# Getting on a hook
# or PostgreSQL extensibility

Alexey Kondratov

Postgres Professional

PostgreSQL @ FOSDEM’21, February 6-7

# PostgreSQL extensibility

- Custom types, operators.
- Access methods.
- PL/pgSQL scripting language.
- Functions, triggers, extensions and so on.
- …

A lot of info in the docs [1].

[1] https://www.postgresql.org/docs/current/extend.html

# PostgreSQL extensibility

- Custom types, operators.
- Access methods.
- PL/pgSQL scripting language.
- Functions, triggers, extensions and so on.
- …
- **Hooks** (*and* ***callbacks***).

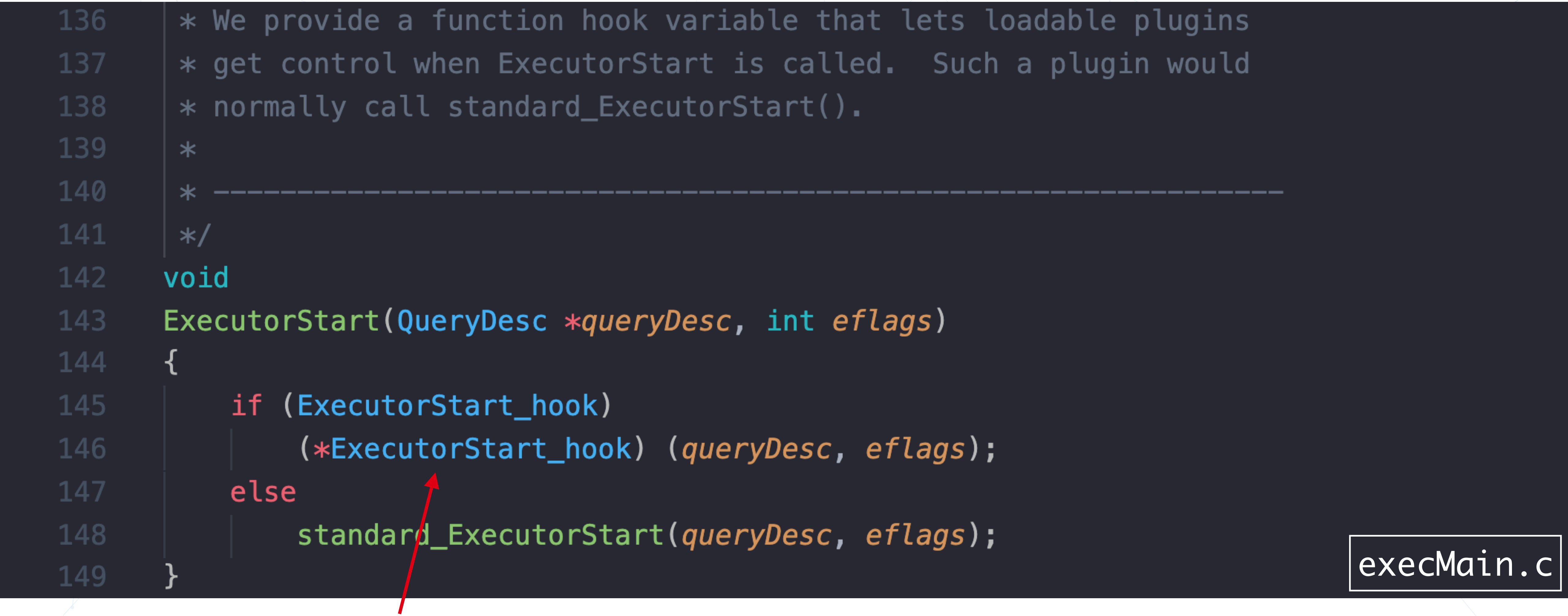
A lot of info in the docs [1].

[1] https://www.postgresql.org/docs/current/extend.html

# What is a hook?

- Function or more precisely **a global pointer to a function**.
- Being defined it will be called by PostgreSQL at some specific moment.
- Scattered all over the PostgreSQL core.
- Extensions (shared libraries) can set these hooks to peek into the PostgreSQL internal state.

# Hooks: pointer

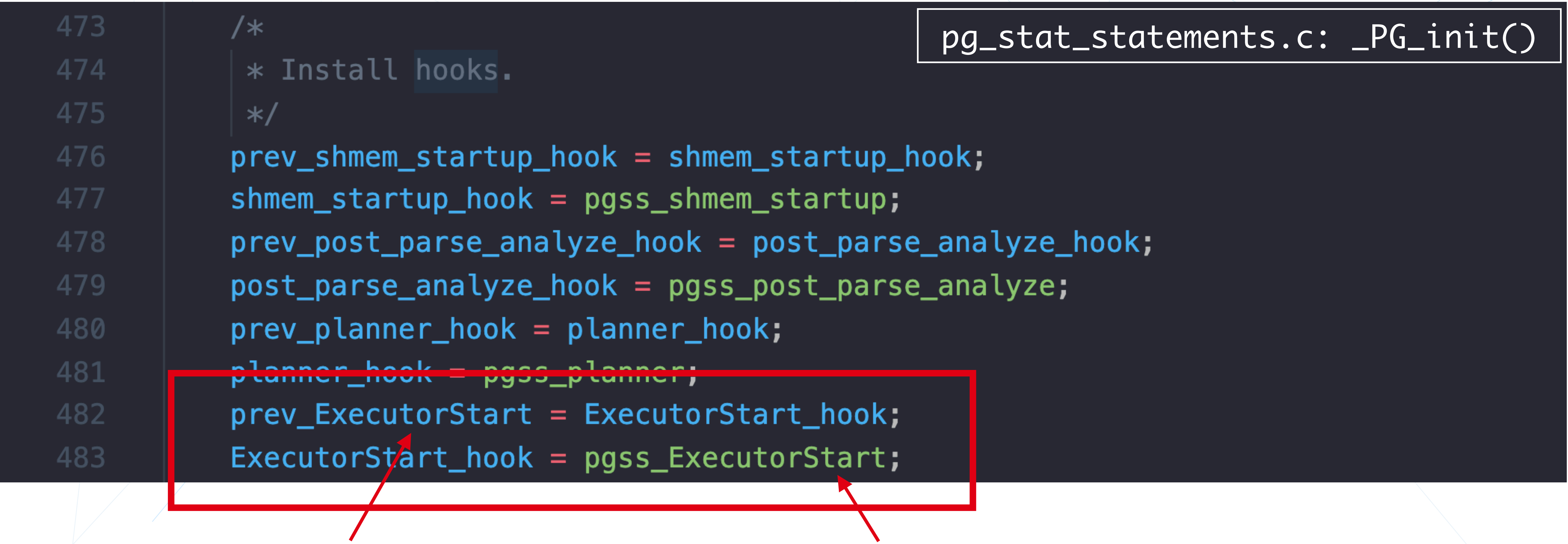


```
 * We provide a function hook variable that lets loadable plugins
 * get control when ExecutorStart is called.  Such a plugin would
 * normally call standard_ExecutorStart().
 *
 * ----------------------------------------------------------------
 */
void
ExecutorStart(QueryDesc *queryDesc, int eflags)
{
    if (ExecutorStart_hook)
        (*ExecutorStart_hook) (queryDesc, eflags);
    else
        standard_ExecutorStart(queryDesc, eflags);
}
```

