flowr

Streamlining Computing Workflows

Sahil Seth

Flowr documentation, version (0.9.7.9031)

Contents

Get started

```
library(flowr)
```

```
setup()
```

This will copy the flowr helper script to `~/bin`. Please make sure that this folder is in your `$PATH` variable. For more details refer to setup's help section.

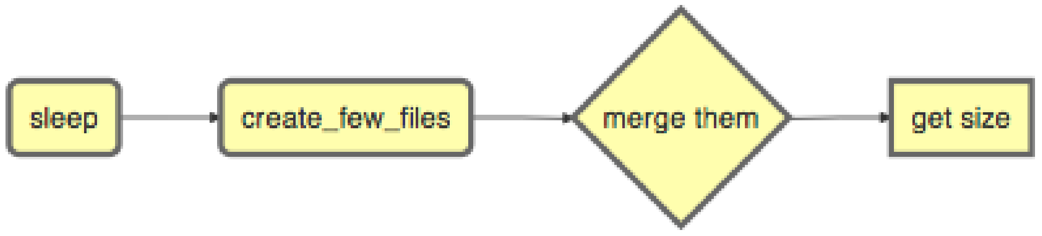
Running flowr from the terminal will fetch you the following:

```
Usage: flowr function [arguments]

status          Detailed status of a flow(s).
rerun           rerun a previously failed flow
kill            Kill the flow, upon providing working directory
fetch_pipes     Checking what modules and pipelines are available; flowr fetch_pipes

Please use 'flowr -h function' to obtain further information about the usage of a specific function.
```

Toy example



Consider, a simple example where we have **three** instances of linux's `sleep` command. After its completion **three** tmp files are created with some random data. Then, a merging step follows, combining the tmp files into one big file. Next, we use `du` to calculate the size of the merged file.

NGS context This is quite similar in structure to a typical workflow from where a series of alignment and sorting steps may take place on the raw fastq files. Followed by merging of the resulting bam files into one large file per-sample and further downstream processing.

To create this flow in flowr, we need the actual commands to run; and a set of instructions regarding how to stich the individual steps into a coherent pipeline.

Here is a table with the commands we would like to run (or `flow mat`).

samplename	jobname	cmd
sample1	sleep	sleep 10 && sleep 2;echo hello
sample1	sleep	sleep 11 && sleep 8;echo hello
sample1	sleep	sleep 11 && sleep 17;echo hello
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_1

samplename	jobname	cmd
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_2
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_3
sample1	merge	cat sample1_tmp_1 sample1_tmp_2 sample1_tmp_3 > sample1_merged
sample1	size	du -sh sample1_merged; echo MY shell: $SHELL

Further, we use an additional file specifying the relationship between the steps, and also other resource requirements: flow_def.

jobname	sub_type	prev_jobs	dep_type	queue	memory_reserved	walltime	cpu_reserved	platform	jo
sleep	scatter	none	none	short	2000	1:00	1	torque	
create_tmp	scatter	sleep	serial	short	2000	1:00	1	torque	
merge	serial	create_tmp	gather	short	2000	1:00	1	torque	
size	serial	merge	serial	short	2000	1:00	1	torque	

Note: Each row in a flow mat relates to one job. Jobname column is used to link flow definition with flow mat. Also, values in previous jobs (prev_jobs) are derived from jobnames.

Stitch it

We use the two files described above and stich them to create a `flow object` (which contains all the information we need for cluster submission).

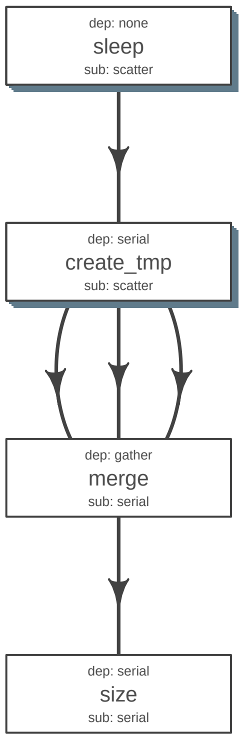
```
fobj <- to_flow(x = flow_mat,
                                def = flow_def,
                                flowname = "example1", ## give it a name
                                platform = "lsf")      ## override platform mentioned in flow def
```

Refer to to_flow's help section for more details.

Plot it

We can use `plot_flow` to quickly visualize the flow; this really helps when developing complex workflows.

```
plot_flow(fobj)     # ?plot_flow for more information
plot_flow(flow_def) # plot_flow works on flow definition as well
```



Refer to plot_flow's help section for more details.

Dry Run

Dry run: Quickly perform a dry run, of the submission step. This creates all the folder and files, and skips submission to the cluster. This helps in debugging etc.

```
submit_flow(fobj)
```

```
Test Successful!
You may check this folder for consistency. Also you may re-run submit with execute=TRUE
 ~/flowr/sleep_pipe-20150520-15-18-27-5mSd32G0
```

Submit it

Submit to the cluster !

```
submit_flow(fobj, execute = TRUE)
```

```
Flow has been submitted. Track it from terminal using:
flowr status x=~/flowr/type1-20150520-15-18-46-sySOzZnE
```

Refer to submit_flow's help section for more details.

Check its status

One may periodically run `status` to monitor the status of a flow.

```
flowr status x=~/flowr/runs/sleep_pipe-20150520*

|          | total| started| completed| exit_status|    status|
|:---------|-----:|-------:|---------:|-----------:|---------:|
|001.sleep |    10|      10|        10|           0| completed|
|002.tmp   |    10|      10|        10|           0| completed|
|003.merge |     1|       1|         1|           0| completed|
|004.size  |     1|       1|         1|           0| completed|
```

Alternatively, to check a summarized status of several flows, use the parent folder, for example:

```
flowr status x=~/flowr/runs

Showing status of: ~/flowr/runs
|          | total| started| completed| exit_status|    status|
|:---------|-----:|-------:|---------:|-----------:|---------:|
|001.sleep |    30|      30|        10|           0|processing|
|002.tmp   |    30|      30|        10|           0|processing|
|003.merge |     3|       3|         1|           0|   pending|
|004.size  |     3|       3|         1|           0|   pending|
```

Scalability: Quickly submit, and check a summarized OR detailed status on ten or hundreds of flows.

Refer to status's help section for more details.

