями решения являются:

5.2. Поточковая репликация (Streaming Replication)

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

К недостаткам можно отнести:

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

Установка

Для начала нам потребуется PostgreSQL не ниже 9 версии. Все работы, как полагается, будут проводиться на Linux.

Настройка

Обозначим мастер сервер как `masterdb(192.168.0.10)` и слейв как `slavedb(192.168.0.20)`.

Предварительная настройка

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

Листинг 5.1 Создаем пользователя `userssh`

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

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

Листинг 5.2 Логинимся под пользователем `postgres`

```
Line 1 $ su postgres
```

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

Листинг 5.3 Генерируем RSA-ключ

```
Line 1 $ ssh-keygen -t rsa -P ""
- Generating public/private rsa key pair.
- Enter file in which to save the key (/var/lib/postgresql/.ssh/id_rsa):
- Created directory '/var/lib/postgresql/.ssh'.
```

5.2. Поточковая репликация (Streaming Replication)

```
5 Your identification has been saved in /var/lib/postgresql/.
  ssh/id_rsa.
- Your public key has been saved in /var/lib/postgresql/.ssh/
  id_rsa.pub.
- The key fingerprint is:
- 16:08:27:97:21:39:b5:7b:86:e1:46:97:bf:12:3d:76
  postgres@localhost
```

И добавляем его в список авторизованных ключей:

Листинг 5.4 Добавляем его в список авторизованных ключей

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

Проверить работоспособность соединения можно просто написав:

Листинг 5.5 Пробуем зайти на ssh без пароля

```
Line 1 $ ssh localhost
```

Не забываем предварительно инициализировать sshd:

Листинг 5.6 Запуск sshd

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

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

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

Листинг 5.7 Мастер `pg_hba.conf`

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

Листинг 5.8 Слейв `pg_hba.conf`

```
Line 1 host replication all 192.168.0.10/32 trust
```

Не забываем после этого перезагрузить postgresql на обоих серверах.

Настройка мастера

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

Листинг 5.9 Настройка мастера

```
Line 1 # To enable read-only queries on a standby server, wal_level
      must be set to
- # "hot_standby". But you can choose "archive" if you never
      connect to the
- # server in standby mode.
- wal_level = hot_standby
```

5.2. Поточковая репликация (Streaming Replication)

```
5
- # Set the maximum number of concurrent connections from the
  standby servers.
- max_wal_senders = 5
-
- # To prevent the primary server from removing the WAL
  segments required for
10 # the standby server before shipping them, set the minimum
  number of segments
- # retained in the pg_xlog directory. At least
  wal_keep_segments should be
- # larger than the number of segments generated between the
  beginning of
- # online-backup and the startup of streaming replication. If
  you enable WAL
- # archiving to an archive directory accessible from the
  standby, this may
15 # not be necessary.
- wal_keep_segments = 32
-
- # Enable WAL archiving on the primary to an archive
  directory accessible from
- # the standby. If wal_keep_segments is a high enough number
  to retain the WAL
20 # segments required for the standby server, this may not be
  necessary.
- archive_mode = on
- archive_command = 'cp %p /path_to/archive/%f'
```

Давайте по порядку:

- `wal_level = hot_standby` — сервер начнет писать в WAL логи так же как и при режиме «archive», добавляя информацию, необходимую для восстановления транзакции (можно также поставить `archive`, но тогда сервер не может быть слейвом при необходимости);
- `max_wal_senders = 5` — максимальное количество слейвов;
- `wal_keep_segments = 32` — минимальное количество файлов с WAL сегментами в `pg_xlog` директории;
- `archive_mode = on` — позволяем сохранять WAL сегменты в указанное переменной `archive_command` хранилище. В данном случае в директорию `/path/to/archive/`;

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

5.2. Поточковая репликация (Streaming Replication)

Листинг 5.10 recovery.conf для синхронной репликации на слейве

```
Line 1 restore_command = 'cp /mnt/server/archivedir/%f %p'
        # e.g. 'cp /mnt/server/archivedir/%f %p'
- standby_mode = on
- primary_conninfo = 'host=masterdb port=59121 user=
    replication password=replication application_name=
    newcluster' # e.g. 'host=localhost port=5432'
- trigger_file = '/tmp/trig_f_newcluster'
```

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

Настройка слейва

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

Переместимся на `masterdb` сервер и выполним в консоли:

Листинг 5.11 Выполняем на мастере

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

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

Листинг 5.12 Выполняем на мастере

```
Line 1 $ rsync -C -a --delete -e ssh --exclude postgresql.conf --
    exclude postmaster.pid \
- --exclude postmaster.opts --exclude pg_log --exclude pg_xlog
    \
- --exclude recovery.conf master_db_datadir/ slavedb_host:
    slave_db_datadir/
```

где

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

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

Листинг 5.13 Выполняем на мастере

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

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

Листинг 5.14 Выполняем на слейве

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

5.2. Поточковая репликация (Streaming Replication)

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

Листинг 5.15 Конфиг слейва

```
Line 1 hot_standby = on
```

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

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

Листинг 5.16 Конфиг recovery.conf

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

где

- `standby_mode='on'` — указываем серверу работать в режиме слейв;
- `primary_conninfo` — настройки соединения слейва с мастером;
- `trigger_file` — указываем триггер файл, при наличии которого будет остановлена репликация;

5.2. Поточковая репликация (Streaming Replication)

- `restore_command` — команда, которой будут восстанавливаться WAL логи. В нашем случае через `scp` копируем с `masterdb` (`masterdb_host` - хост `masterdb`);

Теперь можем запустить PostgreSQL на `slavedb`.

Тестирование репликации

В результате можем посмотреть отставание слейвов от мастера с помощью таких команд:

Листинг 5.17 Тестирование репликации

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

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

Листинг 5.18 Состояние слейвов

```
Line 1 # SELECT * from pg_stat_replication ;
-  procpid | usesysid |  username  | application_name |
-  client_addr | client_hostname | client_port |
-  backend_start      | state  | sent_location |
-  write_location | flush_location | replay_location |
-  sync_priority | sync_state
-  --
-  -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

5.2. Поточковая репликация (Streaming Replication)

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

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

Листинг 5.19 Состояние слейва

```
Line 1 # SELECT * from pg_stat_database_conflicts ;
-      datid | datname | confl_tablespace | confl_lock |
confl_snapshot | confl_bufferpin | confl_deadlock
-      --
-      ----+-----+-----+-----+-----+-----+
-          1 | template1 |          0 |          0 |
          0 |          0 |          0 |          0 |
5      11979 | template0 |          0 |          0 |
          0 |          0 |          0 |          0 |
-      11987 | postgres |          0 |          0 |
          0 |          0 |          0 |          0 |
-      16384 | marc     |          0 |          0 |
          1 |          0 |          0 |          0 |
```

Еще проверить работу репликации можно с помощью утилиты `ps`:

Листинг 5.20 Тестирование репликации

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

репликацию

Давайте проверим репликацию и выполним на мастере:

Листинг 5.21 Выполняем на мастере

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

5.2. Поточковая репликация (Streaming Replication)

```
- INSERT 0 1
- test_db=#
```

Теперь проверим на слейве результат:

Листинг 5.22 Выполняем на слейве

```
Line 1 $ psql test_db
- test_db=# select * from test3;
- id | name
- ----+-----
5    1 | test1
- (1 row)
```

Как видим, таблица с данными успешно скопирована с мастера на слейв. Более подробно по настройке данной репликации можно почитать из [официальной wiki](#).

Общие задачи

Переключение на слейв при падении мастера

Достаточно создать триггер файл (`trigger_file`) на слейве, который перестанет читать данные с мастера.

Остановка репликации на слейве

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

Перезапуск репликации после сбоя

Повторяем операции из раздела «[Настройка слейва](#)». Хотелось заметить, что мастер при этом не нуждается в остановке при выполнении данной задачи.

Перезапуск репликации после сбоя слейва

Перезагрузить PostgreSQL на слейве после устранения сбоя.

Повторно синхронизировать репликации на слейве

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

5.2. Поточковая репликация (Streaming Replication)

Repmgr

Repmgr — набор инструментов для управления потоковой репликацией и восстановления после сбоя кластера PostgreSQL серверов. Он автоматизирует настройку резервных серверов, мониторинг репликации, а также помогает выполнять задачи администрированию кластера, такие как отказоустойчивость (failover) или переключение мастера-слейва (слейв становится мастером, а мастер - слейвом). Repmgr работает с версии PostgreSQL 9.3 и выше.

Repmgr состоит из двух утилит:

- **repmgr** — инструмент командной строки (cli), который используется для административных задач, таких как:
 - создание слейвов;
 - переключение слейва в режим мастера;
 - переключение между собой мастер и слейв серверов;
 - отображение состояния кластера;
- **repmgrd** — демон, который мониторит кластер серверов и выполняет такие задачи:
 - мониторинг и логирование эффективности репликации;
 - автоматическое переключение слейва в мастер при обнаружении проблем у текущего мастера (failover);
 - посылка сообщений о событиях в кластере через заданные пользователем скрипты;

Пример использования: автоматическое переключение слейва в мастер

Для использования failover потребуется добавить `repmgr_funcs` в `postgresql.conf`:

Листинг 5.23 repmgr_funcs

```
Line 1 shared_preload_libraries = 'repmgr_funcs'
```

И добавить настройки в `repmgr.conf`:

Листинг 5.24 repmgr.conf

```
Line 1 failover=automatic
- promote_command='repmgr standby promote -f /etc/repmgr.conf
  --log-to-file '
- follow_command='repmgr standby follow -f /etc/repmgr.conf --
  log-to-file '
```

Для демонстрации автоматического failover, настроен кластер с тремя узлами репликации (один мастер и два слейв сервера), так что таблица `repl_nodes` выглядит следующим образом:

5.2. Поточковая репликация (Streaming Replication)

Листинг 5.25 repl_nodes

```
Line 1 # SELECT id, type, upstream_node_id, priority, active FROM
      repmgr_test.repl_nodes ORDER BY id;
- id | type | upstream_node_id | priority | active
- ---+-----+-----+-----+-----
- 1 | master | | 100 | t
5 2 | standby | 1 | 100 | t
- 3 | standby | 1 | 100 | t
- (3 rows)
```

После запуска `repmgrd` демона на каждом сервере в режиме ожидания, убеждаемся что он мониторит кластер:

Листинг 5.26 logs

```
Line 1 [2016-01-05 13:15:40] [INFO] checking cluster configuration
      with schema 'repmgr_test'
- [2016-01-05 13:15:40] [INFO] checking node 2 in cluster '
      test'
- [2016-01-05 13:15:40] [INFO] reloading configuration file
      and updating repmgr tables
- [2016-01-05 13:15:40] [INFO] starting continuous standby
      node monitoring
```

Теперь остановим мастер базу:

Листинг 5.27 Остановка текущего мастера

```
Line 1 pg_ctl -D /path/to/node1/data -m immediate stop
```

`repmgrd` автоматически замечает падение мастера и переключает один из слейвов в мастер:

Листинг 5.28 Переключение слейва в мастер

```
Line 1 [2016-01-06 18:32:58] [WARNING] connection to upstream has
      been lost, trying to recover... 15 seconds before
      failover decision
- [2016-01-06 18:33:03] [WARNING] connection to upstream has
      been lost, trying to recover... 10 seconds before
      failover decision
- [2016-01-06 18:33:08] [WARNING] connection to upstream has
      been lost, trying to recover... 5 seconds before failover
      decision
- ...
5 [2016-01-06 18:33:18] [NOTICE] this node is the best
      candidate to be the new master, promoting...
- ...
- [2016-01-06 18:33:20] [NOTICE] STANDBY PROMOTE successful
```

Также переключает оставшийся слейв на новый мастер:

5.2. Поточковая репликация (Streaming Replication)

Листинг 5.29 Переключение слейва на новый мастер

```
Line 1 [2016-01-06 18:32:58] [WARNING] connection to upstream has
      been lost , trying to recover... 15 seconds before
      failover decision
- [2016-01-06 18:33:03] [WARNING] connection to upstream has
      been lost , trying to recover... 10 seconds before
      failover decision
- [2016-01-06 18:33:08] [WARNING] connection to upstream has
      been lost , trying to recover... 5 seconds before failover
      decision
- ...
5 [2016-01-06 18:33:23] [NOTICE] node 2 is the best candidate
      for new master , attempting to follow...
- [2016-01-06 18:33:23] [INFO] changing standby's master
- ...
- [2016-01-06 18:33:25] [NOTICE] node 3 now following new
      upstream node 2
```

Таблица `repl_nodes` будет обновлена, чтобы отразить новую ситуацию — старый мастер `node1` помечен как неактивный, и слейв `node3` теперь работает от нового мастера `node2`:

Листинг 5.30 Результат после failover

```
Line 1 # SELECT id , type , upstream_node_id , priority , active from
      repl_nodes ORDER BY id ;
- id | type | upstream_node_id | priority | active
- ---+-----+-----+-----+-----
- 1 | master | | 100 | f
5 2 | master | | 100 | t
- 3 | standby | 2 | 100 | t
- (3 rows)
```

В таблицу `repl_events` будут добавлены записи того, что произошло с каждым сервером во время failover:

Листинг 5.31 Результат после failover

```
Line 1 # SELECT node_id , event , successful , details from
      repmgr_test.repl_events where event_timestamp>=
      '2016-01-06 18:30';
- node_id | event | successful |
- details
- --
- -----+-----+-----+-----
- 2 | standby_promote | t | node 2
- was successfully promoted to master
5 2 | repmgrd_failover_promote | t | node 2
- promoted to master; old master 1 marked as failed
```

5.2. Поточковая репликация (Streaming Replication)

```
-          3 | repmgrd_failover_follow | t          | node 3  
-    now following new upstream node 2  
- (3 rows)
```

Заключение

Более подробно по функционалу, его настройках и ограничениях доступно в [официальном репозитории](#).

Patroni

Patroni — это демон на Python, позволяющий автоматически обслуживать кластеры PostgreSQL с потоковой репликацией.

Особенности:

- Использует потоковую PostgreSQL репликацию (асинхронная и синхронная репликация);
- Интеграция с **Kubernetes**;
- Поддержания актуальности кластера и выборов мастера используются распределенные **DCS** хранилища (поддерживаются **Zookeeper**, **Etcd** или **Consul**);
- Автоматическое «service discovery» и динамическая реконфигурация кластера;
- Состояние кластера можно получить как запросами в DCS, так и напрямую к Patroni через HTTP запросы;

Информация по настройке и использованию Patroni находится в [официальной документации](#) проекта.

Stolon

Stolon — это демон на Go, позволяющий автоматически обслуживать кластеры PostgreSQL с потоковой репликацией.

Особенности:

- Использует потоковую PostgreSQL репликацию (асинхронная и синхронная репликация);
- Интеграция с **Kubernetes**;
- Поддержания актуальности кластера и выборов мастера используются распределенные **DCS** хранилища (поддерживаются **Etcd** или **Consul**);
- Автоматическое «service discovery» и динамическая реконфигурация кластера;
- Состояние кластера можно получить как запросами в DCS, так и через `stolonctl` клиент;

5.2. Потокная репликация (Streaming Replication)

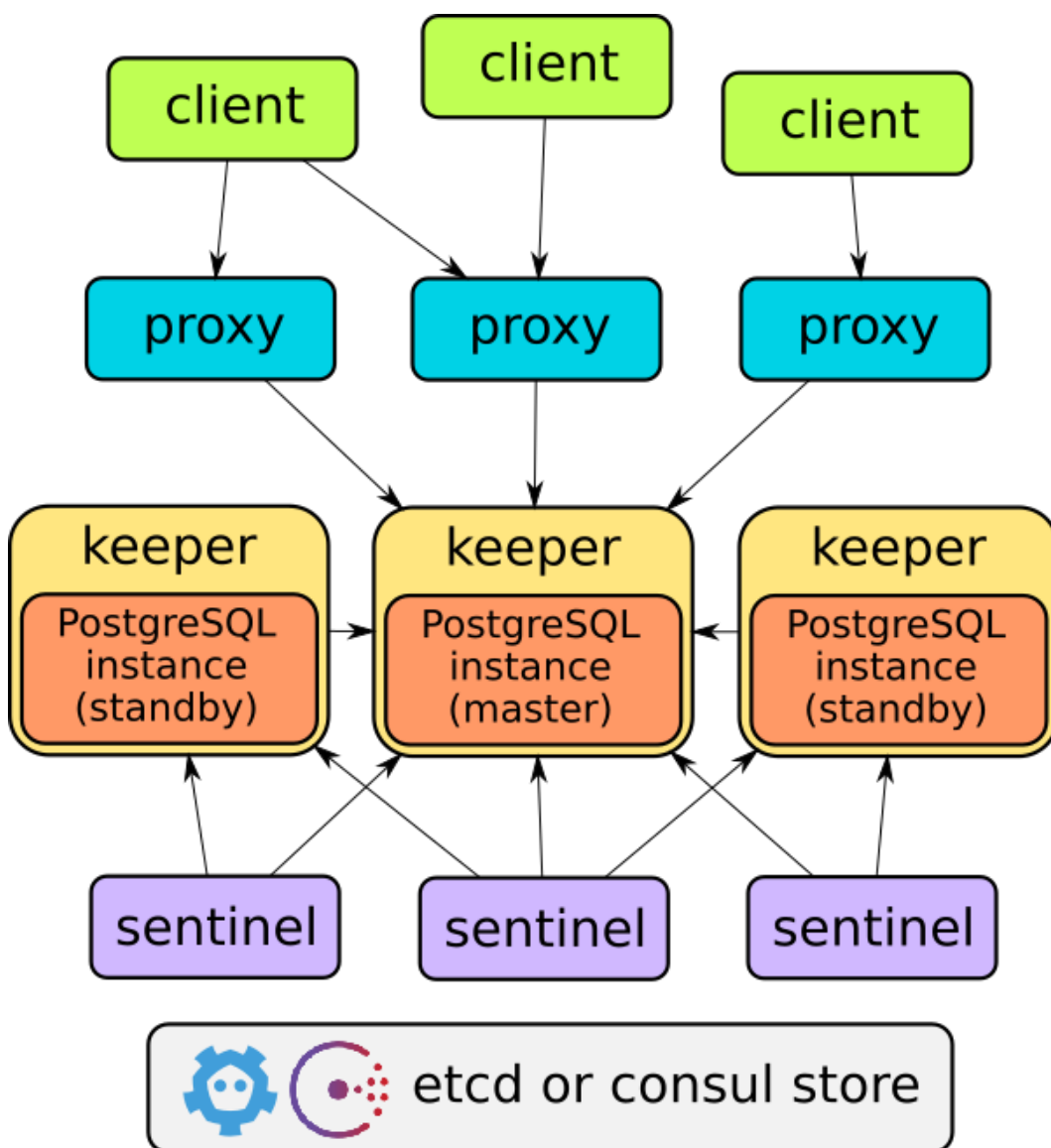


Рис. 5.1: Stolon архитектура

Stolon состоит из 3 основных компонентов:

- keeper (хранитель): управляет экземпляром PostgreSQL;
- sentinel: обнаруживает и контролирует keeper-ров и вычисляет оптимальное состояние кластера;
- proxy: точка доступа клиента, обеспечивает подключение к верному PostgreSQL мастеру в кластере;

Информация по настройке и использованию Stolon находится в [официальной документации](#) проекта.

5.3 PostgreSQL Bi-Directional Replication (BDR)

BDR (Bi-Directional Replication) это новая функциональность добавленная в ядро PostgreSQL которая предоставляет расширенные средства для репликации. На данный момент это реализовано в виде небольшого патча и модуля для 9.4 версии. Заявлено что полностью будет только в PostgreSQL 9.6 (разработчики решили не заниматься поддержкой патча для 9.5, а сосредоточиться на добавление патчей в сам PostgreSQL). BDR позволяет создавать географически распределенные асинхронные мульти-мастер конфигурации используя для этого встроенную логическую потоковую репликацию LLSR (Logical Log Streaming Replication).

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

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

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

5.4. Pglogical

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

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

5.4 Pglogical

Pglogical — это расширение для PostgreSQL, которое использует логическое декодирование через publish/subscribe модель. Данное расширение базируется на BDR проекте ([5.3 PostgreSQL Bi-Directional Replication \(BDR\)](#)). Расширение работает только начиная с версии PostgreSQL 9.4 и выше (из-за логического декодирования). Для разных вариаций обнаружения и разрешения конфликтов требуется версия 9.5 и выше.

Используются следующие термины для описания pglogical:

- Nodes(ноды, узлы) — экземпляры баз данных PostgreSQL;
- Provider и subscriber — роли узлов. Provider выполняют выдачу данных и изменений для subscriber-ов;
- Replication set (набор для репликации) — коллекция таблиц и последовательностей для репликации;

Сценарии использования pglogical:

- Обновление между версиями PostgreSQL (например с 9.4 на 9.5);
- Полная репликация базы данных;
- Выборочная репликация таблиц;
- Сбор данных с нескольких баз данных в одну;

Архитектурные детали:

- Pglogical работает на уровне каждой базы данных, а не на весь сервер;
- Provider (publisher) может «кормить» несколько subscriber-ов без дополнительных накладных расходов записи на диск;

5.4. Pglogical

- Один subscriber может объединить изменения из нескольких provider-ов и использовать систему обнаружения и разрешения конфликтов между изменениями;
- Каскадная репликация осуществляется в виде переадресации изменений;

Установка и настройка

Установить pglogical можно по [данной документации](#). Далее требуется настроить логический декодинг в PostgreSQL:

Листинг 5.32 postgresql.conf

```
Line 1 wal_level = 'logical'
- max_worker_processes = 10 # one per database needed on
  provider node
-
  subscriber node # one per node needed on
- max_replication_slots = 10 # one per node needed on
  provider node
5 max_wal_senders = 10 # one per node needed on
  provider node
- shared_preload_libraries = 'pglogical'
```

Если используется PostgreSQL 9.5+ и требуются механизмы разрешения конфликтов, то требуется добавить дополнительные опции:

Листинг 5.33 postgresql.conf

```
Line 1 track_commit_timestamp = on # needed for last/first update
  wins conflict resolution
-
  PostgreSQL 9.5+ # property available in
```

В `pg_hba.conf` нужно разрешить replication соединения с локального хоста для пользователя с привилегией репликации. После перезапуска базы нужно активировать расширение на всех нодах:

Листинг 5.34 Активируем расширение

```
Line 1 CREATE EXTENSION pglogical;
```

Далее на master (мастер) создаем provider (процесс, который будет выдавать изменения для subscriber-ов) ноду:

Листинг 5.35 Создаем provider

```
Line 1 SELECT pglogical.create_node(
-   node_name := 'provider1',
-   dsn := 'host=providerhost port=5432 dbname=db'
- );
```

И добавляем все таблицы в `public` схеме:

5.4. Pglogical

Листинг 5.36 Добавляем в replication set все таблицы в public схеме

```
Line 1 SELECT pglogical.replication_set_add_all_tables('default',  
          ARRAY['public']);
```

Далее переходим на slave (слейв) и создаем subscriber ноду:

Листинг 5.37 Создаем subscriber

```
Line 1 SELECT pglogical.create_node(  
-     node_name := 'subscriber1',  
-     dsn := 'host=thishost port=5432 dbname=db'  
- );
```

После этого создаем «подписку» на provider ноду, которая начнет синхронизацию и репликацию в фоне:

Листинг 5.38 Активируем subscriber

```
Line 1 SELECT pglogical.create_subscription(  
-     subscription_name := 'subscription1',  
-     provider_dsn := 'host=providerhost port=5432 dbname=db'  
- );
```

Если все сделано верно, subscriber через определенный интервал времени subscriber нода должна получить точную копию всех таблиц в public схеме с master хоста.

Разрешение конфликтов

Если используется схема, где subscriber нода подписана на данные из нескольких provider-ов, или же на subscriber дополнительно производятся локальные изменения данных, могут возникать конфликты для новых изменений. В pglogical встроен механизм для обнаружения и разрешения конфликтов. Настройка данного механизма происходит через `pglogical.conflict_resolution` ключ. Поддерживаются следующие значения:

- `error` - репликация остановится на ошибке, если обнаруживается конфликт и потребуются ручное действие для его разрешения;
- `apply_remote` - всегда применить изменения, который конфликтуют с локальными данными. Значение по умолчанию;
- `keep_local` - сохранить локальную версию данных и игнорировать конфликтующие изменения, которые исходят от provider-a;
- `last_update_wins` - версия данных с самым новым коммитом (newest commit timestamp) будет сохранена;
- `first_update_wins` - версия данных с самым старым коммитом (oldest commit timestamp) будет сохранена;

Когда опция `track_commit_timestamp` отключена, единственное допустимое значение для `pglogical.conflict_resolution` может быть `apply_remote`. Поскольку `track_commit_timestamp` не доступен в PostgreSQL 9.4, данная опция установлена по умолчанию в `apply_remote`.

Ограничения и недостатки

- Для работы требуется суперпользователь;
- **UNLOGGED** и **TEMPORARY** таблицы не реплицируются;
- Для каждой базы данных нужно настроить отдельный provider и subscriber;
- Требуется primary key или **replica identity** для репликации;
- Разрешен только один уникальный индекс/ограничение/основной ключ на таблицу (из-за возможных конфликтов). Возможно использовать больше, но только в случае если subscriber читает только с одного provider и не производится локальных изменений данных на нем;
- Автоматическая репликация DDL не поддерживается. У pglogical есть команда `pglogical.replicate_ddl_command` для запуска DDL на provider и subscriber;
- Ограничения на foreign ключи не выполняются на subscriber-рах;
- При использовании **TRUNCATE ... CASCADE** будет выполнен **CASCADE** только на provider;
- Последовательности реплицируются периодически, а не в режиме реального времени;

5.5 Slony-I

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

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

Установка

Установка на Ubuntu производится простой командой:

Листинг 5.39 Установка

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

Настройка

Рассмотрим установку на гипотетическую базу данных customers. Исходные данные:

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

Подготовка master базы

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

Листинг 5.40 Подготовка master-сервера

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

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

Подготовка slave базы

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

Листинг 5.41 Подготовка одного slave-сервера

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

должна подключать нас к мастер-серверу (после ввода пароля, желательно).

Теперь устанавливаем на slave-хост сервер PostgreSQL. Следующего обычно не требуется, сразу после установки Postgres «up and ready», но в случае каких-то ошибок можно начать «с чистого листа», выполнив следующие команды (предварительно сохранив конфигурационные файлы и остановив `postmaster`):

Листинг 5.42 Подготовка одного slave-сервера

```
Line 1 pgsq@customers_slave$ rm -rf $PGDATA
- pgsq@customers_slave$ mkdir $PGDATA
- pgsq@customers_slave$ initdb -E UTF8 -D $PGDATA
```

5.5. Slony-I

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

Далее запускаем postmaster. Обычно требуется определённый владелец для реплицируемой БД. В этом случае необходимо создать его тоже:

Листинг 5.43 Подготовка одного slave-сервера

```
Line 1 postgresql@customers_slave$ createuser -a -d customers_owner
- postgresql@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`):

Листинг 5.44 Инициализация БД и plpgsql на slave

```
Line 1 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`.

Далее создаём таблицы и всё остальное на slave базе:

Листинг 5.45 Инициализация БД и plpgsql на slave

```
Line 1 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 пускаться без пароля, а вот для `psql -U slony -h customers_slave.com customers` придётся набрать пароль (`slony_user_pass`). Важно: подразумевается что сейчас на мастер-хосте ещё не установлен Slony (речь не про `make install`), то есть в БД нет таблиц `sl_*`, триггеров и прочего.

5.5. Slony-I

Инициализация кластера

Сейчас мы имеем два сервера PostgreSQL которые свободно «видят» друг друга по сети, на одном из них находится мастер-база с данными, на другом — только структура базы. Далее мастер-хосте запускаем скрипт:

Листинг 5.46 Инициализация кластера

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


5.5. Slony-I

```
- set add table ( set id = 1, origin = 1, id = 5, full
    qualified
- name = 'public.customers_something',
40 comment = 'Table public.customers_something );
- echo ' done';
-
- echo 'done adding';
- store node ( id = 2, comment = 'Node 2, $HOST2' );
45 echo 'stored node';
- store path ( server = 1, client = 2, conninfo = 'dbname=
    $DBNAME1 host=$HOST1
- port=$PORT1 user=slony password=slony_user_password' );
- echo 'stored path';
- store path ( server = 2, client = 1, conninfo = 'dbname=
    $DBNAME2 host=$HOST2
50 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 инициализируется кластер, создается replication set, включаются в него две таблицы. Нужно перечислить все таблицы, которые нужно реплицировать. Replication set запоминается раз и навсегда. Чтобы добавить узел в схему репликации не нужно заново инициализировать set. Если в набор добавляется или удаляется таблица нужно переподписать все узлы. То есть сделать `unsubscribe` и `subscribe` заново.

Подписываем slave-узел на replication set

Далее запускаем на слейве:

Листинг 5.47 Подписываем slave-узел на replication set

```
Line 1 #!/bin/sh
-
- CLUSTER=customers_rep
-
5 DBNAME1=customers
- DBNAME2=customers
-
- HOST1=customers_master.com
- HOST2=customers_slave.com
10
- PORT1=5432
- PORT2=5432
-
- SLONY_USER=slony
15
```

5.5. Slony-I

```
- slonik <<EOF
- cluster name = $CLUSTER;
- node 1 admin conninfo = 'dbname=$DBNAME1 host=$HOST1
- port=$PORT1 user=slony password=slony_user_password';
20 node 2 admin conninfo = 'dbname=$DBNAME2 host=$HOST2
- port=$PORT2 user=slony password=slony_user_password';
-
- echo 'subscribing';
- subscribe set ( id = 1, provider = 1, receiver = 2, forward
    = no);
25
- EOF
```

Старт репликации

Теперь, на обоих узлах необходимо запустить демона репликации.

Листинг 5.48 Старт репликации

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

и

Листинг 5.49 Старт репликации

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

Слоны обмениваются сообщениями и начнут передачу данных. Начальное наполнение происходит с помощью `COPY` команды, слейв база в это время полностью блокируется.

Общие задачи

Добавление ещё одного узла в работающую схему репликации

Требуется выполнить [5.5](#) и [5.5](#) этапы. Новый узел имеет `id = 3`. Находится на хосте `customers_slave3.com`, «видит» мастер-сервер по сети и мастер может подключиться к его PostgreSQL. После дублирования структуры (п [5.5.2](#)) делается следующее:

Листинг 5.50 Общие задачи

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

5.5. Slony-I

- EOF

Это нужно чтобы удалить схему, триггеры и процедуры, которые были сдублированы вместе с таблицами и структурой БД. Инициализировать кластер не надо. Вместо этого записываем информацию о новом узле в сети:

Листинг 5.51 Общие задачи

```
Line 1 #!/bin/sh
-
- CLUSTER=customers_rep
-
5 DBNAME1=customers
- DBNAME3=customers
-
- HOST1=customers_master.com
- HOST3=customers_slave3.com
10
- PORT1=5432
- PORT2=5432
-
- SLONY_USER=slony
15
- slonik <<EOF
- cluster name = $CLUSTER;
- node 1 admin conninfo = 'dbname=$DBNAME1 host=$HOST1
- port=$PORT1 user=slony password=slony_user_pass';
20 node 3 admin conninfo = 'dbname=$DBNAME3
- host=$HOST3 port=$PORT2 user=slony password=slony_user_pass'
- ;
-
- echo 'done adding';
-
25 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';
30 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 );
35 store listen ( origin = 3, provider = 3, receiver = 1 );
```

5.5. Slony-I

- EOF

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

Листинг 5.52 Общие задачи

```
Line 1 #!/bin/sh
-
- CLUSTER=customers_rep
-
5 DBNAME1=customers
- DBNAME3=customers
-
- HOST1=customers_master.com
- HOST3=customers_slave3.com
10
- PORT1=5432
- PORT2=5432
-
- SLONY_USER=slony
15
- slonik <<EOF
- cluster name = $CLUSTER;
- node 1 admin conninfo = 'dbname=$DBNAME1 host=$HOST1
- port=$PORT1 user=slony password=slony_user_pass';
20 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);
25
- EOF
```

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

Листинг 5.53 Общие задачи

```
Line 1 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, имею-

5.5. Slony-I

щиеся же отваливаются с примерно следующей диагностикой:

Листинг 5.54 Устранение неисправностей

```
Line 1 %slon customers_rep "dbname=customers user=slony_user"
- CONFIG main: slon version 1.0.5 starting up
- CONFIG main: local node id = 3
- CONFIG main: loading current cluster configuration
5 CONFIG storeNode: no_id=1 no_comment='CustomersDB
- replication cluster'
- CONFIG storeNode: no_id=2 no_comment='Node 2,
- node2.example.com'
- CONFIG storeNode: no_id=4 no_comment='Node 4,
10 node4.example.com'
- CONFIG storePath: pa_server=1 pa_client=3
- pa_conninfo="dbname=customers
- host=mainhost.com port=5432 user=slony_user
- password=slony_user_pass" pa_connretry=10
15 CONFIG storeListen: li_origin=1 li_receiver=3
- li_provider=1
- CONFIG storeSet: set_id=1 set_origin=1
- set_comment='CustomersDB replication set'
- WARN remoteWorker_wakeup: node 1 - no worker thread
20 CONFIG storeSubscribe: sub_set=1 sub_provider=1 sub_forward=
'f'
- WARN remoteWorker_wakeup: node 1 - no worker thread
- CONFIG enableSubscription: sub_set=1
- WARN remoteWorker_wakeup: node 1 - no worker thread
- CONFIG main: configuration complete - starting threads
25 CONFIG enableNode: no_id=1
- CONFIG enableNode: no_id=2
- CONFIG enableNode: no_id=4
- ERROR remoteWorkerThread_1: "begin transaction; set
- transaction isolation level
30 serializable; lock table "_customers_rep".sl_config_lock;
select
- "_customers_rep".enableSubscription(1, 1, 4);
- notify "_customers_rep_Event"; notify "
- _customers_rep_Confirm";
- insert into "_customers_rep".sl_event (ev_origin, ev_seqno,
- ev_timestamp, ev_minxid, ev_maxxid, ev_xip,
35 ev_type, ev_data1, ev_data2, ev_data3, ev_data4) values
- ('1', '219440',
- '2005-05-05 18:52:42.708351', '52501283', '52501292',
- ''52501283'', 'ENABLE_SUBSCRIPTION',
- '1', '1', '4', 'f'); insert into "_customers_rep".
40 sl_confirm (con_origin, con_received,
- con_seqno, con_timestamp) values (1, 3, '219440',
```

5.5. Slony-I

```
- CURRENT_TIMESTAMP); commit transaction;"
- PGRES_FATAL_ERROR ERROR: insert or update on table
- "sl_subscribe" violates foreign key
45 constraint "sl_subscribe-sl_path-ref"
- DETAIL: Key (sub_provider,sub_receiver)=(1,4)
- is not present in table "sl_path".
- INFO remoteListenThread_1: disconnecting from
- 'dbname=customers host=mainhost.com
50 port=5432 user=slony_user password=slony_user_pass'
- %
```

Это означает что в служебной таблице `_имя< кластера>.sl_path`, например `_customers_rep.sl_path` на уже имеющихся узлах отсутствует информация о новом узле. В данном случае, id нового узла 4, пара (1,4) в `sl_path` отсутствует. Чтобы это устранить, нужно выполнить на каждом из имеющихся узлов приблизительно следующий запрос:

Листинг 5.55 Устранение неисправностей

```
Line 1 $ 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, то 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 может наблюдаться как со временем растёт нагрузка на master-сервере, в списке активных бекендов — постоянные SELECT-ы со слейвов. В `pg_stat_activity` видны примерно такие запросы:

Листинг 5.56 Устранение неисправностей

```
Line 1 select ev_origin, ev_seqno, ev_timestamp, ev_minxid,
- ev_maxxid, ev_xip,
- ev_type, ev_data1, ev_data2, ev_data3, ev_data4, ev_data5,
- ev_data6,
- ev_data7, ev_data8 from "_customers_rep".sl_event e where
- (e.ev_origin = '2' and e.ev_seqno > '336996') or
5 (e.ev_origin = '3' and e.ev_seqno > '1712871') or
- (e.ev_origin = '4' and e.ev_seqno > '721285') or
- (e.ev_origin = '5' and e.ev_seqno > '807715') or
- (e.ev_origin = '1' and e.ev_seqno > '3544763') or
- (e.ev_origin = '6' and e.ev_seqno > '2529445') or
10 (e.ev_origin = '7' and e.ev_seqno > '2512532') or
- (e.ev_origin = '8' and e.ev_seqno > '2500418') or
```

5.6. Londiste

```
- (e.ev_origin = '10' and e.ev_seqno > '1692318')  
- order by e.ev_origin, e.ev_seqno;
```

где `_customers_rep` — имя схемы из примера. Таблица `sl_event` почему-то разрастается со временем, замедляя выполнение этих запросов до неприемлемого времени. Удаляем ненужные записи:

Листинг 5.57 Устранение неисправностей

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

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

5.6 Londiste

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

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

К недостаткам можно отнести:

- триггерная репликация, что ухудшает производительность базы;

Установка

Установка будет проводиться на Debian сервере. Поскольку Londiste — это часть Skytools, то нам нужно ставить этот пакет:

Листинг 5.58 Установка

```
Line 1 % sudo aptitude install skytools
```

5.6. Londiste

В некоторых системах может содержаться пакет версии 2.x, который не поддерживает каскадную репликацию, отказоустойчивость (failover) и переключение между серверами (switchover). По этой причине он не будет рассматриваться. Скачать самую последнюю версию пакета можно с [официального сайта](#). На момент написания главы последняя версия была 3.2. Начнем установку:

Листинг 5.59 Установка

```
Line 1 $ wget http://pgfoundry.org/frs/download.php/3622/skytools
      -3.2.tar.gz
- $ tar zxvf skytools-3.2.tar.gz
- $ cd skytools-3.2/
- # пакеты для сборки deb
5 $ sudo aptitude install build-essential autoconf \
- automake autotools-dev dh-make \
- debhelper devscripts fakeroot xutils lintian pbuilder \
- python-all-dev python-support xmlto asciidoc \
- libevent-dev libpq-dev libtool
10 # python-psycopg нужен для работы Londiste
- $ sudo aptitude install python-psycopg2 postgresql-server-
      dev-all
- # данной командой собираем deb пакет
- $ make deb
- $ cd ../
15 # ставим skytools
- $ dpkg -i *.deb
```

Для других систем можно собрать Skytools командами:

Листинг 5.60 Установка

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

Далее проверяем правильность установки:

Листинг 5.61 Установка

```
Line 1 $ londiste3 -V
- londiste3, Skytools version 3.2
- $ pgqd -V
- bad switch: usage: pgq-ticker [switches] config.file
5 Switches:
- -v          Increase verbosity
- -q          No output to console
- -d          Daemonize
- -h          Show help
10 -V          Show version
- --ini       Show sample config file
```


5.6. Londiste

- -s Stop - send SIGINT to running process
- -k Kill - send SIGTERM to running process
- -r Reload - send SIGHUP to running process

Настройка

Обозначения:

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

Конфигурация репликаторов

Сначала создается конфигурационный файл для master базы (конфиг будет `/etc/skytools/master-londiste.ini`):

Листинг 5.62 Конфигурация репликаторов

```
Line 1 [londiste3]
- job_name = master_l3simple
- db = dbname=l3simple
- queue_name = replika
5 logfile = /var/log/skytools/master_l3simple.log
- pidfile = /var/pid/skytools/master_l3simple.pid
-
- # Задержка между проверками наличия активности
- # новых( пакетов данных) в секундах
10 loop_delay = 0.5
```

Инициализируем Londiste для master базы:

Листинг 5.63 Инициализируем Londiste

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

5.6. Londiste

```
15 INFO Node "master-node" initialized for queue "replika" with
    type "root"
- INFO Done
```

где `master-server` — это имя провайдера (мастера базы).
Теперь запустим демон:

Листинг 5.64 Запускаем демон для master базы

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

Если нужно перезагрузить демон (например при изменении конфигурации), то можно воспользоваться параметром `-r`:

Листинг 5.65 Перегрузка демона

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

Для остановки демона есть параметр `-s`:

Листинг 5.66 Остановка демона

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

или если потребуется «убить» (`kill`) демон:

Листинг 5.67 Остановка демона

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

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

Листинг 5.68 Демон для ticker

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

Перейдем к `slave` базе. Для начала нужно создать базу данных:

Листинг 5.69 Копирования структуры базы

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

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

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

5.6. Londiste

Листинг 5.70 Создаём конфигурацию для slave

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

Инициализируем Londiste для slave базы:

Листинг 5.71 Инициализируем Londiste для slave

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

Теперь можем запустить демон:

Листинг 5.72 Запускаем демон для slave базы

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

Или же через главный демон:

Листинг 5.73 Запускаем демон для slave базы

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

Создаём конфигурацию для PgQ ticker

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

Листинг 5.74 PgQ ticker конфиг

```
Line 1 [pgqd]
- logfile = /var/log/skytools/pgqd.log
- pidfile = /var/pid/skytools/pgqd.pid
```

Запускаем демон:

Листинг 5.75 Запускаем PgQ ticker

```
Line 1 $ pgqd -d /etc/skytools/pgqd.ini
```

5.6. Londiste

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

Или же через глобальный демон:

Листинг 5.76 Запускаем PgQ ticker

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

Теперь можно увидеть статус кластера:

Листинг 5.77 Статус кластера

```
Line 1 $ londiste3 /etc/skytools/master-londiste.ini status
- Queue: replika Local node: slave1-node
-
- master-node (root)
5 | Tables: 0/0/0
- | Lag: 44s, Tick: 5
- +---: slave1-node (leaf)
- Tables: 0/0/0
- Lag: 44s, Tick: 5
10
- $ londiste3 /etc/skytools/master-londiste.ini members
- Member info on master-node@replika:
- node_name dead node_location
- -----
15 master-node False dbname=l3simple host=
- master-host
- slave1-node False dbname=l3simple host=
- slave1-host
```

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

Листинг 5.78 Добавляем таблицы

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

В данном примере используется параметр `--all`, который означает все таблицы, но вместо него вы можете перечислить список конкретных таблиц, если не хотите реплицировать все. Если имена таблиц отличаются на

5.6. Londiste

master и slave, то можно использовать `--dest-table` параметр при добавлении таблиц на slave базе. Также, если вы не перенесли структуру таблиц заранее с master на slave базы, то это можно сделать автоматически через `--create` параметр (или `--create-full`, если нужно перенести полностью всю схему таблицы).

Подобным образом добавляем последовательности (`sequences`) для репликации:

Листинг 5.79 Добавляем последовательности

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

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

Листинг 5.80 Клонирование структуры базы

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

Далее проверяем состояние репликации:

Листинг 5.81 Статус кластера

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

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

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

Через небольшой интервал времени все таблицы должны синхронизироваться:

5.6. Londiste

Листинг 5.82 Статус кластера

```
Line 1 $ londiste3 /etc/skytools/master-londiste.ini status
- Queue: replika Local node: master-node
-
- master-node (root)
5 | Tables: 4/0/0
- | Lag: 31s, Tick: 20
- +---: slave1-node (leaf)
- Tables: 4/0/0
- Lag: 31s, Tick: 20
```

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

Листинг 5.83 Статус таблиц и последовательностей

```
Line 1 $ londiste3 /etc/skytools/master-londiste.ini tables
- Tables on node
- table_name merge_state table_attrs
- -----
5 public.pgbench_accounts ok
- public.pgbench_branches ok
- public.pgbench_history ok
- public.pgbench_tellers ok
-
10 $ londiste3 /etc/skytools/master-londiste.ini seqs
- Sequences on node
- seq_name local last_value
- -----
- public.pgbench_history_hid_seq True 33345
```

Проверка

Для проверки будем использовать `pgbench` утилиту. Запустим добавление данных в таблицу и смотрим в логи одновременно:

Листинг 5.84 Проверка

```
Line 1 $ pgbench -T 10 -c 5 l3simple
- $ tail -f /var/log/skytools/slave1_l3simple.log
- INFO {count: 1508, duration: 0.307, idle: 0.0026}
- INFO {count: 1572, duration: 0.3085, idle: 0.002}
5 INFO {count: 1600, duration: 0.3086, idle: 0.0026}
- INFO {count: 36, duration: 0.0157, idle: 2.0191}
```

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

5.6. Londiste

Каскадная репликация

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

Листинг 5.85 Конфиг для slave2

