Rollup diff compression

Blagoj Dimovski, Barry WhiteHat
ZKTeam, Ethereum foundation

Contents

1 Introduction

Airdrops on the Ethereum network are increasingly popular and common. The structure and complexity between different airdrops scenarios can be vastly different. Although the cost for each airdrop scenario can be different depending on the complexity of the airdrop, the costs for each scenario on L1 are high and can be optimized. This is mainly due to the amount of data that needs to be stored on the blockchain. The store operation is one of the most expensive operations in terms of gas cost, as described in the Ethereum yellow paper[2]. The most promissing L2 scaling solutions at the moment, such as Optimistic and ZK rollups are a way to reduce this costs. Their main limitation is the amount of data that they can publish on L1. In this research we look for ways to optimize the amount of data that needs to be published on L1, by looking into different data compression strategies for the most complex airdrop scenario.

The main benefit of the rollup solutions is that the data processing and storage can be done offchain and only the relevant state changes can be published on the L1 chain as a proof for the changes on the L2. Compressing the state changes (diff compression) is desirable because of the rollup limitations and the gas cost reduction. What we're looking for is to find an optimal strategy for compressing the data, so that the storage costs are minimized for the rollup and the published data on L1 is also minimized.

Because of the difference in the airdrop scenarios, there is no "One size fits all" solution. The airdrop distributions can be different in size and form. Using the same compression strategy might give good results for one type of airdrop and bad for other. In this paper we've started with experimenting with the most complex airdrop scenario: multiple distribution iterative airdrop, with new users per distribution and different (non-grouped) user balances. We've used the data from the Reddit r/FortNiteBR airdrop, which currently consists of 10 distributions.

The goal for this paper is to find the optimal strategies for different airdrop scenarios, by comparing the gas costs for different strategies based on previous data. After this we propose an implementation which will utilize the most appropriate strategy based on the airdrop scenario. The strategy is determined by the airdrop structure and it's data. At the end we will test the implementation and present the real gas costs for different airdrop scenarios. This research is extension of the work from Barry WhiteHat[1].

2 Dataset

2.1 Description

The data used for the initial experiments in this research is the data from the Reddit's r/FortNiteBR community points airdrop distribution. The community points are a way to award the reddit users for their participation in a specific subreddit (community). Community points as of now could be used for different purposes at the related subbreddit they're issued for, such as: purchasing perks, tipping and voting. The community points are owned entirely by the users and are distributed through the Ethereum Rinkeby testnet. The r/FortNiteBR community points are called BRICKS. The airdrop for this subreddit consists of 10 distributions as of now. Each airdrop distribution is conducted once a month. List of each user's karma points is published each 4 weeks on the specific subreddit, community votes for the list's validity and once everything is approved, the Distribution contract on the Ethereum Rinkeby testnet is "fed" with the distribution data. This is triggered by Reddit. Reddit also issues signed claims for each user. Each user can then redeem their claim onchain.

