Contents

Index

Dahak

Dahak is a software suite that integrates state-of-the-art open source tools for metagenomic analyses. Tools in the dahak software suite will perform various steps in metagenomic analysis workflows including data pre-processing, metagenome assembly, taxonomic and functional classification, genome binning, and gene assignment. We aim to deliver the analytical framework as a robust and reliable containerized workflow system, which will be free from dependency, installation, and execution problems typically associated with other open-source bioinformatics solutions. This will maximize the transparency, data provenance (i.e., the process of tracing the origins of data and its movement through the workflow), and reproducibility.

Benchmarking Data

For purposes of benchmarking this project will use the following datasets:

Dataset	Description
Shakya complete	Complete metagenomic dataset from Shakya et al., 2013* containing bacterial and archaeal genomes
Shakya subset 50	50 percent of the reads from Shakya complete
Shakya subset 25	25 percent of the reads from Shakya complete
Shakya subset 10	10 percent of the reads from Shakya complete

*Shakya, M., C. Quince, J. H. Campbell, Z. K. Yang, C. W. Schadt and M. Podar (2013). "Comparative metagenomic and rRNA microbial diversity characterization using archaeal and bacterial synthetic communities." Environ Microbiol 15(6): 1882-1899.

Requirements and Installation

Dahak is not a standalone program, but rather a collection of workflows that are defined in Snakemake files. These workflows utilize Bioconda, Biocontainers, and Docker/Singularity containerization technologies to install and run software for different tasks.

The following software is required to run Dahak workflows:

REQUIRED:

- Python 3
- Snakemake
- Conda
- Singularity or Docker

TARGET PLATFORM:

- (Required) Singularity >= 2.4 (does not require sudo access) or Docker (requires sudo access)
- (Recommended) Ubuntu 16.04 (Xenial)
- (Recommended) Sun Grid Compute Engine

See the Installing page for detailed instructions on installing each of the required components listed above, including Singularity and Docker.

See the Quickstart page for instructions on getting started running dahak workflows with Snakemake.

Workflows

Dahak provides a set of workflow components that all fit together to perform various useful tasks.

See the Running Workflows page for some background on how to run workflows using singularity and snakemake.

See the Quickstart guide if you just want to get up and running with workflows.

Our **target compute system** is a generic cluster running Sun Grid Engine in an HPC environment; all Snakemake files are written for this target system.

List of workflows:

- Read Filtering
- Assembly
- Metagenomic Comparison
- Taxonomic Classification
- Variant Calling
- Functional Inference

To get started with a particular workflow, select it from the navigation menu on the left side of the page.

Parameters and Configuration

See the Parameters and Configuration page for details about controlling how each workflow operates, and how to use parameter presets.

Contributing

Please read CONTRIBUTING.md for details on our code of conduct and the process for submitting pull requests to us.

Contributors

Phillip Brooks1, Charles Reid1, Bruce Budowle2, Chris Grahlmann3, Stephanie L. Guertin3, F. Curtis Hewitt3, Alexander F. Koeppel4, Oana I. Lungu3, Krista L. Ternus3, Stephen D. Turner4,5, C. Titus Brown1

1School of Veterinary Medicine, University of California Davis, Davis, CA, United States of America

2Institute of Applied Genetics, Department of Molecular and Medical Genetics, University of North Texas Health Science Center, Fort Worth, Texas, United States of America

3Signature Science, LLC, Austin, Texas, United States of America

4Department of Public Health Sciences, University of Virginia, Charlottesville, VA, United States of America

5Bioinformatics Core, University of Virginia School of Medicine, Charlottesville, VA, United States of America

See also the list of contributors who participated in this project.

License

Installing

Installation Instructions

Before getting started with Dahak, you will need the following software installed:

REQUIRED:

- Docker or Singularity
- Python 3
- Conda
- Snakemake

The instructions below will help you get started running the software above.

Installing Docker or Singularity

(Recommended) Singularity

Singularity is a tool for running Docker containers in higher security environments (e.g., high performance computing clusters) where permissions are restricted. If you wish to use Docker directly and have root access (e.g., with AWS/cloud machines), see the "Getting Started with Docker" section below.

Installing a stable version of Singularity is recommended. Stable versions can be obtained from Singularity's Releases on Github.

(In an HPC environment, these commands are run by the system administrator.)

```
### Latest
VERSION=2.5.1

### More widely available
VERSION=2.4.6

wget https://github.com/singularityware/singularity/releases/download/$VERSION/singularity-$
tar xvf singularity-$VERSION.tar.gz
cd singularity-$VERSION
./configure --prefix=/usr/local
make
sudo make install
```

To check whether Singularity is installed (in an HPC environment, this command is run by the user), check the version:

```
singularity --version
```

(Optional) Docker

If you wish to follow along with the walkthroughs, which cover the use of Docker containers to run the workflows interactively, you will need to install Docker, which requires root access.

Alternatively, Dahak provides Snakefiles for automating these workflows without requiring root access by using Singularity (see instructions above).

First, update your machine:

```
### Update aptitude and install dependencies
sudo apt-get -y update && sudo apt-get install zlib1g-dev ncurses-dev
```

Next, install Docker:

```
### Install Docker
wget -qO- https://get.docker.com/ | sudo sh
```

```
sudo usermod -aG docker ubuntu
```

Installing Python

To run Dahak workflows, we utilize Snakemake, a Python build tool. To use Snakemake, we must first install Python.

To ensure a universal installation process, we will use pyenv to install the correct versions of Python and Conda.

There are many versions of Python, so your setup may vary. We provide installation instructions using two methods:

- Aptitude-installed Python (requires root)
- Pyenv-managed Python (non-root)

Pyenv is a command-line tool for managing multiple Python versions, including Conda.

(Recommended) Installing Aptitude Python

To install Python with Aptitude:

```
sudo apt-get -y update
sudo apt-get -y install python-pip python-dev
```

(Optional) Installing Pyenv

Start by running the pyenv installer:

```
### Install pyenv
curl -L https://raw.githubusercontent.com/pyenv/pyenv-installer/master/bin/pyenv-installer
```

Add pyenv's bin directory to `$PATH` (should already be in `~/.bash_profile` but just in case):

```
echo 'export PATH="~/.pyenv/bin:$PATH"' >> ~/.bash_profile
source ~/.bash_profile
```

You should see a path with a `.pyenv` directory in it when you type `which pyenv`.

You can now install various Python versions (we will install a version of Conda below):

```
PYVERSION="3.6.5"
PYVERSION="anaconda3-5.1.0"
PYVERSION="miniconda3-4.3.30"
```

To install it:

```
pyenv install $PYVERSION
```

To make it available on the `$PATH` (to make it the version of Python that `python` on the command line points to):

```
pyenv global $PYVERSION
eval "$(pyenv init -)"
```

To make this change permanent, you can add the pyenv init statement to your `~/.bash_profile`:

```
echo 'eval "$(pyenv init -)"' >> ~/.bash_profile
source ~/.bash_profile
```

You should also have a version of `pip` associated with `python`:

```
pip install --upgrade pip
```

Installing Conda

Installing Conda with Python

Once Python is installed, install the latest Miniconda from continuum.io:

```
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
```

Now check to ensure conda is installed and is the correct version:

```
which conda
conda --version
python --version
```

Add the required conda channels:

```
conda update
conda config --add channels r
conda config --add channels defaults
conda config --add channels conda-forge
conda config --add channels bioconda
```

Installing Conda with Pyenv

Once pyenv is installed, use it to install Miniconda 4.3.30 with Python 3:

```
CONDA="miniconda3-4.3.30"
pyenv install $CONDA
pyenv global $CONDA
eval "$(pyenv init -)"
```

You can also add this to your bash profile to ensure that the global pyenv Python version is always the first version of Python on your path:

```
echo 'eval "$(pyenv init -)"' >> ~/.bash_profile
source ~/.bash_profile
```

Now check to make sure you have the pyenv-installed version of conda:

```
which conda
conda --version
python --version
```

Add the required conda channels:

```
conda update
conda config --add channels r
conda config --add channels defaults
conda config --add channels conda-forge
conda config --add channels bioconda
```

Create a Conda Environment

Once conda is installed, you can create a conda environment called `dahak` as detailed in the conda documentation:

```
conda create --name dahak
```

Installing Snakemake

Now install snakemake from the bioconda channel, or install it using pip:

```
conda install -c bioconda snakemake
```

or

```
pip install snakemake
```

Note that this pip will correspond to the version of Python and Conda that are on the path.

Finally, install the Open Science Framework CLI client using pip:

```
pip install --user osfclient
```

Workflows Overview

Metagenomic Workflows

When analyzing metagenomic data, different workflows are broken down into atomic operations. Each directory here corresponds to an atomic operation (a workflow component).

A flowchart illustrating how each workflow component fits together with tools into the overall process is included below:

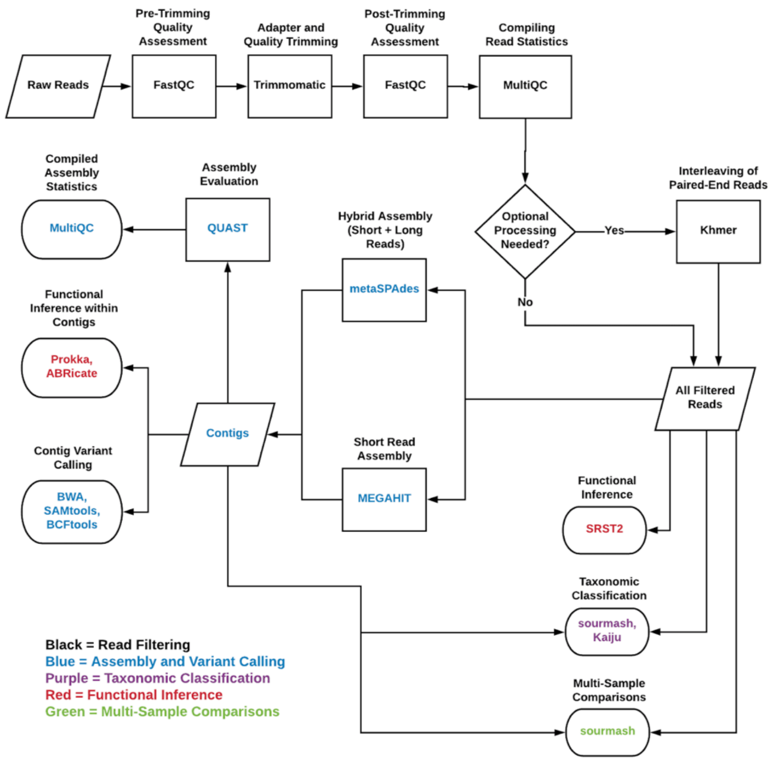

Pre-Trimming Quality Assessment
Adapter and Quality Trimming
Post-Trimming Quality Assessment
Compiling Read Statistics
Raw Reads
FastQC
Trimmomatic
FastQC
MultiQC
Compiled Assembly Statistics
Assembly Evaluation
MultiQC
QUAST
Hybrid Assembly (Short + Long Reads)
metaSPAdes
Optional Processing Needed?
Yes
No
Interleaving of Paired-End Reads
Khmer
Functional Inference within Contigs
Prokka, ABRicate
Contigs
All Filtered Reads
Contig Variant Calling
BWA, SAMtools, BCFtools
Short Read Assembly
MEGAHIT
Functional Inference
SRST2
Taxonomic Classification
sourmash, Kaiju
Black = Read Filtering
Blue = Assembly and Variant Calling
Purple = Taxonomic Classification
Red = Functional Inference
Green = Multi-Sample Comparisons
Multi-Sample Comparisons
sourmash

Figure 1: workflow flowchart

Each workflow has its own Snakefile. The Snakefile is composed of a list of simple rules that specify how an input file is turned into an output file.

Dahak Workflows

Dahak is designed to chain together tools for various tasks. Each task is accomplished by a component of a workflow, with some components being re-used for multiple tasks. This document describes the components that compose each workflow.

What’s Here?

The following workflows are required deliverables:

- Taxonomic characterization of bulk metagenome data sets with the sourmash tool against public and private reference databases;
- Assembly-based approaches to give higher-confidence gene identity assignment than raw read assignment alone;
- MinHash-based taxonomic description of data sets;
- Full-set and marker gene analysis of hybrid assembly/read collections to characterize taxonomic content;
- Full-set gene analysis of hybrid assembly/read collections to characterize functional content;
- Taxonomic and functional analysis performed on reads left out of the assembly;
- Rapid k-mer-based ordination analyses of many samples to provide sample groupings and identify potential outliers; and
- Interactive Jupyter notebooks for interpretation of results.

Workflow Diagrams

In the following diagrams, the colors denote the following:

- Blue - metagenome assembly, alignment, variant calling
- Red - contig annotation and gene assignment
- Purple -taxonomic and functional analysis of reads
- Green - sample comparison

Terminal software/outputs are denoted by ovals.

Dataset Construction

See `/dataset_construction` directory.

Constructs SBTs from external genomic databases and saves them to disk so that they can then be shared and loaded.

