Toil Documentation

Release 3.23.1

UCSC Computational Genomics Lab

Feb 07, 2020

Getting Started

1 Installation **3**
- 1.1 Preparing Your Python Runtime Environment . . . 3
- 1.2 Basic Installation . . . 4
- 1.3 Installing Toil with Extra Features . . . 4
- 1.4 Building from Source . . . 5

2 Quickstart Examples **7**
- 2.1 Running a basic workflow . . . 7
- 2.2 Running a basic CWL workflow . . . 8
- 2.3 Running a basic WDL workflow . . . 9
- 2.4 A (more) real-world example . . . 9
- 2.5 Launching a Toil Workflow in AWS . . . 15
- 2.6 Running a CWL Workflow on AWS . . . 16
- 2.7 Running a Workflow with Autoscaling - Cactus . . . 17

3 Introduction **21**
- 3.1 Job Store . . . 21
- 3.2 Batch System . . . 22
- 3.3 Provisioner . . . 22

4 Commandline Options **23**
- 4.1 The Job Store . . . 23
- 4.2 Commandline Options . . . 23
- 4.3 Restart Option . . . 28
- 4.4 Running Workflows with Services . . . 28
- 4.5 Setting Options directly with the Toil Script . . . 28

5 Toil Debugging **31**
- 5.1 Introspecting the Jobstore . . . 31
- 5.2 Stats and Status . . . 31
- 5.3 Using a Python debugger . . . 32

6 Running in the Cloud **33**
- 6.1 Managing a Cluster of Virtual Machines (Provisioning) . . . 33
- 6.2 Storage (Toil jobStore) . . . 33

7 Cloud Platforms **35**

7.1 Running in AWS . . . 35
7.2 Running in Google Compute Engine (GCE) . . . 41
7.3 Cluster Utilities . . . 44
7.4 Stats Command . . . 45
7.5 Status Command . . . 47
7.6 Clean Command . . . 47
7.7 Launch-Cluster Command . . . 47
7.8 Ssh-Cluster Command . . . 49
7.9 Rsync-Cluster Command . . . 49
7.10 Destroy-Cluster Command . . . 50
7.11 Kill Command . . . 50

8 HPC Environments **51**
8.1 Standard Output/Error from Batch System Jobs . . . 51

9 CWL in Toil **53**
9.1 Running CWL Locally . . . 53
9.2 Detailed Usage Instructions . . . 53
9.3 Running CWL in the Cloud . . . 54
9.4 Running CWL within Toil Scripts . . . 55
9.5 Toil & CWL Tips . . . 56

10 WDL in Toil **61**
10.1 How to Run a WDL file in Toil . . . 61
10.2 ENCODE Example from ENCODE-DCC . . . 61
10.3 GATK Examples from the Broad . . . 62
10.4 toilwdl.py Options . . . 63
10.5 Running WDL within Toil Scripts . . . 63
10.6 WDL Specifications . . . 64

11 Developing a Workflow **65**
11.1 Scripting Quick Start . . . 65
11.2 Job Basics . . . 66
11.3 Invoking a Workflow . . . 66
11.4 Specifying Commandline Arguments . . . 67
11.5 Resuming a Workflow . . . 68
11.6 Functions and Job Functions . . . 68
11.7 Workflows with Multiple Jobs . . . 69
11.8 Dynamic Job Creation . . . 71
11.9 Promises . . . 71
11.10 Promised Requirements . . . 73
11.11 FileID . . . 74
11.12 Managing files within a workflow . . . 74
11.13 Using Docker Containers in Toil . . . 77
11.14 Services . . . 78
11.15 Checkpoints . . . 79
11.16 Encapsulation . . . 80
11.17 Depending on Toil . . . 81
11.18 Best Practices for Dockerizing Toil Workflows . . . 81

12 Toil Class API **83**

13 Job Store API **85**

14 Toil Job API **93**

14.1 FunctionWrappingJob . . . 93
14.2 JobFunctionWrappingJob . . . 93
14.3 EncapsulatedJob . . . 94
14.4 Promise . . . 96

15 Job Methods API **99**

16 Job.Runner API **105**

17 job.fileStore API **107**

18 Batch System API **111**
18.1 Batch System Enivronmental Variables . . . 111
18.2 Batch System API . . . 112

19 Job.Service API **115**

20 Exceptions API **117**

21 Running Tests **119**
21.1 Running Tests with pytest . . . 120
21.2 Running Integration Tests . . . 120
21.3 Test Environment Variables . . . 120
21.4 Using Docker with Quay . . . 121
21.5 Running Mesos Tests . . . 121

22 Developing with Docker **123**
22.1 Making Your Own Toil Docker Image . . . 123
22.2 Running a Cluster Locally . . . 124

23 Maintainer's Guidelines **127**
23.1 Naming Conventions . . . 127
23.2 Pull Requests . . . 128

24 Toil Architecture **129**
24.1 Optimizations . . . 131
24.2 Toil support for Common Workflow Language . . . 132

25 Minimum AWS IAM permissions **135**

26 Auto-Deployment **137**
26.1 Auto Deployment with Sibling Modules . . . 138
26.2 Auto-Deploying a Package Hierarchy . . . 139
26.3 Relying on Shared Filesystems . . . 140

27 Environment Variables **141**

Index **145**

Toil is an open-source pure-Python workflow engine that lets people write better pipelines.

Check out our website for a comprehensive list of Toil's features and read our paper to learn what Toil can do in the real world. Please subscribe to our low-volume announce mailing list and feel free to also join us on GitHub and Gitter.

If using Toil for your research, please cite

> Vivian, J., Rao, A. A., Nothaft, F. A., Ketchum, C., Armstrong, J., Novak, A., ... Paten, B. (2017). Toil enables reproducible, open source, big biomedical data analyses. Nature Biotechnology, 35(4), 314–316. http://doi.org/10.1038/nbt.3772

CHAPTER 1

Installation

This document describes how to prepare for and install Toil. Note that Toil requires that the user run all commands inside of a Python virtualenv. Instructions for installing and creating a Python virtual environment are provided below.

1.1 Preparing Your Python Runtime Environment

Toil currently supports Python 2.7, 3.5, and 3.6, and requires a virtualenv to be active to install.

If not already present, please install the latest Python `virtualenv` using pip:

```
$ sudo pip install virtualenv
```

And create a virtual environment called `venv` in your home directory:

```
$ virtualenv ~/venv
```

If the user does not have root privileges, there are a few more steps, but one can download a specific virtualenv package directly, untar the file, create, and source the virtualenv (version 15.1.0 as an example) using

```
$ curl -O https://pypi.python.org/packages/d4/0c/
↪9840c08189e030873387a73b90ada981885010dd9aea134d6de30cd24cb8/virtualenv-15.1.0.tar.
↪gz
$ tar xvfz virtualenv-15.1.0.tar.gz
$ cd virtualenv-15.1.0
$ python virtualenv.py ~/venv
```

Now that you've created your virtualenv, activate your virtual environment:

```
$ source ~/venv/bin/activate
```

1.2 Basic Installation

If you need only the basic version of Toil, it can be easily installed using pip:

```
$ pip install toil
```

Now you're ready to run *your first Toil workflow*!

(If you need any of the extra features don't do this yet and instead skip to the next section.)

1.3 Installing Toil with Extra Features

Python headers and static libraries

Needed for the `mesos`, `aws`, `google`, and `encryption` extras.

On Ubuntu:

```
$ sudo apt-get install build-essential python-dev
```

On macOS:

```
$ xcode-select --install
```

Encryption specific headers and library

Needed for the `encryption` extra.

On Ubuntu:

```
$ sudo apt-get install libssl-dev libffi-dev
```

On macOS:

```
$ brew install libssl libffi
```

Or see Cryptography for other systems.

Some optional features, called *extras*, are not included in the basic installation of Toil. To install Toil with all its bells and whistles, first install any necessary headers and libraries (*python-dev*, *libffi-dev*). Then run

```
$ pip install toil[aws,mesos,google,encryption,cwl]
```

or

```
$ pip install toil[all]
```

Here's what each extra provides:

Extra	Description
`all`	Installs all extras (though htcondor is linux-only and will be skipped if not on a linux computer).
`aws`	Provides support for managing a cluster on Amazon Web Service (AWS) using Toil's built in *Cluster Utilities*. Clusters can scale up and down automatically. It also supports storing workflow state.
`google`	Experimental. Stores workflow state in Google Cloud Storage.

`mesos`	Provides support for running Toil on an Apache Mesos cluster. Note that running Toil on other batch systems does not require an extra. The `mesos` extra requires the following native dependencies:
	• Apache Mesos (Tested with Mesos v1.0.0)
	• *Python headers and static libraries*
	--- 	
	Important: If launching toil remotely on a mesos instance, to install Toil with the `mesos` extra in a virtualenv, be sure to create that virtualenv with the `--system-site-packages` flag (only use remotely!):
	`$ virtualenv ~/venv --system-site-packages`
	Otherwise, you'll see something like this:
	`ImportError: No module named mesos.native`
	--- 	

`htcondor`	Support for the htcondor batch system. This currently is a linux only extra.
`encryption`	Provides client-side encryption for files stored in the AWS job store. This extra requires the following native dependencies: • *Python headers and static libraries* • *libffi headers and library*
`cwl`	Provides support for running workflows written using the Common Workflow Language.
`wdl`	Provides support for running workflows written using the Workflow Description Language. This extra has no native dependencies.

1.4 Building from Source

If developing with Toil, you will need to build from source. This allows changes you make to Toil to be reflected immediately in your runtime environment.

First, clone the source:

```
$ git clone https://github.com/DataBiosphere/toil.git
$ cd toil
```

Then, create and activate a virtualenv:

```
$ virtualenv venv
$ . venv/bin/activate
```

From there, you can list all available Make targets by running `make`. First and foremost, we want to install Toil's build requirements (these are additional packages that Toil needs to be tested and built but not to be run):

```
$ make prepare
```

Now, we can install Toil in development mode (such that changes to the source code will immediately affect the virtualenv):

```
$ make develop
```

Or, to install with support for all optional *Installing Toil with Extra Features*:

```
$ make develop extras=[aws,mesos,google,encryption,cwl]
```

Or:

```
$ make develop extras=[all]
```

To build the docs, run `make develop` with all extras followed by

```
$ make docs
```

To run a quick batch of tests (this should take less than 30 minutes) run

```
$ export TOIL_TEST_QUICK=True; make test
```

For more information on testing see *Running Tests*.

CHAPTER 2

Quickstart Examples

2.1 Running a basic workflow

A Toil workflow can be run with just three steps:

1. Install Toil (see *Installation*)
2. Copy and paste the following code block into a new file called `helloWorld.py`:

```
from toil.common import Toil
from toil.job import Job

def helloWorld(message, memory="1G", cores=1, disk="1G"):
    return "Hello, world!, here's a message: %s" % message

if __name__ == "__main__":
    parser = Job.Runner.getDefaultArgumentParser()
    options = parser.parse_args()
    options.clean = "always"
    with Toil(options) as toil:
        output = toil.start(Job.wrapFn(helloWorld, "You did it!"))
    print(output)
```

3. Specify the name of the *job store* and run the workflow:

```
(venv) $ python helloWorld.py file:my-job-store
```

Note: Don't actually type `(venv) $` in at the beginning of each command. This is intended only to remind the user that they should have their *virtual environment* running.

Congratulations! You've run your first Toil workflow using the default *Batch System*, `singleMachine`, using the `file` job store.

Toil uses batch systems to manage the jobs it creates.

The `singleMachine` batch system is primarily used to prepare and debug workflows on a local machine. Once validated, try running them on a full-fledged batch system (see *Batch System API*). Toil supports many different batch systems such as Apache Mesos and Grid Engine; its versatility makes it easy to run your workflow in all kinds of places.

Toil is totally customizable! Run `python helloWorld.py --help` to see a complete list of available options.

For something beyond a "Hello, world!" example, refer to *A (more) real-world example*.

2.2 Running a basic CWL workflow

The Common Workflow Language (CWL) is an emerging standard for writing workflows that are portable across multiple workflow engines and platforms. Running CWL workflows using Toil is easy.

1. First ensure that Toil is installed with the `cwl` extra (see *Installing Toil with Extra Features*):

   ```
   (venv) $ pip install 'toil[cwl]'
   ```

   This installs the `toil-cwl-runner` and `cwltoil` executables.

2. Copy and paste the following code block into `example.cwl`:

   ```
   cwlVersion: v1.0
   class: CommandLineTool
   baseCommand: echo
   stdout: output.txt
   inputs:
     message:
       type: string
       inputBinding:
         position: 1
   outputs:
     output:
       type: stdout
   ```

   and this code into `example-job.yaml`:

   ```
   message: Hello world!
   ```

3. To run the workflow simply enter

   ```
   (venv) $ toil-cwl-runner example.cwl example-job.yaml
   ```

   Your output will be in `output.txt`:

   ```
   (venv) $ cat output.txt
   Hello world!
   ```

To learn more about CWL, see the CWL User Guide (from where this example was shamelessly borrowed).

To run this workflow on an AWS cluster have a look at *Running a CWL Workflow on AWS*.

For information on using CWL with Toil see the section *CWL in Toil*

2.3 Running a basic WDL workflow

The Workflow Description Language (WDL) is another emerging language for writing workflows that are portable across multiple workflow engines and platforms. Running WDL workflows using Toil is still in alpha, and currently experimental. Toil currently supports basic workflow syntax (see *WDL in Toil* for more details and examples). Here we go over running a basic WDL helloworld workflow.

1. First ensure that Toil is installed with the `wdl` extra (see *Installing Toil with Extra Features*):

```
(venv) $ pip install 'toil[wdl]'
```

   This installs the `toil-wdl-runner` executable.

2. Copy and paste the following code block into `wdl-helloworld.wdl`:

```
    workflow write_simple_file {
      call write_file
    }
    task write_file {
      String message
      command { echo ${message} > wdl-helloworld-output.txt }
      output { File test = "wdl-helloworld-output.txt" }
    }

and this code into ``wdl-helloworld.json``::

    {
      "write_simple_file.write_file.message": "Hello world!"
    }
```

3. To run the workflow simply enter

```
(venv) $ toil-wdl-runner wdl-helloworld.wdl wdl-helloworld.json
```

   Your output will be in `wdl-helloworld-output.txt`:

```
(venv) $ cat wdl-helloworld-output.txt
Hello world!
```

To learn more about WDL, see the main WDL website .

2.4 A (more) real-world example

For a more detailed example and explanation, we've developed a sample pipeline that merge-sorts a temporary file. This is not supposed to be an efficient sorting program, rather a more fully worked example of what Toil is capable of.

2.4.1 Running the example

1. Download `the example code`
2. Run it with the default settings:

```
(venv) $ python sort.py file:jobStore
```

The workflow created a file called `sortedFile.txt` in your current directory. Have a look at it and notice that it contains a whole lot of sorted lines!

This workflow does a smart merge sort on a file it generates, `fileToSort.txt`. The sort is *smart* because each step of the process—splitting the file into separate chunks, sorting these chunks, and merging them back together—is compartmentalized into a **job**. Each job can specify its own resource requirements and will only be run after the jobs it depends upon have run. Jobs without dependencies will be run in parallel.

Note: Delete `fileToSort.txt` before moving on to #3. This example introduces options that specify dimensions for `fileToSort.txt`, if it does not already exist. If it exists, this workflow will use the existing file and the results will be the same as #2.

---

3. Run with custom options:

```
(venv) $ python sort.py file:jobStore --numLines=5000 --lineLength=10 --
↪overwriteOutput=True --workDir=/tmp/
```

Here we see that we can add our own options to a Toil script. As noted above, the first two options, `--numLines` and `--lineLength`, determine the number of lines and how many characters are in each line. `--overwriteOutput` causes the current contents of `sortedFile.txt` to be overwritten, if it already exists. The last option, `--workDir`, is an option built into Toil to specify where temporary files unique to a job are kept.

2.4.2 Describing the source code

To understand the details of what's going on inside. Let's start with the `main()` function. It looks like a lot of code, but don't worry—we'll break it down piece by piece.

```
def main(options=None):
    if not options:
        # deal with command line arguments
        parser = ArgumentParser()
        Job.Runner.addToilOptions(parser)
        parser.add_argument('--numLines', default=defaultLines, help='Number of lines␣
↪in file to sort.', type=int)
        parser.add_argument('--lineLength', default=defaultLineLen, help='Length of␣
↪lines in file to sort.', type=int)
        parser.add_argument("--fileToSort", help="The file you wish to sort")
        parser.add_argument("--outputFile", help="Where the sorted output will go")
        parser.add_argument("--overwriteOutput", help="Write over the output file if␣
↪it already exists.", default=True)
        parser.add_argument("--N", dest="N",
                            help="The threshold below which a serial sort function is␣
↪used to sort file. "
                                 "All lines must of length less than or equal to N or␣
↪program will fail",
                            default=10000)
        parser.add_argument('--downCheckpoints', action='store_true',
                            help='If this option is set, the workflow will make␣
↪checkpoints on its way through'
                                 'the recursive "down" part of the sort')
        parser.add_argument("--sortMemory", dest="sortMemory",
                            help="Memory for jobs that sort chunks of the file.",
                            default=None)
```

(continues on next page)

(continued from previous page)

```
        parser.add_argument("--mergeMemory", dest="mergeMemory",
                            help="Memory for jobs that collate results.",
                            default=None)

        options = parser.parse_args()
    if not hasattr(options, "sortMemory") or not options.sortMemory:
        options.sortMemory = sortMemory
    if not hasattr(options, "mergeMemory") or not options.mergeMemory:
        options.mergeMemory = sortMemory

    # do some input verification
    sortedFileName = options.outputFile or "sortedFile.txt"
    if not options.overwriteOutput and os.path.exists(sortedFileName):
        print("the output file {} already exists. Delete it to run the sort example␣
↪again or use --overwriteOutput=True".format(sortedFileName))
        exit()

    fileName = options.fileToSort
    if options.fileToSort is None:
        # make the file ourselves
        fileName = 'fileToSort.txt'
        if os.path.exists(fileName):
            print("Sorting existing file: {}".format(fileName))
        else:
            print('No sort file specified. Generating one automatically called: {}.'.
↪format(fileName))
            makeFileToSort(fileName=fileName, lines=options.numLines, lineLen=options.
↪lineLength)
    else:
        if not os.path.exists(options.fileToSort):
            raise RuntimeError("File to sort does not exist: %s" % options.fileToSort)

    if int(options.N) <= 0:
        raise RuntimeError("Invalid value of N: %s" % options.N)

    # Now we are ready to run
    with Toil(options) as workflow:
        sortedFileURL = 'file://' + os.path.abspath(sortedFileName)
        if not workflow.options.restart:
            sortFileURL = 'file://' + os.path.abspath(fileName)
            sortFileID = workflow.importFile(sortFileURL)
            sortedFileID = workflow.start(Job.wrapJobFn(setup, sortFileID,␣
↪int(options.N), options.downCheckpoints, options=options,
                                                        memory=sortMemory))
        else:
            sortedFileID = workflow.restart()
        workflow.exportFile(sortedFileID, sortedFileURL)
```

First we make a parser to process command line arguments using the argparse module. It's important that we add the call to `Job.Runner.addToilOptions()` to initialize our parser with all of Toil's default options. Then we add the command line arguments unique to this workflow, and parse the input. The help message listed with the arguments should give you a pretty good idea of what they can do.

Next we do a little bit of verification of the input arguments. The option `--fileToSort` allows you to specify a file that needs to be sorted. If this option isn't given, it's here that we make our own file with the call to `makeFileToSort()`.

Finally we come to the context manager that initializes the workflow. We create a path to the input file prepended with

`'file://'` as per the documentation for *toil.common.Toil()* when staging a file that is stored locally. Notice that we have to check whether or not the workflow is restarting so that we don't import the file more than once. Finally we can kick off the workflow by calling *toil.common.Toil.start()* on the job `setup`. When the workflow ends we capture its output (the sorted file's fileID) and use that in *toil.common.Toil.exportFile()* to move the sorted file from the job store back into "userland".

Next let's look at the job that begins the actual workflow, `setup`.

```
def setup(job, inputFile, N, downCheckpoints, options):
    """
    Sets up the sort.
    Returns the FileID of the sorted file
    """
    RealtimeLogger.info("Starting the merge sort")
    return job.addChildJobFn(down,
                             inputFile, N, 'root',
                             downCheckpoints,
                             options = options,
                             preemptable=True,
                             memory=sortMemory).rv()
```

`setup` really only does two things. First it writes to the logs using `Job.log()` and then calls `addChildJobFn()`. Child jobs run directly after the current job. This function turns the 'job function' `down` into an actual job and passes in the inputs including an optional resource requirement, `memory`. The job doesn't actually get run until the call to `Job.rv()`. Once the job `down` finishes, its output is returned here.

Now we can look at what `down` does.

```
def down(job, inputFileStoreID, N, path, downCheckpoints, options, memory=sortMemory):
    """
    Input is a file, a subdivision size N, and a path in the hierarchy of jobs.
    If the range is larger than a threshold N the range is divided recursively and
    a follow on job is then created which merges back the results else
    the file is sorted and placed in the output.
    """

    RealtimeLogger.info("Down job starting: %s" % path)

    # Read the file
    inputFile = job.fileStore.readGlobalFile(inputFileStoreID, cache=False)
    length = os.path.getsize(inputFile)
    if length > N:
        # We will subdivide the file
        RealtimeLogger.critical("Splitting file: %s of size: %s"
                % (inputFileStoreID, length))
        # Split the file into two copies
        midPoint = getMidPoint(inputFile, 0, length)
        t1 = job.fileStore.getLocalTempFile()
        with open(t1, 'w') as fH:
            fH.write(copySubRangeOfFile(inputFile, 0, midPoint+1))
        t2 = job.fileStore.getLocalTempFile()
        with open(t2, 'w') as fH:
            fH.write(copySubRangeOfFile(inputFile, midPoint+1, length))
        # Call down recursively. By giving the rv() of the two jobs as inputs to the
↪follow-on job, up,
        # we communicate the dependency without hindering concurrency.
        result = job.addFollowOnJobFn(up,
                                      job.addChildJobFn(down, job.fileStore.
↪writeGlobalFile(t1), N, path + '/0',
```

(continues on next page)

(continued from previous page)

```
                                                    downCheckpoints,␣
↪checkpoint=downCheckpoints, options=options,
                                                    preemptable=True,␣
↪memory=options.sortMemory).rv(),
                                    job.addChildJobFn(down, job.fileStore.
↪writeGlobalFile(t2), N, path + '/1',
                                                    downCheckpoints,␣
↪checkpoint=downCheckpoints, options=options,
                                                    preemptable=True,␣
↪memory=options.mergeMemory).rv(),
                                    path + '/up', preemptable=True, options=options,␣
↪memory=options.sortMemory).rv()
    else:
        # We can sort this bit of the file
        RealtimeLogger.critical("Sorting file: %s of size: %s"
                % (inputFileStoreID, length))
        # Sort the copy and write back to the fileStore
        shutil.copyfile(inputFile, inputFile + '.sort')
        sort(inputFile + '.sort')
        result = job.fileStore.writeGlobalFile(inputFile + '.sort')

    RealtimeLogger.info("Down job finished: %s" % path)
    return result
```

Down is the recursive part of the workflow. First we read the file into the local filestore by calling `job.fileStore.readGlobalFile()`. This puts a copy of the file in the temp directory for this particular job. This storage will disappear once this job ends. For a detailed explanation of the filestore, job store, and their interfaces have a look at *Managing files within a workflow*.

Next `down` checks the base case of the recursion: is the length of the input file less than `N` (remember `N` was an option we added to the workflow in `main`)? In the base case, we just sort the file, and return the file ID of this new sorted file.

If the base case fails, then the file is split into two new tempFiles using `job.fileStore.getLocalTempFile()` and the helper function `copySubRangeOfFile`. Finally we add a follow on Job `up` with `job.addFollowOnJobFn()`. We've already seen child jobs. A follow-on Job is a job that runs after the current job and *all* of its children (and their children and follow-ons) have completed. Using a follow-on makes sense because `up` is responsible for merging the files together and we don't want to merge the files together until we *know* they are sorted. Again, the return value of the follow-on job is requested using `Job.rv()`.

Looking at `up`

```
def up(job, inputFileID1, inputFileID2, path, options, memory=sortMemory):
    """
    Merges the two files and places them in the output.
    """

    RealtimeLogger.info("Up job starting: %s" % path)

    with job.fileStore.writeGlobalFileStream() as (fileHandle, outputFileStoreID):
        fileHandle = codecs.getwriter('utf-8')(fileHandle)
        with job.fileStore.readGlobalFileStream(inputFileID1) as inputFileHandle1:
            inputFileHandle1 = codecs.getreader('utf-8')(inputFileHandle1)
            with job.fileStore.readGlobalFileStream(inputFileID2) as inputFileHandle2:
                inputFileHandle2 = codecs.getreader('utf-8')(inputFileHandle2)
                RealtimeLogger.info("Merging %s and %s to %s"
                    % (inputFileID1, inputFileID2, outputFileStoreID))
```

(continues on next page)

(continued from previous page)

```
                merge(inputFileHandle1, inputFileHandle2, fileHandle)
        # Cleanup up the input files - these deletes will occur after the completion␣
↪is successful.
        job.fileStore.deleteGlobalFile(inputFileID1)
        job.fileStore.deleteGlobalFile(inputFileID2)

        RealtimeLogger.info("Up job finished: %s" % path)

        return outputFileStoreID
```

we see that the two input files are merged together and the output is written to a new file using `job.fileStore.writeGlobalFileStream()`. After a little cleanup, the output file is returned.

Once the final `up` finishes and all of the `rv()` promises are fulfilled, `main` receives the sorted file's ID which it uses in `exportFile` to send it to the user.

There are other things in this example that we didn't go over such as *Checkpoints* and the details of much of the *Toil Class API*.

At the end of the script the lines

```
if __name__ == '__main__'
    main()
```

are included to ensure that the main function is only run once in the '__main__' process invoked by you, the user. In Toil terms, by invoking the script you created the *leader process* in which the `main()` function is run. A *worker process* is a separate process whose sole purpose is to host the execution of one or more jobs defined in that script. In any Toil workflow there is always one leader process, and potentially many worker processes.

When using the single-machine batch system (the default), the worker processes will be running on the same machine as the leader process. With full-fledged batch systems like Mesos the worker processes will typically be started on separate machines. The boilerplate ensures that the pipeline is only started once—on the leader—but not when its job functions are imported and executed on the individual workers.

Typing `python sort.py --help` will show the complete list of arguments for the workflow which includes both Toil's and ones defined inside `sort.py`. A complete explanation of Toil's arguments can be found in *Commandline Options*.

2.4.3 Logging

By default, Toil logs a lot of information related to the current environment in addition to messages from the batch system and jobs. This can be configured with the `--logLevel` flag. For example, to only log `CRITICAL` level messages to the screen:

```
(venv) $ python sort.py file:jobStore --logLevel=critical --overwriteOutput=True
```

This hides most of the information we get from the Toil run. For more detail, we can run the pipeline with `--logLevel=debug` to see a comprehensive output. For more information, see *Commandline Options*.

2.4.4 Error Handling and Resuming Pipelines

With Toil, you can recover gracefully from a bug in your pipeline without losing any progress from successfully completed jobs. To demonstrate this, let's add a bug to our example code to see how Toil handles a failure and how we can resume a pipeline after that happens. Add a bad assertion at line 52 of the example (the first line of `down()`):

```
def down(job, inputFileStoreID, N, downCheckpoints, memory=sortMemory):
    ...
    assert 1 == 2, "Test error!"
```

When we run the pipeline, Toil will show a detailed failure log with a traceback:

```
(venv) $ python sort.py file:jobStore
...
---TOIL WORKER OUTPUT LOG---
...
m/j/jobonrSMP    Traceback (most recent call last):
m/j/jobonrSMP      File "toil/src/toil/worker.py", line 340, in main
m/j/jobonrSMP        job._runner(jobGraph=jobGraph, jobStore=jobStore,␣
↪fileStore=fileStore)
m/j/jobonrSMP      File "toil/src/toil/job.py", line 1270, in _runner
m/j/jobonrSMP        returnValues = self._run(jobGraph, fileStore)
m/j/jobonrSMP      File "toil/src/toil/job.py", line 1217, in _run
m/j/jobonrSMP        return self.run(fileStore)
m/j/jobonrSMP      File "toil/src/toil/job.py", line 1383, in run
m/j/jobonrSMP        rValue = userFunction(*((self,) + tuple(self._args)), **self._
↪kwargs)
m/j/jobonrSMP      File "toil/example.py", line 30, in down
m/j/jobonrSMP        assert 1 == 2, "Test error!"
m/j/jobonrSMP    AssertionError: Test error!
```

If we try and run the pipeline again, Toil will give us an error message saying that a job store of the same name already exists. By default, in the event of a failure, the job store is preserved so that the workflow can be restarted, starting from the previously failed jobs. We can restart the pipeline by running

```
(venv) $ python sort.py file:jobStore --restart --overwriteOutput=True
```

We can also change the number of times Toil will attempt to retry a failed job:

```
(venv) $ python sort.py file:jobStore --retryCount 2 --restart --overwriteOutput=True
```

You'll now see Toil attempt to rerun the failed job until it runs out of tries. `--retryCount` is useful for non-systemic errors, like downloading a file that may experience a sporadic interruption, or some other non-deterministic failure.

To successfully restart our pipeline, we can edit our script to comment out line 30, or remove it, and then run

```
(venv) $ python sort.py file:jobStore --restart --overwriteOutput=True
```

The pipeline will run successfully, and the job store will be removed on the pipeline's completion.

2.4.5 Collecting Statistics

Please see the *Stats Command* section for more on gathering runtime and resource info on jobs.

2.5 Launching a Toil Workflow in AWS

After having installed the `aws` extra for Toil during the *Installation* and set up AWS (see *Preparing your AWS environment*), the user can run the basic `helloWorld.py` script (*Running a basic workflow*) on a VM in AWS just by modifying the run command.

Note that when running in AWS, users can either run the workflow on a single instance or run it on a cluster (which is running across multiple containers on multiple AWS instances). For more information on running Toil workflows on a cluster, see *Running in AWS*.

Also! Remember to use the *Destroy-Cluster Command* command when finished to destroy the cluster! Otherwise things may not be cleaned up properly.

1. Launch a cluster in AWS using the *Launch-Cluster Command* command:

```
(venv) $ toil launch-cluster <cluster-name> --keyPairName <AWS-key-pair-name> --
↪leaderNodeType t2.medium --zone us-west-2a
```

   The arguments `keyPairName`, `leaderNodeType`, and `zone` are required to launch a cluster.

