# Getting By With Just psql

PgConf EU - Warsaw
October 2017
Corey Huinker

# Why Use Only psql?

- Restricted Toolchain
  - Training/Maintenance Considerations
  - Regulatory or Auditing Restrictions
  - Security Concerns
  - Container Limitations
  - Installation Hassles
- Obfuscation
  - Application language may only add clutter to code
  - Database access layer may add more heat than light (positional rather than named placeholders, etc)
- Logging for Auditing
  - modes to show the query that was run with all of the positional variables filled out
  - success/failure and row counts printed by default
  - timings are printed (in milliseconds but also in human readable times in v10)
- Features available in newer versions of psql will work when connected to earlier server versions and postgres-ish databases (Vertica, Redshift).

# \ ("slash") commands

- Are psql commands
- Are *never* sent by psql to the server
- Have no meaning to postgres itself
- Have no meaning in other programming languages, unless that language is copying psql
- All operations of psql can be done with \ commands
  - connecting to a database
  - changing format output
  - sending SQL commands to a server
  - changing output location
- Anything that is not a slash command or a buffer terminator (';') is accumulated in a buffer to be sent to the server at a later time
- Many operations can be done with command-line switches as well to set initial state

# Variables

- Available in all supported postgresql versions
- Set on the command line via -v or --set or the \-comands \set and \gset
- Are string type
- Can be used as a simple macro replacement (:var), a quote-safe string literal (:'var') or a quote-safe SQL identifier (:"var), to avoid SQL-injection risks.

```
$ psql test --quiet --set message="The farmer's cow says \"Moo\""
test=# \echo :message
The farmer's cow says "Moo"
test=# \echo :'message'
'The farmer''s cow says "Moo"'
test=# \echo :"message"
"The farmer's cow says ""Moo"""
```

- Undefined variables are not macro-expanded in any way

```
$ psql test --quiet
test=# \echo :some_var :'some_var' :"some_var"
:some_var :'some_var' :"some_var"
```

# Setting Variables - \set

- Available in all supported versions
- Can invoke OS-level commands and environment variables

```
test=# \set yes_please `yes | head -n 1`
test=# \echo :yes_please
Y
test=# \set path `echo $PATH | cut -d ':' -f 1`
test=# \echo :path
/home/corey/bin
```

- Does concatenation without spaces

```
test=# \set xvar x
test=# \set yvar y
test=# \set alphabet :xvar :yvar z
test=# \echo :alphabet
xyz
```

# Using Variables - Sanitizing Input

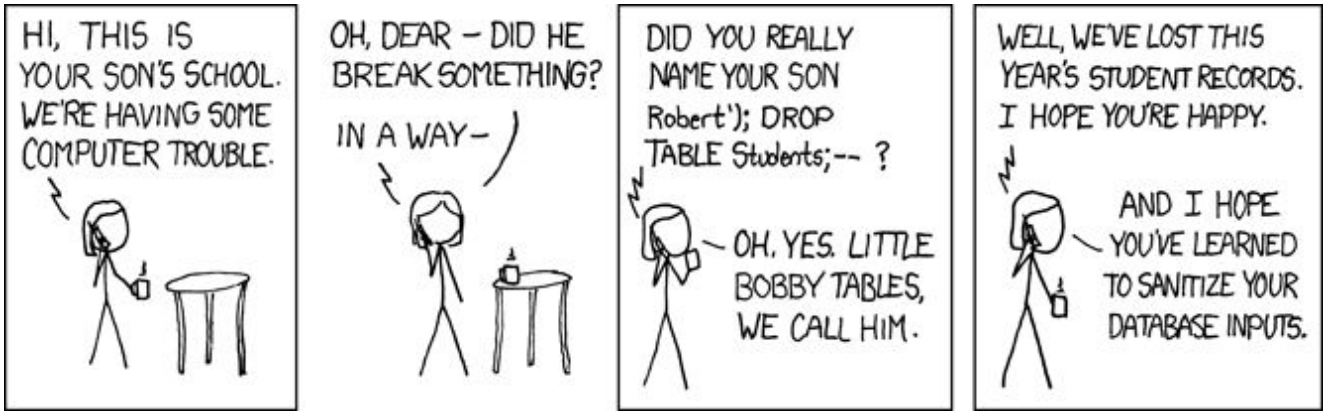
```
$ psql test --set os_user=$( whoami )
test=# CREATE TEMPORARY TABLE user_log (username text);
CREATE TABLE
test=# INSERT INTO user_log(username) VALUES(:'os_user');
INSERT 0 1
test=# SELECT * FROM user_log;
 username
----------
 corey
(1 row)

test=# SELECT count(*) FROM user_log WHERE username = :'os_user';
 count
-------
     1
(1 row)
```