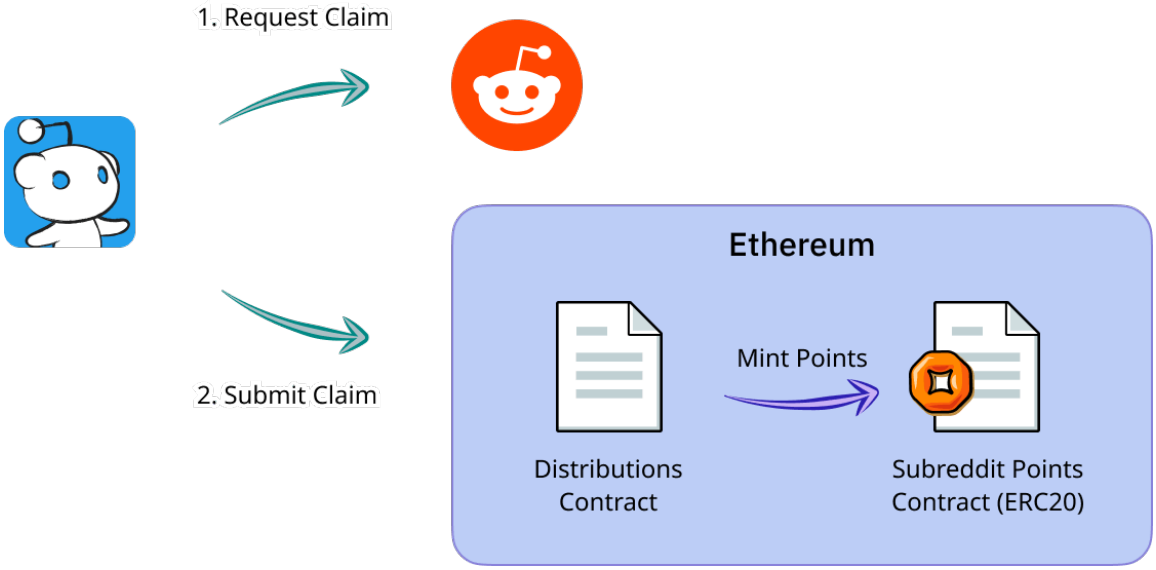

Figure 1: Subreddit points distribution

More details about the community points can be found here on Reddit[1]. Reddit looks

[1]Community points description: https://www.reddit.com/community-points/

for migrating the community points to Ethereum mainnet in the near future. The gas spent would matter much more from financial standpoint and optimisations are necessary.

2.2 Overview

The data consists of 10 distributions. The total amount of airdrops (we refer to an airdrop as a single record containing the user address and amount) for all the distributions is: 220274. The number of unique users for all airdrops is: 102069. Number of users that appear in at least 2 airdrop distributions (repeating users), is: 42944, or 42.07% from the unique users.

More details about the data are displayed in table 1 and appendix A.

Id	Airdrops	Unique users	Repeating users	Percent repeating
1	25393	25393	0	0
2	30448	18131	12317	40.45%
3	23883	10442	13441	56.27%
4	27593	11752	15841	57.40%
5	22247	7749	14498	65.16%
6	18439	5983	12456	67.55%
7	16985	4966	12019	70.76%
8	22979	7928	15051	65.49%
9	16377	4950	11427	69.77%
10	15930	4775	11155	70.02%
Total	220274	102069	42944	42.07%

Table 1: r/FortNiteBR airdrop distributions overview

3 Initial cost assumptions

For this research and the data we're using, we've made the following assumptions:

- Ultra efficient rollup

  We're using an ultra efficient rollup and the gas cost is just the data availability cost. The data availability availability cost can be viewed as:

1. $gasCostBitCalldata = 2$
2. $gasCostBitCommitment = 5$

which leads to $totalGasPerBit = 7$.

- Data size per airdrop

  We can represent the total number of unique users from the distributions until now (102069) with 17 bits (max amount of users that can be represented is 131072). Note that we're not trying to fit all possible users eligible for the reddit airdrop - the subreddit is quite big (1.5 million users). We're making assumptions based on the available data only. From here the user ID can be represented with 17 bits, $userIdBits = 17$.

The karma per user can be represented with 16 (max amount of karma that can be represented is 65536) bits. The maximum karma found in all the distributions is 54981. From here the user balance can be represented with 16 bits, $userBalanceBits = 16$.

This leads to:

$$totalGasCostPerAirdrop = (userIdBits + userBalanceBits) * totalGasCostPerBit$$

$$= (17 + 16) * 7 = 231$$

which can also be formulated as:

$$totalGasCostPerAirdrop = gasCostUserId + gasCostUserBalance$$

$$gasCostUserId = userIdBits * totalGasCostPerBit = 17 * 7 = 119$$

$$gasCostUserBalance = userBalanceBits * totalGasPerBit = 16 * 7 = 112$$

3.1 Naive approach costs

The 'naive' approach consists of publishing the data as it is, without any compression optimizations. This means one record for each airdrop is published in the format:

$$userId : userBalance$$

The gas cost for this approach is easy to estimate, for one distribution it is:

$$gasCostPerDistNaiveX = numAirdropsDist * totalGasCostPerAirdrop$$

For all distributions the naive gas cost is:

$$gasCostTotalNaive = numDists * gasCostPerDistNaive(1..X..N)$$

The table 2 displays the naive the gas cost for the r/FortNiteBR airdrop distribution data. All the following comparisons will be based of this data.

