

# Weighted Distribution Of Tests

## Table of contents

1 Introduction.....	2
2 Statement Of The Problem.....	2
3 Test Cases.....	2
4 Weight Distribution Definition.....	2
5 Accumulator Function.....	3
6 Random Numbers.....	3
7 Test Runner Class.....	4
8 Putting It All Together.....	4
9 Full Script Listing.....	5

## 1. Introduction

This is a step-by-step tutorial on how to schedule tests according to any "weight distribution" you desire. This is an exercise in data structures and random numbers, and as such it does not use any facilities of The Grinder (such as HTTPClient) except for its core TestRunner functionality. Therefore, it is immediately applicable to almost any test scenario.

**Walt Tuvell**

## 2. Statement Of The Problem

Let's assume you have a collection of four kinds of tests you want to run, say CREATE, READ, UPDATE, DELETE. These might be operations on a Web Server, or a Database Server, for example.

Suppose further you want to run your tests using many threads (grinder.threads property in the grinder.properties file), and you want to schedule these threads amongst the tests according to a specified "weighted distribution". As an example, we'll assume you want to run: 20% CREATEs, 40% READs, 30% UPDATEs, 10% DELETEs.

How can you do this?

## 3. Test Cases

Note that the problem statement is independent of the actual tests themselves. So for illustrative purposes, we will choose dummy tests that simply print a message to stdout. (Your tests will likely make use of deeper facilities of The Grinder, such as HTTPClient, etc.)

```
def doCREATEtest():
    print 'Doing CREATE test ...'
def doREADtest():
    print 'Doing READ test ...'
def doUPDATEtest():
    print 'Doing UPDATE test ...'
def doDELETETest():
    print 'Doing DELETE test ...'
```

## 4. Weight Distribution Definition

The most flexible way to define test distribution is by means of "relative weights", that is, numbers which specify the number of times each test is to be run relative to one another (as opposed to an absolute number of runs - for that, see the grinder.runs property in the grinder.properties file).

For our example, we begin by defining our desired weight distribution in a table (Jython dictionary structure) like the following:

```
g_weights = {
```

```
'CREATE': 2,  
'READ'   : 4,  
'UPDATE': 3,  
'DELETE': 1,  
}
```

Since the weights in this table are relative, we could multiply all their values by a constant and arrive at the same weight distribution. (The same goes for division, provided we end up with integers.) If the sum of weights adds up to 100, the weights can be interpreted directly as "percentages". For example, in our example, if we multiplied our weights by 10, we'd end up with exactly the percentage values in the original statement of our example.

Note that string-names in the weight table are arbitrary tags (they will be mapped to the tests in `TestRunner.__call__()`). As a matter of style, the weight table should be placed near the top of your script, so its settings can be modified easily from run to run, according to the test scenarios you want to model.

## 5. Accumulator Function

All the magic of choosing which test to run according to your specified weight distribution is accomplished by the following "accumulator" function:

```
def weightAccumulator(i_dict):  
    keyList = i_dict.keys()  
    keyList.sort() # sorting is optional - order coming-in doesn't  
matter, but determinism is kinda cool  
    listAcc = []  
    weightAcc = 0  
    for key in keyList:  
        weightAcc += i_dict[key]  
        listAcc.append((key, weightAcc))  
    return (listAcc, weightAcc) # order going-out does matter -  
hence "listAcc" instead of "dictAcc"  
  
    g_WeightsAcc, g_WeightsAccMax = weightAccumulator(g_Weights)  
    g_WeightsAccLen, g_WeightsAccMax_1 = len(g_WeightsAcc),  
g_WeightsAccMax-1
```

This accumulator function takes a weight dictionary as input, and transforms it into an accumulated weight list, suitable for random indexing, as we will do below.

As shown above, the accumulator function is called with `g_Weights` as input, and its output is captured in two convenience variables. Two more convenience variables are also defined, for use below.

## 6. Random Numbers

Next, we prepare a random number generator, which we will use to index into our accumulated weight list. There are many choices available (including Jython), but for our purposes here we'll just use the Java standard generator:

```
g_rng = java.util.Random(java.lang.System.currentTimeMillis())  
  
def randNum(i_min, i_max):  
    assert i_min <= i_max
```

```

    range = i_max - i_min + 1 # re-purposing "range" is legal in
Python
    assert range <= 0x7fffffff # because we're using
java.util.Random
    randnum = i_min + g_rng.nextInt(range)
    assert i_min <= randnum <= i_max
    return randnum

```

Here, we've constructed a random number generator, and seeded it with the time-of-day. (For test/simulation purposes, it is counterproductive to use secure random number generators, such as `java.security.SecureRandom`, or a secure seed source.)