2. Copy `helloWorld.py` to the `/tmp` directory on the leader node using the *Rsync-Cluster Command* command:

```
(venv) $ toil rsync-cluster --zone us-west-2a <cluster-name> helloWorld.py :/tmp
```

   Note that the command requires defining the file to copy as well as the target location on the cluster leader node.

3. Login to the cluster leader node using the *Ssh-Cluster Command* command:

```
(venv) $ toil ssh-cluster --zone us-west-2a <cluster-name>
```

   Note that this command will log you in as the `root` user.

4. Run the Toil script in the cluster:

```
$ python /tmp/helloWorld.py aws:us-west-2:my-S3-bucket
```

   In this particular case, we create an S3 bucket called `my-S3-bucket` in the `us-west-2` availability zone to store intermediate job results.

   Along with some other `INFO` log messages, you should get the following output in your terminal window: `Hello, world!, here's a message: You did it!.`

5. Exit from the SSH connection.

```
$ exit
```

6. Use the *Destroy-Cluster Command* command to destroy the cluster:

```
(venv) $ toil destroy-cluster --zone us-west-2a <cluster-name>
```

   Note that this command will destroy the cluster leader node and any resources created to run the job, including the S3 bucket.

2.6 Running a CWL Workflow on AWS

After having installed the `aws` and `cwl` extras for Toil during the *Installation* and set up AWS (see *Preparing your AWS environment*), the user can run a CWL workflow with Toil on AWS.

Also! Remember to use the *Destroy-Cluster Command* command when finished to destroy the cluster! Otherwise things may not be cleaned up properly.

1. First launch a node in AWS using the *Launch-Cluster Command* command:

```
(venv) $ toil launch-cluster <cluster-name> --keyPairName <AWS-key-pair-name> --
↪leaderNodeType t2.medium --zone us-west-2a
```

2. Copy `example.cwl` and `example-job.yaml` from the *CWL example* to the node using the *Rsync-Cluster Command* command:

```
(venv) $ toil rsync-cluster --zone us-west-2a <cluster-name> example.cwl :/tmp
(venv) $ toil rsync-cluster --zone us-west-2a <cluster-name> example-job.yaml :/
↪tmp
```

3. SSH into the cluster's leader node using the *Ssh-Cluster Command* utility:

```
(venv) $ toil ssh-cluster --zone us-west-2a <cluster-name>
```

4. Once on the leader node, it's a good idea to update and install the following:

```
sudo apt-get update
sudo apt-get -y upgrade
sudo apt-get -y dist-upgrade
sudo apt-get -y install git
sudo pip install mesos.cli
```

5. Now create a new `virtualenv` with the `--system-site-packages` option and activate:

```
virtualenv --system-site-packages venv
source venv/bin/activate
```

6. Now run the CWL workflow:

```
(venv) $ toil-cwl-runner --provisioner aws --jobStore aws:us-west-2a:any-name /
↪tmp/example.cwl /tmp/example-job.yaml
```

Tip: When running a CWL workflow on AWS, input files can be provided either on the local file system or in S3 buckets using `s3://` URI references. Final output files will be copied to the local file system of the leader node.

---

7. Finally, log out of the leader node and from your local computer, destroy the cluster:

```
(venv) $ toil destroy-cluster --zone us-west-2a <cluster-name>
```

2.7 Running a Workflow with Autoscaling - Cactus

Cactus is a reference-free, whole-genome multiple alignment program that can be run on any of the cloud platforms Toil supports.

Note: Cloud Independence:

This example provides a "cloud agnostic" view of running Cactus with Toil. Most options will not change between cloud providers. However, each provisioner has unique inputs for `--leaderNodeType`, `--nodeType` and `--zone`. We recommend the following:

Option	Used in	AWS	Google
`--leaderNodeType`	launch-cluster	t2.medium	n1-standard-1
`--zone`	launch-cluster	us-west-2a	us-west1-a
`--zone`	cactus	us-west-2	
`--nodeType`	cactus	c3.4xlarge	n1-standard-8

When executing `toil launch-cluster` with `gce` specified for `--provisioner`, the option `--boto` must be specified and given a path to your .boto file. See *Running in Google Compute Engine (GCE)* for more information about the `--boto` option.

Also! Remember to use the *Destroy-Cluster Command* command when finished to destroy the cluster! Otherwise things may not be cleaned up properly.

1. Download `pestis.tar.gz`
2. Launch a leader node using the *Launch-Cluster Command* command:

```
(venv) $ toil launch-cluster <cluster-name> --provisioner <aws, gce> --
↪keyPairName <key-pair-name> --leaderNodeType <type> --zone <zone>
```

Note: A Helpful Tip

When using AWS, setting the environment variable eliminates having to specify the `--zone` option for each command. This will be supported for GCE in the future.

```
(venv) $ export TOIL_AWS_ZONE=us-west-2c
```

---

3. Create appropriate directory for uploading files:

```
(venv) $ toil ssh-cluster --provisioner <aws, gce> <cluster-name>
$ mkdir /root/cact_ex
$ exit
```

4. Copy the required files, i.e., seqFile.txt (a text file containing the locations of the input sequences as well as their phylogenetic tree, see here), organisms' genome sequence files in FASTA format, and configuration files (e.g. blockTrim1.xml, if desired), up to the leader node:

```
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> pestis-short-
↪aws-seqFile.txt :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> GCF_000169655.
↪1_ASM16965v1_genomic.fna :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> GCF_000006645.
↪1_ASM664v1_genomic.fna :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> GCF_000182485.
↪1_ASM18248v1_genomic.fna :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> GCF_000013805.
↪1_ASM1380v1_genomic.fna :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> setup_
↪leaderNode.sh :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> blockTrim1.
↪xml :/root/cact_ex
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name> blockTrim3.
↪xml :/root/cact_ex
```

5. Log in to the leader node:

```
(venv) $ toil ssh-cluster --provisioner <aws, gce> <cluster-name>
```

6. Set up the environment of the leader node to run Cactus:

```
$ bash /root/cact_ex/setup_leaderNode.sh
$ source cact_venv/bin/activate
(cact_venv) $ cd cactus
(cact_venv) $ pip install --upgrade .
```

7. Run Cactus as an autoscaling workflow:

```
(cact_venv) $ TOIL_APPLIANCE_SELF=quay.io/ucsc_cgl/toil:3.14.0 cactus --
↪provisioner <aws, gce> --nodeType <type> --maxNodes 2 --minNodes 0 --retry 10 --
↪batchSystem mesos --disableCaching --logDebug --logFile /logFile_pestis3 --
↪configFile /root/cact_ex/blockTrim3.xml <aws, google>:<zone>:cactus-pestis /
↪root/cact_ex/pestis-short-aws-seqFile.txt /root/cact_ex/pestis_output3.hal
```

Note: Pieces of the Puzzle:

`TOIL_APPLIANCE_SELF=quay.io/ucsc_cgl/toil:3.14.0` — specifies the version of Toil being used, 3.14.0; if the latest one is desired, please eliminate.

`--nodeType` — determines the instance type used for worker nodes. The instance type specified here must be on the same cloud provider as the one specified with `--leaderNodeType`

`--maxNodes 2` — creates up to two instances of the type specified with `--nodeType` and launches Mesos worker containers inside them.

`--logDebug` — equivalent to `--logLevel DEBUG`.

`--logFile /logFile_pestis3` — writes logs in a file named *logFile_pestis3* under `/` folder.

`--configFile` — this is not required depending on whether a specific configuration file is intended to run the alignment.

`<aws, google>:<zone>:cactus-pestis` — creates a bucket, named `cactus-pestis`, with the specified cloud provider to store intermediate job files and metadata. **NOTE**: If you want to use a GCE-based jobstore, specify `google` here, not `gce`.

The result file, named `pestis_output3.hal`, is stored under `/root/cact_ex` folder of the leader node.

Use `cactus --help` to see all the Cactus and Toil flags available.

---

8. Log out of the leader node:

```
(cact_venv) $ exit
```

9. Download the resulted output to local machine:

```
(venv) $ toil rsync-cluster --provisioner <aws, gce> <cluster-name>  :/root/cact_
↪ex/pestis_output3.hal <path-of-folder-on-local-machine>
```

10. Destroy the cluster:

```
(venv) $ toil destroy-cluster --provisioner <aws, gce> <cluster-name>
```

CHAPTER 3

Introduction

Toil runs in various environments, including *locally* and *in the cloud* (Amazon Web Services and Google Compute Engine). Toil also supports two DSLs: *CWL* and (Amazon Web Services and Google Compute Engine). Toil also supports two DSLs: *CWL* and *WDL* (experimental).

Toil is built in a modular way so that it can be used on lots of different systems, and with different configurations. The three configurable pieces are the

- *Job Store API*: A filepath or url that can host and centralize all files for a workflow (e.g. a local folder, or an AWS s3 bucket url).
- *Batch System API*: Specifies either a local single-machine or a currently supported HPC environment (lsf, parasol, mesos, slurm, torque, htcondor, or gridengine). Mesos is a special case, and is launched for cloud environments.
- *Provisioner*: For running in the cloud only. This specifies which cloud provider provides instances to do the "work" of your workflow.

3.1 Job Store

The job store is a storage abstraction which contains all of the information used in a Toil run. This centralizes all of the files used by jobs in the workflow and also the details of the progress of the run. If a workflow crashes or fails, the job store contains all of the information necessary to resume with minimal repetition of work.

Several different job stores are supported, including the file job store and cloud job stores.

3.1.1 File Job Store

The file job store is for use locally, and keeps the workflow information in a directory on the machine where the workflow is launched. This is the simplest and most convenient job store for testing or for small runs.

For an example that uses the file job store, see *Running a basic workflow*.

3.1.2 Cloud Job Stores

Toil currently supports the following cloud storage systems as job stores:

- *AWS Job Store*: An AWS S3 bucket formatted as "aws:<zone>:<bucketname>" where only numbers, letters, and dashes are allowed in the bucket name. Example: *aws:us-west-2:my-aws-jobstore-name*.
- *Google Job Store*: A Google Cloud Storage bucket formatted as "gce:<zone>:<bucketname>" where only numbers, letters, and dashes are allowed in the bucket name. Example: *gce:us-west2-a:my-google-jobstore-name*.

These use cloud buckets to house all of the files. This is useful if there are several different worker machines all running jobs that need to access the job store.

3.2 Batch System

A Toil batch system is either a local single-machine (one computer) or a currently supported HPC cluster of computers (lsf, parasol, mesos, slurm, torque, htcondor, or gridengine). Mesos is a special case, and is launched for cloud environments. These environments manage individual worker nodes under a leader node to process the work required in a workflow. The leader and its workers all coordinate their tasks and files through a centralized job store location.

See *Batch System API* for a more detailed description of different batch systems.

3.3 Provisioner

The Toil provisioner provides a tool set for running a Toil workflow on a particular cloud platform.

The *Cluster Utilities* are command line tools used to provision nodes in your desired cloud platform. They allows you to launch nodes, ssh to the leader, and rsync files back and forth.

For detailed instructions for using the provisioner see *Running in AWS* or *Running in Google Compute Engine (GCE)*.

CHAPTER 4

Commandline Options

A quick way to see all of Toil's commandline options is by executing the following on a toil script:

```
$ toil example.py --help
```

For a basic toil workflow, Toil has one mandatory argument, the job store. All other arguments are optional.

4.1 The Job Store

Running toil scripts requires a filepath or url to a centralizing location for all of the files of the workflow. This is Toil's one required positional argument: the job store. To use the *quickstart* example, if you're on a node that has a large **/scratch** volume, you can specify that the jobstore be created there by executing: `python HelloWorld.py /scratch/my-job-store`, or more explicitly, `python HelloWorld.py file:/scratch/my-job-store`.

Syntax for specifying different job stores:

Local: `file:job-store-name`

AWS: `aws:region-here:job-store-name`

Google: `google:projectID-here:job-store-name`

Different types of job store options can be found below.

4.2 Commandline Options

Core Toil Options

--workDir WORKDIR Absolute path to directory where temporary files generated during the Toil run should be placed. Temp files and folders, as well as standard output and error from batch system jobs (unless –noStdOutErr), will be placed in a directory toil-<workflowID> within

workDir. The workflowID is generated by Toil and will be reported in the workflow logs. Default is determined by the variables (TMPDIR, TEMP, TMP) via mkdtemp. This directory needs to exist on all machines running jobs; if capturing standard output and error from batch system jobs is desired, it will generally need to be on a shared file system.

--noStdOutErr Do not capture standard output and error from batch system jobs.

--stats Records statistics about the toil workflow to be used by 'toil stats'.

--clean=STATE Determines the deletion of the jobStore upon completion of the program. Choices: 'always', 'onError','never', or 'onSuccess'. The –stats option requires information from the jobStore upon completion so the jobStore will never be deleted with that flag. If you wish to be able to restart the run, choose 'never' or 'onSuccess'. Default is 'never' if stats is enabled, and 'onSuccess' otherwise

--cleanWorkDir STATE Determines deletion of temporary worker directory upon completion of a job. Choices: 'always', 'never', 'onSuccess'. Default = always. WARNING: This option should be changed for debugging only. Running a full pipeline with this option could fill your disk with intermediate data.

--clusterStats FILEPATH If enabled, writes out JSON resource usage statistics to a file. The default location for this file is the current working directory, but an absolute path can also be passed to specify where this file should be written. This option only applies when using scalable batch systems.

--restart If –restart is specified then will attempt to restart existing workflow at the location pointed to by the –jobStore option. Will raise an exception if the workflow does not exist.

Logging Options

Toil hides stdout and stderr by default except in case of job failure. Log levels in toil are based on priority from the logging module:

--logOff Only CRITICAL log levels are shown. Equivalent to `--logLevel=OFF` or `--logLevel=CRITICAL`.

--logCritical Only CRITICAL log levels are shown. Equivalent to `--logLevel=OFF` or `--logLevel=CRITICAL`.

--logError Only ERROR, and CRITICAL log levels are shown. Equivalent to `--logLevel=ERROR`.

--logWarning Only WARN, ERROR, and CRITICAL log levels are shown. Equivalent to `--logLevel=WARNING`.

--logInfo All log statements are shown, except DEBUG. Equivalent to `--logLevel=INFO`.

--logDebug All log statements are shown. Equivalent to `--logLevel=DEBUG`.

--logLevel=LOGLEVEL May be set to: `OFF` (or `CRITICAL`), `ERROR`, `WARN` (or `WARNING`), `INFO`, or `DEBUG`.

--logFile FILEPATH Specifies a file path to write the logging output to.

--rotatingLogging Turn on rotating logging, which prevents log files from getting too big (set using `--maxLogFileSize BYTESIZE`).

--maxLogFileSize BYTESIZE Sets the maximum log file size in bytes (`--rotatingLogging` must be active).

Batch System Options

--batchSystem BATCHSYSTEM The type of batch system to run the job(s) with, currently can be one of LSF, Mesos, Slurm, Torque, HTCondor, singleMachine, parasol, gridEngine'. (default: singleMachine)

--parasolCommand PARASOLCOMMAND The name or path of the parasol program. Will be looked up on PATH unless it starts with a slash. (default: parasol)

--parasolMaxBatches PARASOLMAXBATCHES Maximum number of job batches the Parasol batch is allowed to create. One batch is created for jobs with a unique set of resource requirements. (default: 1000)

--scale SCALE A scaling factor to change the value of all submitted tasks' submitted cores. Used in singleMachine batch system. (default: 1)

--linkImports When using Toil's importFile function for staging, input files are copied to the job store. Specifying this option saves space by symlinking imported files. As long as caching is enabled Toil will protect the file automatically by changing the permissions to read-only.

--mesosMaster MESOSMASTERADDRESS The host and port of the Mesos master separated by a colon. (default: 169.233.147.202:5050)

Autoscaling Options

--provisioner CLOUDPROVIDER The provisioner for cluster auto-scaling. The currently supported choices are 'aws' or 'gce'. The default is None.

--nodeTypes NODETYPES List of node types separated by commas. The syntax for each node type depends on the provisioner used. For the cgcloud and AWS provisioners this is the name of an EC2 instance type, optionally followed by a colon and the price in dollars to bid for a spot instance of that type, for example 'c3.8xlarge:0.42'. If no spot bid is specified, nodes of this type will be non-preemptable. It is acceptable to specify an instance as both preemptable and non-preemptable, including it twice in the list. In that case, preemptable nodes of that type will be preferred when creating new nodes once the maximum number of preemptable-nodes have been reached.

--nodeOptions NODEOPTIONS Options for provisioning the nodes. The syntax depends on the provisioner used. Neither the CGCloud nor the AWS provisioner support any node options.

--minNodes MINNODES Minimum number of nodes of each type in the cluster, if using auto-scaling. This should be provided as a comma-separated list of the same length as the list of node types. default=0

--maxNodes MAXNODES Maximum number of nodes of each type in the cluster, if using autoscaling, provided as a comma-separated list. The first value is used as a default if the list length is less than the number of nodeTypes. default=10

--preemptableCompensation PREEMPTABLECOMPENSATION The preference of the autoscaler to replace preemptable nodes with non-preemptable nodes, when preemptable nodes cannot be started for some reason. Defaults to 0.0. This value must be between 0.0 and 1.0, inclusive. A value of 0.0 disables such compensation, a value of 0.5 compensates two missing preemptable nodes with a non-preemptable one. A value of 1.0 replaces every missing pre-emptable node with a non-preemptable one.

--nodeStorage NODESTORAGE Specify the size of the root volume of worker nodes when they are launched in gigabytes. You may want to set this if your jobs require a lot of disk space. The default value is 50.

--metrics Enable the prometheus/grafana dashboard for monitoring CPU/RAM usage, queue size, and issued jobs.

--defaultMemory INT The default amount of memory to request for a job. Only applicable to jobs that do not specify an explicit value for this requirement. Standard suffixes like K, Ki, M, Mi, G or Gi are supported. Default is 2.0G

--defaultCores FLOAT The default number of CPU cores to dedicate a job. Only applicable to jobs that do not specify an explicit value for this requirement. Fractions of a core (for example 0.1) are supported on some batch systems, namely Mesos and singleMachine. Default is 1.0

--defaultDisk INT The default amount of disk space to dedicate a job. Only applicable to jobs that do not specify an explicit value for this requirement. Standard suffixes like K, Ki, M, Mi, G or Gi are supported. Default is 2.0G

--maxCores INT The maximum number of CPU cores to request from the batch system at any one time. Standard suffixes like K, Ki, M, Mi, G or Gi are supported.

--maxMemory INT The maximum amount of memory to request from the batch system at any one time. Standard suffixes like K, Ki, M, Mi, G or Gi are supported.

--maxDisk INT The maximum amount of disk space to request from the batch system at any one time. Standard suffixes like K, Ki, M, Mi, G or Gi are supported.

--retryCount RETRYCOUNT Number of times to retry a failing job before giving up and labeling job failed. default=1

--maxJobDuration MAXJOBDURATION Maximum runtime of a job (in seconds) before we kill it (this is a lower bound, and the actual time before killing the job may be longer).

--rescueJobsFrequency RESCUEJOBSFREQUENCY Period of time to wait (in seconds) between checking for missing/overlong jobs, that is jobs which get lost by the batch system.

--maxServiceJobs MAXSERVICEJOBS The maximum number of service jobs that can be run concurrently, excluding service jobs running on preemptable nodes. default=9223372036854775807

--maxPreemptableServiceJobs MAXPREEMPTABLESERVICEJOBS The maximum number of service jobs that can run concurrently on

preemptable nodes. default=9223372036854775807

--deadlockWait DEADLOCKWAIT The minimum number of seconds to observe the cluster stuck running only the same service jobs before throwing a deadlock exception. default=60

--statePollingWait STATEPOLLINGWAIT Time, in seconds, to wait before doing a scheduler query for job state. Return cached results if within the waiting period.

Miscellaneous Options

--disableCaching Disables caching in the file store. This flag must be set to use a batch system that does not support caching such as Grid Engine, Parasol, LSF, or Slurm.

--disableChaining Disables chaining of jobs (chaining uses one job's resource allocation for its successor job if possible).

--maxLogFileSize MAXLOGFILESIZE The maximum size of a job log file to keep (in bytes), log files larger than this will be truncated to the last X bytes. Setting this option to zero will prevent any truncation. Setting this option to a negative value will truncate from the beginning. Default=62.5 K

--writeLogs FILEPATH Write worker logs received by the leader into their own files at the specified path. Any non-empty standard output and error from failed batch system jobs will also be written into files at this path. The current working directory will be used if a path is not specified explicitly. Note: By default only the logs of failed jobs are returned to leader. Set log level to 'debug' to get logs back from successful jobs, and adjust 'maxLogFileSize' to control the truncation limit for worker logs.

--writeLogsGzip FILEPATH Identical to –writeLogs except the logs files are gzipped on the leader.

--realTimeLogging Enable real-time logging from workers to masters.

--sseKey SSEKEY Path to file containing 32 character key to be used for server-side encryption on awsJobStore or googleJobStore. SSE will not be used if this flag is not passed.

--setEnv NAME NAME=VALUE or NAME, -e NAME=VALUE or NAME are also valid. Set an environment variable early on in the worker. If VALUE is omitted, it will be looked up in the current environment. Independently of this option, the worker will try to emulate the leader's environment before running a job. Using this option, a variable can be injected into the worker process itself before it is started.

--servicePollingInterval SERVICEPOLLINGINTERVAL Interval of time service jobs wait between polling for the existence of the keep-alive flag (default=60)

--debugWorker Experimental no forking mode for local debugging. Specifically, workers are not forked and stderr/stdout are not redirected to the log. (default=False)

4.3 Restart Option

In the event of failure, Toil can resume the pipeline by adding the argument `--restart` and rerunning the python script. Toil pipelines can even be edited and resumed which is useful for development or troubleshooting.

4.4 Running Workflows with Services

Toil supports jobs, or clusters of jobs, that run as *services* to other *accessor* jobs. Example services include server databases or Apache Spark Clusters. As service jobs exist to provide services to accessor jobs their runtime is dependent on the concurrent running of their accessor jobs. The dependencies between services and their accessor jobs can create potential deadlock scenarios, where the running of the workflow hangs because only service jobs are being run and their accessor jobs can not be scheduled because of too limited resources to run both simultaneously. To cope with this situation Toil attempts to schedule services and accessors intelligently, however to avoid a deadlock with workflows running service jobs it is advisable to use the following parameters:

- `--maxServiceJobs`: The maximum number of service jobs that can be run concurrently, excluding service jobs running on preemptable nodes.
- `--maxPreemptableServiceJobs`: The maximum number of service jobs that can run concurrently on preemptable nodes.

Specifying these parameters so that at a maximum cluster size there will be sufficient resources to run accessors in addition to services will ensure that such a deadlock can not occur.

If too low a limit is specified then a deadlock can occur in which toil can not schedule sufficient service jobs concurrently to complete the workflow. Toil will detect this situation if it occurs and throw a `toil.DeadlockException` exception. Increasing the cluster size and these limits will resolve the issue.

4.5 Setting Options directly with the Toil Script

It's good to remember that commandline options can be overridden in the Toil script itself. For example, *toil.job.Job.Runner.getDefaultOptions()* can be used to run toil with all default options, and in this example, it will override commandline args to run the default options and always run with the "./toilWorkflow" directory specified as the jobstore:

```
options = Job.Runner.getDefaultOptions("./toilWorkflow") # Get the options object

with Toil(options) as toil:
    toil.start(Job())  # Run the script
```

However, each option can be explicitly set within the script by supplying arguments (in this example, we are setting `logLevel = "DEBUG"` (all log statements are shown) and `clean="ALWAYS"` (always delete the jobstore) like so:

```
options = Job.Runner.getDefaultOptions("./toilWorkflow") # Get the options object
options.logLevel = "DEBUG" # Set the log level to the debug level.
options.clean = "ALWAYS" # Always delete the jobStore after a run

with Toil(options) as toil:
    toil.start(Job())  # Run the script
```

However, the usual incantation is to accept commandline args from the user with the following:

```
parser = Job.Runner.getDefaultArgumentParser() # Get the parser
options = parser.parse_args() # Parse user args to create the options object

with Toil(options) as toil:
    toil.start(Job())  # Run the script
```

Which can also, of course, then accept script supplied arguments as before (which will overwrite any user supplied args):

```
parser = Job.Runner.getDefaultArgumentParser() # Get the parser
options = parser.parse_args() # Parse user args to create the options object
options.logLevel = "DEBUG" # Set the log level to the debug level.
options.clean = "ALWAYS" # Always delete the jobStore after a run

with Toil(options) as toil:
    toil.start(Job())  # Run the script
```

CHAPTER 5

Toil Debugging

Toil has a number of tools to assist in debugging. Here we provide help in working through potential problems that a user might encounter in attempting to run a workflow.

5.1 Introspecting the Jobstore

Note: Currently these features are only implemented for use locally (single machine) with the fileJobStore.

To view what files currently reside in the jobstore, run the following command:

```
$ toil debug-file file:path-to-jobstore-directory --listFilesInJobStore
```

When run from the commandline, this should generate a file containing the contents of the job store (in addition to displaying a series of log messages to the terminal). This file is named "jobstore_files.txt" by default and will be generated in the current working directory.

If one wishes to copy any of these files to a local directory, one can run for example:

```
$ toil debug-file file:path-to-jobstore --fetch overview.txt *.bam *.fastq --
↪localFilePath=/home/user/localpath
```

To fetch `overview.txt`, and all `.bam` and `.fastq` files. This can be used to recover previously used input and output files for debugging or reuse in other workflows, or use in general debugging to ensure that certain outputs were imported into the jobStore.

5.2 Stats and Status

See *Stats Command* for more about gathering statistics about job success, runtime, and resource usage from workflows.

5.3 Using a Python debugger

If you execute a workflow using the `--debugWorker` flag, Toil will not fork in order to run jobs, which means you can either use pdb, or an IDE that supports debugging Python as you would normally. Note that the `--debugWorker` flag will only work with the `singleMachine` batch system (the default), and not any of the custom job schedulers.

CHAPTER 6

Running in the Cloud

Toil supports Amazon Web Services (AWS) and Google Compute Engine (GCE) in the cloud and has autoscaling capabilities that can adapt to the size of your workflow, whether your workflow requires 10 instances or 20,000.

Toil does this by creating a virtual cluster with Apache Mesos. Apache Mesos requires a leader node to coordinate the workflow, and worker nodes to execute the various tasks within the workflow. As the workflow runs, Toil will "autoscale", creating and terminating workers as needed to meet the demands of the workflow.

Once a user is familiar with the basics of running toil locally (specifying a *jobStore*, and how to write a toil script), they can move on to the guides below to learn how to translate these workflows into cloud ready workflows.

6.1 Managing a Cluster of Virtual Machines (Provisioning)

Toil can launch and manage a cluster of virtual machines to run using the *provisioner* to run a workflow distributed over several nodes. The provisioner also has the ability to automatically scale up or down the size of the cluster to handle dynamic changes in computational demand (autoscaling). Currently we have working provisioners with AWS and GCE (Azure support has been deprecated).

Toil uses Apache Mesos as the *Batch System*.

See here for instructions for *Running in AWS*.

See here for instructions for *Running in Google Compute Engine (GCE)*.

6.2 Storage (Toil jobStore)

Toil can make use of cloud storage such as AWS or Google buckets to take care of storage needs.

This is useful when running Toil in single machine mode on any cloud platform since it allows you to make use of their integrated storage systems.

For an overview of the job store see *Job Store*.

For instructions configuring a particular job store see:

- *AWS Job Store*
- *Google Job Store*

CHAPTER 7

Cloud Platforms

7.1 Running in AWS

Toil jobs can be run on a variety of cloud platforms. Of these, Amazon Web Services (AWS) is currently the best-supported solution. Toil provides the *Cluster Utilities* to conveniently create AWS clusters, connect to the leader of the cluster, and then launch a workflow. The leader handles distributing the jobs over the worker nodes and autoscaling to optimize costs.

The *Running a Workflow with Autoscaling* section details how to create a cluster and run a workflow that will dynamically scale depending on the workflow's needs.

The *Static Provisioning* section explains how a static cluster (one that won't automatically change in size) can be created and provisioned (grown, shrunk, destroyed, etc.).

7.1.1 Preparing your AWS environment

To use Amazon Web Services (AWS) to run Toil or to just use S3 to host the files during the computation of a workflow, first set up and configure an account with AWS:

1. If necessary, create and activate an AWS account
2. Only needed once, but AWS requires that users "subscribe" to use the Container Linux by CoreOS AMI. You will encounter errors if this is not done.
3. Next, generate a key pair for AWS with the command (do NOT generate your key pair with the Amazon browser):

```
$ ssh-keygen -t rsa
```

4. This should prompt you to save your key. Please save it in

```
~/.ssh/id_rsa
```

5. Now move this to where your OS can see it as an authorized key:

```
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ eval `ssh-agent -s`
$ ssh-add
```

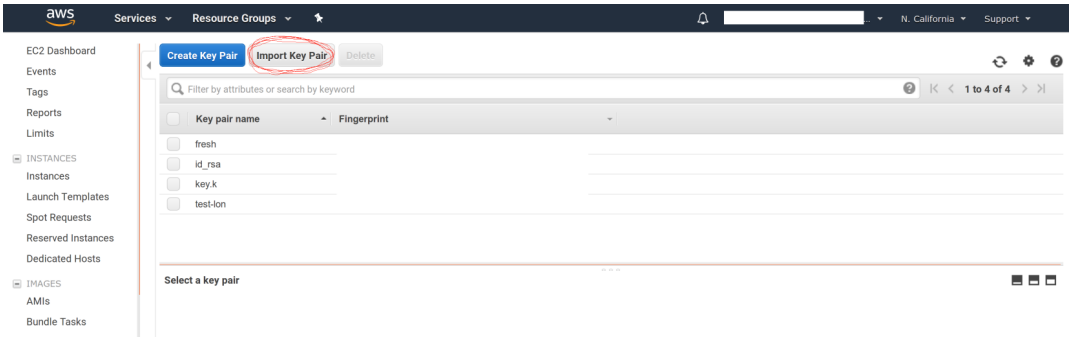
6. You'll also need to chmod your private key (good practice but also enforced by AWS):

```
$ chmod 400 id_rsa
```

7. Now you'll need to add the key to AWS via the browser. For example, on us-west1, this address would accessible at:

```
https://us-west-1.console.aws.amazon.com/ec2/v2/home?region=us-west-1
↪#KeyPairs:sort=keyName
```

8. Now click on the "Import Key Pair" button to add your key:



9. Next, you need to create an AWS access key. First go to the IAM dashboard, again; for "us-west1", the example link would be here:

```
https://console.aws.amazon.com/iam/home?region=us-west-1#/home
```

10. The directions (transcribed from: https://docs.aws.amazon.com/general/latest/gr/managing-aws-access-keys.html) are now:

    (a) On the IAM Dashboard page, choose your account name in the navigation bar, and then choose My Security Credentials.

