Distributed DDL Replication at Global Scale

Gwen Shapira, co-founder, Nile

Nice to meet you! I'm Gwen Shapira

- Co-founder of Nile:
  Serverless Postgres for Modern SaaS
- Presenting on behalf of a smart team
- Previously: Cloud Native Kafka lead @ Confluent, Apache Kafka core committer
- Previously: Data architect (Hadoop, MySQL, Oracle)
- Wrote books, tweet a lot @gwenshap

Nile

Agenda

- Why distributed DDL?
- Step by step DDL flow
- Transactions
- Locks
- Handling Failures

Serverless Postgres for modern SaaS

Launch In Days. Grow For Years

Nile

Modern SaaS:

Nile

SaaS == Multi-Tenant

Nile

SaaS HR – Schema / DB per tenant

Customer 1

EMPNO	ENAME	SAL
7499	ALLEN	1600
7521	WARD	1250
7566	JONES	2975
7654	MARTIN	1250
7698	BLAKE	2850
7782	CLARK	2450
7788	SCOTT	3000
7839	KING	5000
7844	TURNER	1500
7876	ADAMS	1100
7900	JAMES	950
7902	FORD	3000
7934	MILLER	1300

Customer 2

EMPNO	ENAME	SAL
7499	ALLEN	1600
7521	WARD	1250
7566	JONES	2975
7654	MARTIN	1250
7698	BLAKE	2850
7782	CLARK	2450
7788	SCOTT	3000
7839	KING	5000
7844	TURNER	1500
7876	ADAMS	1100
7900	JAMES	950
7902	FORD	3000
7934	MILLER	1300

Customer 3

EMPNO	ENAME	SAL
7499	ALLEN	1600
7521	WARD	1250
7566	JONES	2975
7654	MARTIN	1250
7698	BLAKE	2850
7782	CLARK	2450
7788	SCOTT	3000
7839	KING	5000
7844	TURNER	1500
7876	ADAMS	1100
7900	JAMES	950
7902	FORD	3000
7934	MILLER	1300

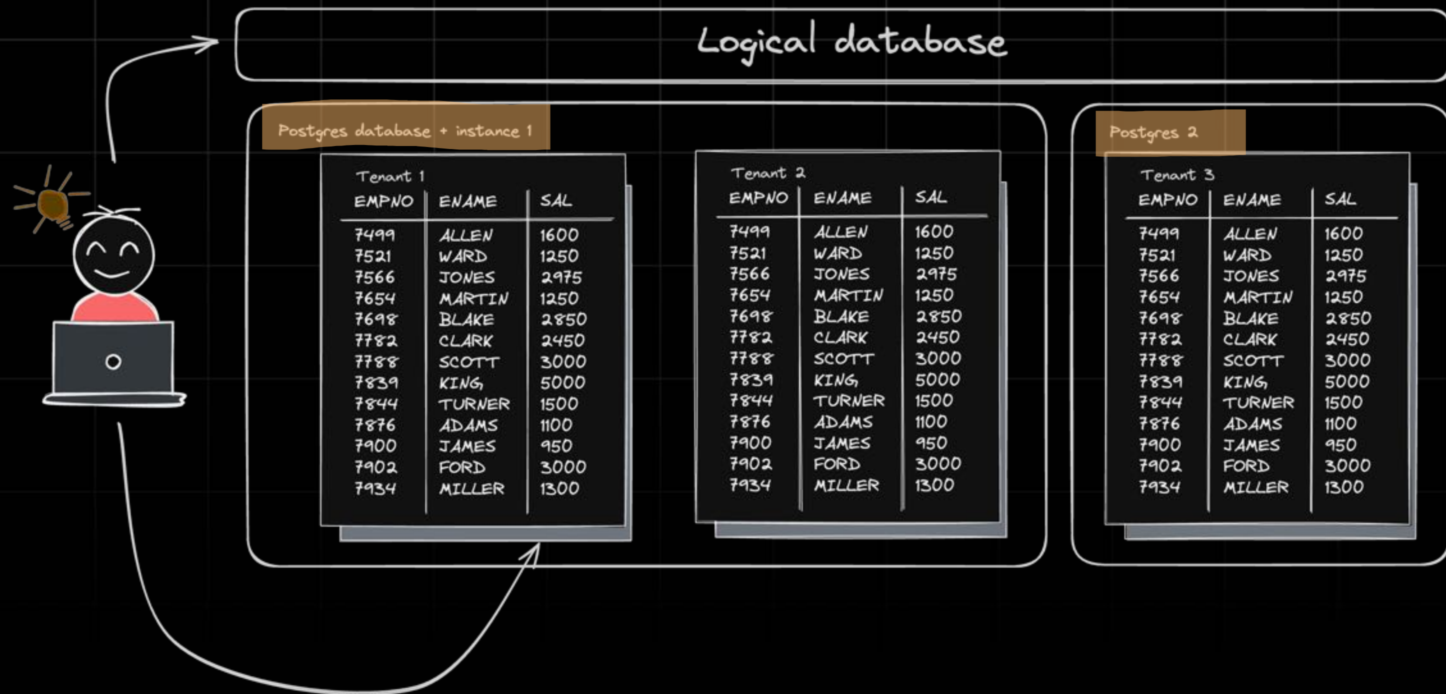
Nile

SaaS HR – Multi-tenant schema

tenant_id	EMPNO	ENAME	SAL
1	7499	ALLEN	1600
1	7521	WARD	1250
2	7566	JONES	2975
1	7654	MARTIN	1250
2	7698	BLAKE	2850
2	7782	CLARK	2450
3	7788	SCOTT	3000
5	7839	KING	5000
3	7844	TURNER	1500
1	7876	ADAMS	1100
2	7900	JAMES	950
5	7902	FORD	3000
2	7934	MILLER	1300

Nile

Nile model: Virtual tenant databases

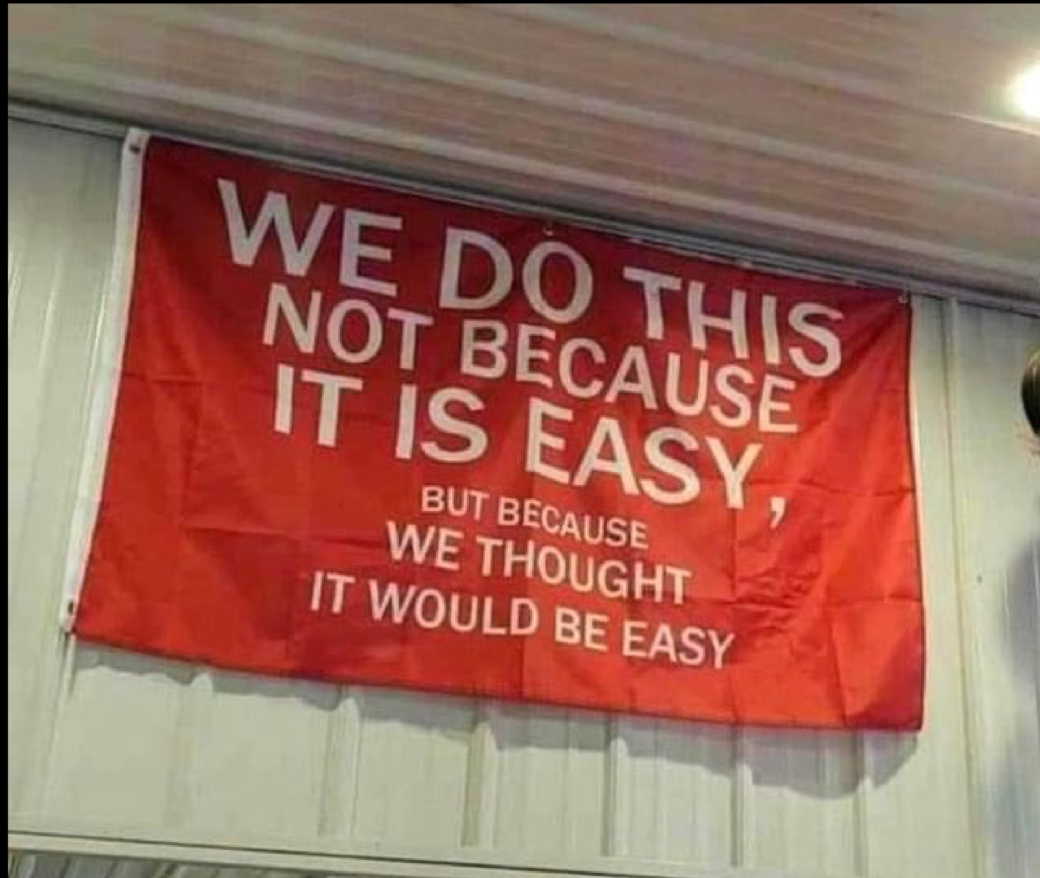


Nile

Our Goals:

- Each DDL applies to all tenants
- At “same time”
- Behave exactly like Postgres
- Completely transparent

Nile

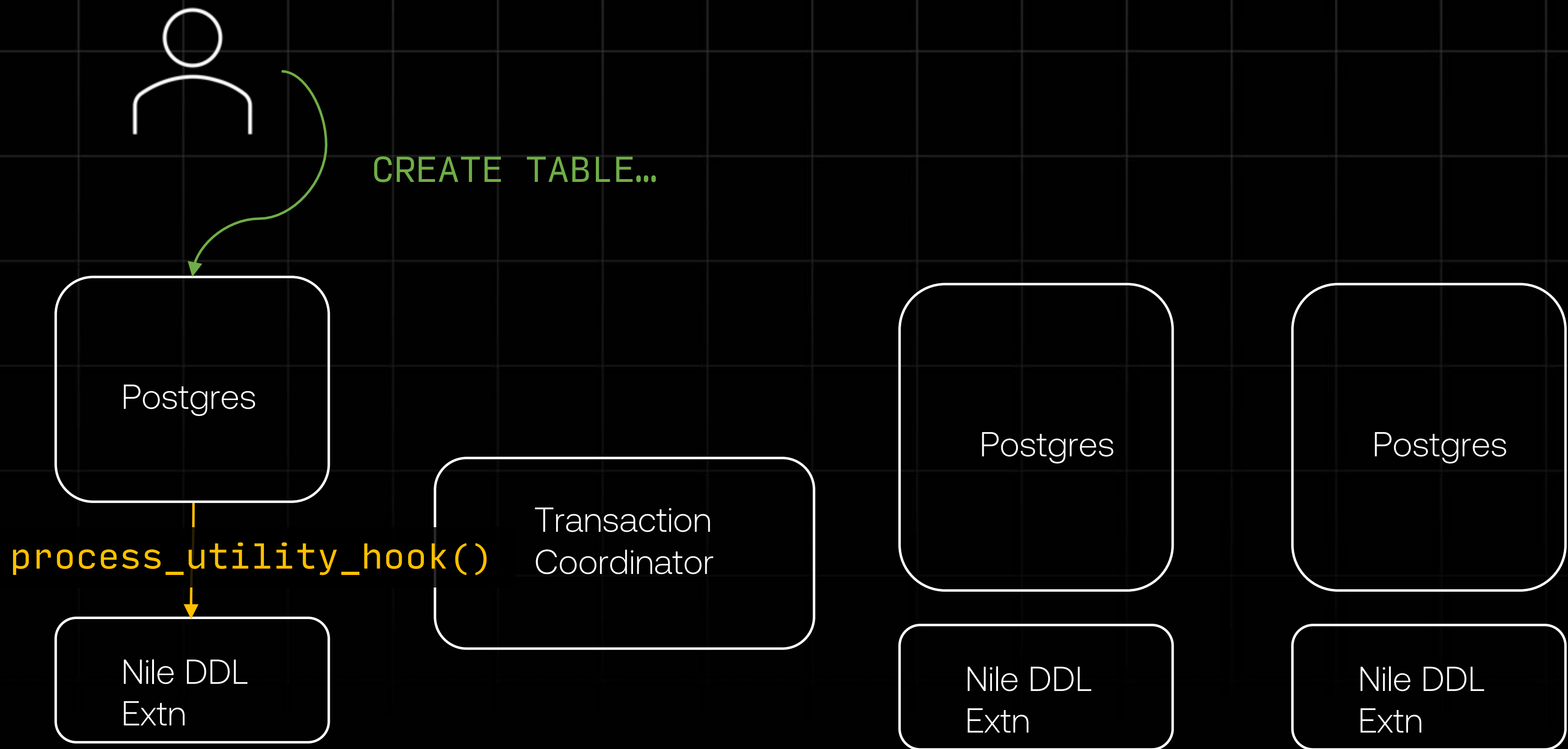


Nile

Distributed DDL: Step by Step

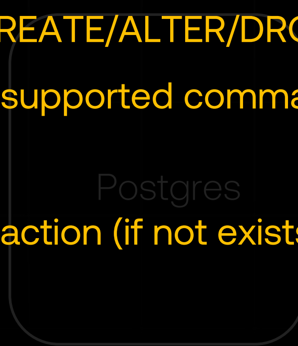
Nile

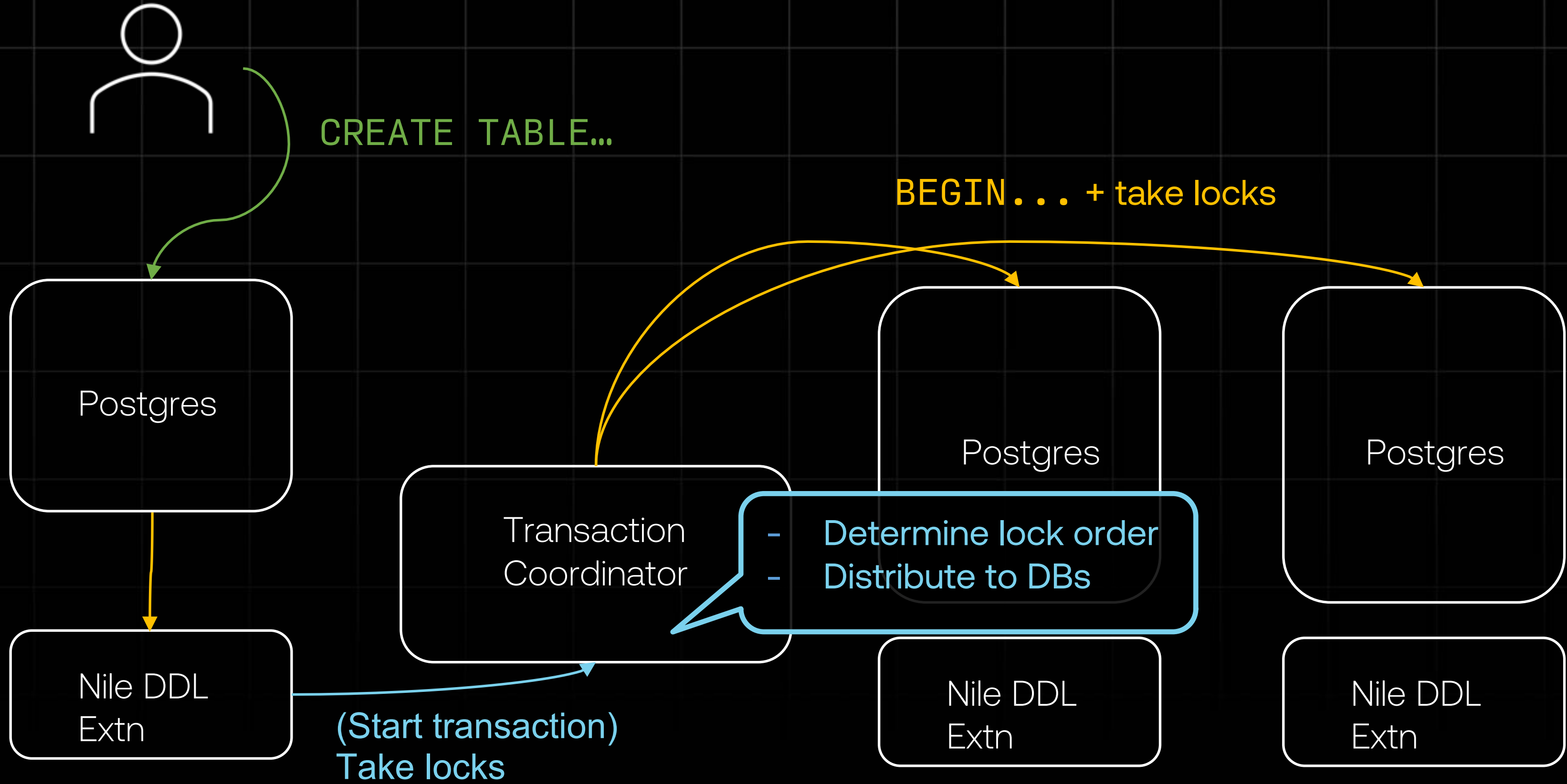
Component Overview


Nile
CREATE TABLE…
Postgres
process_utility_hook()
Nile DDL
Extn
Transaction
Coordinator
Postgres
Nile DDL
Extn
Postgres
Nile DDL
Extn