# Using Variables - SQL Construction

```
test=# \set temp_tab_name user_log_partition_ :os_user
test=# CREATE TEMPORARY TABLE :"temp_tab_name" AS
     SELECT * FROM user_log WHERE username = :'os_user';
SELECT 1
test=# \d user_log_partition_corey
Table "pg_temp_2.user_log_partition_corey"
  Column  | Type | Modifiers
----------+------+-----------
 username | text |
```

Use un-sanitized variables in SQL with extreme caution!




https://xkcd.com/327/

# Setting Variables - \gset

- New in 9.3 (thanks, Pavel!)
- Captures columns of a one-row result set

```
test=# select 'a' as avar \gset
test=# \echo :avar
a
```

- Multi-row results sets are a *psql* error and will set no values (not a DB-error)

```
test=# select 'b' as avar from generate_series(1,10) \gset
more than one row returned for \gset
test=# \echo :avar
a
```

- Variable names can be prefixed

```
test=# select 'a' as avar \gset prefix_
test=# \echo :prefix_avar
a
```

# Setting Variables - \gset

- Beware of name clashes, last (rightmost) column wins

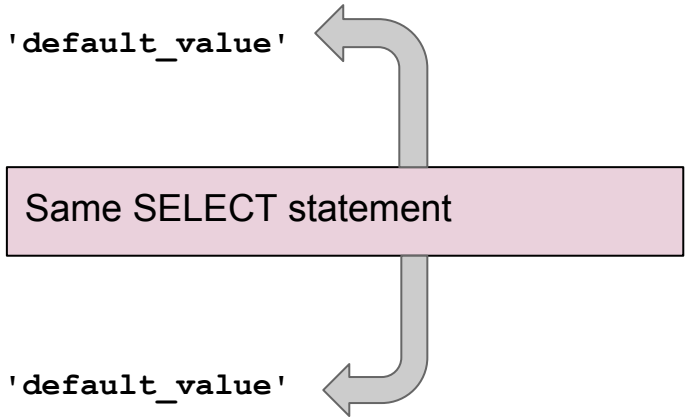
```
test=# select 'a' as avar, 'b' as avar \gset prefix_
test=# \echo :prefix_avar
b
```

- NULL results *un-set* the variable, which is different from \set
- \set doesn't know about NULL, thinks it's the string 'NULL'

```
test=# \set avar a
test=# \echo :avar
a
test=# SELECT NULL as avar \gset
test=# \echo :avar
:avar
test=# \set avar NULL
test=# \echo :avar
NULL
```

# Ugly Hack: Defaults for Variables

```
test=# \set foo abc
test=# \set test_foo :foo
test=# SELECT  CASE
test-#             WHEN  :'test_foo' = ':foo' THEN 'default_value'
test-#             ELSE :'test_foo'
test-#         END AS foo
test-# \gset
test=# \echo :foo
abc
test=# \unset foo
test=# \set test_foo :foo
test=# SELECT  CASE
test-#             WHEN  :'test_foo' = ':foo' THEN 'default_value'
test-#             ELSE :'test_foo'
test-#         END AS foo
test-# \gset
test=# \echo :foo
default_value
```



