cgatcore Documentation

Release 1.0

CGAT Developers

Feb 08, 2023

GETTING STARTED

CGAT-core is a workflow management system that allows users to quickly and reproducibly build scalable data analysis pipelines. CGAT-core is a set of libraries and helper functions used to enable researchers to design and build computational workflows for the analysis of large-scale data-analysis.

Used in combination with CGAT-apps, we have deomonstrated the functionality of our flexible workflow management system using a simple RNA-seq pipeline in cgat-showcase.

CGAT-core is open-sourced, powerful and user-friendly, and has been continually developed as a Next Generation Sequencing (NGS) workflow management system over the past 10 years.

For more advanced examples of cgatcore utilities please refer to our cgat-flow repository, however please be aware that this is in constant development and has many software dependancies.

CHAPTER ONE

CITATION

Our workflow management system is published in F1000 Research:

Cribbs AP, Luna-Valero S, George C et al. CGAT-core: a python framework for building scalable, reproducible computational biology workflows [version 1; peer review: 1 approved, 1 approved with reservations]. F1000Research 2019, 8:377 (https://doi.org/10.12688/f1000research.18674.1)

CHAPTER

TWO

SUPPORT

- Please refer to our *FAQs* section
- For bugs and issues, please raise an issue on github
- For contributions, please refer to our contributor section and github source code.

CHAPTER THREE

EXAMPLES

cgat-showcase
: This is a toy example of how to develop a simple workflow. Please refer to the github page and the documentation.

cgat-flow
: As an example of the flexibility and functionality of CGAT-core, we have developed a set of fully tested production pipelines for automating the analysis of our NGS data. Please refer to the github page for information on how to install and use our code.

Single cell RNA-seq
: The cribbs lab use CGAT-core to develop pseudoalignment pipelines for single cell dropseq methods The sansom lab use the CGAT-core workflow engine to develop single cell sequencing analysis workflows.

CHAPTER

FOUR

SELECTED PUBLICATIONS USING CGAT-CORE

CGAT-core has been developed over the past 10 years and as such has been used in many previously published articles

For a non-comprehensive list of citations please see our :citing and *Citing and Citations*

4.1 Installation

The following sections describe how to install the cgatcore framework.

4.1.1 Conda Installation

The our preffered method of installation is using conda. If you dont have conda installed then please install conda using miniconda or anaconda.

cgatcore is currently installed using the bioconda channel and the recipe can be found on github. To install cgatcore:

```
conda install -c conda-forge -c bioconda cgatcore
```

4.1.2 Pip installation

We recommend installation through conda because it manages the dependancies. However, cgatcore is generally lightweight and can be installed easily using pip package manager. However, you may also have to install other dependancies manually:

```
pip install cgatcore
```

4.1.3 Automated installation

The following sections describe how to install the cgatcore framework.

The preferred method to install the cgatcore is using conda but we have also created a bash installation script, which uses conda under the hood.

Here are the steps:

```
# download installation script:
curl -O https://raw.githubusercontent.com/cgat-developers/cgat-core/master/install.sh

# see help:
```

(continues on next page)

(continued from previous page)

```
bash install.sh

# install the development version (recommended, no production version yet):
bash install.sh --devel [--location </full/path/to/folder/without/trailing/slash>]

# the code is downloaded in zip format by default. If you want to get a git clone, use:
--git # for an HTTPS clone
--git-ssh # for a SSH clone (you need to be a cgat-developer contributor on GitHub to do␣
↪this)

# enable the conda environment as requested by the installation script
# NB: you probably want to automate this by adding the instructions below to your .bashrc
source </full/path/to/folder/without/trailing/slash>/conda-install/etc/profile.d/conda.sh
conda activate base
conda activate cgat-c
```

The installation script will put everything under the specified location. The aim of the script is to provide a portable installation that does not interfere with the existing software. As a result, you will have a conda environment working with the cgat-core which can be enabled on demand according to your needs.

4.1.4 Manual installation

To obtain the latest code, check it out from the public git repository and activate it:

```
git clone https://github.com/cgat-developers/cgat-core.git
cd cgat-core
python setup.py develop
```

Once checked-out, you can get the latest changes via pulling:

```
git pull
```

4.1.5 Installing additonal software

When building your own workflows we recomend using conda to install software into your environment where possible.

This can easily be performed by:

```
conda search <package>
conda install <package>
```

4.1.6 Access libdrmaa shared library

You may also need access to the libdrmaa.so.1.0 C library, which can often be installed as part of the libdrmaa-dev package on most Unixes. Once you have installed that, you may need to tell DRMAA Python where it is installed by setting the DRMAA_LIBRARY_PATH environment variable, if it is not installed in a location that Python usually looks for libraries.

In order to set this correctly every time please add the following line to your bashrc, but set the library path to the location of the libdrmaa.so.1.0:

```
export DRMAA_LIBRARY_PATH=/usr/lib/libdrmaa.so.1.0
```

4.2 Cluster configuration

Currently SGE, SLURM, Torque and PBSPro workload managers are supported. The default cluster options for cgat-core are set for SunGrid Engine (SGE). Therefore, if you would like to run an alternative workload manager then you will need to configure your settings for your cluster. In order to do this you will need to create a `.cgat.yml` within the user`s home directory.

This will allow you to overide the default configurations. To view the hardcoded parameters for cgatcore please see the parameters.py file.

For an example of how to configure a PBSpro workload manager see this link to this config example.

The .cgat.yml is placed in your home directory and when a pipeline is executed it will automatically prioritise the `.cgat.yml` parameters over the cgatcore hard coded parameters. For example, adding the following to the .cgat.yml file will implement cluster settings for PBSpro:

```
    memory_resource: mem

options: -l walltime=00:10:00 -l select=1:ncpus=8:mem=1gb

queue_manager: pbspro

queue: NONE

parallel_environment:  "dedicated"
```

4.3 Running a pipeline

This section provides a tutorial-like introduction of how to run CGAT pipelines. As an example of how we build simple computational pipelines please refer to cgat-showcase. As an example of how we use cgatcore to build more complex computational pipelines, please refer to the code detailed in our cgat-flow repository.

4.3.1 Introduction

A pipeline takes input data and performs a series of automated steps on it to produce some output data.

Each pipeline is usually coupled with a report (usually MultiQC or Rmarkdown) document to summarize and visualize the results.

It really helps if you are familiar with:

- the unix command line to run and debug the pipeline
- python in order to understand what happens in the pipeline
- ruffus in order to understand the pipeline code
- sge (or any other workflow manager) in order to monitor your jobs
- git in order to up-to-date code

4.3.2 Setting up a pipeline

Step 1: Install cgat-showcase (our toy example of a cgatcore pipeline):

Check that your computing environment is appropriate and follow cgat-showcase installation instructions (see Installation instructions).

Step2: Clone the repository

To inspect the code and the layout clone the repository:

```
git clone https://github.com/cgat-developers/cgat-showcase.git
```

When inspecting the respoitory: The source directory will contain the pipeline master script named `cgatshowcase/pipeline_<name>.py`

The default configuration files will be contained in the folder `cgatshowcase/pipeline<Name>/`

All our pipelines are written to be lightweight. Therefore, a module file assoaiated with the pipeline master script, typically named `cgatshowcase/Module<Name>.py`, is usually where code required to run the tasks of the pipleine is located.

Step 3: To run a pipeline you will need to create a working directory and enter it. For example:

```
mkdir version1
cd version1/
```

This is where the pipeline will be executed and files will be generated in this directory.

However, the cgat-showcase example comes with test data and this can be downloaded by running:

```
wget https://www.cgat.org/downloads/public/showcase/showcase_test_data.tar.gz
tar -zxvf showcase_test_data.tar.gz
cd showcase_test_data
```

Step 4: Configure the cluster

Running pipelines on a cluster required the drmaa API settings to be configures and passed to cgatcore. The default cluster engine is SGE, however we also support SLURM and Torque/PBSpro. In order to execute using a non SGE cluster you will need to setup a *.cgat.yml* file in your home directory and specify parameters according to the cluster configuration documentation.

