pg_timetable Documentation

Release master

Jun 24, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

pg_timetable is an advanced job scheduler for PostgreSQL, offering many advantages over traditional schedulers such as **cron** and others. It is completely database driven and provides a couple of advanced concepts.

1.1 Main features

- Tasks can be arranged in chains
- A chain can consist of built-int commands, SQL and executables
- Parameters can be passed to chains
- Missed tasks (possibly due to downtime) can be retried automatically
- Support for configurable repetitions
- Built-in tasks such as sending emails, etc.
- Fully database driven configuration
- Full support for database driven logging
- Cron-style scheduling at the PostgreSQL server time zone
- Optional concurrency protection
- Task and chain can have execution timeout settings

1.2 Quick Start

1. Download pg_timetable executable
2. Make sure your PostgreSQL server is up and running and has a role with `CREATE` privilege for a target database, e.g.

```
my_database=> CREATE ROLE scheduler PASSWORD 'somestrong';
my_database=> GRANT CREATE ON DATABASE my_database TO scheduler;
```

3. Create a new job, e.g. run `VACUUM` each night at 00:30 Postgres server time zone

```
my_database=> SELECT timetable.add_job('frequent-vacuum', '30 * * * *',
↪'VACUUM');
add_job
---------
```

(continues on next page)

(continued from previous page)

```
      3
(1 row)
```

4. Run the **pg_timetable**

```
# pg_timetable postgresql://scheduler:somestrong@localhost/my_database --
↪clientname=vacuumer
```

5. PROFIT!

1.3 Command line options

```
# ./pg_timetable

Application Options:
  -c, --clientname=                          Unique name for application instance [
↪$PGTT_CLIENTNAME]
      --config=                              YAML configuration file
      --no-program-tasks                     Disable executing of PROGRAM tasks [$PGTT_
↪NOPROGRAMTASKS]

Connection:
  -h, --host=                                PostgreSQL host (default: localhost) [
↪$PGTT_PGHOST]
  -p, --port=                                PostgreSQL port (default: 5432) [$PGTT_
↪PGPORT]
  -d, --dbname=                              PostgreSQL database name (default:␣
↪timetable) [$PGTT_PGDATABASE]
  -u, --user=                                PostgreSQL user (default: scheduler) [
↪$PGTT_PGUSER]
      --password=                            PostgreSQL user password [$PGTT_PGPASSWORD]
      --sslmode=[disable|require]            What SSL priority use for connection␣
↪(default: disable)
      --pgurl=                               PostgreSQL connection URL [$PGTT_URL]
      --timeout=                             PostgreSQL connection timeout in seconds␣
↪(default: 90) [$PGTT_TIMEOUT]

Logging:
      --log-level=[debug|info|error]         Verbosity level for stdout and log file␣
↪(default: info)
      --log-database-level=[debug|info|error] Verbosity level for database storing␣
↪(default: info)
      --log-file=                            File name to store logs
      --log-file-format=[json|text]          Format of file logs (default: json)

Start:
  -f, --file=                                SQL script file to execute during startup
      --init                                 Initialize database schema to the latest␣
↪version and exit. Can be used
                                             with --upgrade
```

(continues on next page)

(continued from previous page)

```
      --upgrade                                 Upgrade database to the latest version
      --debug                                   Run in debug mode. Only asynchronous␣
↪chains will be executed

Resource:
      --cron-workers=                           Number of parallel workers for scheduled␣
↪chains (default: 16)
      --interval-workers=                       Number of parallel workers for interval␣
↪chains (default: 16)
      --chain-timeout=                          Abort any chain that takes more than the␣
↪specified number of milliseconds
      --task-timeout=                           Abort any task within a chain that takes␣
↪more than the specified number
                                                of milliseconds

REST:
      --rest-port:                              REST API port (default: 0) [%PGTT_RESTPORT
↪%]
```

1.4 Contributing

If you want to contribute to **pg_timetable** and help make it better, feel free to open an issue or even consider submitting a pull request. You also can give a star to **pg_timetable** project, and to tell the world about it.

1.5 Support

For professional support, please contact Cybertec.

1.6 Authors

Implementation: Pavlo Golub

Initial idea and draft design: Hans-Jürgen Schönig

CHAPTER

TWO

PROJECT BACKGROUND

The pg_timetable project got started back in 2019 for internal scheduling needs at Cybertec.

For more background on the project motivations and design goals see the original series of blogposts announcing the project and the following feature updates.

Cybertec also provides commercial 9-to-5 and 24/7 support for pg_timetable.

- Project announcement
- v2 released
- Start-up improvements
- v3 released
- Exclusive jobs explained
- Asynchronous chain execution
- v4 released
- PostgreSQL schedulers: comparison table

2.1 Project feedback

For feature requests or troubleshooting assistance please open an issue on project's Github page.

CHAPTER

THREE

INSTALLATION

pg_timetable is compatible with the latest supported PostgreSQL versions: 11, 12, 13, 14 (stable); 15 (dev).

Note:

```
CREATE OR REPLACE FUNCTION starts_with(text, text)
RETURNS bool AS
$$
SELECT
    CASE WHEN length($2) > length($1) THEN
        FALSE
    ELSE
        left($1, length($2)) = $2
    END
$$
LANGUAGE SQL
IMMUTABLE STRICT PARALLEL SAFE
COST 5;
```

3.1 Official release packages

You may find binary package for your platform on the official Releases page. Right now **Windows**, **Linux** and **macOS** packages are available.

3.2 Docker

The official docker image can be found here: https://hub.docker.com/r/cybertecpostgresql/pg_timetable

Note: The `latest` tag is up to date with the *master* branch thanks to this github action. In production you probably want to use the latest stable tag.

Run **pg_timetable** in Docker:

```
docker run --rm \
cybertecpostgresql/pg_timetable:latest \
-h 10.0.0.3 -p 54321 -c worker001
```

Run **pg_timetable** in Docker with Environment variables:

```
docker run --rm \
-e PGTT_PGHOST=10.0.0.3 \
-e PGTT_PGPORT=54321 \
cybertecpostgresql/pg_timetable:latest \
-c worker001
```

3.3 Build from sources

1. Download and install Go on your system.
2. Clone **pg_timetable** repo:

```
$ git clone https://github.com/cybertec-postgresql/pg_timetable.git
$ cd pg_timetable
```

3. Run **pg_timetable**:

```
$ go run main.go --dbname=dbname --clientname=worker001 --user=scheduler --
↪password=strongpwd
```

4. Alternatively, build a binary and run it:

```
$ go build
$ ./pg_timetable --dbname=dbname --clientname=worker001 --user=scheduler --
↪password=strongpwd
```

5. (Optional) Run tests in all sub-folders of the project:

```
$ psql --command="CREATE USER scheduler PASSWORD 'somestrong'"
$ createdb --owner=scheduler timetable
$ go test -failfast -timeout=300s -count=1 -p 1 ./...
```

CHAPTER

FOUR

COMPONENTS

The scheduling in **pg_timetable** encompasses three different abstraction levels to facilitate the reuse with other parameters or additional schedules.

Command
: The base level, **command**, defines *what* to do.

