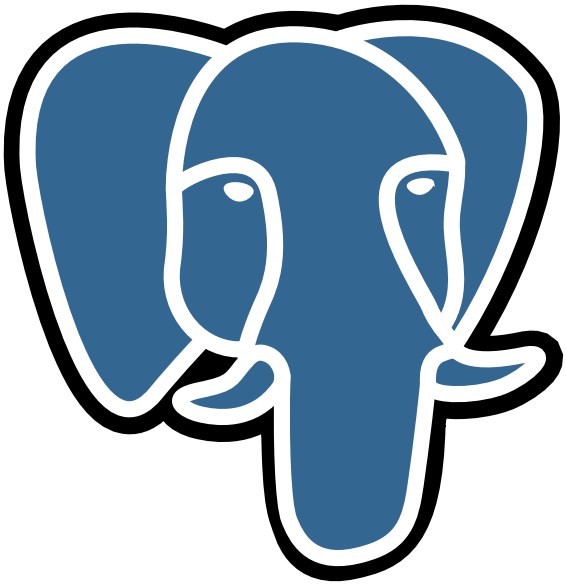

# История о PostgreSQL

© Е.М. Балдин*

Эта статья была опубликована в апрельском номере русскоязычного журнала Linux Format (`http://www.linuxformat.ru`) за 2007 год. Статья размещена с разрешения редакции журнала на сайте `http://www.inp.nsk.su/~baldin/` и до конца сентября месяца все вопросы с размещением статьи в других местах следует решать с редакцией Linux Format. Затем все права на текст возвращаются ко мне.

Текст, представленный здесь, не является точной копией статьи в журнале. Текущий текст в отличии от журнального варианта корректор не просматривал. Все вопросы по содержанию, а так же замечания и предложения следует задавать мне по электронной почте `mailto:E.M.Baldin@inp.nsk.su`.

Текст на текущий момент является просто *текстом*, а не книгой. Поэтому результирующая доводка в целях улучшения восприятия текста не проводилась.

*e-mail: E.M.Baldin@inp.nsk.su

Слон взят с сайта `http://pgfoundry.org/projects/graphics/`. Изображение предоставляется под лицензией BSD.

# Оглавление

Устреми свои мысли на высшее Я, свободный от вожделения и себялюбия, исцелившись от душевной горячки, сражайтесь, Арджуна!

*Зеркало.* Понедельник начинается в субботу.

Глава 6

# Дополнительные главы

Рассказать и предусмотреть всё не реально. Хотя бы по той простой причине, что составляя планы мы изменяем реальность. Изменённая реальность в свою очередь требует изменённых планов и так до бесконечности. Но некоторые особенности жалко не раскрыть чуть более подробно.

## 6.1 Резервное копирование

Если уж завели хранилище информации, то его надо беречь. При этом навязчивая идея на тему порчи данных, переходящая в манию, является обязательной характеристикой нормального администратора базы данных. Только в этом случае можно подстелить соломы в нужном месте, до того как подскользнуться. И то, что при этом весь дом будет покрыт высушенной травой — обычные издержки производства.

### 6.1.1 pg_dump/pg_restore

Просто копировать физические файлы базы данных не самый лучший способ для бэкапа, потому что на момент операции придётся как минимум остановить сервер. Для создания консистентной копии базы данных проще всего воспользоваться программой **pg_dump** (man pg_dump), которая работает как обычный клиент:

```
> pg_dump -Fc «база данных» > «файл резервной копии»
```

Опция `-Fc` определяет формат резервной копии как custom. В этом случае сохраняются не только SQL-структуры, но и большие объекты (lobj).

Для восстановления базы данных из её резервной копии используется зеркальная процедура **pg_restore** (man pg_restore):

```
> pg_restore -d «новая база данных» «файл резервной копии»
```

Используя **pg_restore** с помощью опции `-l` можно получить список всех таблиц находящихся в резервной копии, а с помощь опции `-L` указать список таблиц

которые надо восстановить. Иногда может потребоваться только частичное восстановление данных, например, для отката только конкретной таблицы.