execMain.c

Executed if defined

# Hooks: installation



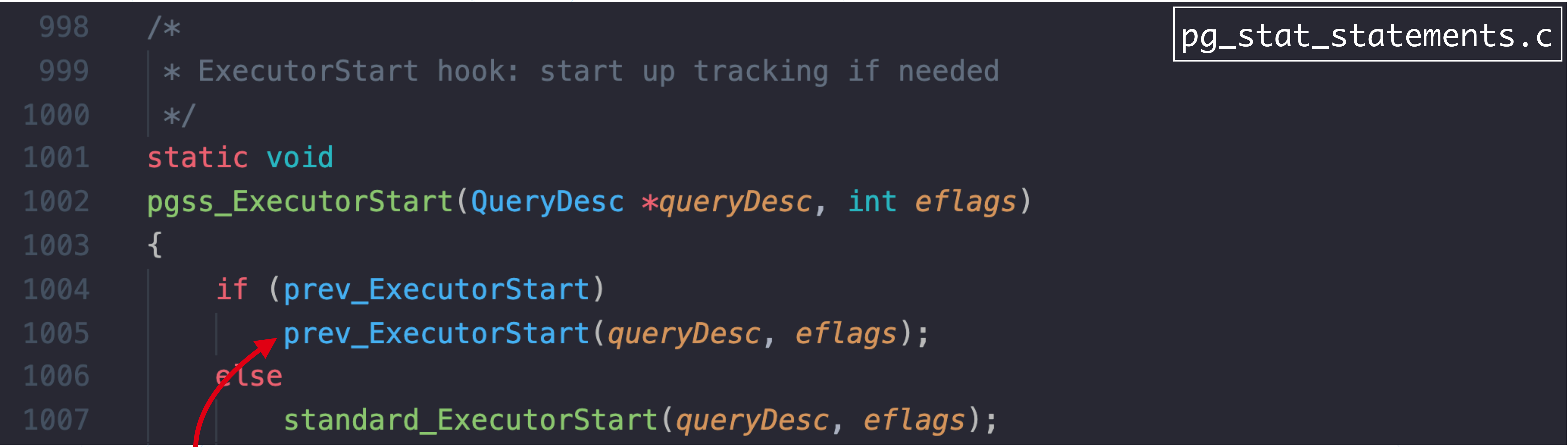
pg_stat_statements.c: _PG_init()

```
/*
 * Install hooks.
 */
prev_shmem_startup_hook = shmem_startup_hook;
shmem_startup_hook = pgss_shmem_startup;
prev_post_parse_analyze_hook = post_parse_analyze_hook;
post_parse_analyze_hook = pgss_post_parse_analyze;
prev_planner_hook = planner_hook;
planner_hook = pgss_planner;
prev_ExecutorStart = ExecutorStart_hook;
ExecutorStart_hook = pgss_ExecutorStart;
```