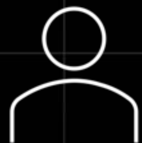
Nile

- Skip everything but CREATE/ALTER/DROP
- Raise errors on yet unsupported commands
- Extract needed locks
- Start distributed transaction (if not exists…)
- Distribute locks

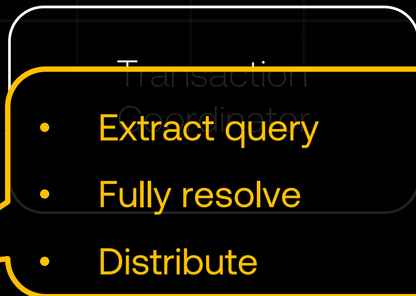



Nile
CREATE TABLE…
BEGIN... + take locks
Postgres
Transaction
Coordinator
- Determine lock order
- Distribute to DBs
Postgres
Postgres
Nile DDL
Extn
(Start transaction)
Take locks
Nile DDL
Extn
Nile DDL
Extn

Nile

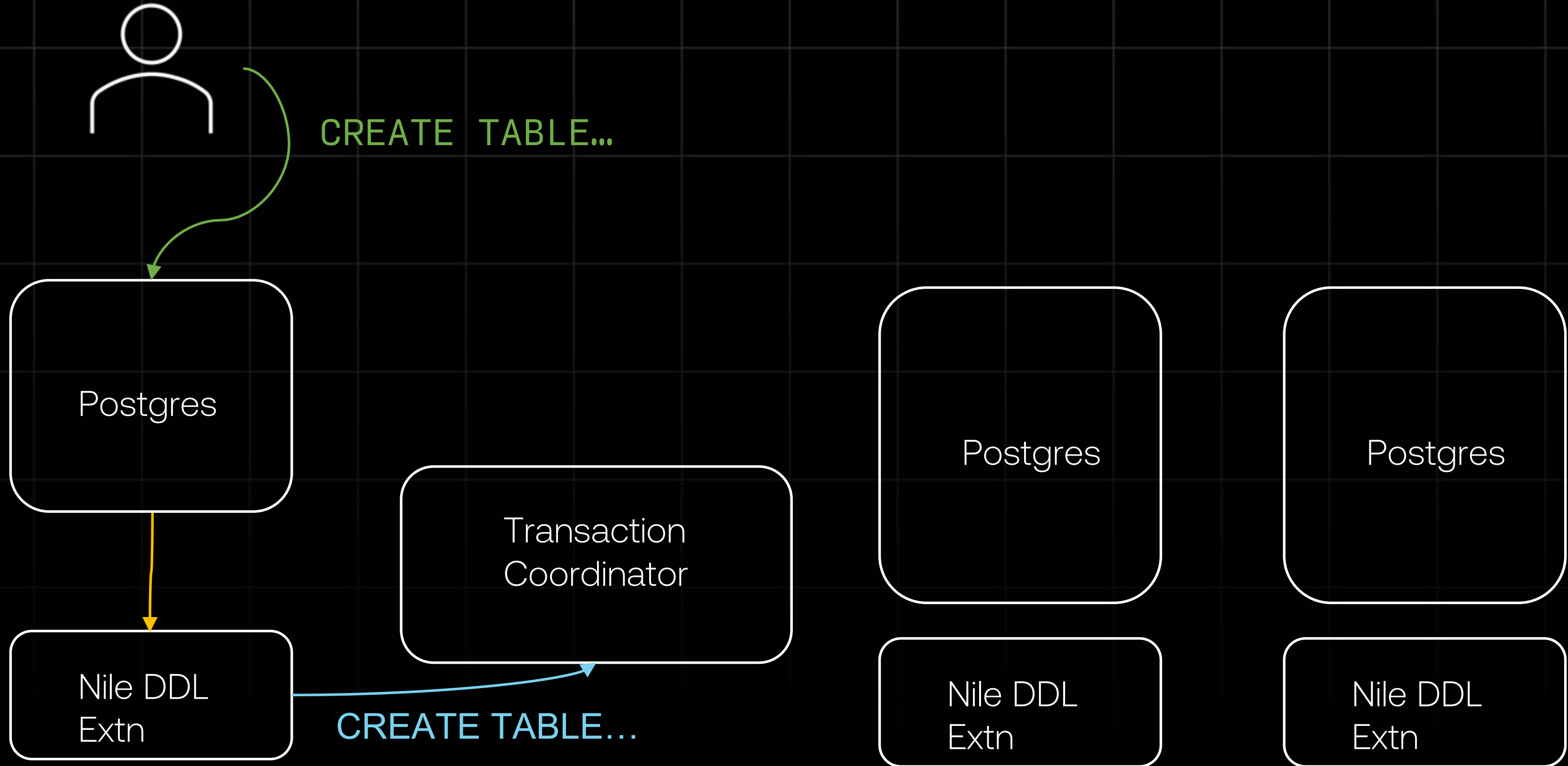


CREATE TABLE…



- Extract query
- Fully resolve
- Distribute

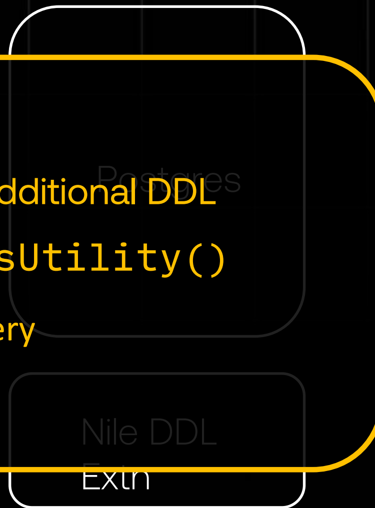
Nile

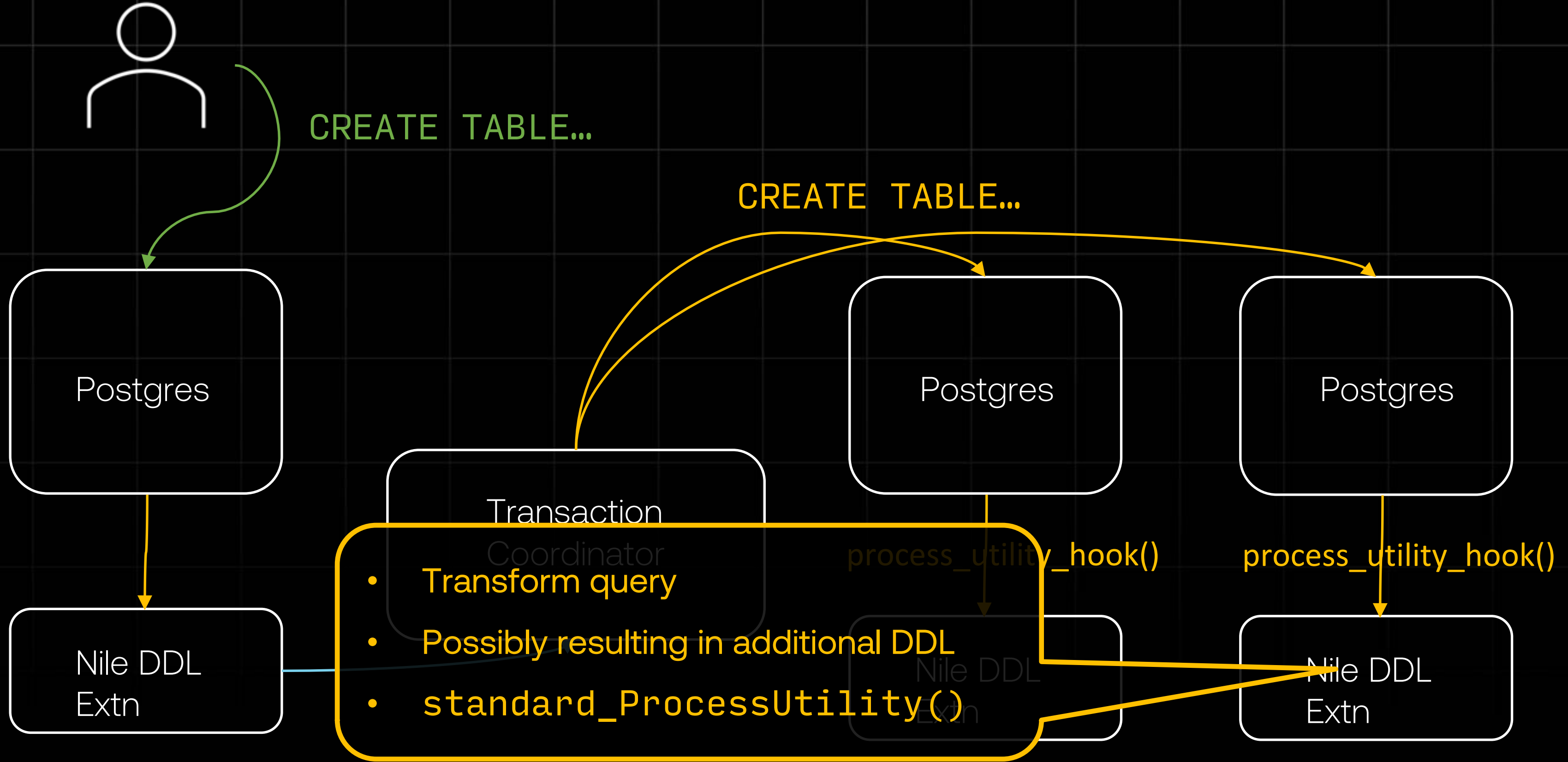


Nile

CREATE TABLE…

- Transform query
- Possibly resulting in additional DDL
- `standard_ProcessUtility()`