Kill it

Incase something goes wrong, one may use to kill command to terminate all the relating jobs.

kill one flow:

```
flowr kill_flow x=flow_wd
```

One may instruct flowr to kill multiple flows, but flowr would confirm before killing.

```
flowr kill x='~/flowr/runs/sleep_pipe'
found multiple wds:
  ~/flowr/runs/sleep_pipe-20150825-16-24-04-0Lv1PbpI
  ~/flowr/runs/sleep_pipe-20150825-17-47-52-5vFIkrMD
Really kill all of them ? kill again with force=TRUE
```

To kill multiple flow, set force=TRUE:

```
kill(x='~/flowr/runs/sleep_pipe*', force = TRUE)
```

Refer to kill's help section for more details.

Re-run a flow

flowr also enables you to re-run a pipeline in case of hardware or software failures.

- **hardware failure**: no change to the pipeline is required, simply rerun it: `rerun(x=flow_wd, start_from=<intermediate step>)`
- **software failure**: either a change to flowmat or flowdef has been made: `rerun(x=flow_wd, mat = new_flowmat, def = new_flowdef, start_from=<intermediate step>)`

Refer to rerun's help section for more details.

Ingredients for building a pipeline

An easy and quick way to build a workflow is to create a set of two tab delimited files. First is a table with commands to run (for each step of the pipeline), while second has details regarding how the modules are stitched together. In the rest of this document we would refer to them as `flow_mat` and `flow_def` respectively (as introduced in the previous sections).

We could read in, examples of both these files to understand their structure.

```
ex = file.path(system.file(package = "flowr"), "pipelines")
flow_mat = as.flowmat(file.path(ex, "sleep_pipe.tsv"))
flow_def = as.flowdef(file.path(ex, "sleep_pipe.def"))
```

1. Flow matrix

describes commands to run:

Each row in flow mat describes one shell command, with additional information regarding the name of the step etc.

Essentially, this is a tab delimited file with three columns:

- `samplename`: A grouping column. The table is split using this column and each subset is treated as an individual flow. Thus we may have one flowmat for a series of samples, and the whole set would be submitted as a batch.
  - If all the commands are for a single sample, one can just repeat a dummy name like sample1 all throughout.
- `jobname`: This corresponds to the name of the step. This should match exactly with the jobname column in flow_def table described below.
- `cmd`: A shell command to run. One can get quite creative here. These could be multiple shell commands separated by a `;` or `&&`, more on this here. Though to keep this clean you may just wrap a multi-line command into a script and just source the bash script from here.

Here is an example flow_mat for the flowr described above.

samplename	jobname	cmd
sample1	sleep	sleep 10 && sleep 2;echo hello
sample1	sleep	sleep 11 && sleep 8;echo hello
sample1	sleep	sleep 11 && sleep 17;echo hello

samplename	jobname	cmd
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_1
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_2
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_3
sample1	merge	cat sample1_tmp_1 sample1_tmp_2 sample1_tmp_3 > sample1_merged
sample1	size	du -sh sample1_merged; echo MY shell: $SHELL

2. Flow definition

defines how to stich pieces of the (work)flow:

Each row in this table refers to one step of the pipeline. It describes the resources used by the step and also its relationship with other steps, especially, the step immediately prior to it.

It is a tab separated file, with a minimum of 4 columns:

- `jobname`: Name of the step
- `sub_type`: Short for submission type, refers to, how should multiple commands of this step be submitted. Possible values are `serial` or `scatter`.
- `prev_job`: Short for previous job, this would be jobname of the previous job. This can be NA/./none if this is a independent/initial step, and no previous step is required for this to start.
- `dep_type`: Short for dependency type, refers to the relationship of this job with the one defined in `prev_job`. This can take values `none`, `gather`, `serial` or `burst`.

These would be explained in detail, below.

Apart from the above described variables, several others defining the resource requirements of each step are also available. These give great amount of flexibility to the user in choosing CPU, wall time, memory and queue for each step (and are passed along to the HPCC platform).

- `cpu_reserved`
- `memory_reserved`
- `nodes`
- `walltime`
- `queue`

This is especially useful for genomics pipelines, since each step may use different amount of resources. For example, in other frameworks, if one step uses 16 cores these would be blocked and not used during processing of several other steps. Thus resulting in blockage of those cores. Flowr prevents this, by being able to tune resources granurly. Example, one may submit few short steps in `short` queue, and longer steps of the same pipeline in say `long` queue.

Most cluster platforms accept these resource arguments. Essentially a file like this is used as a template, and variables defined in curly braces (ex. `{{{CPU}}}`) are filled up using the flow definition file.

If these (resource requirements) columns are not included in the flow definition, their values should be explicitly defined in the submission template. One may customize the templates as described in the cluster support section.

Here is an example of a typical flow_def file.

jobname	sub_type	prev_jobs	dep_type	queue	memory_reserved	walltime	cpu_reserved	platform	jo
sleep	scatter	none	none	short	2000	1:00	1	torque	

jobname	sub_type	prev_jobs	dep_type	queue	memory_reserved	walltime	cpu_reserved	platform	jo
create_tmp	scatter	sleep	serial	short	2000	1:00	1	torque	
merge	serial	create_tmp	gather	short	2000	1:00	1	torque	
size	serial	merge	serial	short	2000	1:00	1	torque	

Example:

Let us use an example flow, to understand submission and dependency types.

Consider three steps A, B and C, where A has 10 commands from A1 to A10, similarly B has 10 commands B1 through B10 and C has a single command, C1. Consider another step D (with D1-D3), which comes after C.

```
step:       A   ----> B  -----> C -----> D
# of cmds  10        10         1        3
```

Submission types

This refers to the sub_type column in flow definition.

- `scatter`: submit all commands as parallel, independent jobs.
  - *Submit A1 through A10 as independent jobs*
- `serial`: run these commands sequentially one after the other.
  - *Wrap A1 through A10, into a single job.*

Dependency types

This refers to the dep_type column in flow definition.

- `none`: independent job.
  - *Initial step A has no dependency*
- `serial`: *one to one* relationship with previous job.
  - *B1 can start as soon as A1 completes.*
- `gather`: *many to one*, wait for **all** commands in previous job to finish then start the current step.
  - *All jobs of B (1-10), need to complete before C1 is started*
- `burst`: *one to many* wait for the previous step which has one job and start processing all cmds in the current step.
  - *D1 to D3 are started as soon as C1 finishes.*

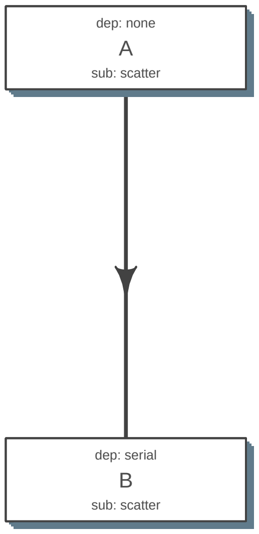
Relationships

Using the above submission and dependency types one can create several types of relationships between former and later jobs. Here are a few examples of relationships one may typically use.

