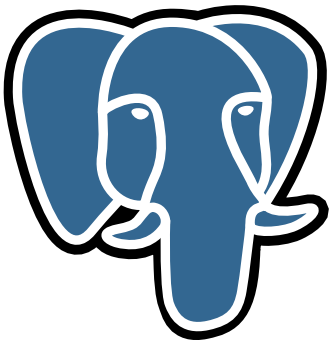

История о PostgreSQL

© Е.М. Балдин*

Этот текст основан на цикле статей, опубликованных с 85го (ноябрь 2006) по 91ый (апрель 2007) номер русскоязычного журнала Linux Format.

Новосибирск, 2006 (ревизия 15.10.2013)

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

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

Оглавление

1. Введение **1**
- 1.1. Это кто такой? . . . 1
  - 1.1.1. Реляционная база данных . . . 1
  - 1.1.2. Открытый исходный код . . . 2
- 1.2. Генеалогия . . . 2
- 1.3. А как оно работает? . . . 3
- 1.4. Установка и запуск . . . 3
- 1.5. Почему? . . . 8
  - 1.5.1. Почему БД? . . . 8
  - 1.5.2. Почему PostgreSQL? . . . 8
- 1.6. Информация о subj . . . 10

2. Работа с базой **11**
- 2.1. SQL . . . 11
- 2.2. Командная строка . . . 13
  - 2.2.1. psql . . . 13
  - 2.2.2. gql-shell . . . 16
  - 2.2.3. dbishell . . . 16
- 2.3. GUI в помощь . . . 16
  - 2.3.1. PgAccess . . . 17
  - 2.3.2. pgAdmin III . . . 18
  - 2.3.3. TOra . . . 20
  - 2.3.4. OpenOffice и SDBC . . . 21
- 2.4. Что выбрать? . . . 22

3. Возможности PostgreSQL **24**
- 3.1. Чуть-чуть про основы . . . 24
- 3.2. Типы данных . . . 25
  - 3.2.1. Числовые типы . . . 25
  - 3.2.2. Символьные типы . . . 26
  - 3.2.3. Бинарные типы . . . 26
  - 3.2.4. Типы даты/времени . . . 26

3.2.5. Логические типы . . . 27
3.2.6. Остальные стандартные типы . . . 27
3.2.7. Определение пользовательских типов . . . 28
3.3. Функции . . . 28
3.3.1. Хранимые процедуры . . . 29
3.3.2. Триггеры . . . 30
3.3.3. Rules . . . 32
3.4. Индексы . . . 32
3.5. Целостность данных . . . 33
3.5.1. Транзакции . . . 33
3.5.2. Ограничения . . . 34
3.5.3. Блокировки . . . 34

4. Интерфейсы **35**
4.1. libpq . . . 35
4.1.1. Открытие и закрытие соединения . . . 36
4.1.2. SQL запросы . . . 39
4.1.3. Большие объекты . . . 42
4.2. ECPG . . . 42
4.3. Всё остальное . . . 44

5. Настройка PostgreSQL **47**
5.1. О железе . . . 47
5.2. Конфигурационные файлы . . . 48
5.2.1. pg_hba.conf . . . 49
5.2.2. postgresql.conf . . . 51
5.3. О том, что думать тоже надо . . . 59

6. Дополнительные главы **61**
6.1. Резервное копирование . . . 61
6.1.1. pg_dump/pg_restore . . . 61
6.1.2. Непрерывный бэкап . . . 62
6.2. Переезд на новую версию PostgreSQL . . . 63
6.3. Репликация слонов . . . 63
6.4. Локаль . . . 66
6.5. VACUUM/ANALYZE . . . 67
6.6. Мониторирование активности базы . . . 67
6.7. log . . . 68

Послесловие **69**

Глава 1.

Введение

Разве же так можно? Разве же
такие вещи алгоритмизируешь?

Магнус Ф. Редькин об определениях
счастья.

Новая информация добывается потом и кровью. Чтобы не утерять найденное — её надо сохранить. А чтобы потом суметь найти необходимое — её следует структурировать. PostgreSQL — предназначен для постоянного[1] хранения структурированных данных[2].

1.1. Это кто такой?

PostgreSQL — это реляционная база данных. PostgreSQL — это программный продукт с открытым исходным кодом и свободной (в прямом смысле этого слова) лицензией. Собственно говоря, этим всё сказано.

1.1.1. Реляционная база данных

Информация в реляционных базах данных хранится в виде обычных плоских двумерных таблиц. Доступ к данным в таблице можно получить по её имени. В таблице есть именованные столбцы (column) и строки (row) — очень простая и понятная концепция. Пользователю предоставляется набор операторов, результатом действий которых так же являются таблицы. Это особенность реляционной базы данных называется *замкнутостъ*. Это очень важное свойство, так в результате любых действий порождаются объекты того же типа, что и объект над которым

[1] Постоянность означает сохранность данных, даже если программа перестала работать.

[2] Хранить можно и *не* структурированные данные, но это уже моветон.

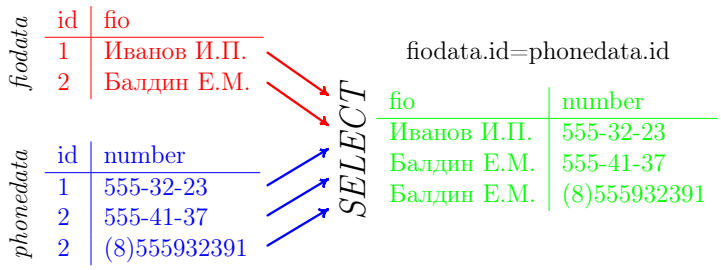

Рис. 1.1. Получение новой таблицы из уже имеющихся

совершались эти самый действия. Следствием замкнутости является возможность применять к результату все имеющиеся в наличии операторы. Иными словами можно пользоваться *вложенными выражениями*[3].

1.1.2. Открытый исходный код

PostgreSQL распространяется под BSD лицензией. Почему не GPL? Ответ разработчиков можно перевести[4] примерно так: «PostgreSQL создавался в Беркли (Berkeley), как, собственно говоря, и лицензия BSD. Эта лицензия служила нам верой и правдой много лет. От добра — добра не ищут. Просьба не начинать опять „флеймить" по этому поводу.»

1.2. Генеалогия

Понятие реляционных баз данных было предложено в 70-ых годах прошлого века сотрудником фирмы IBM Эдгаром Ф. Коддом (Edgar F. Codd). В то время это была революция в сфере хранения данных. Головокружительный успех идей Кодда связан ещё и с тем, что он сумел воплотить математическую абстракцию под названием *реляционная алгебра* в жизнь. Многие ответы на практически вопросы были найдены теоретически с использованием математики.

С тех пор прошло более тридцати лет и новой революции пока не предвидится. Двумерные таблицы ещё долго будут основным методом структурирования информации в силу исключительной простоты решения.

Как и в случае TCP/IP практическое воплощение теории в жизнь началось с того, что DARPA (Defense Advanced Research Projects Agency) дало денег профессору. Профессор Михаил Стоунбрэйкер (Michael Stonebraker) написал реляционную базу данных POSTGRES, первый релиз которой был сделан в 1987 году. Профессор

[3]Вложенные выражения, это многоуровневые выражения, причём, использование имён реальных таблиц обязательно только на самом низком уровне. В остальных случаях в качестве объектов действия могут быть вычисляемые выражения.

[4]Очень вольный перевод.

Стоунбрэйкер писал базу не с нуля. Его проект основывался на одной из самых первых реляционных баз данных Ingres к созданию которой приложил руку сам Кодд — её имя частично присутствует в названии проекта (POST-GRES — после Ingres).

POSTGRES использовался как для реальных дел в качестве СУБД, так и для исследования теории реляционных баз данных в стенах университетов. В 1994 году два студента Андрэ Ю (Andrew Yu) и Джолли Чен (Jolly Chen) добавили движок SQL, который уже к этому моменту стал бесспорным промышленным стандартом для реляционных СУБД. Так появился Postgres95 который в 1996 году сменил имя на PostgreSQL. Имя больше не менялось, но активная разработка не прекращается не на миг. Последняя версия сервера баз данных на февраль 2007 года 8.2.3. Подробнее об истории можно узнать в стандартной документации, идущей с программой или на сайте http://www.postgresql.org.

Семейство Ingres/PostgreSQL породило множество коммерческих реализаций[5] систем управления баз данных, благо лицензия позволяет.

1.3. А как оно работает?

На рис. 1.2 показана схема работы типичного приложения. Приложение посылает запрос процессу POSTMASTER, который существует всегда[6], на подключение. Если запрос на подключение проходит проверку, то POSTMASTER создаёт свою копию. Все дальнейшие операции между базой данных и клиентом проводятся через эту копию POSTMASTER. На каждое соединение создаётся своя копия — это позволяет производить все действия с данными *непосредственно* на сервере.

1.4. Установка и запуск

Установка базы данных это не совсем тривиальная процедура. Лучше довериться стандартной сборке из вашего базового дистрибутива, даже если версия по умолчанию кажется устаревшей[7]. Самостоятельно собирать пакет из исходников лучше только в том случае, если точно известно, что в базовой версии нет необходимой функциональности.

[5]Особенно много потомков у Ingres — та же Sybase. Код Sybase в свою очередь в 1988 году был продан одной известной фирме, которая в 1992 году выпустила продукт в названии которого есть имя этой фирмы и слова «SQL Server». У POSTGRES в прямых потомках ходит Informix.

[6]За исключением тех случаев когда компьютер выключен или сервис был остановлен администратором и *очень* редко по причине какой-либо ошибки. Я не знаю какая статистика у других, но из моего опыта все такого рода ошибки связаны с человеческим фактором.

[7]Базовая версия PostgreSQL в Debian stable (Sarge) на момент написания статьи 7.4.7, в то время как последняя версия базы данных 8.2.3.

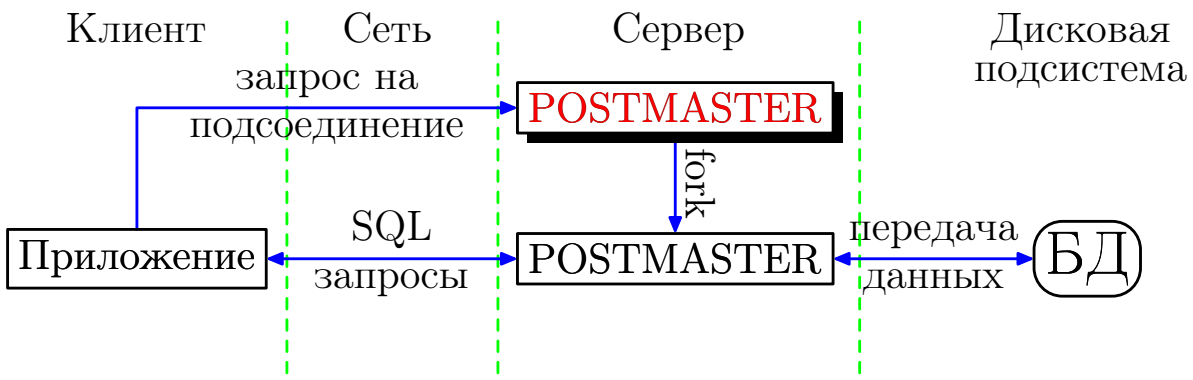

Рис. 1.2. Схема работы приложения с PostgreSQL

Если в будущем необходимо будет сменить версию PostgreSQL, то следует учитывать, что в случае крупных изменений (major releases) могут изменять внутренние форматы системных таблиц и файлов данных. В этих случаях необходимо выполнить процедуру «dump/restore», которая гарантировано сохранит данные при «переезде». В отличии от крупных изменений, небольшие правки (minor releases[8]) как правило не требуют никаких действий со стороны администратора БД.

Число пакетов в дистрибутиве в описании которых упоминается PostgeSQL довольно велико. Например, в Debian (Sarge) таких пакетов 182, что несколько меньше чем число пакетов связанных с именем mysql (212), но превышает число упоминаний InterBase/Firebird (22), sqlite (50) и, естественно, Oracle (19). Это не о чём не говорит, но корреляция, скорее всего, какая-то есть. К счастью все 182 пакета ставить не обязательно — для Debian (Sarge) достаточно двух/трёх пакетов:

```
# устанавливаются исполняемые файлы и файлы настроек
# необходимые для функционирования Базы Данных
> apt-get install postgresql
```

В случае подобной установки в обязательном порядке доставляется базовый набор программ, которые можно ставить на клиентских машинах для удалённой связи с БД — пакет **postgresql-client**.

Если же несмотря ни на что хочется установить всё из исходников самостоятельно, то следует выполнить примерно следующую последовательность действий:

```
> lftp ftp://ftp.postgresql.org/pub/source/v8.2.3/
lftp /pub/source/v8.2.3/> get postgresql-8.2.3.tar.bz2
> tar xvfj postgresql-8.2.3.tar.bz2
> cd postgresql-8.2.3
> ./configure
> make
```

[8]Меняется только последнее число в версии, то есть переход от версии 7.4.0 к версии 7.4.1.

```
> su
> make install
> adduser postgres
> mkdir /usr/local/pgsql/data
> chown postgres /usr/local/pgsql/data
> su - postgres
> /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
> /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data\
+                                 >logfile 2>&1 &
> /usr/local/pgsql/bin/createdb test
> /usr/local/pgsql/bin/psql test
```

Разберём то что происходит поподробнее. После того как с помощью `wget` получен и распакован архив исходников, привычные команды `./configure` и `make` позволяют осуществить сборку PostgreSQL. Установку (`make install`) следует производить под суперпользователем (`su`). После установки необходимо добавить пользователя postgres от имени которого и будет запущен сервер **postmaster**. По умолчанию установка программы производится в директорию `/usr/local/pgsql/`. Для хранения файлов базы предлагается создать директорию `/usr/local/pgsql/data`. Данные должны принадлежать пользователю postgres (команда `chown`). В этой же директории хранятся и файлы настройки.

Дальнейшая настройка производится под пользователем postgres (`su - postgres`). С помощью команды `initdb` производится инициализация хранилища данных, а затем производится запуск сервера `postmaster`. Последние две строчки создают тестовую базу данных test (`createdb`) и проверяют что к ней можно подсоединиться (`psql`). Если всё прошло нормально, то должно появиться приглашение вида:

```
Welcome to psql 8.2.3, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

test=#
```

При установке стандартными средствами дистрибутива описанные выше действия выполняются автоматически кроме последних двух строчек. В случае Debian (Sarge) при установке PostgreSQL можно указать где именно расположить директорию с данными. По умолчанию всё помещается в `/var/lib/postgres/data`. Для других дистрибутивов возможны вариации. Для выяснения подробностей следует изучить README. Например, в случае Debian особенности пакета связанные с дистрибу-

тивом описаны в /usr/share/doc/postgresql/README.Debian.gz. Ниже, если не указано специально, все действия выполняются для дистрибутива Debian (Sarge).

Для администрирования базы данных нет необходимости в суперпользовательских привилегиях. Для этого можно настроить sudo (man sudo), то есть в файл /etc/sudoers (man sudoers) следует добавить примерно следующие строки:

```
# /etc/sudoers
  ...
Host_Alias HOME = localhost
User_Alias DBADM = «ваше имя, если Вы администратор базы данных»
Cmnd_Alias DB  = /etc/init.d/postgresql
DBADM HOME =  NOPASSWD: DB
DBADM HOME = (postgres) NOPASSWD: ALL
  ...
```

Файлы настройки принадлежат пользователю postgres, поэтому для их изменения необходимо иметь возможность заходить под этим пользователем:

```
> sudo -u postgres bash
> whoami
postgres
```

либо добавить себя в группу postgres и разрешить этой группе редактировать конфигурационные файлы в директории /etc/postgres (chgrp плюс chmod g+w).

Скрипт /etc/init.d/postgresql позволяет управлять процессом postmaster

```
> sudo /etc/init.d/postgresql
Usage: /etc/init.d/postgresql {start|stop|autovac-start|
autovac-stop|restart|autovac-restart|reload|force-reload|status}
> sudo /etc/init.d/postgresql status
pg_ctl: postmaster is running (PID: 10868)
Command line was:
/usr/lib/postgresql/bin/postmaster '-D' '/home/postgres/data'
```

Этот же скрипт используется для автоматического запуска сервера при загрузке компьютера. От дистрибутива к дистрибутиву название инициализирующего скрипта может меняться.