- Wait for distributed query




Nile
CREATE TABLE…
CREATE TABLE…
Postgres
Postgres
Postgres
Transaction
Coordinator
process_utility_hook()
process_utility_hook()
Nile DDL
Extn
Nile DDL
Extn
Nile DDL
Extn
Transform query
Possibly resulting in additional DDL
standard_ProcessUtility()

Nile

That’s it.
DDL was distributed.
We are done.

Nile

Transactions

Maybe use `process_utility_hook`?

These are all utility commands:

- `BEGIN`
- `COMMIT`
- `ABORT`
- `ROLLBACK`

`process_utility_hook` isn't useful here

These are all utility commands:

- `BEGIN`
- `COMMIT`
- `ABORT`
- `ROLLBACK`

But...

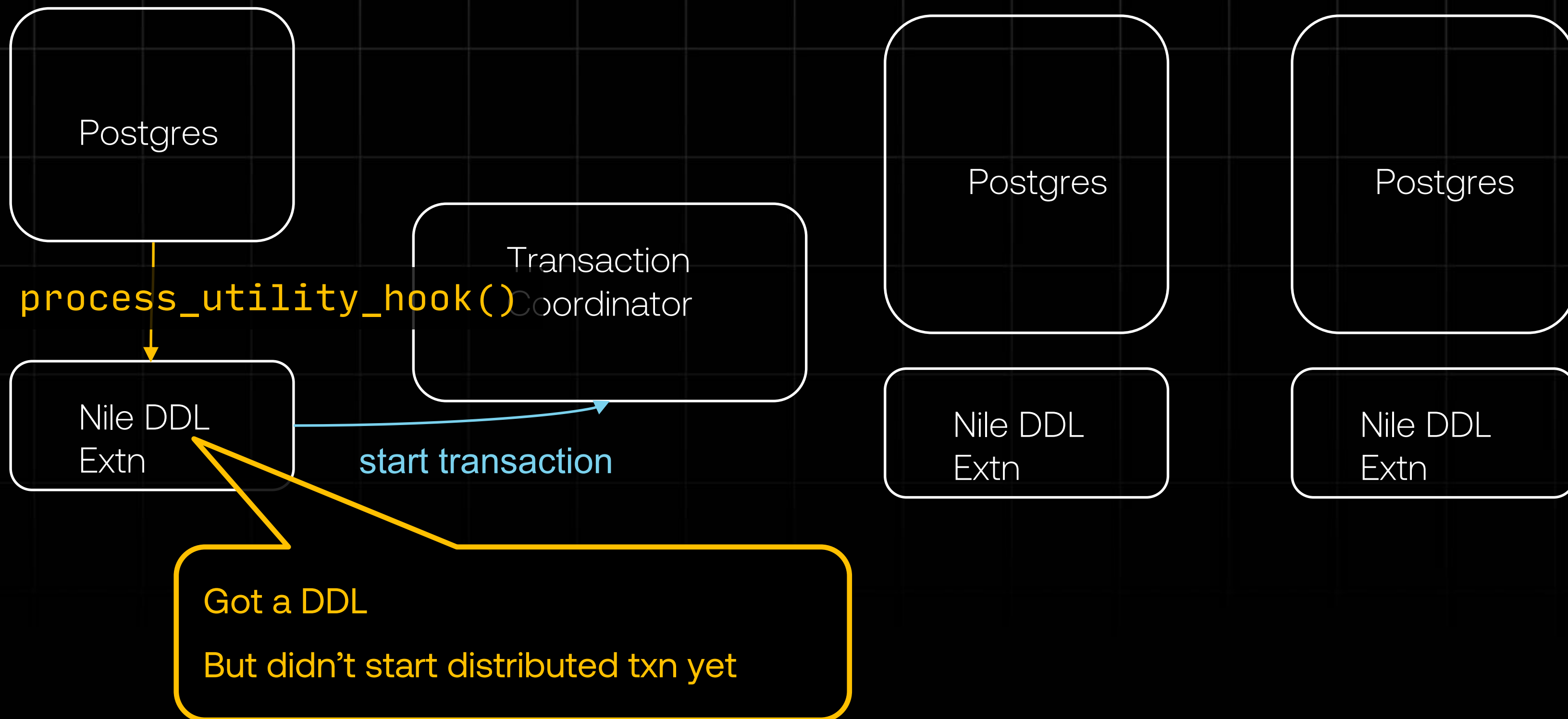
- Intercepting every `BEGIN` is costly
- And unnecessary
- Stand alone DDL is a transaction
- Other implicit transactions
- We need two phase commit