One to One (serial)

```
                A1 --------> B1
                A2 --------> B1
                .. --------> ..
               A10 --------> B10
 dependency submission  dependency submission
    none     scatter      serial     scatter
                 relationship
                  ONE-to-ONE
```

Relationship between steps A and B is best defined as `serial`. Step A (A1 through A10) is submitted as scatter. Further, i^th jobs of B depends on i^th jobs of A. i.e. B1 requires A1 to complete; B2 requires A2 and so on. Also, we note that defining dependency as serial, makes sure that B does not wait for all elements of A to complete.



Many to One (gather)

```
                B1 ----\
                B2 -----\
                ..        -----> C1
                B9 ------/
                B10-----/
 dependency submission  dependency submission
    serial     scatter    gather     serial
                 relationship
                  MANY-to-ONE
```

Since C is a single command which requires all steps of B to complete, intuitively it needs to `gather` pieces of data generated by B. In this case `dep_type` would be `gather` and `sub_type` type would be `serial` since it is a single command.

One to Many (Burst)

```
                     /-----> D1
```

```
                C1 ---------> D2
                     \-----> D3
dependency submission  dependency submission
   gather   serial       burst     scatter
                relationship
                 ONE-to-MANY
```

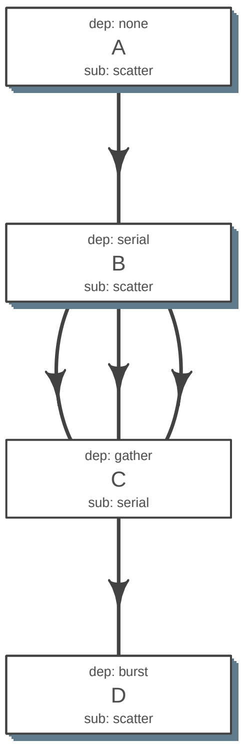
Further, D is a set of three commands (D1-D3), which need to wait for a single process (C1) to complete. They would be submitted as `scatter` after waiting on C in a `burst` type dependency.

In essence, an example flow_def would look like as follows (with additional resource requirements not shown for brevity):

```
ex2def = as.flowdef(file.path(ex, "abcd.def"))
ex2mat = as.flowmat(file.path(ex, "abcd.tsv"))
kable(ex2def[, 1:4])
```

jobname	sub_type	prev_jobs	dep_type
A	scatter	none	none
B	scatter	A	serial
C	serial	B	gather
D	scatter	C	burst

```
plot_flow(ex2def)
```



There is a darker more prominent shadow to indicate scatter steps.

Cluster Support

As of now we have tested this on the following clusters:

Platform	command	status	queue.type
LSF 7	bsub	Beta	lsf
LSF 9.1	bsub	Stable	lsf
Torque	qsub	Stable	torque
Moab	msub	Stable	moab
SGE	qsub	Beta	sge
SLURM	sbatch	alpha	slurm

For more details, refer to the configuration section

Installation

Requirements:

- R version > 3.1, preferred 3.2

```
## for a latest stable version (from DRAT):
install.packages("flowr", repos = "http://sahilseth.github.io/drat")
```

After installation run `setup()`, this will copy the flowr's helper script to `~/bin`. Please make sure that this folder is in your `$PATH` variable.

```
library(flowr)
setup()
```

Running `flowr` from the terminal should now show the following:

```
Usage: flowr function [arguments]

status          Detailed status of a flow(s).
rerun           rerun a previously failed flow
kill            Kill the flow, upon providing working directory
fetch_pipes     Checking what modules and pipelines are available; flowr fetch_pipes

Please use 'flowr -h function' to obtain further information about the usage of a specific function.
```

From this step on, one has the option of typing commands in a R console OR a bash shell (command line). For brevity we will show examples using the shell.

Test

Test a small pipeline on the cluster

This will run a three step pipeline, testing several different relationships between jobs. Initially, we can test this locally, and later on a specific HPCC platform.

```
## This may take about a minute or so.
flowr run x=sleep_pipe platform=local execute=TRUE
## corresponding R command:
run(x='sleep_pipe', platform='local', execute=TRUE)
```

If this completes successfully, we can try this on a computing cluster; where this would submit a few interconnected jobs.

Several platforms are supported out of the box (torque, moab, sge, slurm and lsf), you may use the platform variable to switch between platforms.

```
flowr run x=sleep_pipe platform=lsf execute=TRUE
## other options for platform: torque, moab, sge, slurm, lsf
## this shows the folder being used as a working directory for this flow.
```

Once the submission is complete, we can test the status using `status()` by supplying it the full path as recovered from the previous step.

```
flowr status x=~/flowr/runs/sleep_pipe-samp1-20150923-10-37-17-4WBiLgCm

## we expect to see a table like this when is completes successfully:

|               | total| started| completed| exit_status|status    |
|:--------------|-----:|-------:|---------:|-----------:|:---------|
|001.sleep      |     3|       3|         3|           0|completed |
|002.create_tmp |     3|       3|         3|           0|completed |
|003.merge      |     1|       1|         1|           0|completed |
|004.size       |     1|       1|         1|           0|completed |

## Also we expect a few files to be created:
ls ~/flowr/runs/sleep_pipe-samp1-20150923-10-37-17-4WBiLgCm/tmp
samp1_merged  samp1_tmp_1  samp1_tmp_2  samp1_tmp_3

## If both these checks are fine, we are all set !
```

There are a few places where things may go wrong, you may follow the advanced configuration guide for more details. Feel free to post questions on github issues page.

Advanced Configuration

HPCC Support Overview

Support for several popular cluster platforms is built-in. There is a template, for each platform, which should work out of the box. Further, one may copy and edit them (and save to `~/flowr/conf`) in case some changes are required. Templates from this folder (`~/flowr/conf`), would override defaults.

Here are links to latest templates on github:

- torque
- lsf
- moab

- sge
- slurm, needs testing

Not sure what platform you have?