Task
: The second level, **task**, represents a chain element (step) to run one of the commands. With **tasks** we define order of commands, arguments passed (if any), and how errors are handled.

Chain
: The third level represents a connected tasks forming a chain of tasks. **Chain** defines *if*, *when*, and *how often* a job should be executed.

4.1 Command

Currently, there are three different kinds of commands:

`SQL`
: SQL snippet. Starting a cleanup, refreshing a materialized view or processing data.

`PROGRAM`
: External Command. Anything that can be called as an external binary, including shells, e.g. `bash`, `pwsh`, etc. The external command will be called using golang's exec.CommandContext.

`BUILTIN`
: Internal Command. A prebuilt functionality included in **pg_timetable**. These include:

  - *NoOp*,
  - *Sleep*,
  - *Log*,
  - *SendMail*,
  - *Download*,
  - *CopyFromFile*,
  - *CopyToFile*,
  - *Shutdown*.

4.2 Task

The next building block is a **task**, which simply represents a step in a list of chain commands. An example of tasks combined in a chain would be:

1. Download files from a server
2. Import files
3. Run aggregations
4. Build report
5. Remove the files from disk

Note: All tasks of the chain in **pg_timetable** are executed within one transaction. However, please, pay attention there is no opportunity to rollback `PROGRAM` and `BUILTIN` tasks.

4.2.1 Table timetable.task

chain_id bigint
Link to the chain, if `NULL` task considered to be disabled

task_order DOUBLE PRECISION
Indicates the order of task within a chain.

kind timetable.command_kind
The type of the command. Can be *SQL* (default), *PROGRAM* or *BUILTIN*.

command text
Contains either a SQL command, a path to application or name of the *BUILTIN* command which will be executed.

run_as text
The role as which the task should be executed as.

database_connection text
The connection string for the external database that should be used.

ignore_error boolean
Specify if the next task should proceed after encountering an error (default: `false`).

autonomous boolean
Specify if the task should be executed out of the chain transaction. Useful for `VACUUM`, `CREATE DATABASE`, `CALL` etc.

timeout integer
Abort any task within a chain that takes more than the specified number of milliseconds.

Warning: If the **task** has been configured with `ignore_error` set to `true` (the default value is `false`), the worker process will report a success on execution *even if the task within the chain fails*.

As mentioned above, **commands** are simple skeletons (e.g. *send email*, *vacuum*, etc.). In most cases, they have to be brought to live by passing input parameters to the execution.

4.2.2 Table timetable.parameter

task_id bigint
The ID of the task.

order_id integer
The order of the parameter. Several parameters are processed one by one according to the order.

value jsonb
A JSON value containing the parameters.

4.2.3 Parameter value format

Depending on the **command** kind argument can be represented by different *JSON* values.

Kind

Schema
Example

SQL

array

```
'[ "one", 2, 3.14, false ]'::jsonb
```

PROGRAM

array of strings

```
'["-x", "Latin-ASCII", "-o", "orte_ansi.txt", "orte.txt"]'::jsonb
```

BUILTIN: Sleep

integer

```
'5' :: jsonb
```

BUILTIN: Log

any

```
'"WARNING"'::jsonb
'{"Status": "WARNING"}'::jsonb
```

BUILTIN: SendMail

object

```
'{
    "username":     "user@example.com",
    "password":     "password",
    "serverhost":   "smtp.example.com",
    "serverport":   587,
    "senderaddr":   "user@example.com",
    "ccaddr":       ["recipient_cc@example.com"],
    "bccaddr":      ["recipient_bcc@example.com"],
    "toaddr":       ["recipient@example.com"],
```

(continues on next page)

(continued from previous page)

```
    "subject":      "pg_timetable - No Reply",
    "attachment":   ["/temp/attachments/Report.pdf","config.yaml"],
    "attachmentdata": [{"name": "File.txt", "base64data": "RmlsZSBDb250ZW50"}],
    "msgbody":      "<h2>Hello User,</h2> <p>check some attachments!</p>",
    "contenttype":   "text/html; charset=UTF-8"
}'::jsonb
```

BUILTIN: Download

object

```
'{
    "workersnum": 2,
    "fileurls": ["http://example.com/foo.gz", "https://example.com/bar.csv"],
    "destpath": "."
}'::jsonb
```

BUILTIN: CopyFromFile

object

```
'{
    "sql": "COPY location FROM STDIN",
    "filename": "download/orte_ansi.txt"
}'::jsonb
```

BUILTIN: CopyToFile

object

```
'{
    "sql": "COPY location TO STDOUT",
    "filename": "download/location.txt"
}'::jsonb
```

BUILTIN: Shutdown
value ignored

BUILTIN: NoOp
value ignored

4.3 Chain

Once tasks have been arranged, they have to be scheduled as a **chain**. For this, **pg_timetable** builds upon the enhanced **cron**-string, all the while adding multiple configuration options.

4.3.1 Table timetable.chain

chain_name text
The unique name of the chain.

run_at timetable.cron
Standard *cron*-style value at Postgres server time zone or `@after`, `@every`, `@reboot` clause.

max_instances integer
The amount of instances that this chain may have running at the same time.

timeout integer
Abort any chain that takes more than the specified number of milliseconds.

live boolean
Control if the chain may be executed once it reaches its schedule.

self_destruct boolean
Self destruct the chain after successful execution. Failed chains will be executed according to the schedule one more time.

exclusive_execution boolean
Specifies whether the chain should be executed exclusively while all other chains are paused.

client_name text
Specifies which client should execute the chain. Set this to *NULL* to allow any client.

Note: All chains in **pg_timetable** are scheduled at the PostgreSQL server time zone. You can change the timezone for the **current session** when adding new chains, e.g.

```
SET TIME ZONE 'UTC';

-- Run VACUUM at 00:05 every day in August UTC
SELECT timetable.add_job('execute-func', '5 0 * 8 *', 'VACUUM');
```

CHAPTER

FIVE

GETTING STARTED

A variety of examples can be found in the *Samples*. If you want to migrate from a different scheduler, you can use scripts from *Migration from others schedulers* chapter.

5.1 Add simple job

In a real world usually it's enough to use simple jobs. Under this term we understand:

- job is a chain with only one **task** (step) in it;
- it doesn't use complicated logic, but rather simple **command**;
- it doesn't require complex transaction handling, since one task is implicitely executed as a single transaction.

For such a group of chains we've introduced a special function `timetable.add_job()`.

`timetable.add_job(job_name, job_schedule, job_command, ...) RETURNS BIGINT`

Creates a simple one-task chain

Parameters

- **`job_name`** (*`text`*) – The unique name of the **chain** and **command**.
- **`job_schedule`** (*`timetable.cron`*) – Time schedule in ron syntax at Postgres server time zone
- **`job_command`** (*`text`*) – The SQL which will be executed.
- **`job_parameters`** (*`jsonb`*) – Arguments for the chain **command**. Default: `NULL`.
- **`job_kind`** (*`timetable.command_kind`*) – Kind of the command: *SQL*, *PROGRAM* or *BUILTIN*. Default: `SQL`.
- **`job_client_name`** (*`text`*) – Specifies which client should execute the chain. Set this to *NULL* to allow any client. Default: `NULL`.
- **`job_max_instances`** (*`integer`*) – The amount of instances that this chain may have running at the same time. Default: `NULL`.
- **`job_live`** (*`boolean`*) – Control if the chain may be executed once it reaches its schedule. Default: `TRUE`.
- **`job_self_destruct`** (*`boolean`*) – Self destruct the chain after execution. Default: `FALSE`.
- **`job_ignore_errors`** (*`boolean`*) – Ignore error during execution. Default: `TRUE`.
- **`job_exclusive`** (*`boolean`*) – Execute the chain in the exclusive mode. Default: `FALSE`.