Nile

XactCallback to the Rescue

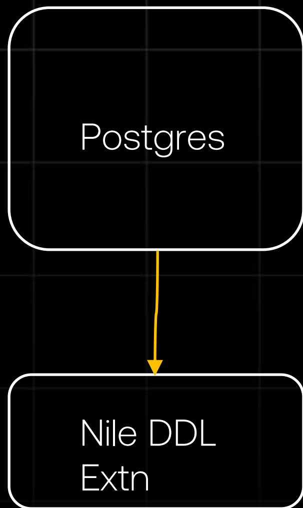
- XACT_EVENT_PRE_COMMIT
- XACT_EVENT_COMMIT
- XACT_EVENT_ABORT

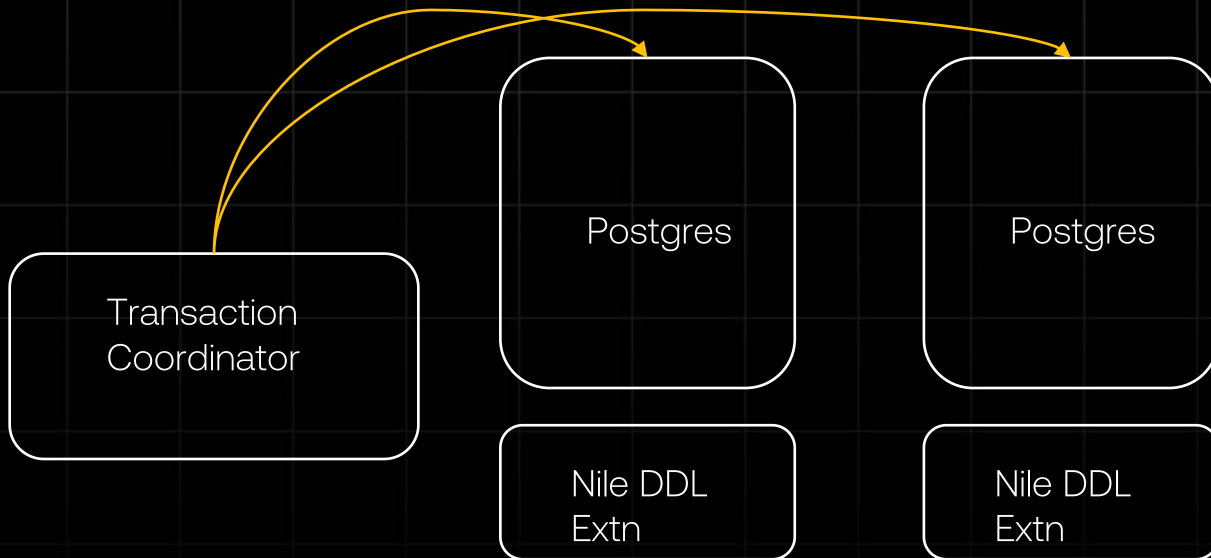
Nile

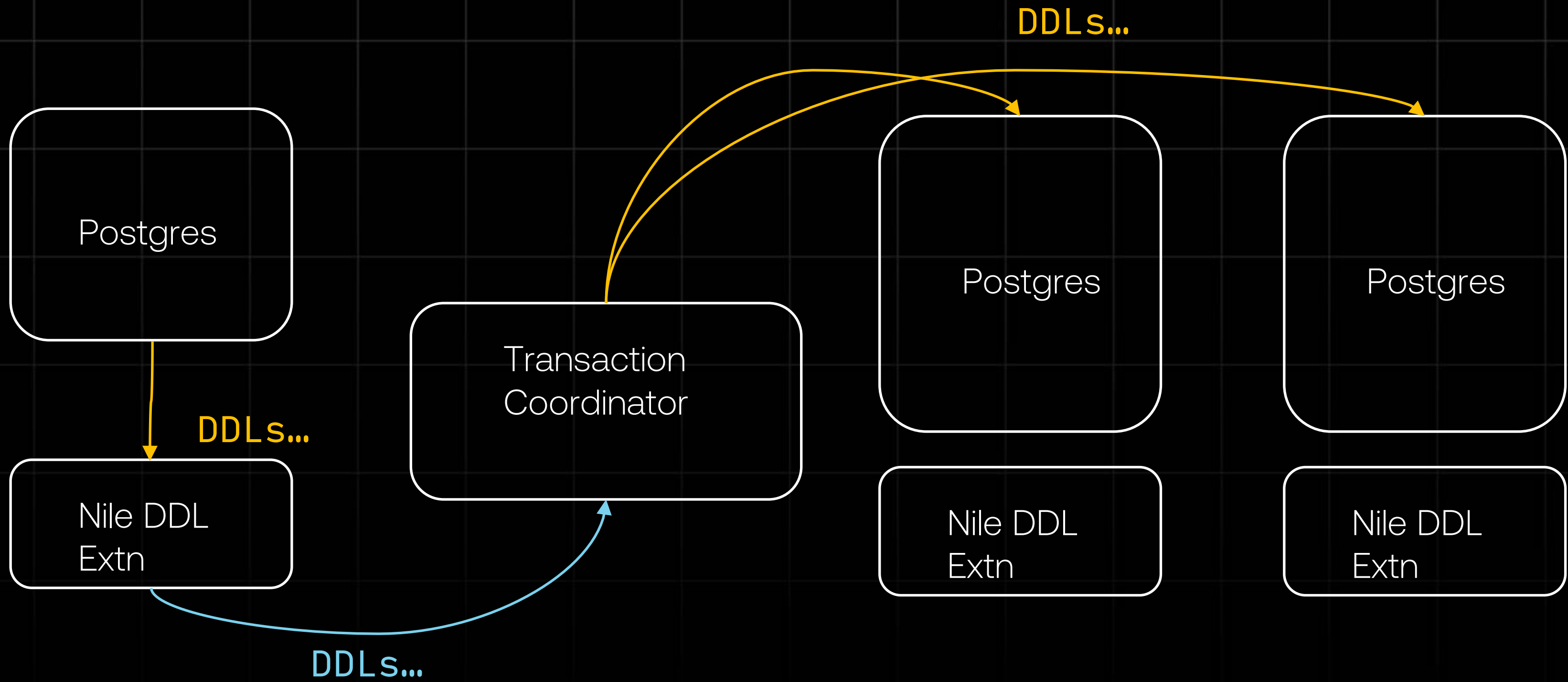


Nile

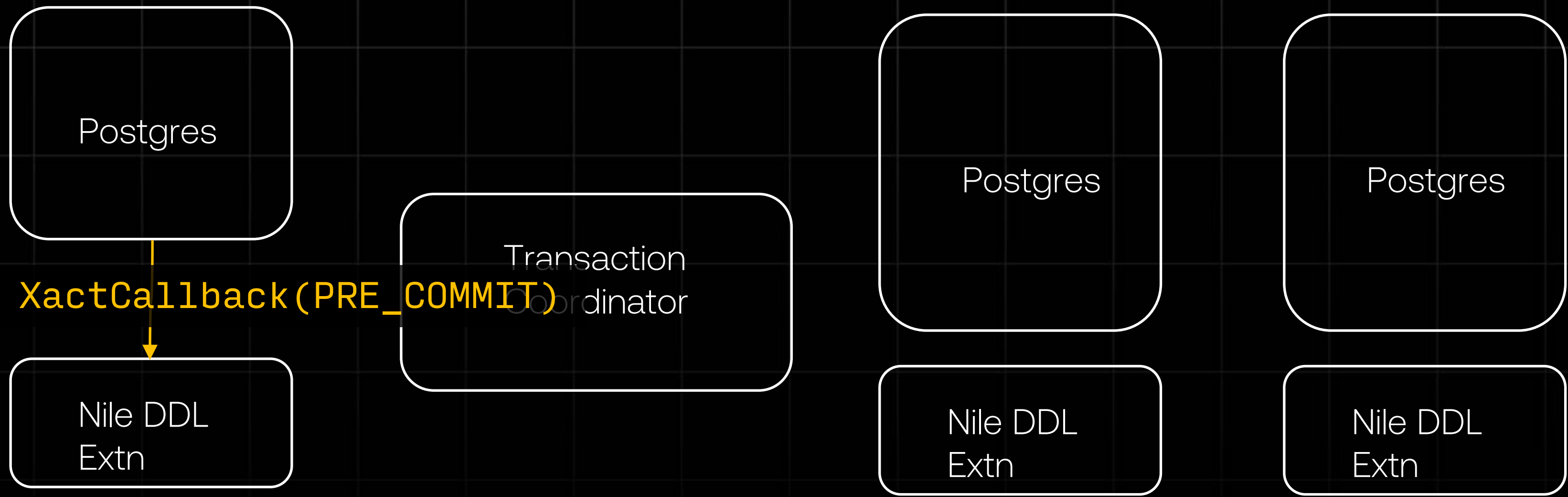
```
SET lock_timeout =
BEGIN
```



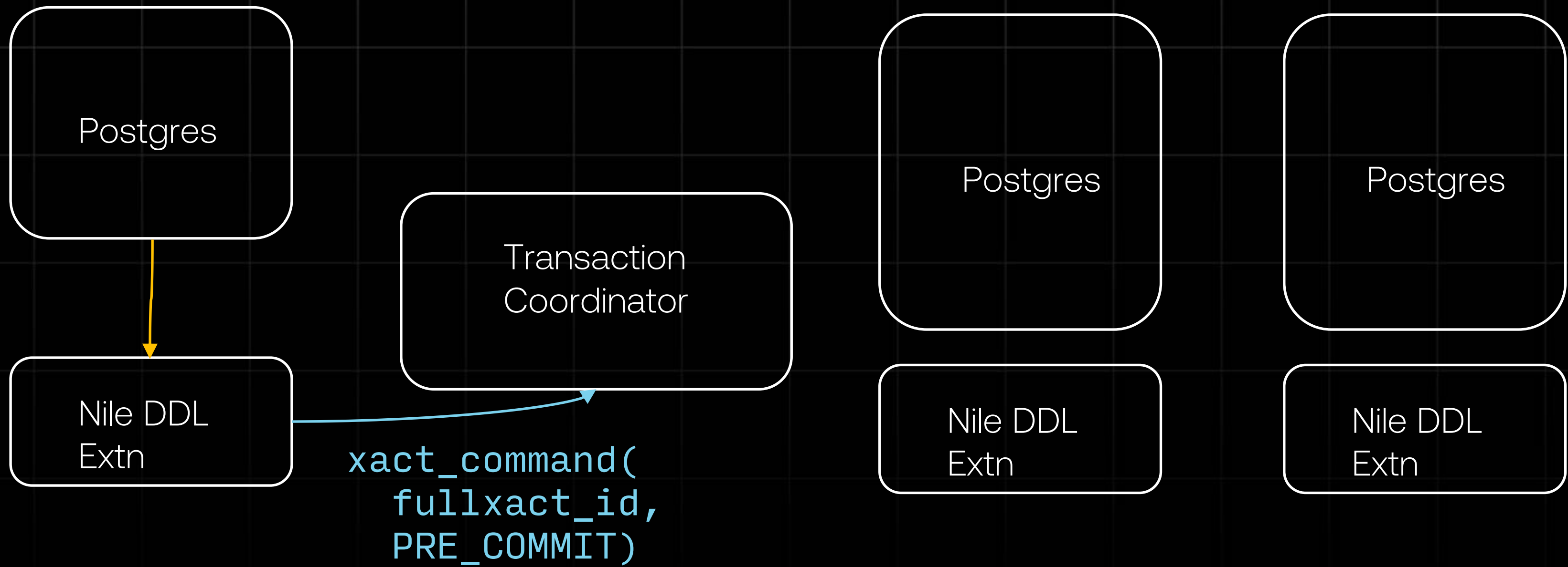


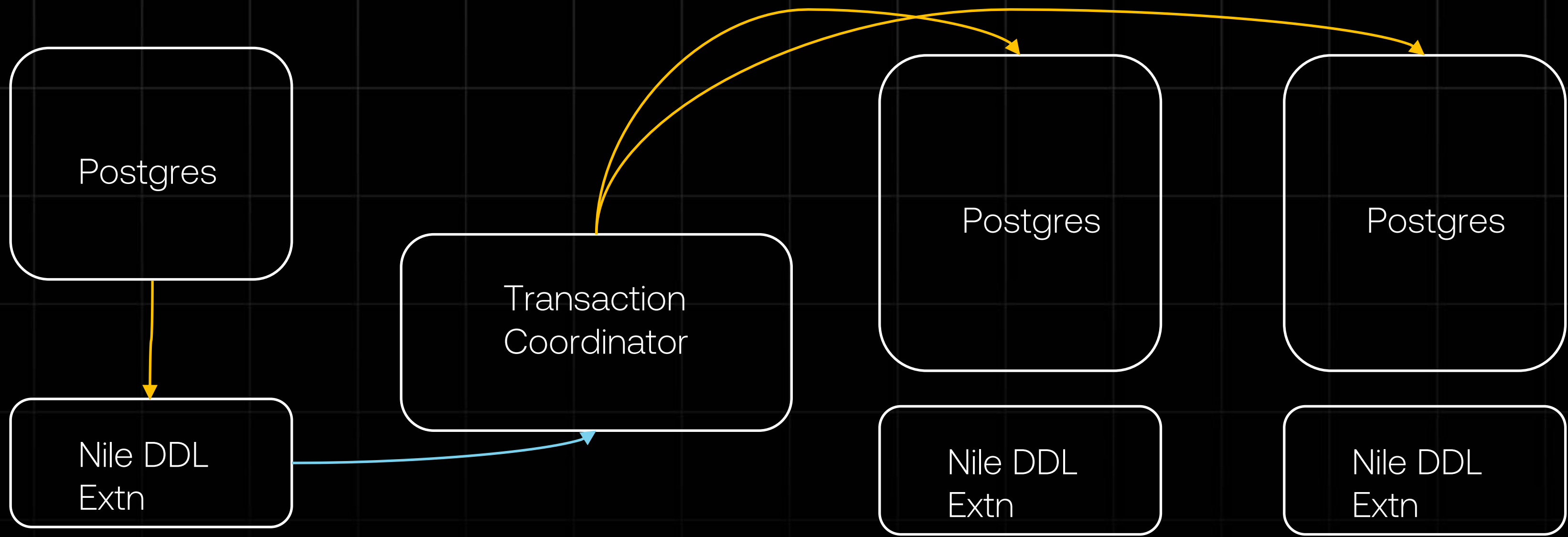

Nile
DDLs…
Postgres
Transaction
Coordinator
Postgres
Postgres
DDLs…
Nile DDL
Extn
Nile DDL
Extn
Nile DDL
Extn
DDLs…

Nile



Nile




Nile
PREPARE TRANSACTION <GID>
Postgres
Transaction Coordinator
Postgres
Postgres
Nile DDL Extn
Nile DDL Extn
Nile DDL Extn

Nile

Synopsis

```