    (b) Expand the Access keys (access key ID and secret access key) section.

    (c) Choose Create New Access Key. Then choose Download Key File to save the access key ID and secret access key to a file on your computer. After you close the dialog box, you can't retrieve this secret access key again.

11. Now you should have a newly generated "AWS Access Key ID" and "AWS Secret Access Key". We can now install the AWS CLI and make sure that it has the proper credentials:

```
$ pip install awscli --upgrade --user
```

12. Now configure your AWS credentials with:

```
$ aws configure
```

13. Add your "AWS Access Key ID" and "AWS Secret Access Key" from earlier and your region and output format:

```
" AWS Access Key ID [****************Q65Q]: "
" AWS Secret Access Key [****************G0ys]: "
```

(continues on next page)

(continued from previous page)

```
" Default region name [us-west-1]: "
" Default output format [json]: "
```

14. Toil also relies on boto, and you'll need to create a boto file containing your credentials as well. To do this, run:

```
$ nano ~/.boto
```

15. Paste in the following (with your actual "AWS Access Key ID" and "AWS Secret Access Key"):

```
[Credentials]
aws_access_key_id = ****************Q65Q
aws_secret_access_key = ****************G0ys
```

16. If not done already, install toil (example uses version 3.12.0, but we recommend the latest release):

```
$ virtualenv venv
$ source venv/bin/activate
$ pip install toil[all]==3.12.0
```

17. Now that toil is installed and you are running a virtualenv, an example of launching a toil leader node would be the following (again, note that we set TOIL_APPLIANCE_SELF to toil version 3.12.0 in this example, but please set the version to the installed version that you are using if you're using a different version):

```
$ TOIL_APPLIANCE_SELF=quay.io/ucsc_cgl/toil:3.12.0 toil launch-cluster␣
↪clustername --leaderNodeType t2.medium --zone us-west-1a --keyPairName id_rsa
```

To further break down each of these commands:

TOIL_APPLIANCE_SELF=quay.io/ucsc_cgl/toil:latest — This is optional. It specifies a mesos docker image that we maintain with the latest version of toil installed on it. If you want to use a different version of toil, please specify the image tag you need from https://quay.io/repository/ucsc_cgl/toil?tag=latest&tab=tags.

toil launch-cluster — Base command in toil to launch a cluster.

clustername — Just choose a name for your cluster.

–leaderNodeType t2.medium — Specify the leader node type. Make a t2.medium (2CPU; 4Gb RAM; $0.0464/Hour). List of available AWS instances: https://aws.amazon.com/ec2/pricing/on-demand/

–zone us-west-1a — Specify the AWS zone you want to launch the instance in. Must have the same prefix as the zone in your awscli credentials (which, in the example of this tutorial is: "us-west-1").

–keyPairName id_rsa — The name of your key pair, which should be "id_rsa" if you've followed this tutorial.

7.1.2 AWS Job Store

Using the AWS job store is straightforward after you've finished *Preparing your AWS environment*; all you need to do is specify the prefix for the job store name.

To run the sort example *sort example* with the AWS job store you would type

```
$ python sort.py aws:us-west-2:my-aws-sort-jobstore
```

7.1.3 Toil Provisioner

The Toil provisioner is included in Toil alongside the `[aws]` extra and allows us to spin up a cluster.

Getting started with the provisioner is simple:

1. Make sure you have Toil installed with the AWS extras. For detailed instructions see *Installing Toil with Extra Features*.
2. You will need an AWS account and you will need to save your AWS credentials on your local machine. For help setting up an AWS account see here. For setting up your AWS credentials follow instructions here.

The Toil provisioner is built around the Toil Appliance, a Docker image that bundles Toil and all its requirements (e.g. Mesos). This makes deployment simple across platforms, and you can even simulate a cluster locally (see *Developing with Docker* for details).

Choosing Toil Appliance Image

When using the Toil provisioner, the appliance image will be automatically chosen based on the pip-installed version of Toil on your system. That choice can be overridden by setting the environment variables `TOIL_DOCKER_REGISTRY` and `TOIL_DOCKER_NAME` or `TOIL_APPLIANCE_SELF`. See *Environment Variables* for more information on these variables. If you are developing with autoscaling and want to test and build your own appliance have a look at *Developing with Docker*.

For information on using the Toil Provisioner have a look at *Running a Workflow with Autoscaling*.

7.1.4 Details about Launching a Cluster in AWS

Using the provisioner to launch a Toil leader instance is simple using the `launch-cluster` command. For example, to launch a cluster named "my-cluster" with a t2.medium leader in the us-west-2a zone, run

```
(venv) $ toil launch-cluster my-cluster --leaderNodeType t2.medium --zone us-west-2a -
↪-keyPairName <your-AWS-key-pair-name>
```

The cluster name is used to uniquely identify your cluster and will be used to populate the instance's `Name` tag. Also, the Toil provisioner will automatically tag your cluster with an `Owner` tag that corresponds to your keypair name to facilitate cost tracking. In addition, the `ToilNodeType` tag can be used to filter "leader" vs. "worker" nodes in your cluster.

The leaderNodeType is an EC2 instance type. This only affects the leader node.

The `--zone` parameter specifies which EC2 availability zone to launch the cluster in. Alternatively, you can specify this option via the `TOIL_AWS_ZONE` environment variable. Note: the zone is different from an EC2 region. A region corresponds to a geographical area like `us-west-2 (Oregon)`, and availability zones are partitions of this area like `us-west-2a`.

By default, Toil creates an IAM role for each cluster with sufficient permissions to perform cluster operations (e.g. full S3, EC2, and SDB access). If the default permissions are not sufficient for your use case (e.g. if you need access to ECR), you may create a custom IAM role with all necessary permissions and set the `--awsEc2ProfileArn` parameter when launching the cluster. Note that your custom role must at least have *these permissions* in order for the Toil cluster to function properly.

For more information on options try:

```
(venv) $ toil launch-cluster --help
```

Static Provisioning

Toil can be used to manage a cluster in the cloud by using the *Cluster Utilities*. The cluster utilities also make it easy to run a toil workflow directly on this cluster. We call this static provisioning because the size of the cluster does not change. This is in contrast with *Running a Workflow with Autoscaling*.

To launch worker nodes alongside the leader we use the `-w` option:

```
(venv) $ toil launch-cluster my-cluster --leaderNodeType t2.small -z us-west-2a --
↪keyPairName your-AWS-key-pair-name --nodeTypes m3.large,t2.micro -w 1,4
```

This will spin up a leader node of type t2.small with five additional workers — one m3.large instance and four t2.micro.

Currently static provisioning is only possible during the cluster's creation. The ability to add new nodes and remove existing nodes via the native provisioner is in development. Of course the cluster can always be deleted with the *Destroy-Cluster Command* utility.

Uploading Workflows

Now that our cluster is launched, we use the *Rsync-Cluster Command* utility to copy the workflow to the leader. For a simple workflow in a single file this might look like

```
(venv) $ toil rsync-cluster -z us-west-2a my-cluster toil-workflow.py :/
```

Note: If your toil workflow has dependencies have a look at the *Auto-Deployment* section for a detailed explanation on how to include them.

Running a Workflow with Autoscaling

Autoscaling is a feature of running Toil in a cloud whereby additional cloud instances are launched to run the workflow. Autoscaling leverages Mesos containers to provide an execution environment for these workflows.

Note: Make sure you've done the AWS setup in *Preparing your AWS environment*.

---

1. Download `sort.py`
2. Launch the leader node in AWS using the *Launch-Cluster Command* command:

   ```
   (venv) $ toil launch-cluster <cluster-name> --keyPairName <AWS-key-pair-name> --
   ↪leaderNodeType t2.medium --zone us-west-2a
   ```

3. Copy the `sort.py` script up to the leader node:

   ```
   (venv) $ toil rsync-cluster <cluster-name> sort.py :/root
   ```

4. Login to the leader node:

   ```
   (venv) $ toil ssh-cluster <cluster-name>
   ```

5. Run the script as an autoscaling workflow:

   ```
   $ python /root/sort.py aws:us-west-2:<my-jobstore-name> --provisioner aws --
   ↪nodeTypes c3.large --maxNodes 2 --batchSystem mesos
   ```

Note: In this example, the autoscaling Toil code creates up to two instances of type *c3.large* and launches Mesos slave containers inside them. The containers are then available to run jobs defined by the *sort.py* script. Toil also creates a bucket in S3 called *aws:us-west-2:autoscaling-sort-jobstore* to store intermediate job results. The Toil autoscaler can also provision multiple different node types, which is useful for workflows that have jobs with varying resource requirements. For example, one could execute the script with `--nodeTypes c3.large,r3.xlarge --maxNodes 5,1`, which would allow the provisioner to create up to five c3.large nodes and one r3.xlarge node for memory-intensive jobs. In this situation, the autoscaler would avoid creating the more expensive r3.xlarge node until needed, running most jobs on the c3.large nodes.

1. View the generated file to sort:

```
$ head fileToSort.txt
```

2. View the sorted file:

```
$ head sortedFile.txt
```

For more information on other autoscaling (and other) options have a look at *Commandline Options* and/or run

```
$ python my-toil-script.py --help
```

Important: Some important caveats about starting a toil run through an ssh session are explained in the *Ssh-Cluster Command* section.

Preemptability

Toil can run on a heterogeneous cluster of both preemptable and non-preemptable nodes. Being preemptable node simply means that the node may be shut down at any time, while jobs are running. These jobs can then be restarted later somewhere else.

A node type can be specified as preemptable by adding a spot bid to its entry in the list of node types provided with the `--nodeTypes` flag. If spot instance prices rise above your bid, the preemptable node whill be shut down.

While individual jobs can each explicitly specify whether or not they should be run on preemptable nodes via the boolean `preemptable` resource requirement, the `--defaultPreemptable` flag will allow jobs without a `preemptable` requirement to run on preemptable machines.

Specify Preemptability Carefully

Ensure that your choices for `--nodeTypes` and `--maxNodes <>` make sense for your workflow and won't cause it to hang. You should make sure the provisioner is able to create nodes large enough to run the largest job in the workflow, and that non-preemptable node types are allowed if there are non-preemptable jobs in the workflow.

Finally, the `--preemptableCompensation` flag can be used to handle cases where preemptable nodes may not be available but are required for your workflow. With this flag enabled, the autoscaler will attempt to compensate for a shortage of preemptable nodes of a certain type by creating non-preemptable nodes of that type, if non-preemptable nodes of that type were specified in `--nodeTypes`.

7.1.5 Dashboard

Toil provides a dashboard for viewing the RAM and CPU usage of each node, the number of issued jobs of each type, the number of failed jobs, and the size of the jobs queue. To launch this dashboard for a toil workflow, include the `--metrics` flag in the toil script command. The dashboard can then be viewed in your browser at localhost:3000 while connected to the leader node through `toil ssh-cluster`. On AWS, the dashboard keeps track of every node in the cluster to monitor CPU and RAM usage, but it can also be used while running a workflow on a single machine. The dashboard uses Grafana as the front end for displaying real-time plots, and Prometheus for tracking metrics exported by toil. In order to use the dashboard for a non-released toil version, you will have to build the containers locally with `make docker`, since the prometheus, grafana, and mtail containers used in the dashboard are tied to a specific toil version.

7.2 Running in Google Compute Engine (GCE)

Toil supports a provisioner with Google, and a *Google Job Store*. To get started, follow instructions for *Preparing your Google environment*.

7.2.1 Preparing your Google environment

Toil supports using the Google Cloud Platform. Setting this up is easy!

1. Make sure that the `google` extra (*Installing Toil with Extra Features*) is installed
2. Follow Google's Instructions to download credentials and set the `GOOGLE_APPLICATION_CREDENTIALS` environment variable
3. Create a new ssh key with the proper format. To create a new ssh key run the command

```
$ ssh-keygen -t rsa -f ~/.ssh/id_rsa -C [USERNAME]
```

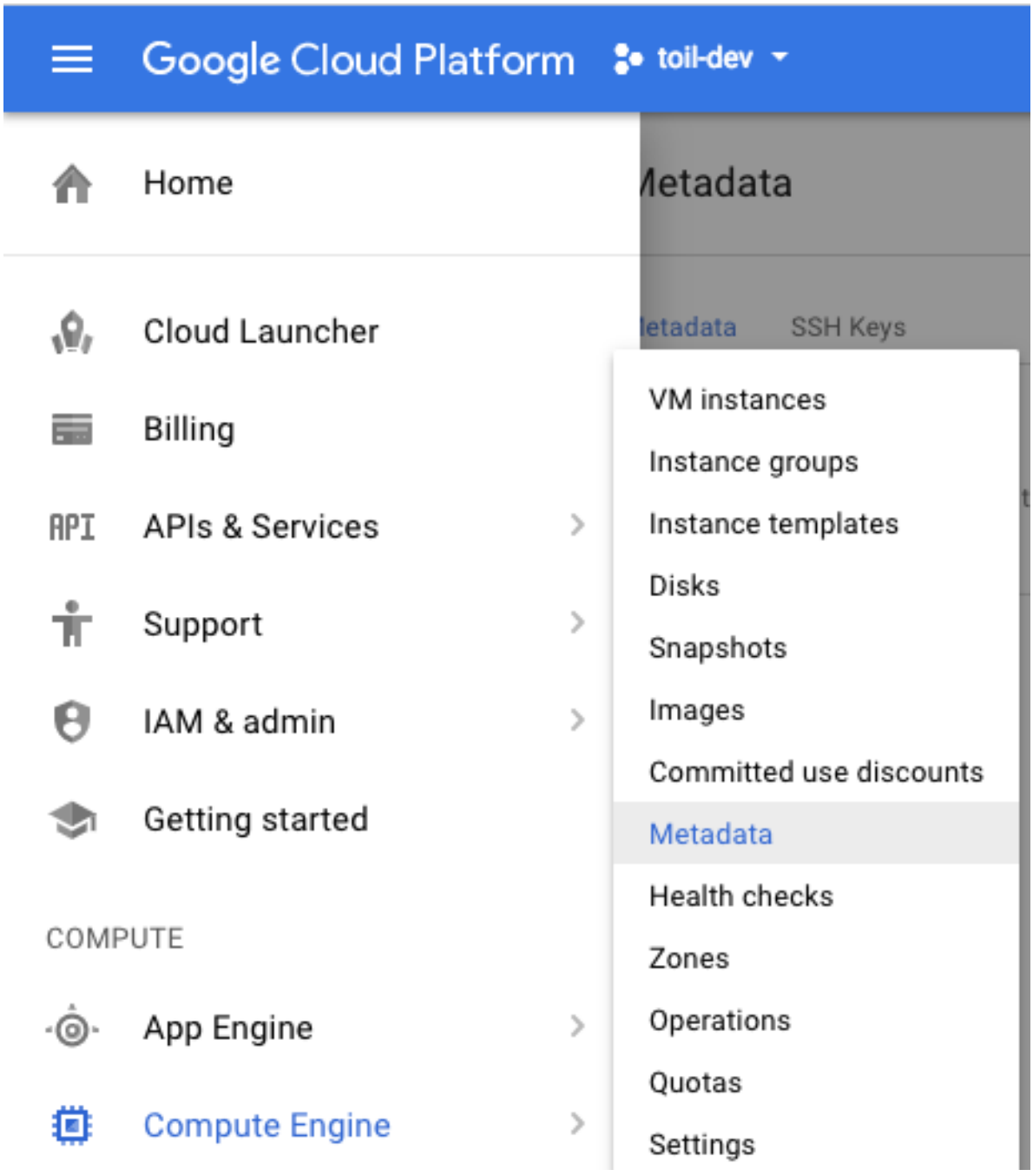
   where `[USERNAME]` is something like `jane@example.com`. Make sure to leave your password blank.

   Warning: This command could overwrite an old ssh key you may be using. If you have an existing ssh key you would like to use, it will need to be called id_rsa and it needs to have no password set.

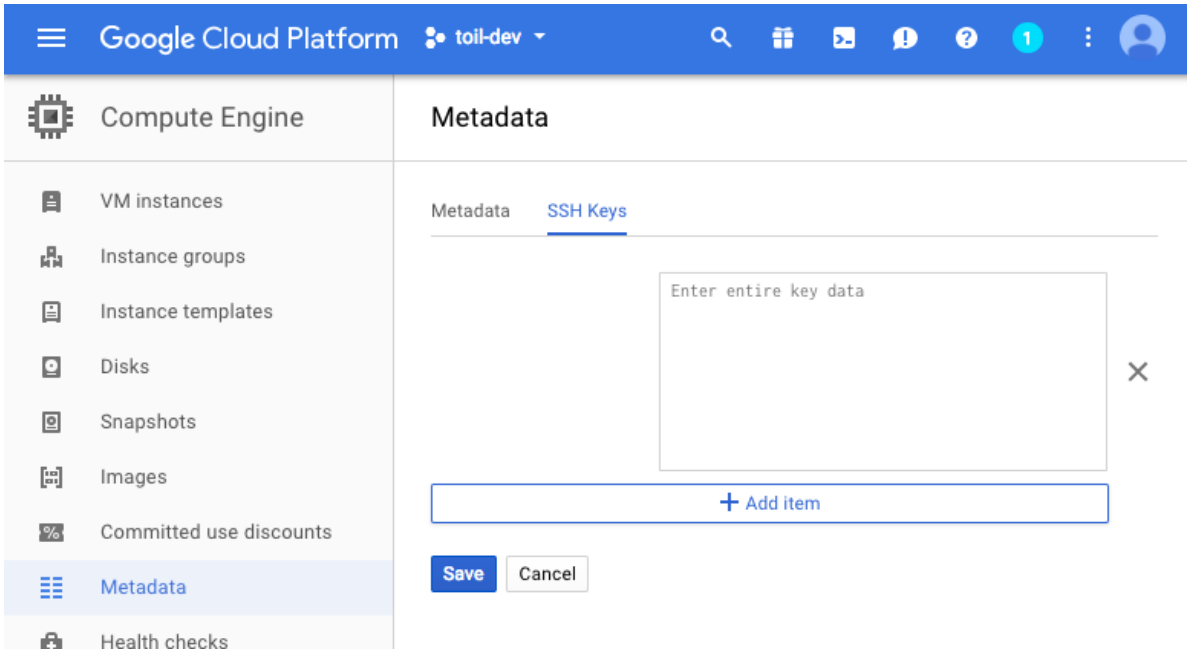
   Make sure only you can read the SSH keys:

```
$ chmod 400 ~/.ssh/id_rsa ~/.ssh/id_rsa.pub
```

4. Add your newly formatted public key to Google. To do this, log into your Google Cloud account and go to metadata section under the Compute tab.



Near the top of the screen click on 'SSH Keys', then edit, add item, and paste the key. Then save:



For more details look at Google's instructions for adding SSH keys.

7.2.2 Google Job Store

To use the Google Job Store you will need to set the `GOOGLE_APPLICATION_CREDENTIALS` environment variable by following Google's instructions.

Then to run the sort example with the Google job store you would type

```
$ python sort.py google:my-project-id:my-google-sort-jobstore
```

7.2.3 Running a Workflow with Autoscaling

Warning: Google Autoscaling is in beta!

The steps to run a GCE workflow are similar to those of AWS (*Running a Workflow with Autoscaling*), except you will need to explicitly specify the `--provisioner gce` option which otherwise defaults to `aws`.

1. Download `sort.py`
2. Launch the leader node in GCE using the *Launch-Cluster Command* command:

   ```
   (venv) $ toil launch-cluster <CLUSTER-NAME> --provisioner gce --leaderNodeType n1-
   ↪standard-1 --keyPairName <SSH-KEYNAME> --zone us-west1-a
   ```

   Where `<SSH-KEYNAME>` is the first part of `[USERNAME]` used when setting up your ssh key. For example if `[USERNAME]` was jane@example.com, `<SSH-KEYNAME>` should be `jane`.

   The `--keyPairName` option is for an SSH key that was added to the Google account. If your ssh key `[USERNAME]` was `jane@example.com`, then your key pair name will be just `jane`.

3. Upload the sort example and ssh into the leader:

```
(venv) $ toil rsync-cluster --provisioner gce <CLUSTER-NAME> sort.py :/root
(venv) $ toil ssh-cluster --provisioner gce <CLUSTER-NAME>
```

4. Run the workflow:

```
$ python /root/sort.py  google:<PROJECT-ID>:<JOBSTORE-NAME> --provisioner gce --
↪batchSystem mesos --nodeTypes n1-standard-2 --maxNodes 2
```

5. Clean up:

```
$ exit  # this exits the ssh from the leader node
(venv) $ toil destroy-cluster --provisioner gce <CLUSTER-NAME>
```

7.3 Cluster Utilities

There are several utilities used for starting and managing a Toil cluster using the AWS provisioner. They are installed via the `[aws]` or `[google]` extra. For installation details see *Toil Provisioner*. The cluster utilities are used for *Running in AWS* and are comprised of `toil launch-cluster`, `toil rsync-cluster`, `toil ssh-cluster`, and `toil destroy-cluster` entry points.

Cluster commands specific to `toil` are:

`status` — Reports runtime and resource usage for all jobs in a specified jobstore (workflow must have originally been run using the –stats option).

`stats` — Inspects a job store to see which jobs have failed, run successfully, etc.

`destroy-cluster` — For autoscaling. Terminates the specified cluster and associated resources.

`launch-cluster` — For autoscaling. This is used to launch a toil leader instance with the specified provisioner.

`rsync-cluster` — For autoscaling. Used to transfer files to a cluster launched with `toil launch-cluster`.

`ssh-cluster` — SSHs into the toil appliance container running on the leader of the cluster.

`clean` — Delete the job store used by a previous Toil workflow invocation.

`kill` — Kills any running jobs in a rogue toil.

For information on a specific utility run:

```
toil launch-cluster --help
```

for a full list of its options and functionality.

The cluster utilities can be used for *Running in Google Compute Engine (GCE)* and *Running in AWS*.

Tip: By default, all of the cluster utilities expect to be running on AWS. To run with Google you will need to specify the `--provisioner gce` option for each utility.

Note: Boto must be configured with AWS credentials before using cluster utilities.

Running in Google Compute Engine (GCE) contains instructions for

---

7.4 Stats Command

To use the stats command, a workflow must first be run using the `--stats` option. Using this command makes certain that toil does not delete the job store, no matter what other options are specified (i.e. normally the option `--clean=always` would delete the job, but `--stats` will override this).

An example of this would be running the following:

```
python discoverfiles.py file:my-jobstore --stats
```

Where `discoverfiles.py` is the following:

```
import subprocess
import os
from toil.common import Toil
from toil.job import Job

class discoverFiles(Job):
    """Views files at a specified path using ls."""
    def __init__(self, path, *args, **kwargs):
        self.path = path
        super(discoverFiles, self).__init__(*args, **kwargs)

    def run(self, fileStore):
        if os.path.exists(self.path):
            subprocess.check_call(["ls", self.path])

def main():
    options = Job.Runner.getDefaultArgumentParser().parse_args()
    options.clean = "always"

    job1 = discoverFiles(path="/sys/", displayName='sysFiles')
    job2 = discoverFiles(path=os.path.expanduser("~"), displayName='userFiles')
    job3 = discoverFiles(path="/tmp/")

    job1.addChild(job2)
    job2.addChild(job3)

    with Toil(options) as toil:
        if not toil.options.restart:
            toil.start(job1)
        else:
            toil.restart()

if __name__ == '__main__':
    main()
```

Notice the `displayName` key, which can rename a job, giving it an alias when it is finally displayed in stats. Running this workflow file should record three job names: `sysFiles` (job1), `userFiles` (job2), and `discoverFiles` (job3). To see the runtime and resources used for each job when it was run, type

```
toil stats file:my-jobstore
```

This should output the following:

```
Batch System: singleMachine
Default Cores: 1  Default Memory: 2097152K
```

(continues on next page)

(continued from previous page)

```
Max Cores: 9.22337e+18
Total Clock: 0.56  Total Runtime: 1.01
Worker
    Count |                                        Time* |                                                
```

```
toil clean file:my-jobstore
```

7.5 Status Command

Continuing the example from the stats section above, if we ran our workflow with the command

```
python discoverfiles.py file:my-jobstore --stats
```

We could interrogate our jobstore with the status command, for example:

```
toil status file:my-jobstore
```

If the run was successful, this would not return much valuable information, something like

```
2018-01-11 19:31:29,739 - toil.lib.bioio - INFO - Root logger is at level 'INFO',
↪'toil' logger at level 'INFO'.
2018-01-11 19:31:29,740 - toil.utils.toilStatus - INFO - Parsed arguments
2018-01-11 19:31:29,740 - toil.utils.toilStatus - INFO - Checking if we have files␣
↪for Toil
The root job of the job store is absent, the workflow completed successfully.
```

Otherwise, the `status` command should return the following:

> There are `x` unfinished jobs, `y` parent jobs with children, `z` jobs with services, `a` services, and `b` totally failed jobs currently in `c`.

7.6 Clean Command

If a Toil pipeline didn't finish successfully, or was run using `--clean=always` or `--stats`, the job store will exist until it is deleted. `toil clean <jobStore>` ensures that all artifacts associated with a job store are removed. This is particularly useful for deleting AWS job stores, which reserves an SDB domain as well as an S3 bucket.

The deletion of the job store can be modified by the `--clean` argument, and may be set to `always`, `onError`, `never`, or `onSuccess` (default).

Temporary directories where jobs are running can also be saved from deletion using the `--cleanWorkDir`, which has the same options as `--clean`. This option should only be run when debugging, as intermediate jobs will fill up disk space.

7.7 Launch-Cluster Command

Running `toil launch-cluster` starts up a leader for a cluster. Workers can be added to the initial cluster by specifying the `-w` option. An example would be

```
$ toil launch-cluster my-cluster --leaderNodeType t2.small -z us-west-2a --
↪keyPairName your-AWS-key-pair-name --nodeTypes m3.large,t2.micro -w 1,4
```

Options are listed below. These can also be displayed by running

```
$ toil launch-cluster --help
```

launch-cluster's main positional argument is the clusterName. This is simply the name of your cluster. If it does not exist yet, Toil will create it for you.

Launch-Cluster Options

--help -h also accepted. Displays this help menu.

--tempDirRoot TEMPDIRROOT Path to the temporary directory where all temp files are created, by default uses the current working directory as the base.

--version Display version.

--provisioner CLOUDPROVIDER -p CLOUDPROVIDER also accepted. The provisioner for cluster auto-scaling. Both AWS and GCE are currently supported.

--zone ZONE -z ZONE also accepted. The availability zone of the leader. This parameter can also be set via the TOIL_AWS_ZONE or TOIL_GCE_ZONE environment variables, or by the ec2_region_name parameter in your .boto file if using AWS, or derived from the instance metadata if using this utility on an existing EC2 instance.

--leaderNodeType LEADERNODETYPE Non-preemptable node type to use for the cluster leader.

--keyPairName KEYPAIRNAME The name of the AWS or ssh key pair to include on the instance.

--boto BOTOPATH The path to the boto credentials directory. This is transferred to all nodes in order to access the AWS jobStore from non-AWS instances.

--tag KEYVALUE KEYVALUE is specified as KEY=VALUE. -t KEY=VALUE also accepted. Tags are added to the AWS cluster for this node and all of its children. Tags are of the form: -t key1=value1 –tag key2=value2. Multiple tags are allowed and each tag needs its own flag. By default the cluster is tagged with: { "Name": clusterName, "Owner": IAM username }.

--vpcSubnet VPCSUBNET VPC subnet ID to launch cluster in. Uses default subnet if not specified. This subnet needs to have auto assign IPs turned on.

--nodeTypes NODETYPES Comma-separated list of node types to create while launching the leader. The syntax for each node type depends on the provisioner used. For the AWS provisioner this is the name of an EC2 instance type followed by a colon and the price in dollars to bid for a spot instance, for example 'c3.8xlarge:0.42'. Must also provide the –workers argument to specify how many workers of each node type to create.

--workers WORKERS -w WORKERS also accepted. Comma-separated list of the number of workers of each node type to launch alongside the leader when the cluster is created. This can be useful if running toil without auto-scaling but with need of more hardware support.

--leaderStorage LEADERSTORAGE Specify the size (in gigabytes) of the root volume for the leader instance. This is an EBS volume.

--nodeStorage NODESTORAGE Specify the size (in gigabytes) of the root volume for any worker instances created when using the -w flag. This is an EBS volume.

Logging Options

--logOff	Same as –logCritical.
--logCritical	Turn on logging at level CRITICAL and above. (default is INFO)
--logError	Turn on logging at level ERROR and above. (default is INFO)
--logWarning	Turn on logging at level WARNING and above. (default is INFO)
--logInfo	Turn on logging at level INFO and above. (default is INFO)
--logDebug	Turn on logging at level DEBUG and above. (default is INFO)
--logLevel LOGLEVEL	Log at given level (may be either OFF (or CRITICAL), ERROR, WARN (or WARNING), INFO or DEBUG). (default is INFO)
--logFile LOGFILE	File to log in.
--rotatingLogging	Turn on rotating logging, which prevents log files getting too big.

7.8 Ssh-Cluster Command

Toil provides the ability to ssh into the leader of the cluster. This can be done as follows:

```
$ toil ssh-cluster CLUSTER-NAME-HERE
```

This will open a shell on the Toil leader and is used to start an *Running a Workflow with Autoscaling* run. Issues with docker prevent using `screen` and `tmux` when sshing the cluster (The shell doesn't know that it is a TTY which prevents it from allocating a new screen session). This can be worked around via

```
$ script
$ screen
```

Simply running `screen` within `script` will get things working properly again.

Finally, you can execute remote commands with the following syntax:

```
$ toil ssh-cluster CLUSTER-NAME-HERE remoteCommand
```

It is not advised that you run your Toil workflow using remote execution like this unless a tool like nohup is used to ensure the process does not die if the SSH connection is interrupted.

For an example usage, see *Running a Workflow with Autoscaling*.

7.9 Rsync-Cluster Command

The most frequent use case for the `rsync-cluster` utility is deploying your Toil script to the Toil leader. Note that the syntax is the same as traditional rsync with the exception of the hostname before the colon. This is not needed in `toil rsync-cluster` since the hostname is automatically determined by Toil.