# Data Structures: Temporary Tables

- Allows for actual data types whereas psql variables are only ever strings
- can do validation with queries and applied check constraints
- can import data through INSERT statements and \copy statements
- can capture data from complex commands via \copy and FROM PROGRAM

```
test=# CREATE TEMPORARY TABLE etc_pwd (uname text, pwd text, uid integer, gid integer,
fullname text, homedir text, shell text);
CREATE TABLE
test=# \copy etc_pwd FROM PROGRAM 'head -n 4 /etc/passwd' (DELIMITER ':')
COPY 4
test=# select * from etc_pwd;
 uname  | pwd | uid | gid | fullname |  homedir  |       shell
--------+-----+-----+-----+----------+-----------+-------------------
 root   | x   |   0 |   0 | root     | /root     | /bin/bash
 daemon | x   |   1 |   1 | daemon   | /usr/sbin | /usr/sbin/nologin
 bin    | x   |   2 |   2 | bin      | /bin      | /usr/sbin/nologin
 sys    | x   |   3 |   3 | sys      | /dev      | /usr/sbin/nologin
(4 rows)
```

# Pushing Data

- COPY TO PROGRAM launches program on server - which might not have the program
- \COPY ... TO PROGRAM uses local client environment
- Allows you to maintain control within psql rather than terminating and passing control back to bash

```
test=# \copy (SELECT * FROM etc_pwd) TO PROGRAM
          'gzip | s3_archive.sh s3://mybucket/pwd_log.gz'
COPY 4
uploaded to s3://mybucket/pwd_log.gz
```

# Pushing Data Alternative: \g

- sends output to a file ( \g filename.txt )
- or a program ( \g | program.sh )
- will attempt default psql formatting unless you set it otherwise
- useful when the "postgres" database isn't actually "postgres" (vertica, redshift, etc)

```
test=# \pset format unaligned
Output format is unaligned.
test=# \pset border 0
Border style is 0.
test=# \pset fieldsep '\t'
Field separator is "     ".
test=# SELECT * FROM etc_pwd \g | gzip > output.txt.gz
test=# \! zcat output.txt.gz
uname     pwd  uid  gid  fullname  homedir   shell
root x    0    0    root /root     /bin/bash
daemon    x    1    1    daemon    /usr/sbin /usr/sbin/nologin
bin  x    2    2    bin  /bin /usr/sbin/nologin
sys  x    3    3    sys  /dev /usr/sbin/nologin
(4 rows)
```

# Metaprogramming: \gexec

- New in 9.6
- Interprets all non-null results in a result set to themselves be SQL statements to be immediately sent to the server for execution in order of arrival (top row first, left to right within a row
- Statements generated can be DML or DDL
- Must be SQL, not psql \-commands
- Normal Error Stop variables are in effect
- No minimum number of rows returned
- Can be used as a primitive finite loop construct
- Whole result set is generated before any result queries are executed

# Metaprogramming: \gexec

```
test=# CREATE TEMPORARY TABLE t (a integer, b integer, c integer);
CREATE TABLE
test=# SELECT format('CREATE INDEX ON t(%I)', attname)
test-# FROM pg_attribute
test-# WHERE attnum > 0
test-# AND attrelid = 't'::regclass
test-# \gexec
CREATE INDEX
CREATE INDEX
CREATE INDEX
test=# \d+ t
                                    Table "pg_temp_3.t"
 Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------+-------------
 a      | integer |           |          |         | plain   |              |
 b      | integer |           |          |         | plain   |              |
 c      | integer |           |          |         | plain   |              |
Indexes:
    "t_a_idx" btree (a)
    "t_b_idx" btree (b)
    "t_c_idx" btree (c)
```

