

JUST ENOUGH POSTGRES COMPATIBILITY TO KEEP YOUR SANITY

Jelte Fennema-Nio (JelteF)

2026-09-30



Who am I?

- Software Engineer @ MotherDuck
- Maintainer of pg_duckdb
- Maintainer of PgBouncer
- Contributing to DuckDB & Postgres
- Worked on Citus @ Microsoft



Who am I?

- Software Engineer @ MotherDuck
- Maintainer of pg_duckdb
- Maintainer of PgBouncer
- Contributing to DuckDB & Postgres
- Worked on Citus @ Microsoft
- All of which are "Postgres compatible"



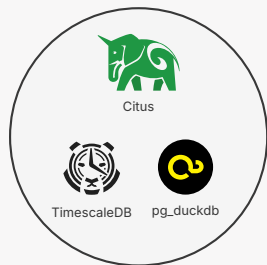
What am I going to talk about?

1. What does "Postgres compatible" mean?
2. The Postgres protocol
3. PgBouncer
4. MotherDuck Postgres Endpoint
5. Building your own Postgres compatible database

“Postgres compatible” databases

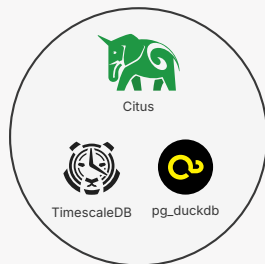
“Postgres compatible” databases

Extensions

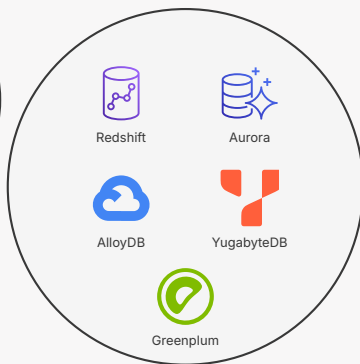


"Postgres compatible" databases

Extensions

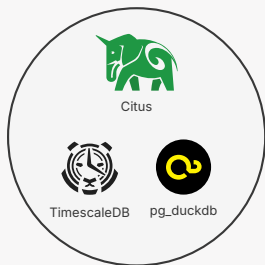


Forks

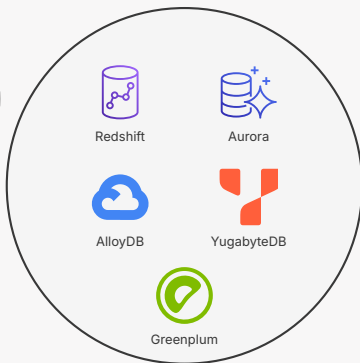


"Postgres compatible" databases

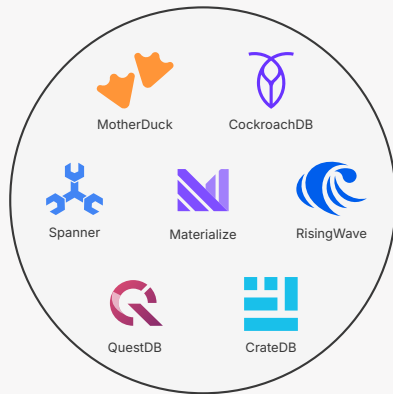
Extensions



Forks



Re-implementations



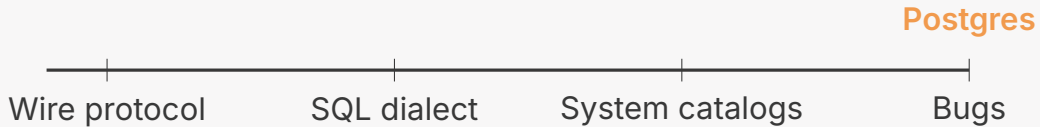
"Only PostgreSQL is fully Postgres compatible."



The compatibility line



The compatibility line



THE PROTOCOL

01010001	00000000
00000000	00000000
00001101	01010011
01000101	01001100 ...

Protocol 101

- TCP connection == database session

Protocol 101

- TCP connection == database session
- One byte message type, four byte length, payload

```
Byte  Int32  ...  
+-----+-----+-----+  
| 'Q' | len  | "SELECT 1\0" |  
+-----+-----+-----+
```

Startup



Client



Server



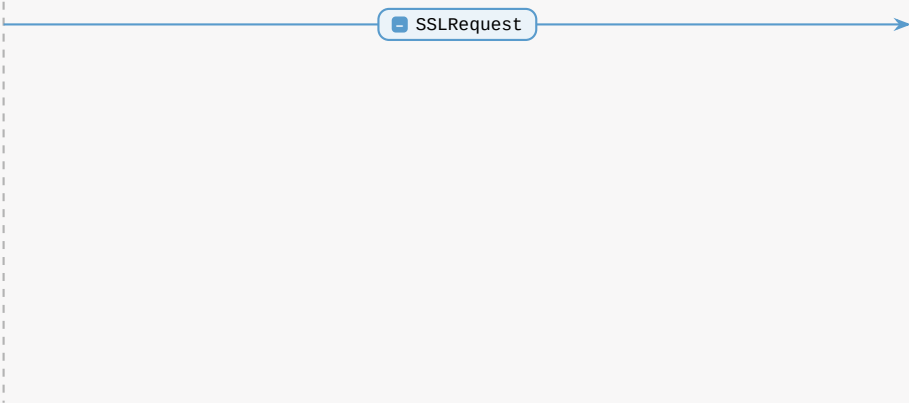
Startup

 **Client**

 **Server**

TLS

 SSLRequest



Startup

 Client

 Server

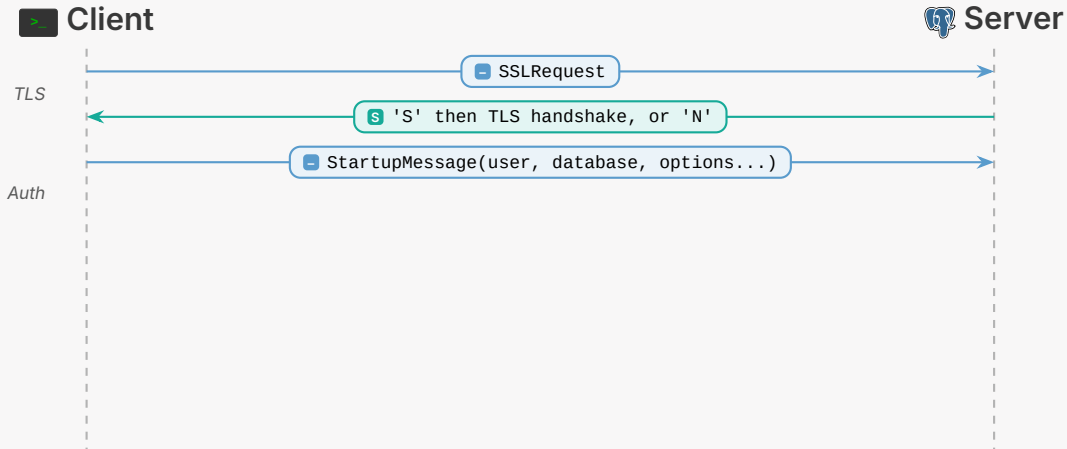
TLS

 SSLRequest

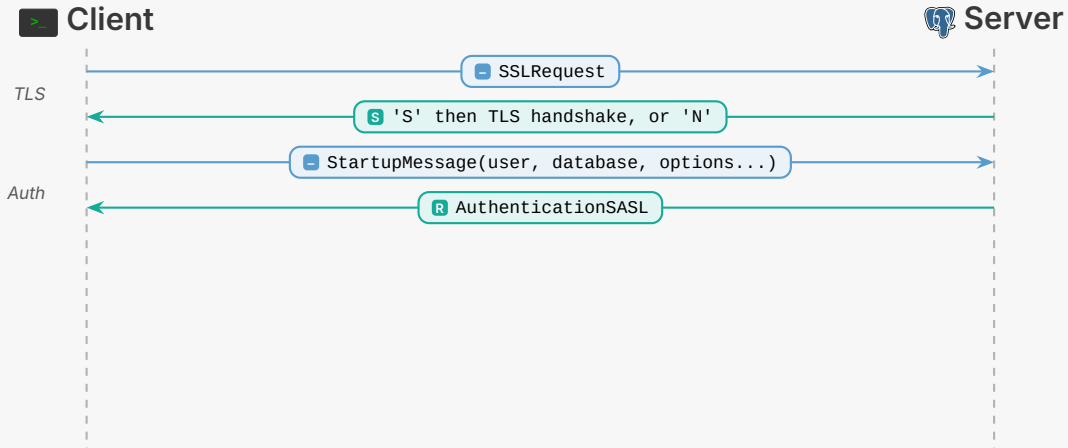
 'S' then TLS handshake, or 'N'