Returns
the ID of the created chain

Return type
integer

5.2 Examples

1. Run `public.my_func()` at 00:05 every day in August Postgres server time zone:

```
SELECT timetable.add_job('execute-func', '5 0 * 8 *', 'SELECT public.my_
↪func()');
```

2. Run *VACUUM* at minute 23 past every 2nd hour from 0 through 20 every day Postgres server time zone:

```
SELECT timetable.add_job('run-vacuum', '23 0-20/2 * * *', 'VACUUM');
```

3. Refresh materialized view every 2 hours:

```
SELECT timetable.add_job('refresh-matview', '@every 2 hours', 'REFRESH␣
↪MATERIALIZED VIEW public.mat_view');
```

4. Clear log table after **pg_timetable** restart:

```
SELECT timetable.add_job('clear-log', '@reboot', 'TRUNCATE timetable.log');
```

5. Reindex at midnight Postgres server time zone on Sundays with reindexdb utility:

   - using default database under default user (no command line arguments)

```
SELECT timetable.add_job('reindex', '0 0 * * 7', 'reindexdb', job_kind␣
↪:= 'PROGRAM');
```

   - specifying target database and tables, and be verbose

```
SELECT timetable.add_job('reindex', '0 0 * * 7', 'reindexdb',
    '["--table=foo", "--dbname=postgres", "--verbose"]'::jsonb, 'PROGRAM
↪');
```

   - passing password using environment variable through `bash` shell

```
SELECT timetable.add_job('reindex', '0 0 * * 7', 'bash',
    '["-c", "PGPASSWORD=5m3R7K4754p4m reindexdb -U postgres -h 192.168.0.
↪221 -v"]'::jsonb,
    'PROGRAM');
```

CHAPTER

SIX

SAMPLES

6.1 Basic

This sample demonstrates how to create a basic one-step chain with parameters. It uses CTE to directly update the **timetable** schema tables.

```
SELECT timetable.add_job(
    job_name            => 'notify every minute',
    job_schedule        => '* * * * *',
    job_command         => 'SELECT pg_notify($1, $2)',
    job_parameters      => '[ "TT_CHANNEL", "Ahoj from SQL base task" ]' :: jsonb,
    job_kind            => 'SQL'::timetable.command_kind,
    job_client_name     => NULL,
    job_max_instances   => 1,
    job_live            => TRUE,
    job_self_destruct   => FALSE,
    job_ignore_errors   => TRUE
) as chain_id;
```

6.2 Download, Transform and Import

This sample demonstrates how to create enhanced three-step chain with parameters. It uses DO statement to directly update the **timetable** schema tables.

```
-- An enhanced example consisting of three tasks:
-- 1. Download text file from internet using BUILT-IN command
-- 2. Remove accents (diacritic signs) from letters using PROGRAM command (can be done with `unaccent` PostgreSQL extension)
-- 3. Import text file as CSV file using BUILT-IN command (can be down with `psql -c /copy`)
DO $$
DECLARE
    v_head_id bigint;
    v_task_id bigint;
    v_chain_id bigint;
BEGIN
    -- Create the chain with default values executed every minute (NULL == '* * * * *':: timetable.cron)
    INSERT INTO timetable.chain (chain_name, live)
```

(continues on next page)

(continued from previous page)

```
    VALUES ('Download locations and aggregate', TRUE)
    RETURNING chain_id INTO v_chain_id;

    -- Step 1. Download file from the server
    -- Create the chain
    INSERT INTO timetable.task (chain_id, task_order, kind, command, ignore_error)
    VALUES (v_chain_id, 1, 'BUILTIN', 'Download', TRUE)
    RETURNING task_id INTO v_task_id;

    -- Create the parameters for the step 1:
    INSERT INTO timetable.parameter (task_id, order_id, value)
        VALUES (v_task_id, 1,
           '{
                "workersnum": 1,
                "fileurls": ["https://www.cybertec-postgresql.com/secret/orte.txt"],
                "destpath": "."
            }'::jsonb);

    RAISE NOTICE 'Step 1 completed. Chain added with ID: %; DownloadFile task added with␣
↪ID: %', v_chain_id, v_task_id;

    -- Step 2. Transform Unicode characters into ASCII
    -- Create the program task to call 'uconv' and name it 'unaccent'
    INSERT INTO timetable.task (chain_id, task_order, kind, command, ignore_error, task_
↪name)
    VALUES (v_chain_id, 2, 'PROGRAM', 'uconv', TRUE, 'unaccent')
    RETURNING task_id INTO v_task_id;

    -- Create the parameters for the 'unaccent' task. Input and output files in this case
    -- Under Windows we should call PowerShell instead of "uconv" with command:
    -- Set-content "orte_ansi.txt" ((Get-content "orte.txt").Normalize("FormD") -replace
↪'\p{M}', '')
    INSERT INTO timetable.parameter (task_id, order_id, value)
        VALUES (v_task_id, 1, '["-x", "Latin-ASCII", "-o", "orte_ansi.txt", "orte.txt"]
↪'::jsonb);

    RAISE NOTICE 'Step 2 completed. Unacent task added with ID: %', v_task_id;

    -- Step 3. Import ASCII file to PostgreSQL table using "CopyFromFile" built-in␣
↪command
    INSERT INTO timetable.task (chain_id, task_order, kind, command)
        VALUES (v_chain_id, 3, 'BUILTIN', 'CopyFromFile')
    RETURNING task_id INTO v_task_id;

    -- Prepare the destination table 'location'
    CREATE TABLE IF NOT EXISTS location(name text);

    -- Add the parameters for the download task. Execute client side COPY to 'location'␣
↪from 'orte_ansi.txt'
    INSERT INTO timetable.parameter (task_id, order_id, value)
        VALUES (v_task_id, 1, '{"sql": "COPY location FROM STDIN", "filename": "orte_
↪ansi.txt" }'::jsonb);
```

(continues on next page)

(continued from previous page)

```
58
59      RAISE NOTICE 'Step 3 completed. Import task added with ID: %', v_task_id;
60  END;
61  $$ LANGUAGE PLPGSQL;
```