Read Filtering and Quality Assessment of Datasets

The read filtering step consists of processing raw reads from a sequencer, such as discarding reads with a high uncertainty value or trimming off adapters.

Tools like Fastqc and Trimmomatic will perform this filtering process for the sequencer's reads.

See Read Filtering Snakemake page for instructions on using this workflow.

Also see the `read_filtering/` directory.

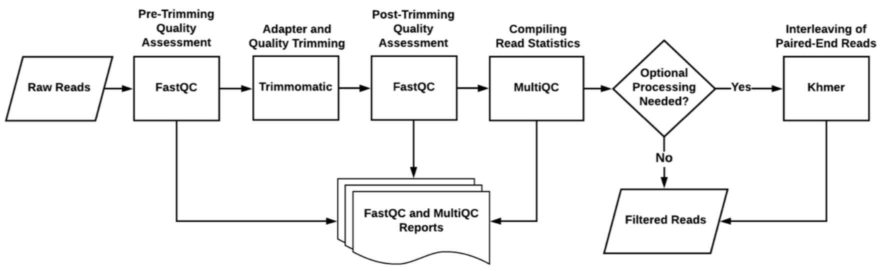

Pre-Trimming Quality Assessment
Adapter and Quality Trimming
Post-Trimming Quality Assessment
Compiling Read Statistics
Interleaving of Paired-End Reads
Raw Reads
FastQC
Trimmomatic
FastQC
MultiQC
Optional Processing Needed?
Yes
Khmer
No
FastQC and MultiQC Reports
Filtered Reads

Figure 2: Quality assessment

Taxonomic Classification Using Custom Database

See Taxonomic Classification Snakemake page for instructions on using this workflow.

Also see `taxonomic_classification/` directory.

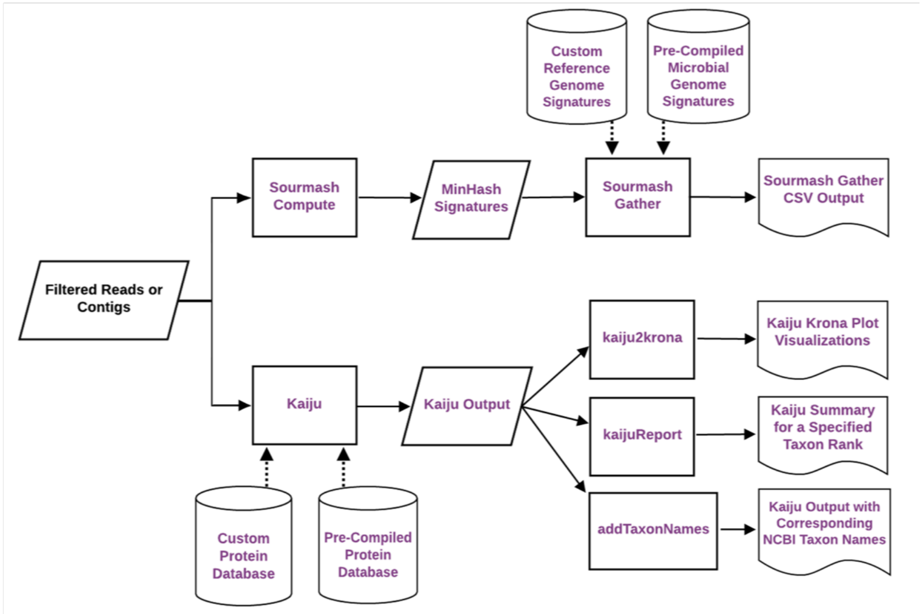

Custom Reference Genome Signatures
Pre-Compiled Microbial Genome Signatures
Sourmash Compute
MinHash Signatures
Sourmash Gather
Sourmash Gather CSV Output
Filtered Reads or Contigs
kaiju2krona
Kaiju Krona Plot Visualizations
Kaiju
Kaiju Output
kaijuReport
Kaiju Summary for a Specified Taxon Rank
addTaxonNames
Kaiju Output with Corresponding NCBI Taxon Names
Custom Protein Database
Pre-Compiled Protein Database

Figure 3: Taxonomic classification

Assembly

The assembly step consists of software to determine the proper order of the reads, and assemble the genome. The assembly tool may use short reads (~350 or fewer reads), or it may use long reads (>1000 reads).

Reads are assembled in order into contigs (chunks of contiguous reads). The contigs are themselves assembled into scaffolds that consist of several contigs.

The Spades tool can handle short or long reads, while the Megahit tool works better for short reads. Pandaseq can merge overlapping reads. Metaquast gives assembly statistics that can help evaluate the assembly (how long, number of fragments, number of contigs, number of scaffolds, etc.).

Typically 30-40% of the reads can be fingerprinted by the assembler.

See Assembly Workflow Snakemake page for instructions on using this workflow.

Also see `assembly/` directory.

Abundance Estimation and Variance Calling

See Variance Calling Workflow Snakemake page for instructions on using this workflow.

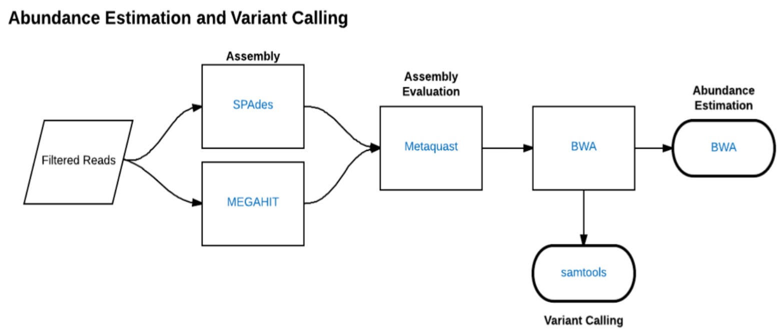


Figure 4: Abundance estimation and variance calling

Sub-Element Identification

See Sub-Element ID Snakemake (*in progress*) for instructions on using this workflow.

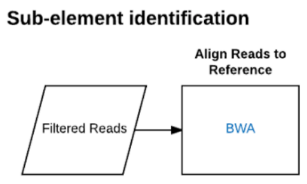


Figure 5: Sub-element ID

Functional Inference

Once the assembly step has been completed, the assembly can be analyzed and annotated using external databases. Prokka is a tool for functional annotation of contigs.

Variant calling searches for common variants of a given gene. Variants are obtained by changing a few genes in an existing genome.

ShotMap was originally used for this step, but was replaced by Miphaser.

See Functional Inference Snakemake page for instructions on using this workflow.

Also see the `functional_inference/` directory.

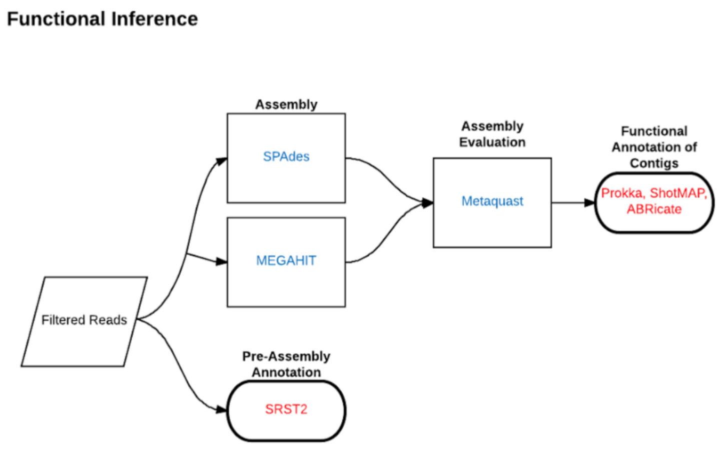


Figure 6: Functional inference

Sample Comparison

Operating at the level of k-mers (representations of the reads), the comparison step is taking the reads that were not fingerprinted by the assembler and seeing if they match genomes of other organisms.

The tool used for comparison is sourmash.

See Sample Comparison Snakemake for instructions on using this workflow.

Also see the `comparison/` directory.

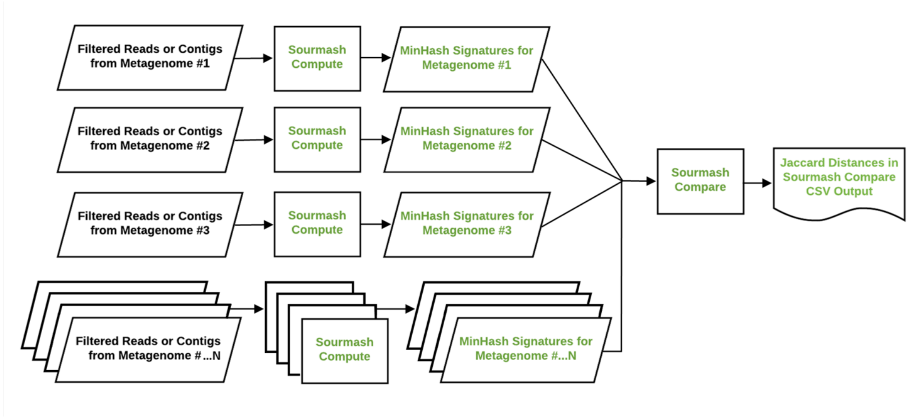

Filtered Reads or Contigs from Metagenome #1
Sourmash Compute
MinHash Signatures for Metagenome #1
Filtered Reads or Contigs from Metagenome #2
Sourmash Compute
MinHash Signatures for Metagenome #2
Filtered Reads or Contigs from Metagenome #3
Sourmash Compute
MinHash Signatures for Metagenome #3
Filtered Reads or Contigs from Metagenome #...N
Sourmash Compute
MinHash Signatures for Metagenome #...N
Sourmash Compare
Jaccard Distances in Sourmash Compare CSV Output

Figure 7: Sample comparison

Running Workflows

Running Workflows

A flowchart illustrating how each workflow component fits together with tools into the overall process is included below:

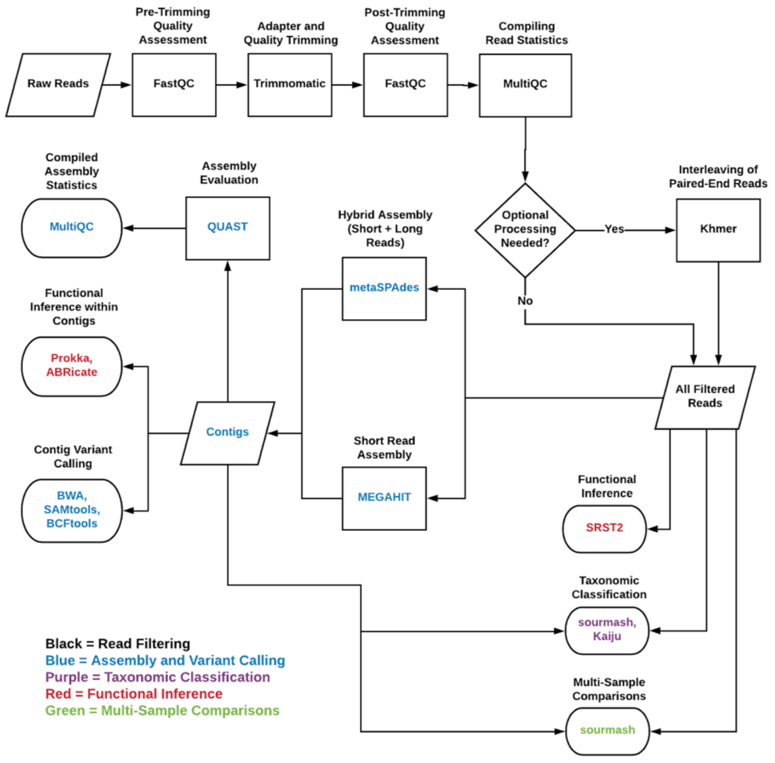
Dahak workflows are run from the command line using Snakemake, a Python package that provides similar capabilities to GNU make. Each workflow consists of a set of Snakemake rules.

Snakemake is a Python program that assembles and runs tasks using a task graph approach. See Installing for instructions on how to install it.

Dahak workflows benefit directly from Snakemake's rich feature set and capabilities. There is an extensive documentation page on executing Snakemake, and its command line options. There are other projects demonstrating ways of creating snakemake-profiles, or platform-specific configuration profiles.

How To Run Workflows

Generally, Snakemake is called by passing command line flags and the name of a target file or rule name:



Raw Reads
Pre-Trimming Quality Assessment
FastQC
Adapter and Quality Trimming
Trimmomatic
Post-Trimming Quality Assessment
FastQC
Compiling Read Statistics
MultiQC
Optional Processing Needed?
Yes
No
Interleaving of Paired-End Reads
Khmer
All Filtered Reads
Hybrid Assembly (Short + Long Reads)
metaSPAdes
Short Read Assembly
MEGAHIT
Contigs
Assembly Evaluation
QUAST
Compiled Assembly Statistics
MultiQC
Functional Inference within Contigs
Prokka, ABRicate
Contig Variant Calling
BWA, SAMtools, BCFtools
Functional Inference
SRST2
Taxonomic Classification
sourmash, Kaiju
Multi-Sample Comparisons
sourmash
Black = Read Filtering
Blue = Assembly and Variant Calling
Purple = Taxonomic Classification
Red = Functional Inference
Green = Multi-Sample Comparisons

Figure 8: workflow flowchart

```
snakemake [FLAGS] <target>
```

What targets are available?

