jeremyaldrich.net Documentation

Release 1.0

Jeremy Aldrich

Apr 23, 2019

Contents

Contents:

CHAPTER 1

About Me

Work stuff: http://linkedin.com/in/jeraldrich

Fun stuff: https://github.com/jeraldrich

Feel free to email at jeremy@jeremyaldrich.net

CHAPTER 2

Contact Me

The best way to contact me is by email at jeremy@jeremyaldrich.net

CHAPTER 3

Multiprocessing SQLAlchemy Largefile Processing

The first step of any analytics data pipeline is to integrate external datasources, filter, and insert into a datastore. Usually, threading is good enough to process each line and insert into the database.

However, when processing large files (1GB+), one cpu core will spike at 100% and the GIL will fight for CPU contention between parsing each line, filtering / inserting the data, and switching between threads.

In this example, I have a simple table called chat_messages which is populated by parsing a large json file of JSON strings. The main point of this is to show how easy it is to use multiprocessing.

I am using python multiprocessing to parse the large file in one process, and filter/insert the data in every other CPU avaliable. This dropped CPU usage from 100% on one core, to 7-12% per core. When inserting 10+ million rows into a mysql database, total time of using twisted / threaded vs multiprocessing dropped from 10+ minutes, to 2 minutes.

I used the consumer producer pattern because then I could easily share one queue between all of the different processes. One producer fills the queue with each parsed line from the log file, and a consumer is spawned per CPU core which consumes each line in the queue.

A mysql session pool is created and shared between every consumer process (SQLAlchemy). When there are 500+ line items in the consumer queue, mysql inserts will be automatically batched.

Here's the full source: https://github.com/jeraldrich/MSLP

My consumer / producer processes are managed by using a multiprocessing manager queue which is wrapped in a class that spawns and joins the producer / consumer processes:

```
from multiprocessing import Process, cpu_count, Manager
from os import sys
import time
import logging
from Queue import Empty

from sqlalchemy.orm import scoped_session, sessionmaker
from sqlalchemy import asc

from producers import LargeFileParser, ChatMessageParser
from consumers import create_mysql_pool, batch_insert
```

(continues on next page)

(continued from previous page)

```
from consumers.models import ChatMessage
from settings import CHAT_LOG


logger = logging.getLogger('chat_message_parser')
logger.setLevel(logging.DEBUG)
logging.basicConfig()
stream_handler = logging.StreamHandler()
stream_handler.setLevel(logging.INFO)
logger.addHandler(stream_handler)


def producer_queue(queue, parser):
    for data in LargeFileParser(CHAT_LOG):
        parsed_data = parser.parse(data)
        queue.put(parsed_data)
    queue.put('STOP')


def consumer_queue(proc_id, queue):

    # shared pooled session per consumer proc
    mysql_pool = create_mysql_pool()
    session_factory = sessionmaker(mysql_pool)
    Session = scoped_session(session_factory)

    while True:
        try:
            time.sleep(0.01)
            consumer_data = queue.get(proc_id, 1)
            if consumer_data == 'STOP':
                logger.info('STOP received')
                # put stop back in queue for other consumers
                queue.put('STOP')
                break
            consumer_data_batch = []
            consumer_data_batch.append(consumer_data)
            if queue.qsize() > 500:
                for i in xrange(50):
                    consumer_data = queue.get(proc_id, 1)
                    consumer_data_batch.append(consumer_data)
            session = Session()
            batch_insert(session, consumer_data_batch)
            # logger.info(consumer_data)
        except Empty:
            pass


class ParserManager(object):

    def __init__(self):
        self.manager = Manager()
        self.queue = self.manager.Queue()
        self.NUMBER_OF_PROCESSES = cpu_count()
        self.parser = ChatMessageParser()

    def start(self):
```

(continues on next page)

(continued from previous page)

```
        self.producer = Process(
            target=producer_queue,
            args=(self.queue, self.parser)
        )
        self.producer.start()

        self.consumers = [
            Process(target=consumer_queue, args=(i, self.queue,))
            for i in xrange(self.NUMBER_OF_PROCESSES)
        ]
        for consumer in self.consumers:
            consumer.start()

    def join(self):
        self.producer.join()
        for consumer in self.consumers:
            consumer.join()

if __name__ == '__main__':
    try:
        manager = ParserManager()
        manager.start()
        manager.join()
    except (KeyboardInterrupt, SystemExit):
        logger.info('interrupt signal received')
        sys.exit(1)
    except Exception, e:
        raise e
```

When using python multiprocessing, you will want to use the multiprocessing module to create all queues and threads. Otherwise, you may get a deadlock when two seperate processes try to read from the same queue at once.