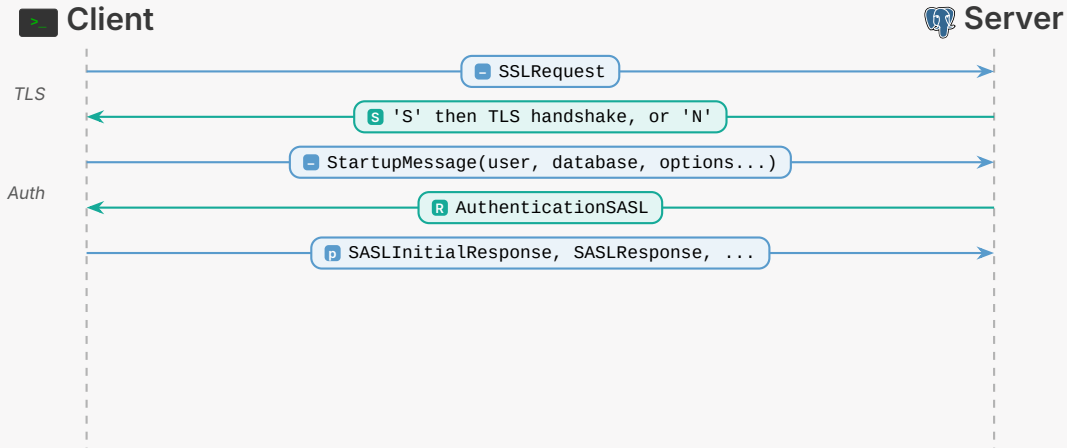
Startup



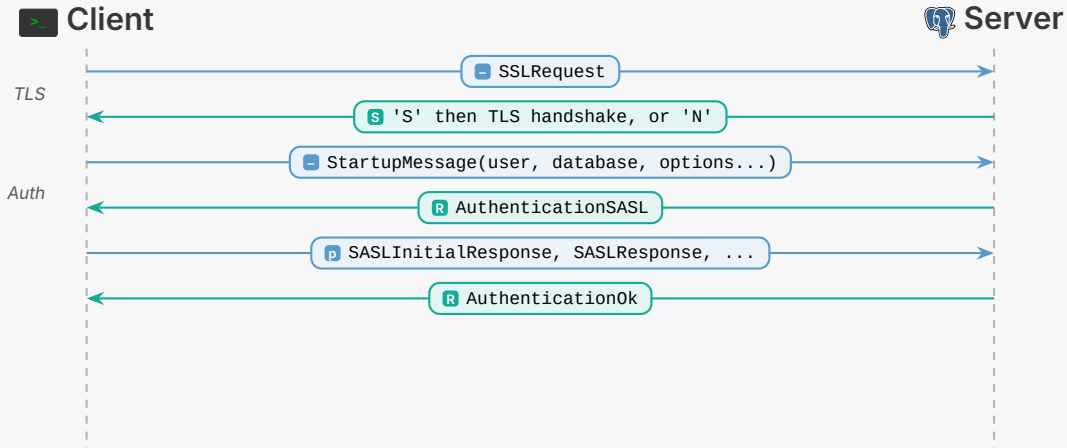
Startup



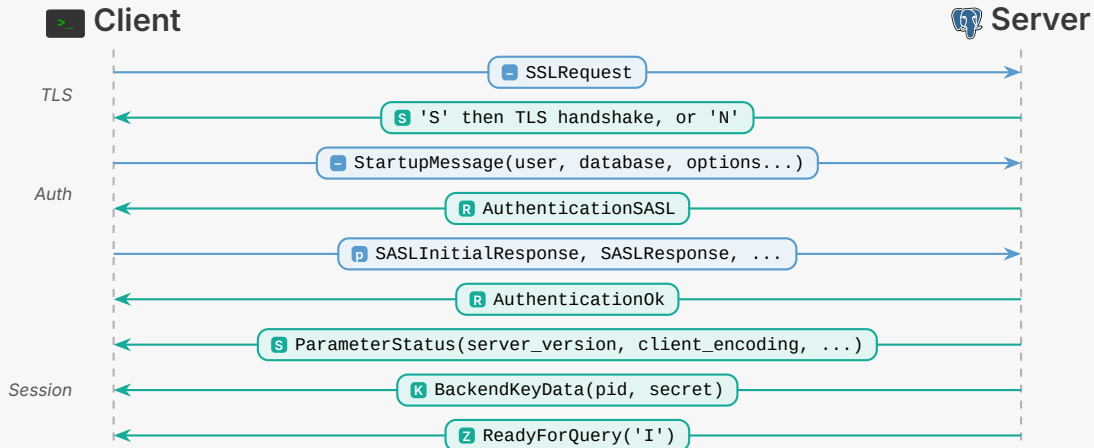
Startup



Startup



Startup



Simple query protocol

 Client



 Server



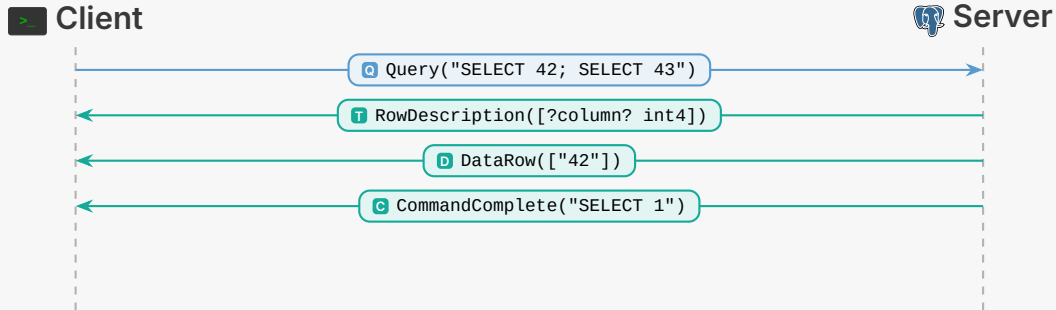
Simple query protocol

 Client

 Server

 Query("SELECT 42; SELECT 43")