You may check the version by running ONE of the following commands:

```
msub --version
## Version: **moab** client 8.1.1
man bsub
##Submits a job to **LSF** by running the specified
qsub --help
```

Here are some helpful guides and details on the platforms:

- PBS: wiki
- Torque: wiki
  - MD Anderson
  - University of Houston
- LSF wiki:
  - Harvard Medicla School uses: LSF HPC 7
  - Also used at Broad
- SGE wiki
  - A tutorial for Sun Grid Engine
  - Another from JHSPH
  - Dependecy info here

Comparison_of_cluster_software

FAQs and help on Solving Issues

Errors in job submission Possible issue: Jobs are not getting submitted

1. Check if the right platform was used for submission.
2. Confirm (with your system admin) that you have the privilege to submit jobs.
3. **Use a custom flowdef**: Many institutions have strict specification on the resource reservations. Make sure that the queue, memory, walltime, etc. requiremets are specified properly
4. **Use a custom submission template**: There are several parameters in the submission script used to submit jobs to the cluster. You may customize this template to suit your needs.

3. Use a custom flowdef

We can copy a example flow definition and customize it to suit our needs. This a tab delimited text file, so make sure that the format is correct after you make any changes.

```
cd ~/flowr/pipelines
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/pipelines/sleep_pipe.def
## check the format
flowr as.flowdef x=~/flowr/pipelines/sleep_pipe.def
```

Run the test with a custom flowdef:

```
flowr run x=sleep_pipe platform=lsf execute=TRUE def=~/flowr/pipelines/sleep_pipe.def
```

4. Use a custom submission template

If you need to customize the HPCC submission template, copy the file for your platform and make your desired changes. For example the MOAB based cluster in our institution does **not** accept the `queue` argument, so we need to comment it out.

Download the template for a specific HPCC platform into `~/flowr/conf`

```
cd ~/flowr/conf ## flowr automatically picks up a template from this folder.
## for MOAB (msub)
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/conf/moab.sh
## for Torque (qsub)
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/conf/torque.sh
## for IBM LSF (bsub)
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/conf/lsf.sh
## for SGE (qsub)
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/conf/sge.sh
## for SLURM (sbatch) [untested]
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/conf/slurm.sh
```

Make the desired changes using your favourite editor and submit again.

Possible issue: Jobs for subsequent steps are not submitting (though first step works fine).

1. **Confirm jobids are parsing fine**: Flowr parses the computing platform's output and extracts job IDs of submitted jobs.
2. **Check dependency string**:

1. Parsing job ids

Flowr parses job IDs to keep a log of all submitted jobs, and also to pass them along as a dependency to subsequent jobs. This is taken care by the parse_jobids() function. Each job scheduler shows the jobs id, when you submit a job, but it may show it in a slightly different fashion. To accommodate this one can use regular expressions as described in the relevant section of the flowr config.

For example LSF may show a string such as:

```
Job <335508> is submitted to queue <transfer>.
## test if it parses correctly
jobid="Job <335508> is submitted to queue <transfer>."
set_opts(flow_parse_lsf = ".*(\<[0-9]*\>).*")
parse_jobids(jobid, platform="lsf")
[1] "335508"
```

In this case *335508* was the job id and regex worked well !

Once we identify the correct regex for the platform you may update the configuration file with it.

```
cd ~/flowr/conf ## flowr automatically reads this file
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/conf/flowr.conf
```

Update the regex pattern and submit again.

2. Check dependency string

After collecting job ids from previous jobs, flowr renders them as a dependency for subsequent jobs. This is handled by render_dependency.PLATFORM functions.

Confirm that the dependency parameter is specified correctly in the submission scripts:

```
wd=~/flowr/runs/sleep_pipe-samp1-20150923-11-20-39-dfvhp5CK ## path to the most recent submission
cat $wd/002.create_tmp/create_tmp_cmd_1.sh
```

Flowr Configuration file Possible issue: Flowr shows too much OR too little information.

There are several verbose levels available (0, 1, 2, 3, ...)

One can change the verbose levels in this file (`~/flowr/conf/flowr.conf`) and check verbosity section in the help pages for more details.

Flowdef resource columns Possible issue: What all resources are supported in the flow definition?

The resource requirement columns of flow definition are passed along to the final (cluster) submission script. For example values in `cpu_reserved` column would be populated as `{{{CPU}}}` in the submission template.

The following table provides a mapping between the flow definition columns and variables in the submission templates:

flowdef variable	submission template variable
nodes	NODES
cpu_reserved	CPU
memory_reserved	MEMORY
email	EMAIL
walltime	WALLTIME
extra_opts	EXTRA_OPTS
	JOBNAME
	STDOUT
	CWD
	DEPENDENCY
	TRIGGER
	CMD

`* These are generated on the fly` and `** This is gathered from flow mat`

Adding a new platform Possible issue: Need to add a new platform

Adding a new platform involves a few steps, briefly we need to consider the following steps where changes would be necessary.

1. **job submission**: One needs to add a new template for the new platform. Several examples are available as described in the previous section.
2. **parsing job ids**: flowr keeps a log of all submitted jobs, and also to pass them along as a dependency to subsequent jobs. This is taken care by the parse_jobids() function. Each job scheduler shows the jobs id, when you submit a job, but each shows it in a slightly different pattern. To accommodate this one can use regular expressions as described in the relevant section of the flowr config.

3. **render dependency**: After collecting job ids from previous jobs, flowr renders them as a dependency for subsequent jobs. This is handled by render_dependency.PLATFORM functions.
4. **recognize new platform**: Flowr needs to be made aware of the new platform, for this we need to add a new class using the platform name. This is essentially a wrapper around the job class

Essentially this requires us to add a new line like: `setClass("torque", contains = "job")`.

5. **killing jobs**: Just like submission flowr needs to know what command to use to kill jobs. This is defined in detect_kill_cmd function.

There are several job scheduling systems available and we try to support the major players. Adding support is quite easy if we have access to them. Your favourite not in the list? re-open this issue, with details on the platform: adding platforms

Possible issue: For other issues upload the error shown in the out files to github.

```
cat $wd/002.create_tmp/create_tmp_cmd_1.out
```

Tutorial: building a pipeline

A pipeline consists of several pieces, most essential of which is a function which generates a flowmat. Additionally, we need a flow definition, which descibes flow of the pipeline. These three files are available under the pipelines folder on github.

```
## to follow this tutorial, you may download them:
url=https://raw.githubusercontent.com/sahilseth/flowr/master/inst/pipelines
cd ~/flowr/pipelines
wget $url/sleep_pipe.R             ## A R script, with sleep_pipe(), which creates a flowmat
wget $url/sleep_pipe.def           ## A tab-delimited flow definition file
wget $url/sleep_pipe.conf          ## An *optional* tab-delim conf file, defining default params
```

To run the aforementioned pipeline, we would follow through these steps:

```
## Single step submission:
fobj = run("sleep_pipe", execute = TRUE);

## Details of the above step:
setwd("~/flowr/pipelines")
## behind the scenes, run does the following:
## optionally, load default parameters
load_opts("sleep_pipe.conf")

## get sleep_pipe() function
source("sleep_pipe.R")

## create a flowmat
flowmat = sleep_pipe()

## read a flow definition.
flowdef = as.flowdef("sleep_pipe.def")

## create flow and submit to cluster
fobj = to_flow(flowmat, flowdef, execute = TRUE)
```

Creating Modules/Pipelines

module: A R function which creates a flow mat, is a module. Using **module + flowdef**, we can run a pipeline.