Так как **pg_dump** и **pg_restore** сконструированы с учётом работы в конвейере, то их удобно использовать в скриптах. Резервная копия представляет из себя в основном ASCII-файл, поэтому при формировании процедуры бэкапа/восстановления имеет смысл предусмотреть фильтр для сжатия данных, например, **bzip2**.

При восстановлении больших объектов (lobj) очень важно, что **pg_restore** отработало без ошибок от начала и до конца. Причина этого в том, что при восстановлении больших объектов создаётся временная таблица, где есть перекодировка из старой нумерации OIDов в новую. Если в процессе восстановления произошло прерывание, то эта таблица теряется и ссылки на большие объекты в таблицах не обновляются. В результате большие объекты в базу данных загружаются, но ссылки на них отсутствуют. Это один из примеров того, как нестандартные расширения могут приводить к неудобствам.

### 6.1.2 Непрерывный бэкап

**pg_dump** умеет создавать резервную копию особо не мешая функционированию базы данных, так как это всего на всего ещё один клиент. Есть одна неприятность в классическом подходе резервирования: информация между бэкапом и крахом базы данных теряется. Иногда это терпимо, так как подобное случается редко, но есть случаи, когда потерянные запросы означают потерянные деньги в полном смысле этого слова. И здесь на помощь приходит журналирование транзакций. Находка для параноика.

Организация непрерывного бэкапа довольно сложная процедура и для её реализации следует обратиться к разделу документации, который так и называется «Online backup and point-in-time recovery (PITR)». В этой главе представлено пошаговое руководство к действию длинной чуть меньше пяти тысяч слов, что составляет около пятнадцати страниц текста на A4.

Основная идея заключается в архивации журнала транзакций. Формально все действия PostgreSQL можно представить как последовательные записи в этом журнале. На диске журнал транзакций разбивается на независимые файлы или сегменты (segment files) размер которых по умолчанию равен 16 Мб. PostgreSQL можно настроить на копирование сегментов в место для бэкапа (параметр `archive_command` в `postgresql.conf`). При этом нет необходимости хранить абсолютно все записи. Достаточно оставлять только те, которые были сделаны после бэкапа. Для локализации времени, которому соответствует резервная копия сделанная во время процедуры бэкапа, используются хранимые функции pg_start_backup/pg_stop_backup.

При восстановлении можно восстановить не только текущее состояние базы данных, но и состояние в котором она была на указанный момент времени. Естественно всё лимитируется объёмом сохранённых сегментов. Таким образом при желании можно организовать своеобразное путешествие в прошлое (point-in-time recovery).

## 6.2 Переезд на новую версию PostgreSQL

По умолчанию при исполнении `pg_dump` на выходе получаются SQL-команды. Так что для восстановления можно воспользоваться **psql**, указав файл резервной копии с помощью ключика `-f`. То есть структура резервной копии зависит только от версии SQL которую поддерживает данный сервер. Это позволяет достаточно легко обновлять PostgreSQL даже если изменяется представление данных внутри самого PostgreSQL, так как SQL и в Африке SQL.

Поэтому переезд с версии на версию гарантировано можно выполнить в четыре этапа:

1. сделать резервную копию с помощью **pg_dumpall**,
2. остановить старый сервер,
3. запустить новый сервер,
4. восстановить базу данных с помощью **pg_restore** или **psql**.

Если меняется только минорная версия PostgreSQL (последняя цифра в версии), то в принципе можно упустить этап 1 и 4. Но в любом случае не следует забывать о фобии потери данных. В принципе можно исключить пункты 2 и 3 воспользовавшись конвейером:

```
> pg_dump -h host1 «БД» | psql -h host2 «БД»
```

host1 и host2 — компьютеры с какого и на какой, соответственно, переезжает база данных.

## 6.3 Репликация слонов

База данных подразумевает централизацию: всё складывается в одно место. Это может стать проблемой. Некоторые проблемы не решаются, но если требуется всего на всего ускорить доступ на чтение, то репликация базы данных может оказаться спасением. Побочным эффектом репликации является повышение надёжности базы, так как число консистентных копий данных увеличивается. Создание кластера баз данных — это глобальный инструмент для решения многих проблем, но и сложности в управлении кластером также будет предостаточно.

