PostgreSQL

The Only Open Source RDBMS*

---

* но это не точно

История

Создан в 1986 году одним из разработчиков Ingres - прародителя Sybase, а в последствии и Microsoft SQL Server, и вообще многих баз. В 1995 году собственный язык запросов был заменен на SQL.

Название расшифровывается примерно так:

Post [in]gres + SQL

Термины

Схема - логическое объединение таблиц в базе данных
База данных - физическое объединение таблиц
Кластер - объединение нескольких баз данных в одной РСУБД

Кластер - историческое понятие, т.к. многие СУБД на заре истории работали только в с одним набором таблиц, никак логически не разделенных, postgres, таким образом, предоставлял кластер баз

Варианты и репозитории

Ванильная сборка (PGDG, PostgreSQL Global Development Group):
http://yum.postgresql.org, http://apt.postgresql.org
https://download.postgresql.org/pub/repos/yum/10/redhat/rhel-7-x86_64/pgdg-centos10-10-2.noarch.rpm

Postgres Professional
http://repo.postgrespro.ru/

2nd Quadrant Postgres-XL
https://www.postgres-xl.org/

Утилиты управления:

- **pgAdmin III, pgAdmin 4 - в официальных репозиториях**
- **OmniDB (2nd Quadrant) https://github.com/OmniDB/OmniDB**
- **DBeaver https://dbeaver.io/**

Основной параметр системы

У большинства утилит сквозные имена параметров, многие из них дублируются в переменных окружения.

Главным параметром является -D или переменная окружения PGDATA, которая указывает на корень дерева файлов PG, от которого рассчитывается положение остальных файлов. Таким образом можно без проблем работать с несколькими инстансами СУБД на одной машине.

Никогда не держите в этом каталоге лишних файлов, все файлы должны быть читаемы и записываемы для самого postgres. Никаких бекапов конфигов, дампов и прочего мусора!

https://www.postgresql.org/docs/current/static/libpq-envars.html
https://www.postgresql.org/docs/current/static/app-postgres.html

Инициализация базы

Как и в случае mysql, перед использованием СУБД надо инициализировать базы.

В процессе инициализации происходит

- **инициализация дерева файлов**
- **создание базовых таблиц template0 и template1**

`template0` - read only база данных, содержащая инициализационный набор данных
`template1` - база-шаблон для создания новых баз

До версии 8.4 основной параметр для инициализации был - установка кодировки. Начиная с 8.4 кодировка может быть установлена для отдельной базы

```
su postgres -c '/usr/pgsql-10/bin/initdb \
-E UTF8 --locale ru_RU.UTF-8 \
-D /var/lib/pgsql/10/data'
```

Подключение к базе. pg_hba.conf

Основная утилита для работы с базой - psql

Вы не можете, как в mysql подключиться к СУБД, или как в ORACLE к TNS Listener, вы всегда подключаетесь к базе. Все системные таблицы и функции всегда доступны из любой базы.

Если еще не решили к какой - то подключайтесь к template1

pg_hba.conf задаёт способ доступа к базам и к репликации из различных источников

trust - не проверять доступ
md5 - проверка по логину-паролю
peer - сопоставление с пользователем системы (доступно только для локальных подключений)

Кроме того, поддерживается pam, ldap, ssl cert, radius и еще много схем
Одно НО: пользователь должен существовать в postgresql

Подключение к базе. pg_hba.conf

Не смотря на то, что по-умолчанию для локальных соединений используется метод peer, это всё-таки не очень безопасно.

**Рекомендую и локальные соединения переводить на логин-пароль.\
Можно использовать файл `pgpass` и переменную окружения PGUSER для автоматического входа в систему.**

`pgpass` - это табличка, из которой по хосту и пользователю выбирается пароль и база. Может лежать в home dir в виде `.pgpass`

По историческим оракловым причинам, считается, что имя базы связано с именем пользователя (а в oracle в принципе пользователей не существует. Существует схема, пароль к схеме и доступ из одной схемы к другой схеме)

Еще немного ораклового

Так же, как и в Oracle, все манипуляции с базой (DML) делаются через системные таблицы и обвязывающие их процедуры и view. Тут нет ни show databases, ни show create table. Но и DML, при этом, транзакционный - можно откатить транзакцию, создающую базу, например.

