flowr Documentation

Release 0.9.7

Sahil Seth

October 12, 2015

Contents

The documentation is outdated, please use: docs.flowr.space

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.

CHAPTER 1

Highlights

- Effectively process a pipeline multi-step pipeline, spawning it across the computing cluster
- **Example:**
  - A typical case with next-generation sequencing, a sample with tens of fastq files)
  - Each file can be processed (aligned) individually, each using multiple cores
  - Say 50 files using 10 cores each, totalling 500 cores across several machines, for one sample
  - flowr further supports processing multiple samples in parrellel, spawning thousands of cores.
- Reproducible, with cleanly structured execution logs
- Track and re-run flows
- Lean and Portable, with easy installation
- Supports multiple platforms (torque, lsf, sge, slurm ...)

1.1 A few lines, to get started:

```
## From the official R repository (may be a few versions behind)
install.packages("flowr")

## OR

install.packages(devtools)
devtools::install_github("sahilseth/flowr")

library(flowr) ## load the library
setup() ## copy flowr bash script; and create a folder flowr under home.
run('sleep', execute=TRUE,  platform='moab') ## submit a simple example
```

- Here is a shiny app, flow_creator which helps you build a flow.
- A few slides providing a quick overview.

CHAPTER 2

Contents:

2.1 Get started

Let us get a latest stable version of flowr, from CRAN. To get the latest features, your could also try the version from github.

```
install.packages('flowr') ## CRAN

install.packages('devtools')
devtools::install_github("sahilseth/flowr")
```

We have a quite handy command-line-interface for flowr, which exposes all functions of the package to terminal. Such that we dont have to open a interactive R session each time. To make this work, run a setup function which copies the 'flowr' helper script to your ~/bin directory. If you would like to do a test drive on its other capabilities, here are a few examples.

```
library(flowr)
```

```
setup()
```

```
## You could try this from the terminal
Rscript -e 'library(flowr);setup()'
## now run the following to confirm that ~/bin is added your PATH variable:
echo $PATH
## if ~/bin is not in your path, run and add the following to your ~/.bashrc
export PATH=$PATH:~/bin
## now run the following from the terminal, to check if setup worked fine.
flowr
```

2.1.1 Toy example

Consider, a simple example where we have **three** instances of the sleep command running (which basically stalls the terminal for few seconds and does nothing). After its completion **three** tmp files are created with some random

data. After this, a merge step follows, which combines them into one big file. Next we use `du` to calculate the size of the resulting file. This flow is shown in the above described figure.

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 some kind of a configuration file to describe which ones go first.

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
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. Each row in a flow mat relates to one job.

Notice how jobname column is being used a key throught the two tables. And how prev_jobs (previous jobs) defines what jobs need to complete before the one described in that row starts.

job-name	sub_type	prev_jobs	dep_type	queue	mem-ory_reserved	wall-time	cpu_reserved	plat-form	jo-bid
sleep	scatter	none	none	short	2000	1:00	1	torque	1
cre-ate_tmp	scatter	sleep	serial	short	2000	1:00	1	torque	2
merge	serial	cre-ate_tmp	gather	short	2000	1:00	1	torque	3
size	serial	merge	serial	short	2000	1:00	1	torque	4

2.1.2 Stitch it

We use the two files descirbed above and stich them to create a `flow object`, which contains all the information we need for submission to the cluster. Additionally we can give a name to this flow, using flowname argument and also override the platform described in `flow def`. Look at `to_flow` help file for more information.

```
fobj <- to_flow(x = flow_mat, def = as.flowdef(flow_def),
    flowname = "example1", platform = "lsf")
```

2.1.3 Plot it

We can use `plot_flow` to quickly visualize the flow; this really helps when developing complex workflows. Additionally, this function also works on the `flow definition` table as well (`plot_flow(flow_def)`.

```
plot_flow(fobj) # ?plot_flow for more information
```

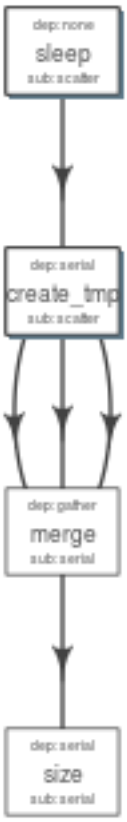

Fig. 2.1: Flow chart describing process for example 1

2.1.4 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. User's may spend some time checking the `*.sh` files for each of the jobs along with pdf of the flow etc.

```
submit_flow(fobj)
```

```
Test Successful!
You may check this folder for consistency. Also you may re-run submit with execute=TRUE
 ~/flowr/type1-20150520-15-18-27-5mSd32G0
```

2.1.5 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")
OR
flowr status x=~/flowr/type1-20150520-15-18-46-sySOzZnE
```

2.1.6 Check its status

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

Note: Please make sure to include `x=~` in status, to expicitly define the variable. Also unlike other command line tools you may skip adding "-" in from of each argument (no need of `-x=~`).