Targets for each workflow are listed on the respective "Snakemake Rules" page for that workflow (see left side navigation menu).

There are two types of targets defined:

Target Files: The user can ask Snakemake to generate a particular file, and Snakemake will dynamically determine the rules that are required to generate the requested file. Snakemake uses a dependency graph to determine what rules to run to generate the file.

Build Rules: There are rules that do not themselves do anything but that trigger all of the rules in a given workflow. (The build rules work by assembling filenames and passing target filenames to Snakemake.)

What targets should I use?

Users should use the build rules to trigger workflows.

The build rules require workflow configuration details to be set using Snakemake's configuration dictionary. See the Snakemake Configuration page for details.

Each workflow has a set of "build rules" that will trigger rules for a given workflow or portion of a workflow. Available build rules for each workflow are listed on the respective "Snakemake Rules" page for that workflow (see left side navigation menu).

The build rules require some information about which read files to run the workflow on; the information required is covered on each "Snakemake Rules" page.

The Quick Start covers some examples.

How do I specify workflow parameters?

Workflow parameters are specified by passing a JSON configuration file to Snakemake.

The default workflow parameter values are set in `default_workflowparams.settings`. Any of these values can be overridden using a custom JSON file, as described above and on the Workflow Configuration page.

For example, to override the default version of trimmomatic (0.36) and use 0.38 instead, the following JSON would override the version to `0.38--5`:

```
{
    "biocontainers" : {
        "trimmomatic" : {
            "use_local" : false,
            "quayurl" : "quay.io/biocontainers/trimmomatic",
            "version" : "0.38--5"
        }
    }
}
```

This can be placed in a JSON file like `config/custom_trimmomatic.json` (in the `workflows/` directory) and passed to Snakemake using the `--config` flag like:

```
snakemake --config=config/custom_trimmomatic.json \
        [FLAGS] <target>
```

How do I use Snakemake with Singularity?

Singularity is a containerization technology similar to Docker but without the need for root access. Snakefiles in Dahak contain `singularity:` directives, which specify a Singularity image to pull and use to run the given commands. These directives are ignored by default, Snakemake must be run with the `--use-singularity` flag to run each command through a singularity container:

```
snakemake --use-singularity <target>
```

When Singularity containers are run, a host directory can be bind-mounted inside the container to provide a shared-access folder on the host filesystem.

To specify a directory for Singularity to bind-mount, use the `SINGULARITY_BINDPATH` environment variable:

```
SINGULARITY_BINDPATH="my_data:/data" snakemake --use-singularity <target>
```

This bind-mounts the directory `my_data/` into the Singularity container at `/data/`.

Where will data files live?

To set the scratch/working directory for the Snakemake workflows, in which all intermediate and final data files will be placed, set the `data_dir` key in the Snakemake configuration dictionary. If this option is not set, it will be `data` by default.

(No trailing slash is needed when specifying the directory name.)

!!! warning “Singularity Bind Path”

```
If you use a custom directory by setting the `data_dir` key,
you must also adjust the `SINGULARITY_BINDPATH` variable
accordingly.
```

For example, to put all intermediate files into the `work/` directory instead of the `data/` directory, the following very short JSON file could be used for the Snakemake configuration dictionary (this would use default values for everything except `data_dir`):

```
{
    "data_dir" : "work"
}
```

This JSON file can be used as the Snakemake configuration dictionary by passing the JSON file name to the `--configfile` flag to Snakemake and updating the Singularity environment variable:

```
SINGULARITY_BINDPATH="work:/work" \
        snakemake --configfile=config/custom_scratch.settings \
        [FLAGS] <target>
```

How do I customize my workflow with custom configuration files?

See the Snakemake Configuration page.

Summary

All together, the command to run a Dahak workflow will look like this:

```
SINGULARITY_BINDPATH="data:/data" snakemake \
    --configfile my_workflow_params.json \
    --use-singularity \
    <target>
```

Workflow Configuration

Snakemake Configuration

The `workflows/config/` directory contains files used to set key-value pairs in the configuration dictionary used by Snakemake.

To use a custom configuration file (JSON or YAML format), use the `--configfile` flag:

```
snakemake --configfile my_workflow_params.json ...
```

Data Files, Workflow Config, and Parameters

The `dahak/workflows/config/` directory contains configuration files for Dahak workflows. There are three types of configuration details that the user may wish to customize:

- **Data files** (a list of names and URLs for read files; these read files do not all necessarily need to be used in the workflow)
  - See `dahak/workflows/config/example_datafiles.json` for an example
- **Workflow configuration** (specifying which read files to run each workflow on)
  - See `dahak/workflows/config/example_workflowconfig.json` for an example
- **Workflow parameters** (parameters specifying how the workflow will run)
  - See `dahak/workflows/config/example_workflowparams.json` for an example

How do I specify my data files?

To set the names and URLs of read files, set the `files` key in the Snakemake configuration dictionary to a list of key-value pairs, where the key is the name of the read file and the value is the URL of the file (do not include `http://` or `https://` in the URL).

For example, the following JSON block will provide a list of reads and their corresponding URLs:

```
{
    "files" : {
        "SRR606249_1_reads.fq.gz" :           "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_2_reads.fq.gz" :           "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset25_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset25_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset50_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset50_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    }
}
```

This can be placed in a JSON file like `dahak/workflows/config/custom_datafiles.json` and passed to Snakemake using the `--config` flag like:

```
snakemake --config=config/custom_datafiles.json \
        [FLAGS] <target>
```

NOTE: Dahak assumes that read files are not present on the local machine and uses a rule to download the reads from the given URL if they are not present. If your read files *are* present locally, they must be put into the temporary directory (set by `data_dir` key in the Snakemake configuration dictionary, `data/` by default) so that Snakemake will find them. If Snakemake finds the read files, the download command will not be run. You can use an empty string for the URLs in this case.

How do I specify my workflow configuration?

The workflow configuration file specifies details about *which* workflow should be run. This is typically just the name of the samples to run the workflow on, and a quality value or a k value.

Set the `workflows` key of the Snakemake configuration dictionary to a dictionary containing details about the workflow you want to run. The workflow configuration values are values that will end up in the final output file's filename.

```
{
    "workflows" : {
        "name_of_workflow" : {
            "workflow_option" : "workflow_option_value"
        }
    }
}
```

For example, here is what the workflow configuration looks like for the assembly workflow:

```
{
    "workflows" : {
        "assembly_workflow_metaspades" : {
            "sample"    : ["SRR606249_subset10","SRR606249_subset25"],
            "qual"      : ["2","30"],
        },

        "assembly_workflow_megahit" : {
            "sample"    : ["SRR606249_subset10","SRR606249_subset25"],
            "qual"      : ["2","30"],
        },

        "assembly_workflow_all" : {
            "sample"    : ["SRR606249_subset10","SRR606249_subset25"],
            "qual"      : ["2","30"],
        },

    }
}
```

Each workflow component has a "Snakemake" page that covers workflow configuration options and details for the respective workflow. Use the navigation menu on the left and select the workflow component of interest, then pick the "Snakemake" page.

How do I specify my workflow parameters?

Workflow parameters specify parameters that are used when executing individual workflow steps. These parameters are not incorporated in the final filename and are usually more extensive.

These are set using a key that is the name of the workflow component. For example, the assembly workflow parameters section of the workflow parameters file looks like this:

```
{
    "assembly" : {
        "assembly_patterns" : {
            "metaspades_pattern" : "{sample}.trim{qual}_metaspades.contigs.fa",
            "megahit_pattern" : "{sample}.trim{qual}_megahit.contigs.fa",
            "assembly_pattern" : "{sample}.trim{qual}_{assembler}.contigs.fa",
            "quast_pattern" : "{sample}.trim{qual}_{assembler}_quast/report.html",
            "multiqc_pattern" : "{sample}.trim{qual}_{assembler}_multiqc/report.html",
        }
    }
}
```

Each workflow component has a "Snakemake" page that covers workflow parameters and details for the respective workflow. Use the navigation menu on the left and select the workflow component of interest, then pick the "Snakemake" page.

.settings (Defaults) vs .json (Overriding Defaults)

The `*.settings` files are Python scripts that set the default Snakemake configuration dictionary. Each `*.settings` file is prefixed by `default_` because it is setting default values.

If the user does not specify a particular configuration dictionary key-value pair, then the default key-value pair will be used. However, if the user sets a key-value pair, it will override the default value.

This allows the user to customize any configuration key-value pair used by a workflow without having to explicitly specify every configuration key-value pair.

For example, by default the version of each container image for each program obtained from the biocontainers project is specified in the top-level `biocontainers` key in `default_parameters.settings`:

```
config_default = {

    ...

    "biocontainers" : {
        "sourmash" : {
            "use_local" : False,
            "quayurl" : "quay.io/biocontainers/sourmash",
            "version" : "2.0.0a3--py36_0"
        },
        "trimmomatic" : {
            "use_local" : False,
            "quayurl" : "quay.io/biocontainers/trimmomatic",
            "version" : "0.36--5"
        },
        "khmer" : {
            "use_local" : False,
            "quayurl" : "quay.io/biocontainers/khmer",
            "version" : "2.1.2--py35_0"
        },

        ...
```

(Note the `*.settings` files and the above code are Python, not JSON.)

If the user wishes to bump the version of trimmomatic to (e.g.) 0.38, but not change the version of khmer or sourmash, the user can specify a JSON file with the following configuration block:

```
{
    "biocontainers" : {
        "trimmomatic" : {
            "use_local" : false,
            "quayurl" : "quay.io/biocontainers/trimmomatic",
            "version" : "0.38--5"
        }
    }
}
```

This can be placed in a JSON file like `dahak/workflows/config/custom_workflowparams.json` and passed to Snakemake using the `--config` flag like:

```
snakemake --config=config/custom_workflowparams.json \
        [FLAGS] <target>
```

This will only override the container version used for trimmomatic, and will use the defaults for all other container images.

Quick Start

Quick Start

Let's run through the entire process start to finish.

Useful Snakemake Flags

Two useful snakemake flags that you can add are:

- `--dryrun` or `-n`: do a dry run of the workflow but do not actually run any commands
- `--printshellcmds` or `-p`: print the shell commands that are being executed (or would be executed if combined with `--dryrun`)

Read Filtering

We will run two variations of the read filtering workflow, and perform a quality assessment of our reads both before and after quality trimming.

Before you begin, make sure you have Singularity installed as in the Installing documentation.

Start by cloning a copy of the repository:

```
git clone https://github.com/dahak-metagenomics/dahak
```

then move into the `workflows/` directory of the Dahak repository:

```
cd dahak/workflows/
```

Now create a JSON file that defines a Snakemake configuration dictionary. This file should:

- Provide URLs at which each read filtering file can be accessed
- Provide a set of quality trimming values to use (2 and 30)
- Specify which read files should be used for the workflow
- Specify a container image from the biocontainers project to use with Singluarity
- Set all read filtering parameters

(See the Read Filtering Snakemake page for details on these options.)

Copy and paste the following:

```
cat > readfilt.json <<EOF
{
    "files" : {
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    },

    "workflows" : {
        "read_filtering_pretrim_workflow" : {
            "sample"    : ["SRR606249_subset10"],
        },
        "read_filtering_posttrim_workflow" : {
            "sample"    : ["SRR606249_subset10"],
        },
    },
}
EOF
```

This creates a workflow configuration file `readfilt.json` that will download the example data files and configure one or more workflows.

We will run two workflows: one pre-trimming quality assessment, and one post-trimming quality assessment, so we call Snakemake and pass it two build targets: `read_filtering_pretrim_workflow` and `read_filtering_posttrim_workflow`.

```
export SINGULARITY_BINDPATH="data:/data"

snakemake --use-singularity \
          --configfile=readfilt.json \
          read_filtering_pretrim_workflow read_filtering_posttrim_workflow
```

This command outputs FastQC reports for the untrimmed reads as well as the reads trimmed at both quality cutoffs, and also outputs the trimmed PE reads and the orphaned reads. All files are placed in the `data/` subdirectory.

Assembly

We will run two assembler workflows using the Megahit assembler workflow implemented in Dahak.

(See the Assembly Snakemake page for details on these options.)

Create a JSON file that defines a Snakemake configuration dictionary:

```
cat > assembly.json <<EOF
{
    "files" : {
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    },

    "workflows" : {
        "assembly_workflow_megahit" : {
            "sample"    : ["SRR606249_subset10"],
            "qual"      : ["2","30"],
        }
    },
}
EOF
```

To run the assembly workflow with both assemblers, we call Snakemake with the `assembly_workflow_all` target.

```
export SINGULARITY_BINDPATH="data:/data"

snakemake --use-singularity \
          --configfile=assembly.json \
          assembly_workflow_megahit
```

Comparison

In this section we will run a comparison workflow to compute sourmash signatures for both filtered reads and assemblies, and compare the computed signatures to a reference database.

Create a config file:

(See the Comparison Snakemake page for details on these options.)