6.3 Run tasks in autonomous transaction

This sample demonstrates how to run special tasks out of chain transaction context. This is useful for special routines and/or non-transactional operations, e.g. *CREATE DATABASE*, *REINDEX*, *VACUUM*, *CREATE TABLESPACE*, etc.

```
1   -- An advanced example showing how to use atutonomous tasks.
2   -- This one-task chain will execute test_proc() procedure.
3   -- Since procedure will make two commits (after f1() and f2())
4   -- we cannot use it as a regular task, because all regular tasks
5   -- must be executed in the context of a single chain transaction.
6   -- Same rule applies for some other SQL commands,
7   -- e.g. CREATE DATABASE, REINDEX, VACUUM, CREATE TABLESPACE, etc.
8   CREATE OR REPLACE FUNCTION f (msg TEXT) RETURNS void AS $$
9   BEGIN
10      RAISE notice '%', msg;
11  END;
12  $$ LANGUAGE PLPGSQL;
13
14  CREATE OR REPLACE PROCEDURE test_proc () AS $$
15  BEGIN
16      PERFORM f('hey 1');
17      COMMIT;
18      PERFORM f('hey 2');
19      COMMIT;
20  END;
21  $$
22  LANGUAGE PLPGSQL;
23
24  WITH
25      cte_chain (v_chain_id) AS (
26          INSERT INTO timetable.chain (chain_name, run_at, max_instances, live, self_
    ↪destruct)
27          VALUES (
28              'call proc() every 10 sec', -- chain_name,
29              '@every 10 seconds',        -- run_at,
30              1,      -- max_instances,
31              TRUE,   -- live,
32              FALSE -- self_destruct
33          ) RETURNING chain_id
34      ),
35      cte_task(v_task_id) AS (
36          INSERT INTO timetable.task (chain_id, task_order, kind, command, ignore_error,␣
    ↪autonomous)
37          SELECT v_chain_id, 10, 'SQL', 'CALL test_proc()', TRUE, TRUE
38          FROM cte_chain
```

(continues on next page)

(continued from previous page)

```
        RETURNING task_id
    )
SELECT v_chain_id, v_task_id FROM cte_task, cte_chain;
```

6.4 Shutdown the scheduler and terminate the session

This sample demonstrates how to shutdown the scheduler using special built-in task. This can be used to control maintenance windows, to restart the scheduler for update purposes, or to stop session before the database should be dropped.

```
-- This one-task chain (aka job) will terminate pg_timetable session.
-- This is useful for maintaining purposes or before database being destroyed.
-- One should take care of restarting pg_timetable if needed.

SELECT timetable.add_job (
    job_name     => 'Shutdown pg_timetable session on schedule',
    job_schedule => '* * 1 * *',
    job_command  => 'Shutdown',
    job_kind     => 'BUILTIN'
);
```

6.5 Access previous task result code and output from the next task

This sample demonstrates how to check the result code and output of a previous task. If the last task failed, that is possible only if *ignore_error boolean = true* is set for that task. Otherwise, a scheduler will stop the chain. This sample shows how to calculate failed, successful, and the total number of tasks executed. Based on these values, we can calculate the success ratio.

```
WITH
    cte_chain (v_chain_id) AS ( -- let's create a new chain and add tasks to it later
        INSERT INTO timetable.chain (chain_name, run_at, max_instances, live)
        VALUES ('many tasks', '* * * * *', 1, true)
        RETURNING chain_id
    ),
    cte_tasks(v_task_id) AS ( -- now we'll add 500 tasks to the chain, some of them will␣
↪fail
        INSERT INTO timetable.task (chain_id, task_order, kind, command, ignore_error)
        SELECT v_chain_id, g.s, 'SQL', 'SELECT 1.0 / round(random())::int4;', true
        FROM cte_chain, generate_series(1, 500) AS g(s)
        RETURNING task_id
    ),
    report_task(v_task_id) AS ( -- and the last reporting task will calculate the␣
↪statistic
        INSERT INTO timetable.task (chain_id, task_order, kind, command)
        SELECT v_chain_id, 501, 'SQL', $CMD$DO
$$
DECLARE
    s TEXT;
```

(continues on next page)

(continued from previous page)

```
BEGIN
    WITH report AS (
        SELECT
        count(*) FILTER (WHERE returncode = 0) AS success,
        count(*) FILTER (WHERE returncode != 0) AS fail,
        count(*) AS total
        FROM timetable.execution_log
        WHERE chain_id = current_setting('pg_timetable.current_chain_id')::bigint
          AND txid = txid_current()
    )
    SELECT 'Tasks executed:' || total ||
         '; succeeded: ' || success ||
         '; failed: ' || fail ||
         '; ratio: ' || 100.0*success/GREATEST(total,1)
    INTO s
    FROM report;
    RAISE NOTICE '%', s;
END;
$$
$CMD$
        FROM cte_chain
        RETURNING task_id
    )
SELECT v_chain_id FROM cte_chain
```

CHAPTER

SEVEN

MIGRATION FROM OTHERS SCHEDULERS

7.1 Migrate jobs from pg_cron to pg_timetable

If you want to quickly export jobs scheduled from *pg_cron* to *pg_timetable*, you can use this SQL snippet:

```
SELECT timetable.add_job(
    job_name            => COALESCE(jobname, 'job: ' || command),
    job_schedule        => schedule,
    job_command         => command,
    job_kind            => 'SQL',
    job_live            => active
) FROM cron.job;
```

The *timetable.add_job()*, however, has some limitations. First of all, the function will mark the task created as **autonomous**, specifying scheduler should execute the task out of the chain transaction. It's not an error, but many autonomous chains may cause some extra connections to be used.

Secondly, database connection parameters are lost for source *pg_cron* jobs, making all jobs local. To export every information available precisely as possible, use this SQL snippet under the role they were scheduled in *pg_cron*:

```
SET ROLE 'scheduler'; -- set the role used by pg_cron

WITH cron_chain AS (
    SELECT
        nextval('timetable.chain_chain_id_seq'::regclass) AS cron_id,
        jobname,
        schedule,
        active,
        command,
        CASE WHEN
            database != current_database()
            OR nodename != 'localhost'
            OR username != CURRENT_USER
            OR nodeport != inet_server_port()
        THEN
            format('host=%s port=%s dbname=%s user=%s', nodename, nodeport, database,␣
↪username)
        END AS connstr
    FROM
        cron.job
),
```

(continues on next page)

(continued from previous page)

```
cte_chain AS (
        INSERT INTO timetable.chain (chain_id, chain_name, run_at, live)
            SELECT
                    cron_id, COALESCE(jobname, 'cronjob' || cron_id), schedule, active
            FROM
                    cron_chain
),
cte_tasks AS (
        INSERT INTO timetable.task (chain_id, task_order, kind, command, database_
↪connection)
            SELECT
                    cron_id, 1, 'SQL', command, connstr
            FROM
                    cron_chain
            RETURNING
                chain_id, task_id
)
SELECT * FROM cte_tasks;
```

CHAPTER

EIGHT

REST API

pg_timetable has a rich REST API, which can be used by external tools in order to perform start/stop/reinitialize/restarts/reloads, by any kind of tools to perform HTTP health checks, and of course, could also be used for monitoring.

Below you will find the list of **pg_timetable** REST API endpoints.

8.1 Health check endpoints

Currently, there are two health check endpoints available:

`GET /liveness`
: Always returns HTTP status code `200` what only indicates that **pg_timetable** is running.

