Automating Fetches with Python

How to Automate the Fetching Process

The Fetching process is one of the most essential processes for a production search engine. Automation of that process is also essential. This brief document will cover the JobStream.py python script that is used to automate the fetching process including fetching, updating the crawl database, and merging fetches into single segments. Please note that it is assumed that storage of the fetches is occuring on the Hadoop DFS (although the script could be altered to run on the local file system).

The JobStream.py Process

JobStream.py is a single class python script that automates the webpage fetching / update process.

The job starts by dumping the crawl database to a local disk from the master crawl database in the distributed file system (dfs). The dump files are then parsed to extract only unfetched urls which are appended to a single output file. Currently this is setup for only http urls. This url file is then split into multiple runs of x number of pages each. This is performed in a temp folder on the local file system. Each run is then consecutively loaded into the dfs, and run through the inject, generate, fetch, updatedb, and readdb commands. This happens in a temp folder on the dfs. There are options to setup x number of fetch runs before merging occurs.

The master and temp crawldb and segments folders are then merged into a new folder and then temp folders on the dfs as well as the temp folders on the local file system are deleted. The master folder is renamed to crawl-backup thereby giving us a single backup of previous data. This backup folder is removed and replaced for each url split.

The process removes the local url load directory before moving on to the next url split. The script finishes when all url splits have been run. There are also options to stop the script after the current fetch-merge run. You can then resume from that point at a later time.

The JobStream.py script automates only the fetching, updating, and merging processes. Inverting links and indexing are not covered yet although development of a more complete automation framework is underway.

The JobStream Options

To start using the JobStream file you will probably want to set some configuration variables in the main def of the script. If you prefer you can override most of these options on the command line. This is at the bottom of the script staring on line 374 and looks like this:

```
"""
Main method that starts up the script.
"""
def main(argv):

  # set the default values
  resume = 0
  execute = 0
  checkfile = "jobstream.stop"
  logconf = "logging.conf"
  jobdir = "/d01/jobstream"
  nutchdir = "/d01/revolution"
  masterdir = "crawl"
  backupdir = "crawl-backup"
  dfsdumpdir = "crawldump"
  tempdir = "crawltemp"
  splitsize = 500000
  fetchmerge = 3
```

Th checkfile variable is the name of the stop file to check for. If this file is present in the same directory as the JobStream.py script, the script will stop executing after the current fetch-update-merge run.

The logconf variable is the name of the logging file. You can configure logging for this script in a logging conf file that is in the same directory as the JobStream script. A sample logging file is provided later. Logging is set to go to the console by default.

The jobdir variable is the path of the local job directory. The one that the JobStream script live in and execute out of.

The nutchdir variable is the path to the base of the local nutch installation.

The masterdir variable is the path to the master directory on the dfs. This is the dfs directory that holds the crawldb and segments. The is also what the temp directory that holds the operations on the dfs will be named after the merge processes are finished and the old master directory is backed up.

The backupdir variable is the path to the backup directory on the dfs. This is where the old master directory will be moved to when the newly fetched and merged master is complete.

The dfsdump variable is the location on the dfs where the crawl database will be dumped to before being moved to the local file system for processing. This is deleted as soon as the files are moved to local.

The tempdir variable is not yet implemented but will be the location on the dfs where the temporary fetching and merging operations will occur. For right now this is hardcoded as crawltemp in the users directory on the dfs.

The splitsize variable is the number of urls to fetch in each load. The default is to fetch 500,000 urls in each load.

The fetchmerge variable is the number of fetches to run before merging. We want to keep the database up to date but we also don't want to waste processing time. By having more fetches per merge, the merge process can merge multiple segments and crawl database in a single execution instead of once per fetch. You can set this to 1 if you want to update once per fetch but the entire process will just take longer to complete. It is recommended to set this to between 3 and 5 depending on your bandwidth, processing power, and the size of your crawl databases and segments. The default is three fetches merged after the third fetch is complete.