By seperating the producer and consumers, the main flow of the program becomes very simple to manage. You can immediatly tell from the code what is going on, and add other SQLAlchemy models as needed.

An important thing to note: A mysql session pool is created per consumer process. You will want to do this with SQLAlchemy, because global session pools cannot be shared among multiple processes.

Here's what the SQLAlchemy model looks like:

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, String, TIMESTAMP


Base = declarative_base()


class ChatMessage(Base):
    __tablename__ = 'chat_messages'
    __table_args__ = {'mysql_engine': 'InnoDB'}

    id = Column('id', String(255), primary_key=True)
    _from = Column('from', String(255), nullable=False)
    _type = Column('type', String(50))
    site_id = Column('site_id', String(50), nullable=False, index=True)
    data = Column('data', String(255), default='')
    timestamp = Column('timestamp', TIMESTAMP, nullable=False, index=True)
```

(continues on next page)

(continued from previous page)

```
    def __repr__(self):
        return "id='{id}',ts='{ts}',type='{t}',data='{data}'>".format(
            id=self.id,
            ts=self.timestamp,
            t=self._type,
            data=self.data,
        )
```

Here's what I use to filter each json line and create a new SQLAlchemy object:

```
import json
import logging
from datetime import datetime

from consumers.models import ChatMessage


logger = logging.getLogger('chat_message_parser')

class ChatMessageParser():

    def parse(self, data):
        """
        parse each line in the log and populate log data with matched pattern
        """
        json_message = json.loads(data)
        # logger.info('json data is {0}'.format(json_message['data']))
        chat_message = None
        if json_message['type'] == 'message':
            chat_message = ChatMessage(
                id=json_message['id'],
                _from=json_message['from'],
                _type=json_message['type'],
                site_id=json_message['site_id'],
                data=json_message['data']['message'],
                timestamp=json_message['timestamp'],
            )
        elif json_message['type'] == 'status':
            chat_message = ChatMessage(
                id=json_message['id'],
                _from=json_message['from'],
                _type=json_message['type'],
                site_id=json_message['site_id'],
                data=json_message['data']['status'],
                timestamp=json_message['timestamp'],
            )
        else:
            logger.error('Invalid json status detected'.format(chat_message))
            return None
        # convert timestamp str to datetime
        chat_message.timestamp = datetime.fromtimestamp(int(json_message['timestamp
↪']))

        return chat_message
```

For my producer, a large file is split up into chunks, and then each chunk yields a line into the consumer queue:

```
from itertools import chain, islice
import os
import time
import logging

logger = logging.getLogger('chat_message_parser')

class LargeFileParser(object):

    def __init__(self, filename):
        self.filename = filename
        self.split_files = []
        # lines per split file
        self.split_every = 100000
        self._split_large_file()

    def __iter__(self):
        logger.info('yielding')
        while self.split_files:
            split_file = self.split_files.pop()
            with open(split_file, 'rU') as f:
                lines = f.readlines()
                for line in lines:
                    yield line
            logger.info('removing split_file')
            os.remove(split_file)
        lines = None
        logger.info('end')