Id	Airdrops	Gas cost	Cumulative gas cost
1	25393	5865783	5865783
2	30448	7033488	12899271
3	23883	5516973	18416244
4	27593	6373983	24790227
5	22247	5139057	29929284
6	18439	4259409	34188693
7	16985	3923535	38112228
8	22979	5308149	43420377
9	16377	3783087	47203464
10	15930	3679830	50883294
Total	220274	50883294	50883294

Table 2: Naive approach gas costs

4 Gas saving strategies

4.1 Approach 1: Grouping the data by amounts, per distribution

Analyzing the data, the most intuitive approach for data compression is grouping the users by their amounts. From the distribution we can see that many of the amounts are repeating. For example for the first month, from total of 25393 airdrops, only 1409 amounts were unique. We can see from the data that in each distribution, the amounts from 1 to 10 are always most dominant (please refer to the charts in appendices A and B).

We can determine the gas cost for this approach by computing the gas cost for the unique balances and adding that with the gas cost for all user ids.

The gas cost per distribution would then be:

$$gasCostDist = gasCostUserIdsDist + gasCostUniqueBalancesDist$$

$$gasCostUserIdsDist = numAirdropsDist * gasCostUserId$$

$$gasCostUniqueBalancesDist = numUniqueBalancesDist * gasCostUserBalance$$

More intuitively this would be described as grouping the users with the same amount in the same group. For example if the users with ids: user1, user2 and user3 have the same amount of 1 for some distribution, we can group them in single group g1 and represent the group as g1 = user1,user2,user3:1. This way the gas that needs to be paid for the group will be:

$$gasCostUserId1 + gasCostUserId2 + gasCostUserId3 + gasCostUserBalance1$$

We're paying the $gasCostUserBalance$ for amount1 only once, where in the naive approach we would pay this cost 3 times.

In this approach we concentrate on the data per distribution, we don't make data assumptions or use the data from the previous distributions.

Gas costs for this approach and gas cost savings compared to the naive approach are displayed in 3.

Id	Airdrops	Unique amounts	Gas compressed	Gas naive	Gas saved	Gas saved %
1	25393	1409	3179575	5865783	2686208	45.79%
2	30448	1527	3794336	7033488	3239152	46.05%
3	23883	1408	2999773	5516973	2517200	45.62%
4	27593	1567	3459071	6373983	2914912	45.73%
5	22247	1387	2802737	5139057	2336320	45.46%
6	18439	1127	2320465	4259409	1938944	45.52%
7	16985	1128	2147551	3923535	1775984	45.26%
8	22979	1286	2878533	5308149	2429616	45.77%
9	16377	1003	2061199	3783087	1721888	45.51%
10	15930	1000	2007670	3679830	1672160	45.44%
Total	220274	/	27650910	50883294	23232384	45.65%

Table 3: Approach 1 Savings

4.2 Approach 2: Making groups of users that repeat across all distributions (permutations)

The second approach we took is making groups of users that repeat across all distributions. This approach serves just for data analysis and comparison with the other approaches and can't be implemented in practice for retrospective airdrop (such as the one that we're analysing), because it requires upfront knowledge of the distributions for the next months. The main idea for this approach is finding groups of repeating users across the whole dataset and reusing the groups. In contrast to grouping by same amount with the previous approach, now we're grouping the users by their appearance in some distribution. Each group represents a permutation of user appearances across all the distributions, from the repeating users. Each user can be only part of one group.

4.2.1 Intuition behind the idea

Intuitively we can describe the grouping (permuting the users) as follows: Find all the users that appeared in the first and second distribution only. Find all the users that appeared in the first, second and third distribution only. Find all the users that appeared in the

first and the third distribution only, and so on. This process ensures that all of the groups contain unique users. For 10 distributions, the number of groups is 999.