```
flowr status x=~/flowr/type1-20150520-15-18-46-sySOzZnE

Showing status of: /rsrch2/iacs/iacs_dep/sseth/flowr/type1-20150520-15-18-46-sySOzZnE
|          | 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, skip the full path, and mention only the parent direcotry, for example:

```
flowr status x=~/flowr/type1-20150520-15-18-46-sySOzZnE

Showing status of: /rsrch2/iacs/iacs_dep/sseth/flowr/type1-20150520-15-18-46-sySOzZnE
|          | 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.

2.1.7 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.

```
kill(x='fastq_haplotyper*')
Flowr: streamlining workflows
found multiple wds:
./fastq_haplotyper-MS132-20150825-16-24-04-0Lv1PbpI
/fastq_haplotyper-MS132-20150825-17-47-52-5vFIkrMD
Really kill all of them ? kill again with force=TRUE
```

To kill multiple, set force=TRUE:

```
kill(x='fastq_haplotyper*', force = TRUE)
```

While submission is in progress, and you figure, 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 with details regarding job ids etc are not created, thus flowr can't associate submitted jobs with flow instance (hence can't kill them). In such a situation you may resort to killing them manually.

```
## manual killing:
jobids=$(qstat | grep 'mypattern')
qdel $jobids
```

2.1.8 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>)`

In either case there are two things which are always required, a `flow_wd` (the folder created by flowr which contains execution logs) and name of the step from where we want to start execution. Refer to the help section for more details.

2.2 Ingredients for building a pipeline

An easy and quick way to build a workflow is create to create a set of two tab delimited files. First is a table with commands to run (for each module 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 introduces in the above sections).

Both these files have a `jobname` column which is used as a ID to connect them to each other.

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

```
## ------ load some example data
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"))
```

2.2.1 1. Flow Definition

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 a typical setup, if one step uses 16 cores these would be blocked and not used during processing of several other steps. Thus resulting in blockage and high cluster load (even when actual CPU usage may be low). Being able to tune them, makes this setup quite efficient.

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.

> **Warning:** If these (resource requirements) columns not included in the flow_def, their values should be explicitly defined in the submission template.

Here is an example of a typical flow_def file.

job-name	sub_type	prev_jobs	dep_type	queue	mem-ory_reserved	wall-time	cpu_reserved	plat-form	jo-bid
sleep	scatter	none	none	short	2000	1:00	1	torque	1
cre-ate_tmp	scatter	sleep	serial	short	2000	1:00	1	torque	2
merge	serial	cre-ate_tmp	gather	short	2000	1:00	1	torque	3
size	serial	merge	serial	short	2000	1:00	1	torque	4

2.2.2 2. Flow mat: A table with shell commands to run

This is also a tab separated table, with a minimum of three columns as defined below:

- `samplename`: A grouping column. The table is split using this column and each subset is treated as a individual flow. This makes it very easy to process multiple samples using a single submission command.
  - 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 defined above.
- `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.

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
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

Example:

A —-> B —-> C —-> D

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.

2.3 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.*

2.4 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.*

2.5 Relationships

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

2.5.1 Serial: one to one relationship

[scatter] —serial—> [scatter]

A is submitted as scatter, A1 through A10. Further B1, requires A1 to complete; B2 requires A2 and so on, but they need not wait for all of step A jobs to complete. Also B1 through B10 are independent of each other.

To set this up, A and B would have `sub_type` `scatter` and B would have `dep_type` as `serial`. Further, since A is an initial step its `dep_type` and `prev_job` would defined as `none`.

2.5.2 Gather: many to one relationship

[scatter] —gather—> [serial]

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.

2.5.3 Burst: one to many relationship

[serial] —burst—> [scatter]

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 and 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"))
fobj = suppressMessages(to_flow(x = ex2mat, def = ex2def))
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(fobj)
```

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

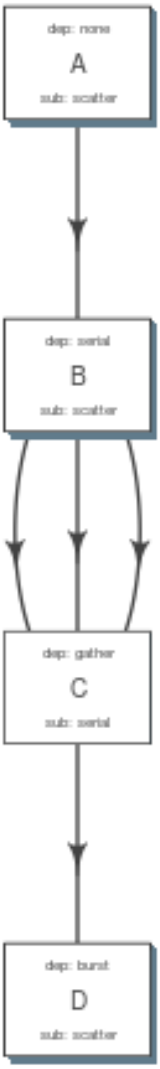

Fig. 2.2: plot of chunk build_pipe_plt_abcd

2.5.4 Passing of flow definition resource columns

The resource requirement columns of flow definition are passed along to the final (cluster) submission script.

