Package 'taskqueue'

December 3, 2025

Type Package

Title Task Queue for Parallel Computing Based on PostgreSQL

Version 0.2.0

Description Implements a task queue system for asynchronous parallel computing using 'PostgreSQL' <https://www.postgresql.org/> as a backend. Designed for embarrassingly parallel problems where tasks do not communicate with each other. Dynamically distributes tasks to workers, handles uneven load balancing, and allows new workers to join at any time. Particularly useful for running large numbers of independent tasks on high-performance computing (HPC) clusters with 'SLURM' <https://slurm.schedmd.com/> job schedulers.

URL https://taskqueue.bangyou.me/,
https://github.com/byzheng/taskqueue

BugReports https://github.com/byzheng/taskqueue/issues

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Imports settings, stats, stringr, RPostgres, ggplot2, whisker, rlang, shiny, DBI, ssh

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation no

Author Bangyou Zheng [aut, cre]

Maintainer Bangyou Zheng <bangyou.zheng@csiro.au>

Repository CRAN

Date/Publication 2025-12-03 20:30:02 UTC

Contents

.check_absolute_path *Check absolute path for system*

Description

Check absolute path for system

Usage

```
.check_absolute_path(path)
```

Arguments

path	File path to check

Value

No return is expected

.check_linux_absolute_path *Check absolute path for Linux*

Description

Check absolute path for Linux

Usage

```
.check_linux_absolute_path(path)
```

Arguments

path	File path to check

Value

No return is expected

.check_windows_absolute_path *Check absolute path for Windows*

Description

Check absolute path for Windows

Usage

```
.check_windows_absolute_path(path)
```

Arguments

path	File path to check

Value

No return is expected

db_clean *Clean All Tables and Definitions from Database*

Description

Removes all taskqueue-related tables, types, and data from the PostgreSQL database. This is a destructive operation that cannot be undone.

Usage

```
db_clean()
```

Details

This function drops:

- All project task tables
- The `project_resource` table
- The `project` table
- The `resource` table
- All custom types (e.g., `task_status`)

Warning: This permanently deletes all projects, tasks, and configurations. Use with extreme caution, typically only for testing or complete resets.

After cleaning, you must call `db_init` to recreate the schema before using taskqueue again.

Value

Invisibly returns NULL. Called for side effects (dropping database objects).

See Also

db_init

Examples

```
## Not run:
# Not run:
# Clean entire database (destructive!)
db_clean()

# Reinitialize after cleaning
db_init()

## End(Not run)
```

---

db_connect *Connect to PostgreSQL Database*

Description

Establishes a connection to the PostgreSQL database using credentials from environment variables or `taskqueue_options()`. If a valid connection is provided, it returns that connection instead of creating a new one.

Usage

```
db_connect(con = NULL)
```

Arguments

con	An existing database connection object. If provided and valid, this connection is returned. If NULL (default), a new connection is created.

Details

Connection parameters are read from environment variables set in `.Renviron`:

- `PGHOST`: Database server hostname
- `PGPORT`: Database server port (typically 5432)
- `PGUSER`: Database username
- `PGPASSWORD`: Database password
- `PGDATABASE`: Database name

The function automatically sets `client_min_messages` to WARNING to reduce console output noise.

Value

A PqConnection object from the RPostgres package that can be used for database operations.

See Also

db_disconnect, taskqueue_options

Examples

```
## Not run:
# Not run:
# Create a new connection
con <- db_connect()

# Reuse existing connection
con2 <- db_connect(con)

# Always disconnect when done
db_disconnect(con)

## End(Not run)
```

---

db_disconnect *Disconnect from PostgreSQL Database*

Description

Safely closes a database connection. Checks if the connection is valid before attempting to disconnect.

Usage

```
db_disconnect(con)
```

Arguments

con	A connection object as produced by db_connect or DBI::dbConnect.

Details

This function wraps RPostgres::dbDisconnect() with a validity check to avoid errors when disconnecting an already-closed connection.

Value

Invisibly returns NULL. Called for side effects.

See Also

db_connect

Examples

```
## Not run:
# Not run:
# Connect and disconnect
con <- db_connect()
# ... perform database operations ...
db_disconnect(con)

# Safe to call on.exit to ensure cleanup
con <- db_connect()
on.exit(db_disconnect(con), add = TRUE)

## End(Not run)
```

---

db_init *Initialize PostgreSQL Database for taskqueue*

Description

Creates the necessary database schema for taskqueue, including all required tables, types, and constraints. This function must be run once before using taskqueue for the first time.