Step 5: Our pipelines are written with minimal hard coded options. Therefore, to run a pipeline an initial configuration file needs to be generated. A configuration file with all the default values can be obtained by running:

```
cgatshowcase <name> config
```

For example, if you wanted to run the transdiffexprs pipeline you would run:

```
cgatshowcase transdiffexprs config
```

This will create a new `pipeline.yml` file. **YOU MUST EDIT THIS FILE**. The default values are unlikely to be configured correctly for your data. The configuration file should be well documented and the format is simple. The documenation for the ConfigParser python module contains the full specification.

Step 6: Add the input files. The required input is specific for each pipeline in the documentation string at the; read the pipeline documentation to find out exactly which files are needed and where they should be put. Commonly, a pipeline works from input files linked into the working directory and named following pipeline specific conventions.

Step 7: You can check if all the external dependencies to tools and R packages are satisfied by running:

```
cgatshowcase <name> check
```

4.3.3 Running a pipeline

pipelines are controlled by a single python script called `pipeline_<name>.py` that lives in the source directory. Command line usage information is available by running:

```
cgatshowcase <name> --help
```

Alternatively, you can call the python script directly:

```
python /path/to/code/cgatshowcase/pipeline_<name>.py --help
```

The basic syntax for `pipeline_<name>.py` is:

```
cgatshowcase <name> [workflow options] [workflow arguments]
```

For example, to run the readqc pipeline you would run the following:

```
cgatshowcase readqc make full
```

`workflow options` can be one of the following:

make <task>

  run all tasks required to build task

show <task>

  show tasks required to build task without executing them

plot <task>

  plot image of workflow (requires inkscape) of pipeline state for task

touch <task>

  touch files without running task or its pre-requisites. This sets the timestamps for files in task and its pre-requisites such that they will seem up-to-date to the pipeline.

config

write a new configuration file `pipeline.ini` with default values. An existing configuration file will not be overwritten.

clone <srcdir>

clone a pipeline from `srcdir` into the current directory. Cloning attempts to conserve disk space by linking.

In case you are running a long pipeline, make sure you start it appropriately, for example:

```
nice -19 nohup cgatshowcase <name> make full -v5 -c1
```

This will keep the pipeline running if you close the terminal.

Fastq naming convention

Most of our pipelines assume that input fastq files follows the following naming convention (with the read inserted between the fastq and the gz. The reason for this is so that regular expressions do not have to acount for the read within the name. It is also more explicit:

```
sample1-condition.fastq.1.gz
sample1-condition.fastq.2.gz
```

Additional pipeline options

In addition to running the pipeline with default command line options, running a pipeline with –help will allow you to see additional options for `workflow arguments` when running the pipelines. These will modify the way the pipeline in ran.

- *-no-cluster*

This option allows the pipeline to run locally.

- *-input-validation*

This option will check the pipeline.ini file for missing values before the pipeline starts.

- *-debug*

Add debugging information to the console and not the logfile

- *-dry-run*

Perform a dry run of the pipeline (do not execute shell commands)

- *-exceptions*

Echo exceptions immidietly as they occur.

-c - -checksums

Set the level of ruffus checksums.

4.3.4 Building pipeline reports

We always associate some for of reporting with our pipelines to display summary information as a set of nicely formatted html pages.

Currently in CGAT we have 3 preferred types of report generation.

- MultiQC report (for general alignment and tool reporting)
- R markdown (for bespoke reporting)
- Jupyter notebook (for bespoke reporting)

To determine which type of reporting is implimented for each pipeline, refer to the specific pipeline documentation at the beginning of the script.

Reports are generated using the following command once a workflow has completed:

```
cgatshowcase <name> make build_report
```

MultiQC report

MultiQC is a python framework for automating reporting and we have imliemnted it in the majority of our workflows to generate QC stats for frequently used tools (mostly in our generic workflows).

R markdown

R markdown report generation is very useful for generating bespoke reports that require user defined reporting. We have implimented this in our bamstats workflow.

Jupyter notebook

Jupyter notebook is a second approach that we use to produce bespoke reports. An example is also implimented in our bamstats workflow.

4.3.5 Troubleshooting

Many things can go wrong while running the pipeline. Look out for

- bad input format. The pipeline does not perform sanity checks on the input format. If the input is bad, you might see wrong or missing results or an error message.
- pipeline disruptions. Problems with the cluster, the file system or the controlling terminal might all cause the pipeline to abort.
- bugs. The pipeline makes many implicit assumptions about the input files and the programs it runs. If program versions change or inputs change, the pipeline might not be able to deal with it. The result will be wrong or missing results or an error message.

If the pipeline aborts, locate the step that caused the error by reading the logfiles and the error messages on stderr (`nohup.out`). See if you can understand the error and guess the likely problem (new program versions, badly formatted input, . . .). If you are able to fix the error, remove the output files of the step in which the error occured and restart the pipeline. Processing should resume at the appropriate point.

Note: Look out for upstream errors. For example, the pipeline might build a geneset filtering by a certain set of contigs. If the contig names do not match, the geneset will be empty, but the geneset building step might conclude successfully. However, you might get an error in any of the downstream steps complaining that the gene set is empty. To fix this, fix the error and delete the files created by the geneset building step and not just the step that threw the error.

Common pipeline errors

One of the most common errors when runnig the pipeline is:

```
GLOBAL_SESSION = drmaa.Session()
NameError: name 'drmaa' is not defined
```

This error occurrs because you are not connected to the cluster. Alternatively you can run the pipleine in local mode by adding - *-no-cluster* as a command line option.

Updating to the latest code version

To get the latest bugfixes, go into the source directory and type:

```
git pull
```

The first command retrieves the latest changes from the master repository and the second command updates your local version with these changes.

Using qsub commands

We would always recommend using cgat-core to perform the job submission as this is handled in the background without the need to use qsub commands. However, if users wish to use qsub then it is perfectly simple to do so. Since our statements to P.run() are essentially commandline scripts then you can write the qsub as you would normally do when sending a script to the commandline. For example:

```
statement = "qsub [commands] echo 'This is where you would put commands you wan ran' "

P.run(statament)
```

When running the pipeline make sure you specify *–no-cluster* as a commandlie option and your good to go.

4.4 Running a pipeline - Tutorial

Before beginning this tutorial make sure you have the CGAT-core installed correctly, please see here (see *Installation*) for installation instructions.

As a tutorial example of how to run a CGAT workflow we will run the cgat-showcase pipeline. Therefore, you will also need to install the cgat-showcase (see instructions)

The aim of this pipeline is to perform pseaudoalignment using kallisto. The pipeline can be ran locally or dirtributed accross a cluster. This tutorial will explain the steps required to run this pipeline. Further documentation on cgat-showcase can be found here.

The cgat-showcase highlights some of the functionality of cgat-core. However, we also have our utility pipelines contained in the cgat-flow repository which demonstrate our advanced pipelines for next-generation sequencing analysis (see cgat-flow).

4.4.1 Tutorial start

1. First download the tutorial data:

```
mkdir showcase
cd showcase
wget https://www.cgat.org/downloads/public/showcase/showcase_test_data.tar.gz
tar -zxvf showcase_test_data.tar.gz
```

2. Next we will generate a configuration yml file so the pipeline output can be modified:

```
cd showcase_test_data
cgatshowcase transdiffexpres config
```

or you can alternatively call the workflow file directly:

```
python /path/to/file/pipeline_transdiffexpres.py config
```

This will generate a **pipeline.yml** file containing the configuration parameters than can be used to modify the output of the pipleine. However, for this tutorial you do not need to modify the parameters to run the pipeline. In the modify_config section below I have detailed how you can modify the config file to change the output of the pipeline.

3. Next we will run the pipleine:

```
cgatshowcase transdiffexpres make full -v5 --no-cluster
```

This `--no-cluster` will run the pipeline locally if you do not have access to a cluster. Alternatively if you have a cluster remove the `--no-cluster` option and the pipleine will distribute your jobs accross the cluster.

Note: There are many commandline options available to run the pipeline. To see available options please run `cgatshowcase --help`.

4. Generate a report

The final step is to generate a report to display the output of the pipeline. We have a preference for using MultiQC for generate bioinformatics tools (such as mappers and pseudoaligners) and Rmarkdown for generating custom reports. In order to generate these run the command:

```
cgatshowcase transdiffexprs make build_report -v 5 --no-cluster
```

This will generate a MultiQC report in the folder *MultiQC_report.dir/* and an Rmarkdown report in *R_report.dir/*.