The following table provides a mapping between the flow definition columns and variables in the submission template (*pipelines below*).

flow_def_column	hpc_script_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 **: This is gathered from flow_mat

2.6 Available Pipelines

Here are some of the available piplines along with their respective locations

name	def	conf	pipe
sleep_pipe	sleep_pipe.def	NA	/home/travis/build/sahilseth/flowr/inst/pipelines/sleep_pipe.R

2.7 Cluster Support

Support for several popular cluster platforms are built-in. There is a template, for each platform, which should would 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

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

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 accomodate this one can use regular expressions as described in the relevent section of the flowr config.

For example LSF may show a string such as:

```
Job <335508> is submitted to queue <transfer>.
```

```
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 !

3. **render dependency**: After collecting job ids from previous jobs, flowr render 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

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

Platform	command	status	queue.type
LSF 7	bsub	Not tested	lsf
LSF 9.1	bsub	Yes	lsf
Torque	qsub	Yes	torque
SGE	qsub	Beta	sge
SLURM	sbatch	under-dev	slurm

*queue short-name used in flow

- 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

2.8 Example of building a pipeline

A pipeline consists of several pieces, namely, a function which generates a flowmat, a flowdef and optionally a text file with parameters and paths to tools used as part of the pipeline.

A R function which creates a flow mat, is a module. Further a module with a flow definition is a pipeline.

We beleive pipeline and modules may be interchangeble, in the sense that a *smaller* pipeline may be included as part of a larger pipeline. In flowr a module OR pipeline always returns a flowmat. The only differnce being, a pipeline also has a correspomding flow definition file. As such, creating a flow definition for a module enables flowr to run it, hence a module **elevates**, becoming a pipeline. This lets the user mix and match several modules/pipelines to create a customized larger pipeline(s).

Let us follow through an example, providing more details regarding this process. Here are a few examples of modules, three functions `sleep`, `create_tmp` and `merge_size` each returning a flowmat.

2.8.1 Define modules

```
#' @param x number of sleep commands
sleep <- function(x, samplename){
    cmd = list(sleep = sprintf("sleep %s && sleep %s;echo 'hello'",
        abs(round(rnorm(x)*10, 0)),
        abs(round(rnorm(x)*10, 0))))
    flowmat = to_flowmat(cmd, samplename)
    return(list(flowmat = flowmat))
}

#' @param x number of tmp commands
create_tmp <- function(x, samplename){
    ## Create 100 temporary files
    tmp = sprintf("%s_tmp_%s", samplename, 1:x)
    cmd = list(create_tmp = sprintf("head -c 100000 /dev/urandom > %s", tmp))
    ## --- convert the list into a data.frame
    flowmat = to_flowmat(cmd, samplename)
    return(list(flowmat = flowmat, outfiles = tmp))
}

#' @param x vector of files to merge
merge_size <- function(x, samplename){
    ## Merge them according to samples, 10 each
    mergedfile = paste0(samplename, "_merged")
    cmd_merge <- sprintf("cat %s > %s",
        paste(x, collapse = " "), ## input files
        mergedfile)
    ## get the size of merged files
    cmd_size = sprintf("du -sh %s; echo 'MY shell:' $SHELL", mergedfile)

    cmd = list(merge = cmd_merge, size = cmd_size)
    ## --- convert the list into a data.frame
    flowmat = to_flowmat(cmd, samplename)
    return(list(flowmat = flowmat, outfiles = mergedfile))
}
```

We then define another function `sleep_pipe` which calls the above defined **modules**; fetches flowmat from each, creating a larger flowmat. This time we will define a flowdef for the `sleep_pipe` function, elevating its status from module to a pipeline.

2.8.2 Define the pipeline

```
#' @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))
}
```

2.8.3 Generate a flowmat

Here is how the generated flowmat looks like.

```
out = sleep_pipe(x = 3, "sample1")
flowmat = out$flowmat
```

samplename	jobname	cmd
sample1	sleep	sleep 16 && sleep 17;echo 'hello'
sample1	sleep	sleep 26 && sleep 8;echo 'hello'
sample1	sleep	sleep 6 && sleep 22;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

2.8.4 Create 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. Thus we will enable parallel process where possible, making this into a better pipeline.

Here is how it looks presently:

```
def = to_flowdef(flowmat)
```

```
## Creating a skeleton flow definition
## Following jobnames detected: sleep create_tmp merge size
## checking submission and dependency types...
```