После настройки сервера необходимо создать базу данных:

```
> sudo -u postgres createdb «имя БД»
CREATE DATABASE
```

и завести пользователя:

```
> sudo -u postgres createuser «имя пользователя»
Разрешить новому пользователю создавать базы? (y/n) n
Разрешить новому пользователю создавать пользователей? (y/n) n
```

```
CREATE USER
```

Для того чтобы пройти проверку при запросе на подключения необходимо, чтобы конфигурационный файл `pg_hba.conf` был соответствующим образом настроен. Например, чтобы можно было подключаться к базе данных под тем же именем под которым Вы работаете, в `pg_hba.conf` должны быть примерно следующие строки:

```
#/etc/postgresql/pg_hba.conf
# TYPE  DATABASE  USER IP-ADDRESS   IP-MASK        METHOD
# connections by UNIX sockets
local   all       all                             ident sameuser
# All IPv4 connections from localhost
host    all       all  127.0.0.1 255.255.255.255 ident sameuser
```

Здесь в качестве метода идентификации используется метод `ident sameuser`[9]. Создав пользователя в соответствии с текущей учётной записью можно подсоединиться к PostgreSQL и начать общаться с сервером базы данных на его родном языке SQL:

```
> psql «имя БД»

«имя БД»=> SELECT fio,number FROM fiodata, phonedata
«имя БД»=>        WHERE fiodata.id=phonedata.id;
     fio     |     number
-------------+----------------
 Иванов И.П. | 555-32-23
 Балдин Е.М. | 555-41-37
 Балдин Е.М. | (+7)5559323919
(записей: 3)
```

К вопросу о номере порта По умолчанию для создания TCP/IP соединения postmaster использует порт (port) за номером 5432. Если номер порта отличается от установленного по умолчанию, то **postmaster** должен быть запущен с ключом `-p [номер порта]`. Для выяснения наверняка достаточно выполнить:

```
> ps axw  grep postmaster  grep -v grep
 4181 ?        S      0:00 /usr/lib/postgresql/bin/postmaster
-D /home/postgres/data
```

Так же номер порта может храниться в переменной окружения `$PGPORT`.

[9] Существует более либеральный метод проверки `trust` — в этом случае пускается кто угодно и под каким угодно пользователем. То есть метод «двери настежь» — некоторым нравится.

1.5. Почему?

Люди обычно работают с текстовыми файлами. Подавляющий объём структурированной информации до сих пор доставляется до нашего сознания через текст. Без помощи компьютера, просто набивая текст руками, информации производится не так уж и много. Необходимость базы данных в начале пути накопления личной информации не является очевидной. Всё сделанное в принципе реально окинуть взглядом.

1.5.1. Почему БД?

То, что создал один человек, другой человек с большой долей вероятность освоить в состоянии, но разобраться с наследием двух и более людей становится трудновато. А если это не наследие, а информация идущая в реальном времени из многих (десятков, сотен, тысяч, миллионов) источников? Для начала всё это надо куда-то сохранить, то есть необходимо надёжное хранилище по возможности ни от чего независимое.

Ну это ещё пол беды: данные надо как-то извлечь, причём извлечь надо не абсолютно все данные, иначе человеческий мозг в них утонет, а только нужные. Компьютеры человеческие мысли пока ещё надёжно[10] не читают, поэтому для начала необходимо нужные данные как-то пометить и лучше это сделать в момент «укладки» в хранилище. То есть хранилище должно быть структурированным, причём структуру можно задавать извне до появления данных.

Когда данных немного — жить можно и так, оставляя ключевую информацию на обрывке листика, надеясь что он не затеряется. Обрывок листика слабо отличается от записи в каком-то файле. Текстовые утилиты типа **grep** существенно облегчают поиск информации, но всегда в конце концов настаёт момент, когда данных становится либо слишком много, либо они слишком часто изменяются и нужно вводить систему — Систему Управления Базой Данных или СУБД.

1.5.2. Почему PostgreSQL?

Когда я примерно семь лет назад пытался понять какую СУБД следует использовать для обеспечения эксперимента в котором я участвую до сих пор, то выбора просто не существовало. Из свободных СУБД только PostgreSQL на тот момент обладал необходимой функциональностью. На сегодня вопрос выбора немного усложнился: подросла в хорошем смысле этого слова MySQL (в последней 5ой версии, говорят, наконец-то даже триггеры появились), были открыты исходники проекта

[10] Удачные опыты по управлению курсором мыши или манипулятором уже зафиксированы, правда, до сих для этого требуются имплантанты. Без имплантантов операторы могут передавать только самые простейшие команды и вряд ли в ближайшее время ситуация кардинально изменится.

Firebird, в девичестве Interbase от фирмы Borland, да и «игрушечные» проекты типа SQLite тоже не лишены определённых преимуществ. Ну и, естественно, свет клином на открытых разработках не сошёлся — тот же Oracle предлагает свои СУБД для изучения. И всё-таки я *выбираю* PostgreSQL — решение шестилетней давности меня не разочаровало. На редкость устойчивая к внешним воздействиям программа с абсолютно предсказуемым поведением. Даже те случаи, которые мне по неопытности показались «граблями» оказались «фичами» ☺.

Одной из основных целей, которая была поставлена при разработке PostgreSQL является соответствие стандартам. PostgreSQL очень строго следовал ANSI SQL-92, SQL-99 (SQL-2 и SQL-3, соответственно), а теперь и ANSI SQL:2003. Мало кому[11] удаётся похвастаться подобным соответствием стандартам.

В дополнение к стандартам PostgreSQL поддерживает множество полезных расширений. Примером мелкого, но полезного расширения не входящего в стандарт SQL является дополнения к условию для SELECT вида LIMIT/OFFSET[12], которые позволяют получить только указанные строки из результата запроса. PostgreSQL полностью поддерживает механизм транзакций (transactions), вложенные запросы (subselects), триггеры (triggers), представления (views), функциональные индексы, ссылочную целостность по внешнему ключу (foreign key referential integrity), изощрённые типы блокировок (sophisticated locking) и многое другое.

К названию PostgreSQL обычно прибавляется слово объектная, то есть полное название звучит как объектно-реляционная база данных PostgreSQL. Пользователю предоставляются необходимые инструменты для создания новых типов данных, функций, операторов и своих методов индексирования. Подобные возможности позволяют работать с довольно нестандартными данными, например, с картографическими объектами — PostGIS (http://postgis.refractions.net/).

Размер базы данных, управляемой PostgreSQL не ограничен, так же нет ограничения и на число строк в таблице. Да вообще есть ли ограничения у этого чуда? Да, есть: ваша таблица не может быть больше чем 32 Тбайта, а число столбцов в таблице не может быть больше 250–1600 в зависимости от типа данных. Много это или мало? Зависит от задачи: я, например, как-то упёрся в ограничение по числу столбцов, но скорее по неопытности нежели по необходимости. Описанное выше верно для версии PostgreSQL 8.2.3. Возможно в будущем будут сняты и эти ограничения.

Существует родные интерфейсы для работы с PostgreSQL из языков Java (JDBC), Perl, Python, Ruby, C, C++, PHP, Lisp, Scheme и всего что может связаться через ODBC. PostgreSQL поддерживает хранимые процедуры которые можно написать на множестве языков программирования, включая Java, Perl, Python, Ruby, Tcl, C/C++ и родном для PostgreSQL PL/pgSQL.

[11] Возможно, что вообще некому. В большинстве случаев следование стандарту заканчиваться на вводном (entry) уровне SQL-92.

[12] Мне эти инструкции в своё время сильно облегчили жизнь, точнее увеличили скорость выполнения нужных мне запросов.

По результатам автоматизированного тестирования, проведённом в 2005 году (http://www.postgresql.org/about/news.363) в коде PostgreSQL было обнаружено 20 дефектов, что соответствует 1 ошибке на 39 тысяч строк кода. Для сравнения аналогичное тестирование примерно в то же время выявило в ядре Linux по одному дефекту на 10 тысяч строк кода, а в MySQL одно проблемное место приходится на 4 тысячи строк кода. Это не о чём не говорит, так сказать, мелочь, зато разработчикам и пользователям PostgreSQL приятно.

1.6. Информация о subj

Книг по PostgreSQL, выпущенных на русском языке, относительно[13] не много, но они есть и количество их будет расти. Эта область технических знаний не так популярна, как следовало бы. Очевидно, что в будущем без надёжных хранилищ данных будет непросто управляться со всё возрастающим потоком информации.

С другой стороны наличие отличной документации (в том числе и русскоязычной) позволяет достаточно безболезненно «войти в тему». Вполне можно обойтись и без специфичных для PostgreSQL возможностей, а для изучения основ SQL годится любая нормальная книга, коих довольно много. Для введения вполне сгодится «SQL» от Мартина Грабера. Собственно говоря, хватит и стандартной документации, которая идёт в дистрибутиве.

Основной сайт PostgreSQL http://www.postgresql.org. Там расположено в том числе и первичное хранилище обширной документации, в которой есть фактически вся «мудрость мира», имеющая хоть какое-то отношение к PostgreSQL — надо только уметь читать.

По адресу http://www.linuxshare.ru/postgresql/[14] представлена русскоязычная версия сайта. Там же можно найти информация о русскоязычном тематическом списке рассылки pgsql-ru-general@postgresql.org[15]. Список не сильно активный, но если хочется перемолвиться о «subj» по русски вполне сгодится.

[13] Например, относительно числа книг по PHP+MySQL.

[14] Виктору Вислобокову, который поддерживает этот ресурс, очевидно нужна помощь.

[15] Подписаться на список рассылки можно, послав письмо на адрес majordomo@postgresql.org. В теле письма должна быть указана строчка: subscribe pgsql-ru-general.

Глава 2.

Работа с базой

Это прибор, — С ним работают

Витька Корнеев о диване-трансляторе.

В прошлой части было описано как создать базу данных и запустить **postmaster**. Дело за малым: надо научиться сохранять данные и доступаться до них. Для этого следует договориться с **postmaster** — благо его «родной» язык довольно высокоуровневый.

Как и в предыдущей части всё рассматривается с точки зрения дистрибутива Debian (Sarge). При прочтении на это следует делать поправки.

2.1. SQL

В качестве языка общения с реляционными базами данных в подавляющем большинстве случаев используется язык SQL. Изначально эти три буквы были сокращением фразы Structured Query Language (язык структурированных запросов). Сейчас, когда язык стал стандартом, SQL уже не является аббревиатурой — это обычное название, которое произносится как «эс-кью-эл». Несмотря на это, даже англоязычные специалисты по прежнему часто называют SQL «сиквел». По-русски также часто говорят «эс-ку-эль».

У этого языка есть недостатки, что приводит к тому что в реальности он дополняется различными расширениями. Кстати, сам Кодд — «отец» реляционных баз данных считал SQL неудачной реализацией его теории. Но на сегодня это мощный открытый промышленный стандарт, который позволяет решать множество типовых задач по созданию, модификации и управления данными — он есть здесь и сейчас.

За время своего существования SQL претерпел несколько ревизий. В таблице 2.1 перечислены основные ревизии стандарта.

Таблица 2.1. Ревизии SQL

Год	Ревизия	Нововведения
1986	SQL-86, SQL-87	Первая версия стандарта ANSI. Принят ISO в 1987 году. Стандартизация синтаксиса.
1989	SQL-89	Стандартизован механизм ссылочной целостности.
1992	SQL-92 (SQL-2)	Множество нововведений. В отличии от предыдущих версий, где стандарт просто сертифицировал уже имеющиеся на рынке реляционных БД возможности были заложены основы для развития языка. Введены три уровня соответствия стандарту: Entry (начальный), Intermediate (промежуточный), Full (полный). Мало какая из баз данных поддерживает SQL-92 больше чем Entry.
1999	SQL:1999 (SQL-3)	Добавлены регулярные выражения, рекурсивные запросы, триггеры. Определена интеграция с объектно-ориентированным подходом. Вместо трёх уровней соответствия введён набор свойств (features).
2003	SQL:2003	Стандартизованы XML-зависимые нововведения, интервальные функции (window functions), стандартные последовательности и столбцы с автоматически генерируемыми значениями.

Степень соответствия PostgreSQL стандарту SQL:2003 рассмотрена в Приложении D (Appendix D. SQL Conformance) стандартной документации.

В стандартной же документации есть и простейший учебник, и исчерпывающий справочник по SQL. Существует море литературы в которой подробно и не очень рассказывается что же это за «зверь такой» SQL. Необходимый для «вхождения в технологию» минимум настолько прост, что основы изучаются в пределах одного дня вдумчивого чтения учебника.

Для того чтобы данные куда-то сохранить, необходимо создать «хранилище» для них — таблицу/таблицы:

```
CREATE TABLE fiodata (id int,fio text)
CREATE TABLE phonedata (id int,number text)
```

Теперь можно добавлять данные:

```
INSERT INTO fiodata VALUES (1,'Иванов И.П.')
INSERT INTO phonedata VALUES (1,'555-32-23')
```

и так далее. Созданы две обычные таблицы «без наворотов» в одной хранятся имена, а в другой телефоны. Сопоставление телефонов именам идёт через поля `id`. Зачем так? На одно имя может быть заведено несколько телефонов, а на одном телефоне может «сидеть» несколько человек.

Теперь надо данные извлечь и в этом нам поможет оператор SELECT. Собственно говоря, пользователю кроме этого оператора больше ничего знать и не надо — все выборки делаются с помощью него. Выведем все имена и соответствующие им телефоны:

```
SELECT fio, number FROM fiodata,phonedata WHERE fiodata.id=phonedata.id
```

SQL заслуживает большего чем это «микровведение», но его и так и сяк придётся изучить тем, кто реально хочет связаться с базами данных. То есть надо читать книжки. А если подходить к делу серьёзно, то кроме описания SQL следует изучить и основы реляционных баз данных, того же К. Дж. Дейта (C. J. Date) — но это уже совсем другая история.

2.2. Командная строка

Когда набирается текст, а SQL это именно текст, то лучше чтобы ничего вокруг не отвлекало. Надёжная, «толстая» и дешевая связь вещь хорошая, только вот не всегда она случается. Часто командная строка вне конкуренции.

2.2.1. psql

Вместе с пакетом **postgresql-client** поставляется утилита psql — интерактивная оболочка для «разговоров» с PostgreSQL. Она же — лучший инструмент для администрирования.

Пусть существует база данных test, в которой заведены таблицы fiodata и phonedata, описанные в предыдущем разделе. Подсоединимся к базе и что-нибудь «спросим» у неё:

```
> psql test
Добро пожаловать в psql 7.4.7 Интерактивный Терминал PostgreSQL.

Наберите:  \copyright для условий распространения
           \h для подсказки по SQL командам
           \? для подсказки по внутренним slash-командам
           \g или ";"для завершения и выполнения запроса
           \q для выхода

test=> SELECT fio, number
test->   FROM fiodata,phonedata WHERE fiodata.id=phonedata.id;
     fio     |     number
-------------+----------------
 Иванов И.П. | 555-32-23
 Балдин Е.М. | 555-41-37
```

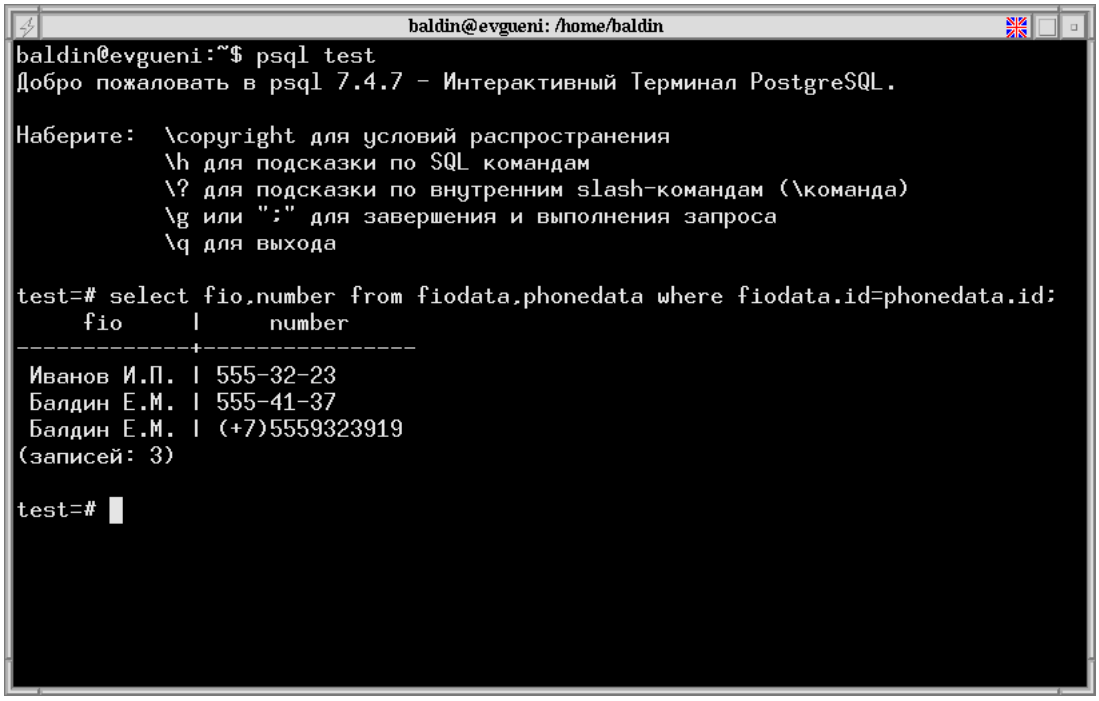

Рис. 2.1. Окно psql

```
 Балдин Е.М. | (+7)5559323919
(записей: 3)
```

Если хочется подсоединиться к серверу на другой машине, то нужно указать имя машины после ключика `-h`. Ключик `-U` позволяет сменить имя пользователя по умолчанию.

psql позволяет передавать SQL-команды серверу. Обратите внимание, что для завершения SQL-команды используется точка с запятой «;».