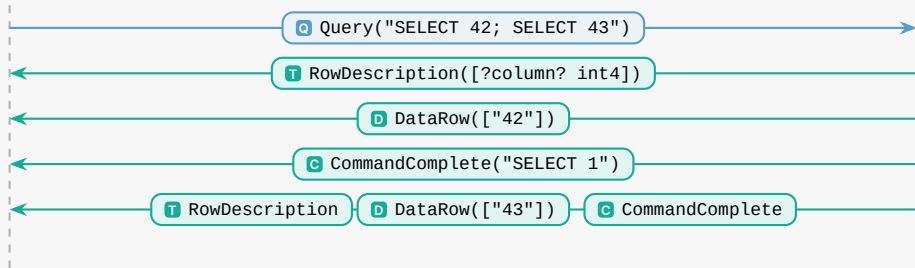
Simple query protocol



Simple query protocol

 Client

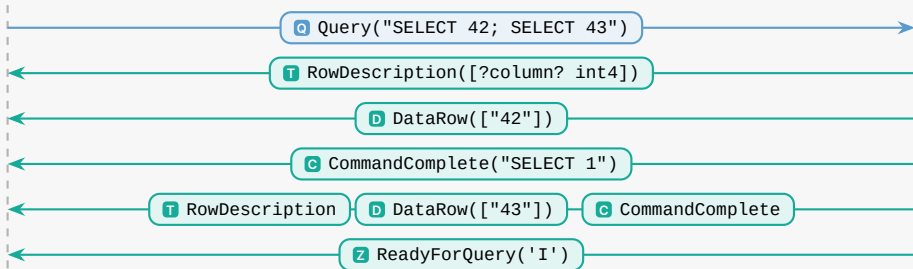
 Server



Simple query protocol

 Client

 Server



Extended query protocol

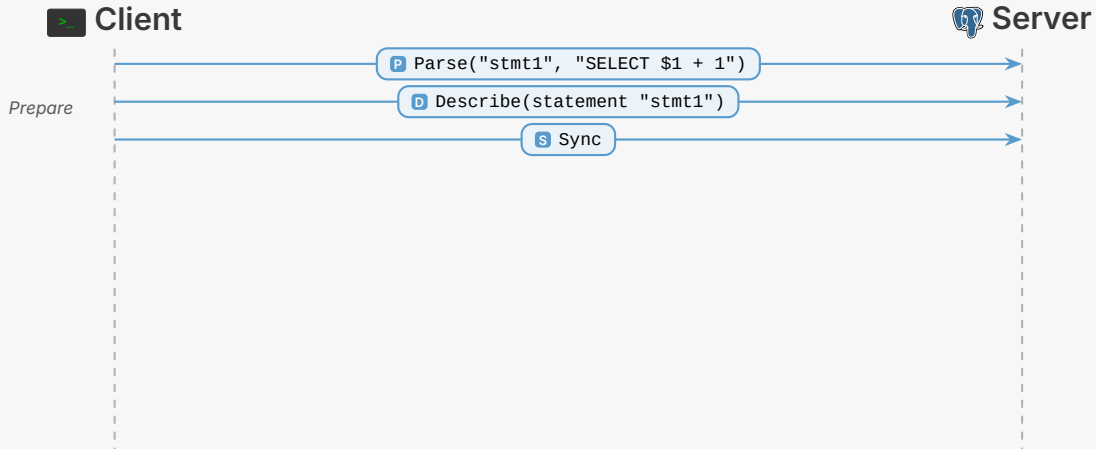
 Client



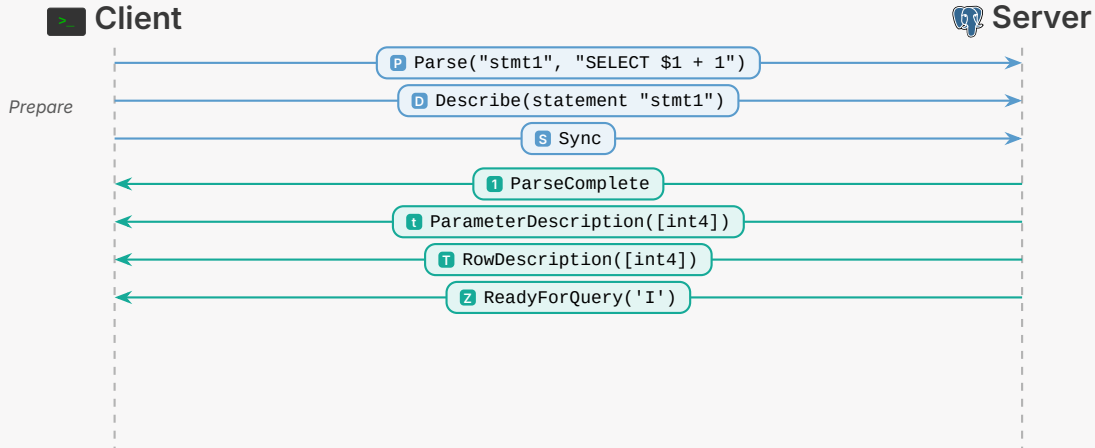
 Server



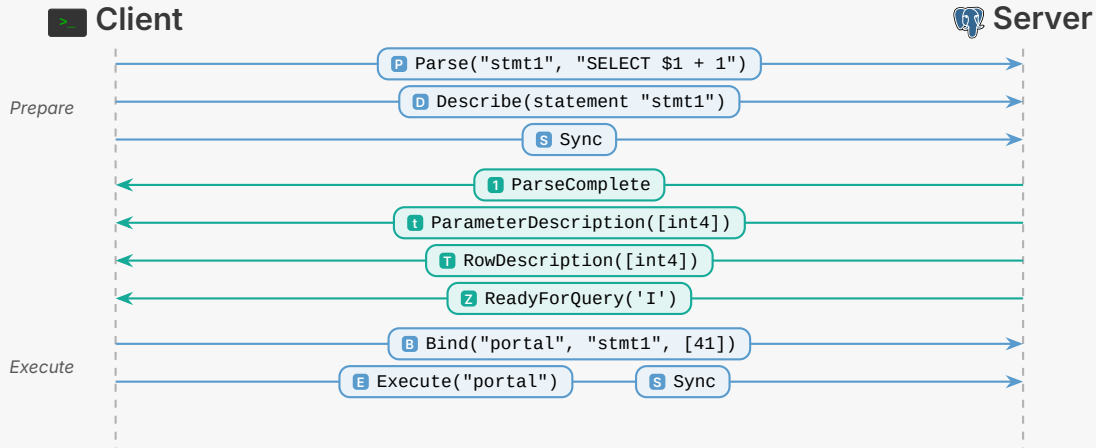
Extended query protocol



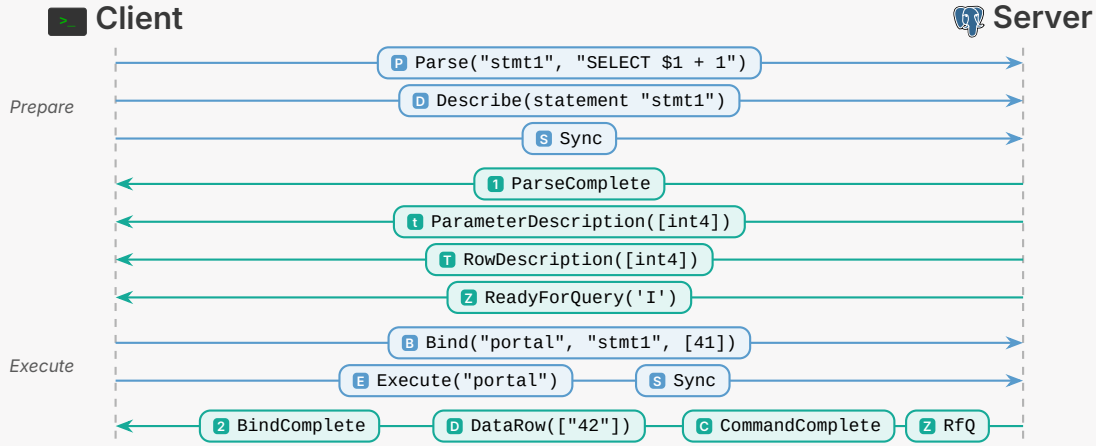
Extended query protocol



Extended query protocol



Extended query protocol



The rest

- Terminate
- CancelRequest, on a new connection
- COPY
- LISTEN / NOTIFY
- Replication