В psql есть алиасы, упрощающие работу:

`\l` - список баз
`\dt` - список таблиц
`\d <table>` - описание таблицы

И вообще - `\?`

И еще немного особенностей

В postgresql строка, заключенная в двойные кавычки - всегда отсылка к именам системных объектов (поля, таблицы)
Строка в одинарных кавычках - строковая константа

```
select * from "employee" where "employee_name"='elina';
```

Организация файловой системы

Директория продукта:
`/usr/pgsql-<version>`
Клиентские утилиты вынесены симлинками в /usr/bin

Основная директория - домашняя для пользователя postgres
`/var/lib/pgsql`

Директория текущей версии

`/var/lib/pgsql/<version>/data` - PGDATA для текущей версии

Дело в том, что обновление версии в пределах одного мажора делается с минимальным downtime с помощью утилиты `pg_upgrade`
Обновление мажорных версий рекомендуется делать через полный dump/restore, для чего рядом ставится новая версия с отдельной иерархией

Ахиллесова пята (левая)

На каждое соединение запускается отдельный процесс (backend)
С другими процессами он общается через shared memory
В некоторых случаях спасает дополнительный софт - pgpool

Ахиллесова пята (правая)

MVCC организован таким образом (в отличие от других РСУБД), что записи не меняются и не удаляются. У каждого тупла (tuple) есть системные поля xmin (id транзакции, создавшей запись) и xmax (id транзакции изменившей или удалившей запись), таким образом создаётся некоторое окно видимости.

1. **Для удаления записей существует процесс autovacuum**
2. **Долговисящие транзакции мешают vacuum process, и база разрастается**

Ахиллесова пята (средняя)

**в простонародии - wraparound, или обнуление счетчика транзакций.
По условиям, не должно существовать в системе транзакций с номерами, отстающими на более, чем maxint32/2-1. При несоблюдении этого условия база останавливается. Решение - postgres pro, с 64-битными транзакциями**

Параметры системы. GUC.

GUC - Global Unified Configuration, подсистема конфигурации Postgres, которая собирает все источники конфигурации воедино, отображая её в системной таблице pg_settings.

Параметры могут быть заданы в:

- **параметрах командной строки основного демона postgres (`-c name=value`)**
- **переменных окружения (PGDATA, etc)**
- **postgres.conf**
- **postgres.auto.conf (результат работы оператора ALTER SYSTEM, 9.4+)**

В pg_settings можно увидеть откуда параметр применился, какие значения по-умолчанию, можно ли поменять на лету, в каких единицах считается и краткое объяснение. Очень удобно.

Как postgres обеспечивает надёжность

...и вообще общается с дисками

Все операции всегда выполняются внутри транзакции, явной или неявной.
Бекенд (backend) изменяет данные в страницах shared memory. Но перед тем как сказать клиенту, что транзакция прошла успешно, все измененные страницы записываются как есть в WAL (write ahead log, pg_xlog (<10) или pg_wal (10+)) иногда, но редко, бекенд сам что-то пишет в таблицы.

Процесс bgwriter время от времени проходится по shared memory и сбрасывает данные в таблицы, помечая сохраненные страницы, как "чистые"

Процесс checkpointer запоминает минимальный id текущих транзакций и начинает сбрасывать данные в файлы. По завершению процедуры номер транзакции попадает в т.н. control data. Чекпоинт инициируется по времени, либо вручную командой CHECKPOINT.

При восстановления после сбоя номер транзакции из control data (чекпоинт) ищется в WAL (в имени WAL закодирован диапазон XID, что бы быстрее искать), и начиная с этой записи WAL накладывается на файлы базы. Т.о. получаем консистентность.

Посмотреть, что еще интересного в control data:
```
/usr/pgsql-10/bin/pg_controldata -D /var/lib/pgsql/10/data
```

Основные параметры

которые вы должны сразу настроить

`shared_buffers` - От 1/8 до 1/2 всей памяти

`effective_cache_size` - На какой объем системного кеша может рассчитывать планировщик запросов.

`RAM - work_mem * max_connections - shared_buffers`

`max_connections` - максимальное число подключений, каждое из которых может расходовать

`work_mem` - память для сессии, 32..128M. Это рекомендательное значение для планировщика, а не жесткое ограничение.

`maintenance_work_mem` - память для checkpointer, autovacuum, etc. 1/32..1/16 RAM или 256MB..4GB

`temp_buffers` - для каждой сессии под временные таблицы, 32Mb

`efective_io_concurrency` - 1 - один диск, 2 - RAID10

`random_page_cost` - 1.5..2 для RAID10, 1.1..1.3 - для SSD

`fsync` и `autovacuum` ДОЛЖНЫ БЫТЬ ВКЛЮЧЕНЫ ВСЕГДА!!!