Для репликации PostgreSQL существует несколько решений, как закрытых[1], так и свободных. Самой популярной свободной системой репликации является Slony I (`http://slony.info/`). Slony I поддерживает master/slaves репликацию[2]. Возможные преимущества которые можно получить, наладив репликацию:

[1] Например, `http://www.commandprompt.com/products/mammothreplicator` — Mammoth PostgreSQL + Replication.

[2] Имя Slony-II зарезервировано для версии, которая будет поддерживает multi-master режим. На текущий момент будущее этой версии довольно туманно. Организовать надёжное решение

- Организуются дополнительные копии данных, которые никогда лишними не бывают. Помним о благотворном влиянии паранойи.
- Разгружается центральный сервер, теперь он может заниматься действительно важными делами не отвлекаясь на мелочи.

  Например, процедура полного бэкапа довольно ресурсоёмкая. Вполне можно поручить это задание одному из вспомогательных серверов. Аналогично можно организовать сервер, который имеет очень длинный лог транзакций, чтобы можно было откатиться максимально далеко по времени в случае необходимости.
- Можно перенести вспомогательный сервер поближе к клиенту, чтобы не было проблем со временем, которое уходит на подсоединение к базе данных,
- Дополнительные сервера позволяют таки доступиться к данным, даже если связь с центральным сервером полностью потеряна.

Вспомогательные сервера вовсе не обязаны получать обновления непосредственно с главного сервера (Master to multiple cascades Slaves). Любой сервер, который получает данные из надёжного источника может быть сконфигурирован так, чтобы рассылать эти данные далее по цепочке. Данная особенность позволяет легко масштабировать систему. Развернуть и запустить репликацию можно не останавливая центральный сервер.

Для для привязки к событиям INSERT/DELETE/UPDATE используются триггеры PostgreSQL. Выполнения действий реализуются через хранимые процедуры. Слежением за выполнением репликацию занимается системный демон slon, то есть для работы он должен быть запущен на каждом из узлов кластера. Администрирование осуществляется посредством командного процессора slonik.

Административная утилита slonik реализована как программа, ориентированная на выполнение в командной строке и в скриптах. Синтаксис команд воспринимаемых slonik'ом напоминает SQL. Команды следует передавать на STDIN. Перед исполнением запроса slonik анализирует синтаксис и в случае наличия проблем, команда не исполняется и выдаётся сообщение об ошибке.

Подробно о настройке кластера можно прочитать в документации к пакету. На русском есть написанное Евгением Кузиным пошаговое руководство, правда возможно уже устаревшее: `http://www.kuzin.net/work/sloniki-privet.html`. В случае возникновения проблем для начала следует поискать решение в стандартном FAQ: `http://linuxfinances.info/info/faq.html`.

В качестве побочного эффекта репликации её можно использовать при обновлении версии сервера PostgreSQL. Это удобно когда объём базы данных становится очень большой и останавливать её на момент смены версии не очень бы хотелось.

---

для требуемого режима очень сложно в силу большого количества принципиальных проблем `http://www.dbspecialists.com/presentations/mm_replication.html`.

Для реализации multi-master режима пользователем PostgreSQL поддерживает отложенные транзакции (two-phase commit). Two-phase commit реализуется с помощью SQL-запросов PREPARE TRANSACTION и COMMIT PREPARED.

**Принципиальные ограничения** Большие объекты не реплицируются. Это происходит потому, что Slony I работает на триггерах, а операции с большими объектами триггерным механизмом не отлавливаются. То есть реплицируются только таблицы и последовательности. Для того чтобы репликация работала автоматически лучше отказаться от больших объектов, благо существуют соответствующие бинарные типы данных, вполне годящиеся на их замену.

На начало марта 2007 года последняя версия Slony I была 1.2.2. Для функционирования этой версии необходим PostgreSQL старше 7.3.3, так как требуется обязательная поддержка пространства имён (namespace). При репликации предполагается, что все базы данных создавались с указанием одной и той же кодовой страницы[3] и текущая кодовая страница с ней совпадает. Задача временной синхронизации серверов выходит за рамки функционирования Slon'ов — для этого следует озадачиться созданием специальной службы (Network Time Protocol `www.ntp.org`).

