The pg_cron extension

Schedule and manage cron jobs directly within your Neon Postgres database

The `pg_cron` extension provides a simple, cron-based job scheduler for Postgres. It operates directly within your database, allowing you to schedule standard SQL commands or calls to stored procedures using familiar cron syntax. This eliminates the need for external cron utilities for many database maintenance and automation tasks.

This guide provides an introduction to the `pg_cron` extension. You'll learn how to enable the extension, schedule jobs, understand the cron syntax, manage and monitor your scheduled tasks, and about considerations specific to the Neon environment.

> **KEY DETAILS ABOUT USING PG_CRON WITH NEON**
>
> Please note that `pg_cron` jobs will only run when your compute is active. We therefore recommend only using `pg_cron` on computes that run 24/7 or where you have disabled scale to zero.

Enable the `pg_cron` extension

To install `pg_cron` on Neon, you must first enable it by setting the `cron.database_name` parameter to the name of the database where you want to install `pg_cron`. This requires making an Update compute endpoint API call.

The `cron.database_name` parameter is passed to your Postgres instance through the `pg_settings` option in the endpoint settings object. The following `Update endpoint` API example shows where to specify your Neon `project_id`, `endpoint_id`, Neon API key, and database name.

The `project_id` and `endpoint_id` values can be obtained from the Neon Console or using the Neon API. In the Neon Console, the `project_id` is found on your project's **Settings** page, and will look something like this: `young-sun-12345678`. The `endpoint_id` is found on the

Compute tab on your **Branches** page, where it is referred to as the **Endpoint ID**. It will have an `ep` prefix, and look similar to this: `ep-still-rain-abcd1234` .

```
curl --request PATCH \
     --url https://console.neon.tech/api/v2/projects/<project_id>/endpoints/<endpoint_id>
     --header 'accept: application/json' \
     --header 'authorization: Bearer $NEON_API_KEY$' \
     --header 'content-type: application/json' \
     --data '
{
  "endpoint": {
    "settings": {
      "pg_settings": {
        "cron.database_name": "your_dbname"
      }
    }
  }
}
'
```

After setting `cron.database_name` , you must restart your compute to apply the new setting. You can do this using the Restart compute endpoint API. Specify the same `project_id` and `endpoint_id` used to set the `cron.database_name` parameter above. **Please note that restarting your compute endpoint will drop current connections to your database.**

```
curl --request POST \
     --url https://console.neon.tech/api/v2/projects/<project_id>/endpoints/<endpoint_id>
     --header 'accept: application/json' \
     --header 'authorization: Bearer $NEON_API_KEY'
```

> **NOTE**
>
> The Restart compute endpoint API only works on an active compute. If your compute is idle, you can start it by running a query to wake it up or running the Start compute endpoint API. For more information and other compute restart options, see Restart a compute.

You can then install the `pg_cron` extension by running the following `CREATE EXTENSION` statement in the Neon SQL Editor or from a client such as psql that is connected to your Neon database.

```
CREATE EXTENSION IF NOT EXISTS pg_cron;
```

If you have trouble with this setup, please reach out to Neon Support or find us on Discord.

`pg_cron` version availability

Please refer to the list of all extensions available in Neon for up-to-date extension version information.

Cron schedule syntax

`pg_cron` uses the standard cron syntax, with the following fields:

```
┌──────────── min (0 - 59)
│ ┌──────────── hour (0 - 23)
│ │ ┌──────────── day of month (1 - 31) or last day of the month ($)
│ │ │ ┌──────────── month (1 - 12)
│ │ │ │ ┌──────────── day of week (0 - 6) (0 to 6 are Sunday to
│ │ │ │ │                  Saturday, or use names; 7 is also Sunday)
│ │ │ │ │
│ │ │ │ │
* * * * *
```

You can use the following special characters:

- `*` : Represents all values within the field.
- `,` : Specifies a list of values (e.g., `1,3,5` for specific days).
- `-` : Specifies a range of values (e.g., `10-12` for hours 10, 11, and 12).
- `/` : Specifies step values (e.g., `*/15` in the minutes field means "every 15 minutes").