PREPARE TRANSACTION transaction_id
```

Description

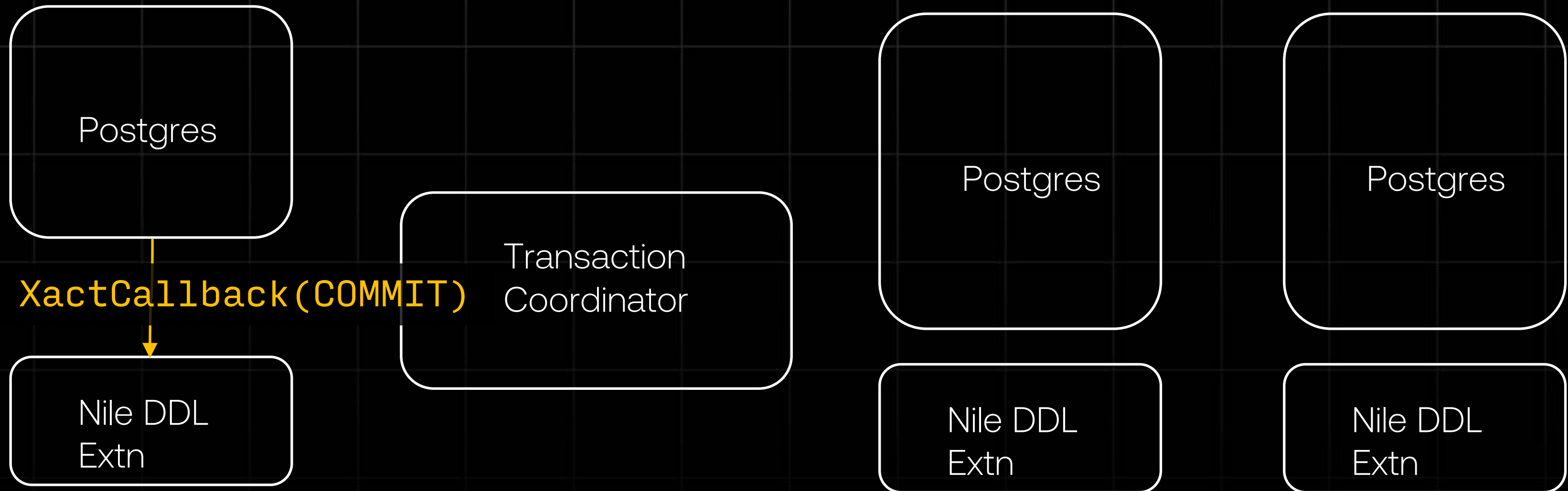
`PREPARE TRANSACTION` prepares the current transaction for two-phase commit. After this command, the transaction is no longer associated with the current session; instead, its state is fully stored on disk, and there is a very high probability that it can be committed successfully, even if a database crash occurs before the commit is requested.

Once prepared, a transaction can later be committed or rolled back with `COMMIT PREPARED` or `ROLLBACK PREPARED`, respectively. Those commands can be issued from any session, not only the one that executed the original transaction.