Here is an example of its usage:

```
$ toil rsync-cluster CLUSTER-NAME-HERE \
   ~/localFile :/remoteDestination
```

7.10 Destroy-Cluster Command

The `destroy-cluster` command is the advised way to get rid of any Toil cluster launched using the *Launch-Cluster Command* command. It ensures that all attached nodes, volumes, security groups, etc. are deleted. If a node or cluster is shut down using Amazon's online portal residual resources may still be in use in the background. To delete a cluster run

```
$ toil destroy-cluster CLUSTER-NAME-HERE
```

7.11 Kill Command

To kill all currently running jobs for a given jobstore, use the command

```
toil kill file:my-jobstore
```

CHAPTER 8

HPC Environments

Toil is a flexible framework that can be leveraged in a variety of environments, including high-performance computing (HPC) environments. Toil provides support for a number of batch systems, including Grid Engine, Slurm, Torque and LSF, which are popular schedulers used in these environments. Toil also supports HTCondor, which is a popular scheduler for high-throughput computing (HTC). To use one of these batch systems specify the "–batchSystem" argument to the toil script.

Due to the cost and complexity of maintaining support for these schedulers we currently consider them to be "community supported", that is the core development team does not regularly test or develop support for these systems. However, there are members of the Toil community currently deploying Toil in HPC environments and we welcome external contributions.

Developing the support of a new or existing batch system involves extending the abstract batch system class *toil.batchSystems.abstractBatchSystem.AbstractBatchSystem*.

8.1 Standard Output/Error from Batch System Jobs

Standard output and error from batch system jobs (except for the Parasol and Mesos batch systems) are redirected to files in the `toil-<workflowID>` directory created within the temporary directory specified by the `--workDir` option; see *Commandline Options*. Each file is named as follows: `toil_job_<Toil job ID>_batch_<name of batch system>_<job ID from batch system>_<file description>.log`, where `<file description>` is `std_output` for standard output, and `std_error` for standard error. HTCondor will also write job event log files with `<file description> = job_events`.

If capturing standard output and error is desired, `--workDir` will generally need to be on a shared file system; otherwise if these are written to local temporary directories on each node (e.g. `/tmp`) Toil will not be able to retrieve them. Alternatively, the `--noStdOutErr` option forces Toil to discard all standard output and error from batch system jobs.

CHAPTER 9

CWL in Toil

The Common Workflow Language (CWL) is an emerging standard for writing workflows that are portable across multiple workflow engines and platforms. Toil has full support for the CWL v1.0.1 specification.

9.1 Running CWL Locally

The *toil-cwl-runner* command provides cwl-parsing functionality using cwltool, and leverages the job-scheduling and batch system support of Toil.

To run in local batch mode, provide the CWL file and the input object file:

```
$ toil-cwl-runner example.cwl example-job.yml
```

For a simple example of CWL with Toil see *Running a basic CWL workflow*.

9.1.1 Note for macOS + Docker + Toil

When invoking CWL documents that make use of Docker containers if you see errors that look like

```
docker: Error response from daemon: Mounts denied:
The paths /var/...tmp are not shared from OS X and are not known to Docker.
```

you may need to add

```
export TMPDIR=/tmp/docker_tmp
```

either in your startup file (`.bashrc`) or add it manually in your shell before invoking toil.

9.2 Detailed Usage Instructions

Help information can be found by using this toil command:

```
$ toil-cwl-runner -h
```

A more detailed example shows how we can specify both Toil and cwltool arguments for our workflow:

```
$ toil-cwl-runner \
    --singularity \
    --jobStore my_jobStore \
    --batchSystem lsf \
    --workDir `pwd` \
    --outdir `pwd` \
    --logFile cwltoil.log \
    --writeLogs `pwd` \
    --logLevel DEBUG \
    --retryCount 2 \
    --disableCaching \
    --maxLogFileSize 20000000000 \
    --stats \
    standard_bam_processing.cwl \
    inputs.yaml
```

In this example, we set the following options, which are all passed to Toil:

`--singularity`: Specifies that all jobs with Docker fornat containers specified should be run using the Singularity container engine instead of the Docker container engine.

`--jobStore`: Path to a folder that already exists, which will contain the Toil jobstore and all related job-tracking information.

`--batchSystem`: Use the specified HPC or Cloud-based cluster platform.

`--workDir`: The directory where all temporary files will be created for the workflow. A subdirectory of this will be set as the `$TMPDIR` environment variable and this subdirectory can be referenced using the CWL parameter reference `$(runtime.tmpdir)` in CWL tools and workflows.

`--outdir`: Directory where final `File` and `Directory` outputs will be written. References to these and other output types will be in the JSON object printed to the stdout stream after workflow execution.

`--logFile`: Path to the main logfile with logs from all jobs.

`--writeLogs`: Directory where all job logs will be stored.

`--retryCount`: How many times to retry each Toil job.

`--disableCaching`: Currently required for batch systems (LSF, slurm, gridengine, htcondor, torque)

`--maxLogFileSize`: Logs that get larger than this value will be truncated.

`--stats`: Save resources usages in json files that can be collected with the `toil stats` command after the workflow is done.

9.3 Running CWL in the Cloud

To run in cloud and HPC configurations, you may need to provide additional command line parameters to select and configure the batch system to use.

To run a CWL workflow in AWS with toil see *Running a CWL Workflow on AWS*.

9.4 Running CWL within Toil Scripts

A CWL workflow can be run indirectly in a native Toil script. However, this is not the *standard* way to run CWL workflows with Toil and doing so comes at the cost of job efficiency. For some use cases, such as running one process on multiple files, it may be useful. For example, if you want to run a CWL workflow with 3 YML files specifying different samples inputs, it could look something like:

```
from toil.job import Job
from toil.common import Toil
import subprocess
import os


def initialize_jobs(job):
    job.fileStore.logToMaster('initialize_jobs')


def runQC(job, cwl_file, cwl_filename, yml_file, yml_filename, outputs_dir, output_
↪num):
    job.fileStore.logToMaster("runQC")
    tempDir = job.fileStore.getLocalTempDir()

    cwl = job.fileStore.readGlobalFile(cwl_file, userPath=os.path.join(tempDir, cwl_
↪filename))
    yml = job.fileStore.readGlobalFile(yml_file, userPath=os.path.join(tempDir, yml_
↪filename))

    subprocess.check_call(["cwltoil", cwl, yml])

    output_filename = "output.txt"
    output_file = job.fileStore.writeGlobalFile(output_filename)
    job.fileStore.readGlobalFile(output_file, userPath=os.path.join(outputs_dir,
↪"sample_" + output_num + "_" + output_filename))
    return output_file


if __name__ == "__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"
    with Toil(options) as toil:

        # specify the folder where the cwl and yml files live
        inputs_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)),
↪"cwlExampleFiles")
        # specify where you wish the outputs to be written
        outputs_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)),
↪"cwlExampleFiles")

        job0 = Job.wrapJobFn(initialize_jobs)

        cwl_filename = "hello.cwl"
        cwl_file = toil.importFile("file://" + os.path.abspath(os.path.join(inputs_
↪dir, cwl_filename)))

        # add list of yml config inputs here or import and construct from file
        yml_files = ["hello1.yml", "hello2.yml", "hello3.yml"]
```

(continues on next page)

(continued from previous page)

```
        i = 0
        for yml in yml_files:
            i = i + 1
            yml_file = toil.importFile("file://" + os.path.abspath(os.path.
↪join(inputs_dir, yml)))
            yml_filename = yml
            job = Job.wrapJobFn(runQC, cwl_file, cwl_filename, yml_file, yml_filename,
↪ outputs_dir, output_num=str(i))
            job0.addChild(job)

        toil.start(job0)
```

9.5 Toil & CWL Tips

See logs for just one job by using the full log file

This requires knowing the job's toil-generated ID, which can be found in the log files.

```
cat cwltoil.log | grep jobVM1fIs
```

Grep for full tool commands from toil logs

This gives you a more concise view of the commands being run (note that this information is only available from Toil when running with *–logDebug*).

```
pcregrep -M "\[job .*\.cwl.*$\n(.*          .*$\n)*" cwltoil.log
#          ^allows for multiline matching
```

Find Bams that have been generated for specific step while pipeline is running:

```
find . | grep -P '^./out_tmpdir.*_MD\.bam$'
```

See what jobs have been run

```
cat log/cwltoil.log | grep -oP "\[job .*.cwl\]" | sort | uniq
```

or:

```
cat log/cwltoil.log | grep -i "issued job"
```

Get status of a workflow

```
$ toil status /home/johnsoni/TEST_RUNS_3/TEST_run/tmp/jobstore-09ae0acc-c800-11e8-
↪9d09-70106fb1697e
<hostname> 2018-10-04 15:01:44,184 MainThread INFO toil.lib.bioio: Root logger is at␣
↪level 'INFO', 'toil' logger at level 'INFO'.
<hostname> 2018-10-04 15:01:44,185 MainThread INFO toil.utils.toilStatus: Parsed␣
↪arguments
<hostname> 2018-10-04 15:01:47,081 MainThread INFO toil.utils.toilStatus: Traversing␣
↪the job graph gathering jobs. This may take a couple of minutes.

Of the 286 jobs considered, there are 179 jobs with children, 107 jobs ready to run,␣
↪0 zombie jobs, 0 jobs with services, 0 services, and 0 jobs with log files␣
↪currently in file:/home/user/jobstore-09ae0acc-c800-11e8-9d09-70106fb1697e.
```

Toil Stats

You can get run statistics broken down by CWL file. This only works once the workflow is finished:

```
$ toil stats /path/to/jobstore
```

The output will contain CPU, memory, and walltime information for all CWL job types:

```
<hostname> 2018-10-15 12:06:19,003 MainThread INFO toil.lib.bioio: Root logger is at level 'INFO', 'toil' logger at level 'INFO'.
<hostname> 2018-10-15 12:06:19,004 MainThread INFO toil.utils.toilStats: Parsed arguments
<hostname> 2018-10-15 12:06:19,004 MainThread INFO toil.utils.toilStats: Checking if we have files for toil
<hostname> 2018-10-15 12:06:19,004 MainThread INFO toil.utils.toilStats: Checked arguments
Batch System: lsf
Default Cores: 1  Default Memory: 10485760K
Max Cores: 9.22337e+18
Total Clock: 106608.01  Total Runtime: 86634.11
Worker
    Count |                                             Time* |                                                 Wait |                                                Memory
        n |      min    med*     ave      max     total |      min     med      ave      max      total |      min      med      ave      max    total
     1659 |     0.00    0.80  264.87 12595.59 439424.40 |     0.00    0.46   449.05 42240.74 744968.80 |  -35336.69     0.16   -184.17  4230.65 -305544.39 |      48K    223K   1020K  40235K 1692300K
Job
 Worker Jobs  |     min    med    ave    max
              |    1077   1077   1077   1077
    Count |                                             Time* |                                                 Wait |                                                Memory
        n |      min    med*     ave      max     total |      min     med      ave      max      total |      min      med      ave      max    total
     1077 |     0.04    1.18  407.06 12593.43 438404.73 |     0.01    0.28   691.17 42240.35 744394.14 |  -35336.83     0.27   -284.11  4230.49 -305989.41 |     135K    268K   1633K  40235K 1759734K
 ResolveIndirect
    Count |                                             Time* |                                                 Wait |                                                Memory
        n |      min    med*     ave      max     total |      min     med      ave      max      total |      min      med      ave      max    total
      205 |     0.04    0.07    0.16     2.29     31.95 |     0.01    0.02     0.02     0.14     3.60 |     0.02     0.05     0.14     2.28     28.35 |     190K    266K    256K    314K   52487K
 CWLGather
    Count |                                             Time* |                                                 Wait |                                                Memory
        n |      min    med*     ave      max     total |      min     med      ave      max      total |      min      med      ave      max    total
```

(continues on next page)

(continued from previous page)

```
      40 |     0.05    0.17    0.29     1.90     11.62 |     0.01    0.02     0.02
↪  0.05      0.80 |       0.03     0.14      0.27     1.88      10.82 |     188K
↪265K    250K    316K   10039K
CWLWorkflow
   Count |                                          Time* |
↪           Clock |                                                Wait |
↪                       Memory
       n |      min    med*     ave      max     total |      min     med      ave
↪   max     total |        min      med       ave      max      total |      min
↪med     ave     max    total
     205 |     0.09    0.40    0.98    13.70    200.82 |     0.04    0.15     0.16
↪  1.08     31.78 |       0.04     0.26      0.82    12.62     169.04 |     190K
↪270K    257K    316K   52826K
file:///home/johnsoni/pipeline_0.0.39/ACCESS-Pipeline/cwl_tools/expression_tools/
↪group_waltz_files.cwl
   Count |                                          Time* |
↪           Clock |                                                Wait |
↪                       Memory
       n |      min    med*     ave      max     total |      min     med      ave
↪   max     total |        min      med       ave      max      total |      min
↪med     ave     max    total
      99 |     0.29    0.49    0.59     2.50     58.11 |     0.14    0.26     0.29
↪  1.04     28.95 |       0.14     0.22      0.29     1.48      29.16 |     135K
↪135K    135K    136K   13459K
file:///home/johnsoni/pipeline_0.0.39/ACCESS-Pipeline/cwl_tools/expression_tools/
↪make_sample_output_dirs.cwl
   Count |                                          Time* |
↪           Clock |                                                Wait |
↪                       Memory
       n |      min    med*     ave      max     total |      min     med      ave
↪   max     total |        min      med       ave      max      total |      min
↪med     ave     max    total
      11 |     0.34    0.52    0.74     2.63      8.18 |     0.20    0.30     0.41
↪  1.17      4.54 |       0.14     0.20      0.33     1.45       3.65 |     136K
↪136K    136K    136K    1496K
file:///home/johnsoni/pipeline_0.0.39/ACCESS-Pipeline/cwl_tools/expression_tools/
↪consolidate_files.cwl
   Count |                                          Time* |
↪           Clock |                                                Wait |
↪                       Memory
       n |      min    med*     ave      max     total |      min     med      ave
↪   max     total |        min      med       ave      max      total |      min
↪med     ave     max    total
       8 |     0.31    0.59    0.71     1.80      5.69 |     0.18    0.35     0.37
↪  0.63      2.94 |       0.13     0.27      0.34     1.17       2.75 |     136K
↪136K    136K    136K    1091K
file:///home/johnsoni/pipeline_0.0.39/ACCESS-Pipeline/cwl_tools/bwa-mem/bwa-mem.cwl
   Count |                                          Time* |
↪           Clock |                                                Wait |
↪                       Memory
       n |      min    med*     ave      max     total |      min     med      ave
↪   max     total |        min      med       ave      max      total |      min
↪med     ave     max    total
      22 |   895.76 3098.13 3587.34 12593.43  78921.51 |  2127.02 7910.31  8123.06
↪16959.13 178707.34 |  -11049.84 -3827.96  -4535.72    19.49  -99785.83 |    5659K
↪5950K   5854K   6128K  128807K
```

Understanding toil log files

There is a *worker_log.txt* file for each job, this file is written to while the job is running, and deleted after the job finishes. The contents are printed to the main log file and transferred to a log file in the *–logDir* folder once the job is completed successfully.

The new log file will be named something like:

```
file:<path to cwl tool>.cwl_<job ID>.log

file:---home-johnsoni-pipeline_1.1.14-ACCESS--Pipeline-cwl_tools-marianas-
↪ProcessLoopUMIFastq.cwl_I-O-jobfGsQQw000.log
```

This is the toil job command with spaces replaced by dashes.

CHAPTER 10

WDL in Toil

Support is still in the alpha phase and should be able to handle basic wdl files. See the specification below for more details.

10.1 How to Run a WDL file in Toil

Recommended best practice when running wdl files is to first use the Broad's wdltool for syntax validation and generating the needed json input file. Full documentation can be found on the repository, and a precompiled jar binary can be downloaded here: wdltool (this requires java7).

That means two steps. First, make sure your wdl file is valid and devoid of syntax errors by running

```
java -jar wdltool.jar validate example_wdlfile.wdl
```

Second, generate a complementary json file if your wdl file needs one. This json will contain keys for every necessary input that your wdl file needs to run:

```
java -jar wdltool.jar inputs example_wdlfile.wdl
```

When this json template is generated, open the file, and fill in values as necessary by hand. WDL files all require json files to accompany them. If no variable inputs are needed, a json file containing only '{}' may be required.

Once a wdl file is validated and has an appropriate json file, workflows can be run in toil using:

```
toil-wdl-runner example_wdlfile.wdl example_jsonfile.json
```

See options below for more parameters.

10.2 ENCODE Example from ENCODE-DCC

To follow this example, you will need docker installed. The original workflow can be found here: https://github.com/ENCODE-DCC/pipeline-container

We've included the wdl file and data files in the toil repository needed to run this example. First, download the example code and unzip. The file needed is "testENCODE/encode_mapping_workflow.wdl".

Next, use wdltool (this requires java7) to validate this file:

```
java -jar wdltool.jar validate encode_mapping_workflow.wdl
```

Next, use wdltool to generate a json file for this wdl file:

```
java -jar wdltool.jar inputs encode_mapping_workflow.wdl
```

This json file once opened should look like this:

```
{
"encode_mapping_workflow.fastqs": "Array[File]",
"encode_mapping_workflow.trimming_parameter": "String",
"encode_mapping_workflow.reference": "File"
}
```

The trimming_parameter should be set to 'native'. Download the example code and unzip. Inside are two data files required for the run

```
ENCODE_data/reference/GRCh38_chr21_bwa.tar.gz    ENCODE_data/ENCFF000VOL_chr21.
fq.gz
```

Editing the json to include these as inputs, the json should now look something like this:

```
{
"encode_mapping_workflow.fastqs": ["/path/to/unzipped/ENCODE_data/ENCFF000VOL_chr21.
↪fq.gz"],
"encode_mapping_workflow.trimming_parameter": "native",
"encode_mapping_workflow.reference": "/path/to/unzipped/ENCODE_data/reference/GRCh38_
↪chr21_bwa.tar.gz"
}
```

The wdl and json files can now be run using the command

```
toil-wdl-runner encode_mapping_workflow.wdl encode_mapping_workflow.json
```

This should deposit the output files in the user's current working directory (to change this, specify a new directory with the '-o' option).

10.3 GATK Examples from the Broad

Simple examples of WDL can be found on the Broad's website as tutorials: https://software.broadinstitute.org/wdl/documentation/topic?name=wdl-tutorials.

One can follow along with these tutorials, write their own wdl files following the directions and run them using either cromwell or toil. For example, in tutorial 1, if you've followed along and named your wdl file 'helloHaplotypeCall.wdl', then once you've validated your wdl file using wdltool (this requires java7) using

```
java -jar wdltool.jar validate helloHaplotypeCaller.wdl
```

and generated a json file (and subsequently typed in appropriate filepaths* and variables) using

```
java -jar wdltool.jar inputs helloHaplotypeCaller.wdl
```

- Absolute filepath inputs are recommended for local testing.

then the wdl script can be run using

```
toil-wdl-runner helloHaplotypeCaller.wdl helloHaplotypeCaller_inputs.json
```

10.4 toilwdl.py Options

'-o' or **'–output_directory'**: Specifies the output folder, and defaults to the current working directory if not specified by the user.

'–gen_parse_files': Creates "AST.out", which holds a printed AST of the wdl file and "mappings.out", which holds the printed task, workflow, csv, and tsv dictionaries generated by the parser.

'–dont_delete_compiled': Saves the compiled toil python workflow file for debugging.

Any number of arbitrary options may also be specified. These options will not be parsed immediately, but passed down as toil options once the wdl/json files are processed. For valid toil options, see the documentation: http://toil.readthedocs.io/en/latest/running/cliOptions.html

10.5 Running WDL within Toil Scripts

Note: A cromwell.jar file is needed in order to run a WDL workflow.

A WDL workflow can be run indirectly in a native Toil script. However, this is not the *standard* way to run WDL workflows with Toil and doing so comes at the cost of job efficiency. For some use cases, such as running one process on multiple files, it may be useful. For example, if you want to run a WDL workflow with 3 JSON files specifying different samples inputs, it could look something like:

```
from toil.job import Job
from toil.common import Toil
import subprocess
import os


def initialize_jobs(job):
    job.fileStore.logToMaster("initialize_jobs")


def runQC(job, wdl_file, wdl_filename, json_file, json_filename, outputs_dir, jar_loc,
↪output_num):
    job.fileStore.logToMaster("runQC")
    tempDir = job.fileStore.getLocalTempDir()

    wdl = job.fileStore.readGlobalFile(wdl_file, userPath=os.path.join(tempDir, wdl_
↪filename))
    json = job.fileStore.readGlobalFile(json_file, userPath=os.path.join(tempDir,␣
↪json_filename))

    subprocess.check_call(["java","-jar",jar_loc,"run",wdl,"--inputs",json])

    output_filename = "output.txt"
    output_file = job.fileStore.writeGlobalFile(outputs_dir + output_filename)
    job.fileStore.readGlobalFile(output_file, userPath=os.path.join(outputs_dir,
↪"sample_" + output_num + "_" + output_filename))
    return output_file


if __name__ == "__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
```

(continues on next page)

(continued from previous page)

```
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:

        # specify the folder where the wdl and json files live
        inputs_dir = "wdlExampleFiles/"
        # specify where you wish the outputs to be written
        outputs_dir = "wdlExampleFiles/"
        # specify the location of your cromwell jar
        jar_loc = os.path.abspath("wdlExampleFiles/cromwell-35.jar")

        job0 = Job.wrapJobFn(initialize_jobs)

        wdl_filename = "hello.wdl"
        wdl_file = toil.importFile("file://" + os.path.abspath(os.path.join(inputs_
↪dir, wdl_filename)))


        # add list of yml config inputs here or import and construct from file
        json_files = ["hello1.json", "hello2.json", "hello3.json"]
        i = 0
        for json in json_files:
            i = i + 1
            json_file = toil.importFile("file://" + os.path.join(inputs_dir, json))
            json_filename = json
            job = Job.wrapJobFn(runQC, wdl_file, wdl_filename, json_file, json_
↪filename, outputs_dir, jar_loc, output_num=str(i))
            job0.addChild(job)

        toil.start(job0)
```

10.6 WDL Specifications

WDL language specifications can be found here: https://github.com/broadinstitute/wdl/blob/develop/SPEC.md

Implementing support for more features is currently underway, but a basic roadmap so far is:

CURRENTLY IMPLEMENTED:

- Scatter
- Many Built-In Functions
- Docker Calls
- Handles Priority, and Output File Wrangling
- Currently Handles Primitives and Arrays

TO BE IMPLEMENTED:

- Integrate Cloud Autoscaling Capacity More Robustly
- WDL Files That "Import" Other WDL Files (Including URI Handling for 'http://' and 'https://')

CHAPTER 11

Developing a Workflow

This tutorial walks through the features of Toil necessary for developing a workflow using the Toil Python API.

Note: "script" and "workflow" will be used interchangeably

11.1 Scripting Quick Start

To begin, consider this short toil script which illustrates defining a workflow:

```
from toil.common import Toil
from toil.job import Job

def helloWorld(message, memory="2G", cores=2, disk="3G"):
    return "Hello, world!, here's a message: %s" % message

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "OFF"
    options.clean = "always"

    hello_job = Job.wrapFn(helloWorld, "Woot")

    with Toil(options) as toil:
        print(toil.start(hello_job)) #Prints Hello, world!, ...
```

The workflow consists of a single job. The resource requirements for that job are (optionally) specified by keyword arguments (memory, cores, disk). The script is run using *toil.job.Job.Runner.getDefaultOptions()*. Below we explain the components of this code in detail.

11.2 Job Basics

The atomic unit of work in a Toil workflow is a *Job*. User scripts inherit from this base class to define units of work. For example, here is a more long-winded class-based version of the job in the quick start example:

```
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return "Hello, world!, here's a message: %s" % self.message
```

In the example a class, HelloWorld, is defined. The constructor requests 2 gigabytes of memory, 2 cores and 3 gigabytes of local disk to complete the work.

The *toil.job.Job.run()* method is the function the user overrides to get work done. Here it just logs a message using *toil.job.Job.log()*, which will be registered in the log output of the leader process of the workflow.

11.3 Invoking a Workflow

We can add to the previous example to turn it into a complete workflow by adding the necessary function calls to create an instance of HelloWorld and to run this as a workflow containing a single job. This uses the *toil.job.Job.Runner* class, which is used to start and resume Toil workflows. For example:

```
from toil.common import Toil
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return "Hello, world!, here's a message: %s" % self.message

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "OFF"
    options.clean = "always"

    hello_job = HelloWorld("Woot")

    with Toil(options) as toil:
        print(toil.start(hello_job))
```

Note: Do not include a . in the name of your python script (besides *.py* at the end). This is to allow toil to import the types and functions defined in your file while starting a new process.

Alternatively, the more powerful *toil.common.Toil* class can be used to run and resume workflows. It is used as a context manager and allows for preliminary setup, such as staging of files into the job store on the leader node. An

instance of the class is initialized by specifying an options object. The actual workflow is then invoked by calling the *toil.common.Toil.start()* method, passing the root job of the workflow, or, if a workflow is being restarted, *toil.common.Toil.restart()* should be used. Note that the context manager should have explicit if else branches addressing restart and non restart cases. The boolean value for these if else blocks is toil.options.restart.

For example:

```
from toil.job import Job
from toil.common import Toil

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        self.log("Hello, world!, I have a message: {}".format(self.message))

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        if not toil.options.restart:
            job = HelloWorld("Woot!")
            toil.start(job)
        else:
            toil.restart()
```

The call to *toil.job.Job.Runner.getDefaultOptions()* creates a set of default options for the workflow. The only argument is a description of how to store the workflow's state in what we call a *job-store*. Here the job-store is contained in a directory within the current working directory called "toilWorkflowRun". Alternatively this string can encode other ways to store the necessary state, e.g. an S3 bucket object store location. By default the job-store is deleted if the workflow completes successfully.

The workflow is executed in the final line, which creates an instance of HelloWorld and runs it as a workflow. Note all Toil workflows start from a single starting job, referred to as the *root* job. The return value of the root job is returned as the result of the completed workflow (see promises below to see how this is a useful feature!).

11.4 Specifying Commandline Arguments

To allow command line control of the options we can use the *toil.job.Job.Runner.getDefaultArgumentParser()* method to create a *argparse.ArgumentParser* object which can be used to parse command line options for a Toil script. For example:

```
from toil.common import Toil
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return "Hello, world!, here's a message: %s" % self.message
```

(continues on next page)

(continued from previous page)

```
if __name__=="__main__":
    parser = Job.Runner.getDefaultArgumentParser()
    options = parser.parse_args()
    options.logLevel = "OFF"
    options.clean = "always"

    hello_job = HelloWorld("Woot")

    with Toil(options) as toil:
        print(toil.start(hello_job))
```

Creates a fully fledged script with all the options Toil exposed as command line arguments. Running this script with "–help" will print the full list of options.

Alternatively an existing `argparse.ArgumentParser` or `optparse.OptionParser` object can have Toil script command line options added to it with the *`toil.job.Job.Runner.addToilOptions()`* method.

11.5 Resuming a Workflow

In the event that a workflow fails, either because of programmatic error within the jobs being run, or because of node failure, the workflow can be resumed. Workflows can only not be reliably resumed if the job-store itself becomes corrupt.

Critical to resumption is that jobs can be rerun, even if they have apparently completed successfully. Put succinctly, a user defined job should not corrupt its input arguments. That way, regardless of node, network or leader failure the job can be restarted and the workflow resumed.

To resume a workflow specify the "restart" option in the options object passed to *`toil.common.Toil.start()`*. If node failures are expected it can also be useful to use the integer "retryCount" option, which will attempt to rerun a job retryCount number of times before marking it fully failed.

In the common scenario that a small subset of jobs fail (including retry attempts) within a workflow Toil will continue to run other jobs until it can do no more, at which point *`toil.common.Toil.start()`* will raise a `toil.leader.FailedJobsException` exception. Typically at this point the user can decide to fix the script and resume the workflow or delete the job-store manually and rerun the complete workflow.

11.6 Functions and Job Functions

Defining jobs by creating class definitions generally involves the boilerplate of creating a constructor. To avoid this the classes *`toil.job.FunctionWrappingJob`* and `toil.job.JobFunctionWrappingTarget` allow functions to be directly converted to jobs. For example, the quick start example (repeated here):

```
from toil.common import Toil
from toil.job import Job

def helloWorld(message, memory="2G", cores=2, disk="3G"):
    return "Hello, world!, here's a message: %s" % message

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "OFF"
```

(continues on next page)

(continued from previous page)

```
    options.clean = "always"

    hello_job = Job.wrapFn(helloWorld, "Woot")

    with Toil(options) as toil:
        print(toil.start(hello_job)) #Prints Hello, world!, ...
```

Is equivalent to the previous example, but using a function to define the job.

The function call:

```
Job.wrapFn(helloWorld, "Woot")
```

Creates the instance of the `toil.job.FunctionWrappingTarget` that wraps the function.

The keyword arguments *memory*, *cores* and *disk* allow resource requirements to be specified as before. Even if they are not included as keyword arguments within a function header they can be passed as arguments when wrapping a function as a job and will be used to specify resource requirements.

We can also use the function wrapping syntax to a *job function*, a function whose first argument is a reference to the wrapping job. Just like a *self* argument in a class, this allows access to the methods of the wrapping job, see `toil.job.JobFunctionWrappingTarget`. For example:

```
from toil.common import Toil
from toil.job import Job

def helloWorld(job, message):
    job.log("Hello world, I have a message: {}".format(message))

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    hello_job = Job.wrapJobFn(helloWorld, "Woot!")

    with Toil(options) as toil:
        toil.start(hello_job)
```

Here `helloWorld()` is a job function. It uses the *`toil.job.Job.log()`* to log a message that will be printed to the output console. Here the only subtle difference to note is the line:

```
hello_job = Job.wrapJobFn(helloWorld, "Woot")
```

Which uses the function *`toil.job.Job.wrapJobFn()`* to wrap the job function instead of *`toil.job.Job.wrapFn()`* which wraps a vanilla function.

11.7 Workflows with Multiple Jobs

A *parent* job can have *child* jobs and *follow-on* jobs. These relationships are specified by methods of the job class, e.g. *`toil.job.Job.addChild()`* and *`toil.job.Job.addFollowOn()`*.

Considering a set of jobs the nodes in a job graph and the child and follow-on relationships the directed edges of the graph, we say that a job B that is on a directed path of child/follow-on edges from a job `A` in the job graph is a *successor* of `A`, similarly `A` is a *predecessor* of `B`.