```
plot_flow(suppressMessages(to_flow(flowmat, def)))
```

```
## checking submission and dependency types...
```

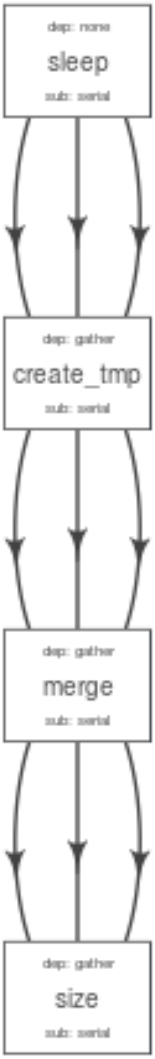

Fig. 2.3: plot of chunk unnamed-chunk-18

After making the desired changes, the new pipeline looks better. Alternatively, one may write this to a file and make other desired changes in resource requirements.

Pipeline follows the following steps, with dependencies mentioned in ():

- multiple sleep commands would run in parallel (none, first step)
- 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)

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

job-name	sub_type	prev_jobs	dep_type	queue	mem-ory_reserved	wall-time	cpu_reserved	plat-form	jo-bid
sleep	scatter	none	none	short	2000	1:00	1	torque	1
cre-ate_tmp	scatter	sleep	serial	short	2000	1:00	1	torque	2
merge	serial	cre-ate_tmp	gather	short	2000	1:00	1	torque	3
size	serial	merge	serial	short	2000	1:00	1	torque	4

```
plot_flow(suppressMessages(to_flow(flowmat, def)))
```

2.9 FAQs

Please visits the github issues with question tag for details FAQS.

Q What platforms are supported

A LSF, torque, SGE, ... more here.

Q Am getting a error in *devtools:::install_github("sahilseth/flowr")*

```
error:14090086:SSL routines:SSL3_GET_SERVER_CERTIFICATE:certificate verify failed
```

A This is basically a issue with httr (link) Try this:

```
install.packages("RCurl")
devtools:::install_github("sahilseth/flowr")

If not then try this:
```

```
install.packages("httr");
library(httr);
set_config( config( ssl.verifypeer = 0L ) )
devtools:::install_github("sahilseth/flowr")
```

2.10 Help on Available functions

2.10.1 check_args

checks all the arguments in the parent frame. None of them should be null.

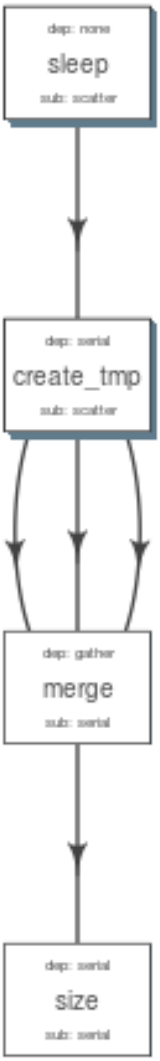

Fig. 2.4: plot of chunk unnamed-chunk-20

Description

This function may be optionally moved to a more generic package.

Usage

```
check_args(ignore, select)
```

Arguments

ignore optionally ignore a few variables for checking.

select optionally only check a few variables of the function.

Examples

Aliases: check_args .. Keywords:

2.10.2 check

Check consistency of flowdef and flowmat

Description

Currently checks S3 flowdef & flowmat for consistency.

Usage

```
check(x, ...)

## method for class 'flowmat'
check(x, ...)

## method for class 'flowdef'
check(x, verbose = get_opts("verbose"), ...)
```

Arguments

x a flowdef or flowmat object

... suppled to `check.classname` function

verbose be chatty

Examples

Aliases: check check.flowdef check.flowmat .. Keywords:

2.10.3 fetch

A generic functions to search for files

Description

These functions help in searching for specific files in the user's space.

fetch_pipes(): Fetches pipelines in the following places, - - available in 'pipelines' folders in flowr and ngsflows packages. - - ~/flowr/pipelines - - github repos (currently not supported)

fetch_conf(): Fetches configuration files in the following places,

- – available in 'conf' folders in flowr and ngsflows packages.
- – ~/flowr/conf folder

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

Usage

```
fetch(x, places, urls, verbose = get_opts("verbose"))

fetch_pipes(x, places, last_only = FALSE, urls = get_opts("flowr_pipe_urls"), silent = FALSE, ask = T

fetch_conf(x = "flowr.conf", places, ...)
```

Arguments

x name of the file to search for

places places (paths) to look for it. Its best to use the defaults

urls urls to look for, works well for pipelines.

verbose be chatty?

last_only [fetch_pipes only]. If multiple pipelines match the pattern, return the last one.

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

ask ask before downloading or copying, not used !

... not used

Examples

```
{
fetch_conf("torque.sh")
}
[1] "/Library/Frameworks/R.framework/Versions/3.2/Resources/library/flowr/conf/torque.sh"
```

Aliases: fetch fetch_conf fetch_pipes .. Keywords:

2.10.4 flow

Flow constructor

Description

Flow constructor

Usage