This completes the tutorial for running the transdiffexprs pipeline for cgat-showcase, hope you find it as useful as we do for writing workflows within python.

4.5 Writing a workflow

4.5.1 Our workflow philosophy

The explicit aim of CGAT-core is to allow users to quickly and easily build their own computational pipelines that will speed up your analysis workflow.

When building pipelines it is often useful to keep in mind the following philosophy:

Flexibility
There are always new tools and insights that could be incorporated into a pipeline. Ultimately, a pipeline should be flexible and the code should not constrain you when implimenting new features.

Scriptability
The pipeline should be scriptable, i.e, the whole pipeline can be run within another pipeline. Similarly, parts of a pipeline can be duplicated to process several data streams in parallel. This is a crucial feature in genome studies as a single analysis will not permit making inferences by itself. When we write a pipeline we usually attempt to write a command line script (and include it in the CGAT-apps repository) and then run this script as a command line statement in the pipeline.

Reproducibility
The pipeline is fully automated. The same inputs and configuration will produce the same outputs.

Reusability
The pipeline should be able to be re-used on similar data, preferably only requiring changes to a configuration file (pipeline.yml).

Archivability
Once finished, the whole project should be able to be archived without too many major dependencies on external data. This should be a simple process and hence all project data should be self-contained. It should not involve going through various directories or databases to figure out which files and tables belong to a project or a project depends on.

4.5.2 Building a pipeline

The best way to build a pipeline is to start from an example. In cgat-showcase we have a toy example of an RNA-seq analysis pipeline that aims to show users how simple workflows can be generated with minimal code. cgat-flow demonstrates a set of complex workflows.

For a step by step tutorial on how to run the pipelines please refer to our *Running a pipeline - Tutorial*.

For help on how to construct pipelines from scratch please continue reading for more information.

In an empty directory you will need to make a new directory and then a python file with the same name. For example:

```
mkdir test && touch pipeline_test.py
```

All pipelines require a yml configuration file that will allow you to add configurable values to modify the behaviour of your code. This is placed within the test/ directory, which should have the same name as the name of your pipeline_test.py file:

```
touch test/pipeline.yml
```

In order to help with debugging and reading our code, our pipelines are written so that a pipeline task file contains Ruffus tasks and calls functions in an associated module file, which contains all of the code to transform and analyse the data.

Therefore, if you wish to create a module file, we usually save this file in the following convention, `ModuleTest.py` and it can be imported into the main pipeline task file (`pipeline_test.py`)as:

```
import ModuleTest
```

This section describes how pipelines can be constructed using the `pipeline` module in cgat-core. The pipeline module contains a variety of useful functions for pipeline construction.

4.5.3 pipeline input

Pipelines are executed within a dedicated working directory. They usually require the following files within this directory:

- a pipeline configuration file `pipeline.yml`
- input data files, usually listed in the documentatuion of each pipeline

Other files that might be used in a pipeline are:

- external data files such as genomes that a referred to by they their full path name.

The pipelines will work from the input files in the working directory, usually identified by their suffix. For example, a mapping pipeline might look for any `*.fastq.gz` files in the directory, run QC on these and map the reads to a genome sequence etc.

4.5.4 pipeline output

The pipeline will create files and database tables in the working directory. When building a pipeline, you can choose any file/directory layout that suits your needs. Some prefer flat hierarchies with many files, while others prefer deep directories.

4.5.5 Guidelines

To preserve disk space, we always use compressed files as much as possible. Most data files compress very well, for example fastq files often compress by a factor of 80% or more: a 10Gb file will use just 2Gb.

Working with compressed files is straight-forward using unix pipes and the commands `gzip`, `gunzip` or `zcat`.

If you require random access to a file, load the file into the database and index it appropriately. Genomic interval files can be indexed with tabix to allow random access.

4.5.6 Import statements

In order to run our pipelines you will need to import the cgatcore python modules into your pipeline. For every CGAT pipeline we recommend importing the basic modules as follows. Then any additional modules can be imported as required.

```
from ruffus import *
import cgatcore.experiment as E
from cgatcore import pipeline as P
import cgatcore.iotools as iotools
```

4.5.7 Selecting the appropriate Ruffus decorator

Before starting to write a pipeline it is always best to map out on a whiteboard the the steps and flow of your potential pipeline. This will allow you to identify the input and outputs of each task. Once you have assessed this then the next step is to identify which Ruffus decorator you require. Documentation on each decorator can be found in the ruffus documentation

4.5.8 Running commands within tasks

To run a command line program within a pipeline task, build a statement and call the `pipeline.run()` method:

```
@transform( '*.unsorted', suffix('.unsorted'), '.sorted')
def sortFile( infile, outfile ):

    statement = '''sort %(infile)s > %(outfile)s'''
    P.run(statement)
```

On calling the `pipeline.run()` method, the environment of the caller is examined for a variable called `statement`. The variable is subjected to string substitution from other variables in the local namespace. In the example above, `%(infile)s` and `%(outfile)s` are substituted with the values of the variables `infile` and `outfile`, respectively.

The same mechanism also permits setting configuration parameters, for example:

```
@transform( '*.unsorted', suffix('.unsorted'), '.sorted')
def sortFile( infile, outfile ):

    statement = '''sort -t %(tmpdir)s %(infile)s > %(outfile)s'''
    P.run(statement)
```

will automatically substitute the configuration parameter `tmpdir` into the command. See *ConfigurationValues* for more on using configuration parameters.

The pipeline will stop and return an error if the command exits with an error code.

If you chain multiple commands, only the return value of the last command is used to check for an error. Thus, if an upstream command fails, it will go unnoticed. To detect these errors, insert && between commands. For example:

```
@transform( '*.unsorted.gz', suffix('.unsorted.gz'), '.sorted)
def sortFile( infile, outfile ):

    statement = '''gunzip %(infile)s %(infile)s.tmp &&
                  sort -t %(tmpdir)s %(infile)s.tmp > %(outfile)s &&
                  rm -f %(infile)s.tmp
    P.run(statement)
```

Of course, the statement aboved could be executed more efficiently using pipes:

```
@transform( '*.unsorted.gz', suffix('.unsorted.gz'), '.sorted.gz')
def sortFile( infile, outfile ):

    statement = '''gunzip < %(infile)s
                  | sort -t %(tmpdir)s
                  | gzip > %(outfile)s'''
    P.run(statement)
```

The pipeline inserts code automatically to check for error return codes if multiple commands are combined in a pipe.

4.5.9 Running commands on the cluster

In order to run commands on cluster, use `to_cluster=True`.

To run the command from the previous section on the cluster:

```
@files( '*.unsorted.gz', suffix('.unsorted.gz'), '.sorted.gz')
def sortFile( infile, outfile ):

    to_cluster = True
    statement = '''gunzip < %(infile)s
                   | sort -t %(tmpdir)s
                   | gzip > %(outfile)s'''
    P.run(statement)
```

The pipeline will automatically create the job submission files, submit the job to the cluster and wait for its return.

pipelines will use the command line options `--cluster-queue`, `--cluster-priority`, etc. for global job control. For example, to change the priority when starting the pipeline, use:

```
python <pipeline_script.py> --cluster-priority=-20
```

To set job options specific to a task, you can define additional variables:

```
@files( '*.unsorted.gz', suffix('.unsorted.gz'), '.sorted.gz')
def sortFile( infile, outfile ):

    to_cluster = True
    job_queue = 'longjobs.q'
    job_priority = -10
    job_options= "-pe dedicated 4 -R y"

    statement = '''gunzip < %(infile)s
                   | sort -t %(tmpdir)s
                   | gzip > %(outfile)s'''
    P.run(statement)
```

The above statement will be run in the queue `longjobs.q` at a priority of `-10`. Additionally, it will be executed in the parallel environment `dedicated` with at least 4 cores.