Copy and paste the following:

```
cat > comparison.json <<EOF
{
    "files" : {
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    },

    "workflows" : {
        "comparison_workflow_reads" : {
            "kvalue"    : ["21","31","51"],
        }
    },
}
EOF
```

Now, run the `comparison_workflow_reads` workflow:

```
export SINGULARITY_BINDPATH="data:/data"

snakemake --use-singularity \
          --configfile=comparison.json \
          comparison_workflow_reads
```

Taxonomic Classification

Taxonomic Classification with Sourmash

There are a number of taxonomic classification workflows implemented in Dahak. In this section we cover the use of the sourmash tool for taxonomic classification.

Before you begin, make sure you have everything listed on the Installing page available on your command line.

There are two taxonomic classification build rules that use sourmash: `taxonomic_classification_signatures_workflow` and `taxonomic_classification_gather_workflow`.

Signatures Workflow

The signatures workflow uses sourmash to compute k-mer signatures from read files. This is essentially the same as the compute signatures step in the comparison workflow.

(See the Taxonomic Classification Snakemake page for details on this workflow.)

```
cat > compute.json <<EOF
{
    "files" : {
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    },

    "workflows" : {
        "taxonomic_classification_signatures_workflow" : {
            "sample"  : ["SRR606249_subset10"],
            "qual" : ["2","30"]
        }
    }
}
EOF
```

```
export SINGULARITY_BINDPATH="data:/data"

snakemake --use-singularity \
        --configfile=compute.json \
        taxonomic_classification_signatures_workflow
```

Gather Workflow

The gather workflow uses sourmash to (gather?) signatures computed from read files and compare them to signatures stored in a genome database.

Create a JSON file for the taxonomic classification gather workflow that defines a Snakemake configuration dictionary:

```
cat > gather.json <<EOF
{
    "files" : {
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    },

    "workflows" : {
        "taxonomic_classification_gather_workflow" : {
            "sample"  : ["SRR606249_subset10"],
            "qual" : ["2","30"]
        }
    }
}
EOF
```

To run the gather workflow, we call Snakemake with the `taxonomic_classification_gather_workflow` target.

```
export SINGULARITY_BINDPATH="data:/data"

snakemake --use-singularity \
          --configfile=taxkaiju.json \
          taxonomic_classification_gather_workflow
```

Taxonomic Classification with Kaiju

There are several taxonomic classification workflows in Dahak that use the Kaiju tool as well. This section covers those workflows.

There are three taxonomic classification build rules that use kaiju:

- `taxonomic_classification_kaijureport_workflow`
- `taxonomic_classification_kaijureport_filtered_workflow`
- `taxonomic_classification_kaijureport_filteredclass_workflow`

Kaiju Report Workflow

Create a JSON file that defines a Snakemake configuration dictionary:

```
cat > taxkaiju.json <<EOF
{
    "files" : {
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    },

    "workflows" : {
        "taxonomic_classification_kaijureport_workflow" : {
            "sample"  : ["SRR606249_subset10"],
            "qual" : ["2","30"]
        }
    }
}
EOF
```

To run the taxonomic classification workflow to generate a kaiju report, we call Snakemake with the `taxonomic_classification` target.

```
export SINGULARITY_BINDPATH="data:/data"

snakemake --use-singularity \
          --configfile=taxkaiju.json \
          taxonomic_classification_kaijureport_workflow
```

Kaiju Filtered Species Report Workflow

The filtered kaiju workflow filters for species whose reads compose less than N% of the total reads, where N is a parameter set by the user.

Copy and paste the following:

```
cat > taxkaiju_filtered.json <<EOF
{
    "files" : {
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    },

    "taxonomic_classification" : {
        "filter_taxa" : {
            "pct_threshold" : 1
        }
    },

    "workflows" : {
        "taxonomic_classification_kaijureport_filtered_workflow" : {
            "sample"  : ["SRR606249_subset10"],
            "qual" : ["2","30"]
        }
    }
}
EOF
```

To run the taxonomic classification filtered report workflow, we call Snakemake with the `taxonomic_classification_kaijureport_filtered_workflow` target.

```
export SINGULARITY_BINDPATH="data:/data"

snakemake --use-singularity \
          --configfile=taxkaiju_filtered.json \
          taxonomic_classification_kaijureport_filtered_workflow
```

Kaiju Filtered Species by Class Report Workflow

The last workflow implements filtering but also implements reporting the taxa level reported by kaiju. This iuses the “genus” taxonomic rank level by default.

Copy and paste the following:

```
cat > taxkaiju_filteredclass.json <<EOF
{
    "files" : {
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    },

    "taxonomic_classification" : {
        "kaiju_report" : {
            "taxonomic_rank" : "genus"
        }
    },

    "workflows" : {

        "taxonomic_classification_kaijureport_filteredclass_workflow" : {
            "sample"  : ["SRR606249_subset10"],
            "qual" : ["2","30"]
        },
    }
}
```

To run the taxonomic classification workflow to generate this kaiju report, we call Snakemake with the `taxonomic_classification_kaijureport_filteredclass_workflow` target.

```
export SINGULARITY_BINDPATH="data:/data"

snakemake --use-singularity \
          --configfile=taxkaiju_filtered.json \
          taxonomic_classification_kaijureport_filteredclass_workflow
```

Troubleshooting

Troubleshooting

Errors

Snakemake error: Directory cannot be locked

If you see an error like this when running Snakemake:

```
Building DAG of jobs...
Error: Directory cannot be locked. Please make sure that no other Snakemake process is tryin
If you are sure that no other instances of snakemake are running on this directory, the rema
```

you can try the following:

- Remove locks with `snakemake --unlock`
- Remove or rename `data/` directory
- Run `export SINGULARITY_BINDPATH="data:/data"`
- Re-run the Snakemake command

Overview

Read Filtering Workflow

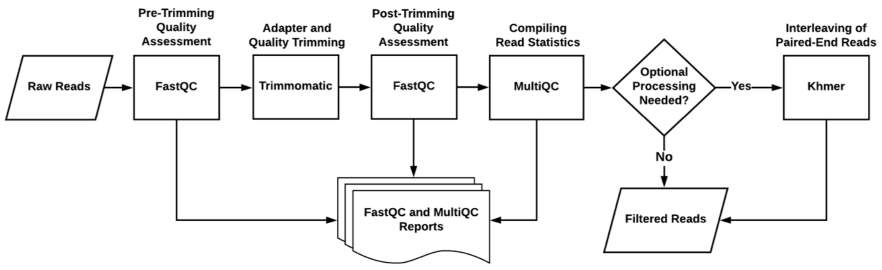

Pre-Trimming Quality Assessment
Adapter and Quality Trimming
Post-Trimming Quality Assessment
Compiling Read Statistics
Interleaving of Paired-End Reads
Raw Reads
FastQC
Trimmomatic
FastQC
MultiQC
Optional Processing Needed?
Yes
Khmer
No
FastQC and MultiQC Reports
Filtered Reads

Figure 9: Quality assessment

The read filtering step consists of processing raw reads from a sequencer, such as discarding reads with a high uncertainty value or trimming off adapters.

Tools like Fastqc and Trimmomatic will perform this filtering process for the sequencer's reads.

More information:

- Read Filtering Walkthrough
- Read Filtering Snakemake
- `workflows/readfiltering/` directory in the repository

Walkthrough

Read Filtering Walkthrough

!!! warning "Important Note"

```
The following walkthrough is **independent of the Snakemake workflows.**
All commands given below are bash commands. This walkthrough also utilizes
Docker, while the Snakemake workflows utilize SIngularity.
```

The following walkthrough covers the steps in the read filtering and quality assessment workflow. This walkthrough covers the use of Docker to interactively run the workflow on a fresh Ubuntu 16.04 (Xenial) image, and requires sudo commands to be run.

This walkthrough presumes that the steps covered on the Installing page have been run, and that a version of Python, Conda, Snakemake, and Docker are available. See the Installing page for instructions on installing required software.

Walkthrough Steps

Starting with a fresh image, go through the installation instructions on the Installing page.

Install the open science framework command-line client:

```
$ pip install osfclient
```

Install docker with the following shell commands:

```
$ wget -qO- https://get.docker.com/ | sudo sh
$ sudo usermod -aG docker ubuntu
```

Make a directory called data and retrieve some data using the osfclient. Specify the path to files.txt or move it to your working directory.

```
mkdir data
cd data

for i in $(cat files.txt)
do
    osf -p dm938 fetch osfstorage/data/${i}
done
```

Link the data and run fastqc

```
mkdir -p ~/data/qc/before_trim

docker run -v ${PWD}:/data -it biocontainers/fastqc fastqc /data/SRR606249_subset10_1.fq.gz

docker run -v ${PWD}:/data -it biocontainers/fastqc fastqc /data/SRR606249_subset10_2.fq.gz
```

Grab the adapter sequences:

```
cd ~/data
curl -O -L http://dib-training.ucdavis.edu.s3.amazonaws.com/mRNAseq-semi-2015-03-04/TruSeq2-
```

Link the data and run trimmomatic

```
for filename in *_1*.fq.gz
do
    # first, make the base by removing .fq.gz using the unix program basename
    base=$(basename $filename .fq.gz)
    echo $base

    # now, construct the base2 filename by replacing _1 with _2
    base2=${base/_1/_2}
    echo $base2

    docker run -v ${PWD}:/data -it quay.io/biocontainers/trimmomatic:0.36--4 trimmomatic PE
                /data/${base2}.fq.gz \
        /data/${base}.trim.fq.gz /data/${base}_se \
        /data/${base2}.trim.fq.gz /data/${base2}_se \
        ILLUMINACLIP:/data/TruSeq2-PE.fa:2:40:15 \
        LEADING:2 TRAILING:2 \
        SLIDINGWINDOW:4:2 \
        MINLEN:25
done
```

Now run fastqc on the trimmed data:

```
mkdir -p ~/data/qc/after_trim


docker run -v ${PWD}:/data -it biocontainers/fastqc fastqc /data/SRR606249_subset10_1.trim.f

docker run -v ${PWD}:/data -it biocontainers/fastqc fastqc /data/SRR606249_subset10_2.trim.f
```

Interleave paired-end reads using khmer. The output file name includes 'trim2' indicating the reads were trimmed at a quality score of 2. If other values were used change the output name accordingly.

```
cd ~/data
for filename in *_1.trim.fq.gz
do
    # first, make the base by removing _1.trim.fq.gz with basename
    base=$(basename $filename _1.trim.fq.gz)
    echo $base

    # construct the output filename
    output=${base}.pe.trim2.fq.gz

    docker run -v ${PWD}:/data -it quay.io/biocontainers/khmer:2.1--py35_0 interleave-reads.
        /data/${base}_1.trim.fq.gz /data/${base}_2.trim.fq.gz --no-reformat -o /data/$output

done
```

Snakemake

Read Filtering Workflow: Snakemake

As mentioned on the Running Workflows page, the Snakemake workflows define **build rules** that trigger all of the rules composing a given workflow.

As a reminder, the Running Workflows page showed how to call Snakemake and ask for a particular target:

```
snakemake [FLAGS] <target>
```

You can replace `<target>` with any of the build rules below.

Build Rules

The build rules are the rules that the end user should be calling. A list of available build rules in the read filtering workflow is given below.

```
read_filtering_pretrim_workflow

    Build rule: trigger the read filtering workflow

read_filtering_posttrim_workflow

    Build rule: trigger the read filtering workflow
```

Pass the name of the build rule directly to Snakemake on the command line:

```
snakemake [FLAGS] read_filtering_pretrim_workflow read_filtering_posttrim_workflow
```

See the Quick Start for details on the process of running this workflow.

Snakemake Configuration Dictionary

There are three types of key-value pairs that can be set in the Snakemake configuration dictionary (also see the Workflow Configuration page).

Data Files Configuration

Set the `files` key to a dictionary containing a list of key-value pairs, where the keys are filenames and values are URLs:

```
{
    "files" : {
        "SRR606249_1_reads.fq.gz" :           "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_2_reads.fq.gz" :           "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset25_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset25_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset50_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset50_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    }
}
```

Put these in a JSON file (e.g., `config/custom_datafiles.json` in the `workflows` directory) and pass the name of the config file to Snakemake using the `--configfile` flag:

```
snakemake --configfile=config/custom_datafiles.json [FLAGS] <target>
```

Workflow Configuration

Set the `workflows` key of the Snakemake configuration dictionary to a dictionary containing details about the workflow you want to run. The workflow configuration values are values that will end up in the final output file's filename.

Here is the structure of the configuration dictionary for read filtering workflows (pre-trimming and post-trimming quality assessment):

```
{
    "workflows" : {

        "read_filtering_pretrim_workflow" : {
            "sample"    : ["SRR606249_subset10","SRR606249_subset25"]
        },

        "read_filtering_posttrim_workflow" : {
            "sample"    : ["SRR606249_subset10","SRR606249_subset25"],
            "qual"   : ["2","30"]
        }
    }
}
```

The `sample` list specifies the prefixes of the sample reads to run the read filtering workflow on. The `qual` list specifies the values to use for quality trimming (for the post-trimming workflow).

