SCHEDULING POSTGRESQL MADE SIMPLE

Taking control of your database jobs

PAVLO GOLUB
Senior Consultant/Developer

Nordic PGDay 2026

Speaker Introduction

Pavlo Golub
Senior Consultant/Developer

- pavlo.golub@cybertec.at
- @PavloGolub
- pashagolub.github.io

About CYBERTEC

- PostgreSQL Services & Support
- PostgreSQL Consulting
- PostgreSQL Remote DBA
- Database Products & Tools

Agenda

- Different levels of database scheduling
- PostgreSQL scheduling approaches
- PostgreSQL scheduling tools available
- **pg_timetable**: Why it is so cool ;)

Why Schedule?

- Maintenance
- Data Import / Export
- Backup / Restore
- Analytical Processing
- Monitoring
- External Actions

Different Levels of Database Scheduling

Three categories:

1. **Built-in Schedulers** — part of the database engine
2. **System Schedulers** — OS or cloud level
3. **PostgreSQL land** — third-party tools built for PostgreSQL

Built-in Schedulers

Some databases ship their own scheduler:

- Microsoft SQL Server
- Oracle
- MySQL (MariaDB)
- DB2

PostgreSQL does not have one.

Built-in Scheduler in PostgreSQL?

> *Many people say it's not necessary, and probably some hackers would oppose it; but mainly I think we just haven't agreed (or even discussed) what the design of such a scheduler would look like. For example, do we want it to be able to just connect and run queries and stuff, or do we want something more elaborate able to start programs such as running pg_dump? What if the program crashes — should it cause the server to restart? And so on. It's not a trivial problem.*

— **Alvaro Herrera**

System Schedulers

Tools available:

- cron, anacron, etc.
- Windows Task Scheduler
- Google Cloud Tasks, Amazon Scheduled Tasks
- Kubernetes CronJob

Cons

They don’t know anything about databases.

Schedulers in “PostgreSQL Land”

Tool	Status
pgAgent	Active
pg_cron	Active
pg_timetable	Active
jpgAgent	Discontinued
pgBucket / runseven	Discontinued
pgAutomator	Discontinued

pgAgent

- The **oldest** PostgreSQL scheduler
- Was part of pgAdmin, now distributed independently
- Written in **C++**
- Stores configuration in the database
- SQL and SHELL tasks
- https://github.com/postgres/pgagent

pg_cron

- Implemented as a **PostgreSQL background worker**
- Written in **C**
- Uses libpq to open a new connection to databases
- **SQL only** tasks
- Jobs executed locally with current user permissions
- Superusers may allow remote execution via sys table
- Need `.pgpass` for remote server auth
- https://github.com/citusdata/pg_cron

pg_timetable

pg_timetable: Creating the Ultimate Scheduler

- Main design principles
- Architecture
- Features
- Demo

pg_timetable: Main Principles

- **1-minute setup**
- Docker image (“cloud ready”)
- One binary written in Go
- Non-invasive
- No extensions or superuser needed

- Schema auto deployment
- Huge number of jobs
- Cross platform support
- SQL, PROGRAM and BUILT-IN tasks

Building Blocks: Tasks And Chains

Tasks

- A task is the **most basic building block**
- Each task has a **kind**: `SQL`, `PROGRAM`, or `BUILTIN`
- Tasks can take parameters (executed once per parameter row)
- e.g. “Download data”, “Aggregate data”, “Send report”, etc.

Chains

- A chain is a **sequence of tasks** wrapped in one transaction
- Arrange tasks in a large sequence of things

Task Types: SQL & PROGRAM

SQL

- Parameterized SQL executed against PostgreSQL (local **or** remote DB)
- Supports `SET ROLE` per task (`run_as`)
- Runs inside the chain's transaction unless `autonomous: true`

PROGRAM

- External binary executed via the OS
- Combined stdout+stderr captured and stored in the execution log
- Disabled safely with `--no-program-tasks`

Task Types: BUILTIN

- `NoOp` — no-op placeholder
- `Sleep` — context-aware sleep
- `Log` — emit a log message
- `SendMail` — full SMTP with attachments
- `Download` — concurrent HTTP downloads
- `CopyFromFile` — `COPY FROM` a local file
- `CopyToFile` — `COPY TO` a local file
- `CopyFromProgram` — pipe program stdout → `COPY FROM`
- `CopyToProgram` — pipe `COPY TO` → program stdin
- `Shutdown` — graceful scheduler shutdown

Building Blocks: Workflow Example

Step	Action
0	Start Transaction
1	Download data
2	Transform data
3	Aggregate
4	Delete file
5	Commit

Design: Scheduling Modes

- **Cron** `"0 2 * * *"` — classic 5-field syntax
- **`@reboot`** — runs once at scheduler startup
- **`@every 10 minutes`** — fixed interval (starts before run)
- **`@after 30 seconds`** — interval starts *after* run finishes