Array jobs can be controlled through the `job_array` variable:

```
@files( '*.in', suffix('.in'), '.out')
def myGridTask( infile, outfile ):

    job_array=(0, nsnps, stepsize)

    statement = '''grid_task.bash %(infile)s %(outfile)s
       > %(outfile)s.$SGE_TASK_ID 2> %(outfile)s.err.$SGE_TASK_ID
    '''
    P.run(statement)
```

Note that the `grid_task.bash` file must be grid engine aware. This means it makes use of the `SGE_TASK_ID`, `SGE_TASK_FIRST`, `SGE_TASK_LAST` and `SGE_TASK_STEPSIZE` environment variables to select the chunk of data it wants to work on.

The job submission files are files called *tmp** in the working directory. These files will be deleted automatically. However, the files will remain after aborted runs to be cleaned up manually.

4.5.10 Useful information regarding decorators

To see a full list of ruffus decorators that control the flow of the pipeline please see the ruffus documentation.

However, during peer review it was pointed out that it would be helpful to include a few examples of how you can modify the infile name and transform it to the output filename. There are a few ways of doing this:

The first way is to capture the suffix so the outfile is placed into the same folder as the infile:

```
# pairs are a tuple of read pairs (read1, read2)
@transform(pairs,
           suffix(.fastq.gz),
           ("_trimmed.fastq.gz", "_trimmed.fastq.gz"))
```

This will transform an input <name_of_file>.fastq.gz and result in an output with a new siffix <name_of_file>_trimmed.fastq.gz.

Another way to add a output file into aother filer is to use a regex:

```
@follows(mkdir("new_folder.dir"))
@transform(pairs,
          regex((\S+).fastq.gz),
          (r"new_folder.dir/\1_trimmed.fastq.gz", r"new_folder.dir/\1_trimmed.fastq.gz"))
```

This can also be achieved using the formatter function:

```
@follows(mkdir("new_folder.dir"))
 @transform(pairs,
           formatter((\S+).fastq.gz),
           ("new_folder.dir/{SAMPLE[0]}_trimmed.fastq.gz", r"new_folder.dir/{SAMPLE[0]}_
↪trimmed.fastq.gz"))
```

4.5.11 Combining commands together

In order to combine commands together you will need to use *&&* to make sure your commands are chained correctly. For example:

```
statement = """
            module load cutadapt &&
            cutadapt ....
            """

P.run(statement)
```

If you didnt have the *&&* then the command will fail because the cutadapt command will be executed as part of the module load statement.

4.5.12 Databases

Loading data into the database

`pipeline.py` offers various tools for working with databases. By default, it is configured to use an sqlite3 database in the working directory called `csvdb`.

Tab-separated output files can be loaded into a table using the `pipeline.load()` function. For example:

```
@jobs_limit(PARAMS.get("jobs_limit_db", 1), "db")
@transform('data_*.tsv.gz', suffix('.tsv.gz'), '.load')
def loadTables(infile, outfile):
   P.load(infile, outfile)
```

The task above will load all tables ending with `tsv.gz` into the database Table names are given by the filenames, i.e, the data in `data_1.tsv.gz` will be loaded into the table `data_1`.

The load mechanism uses the script `csv2db.py` and can be configured using the configuration options in the `database` section of `pipeline.ini`. Additional options can be given via the optional *options* argument:

```
@jobs_limit(PARAMS.get("jobs_limit_db", 1), "db")
@transform('data_*.tsv.gz', suffix('.tsv.gz'), '.load')
def loadTables( infile, outfile ):
   P.load(infile, outfile, "--add-index=gene_id")
```

In order for the load mechanism to be transparent, it is best avoided to call the `csv2db.py` script directly. Instead, use the `pipeline.load()` function. If the `csv2db.py` needs to called at the end of a succession of statements, use the `pipeline.build_load_statement()` method, for example:

```
def loadTranscript2Gene(infile, outfile):
    '''build and load a map of transcript to gene from gtf file
    '''
    load_statement = P.build_load_statement(
        P.toTable(outfile),
        options="--add-index=gene_id "
        "--add-index=transcript_id ")

    statement = '''
    gunzip < %(infile)s
    | python %(scriptsdir)s/gtf2tsv.py --output-map=transcript2gene -v 0
    | %(load_statement)s
    > %(outfile)s'''
    P.run()
```

See also the variants `pipeline.mergeAndLoad()` and *:meth: `pipeline.concatenateAndLoad* to combine multiple tables and upload to the database in one go.

Connecting to a database

To use data in the database in your tasks, you need to first connect to the database. The best way to do this is via the connect() method in pipeline.py.

The following example illustrates how to use the connection:

```
@transform( ... )
def buildCodingTranscriptSet( infile, outfile ):

    dbh = connect()

    statement = '''SELECT DISTINCT transcript_id FROM transcript_info WHERE transcript_
↪biotype = 'protein_coding' '''
    cc = dbh.cursor()
    transcript_ids = set( [x[0] for x in cc.execute(statement)] )
    ...
```

4.5.13 Reports

MultiQC

When using cgat-core to build pipelines we recomend using MultiQC as the default reporting tool for generic thrid party computational biology software.

To run multiQC in our pipelines you only need to run a statement as a commanline task. For example we impliment this in our pipelines as:

```
@follows(mkdir("MultiQC_report.dir"))
@originate("MultiQC_report.dir/multiqc_report.html")
def renderMultiqc(infile):
'''build mulitqc report'''

statement = '''LANG=en_GB.UTF-8 multiqc . -f;
               mv multiqc_report.html MultiQC_report.dir/'''

P.run(statement)
```

Rmarkdown

MultiQC is very useful for running third generation computational biology tools. However, currently it is very difficult to use it as a bespoke reporting tool. Therefore, one was of running bespoke reports is using the Rmarkdown framework and using the render functionality of knitr.

Rendering an Rmarkdown document is very easy if you place the .Rmd file in the same test/ directory as the pipeline.yml. Then the file can easily run using:

```
@follows(mkdir("Rmarkdown.dir"))
@originate("Rmarkdown.dir/report.html")
def render_rmarkdown(outfile):

NOTEBOOK_ROOT = os.path.join(os.path.dirname(__file__), "test")
```

(continues on next page)

(continued from previous page)

```
statement = '''cp %(NOTEBOOK_ROOT)s/report.Rmd Rmarkdown.dir &&
               cd Rmarkdown.dir/ && R -e "rmarkdown::render('report.Rmd',encoding = 'UTF-
↪8')" '''

P.run(statement)
```

This should generate an html output of whatever report your wrote for your particular task.

Jupyter notebook

Another bespoke reporting that we also perform for our pipelines is to use a Jupyter notebook implimentation and execture it in using the commandline. All that is required is that you place your jupyter notebook into the same test/ directory as the pipeline.yml and call the following:

```
@follows(mkdir("jupyter_report.dir"))
@originate("jupyter_report.dir/report.html")
def render_jupyter(outfile):

NOTEBOOK_ROOT = os.path.join(os.path.dirname(__file__), "test")

statement = '''cp %(NOTEBOOK_ROOT)s/report.ipynb jupyter_report.dir/ && cd jupyter_
↪report.dir/ &&
                jupyter nbconvert --ExecutePreprocessor.timeout=None --to html --execute␣
↪*.ipynb --allow-errors;

P.run(statement)
```

4.5.14 Configuration values

Setting up configuration values

There are different ways to pass on configuration values to pipelines. Here we explain the priority for all the possible options so you can choose the best one for your requirements.

The pipeline goes *in order* through different configuration options to load configuration values and stores them in the `PARAMS` dictionary. This order determines a priority so values read in the first place can be overwritten by values read in subsequent steps; i.e. values read lastly have higher priority.

Here is the order in which the configuration values are read:

1. Hard-coded values in `cgatcore/pipeline/parameters.py`.
2. Parameters stored in `pipeline.yml` files in different locations.
3. Variables declared in the ruffus tasks calling `P.run()`; e.g. `job_memory=32G`
4. `.cgat.yml` file in the home directory
5. `cluster_*` options specified in the command line; e.g. `python pipeline_example.py --cluster-parallel=dedicated make full`

This means that configuration values for the cluster provided as command-line options will have the highest priority. Therefore:

```
python pipeline_example.py --cluster-parallel=dedicated make full
```

will overwrite any `cluster_parallel` configuration values given in `pipeline.yml` files. Type:

```
python pipeline_example.py --help
```

to check the full list of available command-line options.