```
flow(jobs = list(new("job")), name = "newflow", desc = "my_super_flow", mode = c("scheduler", "trigge

is.flow(x)
```

Arguments

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

name `character` Name of the flow. Defaults to `'newname'`

Used in *submit_flow* to name the working directories.

desc `character` Description of the flow

This is used to name folders (when submitting jobs, see *submit_flow*). It is good practice to avoid spaces and other special characters. An underscore '_' seems like a good word separator. Defaults to 'my_super_flow'. We usually use this to put sample names of the data.

mode `character` Mode of submission of the flow.

flow_run_path The base path of all the flows you would submit.

Defaults to `~/flows`. Best practice to ignore it.

trigger_path `character`

Defaults to `~/flows/trigger`. Best practice to ignore it.

flow_path `character`

version version of flowr used to create and execute this flow.

status `character` Not used at this time

execute execuction status of flow object.

Examples

```
cmds = rep("sleep 5", 10)
qobj <- queue(platform='torque')
## run the 10 commands in parallel
jobj1 <- job(q_obj=qobj, cmd = cmds, submission_type = "scatter", name = "job1")

## run the 10 commands sequentially, but WAIT for the previous job to complete
## Many-To-One
jobj2 <- job(q_obj=qobj, cmd = cmds, submission_type = "serial",
 dependency_type = "gather", previous_job = "job1", name = "job2")

## As soon as first job on 'job1' is complete
## One-To-One
jobj3 <- job(q_obj=qobj, cmd = cmds, submission_type = "scatter",
 dependency_type = "serial", previous_job = "job1", name = "job3")
```

```
fobj <- flow(jobs = list(jobj1, jobj2, jobj3))

## plot the flow
plot_flow(fobj)
## **Not run**:
# ## dry run, only create the structure without submitting jobs
# submit_flow(fobj)
#
# ## execute the jobs: ONLY works on computing cluster, would fail otherwise
# submit_flow(fobj, execute = TRUE)
# ## **End(Not run)**
```

Aliases: flow is.flow .. Keywords:

2.10.5 get_unique_id

get_unique_id

Description

get_unique_id

Usage

```
get_unique_id(prefix = "id", suffix = "", random_length = 8)
```

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

```
## **Not run**:
# get_unique_id(base = id, random_length = 8)## **End(Not run)**
```

Aliases: get_unique_id .. Keywords:

internal .. Author:

2.10.6 get_wds

Get all the (sub)directories in a folder

Description

Get all the (sub)directories in a folder

Usage

```
get_wds(x)
```

Arguments

x path to a folder

Examples

Aliases: get_wds .. Keywords:

2.10.7 job

job class

Description

job class

Usage

```
job(cmds = "", name = "myjob", q_obj = new("queue"), previous_job = "", cpu = 1, memory, walltime, su
```

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 cpu's 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

```
qobj <- queue(platform="torque")

## torque job with 1 CPU running command 'sleep 2'
jobj <- job(q_obj=qobj, cmd = "sleep 2", cpu=1)

## multiple commands
cmds = rep("sleep 5", 10)

## run the 10 commands in parallel
jobj1 <- job(q_obj=qobj, cmd = cmds, submission_type = "scatter", name = "job1")

## run the 10 commands sequentially, but WAIT for the previous job to complete
jobj2 <- job(q_obj=qobj, cmd = cmds, submission_type = "serial",
   dependency_type = "gather", previous_job = "job1")

fobj <- flow(jobs = list(jobj1, jobj2))

## plot the flow
plot_flow(fobj)
## **Not run**:
# ## dry run, only create the structure without submitting jobs
# submit_flow(fobj)
#
# ## execute the jobs: ONLY works on computing cluster, would fail otherwise
# submit_flow(fobj, execute = TRUE)
#
# ## **End(Not run)**
```

Aliases: job .. Keywords:

2.10.8 kill

Killing a pipline requires files which are created at the END of the submit_flow commands.

Description

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, flow detail files etc are not created, thus flowr can't associate submitted jobs with flow instance.

Usage

```
kill(x, ...)

## method for class 'character'
kill(x, force = FALSE, ...)

## method for class 'flow'
kill(x, kill_cmd, jobid_col = "job_sub_id", ...)
```

Arguments

x either path to flow [character] or fobj object of class *flow*

... not used

force When killing multiple flows, force is neccesary. This makes sure multiple flows are killed by accident.

kill_cmd The command used to kill. Default is 'bkill' (LSF). One can used qdel for 'torque', 'sge' etc.

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

Examples

