59 Branches 31 Tags Go to file Go to file Code

levkk Online resharding (#784) a6afd9b · 6 hours ago

.cargo	Improve JDBC support. Build …	5 months ago
.claude/hooks	Numeric rough draft, includes …	5 months ago
.config	fix(pgdog-plugin): api version c…	last week
.github	fix: replication stream broke du…	4 days ago
dev	Re-introducing plugins (#337)	6 months ago
docker	fix docker compose	10 months ago
docs	fix(pgdog-plugin): api version c…	last week
examples	feat(docker): upgrade psql to 1…	4 months ago
integration	Online resharding (#784)	6 hours ago
pgdog-config	Online resharding (#784)	6 hours ago
pgdog-macros	fix(pgdog-plugin): api version c…	last week
pgdog-plugin	Online resharding (#784)	6 hours ago
pgdog-postgres-types	feat: add timestamp/timestamp…	3 weeks ago
pgdog-stats	Online resharding (#784)	6 hours ago
pgdog-vector	Schema-based sharding (#596)	3 months ago
pgdog	Online resharding (#784)	6 hours ago
plugins	fix(pgdog-plugin): api version c…	last week
sdk/ruby/pgdog	SHOW pgdog.shards comman…	9 months ago
.dockerignore	Fix params sync (#87)	10 months ago
.gitattributes	Remove git lfs	last year
.gitignore	fix(pgdog-plugin): api version c…	last week
.gitmodules	Schema triggers (#36)	11 months ago
AGENTS.md	Incremental Test improvement…	4 months ago
CLAUDE.md	Add some test coverage (#622)	3 months ago
CONTRIBUTING.md	fix(pgdog-plugin): api version c…	last week
Cargo.lock	v0.1.30 (#782)	4 days ago
Cargo.toml	fix(pgdog-plugin): api version c…	last week
Dockerfile	Update `Dockerfile` to accept t…	3 months ago
	save	2 years ago

About

PostgreSQL connection pooler, load balancer and database sharder.

pgdog.dev

#rust #postgresql #load-balancer #sharding #pooler

Readme
AGPL-3.0 license
Contributing
Security policy
Activity
Custom properties
3.7k stars
19 watching
136 forks
Report repository

Releases 29

v0.1.30 Latest
4 days ago

+ 28 releases

Packages 1

pgdog

Contributors 34

+ 20 contributors

Languages

Rust 82.9% **Python** 3.6%
JavaScript 2.6% **Blade** 2.2%
PLpgSQL 2.0% **Java** 1.5%
Other 5.2%

LICENSE		
README.md	feat: added support for AWS E…	last week
SECURITY.md	Create SECURITY.md	2 months ago
cli.sh	feat: track server last sent/rece…	last month
docker-compose.yml	feat(docker): upgrade psql to 1…	4 months ago
example.pgdog.toml	Create parse_queries_enabled…	3 months ago
example.users.toml	Renamed {configs}.toml to exa…	7 months ago
flamegraph.svg	Refactor(net): move wire proto…	6 months ago
mise.toml	chore: add required tools via m…	2 months ago
nextest.toml	One req at a time (#225)	8 months ago
pyrightconfig.json	ActiveRecord tests (#64)	11 months ago

README Contributing AGPL-3.0 license Security

PgDog is a proxy for scaling PostgreSQL. It supports connection pooling, load balancing queries and sharding entire databases. Written in Rust, PgDog is fast, secure and can manage thousands of connections on commodity hardware.

Documentation

📘 PgDog documentation can be **found here**. Any questions? Chat with us on **Discord**.