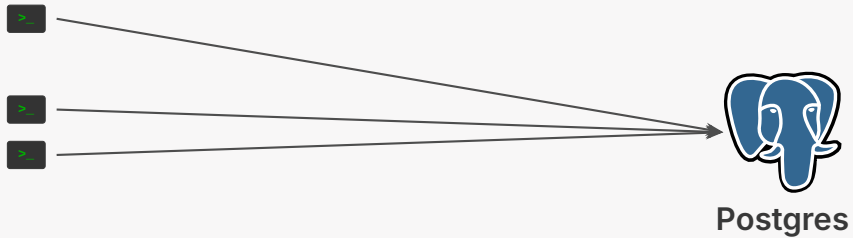
PGBOUNCER



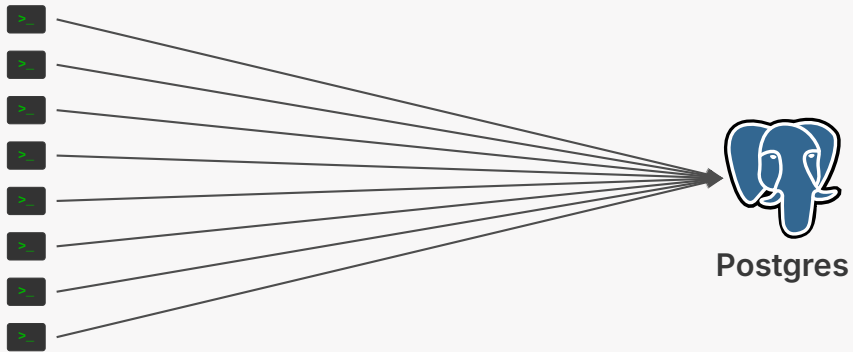
What is PgBouncer?



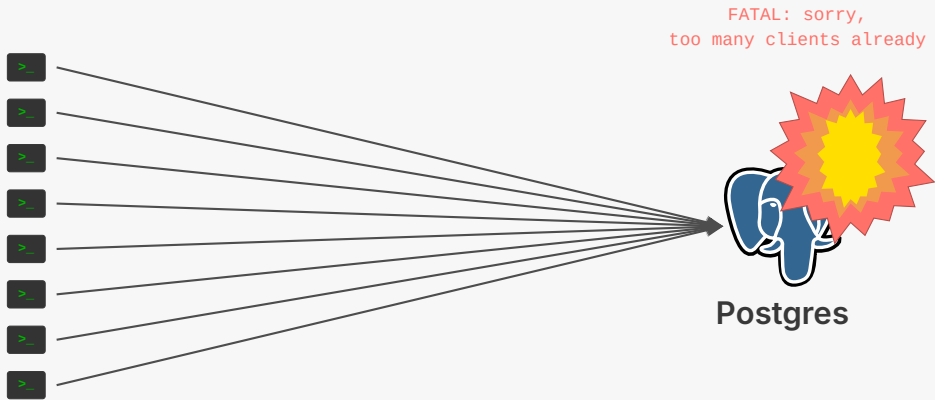
What is PgBouncer?



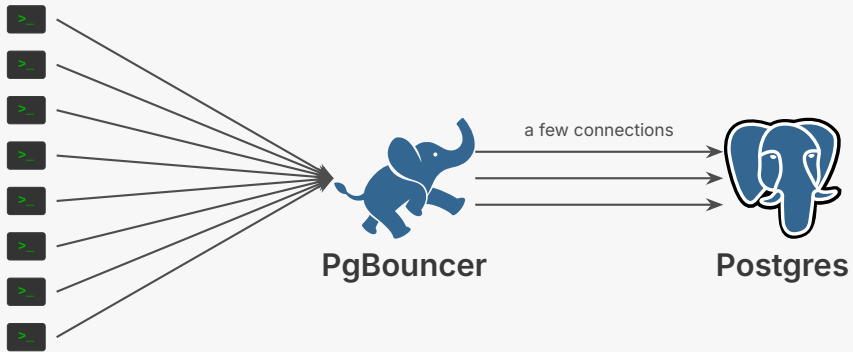
What is PgBouncer?



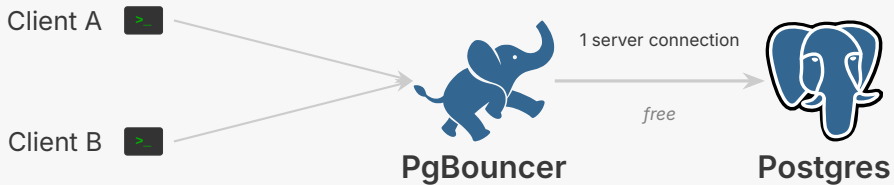
What is PgBouncer?



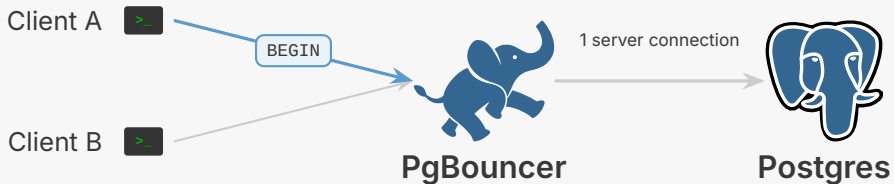
What is PgBouncer?