1) Remember previously defined value

2) Register your own function

# Hooks: design

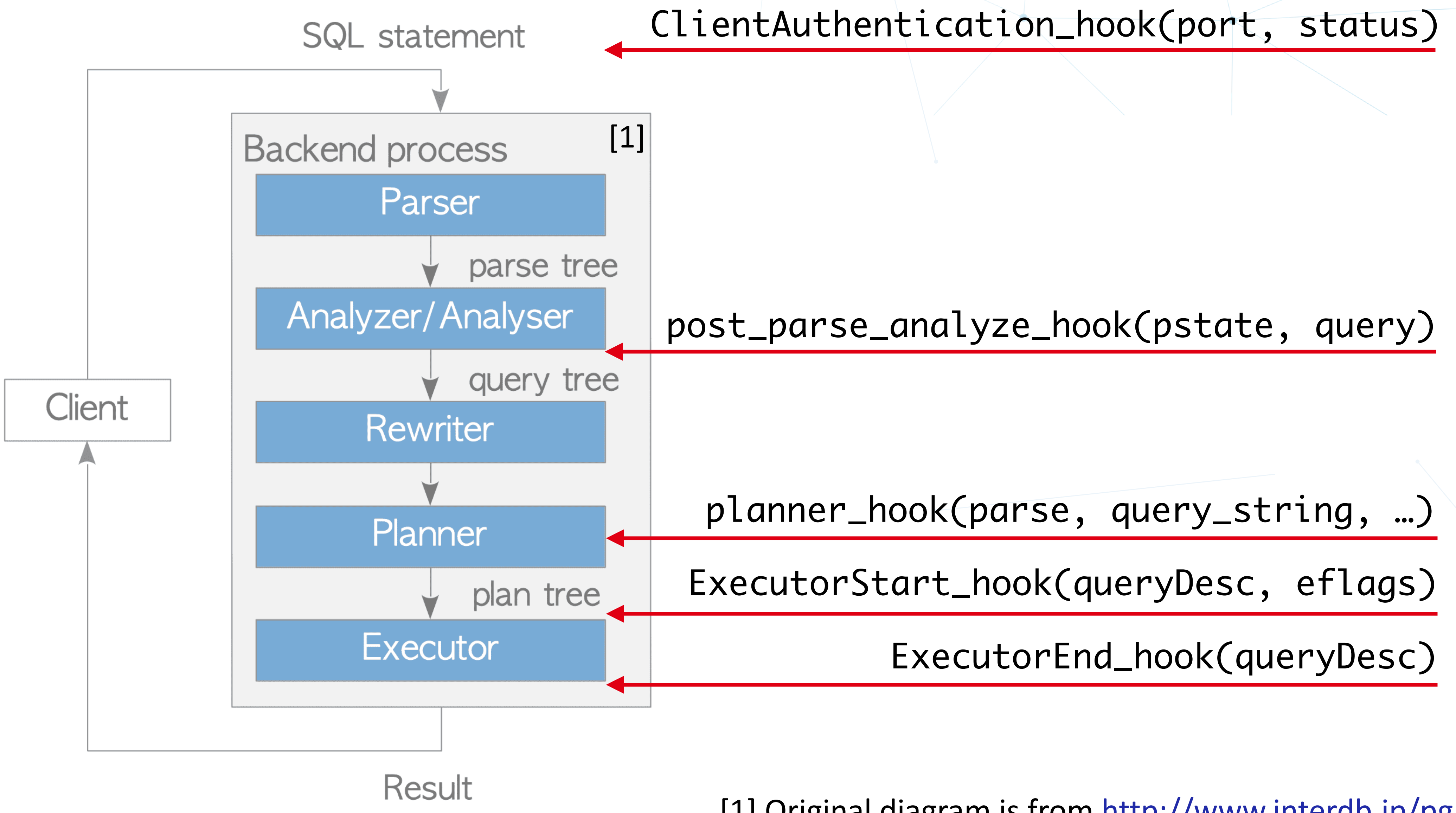


pg_stat_statements.c

```c
/*
 * ExecutorStart hook: start up tracking if needed
 */
static void
pgss_ExecutorStart(QueryDesc *queryDesc, int eflags)
{
	if (prev_ExecutorStart)
		prev_ExecutorStart(queryDesc, eflags);
	else
		standard_ExecutorStart(queryDesc, eflags);
```

Do not forget to call your predecessor!

# Hooks: overview



ClientAuthentication_hook(port, status)

post_parse_analyze_hook(pstate, query)

planner_hook(parse, query_string, …)

ExecutorStart_hook(queryDesc, eflags)

ExecutorEnd_hook(queryDesc)

[1] Original diagram is from http://www.interdb.jp/pg/pgsql03.html.

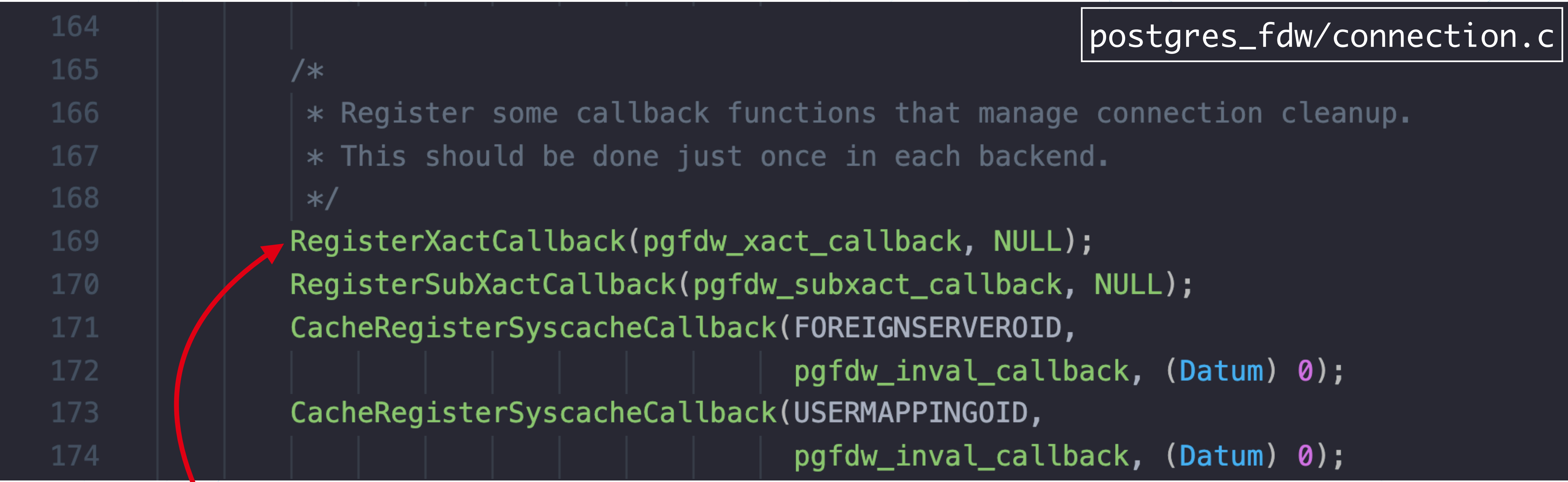
# Hooks: unofficial documentation

- GitHub repo: https://github.com/AmatanHead/psql-hooks
  - Lists hook arguments.
  - Has text description.
- pgPedia: https://pgpedia.info/h/hooks.html
  - There is an interesting change history with commit reference per hook.
- A bit outdated Guillaume Lelarge's slides from PGCon 2012.

# What is a callback?

- Very similar to the hooks.
- But initially designed to be set by multiple users.
- Usually installed by Register*Callback() setter functions: `RegisterXactCallback()`, `RegisterSubXactCallback()`, `RegisterExprContextCallback()`, etc.
- Yet, there are others like: `before_shmem_exit()`, `on_shmem_exit()`.
- Mostly for internal usage.