Quick start

Kubernetes

Helm chart is **here**. To install it, run:

```
helm repo add pgdogdev https://helm.pgdog.dev
helm install pgdog pgdogdev/pgdog
```

AWS

If you're using AWS RDS, you can deploy PgDog using one of two supported methods:

1. Helm chart with EKS, or a self-hosted Kubernetes cluster
2. Terraform module to deploy PgDog on ECS

Try in Docker

You can try PgDog quickly using Docker. Install Docker Compose and run:

```
docker-compose up
```

Once started, you can connect to PgDog with psql or any other PostgreSQL client:

```
PGPASSWORD=postgres psql -h 127.0.0.1 -p 6432 -U postgres
```

The demo comes with 3 shards and 2 sharded tables:

```
INSERT INTO users (id, email) VALUES (1, 'admin@acme.com');
INSERT INTO payments (id, user_id, amount) VALUES (1, 1, 100.0);

SELECT * FROM users WHERE id = 1;
SELECT * FROM payments WHERE user_id = 1;
```

Features

📘 **Configuration**

All PgDog features are configurable and can be turned on and off. PgDog requires 2 configuration files to operate:

1. `pgdog.toml` : hosts, sharding configuration, and other settings
2. `users.toml` : usernames and passwords

Example

Most options have reasonable defaults, so a basic configuration for a single user and database running on the same machine is pretty short:

pgdog.toml

```
[general]
port = 6432
default_pool_size = 10

[[databases]]
name = "pgdog"
host = "127.0.0.1"
```

users.toml

```
[[users]]
name = "alice"
database = "pgdog"
password = "hunter2"
```

If a database in `pgdog.toml` doesn't have a user in `users.toml` , the connection pool for that database will not be created and users won't be able to connect.

If you'd like to try it out locally, create the database and user like so:

```
CREATE DATABASE pgdog;
CREATE USER pgdog PASSWORD 'pgdog' LOGIN;
```

Transaction pooling

📘 **Transactions**

Like PgBouncer, PgDog supports transaction (and session) pooling, allowing thousands of clients to use just a few PostgreSQL server connections.

Unlike PgBouncer, PgDog can parse and handle `SET` statements and startup options, ensuring session state is set correctly when sharing server connections between clients with different parameters.

PgDog also has more advanced connection recovery options, like automatic abandoned transaction rollbacks and connection re-synchronization to avoid churning server connections during an application crash.

Load balancer

📘 **Load balancer**

PgDog is an application layer (OSI Level 7) load balancer for PostgreSQL. It understands the Postgres protocol, can proxy multiple replicas (and primary) and distributes transactions evenly between databases. The load balancer supports 3 strategies: round robin, random and least active connections.

Example

The load balancer is enabled automatically when a database has more than one host:

```
[[databases]]
name = "prod"
host = "10.0.0.1"
role = "primary"

[[databases]]
name = "prod"
host = "10.0.0.2"
role = "replica"
```

Health checks

📘 **Healthchecks**

PgDog maintains a real-time list of healthy hosts. When a database fails a health check, it's removed from the active rotation and queries are re-routed to other replicas. This works like an HTTP load balancer, except it's for your database.

Health checks maximize database availability and protect against bad network connections, temporary hardware failures or misconfiguration.

Single endpoint

📘 **Single endpoint**

PgDog uses `pg_query`, which includes the PostgreSQL native parser. By parsing queries, PgDog can detect writes (e.g. `INSERT`, `UPDATE`, `CREATE TABLE`, etc.) and send them to the primary, leaving the replicas to serve reads (`SELECT`). This allows applications to connect to the same PgDog deployment for both reads and writes.

Transactions

📘 **Load balancer & transactions**

Transactions can execute multiple statements, so in a primary & replica configuration, PgDog routes them to the primary. Clients can indicate a transaction is read-only, in which case PgDog will send it to a replica:

```
BEGIN READ ONLY;
-- This goes to a replica.
SELECT * FROM users LIMIT 1;
COMMIT;
```

Failover

📘 **Failover**

PgDog monitors Postgres replication state and can automatically redirect writes to a different database if a replica is promoted. This doesn't replace tools like Patroni that actually orchestrate failovers. You can use PgDog alongside Patroni (or AWS RDS or other managed Postgres host), to gracefully failover live traffic.