Usage

```
db_init()
```

Details

This function creates:

- Custom PostgreSQL types (e.g., `task_status` enum)
- `project` table for managing projects
- `resource` table for computing resources
- `project_resource` table for project-resource associations

It is safe to call this function multiple times; existing tables and types will not be modified or deleted.

Value

Invisibly returns NULL. Called for side effects (creating database schema).

See Also

db_clean, db_connect

Examples

```
## Not run:
# Not run:
# Initialize database (run once)
db_init()

# Verify initialization
con <- db_connect()
DBI::dbListTables(con)
db_disconnect(con)

## End(Not run)
```

`db_sql` *A Wrapper function for DBI interface*

Description

A Wrapper function for DBI interface

Usage

```
db_sql(sql, method, con = NULL)
```

Arguments

`sql`	multile sql statements
`method`	method of DBI
`con`	a connection

Value

Results of last sql statement with method for DBI interface

`is_db_connect` *Test Database Connection*

Description

Checks whether a connection to the PostgreSQL database can be established with the current configuration.

Usage

```
is_db_connect()
```

Details

This function attempts to create a database connection using the credentials in environment variables or `taskqueue_options()`. It returns FALSE if the connection fails for any reason (wrong credentials, network issues, PostgreSQL not running, etc.).

Useful for testing database configuration before running workers or adding tasks.

Value

Logical. `TRUE` if the database can be connected successfully, `FALSE` otherwise.

See Also

`db_connect`, `taskqueue_options`

Examples

```
## Not run:
# Not run:
# Test connection
if (is_db_connect()) {
  message("Database is accessible")
  db_init()
} else {
  stop("Cannot connect to database. Check .Renviron settings.")
}

## End(Not run)
```

`project_add` *Create a New Project*

Description

Creates a new project in the database for managing a set of related tasks. Each project has its own task table and configuration.

Usage

```
project_add(project, memory = 10)
```

Arguments

`project`	Character string for the project name. Must be unique and cannot be a reserved name (e.g., "config").
`memory`	Memory requirement in gigabytes (GB) for each task in this project. Default is 10 GB.

Details

This function:

- Creates a new entry in the `project` table
- Creates a dedicated task table named `task_<project>`
- Sets default memory requirements for all tasks

If a project with the same name already exists, the memory requirement is updated but the task table remains unchanged.

After creating a project, you must:

1. Assign resources with `project_resource_add`
2. Add tasks with `task_add`
3. Start the project with `project_start`

Value

Invisibly returns NULL. Called for side effects (creating project in database).

See Also

`project_start`, `project_resource_add`, `task_add`, `project_delete`

Examples

```
## Not run:
# Not run:
# Create a project with default memory
project_add("simulation_study")

# Create with higher memory requirement
project_add("big_data_analysis", memory = 64)

# Verify project was created
project_list()

## End(Not run)
```

`project_delete` *Delete a Project*

Description

Permanently removes a project and all associated data from the database. This includes the project configuration, task table, and resource assignments.

Usage

```
project_delete(project, con = NULL)
```

Arguments

project	Character string specifying the project name.
con	An optional database connection. If NULL, a new connection is created and closed automatically.

Details

This function removes:

- The project's task table (task_<project>) and all tasks
- All project-resource associations
- The project entry from the project table

Warning: This is a destructive operation that cannot be undone. All task data and history for this project will be permanently lost.

Log files on resources are NOT automatically deleted. Remove them manually if needed.

Value

Invisibly returns NULL. Called for side effects (deleting project).

See Also

project_add, project_reset, db_clean

Examples

```
## Not run:
# Not run:
# Delete a completed project
project_delete("old_simulation")

# Verify deletion
project_list()

## End(Not run)
```

`project_get` *Get Project Information*

Description

Retrieves detailed information about a specific project from the database.

Usage

```
project_get(project, con = NULL)
```

Arguments

`project`	Character string specifying the project name.
`con`	An optional database connection. If NULL, a new connection is created and closed automatically.

Value

A single-row data frame containing project information with columns:

`id`	Unique project identifier
`name`	Project name
`table`	Name of the task table for this project
`status`	Logical indicating if project is running (TRUE) or stopped (FALSE)
`memory`	Memory requirement in GB for tasks

Stops with an error if the project is not found.