Put these in a JSON file (e.g., `config/custom_workflowconfig.json` in the `workflows` directory) and pass the name of the config file to Snakemake using the `--configfile` flag:

```
snakemake --configfile=config/custom_workflowconfig.json [FLAGS] <target>
```

Workflow Parameters

Set the `read_filtering` key of the Snakemake configuration dictionary to a dictionary containing various child dictionaries and key-value pairs:

```
{
    "read_filtering" : {
        "read_patterns" : {
            "pre_trimming_pattern"  : "{sample}_{direction}_reads.fq.gz",
            "post_trimming_pattern" : "{sample}_{direction}.trim{qual}.fq.gz",
        },

        "direction_labels" : {
            "forward" : "1",
            "reverse" : "2"
        },

        "quality_assessment" : {
            "fastqc_suffix": "fastqc",
        },

        "quality_trimming" : {
            "trim_suffix" : "se"
        },

        "interleaving" : {
            "interleave_suffix" : "pe"
        },

        "adapter_file" : {
            "name" : "TruSeq2-PE.fa",
            "url"  : "http://dib-training.ucdavis.edu.s3.amazonaws.com/mRNAseq-semi-2015-03-
        }
    }
}
```

The `pre_trimming_pattern` must match the filename pattern of the reads that are provided in the `files` key. The `{sample}` and `{direction}` notation is for Snakemake to match wildcards. For example, the pattern

```
    {sample}_{direction}_reads.fq.gz
```

will match the filename

```
    SRR606249_subset10_1_reads.fq.gz
```

such that the wildcard values are `sample=SRR606249_subset10` and `direction=1`.

The `direction_labels` key is used to indicate the suffix used for forward and reverse reads; this is typically `1` and `2` but can be customized by the user if needed.

To use custom values for these parameters, put the configuration dictionary above (or any subset of it) into a JSON file (e.g., `config/custom_workflowparams.json` in the `workflows` directory) and pass the name of the config file to Snakemake using the `--configfile` flag:

```
snakemake --configfile=config/custom_workflowparams.json [FLAGS] <target>
```

Overview

Assembly Workflow

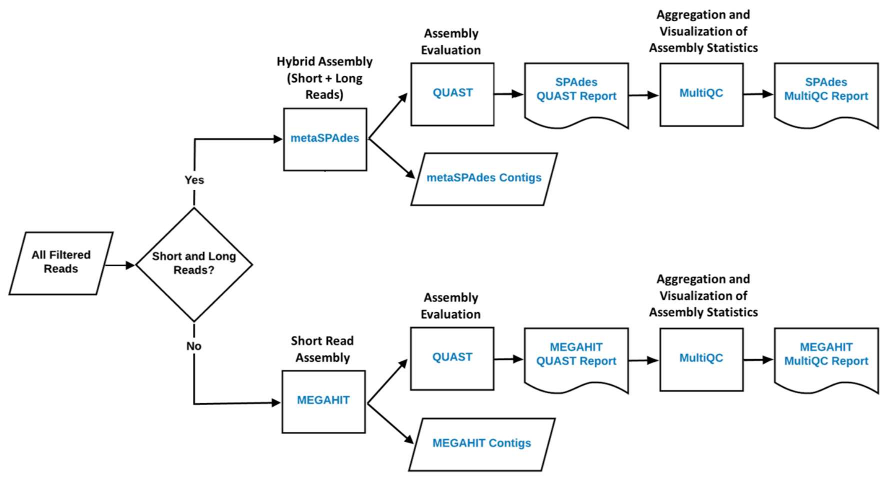


Figure 10: Assembly

The assembly workflow uses assembly-based approaches to give higher-confidence gene identity assignment than raw read assignment alone.

The assembly step determines the proper order of the reads, and assembles the genome. The assembly tool may use short reads (~350 or fewer reads), or it may use long reads (>1000 reads).

Reads are assembled in order into contigs (chunks of contiguous reads). The contigs are themselves assembled into scaffolds that consist of several contigs.

The SPAdes tool can handle short or long reads, while the megahit tool works better for short reads.

Metaquast gives assembly statistics that can help evaluate the assembly (how long, number of fragments, number of contigs, number of scaffolds, etc.).

More information:

- Assembly Walkthrough
- Assembly Snakemake Rules
- `workflows/assembly/` directory in the repository

Walkthrough

Assembly with SPAdes and MEGAHIT

Requirements (from unbuntu 16.04 using filtered reads)

Disk space(60 gb) RAM(32 gb) Updated packages(see read filtering) Docker(see read filtering for installation instructions)

Link the data and run MEGAHIT

```
for filename in *.trim30.fq.gz
do

    base=$(basename $filename .trim30.fq.gz)
    echo $base

    docker run -v ${PWD}:/data -it quay.io/biocontainers/megahit:1.1.1--py36_0 \
    megahit --12 /data/${i} --out-prefix=${base} -o /data/${base}.megahit_output
done
```

Now run Quast to generate some assembly statistics

```
for file in *.megahit_output/*contigs.fa
do

    base=$(basename $file .contigs.fa)
    echo $base

    docker run -v /${PWD}:/data -it quay.io/biocontainers/quast:4.5--boost1.61_1 \
    quast.py /data/${file} -o /data/${base}.megahit_quast_report
done
```

Link the data and run SPAdes

```
for filename in *pe.trim30.fq.gz
do
    base=$(basename $filename .trim30.fq.gz)
    echo $base

    docker run -v ${PWD}:/data -it quay.io/biocontainers/spades:3.10.1--py27_0 \
    metaspades.py --12 /data/${filename} \
    -o /data/${base}.spades_output
done
```

Now run Quast to generate some assembly statistics

```
for file in *.spades_output
do

    base=$(basename $file .spades_output)
    echo ${base}

    docker run -v ${PWD}:/data -it quay.io/biocontainers/quast:4.5--boost1.61_1 \
    quast.py /data/${base}.spades_output/contigs.fasta \
    -o data/${base}.spades_quast_report
done
```

Snakemake

Assembly: Snakemake

As mentioned on the Running Workflows page, the Snakemake workflows define **build rules** that trigger all of the rules composing a given workflow.

As a reminder, the Running Workflows page showed how to call Snakemake and ask for a particular target:

```
snakemake [FLAGS] <target>
```

You can replace `<target>` with any of the build rules below.

Build Rules

The build rules are the rules that the end user should be calling. A list of available build rules in the assembly workflow is given below.

```
assembly_workflow_metaspades
```

```
    Build rule: trigger the metaspades assembly step.

assembly_workflow_megahit

    Build rule: trigger the megahit assembly step.

assembly_workflow_all

    Build rule: trigger the assembly step with all assemblers.
```

Pass the name of the build rule directly to Snakemake on the command line:

```
snakemake [FLAGS] assembly_workflow_all
```

See below for how to configure these workflows. See the Quick Start for details on the process of running this workflow.

Snakemake Configuration Dictionary

There are three types of key-value pairs that can be set in the Snakemake configuration dictionary (also see the Workflow Configuration page).

Data Files Configuration

Set the `files` key to a dictionary containing a list of key-value pairs, where the keys are filenames and values are URLs:

```
{
    "files" : {
        "SRR606249_1_reads.fq.gz" :           "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_2_reads.fq.gz" :           "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset25_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset25_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset50_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset50_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    }
}
```

Put these in a JSON file (e.g., `config/custom_datafiles.json` in the `workflows` directory) and pass the name of the config file to Snakemake using the `--configfile` flag:

```
snakemake --configfile=config/custom_datafiles.json [FLAGS] <target>
```

Workflow Configuration

Set the `workflows` key of the Snakemake configuration dictionary to a dictionary containing details about the workflow you want to run. The workflow configuration values are values that will end up in the final output file's filename.

Here is the structure of the configuration dictionary for assembly workflows:

```
{
    "workflows" : {
        "assembly_workflow_metaspades" : {
            "sample"    : ["SRR606249_subset10","SRR606249_subset25"],
            "qual"      : ["2","30"],
        },

        "assembly_workflow_megahit" : {
            "sample"    : ["SRR606249_subset10","SRR606249_subset25"],
            "qual"      : ["2","30"],
        },

        "assembly_workflow_all" : {
            "sample"    : ["SRR606249_subset10","SRR606249_subset25"],
            "qual"      : ["2","30"],
        },

    }
}
```

The `sample` and `qual` keys are used by Snakemake to generate a list of input files required for the rule, and to generate the name of the output file from the workflow. These should be lists of strings. `sample` should match the read files listed in the `files` section, while `qual` should be the quality trimming value to use.

Put these in a JSON file (e.g., `config/custom_workflowconfig.json` in the `workflows` directory) and pass the name of the config file to Snakemake using the `--configfile` flag:

```
snakemake --configfile=config/custom_workflowconfig.json [FLAGS] <target>
```

Workflow Parameters

Set the `assembly` key of the Snakemake configuration dictionary as shown below:

```
{
    "assembly" : {
        "assembly_patterns" : {
            "metaspades_pattern" : "{sample}.trim{qual}_metaspades.contigs.fa",
            "megahit_pattern" : "{sample}.trim{qual}_megahit.contigs.fa",
            "assembly_pattern" : "{sample}.trim{qual}_{assembler}.contigs.fa",
            "quast_pattern" : "{sample}.trim{qual}_{assembler}_quast/report.html",
            "multiqc_pattern" : "{sample}.trim{qual}_{assembler}_multiqc/report.html",
        }
    }
}
```

To use custom values for these parameters, put the configuration dictionary above (or any subset of it) into a JSON file (e.g., `config/custom_workflowparams.json` in the `workflows` directory) and pass the name of the config file to Snakemake using the `--configfile` flag:

```
snakemake --configfile=config/custom_workflowparams.json [FLAGS] <target>
```

Overview

Sample Comparison Workflow

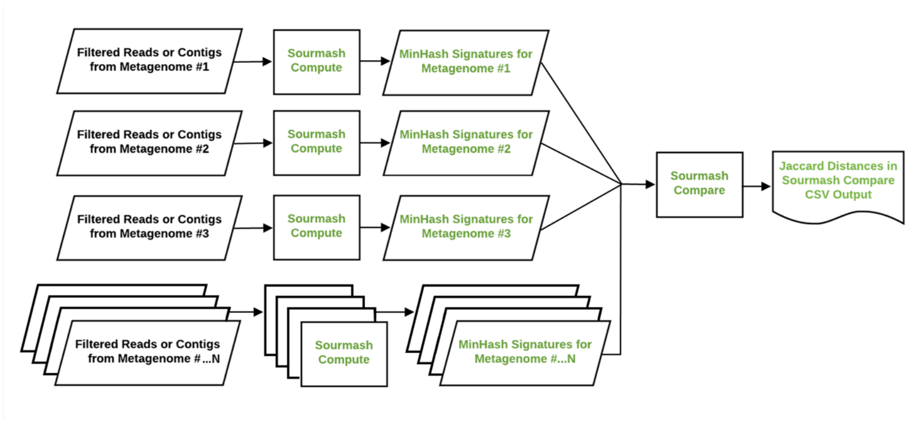

Filtered Reads or Contigs from Metagenome #1
Sourmash Compute
MinHash Signatures for Metagenome #1
Filtered Reads or Contigs from Metagenome #2
Sourmash Compute
MinHash Signatures for Metagenome #2
Filtered Reads or Contigs from Metagenome #3
Sourmash Compute
MinHash Signatures for Metagenome #3
Filtered Reads or Contigs from Metagenome #...N
Sourmash Compute
MinHash Signatures for Metagenome #...N
Sourmash Compare
Jaccard Distances in Sourmash Compare CSV Output

Figure 11: Sample comparison

The sample comparison workflow helps perform rapid k-mer-based ordination analyses of many samples to provide sample groupings and identify potential outliers.

Operating at the level of k-mers (representations of the reads), the comparison step is taking the reads that were not fingerprinted by the assembler and seeing if they match genomes of other organisms.

The tool used for comparison is sourmash.

More information:

- Comparison Walkthrough

- Comparison Snakemake Rules
- `workflows/comparison/` directory in the repository

Walkthrough

Metagenome comparison with sourmash

Sourmash is a tool for calculating and comparing MinHash signatures. Sourmash compare calculates the jaccard similarity of MinHash signatures.