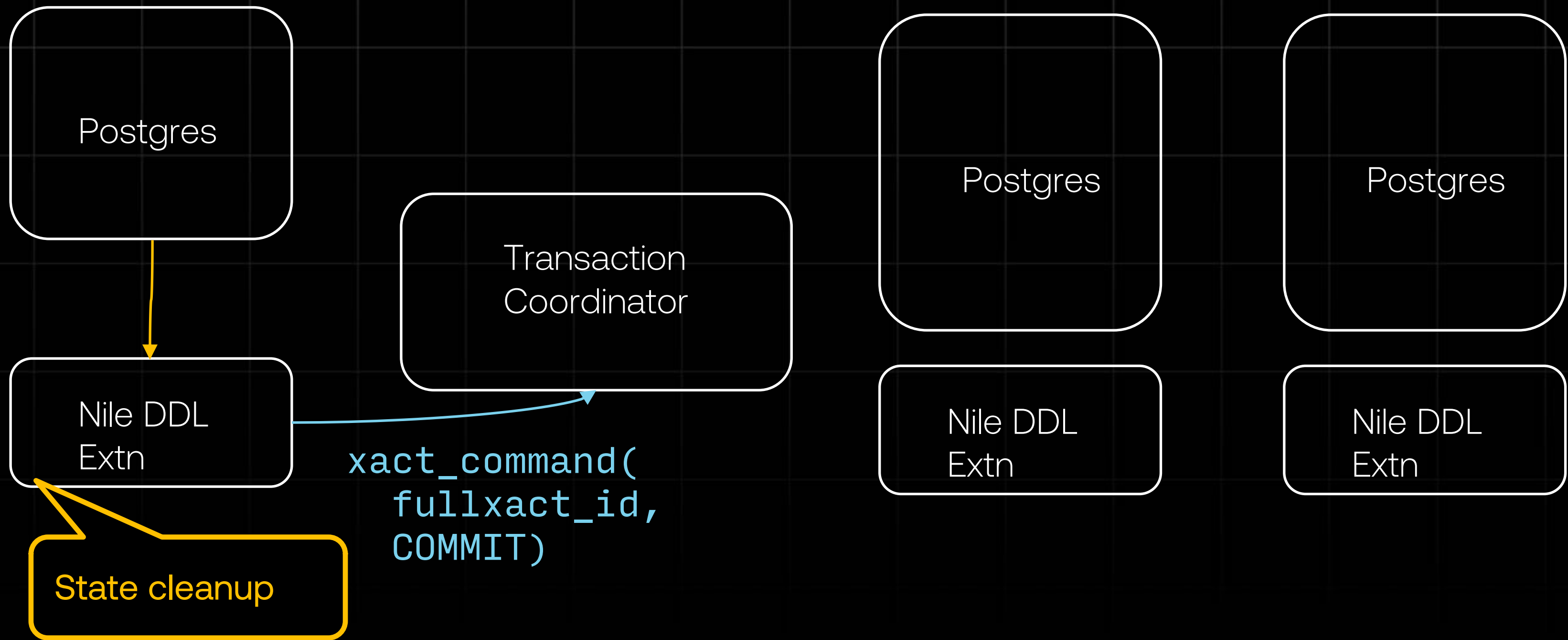
From the point of view of the issuing session, `PREPARE TRANSACTION` is not unlike a `ROLLBACK` command: after executing it, there is no active current transaction, and the effects of the prepared transaction are no longer visible. (The effects will become visible again if the transaction is committed.)

If the `PREPARE TRANSACTION` command fails for any reason, it becomes a `ROLLBACK`: the current transaction is canceled.

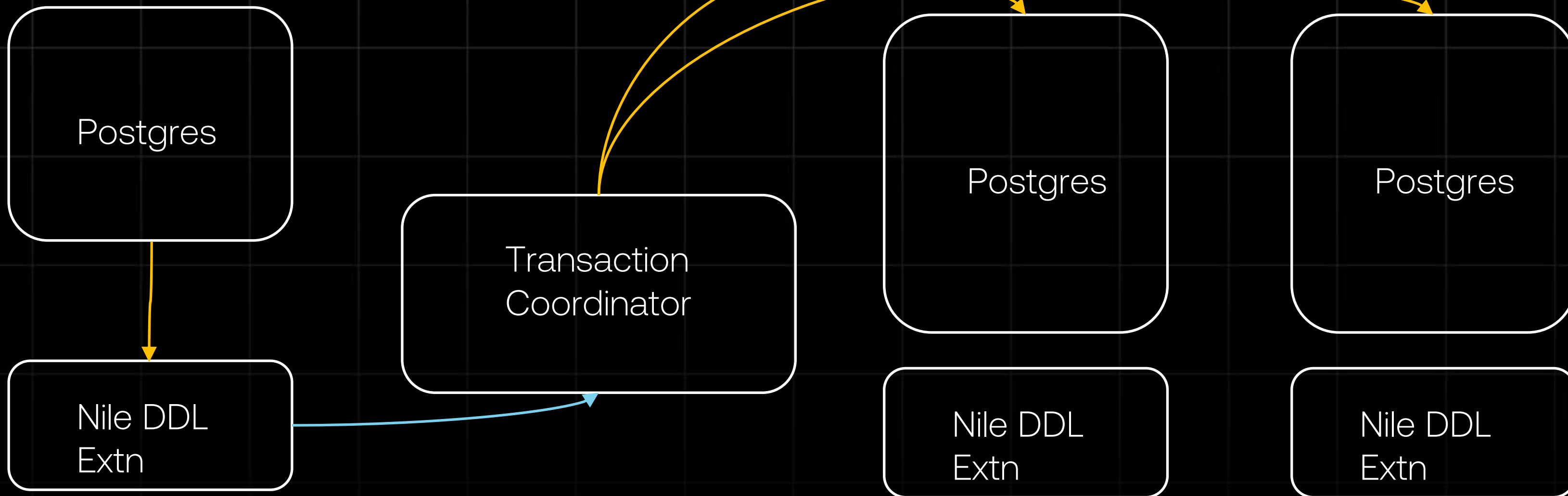
Nile



Nile



Nile



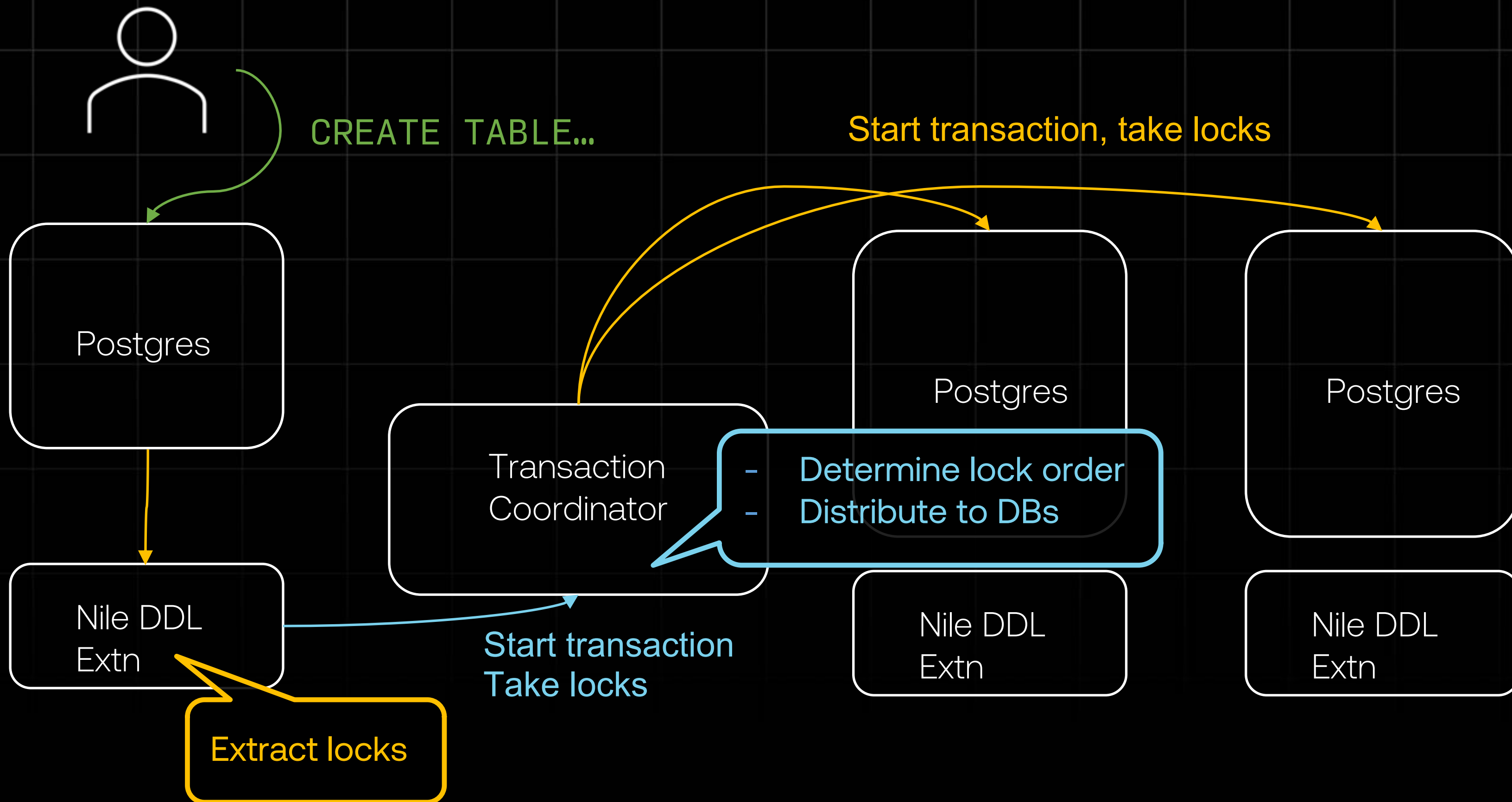
Nile

Locks

Nile

We need just a few simple things

- Fail early
- Don’t get stuck (for too long)
- Don’t deadlock


Nile
CREATE TABLE…
Start transaction, take locks
Postgres
Postgres
Postgres
Transaction
Coordinator
- Determine lock order
- Distribute to DBs
Nile DDL
Extn
Start transaction
Take locks
Nile DDL
Extn
Nile DDL
Extn
Extract locks