The `sleep_pipe` calls the three other functions (**modules**); fetches flowmat from each, then rbinds them, creating a larger flowmat. You may refer to the sleep_pipe.R file for the source.

```
#' @param x number of files to make
sleep_pipe <- function(x = 3, samplename = "samp1"){

    ## call the modules one by one...
    out_sleep = sleep(x, samplename)
    out_create_tmp = create_tmp(x, samplename)
    out_merge_size = merge_size(out_create_tmp$outfiles, samplename)

    ## row bind all the commands
    flowmat = rbind(out_sleep$flowmat,
        out_create_tmp$flowmat,
        out_merge_size$flowmat)

    return(list(flowmat = flowmat, outfiles = out_merge_size$outfiles))
}
```

```
## create a flow matrix
out = sleep_pipe(x = 3, "sample1")
flowmat = out$flowmat
```

samplename	jobname	cmd
sample1	sleep	sleep 1 && sleep 10;echo 'hello'
sample1	sleep	sleep 16 && sleep 4;echo 'hello'
sample1	sleep	sleep 15 && sleep 3;echo 'hello'
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_1
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_2
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_3
sample1	merge	cat sample1_tmp_1 sample1_tmp_2 sample1_tmp_3 > sample1_merged
sample1	size	du -sh sample1_merged; echo 'MY shell:' $SHELL

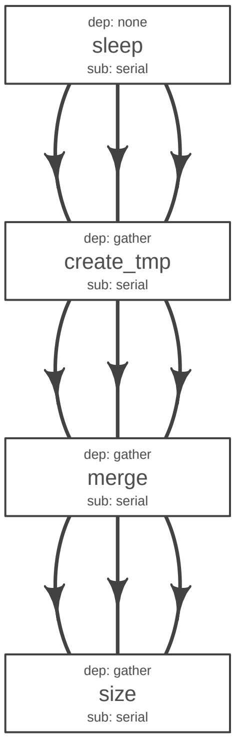
Next, we need a flow definition.

Creating Flow Definition

flowr enables us to quickly create a skeleton flow definition using a flowmat, which we can then alter to suit our needs. A handy function to_flowdef, accepts a `flowmat` and creates a flow definition.

The default skeleton takes a very conservative approach, creating all submissions as `serial` and all dependencies as `gather`. This ensures robustness, compromising efficiency.

```
def = to_flowdef(flowmat) ## create a skeleton flow definition
suppressMessages(plot_flow(def))
```



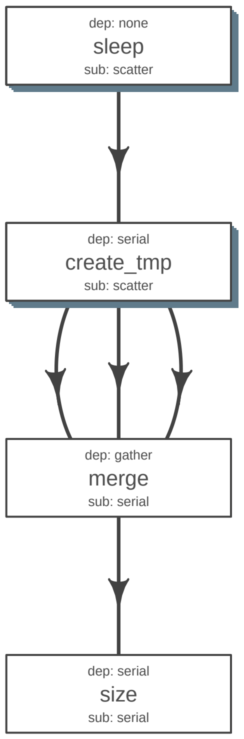
We can make the following changes to make this more efficient (run steps in parallel):

- multiple sleep commands would run as `scatter`/parallel (`none`)
- For each sleep, create_tmp creates a tmp file (`serial`)
- All tmp files are merged; when all are complete (`gather`)
- Then we get size on the resulting file (`serial`)

dependencies mentioned in ()

```
def$sub_type = c("scatter", "scatter", "serial", "serial")
def$dep_type = c("none", "serial", "gather", "serial")
kable(def)
```

jobname	sub_type	prev_jobs	dep_type	queue	memory_reserved	walltime	cpu_reserved	platform	jo
sleep	scatter	none	none	short	2000	1:00	1	torque	
create_tmp	scatter	sleep	serial	short	2000	1:00	1	torque	
merge	serial	create_tmp	gather	short	2000	1:00	1	torque	
size	serial	merge	serial	short	2000	1:00	1	torque	



Tip: Alternatively, one may write this to a file (`write_sheet(def, "sleep_pipe.def")`), make changes in a text editor and read it again (`as.flowdef("sleep_pipe.def")`.

Create flow, submit to cluster

Next, we create a flow object:

```
fobj = to_flow(flowmat, def, flowname = "sleep_pipe")
```

Finally, we can submit this to the cluster:

```
plot_flow(fobj)
submit_flow(fobj) ## dry run
fobj2 = submit_flow(fobj, execute = TRUE) ## submission to LSF cluster

## after submission, we can use the following:
status(fobj2) ## check status
rerun(fobj2)  ## re-run from a intermediate step
kill(fobj2)   ## kill it!
```

You may download a PDF version of this manual here

Streamlining Design and Deployment of Complex Workflows

Authors: Sahil Seth [aut, cre]

Version: 0.9.7.9031

License: MIT + file LICENSE

Description

This framework allows you to design and implement complex pipelines, and deploy them on your institution's computing cluster. This has been built keeping in mind the needs of bioinformatics workflows. However, it is easily extendable to any field where a series of steps (shell commands) are to be executed in a (work)flow.

Depends

R (>= 3.0.2), methods, params (>= 0.2.8), utils

Imports

diagram, whisker, tools

Suggests

reshape2, knitr, ggplot2, openxlsx, testthat

Enhances

(none)

Creating a Flow

to_flow

Create flow objects

Use a set of shell commands and flow definiton to create flow object. vector: a file with flowmat table a named list of commands for a sample. Its best to supply a flowmat instead.

Usage

Arguments

x

path (char. vector) to flow_mat, a data.frame or a list.

. . .