If you don't already have them, retrieve the assembled contigs:

```
for i in $(cat assembly_names.txt)
do

    osf -u philliptbrooks@gmail.com -p dm938 fetch ${i} ${i}
    echo ${i}
done
```

Calculate signatures

Calculate sourmash signatures for reads:

```
for filename in *_1.trim2.fq.gz
do
    #Remove _1.trim2.fq from file name to create base
    base=$(basename $filename _1.trim2.fq.gz)
    echo $base

    docker run -v ${PWD}:/data quay.io/biocontainers/sourmash:2.0.0a1--py35_2 \
        sourmash compute \
        --merge /data/${base}.trim2.fq.gz \
        --scaled 10000 \
        -k 21,31,51 \
        /data/${base}_1.trim2.fq.gz \
        /data/${base}_2.trim2.fq.gz \
        -o /data/${base}.pe.trim2.fq.gz.k21_31_51.sig
done

for filename in *_1.trim30.fq.gz
do
    #Remove _1.trim30.fq from file name to create base
    base=$(basename $filename _1.trim30.fq.gz)
    echo $base

    docker run -v ${PWD}:/data quay.io/biocontainers/sourmash:2.0.0a1--py35_2 \
        sourmash compute \
        --merge /data/${base}.trim30.fq.gz \
        --scaled 10000 \
        -k 21,31,51 \
        /data/${base}_1.trim30.fq.gz \
        /data/${base}_2.trim30.fq.gz \
        -o /data/${base}pe.trim30.fq.gz.k21_31_51.sig
done
```

Calculate sourmash signatures for assemblies:

```
for i in osfstorage/assembly/SRR606249{"_1","_subset10_1","_subset25_1","_subset50_1"}.trim
do
    base=`echo ${i} | awk -F/ '{print $3}'`
    echo ${base}

    docker run -v ${PWD}:/data quay.io/biocontainers/sourmash:2.0.0a1--py35_2 \
            sourmash compute \
            -k 21,31,51 \
            --scaled 10000 \
            /data/${i} \
            -o /data/${base}.k21_31_51.sig
done

for i in osfstorage/assembly/SRR606249{"_1","_subset10_1","_subset25_1","_subset50_1"}.trim
do
        base=`echo ${i} | awk -F/ '{print $3}'`
        echo ${base}

        docker run -v ${PWD}:/data quay.io/biocontainers/sourmash:2.0.0a1--py35_2 \
            sourmash compute \
            -k 21,31,51 \
            --scaled 10000 \
            /data/${i} \
            -o /data/${base}.k21_31_51.sig
done
```

Compare read signatures

Run sourmash compare on all the read signatures:

```
for i in 21 31 51
do

        docker run -v ${PWD}:/data quay.io/biocontainers/sourmash:2.0.0a1--py35_2 \
            sourmash compare \
            /data/SRR606249.pe.trim2.fq.gz.k21_31_51.sig \
            /data/SRR606249.pe.trim30.fq.gz.k21_31_51.sig \
            /data/SRR606249_subset10.pe.trim2.fq.gz.k21_31_51.sig \
            /data/SRR606249_subset10.pe.trim30.fq.gz.k21_31_51.sig \
            /data/SRR606249_subset25.pe.trim2.fq.gz.k21_31_51.sig \
            /data/SRR606249_subset25.pe.trim30.fq.gz.k21_31_51.sig \
            /data/SRR606249_subset50.pe.trim2.fq.gz.k21_31_51.sig \
            /data/SRR606249_subset50.pe.trim30.fq.gz.k21_31_51.sig \
            -k ${i} \
            --csv /data/SRR606249.pe.trim2and30_comparison.k${i}.csv
done
```

Compare assembly signatures

```
for i in 21 31 51
do
    docker run -v ${PWD}:/data quay.io/biocontainers/sourmash:2.0.0a1--py35_2 \
        sourmash compare \
        /data/SRR606249_1.trim2.fq.gz_megahit_output.k21_31_51.sig \
        /data/SRR606249_1.trim2.fq.gz_spades_output.k21_31_51.sig \
        /data/SRR606249_1.trim30.fq.gz_megahit_output.k21_31_51.sig \
        /data/SRR606249_1.trim30.fq.gz_spades_output.k21_31_51.sig \
        /data/SRR606249_subset10_1.trim2.fq.gz_megahit_output.k21_31_51.sig \
        /data/SRR606249_subset10_1.trim2.fq.gz_spades_output.k21_31_51.sig \
        /data/SRR606249_subset10_1.trim30.fq.gz_megahit_output.k21_31_51.sig \
        /data/SRR606249_subset25_1.trim2.fq.gz_spades_output.k21_31_51.sig \
        /data/SRR606249_subset25_1.trim30.fq.gz_megahit_output.k21_31_51.sig \
        /data/SRR606249_subset25_1.trim30.fq.gz_spades_output.k21_31_51.sig \
        /data/SRR606249_subset50_1.trim2.fq.gz_megahit_output.k21_31_51.sig \
        /data/SRR606249_subset50_1.trim2.fq.gz_spades_output.k21_31_51.sig \
        /data/SRR606249_subset50_1.trim30.fq.gz_megahit_output.k21_31_51.sig \
        /data/SRR606249_subset50_1.trim30.fq.gz_spades_output.k21_31_51.sig \
        -k ${i} \
        --csv /data/SRR606249.pe.trim2and30_megahitandspades_comparison.k${i}.csv
done
```

Compare read signatures to assembly signatures

```
for i in 21 31 51
do
        docker run -v ${PWD}:/data quay.io/biocontainers/sourmash:2.0.0a1--py35_2 \
                sourmahs compare \
                /data/SRR606249_1.trim2.fq.gz_megahit_output.k21_31_51.sig \
                /data/SRR606249_1.trim2.fq.gz_spades_output.k21_31_51.sig \
                /data/SRR606249.pe.trim2.fq.gz.k21_31_51.sig \
                /data/SRR606249.pe.trim30.fq.gz.k21_31_51.sig \
                /data/SRR606249_1.trim30.fq.gz_megahit_output.k21_31_51.sig \
                /data/SRR606249_1.trim30.fq.gz_spades_output.k21_31_51.sig \
                -k ${i} \
                --csv /data/SRR606249.pe.trim2and30_readstoassemblies_comparison.k${i}.csv
done
```

Snakemake

Comparison: Snakemake

As mentioned on the Running Workflows page, the Snakemake workflows define **build rules** that trigger all of the rules composing a given workflow.

As a reminder, the Running Workflows page showed how to call Snakemake and ask for a particular target:

```
snakemake [FLAGS] <target>
```

Build Rules

The build rules are the rules that the end user should be calling. A list of available build rules in the taxonomic classification workflow is given below.

List of available build rules in the comparison workflow:

```
comparison_workflow_reads

    Build rule: run sourmash compare on all reads

comparison_workflow_assembly

    Build rule: run sourmash compare on all assemblies

comparison_workflow_reads_assembly

    Build rule: run sourmash compare on all reads and assemblies together
```

Pass the name of the build rule directly to Snakemake on the command line:

```
snakemake [FLAGS] comparison_workflow_reads \
        comparison_workflow_assembly \
        comparison_workflow_reads_assembly
```

See below for how to configure these workflows. See the Quick Start for details on the process of running this workflow.

Snakemake Configuration Dictionary

There are three types of key-value pairs that can be set in the Snakemake configuration dictionary (also see the Workflow Configuration page).

Data Files Configuration

Set the `files` key to a dictionary containing a list of key-value pairs, where the keys are filenames and values are URLs:

```
{
    "files" : {
        "SRR606249_1_reads.fq.gz" :           "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_2_reads.fq.gz" :           "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset25_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset25_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset50_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset50_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    }
}
```

Put these in a JSON file (e.g., `config/custom_datafiles.json` in the `workflows` directory) and pass the name of the config file to Snakemake using the `--configfile` flag:

```
snakemake --configfile=config/custom_datafiles.json [FLAGS] <target>
```

Workflow Configuration

Set the `workflows` key of the Snakemake configuration dictionary to a dictionary containing details about the workflow you want to run. The workflow configuration values are values that will end up in the final output file's filename.

Here is the structure of the configuration dictionary for taxonomic classification workflows:

```
{
    "workflows" : {
        "comparison_workflow_reads": {
            #
            # these parameters determine which reads
            # the comparison workflow will be run on
            #
            "kvalue"    : ["21","31","51"],
        },

        "comparison_workflow_assembly" : {
            #
            # these parameters determine which assembled reads
            # the comparison workflow will be run on
            #
            "kvalue"    : ["21","31","51"],
        },

        "comparison_workflow_reads_assembly" : {
            #
            # these parameters determine which reads and assembled
            # reads the comparison workflow will be run on
            #
            "kvalue"    : ["21","31","51"],
        }
    }
}
```

Put these in a JSON file (e.g., `config/custom_workflowconfig.json` in the `workflows` directory) and pass the name of the config file to Snakemake using the `--configfile` flag:

```
snakemake --configfile=config/custom_workflowconfig.json [FLAGS] <target>
```

Workflow Parameters

Set the `taxonomic_classification` key of the Snakemake configuration dictionary as shown below:

```
{
    "comparison" : {
        "compute_read_signatures" : {
            "scale"         : 10000,
            "kvalues"       : [21,31,51],
            "qual"          : ["2","30"],
            "sig_suffix"    : "_scaled10k.k21_31_51.sig",
            "merge_suffix"  : "_scaled10k.k21_31_51.fq.gz"
        },
        "compare_read_signatures" : {
            "samples" : ["SRR606249_subset10","SRR606249_subset25"],
            "csv_out" : "SRR606249allsamples_trim2and30_read_comparison.k{kvalue}.csv"
        },
        "compute_assembly_signatures" : {
            "scale"         : 10000,
            "kvalues"       : [21,31,51],
            "qual"          : ["2","30"],
            "sig_suffix" : "_scaled10k.k21_31_51.sig",
            "merge_suffix"  : "_scaled10k.k21_31_51.fq.gz"
        },
        "compare_assembly_signatures" : {
            "samples"   : ["SRR606249_subset10","SRR606249_subset25"],
            "assembler" : ["megahit","metaspades"],
            "csv_out"   : "SRR606249_trim2and30_assembly_comparison.k{kvalue}.csv"
        },
        "compare_read_assembly_signatures" : {
            "samples"   : ["SRR606249_subset10"],
            "assembler" : ["megahit","metaspades"],
            "kvalues"   : [21, 31, 51],
            "csv_out"   : "SRR606249_trim2and30_ra_comparison.k{kvalue}.csv"
        }
    }
}
```

(Note that there are multiple "k values" being specified here, but these correspond to different steps in the workflow. The `kvalues` key in the `compute_read_signatures` and `compute_assembly_signatures` section provides the k values used in the intermediate steps of the workflow, and should be a superset of the the k values provided to the build rule.

To use custom values for these parameters, put the configuration dictionary above (or any subset of it) into a JSON file (e.g., `config/custom_workflowparams.json` in the `workflows` directory) and pass the name of the config file to Snakemake using the `--configfile` flag:

```
snakemake --configfile=config/custom_workflowparams.json [FLAGS] <target>
```

Overview

Taxonomic Classification Using Custom Database

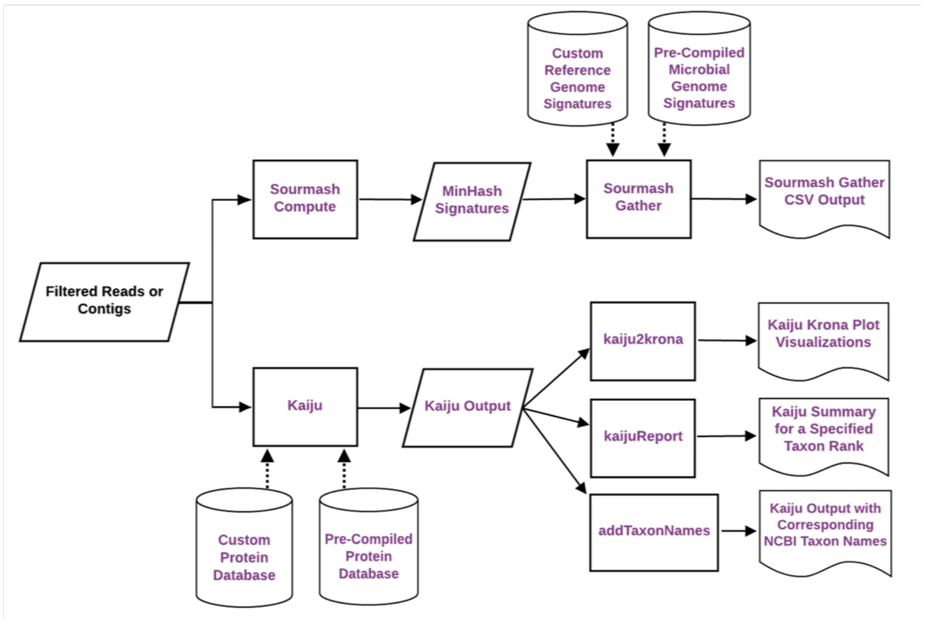

Custom Reference Genome Signatures
Pre-Compiled Microbial Genome Signatures
Sourmash Compute
MinHash Signatures
Sourmash Gather
Sourmash Gather CSV Output
Filtered Reads or Contigs
kaiju2krona
Kaiju Krona Plot Visualizations
Kaiju
Kaiju Output
kaijuReport
Kaiju Summary for a Specified Taxon Rank
addTaxonNames
Kaiju Output with Corresponding NCBI Taxon Names
Custom Protein Database
Pre-Compiled Protein Database

Figure 12: Taxonomic classification

The taxonomic classification workflow characterizes bulk metagenome data sets with the sourmash and kaiju tools using public and private reference databases.

More information:

- Taxonomic Classification Walkthrough
- Taxonomic Classification Snakemake Rules
- `workflows/taxclass/` directory in the repository

Walkthrough

Taxonomic Classification Workflow

To run the taxonomic classification workflow, use the Snakefile in the top level of the repository.