```
Line 1 [londiste3]
- job_name = slave2_l3simple
- db = dbname=l3simple host=slave2-host
- queue_name = replika
5 logfile = /var/log/skytools/slave2_l3simple.log
- pidfile = /var/pid/skytools/slave2_l3simple.pid
-
- # Задержка между проверками наличия активности
- #новых( пакетов данных) в секундах
10 loop_delay = 0.5
```

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

Листинг 5.86 Инициализируем slave2

```
Line 1 $ psql -hslave2-host -d postgres -c "CREATE DATABASE
      l3simple;"
- $ pg_dump -s -npublic l3simple | psql -hslave2-host l3simple
- $ londiste3 /etc/skytools/slave2-londiste.ini create-branch
      slave2-node "dbname=l3simple host=slave2-host" --provider
      ="dbname=l3simple host=master-host"
- INFO plpgsql is installed
5 INFO Installing pgq
- INFO Reading from /usr/share/skytools3/pgq.sql
- INFO pgq.get_batch_cursor is installed
- INFO Installing pgq_ext
- INFO Reading from /usr/share/skytools3/pgq_ext.sql
10 INFO Installing pgq_node
- INFO Reading from /usr/share/skytools3/pgq_node.sql
- INFO Installing londiste
- INFO Reading from /usr/share/skytools3/londiste.sql
- INFO londiste.global_add_table is installed
15 INFO Initializing node
- INFO Location registered
- INFO Location registered
- INFO Subscriber registered: slave2-node
- INFO Location registered
20 INFO Location registered
```

5.6. Londiste

- INFO Location registered
- INFO Node "slave2-node" initialized for queue "replika" with type "branch"
- INFO Done

Далее добавляем все таблицы и последовательности:

Листинг 5.87 Инициализируем slave2

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

И запускаем новый демон:

Листинг 5.88 Инициализируем slave2

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

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

Листинг 5.89 Инициализируем slave3 и slave4

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

В результате получаем такую картину с кластером:

Листинг 5.90 Кластер с каскадной репликацией

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


5.6. Londiste

```
-           | Tables: 4/0/0
15          | Lag: 9s, Tick: 49
-          +---: slave4 -node (branch)
-
-           Tables: 4/0/0
-           Lag: 9s, Tick: 49
```

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

Листинг 5.91 Изменяем топологию

```
Line 1 $ londiste3 /etc/skytools/slave4-londiste.ini change-
        provider --provider="dbname=l3simple host=slave2-host"
- $ londiste3 /etc/skytools/slave4-londiste.ini status
- Queue: replika Local node: slave4-node
-
5 master-node (root)
- | Tables: 4/0/0
- | Lag: 12s, Tick: 56
- +---: slave1-node (leaf)
- | Tables: 4/0/0
10 | Lag: 12s, Tick: 56
- +---: slave2-node (branch)
- | Tables: 4/0/0
- | Lag: 12s, Tick: 56
- +---: slave3-node (branch)
15 | Tables: 4/0/0
- | Lag: 12s, Tick: 56
- +---: slave4-node (branch)
- Tables: 4/0/0
- Lag: 12s, Tick: 56
```

Также топологию можно менять с стороны репликатора через команду takeover:

Листинг 5.92 Изменяем топологию

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

5.6. Londiste

```
-      +---: slave3 -node (branch)
15      |                               Tables: 4/0/0
-      |                               Lag: 9s, Tick: 49
-      +---: slave4 -node (branch)
-      |                               Tables: 4/0/0
-      |                               Lag: 9s, Tick: 49
```

Через команду `drop-node` можно удалить slave из кластера:

Листинг 5.93 Удаляем ноду

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

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

Общие задачи

Проверка состояния слейвов

Данный запрос на мастере дает некоторую информацию о каждой очереди и слейве:

Листинг 5.94 Проверка состояния слейвов

```
Line 1 # SELECT queue_name, consumer_name, lag, last_seen FROM pgq.
      get_consumer_info ();
- queue_name | consumer_name | lag |
- last_seen
- --
- -----+-----+-----+-----
- replika    | .global_watermark | 00:03:37.108259 |
- 00:02:33.013915
```

5.6. Londiste

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

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

Удаление очереди всех событий из мастера

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

Листинг 5.95 Удаление очереди всех событий из мастера

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

Добавление столбца в таблицу

Добавляем в следующей последовательности:

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

Удаление столбца из таблицы

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

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

Устранение неисправностей

Londiste пожирает процессор и lag растет

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

5.7. Bucardo

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

Листинг 5.96 Устранение неисправностей

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

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

Листинг 5.97 Устранение неисправностей

```
Line 1 pgq_lazy_fetch = 500
```

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

5.7 Bucardo

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

Установка

Установка будет проводиться на Debian сервере. Сначала нужно установить `DBIx::Safe Perl` модуль.

Листинг 5.98 Установка

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

Для других систем можно поставить из **ИСХОДНИКОВ**:

Листинг 5.99 Установка

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

5.7. Bucardo

Теперь ставим сам Bucardo. **Скачиваем** его и устанавливаем:

Листинг 5.100 Установка

```
Line 1 $ wget http://bucardo.org/downloads/Bucardo-5.4.1.tar.gz
- $ tar xvfz Bucardo-5.4.1.tar.gz
- $ cd Bucardo-5.4.1
- $ perl Makefile.PL
5 $ make
- $ sudo make install
```

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

Листинг 5.101 Установка

```
Line 1 $ sudo aptitude install postgresql-plperl-9.5
```

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

Листинг 5.102 Установка

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

Теперь можем приступать к настройке репликации.

Настройка

Инициализация Bucardo

Запускаем установку Bucardo:

Листинг 5.103 Инициализация Bucardo

```
Line 1 $ bucardo install
```

Во время установки будут показаны настройки подключения к PostgreSQL, которые можно будет изменить:

Листинг 5.104 Инициализация Bucardo

```
Line 1 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
-
5 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>
10 2. Port: 5432
- 3. User: postgres
- 4. Database: postgres
- 5. PID directory: /var/run/bucardo
```

5.7. Bucardo

После подтверждения настроек, Bucardo создаст пользователя `bucardo` и базу данных `bucardo`. Данный пользователь должен иметь право логиниться через Unix socket, поэтому лучше заранее дать ему такие права в `pg_hba.conf`.

После успешной установки можно проверить конфигурацию через команду `bucardo show all`:

Листинг 5.105 Инициализация Bucardo

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

Настройка баз данных

Теперь нужно настроить базы данных, с которыми будет работать Bucardo. Обозначим базы как `master_db` и `slave_db`. Реплицировать будем `simple_database` базу. Сначала настроим мастер базу:

Листинг 5.106 Настройка баз данных

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

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

Дальше добавляем `slave_db`:

Листинг 5.107 Настройка баз данных

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

Настройка репликации

Теперь требуется настроить синхронизацию между этими базами данных. Делается это командой `sync`:

5.7. Bucardo

Листинг 5.108 Настройка репликации

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

Данная команда устанавливает Bucardo триггеры в PostgreSQL для master-slave репликации. Значения параметров:

- **dbs** — список баз данных, которые следует синхронизировать. Значение **source** или **target** указывает, что это master или slave база данных соответственно (их может быть больше одной);
- **conflict_strategy** — для работы в режиме master-master нужно указать как Bucardo должен решать конфликты синхронизации. Существуют следующие стратегии:
 - **bucardo_source** — при конфликте мы копируем данные с source;
 - **bucardo_target** — при конфликте мы копируем данные с target;
 - **bucardo_skip** — конфликт мы просто не реплицируем. Не рекомендуется для продакшен систем;
 - **bucardo_random** — каждая БД имеет одинаковый шанс, что её изменение будет взято для решение конфликта;
 - **bucardo_latest** — запись, которая была последней изменена решает конфликт;
 - **bucardo_abort** — синхронизация прерывается;
- **tables** — таблицы, которые требуется синхронизировать. Через «all» указываем все;

Для master-master репликации требуется выполнить:

Листинг 5.109 Настройка репликации

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

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

Листинг 5.110 Настройка репликации

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

Для проверки состояния репликации:

5.7. Bucardo

Листинг 5.111 Проверка состояния репликации

```
Line 1 $ bucardo status
- PID of Bucardo MCP: 12122
- Name      State      Last good      Time      Last I/D      Last bad
  Time
- =====+=====+=====+=====+=====+=====+=====
5  delta | Good | 13:28:53 | 13m 6s | 3685/7384 | none
  |
```

Запуск/Остановка репликации

Запуск репликации:

Листинг 5.112 Запуск репликации

```
Line 1 $ bucardo start
```

Остановка репликации:

Листинг 5.113 Остановка репликации

```
Line 1 $ bucardo stop
```

Общие задачи

Просмотр значений конфигурации

Листинг 5.114 Просмотр значений конфигурации

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

Изменения значений конфигурации

Листинг 5.115 Изменения значений конфигурации

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

Например:

Листинг 5.116 Изменения значений конфигурации

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

Перегрузка конфигурации

Листинг 5.117 Перегрузка конфигурации

```
Line 1 $ bucardo reload_config
```

Более подробную информацию можно найти на [официальном сайте](#).

Репликация в другие типы баз данных

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

Листинг 5.118 Установка redis

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

Далее регистрируем redis базу в Bucardo:

Листинг 5.119 Добавление redis базы

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

Создадим группу баз данных под названием `pg_to_redis`:

Листинг 5.120 Группа баз данных

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

И создадим репликацию:

Листинг 5.121 Установка sync

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

После перезапуска Bucardo данные с PostgreSQL таблиц начнут реплицироваться в Redis:

Листинг 5.122 Репликация в redis

```
Line 1 $ pgbench -T 10 -c 5 simple_database
- $ redis - cli monitor
- "HMSET" "pgbench_history:6" "bid" "2" "aid" "36291" "delta"
      "3716" "mtime" "2014-07-11 14:59:38.454824" "hid" "4331"
- "HMSET" "pgbench_history:2" "bid" "1" "aid" "65179" "delta"
      "2436" "mtime" "2014-07-11 14:59:38.500896" "hid" "4332"
```

5.7. Bucardo

```
5 "HMSET" "pgbench_history:14" "bid" "2" "aid" "153001" "delta"
  "-264" "mtime" "2014-07-11 14:59:38.472706" "hid" "4333"
- "HMSET" "pgbench_history:15" "bid" "1" "aid" "195747" "delta"
  "-1671" "mtime" "2014-07-11 14:59:38.509839" "hid" "4334"
- "HMSET" "pgbench_history:3" "bid" "2" "aid" "147650" "delta"
  "3237" "mtime" "2014-07-11 14:59:38.489878" "hid" "4335"
- "HMSET" "pgbench_history:15" "bid" "1" "aid" "39521" "delta"
  "-2125" "mtime" "2014-07-11 14:59:38.526317" "hid" "4336"
- "HMSET" "pgbench_history:14" "bid" "2" "aid" "60105" "delta"
  "2555" "mtime" "2014-07-11 14:59:38.616935" "hid" "4337"
10 "HMSET" "pgbench_history:15" "bid" "2" "aid" "186655" "delta"
  "930" "mtime" "2014-07-11 14:59:38.541296" "hid" "4338"
- "HMSET" "pgbench_history:15" "bid" "1" "aid" "101406" "delta"
  "668" "mtime" "2014-07-11 14:59:38.560971" "hid" "4339"
- "HMSET" "pgbench_history:15" "bid" "2" "aid" "126329" "delta"
  "-4236" "mtime" "2014-07-11 14:59:38.5907" "hid" "4340"
- "DEL" "pgbench_tellers:20"
```

Данные в Redis хранятся в виде хешей:

Листинг 5.123 Данные в redis

```
Line 1 $ redis - cli "HGETALL" "pgbench_history:15"
- 1) "bid"
- 2) "2"
- 3) "aid"
5 4) "126329"
- 5) "delta"
- 6) "-4236"
- 7) "mtime"
- 8) "2014-07-11 14:59:38.5907"
10 9) "hid"
- 10) "4340"
```

Также можно проверить состояние репликации:

Листинг 5.124 Установка redis

```
Line 1 $ bucardo status
- PID of Bucardo MCP: 4655
- Name State Last good Time Last I/D
- Last bad Time
- =====|=====|=====|=====|=====|
5 delta | Good | 14:59:39 | 8m 15s | 0/0
  | none |
- pg_to_redis_sync | Good | 14:59:40 | 8m 14s | 646/2546
  | 14:59:39 | 8m 15s
```

Теперь данные из redis могут использоваться для приложения в виде быстрого кэш хранилища.

5.8 Заключение

Репликация — одна из важнейших частей крупных приложений, которые работают на PostgreSQL. Она помогает распределять нагрузку на базу данных, делать фоновый бэкап одной из копий без нагрузки на центральный сервер, создавать отдельный сервер для логирования или аналитики, прочее.

В главе было рассмотрено несколько видов репликации PostgreSQL. Нельзя четко сказать какая лучше всех. Поточковая репликация — один из самых лучших вариантов для поддержки идентичных кластеров баз данных. Slony-I — громоздкая и сложная в настройке система, но имеющая в своем арсенале множество функций, таких как отказоустойчивости (failover) и переключение между серверами (switchover). В тоже время Londiste имея в своем арсенале подобный функционал, может похвастаться еще компактностью и простой в установке. Bucardo — система которая может быть или master-master, или master-slave репликацией.

Шардинг

Если ешь слона, не пытайся
запихать его в рот целиком

Народная мудрость

6.1 Введение

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

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

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

Обычно, в качестве параметра шардинга выбирают ID пользователя (`user_id`) — это позволяет делить данные по серверам равномерно и просто. Т.о. при получении личных сообщений пользователей алгоритм работы будет такой:

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

Задачу определения конкретного сервера можно решать двумя путями:

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

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

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

- [Postgres-XC](#)
- [Greenplum Database](#)
- [Citus](#)
- [PL/Proxy](#)
- [Stado \(sequel to GridSQL\)](#)

6.2 PL/Proxy

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

6.2. PL/Proxy

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

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

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

Недостатки и ограничения:

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

Установка

1. Скачать [PL/Proxy](#) и распаковать;
2. Собрать PL/Proxy командами `make` и `make install`;

Так же можно установить PL/Proxy из репозитория пакетов. Например в Ubuntu Server достаточно выполнить команду для PostgreSQL 9.6:

Листинг 6.1 Установка

```
Line 1 $ sudo aptitude install postgresql-9.6-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

6.2. PL/Proxy

использовать количество нод равное степеням двойки. База данных будет называться `plproxytest`, а таблица в ней — `users`.

Для начала настроим `node1` и `node2`. Команды, написанные ниже, нужно выполнять на каждой ноде. Сначала создадим базу данных `plproxytest` (если её ещё нет):

Листинг 6.2 Настройка

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

Добавляем табличку `users`:

Листинг 6.3 Настройка

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

Теперь создадим функцию для добавления данных в таблицу `users`:

Листинг 6.4 Настройка

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

С настройкой нод закончено. Приступим к серверу проху. Как и на всех нодах, на главном сервере (`proxy`) должна присутствовать база данных:

Листинг 6.5 Настройка

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

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

Листинг 6.6 Настройка

6.2. PL/Proxy

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

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

Листинг 6.7 Настройка

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

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

Листинг 6.8 Настройка

```
Line 1 CREATE OR REPLACE FUNCTION
- public.get_cluster_partitions(cluster_name text)
- RETURNS SETOF text AS
- $BODY$
5 BEGIN
- IF cluster_name = 'usercluster' THEN
- RETURN NEXT 'dbname=plproxytest host=node1 user=postgres
- ';
- RETURN NEXT 'dbname=plproxytest host=node2 user=postgres
- ';
```


6.2. PL/Proxy

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

И последняя:

Листинг 6.9 Настройка

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

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

Листинг 6.10 Настройка

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

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

Листинг 6.11 Настройка

6.2. PL/Proxy

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

Попробуем извлечь данные. Для этого напишем новую серверную функцию:

Листинг 6.12 Настройка

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

И попробуем её вызвать:

Листинг 6.13 Настройка

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

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

Все ли так просто?

Как видно на тестовом примере ничего сложного в работе с pl/проху нет. Но в реальной жизни все не так просто. Представьте что у вас 16 нод. Это же надо как-то синхронизировать код функций. А что если ошибка закрадётся — как её оперативно исправлять?

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

Вторая проблема, которая не дай бог коснётся вас при разработке такого рода системы, это проблема перераспределения данных в тот момент, когда нам захочется добавить ещё нод в кластер. Планировать эту масштабную операцию придётся очень тщательно, подготовив все сервера заранее, занеся данные и потом в один момент подменив код функции `get_cluster_partitions`.

6.3 Postgres-X2

Postgres-X2 – система для создания мульти-мастер кластеров, работающих в синхронном режиме – все узлы всегда содержат актуальные данные. Postgres-X2 поддерживает опции для увеличения масштабирования кластера как при преобладании операций записи, так и при основной нагрузке на чтение данных: поддерживается выполнение транзакций с распараллеливанием на несколько узлов, за целостностью транзакций в пределах всего кластера отвечает специальный узел GTM (Global Transaction Manager).

Измерение производительности показало, что КПД кластера Postgres-X2 составляет примерно 64%, т. е. кластер из 10 серверов позволяет добиться увеличения производительности системы в целом в 6.4 раза, относительно производительности одного сервера (цифры приблизительные).

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

Хоть Postgres-X2 и выглядит похожим на MultiMaster, но он им не является. Все сервера кластера должны быть соединены сетью с минимальными задержками, никакое географически-распределенное решение с разумной производительностью построить на нем невозможно (это важный момент).

Архитектура

Рис. 6.1 показывает архитектуру Postgres-X2 с тремя её основными компонентами:

1. Глобальный менеджер транзакций (GTM) — собирает и обрабатывает информацию о транзакциях в Postgres-X2, решает вопросы глобального идентификатора транзакции по операциям (для поддержания согласованного представления базы данных на всех узлах). Он обеспечивает поддержку других глобальных данных, таких как последовательности и временные метки. Он хранит данные пользователя, за исключением управляющей информации;
2. Координаторы (coordinators) — обеспечивают точку подключения для клиента (приложения). Они несут ответственность за разбор и выполнение запросов от клиентов и возвращение результатов (при

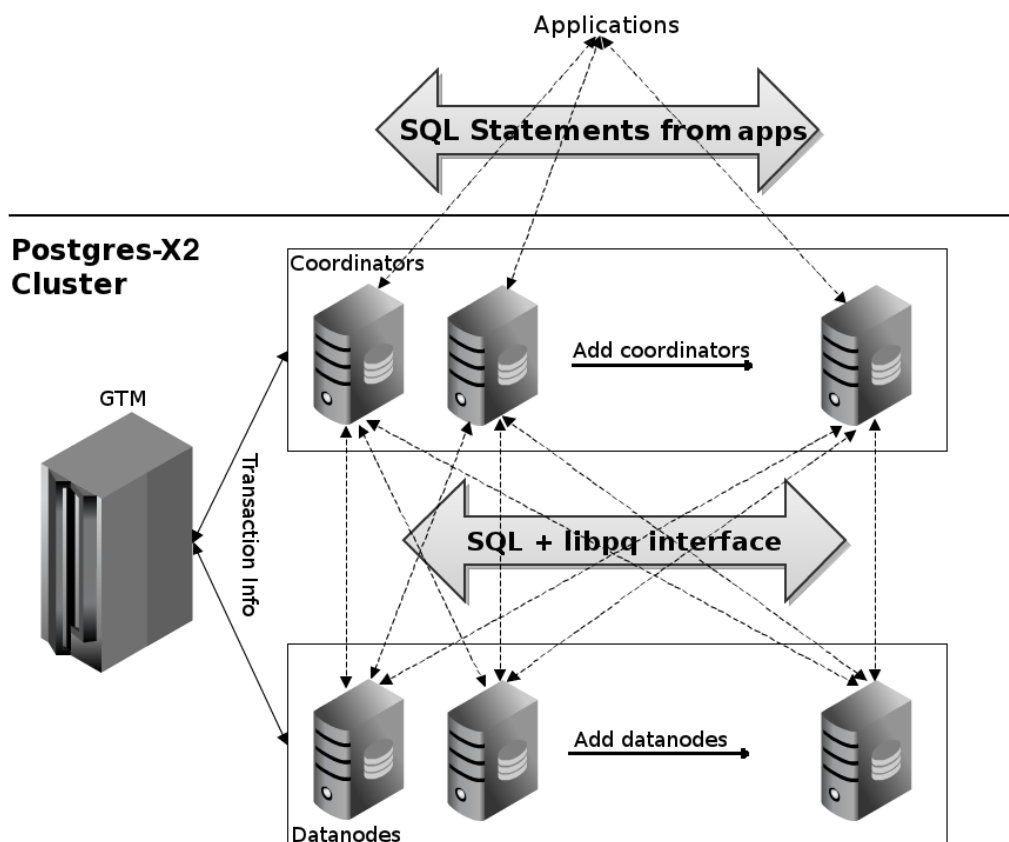


Рис. 6.1: Архитектура Postgres-X2

необходимости). Они не хранят пользовательские данные, а собирают их из обработчиков данных (datanodes) с помощью запросов SQL через PostgreSQL интерфейс. Координаторы также обрабатывают данные, если требуется, и даже управляют двухфазной фиксацией. Координаторы используются также для разбора запросов, составления планов запросов, поиска данных и т.д;

3. Обработчики данных (datanodes) — обеспечивают сохранение пользовательских данных. Datanodes выполняют запросы от координаторов и возвращают им полученный результат;

Установка

Установить Postgres-X2 можно из [исходников](#).

Распределение данных и масштабируемость

Postgres-X2 предусматривает два способа хранения данных в таблицах:

6.3. Postgres-X2

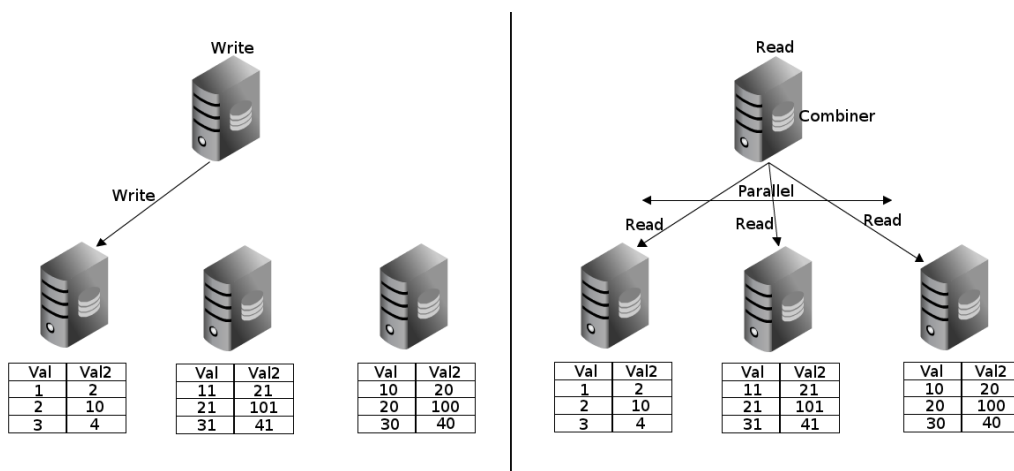


Рис. 6.2: Распределенные таблицы

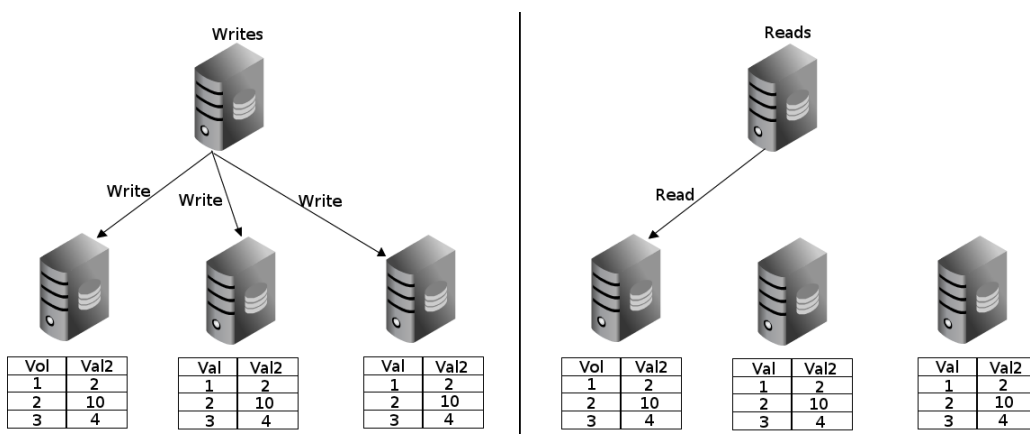


Рис. 6.3: Реплицированные таблицы

1. Распределенные таблицы (distributed tables, рис. 6.2): данные по таблице распределяются на указанный набор обработчиков данных с использованием указанной стратегии (hash, round-robin, modulo). Каждая запись в таблице находится только на одном обработчике данных. Параллельно могут быть записаны или прочитаны данные с различных обработчиков данных. За счет этого значительно улучшена производительность на запись и чтение;
2. Реплицированные таблицы (replicated tables, рис. 6.3): данные по таблице реплицируются (клонироваются) на указанный набор обработчиков данных. Каждая запись в таблице находится на всех обработчиках данных (которые были указаны) и любые изменения дублируются на все обработчики данных. Так как все данные доступны на любом обработчике данных, координатор может собрать все данные из одного узла, что позволяет направить различные запросы на раз-

6.3. Postgres-X2

личные обработчики данных. Таким образом создается балансировка нагрузки и увеличения пропускной способности на чтение;

Таблицы и запросы к ним

После установки работа с Postgres-X2 ведется как с обыкновенным PostgreSQL. Подключаться для работы с данными нужно только к координаторам (по умолчанию координатор работает на порту 5432). Для начала создадим распределенные таблицы:

Листинг 6.14 Создание распределенных таблиц

```
Line 1 CREATE TABLE
- users_with_hash (id SERIAL, type INT, ...)
- DISTRIBUTE by HASH(id);
-
5 CREATE TABLE
- users_with_modulo (id SERIAL, type INT, ...)
- DISTRIBUTE by MODULO(id);
-
- CREATE TABLE
10 users_with_rr robin (id SERIAL, type INT, ...)
- DISTRIBUTE by ROUNDROBIN;
```

На листинге 6.14 создано 3 распределенные таблицы:

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

Листинг 6.15 Данные с координатора и обработчиков данных

```
Line 1 # координатор
- $ psql
- # SELECT id, type from users_with_hash ORDER BY id;
- id | type
5 ---+-----
- 1 | 946
- 2 | 153
- 3 | 484
- 4 | 422
10 5 | 785
- 6 | 906
- 7 | 973
- 8 | 699
- 9 | 434
15 10 | 986
- 11 | 135
- 12 | 1012
```

6.3. Postgres-X2

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

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

Листинг 6.16 Данные с координатора и обработчиков данных

```
Line 1 # координатор
- $ psql
- # SELECT id, type from users_with_modulo ORDER BY id;
- id  | type
5  ---+-----
-    1 |    883
-    2 |    719
-    3 |     29
-    4 |    638
10   5 |    363
-    6 |    946
-    7 |    440
```

6.3. Postgres-X2

```
-      8 |      331
-      9 |      884
15     10 |      199
-     11 |       78
-     12 |      791
-     13 |      345
-     14 |      476
20     15 |      860
-
- # первый обработчик данных
- $ psql -p15432
- # SELECT id, type from users_with_modulo ORDER BY id;
25  id   | type
-  ----+-----
-     2 |   719
-     4 |   638
-     6 |   946
30     8 |   331
-    10 |   199
-    12 |   791
-    14 |   476
-
-
35 # второй обработчик данных
- $ psql -p15433
- # SELECT id, type from users_with_modulo ORDER BY id;
-   id  | type
-  ----+-----
40     1 |   883
-     3 |    29
-     5 |   363
-     7 |   440
-     9 |   884
45    11 |    78
-    13 |   345
-    15 |   860
```

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

Листинг 6.17 Данные с координатора и обработчиков данных

```
Line 1 # координатор
- $ psql
- # SELECT id, type from users_with_rrabin ORDER BY id;
-   id  | type
5  ----+-----
-     1 |   890
-     2 |   198
```


6.3. Postgres-X2

```
-      3 |      815
-      4 |      446
10     5 |        61
-      6 |      337
-      7 |      948
-      8 |      446
-      9 |      796
15     10 |      422
-     11 |      242
-     12 |      795
-     13 |      314
-     14 |      240
20     15 |      733
-
- # первый обработчик данных
- $ psql -p15432
- # SELECT id, type from users_with_rrubin ORDER BY id;
25  id   | type
-  ----+-----
-     2 |   198
-     4 |   446
-     6 |   337
30     8 |   446
-    10 |   422
-    12 |   795
-    14 |   240
-
- # второй обработчик данных
- $ psql -p15433
- # SELECT id, type from users_with_rrubin ORDER BY id;
-  id   | type
-  ----+-----
40     1 |   890
-     3 |   815
-     5 |    61
-     7 |   948
-     9 |   796
45    11 |   242
-    13 |   314
-    15 |   733
```

Теперь создадим реплицированную таблицу:

Листинг 6.18 Создание реплицированной таблицы

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

6.3. Postgres-X2

Естественно данные идентичны на всех обработчиках данных:

Листинг 6.19 Данные с координатора и обработчиков данных

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

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

Листинг 6.20 Выборка записей из распределенной таблицы

```
Line 1 # EXPLAIN VERBOSE SELECT * from users_with_modulo ORDER BY
-      id;
-
-      QUERY PLAN
-
-      ---
-
-      Sort (cost=49.83..52.33 rows=1000 width=8)
5      Output: id, type
-      Sort Key: users_with_modulo.id
-      -> Result (cost=0.00..0.00 rows=1000 width=8)
-      Output: id, type
-      -> Data Node Scan on users_with_modulo (cost
10      =0.00..0.00 rows=1000 width=8)
-      Output: id, type
-      Node/s: dn1, dn2
-      Remote query: SELECT id, type FROM ONLY
-      users_with_modulo WHERE true
-      (9 rows)
```

Как видно на листинге 6.20 координатор собирает данные из обработчиков данных, а потом собирает их вместе.

Подсчет суммы с группировкой по полю из распределенной таблицы:

6.3. Postgres-X2

Листинг 6.21 Выборка записей из распределенной таблицы

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

JOIN между и с участием реплицированных таблиц, а также JOIN между распределенными по одному и тому же полю в таблицах будет выполняются на обработчиках данных. Но JOIN с участием распределенных таблиц по другим ключам будут выполнены на координаторе и скорее всего это будет медленно (листинг 6.22).

Листинг 6.22 Выборка записей из распределенной таблицы

```
Line 1 # EXPLAIN VERBOSE SELECT * from users_with_modulo ,
      users_with_hash WHERE users_with_modulo.id =
      users_with_hash.id;
-
-          QUERY PLAN
-
-  --
-  -----
-
- Nested Loop (cost=0.00..0.01 rows=1 width=16)
5   Output: users_with_modulo.id, users_with_modulo.type,
      users_with_hash.id, users_with_hash.type
-   Join Filter: (users_with_modulo.id = users_with_hash.id)
-   -> Data Node Scan on users_with_modulo (cost=0.00..0.00
      rows=1000 width=8)
-       Output: users_with_modulo.id, users_with_modulo.
      type
-       Node/s: dn1, dn2
```

6.3. Postgres-X2

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

Пример выборки данных из реплицированной таблицы:

Листинг 6.23 Выборка записей из реплицированной таблицы

```
Line 1 # EXPLAIN VERBOSE SELECT * from users_replicated;
- QUERY PLAN
- --
- -----
- Data Node Scan on "__REMOTE_FQS_QUERY__" (cost=0.00..0.00
   rows=0 width=0)
5 Output: users_replicated.id, users_replicated.type
- Node/s: dn1
- Remote query: SELECT id, type FROM users_replicated
- (4 rows)
```

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

Высокая доступность (HA)

По архитектуре у Postgres-X2 всегда есть согласованность данных. По **теореме CAP** в такой системе тяжело обеспечить высокую доступность. Для достижения высокой доступности в распределенных системах требуется избыточность данных, резервные копии и автоматическое восстановление. В Postgres-X2 избыточность данных может быть достигнута с помощью PostgreSQL потоковой (streaming) репликации с hot-standby для обработчиков данных. Каждый координатор способен записывать и читать данные независимо от другого, поэтому координаторы способны заменять друг друга. Поскольку GTM отдельный процесс и может стать точкой отказа, лучше создать GTM-standby как резервную копию. Ну а вот для автоматического восстановления придется использовать сторонние утилиты.

Ограничения

1. Postgres-X2 базируется на PostgreSQL 9.3;

6.4. Postgres-XL

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

Заключение

Postgres-X2 очень перспективное решение для создания кластера на основе PostgreSQL. И хоть это решение имеет ряд недостатков, нестабильно (очень часты случаи падения координаторов при тяжелых запросах) и еще очень молодое, со временем это решение может стать стандартом для масштабирования систем на PostgreSQL.

6.4 Postgres-XL

Postgres-XL – система для создания мульти-мастер кластеров, работающих в синхронном режиме – все узлы всегда содержат актуальные данные. Проект построен на основе кодовой базы Postgres-X2, поэтому архитектурный подход полностью идентичен (глобальный менеджер транзакций (GTM), координаторы (coordinators) и обработчики данных (datanodes)). Более подробно про архитектуру можно почитать в «[6.3 Архитектура](#)» разделе. Поэтому рассмотрим только отличие Postgres-X2 и Postgres-XL.

Postgres-X2 и Postgres-XL

Одно из главных отличий Postgres-XL от Postgres-X2 является улучшенный механизм массово-параллельной архитектуры (massive parallel processing, MPP). Чтобы понять разницу, давайте рассмотрим как Postgres-X2 и Postgres-XL будет обрабатывать разные SQL запросы. Оба этих кластера будут содержать три таблицы **T1**, **T2** и **R1**. **T1** имеет колонки **a1** и **a2**, **T2** — **b1** и **b2**. **T1** распределена в кластере по **a1** полю и **T2** распределена по **b1** полю. **R1** таблица имеет колонки **c1** и **c2** и реплицируется в кластере (`DISTRIBUTE by REPLICATION`).

Для начала, простой запрос вида `SELECT * FROM T1` будет параллельно выполняться на нодах как у Postgres-X2, так и у Postgres-XL. Другой пример запроса `SELECT * FROM T1 INNER JOIN R1 ON T1.a1 = R1.c1` будет также выполняться параллельно обоими кластерами, потому что будет

передан («pushed down») на обработчики данных (datanodes) для выполнения и координатор (coordinators) будет только агрегировать (собирать) результаты запросов. Это будет работать благодаря тому, что **R1** таблица дублируется на каждом обработчике данных. Этот тип запросов будет работать хорошо, когда **T1** является **таблицей фактов** (основной таблицей хранилища данных), в то время как **R1** — **таблицей измерений** (содержит атрибуты событий, сохраненных в таблице фактов).

Теперь рассмотрим другой вариант SQL запроса:

Листинг 6.24 Запрос на распределенные таблицы

```
Line 1 # SELECT * FROM T1 INNER JOIN T2 ON T1.a1 = T2.b2
```

Данный запрос делает **JOIN** по распределенной колонке **a1** в таблице **T1** и по **НЕ** распределенной колонке **b2** в таблице **T2**. В кластере, который состоит из 4 обработчиков данных, колонка в таблице **T1** на первом из них потенциально требуется объединить с колонками таблицы **T2** на всех обработчиках данных в кластере.

У Postgres-X2 в данном случае обработчики данных отправляют все данные по заданному условию в запросе к координатору, который и занимается объединением данных с таблиц. В данном примере отсутствует условие **WHERE**, что значит, что все обработчики данных отправят все содержимое таблиц **T1** и **T2** на координатор, который и будет делать **JOIN** данных. В данной операции будет отсутствовать параллельное выполнение **JOIN** запроса и будут дополнительные накладные расходы на доставку всех данных к координатору. Поэтому в данном случае Postgres-X2 фактически будет медленнее, чем выполнение подобного запроса на обычном PostgreSQL сервере (особенно, если таблицы очень большие).

Postgres-XL будет обрабатывать подобный запрос по-другому. Условие **T1.a1 = T2.b2** говорит о том, что мы объединяем колонку **b2** с колонкой **a1**, которая является ключом распределения для таблицы **T1**. Поэтому, выбрав значения поля **b2**, кластер будет точно знать для каких обработчиков данных требуется полученный результат для объединения с таблицей **T1** (поскольку возможно применить хеш функцию распределения на полученные значения). Поэтому каждый обработчик данных считает с другого обработчика данных требуемые данные по таблице **T2** для объединения со своей таблицей **T1** без участия координатора. Данная возможность прямой коммуникации обработчиков данных с другими обработчиками данных позволяет распараллеливать более сложные запросы в Postgres-XL.

Postgres-XL имеет также другие улучшения производительности (более оптимально обрабатываются последовательности, прочее).

Заключение

Postgres-XL - еще одно перспективное решение для создания кластера на основе Postgres-X2. Разработчики данного решения больше нацелены

на улучшение производительности и стабильности кластера, вместо добавления нового функционала.

6.5 Citus

Citus — горизонтально масштабируемый PostgreSQL кластер. Citus использует механизм расширений PostgreSQL вместо того, что бы использовать модифицированную версию базы (как это делает «6.3 Postgres-X2» или «6.4 Postgres-XL»), что позволяет использовать новые версии PostgreSQL с новыми возможностями, сохраняя при этом совместимость с существующими PostgreSQL инструментами. Кластер предоставляет пользователям результаты запросов в режиме «реального времени» для большого и растущего объема данных (благодаря параллелизации запросов между нодами). Примеры использования:

- аналитика и вывод данных в реальном времени на графики;
- хранение большого набора данных для архива и создание отчетов по ним;
- анализ и сегментация большого объема данных;

Нагрузки, которые требуют большой поток данных между узлами кластера, как правило, не будет хорошо работать с Citus кластером. Например:

- традиционные хранилища данных с длительными и в свободном формате SQL запросами (data warehousing);
- множественные распределенные транзакции между несколькими шардами;
- запросы, которые возвращают данные по тяжелым **ETL** запросам;

Архитектура

На верхнем уровне Citus кластер распределяет данные по PostgreSQL экземплярам. Входящие SQL запросы затем обрабатываются параллельно через эти сервера.

При разворачивании кластера один из экземпляров PostgreSQL выбирается в качестве мастер (master) ноды. Затем остальные добавляются как PostgreSQL воркеры (worker) в конфигурационном файле мастер ноды. После этого все взаимодействие с кластером ведется через мастер ноду с помощью стандартных PostgreSQL интерфейсов. Все данные распределены по воркерам. Мастер хранит только метаданные о воркерах.

Citus использует модульную архитектуру для блоков данных, которая похожа на **HDFS** (Hadoop Distributed File System), но использует PostgreSQL таблицы вместо файлов. Каждая из этих таблиц представляет собой горизонтальный раздел или логический «шард» (shard). Каждый

6.5. Citus

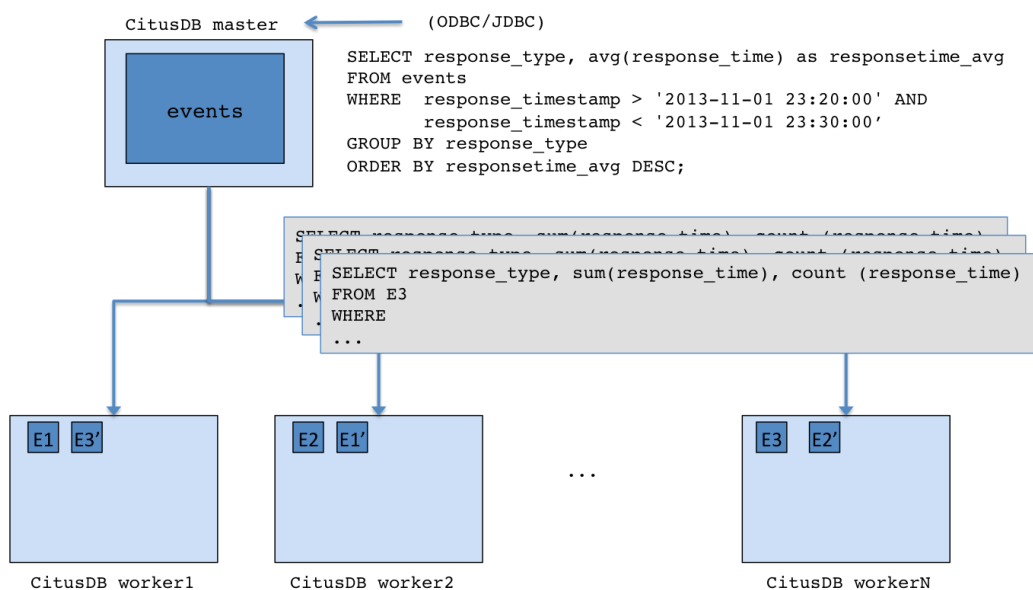


Рис. 6.4: Архитектура Citus кластера

шард дублируется, по крайней мере, на двух воркерах (можно настроить на более высокое значение). В результате, потеря одной машины не влияет на доступность данных. Логическая архитектура шардинга в Citus также позволяет добавлять новые воркеры, чтобы увеличить пропускную способность и вычислительную мощность кластера.

Citus мастер содержит таблицы метаданных для отслеживания всех воркеров и расположение шардов базы данных на них. Эти таблицы также ведут статистику, такую как размер и минимальное/максимальное значений в шардах, которые помогают распределению SQL запросов Citus планировщику. Таблицы метаданных небольшие (обычно несколько мегабайт), и могут быть дублированы и быстро восстановлены, если с мастером когда-либо произойдет сбой. Подробнее о таблицах метаданных можно глянуть [в документации](#).

Когда кластер получает SQL запрос, Citus мастер делит его на более мелкие фрагменты запросов, где каждый фрагмент может выполняться независимо на воркере. Это позволяет Citus распределять каждый запрос в кластере, используя вычислительные мощности всех задействованных узлов, а также отдельных ядер на каждом узле. Мастер затем поручает воркерам выполнить запрос, осуществляет контроль за их исполнением, объединяет результаты по запросам и возвращает конечный результат пользователю. Для того, чтобы гарантировать, что все запросы выполняются в масштабируемой манере, мастер также применяет оптимизации, которые сводят к минимуму объем данных, передаваемых по сети.

Citus кластер может легко переносить свои воркеры из-за своей логи-

6.5. Citus

ческой шардинг архитектуры. Если воркер терпит неудачу во время выполнения запроса, Citus завершает запрос, направляя неудачные части запроса другим воркерам, которые имеют копию данных. Если воркер находится в нерабочем состоянии (сервер упал), пользователь может легко произвести ребалансировку кластера, чтобы поддерживать тот же уровень доступности.

Установка

Установка Citus кластера не требует особых усилий. Для использования в боевом окружении лучше изучить [данную документацию](#). Для проверки, что кластер работает и мастер видит воркеры можно выполнить команду `master_get_active_worker_nodes`, которая покажет список воркеров:

Листинг 6.25 Список воркеров

```
Line 1 postgres=# select * from master_get_active_worker_nodes();
-   node_name | node_port
-   -+-----
-   localhost |         9702
5  localhost |         9701
- (2 rows)
```

Распределенные таблицы

Каждая распределенная таблица в Citus содержит столбец, который должен быть выбран в качестве значения для распределения по шардам (возможно выбрать только один столбец). Это информирует базу данных как хранить статистику и распределять запросы по кластеру. Как правило, требуется выбрать столбец, который является наиболее часто используемым в запросах **WHERE**. В таком случае запросы, которые в фильтре используют данный столбец, будут выполняться на шардах, которые выбираются по условию фильтрации. Это помогает значительно уменьшить количество вычислений на шардах.

Следующим шагом после выбора столбца на распределения будет определение правильного метода распределения данных в таблицу. В целом, существует два шаблона таблиц: распределенные по времени (время создания заказа, запись логов, прочее) и распределение по идентификатору (ID пользователя, ID приложения, прочее). Citus поддерживает оба метода распределения: `append` и `hash` соответственно.