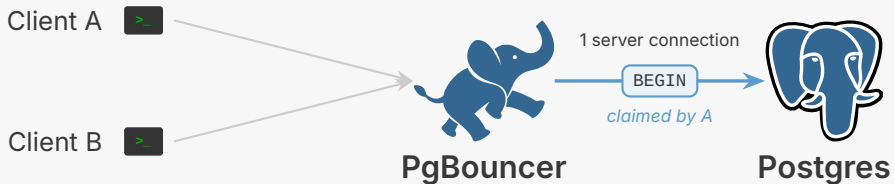
Transaction pooling



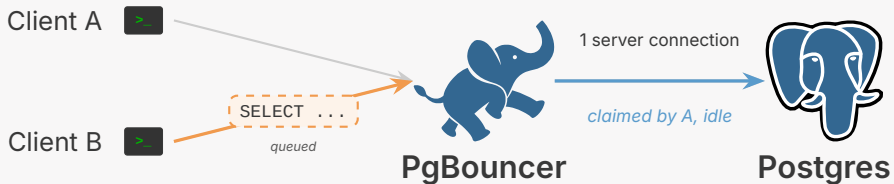
Transaction pooling



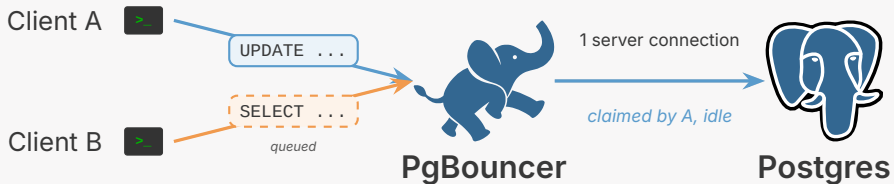
Transaction pooling



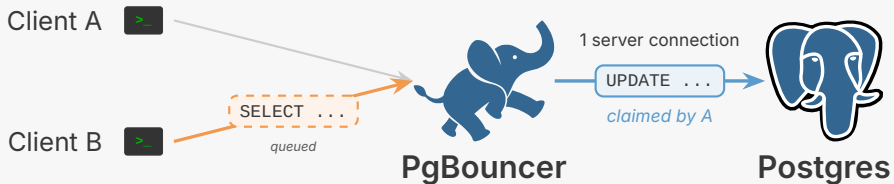
Transaction pooling



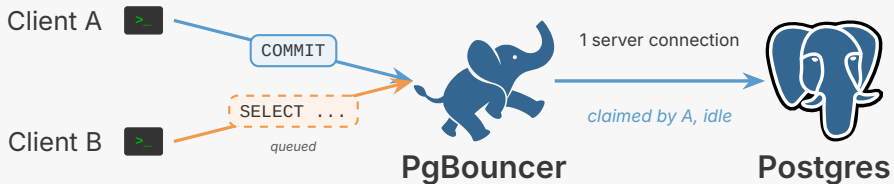
Transaction pooling



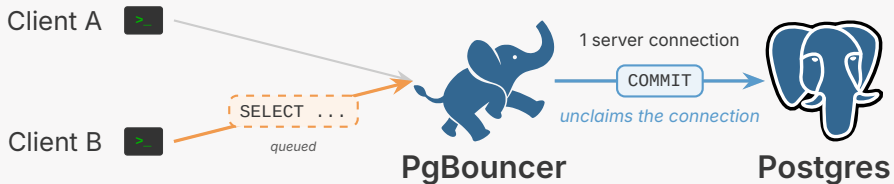
Transaction pooling



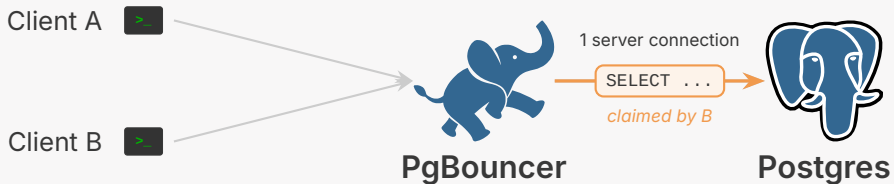
Transaction pooling



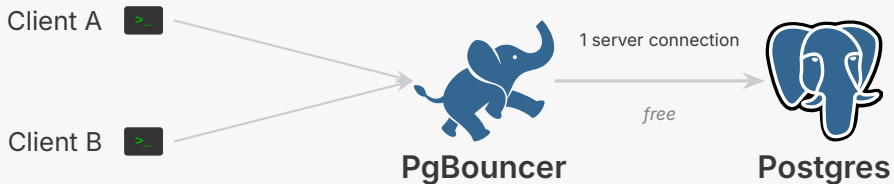
Transaction pooling



Transaction pooling

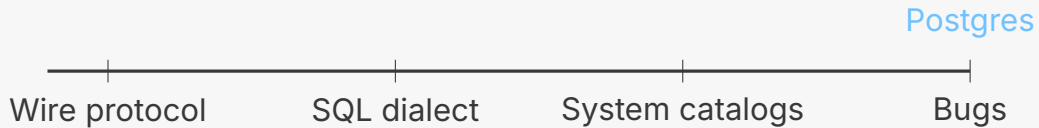


Transaction pooling

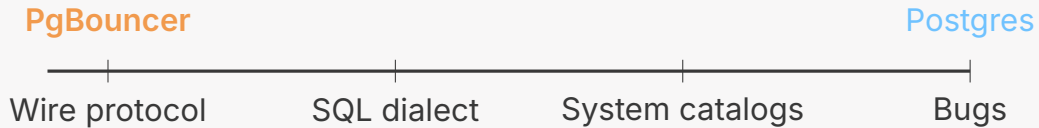


A client holds the server connection for the whole transaction, even while idle

PgBouncer on the line

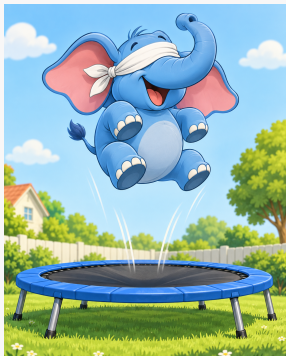


PgBouncer on the line



Design decision

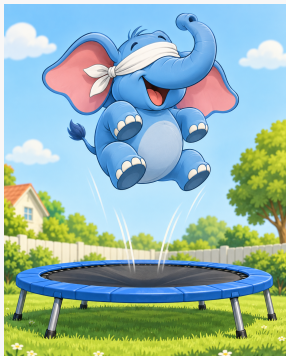
“PgBouncer never looks inside a query string.”



Design decision

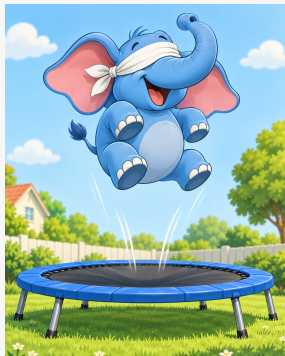
“PgBouncer never looks inside a query string.”

- No parser to keep in sync with Postgres
- No CPU spent on parsing



Design decision

“PgBouncer never looks inside a query string.”



- No parser to keep in sync with Postgres
- No CPU spent on parsing
- But transaction pooling needs to know about session state

Hoop 1: Transaction state

 Client

⋮

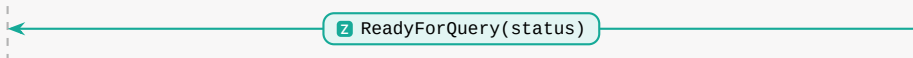
 Server

⋮

Hoop 1: Transaction state

 Client

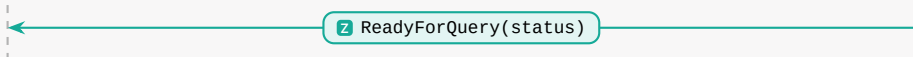
 Server



Hoop 1: Transaction state

 Client

 Server

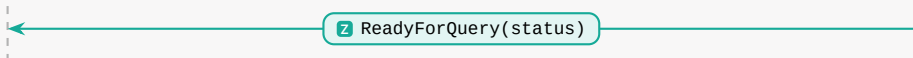


```
ReadyForQuery('I')    idle
ReadyForQuery('T')    in transaction
ReadyForQuery('E')    in failed transaction
```

Hoop 1: Transaction state

 Client

 Server



```
ReadyForQuery('I')    idle
ReadyForQuery('T')    in transaction
ReadyForQuery('E')    in failed transaction
```

- Server connection goes back to the pool on 'I'
- No need to recognise BEGIN, COMMIT, ROLLBACK, END, ABORT, ...

Hoop 2: SET

```
SET TimeZone = 'Europe/Amsterdam';
```

Hoop 2: SET

```
SET TimeZone = 'Europe/Amsterdam';
```

- Changes the *server* connection
- The next client on that connection inherits it
- PgBouncer never saw the SET

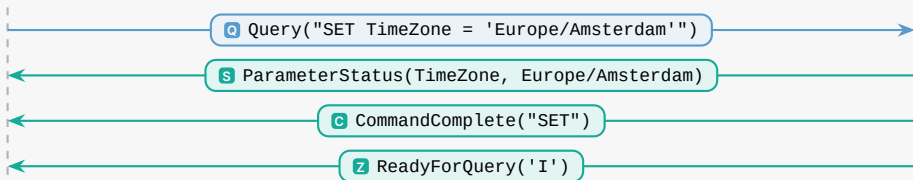
Hoop 2: ParameterStatus to the rescue



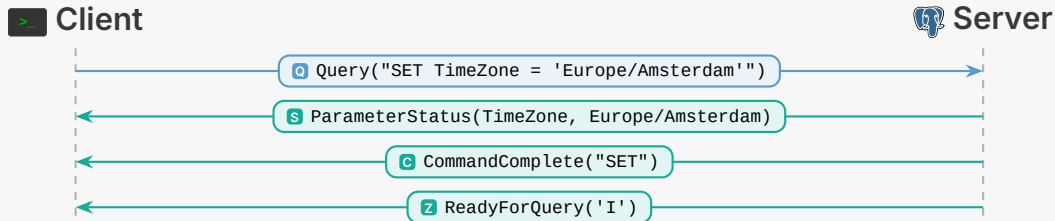
Hoop 2: ParameterStatus to the rescue

 Client

 Server



Hoop 2: ParameterStatus to the rescue



- The server reports some settings: at startup, and whenever they change
- Only the ones marked `GUC_REPORT: client_encoding, DateStyle, TimeZone, server_version, ...`

Hoop 2: SET

- ParameterStatus tells PgBouncer about the reported settings
- PgBouncer remembers them per client
- And replays them when that client gets a new server connection

Hoop 2: SET

- ParameterStatus tells PgBouncer about the reported settings
- PgBouncer remembers them per client
- And replays them when that client gets a new server connection
- Everything else: SET LOCAL, or accept the pain

MOTHERDUCK POSTGRES ENDPOINT



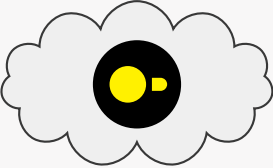
What is DuckDB?