Supplied to specific functions like to_flow.data.frame

def

A flow definition table. Basically a table with resource requirements and mapping of the jobs in this flow

grp_col

column name used to split x (flow_mat). [samplename]

jobname_col

column name with job names. [jobname]

cmd_col

column name with commands. [cmd]

flowname

name of the flow [flowname]

flow_run_path

Path to a folder. Main operating folder for this flow. [get_opts("flow_run_path")][~/flowr/runs].

platform

character vector, specifying the platform to use. local, lsf, torque, moab, sge, slurm, … This over-rides the platform column in flowdef. (optional)

submit

Use submit_flow on flow object this function returns. TRUE/FALSE. [FALSE]

execute

Use submit_flow on flow object this function returns. TRUE/FALSE, an paramter to submit_flow(). [FALSE]

qobj

Depreciated, modify cluster templates instead. A object of class queue.

verbose

A numeric value indicating the amount of messages to produce. Values are integers varying from 0, 1, 2, 3, ... Please refer to the verbose page for more details. [get_opts("verbose")] [1]

desc

Advanced Use. final flow name, please dont change.

Details

The parameter x can be a path to a flow_mat, or a data.frame (as read by read_sheet). This is a minimum three column matrix with three columns: samplename, jobname and cmd

Value

Returns a flow object. If execute=TRUE, fobj is rich with information about where and how the flow was executed. It would include details like jobids, path to exact scripts run etc. To use kill_flow, to kill all the jobs one would need a rich flow object, with job ids present.

Behaviour: What goes in, and what to expect in return?

submit=FALSE & execute=FALSE: Create and return a flow object

submit=TRUE & execute=FALSE: dry-run, Create a flow object then, create a structured execution folder with all the commands

submit=TRUE, execute=TRUE: Do all of the above and then, submit to cluster

Examples

See also

to_flowmat, to_flowdef, to_flowdet, flowopts and submit_flow

to_flowmat

Create a flowmat using a list a commands.

Create a flowmat (data.frame) using a named list a commands. as.flowmat(): reads a file and checks for required columns. If x is data.frame checks for required columns.

Usage

Arguments

x

a named list, where name corresponds to the jobname and value is a vector of commands to run.

…

not used

samplename

character of length 1 or that of nrow(x) [samplename]

grp_col

column used for grouping, default samplename.

jobname_col

column specifying jobname, default jobname

cmd_col

column specifying commands to run, default cmd

Examples

to_flowdef

Flow Definition defines how to stich pieces of the (work)flow into a flow.

This function enables creation of a skeleton flow definition with several default values, using a flowmat. To customize the flowdef, one may supply parameters such as sub_type and dep_type upfront. As such, these params must be of the same length as number of unique jobs using in the flowmat. Each row in this table refers to one step of the pipeline. It describes the resources used by the step and also its relationship with other steps, especially, the step immediately prior to it. Submission types: This refers to the sub_type column in flow definition. Consider an example with three steps A, B and C. A has 10 commands from A1 to A10, similarly B has 10 commands B1 through B10 and C has a single command, C1. Consider another step D (with D1-D3), which comes after C. step (number of sub-processes) A (10) —-> B (10) —–> C (1) —–> D (3)

scatter: submit all commands as parallel, independent jobs. Submit A1 through A10 as independent jobs

serial: run these commands sequentially one after the other. - Wrap A1 through A10, into a single job.

Dependency types This refers to the dep_type column in flow definition.

none: independent job.

serial: one to one relationship with previous job.

B1 can start as soon as A1 completes, and B2 starts just after A2 and so on.

gather: many to one, wait for all commands in the previous job to finish then start the current step.

burst: one to many wait for the previous step which has one job and start processing all cmds in the current step. - D1 to D3 are started as soon as C1 finishes.

Usage

Arguments

x

can a path to a flowmat, flowmat or flow object.

...

not used

sub_type

submission type, one of: scatter, serial. Character, of length one or same as the number of jobnames

dep_type

dependency type, one of: gather, serial or burst. Character, of length one or same as the number of jobnames

prev_jobs

previous job name

queue

Cluster queue to be used

platform

platform of the cluster: lsf, sge, moab, torque, slurm etc.

memory_reserved

amount of memory required.

cpu_reserved

number of cpus required

walltime

amount of walltime required

verbose

A numeric value indicating the amount of messages to produce. Values are integers varying from 0, 1, 2, 3, ... Please refer to the verbose page for more details. [get_opts("verbose")] [1]

Format

This is a tab separated file, with a minimum of 4 columns:

required columns:

jobname: Name of the step

sub_type: Short for submission type, refers to, how should multiple commands of this step be submitted. Possible values are `serial` or `scatter`.

prev_jobs: Short for previous job, this would be the jobname of the previous job. This can be NA/./none if this is a independent/initial step, and no previous step is required for this to start. Additionally, one may use comma(s) to define multiple previous jobs (A,B).

dep_type: Short for dependency type, refers to the relationship of this job with the one defined in `prev_jobs`. This can take values `none`, `gather`, `serial` or `burst`.

resource columns (recommended):

Additionally, one may customize resource requirements used by each step. The format used varies and depends to the computing platform. Thus its best to refer to your institutions guide to specify these.

cpu_reserved integer, specifying number of cores to reserve [1]

memory_reserved Usually in KB [2000]

nodes number of server nodes to reserve, most tools can only use multiple cores on a single node [1]

walltime maximum time allowed for a step, usually in a HH:MM or HH:MM:SS format. [1:00]

queue the queue to use for job submission [short]

check

Check consistency of flowdef and flowmat

Check consistency of flowdef and flowmat, using various rules.

Usage

Arguments

x

a flowdef or flowmat object

. . .

Passed onto either check.flowdef OR check.flowmat functions

verbose

A numeric value indicating the amount of messages to produce. Values are integers varying from 0, 1, 2, 3, ... Please refer to the verbose page for more details. [get_opts("verbose")] [1]

Details

A typical output from flowdef with verbose level: 2

Submiting and managing a flow

submit_flow

Submit a flow to the cluster

Submit a flow to the cluster or perform a dry-run to check and debug issues.

Usage

Arguments

x

a object of class flow.

verbose

logical.

. . .

Advanced use. Any additional parameters are passed on to submit_job function.

execute

logical whether or not to submit the jobs

uuid

character Advanced use. This is the final path used for flow execution. Especially useful in case of re-running a flow.

plot

logical whether to make a pdf flow plot (saves it in the flow working directory).

dump

dump all the flow details to the flow path

.start_jid

Job to start this submission from. Advanced use, should be 1 by default.

Details

NOTE: Even if you want to kill the flow, its best to let submit_flow do its job, when done simply use kill(flow_wd). If submit_flow is interrupted, files like flow_details.rds etc are not created, thus flowr looses the association of jobs with flow instance and cannot monitor, kill or re-run the flow.