`Append` метод подходит для таблиц, в которые записываются данные по времени (упорядочены по времени). Такой тип таблиц отлично справляется с запросами, которые используют фильтры с диапазонами значений по распределенному столбцу (**BETWEEN x AND y**). Это объясняется тем, что

6.5. Citus

мастер хранит диапазоны значений, которые хранятся на шардах, и планировщик может эффективно выбирать шарды, которые содержат данные для SQL запроса.

Hash метод распределения подходит для неупорядоченного столбца (user UUID) или по данным, которые могут записываться в любом порядке. В таком случае Citus кластер будет хранить минимальные и максимальные значения для хеш функций на всех шардах. Эта модель лучше подходит для SQL запросов, включающих фильтры на основе равенства по колонке распределения (`user_uuid='a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11'`).

Hash распределение

Для примера создадим и распределим таблицу по hash методу.

Листинг 6.26 Создание таблицы

```
Line 1 # CREATE TABLE github_events
- (
-     event_id bigint ,
-     event_type text ,
5     event_public boolean ,
-     repo_id bigint ,
-     payload jsonb ,
-     repo jsonb ,
-     actor jsonb ,
10    org jsonb ,
-     created_at timestamp
- );
```

Далее укажем Citus кластеру использовать `repo_id` с hash распределением для `github_events` таблицы.

Листинг 6.27 Создание hash распределения

```
Line 1 # SELECT master_create_distributed_table('github_events', '
      repo_id', 'hash');
```

И создадим шарды для таблицы:

Листинг 6.28 Создание шардов

```
Line 1 # SELECT master_create_worker_shards('github_events', 16, 1)
      ;
```

Данный метод принимает два аргумента в дополнение к имени таблицы: количество шардов и коэффициент репликации. Этот пример позволит создать в общей сложности шестнадцать шардов, где каждый будет владеть частью символического пространства хэша, а данные будут реплицироваться на один воркер.

Далее мы можем заполнить таблицу данными:

6.5. Citus

Листинг 6.29 Загрузка данных

```
Line 1 $ wget http://examples.citusdata.com/github_archive/
        github_events-2015-01-01-{0..5}.csv.gz
- $ gzip -d github_events-2015-01-01-*.gz
```

Листинг 6.30 Загрузка данных

```
Line 1 # \COPY github_events FROM 'github_events-2015-01-01-0.csv'
        WITH (format CSV)
- # INSERT INTO github_events VALUES (2489373118, 'PublicEvent',
        't', 24509048, '{}', '{"id": 24509048, "url": "https://api.
        github.com/repos/SabinaS/csee6868", "name": "SabinaS/
        csee6868"}', '{"id": 2955009, "url": "https://api.github.
        com/users/SabinaS", "login": "SabinaS", "avatar_url": "
        https://avatars.githubusercontent.com/u/2955009?", "
        gravatar_id": ""}', NULL, '2015-01-01 00:09:13');
```

Теперь мы можем обновлять и удалять данные с таблицы:

Листинг 6.31 Изменение данных

```
Line 1 # UPDATE github_events SET org = NULL WHERE repo_id =
        24509048;
- # DELETE FROM github_events WHERE repo_id = 24509048;
```

Для работы `UPDATE` и `DELETE` запросов требуется, что бы он «затрагивал» один шард. Это означает, что условие `WHERE` должно содержать условие, что ограничит выполнение запроса на один шард по распределенному столбцу. Для обновления или удаления данных на нескольких шардах требуется использовать команду `master_modify_multiple_shards`:

Листинг 6.32 Изменение данных на нескольких шардах

```
Line 1 # SELECT master_modify_multiple_shards(
- 'DELETE FROM github_events WHERE repo_id IN (24509048,
        24509049)');
```

Для удаления таблицы достаточно выполнить `DROP TABLE` на мастере:

Листинг 6.33 Удаление таблицы

```
Line 1 # DROP TABLE github_events;
```

Append распределение

Для примера создадим и распределим таблицу по `append` методу.

Листинг 6.34 Создание таблицы

```
Line 1 # CREATE TABLE github_events
- (
```

6.5. Citus

```
-     event_id bigint ,
-     event_type text ,
5     event_public boolean ,
-     repo_id bigint ,
-     payload jsonb ,
-     repo jsonb ,
-     actor jsonb ,
10    org jsonb ,
-     created_at timestamp
- );
```

Далее укажем Citus кластеру использовать `created_at` с `append` распределением для `github_events` таблицы.

Листинг 6.35 Создание hash распределения

```
Line 1 # SELECT master_create_distributed_table('github_events', '
        created_at', 'append');
```

После этого мы можем использовать таблицу и загружать в нее данные:

Листинг 6.36 Загрузка данных

```
Line 1 # SET citus.shard_max_size TO '64MB';
- # \copy github_events from 'github_events-2015-01-01-0.csv'
    WITH (format CSV)
```

По умолчанию команда `\copy` требует два конфигурационных параметра для работы: `citus.shard_max_size` и `citus.shard_replication_factor`.

- `citus.shard_max_size` параметр указывает максимальный размер шарда при использовании команды `\copy` (1Гб по умолчанию). Если файл больше данного параметра, то команда автоматически разобьет файл по нескольким шардам;
- `citus.shard_replication_factor` параметр количество воркеров, на которые шарды будут реплицироваться (2 по умолчанию);

По умолчанию команда `\copy` создает каждый раз новый шард для данных. Если требуется добавлять данные в один и тот же шард, существуют команды `master_create_empty_shard`, которая вернет идентификатор на новый шард, и команда `master_append_table_to_shard` для добавления данных в этот шард по идентификатору.

Для удаления старых данных можно использовать команду `master_apply_delete_command`, которая удаляет старые шарды, которые попадают в переданное условие на удаление:

Листинг 6.37 Удаление старых шардов

```
Line 1 # SELECT * from master_apply_delete_command('DELETE FROM
        github_events WHERE created_at >= ''2015-01-01 00:00:00''
        ');
```

```
- master_apply_delete_command
- -----
-                               3
5 (1 row)
```

Для удаления таблицы достаточно выполнить `DROP TABLE` на мастере:

Листинг 6.38 Удаление таблицы

```
Line 1 # DROP TABLE github_events;
```

Ребалансировка кластера

Логическая архитектура шардинга Citus позволяет масштабировать кластер без каких-либо простоев (no downtime!). Для добавления нового воркера достаточно добавить его в `pg_worker_list.conf` и вызвать на мастере `pg_reload_conf` для загрузки новой конфигурации:

Листинг 6.39 Загрузка новой конфигурации

```
Line 1 # SELECT pg_reload_conf();
```

После этого Citus автоматически начнет использовать данный воркер для новых распределенных таблиц. Если требуется ребалансировать существующие таблицы на новый воркер, то для этого есть команда `rebalance_table_shards`, но, к сожалению, она доступна только в Citus Enterprise (платное решение).

Ограничения

Модель расширения PostgreSQL в Citus позволяет использовать доступные типы данных (JSON, JSONB, другие) и другие расширения в кластере. Но не весь спектр SQL запросов доступен для распределенных таблиц. На текущий момент распределенные таблицы не поддерживают:

- Оконные функции (window functions);
- Общие табличные выражения (CTE);
- `UNION` операции (`UNION/INTERSECT/EXCEPT`);
- Транзакционная семантика для запросов, которые распределены по нескольким шардам;

Заключение

Citus кластер достаточно гибкое и мощное решение для горизонтального масштабирования PostgreSQL. Зрелость данного решения показывает его использование такими игроками на рынке, как CloudFlare, Herp и многими другими.

6.6 Greenplum Database

Greenplum Database (GP) — реляционная СУБД, имеющая массово-параллельную (massive parallel processing) архитектуру без разделения ресурсов (shared nothing). Для подробного понимания принципов работы Greenplum необходимо обозначить основные термины:

- Master instance («мастер») — инстанс PostgreSQL, являющийся одновременно координатором и входной точкой для пользователей в кластере;
- Master host («сервер-мастер») — сервер, на котором работает master instance;
- Secondary master instance — инстанс PostgreSQL, являющийся резервным мастером, включается в работу в случае недоступности основного мастера (переключение происходит вручную);
- Primary segment instance («сегмент») — инстанс PostgreSQL, являющийся одним из сегментов. Именно сегменты непосредственно хранят данные, выполняют с ними операции и отдают результаты мастеру (в общем случае). По сути сегмент — самый обычный инстанс PostgreSQL 8.3.23 с настроенной репликацией в своё зеркало на другом сервере;
- Mirror segment instance («зеркало») — инстанс PostgreSQL, являющийся зеркалом одного из primary сегментов, автоматически принимает на себя роль primary в случае падения одного. Greenplum поддерживает только 1-to-1 репликацию сегментов: для каждого из primary может быть только одно зеркало;
- Segment host («сервер-сегмент») — сервер, на котором работает один или несколько сегментов и/или зеркал;

В общем случае кластер GP состоит из нескольких серверов-сегментов, одного сервера-мастера, и одного сервера-секундари-мастера, соединённых между собой одной или несколькими быстрыми (10g, infiniband) сетями, обычно обособленными (interconnect) (рис 6.5).

Использование нескольких interconnect-сетей позволяет, во-первых, повысить пропускную способность канала взаимодействия сегментов между собой, и во-вторых, обеспечить отказоустойчивость кластера (в случае отказа одной из сетей весь трафик перераспределяется между оставшимися).

При выборе числа серверов-сегментов важно правильно выбрать соотношение кластера «число процессоров/Тб данных» в зависимости от планируемого профиля нагрузки на БД — чем больше процессорных ядер приходится на единицу данных, тем быстрее кластер будет выполнять «тяжёлые» операции, а также работать со сжатыми таблицами.

При выборе числа сегментов в кластере (которое в общем случае к числу серверов никак не привязано) необходимо помнить следующее:

6.6. Greenplum Database

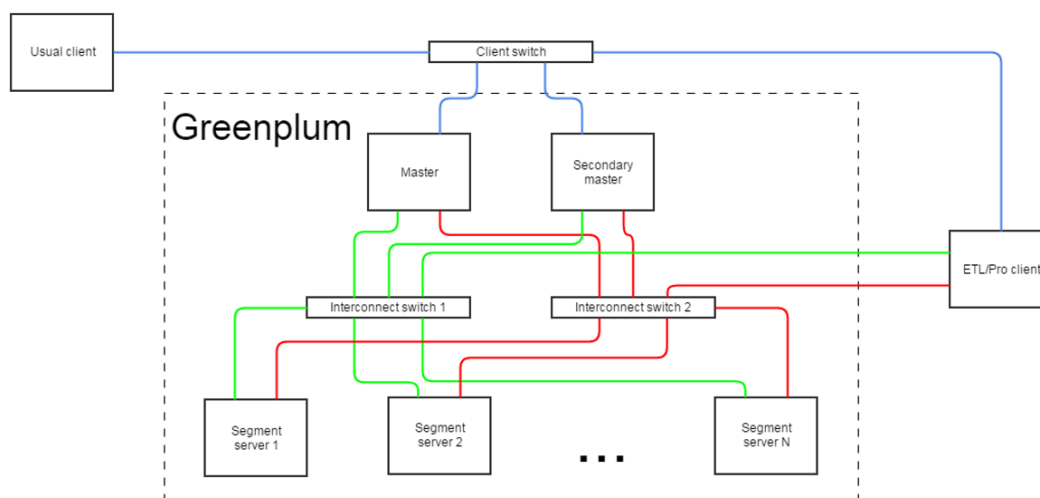


Рис. 6.5: Состав кластера и сетевое взаимодействие элементов. Зелёная и красная линии — обособленные сети interconnect, синяя линия — внешняя, клиентская сеть

- все ресурсы сервера делятся между всеми сегментами на сервере (нагрузкой зеркал, в случае если они располагаются на этих же серверах, можно условно пренебречь);
- каждый запрос на одном сегменте не может потреблять процессорных ресурсов больше, чем одно ядро CPU. Это означает, например, что, если кластер состоит из 32-ядерных серверов с 4-я сегментами GP на борту и используется в среднем для обработки 3-4 одновременных тяжёлых, хорошо утилизирующих CPU, запросов, «в среднем по больнице» CPU не будет утилизироваться оптимально. В данной ситуации лучше увеличить число сегментов на сервере до 6-8;
- штатный процесс бэкапа и рестора данных «из коробки» работает только на кластерах, имеющих одинаковое число сегментов. Восстановить данные, забэкапленные на кластере из 96 сегментов, в кластер из 100 сегментов без напильника будет невозможно;

Хранение данных

В Greenplum реализуется классическая схема шардирования данных. Каждая таблица представляет из себя $N+1$ таблиц на всех сегментах кластера, где N — число сегментов (+1 в этом случае — это таблица на мастере, данных в ней нет). На каждом сегменте хранится $1/N$ строк таблицы. Логика разбиения таблицы на сегменты задаётся ключом (полем) дистрибуции — таким полем, на основе данных которого любую строку можно отнести к одному из сегментов.

Ключ (поле или набор полей) дистрибуции — очень важное понятие в GP. Как было сказано выше, Greenplum работает со скоростью самого мед-

ленного сегмента, это означает, что любой перекоc в количестве данных (как в рамках одной таблицы, так и в рамках всей базы) между сегментами ведёт к деградации производительности кластера, а также к другим проблемам. Именно поэтому следует тщательно выбирать поле для дистрибуции — распределение количества вхождений значений в нём должно быть как можно более равномерным. Правильно ли вы выбрали ключ дистрибуции вам подскажет служебное поле `gp_segment_id`, существующее в каждой таблице — оно содержит номер сегмента, на котором хранится конкретная строка. Важный нюанс: GP не поддерживает `UPDATE` поля, по которому распределена таблица.

Рассмотрим пример (здесь и далее в примерах кластер состоит из 96 сегментов):

Листинг 6.40 Создание распределенной таблицы

```

Line 1 db=# create table distrib_test_table as select
        generate_series(1,20) as num_field distributed by (
            num_field);
- SELECT 20
- db=# select count(1),gp_segment_id from distrib_test_table
        group by gp_segment_id order by gp_segment_id;
-  count | gp_segment_id
5  -----+-----
-      1 |              4
-      1 |              6
-      1 |             15
-      1 |             21
10     1 |             23
-      1 |             25
-      1 |             31
-      1 |             40
-      1 |             42
15     1 |             48
-      1 |             50
-      1 |             52
-      1 |             65
-      1 |             67
20     1 |             73
-      1 |             75
-      1 |             77
-      1 |             90
-      1 |             92
25     1 |             94
-
- db=# truncate table distrib_test_table;
- TRUNCATE TABLE
- db=# insert into distrib_test_table values (1), (1), (1),
        (1), (1), (1), (1), (1), (1), (1), (1), (1),

```



```

    (1), (1), (1), (1), (1), (1);
30 INSERT 0 20
- db=# select count(1),gp_segment_id from distrib_test_table
      group by gp_segment_id order by gp_segment_id;
- count | gp_segment_id
- -----+-----
-      20 |                42

```

В обоих случаях распределена таблица по полю `num_field`. В первом случае вставили в это поле 20 уникальных значений, и, как видно, GP разложил все строки на разные сегменты. Во втором случае в поле было вставлено 20 одинаковых значений, и все строки были помещены на один сегмент.

В случае, если в таблице нет подходящих полей для использования в качестве ключа дистрибуции, можно воспользоваться случайной дистрибуцией (`DISTRIBUTED RANDOMLY`). Поле для дистрибуции можно менять в уже созданной таблице, однако после этого её необходимо перераспределить. Именно по полю дистрибуции Greenplum совершает самые оптимальные `JOIN`: в случае, если в обеих таблицах поля, по которым совершается `JOIN`, являются ключами дистрибуции, `JOIN` выполняется локально на сегменте. Если же это условие не верно, GP придётся или перераспределить обе таблицы по искомому полю, или закинуть одну из таблиц целиком на каждый сегмент (операция `BROADCAST`) и уже затем джойнить таблицы локально на сегментах.

Листинг 6.41 JOIN по ключу дистрибуции

```

Line 1 db=# create table distrib_test_table as select
      generate_series(1,192) as num_field, generate_series
      (1,192) as num_field_2 distributed by (num_field);
- SELECT 192
- db=# create table distrib_test_table_2 as select
      generate_series(1,1000) as num_field, generate_series
      (1,1000) as num_field_2 distributed by (num_field);
- SELECT 1000
5 db=# explain select * from distrib_test_table sq
- db=# left join distrib_test_table_2 sq2
- db=# on sq.num_field = sq2.num_field;
- QUERY PLAN
- --
-----
10 Gather Motion 96:1 (slice1; segments: 96) (cost
    =20.37..42.90 rows=861 width=16)
-   -> Hash Left Join (cost=20.37..42.90 rows=9 width=16)
-       Hash Cond: sq.num_field = sq2.num_field
-       -> Seq Scan on distrib_test_table sq (cost
    =0.00..9.61 rows=9 width=8)

```

```
-          -> Hash (cost=9.61..9.61 rows=9 width=8)
15          -> Seq Scan on distrib_test_table_2 sq2 (
           cost=0.00..9.61 rows=9 width=8)
```

Листинг 6.42 JOIN не по ключу дистрибуции

```
Line 1 db_dev=# explain select * from distrib_test_table sq left
        join distrib_test_table_2 sq2
-      on sq.num_field_2 = sq2.num_field_2;
-
-                                             QUERY PLAN
-
-  -----
5      Gather Motion 96:1 (slice3; segments: 96) (cost
      =37.59..77.34 rows=861 width=16)
-      -> Hash Left Join (cost=37.59..77.34 rows=9 width=16)
-          Hash Cond: sq.num_field_2 = sq2.num_field_2
-          -> Redistribute Motion 96:96 (slice1; segments:
      96) (cost=0.00..26.83 rows=9 width=8)
-              Hash Key: sq.num_field_2
10             -> Seq Scan on distrib_test_table sq (cost
      =0.00..9.61 rows=9 width=8)
-                 -> Hash (cost=26.83..26.83 rows=9 width=8)
-                     -> Redistribute Motion 96:96 (slice2;
      segments: 96) (cost=0.00..26.83 rows=9 width=8)
-                         Hash Key: sq2.num_field_2
-                             -> Seq Scan on distrib_test_table_2
      sq2 (cost=0.00..9.61 rows=9 width=8)
```

Как видно в примере «**JOIN не по ключу дистрибуции**» в плане запроса появляются два дополнительных шага (по одному для каждой из участвующих в запросе таблиц): **Redistribute Motion**. По сути, перед выполнением запроса GP перераспределяет обе таблицы по сегментам, используя логику поля `num_field_2`, а не изначального ключа дистрибуции — поля `num_field`.

Взаимодействие с клиентами

В общем случае всё взаимодействие клиентов с кластером ведётся только через мастер — именно он отвечает клиентам, выдаёт им результат запроса и т. д. Обычные клиенты не имеют сетевого доступа к серверам-сегментам.

Для ускорения загрузки данных в кластер используется **bulk load** — параллельная загрузка данных с/на клиент одновременно с нескольких сегментов. Bulk load возможен только с клиентов, имеющих доступ в интернет-контакты. Обычно в роли таких клиентов выступают ETL-сервера и другие системы, которым необходима загрузка большого объёма данных (на рис 6.5 они обозначены как ETL/Pro client).

6.6. Greenplum Database

Для параллельной загрузки данных на сегменты используется утилита `gpfdist`. По сути, утилита поднимает на удалённом сервере web-сервер, который предоставляет доступ по протоколам `gpfdist` и `http` к указанной папке. После запуска директория и все файлы в ней становятся доступны обычным `wget`. Создадим для примера файл в директории, обслуживаемой `gpfdist`, и обратимся к нему как к обычной таблице.

Листинг 6.43 Пример с `gpfdist`

```
Line 1 # На ETLсервере -:
- bash# for i in {1..1000}; do echo "$i,$(cat /dev/urandom |
  tr -dc 'a-zA-Z0-9' | fold -w 8 | head -n 1)"; done > /tmp
  /work/gpfdist_home/test_table.csv
-
- # Теперь создадим внешнюю таблицу и прочитаем данные из файла
5 # В Greenplum DB:
- db=# create external table ext_test_table
- db=# (id integer, rand varchar(8))
- db=# location ('gpfdist://etl_hostname:8081/test_table.csv')
- db=# format 'TEXT' (delimiter ',', ' NULL ');
10 CREATE EXTERNAL TABLE
- db_dev=# select * from ext_test_table limit 100;
- NOTICE: External scan from gpfdist(s) server will utilize
  64 out of 96 segment databases
- id | rand
- ----+-----
15  1 | UWlonJHO
-   2 | HTyJNA41
-   3 | CBP1Q$n1
-   4 | 0K9y51a3
-   ...
```

Также, но с немного другим синтаксисом, создаются внешние web-таблицы. Их особенность заключается в том, что они ссылаются на `http` протокол, и могут работать с данными, предоставляемыми сторонними web-серверами (`apache`, `nginx` и другие).

В Greenplum также существует возможность создавать внешние таблицы на данные, лежащие на распределённой файловой системе Hadoop (`hdfs`) — за это в GP ответственна отдельная компонента `gphdfs`. Для обеспечения её работы на каждый сервер, входящий в состав кластера GP, необходимо установить библиотеки Hadoop и прописать к ним путь в одной из системных переменных базы. Создание внешней таблицы, обращающейся к данным на `hdfs`, будет выглядеть примерно так:

Листинг 6.44 Пример с `gphdfs`

```
Line 1 db=# create external table hdfs_test_table
- db=# (id int, rand text)
```

6.6. Greenplum Database

- `db=# location ('gp_hdfs://hadoop_name_node:8020/tmp/test_file.csv')`
- `db=# format 'TEXT' (delimiter ',',');`

где `hadoop_name_node` — адрес хоста ноды, `/tmp/test_file.csv` — путь до искомого файла на hdfs.

При обращении к такой таблице Greenplum выясняет у ноды Hadoop расположение нужных блоков данных на датанодах, к которым затем обращается с серверов-сегментов параллельно. Естественно, все ноды кластера Hadoop должны быть в сетях интерконнекта кластера Greenplum. Такая схема работы позволяет достичь значительного прироста скорости даже по сравнению с `gpfdist`. Что интересно, логика выбора сегментов для чтения данных с датанод hdfs является весьма нетривиальной. Например, GP может начать тянуть данные со всех датанод только двумя сегмент-серверами, причём при повторном аналогичном запросе схема взаимодействия может поменяться.

Также есть тип внешних таблиц, которые ссылаются на файлы на сегмент-серверах или файл на мастере, а также на результат выполнения команды на сегмент-серверах или на мастере. К слову сказать, старый добрый `COPY FROM` никуда не делся и также может использоваться, однако по сравнению с описанным выше работает он медленней.

Надёжность и резервирование

Резервирование мастера

Как было сказано ранее, в кластере GP используется полное резервирование мастера с помощью механизма репликации транзакционных логов, контролируемого специальным агентом (`gpsyncagent`). При этом автоматическое переключение роли мастера на резервный инстанс не поддерживается. Для переключения на резервный мастер необходимо:

- убедиться, что основной мастер остановлен (процесс убит и в рабочей директории инстанса мастера отсутствует файл `postmaster.pid`);
- на сервере резервного мастера выполнить команду `gpactivatestandby -d /master_instance_directory`;
- переключить виртуальный ip-адрес на сервер нового мастера (механизм виртуального ip в Greenplum отсутствует, необходимо использовать сторонние инструменты);

Как видно, переключение выполняется совсем не сложно и при принятии определённых рисков может быть автоматизировано.

Резервирование сегментов

Схема резервирования сегментов похожа на таковую для мастера, отличия совсем небольшие. В случае падения одного из сегментов (инстанс

6.6. Greenplum Database

PostgreSQL перестаёт отвечать мастеру в течении таймаута) сегмент помечается как сбойный, и вместо него автоматически запускается его зеркало (по сути, абсолютно аналогичный инстанс PostgreSQL). Репликация данных сегмента в его зеркало происходит на основе кастомной синхронной репликации на уровне файлов.

Стоит отметить, что довольно важное место в процессе планирования архитектуры кластера GP занимает вопрос расположения зеркал сегментов на серверах, благо GP даёт полную свободу в вопросе выбора мест расположения сегментов и их зеркал: с помощью специальной карты расположения сегментов их можно разместить на разных серверах, в разных директориях и заставить использовать разные порты. Рассмотрим два варианта:

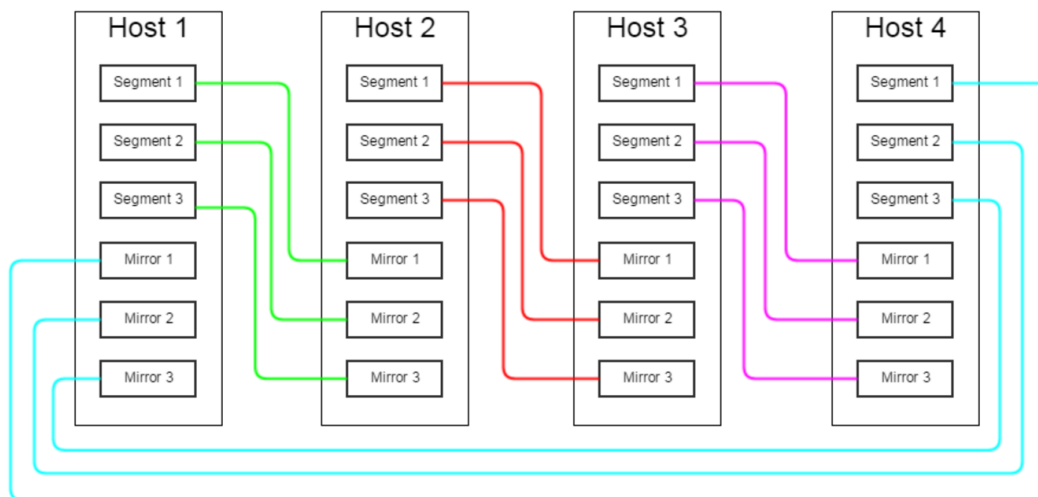


Рис. 6.6: Все зеркала сегментов, располагающихся на хосте N , находятся на хосте $N+1$

При использовании схемы 6.6 при отказе одного из серверов на сервере-соседе оказывается в два раза больше работающих сегментов. Как было сказано выше, производительность кластера равняется производительности самого медленного из сегментов, а значит, в случае отказа одного сервера производительность базы снижается минимум вдвое. Однако, такая схема имеет и положительные стороны: при работе с отказавшим сервером уязвимым местом кластера становится только один сервер — тот самый, куда переехали сегменты.

При использовании схемы 6.7 в случае отказа сервера возросшая нагрузка равномерно распределяется между несколькими серверами, не сильно влияя на общую производительность кластера. Однако, существенно повышается риск выхода из строя всего кластера — достаточно выйти из строя одному из M серверов, соседствующих с вышедшим из строя изначально.

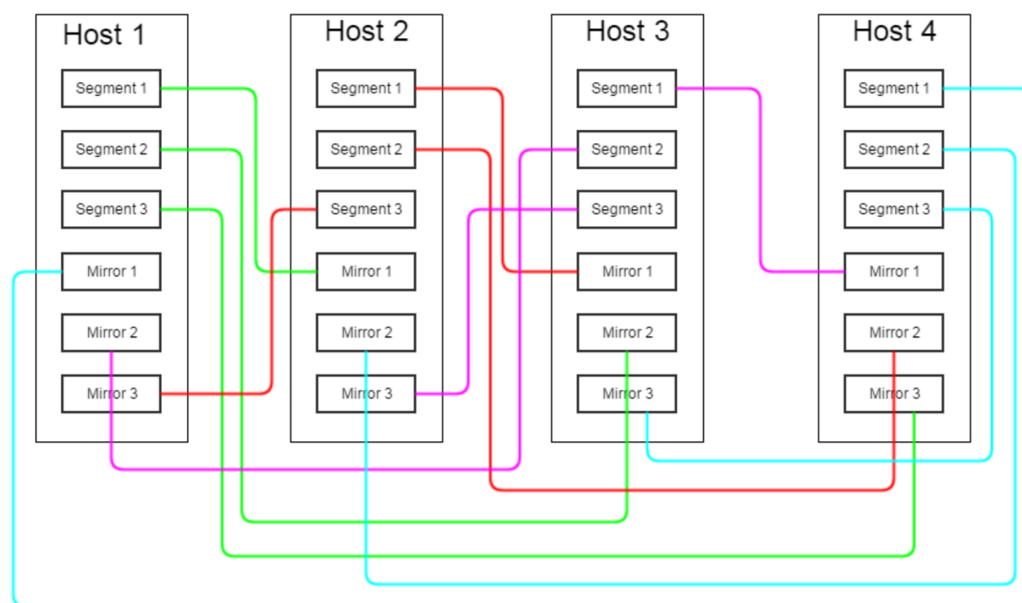


Рис. 6.7: Все зеркала сегментов, располагающихся на хосте N , равномерно «мажутся» на сервера $N+1$, $N+2$... $N+M$, где M – число сегментов на сервере

Истина, как это часто бывает, где-то посередине — можно расположить по несколько зеркал сегментов одного сервера на нескольких других серверах, можно объединять сервера в группы отказоустойчивости, и так далее. Оптимальную конфигурацию зеркал следует подбирать исходя из конкретных аппаратных данных кластера, критичности простоя и так далее.

Также в механизме резервирования сегментов есть ещё один нюанс, влияющий на производительность кластера. В случае выхода из строя зеркала одного из сегментов последний переходит в режим [change tracking](#) — сегмент логирует все изменения, чтобы затем при восстановлении упавшего зеркала применить их к нему, и получить свежую, консистентную копию данных. Другими словами, при падении зеркала нагрузка, создаваемая на дисковую подсистему сервера сегментом, оставшимся без зеркала, существенно возрастает.

При устранении причины отказа сегмента (аппаратные проблемы, кончившееся место на устройстве хранения и прочее) его необходимо вернуть в работу вручную, с помощью специальной утилиты [gprecoverseg](#) (даунтайм СУБД не требуется). По факту эта утилита скопирует скопившиеся на сегменте WA-логи на зеркало и поднимет упавший сегмент/зеркало. В случае, если речь идёт о `primary`-сегменте, изначально он включится в работу как зеркало для своего зеркала, ставшего `primary` (зеркало и основной сегмент будут работать поменявшись ролями). Для того, чтобы вернуть всё на круги своя, потребуется процедура ребаланса — смены ролей. Такая

процедура также не требует даунтайма СУБД, однако на время ребаланса все сессии в БД подвиснут.

В случае, если повреждения упавшего сегмента настолько серьёзны, что простым копированием данных из WA-логов не обойтись, есть возможность использовать полное восстановление упавшего сегмента — в таком случае, по факту, инстанс PostgreSQL будет создан заново, однако за счёт того, что восстановление будет не инкрементальным, процесс восстановления может занять продолжительное время.

Производительность

Оценка производительности кластера Greenplum – понятие довольно растяжимое. Исходные данные: кластер из 24 сегмент-серверов, каждый сервер — 192 Гб памяти, 40 ядер. Число primary-сегментов в кластере: 96. В первом примере мы создаём таблицу с 4-я полями + первичный ключ по одному из полей. Затем мы наполняем таблицу данными (10 000 000 строк) и пробуем выполнить простой **SELECT** с несколькими условиями.

Листинг 6.45 SELECT с условиями

```

Line 1 db=# CREATE TABLE test3
- db=# (id bigint NOT NULL,
- db=# profile bigint NOT NULL,
- db=# status integer NOT NULL,
5 db=# switch_date timestamp without time zone NOT NULL,
- db=# CONSTRAINT test3_id_pkey PRIMARY KEY (id) )
- db=# distributed by (id);
- NOTICE: CREATE TABLE / PRIMARY KEY will create implicit
index "test3_pkey" for table "test3"
- CREATE TABLE
10
- db=# insert into test3 (id , profile ,status , switch_date)
select a, round(random()*100000), round(random()*4), now
() - '1 year'::interval * round(random() * 40) from
generate_series(1,10000000) a;
- INSERT 0 10000000
-
- db=# explain analyze select profile , count(status) from
test3
15 db=# where status <> 2
- db=# and switch_date between '
1970-01-01' and '2015-01-01' group by profile;
-
- Gather Motion 96:1 (slice2; segments: 96) (cost
=2092.80..2092.93 rows=10 width=16)
- Rows out: 100001 rows at destination with 141 ms to first
row, 169 ms to end, start offset by 0.778 ms.

```

6.6. Greenplum Database

```
20 -> HashAggregate (cost=2092.80..2092.93 rows=1 width=16)
-   Group By: test3.profile
-   Rows out: Avg 1041.7 rows x 96 workers. Max 1061 rows (
-   seg20) with 141 ms to end, start offset by 2.281 ms.
-   Executor memory: 4233K bytes avg, 4233K bytes max (seg0).
-   -> Redistribute Motion 96:96 (slice1; segments: 96) (cost
-   =2092.45..2092.65 rows=1 width=16)
25   Hash Key: test3.profile
-   Rows out: Avg 53770.2 rows x 96 workers at destination
-   . Max 54896 rows (seg20) with 71 ms to first row, 117 ms
-   to end, start offset by 5.205 ms.
-   -> HashAggregate (cost=2092.45..2092.45 rows=1 width
-   =16)
-   Group By: test3.profile
-   Rows out: Avg 53770.2 rows x 96 workers. Max 54020
-   rows (seg69) with 71 ms to first row, 90 ms to end, start
-   offset by 7.014 ms.
30   Executor memory: 7882K bytes avg, 7882K bytes max (
-   seg0).
-   -> Seq Scan on test3 (cost=0.00..2087.04 rows=12 width
-   =12)
-   Filter: status <> 2 AND switch_date >= '1970-01-01
-   00:00:00'::timestamp without time zone AND switch_date <=
-   '2015-01-01 00:00:00'::timestamp without time zone
-   Rows out: Avg 77155.1 rows x 96 workers. Max 77743
-   rows (seg26) with 0.092 ms to first row, 31 ms to end,
-   start offset by 7.881 ms.
-   Slice statistics:
35 (slice0) Executor memory: 364K bytes.
-   (slice1) Executor memory: 9675K bytes avg x 96 workers, 9675
-   K bytes max (seg0).
-   (slice2) Executor memory: 4526K bytes avg x 96 workers, 4526
-   K bytes max (seg0).
-   Statement statistics:
-   Memory used: 128000K bytes
40 Total runtime: 175.859 ms
```

Как видно, время выполнения запроса составило 175 мс. Теперь попробуем пример с джойном по ключу дистрибуции одной таблицы и по обычному полю другой таблицы.

Листинг 6.46 JOIN по ключу дистрибуции одной таблицы и по обычному полю другой

```
Line 1 db=# create table test3_1 (id bigint NOT NULL, name text ,
-   CONSTRAINT test3_1_id_pkey PRIMARY KEY (id)) distributed
-   by (id);
-   NOTICE: CREATE TABLE / PRIMARY KEY will create implicit
-   index "test3_1_pkey" for table "test3_1"
```


6.6. Greenplum Database

```
- CREATE TABLE
- db=# insert into test3_1 (id , name) select a, md5(random()
  ::text) from generate_series(1,100000) a;
5 INSERT 0 100000
- db=# explain analyze select test3.*,test3_1.name from test3
  join test3_1 on test3.profile=test3_1.id;
-
- -> Hash Join (cost=34.52..5099.48 rows=1128 width=60)
-   Hash Cond: test3.profile = test3_1.id
10   Rows out: Avg 104166.2 rows x 96 workers. Max 106093 rows
  (seg20) with 7.644 ms to first row, 103 ms to end, start
  offset by 223 ms.
-   Executor memory: 74K bytes avg, 75K bytes max (seg20).
-   Work_mem used: 74K bytes avg, 75K bytes max (seg20).
-   Workfile: (0 spilling, 0 reused)
-   (seg20) Hash chain length 1.0 avg, 1 max, using 1061 of
  262151 buckets.
-   -> Redistribute Motion 96:96 (slice1; segments: 96) (cost
  =0.00..3440.64 rows=1128 width=28)
15   Hash Key: test3.profile
-   Rows out: Avg 104166.7 rows x 96 workers at
  destination. Max 106093 rows (seg20) with 3.160 ms to
  first row, 44 ms to end, start offset by 228 ms.
-   -> Seq Scan on test3 (cost=0.00..1274.88 rows=1128
  width=28)
-   Rows out: Avg 104166.7 rows x 96 workers. Max 104209
  rows (seg66) with 0.165 ms to first row, 16 ms to end,
  start offset by 228 ms.
-   -> Hash (cost=17.01..17.01 rows=15 width=40)
20   Rows in: Avg 1041.7 rows x 96 workers. Max 1061 rows (
  seg20) with 1.059 ms to end, start offset by 227 ms.
-   -> Seq Scan on test3_1 (cost=0.00..17.01 rows=15 width
  =40)
-   Rows out: Avg 1041.7 rows x 96 workers. Max 1061
  rows (seg20) with 0.126 ms to first row, 0.498 ms to end,
  start offset by 227 ms.
-   Slice statistics:
-   (slice0) Executor memory: 364K bytes.
25 (slice1) Executor memory: 1805K bytes avg x 96 workers, 1805
  K bytes max (seg0).
-   (slice2) Executor memory: 4710K bytes avg x 96 workers, 4710
  K bytes max (seg0). Work_mem: 75K bytes max.
-   Statement statistics:
-   Memory used: 128000K bytes
-   Total runtime: 4526.065 ms
```

Время выполнения запроса составило 4.6 секунды. Много это или мало для такого объёма данных — вопрос спорный и лежащий вне этой книги.

Расширение кластера

В жизненном цикле распределённой аналитической БД рано или поздно возникает ситуация, когда объём доступного дискового пространства уже не может вместить всех необходимых данных, а добавление устройств хранения в имеющиеся сервера либо невозможна, либо слишком дорога и сложна (потребуется, как минимум, расширение существующих разделов). Кроме того, добавление одних лишь дисковых мощностей негативно скажется на соотношении «число процессоров/Тб данных», о котором мы говорили в «6.6 Хранение данных». Говоря простым языком, в кластер рано или поздно понадобится вводить новые сервера. Greenplum позволяет добавлять как новые сервера, так и новые сегменты практически без простоя СУБД. Последовательность этого действия примерно такая:

- разработать карту сегментов, согласно которой GP будет размещать новые сегменты и зеркала на новых серверах;
- сделать бэкап необходимых критичных данных (как минимум всех метаданных);
- установить ПО СУБД на новые сервера;
- остановить СУБД (следующий пункт выполняется в даунтайм);
- инициализировать новые сегменты утилитой `gpexpand` (занимает от 5 до 10 минут);
- поднять СУБД (даунтайм окончен);
- перераспределить (`redistribute`) все таблицы;
- собрать статистику (`analyze`) по всем таблицам;

Как видно, хотя процедура расширения и продолжительна, полная недоступность БД при правильных действиях администратора не превысит 20-30 минут.

Особенности эксплуатации

Как обычно, практика вносит в красивую теорию свои коррективы. Поделюсь некоторыми нюансами эксплуатации, выявленными нами за долгое время использования GP. Сразу оговорюсь, что стандартные нюансы PostgreSQL (необходимость `VACUUM`, особенности репликации) в этот перечень не попали:

- Автоматический failover не даёт 100% гарантии переключения на зеркало. Увы, но это так, особенно под нагрузкой, есть риск зависания процессов базы при попытке переключения на зеркало. Частично проблему решает уменьшение таймаута ответа от сегментов до нескольких минут, однако даже в таком случае риск остаётся. Как частное решение проблемы зависания при переключении можно использовать ручное убийство зависшего сегмента или перезагрузку базы;

6.7. Заключение

- Greenplum и OLTP несовместимы. GP — аналитическая БД, предназначенная для небольшого числа одновременных запросов, выполняющих тяжёлые операции над большим объёмом данных. Большое число (более 600 queries per second) лёгких запросов/транзакций, выполняющих одну операцию, негативно сказывается на производительности базы из-за её распределённой архитектуры — каждая транзакция на мастере порождает N транзакций на сегментах. Хорошей практикой является агрегация большого числа `UPDATE/INSERT` в батчи;
- Отсутствие механизма инкрементального бэкапа;
- Свой синтаксис. Несмотря на то, что для клиента Greenplum по сути является PostgreSQL DB, небольшие различия в синтаксисе SQL заставляют использовать стандартный клиентский PostgreSQL-софт с большой осторожностью;
- Отсутствие возможности пометить сегменты как «архивные». Частично этот недостаток можно решить путём использования архивных партиций, находящихся на медленном дешевом tablespace, а также с помощью появившейся в последней на момент написания главы версии GP 4.3.6.0 возможности располагать партиции таблицы во внешних источниках (например, внешних таблицах `gphdfs`, лежащих в кластере Hadoop);
- Greenplum использует PostgreSQL версии 8.3.23, а значит о многих современных плюшках этой замечательной БД придётся забыть;

Заключение

Greenplum — мощный и гибкий инструмент для аналитической обработки больших объёмов данных. Он требует к себе немного другого подхода, чем остальные enterprise-level решения для Data Warehouse («напильник» — любимый инструмент администратора GP). Однако при достаточно низком пороге вхождения и большой унифицированности с PostgreSQL Greenplum является сильным игроком на поле Data Warehouse DB.

6.7 Заключение

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

PgPool-II

Имеется способ сделать
лучше — найди его

Томас Алва Эдисон

7.1 Введение

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

- Объединение соединений

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

- Репликация

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

- Балансировка нагрузки

Если база данных реплицируется, то выполнение запроса `SELECT` на любом из серверов вернет одинаковый результат. pgpool-II использует преимущество функции репликации для уменьшения нагрузки на каждый из серверов PostgreSQL распределяя запросы `SELECT` на несколько серверов, тем самым увеличивая производительность системы в целом. В лучшем случае производительность возрастает

7.2. Установка и настройка

пропорционально числу серверов PostgreSQL. Балансировка нагрузки лучше всего работает в случае когда много пользователей выполняют много запросов в одно и то же время.

- Ограничение лишних соединений

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

- Параллельные запросы

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

`Pgpool-II` общается по протоколу бэкенда и фронтенда PostgreSQL и располагается между ними. Таким образом, приложение базы данных считает что `pgpool-II` — настоящий сервер PostgreSQL, а сервер 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.

7.2 Установка и настройка

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

Листинг 7.1 Установка `pgpool-II`

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

Настройка

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`:

Листинг 7.2 Файлы конфигурации

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

7.2. Установка и настройка

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

Листинг 7.3 Файлы конфигурации

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

Настройка команд PСР

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

Листинг 7.4 Настройка команд PСР

```
Line 1 postgres:e8a48653851e28c69d0506508fb27fc5
```

Для того чтобы зашифровать пароль в формате md5 хэша используется команда `pg_md5`, которая устанавливается как один из исполняемых файлов pgpool-II. `pg_md5` принимает текст в параметре командной строки и отображает md5 хэш как результат.

Листинг 7.5 Настройка команд PСР

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

Команды PСР выполняются по сети, так что в файле `pgpool.conf` должен быть указан номер порта в параметре `psr_port`:

Листинг 7.6 Настройка команд PСР

```
Line 1 psr_port = 9898
```

Подготовка узлов баз данных

Далее требуется 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 данном примере три сервера PostgreSQL размещено в рамках одного хоста вместе с pgpool-II (5432, 5433, 5434 порты соответственно):

7.3. Настройка репликации

Листинг 7.7 Подготовка узлов баз данных

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

В параметрах `backend_hostname`, `backend_port`, `backend_weight` указывается 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. Параметры `backend_weight` все равны 1, что означает что запросы `SELECT` равномерно распределены по трем серверам.

7.3 Настройка репликации

Pgpool-II репликация включает копирование одних и тех же данных на множество узлов базы данных (синхронная репликация). Но данная 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).