Как и всякая человеко-ориентированная оболочка **psql** использует библиотеку Readline. Это означает наличие стандартных горячих emacs-подобных комбинаций символов для общепринятого редактирования ввода командной строки, в том числе и завершение SQL-команд по «TAB». По клавише «TAB» завершаются не только SQL-команды, но и названия таблиц и имена колонок, если это возможно.

psql поддерживает историю команд, которая сохраняется в `.psql_history`. Это так же особенность библиотеки Readline. Полезным является интерактивный поиск по истории команд, который вызывается с помощью комбинации `C^r`.

Кроме команд SQL **psql** имеет набор собственных специальных команд. Все такие команды начинаются с обратной косой черты «\». Число спец-команд довольно обширно и полное их описание можно найти выполнив команду `man psql`. Далее будет перечислены наиболее интересные из них:

\q Закончить работу с **psql**. Выйти из оболочки.

\? Вывести справку по имеющимся спец-командам.

\h [SQL-команда] Вывести помощь по запрашиваемой SQL-команде в форме Бэкуса-Наура (Backus Naur Form). SQL-команда может состоять из нескольких слов. При исполнении \h без аргумента выводится список доступных SQL команд.

\! [shell-команда] Запустить командный интерпретатор и выполнить shell-команду.

\i «файл» Прочить текстовый «файл» и выполнить имеющиеся в нём команды. Удобно для нетривиальных операций.

Имя файла с командами можно передать при запуске **psql** через ключик `-f`. В этом случае после чтения и исполнения всех команд **psql** автоматически прекращает работу.

\o [«файл»] Сохранить результаты выполнения будущих команд в «файл». Если аргумент отсутствует, то вывод переключается на терминал. **psql** имеет набор команд, которые позволяют сформировать вывод как угодно пользователю.

Указать имя файла, в котором следует сохранить результаты, также можно передать при запуске **psql** с помощью ключика `-o`. Этот ключ удобно применять совместно с ключом `-f`.

\d [«регулярное выражение»] Вывести структуру объекта. Годится для таблицы (table), представления (view), индекса (indexes) или последовательности (sequences). Список объектов можно получить добавив первую букву названия объекта `t`, `v`, `i`, `s` к команде `\d`.

В дополнение к вышесказанному, **psql** поддерживает простейший механизм присваивания значений собственным переменным и их интерполяции в SQL-запросах:

```
test=> \set proba 'phonedata'
test=> select * from :proba;
 id |     number
----+----------------
  2 | 555-41-37
  2 | (+7)5559323919
  1 | 555-32-23
(записей: 3)
```

Следует учитывать, что интерполяция переменных не работает, если переменная используется внутри SQL-строки. В любом случае это хоть что-то.

2.2.2. gql-shell

Небольшая **psql**-like оболочка созданная одним человеком. Разработка заморожена. Естественно, **gql-shell** не владеет всеми возможностями **psql**, зато может подсоединяться и «разговаривать» не только с PostgreSQL. Для подсоединения к базе данных используется библиотека GQL (Generic C++ SQL Library). Для работы с PostgreSQL необходимо установить драйвер:

```
> sudo apt-get install gql-shell
> sudo apt-get install libgql-driver-0.5-pg
> gql-shell pg:test
Welcome to gql-shell, the interactive SQL terminal.

Type:  \copyright for distribution terms
\h for help with SQL commands
\? for help on internal slash commands
\g or terminate with semicolon to execute query
\q to quit

test=>
```

2.2.3. dbishell

dbishell — интерактивная оболочка на основе Perl::DBI. Как и в случае с **gql-shell**, поддерживает не только PostgreSQL. **dbishell** представляет из себя скрипт на perl и занимает при установке чуть больше 150 кб.

```
> sudo apt-get install dbishell
> dbishell --driver Pg /
           --dsn host=localhost\;database=test /
           --user baldin
Password:
Using DBIShell::dr::Pg engine

dbi:Pg:host=localhost;database=test:baldin>quit/
```

Для завершения любой команды используется косая черта «/».

2.3. GUI в помощь

Следует признать, что программа с графическим пользовательским интерфейсом выглядит гораздо солиднее какой-то там командной строки. Об эффективности работы в случае необходимости показать, что занят важным делом речь, естественно,

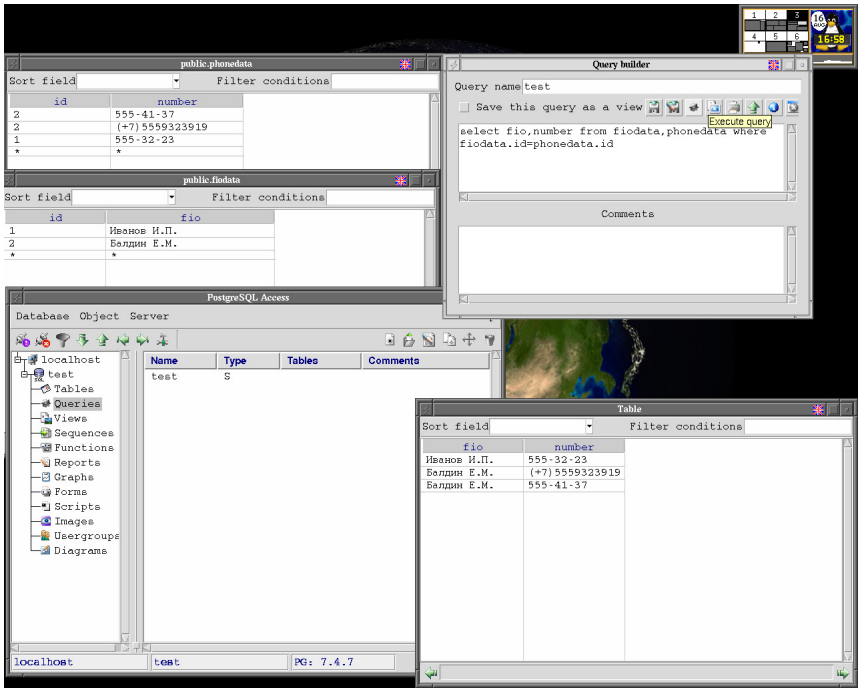

Рис. 2.2. pgaccess явно что-то умеет

не идёт. Зачем один терминал, когда можно открыть кучу красивеньких окошечек с иконками? ☺

2.3.1. PgAccess

Когда обсуждается графический пользовательский интерфейс к PostgreSQL, тут же всплывает слово PgAccess (http://www.pgaccess.org/). PgAccess был создан Константином Теодореску (Constantin Teodorescu) и имеет довольно длительную историю развития. На текущий момент разработка, похоже, заморожена. С другой стороны «нет худа без добра»: новых версий тащить не надо — достаточно поставить то, что идёт стандартно с Вашим дистрибутивом:

```
> sudo apt-get install pgaccess
> pgaccess
```

Для того чтобы подсоединиться к базе данных, необходимо воспользоваться диалогом открытия соединения: **Database** → **Open**. По умолчанию, предполагается что **postmaster** запущен на этом же компьютере (Host: localhost) и «слушает» порт номер *5432*. Если при установке PostgreSQL ничего специально не делалось, то так оно и есть. Далее требуется указать базу данных к которой надо подсоединиться, пользователя и, если необходимо, пароль.

PgAccess — это кросс-платформенный графический интерфейс к PostgreSQL, написанный на чистом Tcl/Tk, и как следствие этого, работает везде, где этот инструментарий имеется, даже на «альтернативной» платформе. Размер дистрибутива по современным меркам крошечный: при установке всё укладывается в 4 Мб.

В программе есть возможность создавать, редактировать и просматривать таблицы, запросы, представление, функции, пользователей, то есть довольно многое из того, что можно делать с помощью SQL. Плюс к этому можно создавать графические формы для ввода/просмотра данных, рисовать простые диаграммы и графики, просматривать картинки, сохранённые в базе данных. Так как программа написана на Tcl/Tk, то есть возможность писать свои скрипты используя объекты уже определённые в PgAccess.

Если хочется «сляпать» на скорую руку формочку, которую можно запустить фактически где угодно после минимального использования напильника, то PgAccess вполне может подойти для этого дела. В самом PgAccess масса ограничений и недостатков, но так как программа относительно небольшая, то её можно доделать «по месту».

Информацию о созданных формах, запросах и тому подобных объектах PgAccess сохраняет непосредственно в базе данных в таблицах, начинающихся с префикса `pga_`. Так что то, что сделано кем-то одним, будет доступно и *всем* пользователям базы.

P.S. После первого запуска необходимо настроить шрифты: **Database** → **Preferences** → **Look & Feel**. Выбор шрифтов по умолчанию не совсем адекватен. С другой стороны при желании это настраивается.

P.P.S. Наличие PgAccess на машине, с моей точки, зрение поощряет нездоровое желание у пользователей что-то «сляпать», а не сделать по человечески. Так что работать с этим предметом надо осторожно, и, если нет необходимости, то лучше убрать его от греха подальше. По мне, так psql *гораздо* удобнее и эффективнее, а самое главное пользователи в БД гораздо реже наведываются ☺.

2.3.2. pgAdmin III

Программа порадовала заявлением, что она наиболее популярная и «фичастая» платформа администрирования и разработки для PostgreSQL, а также своим отсутствием в дистрибутиве Debian (Sarge), посему установка начинается с выкачивания программы на свой диск. Благо на сайте проекта http://www.pgadmin.org/ можно найти сборки под множество дистрибутивов. Есть и специальный репозитарий для Debian — в `/etc/apt/source.list` добавляется строка:

```
deb [MIRROR URL]/pgadmin3/release/debian sarge pgadmin
```

Где вместо [MIRROR URL] подставляется одно из официальных зеркал PostgreSQL, например: ftp://ftp.ru.postgresql.org/pub/mirrors/pgsql, и производится установка программы:

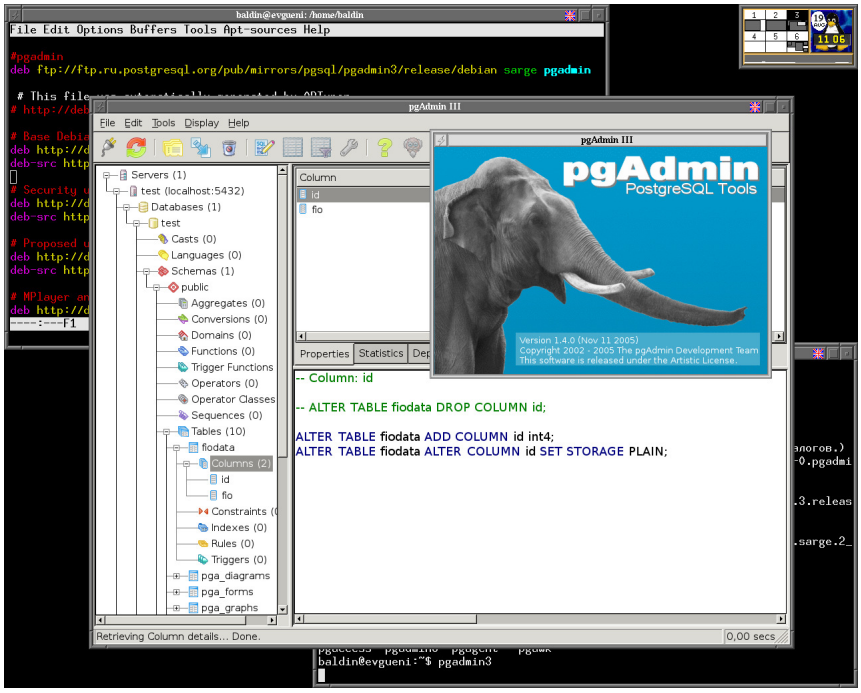

Рис. 2.3. pgAdmin III подробно объясняет что надо «сказать» чтобы создать выбранный объект.

```
> sudo apt-get update
> sudo apt-get install pgadmin3
> pgadmin3
```

При этом скачивается около 7.5 Мб. После запуска можно убедиться, что программа выглядит вполне солидно.

Новое подсоединение подключается через меню **File** → **Add server**. Требуется указать **Address** (localhost для локальной машины), сделать краткое описание соединения (**Description**), выбрать базу данных (**Maintenance DB**) и пользователя. После подсоединения доступны все объекты, которыми может управлять пользователь под которым произведено соединение.

PgAdmin III это продукт для администрирования и управления базами данных под управлением PostgreSQL и его потомков[1]. PgAdmin III содержит в себе графический интерфейс для управления данными, SQL-редактор с графическим представлением EXPLAIN, имеет инструменты для создания и редактирования таблиц, умеет управляться с системой репликации Slony-I и многое другое, что действительно упрощает

[1]В следствии того, что PostgreSQL распространяется под BSD лицензией, имеется несколько коммерческих продуктов, основанных на его коде, например, EnterpriseDB, Pervasive Postgres и SRA PowerGres

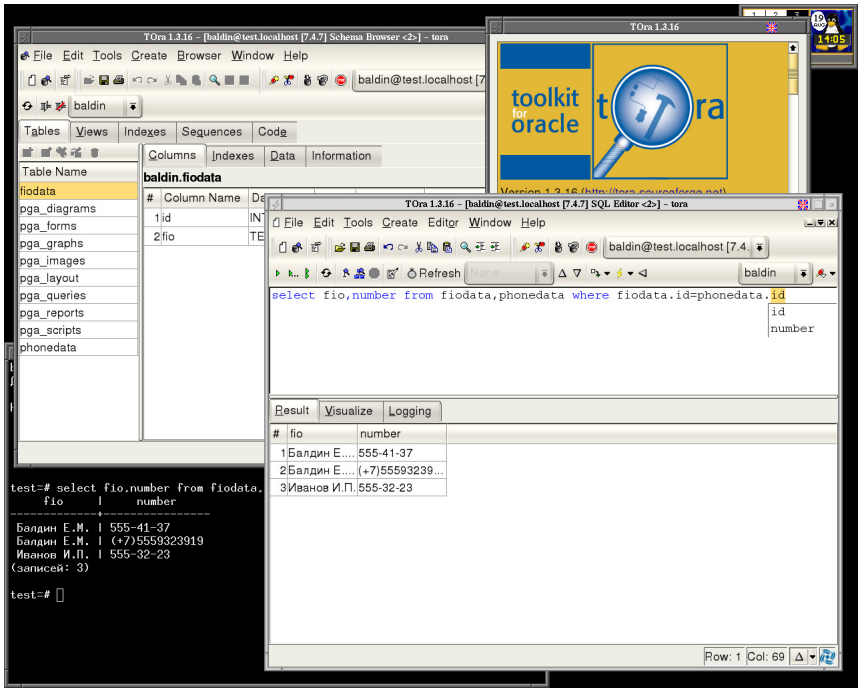

Рис. 2.4. tora знает всё об SQL и кое-что сверх того.

администрирование. И всё-таки PgAdmin III не для пользователя. Пока нет понимая в том, что происходит, не следует уповать на картинки.