Further, we've defined a `randNum()` function that takes minimum and maximum values as input, and returns a random number between them (inclusive of both endpoints).

Note: One advantage of using `java.util.Random` is that it's thread-safe, so we need construct only a single global generator. But that safety comes at the expense of some performance loss, especially if you are using the generator extensively, such as generating massive random file/object content. In that case, you may want to use faster, non-thread-safe generators, constructing one for each thread's private use.

## 7. Test Runner Class

We are now ready to define our `TestRunner` class:

```

class TestRunner:
    def __call__(self):
        opNum = randNum(0, g_WeightsAccMax_1)
        opType = None # flag for assertion below
        for i in range(g_WeightsAccLen):
            if opNum < g_WeightsAcc[i][1]:
                opType = g_WeightsAcc[i][0]
                break
        assert opType in g_Weights.keys()

        if opType=='CREATE': doCREATEtest()
        elif opType=='READ' : doREADtest()
        elif opType=='UPDATE': doUPDATetest()
        elif opType=='DELETE': doDELETetest()
        else : assert False

```

According to The Grinder framework, every worker thread calls `TestRunner.__call__()` in an infinite loop (until it terminates). In our case, for each run, each thread first chooses a random number, `opNum`, and then uses that random number to index into the accumulated weight list. (Well, it's not exactly "indexing" in the array or database access sense, but the idea is the same.) This results in the tag of an operation type, `opType`, to be called. The thread then maps the operation type tag to a test, and calls it.

## 8. Putting It All Together

Our example script is now complete, so we can run it.

Let's say we want to do 10,000 runs. In your `grinder.properties` file, set `grinder.threads=20`, `grinder.runs=500`. Then invoke `startAgent.sh`. You'll see 10,000 lines printed, each saying "Doing XXX test ...", where XXX is one of CREATE, READ, UPDATE, DELETE.

But did you get the weighted distribution of test cases you wanted? For that, you need to count various lines printed out by the test. In a Linux environment, you can do this conveniently by rerunning the test in a pipeline command as follows:

```
startAgent.sh | \
  awk '/^Doing /{count[$0]+=1} END{for (test in count) print test,
count[test]}'
```

A typical run of this command will produce results similar to the following:

```
Doing CREATE test ... 2006
Doing READ test ... 4045
Doing UPDATE test ... 2993
Doing DELETE test ... 956
```

Inspection of these numbers shows you are indeed running the distribution you desired.

## 9. Full Script Listing

```
import java.lang.System, java.util.Random

g_Weights = {
  'CREATE': 2,
  'READ' : 4,
  'UPDATE': 3,
  'DELETE': 1,
}

def doCREATETest():
  print 'Doing CREATE test ...'
def doREADtest():
  print 'Doing READ test ...'
def doUPDATETest():
  print 'Doing UPDATE test ...'
def doDELETETest():
  print 'Doing DELETE test ...'

def weightAccumulator(i_dict):
  keyList = i_dict.keys()
  keyList.sort() # sorting is optional - order coming-in doesn't
  matter, but determinism is kinda cool
  listAcc = []
  weightAcc = 0
  for key in keyList:
    weightAcc += i_dict[key]
    listAcc.append((key, weightAcc))
  return (listAcc, weightAcc) # order going-out does matter - hence
  "listAcc" instead of "dictAcc"

g_WeightsAcc, g_WeightsAccMax = weightAccumulator(g_Weights)
g_WeightsAccLen, g_WeightsAccMax_1 = len(g_WeightsAcc),
g_WeightsAccMax-1

g_rng = java.util.Random(java.lang.System.currentTimeMillis())

def randNum(i_min, i_max):
  assert i_min <= i_max
```

```
range = i_max - i_min + 1 # re-purposing "range" is legal in Python
assert range <= 0x7fffffff # because we're using java.util.Random
randnum = i_min + g_rng.nextInt(range)
assert i_min <= randnum <= i_max
return randnum

class TestRunner:
    def __call__(self):
        opNum = randNum(0, g_WeightsAccMax_1)
        opType = None # flag for assertion below
        for i in range(g_WeightsAccLen):
            if opNum < g_WeightsAcc[i][1]:
                opType = g_WeightsAcc[i][0]
                break
        assert opType in g_Weights.keys()

        if opType=='CREATE': doCREATEtest()
        elif opType=='READ' : doREADtest()
        elif opType=='UPDATE': doUPDATEtest()
        elif opType=='DELETE': doDELETETest()
        else : assert False
```