Examples

plot_flow

Plot a clean and scalable flowchart describing the (work)flow

Plot a flowchart using a flow object or flowdef

Usage

Arguments

x

Object of class flow, or a list of flow objects or a flowdef

. . .

experimental and only for advanced use.

detailed

include submission and dependency types in the plot [TRUE]

type

1 is original, and 2 is a elipse with less details [1]

pdf

create a pdf instead of plotting interactively [FALSE]

pdffile

output file name for the pdf file. [flow_path/flow_details.pdf]

Examples

status

Monitor status of flow(s)

Summarize status of a flow OR multiple flows OR a high-level summary of all flows in a folder.

Usage

Arguments

x

path to the flow root folder or a parent folder to summarize several flows.

use_cache

This skips checking status of jobs which have already been completed a and assumes no new jobs were submitted in the flow(s) being monitored. [FALSE] verbose

A numeric value indicating the amount of messages to produce. Values are integers varying from 0, 1, 2, 3, ... Please refer to the verbose page for more details. [get_opts("verbose")] [1]

out_format

passed onto knitr:::kable. supports: markdown, rst, html. . . [markdown]

. . .

not used

Details

basename(x) is used in a wild card search.

Get status of all the flows: (all flows with sleep_pipe in their name are checked and their status is shown) flowr status x=~/flowr/runs/sleep_pipe*

Provide a high level summary of ALL flows in a folder: flowr status x=~/flowr/runs

Use use_cache=TRUE to speed up checking the status. This assumes that no new jobs have been submitted and skips (re-)checking status of completed jobs.

Once all the jobs have been submitted to the cluster you may always use use_cache=TRUE.

Examples

kill

Kill all jobs submitted to the computing platform, for one or multiple flows

NOTE: This requires files which are created at the end of the submit_flow command. Even if you want to kill the flow, its best to let submit_flow do its job, when done simply use kill(flow_wd). If submit_flow is interrupted, files like flow_details.rds etc are not created, thus flowr looses the association of jobs with flow instance and cannot monitor, kill or re-run the flow.

Usage

Arguments

x

either path to flow wd or object of class flow

. . .

not used

force

You need to set force=TRUE, to kill multiple flows. This makes sure multiple flows are NOT killed by accident.

kill_cmd

The command used to kill. flowr tries to guess this commands, as defined in the detect_kill_cmd(). Supplying it here; fot custom platoforms.

verbose

A numeric value indicating the amount of messages to produce. Values are integers varying from 0, 1, 2, 3, ... Please refer to the verbose page for more details. [get_opts("verbose")] [1]

jobid_col

Advanced use. The column name in flow_details.txt file used to fetch jobids to kill

Examples

rerun

Re-run a pipeline in case of hardware or software failures.

hardware no change required, simple rerun: rerun(x=flow_wd)

software either a change to flowmat or flowdef has been made: rerun(x=flow_wd, mat = new_flowmat, def = new_flowdef)

NOTE: flow_wd: flow working directory, same input as used for status

Usage

Arguments

x

flow working directory

...

passed onto to_flow

mat

(optional) flowmat fetched from previous submission if missing. For more information regarding the format refer to to_flowmat

def

(optional) flowdef fetched from previous submission if missing. For more information regarding the format refer to to_flowdef

start_from

which job to start from, this is a job name.

execute

[logical] whether to execute or not

kill

(optional) logical indicating whether to kill the jobs from the previous execution of flow.

select

select a subset of jobs to rerun [character vector]

ignore

ignore a subset of jobs to rerun [character vector]

verbose

A numeric value indicating the amount of messages to produce. Values are integers varying from 0, 1, 2, 3, ... Please refer to the verbose page for more details. [get_opts("verbose")] [1]

Details

This function fetches details regarding the previous execution from the flow working directory (flow_wd).

It reads the flow object from the flow_details.rds file, and extracts flowdef and flowmat from it using to_flowmat and to_flowdef functions.

Using new flowmat OR flowdef for re-run:

Optionally, if either of flowmat or flowdef are supplied; supplied ones are used instead of those extracted from previous submission.

This functions efficiently updates job details of the latest submission into the previous file; thus information regarding previous job ids and their status is not lost.

Examples

Managing parameters

flowopts

Default options/params used in flowr and ngsflows

There are three helper functions which attempt to manage parameters used by flowr and ngsflows:

get_opts OR opts_flow$get(): show all default options

set_opts OR opts_flow$set(): set default options

load_opts OR opts_flow$load(): load options specified in a tab seperated text file

For more details regarding these funtions refer to params package.

Usage

Arguments

...

get: names of options to fetch

set: a set of options in a name=value format seperated by commas

Format

opts_flow

Details

By default flowr loads, ~/flowr/conf/flowr.conf and ~/flowr/conf/ngsflows.conf

Below is a list of default flowr options, retrieved via

opts_flow$get():

Examples

See also

fetch params read_sheet

setup

Setup and initialize flowr

This functions creates a directory structure in user's home directory. Additionally it creates a shortcut to the flowr helper script in ~/bin.

Usage

Arguments

bin

path to bin folder

flow_base_path

The base of flowr configuration and execution folders.

flow_run_path

Path to a folder. Main operating folder for this flow. [get_opts("flow_run_path")][~/flowr/runs].

flow_conf_path

Flowr configuration folder, used by fetch_conf.

flow_pipe_path

Folder with all pipelines, used by fetch_pipes.

Details

Will add more to this, to identify cluster and aid in other things.

verbose

Verbose levels, defining verboseness of messages

There are several levels of verboseness one can choose from. levels:

level 0 is almost silent, producing only necessary messages

level 1 is good for most purposes, where as,

level 2 is good when developing a new pipeline.

level 3 is good for debugging, especially when getting un-expected results.

One can set the level of verboseness using opts_flow$set(verbose=2), which will be used across flowr and ngsflows packages. Additionally one may set this value in the configurations files: ~/flowr/conf/flowr.conf OR ~/flowr/conf/ngsflows.conf.

Usage

Format

Examples

Managing pipelines

fetch

Two generic functions to search for pipelines and configuration files.

These functions help in searching for specific files in the user's space. fetch_pipes(): Fetches pipelines in the following places, in this specific order:

user's folder: ~/flowr/pipelines

current wd: ./

NOTE: If same pipeline is availabe in multiple places; intitutively, one from the later folder would be selected. As such, giving priority to user's home, and current working directories. fetch_conf(): Fetches configuration files in ALL of the following places:

package: conf folders in flowr and ngsflows packages.

user's folder: ~/flowr/conf folder.

current wd: ./

NOTE: This function would greedily return all matching conf files. One would load all of them in the order returned by this functions. If the same variable is repeated in multiple files, value from the later files would replace those formerly defined. Thus (as explained above), giving priority to options defined in user's home and current working directories. By default flowr loads, flowr.conf and ngsflows.conf. See the details sections, for more explanation on this.