Изначально, pgAdmin разрабатывался под «альтернативную» операционную систему, но на сегодня этот продукт является многоплатформенным решением, и работает не только под Linux, но и под Mac OSX, FreeBSD и даже Solaris. В качестве графической библиотеки при разработки pgAdmin III была выбрана wxWidgets (http://www.wxwidgets.org).

Русский перевод интерфейса, в принципе, существует, но на текущий момент не поддерживается. С другой стороны, это прежде всего инструмент администрирования и управления данными. Но в любом случае разработчики приветствуют новые и обновлённые переводы.

2.3.3. TOra

TOra возникла благодаря тому, что Генри Джонсон (Henrik Johnson) не смог запустить VMWare с альтернативной системой в 2000 году. В то же время ему хотелось иметь графическую утилиту для администрирования Oracle, подобную той, которой пользовались его друзья, так и не отошедшие от альтернативной операционной системы. TOra — это toolkit for Oracle. Так было, но на сегодня, в том числе и вследствие того, что TOra написана на библиотеки qt3, так же можно

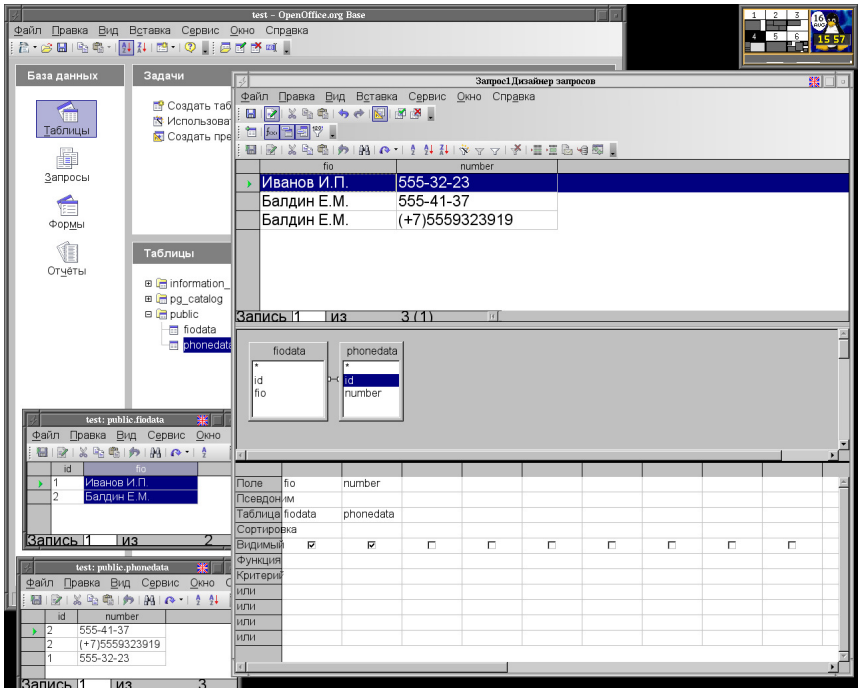

Рис. 2.5. OpenOffice + PostgreSQL

работать и с PostgreSQL. Кроме PostgreSQL дополнительно поддерживается MySQL, и всё, что работает через ODBC.

Установка и запуск TOra просты:

```
> sudo apt-get install tora
> tora
```

tora предлагает диалог создания нового соединения сразу при старте. Требуется указать **Connection provider** (PostgreSQL), Username, Host (localhost), Port (5432) и DataBase.

Вследствие своего прошлого многие возможности TOra привязаны к особенностям Orcale. В случае работы с PostgreSQL, TOra полезна прежде всего как браузер по SQL-объектам, SQL-терминал и изощрённый SQL-редактор. Как и в случае pgAdmin III TOra позволяет создавать и редактировать таблицы с помощью графических диалогов, но не владеет специфичными для PostgreSQL настройками.

TOra — это крепко «сбитый» программный продукт, который позволяет работать с разными реляционными СУБД в пределах одной программы.

2.3.4. OpenOffice и SDBC

Open Office — монстр, но ситуация на сегодня такая, что люди любят монстров, и ничего в обозримом будущем с этим не поделаешь. ☺

Для прямого доступа из OpenOffice к PostgreSQL без промежуточного уровня в виде ODBC/JDBC драйверов разрабатывается postgresql-sdbc драйвер. На сегодня в стандартной поставке OpenOffice этот пакет отсутствует.

Для установки необходимо скачать zip-архив этого драйвера с его домашней странички http://dba.openoffice.org/drivers/postgresql/index.html и положить куда-нибудь у себя на диске *не распаковывая (!)*. Далее, запустив OpenOffice, следует открыть диалог управления пакетами: **Сервис →Управление пакетами...** и с помощью кнопки «Добавить» установить этот пакет. В моём случае после установки пришлось перезапустить OpenOffice.

Для подсоединения к уже существующей базе данных PostgreSQL следует открыть диалог «Мастера базы данных»: **Создать → Базу данных → Выбор базы данных →** поставить галочку **Подключиться к существующей базе данных →** выбрать **postgresql**. Далее, при настройке соединения в следует ввести строчку вида:

```
dbname=«имя БД» host=«адрес сервера»
```

подставив вместо «имя БД» и «адрес сервера» базу данных, которая предварительно уже была создана и адрес сервера на котором «крутится» **postmaster**, например, **dbname=test host=localhost**. Далее, во вкладке «Аутентификация пользователя» необходимо ввести имя пользователя и можно протестировать соединение. Если тест прошёл нормально, то можно продолжить и выполнить соединение.

Во время окончания действия мастера предлагается сохранить всё что проделано в odb-файле (формат «База данных OpenDocument»). Затем это соединение можно будет выполнить простым открытием файла. Туда же сохраняется информация обо всех созданных формах, запросах и отчётах. Как конкретно создаются формы и отчёты — это совсем другая история и относится она не PostgreSQL, а к OpenOffice.

P.S. При выборе таблиц видно, что они в PostgreSQL разбиты на группы. Пользовательские таблицы по умолчанию находятся в группе **public**. В группах **pg_catalog** и **informaion_schema** представлена системная информация и статистика.

2.4. Что выбрать?

Естественно, рассмотрены далеко не все возможные программы общего назначения для работы с PostgreSQL, но даже из того что рассмотрено нельзя выбрать что-то одно. Каждая программа имеет свои особенности и преимущества. **psql** позволяет легко работать удалённо, OpenOffice удовлетворяет нашу любовь к монстрам, PgAdmin III содержит множество подсказок по делу, PgAccess удивляет своей интеграцией с TCL/Tk, а TOra — «красивая» ☺.

Я всегда выбирал **psql**, но это скорее всего связано со специфичностью решаемых мной задач. Я вполне могу представить себе ситуацию, когда наличие, например, TOra значительно облегчит жизнь. Ну и не следует забывать про данные, которые

кто-то должен вводить. Если лень писать специальную программку, которую по хорошему лучше таки написать, то OpenOffice поможет, особенно если вводить не Вам. ☺

Глава 3.

Возможности PostgreSQL

Познание бесконечности требует
бесконечного времени.

отдел *Абсолютного Знания*

В этой статье предпринята попытка сделать краткий обзор имеющихся возможностей PostgreSQL. Не надо иллюзий — «объять необъятное невозможно», поэтому многое интересное осталось за кадром, но всё же «попытка — не пытка».

3.1. Чуть-чуть про основы

Когда говорят про базы данных, то сразу вспоминают принцип ACID: аттомарность (Atomicity), консистентность (Consistency), локализация пользовательских процессов (Isolation) и устойчивость к ошибкам (Durability).

Для обеспечения совместной работы множества пользователей (concurrency), в целях следования заветам ACID PostgreSQL, использует систему управления версиями или MVCC (Multi-Version Concurrency Control). Каждому пользователю при подсоединении MVCC «подсовывает» свою версию или мгновенный снимок (snapshot) базы данных. В этом случае, изменения, производимые пользователем, невидимы другими пользователями до тех пор, пока текущая транзакция[1] (transaction) не подтверждается (commit). Кроме проблем, связанных с ACID, многоверсионность позволяет уменьшить или даже исключить во многих случаях необходимость запретов на изменение данных (locks) при чтении.

Надёжность (reliability) для сохранения данных является одним из основных показателей качества СУБД. Сохранение изменённых данных очень нетривиальная

[1]Транзакция представляет из себя последовательность операций, которая обязана либо выполняться полностью, либо отменяться совсем, как будто это единое целое. При этом обязана соблюдаться целостность данных (consistency) не зависимо от других параллельно идущих транзакций (isolation).

процедура. Всё дело в том, что диски очень *мееедленные*, поэтому прежде чем попасть на диск данные проходят через промежуточные буферы (cache), начиная от собственного кэша базы данных (shared buffers), заканчивая кэшом на самом диске. Никто не сможет гарантировать, что всё, что положено, окажется в безопасном постоянном хранилище в случае возникновения каких-либо проблем. Для максимального уменьшения вероятности потери данных PostgreSQL использует журнал транзакций или Write Ahead Log (WAL). Прежде чем записать данные о проведённой транзакции на диск, информация об изменениях пишется в WAL. Если что-то случилось, то данные можно восстановить по журналу. Если данные в журнал не попали, то соответственно исчезнет вся транзакция — жалко конечно, зато консистентность не нарушается. Следствием использования WAL является отсутствие необходимости «скидывать» данные на диск с помощью `fsync`, так как достаточно убедиться, что записан WAL. Это значительно увеличивает производительность в многопользовательской среде с множеством мелких запросов на изменение данных, так как записать один последовательный файл WAL гораздо проще, чем изменять множество таблиц по всем диску. В качестве бонуса журнал транзакций позволяет организовать *непрерывное* резервное копирование данных (on-line backup) — мечта администратора и возможность «отката» базы данных на любой момент в прошлом (point-in-time recovery) — своеобразная машина времени.

3.2. Типы данных

PostgreSQL поддерживает довольно много стандартных типов данных, как и положено базе данных. Более того, пользователь может определить свои собственный типы данных, если он не найдёт необходимых примитивов среди стандарта.

3.2.1. Числовые типы

Обычные числовые (numeric) типы представлены целыми числами два (smallint), четыре (integer) или восемь байт длиной (bigint), числа с плавающей точкой в четыре (real) и восемь байт (double precision) длиной. Кроме обычных чисел, в случае плавающей точки поддерживаются значения `Infinity`, `-Infinity` и `NaN` — бесконечность (∞), минус бесконечность ($-\infty$) и «не число» (not-a-number), соответственно.

PostgreSQL поддерживает числа с произвольной точностью `numeric(precision, scale)`, где `precision` — число всех знаков в определяемой величине, а `scale` — число знаков в дробной части. PostgreSQL позволяет выполняя действия без накопления ошибки с подобными величинами с точность вплоть до 1000 знаков. Не следует злоупотреблять этим типом данных, так как операции над подобными числами занимают очень много времени.

Битовые поля представлены типами `bit(size)` — битовая строка фиксированной длины `size` и `bit varying(size)` — битовая строка переменной длины с ограничением по размеру `size`.

К числовым типам PostgreSQL относятся и «псевдотипы» serial и bigserial. Эти типы соответствуют типам integer и bigint за исключением того, что при записи новых данных в таблицу с колонкой этого типа, значение по умолчанию в ней увеличивается на единицу — автоматически создаваемая упорядоченная последовательность.

3.2.2. Символьные типы

В стандарте SQL символьный тип определяется как строка определённой длины `character(size)`, где `size` — длина строки. В дополнение к стандарту, PostgreSQL поддерживает строки переменной длины с ограничением `varchar(size)` и без ограничения — `text`.

3.2.3. Бинарные типы

Бинарную строку можно сохранить используя тип bytea. SQL предполагает, что вся информация передаётся как текст, поэтому при передачи данных следует экранировать некоторые из символов.

В PostgreSQL есть специальный тип данных Large Objects. По сути дела, это просто возможность сохранять любые файлы размером вплоть до 2 Гб прямо в базе данных. Операции с подобными объектами выходит за рамки SQL. Для доступа к Large Objects есть специальный программный интерфейс по образу и подобию обычного чтения/записи файла.

3.2.4. Типы даты/времени

Временем в PostgreSQL заведует тип `timestamp` или `timestamp with time zone` — может сохранить дату и время начиная с 4713 г. до н. э. вплоть до 5874897 г. с точность в одну микросекунду (μc), занимает восемь байт. Второй упомянутый тип включает часовой пояс и позволяет автоматически учитывать переход на летнее/зимнее время. С таким диапазоном и точность проблема типа распиаренной «проблемы 2000 года» случится не скоро.

Разницу между двумя датами хранится в столбце типа `interval` — двенадцать байт, поэтому можно хранить информацию о событиях связанных с рождением и смертью вселенной.

Так же есть отдельный тип для календарного времени (`date`) и просто для времени (`time` или `time with timezone`).

PostgreSQL поддерживает множество способов ввода даты и времени. С моей точки зрения СУБД в некоторых случаях проявляется излишний интеллект, поэтому

в качестве способа ввода следует выбрать стандартный ISO, который выглядит примерно так:

```
db=> -- узнаём текущее время с точностью до секунды
db=>select date_trunc('seconds',timestamp with time zone 'now');
       date_trunc
------------------------
 2006-08-26 21:08:14+07
```

В этом случае, никогда не ошибёшься в порядке следования месяца и дня не зависимо от того, какая локаль используется.

Для типа timestamp определены дополнительные константы:

epoch — начало эпохи с точки зрения юниксового времени (четырёхбайтовый `time_t`) `1970-01-01 00:00:00+00`

infinity — позже, чем любое возможное из времён,

−infinity — раньше, чем любое возможное из времён,

now — здесь и сейчас,

today — сегодняшняя полночь, аналогично есть `yesterday` — вчерашняя полночь и, `tomorrow` — завтрашняя полночь.

3.2.5. Логические типы

Логические типы представлены типом boolean. Логично, что он содержит значения либо TRUE ('t', 'true', 'y','yes', '1') — «истина», либо FALSE ('f', 'false', 'n', 'no', '0') — «ложь» . Всё просто, за исключением одного «но» — есть ещё одна возможность: «значение не определено» (NULL). Собственно говоря, это не особенность типа boolean. С тем что значение может быть не определено при использовании SQL необходимо считаться всегда и везде. Вот такая вот логика — вовсе не двоичная.

3.2.6. Остальные стандартные типы

К оставшимся стандартным типам относятся различные геометрические типы данных: типы точки (point), линии (line), отрезка (lseg), прямоугольник (box), пути (path), замкнутого пути (polygon) и окружности (circle). Для системных администраторов будут интересны стандартные типы сетевых IPv4 и IPv6 адресов (cidr или inet) и тип MAC-адреса (macaddr).

Более сложные типы реализуются как дополнения. Яркими примерами служат поддержка географических объектов GIS (http://postgis.refractions.net/) и иерархический тип данных ltree (`contrib/ltree`).

3.2.7. Определение пользовательских типов

Прежде всего следует упомянуть, что PostgreSQL поддерживает массивы. Можно создать массив определённого размера или безразмерный на основе любого стандартного типа или типа определённого пользователем. Поддерживаются многомерные массивы и операции над ними, как то «срезы».

```
db=> --- Создаём массив для игры Тик-Так
db=> create table tictactoe (squares integer[3][3]);
db=> --- |x00| x = 1, 0 = −1
db=> --- |0xx| Вставляем информацию о варианте игры
db=> --- |  x| Крестики начинают и выигрывают
db=> insert into tictactoe
db->     values ('{{1,-1,-1},{-1,1,1},{0,0,1}}');
db=> --- Распечатываем сохранённую позицию
db=> select * from tictactoe ;
           squares
------------------------------
 {{1,-1,-1},{-1,1,1},{0,0,1}}
db=> -- Распечатываем значение первого столбца
db=> select squares[1:3][1:1] from tictactoe ;
    squares
----------------
 {{1},{-1},{0}}
```

Композитный тип (composite type) представляет из себя аналог структуры:

```
db=> CREATE TYPE complex AS (Re real,Im real);
```

В отличии от стандартных встроенных типов использование композитного типа пока имеет некоторые ограничения. Например, нельзя создавать массивы.

PostgreSQL позволяет выйти за рамки стандартного SQL для целей создания своих типов данных и операций над ними подробнее об этом можно узнать изучив документацию по команде `CREATE TYPE`.

