Scenarios

- Existent imbalanced data/gas spent on shards
- Migration of data from one shard to another for other technical/business reasons
- New data from off-chain sources / Eth1

For the purpose of solving this problem we will use dType (dType (Decentralized Type System) on Ethereum 2.0) and the fact that it now has a count

function for stored items. The count

function is not ideal, because it does not measure the actual storage cost, but it gives enough approximation for the current purpose.

Problem Statement

Given a number of shards shard_count

, each loaded with a certain shard_load

and a number of dType storage contracts dtype_count

, each with a certain dtype_load

, we need to find a way to balance the loads across shards.

The result is a list of shards, each with a list of dtype IDs that should be added to that shard and the final data load of that shard.

Solution

The Python code for this solution is: (it can also be read at https://github.com/pipeos-one/dType/blob/f28bc63377f0565b1809c4dd4842242ce25dbd73/docs/research/Data_Load_Balancing_of_Shards.ipynb)

import random

shard_count = 20 dtype_count = 50

Increase average_coef if there are not enough shards for all dtypes

average_coef = 1.3

max_shard_load = 400 max_dtype_load = 2000

Initialize random load values for shards and dtypes

shard_loads_initial = list(enumerate([random.randrange(i, max_shard_load) for i in range(shard_count)])) dtype_loads_initial = list(enumerate([random.randrange(i, max_dtype_load) for i in range(dtype_count)])) shards = [[] for i in range(shard_count)]

next_index_s = 0 next_index_dt = 0 last_index_dt = len(dtype_loads_initial) - 1 last_index_s = len(shard_loads_initial) - 1

Sort loads: ascending for shards, descending for dtypes

shard_loads = sorted(shard_loads_initial, key=lambda tup: tup[1]) dtype_loads = sorted(dtype_loads_initial, key=lambda tup: tup[1], reverse=True)

Calculate average count per shard

average_load_shard = (sum(i[1] for i in dtype_loads) + sum(i[1] for i in shard_loads)) / shard_count average_load_shard *=

average_coef print('average_load_shard', average_load_shard)

Move heavier than average dtypes on the least heaviest shards

for i, dload in dtype_loads: if dload >= average_load_shard: shards[next_index_s].append(i) next_index_s += 1 next_index_dt += 1

Pair heaviest dtypes with lightest shards

and add as many light dtypes on top, as possible

for i, dload in dtype_loads[next_index_dt:]: if last_index_s < next_index_s: print('Needs more shards. Increase average_coef'); break

```
# Add the next heaviest dtype to the next lightest shard
shards[next_index_s].append(i)

# Add as many light dtypes as the average_load_shard permits
load = shard_loads[next_index_s][1] + dload + dtype_loads[last_index_dt][1]
while last_index_dt > next_index_dt and load <= average_load_shard:
    shards[next_index_s].append(dtype_loads[last_index_dt][0])
    last_index_dt -= 1
    load += dtype_loads[last_index_dt][1]

next_index_s += 1
next_index_dt += 1
if next_index_dt > last_index_dt:
    break
```

print('(shard_index, shard_load, dtype_indexes)')

final_shards = [(shard_loads[x][0], sum([dtype_loads_initial[dtype_index][1] for dtype_index in shards[x]]), shards[x]) for x, _ in enumerate(shards)]

print('final_shards', final_shards)

Conclusions

We may need more data about gas/storage costs for this algorithm to be effective. Extended usage of dType will also improve the outcome.