See Also

`project_add`, `project_list`, `project_resource_get`

Examples

```
## Not run:
# Not run:
# Get project details
info <- project_get("simulation_study")
print(info$status)  # Check if running
print(info$memory)  # Memory requirement

## End(Not run)
```

`project_list` *List All Projects*

Description

Retrieves information about all projects in the database.

Usage

```
project_list(con = NULL)
```

Arguments

`con`	An optional database connection. If NULL, a new connection is created and closed automatically.

Details

Returns NULL if the project table doesn't exist (i.e., `db_init` has not been called).

Value

A data frame with one row per project, or NULL if no projects exist. Columns include: id, name, table, status, and memory.

See Also

`project_add`, `project_get`

Examples

```
## Not run:
# Not run:
# List all projects
projects <- project_list()
print(projects)

# Find running projects
running <- projects[projects$status == TRUE, ]

## End(Not run)
```

`project_reset` *Reset a Project*

Description

Resets all tasks in a project to idle status, stops the project, and optionally cleans log files. Useful for restarting a project from scratch.

Usage

```
project_reset(project, log_clean = TRUE)
```

Arguments

`project`	Character string specifying the project name.
`log_clean`	Logical indicating whether to delete log files. Default is TRUE.

Details

This function performs three operations:

1. Resets all tasks to idle status (NULL) using `task_reset`
2. Stops the project using `project_stop`
3. Optionally deletes all log files from resource log folders

Use this when you want to:

- Restart failed tasks
- Re-run all tasks after fixing code
- Clean up before redeploying workers

Warning: Setting `log_clean = TRUE` permanently deletes all log files, which may contain useful debugging information.

Value

Invisibly returns NULL. Called for side effects (resetting tasks and logs).

See Also

`task_reset`, `project_stop`, `project_start`

Examples

```
## Not run:
# Not run:
# Reset project and clean logs
project_reset("simulation_study")

# Reset but keep logs for debugging
project_reset("simulation_study", log_clean = FALSE)

# Restart after reset
project_start("simulation_study")
worker_slurm("simulation_study", "hpc", fun = my_function)

## End(Not run)
```

---

project_resource_add *Assign a Resource to a Project*

Description

Associates a computing resource with a project and configures resource-specific settings like working directory, runtime limits, and worker count.

Usage

```
project_resource_add(
  project,
  resource,
  working_dir,
  account = NULL,
  hours = 1,
  workers = NULL
)
```

Arguments

project	Character string specifying the project name.
resource	Character string specifying the resource name.
working_dir	Absolute path to the working directory on the resource where workers will execute.
account	Optional character string for the account/allocation to use on the resource (relevant for SLURM clusters with accounting). Default is NULL.
hours	Maximum runtime in hours for each worker job. Default is 1 hour.
workers	Maximum number of concurrent workers for this project on this resource. If NULL, uses the resource's maximum worker count.

Details

This function creates or updates the association between a project and resource. Each project can be associated with multiple resources, and settings are resource-specific.

If the project-resource association already exists, only the specified parameters are updated.

The `working_dir` should exist on the resource and contain any necessary input files or scripts.

The `hours` parameter sets the SLURM walltime for worker jobs. Workers will automatically terminate before this limit to avoid being killed mid-task.

Value

Invisibly returns NULL. Called for side effects (adding/updating project-resource association).

See Also

`project_add`, `resource_add`, `worker_slurm`, `project_resource_get`

Examples

```
## Not run:
# Not run:
# Assign resource to project with basic settings
project_resource_add(
  project = "simulation_study",
  resource = "hpc",
  working_dir = "/home/user/simulations"
)

# Assign with specific account and time limit
project_resource_add(
  project = "big_analysis",
  resource = "hpc",
  working_dir = "/scratch/project/data",
  account = "research_group",
  hours = 48,
  workers = 100
)

## End(Not run)
```

`project_resource_add_jobs` *Manage SLURM Job List for Project Resource*

Description

Adds a SLURM job name to the list of active jobs for a project-resource association, or resets the job list.

Usage

```
project_resource_add_jobs(project, resource, job, reset = FALSE)
```

Arguments

project	Character string specifying the project name.
resource	Character string specifying the resource name.
job	Character string with the SLURM job name to add. If missing, the job list is reset to empty.
reset	Logical indicating whether to clear the job list before adding. Default is FALSE. If TRUE, replaces all jobs with job.

Details

The job list is a semicolon-separated string of SLURM job names stored in the database. This list is used by project_stop to cancel all jobs when stopping a project.

Job names are automatically added by worker_slurm when submitting workers.