# Callbacks: registration

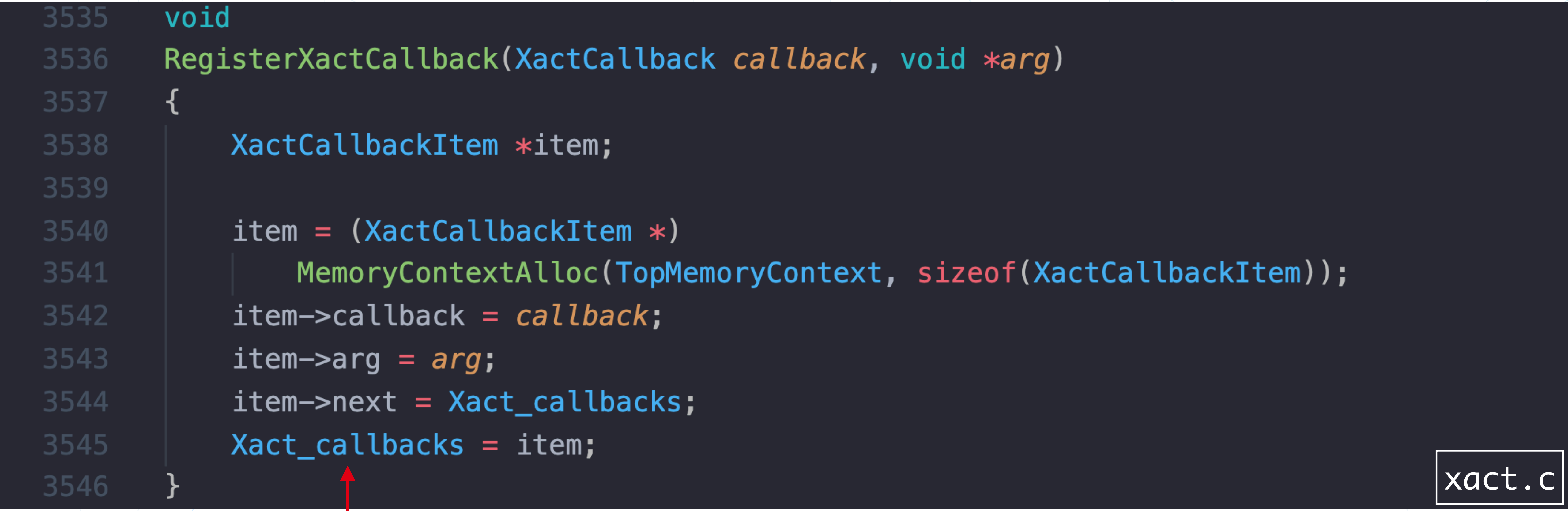


postgres_fdw/connection.c

```c

/*
 * Register some callback functions that manage connection cleanup.
 * This should be done just once in each backend.
 */
RegisterXactCallback(pgfdw_xact_callback, NULL);
RegisterSubXactCallback(pgfdw_subxact_callback, NULL);
CacheRegisterSyscacheCallback(FOREIGNSERVEROID,
                              pgfdw_inval_callback, (Datum) 0);
CacheRegisterSyscacheCallback(USERMAPPINGOID,
                              pgfdw_inval_callback, (Datum) 0);
```

Run setter function to register your own callback

# Callbacks: setter function



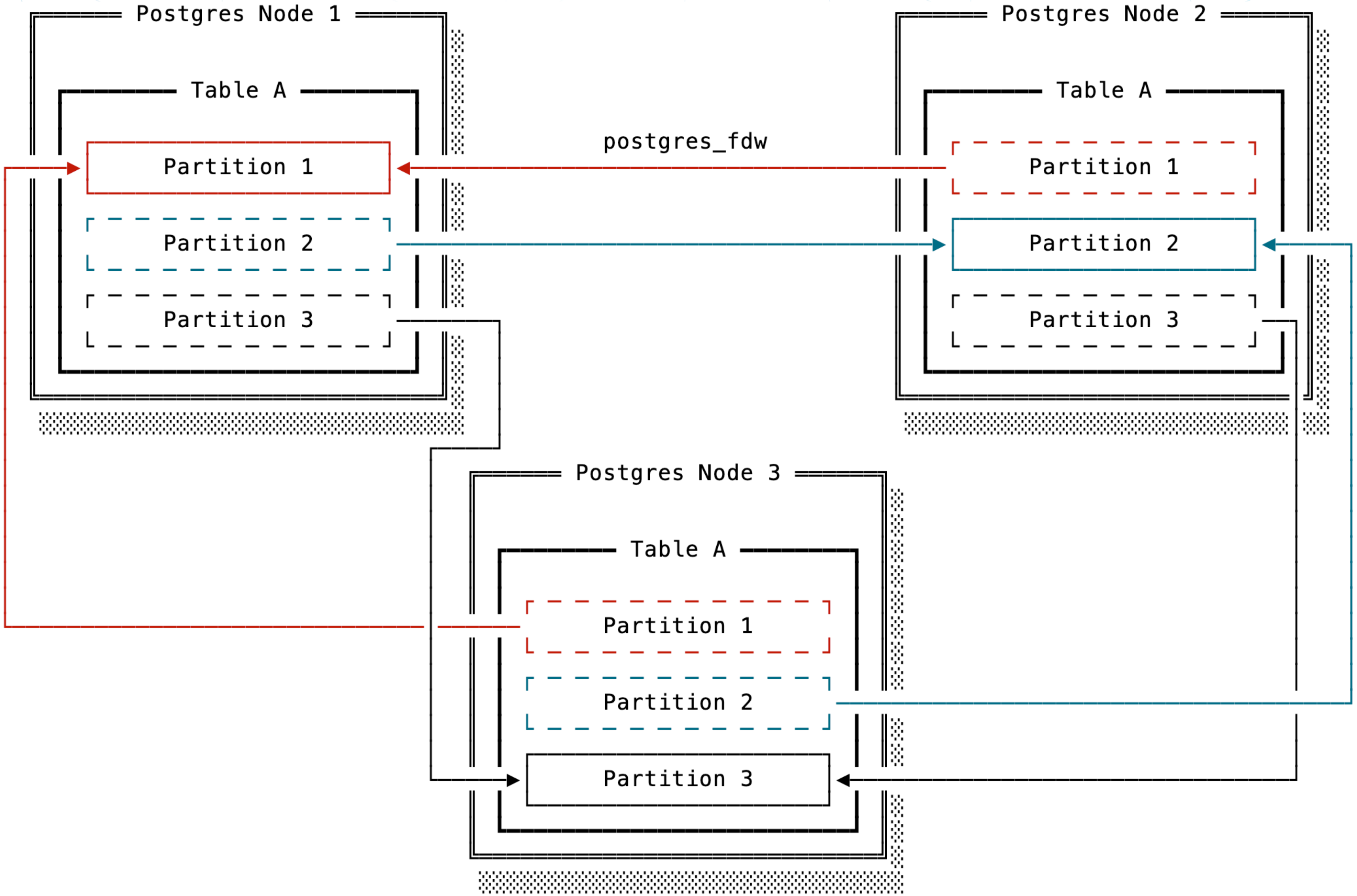
```c
void
RegisterXactCallback(XactCallback callback, void *arg)
{
    XactCallbackItem *item;

    item = (XactCallbackItem *)
        MemoryContextAlloc(TopMemoryContext, sizeof(XactCallbackItem));
    item->callback = callback;
    item->arg = arg;
    item->next = Xact_callbacks;
    Xact_callbacks = item;
}
```