Example

To enable failover, set all database `role` attributes to `auto` and enable replication monitoring (`lsn_check_delay` setting):

```
[general]
lsn_check_delay = 0

[[databases]]
name = "prod"
host = "10.0.0.1"
role = "auto"

[[databases]]
name = "prod"
host = "10.0.0.2"
role = "auto"
```

Sharding

📘 **Sharding**

PgDog is able to manage databases with multiple shards. By using the PostgreSQL parser, PgDog extracts sharding keys and determines the best routing strategy for each query.

For cross-shard queries, PgDog assembles and transforms results in memory, sending all rows to the client as if they are coming from a single database.

Example

Configuring multiple hosts for the same database with different shard numbers (`shard` setting) enables sharding:

```
[[databases]]
name = "prod"
host = "10.0.0.1"
shard = 0

[[databases]]
name = "prod"
host = "10.0.0.2"
shard = 1
```

Note: read below for how to configure query routing. At least one sharded table is required for sharding to work as expected.

Sharding functions

📘 **Sharding functions**

PgDog has two main sharding algorithms:

1. PostgreSQL partition functions (`HASH` , `LIST` , `RANGE`)
2. Using schemas

Partition-based sharding

Partition-based sharding functions are taken directly from Postgres source code. This choice intentionally allows to shard data both with PgDog and with Postgres foreign tables and `postgres_fdw` .

Examples

The `PARTITION BY HASH` algorithm is used by default when configuring sharded tables:

```
[[sharded_tables]]
database = "prod"
column = "user_id"
```

List-based sharding (same as `PARTITION BY LIST` in Postgres) can be configured as follows:

```
# Sharded table definition still required.
[[sharded_tables]]
database = "prod"
column = "user_id"

# Value-specific shard mappings.
[[sharded_mapping]]
database = "prod"
column = "user_id"
values = [1, 2, 3, 4]
shard = 0

[[sharded_mapping]]
database = "prod"
column = "user_id"
values = [5, 6, 7, 8]
shard = 1
```

For range-based sharding, replace the `values` setting with a range, for example:

```
start = 0 # include
end = 5 # exclusive
```

Schema-based sharding

📘 **Schema-based sharding**

Schema-based sharding works on the basis of PostgreSQL schemas. Tables under the same schema are placed on the same shard and all queries that refer to those tables are routed to that shard automatically.

Example

Configuring sharded schemas uses a different configuration from sharded tables:

```
[[sharded_schemas]]
database = "prod"
name = "customer_a"
shard = 0

[[sharded_schemas]]
database = "prod"
name = "customer_b"
shard = 1
```

Queries that refer tables in schema `customer_a` will be sent to shard 0. For example, a query that refers to a table by its fully-qualified name will be sent to one shard only:

```
INSERT INTO customer_a.orders (id, user_id, amount)
VALUES ($1, $2, $3);
```

Alternatively, the schema name can be specified in the `search_path` session variable:

```
SET search_path TO public, customer_a;
-- All subsequent queries will be sent to shard 0.
SELECT * FROM orders LIMIT 1;
```

You can also set the `search_path` for the duration of a single transaction, using `SET LOCAL`, ensuring only that transaction is sent to the desired shard:

```
-- The entire transaction will be sent to shard 1.
BEGIN;
SET LOCAL search_path TO public, customer_b;
SELECT * FROM orders LIMIT 1;
COMMIT;
```

Direct-to-shard queries

📘 **Direct-to-shard queries**

Queries that contain a sharding key are sent to one database only. This is the best case scenario for sharded databases, since the load is uniformly distributed across the cluster.