Below is an example of the JobStream help screen. You can get to this screen at any point in time by using the -h or --help options.

```
JobStream.py [-hrjnlsmc]
             [-e | --execute] runs the jobstream.
             [-h | --help] prints this help and usage message.
             [-r | --resume] to resume a previous run.
             [-l | --logconf] Overrides logging conf file [logging.conf].
             [-j | --jobdir] The job directory, [/d01/jobstream].
             [-n | --nutchdir] The nutch home, [/d01/revolution].
             [-m | --masterdir] The dfs master directory [crawl].
             [-b | --backupdir] The master backup directory, [crawl-backup].
             [-s | --splitsize] The number of urls per load [500000].
             [-f | --fetchmerge] The number of fetches to run before merge [1].
```

The -e or --execute options run the jobstream. Use these option if you are starting a new jobstream process. If you are resuming an old process that you stopped before completion, use the -r or --resume options.

The rest of the options are pretty self explanatory and simply override the options that are set in the main def of the script. To start up a jobstream that will fetch 5 runs of 1,000,000 urls each before merging you would use a command like this.

```
./JobStream.py -s 1000000 -f 5
or alternatively
./JobStream.py --splitsize=1000000 --fetchmerge=5
```

This assumes that you have set basic options in the main def of the script and are overriding basic options.