Currently only supports SLURM resources.

Value

Invisibly returns NULL. Called for side effects (updating job list).

See Also

worker_slurm, project_stop

Examples

```
## Not run:
# Not run:
# Add a job (typically done automatically by worker_slurm)
project_resource_add_jobs("simulation_study", "hpc", "job_12345")

# Reset job list
project_resource_add_jobs("simulation_study", "hpc", reset = TRUE)

## End(Not run)
```

project_resource_get *Get resources of a project*

Description

Get resources of a project

Usage

```
project_resource_get(project, resource = NULL, con = NULL)
```

Arguments

project	project name
resource	resource name
con	connection to database

Value

a table of resources used in the project

project_resource_log_delete *Delete Log Files for a Project Resource*

Description

Removes all log files from the resource's log folder for a specific project. Log files include SLURM output/error files and worker scripts.

Usage

```
project_resource_log_delete(project, resource, con = NULL)
```

Arguments

project	Character string specifying the project name.
resource	Character string specifying the resource name.
con	An optional database connection. If NULL, a new connection is created and closed automatically.

Details

Deletes all files matching the pattern `<project>-<resource>*` from the log folder specified in the resource configuration.

Currently only supports SLURM resources.

This function is automatically called by `project_reset` when `log_clean = TRUE`.

Value

Invisibly returns NULL. Called for side effects (deleting log files).

See Also

`project_reset`, `resource_add`

Examples

```
## Not run:
# Not run:
# Delete logs for specific project-resource
project_resource_log_delete("simulation_study", "hpc")

## End(Not run)
```

`project_start` *Start a Project*

Description

Activates a project to allow workers to begin consuming tasks. Workers will only process tasks from started projects.

Usage

```
project_start(project, con = NULL)
```

Arguments

`project`	Character string specifying the project name.
`con`	An optional database connection. If NULL, a new connection is created and closed automatically.

Details

Starting a project sets its `status` field to TRUE in the database. Workers check this status before requesting new tasks. If a project is stopped (status = FALSE), workers will terminate instead of processing tasks.

You must start a project before deploying workers with `worker` or `worker_slurm`.

Value

Invisibly returns NULL. Called for side effects (updating project status).

See Also

project_stop, project_add, worker, worker_slurm

Examples

```
## Not run:
# Not run:
# Start project to enable workers
project_start("simulation_study")

# Deploy workers after starting
worker_slurm("simulation_study", "hpc", fun = my_function)

## End(Not run)
```

---

project_status *Display Project Status*

Description

Prints a summary of project status including whether it's running and the current status of all tasks.

Usage

```
project_status(project, con = NULL)
```

Arguments

project	Character string specifying the project name.
con	An optional database connection. If NULL, a new connection is created and closed automatically.

Details

Displays:

- Whether the project is running or stopped
- Task status summary from task_status

Use this function to monitor progress and identify failed tasks.

Value

Invisibly returns NULL. Called for side effects (printing status).

See Also

task_status, project_get

Examples

```
## Not run:
# Not run:
# Check project status
project_status("simulation_study")

## End(Not run)
```

`project_stop` *Stop a Project*

Description

Deactivates a project and cancels all running SLURM jobs associated with it. Workers will terminate after completing their current task.

Usage

```
project_stop(project)
```

Arguments

`project`	Character string specifying the project name.

Details

This function:

- Sets the project status to FALSE, preventing workers from taking new tasks
- Cancels all SLURM jobs associated with this project using `scancel`
- Resets the job list for all project resources

Active workers will complete their current task before shutting down. Tasks in `working` status when the project stops should be reset to `idle` using project_reset or task_reset.

Value

Invisibly returns NULL. Called for side effects (stopping project and jobs).

See Also

project_start, project_reset, task_reset

Examples

```
## Not run:
# Not run:
# Stop project and cancel all jobs
project_stop("simulation_study")

# Reset tasks that were in progress
task_reset("simulation_study", status = "working")

## End(Not run)
```

`resource_add` *Add a New Computing Resource*

Description

Registers a new computing resource (HPC cluster or computer) in the database for use with taskqueue projects.