Scheduling Modes: Cron & @reboot

```
-- Classic cron: every day at 02:00
SELECT timetable.add_job(
    job_name     => 'nightly-vacuum',
    job_schedule => '0 2 * * *',
    job_command  => 'VACUUM ANALYZE');

-- Run once at scheduler startup
SELECT timetable.add_job(
    job_name     => 'warm-cache',
    job_schedule => '@reboot',
    job_command  => 'SELECT warm_up_cache()');
```


Scheduling Modes: @every & @after

```
-- @every: fire-and-forget tick (timer starts BEFORE run)
SELECT timetable.add_job(
    job_name     => 'heartbeat',
    job_schedule => '@every 10 seconds',
    job_command  => 'SELECT pg_notify(''heartbeat'', now()::text)');

-- @after: timer starts AFTER run finishes (cool-down)
SELECT timetable.add_job(
    job_name     => 'slow-etl',
    job_schedule => '@after 5 minutes',
    job_command  => 'CALL run_etl()');
```


Design: Enhanced logs

- Per-task execution log with return code
- Parameters recorded per execution
- Database-side log (GUI-friendly)
- File logs with rotation support

Design: Concurrency & Configuration

- Concurrency implemented with **lightweight goroutines**
  - Efficiency matters with hundreds of thousands of jobs
  - Configurable worker pools: `--cron-workers`, `--interval-workers`
- **Fully database-driven configuration**
  - Backups are easy and centralized
  - GUIs can be produced easily
  - Easy to search and modify
  - Simplified versioning
- **YAML chain definitions** — version control friendly
  - Load via `--file my-jobs.yaml`
  - Validate only with `--validate` before import
  - Replace existing chains with `--replace`

Design: Safety & Concurrency Protection

- **Concurrency protection**
  - `max_instances` caps parallel runs of the same chain
  - Example: Ensure only one backup runs at a time
- **Optionally ignore errors** — SAVEPOINTs keep the chain transaction alive
- **Optional exclusive execution** — acquires write lock, pauses all other chains
- **Autonomous tasks** — run outside the chain transaction
  - Required for `VACUUM`, `CREATE DATABASE`, procedures with `COMMIT`

Error Handling: on_error SQL

Each chain can define a SQL snippet executed **when the chain fails**:

```
SELECT timetable.add_job(
    job_name      => 'important-job',
    job_schedule  => '0 2 * * *',
    job_command   => 'CALL nightly_process()',
    job_on_error  => $$
        SELECT pg_notify('monitoring',
            format('{"chain": %s, "status": "failed"}',
                current_setting('pg_timetable.current_chain_id')))
    $$);
```


Error Handling: Retry & Job Control

- **Delayed retry** — reschedule from `on_error` with backoff:

```
PERFORM timetable.notify_chain_start(
    chain_id    => current_setting('pg_timetable.current_chain_id')::bigint,
    worker_name => 'worker001',
    start_delay => interval '5 minutes');
```

- **Session GUCs** inside task SQL:
  - `current_setting('pg_timetable.current_chain_id')`
  - `current_setting('pg_timetable.current_client_name')`
- **pause/resume** without deleting:
  - `timetable.pause_job('job-name')`
  - `timetable.resume_job('job-name')`

Design: Self-Destructive Chains

- Basically for **one-shot / asynchronous execution**
- `self_destruct: true` — chain is deleted after successful execution
- Try to execute once and kill it when done

Why this matters

External applications can trigger one-shot jobs with a single `INSERT` — no need to manage cleanup.

Async Execution via LISTEN/NOTIFY

- Chains can be triggered **immediately**, bypassing the cron schedule
- Works from **any** PostgreSQL connection — no pg_timetable API needed

```
-- Trigger a chain right now
PERFORM timetable.notify_chain_start(
    chain_id    => 42,
    worker_name => 'worker001');

-- Cancel a running chain
PERFORM timetable.notify_chain_stop(42, 'worker001');
```


Async Execution: Delayed Start & Patterns

```
-- Trigger with a delay (e.g. exponential backoff retry)
PERFORM timetable.notify_chain_start(
    chain_id    => 42,
    worker_name => 'worker001',
    start_delay => interval '5 minutes');
```

- Deduplication: repeated NOTIFYs within 60 s are ignored
- Enables **application-triggered** jobs with a plain `INSERT` or `SELECT`
- Enables **retry-with-backoff** from `on_error` SQL
- `self_destruct: true` + NOTIFY = **true async one-shot execution**

Architecture And Components

- **Workers** (Golang goroutines)
- **Config database** (PostgreSQL)
- Optional remote target databases
- **REST API** (optional, `--rest-port`)

- Optional monitoring:
  - pgwatch
  - psql
  - Anything you want …
- Everything is in tables
- YAML chain definitions

REST API

Enable with `--rest-port 8080` (or `PGTT_RESTPORT=8080`)