    def _split_large_file(self):
        if not os.path.isfile(self.filename):
            raise Exception(
                'file does not exist:{0}'.format(self.filename)
            )
        def _chunks(chunk_iterable, n):
           chunk_iterable = iter(chunk_iterable)
           while True:
               yield chain([next(chunk_iterable)], islice(chunk_iterable, n-1))
        with open(self.filename) as bigfile:
            for i, lines in enumerate(_chunks(bigfile, self.split_every)):
                file_split = '{}.{}'.format(self.filename, i)
                with open(file_split, 'w') as f:
                    f.writelines(lines)
                self.split_files.append(file_split)
        #logger.info(self.split_files)
        return True
```

Instead of splitting a large file, you could probably iterate over chunks and use fileseek, but splitting the file up allows me to use multiple consumers if disk IO is not a bottleneck.

CHAPTER 4

Why Python and Go is the Future

When comparing languages, it's easy to compare each one's features and capabilities on paper.

Lost in this comparison is what you gain as a collaborative team by the efficiency and maintainability of a language favoring simplicity and practicality over cleverness and possibilities.

Restrictions like enforced whitespace as part of the code logic, and in most cases, only one way to do something, mean that most Python projects are easily understood and similarly constructed, regardless of who originally wrote it.

Code is about understanding what the code says, how it's piece fits into the larger puzzle, and knowing the estimation and effect of each addition or correction.

Python is almost like pseudocode.

Like english, it makes sense immediately. The characters and expressions used sometimes mathematical, but logic behind the meaning clear.

Go can be used to replace existing Python projects which work just fine in 90% of cases, but you've grown to a scale in which the 10% becomes difficult to reliably service.

There are no classes in Go. This might seem crazy for an enterprise ready language, but this is by language design choice, not language immaturity.

Instead of classes in Go, you have interfaces and structs. Structs are collections of strongly declared types, which can also include interfaces. Interfaces are a collection of structs, or a method doing one thing.

Go by design, encourages a composition pattern approach to object oriented programming rather than an initially elegant, but often times eventually complex, class inheritance hierarchy.

Instead of thinking about all the different types of cars a vehicle can be, design your vehicle so it can be easily used in ways you didn't think of - such as a bike. Instead of defining a base class with an engine and what your idea of 'vehicle' means, design 'vehicle' to mean a collection of individual parts assembled together that serve a specific purpose.

Design 'engine' as a method to propel an object forward, rather than a combustable motor.

This forces you to design methods and functionality so that your methods do one and one thing only.

Unix environments follow the philosophy of doing one thing only and one thing well. This allows the use of individual commands which do one thing, to be chained together with other commands through pipes "|" accepting each other's inputs and outputs until the final result is returned.

The road of software project estimation often meanders when over confidence on how easy you think a solution is, encounters a road block in which your original design did not anticipate. Maybe this special case can be accounted for easily, but other times, it may require a complete change in the underlying design.

Being able to rapidly discover what those edge cases are, and being able to modify your project to account for it, rather than having to refactor a complex class hierarchy, allows you to focus on rapidly iterating and figure out 'what works' right now.

Like a simple command, if your software library, method, or program just does 'one thing and one thing well', it can easily be utilized without the implementor having to be aware of the underlying details.

Go is a strongly type language. This means that variables in Go must be declared as to what type they are. In an interpreted dynamic language like Python, the type is assumed to be the type of the first value you assigned to it, so you do not repeat yourself, and can crank out a feature without to much plumbing.

At first, it seems the extra step of strongly typing things is unnecessary and annoying. However, once a project grows to a certain size, there's probably going to be some code somewhere at a random place that checks type and casts it into something else, or gnarly conditional branches in the code based off of handling different data structure types.

Describing the structures that your method accepts as input and the type of result it will always return, forces you to think visually how all the individual structures of your program fit together - like lego blocks with clearly defined edges that click and hold firmly in place.

Visualize the data structure that should be returned for a new method first, and work backwards from there to figure out the implementation details.

Things like auto multi core distribution for threads which are many times more memory efficient and performant than python threads, true concurrency, near C like performance, and passing variables by pointers rather than by copying the values result in exponential performance gains once you hit a certain scale.

While a few milliseconds may seem insiginficate, at hundreds of thousands of writes / requests to a database or service per second, those few milliseconds add up.

A real world example: http://highscalability.com/blog/2014/5/7/update-on-disqus-its-still-about-realtime-but-go-demolishes.html

If you are a python / ruby / node web dev, you are familiar with the clown fiesta that is maintaining large amounts of dependencies between different environments.

In Go, you simply type in go [program-name-here] and a binary is compiled for you to run on any operating system. The deployment method is to simply to copy / download the binary and run.

While the languages themselves might be different, both follow a similar design philosophy of simplicity and practicality. The cross over between the two feels natural. The mental cost switching context between them, cheap.

Knowing them both very well allows you to fulfill two very important and distinct stages of a product: Rapidly implement an idea to test the market adoption, and once the 10% edge cases crop up when certain scales are reached, implement the performance critical parts in Go where needed.

CHAPTER 5

Django FileField CSV Validation

Let's say that you have a User that want's to attach a CSV file. You would like to validate that CSV file before saving it to your filesystem and associating to the user.

Django model validators make this easy https://docs.djangoproject.com/en/dev/ref/validators/

A validator is a callback function for a model field that will either return True or return a ValidationError exception message which will then be displayed as an error for that field in Django admin.

Pretty handy.

For this simple example, I have a csv file that needs to be checked for valid filetype, empty cell values on required columns, and missing headers.

The HEADERS map is designed to be referenced by a separate import process not included in this example, but may give you an idea on how to implement additional sanitization checking and database actions.

If you do decide to have additional file data sanitization checks, I would recommend refactoring the import_document_validator into separate smaller validator functions and append them to the validators list. Otherwise, you are probably going to end up with a 200+ line validator function.

Here's an example:

```
import csv

from django.conf import settings
from django.db import models
from django.core.exceptions import ValidationError
from django.utils.translation import ugettext_lazy as _


# used to map csv headers to location fields
HEADERS = {
    'shop_id': {'field':'id', 'required':True},
    'platinum_member': {'field':'platinum_member', 'required':False},
}
```

(continues on next page)

(continued from previous page)

```
def import_document_validator(document):
    # check file valid csv format
    try:
        dialect = csv.Sniffer().sniff(document.read(1024))
        document.seek(0, 0)
    except csv.Error:
        raise ValidationError(u'Not a valid CSV file')
    reader = csv.reader(document.read().splitlines(), dialect)
    csv_headers = []
    required_headers = [header_name for header_name, values in
                        HEADERS.items() if values['required']]
    for y_index, row in enumerate(reader):
        # check that all headers are present
        if y_index == 0:
            # store header_names to sanity check required cells later
            csv_headers = [header_name.lower() for header_name in row if header_name]
            missing_headers = set(required_headers) - set([r.lower() for r in row])
            if missing_headers:
                missing_headers_str = ', '.join(missing_headers)
                raise ValidationError(u'Missing headers: %s' % (missing_headers_str))
            continue
        # ignore blank rows
        if not ''.join(str(x) for x in row):
            continue
        # sanity check required cell values
        for x_index, cell_value in enumerate(row):
            # if indexerror, probably an empty cell past the headers col count
            try:
                csv_headers[x_index]
            except IndexError:
                continue
            if csv_headers[x_index] in required_headers:
                if not cell_value:
                    raise ValidationError(u'Missing required value %s for row %s' %
                                          (csv_headers[x_index], y_index + 1))
    return True

class Import(models.Model):
    imported = models.BooleanField(default=False)
    name = models.CharField(primary_key=True, max_length=255)
    document = models.FileField(upload_to='imports', validators=[import_document_
↪validator])
    import_date = models.DateField(auto_now=True)

    def __unicode__(self):
        return self.name
```

CHAPTER 6

Useful Sysadmin Commands

A collection of helpful Linux / Freebsd commands I find helpful for day to day use

6.1 Finding files and executing actions on them

Find and delete files with filename length of 5:

```
find . -type f -name '?????' -exec rm -f {} \;
```

Remove all data from file without deleting file:

```
truncate -s0 access.log
```

List all file and directory disk space usage sorted by 10 largest directories:

```
sudo du -cks * | sort -rn | head
```

6.2 Listing processes and process system usage

List memory usage by category of process:

```
ps aux | awk '{print $4"\t"$11}' | sort | uniq -c | awk '{print $2" "$1" "$3}' | sort␣
↪-nr
```

List memcache objects:

```
ngrep -W none -T -d any "^(get|set|delete|END|STORED|VALUE|DELETED)" port 11211 | awk
↪'{print $1 " " $2}'
```

6.3 SSH Commands

Forward custom port (local requests to MySQL in this example) to a remote host:

```
ssh -fND 3306 username@bestwebsiteintheworld.com
```

6.4 Troubleshooting Network Traffic

Capture http headers with tcpdump:

```
tcpdump -s 1024 -C 1024000 -w /tmp/httpcapture dst port 80
```

Check for connections to a database not closing (left in TIME_WAIT status):

```
netstat -an | grep TIME_WAIT
```

Capture packets for a particular Destination IP and Port:

```
tcpdump -w packet_capture_results.pcap -i eth0 dst 10.0.1.8 and port 22
```

Capture all packets except those that match packet type filter:

```
tcpdump -i eth0 not arp and not rarp
```

6.5 Tar Commands

Tar a directory and encrypt it in one line:

```
tar cvzf - example_dir | openssl des3 -salt -k secretkey | dd of=encrypted_example_dir
```

To decrypt:

```
dd if=encrypted_example_dir | openssl des3 -d -k secretkey | tar xvzf -
```

CHAPTER 7

Vagrant AWS Puppet Rsync

While deploying from vagrant to AWS using vagrant-aws, you may run into an error if you are sharing folders. To share folders on AWS, you would use rsync.

If rsync is not installed on your base AMI, you may run into an issue where you are unable to share your folders and your box will not provision.

Before initializing puppet, you will want to use a script to install rsync.

Here is my vagrant file + custom init script to resolve this issue:

```
# Vagrantfile API/syntax version. Don't touch unless you know what you're doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "dummy"
  config.ssh.pty = true
  config.vm.define :awsinstance do |awsi|
    # setup ami, aws
    awsi.vm.provider :aws do |aws, override|
        aws.keypair_name = "YOURKEYPAIR"
        aws.access_key_id = ""
        aws.secret_access_key = ""
        aws.region = "us-east-1"
        # debian wheezy 7.6
        aws.ami = "ami-c4ab67ac"
        override.ssh.private_key_path = "YOURPRIVATEKEY"
        override.ssh.username = "USERNAME"
    end
    # setup puppet
    awsi.vm.hostname = "myvm"
    awsi.vm.provision :shell, :path => "shell/init.sh"
    awsi.vm.provision :puppet do |awspuppet|
        awspuppet.manifest_file = "nodes.pp"
        awspuppet.manifests_path = "/myapp/manifests/manifests"
        awspuppet.module_path = "/myapp/manifests/modules"
```

(continues on next page)

(continued from previous page)

```
    end
    # mount folders
    awsi.vm.synced_folder "/myapp/stuff", "/myapp/stuff",
                          owner: "admin",
                          group: "root",
                          :mount_options => ['dmode=775','fmode=775'], type: "rsync"
  end
  config.ssh.forward_agent = false
end
```

The shell/init.sh script (which is ran before puppet modules are initialized):

```
#!/bin/sh
$(which rsync > /dev/null 2>&1)
FOUND_RSYNC=$?
if [ "$FOUND_RSYNC" -ne '0' ]; then
    echo 'Attempting to install rsync'
    $(which apt-get > /dev/null 2>&1)
    FOUND_APT=$?
    $(which yum > /dev/null 2>&1)
    FOUND_YUM=$?

    if [ "${FOUND_YUM}" -eq '0' ]; then
        yum -q -y makecache
        yum -q -y install rsync
        echo 'rsync installed.'
        yum -q -y install puppet
        echo 'puppet installed.'
    elif [ "${FOUND_APT}" -eq '0' ]; then
        apt-get -q -y update
        apt-get -q -y install rsync
        echo 'rsync installed.'
        apt-get -q -y install puppet
        echo 'puppet installed.'
    else
        echo 'No package installer available'
    fi
fi
# Set the first portion of this to match the node defined in your manifest
# example node of crunch should be crunch.X
hostname myvm.va.localdomain
```