Настройка репликации

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

Листинг 7.8 Настройка репликации

```
Line 1 replication_mode = true
```

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

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

Листинг 7.9 Настройка репликации

```
Line 1 load_balance_mode = true
```

7.4. Параллельное выполнение запросов

Проверка репликации

После настройки `pgpool.conf` и перезапуска `pgpool-II`, можно проверить репликацию в действии. Для этого создадим базу данных, которую будем реплицировать (базу данных нужно создать на всех узлах):

Листинг 7.10 Проверка репликации

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

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

Листинг 7.11 Проверка репликации

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

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

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

Для проверки указанной выше информации на всех узлах используем простой скрипт на shell:

Листинг 7.12 Проверка репликации

```
Line 1 for port in 5432 5433 5434; do
- >     echo $port
- >     for table_name in branches tellers accounts history;
      do
- >         echo $table_name
5 >         psql -c "SELECT count(*) FROM $table_name" -p \
- >         $port bench_replication
- >     done
- > done
```

7.4 Параллельное выполнение запросов

`Pgpool-II` позволяет использовать распределение для таблиц. Данные из разных диапазонов сохраняются на двух или более узлах базы данных параллельным запросом. Более того, одни и те же данные на двух и более узлах базы данных могут быть воспроизведены с использованием распределения.

7.4. Параллельное выполнение запросов

Чтобы включить параллельные запросы в pgpool-II вы должны установить еще одну базу данных, называемую «системной базой данных» («System Database») (далее будет называться SystemDB). SystemDB хранит определяемые пользователем правила, определяющие какие данные будут сохраняться на каких узлах базы данных. Также SystemDB используется чтобы объединить результаты возвращенные узлами базы данных посредством dblink.

Настройка

Чтобы включить функцию выполнения параллельных запросов требуется установить для параметра `parallel_mode` значение `true` в файле `pgpool.conf`:

Листинг 7.13 Настройка параллельного запроса

```
Line 1 parallel_mode = true
```

Установка параметра `parallel_mode` равным `true` не запустит параллельные запросы автоматически. Для этого pgpool-II нужна SystemDB и правила определяющие как распределять данные по узлам базы данных. Также SystemDB использует dblink для создания соединений с pgpool-II. Таким образом, нужно установить значение параметра `listen_addresses` таким образом чтобы pgpool-II принимал эти соединения:

Листинг 7.14 Настройка параллельного запроса

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

Нужно обратить внимание, что репликация не реализована для таблиц, которые распределяются посредством параллельных запросов. Поэтому:

Листинг 7.15 Настройка параллельного запроса

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

или

Листинг 7.16 Настройка параллельного запроса

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

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

7.4. Параллельное выполнение запросов

данных может хранить системную базу данных, а `pgpool-II` может использоваться для распределения нагрузки каскадным подключением.

Создадим `SystemDB` на узле с портом 5432. Далее приведен список параметров конфигурации для `SystemDB`:

Листинг 7.17 Настройка SystemDB

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

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

Листинг 7.18 Настройка SystemDB

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

Установка dblink

Далее требуется установить `dblink` в базу данных «`pgpool`». `Dblink` — один из инструментов включенных в каталог `contrib` исходного кода PostgreSQL. После того как `dblink` был установлен в вашей системе мы добавим функции `dblink` в базу данных «`pgpool`».

Листинг 7.19 Установка dblink

```
Line 1 $ psql -c "CREATE EXTENSION dblink;" -p 5432 pgpool
```

Создание таблицы dist_def

Следующим шагом мы создадим таблицу с именем `dist_def`, в которой будут храниться правила распределения данных. Поскольку `pgpool-II` уже был установлен, файл с именем `system_db.sql` должен быть установлен в `/usr/local/share/system_db.sql` (имейте в виду, что на вашей системе каталог установки может быть другой). Файл `system_db.sql` содержит директивы для создания специальных таблиц, включая и таблицу `dist_def`. Выполним следующую команду для создания таблицы `dist_def`:

Листинг 7.20 Создание таблицы dist_def

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


7.4. Параллельное выполнение запросов

```
- dbname text , -- имя базы данных
- schema_name text , -- имя схемы
- table_name text , -- имя таблицы
5 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 данном примере будут определены 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`. Будем использоваться этот файл с примером для создания правил распределения для `pgbench`. Выполним следующую команду в каталоге с распакованным исходным кодом `pgpool-II`:

Листинг 7.23 Установка правил распределения данных

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

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

Листинг 7.24 Установка правил распределения данных

```
Line 1 INSERT INTO pgpool_catalog.dist_def VALUES (
-     'bench_parallel',
-     'public',
-     'accounts',
5     'aid',
-     ARRAY['aid', 'bid', 'abalance', 'filler'],
-     ARRAY['integer', 'integer', 'integer',
-     'character(84)'],
-     'pgpool_catalog.dist_def_accounts'
10 );
```

Теперь мы должны создать функцию распределения данных для таблицы `accounts`. Возможно использовать одну и ту же функцию для разных таблиц. 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:

Листинг 7.25 Установка правил распределения данных

```
Line 1 CREATE OR REPLACE FUNCTION
```

7.4. Параллельное выполнение запросов

```
- pgpool_catalog.dist_def_branches(anelement)
- RETURNS integer AS $$
-     SELECT CASE WHEN $1 > 0 AND $1 <= 1 THEN 0
5         WHEN $1 > 1 AND $1 <= 2 THEN 1
-         ELSE 2
-     END;
- $$ LANGUAGE sql;
```

Установка правил репликации

Правило репликации — это то что определяет какие таблицы должны быть использованы для выполнения репликации. Здесь это сделано при помощи `pgbench` с зарегистрированными таблицами `branches` и `tellers`. Как результат, стало возможно создание таблицы `accounts` и выполнение запросов, использующих таблицы `branches` и `tellers`:

Листинг 7.26 Установка правил репликации

```
Line 1 INSERT INTO pgpool_catalog.replicate_def VALUES (
-     'bench_parallel',
-     'public',
-     'branches',
5     ARRAY['bid', 'bbalance', 'filler'],
-     ARRAY['integer', 'integer', 'character(88)']
- );
-
- INSERT INTO pgpool_catalog.replicate_def VALUES (
10     'bench_parallel',
-     'public',
-     'tellers',
-     ARRAY['tid', 'bid', 'tbalance', 'filler'],
-     ARRAY['integer', 'integer', 'integer', 'character(84)']
15 );
```

Подготовленный файл `replicate_def_pgbench.sql` находится в каталоге `sample`:

Листинг 7.27 Установка правил репликации

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

Проверка параллельного запроса

После настройки `pgpool.conf` и перезапуска `pgpool-II` возможно провести проверку работоспособности параллельных запросов. Сначала требуется создать базу данных, которая будет распределена. Эту базу данных нужно создать на всех узлах:

7.5. Master-slave режим

Листинг 7.28 Проверка параллельного запроса

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

Затем запустим `pgbench` с параметрами `-i -s 3`:

Листинг 7.29 Проверка параллельного запроса

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

Один из способов проверить корректно ли были распределены данные — выполнить запрос `SELECT` посредством `pgrool-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 `pgrool-II` используем простой скрипт на `shell`. Приведенный ниже скрипт покажет минимальное и максимальное значение в таблице `accounts` используя для соединения порты 5432, 5433, 5434 и 9999.

Листинг 7.30 Проверка параллельного запроса

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

7.5 Master-slave режим

Этот режим предназначен для использования `pgrool-II` с другой репликацией (например `streaming`, `londiste`). Информация про БД указывается как для репликации. `master_slave_mode` и `load_balance_mode` устанавливается в `true`. `pgrool-II` будет посылать запросы `INSERT/UPDATE/DELETE` на master базу данных (1 в списке), а `SELECT` — использовать балансировку нагрузки, если это возможно. При этом, DDL и DML для временной таблицы может быть выполнен только на мастере. Если нужен `SELECT` только на мастере, то для этого нужно использовать комментарий `/*NO LOAD BALANCE*/` перед `SELECT`.

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

Streaming Replication (Потоковая репликация)

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

Листинг 7.31 Скрипт выполняется при падении нода PostgreSQL

```

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

```

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

Листинг 7.32 Что выполнять при падении нода

```

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

```

где `/tmp/trigger_file` — триггер файл, указанный в конфиге `recovery.conf`. Теперь, если мастер СУБД упадет, слейв будет переключен из режима восстановления в обычный и сможет принимать запросы на запись.

7.6 Онлайн восстановление

Pgpool-II в режиме репликации может синхронизировать базы данных и добавлять их как новые ноды. Называется это «онлайн восстановление». Этот метод также может быть использован когда нужно вернуть в репликацию упавший нод базы данных. Вся процедура выполняется в два задания. Несколько секунд или минут клиент может ждать подключения к pgpool, в то время как восстанавливается узел базы данных. Онлайн восстановление состоит из следующих шагов:

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

Для работы онлайн восстановления потребуется указать следующие параметры:

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

Streaming Replication (Потоковая репликация)

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

Для настройки онлайн восстановления потребуется:

- Установить `recovery_user`. Обычно это «postgres»;
- Установить `recovery_password` для `recovery_user` для подключения к СУБД;
- Настроить `recovery_1st_stage_command`. Для этого можно создать скрипт `basebackup.sh` и положим его в папку с данными мастера (`$PGDATA`), установив ему права на выполнение:

Листинг 7.33 basebackup.sh

```
Line 1  #! /bin/sh
- # Recovery script for streaming replication.
- # This script assumes that DB node 0 is primary, and 1
-   is standby.
- #
5  datadir=$1
-  desthost=$2
-  destdir=$3
-
-  psql -c "SELECT pg_start_backup('Streaming Replication ',
-    true)" postgres
10
-  rsync -C -a --delete -e ssh --exclude postgresql.conf --
-    exclude postmaster.pid \
-  --exclude postmaster.opts --exclude pg_log --exclude
-    pg_xlog \
-  --exclude recovery.conf $datadir/ $desthost:$destdir/
-
15  ssh -T localhost mv $destdir/recovery.done $destdir/
-    recovery.conf
-
-  psql -c "SELECT pg_stop_backup()" postgres
```

При восстановлении слейва, скрипт запускает бэкап мастера и через `rsync` утилиту передает данные с мастера на слейв. Для этого необходимо настроить `ssh` так, чтобы `recovery_user` мог заходить с мастера на слейв без пароля. Далее добавим скрипт на выполнение для первого этапа онлайн восстановления:

7.7. Заключение

Листинг 7.34 `recovery_1st_stage_command`

```
Line 1 recovery_1st_stage_command = 'basebackup.sh'
```

- Оставляем `recovery_2nd_stage_command` пустым. После успешного выполнения первого этапа онлайн восстановления, разницу в данных, что успели записаться во время работы скрипта `basebackup.sh`, слейв инстанс заберет через WAL файлы с мастера;
- Устанавливаем C и SQL функции для работы онлайн восстановления на каждый инстанс СУБД:

Листинг 7.35 Устанавливаем C и SQL функции

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

После этого возможно использовать `psr_recovery_node` для онлайн восстановления упавших слейвов.

7.7 Заключение

PgPool-II — прекрасное средство, функционал которого может помочь администраторам баз данных при масштабировании PostgreSQL.

Мультиплексоры соединений

Если сразу успеха не добились, попробуйте снова и снова. Затем оставьте эти попытки. Какой смысл глупо упорствовать?

Уильям Клод Филдс

8.1 Введение

Мультиплексоры соединений (программы для создания пула соединений) позволяют уменьшить накладные расходы на базу данных, в случае, когда огромное количество физических соединений ведет к падению производительности PostgreSQL. Это особенно важно на Windows, когда система ограничивает большое количество соединений. Это также важно для веб-приложений, где количество соединений может быть очень большим.

Для PostgreSQL существует PgBouncer и Pgpool-II, которые работают как мультиплексоры соединений.

8.2 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;

8.2. PgBouncer

- Transaction Pooling — клиент владеет соединением с бэкендом только в течение транзакции. Когда PgBouncer замечает, что транзакция завершилась, он возвращает соединение назад в пул;
- Statement Pooling — наиболее агрессивный режим. Соединение с бэкендом возвращается назад в пул сразу после завершения запроса. Транзакции с несколькими запросами в этом режиме не разрешены, так как они гарантировано будут отменены. Также не работают подготовленные выражения (prepared statements) в этом режиме;

К достоинствам PgBouncer относятся:

- малое потребление памяти (менее 2 КБ на соединение);
- отсутствие привязки к одному серверу баз данных;
- реконфигурация настроек без рестарта.

Базовая утилита запускается просто:

Листинг 8.1 PgBouncer

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

Пример конфига:

Листинг 8.2 PgBouncer

```
Line 1 [databases]
- template1 = host=127.0.0.1 port=5432 dbname=template1
- [pgbouncer]
- listen_port = 6543
5 listen_addr = 127.0.0.1
- auth_type = md5
- auth_file = userlist.txt
- logfile = pgbouncer.log
- pidfile = pgbouncer.pid
10 admin_users = someuser
```

Нужно создать файл пользователей `userlist.txt` примерно такого содержания: `"someuser" "same_password_as_in_server"`. Административный доступ из консоли к базе данных `pgbouncer` можно получить через команду ниже:

Листинг 8.3 PgBouncer

```
Line 1 $ psql -h 127.0.0.1 -p 6543 pgbouncer
```

Здесь можно получить различную статистическую информацию с помощью команды `SHOW`.

8.3 PgPool-II vs PgBouncer

Если сравнивать PgPool-II и PgBouncer, то PgBouncer намного лучше работает с пулами соединений, чем PgPool-II. Если вам не нужны остальные возможности, которыми владеет PgPool-II (ведь пулы коннектов это мелочи к его функционалу), то конечно лучше использовать PgBouncer.

- PgBouncer потребляет меньше памяти, чем PgPool-II;
- у PgBouncer возможно настроить очередь соединений;
- в PgBouncer можно настраивать псевдо базы данных (на сервере они могут называться по-другому);

Также возможен вариант использования PgBouncer и PgPool-II совместно.

Кэширование в PostgreSQL

Чтобы что-то узнать, нужно
уже что-то знать

Станислав Лем

9.1 Введение

Кэш или кеш — промежуточный буфер с быстрым доступом, содержащий информацию, которая может быть запрошена с наибольшей вероятностью. Кэширование **SELECT** запросов позволяет повысить производительность приложений и снизить нагрузку на PostgreSQL. Преимущества кэширования особенно заметны в случае с относительно маленькими таблицами, имеющими статические данные, например, справочными таблицами.

Многие СУБД могут кэшировать SQL запросы, и данная возможность идет у них, в основном, «из коробки». PostgreSQL не обладает подобным функционалом. Почему? Во-первых, мы теряем транзакционную чистоту происходящего в базе. Что это значит? Управление конкурентным доступом с помощью многоверсионности (MVCC — MultiVersion Concurrency Control) — один из механизмов обеспечения одновременного конкурентного доступа к БД, заключающийся в предоставлении каждому пользователю «снимка» БД, обладающего тем свойством, что вносимые данным пользователем изменения в БД невидимы другим пользователям до момента фиксации транзакции. Этот способ управления позволяет добиться того, что пишущие транзакции не блокируют читающих, и читающие транзакции не блокируют пишущих. При использовании кэширования, которому нет дела к транзакциям СУБД, «снимки» БД могут быть с неверными данными. Во-вторых, кэширование результатов запросов, в основном, должно происходить на стороне приложения, а не СУБД. В таком случае управление кэшированием может работать более гибко (включать-

9.2. Pgmemcache

ся и отключаться где потребуется для приложения), а СУБД будет заниматься своей непосредственной целью — хранением и обеспечением целостности данных.

Для организации кэширования существует два инструмента для PostgreSQL:

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

9.2 Pgmemcache

Memcached — программное обеспечение, реализующее сервис кэширования данных в оперативной памяти на основе хэш-таблицы. С помощью клиентской библиотеки позволяет кэшировать данные в оперативной памяти множества доступных серверов. Распределение реализуется путём сегментирования данных по значению хэша ключа по аналогии с сокетом хэш-таблицы. Клиентская библиотека, используя ключ данных, вычисляет хэш и использует его для выбора соответствующего сервера. Ситуация сбоя сервера трактуется как промах кэша, что позволяет повышать отказоустойчивость комплекса за счет наращивания количества memcached серверов и возможности производить их горячую замену.

Pgmemcache — это PostgreSQL API библиотека на основе libmemcached для взаимодействия с memcached. С помощью данной библиотеки PostgreSQL может записывать, считывать, искать и удалять данные из memcached.

Установка

Поскольку Pgmemcache идет как модуль, то потребуется PostgreSQL с PGXS (если уже не установлен, поскольку в сборках для Linux присутствует PGXS). Также потребуется memcached и libmemcached библиотека версии не ниже 0.38. После скачивания и распаковки исходников достаточно выполнить в консоли:

Листинг 9.1 Установка из исходников

```
Line 1 $ make
- $ sudo make install
```

Настройка

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

9.2. Pgmemcache

Листинг 9.2 Настройка

```
Line 1 % psql [mydbname] [pguser]
- [mydbname]=# CREATE EXTENSION pgmemcache;
```

Теперь можно добавлять сервера memcached через `memcache_server_add` и работать с кэшем. Но есть одно но. Все сервера memcached придется задавать при каждом новом подключении к PostgreSQL. Это ограничение можно обойти, если настроить параметры в `postgresql.conf` файле:

- Добавить `pgmemcache` в `shared_preload_libraries` (автозагрузка библиотеки `pgmemcache` во время старта PostgreSQL);
- Добавить `pgmemcache` в `custom_variable_classes` (устанавливаем переменную для `pgmemcache`);
- Создаем `pgmemcache.default_servers`, указав в формате «host:port» (port - опционально) через запятую. Например:

Листинг 9.3 Настройка default_servers

```
Line 1 pgmemcache.default_servers = '127.0.0.1,
192.168.0.20:11211' # подключили два сервера memcached
```

- Также можем настроить работу самой библиотеки `pgmemcache` через `pgmemcache.default_behavior`. Настройки соответствуют настройкам `libmemcached`. Например:

Листинг 9.4 Настройка pgmemcache

```
Line 1 pgmemcache.default_behavior='BINARY_PROTOCOL:1'
```

Теперь не требуется при подключении к PostgreSQL указывать сервера memcached.

Проверка

После успешной установки и настройки `pgmemcache` становится доступен список команд для работы с memcached серверами.

Посмотрим работу в СУБД данных функций. Для начала получим информацию о memcached серверах:

Листинг 9.5 Проверка memcache_stats

```
Line 1 pgmemcache=# SELECT memcache_stats();
- memcache_stats
- -----
-
-
5 Server: 127.0.0.1 (11211)
- pid: 1116
- uptime: 70
```


9.2. Pgmemcache

Таблица 9.1: Список команд pgmemcache

Команда	Описание
memcache_server_add('hostname:port':TEXT) memcache_server_add('hostname':TEXT)	Добавляет memcached сервер в список доступных серверов. Если порт не указан, по умолчанию используется 11211.
memcache_add(key::TEXT, value::TEXT, expire::TIMESTAMPZ) memcache_add(key::TEXT, value::TEXT, expire::INTERVAL) memcache_add(key::TEXT, value::TEXT)	Добавляет ключ в кэш, если ключ не существует.
newval = memcache_decr(key::TEXT, decrement::INT4) newval = memcache_decr(key::TEXT)	Если ключ существует и является целым числом, происходит уменьшение его значения на указанное число (по умолчанию на единицу). Возвращает целое число после уменьшения.
memcache_delete(key::TEXT, hold_timer::INTERVAL) memcache_delete(key::TEXT)	Удаляет указанный ключ. Если указать таймер, то ключ с таким же названием может быть добавлен только после окончания таймера.
memcache_flush_all()	Очищает все данные на всех memcached серверах.
value = memcache_get(key::TEXT)	Выбирает ключ из кэша. Возвращает NULL, если ключ не существует, иначе — текстовую строку.
memcache_get_multi(keys::TEXT[]) memcache_get_multi(keys::BYTEA[])	Получает массив ключей из кэша. Возвращает список найденных записей в виде «ключ=значение».
newval = memcache_incr(key::TEXT, increment::INT4) newval = memcache_incr(key::TEXT)	Если ключ существует и является целым числом, происходит увеличение его значения на указанное число (по умолчанию на единицу). Возвращает целое число после увеличения.
memcache_replace(key::TEXT, value::TEXT, expire::TIMESTAMPZ) memcache_replace(key::TEXT, value::TEXT, expire::INTERVAL) memcache_replace(key::TEXT, value::TEXT)	Заменяет значение для существующего ключа.
memcache_set(key::TEXT, value::TEXT, expire::TIMESTAMPZ) memcache_set(key::TEXT, value::TEXT, expire::INTERVAL) memcache_set(key::TEXT, value::TEXT)	Создает ключ со значением. Если такой ключ существует — заменяет в нем значение на указанное.
stats = memcache_stats()	Возвращает статистику по всем серверам memcached.

```

- time: 1289598098
- version: 1.4.5
10 pointer_size: 32
- rusage_user: 0.0
- rusage_system: 0.24001
- curr_items: 0
- total_items: 0
15 bytes: 0

```

9.2. Pgmemcache

```
- curr_connections: 5
- total_connections: 7
- connection_structures: 6
- cmd_get: 0
20 cmd_set: 0
- get_hits: 0
- get_misses: 0
- evictions: 0
- bytes_read: 20
25 bytes_written: 782
- limit_maxbytes: 67108864
- threads: 4
-
- (1 row)
```

Теперь сохраним данные в memcached и попробуем их забрать:

Листинг 9.6 Проверка

```
Line 1 pgmemcache=# SELECT memcache_add('some_key', 'test_value');
- memcache_add
- -----
- t
5 (1 row)
-
- pgmemcache=# SELECT memcache_get('some_key');
- memcache_get
- -----
10 test_value
- (1 row)
```

Можно также проверить работу счетчиков в memcached (данный функционал может пригодиться для создания последовательностей):

Листинг 9.7 Проверка

```
Line 1 pgmemcache=# SELECT memcache_add('some_seq', '10');
- memcache_add
- -----
- t
5 (1 row)
-
- pgmemcache=# SELECT memcache_incr('some_seq');
- memcache_incr
- -----
10          11
- (1 row)
-
- pgmemcache=# SELECT memcache_incr('some_seq');
- memcache_incr
```

9.2. Pgmemcache

```
15 -----
-          12
- (1 row)
-
- pgmemcache=# SELECT memcache_incr('some_seq', 10);
20 memcache_incr
- -----
-          22
- (1 row)
-
25 pgmemcache=# SELECT memcache_decr('some_seq');
- memcache_decr
- -----
-          21
- (1 row)
30
- pgmemcache=# SELECT memcache_decr('some_seq');
- memcache_decr
- -----
-          20
35 (1 row)
-
- pgmemcache=# SELECT memcache_decr('some_seq', 6);
- memcache_decr
- -----
40          14
- (1 row)
```

Для работы с `pgmemcache` лучше создать функции и, если требуется, активировать эти функции через триггеры.

Например, приложение кэширует зашифрованные пароли пользователей в `memcached` (для более быстрого доступа), и нам требуется обновлять информацию в кэше, если она изменяется в СУБД. Создаем функцию:

Листинг 9.8 Функция для обновления данных в кэше

```
Line 1 CREATE OR REPLACE FUNCTION auth_passwd_upd() RETURNS TRIGGER
      AS $$
-       BEGIN
-       IF OLD.passwd != NEW.passwd THEN
-           PERFORM memcache_set('user_id_' || NEW.
user_id || '_password', NEW.passwd);
5       END IF;
-       RETURN NEW;
- END;
- $$ LANGUAGE 'plpgsql';
```

Активируем триггер для обновления таблицы пользователей:

Листинг 9.9 Триггер

9.3. Заключение

```
Line 1 CREATE TRIGGER auth_passwd_upd_trg AFTER UPDATE ON passwd
      FOR EACH ROW EXECUTE PROCEDURE auth_passwd_upd();
```

Но данный пример транзакционно небезопасен — при отмене транзакции кэш не вернется на старое значение. Поэтому лучше очищать старые данные:

Листинг 9.10 Очистка ключа в кэше

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

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

Листинг 9.11 Триггер

```
Line 1 CREATE TRIGGER auth_passwd_del_trg AFTER DELETE ON passwd
      FOR EACH ROW EXECUTE PROCEDURE auth_passwd_upd();
```

Данный пример сделан для наглядности, а создавать кэш в memcached на кешированный пароль нового пользователя (или обновленного) лучше через приложение.

Заключение

PostgreSQL с помощью Pgmemcache библиотеки позволяет работать с memcached серверами, что может потребоваться в определенных случаях для кэширования данных напрямую с СУБД. Удобство данной библиотеки заключается в полном доступе к функциям memcached, но вот готовой реализации кэширование SQL запросов тут нет, и её придется дорабатывать вручную через функции и триггеры PostgreSQL.

9.3 Заключение

TODO

Расширения

Гибкость ума может заменить красоту

Стендаль

10.1 Введение

Один из главных плюсов PostgreSQL это возможность расширения его функционала с помощью расширений. В данной статье я затрону только самые интересные и популярные из существующих расширений.

10.2 PostGIS

PostGIS добавляет поддержку для географических объектов в PostgreSQL. По сути PostGIS позволяет использовать PostgreSQL в качестве бэкенда пространственной базы данных для геоинформационных систем (ГИС), так же, как ESRI SDE или пространственного расширения Oracle. PostGIS соответствует OpenGIS «Простые особенности. Спецификация для SQL» и был сертифицирован.

Установка и использование

Для начала инициализируем расширение в базе данных:

Листинг 10.1 Инициализация postgis

```
Line 1 # CREATE EXTENSION postgis;
```

При создании пространственной базы данных автоматически создаются таблица метаданных `spatial_ref_sys` и представления `geometry_columns`,

`geography_columns`, `raster_columns` и `raster_overviews`. Они создаются в соответствии со спецификацией «Open Geospatial Consortium Simple Features for SQL specification», выпущенной OGC и описывающей стандартные типы объектов ГИС, функции для манипуляции ими и набор таблиц метаданных. Таблица `spatial_ref_sys` содержит числовые идентификаторы и текстовые описания систем координат, используемых в пространственной базе данных. Одним из полей этой таблицы является поле `SRID` — уникальный идентификатор, однозначно определяющий систему координат. `SRID` представляет из себя числовой код, которому соответствует некоторая система координат. Например, распространенный код EPSG 4326 соответствует географической системе координат WGS84. Более подробную информацию по таблицам метаданных можно найти в руководстве по PostGIS.

Теперь, имея пространственную базу данных, можно создать несколько пространственных таблиц. Для начала создадим обычную таблицу базы данных, чтобы хранить данные о городе. Эта таблица будет содержать три поля: числовой идентификатор, название города и колонка геометрии, содержащую данные о местоположении городов:

Листинг 10.2 Создание таблицы cities

```
Line 1 # CREATE TABLE cities ( id int4 primary key, name varchar
      (50), the_geom geometry(POINT,4326) );
```

`the_geom` поле указывает PostGIS, какой тип геометрии имеет каждый из объектов (точки, линии, полигоны и т. п.), какая размерность (т.к. возможны и 3-4 измерения — `POINTZ`, `POINTM`, `POINTZM`) и какая система координат. Для данных по городам мы будем использовать систему координат EPSG:4326. Чтобы добавить данные геометрии в соответствующую колонку, используется функция PostGIS `ST_GeomFromText`, чтобы сконвертировать координаты и идентификатор референсной системы из текстового формата:

Листинг 10.3 Заполнение таблицы cities

```
Line 1 # INSERT INTO cities (id, the_geom, name) VALUES (1,
      ST_GeomFromText('POINT(-0.1257 51.508)',4326), 'London,
      England');
- # INSERT INTO cities (id, the_geom, name) VALUES (2,
      ST_GeomFromText('POINT(-81.233 42.983)',4326), 'London,
      Ontario');
- # INSERT INTO cities (id, the_geom, name) VALUES (3,
      ST_GeomFromText('POINT(27.91162491 -33.01529)',4326), '
      East London,SA');
```

Все самые обычные операторы SQL могут быть использованы для выбора данных из таблицы PostGIS:

Листинг 10.4 SELECT cities

10.2. PostGIS

```
Line 1 # SELECT * FROM cities;
-   id |          name          |          the_geom
-   --
-   --+-----+-----+-----+-----+-----+-----+
-   1 | London, England | 0101000020
-     E6100000BBB88D06F016C0BF1B2FDD2406C14940
5   2 | London, Ontario | 0101000020
-     E6100000F4FDD478E94E54C0E7FBA9F1D27D4540
-   3 | East London,SA | 0101000020
-     E610000040AB064060E93B4059FAD005F58140C0
- (3 rows)
```

Это возвращает нам бессмысленные значения координат в шестнадцатеричной системе. Если вы хотите увидеть вашу геометрию в текстовом формате WKT, используйте функцию `ST_AsText(the_geom)` или `ST_AsEwkt(the_geom)`. Вы также можете использовать функции `ST_X(the_geom)`, `ST_Y(the_geom)`, чтобы получить числовые значения координат:

Листинг 10.5 SELECT cities

```
Line 1 # SELECT id , ST_AsText(the_geom) , ST_AsEwkt(the_geom) , ST_X(
-       the_geom) , ST_Y(the_geom) FROM cities;
-   id |          st_astext          |          st_asewkt
-       |          st_x          |          st_y
-   --
-   --+-----+-----+-----+-----+-----+-----+
-   1 | POINT(-0.1257 51.508) | SRID=4326;POINT(-0.1257
-     51.508) | -0.1257 | 51.508
5   2 | POINT(-81.233 42.983) | SRID=4326;POINT(-81.233
-     42.983) | -81.233 | 42.983
-   3 | POINT(27.91162491 -33.01529) | SRID=4326;POINT
-     (27.91162491 -33.01529) | 27.91162491 | -33.01529
- (3 rows)
```

Большинство таких функций начинаются с ST (пространственный тип) и описаны в документации PostGIS. Теперь ответим на практический вопрос: на каком расстоянии в метрах друг от друга находятся три города с названием Лондон, учитывая сферичность земли?

Листинг 10.6 Расстояние до Лондона

```
Line 1 # SELECT p1.name, p2.name, ST_Distance_Sphere(p1.the_geom, p2.
-       the_geom) FROM cities AS p1, cities AS p2 WHERE p1.id >
-       p2.id;
-   name          |          name          | st_distance_sphere
-   -----+-----+-----+-----+-----+-----+
-   London, Ontario | London, England | 5875787.03777356
```

10.3. pgSphere

```
5 East London,SA | London, England | 9789680.59961472
- East London,SA | London, Ontario | 13892208.6782928
- (3 rows)
```

Этот запрос возвращает расстояние в метрах между каждой парой городов. Обратите внимание как часть **WHERE** предотвращает нас от получения расстояния от города до самого себя (расстояние всегда будет равно нулю) и расстояния в обратном порядке (расстояние от Лондона, Англия до Лондона, Онтарио будет таким же как от Лондона, Онтарио до Лондона, Англия). Также можем рассчитать расстояния на сфере, используя различные функции и указывая названия сфероида, параметры главных полуосей и коэффициента обратного сжатия:

Листинг 10.7 Расстояние до Лондона

```
Line 1 # SELECT p1.name, p2.name, ST_Distance_Spheroid(
- #         p1.the_geom, p2.the_geom, 'SPHEROID["GRS_1980
- #         ",6378137,298.257222]'
- #     )
- #     FROM cities AS p1, cities AS p2 WHERE p1.id > p2.id
;
5      name          |          name          | st_distance_spheroid
- -----+-----+-----
- London, Ontario   | London, England       | 5892413.63999153
- East London,SA    | London, England       | 9756842.65715046
- East London,SA    | London, Ontario       | 13884149.4143795
10 (3 rows)
```

Заключение

В данной главе мы рассмотрели как начать работать с PostGIS. Более подробно об использовании расширения можно ознакомиться через [официальную документацию](#).

10.3 pgSphere

pgSphere обеспечивает PostgreSQL сферическими типами данных, а также функциями и операторами для работы с ними. Используется для работы с географическими (может использоваться вместо PostGIS) или астрономическими типами данных.

Установка и использование

Для начала инициализируем расширение в базе данных:

Листинг 10.8 Инициализация pgSphere

```
Line 1 # CREATE EXTENSION pg_sphere;
```


10.3. pgSphere

После этого можем проверить, что расширение функционирует:

Листинг 10.9 Проверка pgSphere

```
Line 1 # SELECT spoly '{ (270d,-10d), (270d,30d), (290d,10d) } ';
-
-   spoly
- --
- -----
-   {(4.71238898038469 , -0.174532925199433) ,(4.71238898038469
-     , 0.523598775598299) ,(5.06145483078356 ,
-     0.174532925199433)}
5   (1 row)
```

И работу индексов:

Листинг 10.10 Проверка pgSphere

```
Line 1 # CREATE TABLE test (
- #   pos spoint NOT NULL
- # );
- CREATE TABLE
5 # CREATE INDEX test_pos_idx ON test USING GIST (pos);
- CREATE INDEX
- # INSERT INTO test(pos) VALUES ('( 10.1d, -90d)'), ('( 10d
-   12m 11.3s, -13d 14m)');
- INSERT 0 2
- # VACUUM ANALYZE test;
10 VACUUM
- # SELECT set_sphere_output('DEG');
-   set_sphere_output
- -----
-   SET DEG
15   (1 row)
-
- # SELECT * FROM test;
-           pos
- -----
20   (10.1d , -90d)
-   (10.2031388888889d , -13.23333333333333d)
-   (2 rows)
- # SET enable_seqscan = OFF;
- SET
25 # EXPLAIN SELECT * FROM test WHERE pos = spoint '(10.1d,-90d
-   )';
-
-           QUERY PLAN
- --
- -----
```

10.4. HStore

```
- Bitmap Heap Scan on test (cost=4.16..9.50 rows=2 width=16)
-   Recheck Cond: (pos = '(10.1d , -90d)'::spoint)
30 -> Bitmap Index Scan on test_pos_idx (cost=0.00..4.16
    rows=2 width=0)
-       Index Cond: (pos = '(10.1d , -90d)'::spoint)
- (4 rows)
```

Заключение

Более подробно об использовании расширения можно ознакомиться через [официальную документацию](#).

10.4 HStore

HStore – расширение, которое реализует тип данных для хранения ключ/значение в пределах одного значения в PostgreSQL (например, в одном текстовом поле). Это может быть полезно в различных ситуациях, таких как строки со многими атрибутами, которые редко выбираются, или полу-структурированные данные. Ключи и значения являются простыми текстовыми строками.

Начиная с версии 9.4 PostgreSQL был добавлен JSONB тип (бинарный JSON). Данный тип является объединением JSON структуры с возможностью использовать индексы, как у Hstore. JSONB лучше Hstore тем, что есть возможность сохранять вложенную структуру данных (nested) и хранить не только текстовые строки в значениях. Поэтому лучше использовать JSONB, если есть такая возможность.

Установка и использование

Для начала активируем расширение:

Листинг 10.11 Активация hstore

```
Line 1 # CREATE EXTENSION hstore;
```

Проверим его работу:

Листинг 10.12 Проверка hstore

```
Line 1 # SELECT 'a=>1,a=>2'::hstore;
-   hstore
-  -----
-   "a"=>"1"
5 (1 row)
```

Как видно в примере [10.12](#) ключи в hstore уникальны. Создадим таблицу и заполним её данными:

10.4. HStore

Листинг 10.13 Проверка hstore

```
Line 1 CREATE TABLE products (  
-     id serial PRIMARY KEY,  
-     name varchar ,  
-     attributes hstore  
5 );  
- INSERT INTO products (name, attributes)  
- VALUES (  
-     'Geek Love: A Novel',  
-     'author      => "Katherine Dunn",  
10    pages        => 368,  
-     category    => fiction '  
- ),  
- (  
-     'Leica M9',  
15    'manufacturer => Leica ,  
-     type          => camera ,  
-     megapixels    => 18,  
-     sensor        => "full-frame 35mm" '  
- ),  
20 ( 'MacBook Air 11',  
-     'manufacturer => Apple ,  
-     type          => computer ,  
-     ram           => 4GB,  
-     storage       => 256GB,  
25    processor     => "1.8 ghz Intel i7 duel core",  
-     weight        => 2.38lbs '  
- );
```

Теперь можно производить поиск по ключу:

Листинг 10.14 Поиск по ключу

```
Line 1 # SELECT name, attributes->'pages' as page FROM products  
-     WHERE attributes ? 'pages';  
-     name          | page  
-     -----+-----  
-     Geek Love: A Novel | 368  
5 (1 row)
```

Или по значению ключа:

Листинг 10.15 Поиск по значению ключа

```
Line 1 # SELECT name, attributes->'manufacturer' as manufacturer  
-     FROM products WHERE attributes->'type' = 'computer';  
-     name          | manufacturer  
-     -----+-----  
-     MacBook Air 11 | Apple  
5 (1 row)
```

10.5. PLV8

Создание индексов:

Листинг 10.16 Индексы

```
Line 1 # CREATE INDEX products_hstore_index ON products USING GIST
      ( attributes );
- # CREATE INDEX products_hstore_index ON products USING GIN (
      attributes );
```

Можно также создавать индекс на ключ:

Листинг 10.17 Индекс на ключ

```
Line 1 # CREATE INDEX product_manufacturer ON products ((products.
      attributes -> 'manufacturer'));
```

Заключение

HStore — расширение для удобного и индексируемого хранения слабо-структурированных данных в PostgreSQL, если нет возможности использовать версию базы 9.4 или выше, где для данной задачи есть встроенный **JSONB** тип данных.

10.5 PLV8

PLV8 является расширением, которое предоставляет PostgreSQL процедурный язык с движком V8 JavaScript. С помощью этого расширения можно писать в PostgreSQL JavaScript функции, которые можно вызывать из SQL.

Установка и использование

Для начала инициализируем расширение в базе данных:

Листинг 10.18 Инициализация plv8

```
Line 1 # CREATE extension plv8;
```

V8 компилирует JavaScript код непосредственно в машинный код и с помощью этого достигается высокая скорость работы. Для примера рассмотрим расчет числа Фибоначчи. Вот функция написана на plpgsql:

Листинг 10.19 Фибоначчи на plpgsql

```
Line 1 CREATE OR REPLACE FUNCTION
-   psqfib(n int) RETURNS int AS $$
-   BEGIN
-       IF n < 2 THEN
5         RETURN n;
-       END IF;
```

10.5. PLV8

```
-     RETURN psqlfib(n-1) + psqlfib(n-2);  
- END;  
- $$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

Замерим скорость её работы:

Листинг 10.20 Скорость расчета числа Фибоначчи на plpgsql

```
Line 1 # SELECT n, psqlfib(n) FROM generate_series(0,25,5) as n;  
-   n | psqlfib  
-   ---+-----  
-   0 |      0  
5   5 |      5  
-  10 |     55  
-  15 |    610  
-  20 |   6765  
-  25 |  75025  
10 (6 rows)  
-  
- Time: 2520.386 ms
```

Теперь сделаем то же самое, но с использованием PLV8:

Листинг 10.21 Фибоначчи на plv8

```
Line 1 CREATE OR REPLACE FUNCTION  
- fib(n int) RETURNS int as $$  
-  
-     function fib(n) {  
5     return n<2 ? n : fib(n-1) + fib(n-2)  
-     }  
-     return fib(n)  
-  
- $$ LANGUAGE plv8 IMMUTABLE STRICT;
```

Замерим скорость работы:

Листинг 10.22 Скорость расчета числа Фибоначчи на plv8

```
Line 1 # SELECT n, fib(n) FROM generate_series(0,25,5) as n;  
-   n | fib  
-   ---+-----  
-   0 |      0  
5   5 |      5  
-  10 |     55  
-  15 |    610  
-  20 |   6765  
-  25 |  75025  
10 (6 rows)  
-  
- Time: 5.200 ms
```

10.5. PLV8

Как видим, PLV8 приблизительно в 484 (2520.386/5.200) раз быстрее `plpgsql`. Можно ускорить работу расчета чисел Фибоначчи на PLV8 за счет кеширования:

Листинг 10.23 Фибоначчи на plv8

```
Line 1 CREATE OR REPLACE FUNCTION
- fibcache(n int) RETURNS int as $$
-   var memo = {0: 0, 1: 1};
-   function fib(n) {
5     if (!(n in memo))
-       memo[n] = fib(n-1) + fib(n-2)
-     return memo[n]
-   }
-   return fib(n);
10 $$ LANGUAGE plv8 IMMUTABLE STRICT;
```

Замерим скорость работы:

Листинг 10.24 Скорость расчета числа Фибоначчи на plv8

```
Line 1 # SELECT n, fibcache(n) FROM generate_series(0,25,5) as n;
-   n | fibcache
-   ---+-----
-   0 |         0
5   5 |         5
-  10 |        55
-  15 |       610
-  20 |      6765
-  25 |     75025
10 (6 rows)
-
- Time: 1.202 ms
```

Естественно эти измерения не имеют ничего общего с реальным миром (не нужно каждый день считать числа Фибоначчи в базе данных), но позволяет понять, как V8 может помочь ускорить функции, которые занимаются вычислением чего-либо в базе.

NoSQL

Одним из полезных применений PLV8 может быть создание на базе PostgreSQL документоориентированного хранилища. Для хранения неструктурированных данных можно использовать `hstore` («10.4 HStore»), но у него есть свои недостатки:

- нет вложенности;
- все данные (ключ и значение по ключу) это строка;

10.5. PLV8

Для хранения данных многие документно-ориентированные базы данных используют JSON (MongoDB, CouchDB, Couchbase и т. д.). Для этого, начиная с PostgreSQL 9.2, добавлен тип данных JSON, а с версии 9.4 — JSONB. JSON тип можно добавить для PostgreSQL 9.1 и ниже используя PLV8 и DOMAIN:

Листинг 10.25 Создание типа JSON

```
Line 1 CREATE OR REPLACE FUNCTION
- valid_json(json text)
- RETURNS BOOLEAN AS $$
-   try {
5     JSON.parse(json); return true;
-   } catch(e) {
-     return false;
-   }
- $$ LANGUAGE plv8 IMMUTABLE STRICT;
10
- CREATE DOMAIN json AS TEXT
- CHECK(valid_json(VALUE));
```

Функция `valid_json` используется для проверки JSON данных. Пример использования:

Листинг 10.26 Проверка JSON

```
Line 1 $ CREATE TABLE members ( id SERIAL, profile json );
- $ INSERT INTO members
- VALUES('not good json');
- ERROR: value for domain json
5 violates check constraint "json_check"
- $ INSERT INTO members
- VALUES('{"good": "json", "is": true}');
- INSERT 0 1
- $ SELECT * FROM members;
10
-           profile
- -----
- {"good": "json", "is": true}
- (1 row)
```

Рассмотрим пример использования JSON для хранения данных и PLV8 для их поиска. Для начала создадим таблицу и заполним её данными:

Листинг 10.27 Таблица с JSON полем

```
Line 1 $ CREATE TABLE members ( id SERIAL, profile json );
- $ SELECT count(*) FROM members;
-   count
- -----
5  1000000
- (1 row)
```

```
-
- Time: 201.109 ms
```

В `profile` поле мы записали приблизительно такую структуру JSON:

Листинг 10.28 JSON структура

```
Line 1 {
-   "name": "Litzy Satterfield",
-   "age": 24,
-   "siblings": 2,
5   "faculty": false,
-   "numbers": [
-     {
-       "type": "work",
-       "number": "684.573.3783 x368"+
10  },
-     {
-       "type": "home",
-       "number": "625.112.6081 "
-     }
15  ]
- }
```

Теперь создадим функцию для вывода значения по ключу из JSON (в данном случае ожидаем цифру):

Листинг 10.29 Функция для JSON

```
Line 1 CREATE OR REPLACE FUNCTION get_numeric(json_raw json, key
      text)
- RETURNS numeric AS $$
-   var o = JSON.parse(json_raw);
-   return o[key];
5 $$ LANGUAGE plv8 IMMUTABLE STRICT;
```

Теперь мы можем произвести поиск по таблице, фильтруя по значениям ключей `age`, `siblings` или другим числовым полям:

Листинг 10.30 Поиск по данным JSON

```
Line 1 $ SELECT * FROM members WHERE get_numeric(profile, 'age') =
      36;
- Time: 9340.142 ms
- $ SELECT * FROM members WHERE get_numeric(profile, 'siblings
      ') = 1;
- Time: 14320.032 ms
```

Поиск работает, но скорость очень маленькая. Чтобы увеличить скорость, нужно создать функциональные индексы:

Листинг 10.31 Создание индексов

10.5. PLV8

```
Line 1 $ CREATE INDEX member_age ON members (get_numeric(profile , '
      age '));
- $ CREATE INDEX member_siblings ON members (get_numeric(
      profile , 'siblings '));
```

С индексами скорость поиска по JSON станет достаточно высокая:

Листинг 10.32 Поиск по данным JSON с индексами

```
Line 1 $ SELECT * FROM members WHERE get_numeric(profile , 'age') =
      36;
- Time: 57.429 ms
- $ SELECT * FROM members WHERE get_numeric(profile , 'siblings
      ') = 1;
- Time: 65.136 ms
5 $ SELECT count(*) from members where get_numeric(profile , '
      age') = 26 and get_numeric(profile , 'siblings') = 1;
- Time: 106.492 ms
```

Получилось отличное документоориентированное хранилище из PostgreSQL. Если используется PostgreSQL 9.4 или новее, то можно использовать JSONB тип, у которого уже есть возможность создавать индексы на требуемые ключи в JSON структуре.