`GET /readiness`
: Returns HTTP status code `200` when the **pg_timetable** is running and the scheduler is in the main loop processing chains. If the scheduler connects to the database, creates the database schema, or upgrades it, it will return HTTP status code `503`.

CHAPTER

NINE

DATABASE SCHEMA

pg_timetable is a database driven application. During the first start the necessary schema is created if absent.

9.1 Main tables and objects

```
CREATE SCHEMA timetable;

-- define migrations you need to apply
-- every change to this file should populate this table.
-- Version value should contain issue number zero padded followed by
-- short description of the issue\feature\bug implemented\resolved
CREATE TABLE timetable.migration(
    id INT8 NOT NULL,
    version TEXT NOT NULL,
    PRIMARY KEY (id)
);

INSERT INTO
    timetable.migration (id, version)
VALUES
    (0, '00259 Restart migrations for v4'),
    (1, '00305 Fix timetable.is_cron_in_time'),
    (2, '00323 Append timetable.delete_job function'),
    (3, '00329 Migration required for some new added functions'),
    (4, '00334 Refactor timetable.task as plain schema without tree-like dependencies'),
    (5, '00381 Rewrite active chain handling'),
    (6, '00394 Add started_at column to active_session and active_chain tables'),
    (7, '00417 Rename LOG database log level to INFO'),
    (8, '00436 Add txid column to timetable.execution_log');

CREATE DOMAIN timetable.cron AS TEXT CHECK(
    substr(VALUE, 1, 6) IN ('@every', '@after') AND (substr(VALUE, 7) :: INTERVAL) IS␣
↪NOT NULL
    OR VALUE = '@reboot'
    OR VALUE ~ '^(((\d+,)+\d+|(\d+(\/|-)\d+)|(\*(\/|-)\d+)|\d+|\*) +){4}(((\d+,)+\d+|(\
↪d+(\/|-)\d+)|(\*(\/|-)\d+)|\d+|\*) ?)$'
);

COMMENT ON DOMAIN timetable.cron IS 'Extended CRON-style notation with support of␣
↪interval values';
```

(continues on next page)

(continued from previous page)

```

CREATE TABLE timetable.chain (
    chain_id            BIGSERIAL   PRIMARY KEY,
    chain_name          TEXT        NOT NULL UNIQUE,
    run_at              timetable.cron,
    max_instances       INTEGER,
    timeout             INTEGER     DEFAULT 0,
    live                BOOLEAN     DEFAULT FALSE,
    self_destruct       BOOLEAN     DEFAULT FALSE,
    exclusive_execution BOOLEAN     DEFAULT FALSE,
    client_name         TEXT
);

COMMENT ON TABLE timetable.chain IS
    'Stores information about chains schedule';
COMMENT ON COLUMN timetable.chain.run_at IS
    'Extended CRON-style time notation the chain has to be run at';
COMMENT ON COLUMN timetable.chain.max_instances IS
    'Number of instances (clients) this chain can run in parallel';
COMMENT ON COLUMN timetable.chain.timeout IS
    'Abort any chain that takes more than the specified number of milliseconds';
COMMENT ON COLUMN timetable.chain.live IS
    'Indication that the chain is ready to run, set to FALSE to pause execution';
COMMENT ON COLUMN timetable.chain.self_destruct IS
    'Indication that this chain will delete itself after successful run';
COMMENT ON COLUMN timetable.chain.exclusive_execution IS
    'All parallel chains should be paused while executing this chain';
COMMENT ON COLUMN timetable.chain.client_name IS
    'Only client with this name is allowed to run this chain, set to NULL to allow any␣
↪client';

CREATE TYPE timetable.command_kind AS ENUM ('SQL', 'PROGRAM', 'BUILTIN');

CREATE TABLE timetable.task (
    task_id             BIGSERIAL               PRIMARY KEY,
    chain_id            BIGINT                  REFERENCES timetable.chain(chain_id) ON␣
↪UPDATE CASCADE ON DELETE CASCADE,
    task_order          DOUBLE PRECISION        NOT NULL,
    task_name           TEXT,
    kind                timetable.command_kind  NOT NULL DEFAULT 'SQL',
    command             TEXT                    NOT NULL,
    run_as              TEXT,
    database_connection TEXT,
    ignore_error        BOOLEAN                 NOT NULL DEFAULT FALSE,
    autonomous          BOOLEAN                 NOT NULL DEFAULT FALSE,
    timeout             INTEGER                 DEFAULT 0
);

COMMENT ON TABLE timetable.task IS
    'Holds information about chain elements aka tasks';
COMMENT ON COLUMN timetable.task.chain_id IS
    'Link to the chain, if NULL task considered to be disabled';
```

(continues on next page)

(continued from previous page)

```
COMMENT ON COLUMN timetable.task.task_order IS
    'Indicates the order of task within a chain';
COMMENT ON COLUMN timetable.task.run_as IS
    'Role name to run task as. Uses SET ROLE for SQL commands';
COMMENT ON COLUMN timetable.task.ignore_error IS
    'Indicates whether a next task in a chain can be executed regardless of the success␣
↪of the current one';
COMMENT ON COLUMN timetable.task.kind IS
    'Indicates whether "command" is SQL, built-in function or an external program';
COMMENT ON COLUMN timetable.task.command IS
    'Contains either an SQL command, or command string to be executed';
COMMENT ON COLUMN timetable.task.timeout IS
    'Abort any task within a chain that takes more than the specified number of␣
↪milliseconds';

-- parameter passing for a chain task
CREATE TABLE timetable.parameter(
    task_id     BIGINT  REFERENCES timetable.task(task_id)
                        ON UPDATE CASCADE ON DELETE CASCADE,
    order_id    INTEGER CHECK (order_id > 0),
    value       JSONB,
    PRIMARY KEY (task_id, order_id)
);

COMMENT ON TABLE timetable.parameter IS
    'Stores parameters passed as arguments to a chain task';

CREATE UNLOGGED TABLE timetable.active_session(
    client_pid  BIGINT  NOT NULL,
    client_name TEXT    NOT NULL,
    server_pid  BIGINT  NOT NULL,
    started_at  TIMESTAMPTZ DEFAULT now()
);

COMMENT ON TABLE timetable.active_session IS
    'Stores information about active sessions';

CREATE TYPE timetable.log_type AS ENUM ('DEBUG', 'NOTICE', 'INFO', 'ERROR', 'PANIC',
↪'USER');

CREATE OR REPLACE FUNCTION timetable.get_client_name(integer) RETURNS TEXT AS
$$
    SELECT client_name FROM timetable.active_session WHERE server_pid = $1 LIMIT 1
$$
LANGUAGE sql;

CREATE TABLE timetable.log
(
    ts              TIMESTAMPTZ         DEFAULT now(),
    pid             INTEGER             NOT NULL,
    log_level       timetable.log_type  NOT NULL,
    client_name     TEXT                DEFAULT timetable.get_client_name(pg_backend_
↪pid()),
```

(continues on next page)