```
## **Not run**:
#
# ## example for terminal
# ## flowr kill_flow x=path_to_flow_directory
# ## In case path matches multiple folders, flowr asks before killing
# kill(x='fastq_haplotyper*')
#  Flowr: streamlining workflows
#  found multiple wds:
#  /fastq_haplotyper-MS132-20150825-16-24-04-0Lv1PbpI
#  /fastq_haplotyper-MS132-20150825-17-47-52-5vFIkrMD
#  Really kill all of them ? kill again with force=TRUE
#
# ## submitting again with force=TRUE will kill them:
# kill(x='fastq_haplotyper*', force = TRUE)
# ## **End(Not run)**
```

Aliases: kill kill.character kill.flow .. Keywords:

2.10.9 flowopts

Default options/params used in ngsflows and flowr

Description

There are three helper functions which attempt to manage params 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.

Usage

```
flowopts

opts_flow
```

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()`: <pre>

|name |value | |:————————|:————————| |default_regex |(.*) | |flow_base_path |~/flowr | |flow_conf_path |~/flowr/conf | |flow_parse_lsf |.*(<[0-9]*>).* | |flow_parse_moab |(.*) | |flow_parse_sge |(.*) | |flow_parse_slurm |(.*) | |flow_parse_torque |(.?)..* | |flow_pipe_paths |~/flowr/pipelines | |flow_pipe_urls |~/flowr/pipelines | |flow_platform |local | |flow_run_path |~/flowr/runs | |my_conf_path |~/flowr/conf | |my_dir |path/to/a/folder | |my_path |~/flowr | |my_tool_exe |/usr/bin/ls | |time_format |%a %b %e %H:%M:%S CDT %Y | |verbose |FALSE | </pre>

Examples

```
## Set options: set_opts()
opts = set_opts(flow_run_path = "~/mypath")
## OR if you would like to supply a long list of options:
opts = set_opts(.dots = list(flow_run_path = "~/mypath"))

## load options from a configuration file: load_opts()
myconfile = fetch_conf("flowr.conf")
load_opts(myconfile)
**Reading file, using 'V1' as id_column to remove empty rows.**<strong class='warning'>Warning messag


Seems like these paths do not exist, this may cause issues later:

</strong>

|name              |value                     |
|:-----------------|:-------------------------|
|flow_parse_slurm  |(.*)                      |
|flow_parse_sge    |(.*)                      |
|flow_parse_lsf    |.*(\<[0-9]*\>).*          |
|flow_parse_torque |(.?)\..*                  |
|flow_pipe_urls    |~/flowr/pipelines         |
|flow_pipe_paths   |~/flowr/pipelines         |
|flow_conf_path    |~/flowr/conf              |
|flow_base_path    |~/flowr                   |
|var               |                          |
|time_format       |%a %b %e %H:%M:%S CDT %Y  |

## Fetch options: get_opts()
get_opts("flow_run_path")
 flow_run_path
"~/flowr/runs"
get_opts()


|name              |value                     |
|:-----------------|:-------------------------|
```

```
|flow_base_path    |~/flowr                    |
|flow_conf_path    |~/flowr/conf               |
|flow_parse_lsf    |.*(\<[0-9]*\>).*           |
|flow_parse_moab   |(.*)                       |
|flow_parse_sge    |(.*)                       |
|flow_parse_slurm  |(.*)                       |
|flow_parse_torque |(.?)\..*                   |
|flow_pipe_paths   |~/flowr/pipelines          |
|flow_pipe_urls    |~/flowr/pipelines          |
|flow_platform     |local                      |
|flow_run_path     |~/flowr/runs               |
|time_format       |%a %b %e %H:%M:%S CDT %Y   |
|var               |                           |
|verbose           |FALSE                      |
```

Aliases: flowopts opts_flow .. Keywords:

datasets .. Author:

2.10.10 plot_flow

plot_flow

Description

plot the flow object

plot_flow.character: works on a flowdef file.

Usage

```
plot_flow(x, ...)

## method for class 'flow'
plot_flow(x, ...)

## method for class 'list'
plot_flow(x, ...)

## method for class 'character'
plot_flow(x, ...)

## method for class 'flowdef'
plot_flow(x, detailed = TRUE, type = c("1", "2"), pdf = FALSE, pdffile, ...)
```

Arguments

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

... experimental

detailed include some details

type 1 is original, and 2 is a elipse with less details

pdf create a pdf instead of plotting interactively

pdffile output file name for the pdf file

Examples

```
qobj = queue(type="lsf")
cmds = rep("sleep 5", 10)
jobj1 <- job(q_obj=qobj, cmd = cmds, submission_type = "scatter", name = "job1")
jobj2 <- job(q_obj=qobj, name = "job2", cmd = cmds, submission_type = "scatter",
             dependency_type = "serial", previous_job = "job1")
fobj <- flow(jobs = list(jobj1, jobj2))
plot_flow(fobj)