xact.c

Keeps a list of registered callbacks

# Example time

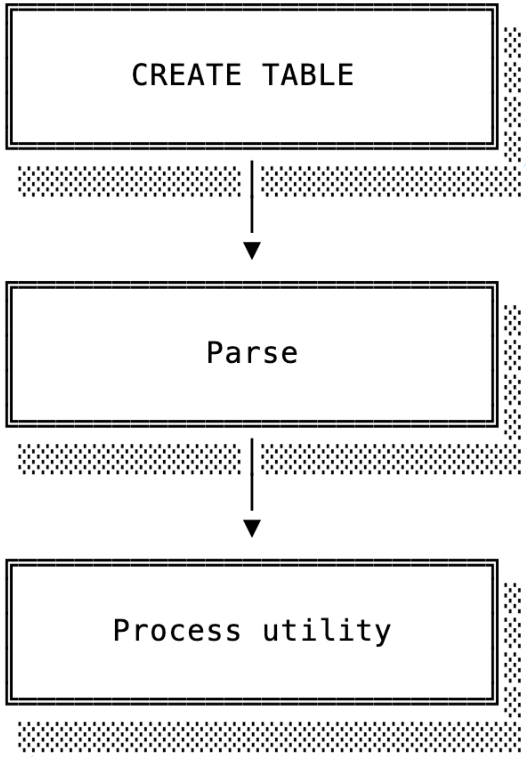
# Distributed PostgreSQL



# Distributed DDL

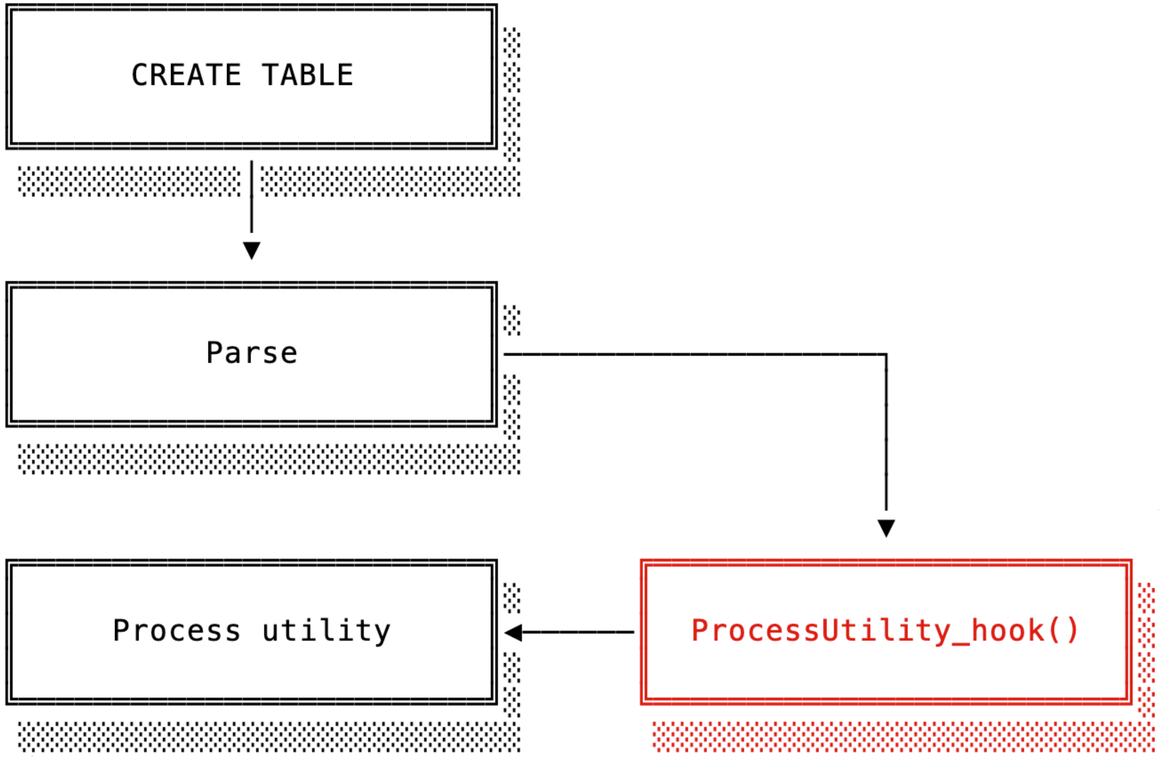
- **Broadcast** specific (or all) DDL across a number of PostgreSQL nodes.
- Create distributed (sharded / partitioned) tables with familiar interface → **extend CREATE TABLE statement syntax**.
- This operation should be **atomic**, i.e. either committed or aborted on all PostgreSQL instances → **use two-phase commit (2PC)**.
- Do everything from the extension → **no core modifications**!