The JobStream.py Script

```
#!/usr/bin/python

import sys
import getopt
import re
import logging
import logging.config
import commands
import os
import os.path

"""
---------------------------------------------------------------------------
The JobStream class automates the webpage fetching / update process.

The job starts by dumping the crawl database to a local disk from the master
crawl database in the distributed file system (dfs).  The dump files are then
parsed to extract only unfetched urls which are appended to a single output
file.  This url file is then split into multiple runs of x number of pages
each.  Each run is then consecutively loaded into the dfs, and run through
the inject, generate, fetch, updatedb, and readdb commands.  This happens in
a temp folder.

The master and temp crawldb and segments folders are then merged into a new
folder and then temp folders are deleted.  The master folder is renamed to
crawl-backup thereby giving us a single backup of previous data.  This backup
folder is removed and replaced for each url split.

The process removes the local url load directory before moving on to the next
```

```
url split.  The script finishes when all url splits have been run.

Use the -h or the --help flag to get a listing of options.

Program: JobStream Automation
Author: Dennis E. Kubes
Date: December 12, 2006
Revision: 1.2

Revision      | Author            | Comment
--------------------------------------------------------------------------------
20060906-1.0  Dennis E. Kubes     Initial creation of JobStream script.
20060907-1.1                      Added command line options for configuration,
                                  added comments, fixed bugs, added resume and
                                  stop after current url run completes.
20061205-1.2                      Added command line options and logic to run
                                  multiple fetches before merging.  This is to
                                  improve performance by not having to merge
                                  as often.
--------------------------------------------------------------------------------
TODO: Add dump and temp directory configuration options
--------------------------------------------------------------------------------
"""

class JobStream:

  nutchdir = ""
  masterdir = ""
  backupdir = ""
  log = logging.getLogger("jobstream")

  """
  Constructor for the JobStream class.  Passes in the nutch home directory and
  the nutch master directory
  """
  def __init__(self, nutchdir, masterdir, backupdir):

    # set the nutch directory home (where the bin directory for running the
    # nutch commands is), and the master crawl directory on the dfs
    self.nutchdir = nutchdir
    self.masterdir = masterdir
    self.backupdir = backupdir

  """
  Checks the status of result codes in the returned result array and raises an
  error if the result code is not a successful exit.
  """
  def checkStatus(self, result, err):
   if result[0] != 0:
      raise err + " " + result[1]

  """
  Dumps the master crawl database first to the dfs at the dfsdir location and
  then copies to the local file system at localdir.  The dfsdir is removed
  once the dump is copied to the local filesystem.
  """
  def dumpCrawlDb(self, dfsdir, localdir):

    # create the nutch commands
    crawldb = self.masterdir + "/crawldb"
    dump = self.nutchdir + "/bin/nutch readdb " + crawldb + " -dump " + dfsdir
    copylocal = self.nutchdir + "/bin/hadoop dfs -copyToLocal " + dfsdir + " " + localdir
    deletetemp = self.nutchdir + "/bin/hadoop dfs -rm " + dfsdir

    # execute the nutch commands and get each exit code, if it exited
    # with an error raise an exception
    nutchcmds = (dump, copylocal, deletetemp)
    for curcmd in nutchcmds:
      self.log.info("Running: " + curcmd)
      result = commands.getstatusoutput(curcmd)
      self.checkStatus(result, "Error occurred while running command " + curcmd)
```

```
"""
Parses the crawl dump files on the local filesystem for unfetched urls and
appends all of the unfetched urls to a single output file.
"""
def parseCrawlDump(self, indir, outfile):

  # open the output file in append mode
  outhandle = open(outfile, "a")

  # loop through each file in the dump directory but only get the
  # part-xxxxx files because there are .crc file in the same directory
  for dumpfile in os.listdir(indir):

    if dumpfile[0:4] == "part":

      curfile = indir + "/" + dumpfile
      self.log.info("Processing dump file: " + curfile)

      # set the input and output files