### Gather: many to one relationship
jobj1 <- job(q_obj=qobj, cmd = cmds, submission_type = "scatter", name = "job1")
jobj2 <- job(q_obj=qobj, name = "job2", cmd = cmds, submission_type = "scatter",
             dependency_type = "gather", previous_job = "job1")
fobj <- flow(jobs = list(jobj1, jobj2))
plot_flow(fobj)

### Burst: one to many relationship
jobj1 <- job(q_obj=qobj, cmd = cmds, submission_type = "serial", name = "job1")
jobj2 <- job(q_obj=qobj, name = "job2", cmd = cmds, submission_type = "scatter",
             dependency_type = "burst", previous_job = "job1")
fobj <- flow(jobs = list(jobj1, jobj2))
plot_flow(fobj)
```

Aliases: plot_flow plot_flow.character plot_flow.flow plot_flow.flowdef plot_flow.list .. Keywords:

2.10.11 queue

Create a `queue` object which containg details about how a job is submitted.

Description

This function defines the queue used to submit jobs to the cluster. In essence details about the computing cluster in use.

Usage

```
queue(object, platform = c("local", "lsf", "torque", "sge", "moab"), format = "", queue = "long", wal
```

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

```
qobj <- queue(platform='lsf')
```

Aliases: queue .. Keywords:

queue .. Author:

2.10.12 rerun

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

Description

- **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

```
rerun(x, ...)

## method for class 'character'
rerun(x, ...)

## method for class 'flow'
rerun(x, mat, def, start_from, execute = TRUE, kill = TRUE, ...)
```

Arguments

x flow working directory

... not used

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.

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. **New flowmat / flowdef** for re-run: Optionally, if either of these (flowmat or flowdef) are supplied, supplied ones are used instead for the new 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

```
## **Not run**:
# rerun_flow(wd = wd, fobj = fobj, execute = TRUE, kill = TRUE)
# ## **End(Not run)**
```

Aliases: rerun rerun.character rerun.flow .. Keywords:

2.10.13 run

run pipelines

Description

Running examples flows This wraps a few steps: Get all the commands to run (flow_mat) Create a *flow* object, using flow_mat and a default flowdef (picked from the same folder). Use *submit_flow()* to submit this to the cluster.

Usage

```
run(x, platform, def, flow_run_path = get_opts("flow_run_path"), execute = FALSE, ...)

run_pipe(x, platform, def, flow_run_path = get_opts("flow_run_path"), execute = FALSE, ...)
```

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 specified in x

Examples

Aliases: run run_flow run_pipe .. Keywords:

2.10.14 setup

Setup and initialize some scripts.

Description

Setup and initialize some scripts.

Usage

```
setup(bin = "~/bin", flow_base_path = get_opts("flow_base_path"))
```

Arguments

bin path to bin folder

flow_base_path the root folder for all flowr operations

Details

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

Examples

Aliases: setup .. Keywords:

2.10.15 status

status

Description

Summarize status of executed flow(x)

Usage

```
status(x, out_format = "markdown")

get_status(x, ...)

## method for class 'character'
get_status(x, out_format = "markdown", ...)

## method for class 'data.frame'
get_status(x, ...)

## method for class 'flow'
get_status(x, out_format = "markdown", ...)
```

Arguments

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

out_format passed onto knitr:::kable. supports: markdown, rst, html...

... not used

Details

basename(x) is used in a wild card search. - If x is a path with a single flow, it outputs the status of one flow. - If the path has more than one flow then this could give a summary of **all** of them. - Instead if x is supplied with paths to more than one flow, then this individually prints status of each. Alternatively, x can also be a flow object

Examples

```
## **Not run**:
# status(x = "~/flowr/runs/sleep_pipe*")
# ## an example for running from terminal
# flowr status x=path_to_flow_directory cores=6
# ## **End(Not run)**
```

Aliases: get_status get_status.character get_status.data.frame get_status.flow status .. Keywords:

2.10.16 submit_flow

submit_flow

Description

submit_flow

Usage

```
submit_flow(x, verbose = get_opts("verbose"), ...)

## method for class 'list'
submit_flow(x, verbose = get_opts("verbose"), ...)

## method for class 'flow'
submit_flow(x, verbose = get_opts("verbose"), execute = FALSE, uuid, plot = TRUE, dump = TRUE, .start
```

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.

Examples

```
## **Not run**:
# submit_flow(fobj = fobj, ... = ...)## **End(Not run)**
```

Aliases: submit_flow submit_flow.flow submit_flow.list .. Keywords:

2.10.17 test_queue

test_queue

Description

This function attempts to test the submission of a job to the queue. We would first submit one single job, then submit another with a dependency to see if configuration works. This would create a folder in home called 'flows'.

Usage

```
test_queue(q_obj, verbose = TRUE, ...)
```

Arguments

q_obj queue object

verbose toggle

... These params are passed onto `queue`. `?queue`, for more information

Examples

```
## **Not run**:
# test_queue(q_obj = q_obj, ... = ...)## **End(Not run)**
```

Aliases: test_queue .. Keywords:

2.10.18 to_flow

Create flow objects

Description

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

```
to_flow(x, ...)