Usage

```
resource_add(
  name,
  type = c("slurm", "computer"),
  host,
  workers,
  log_folder,
  username = NULL,
  nodename = strsplit(host, "\\.")[[1]][1],
  con = NULL
)
```

Arguments

`name`	Character string for the resource name. Must be unique.
`type`	Type of resource. Currently supported: `"slurm"` for SLURM clusters or `"computer"` for standalone machines.
`host`	Hostname or IP address of the resource. For SLURM clusters, this should be the login/head node.
`workers`	Maximum number of concurrent workers/cores available on this resource (integer).
`log_folder`	Absolute path to the directory where log files will be stored. Must be an absolute path (Linux or Windows format). Directory will contain subdirectories for each project.
`username`	Username for SSH connection to the resource. If NULL (default), uses the current user from `Sys.info()["user"]`.

nodename	Node name as obtained by `Sys.info()["nodename"]` on the resource. Default extracts the hostname from `host`.
con	An optional database connection. If NULL, a new connection is created and closed automatically.

Details

The `log_folder` is critical for troubleshooting. It stores:

- SLURM job output and error files
- Task execution logs
- R worker scripts

Choose a high-speed storage location if possible due to frequent I/O operations.

If a resource with the same `name` already exists, this function will fail due to uniqueness constraints.

Value

Invisibly returns NULL. Called for side effects (adding resource to database).

See Also

`resource_get`, `resource_list`, `project_resource_add`

Examples

```
## Not run:
# Not run:
# Add a SLURM cluster resource
resource_add(
  name = "hpc",
  type = "slurm",
  host = "hpc.university.edu",
  workers = 500,
  log_folder = "/home/user/taskqueue_logs/"
)

# Add with explicit username
resource_add(
  name = "hpc2",
  type = "slurm",
  host = "cluster.lab.org",
  workers = 200,
  log_folder = "/scratch/taskqueue/logs/",
  username = "johndoe"
)

# Verify resource was added
resource_list()

## End(Not run)
```

resource_get *Get Information for a Specific Resource*

Description

Retrieves detailed information about a single computing resource by name.

Usage

```
resource_get(resource, con = NULL)
```

Arguments

resource	Character string specifying the resource name.
con	An optional database connection. If NULL, a new connection is created and closed automatically.

Details

The returned data frame contains all resource configuration details needed for worker deployment, including connection information and resource limits.

Value

A single-row data frame containing resource information. Stops with an error if the resource is not found.

See Also

resource_add, resource_list

Examples

```
## Not run:
# Not run:
# Get specific resource
hpc_info <- resource_get("hpc")
print(hpc_info$workers)  # Maximum workers
print(hpc_info$log_folder)  # Log directory

## End(Not run)
```

`resource_list` *List All Computing Resources*

Description

Retrieves all computing resources registered in the database.

Usage

```
resource_list()
```

Value

A data frame containing information about all resources, with columns:

`id`	Unique resource identifier
`name`	Resource name
`type`	Resource type (e.g., "slurm", "computer")
`host`	Hostname or IP address
`username`	Username for SSH connection
`nodename`	Node name as reported by Sys.info()
`workers`	Maximum number of concurrent workers
`log_folder`	Absolute path to log file directory

See Also

`resource_add`, `resource_get`

Examples

```
## Not run:
# Not run:
# List all resources
resources <- resource_list()
print(resources)

# Find SLURM resources
slurm_resources <- resources[resources$type == "slurm", ]

## End(Not run)
```

shiny_app *Launch Shiny App for Monitoring Projects*

Description

Starts an interactive Shiny application to monitor task progress and runtime statistics for taskqueue projects.

Usage

```
shiny_app()
```

Details

The Shiny app provides:

- Project selector dropdown
- Real-time task status table (updates every 5 seconds)
- Runtime distribution histogram for completed tasks

Useful for monitoring long-running projects and identifying performance issues.

Value

Does not return while the app is running. Stops when the app is closed.

See Also

project_status, task_status

Examples

```
## Not run:
# Not run:
# Launch monitoring app
shiny_app()

## End(Not run)
```

table_exist *Check whether a table is existed*

Description

Check whether a table is existed

Usage

```
table_exist(table, con = NULL)
```

Arguments

table	table name
con	a connection

Value

logical value

taskqueue_options *Set or Get taskqueue Options*

Description

Configure or retrieve database connection parameters for taskqueue. Options are typically set via environment variables in `.Renviron`, but can be overridden programmatically.

Usage

```
taskqueue_options(...)
```

Arguments

...	Option names to retrieve values (as strings), or key=value pairs to set options. All option names must be specified.

Details

By default, options are read from environment variables set in `~/.Renviron`. Use this function to override defaults temporarily or check current settings.

Changes are session-specific and don't modify environment variables.

Value

If no arguments: list of all option values. If argument names only: list of specified option values. If setting values: invisibly returns updated options.