You are encouraged to include the following snippet at the beginning of your pipeline script to setup proper configuration values for your analyses:

```
# load options from the config file
from cgatcore import pipeline as P
# load options from the config file
P.get_parameters(
 ["%s/pipeline.yml" % os.path.splitext(__file__)[0],
  "../pipeline.yml",
  "pipeline.yml"])
```

The method `pipeline.getParameters()` reads parameters from the `pipeline.yml` located in the current working directory and updates `PARAMS`, a global dictionary of parameter values. It automatically guesses the type of parameters in the order of `int()`, `float()` or `str()`. If a configuration variable is empty (`var=`), it will be set to `None`.

However, as explained above, there are other `pipeline.yml` files that are read by the pipeline at start up. In order to get the priority of them all, you can run:

```
python pipeline_example.py printconfig
```

to see a complete list of `pipeline.yml` files and their priorities.

Using configuration values

Configuration values are accessible via the `PARAMS` variable. The `PARAMS` variable is a dictionary mapping configuration parameters to values. Keys are in the format `section_parameter`. For example, the key `bowtie_threads` will provide the configuration value of:

```
bowtie:
    threads: 4
```

In a script, the value can be accessed via `PARAMS["bowtie_threads"]`.

Undefined configuration values will throw a `ValueError`. To test if a configuration variable exists, use:

```
if 'bowtie_threads' in PARAMS: pass
```

To test, if it is unset, use:

```
if 'bowie_threads' in PARAMS and not PARAMS['botwie_threads']:
   pass
```

4.5.15 Task specific parameters

Task specific parameters can be set by creating a task specific section in the `pipeline.yml`. The task is identified by the output filename. For example, given the following task:

```
@files( '*.fastq', suffix('.fastq'), '.bam')
def mapWithBowtie( infile, outfile ):
   ...
```

and the files `data1.fastq` and `data2.fastq` in the working directory, two output files `data.bam` and `data2.bam` will be created on executing `mapWithBowtie`. Both will use the same parameters. To set parameters specific to the execution of `data1.fastq`, add the following to `pipeline.yml`:

```
data1.fastq:
    bowtie_threads: 16
```

This will set the configuration value `bowtie_threads` to 16 when using the command line substitution method in `pipeline.run()`. To get an task-specific parameter values in a python task, use:

```
@files( '*.fastq', suffix('.fastq'), '.bam')
def mytask( infile, outfile ):
    MY_PARAMS = P.substitute_parameters( locals() )
```

Thus, task specific are implemented generically using the `pipeline.run()` mechanism, but pipeline authors need to explicitely code for track specific parameters.

4.5.16 Using different conda environments

In addition to running a pipeline using your default conda environment, specifying *job_condaenv="<name of conda environment>"* to the P.run() function allows you run the statement using a different conda environment. For example:

```
@follows(mkdir("MultiQC_report.dir"))
@originate("MultiQC_report.dir/multiqc_report.html")
def renderMultiqc(infile):
'''build mulitqc report'''

statement = '''LANG=en_GB.UTF-8 multiqc . -f;
               mv multiqc_report.html MultiQC_report.dir/'''

P.run(statement, job_condaenv="multiqc")
```

This can be extremely useful when you have python 2 only code but are running in a python 3 environment. Or more importantly, when you have conflicting dependancies in software and you need to seperate them out into two different environments.xs

4.6 Setting run parameters

Our workflows are executed using defaults that specify parameters for setting requirements for memory, threads, environment, e.c.t. Each of these parameters can be modified within the pipeline.

4.6.1 Modifiable run parameters

- *job_memory*: Number of slots (threads/cores/CPU) to use for the task. Default: "4G"
- *job_total_memory*: Total memory to use for a job.
- *to_cluster*: Send the job to the cluster. Default: True
- *without_cluster*: When this is set to True the job is ran locally. Default: False
- *cluster_memory_ulimit*: Restrict virtual memory. Default: False
- *job_condaenv*: Name of the conda environment to use for each job. Default: will use the one specified in bashrc
- *job_array*: If set True, run statement as an array job. Job_array should be tuple with start, end, and increment. Default: False

4.6.2 Specifying parameters to job

Parameters can be set within a pipeline task as follows:

```
@transform( '*.unsorted', suffix('.unsorted'), '.sorted')
def sortFile( infile, outfile ):

  statement = '''sort -t %(tmpdir)s %(infile)s > %(outfile)s'''

  P.run(statement,
        job_condaenv="sort_environment",
        job_memory=30G,
        job_threads=2,
        without_cluster = False,
        job_total_memory = 50G)
```

4.7 Writing a workflow- Tutorial

The explicit aim of cgat-core is to allow users to quickly and easily build their own computational pipelines that will speed up your analysis workflow.

4.7.1 Installation of cgat-core

In order to begin writing a pipeline you will need to install the cgat-core code (see *Installation*) for installation instructions.

4.7.2 Tutorial start

Setting up the pipleine

1. First navigate to a directory where you want to start building your code:

```
mkdir test && cd test && mkdir configuration && touch configuration/pipeline.yml &&␣
↪touch pipeline_test.py && touch ModuleTest.py
```

This will create a directory called test in the current directory with the following layout:

```
|-- configuration
|   `-- pipeline.yml
`-- pipeline_test.py
 -- ModuleTest.py
```

The layout has the following components:

pipeline_test.py
This is the file that will contain all of the ruffus workflows, the file needs the format pipeline_<name>.py

test/
Directory containing the configuration yml file. The directory needs to be named the same as the pipeline_<name>.py file. This folder will contain the *pipeline.yml* configuration file.

ModuleTest.py
This file will contain functions that will be imported into the main ruffus workflow file, pipeline_test.py

2. View the source code within pipeline_test.py

This is where the ruffus tasks will be written. The code begins with a doc string detailing the pipeline functionality. You should use this section to document your pipeline.

```
'''This pipeline is a test and this is where the documentation goes '''
```

The pipeline then needs a few utility functions to help with executing the pipeline.

Import statements You will need to import ruffus and cgatcore utilities

```
from ruffus import *
import cgatcore.experiment as E
from cgatcore import pipeline as P
```

Importing ruffus allows ruffus decorators to be used within out pipeline

Importing experiment from cgatcore is a module that contains ultility functions for argument parsion, logging and record keeping within scripts.

Importing pipeline from cgatcore allows users to run utility functions for interfacing with CGAT ruffus pipelines with an HPC cluster, uploading data to a database, provides paramaterisation and more.

You'll also need python modules:

```
import os
import sys
```

Config parser: This code helps with parsing the pipeline.yml file:

```
# load options from the config file
PARAMS = P.get_parameters(
    ["%s/pipeline.yml" % os.path.splitext(__file__)[0],
    "../pipeline.yml",
    "pipeline.yml"])
```

Pipeline configuration: We will add configurable variables to our pipeline.yml file so that we can modify the output of out pipeline. With *pipeline.yml* open, copy and paste the following into the file.

```
database:
   name: "csvdb"
```

When you come to run the pipeline the configuration variables (in this case csvdb) can be accessed in the pipeline by PARAMS["database_name"].

Database connection: This code helps with connecting to a sqlite database:

```
def connect():
     '''utility function to connect to database.

     Use this method to connect to the pipeline database.
     Additional databases can be attached here as well.

     Returns an sqlite3 database handle.
     '''

     dbh = sqlite3.connect(PARAMS["database_name"])

     return dbh
```

Commandline parser: This bit of code allows pipeline to parse arguments.

```
def main(argv=None):
    if argv is None:
        argv = sys.argv
    P.main(argv)


if __name__ == "__main__":
    sys.exit(P.main(sys.argv))
```

Running test pipeline

You now have the bare bones layout of the pipeline and you now need code to execute. Below you will find example code that you can copy and paste into your pipeline_test.py file. The code includes two ruffus **@transform** tasks that parse the pipeline.yml. The first function called `countWords` is then called which contains a statement that counts the number of words in the file. The statement is then ran using `P.run()` function.

The second ruffus **@transform** function called `loadWordCounts` takes as an input the output of the function countWords and loads the number of words to a sqlite database using `P.load()`.