Процедуры изменения схемы базы данных (database schema, DDL — Язык определения данных), следует производить посредством передачи команд через **slonik** посредством префикса EXECUTE SCRIPT. Это гарантирует, что, например, изменение числа столбцов в таблице произойдёт во всём кластере до того как туда начнут добавляться данные.

## 6.4 Локаль

Локаль (locale) — это набор соглашений, специфических для отдельно взятого языка в отдельно взятой стране[4]. Локаль и кодовая страница базы данных выбирается при её создании с помощью команды **initdb**:

```
> initdb --locale=ru_RU.UTF-8 --lc-numeric=POSIX
```

В зависимости от локаль результат выполнения SQL-запросов может отличаться. Например, это проявляется при сортировке текстовых данных или при выполнении функций upper/lower/initcap.

Для корректной работы базы данных с устанавливаемой локалью необходимо, чтобы данная локаль поддерживалось системой. Вывести список поддерживаемых локалей можно с помощью команды `locale -a`.

Так как локализация проводилось Олегом Бартуновым, то все русские кодовые страницы поддерживаются. При наборе русских текстов можно использовать следующие из них: KOI8 (aka KOI8R), WIN1251 (aka WIN), WIN866 (aka ALT), ISO_8859_5, UTF8 (aka Unicode) и MULE_INTERNAL[5].

Кодовая страница клиента может отличаться от кодовой страницы сервера. Например в сессии **psql** кодовую страницу можно установить следующим образом:

[3] Это замечание относится к ключику `--encoding` команды `createdb`.

[4] В общем случае говорить, что локаль определяется только страной, неправильно. Например, в Канаде могут быть определены две локали: язык "Канада/Английский"и язык "Канада/Французский". Аналогично язык "Великобритания/Английский"не эквивалентен языку "Американский/Английский".

[5] То, что используется в emacs.

```
mydb=> \encoding KOI8R
mydb=> show CLIENT_ENCODING;
 client_encoding
-----------------
 KOI8R
(1 запись)
```

При этом на самом деле используется PQsetClientEncoding() — функция **libpq**, которая в свою очередь выполняет SQL-запрос SET CLIENT_ENCODING TO.

После выставки кодовой страницы клиента PostgreSQL выполняет автоматическое преобразование запросов между кодовыми страницами сервер/клиент, если это конечно возможно. Для русских кодовых страниц все варианты преобразований имеются. При желании с помощью SQL-запрос CREATE CONVERSION можно создать свою таблицу преобразования.

## 6.5 VACUUM/ANALYZE

Администрируя PostgreSQL, следует помнить, что для его нормального функционирования следует регулярно «мыть руки» и «чистить зубы», то есть исполнять команды VACUUM и ANALYZE. Это необходимо по той причине, что иначе не получится заново использовать дисковое пространство, которое занимают ранее удалённые или изменённые строки и не удастся обновить статистику для планировщика запросов. И то и другое отрицательно сказывается на эффективности использования ресурсов и производительности запросов.

Начиная с версии PostgreSQL 8.1 сервер может самостоятельно автоматически запускать ещё один системный процесс, который, соответственно, так и называется autovacuum daemon. Все настройки для этого процесса хранятся в `postgresql.conf`. К значениям этих параметров следует отнестись крайне внимательно.

Если по каким-то причинам демон было решено не запускать, то в любом случае необходимо производить сборку мусора и набор статистики в ручную с помощью команды **vacuumdb** (man vacuumdb):

```
> vacuumdb -ze
VACUUM ANALYZE;
VACUUM
```

## 6.6 Мониторирование активности базы

Текущую активность базы данных легко оценить с помощью команды **ps**:

```
> ps auxww | grep ^postgres
postgres ... postmaster -i
postgres ... postgres: writer process
```

```
postgres ... postgres: stats buffer process
postgres ... postgres: stats collector process
postgres ... postgres: baldin mydbase [local] idle
```

Так как для каждого клиента создаётся своя копия процесса **postmaster**, то это позволяет подсчитать число активных клиентов. Статусная строка даёт информацию о состоянии клиента. Фразы writer process, stats buffer process и stats collector process соответствуют системным процессам, запущенным самим PostgreSQL при старте. Пользовательские процессы имеют статусную строку вида

```
postgres: «пользователь» «база» «хост» «статус»
```