Example:

```
-- user_id is the sharding key.
SELECT * FROM users WHERE user_id = $1;
```

Cross-shard queries

- 📘 **Cross-shard queries**
- 📘 **SELECT**
- 📘 **INSERT**
- 📘 **UPDATE and DELETE**
- 📘 **DDL**

Queries with multiple sharding keys or without one are sent to all databases and results are post-processed and assembled in memory. PgDog then sends the final result to the client.

Currently, support for certain SQL features in cross-shard queries is limited. However, the list of supported ones keeps growing:

Feature	Supported	Notes
Aggregates	Partial	`count`, `min`, `max`, `stddev`, `variance`, `sum`, `avg` are supported.
`ORDER BY`	Partial	Column in `ORDER BY` clause must be present in the result set.
`GROUP BY`	Partial	Same as `ORDER BY`, referenced columns must be present in result set.
Multi-tuple `INSERT`	Supported	PgDog generates one statement per tuple and executes them automatically.
Sharding key `UPDATE`	Supported	PgDog generates a `SELECT`, `INSERT` and `DELETE` statements and execute them automatically.
Subqueries	No	The same subquery is executed on all shards.
CTEs	No	The same CTE is executed on all shards.

Using `COPY`

📘 **Copy**

PgDog has a text, CSV & binary parser and can split rows sent via `COPY` command between all shards automatically. This allows clients to ingest data into sharded PostgreSQL without preprocessing

Example

```
COPY orders (id, user_id, amount) FROM STDIN CSV HEADER;
```

Columns must be specified in the `COPY` statement, so PgDog can infer the sharding key automatically, but are optional in the data file.

Consistency (two-phase commit)

📘 **Two-phase commit**

To make sure cross-shard writes are atomic, PgDog supports Postgres' two-phase transactions. When enabled, PgDog handles `COMMIT` statements sent by clients by executing the 2pc exchange on their behalf:

```
PREPARE TRANSACTION '__pgdog_unique_id';
COMMIT PREPARED '__pgdog_unique_id';
```

In case the client disconnects or Postgres crashes, PgDog will automatically rollback the transaction if it's in phase I and commit it if it's in phase II.

Unique identifiers

📘 **Unique IDs**

While applications can use `UUID` (v4 and now v7) to generate unique primary keys, PgDog supports creating unique `BIGINT` identifiers, without using a sequence:

```
SELECT pgdog.unique_id();
```

This uses a timestamp-based algorithm, can produce millions of unique numbers per second and doesn't require an expensive cross-shard index to guarantee uniqueness.

Shard key updates

PgDog supports changing the sharding key for a row online. Under the hood, it will execute 3 statements to make it happen:

1. `SELECT` to get the entire row from its original shard
2. `INSERT` to write the new, changed row to the new shard
3. `DELETE` to remove it from the old shard

This happens automatically, and the client can retrieve the new row as normal:

```
UPDATE orders SET user_id = 5 WHERE user_id = 1 RETURNING *;
-- This will return the new row
```

Note: Only one row can be updated at a time and if a query attempts to update multiple, PgDog will abort the transaction.

To enable shard key updates, add this to `pgdog.toml`:

```
[rewrite]
enabled = true
shard_key = "rewrite" # options: ignore (possible data loss), error (block shard key update)
```

Multi-tuple inserts

PgDog can handle multi-tuple `INSERT` queries by sending each tuple to the right shard, e.g.:

```
INSERT INTO payments
    (id, user_id, amount) -- user_id is the sharding key
VALUES
(pgdog.unique_id(), 1, 25.00), -- Tuples go to different shards
(pgdog.unique_id(), 5, 55.0); -- Each tuple gets a unique primary key because unique ID function is invoked twice
```

This happens automatically, if enabled:

```
[rewrite]
enabled = true
split_inserts = "rewrite" # other options: ignore, error
```

Re-sharding

- 📘 **Re-sharding**
- 📘 **Schema sync**
- 📘 **Data sync**

PgDog understands the PostgreSQL logical replication protocol and can orchestrate data splits between databases, in the background and without downtime. This allows to shard existing databases and add more shards to existing clusters in production, without impacting database operations.