	Embedded	Client-server
Transactional	 SQLite	 Postgres
Analytics		

What is DuckDB?

	Embedded	Client-server
Transactional	 SQLite	 Postgres
Analytics	 DuckDB	

What is DuckDB?

	Embedded	Client-server
Transactional	 SQLite	 Postgres
Analytics	 DuckDB	

What is DuckDB?

	Embedded	Client-server
Transactional	 SQLite	 Postgres
Analytics	 DuckDB	 MotherDuck

What is the MotherDuck Postgres Endpoint?

- BI tools, ORMs and legacy apps speak Postgres

What is the MotherDuck Postgres Endpoint?

- BI tools, ORMs and legacy apps speak Postgres

“Make MotherDuck pretend to be Postgres.”



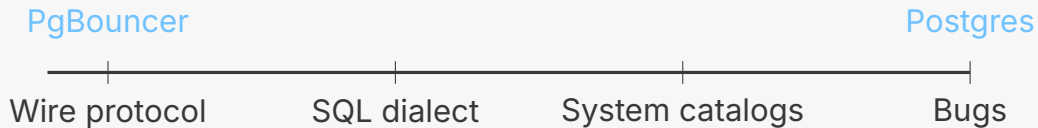
DuckDB is already close to Postgres

- DuckDB's SQL is modelled on Postgres
- Deviates on purpose only with good reason

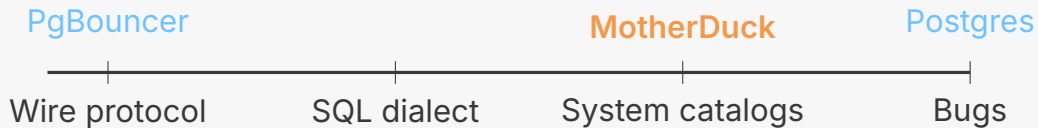
And deviates by accident too



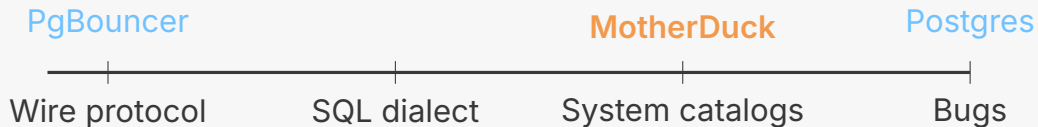
MotherDuck on the line



MotherDuck on the line

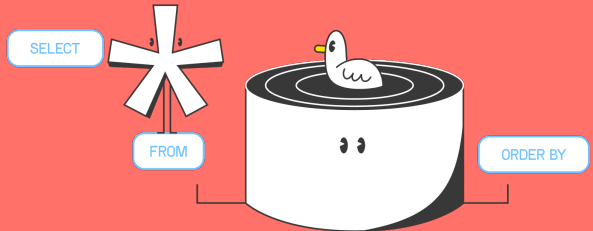


MotherDuck on the line



- Where does the protocol *end*?

THE SQL DIALECT



The SQL dialect bleeds into the protocol



Type OIDs

OID	Type
16	bool
23	int4
25	text
1184	timestamptz
1700	numeric

Type OIDs

OID	Type
16	bool
23	int4
25	text
1184	timestamptz
1700	numeric

- Not part of the protocol spec!

Text or binary?

- Simple query protocol: always text
- Extended query protocol: client picks per column
- The spec says how to choose, but not what the values look like

Text or binary?

- Simple query protocol: always text
- Extended query protocol: client picks per column
- The spec says how to choose, but not what the values look like

In practice both formats and type OIDs
are part of the protocol.

Text format

Type	Postgres	DuckDB
bool	t	true
int4[]	{1, 2, 3}	[1, 2, 3]
text[]	{"a b", NULL, "\"q\""}	[a b, NULL, "q"]
bytea	\x0102	\x01\x02
struct	(1, foo)	{'a': 1, 'b': foo}

Binary format

- Documented nowhere: "just look at the code"

Binary format

- Documented nowhere: "just look at the code"
- `timestamp`: microseconds since 2000-01-01
- `numeric`: base 10000 digits, weight, sign, scale

Binary format

- Documented nowhere: "just look at the code"
- `timestamp`: microseconds since 2000-01-01
- `numeric`: base 10000 digits, weight, sign, scale
- No escape hatch: a client that asks for binary expects binary
- Default for `pgx` (Go), `Npgsql` (.NET), `asyncpg` (Python)

SET commands you don't have

```
SET extra_float_digits = 3;           -- JDBC
SET application_name = '...';
SET datestyle TO 'ISO';               -- psycopg2
SET statement_timeout = 30000;        -- ORMs
SET ssl_renegotiation_limit = 0;      -- old JDBC
SET ROLE NONE;
```

- Sent before the first real query

ParameterStatus at startup

```
ParameterStatus(server_version, 18.2)
ParameterStatus(client_encoding, UTF8)
ParameterStatus(standard_conforming_strings, on)
ParameterStatus(integer_datetimes, on)
ParameterStatus(DateStyle, ISO, YMD)
ParameterStatus(TimeZone, UTC)
```

ParameterStatus at startup

```
ParameterStatus(server_version, 18.2)
ParameterStatus(client_encoding, UTF8)
ParameterStatus(standard_conforming_strings, on)
ParameterStatus(integer_datetimes, on)
ParameterStatus(DateStyle, ISO, YMD)
ParameterStatus(TimeZone, UTC)
```

- Clients rely on these even if they never SET anything
- `standard_conforming_strings`: how libpq and JDBC escape strings
- `DateStyle`: how text dates are parsed
- `integer_datetimes`: how binary timestamps are decoded

Version

What DuckDB says:

```
SELECT version();  
"version"()  
-----  
v1.5.5
```

Version

What DuckDB says:

```
SELECT version();  
"version"()  
-----  
v1.5.5
```

What clients do with it:

```
PostgreSQL (\d+)\.(\d+)      -- no match, or worse: Postgres 1.5
```

Version, take two

```
SELECT version();
```

```
          "version"()
```

```
-----  
PostgreSQL 18.2 on MotherDuck running DuckDB v1.5.5
```

Version, take two

```
SELECT version();
```

```
          "version"()
```

```
-----  
PostgreSQL 18.2 on MotherDuck running DuckDB v1.5.5
```

knex, a Node.js query builder:

```
Cannot read properties of null (reading '1')
```

Column names

Without an alias, DuckDB names a column after the expression:

```
SELECT 1 + 1, version();  
(1 + 1) | "version"()  
-----+-----  
2 | PostgreSQL 18.2 on ...
```

Column names

Without an alias, DuckDB names a column after the expression:

```
SELECT 1 + 1, version();  
(1 + 1) | "version"()  
-----+-----  
2 | PostgreSQL 18.2 on ...
```

Postgres doesn't:

```
SELECT 1 + 1, version();  
?column? | version  
-----+-----  
2 | PostgreSQL 18.2 on ...
```

Version, take three

```
SELECT version();
```

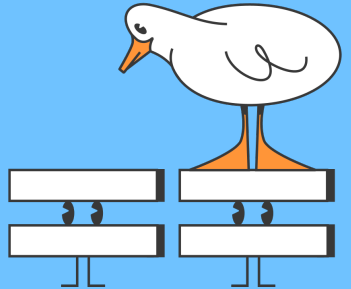
```
version
```

```
-----  
PostgreSQL 18.2 on MotherDuck running DuckDB v1.5.5
```

Compatibility is whack-a-mole



SYSTEM CATALOGS



System catalogs

- `information_schema` is *SQL Standard*
- DuckDB has it and plenty of tools use it

System catalogs

- information_schema is *SQL Standard*
- DuckDB has it and plenty of tools use it
- But most tools also need something Postgres specific like:

```
SELECT udt_name FROM information_schema.columns WHERE ...  
-- int4, _text, timestamptz: Postgres' internal type names
```