# \gexec: Rebuild Indexes

```
test=# SELECT  'BEGIN'
test-# UNION ALL
test-# SELECT  format('DROP INDEX %s', indexrelid::regclass::text)
test-# FROM pg_index
test-# WHERE indrelid = 't'::regclass
test-# UNION ALL
test-# SELECT  'INSERT INTO t SELECT a.a, a.a % 10, a.a % 100 FROM generate_series(1,1000000) as a(a)'
test-# UNION ALL
test-# SELECT  pg_get_indexdef(indexrelid)
test-# FROM pg_index
test-# WHERE indrelid = 't'::regclass
test-# UNION ALL
test-# SELECT  'COMMIT'
test-# \gexec
BEGIN
DROP INDEX
DROP INDEX
DROP INDEX
INSERT 0 1000000
CREATE INDEX
CREATE INDEX
CREATE INDEX
COMMIT
```

# \gexec: Avoid Losing Indexes on Fail

```
test=# BEGIN;
BEGIN
test=# SELECT  format('DROP INDEX %s', indexrelid::regclass::text) FROM pg_index
test-# WHERE   indrelid = 't'::regclass UNION ALL
test-# SELECT  'SELECT 1 / 0' UNION ALL
test-# SELECT  pg_get_indexdef(indexrelid) FROM    pg_index
test-# WHERE   indrelid = 't'::regclass \gexec
DROP INDEX
DROP INDEX
DROP INDEX
ERROR:  division by zero
ERROR:  current transaction is aborted, commands ignored until end of transaction block
ERROR:  current transaction is aborted, commands ignored until end of transaction block
ERROR:  current transaction is aborted, commands ignored until end of transaction block
test=# COMMIT;
ROLLBACK
test=# \d t
               Table "pg_temp_3.t"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 a      | integer |           |          |
 b      | integer |           |          |
 c      | integer |           |          |
Indexes:
    "t_a_idx" btree (a)
    "t_b_idx" btree (b)
    "t_c_idx" btree (c)
```

Swap the INSERT statement for a statement guaranteed to fail

# Conditionals: Prior to Version 10

- It was do-able...sort of:

```
test=# CREATE TEMPORARY TABLE my_table AS SELECT 1 as x;
SELECT 1
test=#
test=# SELECT  CASE
test-#             WHEN EXISTS(SELECT NULL FROM my_table)
test-#             THEN '\echo not empty \q'
test-#             ELSE 'DROP TABLE my_table;'
test-#         END AS cmd
test-# \gset
test=# :cmd
not empty
```

- Not clear
- Not very expressive
- Very hard to do multiple statements
- Probably impossible to do nested conditionals
- These sorts of very minor branching issues often forced programmers to use an application language

# Conditionals: New in Version 10

- `\if, \elif, \else, \endif`
- `\if` and `\elif` take one token and evaluate it for psql-boolean truth
  `true, false, 1, 0, on, off, yes, no`
  ...or any unambiguous case insensitive leading substrings of one of those

```
test=# \if tR \echo good enough \endif
good enough
```

- Any other values raise a warning and are treated as false

```
test=# \if 42 \echo good enough \endif
unrecognized value "42" for "\if expression": Boolean expected
\echo command ignored; use \endif or Ctrl-C to exit current \if block
test=#
```

- Cannot (yet) do more complex expression evaluation

# Queries with \gset: Decider Of Truth!

```
test=# SELECT EXISTS (SELECT NULL FROM pg_class
                      WHERE relname = 'my_table') as my_table_exists
test-# \gset
test=# \if :my_table_exists
test@# drop table my_table;
query ignored; use \endif or Ctrl-C to exit current \if block
test-# \endif
```

Pros:

- Full expressiveness of SQL in determining complex truth
- Much of what you wanted to know is in the database anyway

Cons:

- Database roundtrip
- Clutters logs with trivial math equations (example: `SELECT 3 > 4`)
- psql might not be connected to a database at the moment