Workflow rules for taxonomic classification are defined in the file `workflows/taxonomic_classification/Snake` are defined in `workflows/read_filtering/Snakefile`.

List of Rules

The build rules trigger portions of the workflow to run. These are listed below:

- `genbank` - download pre-assembled SBTs for sourmash built from genome and protein databases (Genbank).
- `sbts` - unpack pre-assembled SBTs for sourmash built from genome and protein databases (Genbank).
- `merge` - merge read files and calculate signature file
- `usekaij` - download and unpack the kaiju database
- `runkaiju` - run the kaiju classifier on specified input files
- `runkrona` - run the krona tool to visualize kaiju output

You can see a list of all available rules and brief descriptions of each by using the Snakemake list command:

```
snakemake -l
```

Running Rules

To run a rule, call Snakemake with specified environment variables, command line flags, and options, and pass it the name of the rule. For example, the following command uses Singularity to run all rules that have a singularity directive:

```
SINGULARITY_BINDPATH="data:/data" snakemake --with-singularity post_trim
```

Walkthrough

The following walkthrough covers the steps in the taxonomic classification workflow.

This workflow covers the use of Docker to interactively run the workflow on a fresh Ubuntu 16.04 (Xenial) image, and requires sudo commands to be run.

The Snakefiles contained in this repository utilize Singularity containers instead of Docker containers, but run analogous commands to those given in this walkthrough.

This walkthrough presumes that the steps covered on the Installing page have been run, and that a version of Python, Conda, and Snakemake are available. See the Installing page for instructions on installing required software.

Walkthrough Steps

Additional requirements:

- At least 120 GB disk space
- At least 64 GB RAM

Software required:

- Docker (covered in this walkthrough) or Singularity (if using Snakefile)
- Updated packages (see the Installing page)
- (Optional) OSF CLI (command-line interface; see Installing and Read Filtering Snakemake pages)

Sourmash is a tool for calculating and comparing MinHash signatures. Sourmash gather allows us to taxonomically classify the components of a metagenome by comparing hashes in our dataset to hashes in a sequence bloom tree (SBT) representing genomes.

First, let's download two SBTs containing hashes that represent the microbial genomes in the NCBI GenBank and RefSeq databases.

(Note: these are each several GB in size, and unzipped they take up around 100 GB of space.)

```
mkdir data/
cd data/

curl -O https://s3-us-west-1.amazonaws.com/spacegraphcats.ucdavis.edu/microbe-refseq-sbt-k51
curl -O https://s3-us-west-1.amazonaws.com/spacegraphcats.ucdavis.edu/microbe-refseq-sbt-k31
curl -O https://s3-us-west-1.amazonaws.com/spacegraphcats.ucdavis.edu/microbe-refseq-sbt-k21
curl -O https://s3-us-west-1.amazonaws.com/spacegraphcats.ucdavis.edu/microbe-genbank-sbt-k5
curl -O https://s3-us-west-1.amazonaws.com/spacegraphcats.ucdavis.edu/microbe-genbank-sbt-k3
curl -O https://s3-us-west-1.amazonaws.com/spacegraphcats.ucdavis.edu/microbe-genbank-sbt-k2
```

```
tar xzf microbe-refseq-sbt-k51-2017.05.09.tar.gz
tar xzf microbe-refseq-sbt-k31-2017.05.09.tar.gz
tar xzf microbe-refseq-sbt-k21-2017.05.09.tar.gz
tar xzf microbe-genbank-sbt-k51-2017.05.09.tar.gz
tar xzf microbe-genbank-sbt-k31-2017.05.09.tar.gz
tar xzf microbe-genbank-sbt-k21-2017.05.09.tar.gz

rm -r microbe-refseq-sbt-k51-2017.05.09.tar.gz
rm -r microbe-refseq-sbt-k31-2017.05.09.tar.gz
rm -r microbe-refseq-sbt-k21-2017.05.09.tar.gz
rm -r microbe-genbank-sbt-k51-2017.05.09.tar.gz
rm -r microbe-genbank-sbt-k31-2017.05.09.tar.gz
rm -r microbe-genbank-sbt-k21-2017.05.09.tar.gz

cd ../
```

If you have not made the trimmed data, you can download it from OSF.

Start with a file that contains two columns, a filename and a location, separated by a space.

The location should be a URL (currently works) or a local path (not implemented yet). If no location is given, the script will look for the given file in the current directory.

Example data file **trimmed.data**:

```
SRR606249_1.trim2.fq.gz            https://osf.io/tzkjr/download
SRR606249_2.trim2.fq.gz            https://osf.io/sd968/download
SRR606249_subset50_1.trim2.fq.gz   https://osf.io/acs5k/download
SRR606249_subset50_2.trim2.fq.gz   https://osf.io/bem28/download
SRR606249_subset25_1.trim2.fq.gz   https://osf.io/syf3m/download
SRR606249_subset25_2.trim2.fq.gz   https://osf.io/zbcrx/download
SRR606249_subset10_1.trim2.fq.gz   https://osf.io/ksu3e/download
SRR606249_subset10_2.trim2.fq.gz   https://osf.io/k9tqn/download
```

This can be turned into download commands using a shell script (cut and xargs) or using a Python script.

```
import subprocess

with open('trimmed.data','r') as f:
    for ln in f.readlines():
        line = ln.split()
        cmd = ["wget",line[1],"-O",line[0]]
        print("Calling command %s"%(" ".join(cmd)))
        subprocess.call(cmd)
```

Next, calculate signatures for our data:

```
sourmashurl="quay.io/biocontainers/sourmash:2.0.0a3--py36_0"

for filename in *_1.trim2.fq.gz
do
    #Remove _1.trim.fq from file name to create base
    base=$(basename $filename _1.trim2.fq.gz)
    sigsuffix=".trim2.scaled10k.k21_31_51.sig"
    echo $base

    if [ -f ${base}${sigsuffix} ]
    then
        # skip
        echo "Skipping file ${base}${sigsuffix}, file exists."
    else
        docker run \
            -v ${PWD}:/data \
            ${sourmashurl} \
            sourmash compute \
            --merge /data/${base}.trim2.fq.gz \
            --track-abundance \
            --scaled 10000 \
            -k 21,31,51 \
            /data/${base}_1.trim2.fq.gz \
            /data/${base}_2.trim2.fq.gz \
            -o /data/${base}${sigsuffix}
    fi
done

for filename in *_1.trim30.fq.gz
do
    #Remove _1.trim.fq from file name to create base
    base=$(basename $filename _1.trim30.fq.gz)
    sigsuffix=".trim30.scaled10k.k21_31_51.sig"
    echo $base

    if [ -f ${base}${sigsuffix} ]
    then
        # skip
        echo "Skipping file ${base}${sigsuffix}, file exists."
    else
        docker run \
            -v ${PWD}:/data \
            ${sourmashurl} \
            sourmash compute \
            --merge /data/${base}.trim30.fq.gz \
            --track-abundance \
            --scaled 10000 \
            -k 21,31,51 \
            /data/${base}_1.trim30.fq.gz \
            /data/${base}_2.trim30.fq.gz \
            -o /data/${base}${sigsuffix}
    fi
done
```

And compare those signatures to our database to classify the components.

```
sourmashurl="quay.io/biocontainers/sourmash:2.0.0a3--py36_0"
for kmer_len in 21 31 51
do
    for sig in *sig
    do
        docker run \
            -v ${PWD}:/data \
            ${sourmashurl} \
            sourmash gather \
            -k ${kmer_len} \
            ${sig} \
            genbank-k${kmer_len}.sbt.json \
            refseq-k${kmer_len}.sbt.json \
            -o ${sig}.k${kmer_len}.gather.output.csv \
            --output-unassigned ${sig}.k${kmer_len}gather_unassigned.output.csv \
            --save-matches ${sig}.k${kmer_len}.gather_matches
    done
done
```

Now, let’s download and unpack the kaiju database (this takes about 15 minutes on my machine):

```
kaijudir="${PWD}/kaijudb"
tarfile="kaiju_index_nr_euk.tgz"

mkdir ${kaijudir}
curl -LO "http://kaiju.binf.ku.dk/database/${tarfile}"
tar xzf ${tarfile}
rm -f ${tarfile}
```

and then link the data and run kaiju:

```
for filename in *1.trim2.fq.gz
do
    #Remove _1.trim2.fq from file name to create base
    base=$(basename $filename _1.trim2.fq.gz)
    echo $base

    # Command to run container interactively:
```

```
    #docker run -it --rm -v ${PWD}:/data quay.io/biocontainers/kaiju:1.6.1--pl5.22.0_0 /bin/

    docker run \
        -v ${PWD}:/data \
        quay.io/biocontainers/kaiju:1.6.1--pl5.22.0_0 \
        kaiju \
        -x \
        -v \
        -t /data/kaijudb/nodes.dmp \
        -f /data/kaijudb/kaiju_db_nr_euk.fmi \
        -i /data/${base}_1.trim2.fq.gz \
        -j /data/${base}_2.trim2.fq.gz \
        -o /data/${base}.kaiju_output.trim2.out \
        -z 4
done

for filename in *1.trim30.fq.gz
do
    #Remove _1.trim30.fq from file name to create base
    base=$(basename $filename _1.trim30.fq.gz)
    echo $base

    docker run \
        -v ${PWD}:/data \
        quay.io/biocontainers/kaiju:1.6.1--pl5.22.0_0 \
        kaiju \
        -x \
        -v \
        -t /data/kaijudb/nodes.dmp \
        -f data/kaijudb/kaiju_db_nr_euk.fmi \
        -i /data/${base}_1.trim30.fq.gz \
        -j /data/${base}_2.trim30.fq.gz \
        -o /data/${base}.kaiju_output.trim30.out \
        -z 4
done
```

Next, convert kaiju output to format readable by krona. Note that the taxonomic rank for classification (e.g. genus) is determined with the -r flag.

```
kaijuurl="quay.io/biocontainers/kaiju:1.6.1--pl5.22.0_0"
kaijudir="kaijudb"
for i in *trim{"2","30"}.out
do
    docker run \
        -v ${PWD}:/data \
        ${kaijuurl} \
        kaiju2krona \
        -v \
        -t \
        /data/${kaijudir}/nodes.dmp \
        -n /data/${kaijudir}/names.dmp \
        -i /data/${i} \
        -o /data/${i}.kaiju.out.krona
done

for i in *trim{"2","30"}.out
do
    docker run \
        -v ${PWD}:/data \
        ${kaijuurl} \
        kaijuReport \
        -v \
        -t \
        /data/${kaijudir}/nodes.dmp \
        -n /data/${kaijudir}/names.dmp \
        -i /data/${i} \
        -r genus \
        -o /data/${i}.kaiju_out_krona.summary
done
```

Now let's filter out taxa with low abundances by obtaining genera that comprise at least 1 percent of the total reads:

```
kaijuurl="quay.io/biocontainers/kaiju:1.6.1--pl5.22.0_0"
for i in *trim{"2","30"}.out
do
    docker run \
        -v ${PWD}:/data \
        ${kaijuurl} \
        kaijuReport
        -v \
        -t /data/kaijudb/nodes.dmp \
        -n /data/kaijudb/names.dmp \
        -i /data/${i} \
        -r genus \
        -m 1 \
        -o /data/${i}.kaiju_out_krona.1percenttotal.summary
done
```

Now for comparison, let's take the genera that comprise at least 1 percent of all of the classified reads:

```
for i in *trim{"2","30"}.out
do
    docker run \
        -v ${PWD}:/data \
        ${kaijuurl} \
        kaijuReport \
        -v \
        -t /data/kaijudb/nodes.dmp \
        -n /data/kaijudb/names.dmp \
        -i /data/${i} \
        -r genus \
        -m 1 \
        -u \
        -o /data/${i}.kaiju_out_krona.1percentclassified.summary
done
```

Download the krona image from quay.io so we can visualize the results from kaiju:

```
kaijudir="${PWD}/kaijudb"
suffix="kaiju_out_krona"
kronaurl="quay.io/biocontainers/krona:2.7--pl5.22.0_1"
docker pull ${kronaurl}
```

Generate krona html with output from all of the reads:

```
suffix="kaiju_out_krona"
for i in *${suffix}.summary
do
        docker run \
            -v ${kaijudir}:/data \
            ${kronaurl} \
            ktImportText \
            -o /data/${i}.${suffix}.html \
            /data/${i}
done
```

Generate krona html with output from genera at least 1 percent of the total reads:

```
suffix="kaiju_out_krona.1percenttotal"
for i in *${suffix}.summary
do
        docker run \
            -v ${kaijudir}:/data \
            ${kronaurl} \
            ktImportText \
            -o /data/${i}.${suffix}.html \
            /data/${i}
done
```

Generate krona html with output from genera at least 1 percent of all classified reads:

```
suffix="kaiju_out_krona.1percentclassified"
for i in *${suffix}.summary
do
        docker run \
            -v ${kaijudir}:/data \
            ${kronaurl} \
            ktImportText \
            -o /data/${i}.${suffix}.html \
            /data/${i}
done
```

Snakemake

Taxonomic Classification: Snakemake

As mentioned on the Running Workflows page, the Snakemake workflows define **build rules** that trigger all of the rules composing a given workflow.