3.3. Функции

Все стандартные типы имеют свои функции, ведь если есть тип, то с ним нужно работать. Число стандартных функций велико[2] и разнообразно. Одних операторов поиска с использованием регулярных выражений целых три штуки: собственное расширение PostgreSQL (`LIKE` и `ILIKE`), оператор соответствующий SQL стандару (`SIMILAR TO`) и POSIX-совместимый оператор (`~` и `~*`). Всё, что только можно было

[2]Больше 1500. Полный список можно вывести, набрав в `psql` команду `\df`

быстро придумать, уже реализовано. А более сложные случаи, например, модуль для полнотекстового поиска tsearch2 (`contrib/tsearch2`) в процессе совершенствования. Придумать что-то выходящее за рамки стандарта тяжело. В этом случае, всегда есть возможность создать свои функции. При желании, ссылаясь на уже имеющуюся функцию, с помощью команды `CREATE OPERATOR` можно определить оператор для своих типов.

3.3.1. Хранимые процедуры

Для создания новых функций используется оператор `CREATE FUNCTION` — вполне предсказуемо. Создаваемые таким образом функции исполняются и хранятся на сервере, отсюда и название — «хранимые процедуры»:

```
db=> -- Создаём и заполняем таблицу
db=> create table AplusB (A integer, B integer);
db=> insert INTO AplusB VALUES (1,1);
db=> insert INTO AplusB VALUES (2,2);
db=> insert INTO AplusB VALUES (3,3);
db=> --  Создаём новую функцию
db=> CREATE FUNCTION plus(integer, integer) RETURNS integer
db->     LANGUAGE SQL as 'SELECT␣$1␣+␣$2;';
CREATE FUNCTION
db=> select A,B,plus(A,B) from AplusB;
  a  |  b  | plus
-----+-----+------
   1 |   1 |    2
   2 |   2 |    4
   3 |   3 |    6
(записей: 3)
```

PostgreSQL поддерживает перегрузку функций. Объектно-ориентированность имеет свои плюсы. Кроме SQL для создания новых функций можно использовать процедурные языки программирования. Для начала работы с процедурным языком его необходимо инициализировать. По умолчанию из соображения безопасности интерфейсы к другим языкам кроме SQL и C недоступны. Для инициализации используется команда `createlang`. Запустить её может только администратор базы данных — тот, кто имеет право создавать базы:

```
# Инициализируем язык PL/pgSQL для базы данных db
> createlang plpgsql db
# Делаем то же самое, но для языка PL/Perl
> createlang plperl db
```

Теперь можно создавать функции с использованием всех прелестей процедурного программирования, вместе с циклами, кои по понятной причине в SQL отсутствуют. Ниже продублирована простейшая функция, которая была описана выше, но теперь уже на PL/pgSQL и на PL/Perl:

```
db=> --  Создаём новую функцию с использование PL/pgSQL
db=>CREATE FUNCTION pgsql_plus(integer,integer) RETURNS integer
db->    LANGUAGE PLPGSQL as 'BEGIN␣return␣$1+$2;␣END;';
CREATE FUNCTION
db=> --  Создаём новую функцию с использование PL/Perl
db=>CREATE FUNCTION perl_plus(integer, integer) RETURNS integer
db->    LANGUAGE PLPERL AS 'return␣$_[0]+$_[1]';
CREATE FUNCTION
db=> --  Проверяем, что всё работает
db=>SELECT pgsql_plus(A,B) FROM AplusB;
db=>SELECT plus(A,B),pgsql_plus(A,B),perl_plus(A,B) from AplusB;
 plus | pgsql_plus | perl_plus
------+------------+-----------
    2 |          2 |         2
    4 |          4 |         4
    6 |          6 |         6
(записей: 3)
```

В стандартной документации подробно описаны идущие вместе с дистрибутивом языки PL/pgSQL, PL/Tcl, PL/Perl, PL/Python и, естественно, C/C++ с SQL. Кроме перечисленных есть поддержка

PL/PHP http://plphp.commandprompt.com/,

PL/java http://gborg.postgresql.org/project/pljava/projdisplay.php,

PL/R http://www.joeconway.com/plr/,

PL/Ruby http://raa.ruby-lang.org/project/pl-ruby,

PL/sh http://plsh.projects.postgresql.org/.

Так же есть возможность подключения своего любимого языка.

3.3.2. Триггеры

Обычно, для решения несложных задач можно удовлетвориться сценарием: «что сказано — то и сделано», но в более сложных случаев от СУБД хотелось бы получать

более сложные реакции на «раздражение». Для управления реакцией СУБД на изменение данных используются триггеры. Для создания триггера используется команда CREATE TRIGGER. Полное описание команды в форме Бэкуса-Наура приведено ниже:

```
CREATE TRIGGER «имя триггера»
    { BEFORE | AFTER } { «событие» [ OR ... ] }
    ON «имя таблицы» [ FOR [ EACH ] { ROW | STATEMENT } ]
    EXECUTE PROCEDURE «исполняемая функция - реакция»
```

Реакция на «событие», (вставкой INSERT, изменение UPDATE или удаление DELETE) может производится по выбору до (BEFORE) или после (AFTER) изменения данных. Выполнение процедуры может производиться для каждой записи (ROW) или для каждого запроса (STATEMENT). Для показательного примера создания триггера возьмём следующую выдуманную задачу: при изменении данных в описанной уже таблице AplusB сумма A и B должна автоматически обновляться в таблице ABresult. Следующее решение *чрезвычайно* не оптимально, зато работает:

```
db=> -- Создаём «результирующую» таблицу
db=> create table ABresult (result integer);
db=> -- Создаём функцию, очищающую ABresult и
db=> -- заполняющую всё суммой A и B из AplusB.
db=> create function ABsumm() returns trigger as
db-> 'BEGIN
db'>          delete from ABresult;
db'>          insert into ABresult values (AplusB.A+AplusB.B);
db'>          return NULL;
db'>  END;'
db-> language 'plpgsql';
db=> -- Создаём триггер
db=> CREATE TRIGGER makeABresult
db=>          AFTER INSERT or UPDATE or DELETE on AplusB
db=>          FOR EACH STATEMENT execute procedure ABsumm();
CREATE TRIGGER
db=> -- Добавляем данных в таблицу AplusB
db=> insert into AplusB VALUES (100,200);
db=> -- Проверяем, что триггер сработал
db=> select * from AplusB,ABresult where A+B=result;
  a  |  b  | result
-----+-----+--------
   1 |   1 |      2
   2 |   2 |      4
   3 |   3 |      6
 100 | 200 |    300
(записей: 4)
```

3.3.3. Rules

Кроме триггеров PostgreSQL обладает ещё одним способом управления реакции СУБД на запросы — это Rules или «правила». Для создания «правил» используется команда `CREATE RULE`. Основным отличием «правила» от триггера в том, что триггер — это реакция системы на изменение данных, а «правило» позволяет изменять сам запрос, в том числе и запрос на получение данных (SELECT). В частности одно из довольно удобных расширений PostgreSQL — представление или виртуальная таблица (view), реализовано с помощью «правил».

3.4. Индексы

Традиционно для ускорения поиска информацию индексируют. Если данных не сильно много, то можно прожить и так. Серьёзные же задачи требуют серьёзных объёмов, поэтому без индексов никак.

Создание индексов — это ответственность создателя БД. Создание индекса, как можно догадаться, производится с помощью команды `CREATE INDEX`:

```
CREATE [UNIQUE] INDEX «имя индекса» ON table [USING «алгоритм»]
    ( { «имя столбца» | ( «выражение» ) } [, ...] )
    [ WHERE «условие» ]
```

Индекс при желании может быть уникальным (UNIQUE). В этом случае, при создании индекса и при добавлении данных, накладывается дополнительное требование на уникальность параметра, по которому создаётся индекс.

При создании индекса можно выбрать алгоритм, по которому создаётся индекс. По умолчанию выбирается B-tree, но можно ещё указать hash, R-tree или GiST. Алгоритм GiST (http://www.sai.msu.su/~megera/postgres/gist/) был создан на пару Олегом Бартуновым и Фёдором Сигаевым. GiST является не просто ещё одним алгоритмом — это целый конструктор, позволяющим создавать индексы для принципиально новых типов данных. В версии 8.2 PostgreSQL опять же благодаря Олегу и Фёдору был добавлен ещё один метод GIN, а в 8.3 ожидается добавление bitmap-индекса. По алгоритмам создания индексов PostgreSQL одна из самых продвинутых СУБД.

Индекс можно создавать по какому-то из столбцов — самый простой метод. Так же при указании нескольких колонок создаются многоколоночные индексы. Особо следует отметить возможность создания функциональных индексов — в качестве индексы указывается функция от данных таблицы. С помощью функциональных индексов можно реализовать ещё один алгоритм индексации: Reverse index (обращает поле переменной — первый символ считается последним).

Условие (WHERE) при создании индекса позволяет создавать частичные индексы (partial indices). Это полезно в случае если в столбце, по которому создаётся индекс, большинство значений одинаково и поиск надо производить по редким значениям.

Для того чтобы индекс работал как надо необходимо следить, чтобы по базе данных регулярно запускалась процедура ANALYZE, которая собирает статистику о распределении значений в индексах. Собранная статистика в свою очередь позволяет планировщику верно принимать решение о порядке выполнения запроса. Для оптимизации поиска информации временами может оказаться полезна собственная команда PostgreSQL CLUSTER. С помощью этой команды можно упорядочить записи в таблице согласно указанному индексу.

3.5. Целостность данных

Сохранить, записать, а затем быстро достать данные вещь полезная, но как отследить, что данные записаны правильно без ошибок? Для этого необходимо постоянно следить за целостностью данных в условиях многопользовательской системы.

3.5.1. Транзакции

Транзакция — это единый блок операций, который нельзя разорвать. Либо совершается весь блок, либо всё отменяется. PostgreSQL в условиях параллельного доступа распространяет информацию об операциях только по завершению транзакции. Транзакция начинается с оператора BEGIN и заканчивается оператором COMMIT (подтверждение транзакции) или ROLLBACK (отмена транзакции). Возможен режим, когда каждый запрос сам себе транзакция, например, такой режим по умолчанию используется в psql. Для отмены этого режима достаточно набрать BEGIN;. Неудобством при использовании транзакций является то, что в случае ошибки какого-то из запросов приходится отменять всю транзакцию. Для устранения этого недостатка в 8ой версии PostgreSQL были добавлены точки сохранения (savepoints).

```
db=> -- Начинаем транзакцию
db=> BEGIN;
db=> -- Здесь идёт блок операторов, который удачно завершается

db=> -- Ставим метку
db=> SAVEPOINT savepoint_one;
db=> -- Здесь идёт блок операторов, в котором произошла ошибка

db=> -- Откатываемся до установленной метки,
db=> -- а не отменяем всю транзакцию
db=> ROLLBACK TO savepoint_one;
db=> -- Повторяем последний блок

db=> -- Завершаем транзакцию
```

```
db=> COMMIT;
db=> -- Всё, теперь изменения доступны всем
```

3.5.2. Ограничения

Целостность данных обеспечивается не только многоверсионность PostgreSQL, но и «архитектором» таблиц базы данных. При создании таблицы (CREATE TABLE) или позже можно всегда создать ограничение (CONSTRAINT) на диапазон записываемых в таблицу данных. Это может могут простые арифметические условные выражения, требования уникальности (UNIQUE или PRIMARY KEY), так и более сложные ограничения в виде внешних ключей (FOREIGN KEY).

Если какой-то столбец A является внешним ключом (FOREIGN KEY) по отношению к столбцу B (REFERENCES), то это означает, что только данные представленные в столбце B могут появиться в качестве значений столбца A. В случае внешних ключей PostgreSQL осуществляет автоматический контроль ссылочной целостности[3]. Это довольно интересный механизм, который, например, позволяет моделировать иерархические структуры.

3.5.3. Блокировки

Так как пользователь в условиях параллельного доступа к базе данных работает со своим мгновенным снимком (следствие MVCC), то в принципе можно придумать ситуацию, когда полученные данные устаревают, так как во время получения, они были изменены. Если это важно, то PostgreSQL предоставляет полный ассортимент блокировок. С помощью команды `LOCK` можно заблокировать таблицу, а инструкция `SELECT FOR UPDATE` позволяет заблокировать отдельные записи. Следует учитывать, что использование блокировок увеличивает шанс взаимной блокировки (deadlock). PostgreSQL умеет определять случаи возникновения взаимной блокировки и разрешать их путём прекращения одной из транзакций, но на это уходит время.

Послесловие

Хотелось бы сказать, что «нельзя объять необъятное». Единственная проблема в том, что конкретно это рассматриваемое «необъятное» уже «объято». За всеми подробностями следует обратиться к стандартной документации, а в качестве бонуса рекомендую хорошую обзорную статью от Олега Бартунов «Что такое PostgreSQL?»: http://www.sai.msu.su/~megera/postgres/talks/what_is_postgresql.html

[3]Ссылочная целостность — гарантированное отсутствие внешних ключей, ссылающихся на несуществующие записи в этой или других таблицах.

Глава 4.

Интерфейсы

> В институте им очень дорожили, так как попутно он использовался для некоторых уникальных экспериментов и как переводчик при общении со Змеем Горынычем.
>
> АБС о Кощее Бессмертном

Понятно, что в принципе любую базу данных можно заполнить в ручную, правда некоторые придётся заполнять очень долго. СУБД — это просто хранилище, а для заполнения и доступа к хранилищу необходима инфраструктура, и эту инфраструктуру надо создавать. Вот такая она — жизнь.

Родной библиотекой для доступа к PostgreSQL является библиотека libpq. Написана эта библиотека на чистом C, что и не удивительно, так как это основной язык *родной* системы. Все остальные языки важны, но безусловно вторичны. В любом случае мы выбираем их вовсе не поэтому.

4.1. libpq

Чтобы общаться с базой данных много функций не надо: одна функция для открытия соединения, одна для посылки запроса, одна для получения ответа и одна для закрытия соединения. В реальности всё немного сложнее, но суть остаётся.

К вопросу о переносимости Библиотека libpq написана на чистом C, поэтому практически везде, где есть `gcc`, можно организовать связь с PostgreSQL. Мне как-то пришлось это делать для VAX/VMS — всё решилось методом тыка, даже думать почти не потребовалось. Все данные — текст, поэтому вопрос бинарной совместимости платформ попросту отсутствует.

С чего начать Для того чтобы воспользоваться вызовами libpq, необходимо для начала установить её.