Extract Locks

`ALTER / DROP:`

- Need object locks on the relevant objects
- Match the lock level PG will need later

`CREATE:`

- Need our own locks

Nile

How we lock locally? (ALTER / DROP)

- Can’t use LOCK, because it doesn’t work on SCHEMA.

```
if (is_rel)
   LockRelationOid(objid, lockmode);
else
   LockDatabaseObject(classid, objid,
                                 objsubid, lockmode);
```

Nile

How we lock locally? `(CREATE)`

- Can’t object-lock non-existing objects
- PG inserts into pg_catalog and locks the PK index. Can’t do that either.
- Want to leave all advisory locks to users

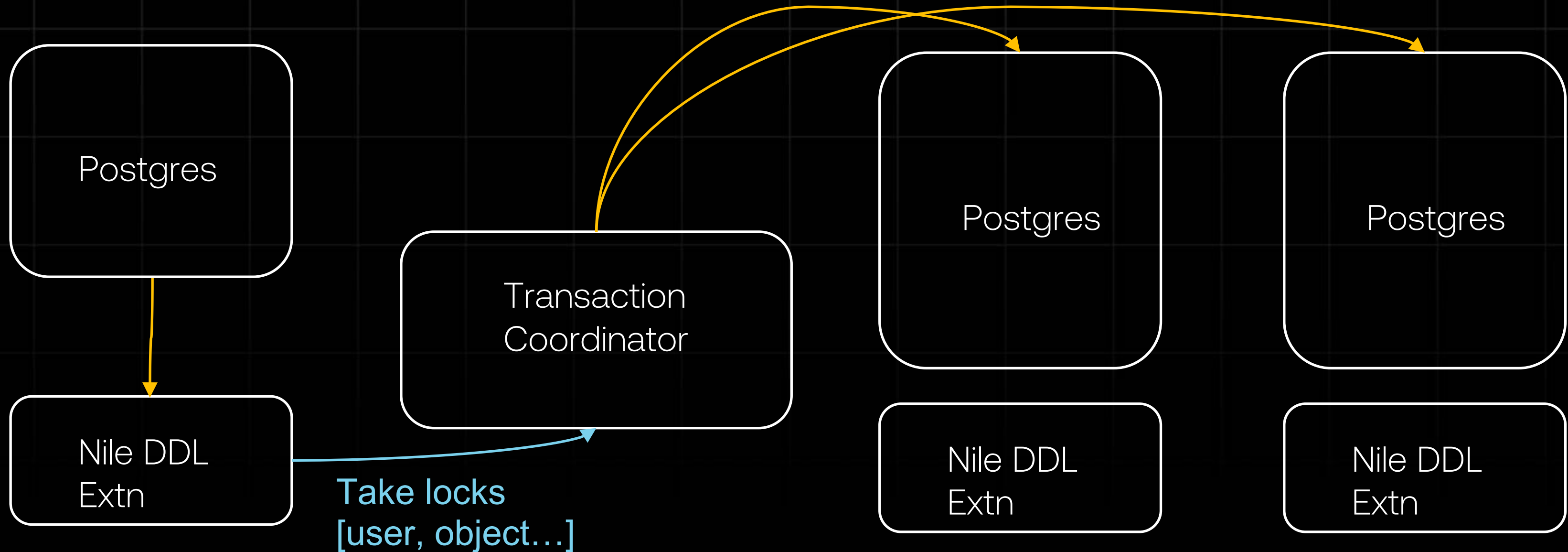
Nile

How we lock locally? (CREATE)

```
locktag.locktag_type = LOCKTAG_USERLOCK;

locktag.locktag_field1 = logical_db_id;
locktag.locktag_field2 = schema_name_hash;
locktag.locktag_field3 = object_name_hash;

res =

LockAcquire(&locktag,AccessExclusiveLock…);
```

Nile

How do we distribute locks?



Nile

Avoiding deadlocks and contention

- Identical lock order on all databases
- Always start with same database
- LOCK_TIMEOUT

Nile

Handling Failures

Nile

2 types of failures

- Failures before PRECOMMIT phase succeeded
- Failures after PRECOMMIT phase succeeded

Nile

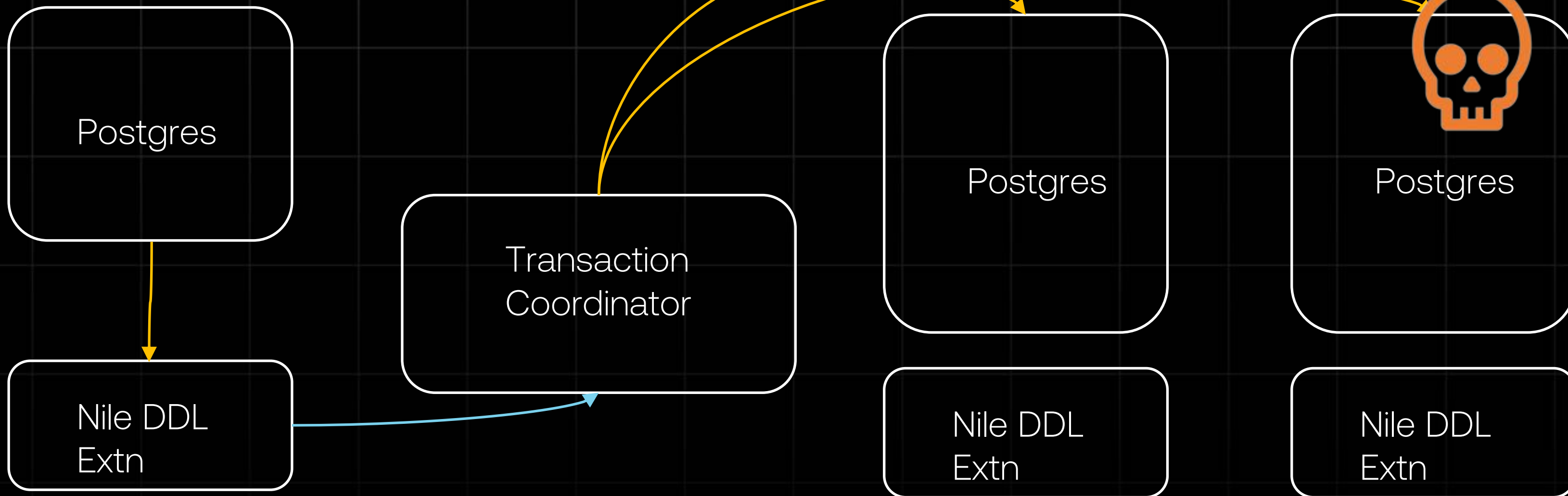
Before PRECOMMIT phase succeeded

- Every error at this point will trigger a rollback
  - DDL errors
  - Timeouts
  - Crashes
- Which the transaction coordinator can distribute
- Exactly same way as distributing a commit

After PRECOMMIT phase succeeded

- Very rare to have COMMIT PREPARED fail
- General solution is full distributed lock manager (Raft, Paxos)
- We are not doing this (yet)
- Instead, we are eventually consistent

Nile



Nile

What do we do?

- Coordinator has the exact state of each transaction
- Can retry a few times
- Fence node (to hide missing tables)
- Eventually return success to user
- Continue retrying once node is back

Serverless Postgres for modern SaaS

Launch In Days. Grow For Years

This is OK because SaaS developers don't do DDL

They run schema migrations.

Nile

Schema migrations

- Either:
  - Stop app, upgrade schema, start new version of app
  - Make backward compatible changes, start new version of app, repeat
- Either way, they are unlikely to experience issues

Nile

To sum it up

Extensions

You can build extensions to handle DDLs via the processUtility hook. Transactions are handled in the XactCallback. The events map to 2PC.

Distributed Txn

It is relatively straight forward to distribute DDLs and transactions. Failure handling is where it gets tricky.

Locks

Locks are surprisingly nuanced. But avoiding long locks and deadlocks is important for developer experience. So we pulled all the tricks.

Thank You!

Follow us on:

Discord

discord.gg/8UuBB84tTy

X (Twitter)

twitter.com/niledatabase

Linkedin

https://www.linkedin.com/company/niledatabase/