Backup

Бекап постгреса - файловый
Процедура с точки зрения postgres выглядит следующим образом:

- **`pg_start_backup('label')` - создание чекпоинта, запись состояния и даты бекапа**
- **копирование файлов**
- **`pg_stop_backup()` - создание нового сегмента wal**
- **копирование WAL**

Проблема: WAL может отротироваться. Решение: откладывать логи заранее с помощью настройки `archive_command`

Восстановление из бекапа

Процедура восстановления:

- **разворачивание файлового бекапа**
- **создание recovery.conf в каталоге PGDATA, в котором**
  - **restore_command - откуда брать файлы. Файлы можно положить сразу.**
  - **archive_cleanup_command - что делать с "отработанными файлами", есть штатная утилита pg_archivecleanup, но можно и своё вписать**

https://habr.com/post/188096/

Репликация

Представляет частный случай восстановления из бекапа. Отличается тем, что WAL файлы передаются по сети процессом wal_sender, postgres не останавливает процесс восстановления.

Репликация делает каталоги двух баз идентичными, т.о. нельзя отреплицировать одну базу.

- **Готовим мастер:**
  - **wal_level устанавливаем в replica или hot_standby,**
  - **wal_senders - в число реплик**
  - **wal_keep_segments - число сохраняемых WAL-файлов на случай сбоя или отставания реплики**
  - **создаем пользователя**
    ```
    create user repluser with replication encrypted password '123'
    ```
  - **в pg_hba.conf добавляем разрешение на репликацию**
    ```
    host replication repluser 192.168.100.2/32 md5
    ```
- **Делаем бекап и разворачиваем его на второй машине**
- **Настраиваем recovery.conf**
  - **standby_mode = 'on'**
  - **primary_conninfo = 'host=192.168.100.2 port=5432 user=postgres'**

Репликация. Hot Standby.

По умолчанию реплика делаетя в режиме standby, т.е. режим чистого восстановления, которое не позволяет подключаться к базе. Для того, что бы можно было читать со слейва, существует режим Hot Standby

На мастере в postgresql.conf:
wal_level = hot_standby

На слейве в recovery.conf:
hot_standby = on

Репликация. Переключение реплики.

Для создания автоматических систем переключения ролей, есть два инструмента:

trigger_file в recovery.conf
pg_ctl promote

Оба этих механизма приводят к отключению от мастера, приведению себя в консистентное состояние и разрешение записи в базу.

При этом postgresql переключается на новый timeline

Репликация. Статус.

```
testdb=# select * from pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
pid              | 21987
usesysid         | 16384
usename          | replica
application_name | walreceiver
client_addr      | 192.168.1.108
client_hostname  |
client_port      | 56674
backend_start    | 2014-11-25 18:30:09.206434+03
backend_xmin     |
state            | streaming
sent_location    | 0/5A2D8E60
write_location   | 0/5A2D8E60
flush_location   | 0/5A2D8E60
replay_location  | 0/5A2D8E60
sync_priority    | 0
sync_state       | async
```

Репликация. Слоты.

Если реплика отстаёт, то её можно потерять навсегда из-за того, что сервер отротировал WAL. Для того, что бы этого не происходило создали механизм обратной связи слейва с мастером - replication slots

Создаются они на мастере командой

```
SELECT
pg_create_physical_replication_slot('standby_slot');
```

На слейве в recovery.conf добавляется строчка

```
primary_slot_name = 'standby_slot'
```

А теперь всё вместе. pg_basebackup

Утилита для создания консистентного бекапа. Умеет сама забирать нужные файлы с сервера (поэтому надо каталог PGDATA держать чистым!) и забирать WAL по протоколу репликации со слотами.

Поэтому в самой базовой конфигурации сервера postgres всегда надо сразу конфигурировать мастер на репликацию.

Бонус. CB/PITR

Continuous Backup / Point In Time Recovery

Существует возможность восстановить систему в любое состояние.

Для этого
Организуется Continuous Backup:
вариант 1: с помощью acrhive_command откладывать логи
вариант 2: с помощью pg_receivewal (pg_receivexlog до 10) сделать слейв

Можно вручную задать точку восстановления функцией
pg_create_restore_point('label_name')

После чего в recovery.conf прописывается один из вариантов настройки recovery_target:
recovery_target = 'immediate' - восстанавливаемся до чекпоинта и всё
recovery_target_name = 'label' - восстанавливаемся до метки из pg_start_backup() или pg_create_restore_point()
recovery_target_time = <время> - восстановление на момент во времени
recovery_target_xid = восстановление до определенной транзакции
recovery_target_lsn = восстановление до определенного номера лога

https://www.postgresql.org/docs/9.4/static/functions-admin.html
https://www.postgresql.org/docs/current/static/recovery-target-settings.html

Домашнее задание.

На данный момент один из вариантов бекапа Postgres является Barman. Попробуйте настроить бекапы с её помощью.

https://www.pgbarman.org/