Endpoint	Description
`GET /liveness`	Process is alive (for probes)
`GET /readiness`	Scheduler loop is running (`503` during init)
`GET /startchain?id=42`	Enqueue chain for immediate execution
`GET /stopchain?id=42`	Cancel a currently running chain

Use cases

- Kubernetes liveness / readiness probes
- External application triggers without a DB connection
- CI/CD pipeline integration

YAML: Chain Properties

```yaml
chains:
  - name: "ETL Pipeline"        # unique identifier
    schedule: "0 2 * * *"       # cron / @reboot / @every / @after
    live: true                   # false = disabled
    max_instances: 1             # max parallel executions
    timeout: 3600000             # chain timeout in ms
    on_error: "SELECT pg_notify('alerts', 'ETL failed')"
    self_destruct: false         # delete chain after success
    exclusive: false             # pause all others while running
    client_name: "worker-1"     # pin to a specific worker instance
```


YAML: Task Properties & Loading

```yaml
tasks:
  - name: "Extract"
    kind: "SQL"              # SQL | PROGRAM | BUILTIN
    command: "CALL extract_data()"
    autonomous: true         # outside chain transaction
    ignore_error: false      # continue chain on failure
    timeout: 30000           # task timeout in ms
    parameters:
      - ["arg1", "arg2"]     # one run per row
```

```
# Import (replace existing chains with same name)
pg_timetable --file etl.yaml --replace
# Validate syntax only, do not import
pg_timetable --file etl.yaml --validate
```


pg_timetable: Supported Environments

- Linux
- macOS
- Windows

- Docker
- Kubernetes
- Cloud (AWS, GCP, Azure)

Comparison Table

Feature	pgAgent	pg_cron	pg_timetable
Language	C++	C	Go
SQL tasks	Yes	Yes	Yes
Shell tasks	Yes	No	Yes
Built-in tasks	No	No	Yes
Chains	No	No	Yes
Cron schedule	Yes	Yes	Yes
Interval schedule	No	No	Yes
Remote exec	No	Limited	Yes
on_error handler	No	No	Yes
REST API	No	No	Yes
YAML config	No	No	Yes
Docker ready	No	No	Yes

Demo

Demo: Setup (Docker)

```
docker run -d \
  -e PGTT_CONNSTR="host=db dbname=scheduler user=scheduler" \
  -p 8080:8080 \
  cybertecpostgresql/pg_timetable:latest \
  --rest-port 8080
```

- Schema is deployed **automatically** on first run
- No PostgreSQL extensions or superuser privileges required
- REST API enabled on port 8080

Demo: Setup (Binary)

```
# Download the binary from GitHub releases and run directly
pg_timetable \
  postgres://scheduler@localhost/scheduler \
  --rest-port 8080 \
  --file demo.yaml \
  --log-level debug
```

- Single binary, no runtime dependencies
- `--file` loads chain definitions from YAML on startup
- `--validate` checks YAML syntax without importing

Demo: Define a Simple Chain

```yaml
chains:
  - name: "hello-world"
    schedule: "@every 10 seconds"
    live: true
    tasks:
      - name: "log message"
        kind: "BUILTIN"
        command: "Log"
        parameters: [["Hello from pg_timetable!"]]
```

```
pg_timetable postgres://scheduler@localhost/scheduler \
  --file hello.yaml
```


Demo: Multi-Task Chain with Error Handling

```yaml
chains:
  - name: "weekly-report"
    schedule: "0 8 * * 1"    # every Monday 08:00
    live: true
    on_error: "SELECT pg_notify('alerts', 'report failed')"
    tasks:
      - name: "generate"
        kind: "SQL"
        command: "CALL generate_weekly_report()"
      - name: "email"
        kind: "BUILTIN"
        command: "SendMail"
        parameters:
          - [{"to": ["team@example.com"], "subject": "Weekly Report"}]
```


Demo: Trigger & Health Check

```
# Readiness probe (503 during startup)
curl http://localhost:8080/readiness

# Trigger a chain on demand (bypasses schedule)
curl "http://localhost:8080/startchain?id=1"

# Cancel a running chain
curl "http://localhost:8080/stopchain?id=1"
```


Demo: Monitor via SQL

```sql
-- Currently running chains
SELECT chain_id, started_at, chain_name
FROM timetable.active_chain
JOIN timetable.chain USING (chain_id);

-- Execution history with parameters used
SELECT started_at, finished, returncode, params
FROM timetable.execution_log
ORDER BY started_at DESC LIMIT 10;

-- Pause / resume without deleting the chain
SELECT timetable.pause_job('weekly-report');
SELECT timetable.resume_job('weekly-report');
```


Be Inspired

> *“I would rather have questions that can’t be answered than answers that can’t be questioned.”*
>
> — **Richard Feynman**

Don’t Be A Stranger

- **CYBERTEC GitHub**
  https://github.com/cybertec-postgresql
- **Personal GitHub**
  https://github.com/pashagolub
- **CYBERTEC Blog**
  https://www.cybertec-postgresql.com/en/blog/