The third `def full()` function is a dummy task that is written to run the whole pipeline. It has an **@follows** function that takes the `loadWordCounts` function. This helps complete the pipeline chain and the pipeline can be ran with the tak name full to execute the whole workflow.

The following code should be pasted just before the **Commandline parser** arguments and after the **database connection** code.

```
# ---------------------------------------------------
# Specific pipeline tasks
@transform("pipeline.yml",
           regex("(.*)\.(.*)"),
           r"\1.counts")
def countWords(infile, outfile):
    '''count the number of words in the pipeline configuration files.'''

    # the command line statement we want to execute
    statement = '''awk 'BEGIN { printf("word\\tfreq\\n"); }
    {for (i = 1; i <= NF; i++) freq[$i]++}
    END { for (word in freq) printf "%%s\\t%%d\\n", word, freq[word] }'
    < %(infile)s > %(outfile)s'''

    # execute command in variable statement.
    #
    # The command will be sent to the cluster.  The statement will be
    # interpolated with any options that are defined in in the
    # configuration files or variable that are declared in the calling
    # function.  For example, %(infile)s will we substituted with the
    # contents of the variable "infile".
    P.run(statement)


@transform(countWords,
           suffix(".counts"),
           "_counts.load")
def loadWordCounts(infile, outfile):
    '''load results of word counting into database.'''
    P.load(infile, outfile, "--add-index=word")

# ---------------------------------------------------
# Generic pipeline tasks
@follows(loadWordCounts)
def full():
    pass
```

To run the pipeline navigate to the working directory and then run the pipeline.

```
python /location/to/code/pipeline_test.py config
python /location/to/code/pipeline_test.py show full -v 5
```

This will place the pipeline.yml in the folder. Then run

```
python /location/to/code/pipeline_test.py  make full -v5 --local
```

The pipeline will then execute and count the words in the yml file.

Modifying the test pipeline to build your own workflows

The next step is to modify the basic code in the pipeline to fit your particular NGS workflow needs. For example, say we wanted to convert a sam file into a bam file then perform flag stats on that output bam file. The code and layout that we just wrote can be easily modified to perform this. We would remove all of the code from the specific pipeline tasks and write our own.

The pipeline will have two steps: 1. Identify all sam files and convert to a bam file. 2. Take the output of step 1 and then perform flagstats on that bam file.

The first step would involve writing a function to identify all *sam* files in a *data.dir/* directory. This first function would accept a sam file then use samtools view to convert it to a bam file. Therefore, we would require an `@transform` function.

The second function would then take the output of the first function, perform samtools flagstat and then output the results as a flat .txt file. Again, an `@transform` function is required to track the input and outputs.

This would be written as follows:

```
@transform("data.dir/*.sam",
           regex("data.dir/(\S+).sam"),
           r"\1.bam")
def bamConvert(infile, outfile):
    'convert a sam file into a bam file using samtools view'

    statement = ''' samtools view -bT /ifs/mirror/genomes/plain/hg19.fasta
                    %(infile)s > %(outfile)s'''

    P.run(statement)

@transform(bamConvert,
           suffix(".bam"),
           "_flagstats.txt")
def bamFlagstats(infile, outfile):
    'perform flagstats on a bam file'

    statement = '''samtools flagstat %(infile)s > %(outfile)s'''

    P.run(statement)
```

To run the pipeline:

```
python /path/to/file/pipeline_test.py make full -v5
```

The bam files and flagstats outputs should then be generated.

Parameterising the code using the .yml file

Having written the basic function of our pipleine, as a philosophy, we try and avoid any hard coded parameters.

This means that any variables can be easily modified by the user without having to modify any code.

Looking at the code above, the hard coded link to the hg19.fasta file can be added as a customisable parameter. This could allow the user to specify any fasta file depending on the genome build used to map and generate the bam file.

In order to do this the `pipeline.yml` file needs to be modified. This can be performed in the following way:

Configuration values are accessible via the `PARAMS` variable. The `PARAMS` variable is a dictionary mapping configuration parameters to values. Keys are in the format `section_parameter`. For example, the key `genome_fasta` will provide the configuration value of:

```
genome:
    fasta: /ifs/mirror/genomes/plain/hg19.fasta
```

In the pipeline.yml, add the above code to the file. In the pipeline_test.py code the value can be accessed via `PARAMS["genome_fasta"]`.

Therefore the code we wrote before for parsing bam files can be modified to

```
@transform("data.dir/*.sam",
           regex("data.dir/(\S+).sam"),
           r"\1.bam")
def bamConvert(infile, outfile):
    'convert a sam file into a bam file using samtools view'

    genome_fasta = PARAMS["genome_fasta"]

    statement = ''' samtools view -bT  %(genome_fasta)s
                    %(infile)s > %(outfile)s'''

    P.run(statement)

@transform(bamConvert,
           suffix(".bam"),
           "_flagstats.txt")
def bamFlagstats(infile, outfile):
    'perform flagstats on a bam file'

    statement = '''samtools flagstat %(infile)s > %(outfile)s'''

    P.run(statement)
```

Running the code again should generate the same output. However, if you had bam files that came from a different genome build then the parameter in the yml file can be modified easily, the output files deleted and the pipeline ran using the new configuration values.

4.8 AWS S3 Storage

This section described how to interact with amazon cloud simple remote storage (S3). In order to interact with the S3 resource we use the boto3 SDK.

This is a work in progress and we would really like feedback for extra features or if there are any bugs then please report them as issues on github.

4.8.1 Setting up credentials

In order to use the AWS remote feature you will need to configure your credentials (The access key and secret key). You can set up these credentials by adding the keys as environment variables in a file *~/.aws/credentials* as detailed in the boto3 configuration page. In brief you will need to add the keys as follows:

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY
aws_secret_access_key = YOUR_SECRET_KEY
```

These access keys can be found within your S3 AWS console and you can access them by following these steps: * Log in to your AWS Management Console. * Click on your user name at the top right of the page. * Click My Security Credentials. * Click Users in left hand menu and select a user. * Click the Security credentials tab. * YOUR_ACCESS_KEY is located in the Access key section

If you have lost YOUR_SECRET_KEY then you will need to Create a new access key, please see AWS documentation on how to do this. Please not that every 90 days AWS will rotate your access keys.

In additon, you may also want to configure the default region:

```
[default]
region=us-east-1
```

Once configuration variables have been created then you are ready to interact with the S3 storage.

4.8.2 Download from AWS S3

Using remote files with AWS can be acieved easily by using *download*, *upload* and *delete_file* functions that are written into a RemoteClass.

Firstly you will need to initiate the class as follows:

```
from cgatcore.remote.aws import *
S3 = S3RemoteObject()
```

In order to download a file and use it within the decorator you can follows the example:

```
@transform(S3.download('aws-test-boto',"pipeline.yml", "./pipeline.yml"),
        regex("(.*)\.(.*)"),
        r"\1.counts")
```

This will download the file *pipeline.yml* in the AWS bucket *aws-test-boto* locally to *./pipeline.yml* and it will be picked up by the decoratory function as normal.

4.8.3 Upload to AWS S3

In order to upload files to aws S3 you simply need to run:

```
S3.upload('aws-test-boto',"pipeline2.yml", "./pipeline.yml")
```

This will upload to the *aws-test-boto* S3 bucket the *./pipeline.yml* file and it will be saved as *pipeline2.yml* in that bucket.

4.8.4 Delete file from AWS S3

In order to delete a file from the AWS S3 bucket then you simply run:

```
S3.delete_file('aws-test-boto',"pipeline2.yml")
```

This will delete the *pipeline2.yml* file from the *aws-test-boto* bucket.

4.8.5 Functional example

As a simple example, the following one function pipeline demonstrates the way you can interact with AWS S3:

```
from ruffus import *
import sys
import os
import cgatcore.experiment as E
from cgatcore import pipeline as P
from cgatcore.remote.aws import *

# load options from the config file
PARAMS = P.get_parameters(
    ["%s/pipeline.yml" % os.path.splitext(__file__)[0],
     "../pipeline.yml",
     "pipeline.yml"])

S3 = S3RemoteObject()


@transform(S3.download('aws-test-boto',"pipeline.yml", "./pipeline.yml"),
       regex("(.*)\.(.*)"),
       r"\1.counts")
def countWords(infile, outfile):
    '''count the number of words in the pipeline configuration files.'''