PLV8 позволяет использовать некоторые JavaScript библиотеки внутри PostgreSQL. Вот пример рендера **Mustache** темплейтов:

Листинг 10.33 Функция для рендера Mustache темплейтов

```
Line 1 CREATE OR REPLACE FUNCTION mustache(template text, view json
      )
- RETURNS text as $$
- // ...400 lines of mustache....js
- return Mustache.render(template, JSON.parse(view))
5 $$ LANGUAGE plv8 IMMUTABLE STRICT;
```

Листинг 10.34 Рендер темплейтов

```
Line 1 $ SELECT mustache(
- 'hello {{#things}}{.} {{/things}}:) {{#data}}{key}{{/
      data}}',
- '{"things": ["world", "from", "postgresql"], "data": {"key
      ": "and me"}}'
- );
5 mustache
- -----
- hello world from postgresql :) and me
- (1 row)
-
10 Time: 0.837 ms
```

Этот пример показывает какие возможности предоставляет PLV8 в PostgreSQL. В действительности рендерить Mustache темплейты в PostgreSQL не лучшая идея.

Заключение

PLV8 расширение предоставляет PostgreSQL процедурный язык с движком V8 JavaScript, с помощью которого можно работать с JavaScript библиотеками, индексировать JSON данные и использовать его как более быстрый язык для вычислений внутри базы.

10.6 Pg_repack

Таблицы в PostgreSQL представлены в виде страниц размером 8 КБ, в которых размещены записи. Когда одна страница полностью заполняется записями, к таблице добавляется новая страница. При удалении записей с помощью **DELETE** или изменении с помощью **UPDATE**, место где были старые записи не может быть повторно использовано сразу же. Для этого процесс очистки `autovacuum`, или команда **VACUUM**, пробегает по изменённым страницам и помечает такое место как свободное, после чего новые записи могут спокойно записываться в это место. Если `autovacuum` не справляется, например в результате активного изменения большого количества данных или просто из-за плохих настроек, то к таблице будут излишне добавляться новые страницы по мере поступления новых записей. И даже после того как очистка дойдёт до наших удалённых записей, новые страницы останутся. Получается, что таблица становится более разряженной в плане плотности записей. Это и называется эффектом раздувания таблиц (`table bloat`).

Процедура очистки, `autovacuum` или **VACUUM**, может уменьшить размер таблицы, убрав полностью пустые страницы, но только при условии, что они находятся в самом конце таблицы. Чтобы максимально уменьшить таблицу в PostgreSQL есть **VACUUM FULL** или **CLUSTER**, но оба эти способа требуют «`exclusively locks`» на таблицу (то есть в это время с таблицы нельзя ни читать, ни писать), что далеко не всегда является подходящим решением.

Для решения подобных проблем существует утилита `pg_repack`. Эта утилита позволяет сделать **VACUUM FULL** или **CLUSTER** команды без блокировки таблицы. Для чистки таблицы `pg_repack` создает точную её копию в `repack` схеме базы данных (ваша база по умолчанию работает в `public` схеме) и сортирует строки в этой таблице. После переноса данных и чистки мусора утилита меняет схему у таблиц. Для чистки индексов утилита создает новые индексы с другими именами, а по выполнению работы меняет их на первоначальные. Для выполнения всех этих работ потребуется дополнительное место на диске (например, если у вас 100 ГБ данных, и

из них 40 ГБ - распухание таблиц или индексов, то вам потребуется 100 ГБ + (100 ГБ - 40 ГБ) = 160 ГБ на диске минимум). Для проверки «распухания» таблиц и индексов в вашей базе можно воспользоваться SQL запросом из раздела «14.2 Размер распухания (bloat) таблиц и индексов в базе данных».

Существует ряд ограничений в работе pg_repack:

- Не может очистить временные таблицы;
- Не может очистить таблицы с использованием GIST индексов;
- Нельзя выполнять DDL (Data Definition Language) на таблице во время работы;

Пример использования

Выполнить команду **CLUSTER** всех кластеризованных таблиц и **VACUUM FULL** для всех не кластеризованных таблиц в **test** базе данных:

[Скачать Листинг](#)

```
Line 1 $ pg_repack test
```

Выполните команду **VACUUM FULL** на **foo** и **bar** таблицах в **test** базе данных (кластеризация таблиц игнорируется):

[Скачать Листинг](#)

```
Line 1 $ pg_repack --no-order --table foo --table bar test
```

Переместить все индексы таблицы **foo** в неймспейс **tbs**:

[Скачать Листинг](#)

```
Line 1 $ pg_repack -d test --table foo --only-indexes --tablespace  
tbs
```

Pgcompact

Существует еще одно решение для борьбы с раздуванием таблиц. При обновлении записи с помощью **UPDATE**, если в таблице есть свободное место, то новая версия пойдет именно в свободное место, без выделения новых страниц. Предпочтение отдается свободному месту ближе к началу таблицы. Если обновлять таблицу с помощью «fake updates» (**some_column = some_column**) с последней страницы, в какой-то момент все записи с последней страницы перейдут в свободное место в предшествующих страницах таблицы. Таким образом, после нескольких таких операций, последние страницы окажутся пустыми и обычный не блокирующий **VACUUM** сможет отрезать их от таблицы, тем самым уменьшив размер. В итоге, с помощью такой техники можно максимально сжать таблицу, при этом не

вызывая критичных блокировок, а значит без помех для других сессий и нормальной работы базы. Для автоматизации этой процедуры существует утилита `pgcompact`.

Основные характеристики утилиты:

- не требует никаких зависимостей кроме Perl $\geq 5.8.8$, можно просто скопировать `pgcompact` на сервер и работать с ним;
- работает через адаптеры `DBD::Pg`, `DBD::PgPP` или даже через стандартную утилиту `psql`, если первых двух на сервере нет;
- обработка как отдельных таблиц, так и всех таблиц внутри схемы, базы или всего кластера;
- возможность исключения баз, схем или таблиц из обработки;
- анализ эффекта раздувания и обработка только тех таблиц, у которых он присутствует, для более точных расчетов рекомендуется установить расширение `pgstattuple`;
- анализ и перестроение индексов с эффектом раздувания;
- анализ и перестроение уникальных ограничений (`unique constraints`) и первичных ключей (`primary keys`) с эффектом раздувания;
- инкрементальное использование, т. е. можно остановить процесс сжатия без ущерба чему-либо;
- динамическая подстройка под текущую нагрузку базы данных, чтобы не влиять на производительность пользовательских запросов (с возможностью регулировки при запуске);
- рекомендации администраторам, сопровождаемые готовым DDL, для перестроения объектов базы, которые не могут быть перестроены в автоматическом режиме;

Рассмотрим пример использования данной утилиты. Для начала создадим таблицу:

Листинг 10.35 Создаем test таблицу

```
Line 1 # create table test (  
-     id int4,  
-     int_1 int4,  
-     int_2 int4,  
5     int_3 int4,  
-     ts_1 timestamptz,  
-     ts_2 timestamptz,  
-     ts_3 timestamptz,  
-     text_1 text,  
10    text_2 text,  
-     text_3 text  
- );
```

После этого заполним её данными:

Листинг 10.36 Заполняем данными test таблицу

```

Line 1 # insert into test
- select
-     i,
-     cast(random() * 10000000 as int4),
5     cast(random()*10000000 as int4),
-     cast(random()*10000000 as int4),
-     now() - '2 years'::interval * random(),
-     now() - '2 years'::interval * random(),
-     now() - '2 years'::interval * random(),
10    repeat('text_1 ', cast(10 + random() * 100 as int4)),
-     repeat('text_2 ', cast(10 + random() * 100 as int4)),
-     repeat('text_2 ', cast(10 + random() * 100 as int4))
- from generate_series(1, 1000000) i;

```

И добавим индексы:

Листинг 10.37 Индексы для test

```

Line 1 # alter table test add primary key (id);
- ALTER TABLE
-
- # create unique index i1 on test (int_1, int_2);
5 CREATE INDEX
-
- # create index i2 on test (int_2);
- CREATE INDEX
-
10 # create index i3 on test (int_3, ts_1);
- CREATE INDEX
-
- # create index i4 on test (ts_2);
- CREATE INDEX
15
- # create index i5 on test (ts_3);
- CREATE INDEX
-
- # create index i6 on test (text_1);
20 CREATE INDEX

```

В результате получим test таблицу, как показано на [10.38](#):

Листинг 10.38 Таблица test

```

Line 1 # \d test
-
-          Table "public.test"
-   Column |          Type          | Modifiers
-   -----+-----+-----
5  id      | integer                | not null
-  int_1   | integer                |

```

10.6. Pg_repack

```
- int_2 | integer |
- int_3 | integer |
- ts_1  | timestamp with time zone |
10 ts_2  | timestamp with time zone |
- ts_3  | timestamp with time zone |
- text_1 | text |
- text_2 | text |
- text_3 | text |
15 Indexes:
- "test_pkey" PRIMARY KEY, btree (id)
- "i1" UNIQUE, btree (int_1, int_2)
- "i2" btree (int_2)
- "i3" btree (int_3, ts_1)
20 "i4" btree (ts_2)
- "i5" btree (ts_3)
- "i6" btree (text_1)
```

Размер таблицы и индексов:

Листинг 10.39 Размер таблицы и индексов

```
Line 1 # SELECT nspname || '.' || relname AS "relation",
- pg_size_pretty(pg_total_relation_size(C.oid)) AS "
total_size"
- FROM pg_class C
- LEFT JOIN pg_namespace N ON (N.oid = C.renamespace)
5 WHERE nspname NOT IN ('pg_catalog', 'information_schema')
- AND nspname !~ '^pg_toast'
- ORDER BY pg_total_relation_size(C.oid) DESC
- LIMIT 20;
- relation | total_size
10 -----+-----
- public.test | 1705 MB
- public.i6 | 242 MB
- public.i3 | 30 MB
- public.i1 | 21 MB
15 public.i2 | 21 MB
- public.i4 | 21 MB
- public.i5 | 21 MB
- public.test_pkey | 21 MB
- (8 rows)
```

Теперь давайте создадим распухание таблицы. Для этого удалим 95% записей в ней:

Листинг 10.40 Удаляем 95% записей

```
Line 1 # DELETE FROM test WHERE random() < 0.95;
- DELETE 950150
```

Далее сделаем `VACUUM` и проверим размер заново:

10.6. Pg_repack

Листинг 10.41 Размер таблицы и индексов

```
Line 1 # VACUUM;
- VACUUM
- # SELECT nspname || '.' || relname AS "relation",
-       pg_size_pretty(pg_total_relation_size(C.oid)) AS "
-       total_size"
5  FROM pg_class C
-  LEFT JOIN pg_namespace N ON (N.oid = C.renamespace)
-  WHERE nspname NOT IN ('pg_catalog', 'information_schema')
-         AND nspname !~ '^pg_toast'
-  ORDER BY pg_total_relation_size(C.oid) DESC
10  LIMIT 20;
-      relation      | total_size
-  -----+-----
-  public.test       | 1705 MB
-  public.i6         | 242 MB
15  public.i3         | 30 MB
-  public.i1         | 21 MB
-  public.i2         | 21 MB
-  public.i4         | 21 MB
-  public.i5         | 21 MB
20  public.test_pkey | 21 MB
-  (8 rows)
```

Как видно из результата, размер не изменился. Теперь попробуем убрать через `pgcompact` распухание таблицы и индексов (для этого дополнительно добавим в базу данных расширение `pgstattuple`):

Листинг 10.42 Запуск `pgcompact`

```
Line 1 # CREATE EXTENSION pgstattuple;
- # \q
-
- $ git clone https://github.com/grayhemp/pgtoolkit.git
5 $ cd pgtoolkit
- $ time ./bin/pgcompact -v info -d analytics_prod --reindex
-       2>&l | tee pgcompact.output
- Sun Oct 30 11:01:18 2016 INFO Database connection method:
-       psql.
- Sun Oct 30 11:01:18 2016 dev INFO Created environment.
- Sun Oct 30 11:01:18 2016 dev INFO Statistics calculation
-       method: pgstattuple.
10 Sun Oct 30 11:02:03 2016 dev, public.test INFO Vacuum
-       initial: 169689 pages left, duration 45.129 seconds.
- Sun Oct 30 11:02:13 2016 dev, public.test INFO Bloat
-       statistics with pgstattuple: duration 9.764 seconds.
- Sun Oct 30 11:02:13 2016 dev, public.test NOTICE Statistics:
-       169689 pages (218233 pages including toasts and indexes)
```

10.6. Pg_repack

```
, approximately 94.62% (160557 pages) can be compacted
reducing the size by 1254 MB.
- Sun Oct 30 11:02:13 2016 dev, public.test INFO Update by
column: id.
- Sun Oct 30 11:02:13 2016 dev, public.test INFO Set pages/
round: 10.
15 Sun Oct 30 11:02:13 2016 dev, public.test INFO Set pages/
vacuum: 3394.
- Sun Oct 30 11:04:56 2016 dev, public.test INFO Progress: 0%,
260 pages completed.
- Sun Oct 30 11:05:45 2016 dev, public.test INFO Cleaning in
average: 85.8 pages/second (0.117 seconds per 10 pages).
- Sun Oct 30 11:05:48 2016 dev, public.test INFO Vacuum
routine: 166285 pages left, duration 2.705 seconds.
- Sun Oct 30 11:05:48 2016 dev, public.test INFO Set pages/
vacuum: 3326.
20 Sun Oct 30 11:05:57 2016 dev, public.test INFO Progress: 2%,
4304 pages completed.
- Sun Oct 30 11:06:19 2016 dev, public.test INFO Cleaning in
average: 841.6 pages/second (0.012 seconds per 10 pages).
- Sun Oct 30 11:06:23 2016 dev, public.test INFO Vacuum
routine: 162942 pages left, duration 4.264 seconds.
- Sun Oct 30 11:06:23 2016 dev, public.test INFO Set pages/
vacuum: 3260.
- Sun Oct 30 11:06:49 2016 dev, public.test INFO Cleaning in
average: 818.1 pages/second (0.012 seconds per 10 pages).
25 Sun Oct 30 11:06:49 2016 dev, public.test INFO Vacuum
routine: 159681 pages left, duration 0.325 seconds.
- Sun Oct 30 11:06:49 2016 dev, public.test INFO Set pages/
vacuum: 3194.
- Sun Oct 30 11:06:57 2016 dev, public.test INFO Progress: 6%,
10958 pages completed.
- Sun Oct 30 11:07:23 2016 dev, public.test INFO Cleaning in
average: 694.8 pages/second (0.014 seconds per 10 pages).
- Sun Oct 30 11:07:24 2016 dev, public.test INFO Vacuum
routine: 156478 pages left, duration 1.362 seconds.
30 Sun Oct 30 11:07:24 2016 dev, public.test INFO Set pages/
vacuum: 3130.
- ...
- Sun Oct 30 11:49:02 2016 dev NOTICE Processing complete.
- Sun Oct 30 11:49:02 2016 dev NOTICE Processing results: size
reduced by 1256 MB (1256 MB including toasts and indexes
) in total.
- Sun Oct 30 11:49:02 2016 NOTICE Processing complete: 0
retries from 10.
35 Sun Oct 30 11:49:02 2016 NOTICE Processing results: size
reduced by 1256 MB (1256 MB including toasts and indexes)
```


10.6. Pg_repack

```
        in total , 1256 MB (1256 MB) dev .
- Sun Oct 30 11:49:02 2016 dev INFO Dropped environment .
-
- real    47m44.831s
- user    0m37.692s
40 sys    0m16.336s
```

После данной процедуры проверим размер таблицы и индексов в базе:

Листинг 10.43 Размер таблицы и индексов

```
Line 1 # VACUUM;
- VACUUM
- # SELECT nspname || '.' || relname AS "relation",
-        pg_size_pretty(pg_total_relation_size(C.oid)) AS "
-        total_size"
5 FROM pg_class C
- LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
- WHERE nspname NOT IN ('pg_catalog', 'information_schema')
-        AND nspname !~ '^pg_toast'
- ORDER BY pg_total_relation_size(C.oid) DESC
10 LIMIT 20;
-      relation      | total_size
- -----+-----
- public.test       | 449 MB
- public.i6         | 12 MB
15 public.i3         | 1544 kB
- public.i1         | 1112 kB
- public.i2         | 1112 kB
- public.test_pkey  | 1112 kB
- public.i4         | 1112 kB
20 public.i5         | 1112 kB
- (8 rows)
```

Как видно, в результате размер таблицы сократился до 449 МБ (было 1705 МБ). Индексы тоже стали меньше (например, `i6` был 242 МБ, а стал 12 МБ). Операция заняла 47 минут и обрабатывала в среднем 600 страниц в секунду (4.69 МБ в секунду). Можно ускорить выполнение этой операции через `--delay-ratio` параметр (задержка между раундами выполнения, по умолчанию 2 секунды) и `--max-pages-per-round` параметр (количество страниц, что будет обработано за один раунд, по умолчанию 10). Более подробно по параметрам `pgcompact` можно ознакомиться через команду `pgcompact --man`.

Заключение

`Pg_repack` и `pgcompact` — утилиты, которые могут помочь в борьбе с распуханием таблиц в PostgreSQL «на лету».

10.7 Pg_prewarm

Модуль `pg_prewarm` обеспечивает удобный способ загрузки данных объектов (таблиц, индексов, прочего) в буферный кэш PostgreSQL или операционной системы. Данный модуль добавлен в contrib начиная с PostgreSQL 9.4.

Установка и использование

Для начала нужно установить модуль:

[Скачать](#) Листинг

```
Line 1 $ CREATE EXTENSION pg_prewarm;
```

После установки доступна функция `pg_prewarm`:

[Скачать](#) Листинг

```
Line 1 $ SELECT pg_prewarm('pgbench_accounts');
-      pg_prewarm
-      -----
-              4082
5 (1 row)
```

Первый аргумент — объект, который требуется предварительно загрузить в память. Второй аргумент — «режим» загрузки в память, который может содержать такие варианты:

- `prefetch` — чтение файла ядром системы в асинхронном режиме (в фоновом режиме). Это позволяет положить содержимое файла в кэш ядра системы. Но этот режим работает не на всех платформах;
- `read` — результат похож на `prefetch`, но делается синхронно (а значит медленнее). Работает на всех платформах;
- `buffer` — в этом режиме данные будут грузиться в PostgreSQL `shared_buffers`. Этот режим используется по умолчанию;

Третий аргумент называется «fork». О нем не нужно беспокоиться. Возможные значения: «main» (используется по умолчанию), «fsm», «vm».

Четвертый и пятый аргументы указывают диапазон страниц для загрузки данных. По умолчанию загружается весь объект в память, но можно решить, например, загрузить только последние 1000 страниц:

[Скачать](#) Листинг

```
Line 1 $ SELECT pg_prewarm(
-       'pgbench_accounts',
-       first_block := (
-         SELECT pg_relation_size('pgbench_accounts') /
-         current_setting('block_size')::int4 - 1000
5       )
- );
```

Заключение

Pg_prewarm — расширение, которое позволяет предварительно загрузить («подогреть») данные в буферной кэш PostgreSQL или операционной системы.

10.8 Smlar

Поиск похожести в больших базах данных является важным вопросом в настоящее время для таких систем как блоги (похожие статьи), интернет-магазины (похожие продукты), хостинг изображений (похожие изображения, поиск дубликатов изображений) и т. д. PostgreSQL позволяет сделать такой поиск более легким. Прежде всего необходимо понять, как мы будем вычислять сходство двух объектов.

Похожесть

Любой объект может быть описан как список характеристик. Например, статья в блоге может быть описана тегами, продукт в интернет-магазине может быть описан размером, весом, цветом и т. д. Это означает, что для каждого объекта можно создать цифровую подпись — массив чисел, описывающих объект (**отпечатки пальцев, n-grams**). То есть нужно создать массив из цифр для описания каждого объекта.

Расчет похожести

Есть несколько методов вычисления похожести сигнатур объектов. Прежде всего, легенда для расчетов:

- N_a, N_b — количество уникальных элементов в массивах;
- N_u — количество уникальных элементов при объединении массивов;
- N_i — количество уникальных элементов при пересечении массивов.

Один из простейших расчетов похожести двух объектов - количество уникальных элементов при пересечении массивов делить на количество уникальных элементов в двух массивах:

$$S(A, B) = \frac{N_i}{(N_a + N_b)} \quad (10.1)$$

или проще

$$S(A, B) = \frac{N_i}{N_u} \quad (10.2)$$

Преимущества:

- Легко понять;
- Скорость расчета: $N * \log N$;
- Хорошо работает на похожих и больших N_a и N_b ;

Также похожесть можно рассчитать по **формуле косинусов**:

$$S(A, B) = \frac{N_i}{\sqrt{N_a * N_b}} \quad (10.3)$$

Преимущества:

- Скорость расчета: $N * \log N$;
- Отлично работает на больших N ;

Но у обоих этих методов есть общие проблемы:

- Если элементов мало, то разброс похожести невелик;
- Глобальная статистика: частые элементы ведут к тому, что вес ниже;
- Спамеры и недобросовестные пользователи могут разрушить работу алгоритма и он перестанет работать на Вас;

Для избежания этих проблем можно воспользоваться **TF/IDF метрикой**:

$$S(A, B) = \frac{\sum_{i < N_a, j < N_b, A_i = B_j} TF_i * TF_j}{\sqrt{\sum_{i < N_a} TF_i^2 * \sum_{j < N_b} TF_j^2}} \quad (10.4)$$

где инвертированный вес элемента в коллекции:

$$IDF_{element} = \log\left(\frac{N_{objects}}{N_{objects\ with\ element}} + 1\right) \quad (10.5)$$

и вес элемента в массиве:

$$TF_{element} = IDF_{element} * N_{occurrences} \quad (10.6)$$

Все эти алгоритмы встроены в smlar расширение. Главное понимать, что для TF/IDF метрики требуется вспомогательная таблица для хранения данных, по сравнению с другими простыми метриками.

Smlar

Олег Бартунов и Теодор Сигаев разработали PostgreSQL расширение **smlar**, которое предоставляет несколько методов для расчета похожести массивов (все встроенные типы данных поддерживаются) и оператор для расчета похожести с поддержкой индекса на базе GIST и GIN. Для начала установим это расширение:

10.8. Smlar

Листинг 10.44 Установка smlar

```
Line 1 $ git clone git://sigaev.ru/smlar
- $ cd smlar
- $ USE_PGXS=1 make && make install
```

Теперь проверим расширение:

Листинг 10.45 Проверка smlar

```
Line 1 $ psql
- psql (9.5.1)
- Type "help" for help.
-
5 test=# CREATE EXTENSION smlar;
- CREATE EXTENSION
-
- test=# SELECT smlar( '{1,4,6}'::int[], '{5,4,6}'::int[] );
- smlar
10 -----
- 0.666667
- (1 row)
-
- test=# SELECT smlar( '{1,4,6}'::int[], '{5,4,6}'::int[], 'N.i
- / sqrt(N.a * N.b)' );
15 smlar
- -----
- 0.666667
- (1 row)
```

Методы, которые предоставляет это расширение:

- `float4 smlar(anyarray, anyarray)` — вычисляет похожесть двух массивов. Массивы должны быть одного типа;
- `float4 smlar(anyarray, anyarray, bool useIntersect)` — вычисляет похожесть двух массивов составных типов. Составной тип выглядит следующим образом:

Листинг 10.46 Составной тип

```
Line 1 CREATE TYPE type_name AS (element_name anytype,
weight_name float4);
```

`useIntersect` параметр для использования пересекающихся элементов в знаменателе;

- `float4 smlar(anyarray a, anyarray b, text formula)` — вычисляет похожесть двух массивов по данной формуле, массивы должны быть того же типа. Доступные переменные в формуле:

- `N.i` — количество общих элементов обоих массивов (пересечение);

10.8. Smlar

- N.a — количество уникальных элементов первого массива;
- N.b — количество уникальных элементов второго массива;
- `anyarray % anyarray` — возвращает истину, если похожесть массивов больше, чем указанный предел. Предел указывается в конфиге PostgreSQL:

Листинг 10.47 Smlar предел

```
Line 1 custom_variable_classes = 'smlar'
- smlar.threshold = 0.8 # предел от 0 до 1
```

Также в конфиге можно указать дополнительные настройки для smlar:

Листинг 10.48 Smlar настройки

```
Line 1 custom_variable_classes = 'smlar'
- smlar.threshold = 0.8 # предел от 0 до 1
- smlar.type = 'cosine' # по какой формуле производить расчет
  похожести: cosine, tfidf, overlap
- smlar.stattable = 'stat' # Имя таблицы для хранения
  статистики при работе по формуле tfidf
```

Более подробно можно прочитать в README этого расширения.

GiST и GIN индексы поддерживаются для оператора %.

Пример: поиск дубликатов картинок

Рассмотрим простой пример поиска дубликатов картинок. Алгоритм помогает найти похожие изображения, которые, например, незначительно отличаются (изображение обесцветили, добавили водяные знаки, пропустили через фильтры). Но, поскольку точность мала, то у алгоритма есть и позитивная сторона — скорость работы. Как можно определить, что картинки похожи? Самый простой метод — сравнивать попиксельно два изображения. Но скорость такой работы будет невелика на больших разрешениях. Тем более, такой метод не учитывает, что могли изменять уровень света, насыщенность и прочие характеристики изображения. Нам нужно создать сигнатуру для картинок в виде массива цифр:

- Создаем пиксельную матрицу к изображению (изменения размера изображения к требуемому размеру пиксельной матрице), например 15X15 пикселей (Рис. 10.1);
- Рассчитаем интенсивность каждого пикселя (интенсивность вычисляется по формуле $0.299 * \text{красный} + 0.587 * \text{зеленый} + 0.114 * \text{синий}$). Интенсивность поможет нам находить похожие изображения, не обращая внимание на используемые цвета в них;

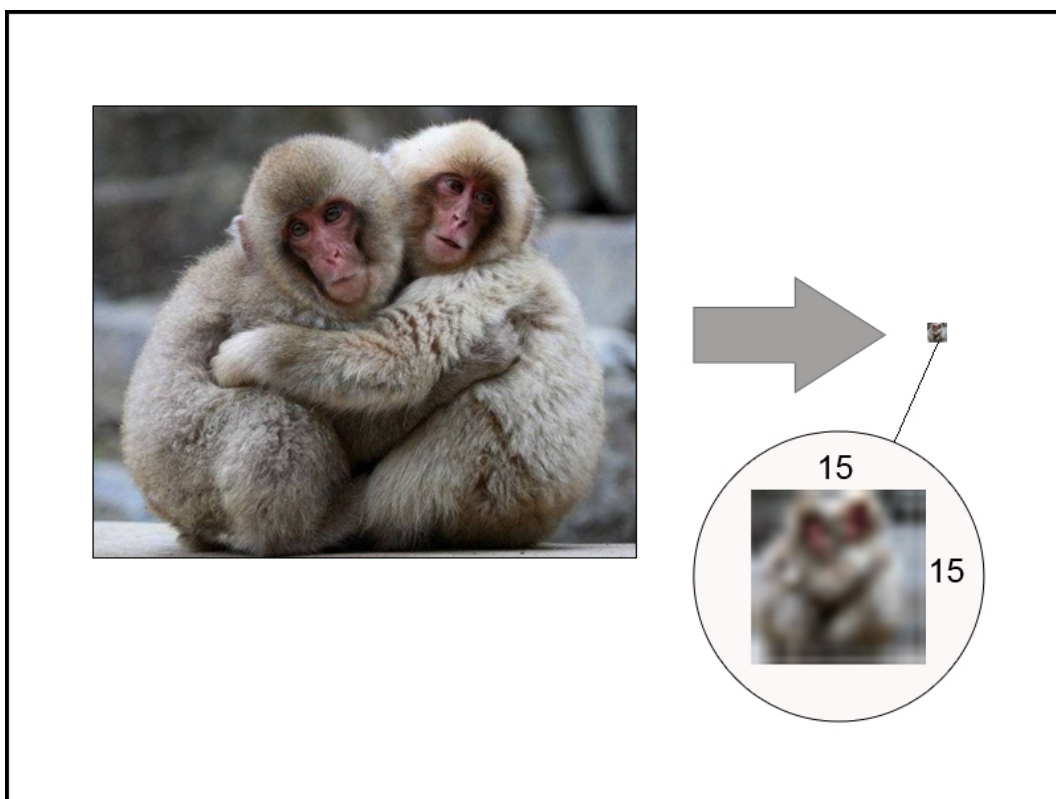


Рис. 10.1: Пиксельная матрица

- Узнаем отношение интенсивности каждого пикселя к среднему значению интенсивности по всей матрице(Рис. 10.2);
- Генерируем уникальное число для каждой ячейки (отношение интенсивности + координаты ячейки);
- Сигнатура для картинке готова;

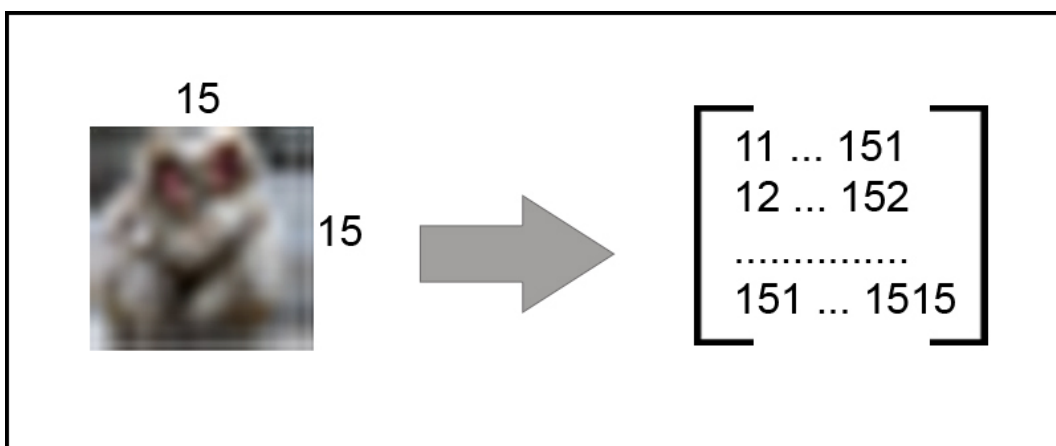


Рис. 10.2: Пиксельная матрица

10.8. Smlar

Создаем таблицу, где будем хранить имя картинки, путь к ней и её сигнатуру:

Листинг 10.49 Таблица для изображений

```
Line 1 CREATE TABLE images (  
-   id serial PRIMARY KEY,  
-   name varchar(50),  
-   img_path varchar(250),  
5   image_array integer []  
- );
```

Создадим GIN или GIST индекс:

Листинг 10.50 Создание GIN или GIST индекса

```
Line 1 CREATE INDEX image_array_gin ON images USING GIN(image_array  
-   _int4_sml_ops);  
- CREATE INDEX image_array_gist ON images USING GIST(  
-   image_array _int4_sml_ops);
```

Теперь можно произвести поиск дубликатов:

Листинг 10.51 Поиск дубликатов

```
Line 1 test=# SELECT count(*) from images;  
-   count  
-   -----  
-   1000000  
5 (1 row)  
-  
- test=# EXPLAIN ANALYZE SELECT count(*) FROM images WHERE  
-   images.image_array % '  
-   {1010259,1011253,...,2423253,2424252}' :: int [];  
-  
- Bitmap Heap Scan on images (cost=286.64..3969.45 rows=986  
-   width=4) (actual time=504.312..2047.533 rows=200000 loops  
-   =1)  
10 Recheck Cond: (image_array % '  
-   {1010259,1011253,...,2423253,2424252}' :: integer [])  
-   -> Bitmap Index Scan on image_array_gist (cost  
-   =0.00..286.39 rows=986 width=0) (actual time  
-   =446.109..446.109 rows=200000 loops=1)  
-   Index Cond: (image_array % '  
-   {1010259,1011253,...,2423253,2424252}' :: integer [])  
-   Total runtime: 2152.411 ms  
- (5 rows)
```

где '{1010259,...,2424252}' :: int [] — сигнатура изображения, для которой пытаемся найти похожие изображения. С помощью `smlar.threshold` управляем % похожести картинок (при каком проценте они будут попадать в выборку).

10.9. Multicorn

Дополнительно можем добавить сортировку по самым похожим изображениям:

Листинг 10.52 Добавляем сортировку по сходству картинок

```
Line 1 test=# EXPLAIN ANALYZE SELECT smmlar(images.image_array, '
      {1010259,...,2424252}'::int[]) as similarity FROM images
      WHERE images.image_array % '{1010259,1011253,
      ...,2423253,2424252}'::int[] ORDER BY similarity DESC;
-
-
- Sort (cost=4020.94..4023.41 rows=986 width=924) (actual
      time=2888.472..2901.977 rows=200000 loops=1)
5  Sort Key: (smmlar(image_array, '{...,2424252}'::integer[]))
-  Sort Method: quicksort Memory: 15520kB
-  -> Bitmap Heap Scan on images (cost=286.64..3971.91
      rows=986 width=924) (actual time=474.436..2729.638 rows
      =200000 loops=1)
-    Recheck Cond: (image_array % '{...,2424252}'::
      integer[])
-    -> Bitmap Index Scan on image_array_gist (cost
      =0.00..286.39 rows=986 width=0) (actual time
      =421.140..421.140 rows=200000 loops=1)
10   Index Cond: (image_array % '{...,2424252}'::
      integer[])
- Total runtime: 2912.207 ms
- (8 rows)
```

Заключение

Smlar расширение может быть использовано в системах, где нам нужно искать похожие объекты, такие как: тексты, темы, блоги, товары, изображения, видео, отпечатки пальцев и прочее.

10.9 Multicorn

Multicorn — расширение для PostgreSQL версии 9.1 или выше, которое позволяет создавать собственные FDW (Foreign Data Wrapper), используя язык программирования **Python**. Foreign Data Wrapper позволяют подключиться к другим источникам данных (другая база, файловая система, REST API, прочее) в PostgreSQL и были представлены с версии 9.1.

Пример

Установка будет проводиться на Ubuntu Linux. Для начала нужно установить требуемые зависимости:

Листинг 10.53 Multicorn

```
Line 1 $ sudo aptitude install build-essential postgresql-server-
      dev-9.3 python-dev python-setuptools
```

Следующим шагом установим расширение:

Листинг 10.54 Multicorn

```
Line 1 $ git clone git@github.com:Kozea/Multicorn.git
- $ cd Multicorn
- $ make && sudo make install
```

Для завершения установки активируем расширение для базы данных:

Листинг 10.55 Multicorn

```
Line 1 # CREATE EXTENSION multicorn;
- CREATE EXTENSION
```

Рассмотрим какие FDW может предоставить Multicorn.

Реляционная СУБД

Для подключения к другой реляционной СУБД Multicorn использует **SQLAlchemy** библиотеку. Данная библиотека поддерживает SQLite, PostgreSQL, MySQL, Oracle, MS-SQL, Firebird, Sybase, и другие базы данных. Для примера настроим подключение к MySQL. Для начала нам потребуется установить зависимости:

Листинг 10.56 Multicorn

```
Line 1 $ sudo aptitude install python-sqlalchemy python-mysqldb
```

В MySQL базе данных «testing» у нас есть таблица «companies»:

Листинг 10.57 Multicorn

```
Line 1 $ mysql -u root -p testing
-
- mysql> SELECT * FROM companies;
- +-----+-----+-----+-----+-----+-----+
5 | id | created_at          | updated_at          |
- +-----+-----+-----+-----+-----+-----+
- | 1 | 2013-07-16 14:06:09 | 2013-07-16 14:06:09 |
- | 2 | 2013-07-16 14:30:00 | 2013-07-16 14:30:00 |
- | 3 | 2013-07-16 14:33:41 | 2013-07-16 14:33:41 |
10 | 4 | 2013-07-16 14:38:42 | 2013-07-16 14:38:42 |
- | 5 | 2013-07-19 14:38:29 | 2013-07-19 14:38:29 |
- +-----+-----+-----+-----+-----+-----+
- 5 rows in set (0.00 sec)
```

В PostgreSQL мы должны создать сервер для Multicorn:

Листинг 10.58 Multicorn

```
Line 1 # CREATE SERVER alchemy_srv foreign data wrapper multicorn
      options (
-      wrapper 'multicorn.sqlalchemyfdw.SqlAlchemyFdw'
-    );
- CREATE SERVER
```

Теперь мы можем создать таблицу, которая будет содержать данные из MySQL таблицы «companies»:

Листинг 10.59 Multicorn

```
Line 1 # CREATE FOREIGN TABLE mysql_companies (
-   id integer ,
-   created_at timestamp without time zone ,
-   updated_at timestamp without time zone
5 ) server alchemy_srv options (
-   tablename 'companies' ,
-   db_url 'mysql://root:password@127.0.0.1/testing'
- );
- CREATE FOREIGN TABLE
```

Основные опции:

- `db_url` (string) — SQLAlchemy настройки подключения к базе данных (примеры: `mysql://<user>:<password>@<host>/<dbname>`, `mssql:mssql://<user>:<password>@<dsname>`). Подробнее можно узнать из [SQLAlchemy документации](#);
- `tablename` (string) — имя таблицы в подключенной базе данных.

Теперь можем проверить, что все работает:

Листинг 10.60 Multicorn

```
Line 1 # SELECT * FROM mysql_companies ;
-   id |          created_at          |          updated_at
-   ---+-----+-----+-----+-----+-----+-----
-   1 | 2013-07-16 14:06:09 | 2013-07-16 14:06:09
5   2 | 2013-07-16 14:30:00 | 2013-07-16 14:30:00
-   3 | 2013-07-16 14:33:41 | 2013-07-16 14:33:41
-   4 | 2013-07-16 14:38:42 | 2013-07-16 14:38:42
-   5 | 2013-07-19 14:38:29 | 2013-07-19 14:38:29
- (5 rows)
```

IMAP сервер

Multicorn может использоваться для получения писем по IMAP протоколу. Для начала установим зависимости:

Листинг 10.61 Multicorn

```
Line 1 $ sudo aptitude install python-pip
- $ sudo pip install imapclient
```

Следующим шагом мы должны создать сервер и таблицу, которая будет подключена к IMAP серверу:

Листинг 10.62 Multicorn

```
Line 1 # CREATE SERVER multicorn_imap FOREIGN DATA WRAPPER
      multicorn options ( wrapper 'multicorn.imapfdw.' )
      ;
- CREATE SERVER
- # CREATE FOREIGN TABLE my_inbox (
-   "Message-ID" character varying ,
5   "From" character varying ,
-   "Subject" character varying ,
-   "payload" character varying ,
-   "flags" character varying [] ,
-   "To" character varying ) server multicorn_imap options (
10   host 'imap.gmail.com' ,
-   port '993' ,
-   payload_column 'payload' ,
-   flags_column 'flags' ,
-   ssl 'True' ,
15   login 'example@gmail.com' ,
-   password 'supersecretpassword'
- );
- CREATE FOREIGN TABLE
```

Основные опции:

- `host (string)` — IMAP хост;
- `port (string)` — IMAP порт;
- `login (string)` — IMAP логин;
- `password (string)` — IMAP пароль;
- `payload_column (string)` — имя поля, которое будет содержать текст письма;
- `flags_column (string)` — имя поля, которое будет содержать IMAP флаги письма (массив);
- `ssl (boolean)` — использовать SSL для подключения;
- `imap_server_charset (string)` — кодировка для IMAP команд. По умолчанию UTF8.

Теперь можно получить письма через таблицу «my_inbox»:

Листинг 10.63 Multicorn

```
Line 1 # SELECT flags , "Subject" , payload FROM my_inbox LIMIT 10;
```

10.9. Multicorn

```
-          flags          |          Subject          |
-          payload        |                               |
-  --
-  -----+-----+-----
-  {$MailFlagBit1,"\\Flagged","\\Seen"} | Test email          |
-  Test email\r          +                               |
5
-  {"\\Seen"} | Test second email |
-  Test second email\r+                               |
-
- (2 rows)
```

RSS

Multicorn может использовать **RSS** как источник данных. Для начала установим зависимости:

Листинг 10.64 Multicorn

```
Line 1 $ sudo aptitude install python-lxml
```

Как и в прошлые разы, создаем сервер и таблицу для RSS ресурса:

Листинг 10.65 Multicorn

```
Line 1 # CREATE SERVER rss_srv foreign data wrapper multicorn
-      options (
-        wrapper 'multicorn.rssfdw.RssFdw'
-      );
- CREATE SERVER
5 # CREATE FOREIGN TABLE my_rss (
-   "pubDate" timestamp,
-   description character varying,
-   title character varying,
-   link character varying
10 ) server rss_srv options (
-   url 'http://news.yahoo.com/rss/entertainment'
- );
- CREATE FOREIGN TABLE
```

Основные опции:

- `url (string)` — URL RSS ленты.

Кроме того, вы должны быть уверены, что PostgreSQL база данных использует UTF-8 кодировку (в другой кодировке вы можете получить ошибки). Результат таблицы «my_rss»:

10.9. Multicorn

Листинг 10.66 Multicorn

```
Line 1 # SELECT "pubDate", title , link from my_rss ORDER BY "
      pubDate" DESC LIMIT 10;
-      pubDate      |                               title
-      link        |
-      --
-      -----+-----
-      2013-09-28 14:11:58 | Royal Mint coins to mark Prince
      George christening | http://news.yahoo.com/royal-mint-
      coins-mark-prince-george-christening-115906242.html
5      2013-09-28 11:47:03 | Miss Philippines wins Miss World in
      Indonesia          | http://news.yahoo.com/miss-philippines-
      wins-miss-world-indonesia-144544381.html
-      2013-09-28 10:59:15 | Billionaire 's daughter in NJ court in
      will dispute | http://news.yahoo.com/billionaires-
      daughter-nj-court-dispute-144432331.html
-      2013-09-28 08:40:42 | Security tight at Miss World final in
      Indonesia        | http://news.yahoo.com/security-tight-miss-
      -world-final-indonesia-123714041.html
-      2013-09-28 08:17:52 | Guest lineups for the Sunday news
      shows             | http://news.yahoo.com/guest-lineups-
      sunday-news-shows-183815643.html
-      2013-09-28 07:37:02 | Security tight at Miss World crowning
      in Indonesia     | http://news.yahoo.com/security-tight-miss-
      -world-crowning-indonesia-113634310.html
10     2013-09-27 20:49:32 | Simons stamps his natural mark on
      Dior              | http://news.yahoo.com/simons-stamps-
      natural-mark-dior-223848528.html
-      2013-09-27 19:50:30 | Jackson jury ends deliberations until
      Tuesday          | http://news.yahoo.com/jackson-jury-ends-
      deliberations-until-tuesday-235030969.html
-      2013-09-27 19:23:40 | Eric Clapton-owned Richter painting
      to sell in NYC | http://news.yahoo.com/eric-clapton-owned-
      -richter-painting-sell-nyc-201447252.html
-      2013-09-27 19:14:15 | Report: Hollywood is less gay-
      friendly off-screen | http://news.yahoo.com/report-
      hollywood-less-gay-friendly-off-screen-231415235.html
-      (10 rows)
```

CSV

Multicorn может использовать CSV файл как источник данных. Как и в прошлые разы, создаем сервер и таблицу для CSV ресурса:

Листинг 10.67 Multicorn

10.9. Multicorn

```
Line 1 # CREATE SERVER csv_srv foreign data wrapper multicorn
        options (
-         wrapper 'multicorn.csvfdw.CsvFdw'
-     );
- CREATE SERVER
5 # CREATE FOREIGN TABLE csvtest (
-     sort_order numeric,
-     common_name character varying,
-     formal_name character varying,
-     main_type character varying,
10     sub_type character varying,
-     sovereignty character varying,
-     capital character varying
- ) server csv_srv options (
-     filename '/var/data/countrylist.csv',
15     skip_header '1',
-     delimiter ',');
- CREATE FOREIGN TABLE
```

Основные опции:

- `filename` (`string`) — полный путь к CSV файлу;
- `delimiter` (`character`) — разделитель в CSV файле (по умолчанию «,»);
- `quotechar` (`character`) — кавычки в CSV файле;
- `skip_header` (`integer`) — число строк, которые необходимо пропустить (по умолчанию 0).