The $groupId$ for each group can be represented as string of ones and zeroes. The length of the string is equal to the number of distributions and for each distribution we have either one or zero in the string. One means that the user was part of the distribution, while zero means that the user was not part of the distribution.

The group of users that were part of the first and second distribution only can be represented as:

$$groupId = 1100000000$$

where the group of users that were part of all distributions can be represented as:

$$groupId = 1111111111$$

For each repeating user we can find the group that he is part of, by looking at which distributions he appeared in. This way we can form the $groupId$ for each user. To find how many users each group contains, we simply need to find how many users have the same $groupId$.

For example if the user with $userId = user1$ was part of the first and the third distribution only, his $groupId$ would be 1010000000. Then if the user with $userId = user2$ was part of the first and the third distribution only he would have the same $groupId$. The number of users with $groupId = 1010000000$, $numUsers\_1010000000$ would be 2.

4.2.2 Costs

The gas cost for the first distribution is the same as in the naive approach, because there are no previous distributions and all of the data needs to be published.

For the rest of the distributions by using the groups, we can determine the gas cost for each distribution as:

1. For each group which users are part of the previous distributions (existing users): Cost of permuting the distribution of where the group's users from the current distribution appeared previously ($prevDistPermCost$) + the gas cost for publishing the $userBalance$ amounts for each user in the group ($groupUserBalances$). The

prevDistPermCost is equal to the number of users in the previous distribution multiplied by *totalGasPerBit*. This is because we only need 1 bit per user for the permutation. The gas cost for publishing the *userBalance* for each user in the group equals to: number of users in the group multiplied by *gasCostUserBalance*.

2. The gas cost for the new users that appear in this distribution, which is the number of new users in the distribution multiplied by *totalGasCostPerAirdrop*.

Based on the current data, this approach results in negative savings. The number of groups is large for the 10 distributions of data - 999, and the number of users per group is small relative to users per distribution. The member count of the group of users that appeared in all distributions is 1295. However, there are many groups with member count less than 10, even there are groups with 1 member only. This results in a lot of gas costs per permutation (*prevDistPermCost*) for all the groups and this approach is worse than the naive and the 'Grouping the data by amounts, per distribution' approach.

The gas costs for this approach are displayed in the following table:

Id	Airdrops	Repeating users	Gas compressed	Gas naive	Gas saved	Gas saved %
1	25393	0	5865783	5865783	0	0
2	30448	12317	49731159	7033488	-42697671	-607.06%
3	23883	13441	78095742	5516973	-72578769	-1315.55%
4	27593	15841	84119364	6373983	-77745381	-1219.72%
5	22247	14498	91186323	5139057	-86047266	-1674.37%
6	18439	12456	85716965	4259409	-81457556	-1912.41%
7	16985	12019	75525198	3923535	-71601663	-1824.92%
8	22979	15051	70143077	5308149	-64834928	-1221.42%
9	16377	11427	75497854	3783087	-71714767	-1895.66%
10	15930	11155	66969380	3679830	-63289550	-1719.90%
Total	220274	42944	27650910	50883294	-631967551	-1241.99%

Table 4: Approach 2 Savings

4.3 Approach 3: Adding only new unique users and referencing the old ones, with previous data only

This approach is very similar to the previous one, with one important distinction: the calculations and are based on the previous distributions only. Looking from implementation perspective, it's very possible to implement this approach in practice because all of the data is already available.

The main idea behind this approach is for the repeating users to be referenced to the previous distribution and the data for new users to be fully published. For the first distribution all of the data will need to be published ($userId : userBalance$ pairs). For the next distributions, for the repeating users we will only need to publish the $userBalance$ for each repeating user and use one bit to reference the $userId$. For the new users both the $userId$ and the $userBalance$ will need to be published. Additionally one bit will be needed for the users that were part of the previous distributions and not the current one. This is because we keep the users sorted and need to mark off the non-receiving addresses as well. This means that for each distribution we will need to allocate at least one bit for the users that appeared in all of the previous distributions. This is the main downside of this approach, because of the data compounding with each round. Looked from a single distribution perspective, sometimes it can lead to increased costs compared to naive approach - for example if the amount of users from the previous distributions is big and the current distribution contains only a few airdrops.