(continued from previous page)

```
    message         TEXT,
    message_data    jsonb
);

COMMENT ON TABLE timetable.log IS
    'Stores log entries of active sessions';

CREATE TABLE timetable.execution_log (
    chain_id    BIGINT,
    task_id     BIGINT,
    txid        INTEGER NOT NULL,
    last_run    TIMESTAMPTZ DEFAULT now(),
    finished    TIMESTAMPTZ,
    pid         BIGINT,
    returncode  INTEGER,
    kind        timetable.command_kind,
    command     TEXT,
    output      TEXT,
    client_name TEXT        NOT NULL
);

COMMENT ON TABLE timetable.execution_log IS
    'Stores log entries of executed tasks and chains';

CREATE UNLOGGED TABLE timetable.active_chain(
    chain_id    BIGINT  NOT NULL,
    client_name TEXT    NOT NULL,
    started_at  TIMESTAMPTZ DEFAULT now()
);

COMMENT ON TABLE timetable.active_chain IS
    'Stores information about active chains within session';

CREATE OR REPLACE FUNCTION timetable.try_lock_client_name(worker_pid BIGINT, worker_name␣
↪TEXT)
RETURNS bool AS
$CODE$
BEGIN
    IF pg_is_in_recovery() THEN
        RAISE NOTICE 'Cannot obtain lock on a replica. Please, use the primary node';
        RETURN FALSE;
    END IF;
    -- remove disconnected sessions
    DELETE
        FROM timetable.active_session
        WHERE server_pid NOT IN (
            SELECT pid
            FROM pg_catalog.pg_stat_activity
            WHERE application_name = 'pg_timetable'
        );
    DELETE
        FROM timetable.active_chain
```

(continues on next page)

(continued from previous page)

```
        WHERE client_name NOT IN (
            SELECT client_name FROM timetable.active_session
        );
    -- check if there any active sessions with the client name but different client pid
    PERFORM 1
        FROM timetable.active_session s
        WHERE
            s.client_pid <> worker_pid
            AND s.client_name = worker_name
        LIMIT 1;
    IF FOUND THEN
        RAISE NOTICE 'Another client is already connected to server with name: %',␣
↪worker_name;
        RETURN FALSE;
    END IF;
    -- insert current session information
    INSERT INTO timetable.active_session(client_pid, client_name, server_pid) VALUES␣
↪(worker_pid, worker_name, pg_backend_pid());
    RETURN TRUE;
END;
$CODE$
STRICT
LANGUAGE plpgsql;

```

9.2 Jobs related functions

```
-- add_task() will add a task to the same chain as the task with `parent_id`
CREATE OR REPLACE FUNCTION timetable.add_task(
    IN kind timetable.command_kind,
    IN command TEXT,
    IN parent_id BIGINT,
    IN order_delta DOUBLE PRECISION DEFAULT 10
) RETURNS BIGINT AS $$
    INSERT INTO timetable.task (chain_id, task_order, kind, command)
        SELECT chain_id, task_order + $4, $1, $2 FROM timetable.task WHERE task_id = $3
        RETURNING task_id
$$ LANGUAGE SQL;

COMMENT ON FUNCTION timetable.add_task IS 'Add a task to the same chain as the task with␣
↪parent_id';

-- add_job() will add one-task chain to the system
CREATE OR REPLACE FUNCTION timetable.add_job(
    job_name            TEXT,
    job_schedule        timetable.cron,
    job_command         TEXT,
    job_parameters      JSONB DEFAULT NULL,
    job_kind            timetable.command_kind DEFAULT 'SQL'::timetable.command_kind,
    job_client_name     TEXT DEFAULT NULL,
```

(continues on next page)

(continued from previous page)

```
    job_max_instances   INTEGER DEFAULT NULL,
    job_live            BOOLEAN DEFAULT TRUE,
    job_self_destruct   BOOLEAN DEFAULT FALSE,
    job_ignore_errors   BOOLEAN DEFAULT TRUE,
    job_exclusive       BOOLEAN DEFAULT FALSE
) RETURNS BIGINT AS $$
    WITH
        cte_chain (v_chain_id) AS (
            INSERT INTO timetable.chain (chain_name, run_at, max_instances, live, self_destruct, client_name, exclusive_execution)
            VALUES (job_name, job_schedule,job_max_instances, job_live, job_self_destruct, job_client_name, job_exclusive)
            RETURNING chain_id
        ),
        cte_task(v_task_id) AS (
            INSERT INTO timetable.task (chain_id, task_order, kind, command, ignore_error, autonomous)
            SELECT v_chain_id, 10, job_kind, job_command, job_ignore_errors, TRUE
            FROM cte_chain
            RETURNING task_id
        ),
        cte_param AS (
            INSERT INTO timetable.parameter (task_id, order_id, value)
            SELECT v_task_id, 1, job_parameters FROM cte_task, cte_chain
        )
        SELECT v_chain_id FROM cte_chain
$$ LANGUAGE SQL;

COMMENT ON FUNCTION timetable.add_job IS 'Add one-task chain (aka job) to the system';

-- notify_chain_start() will send notification to the worker to start the chain
CREATE OR REPLACE FUNCTION timetable.notify_chain_start(
    chain_id BIGINT,
    worker_name TEXT
) RETURNS void AS $$
    SELECT pg_notify(
        worker_name,
        format('{"ConfigID": %s, "Command": "START", "Ts": %s}',
        chain_id,
        EXTRACT(epoch FROM clock_timestamp())::bigint)
    )
$$ LANGUAGE SQL;

COMMENT ON FUNCTION timetable.notify_chain_start IS 'Send notification to the worker to start the chain';

-- notify_chain_stop() will send notification to the worker to stop the chain
CREATE OR REPLACE FUNCTION timetable.notify_chain_stop(
    chain_id BIGINT,
    worker_name TEXT
) RETURNS void AS  $$
    SELECT pg_notify(
```

(continues on next page)

(continued from previous page)

```
        worker_name,
        format('{"ConfigID": %s, "Command": "STOP", "Ts": %s}',
            chain_id,
            EXTRACT(epoch FROM clock_timestamp())::bigint)
        )
$$ LANGUAGE SQL;

COMMENT ON FUNCTION timetable.notify_chain_stop IS 'Send notification to the worker to stop the chain';

-- move_task_up() will switch the order of the task execution with a previous task within the chain
CREATE OR REPLACE FUNCTION timetable.move_task_up(IN task_id BIGINT) RETURNS boolean AS $$
        WITH current_task (ct_chain_id, ct_id, ct_order) AS (
                SELECT chain_id, task_id, task_order FROM timetable.task WHERE task_id = $1
        ),
        tasks(t_id, t_new_order) AS (
                SELECT task_id, COALESCE(LAG(task_order) OVER w, LEAD(task_order) OVER w)
                FROM timetable.task t, current_task ct
                WHERE chain_id = ct_chain_id AND (task_order < ct_order OR task_id = ct_id)
                WINDOW w AS (PARTITION BY chain_id ORDER BY ABS(task_order - ct_order))
                LIMIT 2
        ),
        upd AS (
                UPDATE timetable.task t SET task_order = t_new_order
                FROM tasks WHERE tasks.t_id = t.task_id AND tasks.t_new_order IS NOT NULL
                RETURNING true
        )
        SELECT COUNT(*) > 0 FROM upd
$$ LANGUAGE SQL;

COMMENT ON FUNCTION timetable.move_task_up IS 'Switch the order of the task execution with a previous task within the chain';

-- move_task_down() will switch the order of the task execution with a following task within the chain
CREATE OR REPLACE FUNCTION timetable.move_task_down(IN task_id BIGINT) RETURNS boolean AS $$
        WITH current_task (ct_chain_id, ct_id, ct_order) AS (
                SELECT chain_id, task_id, task_order FROM timetable.task WHERE task_id = $1
        ),
        tasks(t_id, t_new_order) AS (
                SELECT task_id, COALESCE(LAG(task_order) OVER w, LEAD(task_order) OVER w)
                FROM timetable.task t, current_task ct
                WHERE chain_id = ct_chain_id AND (task_order > ct_order OR task_id = ct_id)
                WINDOW w AS (PARTITION BY chain_id ORDER BY ABS(task_order - ct_order))
                LIMIT 2
```