Результат таблицы «csvtest»:

Листинг 10.68 Multicorn

```
Line 1 # SELECT * FROM csvtest LIMIT 10;
- sort_order | common_name | formal_name
-           | main_type   | sub_type |
- sovereignty | capital
- --
- -----+-----+-----
-           1 | Afghanistan | Islamic State of
Afghanistan | Independent State |
-           | Kabul
5           2 | Albania | Republic of Albania
-           | Independent State |
-           | Tirana
-           3 | Algeria | People's Democratic
Republic of Algeria | Independent State |
-           | Algiers
-           4 | Andorra | Principality of Andorra
-           | Independent State |
-           | Andorra la Vella
```

10.9. Multicorn

```
-          5 | Angola                | Republic of Angola
           | Independent State |
| Luanda
-          6 | Antigua and Barbuda |
           | Independent State |
| Saint John's
10         7 | Argentina              | Argentine Republic
           | Independent State |
| Buenos Aires
-          8 | Armenia                | Republic of Armenia
           | Independent State |
| Yerevan
-          9 | Australia              | Commonwealth of Australia
           | Independent State |
| Canberra
-         10 | Austria                | Republic of Austria
           | Independent State |
| Vienna
- (10 rows)
```

Другие FDW

Multicorn также содержит FDW для LDAP и файловой системы. LDAP FDW может использоваться для доступа к серверам по [LDAP протоколу](#). FDW для файловой системы может быть использован для доступа к данным, хранящимся в различных файлах в файловой системе.

Собственный FDW

Multicorn предоставляет простой интерфейс для написания собственных FDW. Более подробную информацию вы можете найти по [этой ссылке](#).

PostgreSQL 9.3+

В PostgreSQL 9.1 и 9.2 была представлена реализация FDW только на чтение. Начиная с версии 9.3, FDW может писать во внешние источники данных. Сейчас Multicorn поддерживает запись данных в другие источники, начиная с версии 1.0.0.

Заключение

Multicorn — расширение для PostgreSQL, которое позволяет использовать встроенные FDW или создавать собственные на Python.

10.10 Pgaudit

Pgaudit — расширение для PostgreSQL, которое позволяет собирать события из различных источников внутри PostgreSQL и записывает их в формате CSV с временной меткой, информацией о пользователе, объекте, который был затронут командой (если такое произошло) и полным текстом команды. Поддерживает все DDL, DML (включая **SELECT**) и прочие команды. Данное расширение работает в PostgreSQL 9.3 и выше.

После установки расширения нужно добавить в конфиг PostgreSQL настройки расширения:

Листинг 10.69 Pgaudit

```
Line 1 shared_preload_libraries = 'pgaudit'
-
- pgaudit.log = 'read, write, user'
```

Далее перегрузить базу данных и установить расширение для базы:

Листинг 10.70 Pgaudit

```
Line 1 # CREATE EXTENSION pgaudit;
```

После этого в логах можно увидеть подобный результат от pgaudit:

Листинг 10.71 Pgaudit

```
Line 1 LOG:  [AUDIT],2014-04-30 17:13:55.202854+09,auditdb,ianb,
      ianb,DEFINITION,CREATE TABLE,TABLE,public.x,CREATE TABLE
      public.x (a pg_catalog.int4 , b pg_catalog.int4 )
      WITH (oids=OFF)
- LOG:  [AUDIT],2014-04-30 17:14:06.548923+09,auditdb,ianb,
      ianb,WRITE,INSERT,TABLE,public.x,INSERT INTO x VALUES
      (1,1);
- LOG:  [AUDIT],2014-04-30 17:14:21.221879+09,auditdb,ianb,
      ianb,READ,SELECT,TABLE,public.x,SELECT * FROM x;
- LOG:  [AUDIT],2014-04-30 17:15:25.620213+09,auditdb,ianb,
      ianb,READ,SELECT,VIEW,public.v_x,SELECT * from v_x;
5 LOG:  [AUDIT],2014-04-30 17:15:25.620262+09,auditdb,ianb,
      ianb,READ,SELECT,TABLE,public.x,SELECT * from v_x;
- LOG:  [AUDIT],2014-04-30 17:16:00.849868+09,auditdb,ianb,
      ianb,WRITE,UPDATE,TABLE,public.x,UPDATE x SET a=a+1;
- LOG:  [AUDIT],2014-04-30 17:16:18.291452+09,auditdb,ianb,
      ianb,ADMIN,VACUUM,,VACUUM x;
- LOG:  [AUDIT],2014-04-30 17:18:01.08291+09,auditdb,ianb,ianb,
      ,DEFINITION,CREATE FUNCTION,FUNCTION,public.func_x(),
      CREATE FUNCTION public.func_x() RETURNS pg_catalog.int4
      LANGUAGE sql VOLATILE CALLED ON NULL INPUT SECURITY
      INVOKER COST 100.000000 AS $dprs_$SELECT a FROM x LIMIT
      1;$dprs_$
```

Более подробную информацию про настройку расширения можно найти в официальном [README](#).

10.11 Ltree

Ltree — расширение, которое позволяет хранить древовидные структуры в виде меток, а также предоставляет широкие возможности поиска по ним.

Почему Ltree?

- Реализация алгоритма Materialized Path (довольно быстра, как на запись, так и на чтение);
- Как правило данное решение будет быстрее, чем использование СТЕ (Common Table Expressions) или рекурсивной функции (постоянно будут пересчитываться ветвления);
- Встроены механизмы поиска по дереву;
- Индексы;

Установка и использование

Для начала активируем расширение для базы данных:

Листинг 10.72 Ltree

```
Line 1 # CREATE EXTENSION ltree;
```

Далее создадим таблицу комментариев, которые будут храниться как дерево:

Листинг 10.73 Ltree

```
Line 1 CREATE TABLE comments (user_id integer, description text,  
    path ltree);  
- INSERT INTO comments (user_id, description, path) VALUES (  
    1, md5(random()::text), '0001');  
- INSERT INTO comments (user_id, description, path) VALUES (  
    2, md5(random()::text), '0001.0001.0001');  
- INSERT INTO comments (user_id, description, path) VALUES (  
    2, md5(random()::text), '0001.0001.0001.0001');  
5 INSERT INTO comments (user_id, description, path) VALUES (  
    1, md5(random()::text), '0001.0001.0001.0002');  
- INSERT INTO comments (user_id, description, path) VALUES (  
    5, md5(random()::text), '0001.0001.0001.0003');  
- INSERT INTO comments (user_id, description, path) VALUES (  
    6, md5(random()::text), '0001.0002');  
- INSERT INTO comments (user_id, description, path) VALUES (  
    6, md5(random()::text), '0001.0002.0001');
```

```

- INSERT INTO comments (user_id, description, path) VALUES (
  6, md5(random()::text), '0001.0003');
10 INSERT INTO comments (user_id, description, path) VALUES (
  8, md5(random()::text), '0001.0003.0001');
- INSERT INTO comments (user_id, description, path) VALUES (
  9, md5(random()::text), '0001.0003.0002');
- INSERT INTO comments (user_id, description, path) VALUES (
  11, md5(random()::text), '0001.0003.0002.0001');
- INSERT INTO comments (user_id, description, path) VALUES (
  2, md5(random()::text), '0001.0003.0002.0002');
- INSERT INTO comments (user_id, description, path) VALUES (
  5, md5(random()::text), '0001.0003.0002.0003');
15 INSERT INTO comments (user_id, description, path) VALUES (
  7, md5(random()::text), '0001.0003.0002.0002.0001');
- INSERT INTO comments (user_id, description, path) VALUES (
  20, md5(random()::text), '0001.0003.0002.0002.0002');
- INSERT INTO comments (user_id, description, path) VALUES (
  31, md5(random()::text), '0001.0003.0002.0002.0003');
- INSERT INTO comments (user_id, description, path) VALUES (
  22, md5(random()::text), '0001.0003.0002.0002.0004');
- INSERT INTO comments (user_id, description, path) VALUES (
  34, md5(random()::text), '0001.0003.0002.0002.0005');
20 INSERT INTO comments (user_id, description, path) VALUES (
  22, md5(random()::text), '0001.0003.0002.0002.0006');

```

Не забываем добавить индексы:

Листинг 10.74 Ltree

```

Line 1 # CREATE INDEX path_gist_comments_idx ON comments USING GIST
      (path);
- # CREATE INDEX path_comments_idx ON comments USING btree(
      path);

```

В данном примере я создаю таблицу `comments` с полем `path`, которое и будет содержать полный путь к этому комментарию в дереве (я использую 4 цифры и точку для делителя узлов дерева). Для начала найдем все комментарии, у которых путь начинается с `0001.0003`:

Листинг 10.75 Ltree

```

Line 1 # SELECT user_id, path FROM comments WHERE path <@ '
      0001.0003';
-   user_id |          path
-   -----+-----
-         6 | 0001.0003
5         8 | 0001.0003.0001
-         9 | 0001.0003.0002
-        11 | 0001.0003.0002.0001
-         2 | 0001.0003.0002.0002

```

10.11. Ltree

```
-      5 | 0001.0003.0002.0003
10     7 | 0001.0003.0002.0002.0001
-     20 | 0001.0003.0002.0002.0002
-     31 | 0001.0003.0002.0002.0003
-     22 | 0001.0003.0002.0002.0004
-     34 | 0001.0003.0002.0002.0005
15    22 | 0001.0003.0002.0002.0006
- (12 rows)
```

И проверим как работают индексы:

Листинг 10.76 Ltree

```
Line 1 # SET enable_seqscan=false ;
- SET
- # EXPLAIN ANALYZE SELECT user_id , path FROM comments WHERE
-   path <@ '0001.0003' ;
-
-   QUERY PLAN
5  --
-   -----
-
-   Index Scan using path_gist_comments_idx on comments (cost
-     =0.00..8.29 rows=2 width=38) (actual time=0.023..0.034
-     rows=12 loops=1)
-     Index Cond: (path <@ '0001.0003'::ltree)
-   Total runtime: 0.076 ms
- (3 rows)
```

Данную выборку можно сделать другим запросом:

Листинг 10.77 Ltree

```
Line 1 # SELECT user_id , path FROM comments WHERE path ~ '
-   0001.0003.*' ;
-   user_id |          path
-   -----+-----
-           6 | 0001.0003
5           8 | 0001.0003.0001
-           9 | 0001.0003.0002
-          11 | 0001.0003.0002.0001
-           2 | 0001.0003.0002.0002
-           5 | 0001.0003.0002.0003
10          7 | 0001.0003.0002.0002.0001
-          20 | 0001.0003.0002.0002.0002
-          31 | 0001.0003.0002.0002.0003
-          22 | 0001.0003.0002.0002.0004
-          34 | 0001.0003.0002.0002.0005
15          22 | 0001.0003.0002.0002.0006
- (12 rows)
```

Не забываем про сортировку дерева:

Листинг 10.78 Ltree

```

Line 1 # INSERT INTO comments (user_id, description, path) VALUES (
      9, md5(random()::text), '0001.0003.0001.0001');
- # INSERT INTO comments (user_id, description, path) VALUES (
      9, md5(random()::text), '0001.0003.0001.0002');
- # INSERT INTO comments (user_id, description, path) VALUES (
      9, md5(random()::text), '0001.0003.0001.0003');
- # SELECT user_id, path FROM comments WHERE path ~ '
      0001.0003.*';
5  user_id |          path
-  -----+-----
-         6 | 0001.0003
-         8 | 0001.0003.0001
-         9 | 0001.0003.0002
10        11 | 0001.0003.0002.0001
-         2 | 0001.0003.0002.0002
-         5 | 0001.0003.0002.0003
-         7 | 0001.0003.0002.0002.0001
-        20 | 0001.0003.0002.0002.0002
15        31 | 0001.0003.0002.0002.0003
-        22 | 0001.0003.0002.0002.0004
-        34 | 0001.0003.0002.0002.0005
-        22 | 0001.0003.0002.0002.0006
-         9 | 0001.0003.0001.0001
20         9 | 0001.0003.0001.0002
-         9 | 0001.0003.0001.0003
- (15 rows)
- # SELECT user_id, path FROM comments WHERE path ~ '
      0001.0003.*' ORDER by path;
-  user_id |          path
25  -----+-----
-         6 | 0001.0003
-         8 | 0001.0003.0001
-         9 | 0001.0003.0001.0001
-         9 | 0001.0003.0001.0002
30         9 | 0001.0003.0001.0003
-         9 | 0001.0003.0002
-        11 | 0001.0003.0002.0001
-         2 | 0001.0003.0002.0002
-         7 | 0001.0003.0002.0002.0001
35        20 | 0001.0003.0002.0002.0002
-        31 | 0001.0003.0002.0002.0003
-        22 | 0001.0003.0002.0002.0004
-        34 | 0001.0003.0002.0002.0005
-        22 | 0001.0003.0002.0002.0006

```

10.11. Ltree

```
40      5 | 0001.0003.0002.0003
- (15 rows)
```

Для поиска можно использовать разные модификаторы. Пример использования «или» (|):

Листинг 10.79 Ltree

```
Line 1 # SELECT user_id, path FROM comments WHERE path ~ '
      0001.*{1,2}.0001|0002.*' ORDER by path;
-   user_id |          path
-   -----+-----
-         2 | 0001.0001.0001
5         2 | 0001.0001.0001.0001
-         1 | 0001.0001.0001.0002
-         5 | 0001.0001.0001.0003
-         6 | 0001.0002.0001
-         8 | 0001.0003.0001
10        9 | 0001.0003.0001.0001
-         9 | 0001.0003.0001.0002
-         9 | 0001.0003.0001.0003
-         9 | 0001.0003.0002
-        11 | 0001.0003.0002.0001
15         2 | 0001.0003.0002.0002
-         7 | 0001.0003.0002.0002.0001
-        20 | 0001.0003.0002.0002.0002
-        31 | 0001.0003.0002.0002.0003
-        22 | 0001.0003.0002.0002.0004
20        34 | 0001.0003.0002.0002.0005
-        22 | 0001.0003.0002.0002.0006
-         5 | 0001.0003.0002.0003
- (19 rows)
```

Например, найдем прямых потомков от 0001.0003:

Листинг 10.80 Ltree

```
Line 1 # SELECT user_id, path FROM comments WHERE path ~ '
      0001.0003.*{1}' ORDER by path;
-   user_id |          path
-   -----+-----
-         8 | 0001.0003.0001
5         9 | 0001.0003.0002
- (2 rows)
```

Можно также найти родителя для потомка «0001.0003.0002.0002.0005»:

Листинг 10.81 Ltree

```
Line 1 # SELECT user_id, path FROM comments WHERE path = subpath('
      0001.0003.0002.0002.0005', 0, -1) ORDER by path;
-   user_id |          path
```

10.12. PostPic

```
- ----+-----  
-          2 | 0001.0003.0002.0002  
5 (1 row)
```

Заключение

Ltree — расширение, которое позволяет хранить и удобно управлять Materialized Path в PostgreSQL.

10.12 PostPic

PostPic - расширение для PostgreSQL, которое позволяет обрабатывать изображения в базе данных, как PostGIS делает это с пространственными данными. Он добавляет новый типа поля `image`, а также несколько функций для обработки изображений (обрезка краев, создание миниатюр, поворот и т. д.) и извлечений его атрибутов (размер, тип, разрешение). Более подробно о возможностях расширения можно ознакомиться на [официальной странице](#).

10.13 Fuzzystmatch

Fuzzystmatch расширение предоставляет несколько функций для определения сходства и расстояния между строками. Функция `soundex` используется для согласования сходно звучащих имен путем преобразования их в одинаковый код. Функция `difference` преобразует две строки в `soundex` код, а затем сообщает количество совпадающих позиций кода. В `soundex` код состоит из четырех символов, поэтому результат будет от нуля до четырех: 0 — не совпадают, 4 — точное совпадение (таким образом, функция названа неверно — как название лучше подходит `similarity`):

Листинг 10.82 soundex

```
Line 1 # CREATE EXTENSION fuzzystmatch;  
- CREATE EXTENSION  
- # SELECT soundex('hello world!');  
- soundex  
5 -----  
- H464  
- (1 row)  
-  
- # SELECT soundex('Anne'), soundex('Ann'), difference('Anne',  
- 'Ann');  
10 soundex | soundex | difference  
- ----+-----+-----  
- A500 | A500 | 4
```

10.13. Fuzzystmatch

```
- (1 row)
-
15 # SELECT soundex('Anne'), soundex('Andrew'), difference('
      Anne', 'Andrew');
-   soundex | soundex | difference
-   -----+-----+-----
-   A500    | A536    |          2
- (1 row)
20
- # SELECT soundex('Anne'), soundex('Margaret'), difference('
      Anne', 'Margaret');
-   soundex | soundex | difference
-   -----+-----+-----
-   A500    | M626    |          0
25 (1 row)
-
- # CREATE TABLE s (nm text);
- CREATE TABLE
- # INSERT INTO s VALUES ('john'), ('joan'), ('wobbly'), ('
      jack');
30 INSERT 0 4
- # SELECT * FROM s WHERE soundex(nm) = soundex('john');
-   nm
-   -----
-   john
35   joan
- (2 rows)
-
- # SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
-   nm
40 -----
-   john
-   joan
-   jack
- (3 rows)
```

Функция `levenshtein` вычисляет **расстояние Левенштейна** между двумя строками. `levenshtein_less_equal` ускоряет функцию `levenshtein` для малых значений расстояния:

Листинг 10.83 levenshtein

```
Line 1 # SELECT levenshtein('GUMBO', 'GAMBOL');
-   levenshtein
-   -----
-           2
5 (1 row)
-
- # SELECT levenshtein('GUMBO', 'GAMBOL', 2, 1, 1);
```


10.14. Pg_trgm

```
- levenshtein
- -----
10          3
- (1 row)
-
- # SELECT levenshtein_less_equal('extensive', 'exhaustive',
-     2);
- levenshtein_less_equal
15 -----
-          3
- (1 row)
-
- test=# SELECT levenshtein_less_equal('extensive', '
-     exhaustive', 4);
20 levenshtein_less_equal
- -----
-          4
- (1 row)
```

Функция `metaphone`, как и `soundex`, построена на идее создания кода для строки: две строки, которые будут считаться похожими, будут иметь одинаковые коды. Последним параметром указывается максимальная длина `metaphone` кода. Функция `dmetaphone` вычисляет два «как звучит» кода для строки — «первичный» и «альтернативный»:

Листинг 10.84 metaphone

```
Line 1 # SELECT metaphone('GUMBO', 4);
- metaphone
- -----
- KM
5 (1 row)
- # SELECT dmetaphone('postgresl');
- dmetaphone
- -----
- PSTK
10 (1 row)
-
- # SELECT dmetaphone_alt('postgresl');
- dmetaphone_alt
- -----
15 PSTK
- (1 row)
```

10.14 Pg_trgm

Автодополнение — функция в программах, предусматривающих интерактивный ввод текста по дополнению текста по введённой его части.

Реализуется это простым `LIKE 'some%'` запросом в базу, где «some» — то, что пользователь успел ввести в поле ввода. Проблема в том, что в огромной таблице такой запрос будет работать очень медленно. Для ускорения запроса типа `LIKE 'bla%'` можно использовать `text_pattern_ops` для `text` поля или `varchar_pattern_ops` для `varchar` поля класс операторов в определении индекса (данные типы индексов не будут работать для стандартных операторов `<`, `<=`, `=>`, `>` и для работы с ними придется создать обычный `btree` индекс).

Листинг 10.85 `text_pattern_ops`

```
Line 1 # create table tags (  
- # tag text primary key,  
- # name text not null,  
- # shortname text,  
5 # status char default 'S',  
- #  
- # check( status in ('S', 'R') )  
- # );  
- NOTICE: CREATE TABLE / PRIMARY KEY will create implicit  
index "tags_pkey" for table "tags"  
10 CREATE TABLE  
-  
- # CREATE INDEX i_tag ON tags USING btree(lower(tag)  
text_pattern_ops);  
- CREATE INDEX  
-  
15 # EXPLAIN ANALYZE select * from tags where lower(tag) LIKE  
lower('0146%');  
-  
- QUERY  
-  
- PLAN  
- --  
-----  
- Bitmap Heap Scan on tags (cost=5.49..97.75 rows=121 width  
=26) (actual time=0.025..0.025 rows=1 loops=1)  
- Filter: (lower(tag) ~~ '0146%'::text)  
20 -> Bitmap Index Scan on i_tag (cost=0.00..5.46 rows=120  
width=0) (actual time=0.016..0.016 rows=1 loops=1)  
- Index Cond: ((lower(tag) ~>= '0146'::text) AND (  
lower(tag) ~< '0147'::text))  
- Total runtime: 0.050 ms  
- (5 rows)
```

Для более сложных вариантов поиска, таких как `LIKE '%some%'` или `LIKE 'so%me%'` такой индекс не будет работать, но эту проблему можно решить через расширение.

`Pg_trgm` — PostgreSQL расширение, которое предоставляет функции и

операторы для определения схожести алфавитно-цифровых строк на основе триграмм, а также классы операторов индексов, поддерживающие быстрый поиск схожих строк. Триграмма — это группа трёх последовательных символов, взятых из строки. Можно измерить схожесть двух строк, подсчитав число триграмм, которые есть в обеих. Эта простая идея оказывается очень эффективной для измерения схожести слов на многих естественных языках. Модуль `pg_trgm` предоставляет классы операторов индексов GiST и GIN, позволяющие создавать индекс по текстовым колонкам для очень быстрого поиска по критерию схожести. Эти типы индексов поддерживают `%` и `<->` операторы схожести и дополнительно поддерживают поиск на основе триграмм для запросов с `LIKE`, `ILIKE`, `~` и `~*` (эти индексы не поддерживают простые операторы сравнения и равенства, так что может понадобиться и обычный `btree` индекс).

Листинг 10.86 `pg_trgm`

```
Line 1 # CREATE TABLE test_trgm (t text);
- # CREATE INDEX trgm_idx ON test_trgm USING gist (t
-   gist_trgm_ops);
- -- or
- # CREATE INDEX trgm_idx ON test_trgm USING gin (t
-   gin_trgm_ops);
```

После создания GiST или GIN индекса по колонке `t` можно осуществлять поиск по схожести. Пример запроса:

Листинг 10.87 `pg_trgm`

```
Line 1 SELECT t, similarity(t, 'word') AS sml
- FROM test_trgm
- WHERE t % 'word'
- ORDER BY sml DESC, t;
```

Он выдаст все значения в текстовой колонке, которые достаточно схожи со словом `word`, в порядке сортировки от наиболее к наименее схожим. Другой вариант предыдущего запроса (может быть довольно эффективно выполнен с применением индексов GiST, а не GIN):

Листинг 10.88 `pg_trgm`

```
Line 1 SELECT t, t <-> 'word' AS dist
- FROM test_trgm
- ORDER BY dist LIMIT 10;
```

Начиная с PostgreSQL 9.1, эти типы индексов также поддерживают поиск с операторами `LIKE` и `ILIKE`, например:

Листинг 10.89 `pg_trgm`

```
Line 1 SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
```

Начиная с PostgreSQL 9.3, индексы этих типов также поддерживают поиск по регулярным выражениям (операторы `~` и `~*`), например:

Листинг 10.90 pg_trgm

```
Line 1 SELECT * FROM test_trgm WHERE t ~ '(foo|bar)';
```

Относительно поиска по регулярному выражению или с `LIKE`, нужно принимать в расчет, что при отсутствии триграмм в искомом шаблоне поиск сводится к полному сканированию индекса. Выбор между индексами GiST и GIN зависит от относительных характеристик производительности GiST и GIN, которые здесь не рассматриваются. Как правило, индекс GIN быстрее индекса GiST при поиске, но строится или обновляется он медленнее; поэтому GIN лучше подходит для статических, а GiST для часто изменяемых данных.

10.15 Cstore_fdw

`Cstore_fdw` расширение реализует модель хранения данных на базе **семейства столбцов** (column-oriented systems) для PostgreSQL (колоночное хранение данных). Такое хранение данных обеспечивает заметные преимущества для аналитических задач (**OLAP**, **data warehouse**), поскольку требуется считывать меньше данных с диска (благодаря формату хранения и компрессии). Расширение использует **Optimized Row Columnar (ORC)** формат для размещения данных на диске, который имеет следующие преимущества:

- Уменьшение (сжатие) размера данных в памяти и на диске в 2-4 раза. Можно добавить в расширение другой кодек для сжатия (алгоритм Лемпеля-Зива, LZ присутствует в расширении);
- Считывание с диска только тех данных, которые требуются. Повышается производительность по I/O диска для других запросов;
- Хранение минимального/максимального значений для групп полей (skip index, индекс с пропусками), что помогает пропустить не требуемые данные на диске при выборке;

Установка и использование

Для работы `cstore_fdw` требуется `protobuf-c` для сериализации и десериализации данных. Далее требуется добавить в `postgresql.conf` расширение:

Листинг 10.91 Cstore_fdw

```
Line 1 shared_preload_libraries = 'cstore_fdw'
```

И активировать его для базы:

Листинг 10.92 Cstore_fdw

```
Line 1 # CREATE EXTENSION cstore_fdw;
```

Для загрузки данных в cstore таблицы существует два варианта:

10.15. Cstore_fdw

- Использование команды `COPY` для загрузки или добавления данных из файлов или `STDIN`;
- Использование конструкции `INSERT INTO cstore_table SELECT ...` для загрузки или добавления данных из другой таблицы;

Cstore таблицы не поддерживают `INSERT` (кроме выше упомянутого `INSERT INTO ... SELECT`), `UPDATE` или `DELETE` команды.

Для примера загрузим тестовые данные:

Листинг 10.93 Cstore_fdw

```
Line 1 $ wget http://examples.citusdata.com/customer_reviews_1998.csv.gz
- $ wget http://examples.citusdata.com/customer_reviews_1999.csv.gz
-
- $ gzip -d customer_reviews_1998.csv.gz
5 $ gzip -d customer_reviews_1999.csv.gz
```

Далее загрузим эти данные в cstore таблицу (расширение уже активировано для PostgreSQL):

Листинг 10.94 Cstore таблицы

```
Line 1 -- create server object
- CREATE SERVER cstore_server FOREIGN DATA WRAPPER cstore_fdw;
-
- -- create foreign table
5 CREATE FOREIGN TABLE customer_reviews
- (
-     customer_id TEXT,
-     review_date DATE,
-     review_rating INTEGER,
10     review_votes INTEGER,
-     review_helpful_votes INTEGER,
-     product_id CHAR(10),
-     product_title TEXT,
-     product_sales_rank BIGINT,
15     product_group TEXT,
-     product_category TEXT,
-     product_subcategory TEXT,
-     similar_product_ids CHAR(10) []
- )
20 SERVER cstore_server
- OPTIONS(compression 'pglz');
-
- COPY customer_reviews FROM '/tmp/customer_reviews_1998.csv'
- WITH CSV;
- COPY customer_reviews FROM '/tmp/customer_reviews_1999.csv'
- WITH CSV;
```



```

- GROUP BY
-     title_length_bucket
- ORDER BY
-     title_length_bucket;
35 title_length_bucket | review_average | count
- -----+-----+-----
-           1 |           4.26 | 139034
-           2 |           4.24 | 411318
-           3 |           4.34 | 245671
40           4 |           4.32 | 167361
-           5 |           4.30 | 118422
-           6 |           4.40 | 116412
- (6 rows)
45 Time: 1285.059 ms

```

Заключение

Более подробно об использовании расширения можно ознакомиться через [официальную документацию](#).

10.16 Postgresql-hll

На сегодняшний день широко распространена задача подсчета количества уникальных элементов (count-distinct problem) в потоке данных, которые могут содержать повторяющиеся элементы. Например, сколько уникальных IP-адресов подключалось к серверу за последний час? Сколько различных слов в большом куске текстов? Какое количество уникальных посетителей побывало на популярном сайте за день? Сколько уникальных URL было запрошено через прокси-сервер? Данную задачу можно решить «в лоб»: пройтись по всем элементам и убрать дубликаты, после этого посчитать их количество (например использовать множество, set). Трудности в таком подходе возникают при увеличении масштаба. С минимальными затратами можно подсчитать тысячу или даже миллион уникальных посетителей, IP-адресов, URL или слов. А что если речь идет о 100 миллионах уникальных элементов на один сервер при наличии тысяч серверов? Теперь это уже становится интересным.

Текущее решение проблемы будет выглядеть так: необходимо сформировать множества (set) уникальных элементов для каждого из 1000 серверов, каждое из которых может содержать около 100 миллионов уникальных элементов, а затем подсчитать количество уникальных элементов в объединении этих множеств. Другими словами, мы имеем дело с распределенным вариантом задачи подсчета уникальных элементов. Хотя это решение является вполне логичным, на практике этот подход обойдется

высокой ценой. Для примера возьмем URL, средняя длина которого составляет 76 символов. В нашем случае один сервер обслуживает около 100 миллионов уникальных URL, следовательно, размер файла с их перечнем составит около 7.6 ГБ. Даже если каждый URL преобразовать в 64-битный хеш, размер файла составит 800 МБ. Это намного лучше, но не забывайте, что речь идет о 1000 серверов. Каждый сервер отправляет файл с перечнем уникальных URL на центральный сервер, следовательно, при наличии 1000 серверов функция объединения множеств должна обрабатывать 800 ГБ данных. Если такая операция должна выполняться часто, тогда необходимо будет либо установить систему для обработки больших данных (и нанять команду для ее обслуживания), либо найти другое решение.

И вот на сцену выходит **HyperLogLog** алгоритм. Этот алгоритм реализует вероятностный подход к задаче подсчета уникальных элементов и базируется на двух следующих положениях:

- вероятность того, что любой данный бит двоичного представления случайного числа равен единице, составляет 50%;
- вероятность того, что совместно произойдут два независимых случайных события A и B , вычисляется по формуле $P(A) * P(B)$. Таким образом, если вероятность равенства единице одного любого бита случайного числа составляет 50%, тогда вероятность равенства единице двух любых битов составляет 25%, трех — 12,5% и т.д;

Вспомним еще одно базовое положение теории вероятностей, согласно которому ожидаемое количество испытаний, необходимое для наступления события, вычисляется по формуле $1/P(event)$. Следовательно, если $P(one\ specific\ bit\ set) = 50\%$, то ожидаемое количество испытаний равно 2. Для двух битов — 4, для трех битов — 8 и т. д.

В общем случае входные значения не являются равномерно распределенными случайными числами, поэтому необходим способ преобразования входных значений к равномерному распределению, т. е. необходима хеш-функция. Обратите внимание, в некоторых случаях распределение, получаемое на выходе хеш-функции, не оказывает существенное влияние на точность системы. Однако HyperLogLog очень чувствителен в этом отношении. Если выход хеш-функции не соответствует равномерному распределению, алгоритм теряет точность, поскольку не выполняются базовые допущения, лежащие в его основе.

Рассмотрим алгоритм подробно. Вначале необходимо хешировать все элементы исследуемого набора. Затем нужно подсчитать количество последовательных начальных битов, равных единице, в двоичном представлении каждого хеша и определить максимальное значение этого количества среди всех хешей. Если максимальное количество единиц обозначить n , тогда количество уникальных элементов в наборе можно оценить, как

2^n . То есть, если максимум один начальный бит равен единице, тогда количество уникальных элементов, в среднем, равно 2; если максимум три начальных бита равны единице, в среднем, мы можем ожидать 8 уникальных элементов и т. д.

Подход, направленный на повышение точности оценки и являющийся одной из ключевых идей HyperLogLog, заключается в следующем: разделяем хеши на подгруппы на основании их конечных битов, определяем максимальное количество начальных единиц в каждой подгруппе, а затем находим среднее. Этот подход позволяет получить намного более точную оценку общего количества уникальных элементов. Если мы имеем m подгрупп и n уникальных элементов, тогда, в среднем, в каждой подгруппе будет n/m уникальных элементов. Таким образом, нахождение среднего по всем подгруппам дает достаточно точную оценку величины $\log_2(n/m)$, а отсюда легко можно получить необходимое нам значение. Более того, HyperLogLog позволяет обрабатывать по отдельности различные варианты группировок, а затем на основе этих данных находить итоговую оценку. Следует отметить, что для нахождения среднего HyperLogLog использует среднее гармоническое, которое обеспечивает лучшие результаты по сравнению со средним арифметическим (более подробную информацию можно найти в оригинальных публикациях, посвященных [LogLog](#) и [HyperLogLog](#)).

Вернемся к задаче. По условию существует 1000 серверов и 100 миллионов уникальных URL на каждый сервер, следовательно, центральный сервер должен обрабатывать 800 ГБ данных при каждом выполнении простого варианта алгоритма. Это также означает, что 800 ГБ данных каждый раз необходимо передавать по сети. HyperLogLog меняет ситуацию кардинально. Согласно анализу, проведенному авторами оригинальной публикации, HyperLogLog обеспечивает точность около 98% при использовании всего 1.5 КБ памяти. Каждый сервер формирует соответствующий файл размером 1.5 КБ, а затем отправляет его на центральный сервер. При наличии 1000 серверов, центральный сервер обрабатывает всего 1.5 МБ данных при каждом выполнении алгоритма. Другими словами, обрабатывается лишь 0.0002% данных по сравнению с предыдущим решением. Это полностью меняет экономический аспект задачи. Благодаря HyperLogLog, возможно выполнять такие операции чаще и в большем количестве. И все это ценой всего лишь 2% погрешности.

Для работы с этим алгоритмом внутри PostgreSQL было создано расширение [postgresql-hll](#). Оно добавляет новый тип поля [hll](#), который представляет собой HyperLogLog структуру данных. Рассмотрим пример его использования.

Установка и использование

Для начала инициализируем расширение в базе данных:

Листинг 10.96 Инициализация hll

```
Line 1 # CREATE EXTENSION hll;
```

Давайте предположим, что есть таблица `users_visits`, которая записывает визиты пользователей на сайт, что они сделали и откуда они пришли. В таблице сотни миллионов строк.

Листинг 10.97 `users_visits`

```
Line 1 CREATE TABLE users_visits (
-   date           date ,
-   user_id        integer ,
-   activity_type  smallint ,
5   referrer       varchar(255)
- );
```

Требуется получать очень быстро представление о том, сколько уникальных пользователей посещают сайт в день на админ панели. Для этого создадим агрегатную таблицу:

Листинг 10.98 `daily_uniques`

```
Line 1 CREATE TABLE daily_uniques (
-   date           date UNIQUE,
-   users          hll
- );
5
- -- Fill it with the aggregated unique statistics
- INSERT INTO daily_uniques(date , users)
-   SELECT date , hll_add_agg(hll_hash_integer(user_id))
-   FROM users_visits
10  GROUP BY 1;
```

Далее хэшируется `user_id` и собираются эти хэш-значения в один `hll` за день. Теперь можно запросить информацию по уникальным пользователям за каждый день:

Листинг 10.99 `daily_uniques` по дням

```
Line 1 # SELECT date , hll_cardinality(users) FROM daily_uniques;
-   date           | hll_cardinality
-   -----+-----
-   2017-02-21    |           23123
5   2017-02-22    |           59433
-   2017-02-23    |          2134890
-   2017-02-24    |          3276247
- (4 rows)
```

Можно возразить, что такую задачу можно решить и через `COUNT DISTINCT` и это будет верно. Но в примере только ответили на вопрос: «Сколько уникальных пользователей посещает сайт каждый день?». А

10.17. Tsearch2

что, если требуется знать сколько уникальных пользователей посетили сайт за неделю?

Листинг 10.100 daily_uniques за неделю

```
Line 1 SELECT hll_cardinality(hll_union_agg(users)) FROM
        daily_uniques WHERE date >= '2017-02-20'::date AND date
        <= '2017-02-26'::date;
```

Или выбрать уникальных пользователей за каждый месяц в течение года?

Листинг 10.101 daily_uniques за каждый месяц

```
Line 1 SELECT EXTRACT(MONTH FROM date) AS month, hll_cardinality(
        hll_union_agg(users))
- FROM daily_uniques
- WHERE date >= '2016-01-01' AND
-        date < '2017-01-01'
5 GROUP BY 1;
```

Или узнать количество пользователей, что посетили сайт вчера, но не сегодня?

Листинг 10.102 daily_uniques за вчера но не сегодня

```
Line 1 SELECT date, (#hll_union_agg(users) OVER two_days) - #users
        AS lost_uniques
- FROM daily_uniques
- WINDOW two_days AS (ORDER BY date ASC ROWS 1 PRECEDING);
```

Это всего пара примеров типов запросов, которые будут возвращать результат в течение миллисекунд благодаря `hll`, но потребует либо полностью отдельные предварительно созданные агрегирующие таблицы или `self join/generate_series` фокусы в `COUNT DISTINCT` мире.

Заключение

Более подробно об использовании расширения можно ознакомиться через [официальную документацию](#).

10.17 Tsearch2

Как и многие современные СУБД, PostgreSQL имеет встроенный механизм полнотекстового поиска. Отметим, что операторы поиска по текстовым данным существовали очень давно, это операторы `LIKE`, `ILIKE`, `~`, `~*`. Однако, они не годились для эффективного полнотекстового поиска, так как:

- У них не было лингвистической поддержки, например, при поиске слова `satisfies` будут не найдены документы со словом `satisfy` и никакими регулярными выражениями этому не помочь. В принципе, используя `OR` и все формы слова, можно найти все необходимые документы, но это очень неэффективно, так как в некоторых языках могут быть слова со многими тысячами форм!;
- Они не предоставляют никакой информации для ранжирования (сортировки) документов, что делает такой поиск практически бесполезным, если только не существует другой сортировки или в случае малого количества найденных документов;
- Они, в целом, очень медленные из-за того, что они каждый раз просматривают весь документ и не имеют индексной поддержки;

Для улучшения ситуации Олег Бартунов и Федор Сигаев предложили и реализовали новый полнотекстовый поиск, существовавший как модуль расширения и интегрированный в PostgreSQL, начиная с версии 8.3 — [Tsearch2](#).

Идея нового поиска состояла в том, чтобы затратить время на обработку документа один раз и сохранить время при поиске, использовать специальные программы-словари для нормализации слов, чтобы не заботиться, например, о формах слов, учитывать информацию о важности различных атрибутов документа и положения слова из запроса в документе для ранжирования найденных документов. Для этого, требовалось создать новые типы данных, соответствующие документу и запросу, и полнотекстовый оператор для сравнения документа и запроса, который возвращает `TRUE`, если запрос удовлетворяет запросу, и в противном случае - `FALSE`.

PostgreSQL предоставляет возможность как для создания новых типов данных, операторов, так и создания индексной поддержки для доступа к ним, причем с поддержкой конкурентности и восстановления после сбоев. Однако, надо понимать, что индексы нужны только для ускорения поиска, сам поиск обязан работать и без них. Таким образом, были созданы новые типы данных - `tsvector`, который является хранилищем для лексем из документа, оптимизированного для поиска, и `tsquery` - для запроса с поддержкой логических операций, полнотекстовый оператор «две собаки» `@@` и индексная поддержка для него с использованием GiST и GIN. `tsvector` помимо самих лексем может хранить информацию о положении лексемы в документе и ее весе (важности), которая потом может использоваться для вычисления ранжирующей информации.

Установка и использование

Для начала активируем расширение:

```
Листинг 10.103 Активация tsearch2
```

```
Line 1 # CREATE EXTENSION tsearch2;
```

Проверим его работу:

Листинг 10.104 Проверка tsearch2

```
Line 1 # SELECT 'This is test string'::tsvector;
-         tsvector
- -----
-  'This' 'is' 'string' 'test'
5  (1 row)
-
- # SELECT strip(to_tsvector('The air smells of sea water.'));
-         strip
- -----
10  'air' 'sea' 'smell' 'water'
-  (1 row)
```

Заключение

Данное расширение заслуживает отдельной книги, поэтому лучше ознакомиться с ним подробнее в «[Введение в полнотекстовый поиск в PostgreSQL](#)» документе.

10.18 PL/Proxy

[PL/Proxy](#) представляет собой прокси-язык для удаленного вызова процедур и партиционирования данных между разными базами (шардинг). Подробнее можно почитать в «[6.2 PL/Proxy](#)» главе.

10.19 Texcaller

[Texcaller](#) — это удобный интерфейс для командной строки [TeX](#), который обрабатывает все виды ошибок. Он написан в простом C, довольно портативный и не имеет внешних зависимостей, кроме TeX. Неверный TeX документ обрабатывается путем простого возвращения NULL, а не прерывается с ошибкой. В случае неудачи, а также в случае успеха, дополнительная обработка информации осуществляется через NOTICES.

10.20 Pgmemcache

[Pgmemcache](#) — это PostgreSQL API библиотека на основе libmemcached для взаимодействия с memcached. С помощью данной библиотеки PostgreSQL может записывать, считывать, искать и удалять данные из memcached. Подробнее можно почитать в «[9.2 Pgmemcache](#)» главе.

10.21 Prefix

Prefix реализует поиск текста по префиксу (`prefix @> text`). Prefix используется в приложениях телефонии, где маршрутизация вызовов и расходы зависят от вызывающего/вызываемого префикса телефонного номера оператора.

Установка и использование

Для начала инициализируем расширение в базе данных:

Листинг 10.105 Инициализация prefix

```
Line 1 # CREATE EXTENSION prefix;
```

После этого можем проверить, что расширение функционирует:

Листинг 10.106 Проверка prefix

```
Line 1 # select '123'::prefix_range @> '123456';
-   ?column?
-   -----
-   t
5 (1 row)
-
- # select a, b, a | b as union, a & b as intersect
-   from (select a::prefix_range, b::prefix_range
-           from (values ('123', '123'),
10                  ('123', '124'),
-                   ('123', '123[4-5]'),
-                   ('123[4-5]', '123[2-7]'),
-                   ('123', '[2-3]')) as t(a, b)
-           ) as x;
15
-   a          | b          | union      | intersect
-   -----+-----+-----+-----
-   123         | 123        | 123        | 123
-   123         | 124        | 12[3-4]    |
20  123         | 123[4-5]   | 123        | 123[4-5]
-   123[4-5]    | 123[2-7]   | 123[2-7]   | 123[4-5]
-   123         | [2-3]      | [1-3]      |
- (5 rows)
```

В примере [10.107](#) производится поиск мобильного оператора по номеру телефона:

Листинг 10.107 Проверка prefix

```
Line 1 $ wget https://github.com/dimitri/prefix/raw/master/prefixes
-   .fr.csv
- $ psql
```

10.21. Prefix

```
-
- # create table prefixes (
5     prefix    prefix_range primary key,
-     name      text not null,
-     shortname text,
-     status    char default 'S',
-
10     check( status in ('S', 'R') )
- );
- CREATE TABLE
- # comment on column prefixes.status is 'S: - R: reserved';
- COMMENT
15 # \copy prefixes from 'prefixes.fr.csv' with delimiter ';'
-     csv quote '"'
- COPY 11966
- # create index idx_prefix on prefixes using gist(prefix);
- CREATE INDEX
- # select * from prefixes limit 10;
20 prefix | name
-     | shortname | status
- --
-     +-----+-----+-----+-----+-----+-----+
-
- 010001 | COLT TELECOMMUNICATIONS FRANCE
-     | COLT      | S
- 010002 | EQUANT France
-     | EQFR      | S
- 010003 | NUMERICABLE
-     | NURC      | S
25 010004 | PROSODIE
-     | PROS      | S
- 010005 | INTERNATIONAL TELECOMMUNICATION NETWORK France (
-     Vivaction) | ITNF      | S
- 010006 | SOCIETE FRANCAISE DU RADIOTELEPHONE
-     | SFR       | S
- 010007 | SOCIETE FRANCAISE DU RADIOTELEPHONE
-     | SFR       | S
- 010008 | BJT PARTNERS
-     | BJTP      | S
30 010009 | LONG PHONE
-     | LGPH      | S
- 010010 | IPNOTIC TELECOM
-     | TLNW      | S
- (10 rows)
-
- # select * from prefixes where prefix @> '0146640123';
35 prefix | name      | shortname | status
```

10.22. Dblink

```
- -----+-----+-----+-----
- 0146   | FRANCE TELECOM | FRTE       | S
- (1 row)
-
40 # select * from prefixes where prefix @> '0100091234';
- prefix | name       | shortname | status
- -----+-----+-----+-----
- 010009 | LONG PHONE | LGPH      | S
- (1 row)
```

Заключение

Более подробно об использовании расширения можно ознакомиться через [официальную документацию](#).