Supported options

host PostgreSQL server hostname or IP address (from PGHOST)

port PostgreSQL server port, typically 5432 (from PGPORT)

user Database username (from PGUSER)

password Database password (from PGPASSWORD)

database Database name (from PGDATABASE)

See Also

taskqueue_reset, db_connect

Examples

```
# View all current options
taskqueue_options()

# Get specific option
taskqueue_options("host")

# Set options (temporary override)
taskqueue_options(host = "localhost", port = 5432)

# Reset to environment variable values
taskqueue_reset()
```

---

taskqueue_reset *Reset taskqueue Options to Defaults*

Description

Resets all taskqueue options to their default values from environment variables.

Usage

```
taskqueue_reset()
```

Details

This function restores options to the values specified in environment variables (PGHOST, PGPORT, PGUSER, PGPASSWORD, PGDATABASE). Any programmatic changes made via taskqueue_options are discarded.

Useful after temporarily modifying connection parameters.

Value

Invisibly returns NULL. Called for side effects (resetting options).

See Also

taskqueue_options

Examples

```
# Override options temporarily
taskqueue_options(host = "test.server.com")

# Reset to environment variable values
taskqueue_reset()
```

---

task_add *Add Tasks to a Project*

Description

Creates a specified number of tasks in a project's task table. Each task is assigned a unique ID and initially has idle (NULL) status.

Usage

```
task_add(project, num, clean = TRUE, con = NULL)
```

Arguments

project	Character string specifying the project name.
num	Integer specifying the number of tasks to create.
clean	Logical indicating whether to delete existing tasks before adding new ones. Default is TRUE.
con	An optional database connection. If NULL, a new connection is created and closed automatically.

Details

Tasks are created with sequential IDs from 1 to num. Each task initially has NULL status (idle) and will be assigned to workers after the project is started.

If clean = TRUE, all existing tasks are removed using task_clean before adding new tasks. If FALSE, new tasks are added but existing tasks remain (duplicates are ignored due to primary key constraints).

Your worker function will receive the task ID as its first argument.

Value

Invisibly returns NULL. Called for side effects (adding tasks to database).

See Also

task_clean, task_status, worker, project_start

Examples

```
## Not run:
# Not run:
# Add 100 tasks to a project
task_add("simulation_study", num = 100)

# Add tasks without cleaning existing ones
task_add("simulation_study", num = 50, clean = FALSE)

# Check task status
task_status("simulation_study")

## End(Not run)
```

---

task_clean *Remove All Tasks from a Project*

Description

Deletes all tasks from a project's task table. This is a destructive operation that removes all task data and history.

Usage

```
task_clean(project, con = NULL)
```

Arguments

project	Character string specifying the project name.
con	An optional database connection. If NULL, a new connection is created and closed automatically.

Details

Uses SQL TRUNCATE to efficiently remove all rows from the task table. This is faster than DELETE but cannot be rolled back.

Warning: All task history, including completion status and runtime information, will be permanently lost.

This function is automatically called by task_add when clean = TRUE.

Value

Invisibly returns NULL. Called for side effects (truncating task table).

See Also

task_add, task_reset

Examples

```
## Not run:
# Not run:
# Remove all tasks
task_clean("simulation_study")

# Add new tasks
task_add("simulation_study", num = 200)

## End(Not run)
```

---

task_get *Get Detailed Task Information*

Description

Retrieves detailed information about tasks with specified statuses, including execution times and error messages.

Usage

```
task_get(project, status = c("failed"), limit = 10, con = NULL)
```

Arguments

project	Character string specifying the project name.
status	Character vector of statuses to retrieve. Can include "working", "failed", "finished", or "all". Default is "failed".
limit	Maximum number of tasks to return (integer). Default is 10.
con	An optional database connection. If NULL, a new connection is created and closed automatically.

Details

Useful for:

- Debugging failed tasks (examine error messages)
- Analyzing runtime patterns
- Identifying slow tasks

The runtime column is calculated as the difference between finish and start times in seconds.

Specifying status = "all" returns tasks of any status.