(continues on next page)

(continued from previous page)

```
        ),
        upd AS (
                UPDATE timetable.task t SET task_order = t_new_order
                FROM tasks WHERE tasks.t_id = t.task_id AND tasks.t_new_order IS NOT NULL
                RETURNING true
        )
        SELECT COUNT(*) > 0 FROM upd
$$ LANGUAGE SQL;

COMMENT ON FUNCTION timetable.move_task_down IS 'Switch the order of the task execution with a following task within the chain';

-- delete_job() will delete the chain and its tasks from the system
CREATE OR REPLACE FUNCTION timetable.delete_job(IN job_name TEXT) RETURNS boolean AS $$
    WITH del_chain AS (DELETE FROM timetable.chain WHERE chain.chain_name = $1 RETURNING chain_id)
    SELECT EXISTS(SELECT 1 FROM del_chain)
$$ LANGUAGE SQL;

COMMENT ON FUNCTION timetable.delete_job IS 'Delete the chain and its tasks from the system';

-- delete_task() will delete the task from a chain
CREATE OR REPLACE FUNCTION timetable.delete_task(IN task_id BIGINT) RETURNS boolean AS $$
    WITH del_task AS (DELETE FROM timetable.task WHERE task_id = $1 RETURNING task_id)
    SELECT EXISTS(SELECT 1 FROM del_task)
$$ LANGUAGE SQL;

COMMENT ON FUNCTION timetable.delete_task IS 'Delete the task from a chain';
```

9.3 ron related functions

```
CREATE OR REPLACE FUNCTION timetable.cron_split_to_arrays(
    cron text,
    OUT mins integer[],
    OUT hours integer[],
    OUT days integer[],
    OUT months integer[],
    OUT dow integer[]
) RETURNS record AS $$
DECLARE
    a_element text[];
    i_index integer;
    a_tmp text[];
    tmp_item text;
    a_range int[];
    a_split text[];
    a_res integer[];
    allowed_range integer[];
    max_val integer;
```

(continues on next page)

(continued from previous page)

```
    min_val integer;
BEGIN
    a_element := regexp_split_to_array(cron, '\s+');
    FOR i_index IN 1..5 LOOP
        a_res := NULL;
        a_tmp := string_to_array(a_element[i_index],',');
        CASE i_index -- 1 - mins, 2 - hours, 3 - days, 4 - weeks, 5 - DOWs
            WHEN 1 THEN allowed_range := '{0,59}';
            WHEN 2 THEN allowed_range := '{0,23}';
            WHEN 3 THEN allowed_range := '{1,31}';
            WHEN 4 THEN allowed_range := '{1,12}';
        ELSE
            allowed_range := '{0,7}';
        END CASE;
        FOREACH  tmp_item IN ARRAY a_tmp LOOP
            IF tmp_item ~ '^[0-9]+$' THEN -- normal integer
                a_res := array_append(a_res, tmp_item::int);
            ELSIF tmp_item ~ '^[*]+$' THEN -- '*' any value
                a_range := array(select generate_series(allowed_range[1], allowed_
↪range[2]));
                a_res := array_cat(a_res, a_range);
            ELSIF tmp_item ~ '^[0-9]+[-][0-9]+$' THEN -- '-' range of values
                a_range := regexp_split_to_array(tmp_item, '-');
                a_range := array(select generate_series(a_range[1], a_range[2]));
                a_res := array_cat(a_res, a_range);
            ELSIF tmp_item ~ '^[0-9]+[\/][0-9]+$' THEN -- '/' step values
                a_range := regexp_split_to_array(tmp_item, '/');
                a_range := array(select generate_series(a_range[1], allowed_range[2], a_
↪range[2]));
                a_res := array_cat(a_res, a_range);
            ELSIF tmp_item ~ '^[0-9-]+[\/][0-9]+$' THEN -- '-' range of values and '/'␣
↪step values
                a_split := regexp_split_to_array(tmp_item, '/');
                a_range := regexp_split_to_array(a_split[1], '-');
                a_range := array(select generate_series(a_range[1], a_range[2], a_
↪split[2]::int));
                a_res := array_cat(a_res, a_range);
            ELSIF tmp_item ~ '^[*]+[\/][0-9]+$' THEN -- '*' any value and '/' step values
                a_split := regexp_split_to_array(tmp_item, '/');
                a_range := array(select generate_series(allowed_range[1], allowed_
↪range[2], a_split[2]::int));
                a_res := array_cat(a_res, a_range);
            ELSE
                RAISE EXCEPTION 'Value ("%") not recognized', a_element[i_index]
                    USING HINT = 'fields separated by space or tab.'+
                       'Values allowed: numbers (value list with ","), '+
                    'any value with "*", range of value with "-" and step values with "/
↪"!';
            END IF;
        END LOOP;
        SELECT
           ARRAY_AGG(x.val), MIN(x.val), MAX(x.val) INTO a_res, min_val, max_val
```

(continues on next page)

(continued from previous page)

```
        FROM (
            SELECT DISTINCT UNNEST(a_res) AS val ORDER BY val) AS x;
        IF max_val > allowed_range[2] OR min_val < allowed_range[1] THEN
            RAISE EXCEPTION '% is out of range: %', a_res, allowed_range;
        END IF;
        CASE i_index
                  WHEN 1 THEN mins := a_res;
            WHEN 2 THEN hours := a_res;
            WHEN 3 THEN days := a_res;
            WHEN 4 THEN months := a_res;
        ELSE
            dow := a_res;
        END CASE;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE PLPGSQL STRICT;

CREATE OR REPLACE FUNCTION timetable.cron_months(
    from_ts timestamptz,
    allowed_months int[]
) RETURNS SETOF timestamptz AS $$
    WITH
    am(am) AS (SELECT UNNEST(allowed_months)),
    genm(ts) AS ( --generated months
        SELECT date_trunc('month', ts)
        FROM pg_catalog.generate_series(from_ts, from_ts + INTERVAL '1 year', INTERVAL
↪'1 month') g(ts)
    )
    SELECT ts FROM genm JOIN am ON date_part('month', genm.ts) = am.am
$$ LANGUAGE SQL STRICT;

CREATE OR REPLACE FUNCTION timetable.cron_days(
    from_ts timestamptz,
    allowed_months int[],
    allowed_days int[],
    allowed_week_days int[]
) RETURNS SETOF timestamptz AS $$
    WITH
    ad(ad) AS (SELECT UNNEST(allowed_days)),
    am(am) AS (SELECT * FROM timetable.cron_months(from_ts, allowed_months)),
    gend(ts) AS ( --generated days
        SELECT date_trunc('day', ts)
        FROM am,
            pg_catalog.generate_series(am.am, am.am + INTERVAL '1 month'
                - INTERVAL '1 day',  -- don't include the same day of the next month
                INTERVAL '1 day') g(ts)
    )
    SELECT ts
    FROM gend JOIN ad ON date_part('day', gend.ts) = ad.ad
    WHERE extract(dow from ts)=ANY(allowed_week_days)
$$ LANGUAGE SQL STRICT;
```