A parent job's child jobs are run directly after the parent job has completed, and in parallel. The follow-on jobs of a job are run after its child jobs and their successors have completed. They are also run in parallel. Follow-ons allow the easy specification of cleanup tasks that happen after a set of parallel child tasks. The following shows a simple example that uses the earlier `helloWorld()` job function:

```
from toil.common import Toil
from toil.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.log("Hello world, I have a message: {}".format(message))

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    j1 = Job.wrapJobFn(helloWorld, "first")
    j2 = Job.wrapJobFn(helloWorld, "second or third")
    j3 = Job.wrapJobFn(helloWorld, "second or third")
    j4 = Job.wrapJobFn(helloWorld, "last")
    j1.addChild(j2)
    j1.addChild(j3)
    j1.addFollowOn(j4)

    with Toil(options) as toil:
        toil.start(j1)
```

In the example four jobs are created, first `j1` is run, then `j2` and `j3` are run in parallel as children of `j1`, finally `j4` is run as a follow-on of `j1`.

There are multiple short hand functions to achieve the same workflow, for example:

```
from toil.common import Toil
from toil.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.log("Hello world, I have a message: {}".format(message))

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    j1 = Job.wrapJobFn(helloWorld, "first")
    j2 = j1.addChildJobFn(helloWorld, "second or third")
    j3 = j1.addChildJobFn(helloWorld, "second or third")
    j4 = j1.addFollowOnJobFn(helloWorld, "last")

    with Toil(options) as toil:
        toil.start(j1)
```

Equivalently defines the workflow, where the functions *toil.job.Job.addChildJobFn()* and *toil.job.Job.addFollowOnJobFn()* are used to create job functions as children or follow-ons of an earlier job.

Jobs graphs are not limited to trees, and can express arbitrary directed acyclic graphs. For a precise definition of legal graphs see *toil.job.Job.checkJobGraphForDeadlocks()*. The previous example could be specified as a DAG as follows:

```
from toil.common import Toil
from toil.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.log("Hello world, I have a message: {}".format(message))

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    j1 = Job.wrapJobFn(helloWorld, "first")
    j2 = j1.addChildJobFn(helloWorld, "second or third")
    j3 = j1.addChildJobFn(helloWorld, "second or third")
    j4 = j2.addChildJobFn(helloWorld, "last")
    j3.addChild(j4)

    with Toil(options) as toil:
        toil.start(j1)
```

Note the use of an extra child edge to make `j4` a child of both `j2` and `j3`.

11.8 Dynamic Job Creation

The previous examples show a workflow being defined outside of a job. However, Toil also allows jobs to be created dynamically within jobs. For example:

```
from toil.common import Toil
from toil.job import Job

def binaryStringFn(job, depth, message=""):
    if depth > 0:
        job.addChildJobFn(binaryStringFn, depth-1, message + "0")
        job.addChildJobFn(binaryStringFn, depth-1, message + "1")
    else:
        job.log("Binary string: {}".format(message))

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        toil.start(Job.wrapJobFn(binaryStringFn, depth=5))
```

The job function `binaryStringFn` logs all possible binary strings of length `n` (here `n=5`), creating a total of `2^(n+2) - 1` jobs dynamically and recursively. Static and dynamic creation of jobs can be mixed in a Toil workflow, with jobs defined within a job or job function being created at run time.

11.9 Promises

The previous example of dynamic job creation shows variables from a parent job being passed to a child job. Such forward variable passing is naturally specified by recursive invocation of successor jobs within parent jobs. This can

also be achieved statically by passing around references to the return variables of jobs. In Toil this is achieved with promises, as illustrated in the following example:

```
from toil.common import Toil
from toil.job import Job

def fn(job, i):
    job.log("i is: %s" % i, level=100)
    return i+1

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    j1 = Job.wrapJobFn(fn, 1)
    j2 = j1.addChildJobFn(fn, j1.rv())
    j3 = j1.addFollowOnJobFn(fn, j2.rv())

    with Toil(options) as toil:
        toil.start(j1)
```

Running this workflow results in three log messages from the jobs: `i is 1` from `j1`, `i is 2` from `j2` and `i is 3` from `j3`.

The return value from the first job is *promised* to the second job by the call to *`toil.job.Job.rv()`* in the following line:

```
j2 = j1.addChildFn(fn, j1.rv())
```

The value of `j1.rv()` is a *promise*, rather than the actual return value of the function, because `j1` for the given input has at that point not been evaluated. A promise (*`toil.job.Promise`*) is essentially a pointer to for the return value that is replaced by the actual return value once it has been evaluated. Therefore, when `j2` is run the promise becomes 2.

Promises also support indexing of return values:

```
def parent(job):
    indexable = Job.wrapJobFn(fn)
    job.addChild(indexable)
    job.addFollowOnFn(raiseWrap, indexable.rv(2))

def raiseWrap(arg):
    raise RuntimeError(arg) # raises "2"

def fn(job):
    return (0, 1, 2, 3)
```

Promises can be quite useful. For example, we can combine dynamic job creation with promises to achieve a job creation process that mimics the functional patterns possible in many programming languages:

```
from toil.common import Toil
from toil.job import Job

def binaryStrings(job, depth, message=""):
    if depth > 0:
        s = [ job.addChildJobFn(binaryStrings, depth-1, message + "0").rv(),
              job.addChildJobFn(binaryStrings, depth-1, message + "1").rv() ]
```

(continues on next page)

(continued from previous page)

```
        return job.addFollowOnFn(merge, s).rv()
    return [message]

def merge(strings):
    return strings[0] + strings[1]

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.loglevel = "OFF"
    options.clean = "always"

    with Toil(options) as toil:
        print(toil.start(Job.wrapJobFn(binaryStrings, depth=5)))
```

The return value `l` of the workflow is a list of all binary strings of length 10, computed recursively. Although a toy example, it demonstrates how closely Toil workflows can mimic typical programming patterns.

11.10 Promised Requirements

Promised requirements are a special case of *Promises* that allow a job's return value to be used as another job's resource requirements.

This is useful when, for example, a job's storage requirement is determined by a file staged to the job store by an earlier job:

```
from toil.common import Toil
from toil.job import Job, PromisedRequirement
import os

def parentJob(job):
    downloadJob = Job.wrapJobFn(stageFn, "File://"+os.path.realpath(__file__),␣
↪cores=0.1, memory='32M', disk='1M')
    job.addChild(downloadJob)

    analysis = Job.wrapJobFn(analysisJob, fileStoreID=downloadJob.rv(0),
                             disk=PromisedRequirement(downloadJob.rv(1)))
    job.addFollowOn(analysis)

def stageFn(job, url, cores=1):
    importedFile = job.fileStore.importFile(url)
    return importedFile, importedFile.size

def analysisJob(job, fileStoreID, cores=2):
    # now do some analysis on the file
    pass

if __name__ == "__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        toil.start(Job.wrapJobFn(parentJob))
```

Note that this also makes use of the `size` attribute of the *FileID* object. This promised requirements mechanism can

also be used in combination with an aggregator for multiple jobs' output values:

```
def parentJob(job):
    aggregator = []
    for fileNum in range(0,10):
        downloadJob = Job.wrapJobFn(stageFn, "File://"+os.path.realpath(__file__),␣
↪cores=0.1, memory='32M', disk='1M')
        job.addChild(downloadJob)
        aggregator.append(downloadJob)

    analysis = Job.wrapJobFn(analysisJob, fileStoreID=downloadJob.rv(0),
                             disk=PromisedRequirement(lambda xs: sum(xs), [j.rv(1)␣
↪for j in aggregator]))
    job.addFollowOn(analysis)
```

Limitations

Just like regular promises, the return value must be determined prior to scheduling any job that depends on the return value. In our example above, notice how the dependent jobs were follow ons to the parent while promising jobs are children of the parent. This ordering ensures that all promises are properly fulfilled.

11.11 FileID

The `toil.fileStore.FileID` class is a small wrapper around Python's builtin string class. It is used to represent a file's ID in the file store, and has a `size` attribute that is the file's size in bytes. This object is returned by `importFile` and `writeGlobalFile`.

11.12 Managing files within a workflow

It is frequently the case that a workflow will want to create files, both persistent and temporary, during its run. The *toil.fileStores.abstractFileStore.AbstractFileStore* class is used by jobs to manage these files in a manner that guarantees cleanup and resumption on failure.

The *toil.job.Job.run()* method has a file store instance as an argument. The following example shows how this can be used to create temporary files that persist for the length of the job, be placed in a specified local disk of the node and that will be cleaned up, regardless of failure, when the job finishes:

```
from toil.common import Toil
from toil.job import Job

class LocalFileStoreJob(Job):
    def run(self, fileStore):
        # self.TempDir will always contain the name of a directory within the␣
↪allocated disk space reserved for the job
        scratchDir = self.tempDir

        # Similarly create a temporary file.
        scratchFile = fileStore.getLocalTempFile()

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
```

(continues on next page)

(continued from previous page)

```
    options.logLevel = "INFO"
    options.clean = "always"

    # Create an instance of FooJob which will have at least 2 gigabytes of storage␣
↪space.
    j = LocalFileStoreJob(disk="2G")

    #Run the workflow
    with Toil(options) as toil:
        toil.start(j)
```

Job functions can also access the file store for the job. The equivalent of the `LocalFileStoreJob` class is

```
def localFileStoreJobFn(job):
    scratchDir = job.tempDir
    scratchFile = job.fileStore.getLocalTempFile()
```

Note that the `fileStore` attribute is accessed as an attribute of the `job` argument.

In addition to temporary files that exist for the duration of a job, the file store allows the creation of files in a *global* store, which persists during the workflow and are globally accessible (hence the name) between jobs. For example:

```
from toil.common import Toil
from toil.job import Job
import os
import sys

def globalFileStoreJobFn(job):
    job.log("The following example exercises all the methods provided"
            " by the toil.fileStores.abstractFileStore.AbstractFileStore class")

    # Create a local temporary file.
    scratchFile = job.fileStore.getLocalTempFile()

    # Write something in the scratch file.
    with open(scratchFile, 'w') as fH:
        fH.write("What a tangled web we weave")

    # Write a copy of the file into the file-store; fileID is the key that can be␣
↪used to retrieve the file.
    # This write is asynchronous by default
    fileID = job.fileStore.writeGlobalFile(scratchFile)

    # Write another file using a stream; fileID2 is the
    # key for this second file.
    with job.fileStore.writeGlobalFileStream(cleanup=True) as (fH, fileID2):
        if sys.version_info >= (3, 0):
            # if python 3
            fH.write(b"Out brief candle")
        else:
            # if python 2
            fH.write("Out brief candle")

    # Now read the first file; scratchFile2 is a local copy of the file that is read-
↪only by default.
    scratchFile2 = job.fileStore.readGlobalFile(fileID)
```

(continues on next page)

(continued from previous page)

```
    # Read the second file to a desired location: scratchFile3.
    scratchFile3 = os.path.join(job.tempDir, "foo.txt")
    job.fileStore.readGlobalFile(fileID2, userPath=scratchFile3)

    # Read the second file again using a stream.
    with job.fileStore.readGlobalFileStream(fileID2) as fH:
        print(fH.read()) #This prints "Out brief candle"

    # Delete the first file from the global file-store.
    job.fileStore.deleteGlobalFile(fileID)

    # It is unnecessary to delete the file keyed by fileID2 because we used the␣
↪cleanup flag,
    # which removes the file after this job and all its successors have run (if the␣
↪file still exists)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        toil.start(Job.wrapJobFn(globalFileStoreJobFn))
```

The example demonstrates the global read, write and delete functionality of the file-store, using both local copies of the files and streams to read and write the files. It covers all the methods provided by the file store interface.

What is obvious is that the file-store provides no functionality to update an existing "global" file, meaning that files are, barring deletion, immutable. Also worth noting is that there is no file system hierarchy for files in the global file store. These limitations allow us to fairly easily support different object stores and to use caching to limit the amount of network file transfer between jobs.

11.12.1 Staging of Files into the Job Store

External files can be imported into or exported out of the job store prior to running a workflow when the *toil.common.Toil* context manager is used on the leader. The context manager provides methods *toil.common.Toil.importFile()*, and *toil.common.Toil.exportFile()* for this purpose. The destination and source locations of such files are described with URLs passed to the two methods. A list of the currently supported URLs can be found at *toil.jobStores.abstractJobStore.AbstractJobStore.importFile()*. To import an external file into the job store as a shared file, pass the optional `sharedFileName` parameter to that method.

If a workflow fails for any reason an imported file acts as any other file in the job store. If the workflow was configured such that it not be cleaned up on a failed run, the file will persist in the job store and needs not be staged again when the workflow is resumed.

Example:

```
import os
from toil.common import Toil
from toil.job import Job


class HelloWorld(Job):
    def __init__(self, id):
```

(continues on next page)

(continued from previous page)

```
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.inputFileID = id

    def run(self, fileStore):
        with self.fileStore.readGlobalFileStream(self.inputFileID) as fi:
            with self.fileStore.writeGlobalFileStream() as (fo, outputFileID):
                fo.write(fi.read() + 'World!')
        return outputFileID


if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        if not toil.options.restart:
            ioFileDirectory = os.path.join(os.path.dirname(os.path.abspath(__file__)),
↪ "stagingExampleFiles")
            inputFileID = toil.importFile("file://" + os.path.abspath(os.path.
↪join(ioFileDirectory, "in.txt")))
            outputFileID = toil.start(HelloWorld(inputFileID))
        else:
            outputFileID = toil.restart()

        toil.exportFile(outputFileID, "file://" + os.path.abspath(os.path.
↪join(ioFileDirectory, "out.txt")))
```

11.13 Using Docker Containers in Toil

Docker containers are commonly used with Toil. The combination of Toil and Docker allows for pipelines to be fully portable between any platform that has both Toil and Docker installed. Docker eliminates the need for the user to do any other tool installation or environment setup.

In order to use Docker containers with Toil, Docker must be installed on all workers of the cluster. Instructions for installing Docker can be found on the Docker website.

When using Toil-based autoscaling, Docker will be automatically set up on the cluster's worker nodes, so no additional installation steps are necessary. Further information on using Toil-based autoscaling can be found in the *Running a Workflow with Autoscaling* documentation.

In order to use docker containers in a Toil workflow, the container can be built locally or downloaded in real time from an online docker repository like Quay. If the container is not in a repository, the container's layers must be accessible on each node of the cluster.

When invoking docker containers from within a Toil workflow, it is strongly recommended that you use `dockerCall()`, a toil job function provided in `toil.lib.docker`. `dockerCall` leverages docker's own python API, and provides container cleanup on job failure. When docker containers are run without this feature, failed jobs can result in resource leaks. Docker's API can be found at docker-py.

In order to use `dockerCall`, your installation of Docker must be set up to run without `sudo`. Instructions for setting this up can be found here.

An example of a basic `dockerCall` is below:

```
dockerCall(job=job,
           tool='quay.io/ucsc_cgl/bwa',
           workDir=job.tempDir,
           parameters=['index', '/data/reference.fa'])
```

Note the assumption that *reference.fa* file is located in */data*. This is Toil's standard convention as a mount location to reduce boilerplate when calling *dockerCall*. Users can choose their own mount locations by supplying a *volumes* kwarg to *dockerCall*, such as: *volumes={working_dir: {'bind': '/data', 'mode': 'rw'}}*, where *working_dir* is an absolute path on the user's filesystem.

`dockerCall` can also be added to workflows like any other job function:

```
from toil.common import Toil
from toil.job import Job
from toil.lib.docker import apiDockerCall
import os

align = Job.wrapJobFn(apiDockerCall,
                      image='ubuntu',
                      working_dir=os.getcwd(),
                      parameters=['ls', '-lha'])

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
       toil.start(align)
```

cgl-docker-lib contains `dockerCall`-compatible Dockerized tools that are commonly used in bioinformatics analysis.

The documentation provides guidelines for developing your own Docker containers that can be used with Toil and `dockerCall`. In order for a container to be compatible with `dockerCall`, it must have an `ENTRYPOINT` set to a wrapper script, as described in cgl-docker-lib containerization standards. This can be set by passing in the optional keyword argument, 'entrypoint'. Example:

entrypoint=["/bin/bash","-c"]

dockerCall supports currently the 75 keyword arguments found in the python Docker API, under the 'run' command.

11.14 Services

It is sometimes desirable to run *services*, such as a database or server, concurrently with a workflow. The *`toil.job.Job.Service`* class provides a simple mechanism for spawning such a service within a Toil workflow, allowing precise specification of the start and end time of the service, and providing start and end methods to use for initialization and cleanup. The following simple, conceptual example illustrates how services work:

```
from toil.common import Toil
from toil.job import Job

class DemoService(Job.Service):

    def start(self, fileStore):
        # Start up a database/service here
```

(continues on next page)

(continued from previous page)

```
        # Return a value that enables another process to connect to the database
        return "loginCredentials"

    def check(self):
        # A function that if it returns False causes the service to quit
        # If it raises an exception the service is killed and an error is reported
        return True

    def stop(self, fileStore):
        # Cleanup the database here
        pass

j = Job()
s = DemoService()
loginCredentialsPromise = j.addService(s)

def dbFn(loginCredentials):
    # Use the login credentials returned from the service's start method to connect␣
↪to the service
    pass

j.addChildFn(dbFn, loginCredentialsPromise)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        toil.start(j)
```

In this example the DemoService starts a database in the start method, returning an object from the start method indicating how a client job would access the database. The service's stop method cleans up the database, while the service's check method is polled periodically to check the service is alive.

A DemoService instance is added as a service of the root job `j`, with resource requirements specified. The return value from *`toil.job.Job.addService()`* is a promise to the return value of the service's start method. When the promised is fulfilled it will represent how to connect to the database. The promise is passed to a child job of `j`, which uses it to make a database connection. The services of a job are started before any of its successors have been run and stopped after all the successors of the job have completed successfully.

Multiple services can be created per job, all run in parallel. Additionally, services can define sub-services using `toil.job.Job.Service.addChild()`. This allows complex networks of services to be created, e.g. Apache Spark clusters, within a workflow.

11.15 Checkpoints

Services complicate resuming a workflow after failure, because they can create complex dependencies between jobs. For example, consider a service that provides a database that multiple jobs update. If the database service fails and loses state, it is not clear that just restarting the service will allow the workflow to be resumed, because jobs that created that state may have already finished. To get around this problem Toil supports *checkpoint* jobs, specified as the boolean keyword argument `checkpoint` to a job or wrapped function, e.g.:

```
j = Job(checkpoint=True)
```

A checkpoint job is rerun if one or more of its successors fails its retry attempts, until it itself has exhausted its retry attempts. Upon restarting a checkpoint job all its existing successors are first deleted, and then the job is rerun to define new successors. By checkpointing a job that defines a service, upon failure of the service the database and the jobs that access the service can be redefined and rerun.

To make the implementation of checkpoint jobs simple, a job can only be a checkpoint if when first defined it has no successors, i.e. it can only define successors within its run method.

11.16 Encapsulation

Let `A` be a root job potentially with children and follow-ons. Without an encapsulated job the simplest way to specify a job `B` which runs after `A` and all its successors is to create a parent of `A`, call it `Ap`, and then make `B` a follow-on of `Ap`. e.g.:

```
from toil.common import Toil
from toil.job import Job

if __name__=="__main__":
    # A is a job with children and follow-ons, for example:
    A = Job()
    A.addChild(Job())
    A.addFollowOn(Job())

    # B is a job which needs to run after A and its successors
    B = Job()

    # The way to do this without encapsulation is to make a parent of A, Ap, and make␣
↪B a follow-on of Ap.
    Ap = Job()
    Ap.addChild(A)
    Ap.addFollowOn(B)

    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        print(toil.start(Ap))
```

An *encapsulated job* `E(A)` of `A` saves making `Ap`, instead we can write:

```
from toil.common import Toil
from toil.job import Job

if __name__=="__main__":
    # A
    A = Job()
    A.addChild(Job())
    A.addFollowOn(Job())

    # Encapsulate A
    A = A.encapsulate()

    # B is a job which needs to run after A and its successors
    B = Job()
```

(continues on next page)

(continued from previous page)

```
    # With encapsulation A and its successor subgraph appear to be a single job,␣
↪hence:
    A.addChild(B)

    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    options.clean = "always"

    with Toil(options) as toil:
        print(toil.start(A))
```

Note the call to *toil.job.Job.encapsulate()* creates the `toil.job.Job.EncapsulatedJob`.

11.17 Depending on Toil

If you are packing your workflow(s) as a pip-installable distribution on PyPI, you might be tempted to declare Toil as a dependency in your `setup.py`, via the `install_requires` keyword argument to `setup()`. Unfortunately, this does not work, for two reasons: For one, Toil uses Setuptools' *extra* mechanism to manage its own optional dependencies. If you explicitly declared a dependency on Toil, you would have to hard-code a particular combination of extras (or no extras at all), robbing the user of the choice what Toil extras to install. Secondly, and more importantly, declaring a dependency on Toil would only lead to Toil being installed on the leader node of a cluster, but not the worker nodes. Auto-deployment does not work here because Toil cannot auto-deploy itself, the classic "Which came first, chicken or egg?" problem.

In other words, you shouldn't explicitly depend on Toil. Document the dependency instead (as in "This workflow needs Toil version X.Y.Z to be installed") and optionally add a version check to your `setup.py`. Refer to the `check_version()` function in the `toil-lib` project's setup.py for an example. Alternatively, you can also just depend on `toil-lib` and you'll get that check for free.

If your workflow depends on a dependency of Toil, consider not making that dependency explicit either. If you do, you risk a version conflict between your project and Toil. The `pip` utility may silently ignore that conflict, breaking either Toil or your workflow. It is safest to simply assume that Toil installs that dependency for you. The only downside is that you are locked into the exact version of that dependency that Toil declares. But such is life with Python, which, unlike Java, has no means of dependencies belonging to different software components within the same process, and whose favored software distribution utility is incapable of properly resolving overlapping dependencies and detecting conflicts.

11.18 Best Practices for Dockerizing Toil Workflows

Computational Genomics Lab's Dockstore based production system provides workflow authors a way to run Dockerized versions of their pipeline in an automated, scalable fashion. To be compatible with this system of a workflow should meet the following requirements. In addition to the Docker container, a common workflow language descriptor file is needed. For inputs:

- Only command line arguments should be used for configuring the workflow. If the workflow relies on a configuration file, like Toil-RNAseq or ProTECT, a wrapper script inside the Docker container can be used to parse the CLI and generate the necessary configuration file.
- All inputs to the pipeline should be explicitly enumerated rather than implicit. For example, don't rely on one FASTQ read's path to discover the location of its pair. This is necessary since all inputs are mapped to their own isolated directories when the Docker is called via Dockstore.

- All inputs must be documented in the CWL descriptor file. Examples of this file can be seen in both Toil-RNAseq and ProTECT.

For outputs:

- All outputs should be written to a local path rather than S3.
- Take care to package outputs in a local and user-friendly way. For example, don't tar up all output if there are specific files that will care to see individually.
- All output file names should be deterministic and predictable. For example, don't prepend the name of an output file with PASS/FAIL depending on the outcome of the pipeline.
- All outputs must be documented in the CWL descriptor file. Examples of this file can be seen in both Toil-RNAseq and ProTECT.

CHAPTER 12

Toil Class API

The Toil class configures and starts a Toil run.

class toil.common.**Toil**(*options*)

A context manager that represents a Toil workflow, specifically the batch system, job store, and its configuration.

__init__(*options*)

Initialize a Toil object from the given options. Note that this is very light-weight and that the bulk of the work is done when the context is entered.

Parameters **options** (*argparse.Namespace*) – command line options specified by the user

config = None

Type toil.common.Config

start(*rootJob*)

Invoke a Toil workflow with the given job as the root for an initial run. This method must be called in the body of a `with Toil(...) as toil:` statement. This method should not be called more than once for a workflow that has not finished.

Parameters **rootJob** (toil.job.Job) – The root job of the workflow

Returns The root job's return value

restart()

Restarts a workflow that has been interrupted.

Returns The root job's return value

classmethod getJobStore(*locator*)

Create an instance of the concrete job store implementation that matches the given locator.

Parameters **locator** (*str*) – The location of the job store to be represent by the instance

Returns an instance of a concrete subclass of AbstractJobStore

Return type *toil.jobStores.abstractJobStore.AbstractJobStore*

static createBatchSystem(*config*)
Creates an instance of the batch system specified in the given config.

Parameters **config** (*toil.common.Config*) – the current configuration

Return type *batchSystems.abstractBatchSystem.AbstractBatchSystem*

Returns an instance of a concrete subclass of AbstractBatchSystem

importFile(*srcUrl*, *sharedFileName=None*)
Imports the file at the given URL into job store.

See *toil.jobStores.abstractJobStore.AbstractJobStore.importFile()* for a full description

exportFile(*jobStoreFileID*, *dstUrl*)
Exports file to destination pointed at by the destination URL.

See *toil.jobStores.abstractJobStore.AbstractJobStore.exportFile()* for a full description

static getWorkflowDir(*workflowID*, *configWorkDir=None*)
Returns a path to the directory where worker directories and the cache will be located for this workflow.

Parameters

- **workflowID** (*str*) – Unique identifier for the workflow
- **configWorkDir** (*str*) – Value passed to the program using the –workDir flag

Returns Path to the workflow directory

Return type str

writePIDFile()
Write a the pid of this process to a file in the jobstore.

Overwriting the current contents of pid.log is a feature, not a bug of this method. Other methods will rely on always having the most current pid available. So far there is no reason to store any old pids.

CHAPTER 13

Job Store API

The job store interface is an abstraction layer that that hides the specific details of file storage, for example standard file systems, S3, etc. The *AbstractJobStore* API is implemented to support a give file store, e.g. S3. Implement this API to support a new file store.

class toil.jobStores.abstractJobStore.**AbstractJobStore**

Represents the physical storage for the jobs and files in a Toil workflow.

__init__()

Create an instance of the job store. The instance will not be fully functional until either *initialize()* or *resume()* is invoked. Note that the *destroy()* method may be invoked on the object with or without prior invocation of either of these two methods.

initialize(*config*)

Create the physical storage for this job store, allocate a workflow ID and persist the given Toil configuration to the store.

Parameters **config** (*toil.common.Config*) – the Toil configuration to initialize this job store with. The given configuration will be updated with the newly allocated workflow ID.

Raises ***JobStoreExistsException*** – if the physical storage for this job store already exists

writeConfig()

Persists the value of the *AbstractJobStore.config* attribute to the job store, so that it can be retrieved later by other instances of this class.

resume()

Connect this instance to the physical storage it represents and load the Toil configuration into the *AbstractJobStore.config* attribute.

Raises ***NoSuchJobStoreException*** – if the physical storage for this job store doesn't exist

config

The Toil configuration associated with this job store.

Return type toil.common.Config

setRootJob(*rootJobStoreID*)
Set the root job of the workflow backed by this job store

Parameters **rootJobStoreID** (*str*) – The ID of the job to set as root

loadRootJob()
Loads the root job in the current job store.

Raises ***toil.job.JobException*** – If no root job is set or if the root job doesn't exist in this job store

Returns The root job.

Return type toil.jobGraph.JobGraph

createRootJob(**args*, ***kwargs*)
Create a new job and set it as the root job in this job store

Return type toil.jobGraph.JobGraph

getRootJobReturnValue()
Parse the return value from the root job.

Raises an exception if the root job hasn't fulfilled its promise yet.

importFile(*srcUrl*, *sharedFileName=None*, *hardlink=False*)
Imports the file at the given URL into job store. The ID of the newly imported file is returned. If the name of a shared file name is provided, the file will be imported as such and None is returned.

Currently supported schemes are:

- **'s3' for objects in Amazon S3** e.g. s3://bucket/key
- **'file' for local files** e.g. file:///local/file/path
- **'http'** e.g. http://someurl.com/path
- **'gs'** e.g. gs://bucket/file

Parameters

- **srcUrl** (*str*) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket.
- **sharedFileName** (*str*) – Optional name to assign to the imported file within the job store

Returns The jobStoreFileId of the imported file or None if sharedFileName was given

Return type *toil.fileStores.FileID* or None

exportFile(*jobStoreFileID*, *dstUrl*)
Exports file to destination pointed at by the destination URL.

Refer to *AbstractJobStore.importFile()* documentation for currently supported URL schemes.

Note that the helper method _exportFile is used to read from the source and write to destination. To implement any optimizations that circumvent this, the _exportFile method should be overridden by subclasses of AbstractJobStore.

Parameters

- **jobStoreFileID** (*str*) – The id of the file in the job store that should be exported.
- **dstUrl** (*str*) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket.

classmethod getSize(*url*)
returns the size in bytes of the file at the given URL

Parameters **url** (*urlparse.ParseResult*) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an AWS s3 bucket.

destroy()
The inverse of *initialize()*, this method deletes the physical storage represented by this instance. While not being atomic, this method *is* at least idempotent, as a means to counteract potential issues with eventual consistency exhibited by the underlying storage mechanisms. This means that if the method fails (raises an exception), it may (and should be) invoked again. If the underlying storage mechanism is eventually consistent, even a successful invocation is not an ironclad guarantee that the physical storage vanished completely and immediately. A successful invocation only guarantees that the deletion will eventually happen. It is therefore recommended to not immediately reuse the same job store location for a new Toil workflow.

getEnv()
Returns a dictionary of environment variables that this job store requires to be set in order to function properly on a worker.

Return type dict[str,str]

clean(*jobCache=None*)
Function to cleanup the state of a job store after a restart. Fixes jobs that might have been partially updated. Resets the try counts and removes jobs that are not successors of the current root job.

Parameters **jobCache** (*dict[str,toil.jobGraph.JobGraph]*) – if a value it must be a dict from job ID keys to JobGraph object values. Jobs will be loaded from the cache (which can be downloaded from the job store in a batch) instead of piecemeal when recursed into.

batch(***kwds*)
All calls to create() with this context manager active will be performed in a batch after the context manager is released.

Return type None

create(*jobNode*)
Creates a job graph from the given job node & writes it to the job store.

Return type toil.jobGraph.JobGraph

exists(*jobStoreID*)
Indicates whether the job with the specified jobStoreID exists in the job store

Return type bool

getPublicUrl(*fileName*)
Returns a publicly accessible URL to the given file in the job store. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters **fileName** (*str*) – the jobStoreFileID of the file to generate a URL for

Raises ***NoSuchFileException*** – if the specified file does not exist in this job store

Return type str

getSharedPublicUrl(*sharedFileName*)
Differs from *getPublicUrl()* in that this method is for generating URLs for shared files written by *writeSharedFileStream()*.

Returns a publicly accessible URL to the given file in the job store. The returned URL starts with 'http:', 'https:' or 'file:'. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters **sharedFileName** (*str*) – The name of the shared file to generate a publically accessible url for.

Raises ***NoSuchFileException*** – raised if the specified file does not exist in the store

Return type str

load(*jobStoreID*)

Loads the job referenced by the given ID and returns it.

Parameters **jobStoreID** (*str*) – the ID of the job to load

Raises ***NoSuchJobException*** – if there is no job with the given ID

Return type toil.jobGraph.JobGraph

update(*job*)

Persists the job in this store atomically.