# Standard DDL processing



- ○ Get query from the client.
- ○ Parse and plan it.
- ○ Pass it to the `standard_ProcessUtility()`.

# Distributed DDL: broadcast



Utility hook receives:

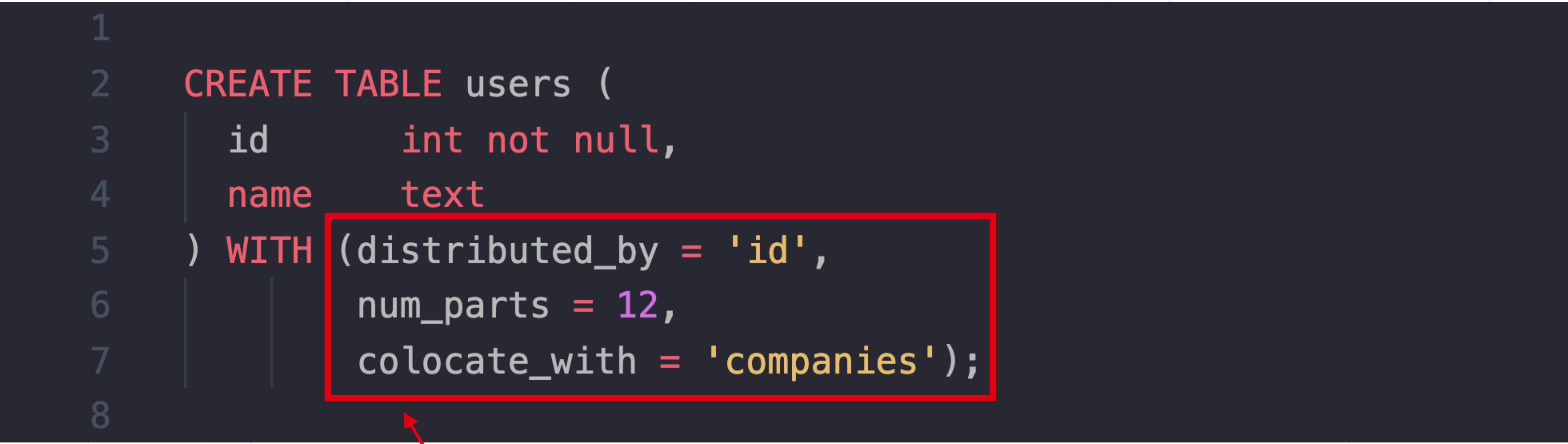
○ Raw text of the statement.

○ Planned statement.

○ **So it can decide whether to send this DDL to other servers or not [1].**

[1] Source code of broadcast example can be found on GitHub postgrespro/shardman.

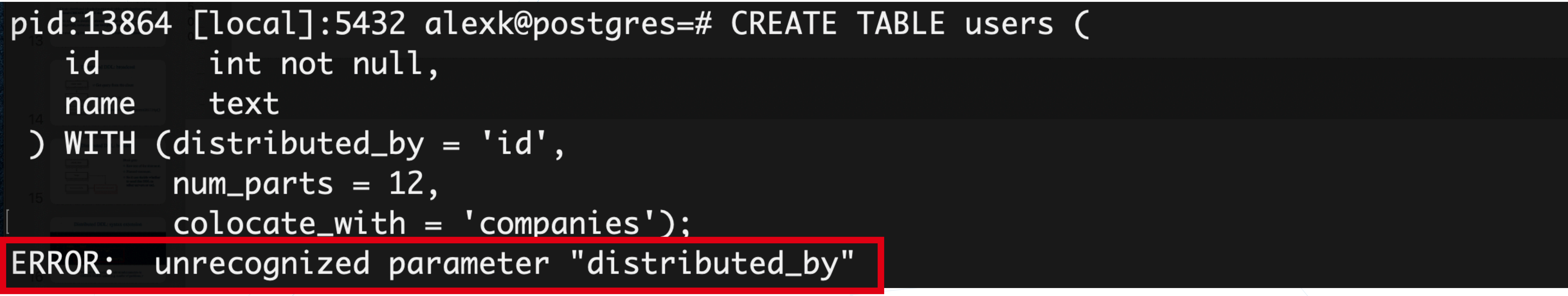
# Distributed DDL: syntax extension



```

CREATE TABLE users (
  id      int not null,
  name    text
) WITH (distributed_by = 'id',
        num_parts = 12,
        colocate_with = 'companies');

```

We would like to add some additional parameters to CREATE TABLE syntax (e.g. **number of partitions**, **partitioning column name**).