      inhandle = open(curfile, "r")

      # setup the regular expressions for searching and matching
      validUrl = "^http://(?! )(\w+[:]?){2,}(/?|[^ \n\r\"]+[\w/])(?=[\s\.,)\'\"\]])"
      urlRegex = re.compile(validUrl)
      unfetchedRegex = re.compile("DB_unfetched")

      # loop over the file, and match lines that are valid http:// urls where the
      # next line says that it is unfetched.  Write those lines to the outfile
      prevline = ""
      for line in inhandle:
        if urlRegex.search(line):
          prevline = line
        elif unfetchedRegex.search(line) and prevline != "":
          fields = prevline.strip("\n").split("\t")
          url = fields[0]
          prevline = ""
          outhandle.write(url + "\n")

      # close the file connections
      inhandle.close()

  # close the output file
  outhandle.close()

"""
Creates the url splits.  Each split will be a file that contains the number of
splitsize urls.  Each split is in its own number directory with a file in the
directory named urls that contains the actual urls to be fetched.
"""
def createUrlLoads(self, splitsize, urllist, outdir):

  # if the directory that holds the url loads doesn't exists, create it
  if not os.path.isdir(outdir):
    os.mkdir(outdir)

  # determine the total number of urls in the file and create a padding string
  # used to name the output folders correctly
  total_urls = 0
  urllinecount = open(urllist, "r")
  for line in urllinecount:
    total_urls += 1
  urllinecount.close()
  numsplits = total_urls / splitsize
  padding = "0" * len(repr(numsplits))

  # create the url load folder
  filenum = 0
  strfilenum = repr(filenum)
  urloutdir = outdir + "/urls-" + padding[len(strfilenum):] + strfilenum
  os.mkdir(urloutdir)
```

```
  urlfile = urloutdir + "/urls"

  # open the input and output files
  self.log.info("Creating load file: " + urlfile)
  inhandle = open(urllist, "r")
  outhandle = open(urlfile, "w")

  # loop through the file
  for linenum, line in enumerate(inhandle):

    # if we have come to a split then close the current file, create a new
    # url folder and open a new url file
    if linenum > 0 and linenum % splitsize == 0:

      filenum += 1
      strfilenum = repr(filenum)
      urloutdir = outdir + "/urls-" + padding[len(strfilenum):] + strfilenum
      os.mkdir(urloutdir)
      urlfile = urloutdir + "/urls"
      self.log.info("Creating load file: " + urlfile)
      outhandle.close()
      outhandle = open(urlfile, "w")

    # write the url to the file
    outhandle.write(line)

  # close the input and output files
  inhandle.close()
  outhandle.close()

"""
Runs all of the url loads.  For each of the url load directories it loads the
url file into the dfs and then runs the inject, generate, fetch, readdb, and
updatedb commands.  This temp load is then merged with the master database to
create a new database.  The old database is stored as backup, the new database
takes the place as the master database, and the temp database is removed.

At the end of each url load the process will check to see if a stopfile is
present on the system.  If the file is present, this indicates to the process
to stop running.  The stop file will be removed and the process will terminate.
The process can be restarted later with the -r or --resume flags and it will
pick up where it left off on the next url load.
"""
def runFetchMerges(self, urllistdir, stopfile, fetchmerges):

  # get the folders for the url directory and sort them, this is reason the
  # folders needs to be named correctly, so they are loaded in order
  urldirs = os.listdir(urllistdir)
  urldirs.sort()
  counter = 0
  numdirs = len(urldirs)

  # variables for the crawldb and segments directories on the dfs
  mastercrawldbdir = self.masterdir + "/crawldb"
  mastersegsdir = self.masterdir + "/segments"

  # for each of the url loads
  while counter < numdirs:

    # list to hold individual fetches and segments
    tempseglist = []
    tempdblist = []

    # run the number of fetches before merging, this allows us to improve
    # performance by not having to merge as often
    for curl in urldirs[counter:counter + fetchmerges]:

      # set the current load directory
      curloaddir = urllistdir + "/" + curl
      self.log.info("Starting current load: " + curloaddir)
```

```
    # create the nutch commands for load, inject, generate
    tempdb = "crawltemp/crawldb" + str(counter)
    tempdblist.append(tempdb)
    load = self.nutchdir + "/bin/hadoop dfs -put " + curloaddir + " crawltemp/urls" + str(counter)
    inject = self.nutchdir + "/bin/nutch inject " + tempdb + " crawltemp/urls" + str(counter)