Value

A data frame with detailed task information:

`id`	Task ID
`status`	Current status
`start`	Start timestamp
`finish`	Finish timestamp
`message`	Error message (for failed tasks) or NULL
`runtime`	Calculated runtime in seconds

See Also

`task_status`, `task_reset`

Examples

```
## Not run:
# Not run:
# Get first 10 failed tasks
failed <- task_get("simulation_study", status = "failed")
print(failed$message)  # View error messages

# Get all finished tasks
finished <- task_get("simulation_study", status = "finished", limit = 1000)
hist(finished$runtime, main = "Task Runtime Distribution")

# Get tasks of any status
all_tasks <- task_get("simulation_study", status = "all", limit = 50)

## End(Not run)
```

---

task_reset *Reset Task Status to Idle*

Description

Resets tasks with specified statuses back to idle (NULL) state, clearing their execution history. This allows them to be picked up by workers again.

Usage

```
task_reset(project, status = c("working", "failed"), con = NULL)
```

Arguments

`project`	Character string specifying the project name.
`status`	Character vector of statuses to reset. Can include "working", "failed", "finished", or "all". Default is c("working", "failed").
`con`	An optional database connection. If NULL, a new connection is created and closed automatically.

Details

Resetting tasks clears:

- Status (set to NULL/idle)
- Start time
- Finish time
- Error messages

Common use cases:

- Reset failed tasks after fixing code: `status = "failed"`
- Reset interrupted tasks: `status = "working"`
- Re-run everything: `status = "all"`

Specifying `status = "all"` resets all tasks regardless of current status.

Value

Invisibly returns NULL. Called for side effects (resetting task status).

See Also

`task_status`, `task_add`, `project_reset`

Examples

```
## Not run:
# Not run:
# Reset only failed tasks
task_reset("simulation_study", status = "failed")

# Reset working tasks (e.g., after project_stop)
task_reset("simulation_study", status = "working")

# Reset all tasks to start over
task_reset("simulation_study", status = "all")

# Check status after reset
task_status("simulation_study")

## End(Not run)
```

`task_status` *Get Task Status Summary*

Description

Returns a summary table showing the number and proportion of tasks in each status for a project.

Usage

```
task_status(project, con = NULL)
```

Arguments

`project`	Character string specifying the project name.
`con`	An optional database connection. If NULL, a new connection is created and closed automatically.

Details

Task statuses:

- **idle** (NULL in database): Task not yet started
- **working**: Task currently being processed by a worker
- **finished**: Task completed successfully
- **failed**: Task encountered an error

Use this function to monitor progress and identify problems.

Value

A data frame with one row per status, containing:

`status`	Task status: "idle", "working", "finished", or "failed"
`count`	Number of tasks with this status (integer)
`ratio`	Proportion of tasks with this status (numeric)

See Also

`task_get`, `task_reset`, `project_status`

Examples

```
## Not run:
# Not run:
# Check task status
status <- task_status("simulation_study")
print(status)

# Calculate completion percentage
finished <- status$count[status$status == "finished"]
total <- sum(status$count)
pct_complete <- 100 * finished / total

## End(Not run)
```

`tq_apply` *Apply a Function with Task Queue (Simplified Workflow)*

Description

A high-level interface for running embarrassingly parallel tasks on HPC clusters. Combines project creation, task addition, and worker scheduling into a single function call, similar to `lapply`.

Usage

```
tq_apply(
  n,
  fun,
  project,
  resource,
  memory = 10,
  hour = 24,
  account = NULL,
  working_dir = getwd(),
  ...
)
```

Arguments

`n`	Integer specifying the number of tasks to run. Your function will be called with arguments 1, 2, ..., n.
`fun`	Function to execute for each task. Must accept the task ID as its first argument. Should save results to disk.
`project`	Character string for project name. Will be created if it doesn't exist, updated if it does.
`resource`	Character string for resource name. Must already exist (created via `resource_add`).
`memory`	Memory requirement in GB for each task. Default is 10 GB.

hour	Maximum runtime in hours for worker jobs. Default is 24 hours.
account	Optional character string for SLURM account/allocation. Default is NULL.
working_dir	Working directory on the cluster where tasks execute. Default is current directory (getwd()).
...	Additional arguments passed to fun for every task.

Details

This function automates the standard taskqueue workflow:

1. Creates or updates the project with specified memory
2. Assigns the resource to the project
3. Adds n tasks (cleaning any existing tasks)
4. Resets all tasks to idle status
5. Schedules workers on the SLURM cluster

Equivalent to manually calling:

```
project_add(project, memory = memory)
project_resource_add(project, resource, working_dir, account, hour, n)
task_add(project, n, clean = TRUE)
project_reset(project)
worker_slurm(project, resource, fun = fun, ...)
```

Before using tq_apply:

- Initialize database: db_init()
- Create resource: resource_add(...)
- Configure .Renviron with database credentials

Your worker function should:

- Take task ID as first argument
- Save results to files (not return values)
- Be idempotent (check if output exists)

Value

Invisibly returns NULL. Called for side effects (scheduling workers).