В Debian (Etch) для этого надо установить пакет `postgresql-dev`:

```
> sudo apt-get install postgresql-dev
```

Для доступа к функциям libpq необходимо включить в исходник include-файл:

```
#include "libpq-fe.h"
```

Скрипт `pg_config` (`man pg_config`) позволяет получить информацию куда помещаются include-файлы, библиотеки и тому подобное:

```
> #сборка программы
> gcc  -o «бинарник» «исходник».c -I‘pg_config --includedir‘ \
 -lpq  ‘pg_config --libs‘
```

4.1.1. Открытие и закрытие соединения

Даже открывать соединение с PostgreSQL можно двумя способами:

```
//открыть соединение
PGconn *PQconnectdb(const char *conninfo);
//то же, но не блокируя программу
PGconn *PQconnectStart(const char *conninfo);
//проверка статуса соединения (после PQconnectStart)
PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

`PQconnectdb` — обычная функция, где на вход подаём текстовую строку `conninfo` с параметрами для соединения с сервером, а на выходе получаем структуру типа `PGconn` с информацией о сделанном соединении и на сколько операция по соединению прошла удачно. В дальнейшем при передача данных эта переменная будет использоваться в качестве параметра.

Передача информации о серевере в качестве строки (`conninfo`) позволяет в случае появления дополнительных параметров не менять внешний интерфейс вызова и легко добавлять дополнительные опции. Пример открытия соединения:

```
const char *conninfo= "dbname␣=␣test␣host=localhost";
PGconn *conn=PQconnectdb(conninfo);
if (PQstatus(conn) != CONNECTION_OK) {
  fprintf(stderr, "Не удалось соединиться с базой данных: %s",
                PQerrorMessage(conn));
  /*завершаем работу*/ ...}
```

Параметры передаются в форме «ключевое слово» = «значение». Пары разделяются обычным пробелом. Пробелы вокруг знака равенства можно опустить. Если

необходимо передать значение с пробелами, то его необходимо заключить в одинарные кавычки '«составное»␣«значение»'. Для передачи одинарной кавычки её необходимо экранировать с помощью обратной косой черты \'. При отсутствии какого-либо параметра в строке conninfo его значение берётся из соответствующей переменной окружения, если такая определена. Если нет, то при открытии соединения используется значение по умолчанию.

Функции открытия соединения распознают следующие параметры и переменные окружения (кое-какие особенности опущены):

host TCP/IP имя узла на котором находится сервер PostgreSQL. Соответствует переменной окружения PGHOST. Значение по умолчанию localhost.

hostaddr Числовой адрес узла на котором находится PostgreSQL (альтернатива host). Соответствует переменной окружения PGHOSTADDR. Значение по умолчанию эквивалентно localhost.

port Номер порта, который «слушает» POSTMASTER. Соответствует переменной окружения PGPORT. Значение по умолчанию обычно 5432.

dbname Имя базы данных. Соответствует переменной окружения PGDATABASE. Значение по умолчанию совпадает с системной учётной записью пользователя.

user Имя пользователя базы данных. Соответствует переменной окружения PGUSER. Значение по умолчанию совпадает с системной учётной записью пользователя.

password Поле пароля, если для аутентификации требуется пароль. Соответствует переменной окружения PGPASSWORD. Если аутентификация требуется, а поле неопределенно, то для доступа используется информация в файле ~/.pgpass. Переменная окружения PGPASSFILE может указать другой файл для проведения аутентификации.

connect_timeout Устанавливает максимальное время ожидания соединения в секундах. С сервером и сетью всякое может случиться. Соответствует переменной окружения PGCONNECT_TIMEOUT. Значение по умолчанию равно 0, что означает что время ожидания равно бесконечности. Не рекомендуется устанавливать значение на ожидание меньше 2 секунд.

options Опции, посылаемые непосредственно серверу, коли такое потребуется. Соответствует переменной окружения PGOPTIONS.

sslmode Определяет порядок действий при SSL-соединении. Принимает четыре возможных значения:

disable — без шифрации,

allow — сначала попробовать соединиться без шифрации, а в случае неудачи постараться установить защищённое соединение,

prefer — сначала попробовать установить защищённое соединения, а в случае неудачи повторить соединение без шифрации,

require — выполнять только защищённое соединение.

Соответствует переменной окружения `PGSSLMODE`. Значение по умолчанию `prefer`.

krbsrvname Имя Kerberos-сервиса. используется для аутентификации с помощью Kerberos-5[1]. Это совершенно отдельная тема для беседы. Соответствует переменной окружения `PGKRBSRVNAME`.

PGDATESTYLE Переменная окружения, позволяющая установить представление времени и даты по умолчанию. Полностью соответствует стандартной SQL-команде `SET datestyle TO...`

PGTZ Переменная окружения, позволяющая установить текущий часовой пояс. Соответствует SQL-команде `SET timezone TO...`

PGCLIENTENCODING Переменная окружения, позволяющая установить «кодировку» клиента. Соответствует SQL-команде `SET client_encoding TO...`

Существует целый класс функций, которые позволяют получить информацию о соединении. Для подробностей следует обратиться к документации. Для начала полезно знать о двух из них:

```
ConnStatusType PQstatus(const PGconn *conn);
char *PQerrorMessage(const PGconn *conn);
```

`PQstatus` возвращает информацию о том, как прошло соединение. Интересны состояния `CONNECTION_OK` — всё хорошо и `CONNECTION_BAD` — ничего не вышло. Функция `PQerrorMessage` позволяет получить текстовую строку с описанием последней возникшей проблемы.

Для того чтобы разорвать соединение используется функция:

```
void PQfinish(PGconn *conn);
```

Внимание: Всегда следует закрывать соединения, когда в них отпадает нужда. Число соединений которые поддерживает POSTMASTER ограничено — очень легко парализовать работу базы данных только открывая новые соединения.

[1]Kerberos — промышленный стандарт для аутентификации и взаимодействия в условиях незащищённого окружения. Алгоритмы Kerberos основаны на шифровании с использованием симметричного криптографического ключа и требует наличие доверенного агента.

4.1.2. SQL запросы

Что ж, до сервера уже «дозвонились», теперь пора с ним поговорить.

Посылка запросов Простейший способ выполнить SQL-запрос, это воспользоваться функцией `PQexec`:

```
PGresult *PQexec(PGconn *conn, const char *command);
```

В качестве параметров передаётся структура соединения `conn` и строка с SQL-командой `command`. Возвращается указатель на структуру типа `PGresult`, где сохраняется информация полученная от СУБД в ответ на запрос. При желании можно в одном запросе отсылать сразу несколько SQL-команд, разделённых точкой с запятой «;», но в этом случае информация сохранённая в структуре `PGresult` будет относиться только к последнему запросу.

По умолчанию каждый `PQexec` считается за отдельную транзакцию, если явно не начать транзакцию с помощью команды `BEGIN`, которая будет продолжаться либо до `COMMIT`, либо до `ROLLBACK`.

Есть более сложный вызов `PQexecParams`, который позволяет передавать вызов и параметры к этому вызову раздельно. Таким образом исчезает необходимость самостоятельно формировать строку SQL-команды и заботиться об экранировании данных, что важно в случае сохранения бинарных последовательностей. В качестве платы из соображения безопасности `PQexecParams` может послать не более одной команды за раз.

В некоторых случаях для увеличения скорости выполнения часто встречающихся запросов полезно обратить внимание на парочку `PQprepare` и `PQexecPrepared`. Эти команды эквивалентны своим SQL-аналогам `PREPARE` и `EXECUTE`. Идея оптимизации состоит в том, что прежде чем выполнить запрос, PostgreSQL сначала анализирует его, затем планирует порядок действий и только потом, собственно, выполняет запрос. Первые два этапа для похожих запросов с разными условиями отбора можно выполнить заранее с помощью команды `PREPARE`. Затем, с помощью команды `EXECUTE`, можно выполнять подобные уже подготовленные (prepared) запросы.

Все упомянутые выше команды работают с сервером БД синхронным образом, то есть посылают запрос и ждут ответа. Клиентское приложение на это время «засыпает» (suspended). В libpq предусмотрен целый класс функций предназначенный для асинхронных операций, не блокирующих клиентское приложение. Их применение усложняет код и логику программы, хотя всё в пределах допустимого. С моей точки зрения лучше организовать всё так, чтобы время использованное на ожидание результатов запроса фатально не влияло на внешние процессы и обеспечить бесперебойную работу сети и сервера базы данных.

Информация о состоянии запроса После выполнения запроса всегда интересно узнать каково его состояние:

```
ExecStatusType PQresultStatus(const PGresult *res);
```

На вход подаётся структура `PGresult`, создаваемая в результате работы `PQexec`-подобных функций, а на выходе получаем информацию о состоянии в виде числа, значение которого можно сравнить со следующими константами:

PGRES_COMMAND_OK — всё прошло хорошо (для запросов, которые не возвращают данные, например, `INSERT`),

PGRES_TUPLES_OK — всё прошло хорошо, плюс получены данные в ответ на запрос (для запросов типа `SELECT` или `SHOW`),

PGRES_EMPTY_QUERY — строка запроса была почему-то пустой,

PGRES_COPY_OUT — идёт передача данных *от* сервера,

PGRES_COPY_IN — идёт передача данных *на* сервер,

PGRES_BAD_RESPONSE — ошибка, ответ сервера не разборчив,

PGRES_NONFATAL_ERROR — ошибка, не смертельно: предупреждение (notice) или информация к сведению (warning),

PGRES_FATAL_ERROR — в процессе выполнения запроса произошла серьёзная ошибка.

Для получения более подробной информации об ошибке следует воспользоваться функцией

```
char *PQresultErrorMessage(const PGresult *res);
```

При вызове этой функции в качестве результата будет сформирована строка с информацией об ошибке или пустая строка если всё прошло хорошо.

Получение данных При получении данных предполагается, что статус запроса соответствует **PGRES_TUPLES_OK**. Теперь, если примерно известно что хочется получить в результате запроса, то для получения данных достаточно четырёх функций:

```
int PQntuples(const PGresult *res);
int PQnfields(const PGresult *res);
char *PQgetvalue(const PGresult *res,
                 int row_number, int column_number);
int PQgetisnull(const PGresult *res,
                 int row_number, int column_number);
```

Первые две функции являются информационными и позволяют узнать сколько в результате запроса получено строк (`PQntuples`) и сколько колонок в каждой такой строке (`PQnfields`). Возьмите на заметку, что 0 строк — это тоже хороший результат.

Функция `PQgetvalue` позволяет получить доступ к полученным данным. В качестве параметров кроме структуры соединения (`res`) передаётся номер строки (`column_number`) и номер колонки (`column_number`). Все данные возвращаются так же в виде текстовой строки, как и посылаются, то есть, эти данные необходимо перевести в привычный формат. Например, в случае целых чисел можно воспользоваться функцией `atoi`.

Следует помнить, что данные SQL могут иметь неопределённое значение (NULL). Если подобная возможность существует, то перед получением значения проверить, а определено ли оно. `PQgetisnull` позволяет разобраться с этой проблемой. По передаваемым параметрам эта функция эквивалентна `PQgetvalue`, а в качестве результата возвращает 1, если значение *не* определено и 0, если опеределено.

Кроме упомянутых существует целый ряд функций, позволяющих получить информацию о полученных данных, как то: имя колонки (`PQfname`), размер передаваемых данных в байтах (`PQgetlength`) и тому подобное. Для экранирования специальных символов при операции с бинарными или текстовыми данными есть набор сервисных функций `PQescape*`.

COPY SQL команда `COPY` является расширением специфичным для PostgreSQL. Основное преимущество SQL — *«всё есть понятный текст»*, в некоторых случаях, когда надо передавать большие объёмы данных, оборачивается недостатком. Функции `PQputCopyData` и `PQgetCopyData` позволяют под час значительно ускорить передачу данных между сервером и клиентом.

Асинхронные сигналы Стандартный SQL не предполагает взаимодействия разных пользователей, кроме как через изменение данных в таблицах. PostgreSQL позволяет посылать асинхронные сигналы с помощью команд `LISTEN` и `NOTIFY`. `LISTEN "имя сигнала"` передаётся серверу как обычная SQL-команда. Если статус запроса становится равным **PGRES_COMMAND_OK**, то это означает, что ранее был выполнен запрос `NOTIFY "имя сигнала"`. Если же инициализация сигнала (`NOTIFY`) ожидается позже регистрации (`LISTEN`), то функция `PQnotifies` позволяет после любого запроса проверить наличие сигнала вновь.

Сборка «мусора» «Мусор» убирать придётся руками. Каждая функция типа `PQexec` создаёт объект типа `PGresult`. После того как вся необходимая информация о результатах запроса получена, следует освободить память, занимаемую этим объектом с помощью команды:

```
void PQclear(PGresult *res);
```

Если утечки памяти Вас не волнуют, то можно этого и не делать. В этом случае следует побеспокоиться о том: «Почему Вас не беспокоят утечки памяти?» ☺

4.1.3. Большие объекты

Ещё один способ сохранять неструктурированные данные в PostgreSQL — это «пихать» их как большие объекты (Large Objects). PostgreSQL предоставляет интерфейс схожим с файловым интерфейсом Unix: open (`lo_open`), read (`lo_read`), write (`lo_write`), lseek (`lo_lseek`) и так далее. Все `lo_*` команды работают со значениями полученными из колонки с типом `oid`. `oid` — это специальный тип данных, который является ссылкой на объект произвольного типа. То есть последовательность работы с большим объектом следующая: создаётся большой объект (`lo_create`). Далее возвращаемый `lo_create` указатель `Oid` используется для записи данных в большой объект (`lo_import`/`lo_write`), а затем этот указатель вставляется в таблицу с помощью стандартных SQL операторов. Чтение происходит в обратном порядке (`lo_export`/`lo_read`). Все операции с большими объектами должны происходить внутри транзакции.

P.S. Необходимость интерфейса больших объектов на текущий момент не так уж и очевидна. Стандартными средствами в PostgreSQL можно сохранять бинарные данные размером вплоть до 1 Гб, что вполне может соперничать с максимальным размером для большого объекта в 2 Гб.

4.2. ECPG

Чтобы не отставать от коммерческих баз данных PostgreSQL имеет свой собственный вариант «встроенного SQL». Эта технология позволяет смешивать обычный язык C с SQL-структурами, примерно следующим образом:

```
// файл test.pgc
#include <stdio.h>
#include <stdlib.h>
// структура для обработки ошибок
EXEC SQL include sqlca;
// реакция в случае ошибки/предупреждения
EXEC SQL whenever sqlwarning sqlprint;
EXEC SQL whenever sqlerror do ExitForError();
void  ExitForError() {
  fprintf(stderr,"Всё, конец - это фатально.\n");
  sqlprint();
  exit(1);
}

int main(int argc, char **argv)
{
// определение переменных, чтобы их можно было использовать
```

```
// инструкциях ECPG
  EXEC SQL BEGIN DECLARE SECTION;
   const char *dbname = "test";
   const char *user = "baldin";
   VARCHAR FIO[128];
   VARCHAR NUMBER[128];
  EXEC SQL END DECLARE SECTION;
// соединение с базой данных
// внешние переменные предваряются двоеточием
  EXEC SQL CONNECT TO :dbname USER :user;
// определение курсора через SELECT
  EXEC SQL DECLARE mycursor  CURSOR FOR
    SELECT fio, number FROM fiodata,phonedata
                       WHERE fiodata.id=phonedata.id;
  EXEC SQL open mycursor;
// чтение данных из курсора
  EXEC SQL FETCH NEXT FROM mycursor INTO :FIO,:NUMBER;
  while (sqlca.sqlcode == 0) { // не 0, если данные больше нет
    printf("ФИО: %s номер: %s\n",FIO.arr,NUMBER.arr);
    EXEC SQL FETCH NEXT FROM mycursor INTO :FIO, :NUMBER;
  }
// разъединение с базой данных
 EXEC SQL DISCONNECT;
}
```

Все SQL-команды начинаются с метки `EXEC SQL`. Эта метка позволяет затем препроцессору `ecpg` обработать и произвести C-исходник. Внутри SQL-команд можно использовать C-переменные. Для этого переменным в начале добавляется двоеточие «:».

Для компиляции выше процитированного исходника (файл `test1.pgc`) необходимо выполнить следующие действия:

```
> # установить ecpg
> sudo apt-get install libecpg-dev
> # запустить препроцессор
> ecpg test1.pgc
> # скомпилировать получившийся исходник
> gcc  -o test1 test1.c -I‘pg_config --includedir‘ -lecpg
> # проверка работоспособности программы
> ./test1
ФИО: Иванов И.П. номер: 555-32-23
ФИО: Балдин Е.М. номер: 555-41-37
ФИО: Балдин Е.М. номер: (+7)5559323919
```

Удобно это или нет — решайте сами.

4.3. Всё остальное

Статья называется «Интерфейсы», а большая часть *посвящена* только одному из них. Дело в том, что этот «один» является родным и наиболее полным, а всё остальное лишь подмножество. В простейшем случае все интерфейсы одинаковы: открыл соединение, послал запрос, обработал результаты запроса, закрыл соединение. Так же заметна энергосберегающая тенденция везде делать ровно один интерфейс на все типы СУБД.

bash Да, да к bash тоже есть свой интерфейс, правда для этого надо патчить его исходники. Возни, конечно, не мало — зато прямо в shell-скриптах можно обращать к базе данных ☺.

Страничка проекта: http://www.psn.co.jp/PostgreSQL/pgbash/index-e.html.

Java Совершенно ожидаемо, что Java общается с PostgreSQL стандартным образом, а именно через JDBC. Поэтому если знаком с Java, то достаточно добыть драйвер JDBC для PostgreSQL, например отсюда: http://jdbc.postgresql.org/ или в Debian (Sarge) набрать

```
> sudo apt-get install libpgjava
```

и, прочитав README к пакету, приступить к работе.

lisp Точнее Common Lisp. Скорее всего эти драйвера подойдут и для других диалектов:

```
> sudo apt-get install cl-pg
#или
> sudo apt-get install cl-sql-postgresql
```

Второй вариант является драйвером для единого интерфейса доступа к SQL-базам данных из Common Lisp CLSQL (http://clsql.b9.com/).

perl Интерфейс для связи с PostgreSQL DBD-Pg используется в perl через DBI[2]. Все подробности на CTAN: http://search.cpan.org/~dbdpg/DBD-Pg/Pg.pm.