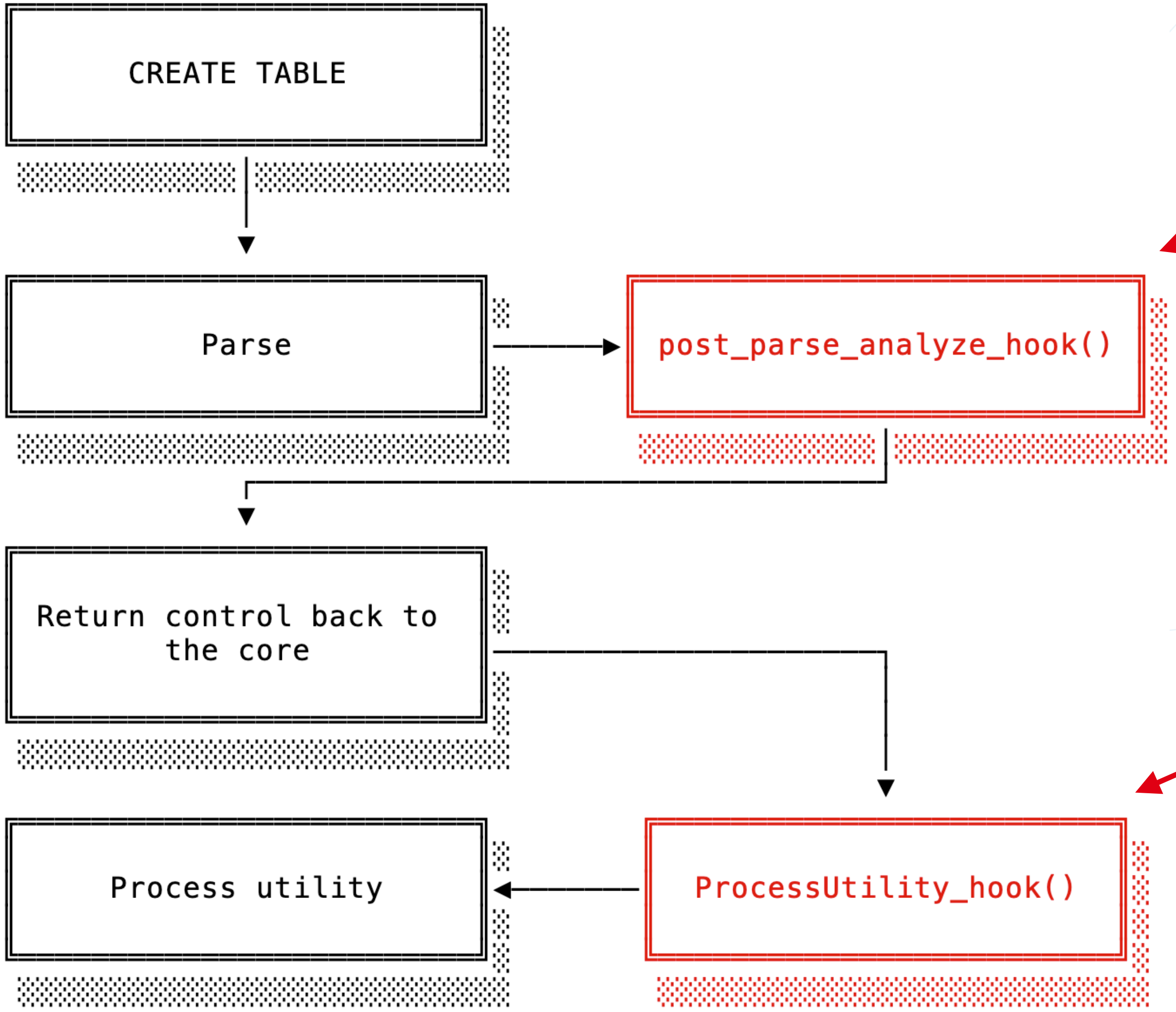
# Distributed DDL: syntax extension



```
pid:13864 [local]:5432 alexk@postgres=# CREATE TABLE users (
   id      int not null,
   name    text
 ) WITH (distributed_by = 'id',
         num_parts = 12,
         colocate_with = 'companies');
ERROR:  unrecognized parameter "distributed_by"
```

Luckily, not a ‘syntax error’, so **parameters are not processed by the parser itself**!

# Distributed DDL: syntax extension



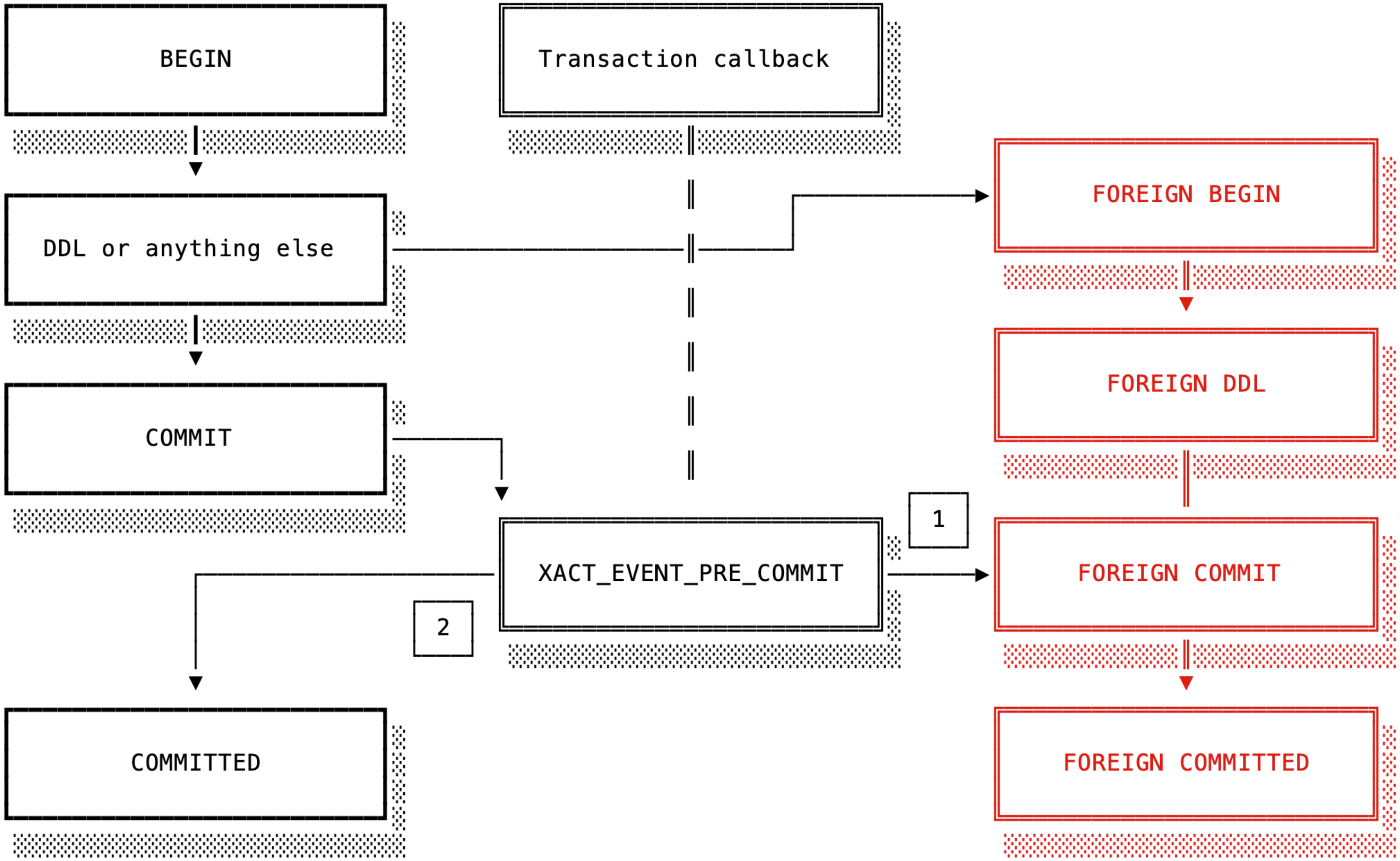
1. Notice, remember and **remove** known additional parameters.