The re-sharding process is done in 5 steps:

1. Create new empty cluster with the desired number of shards
2. Configure it in `pgdog.toml` and run `schema-sync` command to copy table schemas to the new databases
3. Run `data-sync` command to copy and re-shard table data with logical replication (tables are copied in parallel)
4. While keeping previous command running (it streams row updates in real-time), run `schema-sync --data-sync-complete` to create secondary indexes on the new databases (much faster to do this after data is copied)
5. Cutover traffic to new cluster with `MAINTENANCE ON`, `RELOAD`, `MAINTENANCE OFF` command sequence

Cutover can be done atomically with multiple PgDog containers because `RELOAD` doesn't resume traffic, `MAINTENANCE OFF` does, so the config is the same in all containers before queries are resumed. No complex synchronization tooling like etcd or Zookeeper is required.

Monitoring

📘 **Metrics**

PgDog exposes both the standard PgBouncer-style admin database and an OpenMetrics endpoint. The admin database isn't 100% compatible, so we recommend you use OpenMetrics for monitoring. Example Datadog configuration and dashboard are included.

Running PgDog locally

Install the latest version of the Rust compiler from rust-lang.org. Clone this repository and build the project in release mode:

```
cargo build --release
```

It's important to use the release profile if you're deploying to production or want to run performance benchmarks.

Try sharding

Sharded database clusters are set in the config. For example, to set up a 2 shard cluster, you can:

pgdog.toml

```
[[databases]]
name = "pgdog_sharded"
host = "127.0.0.1"
database_name = "shard_0"
shard = 0

[[databases]]
name = "pgdog_sharded"
host = "127.0.0.1"
database_name = "shard_1"
shard = 1

[[sharded_tables]]
database = "pgdog_sharded"
column = "user_id"
```

Don't forget to configure a user:

users.toml

```
[[users]]
database = "pgdog_sharded"
name = "pgdog"
password = "pgdog"
```

And finally, to make it work locally, create the required databases:

```
CREATE DATABASE shard_0;
CREATE DATABASE shard_1;

GRANT ALL ON DATABASE shard_0 TO pgdog;
GRANT ALL ON DATABASE shard_1 TO pgdog;
```

Start PgDog

Running PgDog can be done with Cargo:

```
cargo run --release
```

Command-line options

PgDog supports several command-line options:

- `-c, --config <CONFIG>` : Path to the configuration file (default: `"pgdog.toml"`)
- `-u, --users <USERS>` : Path to the users.toml file (default: `"users.toml"`)
- `-d, --database_url <DATABASE_URL>` : Connection URL(s). Can be specified multiple times to add multiple database connections. When provided, these URLs override database configurations from the config file.

Example using database URLs directly:

```
cargo run --release -- -d postgres://user:pass@localhost:5432/db1 -d postgres://user:pass@localhost:5433/db2
```

You can connect to PgDog with `psql` or any other PostgreSQL client:

```
psql postgres://pgdog:pgdog@127.0.0.1:6432/pgdog
```

🚦 Status 🚦

PgDog is used in production and at scale. Most features are stable, while some are experimental. Check documentation for more details. New sharding features are added almost weekly.

Performance

📘 **Architecture & benchmarks**

PgDog is heavily optimized for performance. We use Rust, Tokio, bytes crate to avoid unnecessary memory allocations, and profile for performance regressions on a regular basis.

License

PgDog is free and open source software, licensed under the AGPL v3. While often misunderstood, this license is very permissive and allows the following without any additional requirements from you or your organization:

- Internal use
- Private modifications for internal use without sharing any source code

You can freely use PgDog to power your PostgreSQL databases without having to share any source code, including proprietary work product or any PgDog modifications you make.

AGPL was written specifically for organizations that offer PgDog *as a public service* (e.g. database cloud providers) and require those