```
> libdbd-pg-perl
```

[2]DBI — унифицированный интерфейс для доступа к данным. Подробнее об этом пакете можно посмотреть на CTAN: http://search.cpan.org/~timb/DBI-1.52/DBI.pm

DBD-Pg охватывает фактически все имеющиеся на сегодня возможности PostgreSQL от больших объектов (large objects), до точек сохранения (savepoints).

PHP О том как использовать PostgreSQL в PHP-проектах можно прочитать здесь: http://www.php.net/manual/en/ref.pgsql.php. Установить драйвер можно, например, так:

```
> sudo apt-get install php5-pgsql
```

Говорят, почти единственной причиной, по которой PHP-разработчики предпочитают MySQL является то, что раньше не было «родной» версии PostgreSQL под альтернативную операционную систему. С версии 8.0 PostgreSQL конкретно *этот* довод «против» уже не работает.

Python Модуль для Python существует уже больше десяти лет. Подробности выясняются здесь: http://www.druid.net/pygresql/. Установка модуля:

```
> sudo apt-get install python-pygresql
```

Более «молодая» и по утверждениям пользователей более стабильная библиотека для связи с PostgreSQL psycopg2 (http://initd.org/projects/psycopg2) так же устанавливается из коробки:

```
>  sudo apt-get install python-psycopg2
```

Ruby Что-то есть здесь: http://ruby.scripting.ca/postgres/. Установка, как обычно:

```
> sudo apt-get install libdbd-pg-ruby
```

ODBC Разработка драйвера идёт на pgFoundry, аскетичная страничка проекта здесь: http://pgfoundry.org/projects/psqlodbc/. Установка:

```
> sudo apt-get install odbc-postgresql
```

Послесловие

Очевидно, что есть много чего ещё. При желании можно самому написать, благо родной C-интерфейс уже знаком, а логика достаточно прозрачна: открыл соединение, обменялся SQL-запросами и обязательно закрыл соединение. С другой стороны не стоит изобретать велосипеда и лучше для начала посмотреть, что было уже сделано, например, здесь: http://techdocs.postgresql.org/oresources.php.

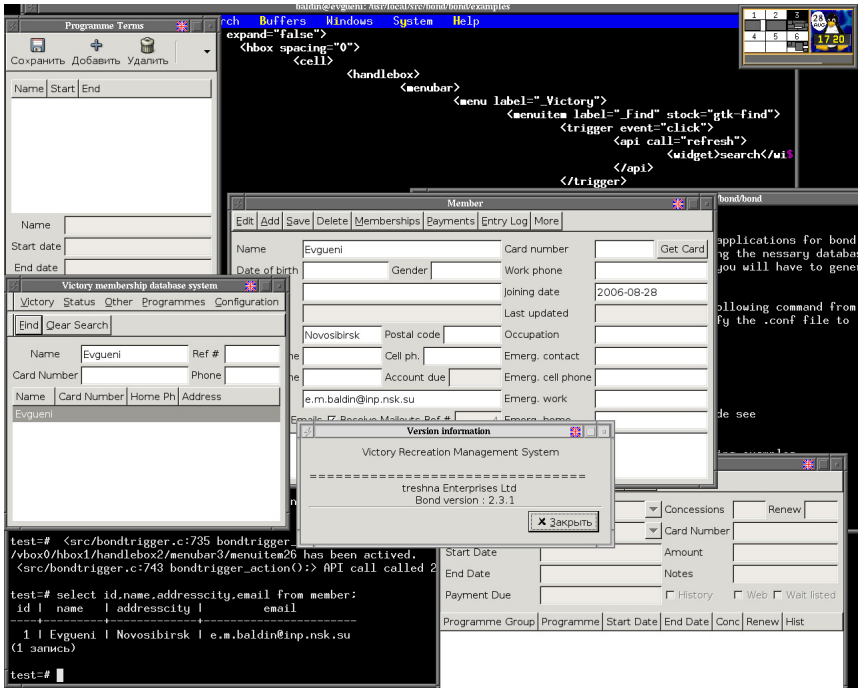

Рис. 4.1. Формочки, XML (правда на заднем фоне и без подсветки) и связь с базой данных — это bond

Врезка про bond

Лень писать всё самому? Но не лень изучать XML? Тогда BOND — это программа для Вас. Сайт проекта http://www.treshna.com/bond/.

Рабочей частью пакета является исполняемый файл `bondfrontend`, который осуществляет связь с базой данных и может «прикинуться» любой формой. Описание формы хранится в обычном xml-файле. Используемый диалект xml подробно описан в документации.

История пакета насчитывает уже пять лет. Программа доступна под GPL, то есть исходники производных продуктов надо открывать, а если хочется пожадничать, то есть версия и для такого случая, но за деньги. Доступна версия и под win32.

Внимательно читаем README и устанавливаем всё что там перечисляется. Сборка осуществляется с помощью `scons`, который позиционируется как замена `make` с функциональность `automake`/`autoconf` и синтаксисом от Python. Установка производится по умолчанию `/usr/local/`. Для установки (`scons install`) необходимы привилегии системного администратора. Далее читаем документацию и изучаем директорию `examples`.

Глава 5.

Настройка PostgreSQL

> — Между прочим, — сказал Роман громко, — уже в течение двух минут я пытаюсь его пассивизировать, и совершенно безрезультатно
>
> «Понедельник начинается в субботу» АБС

«Тюнинг» — это не операция — это образ жизни. Очевидно, что когда необходимые характеристики можно улучшить несколькими способами, то всеми способами их и надо улучшать. Но опять же следует помнить, что избыточная и ранняя оптимизация — корень многих зол. Если система работает и «не кашляет», то может быть не стоит её «настраивать»?

5.1. О железе

«Театр начинается с вешалки», а большая база данных начинается с выбора сервера. PostgreSQL умудряется работать почти при любой конфигурации, но если Вас интересует результат, то следует знать на что обращать внимание. Очевидно, что рамки на железо диктуются исключительно объёмом денежных ресурсов, но в имеющихся пределах всегда можно что-то подвигать.

Как и для всякой программы, работающей с большим объёмом данных, дисковая подсистема является для PostgreSQL определяющей. Поэтому если есть возможность, то лучше вложиться именно в неё. В противоположность дисковой подсистеме процессор нагружается не очень сильно. Поэтому для сервера достаточно, чтобы процессор просто был, хотя лучше чтобы он был не один. Если денег на покупку хорошего SCSI диска нет, то следует вложиться в память.

К вопросу о дисках можно сказать, что чем их больше — тем лучше. По возможности следует выделить отдельный диск для журнала транзакций (`pg_xlog`). Избыток дисков так же позволит собрать из них RAID. Хоть никто и не отменяет

бэкапа, но дополнительная избыточность для дисковой подсистемы, как и источники бесперебойного питания, сэкономят массу сил и нервов.

Относительно недорогие системы снабжены дешёвыми встроенными RAID-контроллерами на четыре диска. Не следует использовать эти контроллеры, а лучше настроить софтверный RAID и не надо использовать RAID 5[1] для небольшого числа дисков. До 6 дисков включительно лучше RAID 1+0[2]. Избыточность во всём — это только похоже на паранойю.

На сервер, где уже работает PostgreSQL не следует «подселять» другие базы данных или программы, осуществляющие интенсивный обмен с дисковой памятью. А вот программы, которые вместо этого интенсивно используют процессор, например, Apache, вполне уживаются если памяти достаточно.

Собственно говоря, можно особо не стараться. PostgreSQL вполне себе работает и на обычном пользовательском компьютере. Более того пришло время серьёзных баз данных на десктопе. Существует куча приложений, которые занимаются индексированием и каталогизацией, при этом создавая свои доморощенные «базки». А ведь решение очевидно и PostgreSQL вполне может стать им. Да и самим данным вовсе не помешает оказаться в нормальной базе специально для этих данных созданной. Это «толстый» такой намёк, так как хорошее хранилище данных для Вашей, ещё не написанной программы, на дороге не валяется.

5.2. Конфигурационные файлы

Настройка конфигурационных файлов — это не единственный способ настройки сервера базы данных. Умолчания можно изменить непосредственно при сборке из исходников. Значения можно передавать непосредственно серверу `postmaster` в командной строке, используя ключик `-c`. Так же можно определить переменную окружения `PGOPTIONS`, а значения некоторых переменных можно изменить прямо в процессе выполнения запросов.

Концентрация внимания на конфигурационных файлах объясняется тем, что в любом случае их придётся настраивать. По умолчанию PostgreSQL сконфигурирован в расчёте на минимальное потребление ресурсов, и это не может не сказаться на скорости. Что подходит для локальной записной книжки — не годится для боевого сервера.

Все настройки очень подробно описаны в документации. Для любителей «выжимать из программы всё» существует специальный список рассылки psql-performance: http://archives.postgresql.org/pgsql-performance/. В документации на странице Power PostgreSQL http://www.powerpostgresql.com/Docs также можно найти некоторое количество полезных подсказок.

Для настроек PostgreSQL используются файлы:

[1] Один диск в массиве выделяется под контрольные суммы

[2] зеркалирование (1) + объединение (0)

pg_hba.conf — политика доступа и идентификации пользователей,

postgresql.conf — собственно говоря, настройки сервера.

5.2.1. pg_hba.conf

Часто для локальных нужд при использовании PostgreSQL в качестве своей личной записной книжки, склада данных или даже «помойки» не требуется открывать сетевой доступ к базе данных. По умолчанию PostgreSQL настроен так, что каждый локальный пользователь может подсоединиться к базе совпадающей по названию с регистрационным именем клиента, при условии что такая база данных уже создана.

Но это не значит, что так бывает всегда. PostgreSQL предоставляет свои механизмы для управления пользователями с помощью тройки команд `CREATE USER`, `DROP USER` и `ALTER USER`. В случае не совпадающих регистрационных имен в СУБД PostgreSQL и в системе или при доступе к базе данных с других компьютеров необходимо «обговорить» правила доступа к данным на сервере.

Для настройки политики доступа к серверу волей-неволей придётся заглянуть в файл `pg_hba.conf`[3]. Тело файла представляют из себя однострочные записи, каждая из которых регулирует правила получения доступа для конкретной машины или для целой группы IP. Файл `pg_hba.conf` выглядит примерно так:

```
# Разрешаем доступ через локальные unix-сокеты абсолютно
#всем пользователям к базам данным, совпадающим по названию
#с регистрационными именами
local   all          all                          ident sameuser
# Доверяем пользователю alex с указанного IP безгранично
#в рамках базы данных photos
host    photos       alex   130.255.204.48/32  trust
# Требуем пароль от пользователя baldin при доступе с
#компьютеров из сети 128.138.242.192/27 к базам данных
#data и photos
hostssl data,photos  baldin 128.138.242.192/27  md5
```

В файле по умолчанию присутствует подробная информация о формате записей, первое поле которых определяет тип записи:

- local — эта запись определяет политику для локального доступа через локальные UNIX-сокеты.
- host — эта запись определяет политику для сетевого TCP/IP соединения и годится для соединений с использованием SSL и без него. Для того чтобы можно было доступаться к базе данных по сети необходимо правильно настроить listen_addresses в `postgresql.conf`.

[3] hba — host-based authentication.

- hostssl — определяется политика для сетевого соединения с обязательным использованием SSL.
- hostnossl — антипод hostssl.

Второе поле представляет из себя имя базы данных для которой определяется политика. Имя **all** зарезервировано для всех баз данных, а имя **sameuser** для базы данных, совпадающей с именем пользователя. Имена баз данных можно перечислять через запятую. Так же в качестве имени можно добавить имя файла со списком баз, разделённых запятой или пробелами. Для этого к имени файла следует добавить символ «@» в качестве префикса.

Третье поле — имя пользователя. Как и в случае имён баз данных можно работать со списками. Имя **all** зарезервировано для всех пользователей. Так же доступ можно открыть для группы пользователей (role), для этого перед именем группы следует поставить знак «+».

Следующие ноль (в случае записи local), одно (нотация CIDR[4]) или два поля (адрес и сетевая маска) занимает сетевой адрес компьютера или подсети для которого настраивается политика доступа.

Предпоследнее обязательное поле отведено под метод авторизации:

- trust — полностью доверяем этому клиенту.
- reject — отказ в доступе.
- ident — доступ по регистрационной записи клиента. Часто применяется для локальных соединений. RFC 1413.
- md5 — авторизация по паролю зашифрованному с помощью алгоритма md5.

  Если клиент использует библиотеку для доступа к PostgreSQL версии старше 7.2, то вместо метода md5 следует использовать метод crypt.

  Пароль при желании и значительной степени бесшабашности можно передавать открытым текстом с помощью метода password.
- pam — авторизация с помощью **P**luggable **A**uthentication **M**odules. Этот сервис предоставляется операционной системой.

  Подробнее о PAM написано здесь: http://www.kernel.org/pub/linux/libs/pam/.
- krb4 и krb5 — авторизация с использованием механизма Kerberos версии 4 и 5, соответственно. Это индустриальный стандарт авторизации. То есть, кому надо — тот знает что это такое.

После метода ему можно передать опции в последнем необязательном поле.

[4] Classless Inter-Domain Routing

5.2.2. postgresql.conf

Настройки в `postgresql.conf` разбиты по группам и подробно задокументированы прямо в файле. Здесь будут описаны не все настройки, но важнейшие из них будут отмечены.

Настройка соединений и авторизация (Connections and Authentication)

Политика авторизации настраивается в `pg_hba.conf`. Здесь же собраны в основном технические параметры.

- Настройка соединений (connection settings).

  listen_addresses После настройки `pg_hba.conf` можно и нужно смело устанавливать `*` — слушаем все интерфейсы, которые есть в наличии. По умолчанию (localhost) запросы принимаются только от локальных пользователей в том числе и через интерфейс обратной связи (loopback-интерфейс `127.0.0.1`).

  port Номер порта, который слушает сервер в ожидании соединений. По умолчанию он равен 5432.

  max_connections Число клиентов, которые могут подсоединяться к базе данных одновременно не может быть бесконечным. Каждое подсоединение порождает ещё один процесс postmaster, что, естественно, требует ресурсов. Средней «паршивости» современный однопроцессорный компьютер со стандартным наполнении без особых проблем может обслуживать 100-200 соединений, но, например, 600 активных соединений будут уже явной проблемой.

  Любая попытка подсоединиться сверх указанного лимита приведёт к отказу от обслуживания. Плохо написанная программа в цикле открывающая, но не закрывающая за собой соединения, легко создаст проблему.

  Если число клиентов жёстко ограничено, то имеет смысл уменьшить этот параметр до минимально возможного значения.

  superuser_reserved_connections Число соединений, которые зарезервированы для суперпользователя, чтобы он мог всегда зайти, разобраться в чём дело, а затем принять меры. Не стоит совсем отказываться зарезервированных соединений, причём одного зарезервированного соединения может оказаться не достаточно — 2 это минимум.

- Безопасность и авторизация (security and authentication)

  authentication_timeout Время ожидания в секундах для прохождения авторизации. По умолчанию это время равно одной минуте. Не позволяет

клиенту «зависнуть» и заблокировать ресурс соединения на очень долгое время.

ssl Разрешается доступ через ssl. Для работы через SSL необходимо создать публичный ключ и сертификат. Это требует некоторых усилий, зато позволяет немного успокоить себя на тему безопасности сетевых соединений.

Управление ресурсами (Resource Consumption)

Правильная оценка имеющихся ресурсов — путь к эффективном планированию. А эффективное планирование позволяет в процессе работы не сильно увеличивать окружающую нас энтропию и в то же время добиваться поставленной цели. Очевидные истины.

- Память (memory)

  shared_buffers Объём совместно используемой памяти, выделяемой сервером PostgreSQL для кэширования данных, определяется числом страниц (shared_buffers) по 8 килобайт каждая. Естественно, данные умеет кэшировать не только сам PostgreSQL, но и операционная система сама по себе делает это очень неплохо. Поэтому нет необходимости отводить под кэш всю наличную оперативную память. Оптимальное число shared_buffers зависит от многих факторов, поэтому проще для начала принять следующие ориентиры:

  – Обычный настольный компьютер с 512 Мб и небольшой базой данных — 8–16 Мб или 1000–2000 страниц.

  – Не сильно выдающийся сервер предназначенный для обслуживания базы данных с объёмом оперативной памяти 1 Гб и базой данных около 10 Гб— 80–160 Мб или 10000–20000 страниц.

  – Сервер посерьёзнее с несколькими процессорами на борту, с объёмом памяти в 8 Гб и базой данных занимающей свыше 100 Гб обслуживающий несколько сотен активных соединений одновременно — 400 Мб или 50000 страниц.

  work_mem Под каждый запрос можно выделить личный ограниченный объём памяти для работы. Этот объём может использоваться для сортировки, объединения и других подобных операций. При превышении этого объёма сервер начинает использовать временные файлы на диске, что может существенно замедлить скорость обработки запросов. Предел для work_mem можно вычислить, разделив объём доступной памяти (физическая память минус объём занятый под другие программы и под совместно используемые страницы shared_buffers) на максимальное число одновременно используемых активных соединений.

При необходимости, например, выполнения очень объёмных операций, допустимый лимит можно изменять прямо во время выполнения запроса. Поэтому нет нужды изначально задавать теоретический предел.

maintenance_work_mem Эта память используется для выполнения операций по сбору статистики (ANALYZE), сборке мусора (VACUUM), создания индексов (CREATE INDEX) и для добавления внешних ключей. Размер выделяемой под эти операции памяти должен быть сравним с физическим размером самого большого индекса на диске. Как и в случае work_mem эта переменная может быть установлена прямо во время выполнения запроса.

max_prepared_transactions Определяет максимальное число подготовленных транзакций (команда PREPARE TRANSACTION). Подготовленные транзакции выполняются, но результат их не будет доступен пока их не подтвердят (COMMIT). Так же можно такие транзакции и отклонить (ROLLBACK). Если эта сущность нигде не используется, то переменную можно занулить.

- Карта неиспользованного пространства (free space map)

  При удалении записи не удаляются физически, а только помечаются как удалённые. Именно таким образом мусор и собирается.

  max_fsm_pages Для целей «сборки мусора» полезно знать, где этот мусор находится. Число страниц, отведённых под эту задачу, должно быть больше, чем число удалённых или изменённых записей между сборками мусора (VACUUM). Если страниц достаточно выполнение жёстких оптимизирующих операций, таких как VACUUM FULL или REINDEX никогда и не понадобится. Так как объём требуемой для этого памяти не очень большой (по 6 байт на страницу), то жадничать не стоит. Проще всего узнать необходимое число max_fsm_pages запустив, VACUUM VERBOSE ANALYZE.

  max_fsm_relations Число таблиц для которых создаются карты неиспользованного пространства. По умолчанию это число равно 1000. В случае большего числа используемых таблиц это значение можно и нужно увеличить, тем более что на каждую таблицу требуется всего по семь байт.

- Системные ресурсы (kernel resource usage)

  preload_libraries Если для исполнения запроса требуется загрузить какую-либо разделяемую библиотеку, то действует правило: «загружаем при первом использовании», что замедляет исполнение самого первого такого запроса. Это можно обойти, загрузив необходимые библиотеке при старте сервера, то есть воспользоваться формулой: память в обмен на скорость.

Таким образом можно подгрузить разделяемую библиотеку для используемых в запросах процедурных языков.

- Оценка стоимости сборки мусора (cost-based vacuum delay)

  Немного подробнее про сборку мусора будет рассказываться далее. Обычно нет необходимости заглядывать в этот раздел, так как операции по сборке мусора (VACUUM) и анализа (ANALYZE) выполняются достаточно быстро.

- Запись в фоне (background writer)

  Начиная с версии PostgreSQL 8.0 вместе с основным сервером стартует процесс для записи данных в фоне. При выполнении запроса нет необходимости ждать самого акта записи, так как это гарантировано сделает background writer.

Журнал транзакций (Write Ahead Log)

Наличие журнала транзакций или WAL (write ahead log) позволяет увеличить скорость выполнения операций требующих изменения данных в следствии того, что в журнал информация об изменениях пишется последовательно, а изменения в самих таблицах могут быть отложены до «лучших времён» — своеобразный кэш только на диске. Если же база данных используется в основном для чтения, то в журнале транзакций нет особой необходимости, но это не повод от него отказываться.

- Настройки (settings)

  fsync По умолчанию эта опция включена (true). В этом случае PostgreSQL пытается записать данные на диск физически. Это на самом деле не такая уж и простая операции, так как кэши существуют не только в системе, но и в контроллерах и в дисках. Вполне можно представить себе такую ситуацию, что слишком умный диск для увеличения своей производительности в тестах рапортует о том, что данные записаны, а при перебое с электричеством выясняется, что это была просто шутка. Сбои самого сервера не приводят к порчи данных, но сервер живёт в окружении далеко не идеальной операционной системе, которая в свою очередь управляет далеко не идеальными физическими устройствами. Так что проверенное железо, источники бесперебойного питания, бэкап, бэкап и ещё раз бэкап.

  Если Вы доверяете своему железу, то эту опцию можно выключить. Здесь можно поменять немного безопасности на скорость. Хотя более оптимальным решением является перенос журнального файла на отдельный быстрый диск. То есть прикупить скорость за деньги.

- Контрольные точки (checkpoints)

По свершению каких-то определённых условий или истечению контрольного времени сервер гарантировано переносит данные, записанные в WAL непосредственно в таблицы, даже если очень сильно занят на других запросах.

checkpoint_segments Объём кэша на диске. Физический объём места на диске, требуемый под кэш вычисляется по формуле:

$$(\text{checkpoint\_segments} \times 2 + 1) \times 16\,\text{Мб}.$$

Следует выделить столько, сколько не жалко, осознавая, что 32 сегмента займёт на диске свыше 1 Гб.

checkpoint_timeout Время, через которое WAL очищается насильно. Значение по умолчанию 300 секунд.

checkpoint_warning Если кэш а диске заполняется быстрее чем число секунд checkpoint_warning, то посылается предупреждение, которое будет передано в журнальный файл. Это намёк, что кэш на диске следует увеличить.

- Архивация (archiving)

  archive_command Для целей создания непрерывного резервного копирования (это возможность для настоящих параноиков) журнал необходимо копировать куда-то ещё. В этом случае здесь должна быть команда, которая будет использоваться системой для копирования данных. Подробности следует искать в документации в разделе «On-line backup and point-in-time recovery (PITR)».

Планирование запросов (Query Planning)

Здесь можно повлиять на логику действия планировщика. Возможно конкретно для Вашей системы значения по умолчанию не оптимальны, но менять их стоит только после серии тестов для выявления более оптимальной конфигурации под конкретную платформу и конкретные запросы. В большинстве случаев углубляться в тонкости настроек из этого раздела имеет смысл только в случае очень изощрённых запросов.

- Методология планировщика (planner method configuration)

  В этом разделе перечислены алгоритмы, которые можно использовать для извлечения данных. Для целей тестирования какие-то из них можно отключить.

- Оценочные константы (planner cost constants)

effective_cache_size PostgreSQL в своих планах опирается на кэширование файлов, осуществляемое операционной системой. Этот параметр соответствует максимальному размеру объекта, который может поместиться в системный кэш. Установка этого параметра не приводит к увеличению выделяемой памяти. Это значение используется только для оценки.

effective_cache_size можно установить в 1/3 от объёма имеющейся в наличии оперативной памяти, при условии, если вся она отдана в распоряжение PostgreSQL.

Сообщения об ошибках и событиях (Error Reporting and Logging)

Оптимизация возможна только в случае обратной связи. Сервер PostgreSQL может много чего про себя рассказать. Этот раздел настроек посвящён тому, как правильно его об этом попросить.

- Местоположение журнального файла (where to log)

  log_destination Здесь можно выбрать способ записи в журнальный файл: stderr или syslog. Метод stderr хорош для тестирования, но по хорошему журналированием должна заниматься специальная служба, а это нас приводит к необходимости изучить что такое syslog.

  В случае выбора метода stderr придётся определить директорию для журнального файла (log_directory), имя журнального файла (log_filename) и другие параметры (log_rotation_age, log_rotation_size, log_truncate_on_rotation) свойственный службе журналирования.

  В случае выбора метода syslog настроить его с помощью syslog_facility (необходимо знать как пользоваться syslog) и syslog_iden — идентификационный префикс для сообщений получаемых от PostgreSQL.

- По какому случаю создаём запись (when to log)

  Обычно, нет необходимости записывать в дневник информацию о каждом чихе, но при серьёзном разбирательстве данные о числе чихов в секунду и их классификация могут подтолкнуть в нужном направлении.

  Каждому событию можно присвоить какой-то определённый уровень. Например, уровень PANIC означает, что плохо стало всем, а уровень WARNING сообщает просто о подозрительных, но вполне законных событиях. В порядке возрастания подробности уровни имеют примерно следующую классификацию: PANIC, FATAL, LOG, ERROR, WARNING, NOTICE, INFO, DEBUG[1-5]. Уровень можно установить в процессе выполнения запроса.

  Следует осознавать, что журнал необходим при разбирательствах, но его активное использование ведёт к деградации производительности. Обычно эта

уменьшение производительности находится в пределах 5% при условии, что журнальный файл расположен на другом диске нежели журнал транзакций.

log_min_messages Характеризует уровень подробности записей в журнальный файл.

log_error_verbosity Характеризует степень подробности делаемых в журнал записей. Различаются три степени: TERSE, DEFAULT и VERBOSE.

client_min_messages Характеризует уровень подробности сообщений отсылаемых клиенту.

log_min_error_statement Характеризует уровень подробностей записей в журнальный файл создаваемых в результате исполнения SQL-запросов.

- Что именно пишем в журнал (what to log)

  В этом разделе настроек перечислены различные возможные источники записей для журнала. Можно записывать информацию о делаемых соединениях (log_connections), информацию о выполняемых запросах (log_statement), информацию о времени уходящему на выполнение запросов (log_duration) и тому подобное. С помощью переменной log_line_prefix можно настроить идентификацию каждой записи по пользователю, IP, базе данных и так далее.

Сбор статистики (Run-Time Statistics)

Есть ложь, гнусная ложь и статистика. База данных врать не научена, поэтому остаётся только статистика. Этот раздел настроек отвечает за её сбор. Пока нет необходимости в мониторировании активности базы данных, нет необходимости здесь что-то править.

Для того чтобы работала автоматическая сборка мусора опции stats_start_collector и stats_row_level должны быть включены.

Автоматическая сборка мусора (Automatic Vacuuming)

Ну мусор, ну и пусть. Места много — зачем напрягаться, да ещё автоматически? В этом есть какая-то логика, но кроме сборки мусора (VACUUM) производится ещё и анализ (ANALYZE). Периодическое выполнение команды ANALYZE необходимо для нормального функционирования планировщика. Собранная с помощью этой команды статистика позволяет значительно ускорить выполнение SQL-запросов. То есть, если не хочется настраивать автоматическую сборку мусора, то в любом случае её придётся делать только теперь в ручную.

Процесс обычной сборки мусора в PostgreSQL (VACUUM без приставки FULL) не блокирует таблиц и может выполняться в фоне, не мешая выполнению запросов. Начиная с PostgreSQL версии 8.1 процесс автоматической сборки мусора выделяется в отдельный процесс. Эта группа настроек контролирует работу этого процесса.

autovacuum Если Вы лучше чем PostgreSQL знаете когда следует производить сборку мусора, то автоматику можно выключить. Хотя лучше её просто правильно настроить. С другой стороны сборка мусора оттягивает на себя ресурсы системы и если это не допустимо, то её можно отложить на некоторое время.

При настройке службы автоматической сборки мусора и анализа следует понимать, что один из зарезервированных с помощью superuser_reserved_connections слотов может оказаться в нужный момент занят.

autovacuum_naptime Время в секундах через которое база данных проверяется на необходимость в сборке мусора. По умолчанию это происходит раз в минуту.

autovacuum_vacuum_threshold Порог на число удалённых и изменённых записей в любой таблице по превышению которого происходит сборка мусора (VACUUM). По умолчанию этот порог равен 1000 и его вполне можно уменьшить.

autovacuum_analyze_threshold Порог на число вставленных, удалённых и изменённых записей в любой таблице по превышению которого запускается процесс анализа (ANALYZE). По умолчанию это порог равен 500. Никто не запрещает сделать его поменьше.

autovacuum_vacuum_scale_factor Процент изменённых и удалённых записей по отношению к таблице по превышению которого запускается сборка мусора. Значение по умолчанию равно 0.4.

autovacuum_analyze_scale_factor То же, что и предыдущая переменная, но по отношению к анализу. Значение по молчанию равно 0.2.

Настройки для пользователя по умолчанию (Client Connection Defaults)

Настройки из этой группы задают некоторые значения по умолчанию для подсоединившегося к базе данных клиента и не влияют на производительность самого сервера. Исключением является разве что переменная statement_timeout, которая выставляет ограничение в миллисекундах на исполнение запроса. Да и то эта возможность по умолчанию не активирована. Но, если хочется настроить локаль по умолчанию, хотя это личное дело клиента, или порядок выбора объектов относящихся к различным пространствам имён, то можно что-то поправить и здесь.

Управление блокировками (Lock Management)

Очевидно, что блокировок следует избегать всячески, причём делать это надо начинать на этапе проектирования базы данных. К сожалению реальная жизнь отличается от планов.

deadlock_timeout Взаимные блокировки (deadlock) — это чрезвычайно уродливое явление, при котором вошедшие в клинч процессы ожидают освобождение ресурсов, которые сами же и захватили. PostgreSQL умеет разрешать эту проблему путём насильного прерывания одного из процессов. Проверка на deadlock это довольно длительная процедура, поэтому, прежде чем начать проверку блокировки на предмет является ли она взаимной, сервер выжидает указанное время. Значение по умолчанию равно 1000 миллисекунд. Для загруженных серверов имеет смысл это значение увеличить.

max_locks_per_transaction Вопреки своему названию это не жёсткий лимит на число блокировок, осуществляемых в пределах транзакций. Этот параметр входит в формулу устанавливающую предел на число одновременно существующих блокировок, то есть это скорее максимальное среднее:
max_locks_per_transaction × (max_connections + max_prepared_transactions)
В документации сказано, что 64 (число стоящее по умолчанию) — это исторически проверенный предел и чтобы превзойти его требуются определённые усилия.

5.3. О том, что думать тоже надо

Можно идеально настроить сервер, регулярно проводить сборку мусора, можно закупить самое дорогое железо и поставить рядом с ним дизельную электростанцию. Но если таблицы и отношения между ними создавались без плана, а запросы задаются криво, то проблемы гарантировано будут.

Выполнение запросов проверяются с помощью команды EXPLAIN ANALYZE, которая по полочкам разложит как ищется, сортируется, объединяется и группируется информация, какие для этого использовались алгоритмы и какие индексы были задействованы. Для любителей картинок pgAdmin III имеет графический интерфейс к этой команде. Ни в коем случае нельзя пренебрегать индексами и по возможности следует избегать блокировок.

Настройки PostgreSQL для 1C

По адресу http://v8.1c.ru/overview/postgres_patches_notes.htm лежат патчи. Это отличие версии сервера поставляемого с «1С:Предприятием 8» от оригинальных исходников PostgreSQL. Патч `postgresql-1c-8.1.5.patch` несёт в себе изменения в исходном файле настройки. Перечислим их:

- Допускаются сетевые соединения:

```
-#listen_addresses = 'localhost'
+listen_addresses = '*'
```

- немного увеличен размер разделяемой памяти с 8 Мб до 28 Мб:

```
-#shared_buffers = 1000
+shared_buffers = 3500
```

- оценка размера кэша системы изменилась с 8 Мб до 80 Мб:

```
-#effective_cache_size = 1000
+effective_cache_size = 10000
```

- Включён процесс автоматической сборки мусора:

```
-#stats_row_level = off
+stats_row_level = on
-#autovacuum = off
+autovacuum = on
```

- Максимальное среднее число блокировок увеличено более чем в два раза:

```
-#max_locks_per_transaction = 64
+max_locks_per_transaction = 150
```

Мне кажется, что судя по этим изменениям есть куда оптимизировать и сам продукт и настройки к PostgreSQL.

Глава 6.

Дополнительные главы

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

Зеркало. Понедельник начинается в субботу.

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 «On-line 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:

- 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.

---

[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 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`.

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.

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.

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.

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

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.

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 поддерживалась системой. Вывести список поддерживаемых локалей можно с помощью команды `locale -a`.

Так как локализация проводилась 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:

```
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`.

[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.

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.

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. Например, 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.