Additionally, `pg_cron` supports:

- Interval scheduling using `'[1-59] seconds'` (e.g., `'5 seconds'`).
- `'$'` to indicate the last day of the month.

Remember that all schedules in `pg_cron` are interpreted in UTC. When scheduling jobs, ensure your cron expressions are set accordingly. You can use tools like crontab.guru and adjust for the UTC timezone.

Schedule a job

You can schedule jobs using the `cron.schedule()` function. The basic syntax involves providing a cron schedule string and the command to execute.

Let's look at some examples to understand how to schedule jobs with `pg_cron` .

Automating data archival

Imagine you have an `orders` table and you want to archive orders older than 90 days to a separate `orders_archive` table every Sunday at 2:00 AM UTC to maintain performance on your main table.

```
SELECT cron.schedule(
    'archive-old-orders',
    '0 2 * * 0',  -- Runs every Sunday at 2:00 AM UTC
    $$
        WITH OldOrders AS (
            SELECT *
            FROM orders
            WHERE order_date < NOW() - INTERVAL '90 days'
        )
        INSERT INTO orders_archive SELECT * FROM OldOrders;
        DELETE FROM orders WHERE order_id IN (SELECT order_id FROM OldOrders);
    $$
);
```

Here's a breakdown of the command:

- `'archive-old-orders'` : This is the name you're giving to this scheduled job. It helps you identify and manage the job later.
- `'0 2 * * 0'` : This is the cron schedule string.
  - `0` : The job will run when the minute is `0` .
  - `2` : The job will run when the hour is `2` (2 AM UTC).
  - `*` : The job will run every day of the month.
  - `*` : The job will run every month.
  - `0` : The job will run on Sunday (where 0 represents Sunday). Therefore, this job is scheduled to run at 2:00 AM UTC every Sunday.
- `$$ ... $$` : This is a way to define a string literal in PostgreSQL, especially useful for multi-line commands.
- `INSERT INTO orders_archive ...` : This is the SQL command that will be executed. It selects all rows from the `orders` table older than 90 days and inserts them into the `orders_archive` table. (A CTE is used to make sure the same rows are used for both the `INSERT` and `DELETE` commands.)
- `DELETE FROM orders ...` : This command then deletes the archived orders from the main `orders` table.

This example demonstrates how to automate a common database maintenance task, ensuring your main tables remain manageable and performant.

Purging cron job logs

The `cron.job_run_details` table keeps a record of your scheduled job executions. Over time, this table can grow and consume storage. Regularly purging older entries is a good practice to keep its size manageable.

You can schedule a job using `pg_cron` itself to automatically delete old records from `cron.job_run_details`. Here's how you can schedule a job to purge entries older than seven days, running every day at midnight UTC:

```
SELECT cron.schedule(
    'purge-cron-history',
    '0 0 * * *',  -- Runs every day at midnight UTC
    $$
        DELETE FROM cron.job_run_details
        WHERE end_time < NOW() - INTERVAL '7 days';
    $$
);
```

Here's a breakdown of the command:

purge-cron-history: The name of the scheduled job for purging history.

'0 0 * * *': The cron schedule, set to run at minute 0, hour 0 (midnight), every day of the month, every month, and every day of the week (all in UTC).

`DELETE FROM cron.job_run_details WHERE end_time < NOW() - INTERVAL '7 days'`: This is the SQL command that will be executed. It deletes all records from the `cron.job_run_details` table where the end_time is older than seven days from the current time.

Running jobs every `n` seconds

`pg_cron` also lets you schedule a job every `n` seconds, which is not possible with traditional cron jobs. Here `n` can be any value between 1 and 59 inclusive.

For example, to run a job every 10 seconds, you can use the following command:

```
SELECT cron.schedule('every-10-seconds', '10 seconds', 'SELECT 1');
```

View scheduled jobs

To see the jobs currently scheduled in your database, query the `cron.job` table:

```
SELECT * FROM cron.job;
```

This will show you details like the job ID, schedule, command, and the user who scheduled it.

Unschedule jobs

You can remove scheduled jobs using the `cron.unschedule()` function, either by providing the job name or the job ID.