## method for class 'vector'
to_flow(x, def, grp_col, jobname_col, cmd_col, ...)

## method for class 'flowmat'
to_flow(x, def, grp_col, jobname_col, cmd_col, flowname, flow_run_path, platform, submit = FALSE, exe

## method for class 'list'
to_flow(x, def, flowname, flow_run_path, desc, qobj, ...)
```

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). Default: *samplename*

jobname_col column name with job names. Default: *jobname*

cmd_col column name with commands. Default: *cmd*

flowname name of the flow

flow_run_path Path to a folder. Main operating folder for this flow. Default it *get_opts("flow_run_path")*.

platform character vector, specifying the platform to use. local, lsf, torque, moab, sge, slurm, ...

This over-rides the platform column in flowdef.

submit Depreciated. Use submit_flow on flow object this function returns. TRUE/FALSE

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

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

desc Advanced Use. final flow name, please don't change.

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

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

Examples

```
ex = file.path(system.file(package = "flowr"), "pipelines")
flowmat = as.flowmat(file.path(ex, "sleep_pipe.tsv"))
**mat seems to be a file, reading it...****Using `samplename` as the grouping column****Using `jobnar
**def seems to be a file, reading it...**fobj = to_flow(x = flowmat, def = flowdef, flowname = "sleep
**Using flow_run_path default: ~/flowr/runs****
##--- Checking flow definition and flow matrix for consistency...****
##--- Detecting platform...****Will use platform from flow definition****Platform supplied, this will
Working on... sample1****.****.****.****.**
```

Aliases: to_flow to_flow.flowmat to_flow.list to_flow.vector .. Keywords:

2.10.19 to_flowdef

Create a skeleton flow definition using a flowmat.

Description

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.

Usage

```
to_flowdef(x, ...)

## method for class 'flowmat'
to_flowdef(x, sub_type, dep_type, prev_jobs, queue = "short", platform = "torque", memory_reserved =

## method for class 'flow'
to_flowdef(x, ...)

## method for class 'character'
to_flowdef(x, ...)

as.flowdef(x, ...)

is.flowdef(x)
```

Arguments

x can a path to a flowmat, flomat 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 cpu's required

walltime amount of walltime required

x can be a data.frame or a path for a flow definition file

... passed onto check.flowdef

Examples

Aliases: as.flowdef is.flowdef to_flowdef to_flowdef.character to_flowdef.flow to_flowdef.flowmat .. Keywords:

2.10.20 to_flowdet

to_flowdet

Description

to_flowdet

get a flow_details file from the directory structure. This has less information than the one generated using a flow object. Lacks jobids etc...

Usage

```
to_flowdet(x, ...)

## method for class 'rootdir'
to_flowdet(x, ...)

## method for class 'character'
to_flowdet(x, ...)

## method for class 'flow'
to_flowdet(x, ...)
```

Arguments

x this is a wd

... not used

Details

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

Examples

Aliases: to_flowdet to_flowdet.character to_flowdet.flow to_flowdet.rootdir .. Keywords:

2.10.21 to_flowmat

Taking in a named list and returns a two columns data.frame

Description

Taking in a named list and returns a two columns data.frame

as.flowmat(): reads a file and checks for required columns. If x is data.frame checks for required columns.

Usage

```
to_flowmat(x, ...)

## method for class 'list'
to_flowmat(x, samplename, ...)

## method for class 'data.frame'
to_flowmat(x, ...)

## method for class 'flow'
to_flowmat(x, ...)

as.flowmat(x, grp_col, jobname_col, cmd_col, ...)

is.flowmat(x)
```

Arguments

x a named list OR vector. 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)

grp_col column used for grouping, default samplename.

jobname_col column specifying jobname, default jobname

cmd_col column specifying commands to run, default cmd

x a data.frame or path to file with flow details in it.

... not used

Examples

Aliases: as.flowmat is.flowmat to_flowmat to_flowmat.data.frame to_flowmat.flow to_flowmat.list .. Keywords:

2.10.22 whisker_render

Wrapper around whisker.render with some sugar on it...

Description

This is a wrapper around whisker.render

Usage

```
whisker_render(template, data)
```

Arguments

template template used

data a list with variables to be used to fill in the template.

Examples

Aliases: whisker_render .. Keywords:

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

CHAPTER 4

Aknowledgements

- Jianhua Zhang
- Samir Amin
- Kadir Akdemir
- Ethan Mao
- Henry Song
- An excellent resource for writing your own R packages: r-pkgs.had.co.nz