2. Process statement taking into account the specified parameters (i.e. **add partitioning info**, **create partitions** as well, **do broadcast**).

# Distributed DDL: atomicity

- Without 2PC, transaction might end up **COMMITTED** on some nodes and **ABORTED** on others.
- 2PC introduces an intermediate state — **PREPARED**.
- PostgreSQL already has a 2PC infrastructure.

# Distributed DDL: transaction

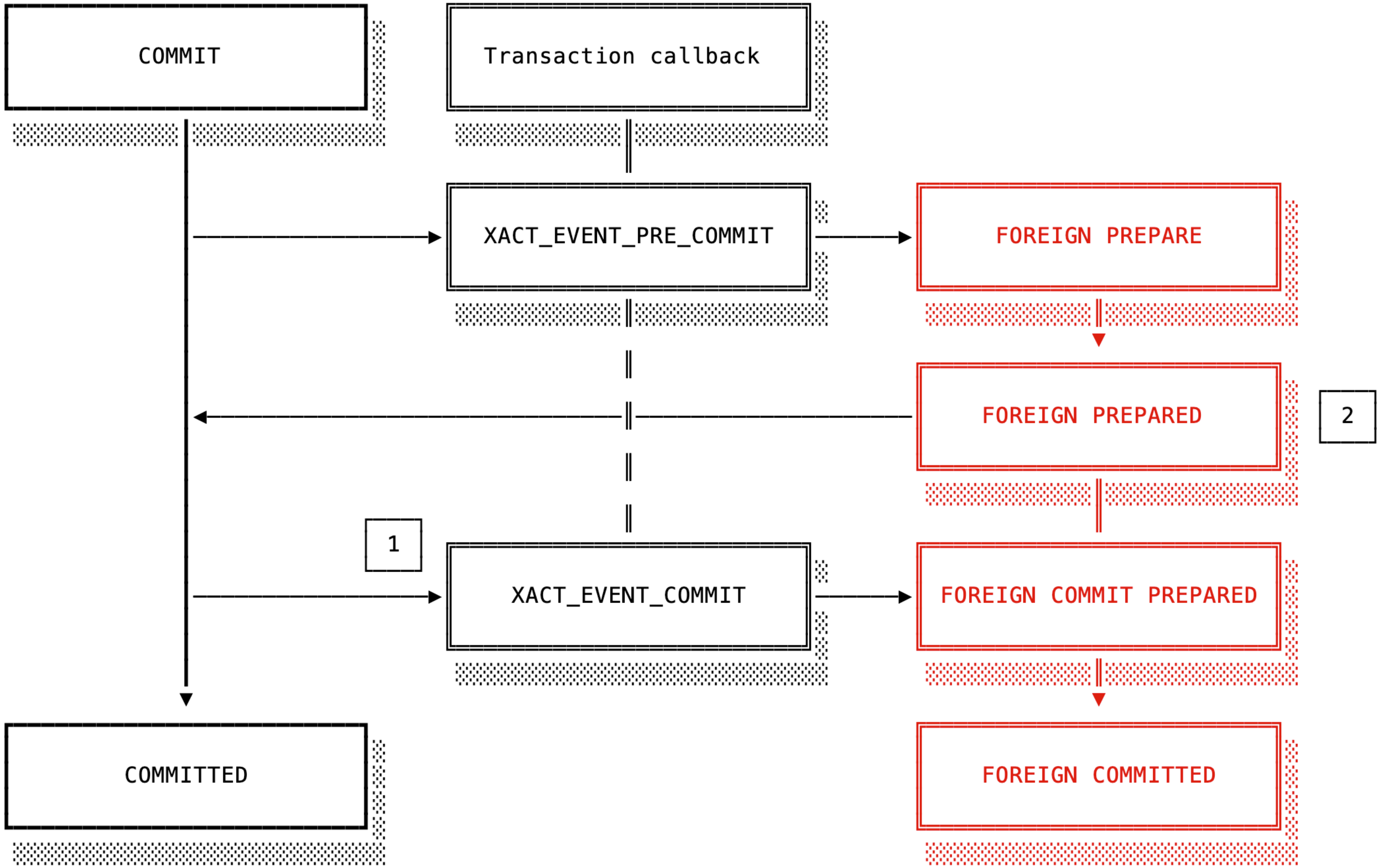


Transaction (xact) callback is used by `postgres_fdw` to:

1. Commit all foreign transactions first.

2. Proceed to local commit.

# Distributed DDL: 2PC



At stage (1) it is too late to abort local transaction and if we will fail to commit all remote xacts, then some of them may be left in the PREPARED state (2). In this case some additional process (resolver) have to either commit them or abort based on the coordinator state.

Simple patch prototype, which adds 2PC into postgres_fdw can be found in the pgsql-hackers mailing list.

# Feedback

If you have any questions or comments:

- kondratov.aleksey@gmail.com
- github.com/ololobus
- twitter.com/ololobuss

Thank you!