    generate = self.nutchdir + "/bin/nutch generate " + tempdb + " crawltemp/segments" + str(counter)

    # run the load, inject, and generate commands, check the results, if bad exit
    nutchcmds = (load, inject, generate)
    for curcmd in nutchcmds:
      self.log.info("Running: " + curcmd)
      result = commands.getstatusoutput(curcmd)
      self.checkStatus(result, "Error occurred while running command" + curcmd)

    # get the current segment to fetch
    self.log.info("Getting segment to fetch.")
    getsegment = self.nutchdir + "/bin/hadoop dfs -ls crawltemp/segments" + str(counter)
    self.log.info("Running: " + getsegment)
    result = commands.getstatusoutput(getsegment)
    self.checkStatus(result, "Error occurred while running command" + getsegment)

    # fetch the current segment
    outar = result[1].splitlines()
    output = outar[-1]
    tempseg = output.split()[0]
    tempseglist.append(tempseg)
    fetch = self.nutchdir + "/bin/nutch fetch " + tempseg
    self.log.info("Starting fetch for: " + tempseg)
    self.log.info("Running: " + fetch)
    result = commands.getstatusoutput(fetch)
    self.checkStatus(result, "Error occurred while running command" + fetch)
    self.log.info("Finished fetch for: " + tempseg)

    # update the crawldb from the current segment
    self.log.info("Updating " + tempdb + " from " + tempseg + ".")
    updatetemp = self.nutchdir + "/bin/nutch updatedb " + tempdb + " " + tempseg
    self.log.info("Running: " + updatetemp)
    result = commands.getstatusoutput(updatetemp)
    self.checkStatus(result, "Error occurred while running command" + updatetemp)

     # remove the current url load directory
    self.log.info("Removing current local load directory: " + curloaddir)
    os.remove(curloaddir + "/urls")
    os.rmdir(curloaddir)

    # log the current url finished
    self.log.info("Finished current load: " + curloaddir)

    #increment the counter
    counter += 1

  # merge the crawldbs
  self.log.info("Merging master and temp crawldbs.")
  crawlmerge = self.nutchdir + "/bin/nutch mergedb mergetemp/crawldb " + \
    mastercrawldbdir + " " + " ".join(tempdblist)
  self.log.info("Running: " + crawlmerge)
  result = commands.getstatusoutput(crawlmerge)
  self.checkStatus(result, "Error occurred while running command" + crawlmerge)

  # merge the segments
  self.log.info("Merging master and temp segments")
  getsegment = self.nutchdir + "/bin/hadoop dfs -ls " + mastersegsdir
  result = commands.getstatusoutput(getsegment)
  self.checkStatus(result, "Error occurred while running command" + getsegment)
  outar = result[1].splitlines()
  output = outar[-1]
  masterseg = output.split()[0]
  mergesegs = self.nutchdir + "/bin/nutch mergesegs mergetemp/segments " + \
    masterseg + " " + " ".join(tempseglist)
  self.log.info("Running: " + mergesegs)
  result = commands.getstatusoutput(mergesegs)
```

```
      self.checkStatus(result, "Error occurred while running command" + mergesegs)

      # back up the master, rename the merged to master, and remove the temp
      self.log.info("Backing up, deleting, and renaming of merge resources")
      rmoldback = self.nutchdir + "/bin/hadoop dfs -rm " + self.backupdir
      self.log.info("Running: " + rmoldback)
      result = commands.getstatusoutput(rmoldback)
      self.checkStatus(result, "Error occurred while running command" + rmoldback)
      masterback = self.nutchdir + "/bin/hadoop dfs -mv " + self.masterdir + " " + self.backupdir
      self.log.info("Running: " + masterback)
      result = commands.getstatusoutput(masterback)
      self.checkStatus(result, "Error occurred while running command" + masterback)
      mergemove = self.nutchdir + "/bin/hadoop dfs -mv mergetemp " + self.masterdir
      self.log.info("Running: " + mergemove)
      result = commands.getstatusoutput(mergemove)
      self.checkStatus(result, "Error occurred while running command" + mergemove)
      deltemp = self.nutchdir + "/bin/hadoop dfs -rm crawltemp"
      self.log.info("Running: " + deltemp)
      result = commands.getstatusoutput(deltemp)
      self.checkStatus(result, "Error occurred while running command" + deltemp)

      # read the current master database stats and put the output into the log
      self.log.info("Reading database statistics")
      dbname =  self.masterdir + "/crawldb"
      readdb = self.nutchdir + "/bin/nutch readdb " + dbname + " -stats"
      self.log.info("Running: " + readdb)
      result = commands.getstatusoutput(readdb)
      self.checkStatus(result, "Error occurred while running command" + readdb)