Usage

Arguments

x

name of the file to search for (without extension). By default fetch_pipes and fetch_conf search for files ending with .R and .conf respectively.

places

places (paths) to look for files matching the name. Defaults are already defined in the function.

urls

urls to look for, works well for pipelines [not implemented yet]

verbose

A numeric value indicating the amount of messages to produce. Values are integers varying from 0, 1, 2, 3, ... Please refer to the verbose page for more details. [get_opts("verbose")] [1]

last_only

fetch_pipes():. If multiple pipelines match the pattern, return the last one. [TRUE]

silent

fetch_pipes(): logical, be silent even if no such pipeline is available. [FALSE]

ask

ask before downloading or copying. [not implemented]

. . .

[not implemented]

Details

For example flowr has a variable flow_run_path where it puts all the execution logs etc. The default value is picked up from packagess internal flowr.conf file. To redefine this value, one could create a new file called ~/flowr/conf/flowr.conf and add a line:

flow_run_pathmy_awesome_path, where is a tab character, since these are tab seperated files.

Also, at any time you can run, load_conf(super_specific_opts.conf); to load custom options.

Examples

See also

flowopts

run

Run automated Pipelines

Run complete pipelines, by wrapping several steps into one convinient function: Taking sleep_pipe as a example.

Use fetch_pipes to get paths to a Rscript, flowdef file and optionally a configuration file with various default options used.

Create a flowmat (using the function defined in the Rscript)

Create a `flow` object, using flowmat created and flowdef (as fetched using fetch_pipes)

Submit the flow to the cluster (using submit_flow)

Usage

Arguments

x

name of the pipeline to run. This is a function called to create a flow_mat.

platform

what platform to use, overrides flowdef

def

flow definition

flow_run_path

passed onto to_flow. Default it picked up from flowr.conf. Typically this is ~/flowr/runs

execute

TRUE/FALSE

...

passed onto the pipeline function as specified in x

Examples

Details on flowr's classes

queue-class

A queue object defines details regarding how a job is submitted

Internal function (used by to_flow), to define the format used to submit a job.

Usage

Arguments

object

this is not used currenlty, ignore.

platform

Required and important. Currently supported values are lsf and torque. [Used by class job]

format

[advanced use] We have a default format for the final command line string generated for lsf and torque.

queue

the type of queue your group usually uses bsub etc.

walltime

max walltime of a job.

memory

The amount of memory reserved. Units depend on the platform used to process jobs

cpu

number of cpus you would like to reserve [Used by class job]

extra_opts

[advanced use] Extra options to be supplied while create the job submission string.

submit_exe

[advanced use] Already defined by platform. The exact command used to submit jobs to the cluster example qsub

nodes

[advanced use] number of nodes you would like to request. Or in case of torque name of the nodes.optional [Used by class job]

jobname

[debug use] name of this job in the computing cluster

email

[advanced use] Defaults to system user, you may put you own email though may get tons of them.

dependency

[debug use] a list of jobs to complete before starting this one

server

[not used] This is not implemented currently. This would specify the head node of the computing cluster. At this time submission needs to be done on the head node of the cluster where flow is to be submitted

verbose

[logical] TRUE/FALSE

cwd

[debug use] Ignore

stderr

[debug use] Ignore

stdout

[debug use] Ignore

...

other passed onto object creation. Example: memory, walltime, cpu

Details

Resources: Can be defined **once** using a queue object and recylced to all the jobs in a flow. If resources (like memory, cpu, walltime, queue) are supplied at the job level they overwrite the one supplied in queue Nodes: can be supplied ot extend a job across multiple nodes. This is purely experimental and not supported.

Server: This a hook which may be implemented in future.

Submission script The platform variable defines the format, and submit_exe; however these two are avaible for someone to create a custom submission command.

Examples

job

Describing details of the job object

Internal function (used by to_flow), which aids in creating a job object.

Usage

Arguments

cmds

the commands to run

name

name of the job

q_obj

queue object

previous_job

character vector of previous job. If this is the first job, one can leave this empty, NA, NULL, ., or . In future this could specify multiple previous jobs.

cpu

no of cpus reserved

memory

The amount of memory reserved. Units depend on the platform used to process jobs

walltime

The amount of time reserved for this job. Format is unique to a platform. Typically it looks like 12:00 (12 hours reserved, say in LSF), in Torque etc. we often see measuring in seconds: 12:00:00

submission_type

submission type: A character with values: scatter, serial. Scatter means all the cmds would be run in parallel as seperate jobs. Serial, they would combined into a single job and run one-by-one.

dependency_type

depedency type. One of none, gather, serial, burst. If previous_job is specified, then this would not be none. [Required]

...

other passed onto object creation. Example: memory, walltime, cpu

Examples

flow-class

Describing the flow class

Internal function (used by to_flow), which aids in creating a flow object.

Usage

Arguments

jobs

list: A list of jobs to be included in this flow

name

character: Name of the flow. [newflow]

desc

character Description of the flow, used to uniquely identify a flow instance. [my_super_flow]

mode

character Mode of submission of the flow (depreciated). [scheduler]

flow_run_path

The base path of all the flows you would submit. [~/flows]

trigger_path

character [~/flows/trigger].

flow_path

character: A unique path identifying a flow instance, populated by submit_flow.

version

version of flowr used to create and execute this flow.

status

character: Status of the flow.

execute

executtion status of flow object. [FALSE]

Examples

Other helpful functions

get_unique_id

get_unique_id

Usage

Arguments

prefix

Default id. Character string to be added in the front.

suffix

Default . Character string to be added in the end.

random_length

Integer, defaults to 8. In our opinion 8 serves well, providing uniqueness and not being much of a eyesore.

Examples

to_flowdet

Create a flow's submission detail file

Create a file describing details regarding jobs ids, submission scripts etc.

Usage

Arguments

x

this is a wd

...

not used

Details

The path provided should contain a flow_detaits.rds file (which is used to extract all the information).

Incase a parent folder with multiple flows is provided information regarding jobids is omitted.

if x is char. assumed a path, check if flow object exists in it and read it. If there is no flow object, try using a simpler function

check_args

Assert none of the arguemnts of a function are null.

Checks all the arguments in the parent function and makes sure that none of them are NULL

Usage

Arguments

ignore

optionally ignore a few variables for checking.

select

optionally only check a few variables of the function.

Details

This function has now been moved to params package.

get_wds

Get all the (sub)directories in a folder

Usage

Arguments

x

path to a folder