10.22 Dblink

Dblink – расширение, которое позволяет выполнять запросы к удаленным базам данных непосредственно из SQL, не прибегая к помощи внешних скриптов.

Установка и использование

Для начала инициализируем расширение в базе данных:

Листинг 10.108 Инициализация dblink

```
Line 1 # CREATE EXTENSION dblink;
```

Для создания подключения к другой базе данных нужно использовать `dblink_connect` функцию, где первым параметром указывается имя подключения, а вторым - опции подключения к базе:

Листинг 10.109 Подключение через dblink

```
Line 1 # SELECT dblink_connect('slave_db', 'host=slave.example.com
port=5432 dbname=exampledb user=admin password=password')
;
- dblink_connect
- -----
- ОК
5 (1 row)
```

При успешном выполнении команды будет выведен ответ «ОК». Теперь можно попробовать считать данные из таблиц через `dblink` функцию:

Листинг 10.110 SELECT

```
Line 1 # SELECT *
```


10.22. Dblink

```
- FROM dblink('slave_db', 'SELECT id, username FROM users
  LIMIT 3')
- AS dblink_users(id integer, username text);
-
5  id |          username
-  ---+-----
-   1 | 64ec7083d7facb7c5d97684e7f415b65
-   2 | 404c3b639a920b5ba814fc01353368f2
-   3 | 153041f992e3eab6891f0e8da9d11f23
10 (3 rows)
```

По завершению работы с сервером, подключение требуется закрыть через функцию `dblink_disconnect`:

Листинг 10.111 dblink_disconnect

```
Line 1 # SELECT dblink_disconnect('slave_db');
-      dblink_disconnect
-      -----
-      ОК
5      (1 row)
```

Курсоры

Dblink поддерживает **курсоры** — инкапсулирующие запросы, которые позволяют получать результат запроса по несколько строк за раз. Одна из причин использования курсоров заключается в том, чтобы избежать переполнения памяти, когда результат содержит большое количество строк.

Для открытия курсора используется функция `dblink_open`, где первый параметр - название подключения, второй - название для курсора, а третий - сам запрос:

Листинг 10.112 dblink_open

```
Line 1 # SELECT dblink_open('slave_db', 'users', 'SELECT id,
  username FROM users');
-      dblink_open
-      -----
-      ОК
5      (1 row)
```

Для получения данных из курсора требуется использовать `dblink_fetch`, где первый параметр - название подключения, второй - название для курсора, а третий - требуемое количество записей из курсора:

Листинг 10.113 dblink_fetch

```
Line 1 # SELECT id, username FROM dblink_fetch('slave_db', 'users',
  2)
- AS (id integer, username text);
```

10.22. Dblink

```
- id | username
- ---+-----
5  1 | 64ec7083d7facb7c5d97684e7f415b65
-  2 | 404c3b639a920b5ba814fc01353368f2
- (2 rows)
-
- # SELECT id, username FROM dblink_fetch('slave_db', 'users',
-      2)
10 AS (id integer, username text);
- id | username
- ---+-----
-  3 | 153041f992e3eab6891f0e8da9d11f23
-  4 | 318c33458b4840f90d87ee4ea8737515
15 (2 rows)
-
- # SELECT id, username FROM dblink_fetch('slave_db', 'users',
-      2)
- AS (id integer, username text);
- id | username
20 ---+-----
-  6 | 5b795b0e73b00220843f82c4d0f81f37
-  8 | c2673ee986c23f62aaeb669c32261402
- (2 rows)
```

После работы с курсором его нужно обязательно закрыть через `dblink_close` функцию:

Листинг 10.114 `dblink_close`

```
Line 1 # SELECT dblink_close('slave_db', 'users');
- dblink_close
- -----
- ОК
5 (1 row)
```

Асинхронные запросы

Последним вариантом для выполнения запросов в `dblink` является асинхронный запрос. При его использовании результаты не будут возвращены до полного выполнения результата запроса. Для создания асинхронного запроса используется `dblink_send_query` функция:

Листинг 10.115 `dblink_send_query`

```
Line 1 # SELECT * FROM dblink_send_query('slave_db', 'SELECT id,
-      username FROM users') AS users;
- users
- -----
- 1
```

5 (1 row)

Результат получается через `dblink_get_result` функцию:

Листинг 10.116 `dblink_get_result`

```
Line 1 # SELECT id, username FROM dblink_get_result('slave_db')
- AS (id integer, username text);
- id | username
- ---+-----
5  1 | 64ec7083d7facb7c5d97684e7f415b65
-  2 | 404c3b639a920b5ba814fc01353368f2
-  3 | 153041f992e3eab6891f0e8da9d11f23
-  4 | 318c33458b4840f90d87ee4ea8737515
-  6 | 5b795b0e73b00220843f82c4d0f81f37
10  8 | c2673ee986c23f62aaeb669c32261402
-  9 | c53f14040fef954cd6e73b9aa2e31d0e
- 10 | 2dbe27fd96cdb39f01ce115cf3c2a517
```

10.23 Postgres_fdw

`Postgres_fdw` — расширение, которое позволяет подключить PostgreSQL к PostgreSQL, которые могут находиться на разных хостах.

Установка и использование

Для начала инициализируем расширение в базе данных:

Листинг 10.117 Инициализация `postgres_fdw`

```
Line 1 # CREATE EXTENSION postgres_fdw;
```

Далее создадим сервер подключений, который будет содержать данные для подключения к другой PostgreSQL базе:

Листинг 10.118 Создание сервера

```
Line 1 # CREATE SERVER slave_db
- FOREIGN DATA WRAPPER postgres_fdw
- OPTIONS (host 'slave.example.com', dbname 'exampledb', port
-         '5432');
```

После этого нужно создать `USER MAPPING`, которое создаёт сопоставление пользователя на внешнем сервере:

Листинг 10.119 `USER MAPPING`

```
Line 1 # CREATE USER MAPPING FOR admin
- SERVER slave_db
- OPTIONS (user 'admin', password 'password');
```

10.23. Postgres_fdw

Теперь можно импортировать таблицы:

Листинг 10.120 Импорт таблицы

```
Line 1 # CREATE FOREIGN TABLE fdw_users (  
-   id                serial ,  
-   username          text not null ,  
-   password          text ,  
5   created_on        timestamptz not null ,  
-   last_logged_on    timestamptz not null  
- )  
- SERVER slave_db  
- OPTIONS (schema_name 'public', table_name 'users');
```

Для того, чтобы не импортировать каждую таблицу отдельно, можно воспользоваться `IMPORT FOREIGN SCHEMA` командой:

Листинг 10.121 Импортируем таблицы

```
Line 1 # IMPORT FOREIGN SCHEMA public  
- LIMIT TO (users, pages)  
- FROM SERVER slave_db INTO fdw;
```

Теперь можно проверить таблицы:

Листинг 10.122 SELECT

```
Line 1 # SELECT * FROM fdw_users LIMIT 1;  
- -[ RECORD 1 ]--+-  
- id           | 1  
- username     | 64ec7083d7facb7c5d97684e7f415b65  
5 password     | b82af3966b49c9ef0f7829107db642bc  
- created_on   | 2017-02-21 05:07:25.619561+00  
- last_logged_on | 2017-02-19 21:03:35.651561+00
```

По умолчанию из таблиц можно не только читать, но и изменять в них данные (`INSERT/UPDATE/DELETE`). `updatable` опция может использоваться для подключения к серверу в режиме «только на чтение»:

Листинг 10.123 Read-only mode

```
Line 1 # ALTER SERVER slave_db  
- OPTIONS (ADD updatable 'false');  
- ALTER SERVER  
- # DELETE FROM fdw_users WHERE id < 10;  
5 ERROR:  foreign table "fdw_users" does not allow deletes
```

Данную опцию можно установить не только на уровне сервера, но и на уровне отдельных таблиц:

Листинг 10.124 Read-only mode для таблицы

```
Line 1 # ALTER FOREIGN TABLE fdw_users  
- OPTIONS (ADD updatable 'false');
```

Postgres_fdw и DBLink

Как можно было заметить, `postgres_fdw` и `dblink` выполняют одну и ту же работу — подключение одной PostgreSQL базы к другой. Что лучше использовать в таком случае?

PostgreSQL FDW (Foreign Data Wrapper) более новый и рекомендуемый метод подключения к другим источникам данных. Хотя функциональность `dblink` похожа на FDW, последний является более SQL совместимым и может обеспечивать улучшенную производительность по сравнению с `dblink` подключениями. Кроме того, в отличие от `postgres_fdw`, `dblink` не способен сделать данные «только на чтение». Это может быть достаточно важно, если требуется обеспечить, чтобы данные в другой базе нельзя было изменять.

В `dblink` подключения работают только в течение работы сессии и их требуется пересоздавать каждый раз. `Postgres_fdw` создает постоянное подключение к другой базе данных. Это может быть как хорошо, так плохо, в зависимости от потребностей.

Из положительных сторон `dblink` можно отнести множество полезных команд, которые позволяют использовать его для программирования полезного функционала. Также `dblink` работает в версиях PostgreSQL 8.3 и выше, в то время как `postgres_fdw` работает только в PostgreSQL 9.3 и выше (такое может возникнуть, если нет возможности обновить PostgreSQL базу).

10.24 Pg_cron

`Pg_cron` — cron-подобный планировщик задач для PostgreSQL 9.5 или выше, который работает как расширение к базе. Он может выполнять несколько задач параллельно, но одновременно может работать не более одного экземпляра задания (если при запуске задачи предыдущий запуск будет еще выполняться, то запуск будет отложен до выполнения текущей задачи).

Установка и использование

После установки расширения требуется добавить его в `postgresql.conf` и перезапустить PostgreSQL:

Листинг 10.125 `pg_cron`

```
Line 1 shared_preload_libraries = 'pg_cron'
```

Далее требуется активировать расширение для `postgres` базы:

Листинг 10.126 `pg_cron`

```
Line 1 # CREATE EXTENSION pg_cron;
```

По умолчанию `pg_cron` ожидает, что все таблицы с метаданными будут находиться в `postgres` базе данных. Данное поведение можно изменить и указать через параметр `cron.database_name` в `postgresql.conf` другую базу данных, где `pg_cron` будет хранить свои данные.

Внутри `pg_cron` использует `libpq` библиотеку, поэтому потребуется разрешить подключения с `localhost` без пароля (`trust` в `pg_hba.conf`) или же создать `.pgpass` файл для настройки подключения к базе.

Для создания `cron` задач используется функция `cron.schedule`:

Листинг 10.127 Cron.schedule

```
Line 1 -- Delete old data on Saturday at 3:30am (GMT)
- SELECT cron.schedule('30 3 * * 6', $$DELETE FROM events
- WHERE event_time < now() - interval '1 week'$$);
- schedule
- -----
5         42
```

Для удаления созданных задач используется `cron.unschedule`:

Листинг 10.128 Cron.unschedule

```
Line 1 -- Vacuum every day at 10:00am (GMT)
- SELECT cron.schedule('0 10 * * *', 'VACUUM');
- schedule
- -----
5         43
-
- -- Stop scheduling a job
- SELECT cron.unschedule(43);
- unschedule
10 -----
-         t
```

В целях безопасности `cron` задачи выполняются в базе данных, в которой `cron.schedule` функция была вызвана с правами доступа текущего пользователя.

Поскольку `pg_cron` использует `libpq` библиотеку, это позволяет запускать `cron` задачи на других базах данных (даже на других серверах). С помощью суперпользователя возможно модифицировать `cron.job` таблицу и добавить в нее параметры подключения к другой базе через `nodename` и `nodeport` поля:

Листинг 10.129 Cron.job

```
Line 1 INSERT INTO cron.job (schedule , command , nodename , nodeport ,
- database , username)
- VALUES ('0 4 * * *', 'VACUUM', 'worker-node-1', 5432, '
- postgres', 'marco');
```

В таком случае нужно будет создать `.pgpass` файл для настройки подключения к базе на другом сервере.

10.25 PGStrom

PGStrom — PostgreSQL расширение, которое позволяет использовать GPU для выполнения некоторых SQL операций. В частности, за счёт привлечения GPU могут быть ускорены такие операции как сравнительный перебор элементов таблиц, агрегирование записей и слияние хэшей. Код для выполнения на стороне GPU генерируется в момент разбора SQL-запроса при помощи специального JIT-компилятора и в дальнейшем выполняется параллельно с другими связанными с текущим запросом операциями, выполняемыми на CPU. Для выполнения заданий на GPU задействован OpenCL. Увеличение производительности операций слияния таблиц (**JOIN**) при использовании GPU увеличивается в десятки раз.

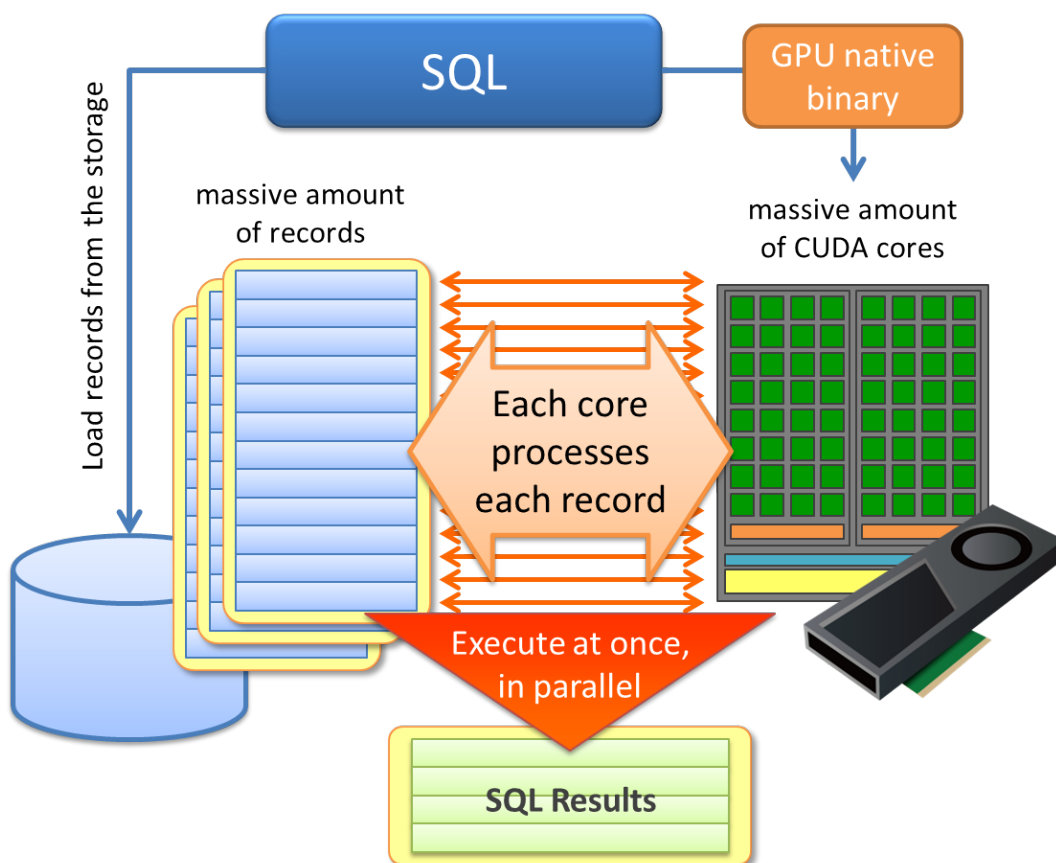


Рис. 10.3: PGStrom

Областью применения PG-Strom являются огромные отчеты с использованием агрегации и объединения таблиц. Эти рабочие нагрузки чаще используются в пакетной обработке данных для **OLAP** систем.

10.26 ZomboDB

ZomboDB — PostgreSQL расширение, которое позволяет использовать **Elasticsearch** индексы внутри базы (используется **интерфейс для методов доступа индекса**). ZomboDB индекс для PostgreSQL ничем не отличается от стандартного btree индекса. Таким образом, стандартные команды SQL полностью поддерживаются, включая **SELECT**, **BEGIN**, **COMMIT**, **ABORT**, **INSERT**, **UPDATE**, **DELETE**, **COPY** и **VACUUM** и данные индексы являются MVCC-безопасными.

На низком уровне ZomboDB индексы взаимодействуют с Elasticsearch сервером через HTTP запросы и автоматически синхронизируются в процессе изменения данных в PostgreSQL базе.

Установка и использование

ZomboDB состоит из двух частей: PostgreSQL расширения (написан на C и SQL/PLPGSQL) и Elasticsearch плагина (написан на Java).

После установки требуется добавить в `postgresql.conf` `zombodb` библиотеку:

Листинг 10.130 Zombodb

```
Line 1 local_preload_libraries = 'zombodb.so'
```

И после перезагрузки PostgreSQL активировать его для базы данных:

Листинг 10.131 Zombodb

```
Line 1 CREATE EXTENSION zombodb;
```

После этого требуется установить Elasticsearch плагин на все ноды сервера и изменить конфигурацию в `elasticsearch.yml`:

Листинг 10.132 Elasticsearch

```
Line 1 threadpool.bulk.queue_size: 1024
- threadpool.bulk.size: 12
-
- http.compression: true
5
- http.max_content_length: 1024mb
- index.query.bool.max_clause_count: 1000000
```

Для примера создадим таблицу с продуктами и заполним её данными:

Листинг 10.133 Products table

```
Line 1 # CREATE TABLE products (
- id SERIAL8 NOT NULL PRIMARY KEY,
- name text NOT NULL,
- keywords varchar(64) [],
5 short_summary phrase,
```


10.26. ZomboDB

```
-     long_description fulltext ,
-     price bigint ,
-     inventory_count integer ,
-     discontinued boolean default false ,
10    availability_date date
- );
-
- # COPY products FROM PROGRAM 'curl https://raw.
-   githubusercontent.com/zombodb/zombodb/master/TUTORIAL-
-   data.dmp';
-
- zdb(record) zombodb функция конвертирует запись в JSON формат
- (обертка поверх row_to_json(record)):
```

Листинг 10.134 Zdb

```
Line 1 # SELECT zdb(products) FROM products WHERE id = 1;
-
-
- -----
- {"id":1,"name":"Magical Widget","keywords":["magical","
-   widget","round"],"short_summary":"A widget that is quite
-   magical","long_description":"Magical Widgets come from
-   the land of Magicville and are capable of things you can'
-   t imagine","price":9900,"inventory_count":42,"
-   discontinued":false,"availability_date":"2015-08-31"}
```

`zdb(regclass, tid)` zombodb функция, которая используется для статического определения ссылок на таблицу/индекс в контексте последовательного сканирования. Благодаря этим двум функциям можно создать zombodb индекс для `products` таблицы:

Листинг 10.135 Zdb

```
Line 1 # CREATE INDEX idx_zdb_products
-       ON products
-       USING zombodb(zdb('products', products.ctid), zdb(
-   products))
-       WITH (url='http://localhost:9200/');
```

Теперь можно проверить работу индекса:

Листинг 10.136 Index usage

```
Line 1 # SELECT id, name, short_summary FROM products WHERE zdb('
-   products', products.ctid) ==> 'sports or box';
-   id | name | short_summary
-   ---+-----+-----
-   2 | Baseball | It's a baseball
5    4 | Box | Just an empty box made of wood
-   (2 rows)
- # EXPLAIN SELECT * FROM products WHERE zdb('products', ctid)
-   ==> 'sports or box';
```

10.27. Заключение

QUERY PLAN

```
10  Index Scan using idx_zdb_products on products  (cost
      =0.00..4.02  rows=2  width=153)
      Index Cond: (zdb('products'::regclass, ctid) ==> 'sports
      or box'::text)
      (2 rows)
```

ZomboDB содержит набор функций для агрегационных запросов. Например, если нужно выбрать уникальный набор ключевых слов для всех продуктов в `keywords` поле вместе с их количеством, то можно воспользоваться `zdb_tally` функцией:

Листинг 10.137 Zdb_tally

```
Line 1 # SELECT * FROM zdb_tally('products', 'keywords', '^.*', '',
      5000, 'term');
      term | count
      -----+-----
      alexander graham bell | 1
5  baseball | 1
      box | 1
      communication | 1
      magical | 1
      negative space | 1
10 primitive | 1
      round | 2
      sports | 1
      square | 1
      widget | 1
15 wooden | 1
      (12 rows)
```

Более подробно с использованием ZomboDB можно ознакомиться в [официальной документации](#).

10.27 Заключение

Расширения помогают улучшить работу PostgreSQL в решении специфических проблем. Расширяемость PostgreSQL позволяет создавать собственные расширения, или же наоборот, не нагружать СУБД лишним, не требуемым функционалом.

Бэкап и восстановление PostgreSQL

Есть два типа администраторов — те, кто не делает бэкапы, и те, кто уже делает

Народная мудрость

Если какая-нибудь неприятность может произойти, она случается

Закон Мэрфи

11.1 Введение

Любой хороший сисадмин знает — бэкапы нужны всегда. Насколько бы надежной ни казалась Ваша система, всегда может произойти случай, который был не учтен, и из-за которого могут быть потеряны данные.

Тоже самое касается и PostgreSQL баз данных. Посыпавшийся винчестер на сервере, ошибка в файловой системе, ошибка в другой программе, которая перетерла весь каталог PostgreSQL и многое другое приведет только к плачевному результату. И даже если у Вас репликация с множеством слейвов, это не означает, что система в безопасности — неверный запрос на мастер (**DELETE/DROP/TRUNCATE**), и у слейвов такая же порция данных (точнее их отсутствие).

Существуют три принципиально различных подхода к резервному копированию данных PostgreSQL:

- SQL бэкап;
- Бэкап уровня файловой системы;
- Непрерывное резервное копирование;

Каждый из этих подходов имеет свои сильные и слабые стороны.

11.2 SQL бэкап

Идея этого подхода в создании текстового файла с командами SQL. Такой файл можно передать обратно на сервер и воссоздать базу данных в том же состоянии, в котором она была во время бэкапа. У PostgreSQL для этого есть специальная утилита — `pg_dump`. Пример использования `pg_dump`:

Листинг 11.1 Создаем бэкап с помощью `pg_dump`

```
Line 1 $ pg_dump dbname > outfile
```

Для восстановления такого бэкапа достаточно выполнить:

Листинг 11.2 Восстанавливаем бэкап

```
Line 1 $ psql dbname < infile
```

При этом базу данных `dbname` потребуется создать перед восстановлением. Также потребуется создать пользователей, которые имеют доступ к данным, которые восстанавливаются (это можно и не делать, но тогда просто в выводе восстановления будут ошибки). Если нам требуется, чтобы восстановление прекратилось при возникновении ошибки, тогда потребуется восстанавливать бэкап таким способом:

Листинг 11.3 Восстанавливаем бэкап

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

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

Листинг 11.4 Бэкап в другую БД

```
Line 1 $ pg_dump -h host1 dbname | psql -h host2 dbname
```

После восстановления бэкапа желательно запустить `ANALYZE`, чтобы оптимизатор запросов обновил статистику.

А что, если нужно сделать бэкап не одной базы данных, а всех, да и еще получить в бэкапе информацию про роли и таблицы? В таком случае у PostgreSQL есть утилита `pg_dumpall`. `pg_dumpall` используется для создания бэкапа данных всего кластера PostgreSQL:

Листинг 11.5 Бэкап кластера PostgreSQL

```
Line 1 $ pg_dumpall > outfile
```

Для восстановления такого бэкапа достаточно выполнить от суперпользователя:

Листинг 11.6 Восстановления бэкапа PostgreSQL

```
Line 1 $ psql -f infile postgres
```

SQL бэкап больших баз данных

Некоторые операционные системы имеют ограничения на максимальный размер файла, что может вызывать проблемы при создании больших бэкапов через `pg_dump`. К счастью, `pg_dump` можете бэкапить в стандартный вывод. Так что можно использовать стандартные инструменты Unix, чтобы обойти эту проблему. Есть несколько возможных способов:

- Использовать сжатие для бэкапа

Можно использовать программу сжатия данных, например GZIP:

Листинг 11.7 Сжатие бэкапа PostgreSQL

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

Восстановление:

Листинг 11.8 Восстановление бэкапа PostgreSQL

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

или

Листинг 11.9 Восстановление бэкапа PostgreSQL

```
Line 1 cat filename.gz | gunzip | psql dbname
```

- Использовать команду `split`

Команда `split` позволяет разделить вывод в файлы меньшего размера, которые являются подходящими по размеру для файловой системы. Например, бэкап делится на куски по 1 мегабайту:

Листинг 11.10 Создание бэкапа PostgreSQL

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

Восстановление:

Листинг 11.11 Восстановление бэкапа PostgreSQL

```
Line 1 $ cat filename* | psql dbname
```

- Использовать пользовательский формат дампа `pg_dump`

PostgreSQL построен на системе с библиотекой сжатия Zlib, поэтому пользовательский формат бэкапа будет в сжатом виде. Это похоже на метод с использованием GZIP, но он имеет дополнительное преимущество — таблицы могут быть восстановлены выборочно. Минус такого бэкапа — восстановить возможно только в такую же версию PostgreSQL (отличаться может только патч релиз, третья цифра после точки в версии):

Листинг 11.12 Создание бэкапа PostgreSQL

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

11.3. Бэкап уровня файловой системы

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

Листинг 11.13 Восстановление бэкапа PostgreSQL

```
Line 1 $ pg_restore -d dbname filename
```

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

11.3 Бэкап уровня файловой системы

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

Листинг 11.14 Бэкап PostgreSQL файлов

```
Line 1 $ tar -cf backup.tar /usr/local/pgsql/data
```

Но есть два ограничения, которые делает этот метод нецелесообразным, или, по крайней мере, уступающим SQL бэкапу:

- PostgreSQL база данных должна быть остановлена, для того, чтобы получить актуальный бэкап (PostgreSQL держит множество объектов в памяти, буферизация файловой системы). Излишне говорить, что во время восстановления такого бэкапа потребуется также остановить PostgreSQL;
- Не получится восстановить только определенные данные с такого бэкапа;

Как альтернатива, можно делать снимки (snapshot) файлов системы (папки с файлами PostgreSQL). В таком случае останавливать PostgreSQL не требуется. Однако, резервная копия, созданная таким образом, сохраняет файлы базы данных в состоянии, как если бы сервер базы данных был неправильно остановлен. Поэтому при запуске PostgreSQL из резервной копии, он будет думать, что предыдущий экземпляр сервера вышел из строя и восстановит данные в соответствии с данными журнала WAL. Это не проблема, просто надо знать про это (и не забыть включить WAL файлы в резервную копию). Также, если файловая система PostgreSQL распределена по разным файловым системам, то такой метод бэкапа будет очень ненадежным — снимки файлов системы должны быть сделаны одновременно. Почитайте документацию файловой системы очень внимательно, прежде чем доверять снимкам файлов системы в таких ситуациях.

Также возможен вариант с использованием `rsync` утилиты. Первым запуском `rsync` мы копируем основные файлы с директории PostgreSQL

11.4. Непрерывное резервное копирование

(PostgreSQL при этом продолжает работу). После этого мы останавливаем PostgreSQL и запускаем повторно `rsync`. Второй запуск `rsync` пройдет гораздо быстрее, чем первый, потому что будет передавать относительно небольшой размер данных, и конечный результат будет соответствовать остановленной СУБД. Этот метод позволяет делать бэкап уровня файловой системы с минимальным временем простоя.

11.4 Непрерывное резервное копирование

PostgreSQL поддерживает упреждающую запись логов (Write Ahead Log, WAL) в `pg_xlog` директорию, которая находится в директории данных СУБД. В логи пишутся все изменения, сделанные с данными в СУБД. Этот журнал существует прежде всего для безопасности во время краха PostgreSQL: если происходят сбои в системе, базы данных могут быть восстановлены с помощью «перезапуска» этого журнала. Тем не менее, существование журнала делает возможным использование третьей стратегии для резервного копирования баз данных: мы можем объединить бэкап уровня файловой системы с резервной копией WAL файлов. Если требуется восстановить такой бэкап, то мы восстанавливаем файлы резервной копии файловой системы, а затем «перезапускаем» с резервной копии файлов WAL для приведения системы к актуальному состоянию. Этот подход является более сложным для администрирования, чем любой из предыдущих подходов, но он имеет некоторые преимущества:

- Не нужно согласовывать файлы резервной копии системы. Любая внутренняя противоречивость в резервной копии будет исправлена путем преобразования журнала (не отличается от того, что происходит во время восстановления после сбоя);
- Восстановление состояния сервера для определенного момента времени;
- Если мы постоянно будем «скармливать» файлы WAL на другую машину, которая была загружена с тех же файлов резервной базы, то у нас будет находящийся всегда в актуальном состоянии резервный сервер PostgreSQL (создание сервера горячего резерва);

Как и бэкап файловой системы, этот метод может поддерживать только восстановление всей базы данных кластера. Кроме того, он требует много места для хранения WAL файлов.

Настройка

Первый шаг — активировать архивирование. Эта процедура будет копировать WAL файлы в архивный каталог из стандартного каталога `pg_xlog`. Это делается в файле `postgresql.conf`:

11.5. Утилиты для непрерывного резервного копирования

Листинг 11.15 Настройка архивирования

```
Line 1 archive_mode = on # enable archiving
- archive_command = 'cp -v %p /data/pgsql/archives/%f'
- archive_timeout = 300 # timeout to close buffers
```

После этого необходимо перенести файлы (в порядке их появления) в архивный каталог. Для этого можно использовать функцию `rsync`. Можно поставить функцию в `cron` и, таким образом, файлы могут автоматически перемещаться между хостами каждые несколько минут:

Листинг 11.16 Копирование WAL файлов на другой хост

```
Line 1 $ rsync -avz --delete prod1:/data/pgsql/archives/ \
- /data/pgsql/archives/ > /dev/null
```

В конце необходимо скопировать файлы в каталог `pg_xlog` на сервере PostgreSQL (он должен быть в режиме восстановления). Для этого необходимо в каталоге данных PostgreSQL создать файл `recovery.conf` с заданной командой копирования файлов из архива в нужную директорию:

Листинг 11.17 recovery.conf

```
Line 1 restore_command = 'cp /data/pgsql/archives/%f "%p"'
```

Документация PostgreSQL предлагает хорошее описание настройки непрерывного копирования, поэтому данная глава не будет углубляться в детали (например, как перенести директорию СУБД с одного сервера на другой, какие могут быть проблемы). Более подробно вы можете прочитать по [этой ссылке](#).

11.5 Утилиты для непрерывного резервного копирования

Непрерывное резервное копирования - один из лучших способов для создания бэкапов и их восстановления. Нередко бэкапы сохраняются на той же файловой системе, на которой расположена база данных. Это не очень безопасно, т.к. при выходе дисковой системы сервера из строя вы можете потерять все данные (и базу, и бэкапы), или попросту столкнуться с тем, что на жестком диске закончится свободное место. Поэтому лучше, когда бэкапы складываются на отдельный сервер или в «облачное хранилище» (например [AWS S3](#)). Чтобы не писать свой «велосипед» для автоматизации этого процесса на сегодняшний день существует набор программ, которые облегчают процесс настройки и поддержки процесса создания бэкапов на основе непрерывного резервного копирования.

WAL-E

WAL-E предназначена для непрерывной архивации PostgreSQL WAL-logs в Amazon S3 или Windows Azure (начиная с версии 0.7) и управления использованием `pg_start_backup` и `pg_stop_backup`. Утилита написана на Python и разработана в компании **Heroku**, где её активно используют.

Установка

У WAL-E есть пара зависимостей: `lzop`, `psql`, `pv` (в старых версиях используется `mbuffer`), python 3.4+ и несколько python библиотек (`gevent`, `boto`, `azure`). Также для удобства настроек переменных среды устанавливается `daemontools`. На Ubuntu это можно все поставить одной командой:

Листинг 11.18 Установка зависимостей для WAL-E

```
Line 1 $ aptitude install git-core python-dev python-setuptools
      python-pip build-essential libevent-dev lzop pv
      daemontools daemontools-run
```

Теперь установим WAL-E:

Листинг 11.19 Установка WAL-E

```
Line 1 $ pip install https://github.com/wal-e/wal-e/archive/v1.0.3.
      tar.gz
```

После успешной установки можно начать работать с WAL-E.

Настройка и работа

Как уже писалось, WAL-E сливает все данные в AWS S3, поэтому нам потребуются «Access Key ID», «Secret Access Key» и «AWS Region» (эти данные можно найти в аккаунте Amazon AWS). Команда для загрузки бэкапа всей базы данных в S3:

Листинг 11.20 Загрузка бэкапа всей базы данных в S3

```
Line 1 AWS_REGION=... AWS_SECRET_ACCESS_KEY=... wal-e
      -k AWS_ACCESS_KEY_ID \
      --s3-prefix=s3://some-bucket/directory/or/whatever \
      backup-push /var/lib/postgresql/9.2/main
```

Где `s3-prefix` — URL, который содержит имя S3 бакета (bucket) и путь к папке, куда следует складывать резервные копии. Команда для загрузки WAL-логов на S3:

Листинг 11.21 Загрузка WAL-логов на S3

```
Line 1 AWS_REGION=... AWS_SECRET_ACCESS_KEY=... wal-e
      -k AWS_ACCESS_KEY_ID \
```

11.5. Утилиты для непрерывного резервного копирования

```
- --s3-prefix=s3://some-bucket/directory/or/whatever \  
- wal-push /var/lib/postgresql/9.2/main/pg_xlog/  
  WAL_SEGMENT_LONG_HEX
```

Для управления этими переменными окружения можно использовать команду `envdir` (идет в поставке с `daemontools`). Для этого создадим `envdir` каталог:

Листинг 11.22 WAL-E с `envdir`

```
Line 1 $ mkdir -p /etc/wal-e.d/env  
- $ echo "aws_region" > /etc/wal-e.d/env/AWS_REGION  
- $ echo "secret-key" > /etc/wal-e.d/env/AWS_SECRET_ACCESS_KEY  
- $ echo "access-key" > /etc/wal-e.d/env/AWS_ACCESS_KEY_ID  
5 $ echo 's3://some-bucket/directory/or/whatever' > /etc/wal-e  
  .d/env/WALE_S3_PREFIX  
- $ chown -R root:postgres /etc/wal-e.d
```

После создания данного каталога появляется возможность запускать WAL-E команды гораздо проще и с меньшим риском случайного использования некорректных значений:

Листинг 11.23 WAL-E с `envdir`

```
Line 1 $ envdir /etc/wal-e.d/env wal-e backup-push ...  
- $ envdir /etc/wal-e.d/env wal-e wal-push ...
```

Теперь настроим PostgreSQL для сбрасывания WAL-логов в S3 с помощью WAL-E. Отредактируем `postgresql.conf`:

Листинг 11.24 Настройка PostgreSQL

```
Line 1 wal_level = hot_standby # или archive, если PostgreSQL < 9.0  
- archive_mode = on  
- archive_command = 'envdir /etc/wal-e.d/env /usr/local/bin/  
  wal-e wal-push %p'  
- archive_timeout = 60
```

Лучше указать полный путь к WAL-E (можно узнать командой `which wal-e`), поскольку PostgreSQL может его не найти. После этого нужно перезагрузить PostgreSQL. В логах базы вы должны увидеть что-то подобное:

Листинг 11.25 Логи PostgreSQL

```
Line 1 2016-11-07 14:52:19 UTC LOG:  database system was shut down  
  at 2016-11-07 14:51:40 UTC  
- 2016-11-07 14:52:19 UTC LOG:  database system is ready to  
  accept connections  
- 2016-11-07 14:52:19 UTC LOG:  autovacuum launcher started  
- 2016-11-07T14:52:19.784+00 pid=7653 wal_e.worker.s3_worker  
  INFO      MSG: begin archiving a file  
5          DETAIL: Uploading "pg_xlog/000000010000000000000001"  
  to "s3://cleverdb-pg-backups/pg/wal_005  
  /000000010000000000000001.lzo".
```

11.5. Утилиты для непрерывного резервного копирования

```
- 2016-11-07 14:52:19 UTC LOG: incomplete startup packet
- 2016-11-07T14:52:28.234+00 pid=7653 wal_e.worker.s3_worker
  INFO      MSG: completed archiving to a file
-         DETAIL: Archiving to "s3://cleverdb-pg-backups/pg/
  wal_005/000000010000000000000001.lzo" complete at 21583.3
  KiB/s.
- 2016-11-07T14:52:28.341+00 pid=7697 wal_e.worker.s3_worker
  INFO      MSG: begin archiving a file
10         DETAIL: Uploading "pg_xlog
  /000000010000000000000002.00000020.backup" to "s3://
  cleverdb-pg-backups/pg/wal_005
  /000000010000000000000002.00000020.backup.lzo".
- 2016-11-07T14:52:34.027+00 pid=7697 wal_e.worker.s3_worker
  INFO      MSG: completed archiving to a file
-         DETAIL: Archiving to "s3://cleverdb-pg-backups/pg/
  wal_005/000000010000000000000002.00000020.backup.lzo"
  complete at 00KiB/s.
- 2016-11-07T14:52:34.187+00 pid=7711 wal_e.worker.s3_worker
  INFO      MSG: begin archiving a file
-         DETAIL: Uploading "pg_xlog/000000010000000000000002"
  to "s3://cleverdb-pg-backups/pg/wal_005
  /000000010000000000000002.lzo".
15 2016-11-07T14:52:40.232+00 pid=7711 wal_e.worker.s3_worker
  INFO      MSG: completed archiving to a file
-         DETAIL: Archiving to "s3://cleverdb-pg-backups/pg/
  wal_005/000000010000000000000002.lzo" complete at 2466.67
  KiB/s.
```

Если ничего похожего в логах не видно, тогда нужно смотреть что за ошибка появляется и исправлять её. Для того, чтобы бэкапить всю базу, достаточно выполнить данную команду:

Листинг 11.26 Загрузка бэкапа всей базы данных в S3

```
Line 1 $ envdir /etc/wal-e.d/env wal-e backup-push /var/lib/
  postgresql/9.2/main
- 2016-11-07T14:49:26.174+00 pid=7493 wal_e.operator.
  s3_operator INFO      MSG: start upload postgres version
  metadata
-         DETAIL: Uploading to s3://cleverdb-pg-backups/pg/
  basebackups_005/base_000000010000000000000006_00000032/
  extended_version.txt.
- 2016-11-07T14:49:32.783+00 pid=7493 wal_e.operator.
  s3_operator INFO      MSG: postgres version metadata
  upload complete
5 2016-11-07T14:49:32.859+00 pid=7493 wal_e.worker.s3_worker
  INFO      MSG: beginning volume compression
-         DETAIL: Building volume 0.
- ...
```

11.5. Утилиты для непрерывного резервного копирования

- HINT: Check that your `archive_command` is executing properly . `pg_stop_backup` can be canceled safely , but the database backup will not be usable without all the WAL segments .
- NOTICE: `pg_stop_backup complete` , all required WAL segments have been archived

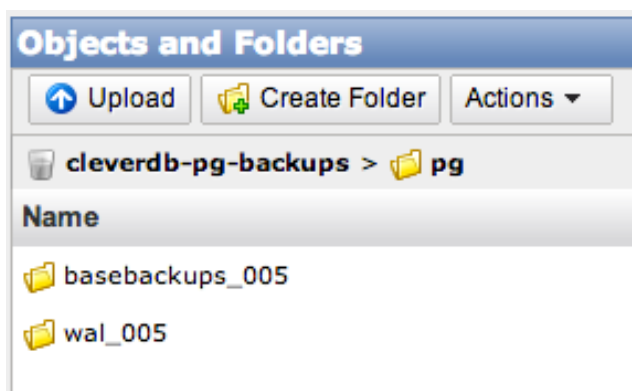


Рис. 11.1: Папка бэкапов на S3

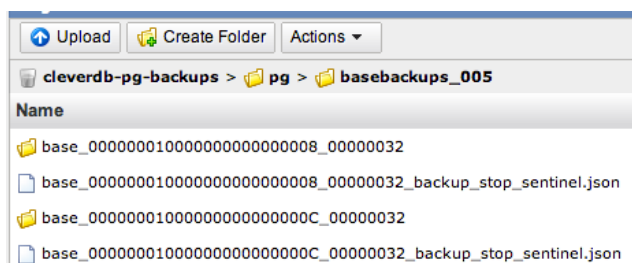


Рис. 11.2: Папка бэкапов базы на S3

Данный бэкап лучше делать раз в сутки (например, добавить в `crontab`). На рис 11.1-11.3 видно как хранятся бэкапы на S3. Все бэкапы сжаты через `lzop`. Данный алгоритм сжимает хуже чем `gzip`, но скорость сжатия намного быстрее (приблизительно 25 Мб/сек используя 5% ЦПУ). Чтобы уменьшить нагрузку на чтение с жесткого диска бэкапы отправляются через `pv` утилиту (опцией `cluster-read-rate-limit` можно ограничить скорость чтения, если это требуется).

Теперь перейдем к восстановлению данных. Для восстановления базы из резервной копии используется `backup-fetch` команда:

Листинг 11.27 Восстановление бэкапа базы из S3

```
Line 1 $ sudo -u postgres bash -c "envdir /etc/wal-e.d/env wal-e  
--s3-prefix=s3://some-bucket/directory/or/whatever backup  
-fetch /var/lib/postgresql/9.2/main LATEST"
```

11.5. Утилиты для непрерывного резервного копирования

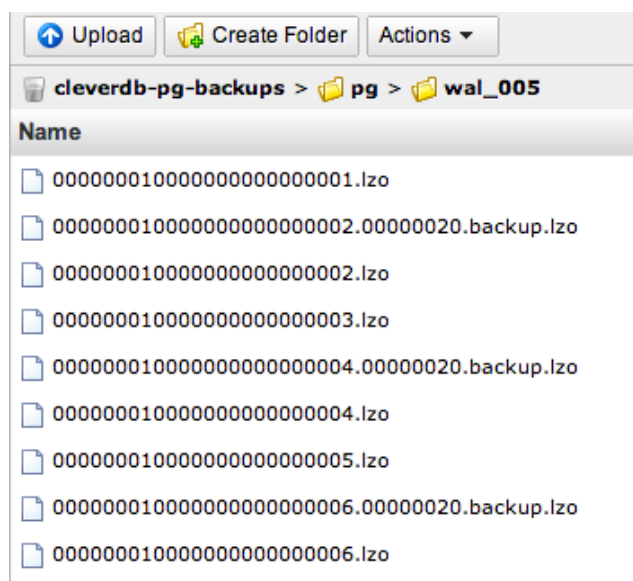


Рис. 11.3: Папка WAL-логов на S3

Где **LATEST** означает, что база восстановится из последнего актуального бэкапа (PostgreSQL в это время должен быть остановлен). Для восстановления из более поздней резервной копии:

Листинг 11.28 Восстановление из поздней резервной копии

```
Line 1 $ sudo -u postgres bash -c "envdir /etc/wal-e.d/env wal-e
--s3-prefix=s3://some-bucket/directory/or/whatever backup
-fetch /var/lib/postgresql/9.2/main
base_LONGWALNUMBER_POSITION_NUMBER"
```

Для получения списка доступных резервных копий есть команда `backup -list`:

Листинг 11.29 Список резервных копий

```
Line 1 $ envdir /etc/wal-e.d/env wal-e backup-list
- name      last_modified    expanded_size_bytes
  wal_segment_backup_start
  wal_segment_offset_backup_start wal_segment_backup_stop
  wal_segment_offset_backup_stop
- base_000000010000000000000008_00000032 2016-11-07T14
  :00:07.000Z                               000000010000000000000008
  00000032
- base_00000001000000000000000C_00000032 2016-11-08T15
  :00:08.000Z                               00000001000000000000000C
  00000032
```

После завершения работы с основной резервной копией для полного восстановления нужно считать WAL-логи (чтобы данные обновились до последнего состояния). Для этого используется `recovery.conf`:

11.5. Утилиты для непрерывного резервного копирования

Листинг 11.30 recovery.conf

```
Line 1 restore_command = 'envdir /etc/wal-e.d/env /usr/local/bin/  
wal-e wal-fetch "%f" "%p"'
```

После создания этого файла нужно запустить PostgreSQL. Через небольшой интервал времени база станет полностью восстановленной.

