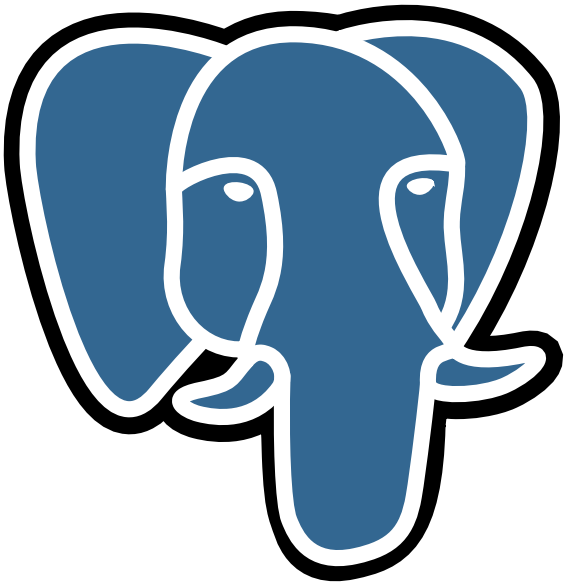

История о PostgreSQL

© Е.М. Балдин*

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

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

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

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

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

Оглавление

Познание бесконечности требует
бесконечного времени.

отдел *Абсолютного Знания*

Глава 3

Возможности PostgreSQL

В этой статье предпринята попытка сделать краткий обзор 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[2] (reliability) для сохранения данных является одним из основных показателей качества СУБД. Сохранение изменённых данных очень нетривиальная 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 дис-

[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).

[2] Этот момент отражён в FAQ fido7.ru.os.cmp следующим образом:
Q51: Народ, а вы стабильным софтом пользоваться не пробовали?
A51: Пробовали, но мэйнфреймы с дизель-генераторами не везде есть.

ке. 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:

```
test=> -- узнаём текущее время с точностью до секунды
test=> 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 типа 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».

```
test=> -- создаём массив для игры Тик–Так
test=> create table tictactoe (squares integer[3][3]);
test=> -- |x00| x = 1, 0 = -1
test=> -- |0xx| вставляем информацию о варианте игры
test=> -- |  x| крестики начинают и выигрывают
test=> insert into tictactoe
test->     values ('{{1,-1,-1},{-1,1,1},{0,0,1}}');
```

```
test=> ---- распечатываем сохранённую позицию
test=> select * from tictactoe ;
          squares
-----------------------------
 {{1,-1,-1},{-1,1,1},{0,0,1}}
test=> -- распечатываем значение первого столбца
test=> select squares[1:3][1:1] from tictactoe ;
   squares
--------------
 {{1},{-1},{0}}
```

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

```
test=> 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[3] 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 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»:

```
test=> -- Создаём и заполняем таблицу
test=> create table AplusB (A integer, B integer);
```

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

```
test=> insert INTO AplusB VALUES (1,1);
test=> insert INTO AplusB VALUES (2,2);
test=> insert INTO AplusB VALUES (3,3);
test=> -- Создаём новую функцию
test=> CREATE FUNCTION plus(integer, integer) RETURNS integer
test-> LANGUAGE SQL as 'SELECT $1 + $2;';
CREATE FUNCTION
test=> 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 начал 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 test
> createlang plpgsql test
# делаем то же самое, но для языка PL/Perl
> createlang plperl test
```

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:

```
test=> -- Создаём новую функцию с использование PL/pgSQL
test=> CREATE FUNCTION pgsql_plus(integer, integer) RETURNS integer
test-> LANGUAGE PLPGSQL as 'BEGIN return $1+$2; END;';
CREATE FUNCTION
test=> -- Создаём новую функцию с использование PL/Perl
test=> CREATE FUNCTION perl_plus(integer, integer) RETURNS integer
test-> LANGUAGE PLPERL as 'return $_[0]+$_[1]';
CREATE FUNCTION
test=> -- Проверяем, что всё работает
test=> select pgsql_plus(A,B) from AplusB;
test=> 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», которое может быть вставкой (INSERT), изменением (UPDATE), или удалением (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:

```
test=> -- Создаём «результирующую» таблицу
test=> create table ABresult (result integer);
test=> -- Создаём функцию, очищающую ABresult и
test=> -- заполняющую всё суммой A и B из AplusB.
```

```
test=> create function ABsumm() returns trigger as
test-> 'BEGIN
test'>          delete from ABresult;
test'>          insert into ABresult values (AplusB.A+AplusB.B);
test'>          return NULL;
test'>  END;'
test-> language 'plpgsql';
test=> -- Создаём триггер
test=> CREATE TRIGGER makeABresult
test=>          AFTER INSERT or UPDATE or DELETE on AplusB
test=>          FOR EACH STATEMENT execute procedure ABsumm();
CREATE TRIGGER
test=> -- Добавляем данных в таблицу AplusB
test=> insert into AplusB VALUES (100,200);
test=> -- проверяем, что триггер сработал
test=> 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 немного, 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).

```
test=> -- начинаем транзакцию
test=> BEGIN;
test=> -- здесь идёт блок операторов, который удачно завершается

test=> -- ставим метку
test=> SAVEPOINT savepoint_one;
test=> -- здесь идёт блок операторов, в котором произошла ошибка

test=> -- откатываемся до установленной метки,
test=> -- а не отменяем всю транзакцию
test=> ROLLBACK TO savepoint_one;
test=> -- повторяем последний блок

test=> -- завершаем транзакцию
test=> COMMIT;
test=> -- всё, теперь изменения доступны всем
```

3.5.2 Ограничения

Целостность данных обеспечивается не только многоверсионность (MVCC) 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[4]. Это довольно интересный механизм, который, например, позволяет моделировать иерархические структуры.

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

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.

3.6 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`