      output = result[1]
      self.log.info(output)

      # check to see if the stop file is present, if it is remove the stopfile
      # and exit the script successfully
      if os.path.isfile(stopfile):
        self.log.info("Found stopfile " + stopfile + ".  Removing stopfile and " +
          "exiting script.")
        os.remove(stopfile)
        sys.exit(0)

"""
Prints out the usage for the command line.
"""
def usage():
  usage = ["JobStream.py [-hrjnlsmc]\n"]
  usage.append("            [-e | --execute] runs the jobstream.\n")
  usage.append("            [-h | --help] prints this help and usage message.\n")
  usage.append("            [-r | --resume] to resume a previous run.\n")
  usage.append("            [-l | --logconf] Overrides logging conf file [logging.conf].\n")
  usage.append("            [-j | --jobdir] The job directory, [/d01/jobstream].\n")
  usage.append("            [-n | --nutchdir] The nutch home, [/d01/revolution].\n")
  usage.append("            [-m | --masterdir] The dfs master directory [crawl].\n")
  usage.append("            [-b | --backupdir] The master backup directory, [crawl-backup].\n")
  usage.append("            [-s | --splitsize] The number of urls per load [500000].\n")
  usage.append("            [-f | --fetchmerge] The number of fetches to run before merging [1].\n")
  message = " ".join(usage)
  print message

"""
Main method that starts up the script.
"""
def main(argv):

  # set the default values
  resume = 0
  execute = 0
  checkfile = "jobstream.stop"
  logconf = "logging.conf"
  jobdir = "/d01/jobstream"
  nutchdir = "/d01/revolution"
  masterdir = "crawl"
  backupdir = "crawl-backup"
```

```
dfsdumpdir = "crawldump"
tempdir = "crawltemp"
splitsize = 500000
fetchmerge = 3

try:

  # process the command line options
  opts, args = getopt.getopt(argv, "hrej:n:l:s:m:b:f:d:t:", ["help","logconf=","jobdir=",
    "resume","nutchdir=","splitsize=","masterdir=","backupdir=","fetchmerge=", "execute",
    "dumpdir=","tempdir="])

  # if no arguments print usage
  if len(argv) == 0:
    usage()
    sys.exit()

  # loop through all of the command line options and set the appropriate
  # values, overriding defaults
  for opt, arg in opts:
    if opt in ("-h", "--help"):
      usage()
      sys.exit()
    elif opt in ("-r", "--resume"):
      resume = 1
    elif opt in ("-e", "--execute"):
      execute = 1
    elif opt in ("-l", "--logconf"):
      logconf = arg
    elif opt in ("-j", "--jobdir"):
      jobdir = arg
    elif opt in ("-n", "--nutchdir"):
      nutchdir = arg
    elif opt in ("-m", "--masterdir"):
      masterdir = arg
    elif opt in ("-b", "--backupdir"):
      backupdir = arg
    elif opt in ("-d", "--dumpdir"):
      dfsdumpdir = arg
    elif opt in ("-t", "--tempdir"):
      tempdir = arg
    elif opt in ("-s", "--splitsize"):
      splitsize = int(arg)
    elif opt in ("-f", "--fetchmerge"):
      fetchmerge = int(arg)

except getopt.GetoptError:

  # if an error happens print the usage and exit with an error
  usage()
  sys.exit(2)

# if we are running the jobstream
if execute:

  # setup the file and folder variables, setup the logging
  logging.config.fileConfig(logconf)
  localdumpdir = jobdir + "/localdump"
  loaddir = jobdir + "/urllists"
  unfetched = localdumpdir + "/unfetched.urls"
  checkpath = jobdir + "/" + checkfile

  # create the job stream object
  jobStream = JobStream(nutchdir, masterdir, backupdir)

  # if we not resuming then dump, parse and create the url loads
  if not resume:
    jobStream.dumpCrawlDb(dfsdumpdir, localdumpdir)
    jobStream.parseCrawlDump(localdumpdir, unfetched)
    jobStream.createUrlLoads(splitsize, unfetched, loaddir)
```

```
    # merging or not run the url loads
    jobStream.runFetchMerges(loaddir, checkpath, fetchmerge)

# if we are running the script from the command line, run the main
# method of the JobStream class
if __name__ == "__main__":
  main(sys.argv[1:])
```

The JobStream Logging.conf File

```
[formatters]
keys=simple

[handlers]
keys=console

[loggers]
keys=root,engine

[formatter_simple]
format=%(name)s :%(levelname)s :  %(message)s

[handler_console]
class=StreamHandler
args=[]
formatter=simple

[logger_root]
level=INFO
handlers=console

[logger_engine]
level=INFO
qualname=jobstream
propagate=0
handlers=console
```

Conclusion

If you want to get further into the operations of the script I suggest reading the source. It is well documented with both documentation and code comments. Alternatively you can send me an email at nutch-dev at dragonflymc dot com if you have any questions.