Parameters **job** (*toil.jobGraph.JobGraph*) – the job to write to this job store

delete(*jobStoreID*)

Removes from store atomically, can not then subsequently call load(), write(), update(), etc. with the job.

This operation is idempotent, i.e. deleting a job twice or deleting a non-existent job will succeed silently.

Parameters **jobStoreID** (*str*) – the ID of the job to delete from this job store

jobs()

Best effort attempt to return iterator on all jobs in the store. The iterator may not return all jobs and may also contain orphaned jobs that have already finished successfully and should not be rerun. To guarantee you get any and all jobs that can be run instead construct a more expensive ToilState object

Returns Returns iterator on jobs in the store. The iterator may or may not contain all jobs and may contain invalid jobs

Return type Iterator[toil.jobGraph.JobGraph]

writeFile(*localFilePath*, *jobStoreID=None*, *cleanup=False*)

Takes a file (as a path) and places it in this job store. Returns an ID that can be used to retrieve the file at a later time. The file is written in a atomic manner. It will not appear in the jobStore until the write has successfully completed.

Parameters

- **localFilePath** (*str*) – the path to the local file that will be uploaded to the job store.
- **jobStoreID** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.

Raises

- ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method
- ***NoSuchJobException*** – if the job specified via jobStoreID does not exist

FIXME: some implementations may not raise this

Returns an ID referencing the newly created file and can be used to read the file in the future.

Return type str

writeFileStream(***kwds*)

Similar to writeFile, but returns a context manager yielding a tuple of 1) a file handle which can be written to and 2) the ID of the resulting file in the job store. The yielded file handle does not need to and should not be closed explicitly. The file is written in a atomic manner. It will not appear in the jobStore until the write has successfully completed.

Parameters

- **jobStoreID** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.

Raises

- ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method
- ***NoSuchJobException*** – if the job specified via jobStoreID does not exist

FIXME: some implementations may not raise this

Returns an ID that references the newly created file and can be used to read the file in the future.

Return type str

getEmptyFileStoreID(*jobStoreID=None*, *cleanup=False*)

Creates an empty file in the job store and returns its ID. Call to fileExists(getEmptyFileStoreID(jobStoreID)) will return True.

Parameters

- **jobStoreID** (*str*) – the id of a job, or None. If specified, the may be associated with that job in a job-store-specific way. This may influence the returned ID.
- **cleanup** (*bool*) – Whether to attempt to delete the file when the job whose jobStoreID was given as jobStoreID is deleted with jobStore.delete(job). If jobStoreID was not given, does nothing.

Returns a jobStoreFileID that references the newly created file and can be used to reference the file in the future.

Return type str

readFile(*jobStoreFileID*, *localFilePath*, *symlink=False*)

Copies or hard links the file referenced by jobStoreFileID to the given local file path. The version will be consistent with the last copy of the file written/updated. If the file in the job store is later modified via updateFile or updateFileStream, it is implementation-defined whether those writes will be visible at localFilePath. The file is copied in an atomic manner. It will not appear in the local file system until the copy has completed.

The file at the given local path may not be modified after this method returns!

Parameters

- **jobStoreFileID** (*str*) – ID of the file to be copied
- **localFilePath** (*str*) – the local path indicating where to place the contents of the given file in the job store

- **symlink** (*bool*) – whether the reader can tolerate a symlink. If set to true, the job store may create a symlink instead of a full copy of the file or a hard link.

readFileStream(***kwds*)
Similar to readFile, but returns a context manager yielding a file handle which can be read from. The yielded file handle does not need to and should not be closed explicitly.

Parameters **jobStoreFileID** (*str*) – ID of the file to get a readable file handle for

deleteFile(*jobStoreFileID*)
Deletes the file with the given ID from this job store. This operation is idempotent, i.e. deleting a file twice or deleting a non-existent file will succeed silently.

Parameters **jobStoreFileID** (*str*) – ID of the file to delete

fileExists(*jobStoreFileID*)
Determine whether a file exists in this job store.

Parameters **jobStoreFileID** (*str*) – an ID referencing the file to be checked

Return type bool

getFileSize(*jobStoreFileID*)
Get the size of the given file in bytes, or 0 if it does not exist when queried.

Note that job stores which encrypt files might return overestimates of file sizes, since the encrypted file may have been padded to the nearest block, augmented with an initialization vector, etc.

Parameters **jobStoreFileID** (*str*) – an ID referencing the file to be checked

Return type int

updateFile(*jobStoreFileID*, *localFilePath*)
Replaces the existing version of a file in the job store. Throws an exception if the file does not exist.

Parameters

- **jobStoreFileID** (*str*) – the ID of the file in the job store to be updated
- **localFilePath** (*str*) – the local path to a file that will overwrite the current version in the job store

Raises

- ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method
- ***NoSuchFileException*** – if the specified file does not exist

updateFileStream(*jobStoreFileID*)
Replaces the existing version of a file in the job store. Similar to writeFile, but returns a context manager yielding a file handle which can be written to. The yielded file handle does not need to and should not be closed explicitly.

Parameters **jobStoreFileID** (*str*) – the ID of the file in the job store to be updated

Raises

- ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method
- ***NoSuchFileException*** – if the specified file does not exist

writeSharedFileStream(***kwds*)
Returns a context manager yielding a writable file handle to the global file referenced by the given name. File will be created in an atomic manner.

Parameters

- **sharedFileName** (*str*) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store
- **isProtected** (*bool*) – True if the file must be encrypted, None if it may be encrypted or False if it must be stored in the clear.

Raises ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method

readSharedFileStream(***kwds*)

Returns a context manager yielding a readable file handle to the global file referenced by the given name.

Parameters **sharedFileName** (*str*) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store

writeStatsAndLogging(*statsAndLoggingString*)

Adds the given statistics/logging string to the store of statistics info.

Parameters **statsAndLoggingString** (*str*) – the string to be written to the stats file

Raises ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method

readStatsAndLogging(*callback*, *readAll=False*)

Reads stats/logging strings accumulated by the writeStatsAndLogging() method. For each stats/logging string this method calls the given callback function with an open, readable file handle from which the stats string can be read. Returns the number of stats/logging strings processed. Each stats/logging string is only processed once unless the readAll parameter is set, in which case the given callback will be invoked for all existing stats/logging strings, including the ones from a previous invocation of this method.

Parameters

- **callback** (*Callable*) – a function to be applied to each of the stats file handles found
- **readAll** (*bool*) – a boolean indicating whether to read the already processed stats files in addition to the unread stats files

Raises ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method

Returns the number of stats files processed

Return type int

CHAPTER 14

Toil Job API

Functions to wrap jobs and return values (promises).

14.1 FunctionWrappingJob

The subclass of Job for wrapping user functions.

class toil.job.**FunctionWrappingJob**(*userFunction*, **args*, ***kwargs*)

Job used to wrap a function. In its *run* method the wrapped function is called.

__init__(*userFunction*, **args*, ***kwargs*)

Parameters **userFunction** (*callable*) – The function to wrap. It will be called with `*args` and `**kwargs` as arguments.

The keywords `memory`, `cores`, `disk`, `preemptable` and `checkpoint` are reserved keyword arguments that if specified will be used to determine the resources required for the job, as *toil.job.Job.__init__()*. If they are keyword arguments to the function they will be extracted from the function definition, but may be overridden by the user (as you would expect).

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters **fileStore** (toil.fileStores.abstractFileStore.AbstractFileStore) – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

14.2 JobFunctionWrappingJob

The subclass of FunctionWrappingJob for wrapping user job functions.

class toil.job.**JobFunctionWrappingJob**(*userFunction*, **args*, ***kwargs*)

A job function is a function whose first argument is a *Job* instance that is the wrapping job for the function. This can be used to add successor jobs for the function and perform all the functions the *Job* class provides.

To enable the job function to get access to the *toil.fileStores.abstractFileStore.AbstractFileStore* instance (see *toil.job.Job.run()*), it is made a variable of the wrapping job called fileStore.

To specify a job's resource requirements the following default keyword arguments can be specified:

- memory
- disk
- cores

For example to wrap a function into a job we would call:

```
Job.wrapJobFn(myJob, memory='100k', disk='1M', cores=0.1)
```

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters **fileStore** (toil.fileStores.abstractFileStore.AbstractFileStore) – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

14.3 EncapsulatedJob

The subclass of Job for *encapsulating* a job, allowing a subgraph of jobs to be treated as a single job.

class toil.job.**EncapsulatedJob**(*job*)

A convenience Job class used to make a job subgraph appear to be a single job.

Let A be the root job of a job subgraph and B be another job we'd like to run after A and all its successors have completed, for this use encapsulate:

```
#  Job A and subgraph, Job B
A, B = A(), B()
A' = A.encapsulate()
A'.addChild(B)
#  B will run after A and all its successors have completed, A and its subgraph of
# successors in effect appear to be just one job.
```

If the job being encapsulated has predecessors (e.g. is not the root job), then the encapsulated job will inherit these predecessors. If predecessors are added to the job being encapsulated after the encapsulated job is created then the encapsulating job will NOT inherit these predecessors automatically. Care should be exercised to ensure the encapsulated job has the proper set of predecessors.

The return value of an encapsulatd job (as accessed by the *toil.job.Job.rv()* function) is the return value of the root job, e.g. A().encapsulate().rv() and A().rv() will resolve to the same value after A or A.encapsulate() has been run.

__init__(*job*)

Parameters **job** (toil.job.Job) – the job to encapsulate.

addChild(*childJob*)
Adds childJob to be run as child of this job. Child jobs will be run directly after this job's *toil.job.Job.run()* method has completed.

Parameters **childJob** (toil.job.Job) –

Returns childJob

Return type *toil.job.Job*

addService(*service*, *parentService=None*)
Add a service.

The *toil.job.Job.Service.start()* method of the service will be called after the run method has completed but before any successors are run. The service's *toil.job.Job.Service.stop()* method will be called once the successors of the job have been run.

Services allow things like databases and servers to be started and accessed by jobs in a workflow.

Raises ***toil.job.JobException*** – If service has already been made the child of a job or another service.

Parameters

- **service** (toil.job.Job.Service) – Service to add.
- **parentService** (toil.job.Job.Service) – Service that will be started before 'service' is started. Allows trees of services to be established. parentService must be a service of this job.

Returns a promise that will be replaced with the return value from *toil.job.Job.Service.start()* of service in any successor of the job.

Return type *toil.job.Promise*

addFollowOn(*followOnJob*)
Adds a follow-on job, follow-on jobs will be run after the child jobs and their successors have been run.

Parameters **followOnJob** (toil.job.Job) –

Returns followOnJob

Return type *toil.job.Job*

rv(**path*)
Creates a *promise* (*toil.job.Promise*) representing a return value of the job's run method, or, in case of a function-wrapping job, the wrapped function's return value.

Parameters **path** (*(Any)*) – Optional path for selecting a component of the promised return value. If absent or empty, the entire return value will be used. Otherwise, the first element of the path is used to select an individual item of the return value. For that to work, the return value must be a list, dictionary or of any other type implementing the *__getitem__()* magic method. If the selected item is yet another composite value, the second element of the path can be used to select an item from it, and so on. For example, if the return value is *[6,{'a':42}]*, *.rv(0)* would select *6* , *rv(1)* would select *{'a':3}* while *rv(1,'a')* would select *3*. To select a slice from a return value that is slicable, e.g. tuple or list, the path element should be a *slice* object. For example, assuming that the return value is *[6, 7, 8, 9]* then *.rv(slice(1, 3))* would select *[7, 8]*. Note that slicing really only makes sense at the end of path.

Returns A promise representing the return value of this jobs *toil.job.Job.run()* method.

Return type *toil.job.Promise*

prepareForPromiseRegistration(*jobStore*)
Ensure that a promise by this job (the promissor) can register with the promissor when another job referring to the promise (the promissee) is being serialized. The promissee holds the reference to the promise (usually as part of the the job arguments) and when it is being pickled, so will the promises it refers to. Pickling a promise triggers it to be registered with the promissor.

Returns

14.4 Promise

The class used to reference return values of jobs/services not yet run/started.

class toil.job.**Promise**(*job*, *path*)
References a return value from a *toil.job.Job.run()* or *toil.job.Job.Service.start()* method as a *promise* before the method itself is run.

Let T be a job. Instances of *Promise* (termed a *promise*) are returned by T.rv(), which is used to reference the return value of T's run function. When the promise is passed to the constructor (or as an argument to a wrapped function) of a different, successor job the promise will be replaced by the actual referenced return value. This mechanism allows a return values from one job's run method to be input argument to job before the former job's run function has been executed.

filesToDelete = set([])
A set of IDs of files containing promised values when we know we won't need them anymore

__init__(*job*, *path*)

Parameters

- **job** (Job) – the job whose return value this promise references
- **path** – see *Job.rv()*

class toil.job.**PromisedRequirement**(*valueOrCallable*, **args*)

__init__(*valueOrCallable*, **args*)
Class for dynamically allocating job function resource requirements involving *toil.job.Promise* instances.

Use when resource requirements depend on the return value of a parent function. PromisedRequirements can be modified by passing a function that takes the *Promise* as input.

For example, let f, g, and h be functions. Then a Toil workflow can be defined as follows:: A = Job.wrapFn(f) B = A.addChildFn(g, cores=PromisedRequirement(A.rv()) C = B.addChildFn(h, cores=PromisedRequirement(lambda x: 2*x, B.rv()))

Parameters

- **valueOrCallable** – A single Promise instance or a function that takes *args as input parameters.
- ***args** (*int or Promise*) – variable length argument list

getValue()
Returns PromisedRequirement value

static convertPromises(*kwargs*)
Returns True if reserved resource keyword is a Promise or PromisedRequirement instance. Converts Promise instance to PromisedRequirement.

Parameters **kwargs** – function keyword arguments

Returns bool

CHAPTER 15

Job Methods API

Jobs are the units of work in Toil which are composed into workflows.

class toil.job.**Job**(*memory=None*, *cores=None*, *disk=None*, *preemptable=None*, *unitName=None*, *checkpoint=False*, *displayName=None*)

Class represents a unit of work in toil.

__init__(*memory=None*, *cores=None*, *disk=None*, *preemptable=None*, *unitName=None*, *checkpoint=False*, *displayName=None*)

This method must be called by any overriding constructor.

Parameters

- **memory** (*int or string convertible by toil.lib.humanize.human2bytes to an int*) – the maximum number of bytes of memory the job will require to run.
- **cores** (*int or string convertible by toil.lib.humanize.human2bytes to an int*) – the number of CPU cores required.
- **disk** (*int or string convertible by toil.lib.humanize.human2bytes to an int*) – the amount of local disk space required by the job, expressed in bytes.
- **preemptable** (*bool*) – if the job can be run on a preemptable node.
- **checkpoint** – if any of this job's successor jobs completely fails, exhausting all their retries, remove any successor jobs and rerun this job to restart the subtree. Job must be a leaf vertex in the job graph when initially defined, see toil.job.Job.checkNewCheckpointsAreCutVertices().

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters **fileStore** (toil.fileStores.abstractFileStore.AbstractFileStore) – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

addChild(*childJob*)
Adds childJob to be run as child of this job. Child jobs will be run directly after this job's *toil.job.Job.run()* method has completed.

Parameters **childJob** (toil.job.Job) –

Returns childJob

Return type *toil.job.Job*

hasChild(*childJob*)
Check if childJob is already a child of this job.

Parameters **childJob** (toil.job.Job) –

Returns True if childJob is a child of the job, else False.

Return type bool

addFollowOn(*followOnJob*)
Adds a follow-on job, follow-on jobs will be run after the child jobs and their successors have been run.

Parameters **followOnJob** (toil.job.Job) –

Returns followOnJob

Return type *toil.job.Job*

hasFollowOn(*followOnJob*)
Check if given job is already a follow-on of this job.

Parameters **followOnJob** (toil.job.Job) –

Returns True if the followOnJob is a follow-on of this job, else False.

Return type bool

addService(*service*, *parentService=None*)
Add a service.

The *toil.job.Job.Service.start()* method of the service will be called after the run method has completed but before any successors are run. The service's *toil.job.Job.Service.stop()* method will be called once the successors of the job have been run.

Services allow things like databases and servers to be started and accessed by jobs in a workflow.

Raises ***toil.job.JobException*** – If service has already been made the child of a job or another service.

Parameters

- **service** (toil.job.Job.Service) – Service to add.
- **parentService** (toil.job.Job.Service) – Service that will be started before 'service' is started. Allows trees of services to be established. parentService must be a service of this job.

Returns a promise that will be replaced with the return value from *toil.job.Job.Service.start()* of service in any successor of the job.

Return type *toil.job.Promise*

addChildFn(*fn*, **args*, ***kwargs*)
Adds a function as a child job.

Parameters **fn** – Function to be run as a child job with `*args` and `**kwargs` as arguments to this function. See toil.job.FunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns The new child job that wraps fn.

Return type *toil.job.FunctionWrappingJob*

addFollowOnFn (*fn*, **args*, ***kwargs*)
Adds a function as a follow-on job.

Parameters **fn** – Function to be run as a follow-on job with `*args` and `**kwargs` as arguments to this function. See toil.job.FunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns The new follow-on job that wraps fn.

Return type *toil.job.FunctionWrappingJob*

addChildJobFn (*fn*, **args*, ***kwargs*)
Adds a job function as a child job. See *toil.job.JobFunctionWrappingJob* for a definition of a job function.

Parameters **fn** – Job function to be run as a child job with `*args` and `**kwargs` as arguments to this function. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns The new child job that wraps fn.

Return type *toil.job.JobFunctionWrappingJob*

addFollowOnJobFn (*fn*, **args*, ***kwargs*)
Add a follow-on job function. See *toil.job.JobFunctionWrappingJob* for a definition of a job function.

Parameters **fn** – Job function to be run as a follow-on job with `*args` and `**kwargs` as arguments to this function. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns The new follow-on job that wraps fn.

Return type *toil.job.JobFunctionWrappingJob*

tempDir
Shortcut to calling `job.fileStore.getLocalTempDir()`. Temp dir is created on first call and will be returned for first and future calls :return: Path to tempDir. See *job.fileStore.getLocalTempDir* :rtype: str

log (*text*, *level=20*)
convenience wrapper for `fileStore.logToMaster()`

static wrapFn (*fn*, **args*, ***kwargs*)
Makes a Job out of a function. Convenience function for constructor of *toil.job.FunctionWrappingJob*.

Parameters **fn** – Function to be run with `*args` and `**kwargs` as arguments. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns The new function that wraps fn.

Return type *toil.job.FunctionWrappingJob*

static wrapJobFn(*fn*, **args*, ***kwargs*)
Makes a Job out of a job function. Convenience function for constructor of *toil.job.JobFunctionWrappingJob*.

Parameters **fn** – Job function to be run with `*args` and `**kwargs` as arguments. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns The new job function that wraps fn.

Return type *toil.job.JobFunctionWrappingJob*

encapsulate()
Encapsulates the job, see *toil.job.EncapsulatedJob*. Convenience function for constructor of *toil.job.EncapsulatedJob*.

Returns an encapsulated version of this job.

Return type *toil.job.EncapsulatedJob*

rv(**path*)
Creates a *promise* (*toil.job.Promise*) representing a return value of the job's run method, or, in case of a function-wrapping job, the wrapped function's return value.

Parameters **path** (*(Any)*) – Optional path for selecting a component of the promised return value. If absent or empty, the entire return value will be used. Otherwise, the first element of the path is used to select an individual item of the return value. For that to work, the return value must be a list, dictionary or of any other type implementing the *__getitem__()* magic method. If the selected item is yet another composite value, the second element of the path can be used to select an item from it, and so on. For example, if the return value is *[6,{'a':42}]*, *.rv(0)* would select *6* , *rv(1)* would select *{'a':3}* while *rv(1,'a')* would select *3*. To select a slice from a return value that is slicable, e.g. tuple or list, the path element should be a *slice* object. For example, assuming that the return value is *[6, 7, 8, 9]* then *.rv(slice(1, 3))* would select *[7, 8]*. Note that slicing really only makes sense at the end of path.

Returns A promise representing the return value of this jobs *toil.job.Job.run()* method.

Return type *toil.job.Promise*

prepareForPromiseRegistration(*jobStore*)
Ensure that a promise by this job (the promissor) can register with the promissor when another job referring to the promise (the promissee) is being serialized. The promissee holds the reference to the promise (usually as part of the the job arguments) and when it is being pickled, so will the promises it refers to. Pickling a promise triggers it to be registered with the promissor.

Returns

checkJobGraphForDeadlocks()
See *toil.job.Job.checkJobGraphConnected()*, toil.job.Job.checkJobGraphAcyclic() and *toil.job.Job.checkNewCheckpointsAreLeafVertices()* for more info.

Raises ***toil.job.JobGraphDeadlockException*** – if the job graph is cyclic, contains multiple roots or contains checkpoint jobs that are not leaf vertices when defined (see toil.job.Job.checkNewCheckpointsAreLeaves()).

getRootJobs()

Returns The roots of the connected component of jobs that contains this job. A root is a job with no predecessors.

:rtype : set of toil.job.Job instances

checkJobGraphConnected()

Raises ***toil.job.JobGraphDeadlockException*** – if *toil.job.Job.getRootJobs()* does not contain exactly one root job.

As execution always starts from one root job, having multiple root jobs will cause a deadlock to occur.

checkJobGraphAcylic()

Raises ***toil.job.JobGraphDeadlockException*** – if the connected component of jobs containing this job contains any cycles of child/followOn dependencies in the *augmented job graph* (see below). Such cycles are not allowed in valid job graphs.

A follow-on edge (A, B) between two jobs A and B is equivalent to adding a child edge to B from (1) A, (2) from each child of A, and (3) from the successors of each child of A. We call each such edge an edge an "implied" edge. The augmented job graph is a job graph including all the implied edges.

For a job graph G = (V, E) the algorithm is `O(|V|^2)`. It is `O(|V| + |E|)` for a graph with no follow-ons. The former follow-on case could be improved!

checkNewCheckpointsAreLeafVertices()

A checkpoint job is a job that is restarted if either it fails, or if any of its successors completely fails, exhausting their retries.

A job is a leaf it is has no successors.

A checkpoint job must be a leaf when initially added to the job graph. When its run method is invoked it can then create direct successors. This restriction is made to simplify implementation.

Raises ***toil.job.JobGraphDeadlockException*** – if there exists a job being added to the graph for which checkpoint=True and which is not a leaf.

defer(*function*, **args*, ***kwargs*)

Register a deferred function, i.e. a callable that will be invoked after the current attempt at running this job concludes. A job attempt is said to conclude when the job function (or the *toil.job.Job.run()* method for class-based jobs) returns, raises an exception or after the process running it terminates abnormally. A deferred function will be called on the node that attempted to run the job, even if a subsequent attempt is made on another node. A deferred function should be idempotent because it may be called multiple times on the same node or even in the same process. More than one deferred function may be registered per job attempt by calling this method repeatedly with different arguments. If the same function is registered twice with the same or different arguments, it will be called twice per job attempt.

Examples for deferred functions are ones that handle cleanup of resources external to Toil, like Docker containers, files outside the work directory, etc.

Parameters

- **function** (*callable*) – The function to be called after this job concludes.
- **args** (*list*) – The arguments to the function
- **kwargs** (*dict*) – The keyword arguments to the function

getTopologicalOrderingOfJobs()

Returns a list of jobs such that for all pairs of indices i, j for which i < j, the job at index i can be run before the job at index j.

Return type list

CHAPTER 16

Job.Runner API

The Runner contains the methods needed to configure and start a Toil run.

class Job.**Runner**
Used to setup and run Toil workflow.

static getDefaultArgumentParser()
Get argument parser with added toil workflow options.

Returns The argument parser used by a toil workflow with added Toil options.

Return type argparse.ArgumentParser

static getDefaultOptions(*jobStore*)
Get default options for a toil workflow.

Parameters **jobStore** (*string*) – A string describing the jobStore for the workflow.

Returns The options used by a toil workflow.

Return type argparse.ArgumentParser values object

static addToilOptions(*parser*)
Adds the default toil options to an optparse or argparse parser object.

Parameters **parser** (*optparse.OptionParser or argparse.ArgumentParser*) – Options object to add toil options to.

static startToil(*job*, *options*)
Deprecated by toil.common.Toil.start. Runs the toil workflow using the given options (see Job.Runner.getDefaultOptions and Job.Runner.addToilOptions) starting with this job. :param toil.job.Job job: root job of the workflow :raises: toil.leader.FailedJobsException if at the end of function their remain failed jobs. :return: The return value of the root job's run function. :rtype: Any

CHAPTER 17

job.fileStore API

The AbstractFileStore is an abstraction of a Toil run's shared storage.

class toil.fileStores.abstractFileStore.**AbstractFileStore**(*jobStore*, *jobGraph*, *localTempDir*, *waitForPreviousCommit*)

Interface used to allow user code run by Toil to read and write files.

Also provides the interface to other Toil facilities used by user code, including:

- normal (non-real-time) logging
- finding the correct temporary directory for scratch work
- importing and exporting files into and out of the workflow

Stores user files in the jobStore, but keeps them separate from actual jobs.

May implement caching.

Passed as argument to the *toil.job.Job.run()* method.

Access to files is only permitted inside the context manager provided by *toil.fileStores.abstractFileStore.AbstractFileStore.open()*.

Also responsible for committing completed jobs back to the job store with an update operation, and allowing that commit operation to be waited for.

__init__(*jobStore*, *jobGraph*, *localTempDir*, *waitForPreviousCommit*)

Create a new file store object.

Parameters

- **jobStore** (toil.jobStores.abstractJobStore.AbstractJobStore) – the job store in use for the current Toil run.
- **jobGraph** (*toil.jobGraph.JobGraph*) – the job graph object for the currently running job.
- **localTempDir** (*str*) – the per-worker local temporary directory, under which per-job directories will be created.

- **waitForPreviousCommit** – the waitForCommit method of the previous job's file store, when jobs are running in sequence on the same worker. Used to prevent this file store's startCommit and the previous job's startCommit methods from running at the same time and racing. If they did race, it might be possible for the later job to be fully marked as completed in the job store before the eralier job was.

static shutdownFileStore(*workflowDir*, *workflowID*)
Carry out any necessary filestore-specific cleanup.

This is a destructive operation and it is important to ensure that there are no other running processes on the system that are modifying or using the file store for this workflow.

This is the intended to be the last call to the file store in a Toil run, called by the batch system cleanup function upon batch system shutdown.

Parameters

- **workflowDir** (*str*) – The path to the cache directory
- **workflowID** (*str*) – The workflow ID for this invocation of the workflow

open(***kwds*)
The context manager used to conduct tasks prior-to, and after a job has been run. File operations are only permitted inside the context manager.

Parameters **job** (toil.job.Job) – The job instance of the toil job to run.

getLocalTempDir()
Get a new local temporary directory in which to write files that persist for the duration of the job.

Returns The absolute path to a new local temporary directory. This directory will exist for the duration of the job only, and is guaranteed to be deleted once the job terminates, removing all files it contains recursively.

Return type str

getLocalTempFile()
Get a new local temporary file that will persist for the duration of the job.

Returns The absolute path to a local temporary file. This file will exist for the duration of the job only, and is guaranteed to be deleted once the job terminates.

Return type str

getLocalTempFileName()
Get a valid name for a new local file. Don't actually create a file at the path.

Returns Path to valid file

Return type str

writeGlobalFile(*localFileName*, *cleanup=False*)
Takes a file (as a path) and uploads it to the job store.

Parameters

- **localFileName** (*string*) – The path to the local file to upload.
- **cleanup** (*bool*) – if True then the copy of the global file will be deleted once the job and all its successors have completed running. If not the global file must be deleted manually.

Returns an ID that can be used to retrieve the file.

Return type *toil.fileStores.FileID*

writeGlobalFileStream(***kwds*)
Similar to writeGlobalFile, but allows the writing of a stream to the job store. The yielded file handle does not need to and should not be closed explicitly.

Parameters **cleanup** (*bool*) – is as in *toil.fileStores.abstractFileStore.AbstractFileStore.writeGlobalFile()*.

Returns A context manager yielding a tuple of 1) a file handle which can be written to and 2) the toil.fileStores.FileID of the resulting file in the job store.

readGlobalFile(*fileStoreID*, *userPath=None*, *cache=True*, *mutable=False*, *symlink=False*)
Makes the file associated with fileStoreID available locally. If mutable is True, then a copy of the file will be created locally so that the original is not modified and does not change the file for other jobs. If mutable is False, then a link can be created to the file, saving disk resources.

If a user path is specified, it is used as the destination. If a user path isn't specified, the file is stored in the local temp directory with an encoded name.

The destination file must not be deleted by the user; it can only be deleted through deleteLocalFile.

Parameters

- **or str fileStoreID** (toil.fileStores.FileID) – job store id for the file
- **userPath** (*string*) – a path to the name of file to which the global file will be copied or hard-linked (see below).
- **cache** (*bool*) – Described in `toil.fileStores.CachingFileStore.readGlobalFile()`
- **mutable** (*bool*) – Described in `toil.fileStores.CachingFileStore.readGlobalFile()`

Returns An absolute path to a local, temporary copy of the file keyed by fileStoreID.

Return type str

readGlobalFileStream(*fileStoreID*)
Similar to readGlobalFile, but allows a stream to be read from the job store. The yielded file handle does not need to and should not be closed explicitly.

Returns a context manager yielding a file handle which can be read from.

getGlobalFileSize(*fileStoreID*)
Get the size of the file pointed to by the given ID, in bytes.

If a FileID or something else with a non-None 'size' field, gets that.

Otherwise, asks the job store to poll the file's size.

Note that the job store may overestimate the file's size, for example if it is encrypted and had to be augmented with an IV or other encryption framing.

Parameters **or str fileStoreID** (toil.fileStores.FileID) – File ID for the file

Returns File's size in bytes, as stored in the job store

Return type int

deleteLocalFile(*fileStoreID*)
Deletes local copies of files associated with the provided job store ID.

The files deleted are all those previously read from this file ID via readGlobalFile by the current job into the job's file-store-provided temp directory, plus the file that was written to create the given file ID, if it was written by the current job from the job's file-store-provided temp directory.

Parameters **or str fileStoreID** (toil.fileStores.FileID) – File Store ID of the file to be deleted.

deleteGlobalFile(*fileStoreID*)
Deletes local files with the provided job store ID and then permanently deletes them from the job store. To ensure that the job can be restarted if necessary, the delete will not happen until after the job's run method has completed.

Parameters **or str fileStoreID** (toil.fileStores.FileID) – the File Store ID of the file to be deleted.

logToMaster(*text*, *level=20*)
Send a logging message to the leader. The message will also be logged by the worker at the same level.