As a reminder, the Running Workflows page showed how to call Snakemake and ask for a particular target:

```
snakemake [FLAGS] <target>
```

You can replace `<target>` with any of the build rules below.

Build Rules

The build rules are the rules that the end user should be calling. A list of available build rules in the taxonomic classification workflow is given below.

```
taxonomic_classification_signatures_workflow

    Build rule: trigger calculation of signatures from reads.

taxonomic_classification_gather_workflow

    Gather and compare read signatures using sourmash gather

taxonomic_classification_kaijureport_workflow

    Run kaiju and generate a report from all results.

taxonomic_classification_kaijureport_filtered_workflow

    Run kaiju and generate a report from filtered
    results (taxa with <X% abundance).

taxonomic_classification_kaijureport_filteredclass_workflow

    Run kaiju and generate a report from filtered, classified
    results (taxa with <X% abundance).
```

Pass the name of the build rule directly to Snakemake on the command line:

```
snakemake [FLAGS] taxonomic_classification_kaijureport_workflow \
        taxonomic_classification_kaijureport_filtered_workflow \
        taxonomic_classification_kaijureport_filteredclass_workflow
```

See below for how to configure these workflows. See the Quick Start for details on the process of running this workflow.

Snakemake Configuration Dictionary

There are three types of key-value pairs that can be set in the Snakemake configuration dictionary (also see the Workflow Configuration page).

Data Files Configuration

Set the `files` key to a dictionary containing a list of key-value pairs, where the keys are filenames and values are URLs:

```
{
    "files" : {
        "SRR606249_1_reads.fq.gz" :           "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_2_reads.fq.gz" :           "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset10_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset25_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset25_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset50_1_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
        "SRR606249_subset50_2_reads.fq.gz" :  "files.osf.io/v1/resources/dm938/providers/osf
    }
}
```

Put these in a JSON file (e.g., `config/custom_datafiles.json` in the `workflows` directory) and pass the name of the config file to Snakemake using the `--configfile` flag:

```
snakemake --configfile=config/custom_datafiles.json [FLAGS] <target>
```

Workflow Configuration

Set the `workflows` key of the Snakemake configuration dictionary to a dictionary containing details about the workflow you want to run. The workflow configuration values are values that will end up in the final output file's filename.

Here is the structure of the configuration dictionary for taxonomic classification workflows:

```
{
    "workflows" : {
        "taxonomic_classification_workflow" : {
            "sample"  : ["SRR606249_subset10","SRR606249_subset25"],
            "qual" : ["2","30"],
        },

        "taxonomic_classification_signatures_workflow" : {
            "sample"  : ["SRR606249_subset10","SRR606249_subset25"],
            "qual" : ["2","30"],
        },

        "taxonomic_classification_gather_workflow" : {
            "sample"  : ["SRR606249_subset10","SRR606249_subset25"],
            "qual" : ["2","30"],
            "kvalues" : ["21","31","51"]
        },

        "taxonomic_classification_kaijureport_workflow" : {
            "sample"  : ["SRR606249_subset10","SRR606249_subset25"],
            "qual" : ["2","30"],
        },

        "taxonomic_classification_kaijureport_filtered_workflow" : {
            "sample"  : ["SRR606249_subset10","SRR606249_subset25"],
            "qual" : ["2","30"],
        },

        "taxonomic_classification_kaijureport_filteredclass_workflow" : {
            "sample"  : ["SRR606249_subset10","SRR606249_subset25"],
            "qual" : ["2","30"],
        }
    }
}
```

Put these in a JSON file (e.g., `config/custom_workflowconfig.json` in the `workflows` directory) and pass the name of the config file to Snakemake using the `--configfile` flag:

```
snakemake --configfile=config/custom_workflowconfig.json [FLAGS] <target>
```

Workflow Parameters

Set the `taxonomic_classification` key of the Snakemake configuration dictionary as shown below:

```
{
    "taxonomic_classification" : {

        "filter_taxa" : {
            "pct_threshold" : 1
        },

        "kaiju" : {
            "dmp1" : "nodes.dmp",
            "dmp2" : "names.dmp",
            "fmi"  : "kaiju_db_nr.fmi",
            "tar"  : "kaiju_index_nr.tgz",
            "url"  : "http://kaiju.binf.ku.dk/database",
            "out"  : "{sample}.kaiju_output.trim{qual}.out"
        },

        "kaiju_report" : {
            "taxonomic_rank" : "genus",
            "pct_threshold"  : 1
        },

        "sourmash" : {
            "sbturl"  : "s3-us-west-1.amazonaws.com/spacegraphcats.ucdavis.edu",
            "sbttar"  : "microbe-{database}-sbt-k{kvalue}-2017.05.09.tar.gz",
            "sbtunpack" : "{database}-k{kvalue}.sbt.json",
            "databases" : ["genbank","refseq"],
            "gather_csv_out"        : "{sample}-k{kvalue}.trim{qual}.gather_output.csv",
            "gather_unassigned_out" : "{sample}-k{kvalue}.trim{qual}.gather_unassigned.csv",
            "gather_matches_out"    : "{sample}-k{kvalue}.trim{qual}.gather_matches.csv"
        },

        "visualize_krona" : {
            "input_summary"  : "{sample}.kaiju_output.trim{qual}.summary",
        }
    }
}
```

To use custom values for these parameters, put the configuration dictionary above (or any subset of it) into a JSON file (e.g., `config/custom_workflowparams.json` in the `workflows` directory) and pass the name of the config file to Snakemake using the `--configfile` flag:

```
snakemake --configfile=config/custom_workflowparams.json [FLAGS] <target>
```

Contributing

How to contribute

This document was inspired by the khmer project getting started documentation.

This document is for people that want to contribute to the dahak metagenomics project. It walks contributors through getting started with GitHub, creating an issue, claiming an issue, making a pull request, and where to store data and newly created Docker images.

Quickstart

Dahak is open source, open data project and we welcome contributions at all levels. We encourage you to submit issues, suggest documentation changes, contribute code, images, workflows etc. Any software included in the workflows must be open source, findable, and reusable. Check out Getting Started, analyze some data, and contribute however you see fit.

How to get started

1. Create a GitHub account.
2. Fork https://github.com/dahak-metagenomics/dahak.

   Visit that page, and then click 'Fork' in the upper right-hand corner. This creates a copy of the dahak source code in your GitHub account. If you're new to GitHub or want a refresher, please check out this awesome tutorial.
3. Clone a copy of dahak into your local environment (or work in your browser!).

   Your shell command should look something like (you can click the 'clone or download' button on the dahak github, copy the link, and insert git clone before it):

   ```
   git clone https://github.com/dahak-metagenomics/dahak.git
   ```

   This makes a copy of the dahak repository on your local machine.

Claiming an issue and starting to develop

1. Find an open issue and claim it.

   Go to the list open issues and claim one you like. Once you've found an issue you like, open it, click 'Assignees', and assign yourself. Once you've assigned yourself make a comment below the issue saying "I'm working on this." That's it; it's all yours. Well not really, because you can always ask for help. Just make another comment below stating what you need some help with and we'll get right back to you.

(Staking your claim is super important because we're trying to avoid people working on the same issue.)

2. In your local copy of the source code, update your master branch from the main dahak branch.

   ```
   git checkout master
   git pull dahak master
   ```

   (This pulls in the latest changes from the master repository)

   If git complains about 'merge conflicts' when you execute `git pull`, please refer to the ***Resolving merge conflicts*** section of the khmer documentation.

   If you are asked to make changes before your pull request can be accepted, you can continue to commit additional changes to the branch associated with your original pull request. The pull request will automatically be updated each time you commit to that branch. Github provides a medium for communicating and providing feedback. Once the pull request is approved, it will be merged into the main branch by the dahak development team.

3. Create a new branch and link it to your fork on GitHub:

   ```
   git checkout -b name-of-branch
   git push -u origin name-of-branch
   ```

   where you replace "name-of-branch" with 2-3 words separated by dashes or underscores describing what issue it fixes.

4. Make some changes and commit them to your branch.

   After you've made a set of cohesive changes, run the command `git status`. This will display a list of all the files git has noticed you changed. Files in the 'untracked' section are files that weren't in the repository before but git has seen.

   To commit these changes you have to 'stage' them using the following command.

   ```
   git add path/to/file
   ```

   Once you've staged your changes, it's time to make a commit (Don't forget to change path/to/file to the actual path to file):

   ```
   git commit -m 'Provide a brief description of the changes you made here'
   ```

   Please make your commit message informative but concise — these messages become a part of the history of the repo and an informative message will help track down changes later. Don't stress over this too much, but before you press the button, please consider whether you will find this commit message useful when a bug pops up 6 months from now and you need to sort through issues to find the right one. Once your changes have been commited, push them to the remote branch:

```
git push origin
```

5. As you develop, please periodically update your branch with changes that have been made to the master branch, and resolve any conflicts that come up.

   ```
   git pull master
   ```

6. Repeat until your ready to commit to master
7. Set up a 'Pull Request' asking to merge your changes into the master dahak repository

   In a web browser, go to your GitHub fork of dahak, e.g.:

   ```
   https://github.com/your-github-username/dahak
   ```

   and you will see a list of 'recently pushed branches' just above the source code listing. On the right side, there should be a green 'Compare and pull request' button. This will add a pull request submission checklist in the following form:

   ```
   Merge Checklist
    - Typos are a sign of poorly maintained code. Has this request been checked with a spe
    - Tutorials should be universally reproducible. If this request modifies a tutorial, d
    - Large diffs to binary or data files can artificially inflate the size of the reposit
   ```

   Next, click "Create Pull Request". This creates a new issue where others can review and make suggestions before your code is added the master dahak repository.

8. Review the pull request checklist and make changes, if necessary.

   Check off as many boxes as possible and make a comment if you need help. If you have an ORCID ID, please add that as a comment. Dahak is an open-source, community-driven project and we'd like to acknowledge your contribution when we publish. Including your ORCID ID helps that process move smoothly.

   As you add new changes, you can keep pushing to your pull request using `git push origin`.

9. When you're ready to have the pull request reviewed, please mention @brooksph, @charlesreid1, @kternus, @stephenturner, @ctb or anyone else on the list of collaborators plus the comment `ready for review`. Often pull requests will require changes, need more work before they can be merged, or simply need to be addressed later. Adding tags can help with organizing. Check out this list for some examples of tags.

10. Once your issue has been reviewed an merged, stand-up, throw your hands in the air, and do a little dance; you're officially a GitHub master and a contributor to the dahak project – we hold you in the highest of regards.

My pull request was merged. What now?

Before continuing on your journey towards your next pull request, there are a couple of steps that you need take to clean up your local copy of dahak.

```
git checkout master
git pull master
git branch -d my-branch-name     # delete the branch locally
git push origin :my-branch-name  # delete the branch on GitHub (your fork)
```

I have a dataset to contribute to the project

Great! A big part of this project is benchmarking tools to determine when and how we should use them. Datasets with interesting composition help us uncover new and interesting things about metagenomics tools. If you have a dataset that you would like to benchmark and/or submit for benchmarking please create an issue and mention @brooksph, @kternus, or @ctb. In general, we'll advise you to make the data publicly available and go crazy characterizing it. We can help you think about the best way to do it but in general we're using the workflows in the workflows/ repository and analyzing the data in Jupyter notebooks. Poke us and we'd be happy to discuss the process. If your dataset is less than 5 GB in size, the open science framework is a great, free place to put it.

I have a tool to contribute to the project

Greater! The more tools the better. A major goal of this project to make more tools easy to use. We opted to do this by using or creating containerized tools. The biocontainers project is leading the way in this effort and we've contributed a few things there. You're not required to contribute to the biocontainers project to add a tool to this project but the tool must be open source and the image must be stored in a public repository like Docker hub or quay.io. Here's an excellent guide to getting started building containers. We're using Singularity to run the containers in our workflows. Docker does not need to be installed.

I have a workflow to contribute to the project

Greatest! New metagenomics analysis tools are created all the time. We're using a small subset that we think encompasses most the methods that are most commonly used to probe metagenomic communities. If you want to include a new tool or workflow, create an issue and we can point you in the right direction. The critical bits are we're stringing together open-source, containerized tools to make workflows using Snakemake. Take a look here for a basic example and here for a bit more flavor. This is a work in progress but the second example is where we're headed.

Contributor Code of Conduct

As contributors and maintainers of this project, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, or religion.

Examples of unacceptable behavior by participants include the use of sexual language or imagery, derogatory comments or personal attacks, trolling, public or private harassment, insults, or other unprofessional conduct.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct. Project maintainers who do not follow the Code of Conduct may be removed from the project team.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by emailing Phil Brooks (ptbrooks@ucdavis.edu) or C. Titus Brown (ctbrown@ucdavis.edu). To report an issue involving either of them, please email Judi Brown Clarke (jbc@egr.msu.edu), Ph.D. the Diversity Director at the BEACON Center for the Study of Evolution in Action, an NSF Center for Science and Technology.

This Code of Conduct is adapted from the Contributor Covenant, version 1.0.0, available from http://contributor-covenant.org/version/1/0/0/