«пользователь», «база» и «хост» соответствуют имени пользователя «пользователь» подсоединявшегося к базе «база» с компьютера «хост». «статус» может принимать следующие параметры:

**idle** — ожидание команды от клиента,

**idle in transaction** — ожидание команды от клиента внутри транзакции (между BEGIN и окончанием транзакции),

**SQL-команда** — выполняется эта команда, например, SELECT,

**waiting** — ждём когда разблокируется занятая другим процессом таблица.

Если в `postgresql.conf` разрешён сбор статистики (опции `stats_start_collector` и `stats_row_level`), то информация об активности базы данных собирается в специальных системных таблицах. Ту же информацию, что получается с помощью **ps** можно извлечь из таблицы pg_stat_activity, а в pg_stat_all_tables лежат данные о числе обращений к каждой из таблиц базы. Подробнее обо всех имеющихся таблицах можно прочитать в главе «Viewing Collected Statistics» стандартной документации. Информация собранная «статистическим сборником» может оказаться полезной для оценки эффективности базы данных и запросов. Например, pg_stat_all_indexes поможет оценить эффективность и частоту использования индексов при реальной работе. Подробную информация о блокировках можно почерпнуть в таблице pg_locks.

## 6.7 log

Когда что-то работает бывает полезно иметь обратную связь. Поэтому лучше чтобы журнальный файл (log) существовал. Создавать ли лог-файл самостоятельно или воспользоваться службой **syslog** это зависит от обстоятельств. Следует только учитывать, что **syslog** на каждой записи производит операцию **sync**, что может серьёзно замедлить доступ к диску на котором лежит журнальный файл. Это так же следует учитывать.

## 6.8 Послесловие

Вот и закончилась серия статей о PostgreSQL. Но надеюсь Ваше взаимодействие с этим замечательным образчиком программного искусства будет только расширяться. С одной стороны довольно странно, что одну программу пришлось рассматривать на протяжении целых шести глав. Причём дальше обзора, как правило, зайти не удавалось. С другой стороны задача сохранения и доступ к уже имеющимся данным является одной из самых важных в буквальном смысле для выживания человечества. Поэтому эта области деятельности в информатике является, пожалуй, самой развитой. Здесь теория переходит в практику.

Далеко не всё упомянуто и далеко не все рассказано достаточно подробно. Та же тема полнотекстового поиска просто сама по себе может занять не одну главу. Картографические и астрофизические типы данных так же имеют за собой фундаментальную базу и интересные сферы приложения. PostgreSQL — это хранилище данных, но за каждым типом данных своя уникальная история и инструкция по использованию. Тема супербольших баз данных, где кластеризация является обязательной суперсложна и не менее интересна. Переход к многопроцессорным архитектурам увеличивает сферы применимости баз данных. Я не сильно удивлюсь, что угрозы, сделать из файловой системы специализированную базу данных, воплотятся в обычную базу данных общего применения, которая будет хранить в себе файловую систему. Хотя бы пользовательские документы, где не особо важна скорость доступа, но естественный полнотекстовый поиск будет весьма кстати. В любом случае в будущем без баз данных делать нечего и PostgreSQL там будет наверняка. Присоединяйтесь.

## Врезка: К вопросу о происхождении Слонов

В документации к пакету Slony I для англоязычной аудитории идёт специальное разъяснение:

- слон — это русский elephant,
- множественная форма от слова слон — это слоны,
- слоник — это маленький elephant.

Термин Slony I — это реверанс в сторону Вадима Михеева, который создал прототип для системы репликации **rserv** на языке **perl**. Проект был спонсирован фирмой Afilias (`http://afilias.info`), которая наняла для этого одного из основных разработчиков PostgreSQL Яна Вейка (Jan Wieck). Со слов Яна с самого начала проект планировался как программа с открытыми исходниками, которые всегда были доступны публично. Это яркий пример того, что коммерческие фирмы могут сделать весомый вклад в открытые разработки без каких-либо задних мыслей, как участники свободного сообщества.