The Problem

We would like to automate the process of installing new servers to the system. We would like to choose a storage so that we can optimize the overall resouces of the system. A certrain capacity, IOPS, and throughput will be requested and we need to find the optimal storage to assign these resources.

Solution Idea

A simple solution but very efficient and scalable.

The idea is we need to find the storage that fits our request best without wasting any resources.

For example: If we have a 30/40 TB storage and a 20/40 TB storage and the request is asking for 10TB then we will assign it the storage with 30/40. But we have to apply this idea to all the fields including IOPS and Throughput.

Implementation

Here is an overview of the structure.

`compute.py extractor.py storages.json priority.json`

extractor.py is responsible for extracting the data of the existing storages and converting them into json format and output them inside the storages.json file.

Filtering the storages

compute.py will take the JSON array supplied by storages.json and filter it over multiple steps in order to find the best match.

First we loop over the all the existing storages to find the storages that have enough capacity/iops/throughput

```python
# Return all storages that have enough capacity, iops, and throughput.
def findAllViableStorages():
    viableStorages = []
    storages = loadStorageData()
    requestedCapacity = int(sys.argv[2])
    requestedIOPS = int(sys.argv[3])
    requestedThroughput = int(sys.argv[4])

    for i in range(len(storages)):
        if storages[i]["capacity"] >= requestedCapacity and
        storages[i]["iops"] >= requestedIOPS and
        storages[i]["throughput"] >= requestedThroughput:

            viableStorages.append(storages[i])
            increaseTotalAttributes(storages[i])

    return viableStorages
```

Figure 1: Filtering

Now once we have an array of all the viable storages we need to know which one fits the best. In order to do that we need to calculate a score for each storage and the lowest score will be the closest to the requested resources.

```
def calculateScore(storage):
    #Normalizing the requested data
    requestedCapacity = normalizeData(int(sys.argv[2]), totalCapacity)
    requestedIOPS = normalizeData(int(sys.argv[3]), totalIOPS)
    requestedThroughput = normalizeData(int(sys.argv[4]), totalThroughput)

    #Normalizing the storage data
    storageCapacity = normalizeData(storage["capacity"], totalCapacity)
    storageIOPS = normalizeData(storage["iops"], totalIOPS)
    storageThroughput = normalizeData(storage["throughput"], totalThroughput)

    #Getting the Multipliers from priority.json
    capacityMulti = getMultiplier("capacityMulti")
    IOPSMulti = getMultiplier("IOPSMulti")
    throughputMulti = getMultiplier("throughputMulti")

    score =  (storageCapacity - requestedCapacity * capacityMulti) +
             (storageIOPS - requestedIOPS * IOPSMulti) +
             (storageThroughput - requestedThroughput * throughputMulti)
    return score
```

Figure 2: Scoring

Normalizing the data

In order to create a score for each storage we need them to have the same units since they are different. We can do this by calculating the total Capacity, IOPS, and Throughput and dividing each of the resources by the total to get a value between 0 and 1.

```
# This returns a value between 0 and 1 to use in the scoring.
def normalizeData(value, total):
    return value/total
```

Figure 3: normalizing

Prioritising

If we wanted to focus more on optimizing capacity for example, we can head into the **priority.json** file and change the capacity multiplier to a higher value. Increasing this value will make the capacity affect the score less, which means we will get a lower score.

```
{
    "capacityMulti": 1,
    "IOPSMulti": 1,
    "throughputMulti": 1
}
```

Figure 4: priority JSON file

Choosing the best storage

After calculating the scores for each storage we simply go over the scores to find the lowest one and return the corresponding storage.

```python
# Find the best match from the storages
def findOptimalStorage():
    storages = findAllViableStorages()
    best_storage = {}
    best_score = float('inf') # Infinity represents a very large number
    # We need to pick the one with the least score in order to best utilize our resources.
    for storage in storages:
        tmp = calculateScore(storage)
        print("Score: " + str(tmp) + " with wwn: " + str(storage["wwn"]))
        if tmp < best_score:
            best_score = tmp
            best_storage = storage
    return best_storage
```

Figure 5: Saving the object with the lowest score

Usage

`python3 compute.py storages.json <requestedCapacity> <requestedIOPS> <requestedThroughput>`

Other ideas

- If we only care about optimizing one resource like Capacity for example we can simply sort all the storages and binary search for the requested capacity and return the one that is equal to it or exactly above it.
- Use Machine learning and Data analysis to find the optimal storage using Historic Data.

Those scripts can be a good base for a machine learning implementation. The machine learning could possibly only make changes to the **priority.json** file and change the multipliers to pick the perfect storage.