Parameters

- **text** – The string to log.
- **level** (*int*) – The logging level.

startCommit(*jobState=False*)
Update the status of the job on the disk.

May start an asynchronous process. Call waitForCommit() to wait on that process.

Parameters **jobState** (*bool*) – If True, commit the state of the FileStore's job, and file deletes. Otherwise, commit only file creates/updates.

waitForCommit()
Blocks while startCommit is running. This function is called by this job's successor to ensure that it does not begin modifying the job store until after this job has finished doing so.

Might be called when startCommit is never called on a particular instance, in which case it does not block.

Returns Always returns True

Return type bool

classmethod shutdown(*dir_*)
Shutdown the filestore on this node.

This is intended to be called on batch system shutdown.

Parameters **dir** – The implementation-specific directory containing the required information for shutting down the file store and removing all its state and all job local temp directories from the node.

class toil.fileStores.**FileID**(*fileStoreID*, *size*)
A small wrapper around Python's builtin string class. It is used to represent a file's ID in the file store, and has a size attribute that is the file's size in bytes. This object is returned by importFile and writeGlobalFile.

Calls into the file store can use bare strings; size will be queried from the job store if unavailable in the ID.

__init__(*fileStoreID*, *size*)
x.__init__(...) initializes x; see help(type(x)) for signature

pack()
Pack the FileID into a string so it can be passed through external code.

classmethod unpack(*packedFileStoreID*)
Unpack the result of pack() into a FileID object.

CHAPTER 18

Batch System API

The batch system interface is used by Toil to abstract over different ways of running batches of jobs, for example Slurm, GridEngine, Mesos, Parasol and a single node. The *toil.batchSystems.abstractBatchSystem.AbstractBatchSystem* API is implemented to run jobs using a given job management system, e.g. Mesos.

18.1 Batch System Enivronmental Variables

Environmental variables allow passing of scheduler specific parameters.

For SLURM:

```
export TOIL_SLURM_ARGS="-t 1:00:00 -q fatq"
```

For TORQUE there are two environment variables - one for everything but the resource requirements, and another - for resources requirements (without the *-l* prefix):

```
export TOIL_TORQUE_ARGS="-q fatq"
export TOIL_TORQUE_REQS="walltime=1:00:00"
```

For GridEngine (SGE, UGE), there is an additional environmental variable to define the parallel environment for running multicore jobs:

```
export TOIL_GRIDENGINE_PE='smp'
export TOIL_GRIDENGINE_ARGS='-q batch.q'
```

For HTCondor, additional parameters can be included in the submit file passed to condor_submit:

```
export TOIL_HTCONDOR_PARAMS='requirements = TARGET.has_sse4_2 == true; accounting_
↪group = test'
```

The environment variable is parsed as a semicolon-separated string of `parameter = value` pairs.

18.2 Batch System API

class toil.batchSystems.abstractBatchSystem.**AbstractBatchSystem**

An abstract (as far as Python currently allows) base class to represent the interface the batch system must provide to Toil.

classmethod supportsAutoDeployment()

Whether this batch system supports auto-deployment of the user script itself. If it does, the *setUserScript()* can be invoked to set the resource object representing the user script.

Note to implementors: If your implementation returns True here, it should also override

Return type bool

classmethod supportsWorkerCleanup()

Indicates whether this batch system invokes workerCleanup() after the last job for a particular workflow invocation finishes. Note that the term *worker* refers to an entire node, not just a worker process. A worker process may run more than one job sequentially, and more than one concurrent worker process may exist on a worker node, for the same workflow. The batch system is said to *shut down* after the last worker process terminates.

Return type bool

setUserScript(*userScript*)

Set the user script for this workflow. This method must be called before the first job is issued to this batch system, and only if *supportsAutoDeployment()* returns True, otherwise it will raise an exception.

Parameters **userScript** (*toil.resource.Resource*) – the resource object representing the user script or module and the modules it depends on.

issueBatchJob(*jobNode*)

Issues a job with the specified command to the batch system and returns a unique jobID.

:param jobNode a toil.job.JobNode

Returns a unique jobID that can be used to reference the newly issued job

Return type int

killBatchJobs(*jobIDs*)

Kills the given job IDs. After returning, the killed jobs will not appear in the results of getRunningBatchJobIDs.

Parameters **jobIDs** (*list[int]*) – list of IDs of jobs to kill

getIssuedBatchJobIDs()

Gets all currently issued jobs

Returns A list of jobs (as jobIDs) currently issued (may be running, or may be waiting to be run). Despite the result being a list, the ordering should not be depended upon.

Return type list[str]

getRunningBatchJobIDs()

Gets a map of jobs as jobIDs that are currently running (not just waiting) and how long they have been running, in seconds.

Returns dictionary with currently running jobID keys and how many seconds they have been running as the value

Return type dict[int,float]

getUpdatedBatchJob(*maxWait*)

Returns information about job that has updated its status (i.e. ceased running, either successfully or with an error). Each such job will be returned exactly once.

Parameters **maxWait** (*float*) – the number of seconds to block, waiting for a result

Return type tuple(str, int, float) or None

Returns If a result is available, returns a tuple (jobID, exitValue, wallTime). Otherwise it returns None. wallTime is the number of seconds (a strictly positive float) in wall-clock time the job ran for, or None if this batch system does not support tracking wall time. Returns None for jobs that were killed.

shutdown()

Called at the completion of a toil invocation. Should cleanly terminate all worker threads.

setEnv(*name*, *value=None*)

Set an environment variable for the worker process before it is launched. The worker process will typically inherit the environment of the machine it is running on but this method makes it possible to override specific variables in that inherited environment before the worker is launched. Note that this mechanism is different to the one used by the worker internally to set up the environment of a job. A call to this method affects all jobs issued after this method returns. Note to implementors: This means that you would typically need to copy the variables before enqueuing a job.

If no value is provided it will be looked up from the current environment.

classmethod setOptions(*setOption*)

Process command line or configuration options relevant to this batch system. The

Parameters **setOption** – A function with signature setOption(varName, parsingFn=None, checkFn=None, default=None) used to update run configuration

CHAPTER 19

Job.Service API

The Service class allows databases and servers to be spawned within a Toil workflow.

class Job.**Service**(*memory=None*, *cores=None*, *disk=None*, *preemptable=None*, *unitName=None*)
Abstract class used to define the interface to a service.

__init__(*memory=None*, *cores=None*, *disk=None*, *preemptable=None*, *unitName=None*)
Memory, core and disk requirements are specified identically to as in *toil.job.Job.__init__()*.

start(*job*)
Start the service.

Parameters **job** (toil.job.Job) – The underlying job that is being run. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

Returns An object describing how to access the service. The object must be pickleable and will be used by jobs to access the service (see *toil.job.Job.addService()*).

stop(*job*)
Stops the service. Function can block until complete.

Parameters **job** (toil.job.Job) – The underlying job that is being run. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

check()
Checks the service is still running.

Raises **exceptions.RuntimeError** – If the service failed, this will cause the service job to be labeled failed.

Returns True if the service is still running, else False. If False then the service job will be terminated, and considered a success. Important point: if the service job exits due to a failure, it should raise a RuntimeError, not return False!

CHAPTER 20

Exceptions API

Toil specific exceptions.

exception toil.job.**JobException**(*message*)
General job exception.

__init__(*message*)
x.__init__(...) initializes x; see help(type(x)) for signature

exception toil.job.**JobGraphDeadlockException**(*string*)
An exception raised in the event that a workflow contains an unresolvable dependency, such as a cycle. See *toil.job.Job.checkJobGraphForDeadlocks()*.

__init__(*string*)
x.__init__(...) initializes x; see help(type(x)) for signature

exception toil.jobStores.abstractJobStore.**ConcurrentFileModificationException**(*jobStoreFileID*)
Indicates that the file was attempted to be modified by multiple processes at once.

__init__(*jobStoreFileID*)

Parameters **jobStoreFileID** (*str*) – the ID of the file that was modified by multiple workers or processes concurrently

exception toil.jobStores.abstractJobStore.**JobStoreExistsException**(*locator*)
Indicates that the specified job store already exists.

__init__(*locator*)
x.__init__(...) initializes x; see help(type(x)) for signature

exception toil.jobStores.abstractJobStore.**NoSuchFileException**(*jobStoreFileID*, *customName=None*, **extra*)
Indicates that the specified file does not exist.

__init__(*jobStoreFileID*, *customName=None*, **extra*)

Parameters

- **jobStoreFileID** (*str*) – the ID of the file that was mistakenly assumed to exist
- **customName** (*str*) – optionally, an alternate name for the nonexistent file
- **extra** (*list*) – optional extra information to add to the error message

exception toil.jobStores.abstractJobStore.**NoSuchJobException**(*jobStoreID*)
Indicates that the specified job does not exist.

__init__(*jobStoreID*)

Parameters **jobStoreID** (*str*) – the jobStoreID that was mistakenly assumed to exist

exception toil.jobStores.abstractJobStore.**NoSuchJobStoreException**(*locator*)
Indicates that the specified job store does not exist.

__init__(*locator*)
x.__init__(...) initializes x; see help(type(x)) for signature

CHAPTER 21

Running Tests

Test make targets, invoked as `$ make <target>`, subject to which environment variables are set (see *Running Integration Tests*).

TARGET	DESCRIPTION
test	Invokes all tests.
integration_test	Invokes only the integration tests.
test_offline	Skips building the Docker appliance and only invokes tests that have no docker dependencies.
integration_test_local	Makes integration tests easier to debug locally by running the integration tests serially and doesn't redirect output. This makes it appears on the terminal as expected.

Before running tests for the first time, initialize your virtual environment following the steps in *Building from Source*.

Run all tests (including slow tests):

```
$ make test
```

Run only quick tests (as of Jul 25, 2018, this was ~ 20 minutes):

```
$ export TOIL_TEST_QUICK=True; make test
```

Run an individual test with:

```
$ make test tests=src/toil/test/sort/sortTest.py::SortTest::testSort
```

The default value for `tests` is `"src"` which includes all tests in the `src/` subdirectory of the project root. Tests that require a particular feature will be skipped implicitly. If you want to explicitly skip tests that depend on a currently installed *feature*, use

```
$ make test tests="-m 'not aws' src"
```

This will run only the tests that don't depend on the `aws` extra, even if that extra is currently installed. Note the distinction between the terms *feature* and *extra*. Every extra is a feature but there are features that are not extras, such

as the `gridengine` and `parasol` features. To skip tests involving both the `parasol` feature and the `aws` extra, use the following:

```
$ make test tests="-m 'not aws and not parasol' src"
```

21.1 Running Tests with pytest

Often it is simpler to use pytest directly, instead of calling the `make` wrapper. This usually works as expected, but some tests need some manual preparation. To run a specific test with pytest, use the following:

```
python -m pytest src/toil/test/sort/sortTest.py::SortTest::testSort
```

For more information, see the pytest documentation.

21.2 Running Integration Tests

These tests are generally only run using in our CI workflow due to their resource requirements and cost. However, they can be made available for local testing:

- Running tests that make use of Docker (e.g. autoscaling tests and Docker tests) require an appliance image to be hosted. First, make sure you have gone through the set up found in *Using Docker with Quay*. Then to build and host the appliance image run the `make` target `push_docker`.

  ```
  $ make push_docker
  ```

- Running integration tests require activation via an environment variable as well as exporting information relevant to the desired tests. Enable the integration tests:

  ```
  $ export TOIL_TEST_INTEGRATIVE=True
  ```

- Finally, set the environment variables for keyname and desired zone:

  ```
  $ export TOIL_X_KEYNAME=[Your Keyname]
  $ export TOIL_X_ZONE=[Desired Zone]
  ```

  Where `X` is one of our currently supported cloud providers (`GCE`, `AWS`).

- See the above sections for guidance on running tests.

21.3 Test Environment Variables

TOIL_TEST_TEMP	An absolute path to a directory where Toil tests will write their temporary files. Defaults to the system's standard temporary directory.
TOIL_TEST_INTEGRATIVE	If True, this allows the integration tests to run. Only valid when running the tests from the source directory via `make test` or `make test_parallel`.
TOIL_AWS_KEYNAME	An AWS keyname (see *Preparing your AWS environment*), which is required to run the AWS tests.
TOIL_SSH_KEYNAME	The SSH key to use for tests.
TOIL_GOOGLE_PROJECTID	A Google Cloud account projectID (see *Running in Google Compute Engine (GCE)*), which is required to to run the Google Cloud tests.
TOIL_TEST_QUICK	If True, long running tests are skipped.

Partial install and failing tests

Some tests may fail with an ImportError if the required extras are not installed. Install Toil with all of the extras do prevent such errors.

21.4 Using Docker with Quay

Docker is needed for some of the tests. Follow the appropriate installation instructions for your system on their website to get started.

When running `make test` you might still get the following error:

```
$ make test
Please set TOIL_DOCKER_REGISTRY, e.g. to quay.io/USER.
```

To solve, make an account with Quay and specify it like so:

```
$ TOIL_DOCKER_REGISTRY=quay.io/USER make test
```

where `USER` is your Quay username.

For convenience you may want to add this variable to your bashrc by running

```
$ echo 'export TOIL_DOCKER_REGISTRY=quay.io/USER' >> $HOME/.bashrc
```

21.5 Running Mesos Tests

If you're running Toil's Mesos tests, be sure to create the virtualenv with `--system-site-packages` to include the Mesos Python bindings. Verify this by activating the virtualenv and running `pip list | grep mesos`. On macOS, this may come up empty. To fix it, run the following:

```
for i in /usr/local/lib/python2.7/site-packages/*mesos*; do ln -snf $i venv/lib/
↪python2.7/site-packages/; done
```

CHAPTER 22

Developing with Docker

To develop on features reliant on the Toil Appliance (the docker image toil uses for AWS autoscaling), you should consider setting up a personal registry on Quay or Docker Hub. Because the Toil Appliance images are tagged with the Git commit they are based on and because only commits on our master branch trigger an appliance build on Quay, as soon as a developer makes a commit or dirties the working copy they will no longer be able to rely on Toil to automatically detect the proper Toil Appliance image. Instead, developers wishing to test any appliance changes in autoscaling should build and push their own appliance image to a personal Docker registry. This is described in the next section.

22.1 Making Your Own Toil Docker Image

Note! Toil checks if the docker image specified by TOIL_APPLIANCE_SELF exists prior to launching by using the docker v2 schema. This should be valid for any major docker repository, but there is an option to override this if desired using the option: *–forceDockerAppliance*.

Here is a general workflow (similar instructions apply when using Docker Hub):

1. Make some changes to the provisioner of your local version of Toil
2. Go to the location where you installed the Toil source code and run

   ```
   $ make docker
   ```

   to automatically build a docker image that can now be uploaded to your personal Quay account. If you have not installed Toil source code yet see *Building from Source*.
3. If it's not already you will need Docker installed and need to log into Quay. Also you will want to make sure that your Quay account is public.
4. Set the environment variable `TOIL_DOCKER_REGISTRY` to your Quay account. If you find yourself doing this often you may want to add

   ```
   export TOIL_DOCKER_REGISTRY=quay.io/<MY_QUAY_USERNAME>
   ```

   to your `.bashrc` or equivalent.

5. Now you can run

```
$ make push_docker
```

which will upload the docker image to your Quay account. Take note of the image's tag for the next step.

6. Finally you will need to tell Toil from where to pull the Appliance image you've created (it uses the Toil release you have installed by default). To do this set the environment variable `TOIL_APPLIANCE_SELF` to the url of your image. For more info see *Environment Variables*.

7. Now you can launch your cluster! For more information see *Running a Workflow with Autoscaling*.

22.2 Running a Cluster Locally

The Toil Appliance container can also be useful as a test environment since it can simulate a Toil cluster locally. An important caveat for this is autoscaling, since autoscaling will only work on an EC2 instance and cannot (at this time) be run on a local machine.

To spin up a local cluster, start by using the following Docker run command to launch a Toil leader container:

```
docker run --entrypoint=mesos-master --net=host -d --name=leader --volume=/home/
↪jobStoreParentDir:/jobStoreParentDir quay.io/ucsc_cgl/toil:3.6.0 --registry=in_
↪memory --ip=127.0.0.1 --port=5050 --allocation_interval=500ms
```

A couple notes on this command: the `-d` flag tells Docker to run in daemon mode so the container will run in the background. To verify that the container is running you can run `docker ps` to see all containers. If you want to run your own container rather than the official UCSC container you can simply replace the `quay.io/ucsc_cgl/toil:3.6.0` parameter with your own container name.

Also note that we are not mounting the job store directory itself, but rather the location where the job store will be written. Due to complications with running Docker on MacOS, I recommend only mounting directories within your home directory. The next command will launch the Toil worker container with similar parameters:

```
docker run --entrypoint=mesos-slave --net=host -d --name=worker --volume=/home/
↪jobStoreParentDir:/jobStoreParentDir quay.io/ucsc_cgl/toil:3.6.0 --work_dir=/var/
↪lib/mesos --master=127.0.0.1:5050 --ip=127.0.0.1 ---attributes=preemptable:False --
↪resources=cpus:2
```

Note here that we are specifying 2 CPUs and a non-preemptable worker. We can easily change either or both of these in a logical way. To change the number of cores we can change the 2 to whatever number you like, and to change the worker to be preemptable we change `preemptable:False` to `preemptable:True`. Also note that the same volume is mounted into the worker. This is needed since both the leader and worker write and read from the job store. Now that your cluster is running, you can run

```
docker exec -it leader bash
```

to get a shell in your leader 'node'. You can also replace the `leader` parameter with `worker` to get shell access in your worker.

Docker-in-Docker issues

If you want to run Docker inside this Docker cluster (Dockerized tools, perhaps), you should also mount in the Docker socket via `-v /var/run/docker.sock:/var/run/docker.sock`. This will give the Docker client inside the Toil Appliance access to the Docker engine on the host. Client/engine version mismatches have been known to cause issues, so we recommend using Docker version 1.12.3 on the host to be compatible with the Docker client installed in the Appliance. Finally, be careful where you write files inside the Toil Appliance - 'child' Docker

containers launched in the Appliance will actually be siblings to the Appliance since the Docker engine is located on the host. This means that the 'child' container can only mount in files from the Appliance if the files are located in a directory that was originally mounted into the Appliance from the host - that way the files are accessible to the sibling container. Note: if Docker can't find the file/directory on the host it will silently fail and mount in an empty directory.

CHAPTER 23

Maintainer's Guidelines

In general, as developers and maintainers of the code, we adhere to the following guidelines:

- We strive to never break the build on master. All development should be done on branches, in either the main Toil repository or in developers' forks.
- Pull requests should be used for any and all changes (except truly trivial ones).
- Pull requests should be in response to issues. If you find yourself making a pull request without an issue, you should create the issue first.

23.1 Naming Conventions

- **Commit messages** *should* be great. Most importantly, they *must*:
  - Have a short subject line. If in need of more space, drop down **two** lines and write a body to explain what is changing and why it has to change.
  - Write the subject line as a command: *Destroy all humans*, not *All humans destroyed*.
  - Reference the issue being fixed in a Github-parseable format, such as *(resolves #1234)* at the end of the subject line, or *This will fix #1234.* somewhere in the body. If no single commit on its own fixes the issue, the cross-reference must appear in the pull request title or body instead.
- **Branches** in the main Toil repository *must* start with `issues/`, followed by the issue number (or numbers, separated by a dash), followed by a short, lowercase, hyphenated description of the change. (There can be many open pull requests with their associated branches at any given point in time and this convention ensures that we can easily identify branches.)

  Say there is an issue numbered #123 titled *Foo does not work*. The branch name would be `issues/123-fix-foo` and the title of the commit would be *Fix foo in case of bar (resolves #123).*

23.2 Pull Requests

- All pull requests must be reviewed by a person other than the request's author.
- Modified pull requests must be re-reviewed before merging. **Note that Github does not enforce this!**
- Pull requests will not be merged unless Travis and Gitlab CI tests pass. Gitlab tests are only run on code in the main Toil repository on some branch, so it is the responsibility of the approving reviewer to make sure that pull requests from outside repositories are copied to branches in the main repository. This can be accomplished with (from a Toil clone):

  ```
  ./contrib/admin/test-pr theirusername their-branch issues/123-fix-description-here
  ```

  This must be repeated every time the PR submitter updates their PR, after checking to see that the update is not malicious.

  If there is no issue corresponding to the PR, after which the branch can be named, the reviewer of the PR should first create the issue.

  Developers who have push access to the main Toil repository are encouraged to make their pull requests from within the repository, to avoid this step.

- Prefer using "Squash and marge" when merging pull requests to master especially when the PR contains a "single unit" of work (i.e. if one were to rewrite the PR from scratch with all the fixes included, they would have one commit for the entire PR). This makes the commit history on master more readable and easier to debug in case of a breakage.

  When squashing a PR from multiple authors, please add Co-authored-by to give credit to all contributing authors.

  See Issue #2816 for more details.

CHAPTER 24

Toil Architecture

The following diagram layouts out the software architecture of Toil.

These components are described below:

- **the leader:** The leader is responsible for deciding which jobs should be run. To do this it traverses the job graph. Currently this is a single threaded process, but we make aggressive steps to prevent it becoming a bottleneck (see *Read-only Leader* described below).
- **the job-store:** Handles all files shared between the components. Files in the job-store are the means by which the state of the workflow is maintained. Each job is backed by a file in the job store, and atomic updates to this state are used to ensure the workflow can always be resumed upon failure. The job-store can also store all user files, allowing them to be shared between jobs. The job-store is defined by the `AbstractJobStore` class. Multiple implementations of this class allow Toil to support different back-end file stores, e.g.: S3, network file systems, Google file store, etc.
- **workers:** The workers are temporary processes responsible for running jobs, one at a time per worker. Each worker process is invoked with a job argument that it is responsible for running. The worker monitors this job and reports back success or failure to the leader by editing the job's state in the file-store. If the job defines successor jobs the worker may choose to immediately run them (see *Job Chaining* below).
- **the batch-system:** Responsible for scheduling the jobs given to it by the leader, creating a worker command for each job. The batch-system is defined by the `AbstractBatchSystem` class. Toil uses multiple existing batch systems to schedule jobs, including Apache Mesos, GridEngine and a multi-process single node implementation that allows workflows to be run without any of these frameworks. Toil can therefore fairly easily be made to run a workflow using an existing cluster.
- **the node provisioner:** Creates worker nodes in which the batch system schedules workers. It is defined by the `AbstractProvisioner` class.
- **the statistics and logging monitor:** Monitors logging and statistics produced by the workers and reports them. Uses the job-store to gather this information.

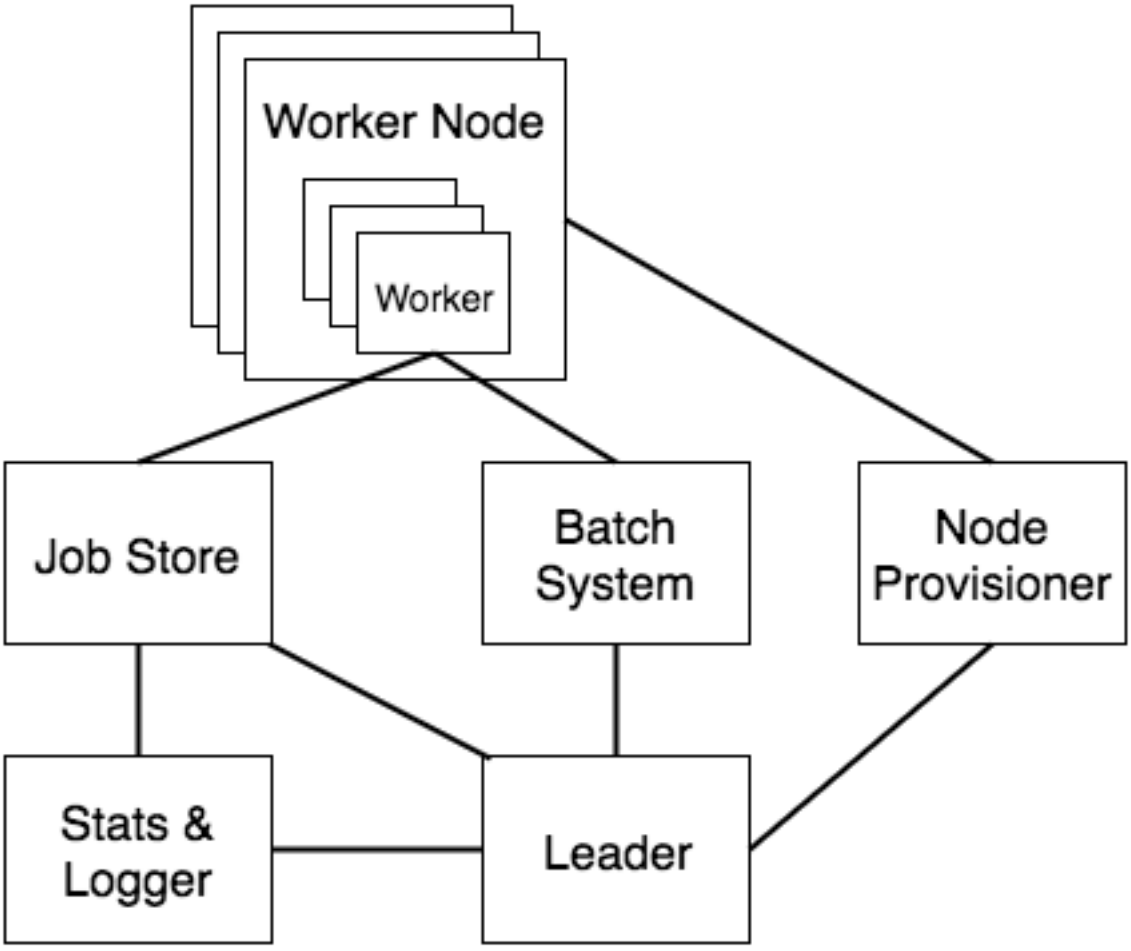


Fig. 1: Figure 1: The basic components of Toil's architecture.

24.1 Optimizations

Toil implements lots of optimizations designed for scalability. Here we detail some of the key optimizations.

24.1.1 Read-only leader

The leader process is currently implemented as a single thread. Most of the leader's tasks revolve around processing the state of jobs, each stored as a file within the job-store. To minimise the load on this thread, each worker does as much work as possible to manage the state of the job it is running. As a result, with a couple of minor exceptions, the leader process never needs to write or update the state of a job within the job-store. For example, when a job is complete and has no further successors the responsible worker deletes the job from the job-store, marking it complete. The leader then only has to check for the existence of the file when it receives a signal from the batch-system to know that the job is complete. This off-loading of state management is orthogonal to future parallelization of the leader.

24.1.2 Job chaining

The scheduling of successor jobs is partially managed by the worker, reducing the number of individual jobs the leader needs to process. Currently this is very simple: if the there is a single next successor job to run and its resources fit within the resources of the current job and closely match the resources of the current job then the job is run immediately on the worker without returning to the leader. Further extensions of this strategy are possible, but for many workflows which define a series of serial successors (e.g. map sequencing reads, post-process mapped reads, etc.) this pattern is very effective at reducing leader workload.

24.1.3 Preemptable node support

Critical to running at large-scale is dealing with intermittent node failures. Toil is therefore designed to always be resumable providing the job-store does not become corrupt. This robustness allows Toil to run on preemptible nodes, which are only available when others are not willing to pay more to use them. Designing workflows that divide into many short individual jobs that can use preemptable nodes allows for workflows to be efficiently scheduled and executed.

24.1.4 Caching

Running bioinformatic pipelines often require the passing of large datasets between jobs. Toil caches the results from jobs such that child jobs running on the same node can directly use the same file objects, thereby eliminating the need for an intermediary transfer to the job store. Caching also reduces the burden on the local disks, because multiple jobs can share a single file. The resulting drop in I/O allows pipelines to run faster, and, by the sharing of files, allows users to run more jobs in parallel by reducing overall disk requirements.

To demonstrate the efficiency of caching, we ran an experimental internal pipeline on 3 samples from the TCGA Lung Squamous Carcinoma (LUSC) dataset. The pipeline takes the tumor and normal exome fastqs, and the tumor rna fastq and input, and predicts MHC presented neoepitopes in the patient that are potential targets for T-cell based immunotherapies. The pipeline was run individually on the samples on c3.8xlarge machines on AWS (60GB RAM,600GB SSD storage, 32 cores). The pipeline aligns the data to hg19-based references, predicts MHC haplotypes using PHLAT, calls mutations using 2 callers (MuTect and RADIA) and annotates them using SnpEff, then predicts MHC:peptide binding using the IEDB suite of tools before running an in-house rank boosting algorithm on the final calls.

To optimize time taken, The pipeline is written such that mutations are called on a per-chromosome basis from the whole-exome bams and are merged into a complete vcf. Running mutect in parallel on whole exome bams requires each mutect job to download the complete Tumor and Normal Bams to their working directories – An operation that

quickly fills the disk and limits the parallelizability of jobs. The script was run in Toil, with and without caching, and Figure 2 shows that the workflow finishes faster in the cached case while using less disk on average than the uncached run. We believe that benefits of caching arising from file transfers will be much higher on magnetic disk-based storage systems as compared to the SSD systems we tested this on.

24.2 Toil support for Common Workflow Language

The CWL document and input document are loaded using the 'cwltool.load_tool' module. This performs normalization and URI expansion (for example, relative file references are turned into absolute file URIs), validates the document against the CWL schema, initializes Python objects corresponding to major document elements (command line tools, workflows, workflow steps), and performs static type checking that sources and sinks have compatible types.

Input files referenced by the CWL document and input document are imported into the Toil file store. CWL documents may use any URI scheme supported by Toil file store, including local files and object storage.

The 'location' field of File references are updated to reflect the import token returned by the Toil file store.

For directory inputs, the directory listing is stored in Directory object. Each individual files is imported into Toil file store.

An initial workflow Job is created from the toplevel CWL document. Then, control passes to the Toil engine which schedules the initial workflow job to run.

When the toplevel workflow job runs, it traverses the CWL workflow and creates a toil job for each step. The dependency graph is expressed by making downstream jobs children of upstream jobs, and initializing the child jobs with an input object containing the promises of output from upstream jobs.

Because Toil jobs have a single output, but CWL permits steps to have multiple output parameters that may feed into multiple other steps, the input to a CWLJob is expressed with an "indirect dictionary". This is a dictionary of input parameters, where each entry value is a tuple of a promise and a promise key. When the job runs, the indirect dictionary is turned into a concrete input object by resolving each promise into its actual value (which is always a dict), and then looking up the promise key to get the actual value for the the input parameter.