    # Upload file to S3
    S3.upload('aws-test-boto',"pipeline2.yml", "/ifs/projects/adam/test_remote/data/
↪pipeline.yml")

    # the command line statement we want to execute
    statement = '''awk 'BEGIN { printf("word\\tfreq\\n"); }
    {for (i = 1; i <= NF; i++) freq[$i]++}
    END { for (word in freq) printf "%%s\\t%%d\\n", word, freq[word] }'
    < %(infile)s > %(outfile)s'''

```

(continues on next page)

(continued from previous page)

```
    P.run(statement)

     # Delete file from S3
     S3.delete_file('aws-test-boto',"pipeline2.yml")

     @follows(countWords)
     def full():
         pass
```

4.9 Google storage

This section describes how to interact with the google cloud storage bucket and blob (files). In order to interact with the cloud resource we use the *google.cloud* API for python.

This is a work in progress and we would really like feedback for extra features or if there are any bugs then please report them as issues on github.

4.9.1 Setting up credentials

In order to use google cloud storage feature you will need to conigure your credentials. This is quite easy with the *gcloud* tool. This tool is ran before exectuing a workflow in the following way:

```
gcloud auth application-default login
```

This sets up a JSON file with all of the credentiaals on your home folder, usually in the file *.config/gcloud/application_default_credentials.json*

Next you will also need to tell the API which project you are using. Projects are usually set in the google console and all have a unique ID. This ID needs to be passed into cgat-core.

This can be achieved in the following ways:

- passing project_id into the JASON file:

```
{
"client_id": "764086051850-6qr4p6gpi6hn506pt8ejuq83di341hur.apps.googleusercontent.
↪com",
"client_secret": "d-FL95Q19q7MQmFpd7hHD0Ty",
"refresh_token": "1/d8JxxulX84r3jiJVlt-xMrpDLcIp3RHuxLHtieDu8uA",
"type": "authorized_user",
"project_id": "extended-cache-163811"
}
```

- project_id can be set in the *bashrc*:

```
export GCLOUD_PROJECT=extended-cache-163811
```

4.9.2 Download from google storage

Using remote files with google cloud can be acieved easily by using *download*, *upload* and *delete_file* functions that are written into a RemoteClass.

Firstly you will need to initiate the class as follows:

```
from cgatcore.remote.google_cloud import *
GC = GCRemoteObject()
```

In order to download a file and use it within the decorator you can follows the example:

```
@transform(GC.download('gc-test',"pipeline.yml", "./pipeline.yml"),
       regex("(.*)\.(.*)"),
       r"\1.counts")
```

This will download the file *pipeline.yml* from the google cloud bucket *gc-test* locally to *./pipeline.yml* and it will be picked up by the decoratory function as normal.

4.9.3 Upload to google cloud

In order to upload files to google cloud you simply need to run:

```
GC.upload('gc-test',"pipeline2.yml", "./pipeline.yml")
```

This will upload to the *gc-test* google cloud bucket the *./pipeline.yml* file and it will be saved as *pipeline2.yml* in that bucket.

4.9.4 Delete file from AWS S3

In order to delete a file from the AWS S3 bucket then you simply run:

```
S3.delete_file('aws-test-boto',"pipeline2.yml")
```

This will delete the *pipeline2.yml* file from the *aws-test-boto* bucket.

4.9.5 Functional example

As a simple example, the following one function pipeline demonstrates the way you can interact with the google cloud:

```
from ruffus import *
import sys
import os
import cgatcore.experiment as E
from cgatcore import pipeline as P
from cgatcore.remote.google_cloud import *

# load options from the config file
PARAMS = P.get_parameters(
    ["%s/pipeline.yml" % os.path.splitext(__file__)[0],
     "../pipeline.yml",
     "pipeline.yml"])
```

(continues on next page)

(continued from previous page)

```

GC = GCRemoteObject()


@transform(GC.download('gc-test',"pipeline.yml", "./pipeline.yml"),
       regex("(.*)\.(.*)"),
       r"\1.counts")
def countWords(infile, outfile):
    '''count the number of words in the pipeline configuration files.'''

    # Upload file to google cloud
    GC.upload('gc-test',"pipeline2.yml", "/ifs/projects/adam/test_remote/data/pipeline.
↪yml")

    # the command line statement we want to execute
    statement = '''awk 'BEGIN { printf("word\\tfreq\\n"); }
    {for (i = 1; i <= NF; i++) freq[$i]++}
    END { for (word in freq) printf "%%s\\t%%d\\n", word, freq[word] }'
    < %(infile)s > %(outfile)s'''

     P.run(statement)

      # Delete file from google cloud
      GC.delete_file('gc-test',"pipeline2.yml")

      @follows(countWords)
      def full():
          pass
```

4.10 Microsoft Azure Storage

This section describes how to interact with Microsoft Azure cloud storage. In order to interact with the Azure cloud storage resource we use the azure SDK.

Like all of our remote connection functionality, this is a work in progress and we are currently in the process of adding extra features. If you have bug reports or comments then please raise them as an issue on github

4.10.1 Setting up credentials

Unlike other remote access providers, the credentials are set up by passing them directly into the initial class as variables as follows:

```
Azure = AzureRemoteObject(account_name = "firstaccount", account_key =
↪"jbiuebcjubncjklncjkln........")
```

These access keys can be found in the Azure portal and locating the storage account. In the settings of the storage account there is a selection "Access keys". The account name and access keys are listed here.

4.10.2 Download from Azure

Using remote files with Azure can be achieved easily by using *download*, *upload* and *delete_file* functions that are written into a RemoteClass.

Firstly you will need to initiate the class as follows:

```
from cgatcore.remote.azure import *
Azure = AzureRemoteObject(account_name = "firstaccount", account_key =
↪"jbiuebcjubncjklncjkln........")
```

In order to download a file and use it within the decorator you can follows the example:

```
@transform(Azure.download('test-azure',"pipeline.yml", "./pipeline.yml"),
       regex("(.*)\.(.*)"),
       r"\1.counts")
```

This will download the file *pipeline.yml* in the Azure container *test-azure* locally to *./pipeline.yml* and it will be picked up by the decoratory function as normal.

4.10.3 Upload to Azure

In order to upload files to Azure you simply need to run:

```
Azure.upload('test-azure',"pipeline2.yml", "./pipeline.yml")
```

This will upload to the *test-azure* Azure container the *./pipeline.yml* file and it will be saved as *pipeline2.yml* in that bucket.

4.10.4 Delete file from Azure

In order to delete a file from the Azure container then you simply run:

```
Azure.delete_file('test-azure',"pipeline2.yml")
```

This will delete the *pipeline2.yml* file from the *test-azure* container.

4.10.5 Functional example

As a simple example, the following one function pipeline demonstrates the way you can interact with AWS S3:

```
from ruffus import *
import sys
import os
import cgatcore.experiment as E
from cgatcore import pipeline as P
from cgatcore.remote.azure import *

# load options from the config file
PARAMS = P.get_parameters(
    ["%s/pipeline.yml" % os.path.splitext(__file__)[0],
     "../pipeline.yml",
```

(continues on next page)

(continued from previous page)

```
     "pipeline.yml"])

Azure = AzureRemoteObject(account_name = "firstaccount", account_key =
↪"jbiuebcjubncjklncjkln........")


@transform(Azure.download('test-azure',"pipeline.yml", "./pipeline.yml"),
       regex("(.*)\.(.*)"),
       r"\1.counts")
def countWords(infile, outfile):
    '''count the number of words in the pipeline configuration files.'''