Small: functions that do nothing

- `pg_get_expr` deparses a stored expression tree, prettily if you ask

```
SELECT pg_get_expr('some_expr', 0, true);  
-- some_expr
```

Small: views that can return nothing

```
CREATE VIEW pg_catalog.pg_roles AS  
SELECT NULL::oid AS oid, NULL::name AS rolname, ...  
WHERE false;
```

Small: views that can return nothing

```
CREATE VIEW pg_catalog.pg_roles AS  
SELECT NULL::oid AS oid, NULL::name AS rolname, ...  
WHERE false;
```

- pg_roles, pg_range, pg_inherits, pg_policy, ...
- Zero rows is fine, a missing table is not

Medium: the reg types

```
SELECT 'orders'::regclass::oid;    -- 16384
```

Medium: the reg types

```
SELECT 'orders'::regclass::oid;      -- 16384
```

```
SELECT 16384::regclass;              -- orders
```

Medium: the reg types

```
SELECT 'orders'::regclass::oid;      -- 16384
```

```
SELECT 16384::regclass;              -- orders
```

```
SELECT indexrelid::regclass FROM pg_index  
-- indrelid is a plain oid column  
WHERE indrelid = 'orders'::regclass;
```

Medium: the reg types

```
SELECT 'orders'::regclass::oid;      -- 16384
```

```
SELECT 16384::regclass;              -- orders
```

```
SELECT indexrelid::regclass FROM pg_index  
-- indrelid is a plain oid column  
WHERE indrelid = 'orders'::regclass;
```

- regclass for tables, regproc for functions, regtype for types, ...

Medium: the reg types and the search_path

- Name resolution follows the search_path in both directions

Medium: the reg types and the search_path

- Name resolution follows the search_path in both directions

```
SET search_path = shop;
```

```
SELECT 16384::regclass;           -- orders
```

```
SELECT 'orders'::regclass::oid;  -- 16384
```

Medium: the reg types and the search_path

- Name resolution follows the search_path in both directions

```
SET search_path = shop;
```

```
SELECT 16384::regclass;           -- orders
```

```
SELECT 'orders'::regclass::oid;  -- 16384
```

```
SET search_path = archive;
```

```
SELECT 16384::regclass;           -- shop.orders
```

```
SELECT 'shop.orders'::regclass::oid; -- 16384
```

Medium: type loading queries

- The common type OIDs are well-known, drivers hardcode them

Medium: type loading queries

- The common type OIDs are well-known, drivers hardcode them
- Some drivers also look at `pg_type` for types they don't know about

Medium: type loading queries

- The common type OIDs are well-known, drivers hardcode them
- Some drivers also look at `pg_type` for types they don't know about
- `pg_type` has ~30 columns, and not one says "this is an array"

Medium: type loading queries

- The common type OIDs are well-known, drivers hardcode them
- Some drivers also look at pg_type for types they don't know about
- pg_type has ~30 columns, and not one says "this is an array"

```
-- pgjdbc (Java): is this an array type?  
SELECT typinput = 'pg_catalog.array_in'::regproc  
FROM pg_type WHERE typename = '_int4';
```

Medium: type loading queries

```
-- Npgsql (.NET): same question, different way
LEFT JOIN pg_proc AS proc ON proc.oid = typ.typereceive
CASE WHEN proc.proname = 'array_recv' THEN 'a' ...
```

Medium: type loading queries

```
-- Npgsql (.NET): same question, different way  
LEFT JOIN pg_proc AS proc ON proc.oid = typ.typereceive  
CASE WHEN proc.proname = 'array_recv' THEN 'a' ...
```

- So pg_type and pg_proc need dummy data for this...
- For functions that they don't have

Medium: `pg_attribute.atttypmod`

The type modifier, next to the type OID, for every column of every table:

```
DECIMAL(18,6)    -> ((18 << 16) | 6) + 4 = 1179658  
VARCHAR(100)     -> 100 + 4              = 104  
anything else    -> -1
```

Medium: `pg_attribute.atttypmod`

The type modifier, next to the type OID, for every column of every table:

```
DECIMAL(18,6)    -> ((18 << 16) | 6) + 4 = 1179658
VARCHAR(100)     -> 100 + 4              = 104
anything else    -> -1
```

- `format_type(atttypid, atttypmod)` decodes it, JDBC computes `COLUMN_SIZE` from it
- Documented nowhere either, time to read the source code

Big: primary keys

pgjdbc's `getPrimaryKeys()`, as sent by every JDBC based tool:

```
SELECT ...,
    (information_schema._pg_expandarray(i.indkey)).n AS KEY_SEQ,
    ...
FROM pg_catalog.pg_class ct
JOIN pg_catalog.pg_attribute a ON (ct.oid = a.attrelid)
JOIN pg_catalog.pg_namespace n ON (ct.relnamespace = n.oid)
JOIN pg_catalog.pg_index i ON (a.attrelid = i.indrelid)
JOIN pg_catalog.pg_class ci ON (ci.oid = i.indexrelid)
WHERE n.nspname = 'main' AND ct.relname = 'orders'
AND i.indisprimary
```

Big: primary keys

pgjdbc's `getPrimaryKeys()`, as sent by every JDBC based tool:

```
SELECT ...,
  (information_schema._pg_expandarray(i.indkey)).n AS KEY_SEQ,
  ...
FROM pg_catalog.pg_class ct
JOIN pg_catalog.pg_attribute a ON (ct.oid = a.attrelid)
JOIN pg_catalog.pg_namespace n ON (ct.relnamespace = n.oid)
JOIN pg_catalog.pg_index i ON (a.attrelid = i.indrelid)
JOIN pg_catalog.pg_class ci ON (ci.oid = i.indexrelid)
WHERE n.nspname = 'main' AND ct.relname = 'orders'
      AND i.indisprimary
```

Big: primary keys

pgjdbc's `getPrimaryKeys()`, as sent by every JDBC based tool:

```
SELECT ...,
  (information_schema._pg_expandarray(i.indkey)).n AS KEY_SEQ,
  ...
FROM pg_catalog.pg_class ct
JOIN pg_catalog.pg_attribute a ON (ct.oid = a.attrelid)
JOIN pg_catalog.pg_namespace n ON (ct.relnamespace = n.oid)
JOIN pg_catalog.pg_index i ON (a.attrelid = i.indrelid)
JOIN pg_catalog.pg_class ci ON (ci.oid = i.indexrelid)
WHERE n.nspname = 'main' AND ct.relname = 'orders'
      AND i.indisprimary
```

Final boss: psql

\d orders, query 1: which table do you mean?

```
SELECT c.oid, n.nspname, c.relname
FROM pg_catalog.pg_class c
      LEFT JOIN pg_catalog.pg_namespace n
            ON n.oid = c.relnamespace
WHERE c.relname OPERATOR(pg_catalog.~) '^(orders)$'
      COLLATE pg_catalog.default
      AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 2, 3;
```

Final boss: psql

\d orders, query 1: which table do you mean?

```
SELECT c.oid, n.nspname, c.relname
FROM pg_catalog.pg_class c
      LEFT JOIN pg_catalog.pg_namespace n
            ON n.oid = c.relnamespace
WHERE c.relname OPERATOR(pg_catalog.~) '^(orders)$'
      COLLATE pg_catalog.default
      AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 2, 3;
```

Final boss: psql

\d orders, query 1: which table do you mean?

```
SELECT c.oid, n.nspname, c.relname
FROM pg_catalog.pg_class c
      LEFT JOIN pg_catalog.pg_namespace n
            ON n.oid = c.relnamespace
WHERE c.relname OPERATOR(pg_catalog.~) '^(orders)$'
      COLLATE pg_catalog.default
      AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 2, 3;
```