See Also

worker, worker_slurm, project_add, task_add, resource_add

Examples

```
## Not run:
# Not run:
# Simple example
my_simulation <- function(i, param) {
  out_file <- sprintf("results/sim_%04d.Rds", i)
  if (file.exists(out_file)) return()
  result <- run_simulation(i, param)
  saveRDS(result, out_file)
}

# Run 100 simulations on HPC
tq_apply(
  n = 100,
  fun = my_simulation,
  project = "my_study",
  resource = "hpc",
  memory = 16,
  hour = 6,
  param = 5
)

# Monitor progress
project_status("my_study")
task_status("my_study")

## End(Not run)
```

`worker` *Execute Tasks as a Worker*

Description

Runs as a worker process that continuously fetches and executes tasks from a project until no tasks remain or the project is stopped.

Usage

```
worker(project, fun, ...)
```

Arguments

`project`	Character string specifying the project name.
`fun`	Function to execute for each task. Must accept the task ID as its first argument. The function should save its results to disk and is not expected to return a value.
`...`	Additional arguments passed to `fun` for every task.

Details

This function implements the worker loop:

1. Request a task from the database (atomically)
2. Update task status to "working"
3. Execute `fun(task_id, ...)`
4. Update task status to "finished" (or "failed" if error)
5. Repeat until no more tasks or stopping condition

Workers automatically:

- Add random delays to reduce database contention
- Track runtime to respect SLURM walltime limits
- Reconnect to database on connection failures
- Log progress and errors to console

Your worker function should:

- Check if output already exists (idempotent)
- Save results to disk (not return them)
- Handle errors gracefully or let them propagate

For SLURM resources, set the `TASKQUEUE_RESOURCE` environment variable to enable automatic walltime management.

Value

Does not return normally. Stops when: no more tasks are available, the project is stopped, or runtime limit is reached (SLURM only).

See Also

`worker_slurm`, `task_add`, `project_start`, `tq_apply`

Examples

```
## Not run:
# Not run:
# Define worker function
my_task <- function(task_id, param1, param2) {
  out_file <- sprintf("results/task_%04d.Rds", task_id)
  if (file.exists(out_file)) return()  # Skip if done

  result <- expensive_computation(task_id, param1, param2)
  saveRDS(result, out_file)
}

# Run worker locally (for testing)
worker("test_project", my_task, param1 = 10, param2 = "value")

## End(Not run)
```

worker_slurm *Create a worker on slurm cluster*

Description

Create a worker on slurm cluster

Usage

```
worker_slurm(
  project,
  resource,
  fun,
  rfile,
  module_r = "R/4.3.1",
  module_pg = "postgresql/16.0",
  modules = NULL,
  pkgs = rev(.packages()),
  submit = TRUE,
  ...
)
```

Arguments

project	Project name.
resource	Resource name.
fun	Function running on workers. See details.
rfile	R script file path. See details.
module_r	Module name for R.
module_pg	Module name for postgresql. See details.
modules	extra modules to load in slurm. See details.
pkgs	A character vector containing the names of packages that must be loaded on worker including all packages in default when worker_slurm is called.
submit	Whether to submit to slurm cluster (TRUE in default). See details.
...	Extra arguments for fun.

Details

There are two ways to pass R scripts into workers (i.e. fun or file). * fun is used for general and simple case which takes the task id as the first argument. A new r script is created in the log folder and running in the workers. The required packages are passed using pkgs. Extra arguments are specified through .... taskqueue_options() is passed into workers. * rfile is used more complicated case. Function worker has to be called at the end of file. No taskqueue_options() is passed into workers. * fun is higher priority with file. A submit file is created in the log folder for each project/resource with random file name. Then system command ssh is used to connect remote slurm host if submit = TRUE.

Value

no return

Examples

```
## Not run:
# Not run:
fun_test <- function(i, prefix) {
    Sys.sleep(runif(1) * 2)
}
worker_slurm("test_project", "slurm", fun = fun_test)
worker_slurm("test_project", "slurm", fun = fun_test, prefix = "a")
worker_slurm("test_project", "slurm", rfile = "rfile.R")
worker_slurm("test_project", "slurm", fun = fun_test, submit = FALSE)

## End(Not run)
```

Index