# Modularity With Includes - Module

```
$ cat move_to_archive.sql
-- requires variable "table_name" to be defined
BEGIN;
-- sanitize table_name and ensure existence of destination table
SELECT  :'table_name'::regclass::text as src_table_name,
        format('%s', :'table_name' || '_archive')::regclass::text as dest_table_name,
        CURRENT_TIMESTAMP - INTERVAL '7 days' as low_water_mark
\gset
WITH del as (
    DELETE FROM :src_table_name
    WHERE created < :'low_water_mark'::timestamptz
    RETURNING * )
INSERT
INTO    :dest_table_name
SELECT  *
FROM    del;
COMMIT;
```

# Modularity With Includes - Usage

```
test=# CREATE TEMPORARY TABLE yep ( x integer, created timestamptz default
current_timestamp);
CREATE TABLE
test=# INSERT INTO yep(x) SELECT  r.r FROM generate_series(1,10000) as r(r);
INSERT 0 10000
test=# CREATE TEMPORARY TABLE yep_archive AS SELECT * FROM yep WHERE false;
SELECT 0
test=# \ir move_to_archive.sql
```

- Sub-script handles un-set variables in a non-destructive way

```
BEGIN
psql:move_to_archive.sql:12: ERROR:  syntax error at or near ":"
LINE 1: SELECT  :'table_name'::regclass::text as src_table_name,
                ^
psql:move_to_archive.sql:22: ERROR:  syntax error at or near ":"
LINE 2:     DELETE FROM :src_table_name
                        ^
ROLLBACK
```

# Modularity With Includes - Usage

- When used correctly, it just works.

```
test=# \set table_name yep
test=# \ir move_to_archive.sql
BEGIN
INSERT 0 0
COMMIT
```

- But even when called correctly, only does work that makes sense.

```
test=# CREATE TEMPORARY TABLE nope AS SELECT * FROM yep;
SELECT 10000
test=# \set table_name nope
test=# \ir move_to_archive.sql
BEGIN
psql:move_to_archive.sql:12: ERROR:  relation "nope_archive" does not exist
psql:move_to_archive.sql:22: ERROR:  current transaction is aborted, commands ignored
until end of transaction block
ROLLBACK
```

# Modularity With Includes

Pros:

- Can do transactions whereas a DO block cannot

Cons:

- Cannot nest transactions. All transaction code must either be in *every* included module or entirely in the calling program.
- Where do you put the GRANT statements?
- Keeping code generic enough to be useful multiple schemas, multiple databases.
- Underlying OS requirements (i.e. is that extension installed) are out of psql's control.

# Looping

A hard problem

- psql interprets commands "on the fly", if there was a looping construct it would have to remember where the loop started.
- Would have to ensure proper nesting of if/then/else blocks within loop constructs
- Difficult to communicate to an interactive user where they are inside loops and blocks
- Code that was part of the loop construct on one iteration might not on the next:

```
\set x 1
\set continue_loop true
\set weird_command '\endwhile'
\while :continue_loop
SELECT :x + 1 as x, (:x > 1) as include_file \gset
\if :include_file
   \ir some_other_file.sql
   :weird_command
\endif
\endwhile
```

# Looping With Recursion

```
$ cat recursion_test.sql
\echo :x
SELECT :x + 1 as x, (:x > :y) as exit_now \gset
\if :exit_now
    \q
\endif
\ir recursion_test.sql

$ psql test -f recursion_test.sql --set x=1 --set y=2000
...
1017
1018
1019
psql:recursion_test.sql:6: recursion_test.sql: Too many open files
```

- You can raise the file limit, but in my test I got a segfault at 5291

# Future Directions:

- `Real expressions for \if, \elif, \set`
- `\gdesc`
- Testing for variable definition with {?var}
- Test-able server version numbers (good for install scripts)
- Making the postgres regression tests more robust without switching to PgTAP

# Questions?