Final boss: psql

\d orders, query 1: which table do you mean?

```
SELECT c.oid, n.nspname, c.relname
FROM pg_catalog.pg_class c
      LEFT JOIN pg_catalog.pg_namespace n
            ON n.oid = c.relnamespace
WHERE c.relname OPERATOR(pg_catalog.~) '^(orders)$'
      COLLATE pg_catalog.default
      AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 2, 3;
```

Final boss: psql

Query 2: what kind of table is it?

```
SELECT c.relchecks, c.relkind, c.relhasindex,  
       c.relhasrules, c.relhastriggers, c.relrowsecurity,  
       c.relforcerowsecurity, false AS relhasoids,  
       c.relispartition, '', c.reltablespace,  
       CASE WHEN c.reloftype = 0 THEN '' ELSE  
         c.reloftype::pg_catalog.regtype::pg_catalog.text END,  
       c.relpersistence, c.relreplident, am.amname  
FROM pg_catalog.pg_class c  
LEFT JOIN pg_catalog.pg_class tc ON (c.reltoastrelid = tc.oid)  
LEFT JOIN pg_catalog.pg_am am ON (c.relam = am.oid)  
WHERE c.oid = '16384';
```

Final boss: psql

Query 2: what kind of table is it?

```
SELECT c.relchecks, c.relkind, c.relhasindex,  
       c.relhasrules, c.relhastriggers, c.relrowsecurity,  
       c.relforcerowsecurity, false AS relhasoids,  
       c.relispartition, '', c.reltablespace,  
       CASE WHEN c.reloftype = 0 THEN '' ELSE  
         c.reloftype::pg_catalog.regtype::pg_catalog.text END,  
       c.relpersistence, c.relreplident, am.amname  
FROM pg_catalog.pg_class c  
LEFT JOIN pg_catalog.pg_class tc ON (c.reltoastrelid = tc.oid)  
LEFT JOIN pg_catalog.pg_am am ON (c.relam = am.oid)  
WHERE c.oid = '16384';
```

Final boss: psql

Query 2: what kind of table is it?

```
SELECT c.relchecks, c.relkind, c.relhasindex,  
       c.relhasrules, c.relhastriggers, c.relrowsecurity,  
       c.relforcerowsecurity, false AS relhasoids,  
       c.relispartition, '', c.reltablespace,  
       CASE WHEN c.reloftype = 0 THEN '' ELSE  
         c.reloftype::pg_catalog.regtype::pg_catalog.text END,  
       c.relpersistence, c.relreplident, am.amname  
FROM pg_catalog.pg_class c  
LEFT JOIN pg_catalog.pg_class tc ON (c.reltoastrelid = tc.oid)  
LEFT JOIN pg_catalog.pg_am am ON (c.relam = am.oid)  
WHERE c.oid = '16384';
```

Final boss: psql

And then four more queries like that...

Final boss: psql

**And then four more queries like that...
...for a single backslash command.**

Final boss: psql

**And then four more queries like that...
...for a single backslash command.**

- And we haven't even talked about tab completion, which does similar things

Where MotherDuck draws the line

"I won't stop until we support \d"

Where MotherDuck draws the line

Things DuckDB doesn't support (yet), and clients ask for anyway:

- Cursors: `DECLARE ... CURSOR, FETCH 100 FROM ...`
- `SERIALIZABLE` isolation
- Range types, `LISTEN / NOTIFY`
- `client_encoding` other than UTF-8

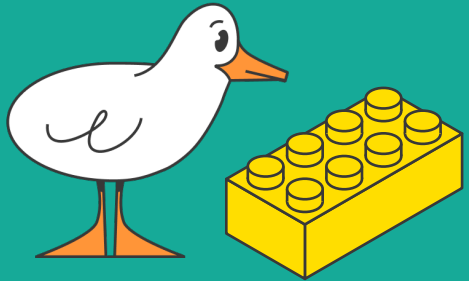
Where MotherDuck draws the line

Things DuckDB doesn't support (yet), and clients ask for anyway:

- Cursors: `DECLARE ... CURSOR, FETCH 100 FROM ...`
- `SERIALIZABLE` isolation
- Range types, `LISTEN / NOTIFY`
- `client_encoding` other than UTF-8

A clear error beats a wrong answer.

BUILDING YOUR OWN



If you want to feel this pain too



If you want to feel this pain too

<https://github.com/jeroenrinzema/psql-wire>

- Pure Go Postgres server wire protocol
- You provide: a function from query string to rows

Hello world: the echo database

Supports exactly one statement: `SELECT '<anything>';`

```
var selectLiteral = regexp.MustCompile(  
    `(?is)^\s*SELECT\s+'(.*)'\s*;?\s*$`)  
var columns = wire.Columns{  
    {Name: "echo", Oid: pgtype.TextOID, Width: -1}}  
  
func main() {  
    wire.ListenAndServe("127.0.0.1:5433", handler)  
}
```

Hello world: the handler

```
func handler(ctx context.Context, q wire.Query) (  
    wire.PreparedStatements, error) {  
    m := selectLiteral.FindStringSubmatch(q.Query)
```

Hello world: the handler

```
func handler(ctx context.Context, q wire.Query) (
    wire.PreparedStatements, error) {
    m := selectLiteral.FindStringSubmatch(q.Query)
    if m == nil {
        err := errors.New("only SELECT '<literal>' is supported")
        return nil, psqlerr.WithCode(err, codes.FeatureNotSupported)
    }
}
```

Hello world: the handler

```
func handler(ctx context.Context, q wire.Query) (
    wire.PreparedStatements, error) {
    m := selectLiteral.FindStringSubmatch(q.Query)
    if m == nil {
        err := errors.New("only SELECT '<literal>' is supported")
        return nil, psqlerr.WithCode(err, codes.FeatureNotSupported)
    }
    handle := func(ctx context.Context, w wire.DataWriter,
        params []wire.Parameter) error {
        w.Row([]any{m[1]})
        return w.Complete("SELECT 1")
    }
```

Hello world: the handler

```
func handler(ctx context.Context, q wire.Query) (
    wire.PreparedStatements, error) {
    m := selectLiteral.FindStringSubmatch(q.Query)
    if m == nil {
        err := errors.New("only SELECT '<literal>' is supported")
        return nil, psqlerr.WithCode(err, codes.FeatureNotSupported)
    }
    handle := func(ctx context.Context, w wire.DataWriter,
        params []wire.Parameter) error {
        w.Row([]any{m[1]})
        return w.Complete("SELECT 1")
    }
    stmt := wire.NewStatement(handle, wire.WithColumns(columns))
    return wire.Prepared(stmt), nil
}
```

Hello world: in action

```
$ psql "host=localhost port=5433 user=jelte"
psql (17.5, server 17.0)

jelte=> SELECT 'hello world';
      echo
-----
hello world
(1 row)
```

Hello world: in action

```
$ psql "host=localhost port=5433 user=jelte"
psql (17.5, server 17.0)

jelte=> SELECT 'hello world';
      echo
-----
hello world
(1 row)
```

Shipped it. Postgres compatible.

Summary

- The wire protocol is the easy part
- PgBouncer gets by on the protocol alone, by never parsing SQL
- The protocol leaks the dialect: formats, OIDs, GUCs, version
- `information_schema` is never enough, clients end up in `pg_catalog`
- Decide who your users are, refuse the rest loudly

We're hiring



MotherDuck is hiring

- Offices in New York, Seattle, San Francisco and Amsterdam
- Work on a fun team building a fast and easy to use data warehouse
- Talk to me after the talk

<https://motherduck.com/careers/>

Questions?

- Slides: <https://github.com/JelteF/slides>
- PgBouncer: <https://github.com/pgbouncer/pgbouncer>
- MotherDuck Postgres Endpoint: <https://motherduck.com/docs/>
- psql-wire: <https://github.com/jeroenrinzema/psql-wire>

Previous talks:

- Prepared statements and cancellations in PgBouncer: <https://youtu.be/X-nCHcZ6vQU>
- Postgres protocol details: <https://youtu.be/nh62VgNj6hY>