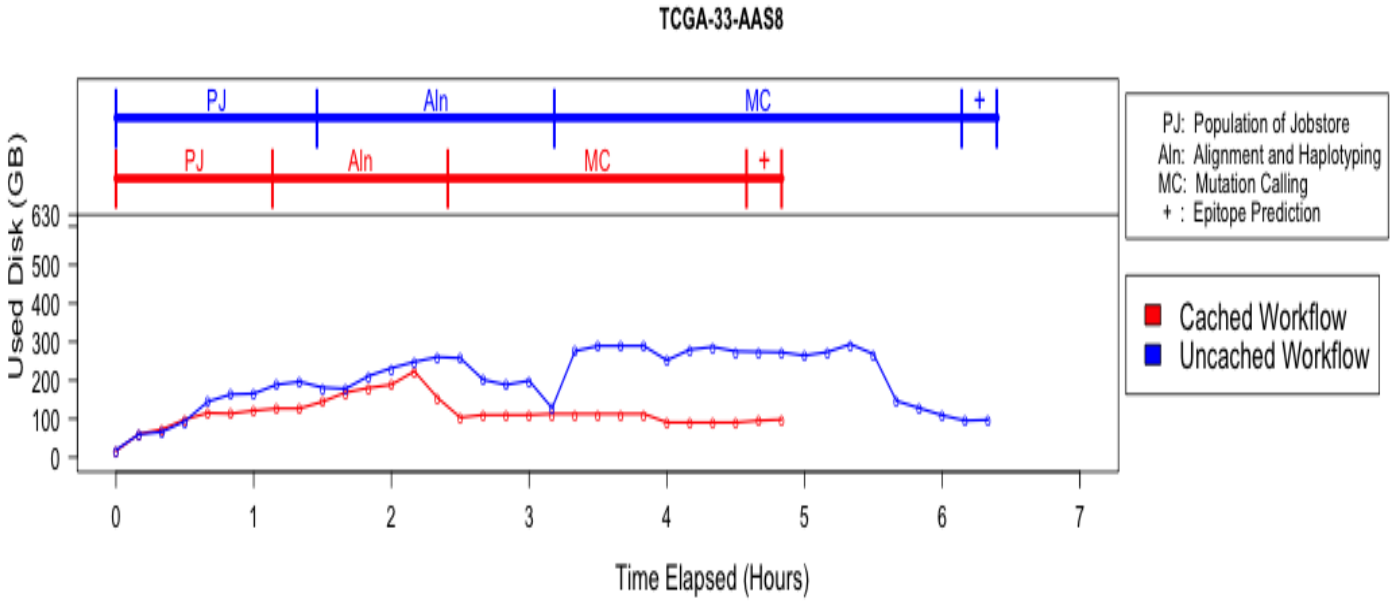
If a workflow step specifies a scatter, then a scatter job is created and connected into the workflow graph as described above. When the scatter step runs, it creates child jobs for each parameterizations of the scatter. A gather job is added as a follow-on to gather the outputs into arrays.

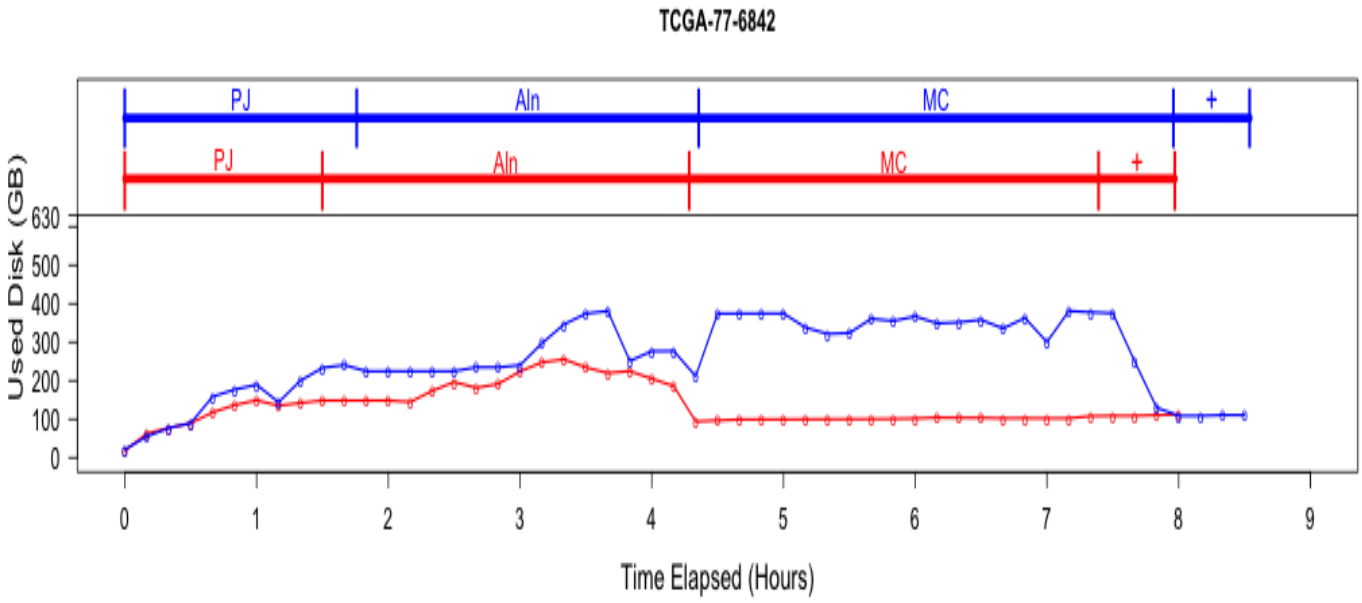
When running a command line tool, it first creates output and temporary directories under the Toil local temp dir. It runs the command line tool using the single_job_executor from CWLTool, providing a Toil-specific constructor for filesystem access, and overriding the default PathMapper to use ToilPathMapper.

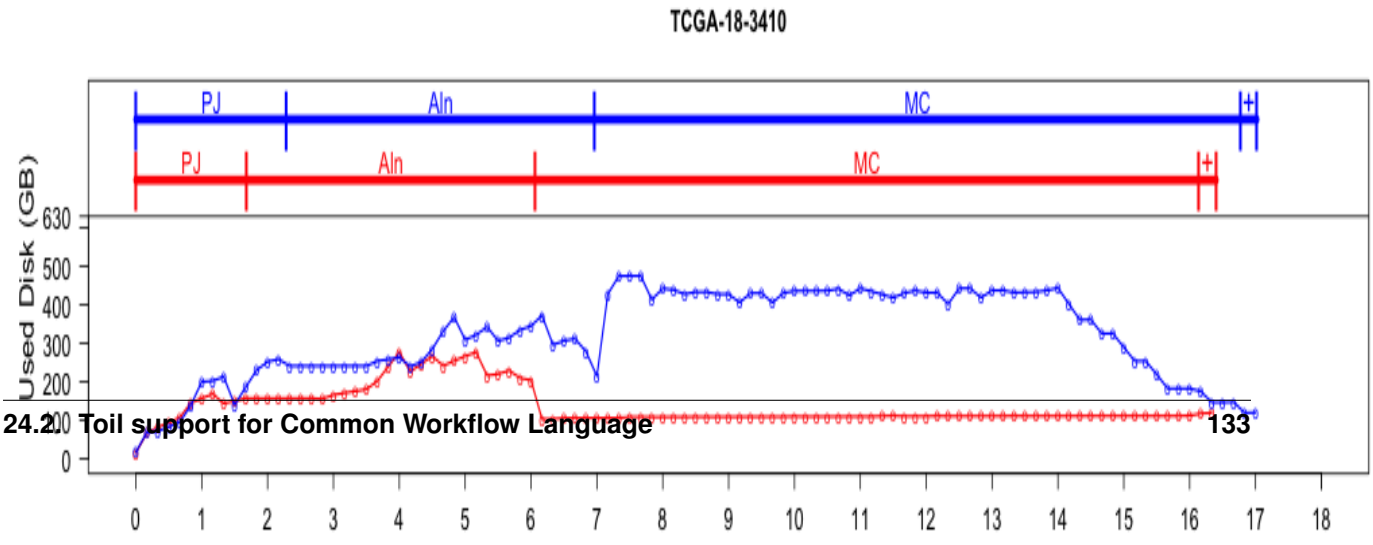
The ToilPathMapper keeps track of a file's symbolic identifier (the Toil FileID), its local path on the host (the value returned by readGlobalFile) and the the location of the file inside the Docker container.

After executing single_job_executor from CWLTool, it gets back the output object and status. If the underlying job failed, raise an exception. Files from the output object are added to the file store using writeGlobalFile and the 'location' field of File references are updated to reflect the token returned by the Toil file store.

When the workflow completes, it returns an indirect dictionary linking to the outputs of the job steps that contribute to the final output. This is the value returned by toil.start() or toil.restart(). This is resolved to get the final output object. The files in this object are exported from the file store to 'outdir' on the host file system, and the 'location' field of File references are updated to reflect the final exported location of the output files.


TCGA-33-AAS8
PJ
Aln
MC
+
PJ
Aln
MC
+
Used Disk (GB)
630
500
400
300
200
100
0
0
1
2
3
4
5
6
7
Time Elapsed (Hours)
PJ: Population of Jobstore
Aln: Alignment and Haplotyping
MC: Mutation Calling
+ : Epitope Prediction
Cached Workflow
Uncached Workflow


TCGA-77-6842
PJ
Aln
MC
+
PJ
Aln
MC
+
Used Disk (GB)
630
500
400
300
200
100
0
0
1
2
3
4
5
6
7
8
9
Time Elapsed (Hours)


TCGA-18-3410
PJ
Aln
MC
+
PJ
Aln
MC
+
Used Disk (GB)
630
500
400
300
200
100
0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

CHAPTER 25

Minimum AWS IAM permissions

Toil requires at least the following permissions in an IAM role to operate on a cluster. These are added by default when launching a cluster. However, ensure that they are present if creating a custom IAM role when *launching a cluster* with the `--awsEc2ProfileArn` parameter.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:*",
                "s3:*",
                "sdb:*",
                "iam:PassRole"
            ],
            "Resource": "*"
        }
    ]
}
```

CHAPTER 26

Auto-Deployment

If you want to run your workflow in a distributed environment, on multiple worker machines, either in the cloud or on a bare-metal cluster, your script needs to be made available to those other machines. If your script imports other modules, those modules also need to be made available on the workers. Toil can automatically do that for you, with a little help on your part. We call this feature *auto-deployment* of a workflow.

Let's first examine various scenarios of auto-deploying a workflow, which, as we'll see shortly cannot be auto-deployed. Lastly, we'll deal with the issue of declaring *Toil as a dependency* of a workflow that is packaged as a setuptools distribution.

Toil can be easily deployed to a remote host. First, assuming you've followed our *Preparing your AWS environment* section to install Toil and use it to create a remote leader node on (in this example) AWS, you can now log into this into using *Ssh-Cluster Command* and once on the remote host, create and activate a virtualenv (noting to make sure to use the `--system-site-packages` option!):

```
$ virtualenv --system-site-packages venv
$ . venv/bin/activate
```

Note the `--system-site-packages` option, which ensures that globally-installed packages are accessible inside the virtualenv. Do not (re)install Toil after this! The `--system-site-packages` option has already transferred Toil and the dependencies from your local installation of Toil for you.

From here, you can install a project and its dependencies:

```
$ tree
.
├── util
│   ├── __init__.py
│   └── sort
│       ├── __init__.py
│       └── quick.py
└── workflow
    ├── __init__.py
    └── main.py
```

(continues on next page)

(continued from previous page)

```
3 directories, 5 files
$ pip install matplotlib
$ cp -R workflow util venv/lib/python2.7/site-packages
```

Ideally, your project would have a `setup.py` file (see setuptools) which streamlines the installation process:

```
$ tree
.
├── util
│   ├── __init__.py
│   └── sort
│       ├── __init__.py
│       └── quick.py
├── workflow
│   ├── __init__.py
│   └── main.py
└── setup.py

3 directories, 6 files
$ pip install .
```

Or, if your project has been published to PyPI:

```
$ pip install my-project
```

In each case, we have created a virtualenv with the `--system-site-packages` flag in the `venv` subdirectory then installed the `matplotlib` distribution from PyPI along with the two packages that our project consists of. (Again, both Python and Toil are assumed to be present on the leader and all worker nodes.)

We can now run our workflow:

```
$ python main.py --batchSystem=mesos ...
```

Important: If workflow's external dependencies contain native code (i.e. are not pure Python) then they must be manually installed on each worker.

Warning: Neither `python setup.py develop` nor `pip install -e .` can be used in this process as, instead of copying the source files, they create `.egg-link` files that Toil can't auto-deploy. Similarly, `python setup.py install` doesn't work either as it installs the project as a Python `.egg` which is also not currently supported by Toil (though it could be in the future).

Also note that using the `--single-version-externally-managed` flag with `setup.py` will prevent the installation of your package as an `.egg`. It will also disable the automatic installation of your project's dependencies.

26.1 Auto Deployment with Sibling Modules

This scenario applies if the user script imports modules that are its siblings:

```
$ cd my_project
$ ls
userScript.py utilities.py
$ ./userScript.py --batchSystem=mesos ...
```

Here `userScript.py` imports additional functionality from `utilities.py`. Toil detects that `userScript.py` has sibling modules and copies them to the workers, alongside the user script. Note that sibling modules will be auto-deployed regardless of whether they are actually imported by the user script–all .py files residing in the same directory as the user script will automatically be auto-deployed.

Sibling modules are a suitable method of organizing the source code of reasonably complicated workflows.

26.2 Auto-Deploying a Package Hierarchy

Recall that in Python, a package is a directory containing one or more `.py` files—one of which must be called `__init__.py`—and optionally other packages. For more involved workflows that contain a significant amount of code, this is the recommended way of organizing the source code. Because we use a package hierarchy, we can't really refer to the user script as such, we call it the user *module* instead. It is merely one of the modules in the package hierarchy. We need to inform Toil that we want to use a package hierarchy by invoking Python's `-m` option. That enables Toil to identify the entire set of modules belonging to the workflow and copy all of them to each worker. Note that while using the `-m` option is optional in the scenarios above, it is mandatory in this one.

The following shell session illustrates this:

```
$ cd my_project
$ tree
.
├── utils
│   ├── __init__.py
│   └── sort
│       ├── __init__.py
│       └── quick.py
└── workflow
    ├── __init__.py
    └── main.py

3 directories, 5 files
$ python -m workflow.main --batchSystem=mesos ...
```

Here the user module `main.py` does not reside in the current directory, but is part of a package called `util`, in a subdirectory of the current directory. Additional functionality is in a separate module called `util.sort.quick` which corresponds to `util/sort/quick.py`. Because we invoke the user module via `python -m workflow.main`, Toil can determine the root directory of the hierarchy–`my_project` in this case–and copy all Python modules underneath it to each worker. The `-m` option is documented here

When `-m` is passed, Python adds the current working directory to `sys.path`, the list of root directories to be considered when resolving a module name like `workflow.main`. Without that added convenience we'd have to run the workflow as `PYTHONPATH="$PWD" python -m workflow.main`. This also means that Toil can detect the root directory of the user module's package hierarchy even if it isn't the current working directory. In other words we could do this:

```
$ cd my_project
$ export PYTHONPATH="$PWD"
$ cd /some/other/dir
$ python -m workflow.main --batchSystem=mesos ...
```

Also note that the root directory itself must not be package, i.e. must not contain an `__init__.py`.

26.3 Relying on Shared Filesystems

Bare-metal clusters typically mount a shared file system like NFS on each node. If every node has that file system mounted at the same path, you can place your project on that shared filesystem and run your user script from there. Additionally, you can clone the Toil source tree into a directory on that shared file system and you won't even need to install Toil on every worker. Be sure to add both your project directory and the Toil clone to `PYTHONPATH`. Toil replicates `PYTHONPATH` from the leader to every worker.

Using a shared filesystem

Toil currently only supports a `tempdir` set to a local, non-shared directory.

26.3.1 Toil Appliance

The term Toil Appliance refers to the Mesos Docker image that Toil uses to simulate the machines in the virtual mesos cluster. It's easily deployed, only needs Docker, and allows for workflows to be run in single-machine mode and for clusters of VMs to be provisioned. To specify a different image, see the Toil *Environment Variables* section. For more information on the Toil Appliance, see the *Running in AWS* section.

CHAPTER 27

Environment Variables

There are several environment variables that affect the way Toil runs.

TOIL_CHECK_ENV	A flag that determines whether Toil will try to refer back to a Python virtual environment in which it is installed when composing commands that may be run on other hosts. If set to `True`, if Toil is installed in the current virtual environment, it will use absolute paths to its own executables (and the virtual environment must thus be available on at the same path on all nodes). Otherwise, Toil internal commands such as `_toil_worker` will be resolved according to the `PATH` on the node where they are executed. This setting can be useful in a shared HPC environment, where users may have their own Toil installations in virtual environments.
TOIL_WORKDIR	An absolute path to a directory where Toil will write its temporary files. This directory must exist on each worker node and may be set to a different value on each worker. The `--workDir` command line option overrides this. On Mesos nodes, `TOIL_WORKDIR` generally defaults to the Mesos sandbox, except on CGCloud-provisioned nodes where it defaults to `/var/lib/mesos`. In all other cases, the system's standard temporary directory is used.
TOIL_KUBERNETES_OWNER	A name prefix for easy identification of Kubernetes jobs. If not set, Toil will use the current user name.
TOIL_APPLIANCE_SELF	The fully qualified reference for the Toil Appliance you wish to use, in the form `REPO/IMAGE:TAG`. `quay.io/ucsc_cgl/toil:3.6.0` and `cket/toil:3.5.0` are both examples of valid options. Note that since Docker defaults to Dockerhub repos, only quay.io repos need to specify their registry.
TOIL_DOCKER_REGISTRY	The registry of the Toil Appliance image you wish to use. Docker will use Dockerhub by default, but the quay.io registry is also very popular and easily specifiable by setting this option to `quay.io`.
TOIL_DOCKER_NAME	The name of the Toil Appliance image you wish to use. Generally this is simply `toil` but this option is provided to override this, since the image can be built with arbitrary names.
TOIL_AWS_SECRET_NAME	For the Kubernetes batch system, the name of a Kubernetes secret which contains a `credentials` file granting access to AWS resources. Will be mounted as `~/.aws` inside Kubernetes-managed Toil containers. Enables the AWSJobStore to be used with the Kubernetes batch system, if the credentials allow access to S3 and SimpleDB.
TOIL_AWS_ZONE	The EC2 zone to provision nodes in if using Toil's provisioner.
TOIL_AWS_AMI	ID of the AMI to use in node provisioning. If in doubt, don't set this variable.
TOIL_AWS_NODE_DEBUG	Determines whether to preserve nodes that have failed health checks. If set to `True`, nodes that fail EC2 health checks won't immediately be terminated so they can be examined and the cause of failure determined. If any EC2 nodes are left behind in this manner, the security group will also be left behind by necessity as it cannot be deleted until all associated nodes have been terminated.
TOIL_SLURM_ARGS	Arguments for sbatch for the slurm batch system. Do not pass CPU or memory specifications here. Instead, define resource requirements for the job. There is no default value for this variable.
TOIL_GRIDENGINE_ARGS	Arguments for qsub for the gridengine batch system. Do not pass CPU or memory specifications here. Instead, define resource requirements for the job. There is no default value for this variable.
TOIL_GRIDENGINE_PE	Parallel environment arguments for qsub and for the gridengine batch system. There is no default value for this variable.
TOIL_TORQUE_ARGS	Arguments for qsub for the Torque batch system. Do not pass CPU or memory specifications here. Instead, define extra parameters for the job such as queue. Example: -q medium Use TOIL_TORQUE_REQS to pass extra values for the -l resource requirements parameter. There is no default value for this variable.
TOIL_TORQUE_REQS	Arguments for the resource requirements for Torque batch system. Do not pass CPU or memory specifications here. Instead, define extra resource requirements as a string that goes after the -l argument to qsub. Example: walltime=2:00:00,file=50gb There is no default value for this variable.
TOIL_LSF_ARGS	Additional arguments for the LSF's bsub command. Instead, define extra parameters for the job such as queue. Example: -q medium. There is no default value for this variable.
TOIL_HTCONDOR_PARAMS	Additional parameters to include in the HTCondor submit file passed to condor_submit. Do not pass CPU or memory specifications here. Instead define extra parameters which may be required by HTCondor. This variable is parsed as a semicolon-separated string of `parameter = value` pairs. Example: `requirements = TARGET.has_sse4_2 == true; accounting_group = test`. There is no default value for this variable.
TOIL_CUSTOM_DOCKER_INIT_COMMAND	Any custom bash command to run in the Toil docker container prior to running the Toil services. Can be used for any custom initialization in the worker and/or primary nodes such as private docker docker authentication. Example for AWS ECR: `pip install awscli && eval $(aws ecr get-login --no-include-email --region us-east-1)`.

- genindex
- search

Index

Symbols

__init__() (toil.common.Toil method), 83
__init__() (toil.fileStores.FileID method), 110
__init__() (toil.fileStores.abstractFileStore.AbstractFileStore method), 107
__init__() (toil.job.EncapsulatedJob method), 94
__init__() (toil.job.FunctionWrappingJob method), 93
__init__() (toil.job.Job method), 99
__init__() (toil.job.Job.Service method), 115
__init__() (toil.job.JobException method), 117
__init__() (toil.job.JobGraphDeadlockException method), 117
__init__() (toil.job.Promise method), 96
__init__() (toil.job.PromisedRequirement method), 96
__init__() (toil.jobStores.abstractJobStore.AbstractJobStore method), 85
__init__() (toil.jobStores.abstractJobStore.ConcurrentFileModificationException method), 117
__init__() (toil.jobStores.abstractJobStore.JobStoreExistsException method), 117
__init__() (toil.jobStores.abstractJobStore.NoSuchFileException method), 117
__init__() (toil.jobStores.abstractJobStore.NoSuchJobException method), 118
__init__() (toil.jobStores.abstractJobStore.NoSuchJobStoreException method), 118

A

AbstractBatchSystem (class in toil.batchSystems.abstractBatchSystem), 112
AbstractFileStore (class in toil.fileStores.abstractFileStore), 107
AbstractJobStore (class in toil.jobStores.abstractJobStore), 85
addChild() (toil.job.EncapsulatedJob method), 94
addChild() (toil.job.Job method), 100
addChildFn() (toil.job.Job method), 100
addChildJobFn() (toil.job.Job method), 101
addFollowOn() (toil.job.EncapsulatedJob method), 95
addFollowOn() (toil.job.Job method), 100
addFollowOnFn() (toil.job.Job method), 101
addFollowOnJobFn() (toil.job.Job method), 101
addService() (toil.job.EncapsulatedJob method), 95
addService() (toil.job.Job method), 100
addToilOptions() (toil.job.Job.Runner static method), 105

B

batch() (toil.jobStores.abstractJobStore.AbstractJobStore method), 87

C

check() (toil.job.Job.Service method), 115
checkJobGraphAcylic() (toil.job.Job method), 103
checkJobGraphConnected() (toil.job.Job method), 102
checkJobGraphForDeadlocks() (toil.job.Job method), 102
checkNewCheckpointsAreLeafVertices() (toil.job.Job method), 103
clean() (toil.jobStores.abstractJobStore.AbstractJobStore method), 87
ConcurrentFileModificationException, 117
config (toil.common.Toil attribute), 83
config (toil.jobStores.abstractJobStore.AbstractJobStore attribute), 85
convertPromises() (toil.job.PromisedRequirement static method), 96
create() (toil.jobStores.abstractJobStore.AbstractJobStore method), 87
createBatchSystem() (toil.common.Toil static method), 83
createRootJob() (toil.jobStores.abstractJobStore.AbstractJobStore method), 86

D

defer() (toil.job.Job method), 103
delete() (toil.jobStores.abstractJobStore.AbstractJobStore method), 88
deleteFile() (toil.jobStores.abstractJobStore.AbstractJobStore method), 90

deleteGlobalFile() (toil.fileStores.abstractFileStore.AbstractFileStore method), 110
deleteLocalFile() (toil.fileStores.abstractFileStore.AbstractFileStore method), 109
destroy() (toil.jobStores.abstractJobStore.AbstractJobStore method), 87

E

encapsulate() (toil.job.Job method), 102
EncapsulatedJob (class in toil.job), 94
exists() (toil.jobStores.abstractJobStore.AbstractJobStore method), 87
exportFile() (toil.common.Toil method), 84
exportFile() (toil.jobStores.abstractJobStore.AbstractJobStore method), 86

F

fileExists() (toil.jobStores.abstractJobStore.AbstractJobStore method), 90
FileID (class in toil.fileStores), 110
filesToDelete (toil.job.Promise attribute), 96
FunctionWrappingJob (class in toil.job), 93

G

getDefaultArgumentParser() (toil.job.Job.Runner static method), 105
getDefaultOptions() (toil.job.Job.Runner static method), 105
getEmptyFileStoreID() (toil.jobStores.abstractJobStore.AbstractJobStore method), 89
getEnv() (toil.jobStores.abstractJobStore.AbstractJobStore method), 87
getFileSize() (toil.jobStores.abstractJobStore.AbstractJobStore method), 90
getGlobalFileSize() (toil.fileStores.abstractFileStore.AbstractFileStore method), 109
getIssuedBatchJobIDs() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem method), 112
getJobStore() (toil.common.Toil class method), 83
getLocalTempDir() (toil.fileStores.abstractFileStore.AbstractFileStore method), 108
getLocalTempFile() (toil.fileStores.abstractFileStore.AbstractFileStore method), 108
getLocalTempFileName() (toil.fileStores.abstractFileStore.AbstractFileStore method), 108
getPublicUrl() (toil.jobStores.abstractJobStore.AbstractJobStore method), 87
getRootJobReturnValue() (toil.jobStores.abstractJobStore.AbstractJobStore method), 86
getRootJobs() (toil.job.Job method), 102
getRunningBatchJobIDs() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem method), 112
getSharedPublicUrl() (toil.jobStores.abstractJobStore.AbstractJobStore method), 87
getSize() (toil.jobStores.abstractJobStore.AbstractJobStore class method), 86
getTopologicalOrderingOfJobs() (toil.job.Job method), 103
getUpdatedBatchJob() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem method), 112
getValue() (toil.job.PromisedRequirement method), 96
getWorkflowDir() (toil.common.Toil static method), 84

H

hasChild() (toil.job.Job method), 100
hasFollowOn() (toil.job.Job method), 100

I

importFile() (toil.common.Toil method), 84
importFile() (toil.jobStores.abstractJobStore.AbstractJobStore method), 86
initialize() (toil.jobStores.abstractJobStore.AbstractJobStore method), 85
issueBatchJob() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem method), 112

J

Job (class in toil.job), 99
Job.Runner (class in toil.job), 105
Job.Service (class in toil.job), 115
JobException, 117
JobFunctionWrappingJob (class in toil.job), 93
JobGraphDeadlockException, 117
jobStore() (toil.jobStores.abstractJobStore.AbstractJobStore method), 88
JobStoreExistsException, 117

K

killBatchJobs() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem method), 112

L

load() (toil.jobStores.abstractJobStore.AbstractJobStore method), 88
loadRootJob() (toil.jobStores.abstractJobStore.AbstractJobStore method), 86
log() (toil.job.Job method), 101
logToMaster() (toil.fileStores.abstractFileStore.AbstractFileStore method), 110

N

NoSuchFileException, 117

NoSuchJobException, 118
NoSuchJobStoreException, 118

O

open() (toil.fileStores.abstractFileStore.AbstractFileStore method), 108

P

pack() (toil.fileStores.FileID method), 110
prepareForPromiseRegistration() (toil.job.EncapsulatedJob method), 95
prepareForPromiseRegistration() (toil.job.Job method), 102
Promise (class in toil.job), 96
PromisedRequirement (class in toil.job), 96

R

readFile() (toil.jobStores.abstractJobStore.AbstractJobStore method), 89
readFileStream() (toil.jobStores.abstractJobStore.AbstractJobStore method), 90
readGlobalFile() (toil.fileStores.abstractFileStore.AbstractFileStore method), 109
readGlobalFileStream() (toil.fileStores.abstractFileStore.AbstractFileStore method), 109
readSharedFileStream() (toil.jobStores.abstractJobStore.AbstractJobStore method), 91
readStatsAndLogging() (toil.jobStores.abstractJobStore.AbstractJobStore method), 91
restart() (toil.common.Toil method), 83
resume() (toil.jobStores.abstractJobStore.AbstractJobStore method), 85
run() (toil.job.FunctionWrappingJob method), 93
run() (toil.job.Job method), 99
run() (toil.job.JobFunctionWrappingJob method), 94
rv() (toil.job.EncapsulatedJob method), 95
rv() (toil.job.Job method), 102

S

setEnv() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem method), 113
setOptions() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem class method), 113
setRootJob() (toil.jobStores.abstractJobStore.AbstractJobStore method), 85
setUserScript() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem method), 112
shutdown() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem method), 113
shutdown() (toil.fileStores.abstractFileStore.AbstractFileStore class method), 110
shutdownFileStore() (toil.fileStores.abstractFileStore.AbstractFileStore static method), 108
start() (toil.common.Toil method), 83
start() (toil.job.Job.Service method), 115
startCommit() (toil.fileStores.abstractFileStore.AbstractFileStore method), 110
startToil() (toil.job.Job.Runner static method), 105
stop() (toil.job.Job.Service method), 115
supportsAutoDeployment() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem class method), 112
supportsWorkerCleanup() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem class method), 112

T

tempDir (toil.job.Job attribute), 101
Toil (class in toil.common), 83

U

unpack() (toil.fileStores.FileID class method), 110
update() (toil.jobStores.abstractJobStore.AbstractJobStore method), 88
updateFile() (toil.jobStores.abstractJobStore.AbstractJobStore method), 90
updateFileStream() (toil.jobStores.abstractJobStore.AbstractJobStore method), 90

W

waitForCommit() (toil.fileStores.abstractFileStore.AbstractFileStore method), 110
wrapFn() (toil.job.Job static method), 101
wrapJobFn() (toil.job.Job static method), 101
writeConfig() (toil.jobStores.abstractJobStore.AbstractJobStore method), 85
writeFile() (toil.jobStores.abstractJobStore.AbstractJobStore method), 88
writeFileStream() (toil.jobStores.abstractJobStore.AbstractJobStore method), 89
writeGlobalFile() (toil.fileStores.abstractFileStore.AbstractFileStore method), 108
writeGlobalFileStream() (toil.fileStores.abstractFileStore.AbstractFileStore method), 108
writePIDFile() (toil.common.Toil method), 84
writeSharedFileStream() (toil.jobStores.abstractJobStore.AbstractJobStore method), 90
writeStatsAndLogging() (toil.jobStores.abstractJobStore.AbstractJobStore method), 91