Для удаления старых резервных копий (или вообще всех) используется команда `delete`:

Листинг 11.31 Удаление резервных копий

```
Line 1 # удаление старых бэкапов старше  
base_00000004000002DF000000A6_03626144  
- $ envdir /etc/wal-e.d/env wal-e delete --confirm before  
base_00000004000002DF000000A6_03626144  
- # удаление всех бэкапов  
- $ envdir /etc/wal-e.d/env wal-e delete --confirm everything  
5 # удалить все старше последних 20 бэкапов  
- $ envdir /etc/wal-e.d/env wal-e delete --confirm retain 20
```

Без опции `--confirm` команды будут запускаться и показывать, что будет удаляться, но фактического удаления не будет производиться (dry run).

Заключение

WAL-E помогает автоматизировать сбор резервных копий с PostgreSQL и хранить их в достаточно дешевом и надежном хранилище — Amazon S3 или Windows Azure.

Barman

Barman, как и WAL-E, позволяет создать систему для бэкапа и восстановления PostgreSQL на основе непрерывного резервного копирования. Barman использует для хранения бэкапов отдельный сервер, который может собирать бэкапы как с одного, так и с нескольких PostgreSQL баз данных.

Установка и настройка

Рассмотрим простой случай с одним экземпляром PostgreSQL (один сервер) и пусть его хост будет `pghost`. Наша задача — автоматизировать сбор и хранение бэкапов этой базы на другом сервере (его хост будет `brhost`). Для взаимодействия эти два сервера должны быть полностью открыты по SSH (доступ без пароля, по ключам). Для этого можно использовать `authorized_keys` файл.

Листинг 11.32 Проверка подключения по SSH

```
Line 1 # Проверка подключения с сервера PostgreSQL (pghost)
```

11.5. Утилиты для непрерывного резервного копирования

- \$ ssh barman@brhost
- # Проверка подключения с сервера бэкапов (brhost)
- \$ ssh postgres@pghost

Далее нужно установить на сервере для бэкапов barman. Сам barman написан на python и имеет пару зависимостей: python 2.6+, `rsync` и python библиотеки (`argh`, `psycopg2`, `python-dateutil`, `distribute`). На Ubuntu все зависимости можно поставить одной командой:

Листинг 11.33 Установка зависимостей barman

```
Line 1 $ aptitude install python-dev python-argh python-psycopg2
      python-dateutil rsync python-setuptools
```

Далее нужно установить barman:

Листинг 11.34 Установка barman

```
Line 1 $ tar -xzf barman-2.1.tar.gz
- $ cd barman-2.1/
- $ ./setup.py build
- $ sudo ./setup.py install
```

Или используем [PostgreSQL Community APT репозиторий](#):

Листинг 11.35 Установка barman

```
Line 1 $ apt-get install barman
```

Теперь перейдем к серверу с PostgreSQL. Для того, чтобы barman мог подключаться к базе данных без проблем, нам нужно выставить настройки доступа в конфигах PostgreSQL:

Листинг 11.36 Отредактировать в postgresql.conf

```
Line 1 listen_address = '*'
```

Листинг 11.37 Добавить в pg_hba.conf

```
Line 1 host all all brhost/32 trust
```

После этих изменений нужно перезагрузить PostgreSQL. Теперь можем проверить с сервера бэкапов подключение к PostgreSQL:

Листинг 11.38 Проверка подключения к базе

```
Line 1 $ psql -c 'SELECT version()' -U postgres -h pghost
-
-
-
-
- PostgreSQL 9.3.1 on x86_64-unknown-linux-gnu, compiled by
  gcc (Ubuntu/Linaro 4.7.2-2ubuntu1) 4.7.2, 64-bit
5 (1 row)
```

Далее создадим папку на сервере с бэкапами для хранения этих самых бэкапов:

11.5. Утилиты для непрерывного резервного копирования

Листинг 11.39 Папка для хранения бэкапов

```
Line 1 $ sudo mkdir -p /srv/barman
- $ sudo chown barman:barman /srv/barman
    Для настройки barman создадим /etc/barman.conf:
```

Листинг 11.40 barman.conf

```
Line 1 [barman]
- ; Main directory
- barman_home = /srv/barman
-
5 ; Log location
- log_file = /var/log/barman/barman.log
-
- ; Default compression level: possible values are None (
-     default), bzip2, gzip or custom
- compression = gzip
10
- ; 'main' PostgreSQL Server configuration
- [main]
- ; Human readable description
- description = "Main PostgreSQL Database"
15
- ; SSH options
- ssh_command = ssh postgres@pghost
-
- ; PostgreSQL connection string
20 conninfo = host=pghost user=postgres
```

Секция «main» (так мы назвали для barman наш PostgreSQL сервер) содержит настройки для подключения к PostgreSQL серверу и базе. Проверим настройки:

Листинг 11.41 Проверка barman настроек

```
Line 1 $ barman show-server main
- Server main:
-     active: true
-     description: Main PostgreSQL Database
5     ssh_command: ssh postgres@pghost
-     conninfo: host=pghost user=postgres
-     backup_directory: /srv/barman/main
-     basebackups_directory: /srv/barman/main/base
-     wals_directory: /srv/barman/main/wals
10     incoming_wals_directory: /srv/barman/main/incoming
-     lock_file: /srv/barman/main/main.lock
-     compression: gzip
-     custom_compression_filter: None
-     custom_decompression_filter: None
```


11.5. Утилиты для непрерывного резервного копирования

```
15     retention_policy: None
-     wal_retention_policy: None
-     pre_backup_script: None
-     post_backup_script: None
-     current_xlog: None
20     last_shipped_wal: None
-     archive_command: None
-     server_txt_version: 9.3.1
-     data_directory: /var/lib/postgresql/9.3/main
-     archive_mode: off
25     config_file: /etc/postgresql/9.3/main/postgresql.
conf
-     hba_file: /etc/postgresql/9.3/main/pg_hba.conf
-     ident_file: /etc/postgresql/9.3/main/pg_ident.conf
-
- #barman check main
30 Server main:
-     ssh: OK
-     PostgreSQL: OK
-     archive_mode: FAILED (please set it to 'on')
-     archive_command: FAILED (please set it accordingly
to documentation)
35     directories: OK
-     compression settings: OK
```

Все хорошо, вот только PostgreSQL не настроен. Для этого на сервере с PostgreSQL отредактируем конфиг базы:

Листинг 11.42 Настройка PostgreSQL

```
Line 1 wal_level = hot_standby # archive для PostgreSQL < 9.0
- archive_mode = on
- archive_command = 'rsync -a %p barman@brhost:
  INCOMING_WALS_DIRECTORY/%f'
```

где `INCOMING_WALS_DIRECTORY` — директория для складывания WAL-логов. Её можно узнать из вывода команды `barman show-server main` (листинг 11.41, указано `/srv/barman/main/incoming`). После изменения настроек нужно перезагрузить PostgreSQL. Теперь проверим статус на сервере бэкапов:

Листинг 11.43 Проверка

```
Line 1 $ barman check main
- Server main:
-     ssh: OK
-     PostgreSQL: OK
5     archive_mode: OK
-     archive_command: OK
-     directories: OK
```

11.5. Утилиты для непрерывного резервного копирования

- `compression settings: OK`

Все готово. Для добавления нового сервера процедуру потребуется повторить, а в `barman.conf` добавить новый сервер.

Создание бэкапов

Получение списка серверов:

Листинг 11.44 Список серверов

```
Line 1 $ barman list -server
- main - Main PostgreSQL Database
```

Запуск создания резервной копии PostgreSQL (сервер указывается последним параметром):

Листинг 11.45 Создание бэкапа

```
Line 1 $ barman backup main
- Starting backup for server main in /srv/barman/main/base
  /20121109T090806
- Backup start at xlog location: 0/3000020
  (00000001000000000000000003, 00000020)
- Copying files.
5 Copy done.
- Asking PostgreSQL server to finalize the backup.
- Backup end at xlog location: 0/30000D8
  (00000001000000000000000003, 000000D8)
- Backup completed
```

Такую задачу лучше выполнять раз в сутки (добавить в cron). Посмотреть список бэкапов для указанной базы:

Листинг 11.46 Список бэкапов

```
Line 1 $ barman list -backup main
- main 20121110T091608 - Fri Nov 10 09:20:58 2012 - Size: 1.0
  GiB - WAL Size: 446.0 KiB
- main 20121109T090806 - Fri Nov 9 09:08:10 2012 - Size: 23.0
  MiB - WAL Size: 477.0 MiB
```

Более подробная информация о выбранной резервной копии:

Листинг 11.47 Информация о выбранной резервной копии

```
Line 1 $ barman show-backup main 20121110T091608
- Backup 20121109T091608:
- Server Name      : main
- Status          : DONE
5 PostgreSQL Version: 90201
- PGDATA directory : /var/lib/postgresql/9.3/main
-
```

11.5. Утилиты для непрерывного резервного копирования

```
- Base backup information :
-   Disk usage           : 1.0 GiB
10  Timeline             : 1
-   Begin WAL            : 000000010000000000000008C
-   End WAL              : 0000000100000000000000092
-   WAL number          : 7
-   Begin time           : 2012-11-10 09:16:08.856884
15  End time             : 2012-11-10 09:20:58.478531
-   Begin Offset        : 32
-   End Offset          : 3576096
-   Begin XLOG           : 0/8C000020
-   End XLOG            : 0/92369120
20
- WAL information :
-   No of files         : 1
-   Disk usage          : 446.0 KiB
-   Last available     : 0000000100000000000000093
25
- Catalog information :
-   Previous Backup    : 20121109T090806
-   Next Backup       : - (this is the latest base backup)
```

Также можно сжимать WAL-логи, которые накапливаются в каталогах командой «cron»:

Листинг 11.48 Архивирование WAL-логов

```
Line 1 $ barman cron
- Processing xlog segments for main
-   00000001000000000000000001
-   00000001000000000000000002
5   00000001000000000000000003
-   00000001000000000000000003.00000020.backup
-   00000001000000000000000004
-   00000001000000000000000005
-   00000001000000000000000006
```

Эту команду требуется добавлять в `crontab`. Частота выполнения данной команды зависит от того, как много WAL-логов накапливается (чем больше файлов - тем дольше она выполняется). Barman может сжимать WAL-логи через `gzip`, `bzip2` или другой компрессор данных (команды для сжатия и распаковки задаются через `custom_compression_filter` и `custom_decompression_filter` соответственно). Также можно активировать компрессию данных при передаче по сети через опцию `network_compression` (по умолчанию отключена). Через опции `bandwidth_limit` (по умолчанию 0, ограничений нет) и `tablespace_bandwidth_limit` возможно ограничить использования сетевого канала.

Для восстановления базы из бэкапа используется команда `recover`:

11.5. Утилиты для непрерывного резервного копирования

Листинг 11.49 Восстановление базы

```
Line 1 $ barman recover --remote-ssh-command "ssh postgres@pghost"
      main 20121109T090806 /var/lib/postgresql/9.3/main
- Starting remote restore for server main using backup
  20121109T090806
- Destination directory: /var/lib/postgresql/9.3/main
- Copying the base backup.
5 Copying required wal segments.
- The archive_command was set to 'false' to prevent data
  losses.
-
- Your PostgreSQL server has been successfully prepared for
  recovery!
-
10 Please review network and archive related settings in the
  PostgreSQL
- configuration file before starting the just recovered
  instance.
-
- WARNING: Before starting up the recovered PostgreSQL server ,
- please review also the settings of the following
  configuration
15 options as they might interfere with your current recovery
  attempt:
-
-   data_directory = '/var/lib/postgresql/9.3/main'
-     # use data in another directory
-   external_pid_file = '/var/run/postgresql/9.3-main.pid'
-     # write an extra PID file
-   hba_file = '/etc/postgresql/9.3/main/pg_hba.conf' #
  host-based authentication file
20   ident_file = '/etc/postgresql/9.3/main/pg_ident.conf'
-     # ident configuration file
```

Barman может восстановить базу из резервной копии на удаленном сервере через SSH (для этого есть опция `remote-ssh-command`). Также barman может восстановить базу, используя **PITR**: для этого используются опции `target-time` (указывается время) или `target-xid` (id транзакции).

Заключение

Barman помогает автоматизировать сбор и хранение резервных копий PostgreSQL данных на отдельном сервере. Утилита проста, позволяет хранить и удобно управлять бэкапами нескольких PostgreSQL серверов.

Pg_arman

Pg_arman — менеджер резервного копирования и восстановления для PostgreSQL 9.5 или выше. Это ответвление проекта `pg_arman`, изначально разрабатываемого в NTT. Теперь его разрабатывает и поддерживает Мишель Пакье. Утилита предоставляет следующие возможности:

- Резервное копирование во время работы базы данных, включая табличные пространства, с помощью всего одной команды;
- Восстановление из резервной копии всего одной командой, с нестандартными вариантами, включая использование **PITR**;
- Поддержка полного и дифференциального копирования;
- Управление резервными копиями со встроенными каталогами;

Использование

Сначала требуется создать «каталог резервного копирования», в котором будут храниться файлы копий и их метаданные. До инициализации этого каталога рекомендуется настроить параметры `archive_mode` и `archive_command` в `postgresql.conf`. Если переменные инициализированы, `pg_arman` может скорректировать файл конфигурации. В этом случае потребуется задать путь к кластеру баз данных: переменной окружения `PGDATA` или через параметр `-D/--pgdata`.

Листинг 11.50 init

```
Line 1 $ pg_arman init -B /path/to/backup/
```

После этого возможен один из следующих вариантов резервного копирования:

- Полное резервное копирование (копируется весь кластер баз данных);

Листинг 11.51 backup

```
Line 1 $ pg_arman backup --backup-mode=full  
- $ pg_arman validate  
-
```

- Дифференциальное резервное копирование: копируются только файлы или страницы, изменённые после последней проверенной копии. Для этого выполняется сканирование записей WAL от позиции последнего копирования до LSN выполнения `pg_start_backup` и все изменённые блоки записываются и отслеживаются как часть резервной копии. Так как просканированные сегменты WAL должны находиться в архиве WAL, последний сегмент, задействованный после запуска `pg_start_backup`, должен быть переключен принудительно;

11.6. Заключение

Листинг 11.52 backup

```
Line 1  $ pg_arman backup --backup-mode=page
-      $ pg_arman validate
-
```

После резервного копирования рекомендуется проверять файлы копий как только это будет возможно. Непроверенные копии нельзя использовать в операциях восстановления и резервного копирования.

До начала восстановления через `pg_arman` PostgreSQL кластер должен быть остановлен. Если кластер баз данных всё ещё существует, команда восстановления сохранит незаархивированный журнал транзакций и удалит все файлы баз данных. После восстановления файлов `pg_arman` создаёт `recovery.conf` в `$PGDATA` каталоге. Этот конфигурационный файл содержит параметры для восстановления. После успешного восстановления рекомендуется при первой же возможности сделать полную резервную копию. Если ключ `--recovery-target-timeline` не задан, целевой точкой восстановления будет `TimeLineID` последней контрольной точки в файле (`$PGDATA/global/pg_control`). Если файл `pg_control` отсутствует, целевой точкой будет `TimeLineID` в полной резервной копии, используемой при восстановлении.

Листинг 11.53 restore

```
Line 1  $ pg_ctl stop -m immediate
-      $ pg_arman restore
-      $ pg_ctl start
```

`Pg_arman` имеет ряд ограничений:

- Требуются права чтения каталога баз данных и записи в каталог резервного копирования. Обычно для этого на сервере БД требуется смонтировать диск, где размещён каталог резервных копий, используя NFS или другую технологию;
- Основные версии `pg_arman` и сервера должны совпадать;
- Размеры блоков `pg_arman` и сервера должны совпадать;
- Если в каталоге с журналами сервера или каталоге с архивом WAL оказываются нечитаемые файлы/каталоги, резервное копирование или восстановление завершится сбоем вне зависимости от выбранного режима копирования;

11.6 Заключение

В любом случае, усилия и время, затраченные на создание оптимальной системы создания бэкапов, будут оправданы. Невозможно предугадать когда произойдут проблемы с базой данных, поэтому бэкапы должны быть настроены для PostgreSQL (особенно, если это продакшн система).

Стратегии масштабирования для PostgreSQL

В конце концов, все решают
люди, не стратегии

Ларри Боссида

12.1 Введение

Многие разработчики крупных проектов сталкиваются с проблемой, когда один-единственный сервер базы данных никак не может справиться с нагрузками. Очень часто такие проблемы происходят из-за неверного проектирования приложения (плохая структура БД для приложения, отсутствие кеширования). Но в данном случае пусть у нас есть «идеальное» приложение, для которого оптимизированы все SQL запросы, используется кеширование, PostgreSQL настроен, но все равно не справляется с нагрузкой. Такая проблема может возникнуть как на этапе проектирования, так и на этапе роста приложения. И тут возникает вопрос: какую стратегию выбрать при возникновении подобной ситуации?

Если Ваш заказчик готов купить супер сервер за несколько тысяч долларов (а по мере роста — десятков тысяч и т. д.), чтобы сэкономить время разработчиков, но сделать все быстро, можете дальше эту главу не читать. Но такой заказчик — мифическое существо и, в основном, такая проблема ложится на плечи разработчиков.

Суть проблемы

Для того, чтобы сделать какой-то выбор, необходимо знать суть проблемы. Существуют два предела, в которые могут уткнуться сервера баз данных:

12.2. Проблема чтения данных

- Ограничение пропускной способности чтения данных;
- Ограничение пропускной способности записи данных;

Практически никогда не возникает одновременно две проблемы, по крайней мере, это маловероятно (если вы, конечно, не Twitter или Facebook пишете). Если вдруг такое происходит — возможно, система неверно спроектирована, и её реализацию следует пересмотреть.

12.2 Проблема чтения данных

Проблема с чтением данных обычно начинается, когда СУБД не в состоянии обеспечить то количество выборок, которое требуется. В основном такое происходит в блогах, новостных лентах и т. д. Хочу сразу отметить, что подобную проблему лучше решать внедрением кеширования, а потом уже думать как масштабировать СУБД.

Методы решения

- PgPool-II v.3 + PostgreSQL v.9 с Streaming Replication — отличное решение для масштабирования на чтение, более подробно можно ознакомиться по [ссылке](#). Основные преимущества:
 - Низкая задержка репликации между мастером и слейвом;
 - Производительность записи падает незначительно;
 - Отказоустойчивость (failover);
 - Пулы соединений;
 - Интеллектуальная балансировка нагрузки — проверка задержки репликации между мастером и слейвом (сам проверяет `pg_current_xlog_location` и `pg_last_xlog_receive_location`);
 - Добавление слейвов СУБД без остановки pgpool-II;
 - Простота в настройке и обслуживании;
- PgPool-II v.3 + PostgreSQL с Slony/Londiste/Bucardo — аналогично предыдущему решению, но с использованием Slony/Londiste/Bucardo. Основные преимущества:
 - Отказоустойчивость (failover);
 - Пулы соединений;
 - Интеллектуальная балансировка нагрузки — проверка задержки репликации между мастером и слейвом;
 - Добавление слейв СУБД без остановки pgpool-II;
 - Можно использовать Postgresql ниже 9 версии;
- Citus — подробнее можно прочитать в «6.5 Citus» главе;
- Postgres-X2 — подробнее можно прочитать в «6.3 Postgres-X2» главе;
- Postgres-XL — подробнее можно прочитать в «6.4 Postgres-XL» главе;

12.3 Проблема записи данных

Обычно такая проблема возникает в системах, которые производят анализ больших объемов данных (например аналог Google Analytics). Данные активно пишутся и мало читаются (или читается только суммарный вариант собранных данных).

Методы решения

Один из самых популярных методов решения проблемы — размазать нагрузку по времени с помощью систем очередей.

- PgQ — это система очередей, разработанная на базе PostgreSQL. Разработчики — компания Skype. Используется в Londiste (подробнее «5.6 Londiste»). Особенности:
 - Высокая производительность благодаря особенностям PostgreSQL;
 - Общая очередь, с поддержкой нескольких обработчиков и нескольких генераторов событий;
 - PgQ гарантирует, что каждый обработчик увидит каждое событие как минимум один раз;
 - События достаются из очереди «пачками» (batches);
 - Чистое API на SQL функциях;
 - Удобный мониторинг;
- Citus — подробнее можно прочитать в «6.5 Citus» главе;
- Postgres-X2 — подробнее можно прочитать в «6.3 Postgres-X2» главе;
- Postgres-XL — подробнее можно прочитать в «6.4 Postgres-XL» главе;

12.4 Заключение

В данной главе показаны только несколько возможных вариантов решения задач масштабирования PostgreSQL. Таких стратегий существует огромное количество и каждая из них имеет как сильные, так и слабые стороны. Самое важное то, что выбор оптимальной стратегии масштабирования для решения поставленных задач остается на плечах разработчиков и/или администраторов СУБД.

Утилиты для PostgreSQL

Ум всегда занят
исследованием чего-либо

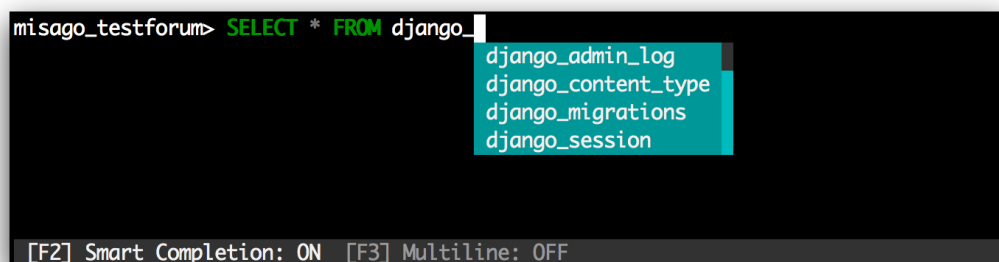
Цицерон

13.1 Введение

В данной главе собраны полезные утилиты для PostgreSQL, которые не упоминались в других разделах книги.

13.2 Pgcli

Pgcli — интерфейс командной строки для PostgreSQL с автозаполнением и подсветкой синтаксиса. Написан на Python.



```
misago_testforum> SELECT * FROM django_
django_admin_log
django_content_type
django_migrations
django_session
[F2] Smart Completion: ON [F3] Multiline: OFF
```

Рис. 13.1: Pgcli автозаполнение

13.3 Pgloader

Pgloader — консольная утилита для переноса данных с CSV файлов, HTTP ресурсов, SQLite, dBase или MySQL баз данных в PostgreSQL. Для быстрой загрузки данных используется COPY протокол. Pgloader содержит модули для преобразование данных, которые позволяют преобразовывать данные во время переноса (например, преобразование набора цифр в IP адрес или разбить строку на два текстовых поля).

13.4 Postgres.app

Postgres.app — полнофункциональный PostgreSQL, который упакован в качестве стандартного Mac приложения, поэтому работает только на Mac OS системах. Приложение имеет красивый пользовательский интерфейс и работает в системной строке меню.

13.5 pgAdmin

pgAdmin — инструмент с графическим интерфейсом для управления PostgreSQL и производных от него баз данных. Он может быть запущен в качестве десктоп или веб-приложения. Написан на Python (с использованием Flask фреймворка) и JavaScript (с использованием jQuery и Bootstrap).

Существуют альтернативные программные продукты для PostgreSQL (как платные, так и бесплатные). Вот примеры бесплатных альтернатив:

- **DBeaver**;
- **DBGlass**;
- **Metabase**;
- **pgModeler**;
- **Pgweb**;
- **Postbird**;
- **SQL Tabs**;

13.6 PostgREST

PostgREST — инструмент создания HTTP REST API для PostgreSQL базы. Написан на Haskell.

13.7 Ngx_postgres

Ngx_postgres — модуль для **Nginx**, который позволяет напрямую работать с PostgreSQL базой. Ответы генерируется в формате RDS (Resty DBD

Stream), поэтому он совместим с `ngx_rds_json`, `ngx_rds_csv` и `ngx_drizzle` модулями.

13.8 Заключение

В данной главе рассмотрено лишь несколько полезных утилит для PostgreSQL. Каждый день для этой базы данных появляется все больше интересных инструментов, которые улучшают, упрощают или автоматизируют работу с данной базой данных.

Полезные мелочи

Быстро найти правильный
ответ на трудный вопрос — ни
с чем не сравнимое
удовольствие

Макс Фрай. Обжора-Хохотун

14.1 Введение

Иногда возникают очень интересные проблемы по работе с PostgreSQL, которые при нахождении ответа поражают своей лаконичностью, красотой и простым исполнением. В данной главе я решил собрать интересные методы решения разных проблем, с которыми сталкиваются люди при работе с PostgreSQL.

14.2 Мелочи

Размер объектов в базе данных

Данный запрос показывает размер объектов в базе данных (например, таблиц и индексов).

[Скачать](#) snippets/biggest_relations.sql

```
Line 1 SELECT nspname || '.' || relname AS "relation",  
-       pg_size_pretty(pg_relation_size(C.oid)) AS "size"  
- FROM pg_class C  
- LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)  
5 WHERE nspname NOT IN ('pg_catalog', 'information_schema')  
- ORDER BY pg_relation_size(C.oid) DESC  
- LIMIT 20;
```

Пример вывода:

Листинг 14.1 Поиск самых больших объектов в БД. Пример вывода

```

Line 1          relation          |      size
-  -----+-----
-  public.accounts              | 326 MB
-  public.accounts_pkey         | 44 MB
5  public.history                | 592 kB
-  public.tellers_pkey          | 16 kB
-  public.branches_pkey         | 16 kB
-  public.tellers                | 16 kB
-  public.branches              | 8192 bytes

```

Размер самых больших таблиц

Данный запрос показывает размер самых больших таблиц в базе данных.

[Скачать snippets/biggest_tables.sql](#)

```

Line 1 SELECT nspname || '.' || relname AS "relation",
-      pg_size_pretty(pg_total_relation_size(C.oid)) AS "
-      total_size"
- FROM pg_class C
- LEFT JOIN pg_namespace N ON (N.oid = C.renamespace)
5 WHERE nspname NOT IN ('pg_catalog', 'information_schema')
-      AND C.relkind <> 'i'
-      AND nspname !~ '^pg_toast'
- ORDER BY pg_total_relation_size(C.oid) DESC
- LIMIT 20;

```

Пример вывода:

Листинг 14.2 Размер самых больших таблиц. Пример вывода

```

Line 1          relation          | total_size
-  -----+-----
-  public.actions                | 4249 MB
-  public.product_history_records | 197 MB
5  public.product_updates        | 52 MB
-  public.import_products        | 34 MB
-  public.products               | 29 MB
-  public.visits                 | 25 MB

```

«Средний» count

Данный метод позволяет узнать приблизительное количество записей в таблице. Для огромных таблиц этот метод работает быстрее, чем обычный count.

[Скачать snippets/count_estimate.sql](#)

```

Line 1 CREATE FUNCTION count_estimate(query text) RETURNS integer
      AS $$
- DECLARE
-     rec    record;
-     rows  integer;
5 BEGIN
-     FOR rec IN EXECUTE 'EXPLAIN ' || query LOOP
-         rows := substring(rec."QUERY PLAN" FROM ' rows=([:
digit:|]|+)' );
-         EXIT WHEN rows IS NOT NULL;
-         END LOOP;
10
-     RETURN rows;
- END;
- $$ LANGUAGE plpgsql VOLATILE STRICT;

```

Пример:

Листинг 14.3 «Средний» count. Пример

```

Line 1 CREATE TABLE foo (r double precision);
- INSERT INTO foo SELECT random() FROM generate_series(1,
      1000);
- ANALYZE foo;
-
5 # SELECT count(*) FROM foo WHERE r < 0.1;
- count
- -----
-      92
- (1 row)
10
- # SELECT count_estimate('SELECT * FROM foo WHERE r < 0.1');
- count_estimate
- -----
-             94
15 (1 row)

```

Случайное число из диапазона

Данный метод позволяет взять случайное число из указанного диапазона (целое или с плавающей запятой).

[Скачать snippets/random_from_range.sql](#)

```

Line 1 CREATE OR REPLACE FUNCTION random(numeric , numeric)
- RETURNS numeric AS
- $$
-     SELECT ($1 + ($2 - $1) * random())::numeric;

```

```
5 $$ LANGUAGE 'sql' VOLATILE;
```

Пример:

Листинг 14.4 Случайное число из диапазона. Пример

```
Line 1 SELECT random(1,10)::int , random(1,10);
-   random |          random
-   -----+-----
-         6 | 5.11675184825435
5 (1 row)
-
- SELECT random(1,10)::int , random(1,10);
-   random |          random
-   -----+-----
10        7 | 1.37060070643201
- (1 row)
```

Алгоритм Луна

Алгоритм Луна или формула Луна — алгоритм вычисления контрольной цифры, получивший широкую популярность. Он используется, в частности, при первичной проверке номеров банковских пластиковых карт, номеров социального страхования в США и Канаде. Алгоритм был разработан сотрудником компании «IBM» Хансом Петером Луном и запатентован в 1960 году.

Контрольные цифры вообще и алгоритм Луна в частности предназначены для защиты от случайных ошибок, а не преднамеренных искажений данных.

Алгоритм Луна реализован на чистом SQL. Обратите внимание, что эта реализация является чисто арифметической.

[Скачать snippets/luhn_algorithm.sql](#)

```
Line 1 CREATE OR REPLACE FUNCTION luhn_verify(int8) RETURNS BOOLEAN
      AS $$
-   -- Take the sum of the
-   -- doubled digits and the even-numbered undoubled digits ,
-   -- and see if
-   -- the sum is evenly divisible by zero.
5 SELECT
-   -- Doubled digits might in turn be two digits. In
-   -- that case ,
-   -- we must add each digit individually rather than
-   -- adding the
-   -- doubled digit value to the sum. Ie if the
-   -- original digit was
-   -- '6' the doubled result was '12' and we must add
-   -- '1+2' to the
```


14.2. Мелочи

```
10         -- sum rather than '12'.
-         MOD(SUM(doubled_digit / INT8 '10' + doubled_digit %
-             INT8 '10'), 10) = 0
- FROM
- -- Double odd-numbered digits (counting left with
- -- least significant as zero). If the doubled digits end up
15 -- having values
- -- > 10 (ie they're two digits), add their digits together.
- (SELECT
-     -- Extract digit 'n' counting left from least
-     significant
-     -- as zero
20     MOD( ( $1::int8 / (10^n)::int8 ), 10::int8)
-     -- Double odd-numbered digits
-     * (MOD(n,2) + 1)
-     AS doubled_digit
-     FROM generate_series(0, CEIL(LOG( $1 ))::INTEGER -
-         1) AS n
25 ) AS doubled_digits;
-
- $$ LANGUAGE 'SQL'
- IMMUTABLE
- STRICT;
30
- COMMENT ON FUNCTION luhn_verify(int8) IS 'Return true iff
-     the last digit of the
- input is a correct check digit for the rest of the input
-     according to Luhn''s
- algorithm.';
- CREATE OR REPLACE FUNCTION luhn_generate_checkdigit(int8)
-     RETURNS int8 AS $$
35 SELECT
-     -- Add the digits, doubling even-numbered digits (
-     counting left
-     -- with least-significant as zero). Subtract the
-     remainder of
-     -- dividing the sum by 10 from 10, and take the
-     remainder
-     -- of dividing that by 10 in turn.
40     ((INT8 '10' - SUM(doubled_digit / INT8 '10' +
-         doubled_digit % INT8 '10') %
-             INT8 '10') % INT8 '10')::INT8
- FROM (SELECT
-     -- Extract digit 'n' counting left from least
-     significant\
-     -- as zero
45     MOD( ($1::int8 / (10^n)::int8), 10::int8 )
```

14.2. Мелочи

```
-         -- double even-numbered digits
-         * (2 - MOD(n,2))
-         AS doubled_digit
-         FROM generate_series(0, CEIL(LOG($1))::INTEGER - 1)
-         AS n
50 ) AS doubled_digits;
-
- $$ LANGUAGE 'SQL'
- IMMUTABLE
- STRICT;
55
- COMMENT ON FUNCTION luhn_generate_checkdigit(int8) IS 'For
- the input
- value, generate a check digit according to Luhn's algorithm
- ';
- CREATE OR REPLACE FUNCTION luhn_generate(int8) RETURNS int8
- AS $$
- SELECT 10 * $1 + luhn_generate_checkdigit($1);
60 $$ LANGUAGE 'SQL'
- IMMUTABLE
- STRICT;
-
- COMMENT ON FUNCTION luhn_generate(int8) IS 'Append a check
- digit generated
65 according to Luhn's algorithm to the input value. The input
- value must be no
- greater than (maxbigint/10).';
- CREATE OR REPLACE FUNCTION luhn_strip(int8) RETURNS int8 AS
- $$
- SELECT $1 / 10;
- $$ LANGUAGE 'SQL'
70 IMMUTABLE
- STRICT;
-
- COMMENT ON FUNCTION luhn_strip(int8) IS 'Strip the least
- significant digit from
- the input value. Intended for use when stripping the check
- digit from a number
75 including a Luhn's algorithm check digit.';
```

Пример:

Листинг 14.5 Алгоритм Луна. Пример

```
Line 1 Select luhn_verify(49927398716);
- luhn_verify
- -----
- t
5 (1 row)
```

```

-
- Select luhn_verify(49927398714);
-   luhn_verify
-   -----
10  f
-   (1 row)

```

Выборка и сортировка по данному набору данных

Выбор данных по определенному набору данных можно сделать с помощью обыкновенного `IN`. Но как сделать подобную выборку и отсортировать данные в том же порядке, в котором передан набор данных? Например:

Дан набор: (2,6,4,10,25,7,9). Нужно получить найденные данные в таком же порядке т. е. 2 2 2 6 6 4 4

[Скачать snippets/order_like_in.sql](#)

```

Line 1 SELECT foo.* FROM foo
- JOIN (SELECT id.val, row_number() over () FROM (VALUES(3),(2)
-   ,(6),(1),(4)) AS
-   id(val)) AS id
- ON (foo.catalog_id = id.val) ORDER BY row_number;

```

где

`VALUES(3),(2),(6),(1),(4)` — наш набор данных

`foo` — таблица, из которой идет выборка

`foo.catalog_id` — поле, по которому ищем набор данных (замена `foo.catalog_id IN (3,2,6,1,4)`)

Quine — запрос который выводит сам себя

Куайн, квайн (англ. quine) — компьютерная программа (частный случай метапрограммирования), которая выдаёт на выходе точную копию своего исходного текста.

[Скачать snippets/quine.sql](#)

```

Line 1 select a || ' from (select ' || quote_literal(a) || b || ',
-   ' || quote_literal(b) || '>::text as b) as quine' from
- (select 'select a || '' from (select '' || quote_literal(a)
-   || b || ''', '' || quote_literal(b) || '>::text as b) as
- quine''>::text as a, '>::text as a'>::text as b) as quine;

```

Поиск дубликатов индексов

Запрос находит индексы, созданные на одинаковый набор столбцов (такие индексы эквивалентны, а значит бесполезны).

[Скачать snippets/duplicate_indexes.sql](#)

```

Line 1 SELECT pg_size_pretty(sum(pg_relation_size(idx))::bigint) AS
      size ,
-      (array_agg(idx))[1] AS idx1 , (array_agg(idx))[2] AS
      idx2 ,
-      (array_agg(idx))[3] AS idx3 , (array_agg(idx))[4] AS
      idx4
- FROM (
5     SELECT indexrelid::regclass AS idx, (indrelid::text ||E'
      \n' || indclass::text ||E'\n' || indkey::text ||E'\n' ||
-                                     coalesce(indexprs::
      text, '' ) ||E'\n' || coalesce(indpred::text, '')) AS KEY
-     FROM pg_index) sub
- GROUP BY KEY HAVING count(*)>1
- ORDER BY sum(pg_relation_size(idx)) DESC;

```

Размер и статистика использования индексов

[Скачать snippets/indexes_statistic.sql](#)

```

Line 1 SELECT
-     t.tablename ,
-     indexname ,
-     c.reltuples AS num_rows,
5     pg_size_pretty(pg_relation_size(quote_ident(t.tablename)
      ::text)) AS table_size ,
-     pg_size_pretty(pg_relation_size(quote_ident(indexrelname)
      ::text)) AS index_size ,
-     CASE WHEN x.is_unique = 1 THEN 'Y'
-     ELSE 'N'
-     END AS UNIQUE,
10     idx_scan AS number_of_scans ,
-     idx_tup_read AS tuples_read ,
-     idx_tup_fetch AS tuples_fetched
- FROM pg_tables t
- LEFT OUTER JOIN pg_class c ON t.tablename=c.relname
15 LEFT OUTER JOIN
-     (SELECT indrelid ,
-      max(CAST(indisunique AS integer)) AS is_unique
-     FROM pg_index
-     GROUP BY indrelid) x
20     ON c.oid = x.indrelid
- LEFT OUTER JOIN
-     ( SELECT c.relname AS ctablename, ipg.relname AS
      indexname, x.indnatts AS number_of_columns, idx_scan ,
      idx_tup_read , idx_tup_fetch ,indexrelname FROM pg_index x

```

```

-         JOIN pg_class c ON c.oid = x.indrelid
-         JOIN pg_class ipg ON ipg.oid = x.indexrelid
25         JOIN pg_stat_all_indexes psai ON x.indexrelid =
psai.indexrelid )
-     AS foo
-     ON t.tablename = foo.ctablename
- WHERE t.schemaname='public'
- ORDER BY 1,2;

```

Размер распухания (bloat) таблиц и индексов в базе данных

Запрос, который показывает «приблизительный» bloat (раздутие) таблиц и индексов в базе:

[Скачать snippets/bloating.sql](#)

```

Line 1 WITH constants AS (
-     SELECT current_setting('block_size')::numeric AS bs, 23 AS
hdr, 4 AS ma
- ), bloat_info AS (
-     SELECT
5     ma, bs, schemaname, tablename,
-     (datawidth+(hdr+ma-(case when hdr%ma=0 THEN ma ELSE hdr%
ma END)))::numeric AS datahdr,
-     (maxfracsum*(nullhdr+ma-(case when nullhdr%ma=0 THEN ma
ELSE nullhdr%ma END))) AS nullhdr2
- FROM (
-     SELECT
10     schemaname, tablename, hdr, ma, bs,
-     SUM((1-null_frac)*avg_width) AS datawidth,
-     MAX(null_frac) AS maxfracsum,
-     hdr+(
-     SELECT 1+count(*)/8
15     FROM pg_stats s2
-     WHERE null_frac <> 0 AND s2.schemaname = s.schemaname
AND s2.tablename = s.tablename
-     ) AS nullhdr
-     FROM pg_stats s, constants
-     GROUP BY 1,2,3,4,5
20 ) AS foo
- ), table_bloat AS (
-     SELECT
-     schemaname, tablename, cc.relpages, bs,
-     CEIL((cc.reltuples*((datahdr+ma-
25     (CASE WHEN datahdr%ma=0 THEN ma ELSE datahdr%ma END))+
nullhdr2+4))/(bs-20::float)) AS otta
- FROM bloat_info
- JOIN pg_class cc ON cc.relname = bloat_info.tablename

```

```

- JOIN pg_namespace nn ON cc.relnamespace = nn.oid AND nn.
  nspname = bloat_info.schemaname AND nn.nspname <> '
  information_schema'
- ), index_bloat AS (
30 SELECT
-   schemaname, tablename, bs,
-   COALESCE(c2.relname, '?') AS iname, COALESCE(c2.reltuples
,0) AS ituples, COALESCE(c2.relpages,0) AS ipages,
-   COALESCE(CEIL((c2.reltuples*(datahdr-12))/(bs-20::float)
),0) AS iotta -- very rough approximation, assumes all
  cols
- FROM bloat_info
35 JOIN pg_class cc ON cc.relname = bloat_info.tablename
- JOIN pg_namespace nn ON cc.relnamespace = nn.oid AND nn.
  nspname = bloat_info.schemaname AND nn.nspname <> '
  information_schema'
- JOIN pg_index i ON indrelid = cc.oid
- JOIN pg_class c2 ON c2.oid = i.indexrelid
- )
40 SELECT
-   type, schemaname, object_name, bloat, pg_size_pretty(
  raw_waste) as waste
- FROM
- (SELECT
-   'table' as type,
45   schemaname,
-   tablename as object_name,
-   ROUND(CASE WHEN otta=0 THEN 0.0 ELSE table_bloat.relpages/
  otta::numeric END,1) AS bloat,
-   CASE WHEN relpages < otta THEN '0' ELSE (bs*(table_bloat.
  relpages - otta)::bigint)::bigint END AS raw_waste
- FROM
50   table_bloat
-   UNION
-   SELECT
-   'index' as type,
-   schemaname,
55   tablename || '::' || iname as object_name,
-   ROUND(CASE WHEN iotta=0 OR ipages=0 THEN 0.0 ELSE ipages/
  iotta::numeric END,1) AS bloat,
-   CASE WHEN ipages < iotta THEN '0' ELSE (bs*(ipages - iotta))
  ::bigint END AS raw_waste
- FROM
-   index_bloat) bloat_summary
60 ORDER BY raw_waste DESC, bloat DESC

```

Литература

- [1] Алексей Борзов (Sad Spirit) borz_off@cs.msu.su
PostgreSQL: настройка производительности
<http://www.phpclub.ru/detail/store/pdf/postgresql-performance.pdf>
- [2] Eugene Kuzin eugene@kuzin.net Настройка репликации в PostgreSQL
с помощью системы Slony-I <http://www.kuzin.net/work/sloniki-privet.html>
- [3] Sergey Konoplev gray.ru@gmail.com Установка Londiste в подробностях
<http://gray-hemp.blogspot.com/2010/04/londiste.html>
- [4] Dmitry Stasyuk Учебное руководство по pgpool-II
<http://undenied.ru/2009/03/04/uchebnoe-rukovodstvo-po-pgpool-ii/>
- [5] Чиркин Дима dmitry.chirkin@gmail.com Горизонтальное масштабирование PostgreSQL с помощью PL/Proху
<http://habrahabr.ru/blogs/postgresql/45475/>
- [6] Иван Блинков wordpress@insight-it.ru Hadoop <http://www.insight-it.ru/masshtabiruemost/hadoop/>
- [7] Pdraig O’Sullivan Up and Running with HadoopDB
<http://posulliv.github.com/2010/05/10/hadoopdb-mysql.html>
- [8] Иван Золотухин Масштабирование PostgreSQL: готовые решения от Skype <http://postgresmen.ru/articles/view/25>
- [9] Streaming Replication. http://wiki.postgresql.org/wiki/Streaming_Replication
- [10] Den Golotyuk Шардинг, партиционирование, репликация - зачем и когда? <http://highload.com.ua/index.php/2009/05/06/шардинг-партиционирование-репликац/>
- [11] Postgres-XC — A PostgreSQL Clustering Solution
<http://www.linuxforu.com/2012/01/postgres-xc-database-clustering-solution/>

Литература

- [12] Введение в PostgreSQL BDR <http://habrahabr.ru/post/227959/>
- [13] Популярный обзор внутренностей базы данных. Часть пятая <http://zamotivator.livejournal.com/332814.html>
- [14] BRIN-индексы в PostgreSQL <http://langtoday.com/?p=485>
- [15] Huge Pages в PostgreSQL <https://habrahabr.ru/post/228793/>
- [16] Greenplum DB <https://habrahabr.ru/company/tinkoff/blog/267733/>
- [17] Введение в PostGIS https://live.osgeo.org/ru/quickstart/postgis_quickstart.html
- [18] Введение в полнотекстовый поиск в PostgreSQL http://www.sai.msu.su/~megeera/postgres/talks/fts_pgsql_intro.html
- [19] pg_arman <https://postgrespro.ru/docs/postgrespro/9.5/pg-arman.html>
- [20] It Probably Works <http://queue.acm.org/detail.cfm?id=2855183>
- [21] Кластер PostgreSQL высокой надежности на базе Patroni, Нароуху, Keepalived <https://habrahabr.ru/post/322036/>