    # Upload file to Azure
    Azure.upload('test-azure',"pipeline2.yml", "/ifs/projects/adam/test_remote/data/
↪pipeline.yml")

    # the command line statement we want to execute
    statement = '''awk 'BEGIN { printf("word\\tfreq\\n"); }
    {for (i = 1; i <= NF; i++) freq[$i]++}
    END { for (word in freq) printf "%%s\\t%%d\\n", word, freq[word] }'
    < %(infile)s > %(outfile)s'''

     P.run(statement)

      # Delete file from Azure
      Azure.delete_file('test-azure',"pipeline2.yml")

      @follows(countWords)
      def full():
          pass
```

4.11 pipeline modules

4.12 Core helper modules

Add links to the other core documentation

4.13 Developers

The following individuals are the main developers of the cgatcore

Andreas Heger

Adam Cribbs

Sebastian Luna Valero

Hania Pavlou

David Sims

Charlotte George

Tom Smith

Ian Sudbery

Jakub Scaber

Mike Morgan

Katy Brown

Nick Ilott

Jethro Johnson

Katherine Fawcett

Steven Sansom

Antonio Berlanga

4.14 Contributing

Contributions are very much encouraged and we greatly appreciate the time and effort people make to help maintain and support out tools. Every contribution helps, please dont be shy, we dont bite.

You can contribute to the development of our software in a number of different ways:

4.14.1 Reporting bug fixes

Bugs are annoying and reporting them will help us to fix your issue.

Bugs can be reported using the issue section in github

When reporting issues, please include:

- Steps in your code/command that led to the bug so it can be reproduced.
- The error message from the log message.
- Any other helpful info, such as the system/cluster engine or version information.

4.14.2 Proposing a new feature/enhancement

If you wish to contribute a new feature to the CGAT-core repository then the best way is to raise this as an issue and label it as an enhancement in github

If you propose a new feature then please:

- Explain how your enhancement will work
- Describe as best as you can how you plan to implement this.
- If you dont think you have the necessary skills to implement this on your own then please say and we will try our best to help (or implement this for you). However, please be aware that this is a community developed software and our volunteers have other jobs. Therefore, we may not be able to work as fast as you hoped.

4.14.3 Pull Request Guidelines

Why not contribute to our project, its a great way of making the project better, your help is always welcome. We follow the fork/pull request model. To update our documentation, fix bugs or add extra enhancements you will need to create a pull request through github.

To create a pull request perform these steps:

1. Create a github account.
2. Create a personal fork of the project on github.
3. Clone the fork onto your local machine. Your remote repo on github is called `origin`.
4. Add the orginal repository as a remote called `upstream`.
5. If you made the fork a while ago then please make sure you `git pull upstream` to keep your repository up to date
6. Create a new branch to work on! We usually name our branches with capital first and last followed by a dash and something unique. For example: `git checkout -b AC-new_doc`.
7. Impliment your fix/enhancement and make sure your code is effectively documented.
8. Our code has tests and these will be ran when a pull request is submitted, however you can run our tests before you make the pull request, we have a number written in the `tests/` directory. To run all tests, type `pytest --pep8 tests`
9. Add or change our documentation in the `docs/` directory.
10. Squash all of your commits into a single commit with gits interactive rebase.
11. Push your branch to your fork on github `git push origin`
12. From your fork in github.com, open a pull request in the correct branch.
13. . . . This is where someone will review your changes and modify them or approve them . . .
14. Once the pull request is approved and merged you can pull the changes from the `upstream` to your local repo and delete your branch.

Note: Always write your commit messages in the present tense. Your commit messages should describe what the commit does to the code and not what you did to the code.

4.15 Citing and Citations

cgatcore has been developed over the past 10 years and as such has been used in a number of previously published scientific artciles.

4.15.1 Citing cgatcore

When using cgatcore for a publication, **please cite the following article** in you paper:

ADD CITATION HERE

4.15.2 More references

A number of publications that have used CGAT-developer tools are listed below, however this list is not an exhastive list:

A ChIP-seq defined genome-wide map of vitamin D receptor binding: associations with disease and evolution SV Ramagopalan, A Heger, AJ Berlanga, NJ Maugeri, MR Lincoln, … Genome research 20 (10), 1352-1360 2010

Sequencing depth and coverage: key considerations in genomic analyses D Sims, I Sudbery, NE Ilott, A Heger, CP Ponting Nature Reviews Genetics 15 (2), 121 2014

KDM2B links the Polycomb Repressive Complex 1 (PRC1) to recognition of CpG islands AM Farcas, NP Blackledge, I Sudbery, HK Long, JF McGouran, NR Rose, … elife 2012

Targeting polycomb to pericentric heterochromatin in embryonic stem cells reveals a role for H2AK119u1 in PRC2 recruitment S Cooper, M Dienstbier, R Hassan, L Schermelleh, J Sharif, … Cell reports 7 (5), 1456-1470 2014

Long non-coding RNAs and enhancer RNAs regulate the lipopolysaccharide-induced inflammatory response in human monocytes NE Ilott, JA Heward, B Roux, E Tsitsiou, PS Fenwick, L Lenzi, I Goodhead, … Nature communications 5, 3979 2014

Population and single-cell genomics reveal the Aire dependency, relief from Polycomb silencing, and distribution of self-antigen expression in thymic epithelia SN Sansom, N Shikama-Dorn, S Zhanybekova, G Nusspaumer, … Genome research 24 (12), 1918-1931 2014

Epigenetic conservation at gene regulatory elements revealed by non-methylated DNA profiling in seven vertebrates HK Long, D Sims, A Heger, NP Blackledge, C Kutter, ML Wright, … Elife 2 2013

The long non-coding RNA Paupar regulates the expression of both local and distal genes KW Vance, SN Sansom, S Lee, V Chalei, L Kong, SE Cooper, PL Oliver, … The EMBO journal 33 (4), 296-311 2014

A genome-wide association study implicates the APOE locus in nonpathological cognitive ageing G Davies, SE Harris, CA Reynolds, A Payton, HM Knight, DC Liewald, … Molecular Psychiatry 19 (1), 76 2014

Predicting long non-coding RNAs using RNA sequencing NE Ilott, CP Ponting Methods 63 (1), 50-59 2013

Next-generation sequencing of advanced prostate cancer treated with androgen-deprivation therapy P Rajan, IM Sudbery, MEM Villasevil, E Mui, J Fleming, M Davis, I Ahmad, … European urology 66 (1), 32-39 2014

The long non-coding RNA Dali is an epigenetic regulator of neural differentiation V Chalei, SN Sansom, L Kong, S Lee, JF Montiel, KW Vance, CP Ponting Elife 3 2014

GAT: a simulation framework for testing the association of genomic intervals A Heger, C Webber, M Goodson, CP Ponting, G Lunter Bioinformatics 29 (16), 2046-2048 2013

De novo point mutations in patients diagnosed with ataxic cerebral palsy R Parolin Schnekenberg, EM Perkins, JW Miller, WIL Davies, … Brain 138 (7), 1817-1832 2015

SPG7 mutations are a common cause of undiagnosed ataxia G Pfeffer, A Pyle, H Griffin, J Miller, V Wilson, L Turnbull, K Fawcett, … Neurology 84 (11), 1174-1176 2015

CDK9 inhibitors define elongation checkpoints at both ends of RNA polymerase II–transcribed genes C Laitem, J Zaborowska, NF Isa, J Kufs, M Dienstbier, S Murphy Nature Structural and Molecular Biology 22 (5), 396 2015

IRF5: RelA interaction targets inflammatory genes in macrophages DG Saliba, A Heger, HL Eames, S Oikonomopoulos, A Teixeira, K Blazek, … Cell reports 8 (5), 1308-1317 2014

UMI-tools: modeling sequencing errors in Unique Molecular Identifiers to improve quantification accuracy T Smith, A Heger, I Sudbery Genome research 27 (3), 491-499 2017

Long noncoding RNAs in B-cell development and activation TF Brazão, JS Johnson, J Müller, A Heger, CP Ponting, VLJ Tybulewicz Blood 128 (7), e10-e19 2016

CGAT: computational genomics analysis toolkit D Sims, NE Ilott, SN Sansom, IM Sudbery, JS Johnson, KA Fawcett, … Bioinformatics 30 (9), 1290-1291 2014

4.16 FAQs

As our workflow develops we will add frequently asked questions here.

In the meantime please add issues to the github page

4.17 Licence

CGAT-core is an open-source project and we have made the cgat-developers repositor available under the open source permissive free MIT software licence, allowing free and full use of the code for both commercial and non-commercial purposes. A copy of the licence is shown below:

4.17.1 MIT License

INDEX