(continues on next page)

(continued from previous page)

```
116
117 CREATE OR REPLACE FUNCTION timetable.cron_times(
118     allowed_hours int[],
119     allowed_minutes int[]
120 ) RETURNS SETOF time AS $$
121     WITH
122     ah(ah) AS (SELECT UNNEST(allowed_hours)),
123     am(am) AS (SELECT UNNEST(allowed_minutes))
124     SELECT make_time(ah.ah, am.am, 0) FROM ah CROSS JOIN am
125 $$ LANGUAGE SQL STRICT;
126
127
128 CREATE OR REPLACE FUNCTION timetable.cron_runs(
129     from_ts timestamp with time zone,
130     cron text
131 ) RETURNS SETOF timestamptz AS $$
132     SELECT cd + ct
133     FROM
134         timetable.cron_split_to_arrays(cron) a,
135         timetable.cron_times(a.hours, a.mins) ct CROSS JOIN
136         timetable.cron_days(from_ts, a.months, a.days, a.dow) cd
137     WHERE cd + ct > from_ts
138     ORDER BY 1 ASC;
139 $$ LANGUAGE SQL STRICT;
140
141 -- is_cron_in_time returns TRUE if timestamp is listed in cron expression
142 CREATE OR REPLACE FUNCTION timetable.is_cron_in_time(
143     run_at timetable.cron,
144     ts timestamptz
145 ) RETURNS BOOLEAN AS $$
146     SELECT
147     CASE WHEN run_at IS NULL THEN
148         TRUE
149     ELSE
150         date_part('month', ts) = ANY(a.months)
151         AND (date_part('dow', ts) = ANY(a.dow) OR date_part('isodow', ts) = ANY(a.dow))
152         AND date_part('day', ts) = ANY(a.days)
153         AND date_part('hour', ts) = ANY(a.hours)
154         AND date_part('minute', ts) = ANY(a.mins)
155     END
156     FROM
157         timetable.cron_split_to_arrays(run_at) a
158 $$ LANGUAGE SQL;
159
160 CREATE OR REPLACE FUNCTION timetable.next_run(cron timetable.cron) RETURNS timestamptz␣
  ↪AS $$
161     SELECT * FROM timetable.cron_runs(now(), cron) LIMIT 1
162 $$ LANGUAGE SQL STRICT;
163
```

9.4 ER-Diagram

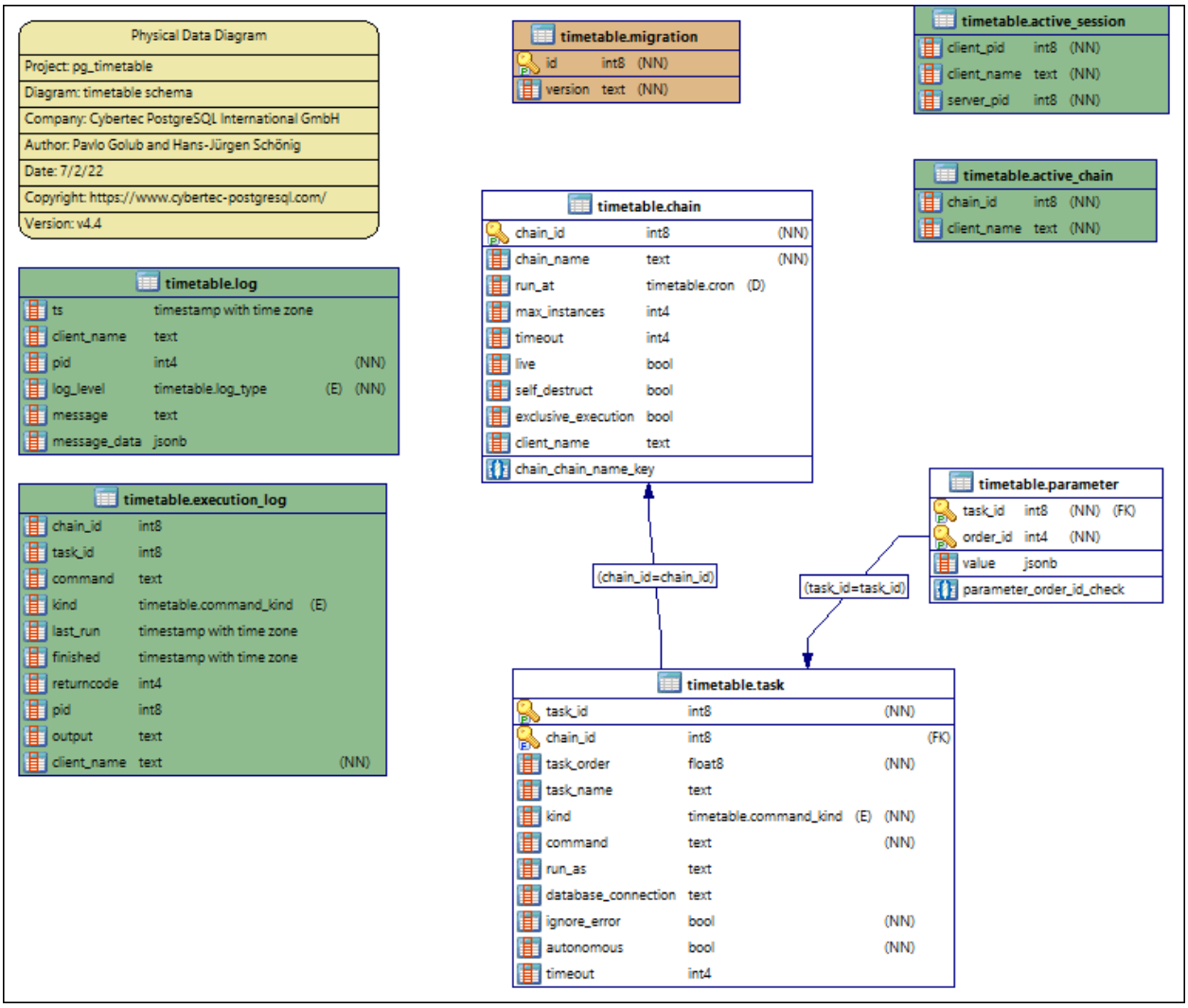

Fig. 1: ER-Diagram

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search