Unschedule by name

Let's say you want to unschedule the job we created earlier to archive old orders:

```
SELECT cron.unschedule('archive-old-orders');
```

Unschedule by ID

You can also unschedule a job by providing the job ID:

```
SELECT cron.unschedule(26);
```

View job run details

The `cron.job_run_details` table provides information about the execution of scheduled jobs.

```
SELECT * FROM cron.job_run_details ORDER BY start_time DESC LIMIT 5;
```

This table includes details like the job ID, run ID, execution status, start and end times, and any return messages.

Running pg_cron jobs in multiple databases

The `pg_cron` extension can only be installed in one database per Postgres cluster (each compute in a Neon project runs a Postgres instance, i.e., a Postgres cluster). If you need to schedule jobs in multiple databases, you can use the `cron.schedule_in_database()` function. This function allows you to create a cron job that runs in a specific database, even if `pg_cron` is installed in a different database.

> ⚠ **FUNCTION NOT SUPPORTED IN NEON**
>
> The `cron.schedule_in_database()` function is currently not supported in Neon.

Example: Scheduling a job in a different database

To schedule a job in another database, use `cron.schedule_in_database()` and specify the target database name:

```sql
SELECT cron.schedule_in_database(
    'my_job',                       -- Job name
    '0 * * * *',                    -- Cron schedule (every hour)
    'my_database',                   -- Target database
    'VACUUM ANALYZE my_table'        -- SQL command to run
);
```

In this example:

- The job named `my_job` runs every hour `(0 * * * *)`.
- It executes `VACUUM ANALYZE my_table` in `my_database`, even if `pg_cron` is installed in another database.

Extension settings

`pg_cron` has several configuration parameters that influence its behavior. These settings are managed by Neon and cannot be directly modified by users. Understanding these settings can be helpful for monitoring and troubleshooting. You can view the current configuration in your Neon database using the following query:

```sql
SELECT * FROM pg_settings WHERE name LIKE 'cron.%';
```

Here are a few key `pg_cron` settings and their descriptions:

Setting	Default	Description
`cron.launch_active_jobs`	`on`	When set to `off`, this setting disables all active `pg_cron` jobs without requiring a server restart.
`cron.log_min_messages`	`WARNING`	This setting determines the minimum severity level of log messages generated by the `pg_cron` launcher background worker.
`cron.log_run`	`on`	When enabled (`on`), details of each job run are logged in the `cron.job_run_details` table.
`cron.log_statement`	`on`	If enabled (`on`), the SQL command of each scheduled job is logged before execution.
`cron.max_running_jobs`	`32`	This parameter defines the maximum number of `pg_cron` jobs that can run concurrently.
`cron.timezone`	`GMT`	Specifies the timezone in which the `pg_cron` background worker operates. **Note:** Although this setting exists, `pg_cron`

Setting	Default	Description
		internally interprets all job schedules in UTC. Changing this parameter has no effect on how schedules are executed.
`cron.use_background_workers`	`off`	When enabled (`on`), `pg_cron` uses background workers instead of direct client connections to execute jobs. This may require adjustments to the `max_worker_processes` PostgreSQL setting.

> **IMPORTANT: SETTING MODIFICATIONS IN NEON**
>
> It's important to note that because `pg_cron` is managed by Neon, modifying these settings requires superuser privileges. Therefore, you cannot directly alter these `pg_cron` configuration parameters yourself. If you have a specific need to adjust any of these settings, please open a support ticket. **After Neon support implements the requested configuration change, you will need to** restart your Neon compute **for the new settings to take effect.**

Conclusion

You have successfully learned how to enable and use the `pg_cron` extension within your Neon Postgres environment. You can now schedule routine database tasks directly within your database, simplifying automation and maintenance. Remember that `pg_cron` schedules are interpreted in UTC and will only run when your compute is active.

Resources

- pg_cron GitHub Repository
- crontab.guru

Need help?

Join our Discord Server to ask questions or see what others are doing with Neon. For paid plan support options, see Support.

← Previous
pgrag

Next →
pg_graphql

NEON

All systems operational

System