4.3.1 Costs

The gas cost for the first distribution is the same as in the naive approach, because there are no previous distributions and all of the data needs to be published.

For the rest of the distributions by using the groups, we can determine the gas cost for each distribution as:

1. For the new users in the current distribution (new users): The full data publishing cost for airdrop, $totalGasCostPerAirdrop$ multiplied by the number of new users.

$$newUsersGasCost = totalGasCostPerAirdrop * numNewUsers$$

2. For the repeating users: For each repeating user: the cost for referencing the $userId$ for each repeating user ($prevUserRefCost$) which is 1 bit, plus the cost for the $userBalance$ ($gasCostUserBalance$), multiplied by the number of repeating users for the distribution.

$$repeatingUsersGasCost = (prevUserRefCost * gasCostUserBalance)$$
$$*numRepeatingUsers$$

3. For the previous users which are not part of the current distribution: The cost for marking off each previous user which is not part of the current distribution $prevUserMarkCost$, which is 1 bit. This cost can be viewed as a placeholder that implies that this user is not part of the current distribution.

$$previousUsersGasCost = prevUserMarkCost * numPrevUsers$$

The total cost for each distribution (other than the first one) can then be calculated as:

$$gasCostDist = newUsersGasCost + repeatingUsersGasCost + previousUsersGasCost$$

The gas costs for this approach are displayed in table 5.

Id	Airdrops	Repeating	G. compressed	G. naive	G. unique users	G. repeating	G. referencing	G. saved	G. saved %
1	25393	0	5865783	5865783	0	0	0	0	0%
2	30448	12317	5745516	7033488	4188261	1465723	91532	1287972	18.31%
3	23883	13441	4222162	5516973	2412102	1599479	210581	1294811	23.46%
4	27593	15841	4866666	6373983	2714712	1885079	266875	1507317	23.64%
5	22247	14498	3873821	5139057	1790019	1725262	358540	1265236	24.62%
6	18439	12456	3291414	4259409	1382073	1482264	427077	967995	22.72%
7	16985	12019	3049424	3923535	1147146	1430261	472017	874111	22.27%
8	22979	15051	4107992	5308149	1831368	1791069	485555	1200157	22.60%
9	16377	11427	3069682	3783087	1143450	1359813	566419	713405	18.85%
10	15930	11155	3033443	3679830	1103025	1327445	602973	646387	17.56%
Total	220274	42944	41125903	50883294	23577939	14066395	3481569	9757391	19.17%

Table 5: Approach 3 Savings

4.4 Approach 4: User amount grouping per distribution

The main idea behind this approach is to group the users in groups with similar amounts. It is similar to the approach 1, but the main difference is that the user amounts are grouped

into one first. There are two sub-approaches for this compression: grouping the users amount by value similarity, grouping the user amounts in fixed number of equal buckets. Both of these approaches are implementable into practice, but an important assumption is made about the amounts. By grouping them, the value of each amount is given less importance and this might be problematic for some airdrops. If the accuracy of each airdrop is highly important and the airdropped amounts must be 100% accurate for each user, this strategy might not be desired. This concern might be mitigated in some scenarios, by parameterizing this approach - introducing a parameter called $groupingRange$. For the approach 4a it represents the maximum distance difference between two numbers in a group. For the approach 4b it represents the number of amounts that need to be grouped together.

4.4.1 Approach 4a Grouping by value similarity

From the data we can observe that the value difference between the largest and the smallest amount is big. We might assume that the airdropped amounts for each user should not be 100% accurate. This makes sense if the airdrop distribution cost is larger than the value lost from compressing the data in this way. For example in some systems and scenarios, airdropping amount X to a certain user is far more financially expensive than the value amount X can have in the system. For scenarios where this kinds of concerns are not important, this approach is perfectly valid for saving on gas costs. For the data that we're analyzing we've chosen few parameters for the $groupingRange$ parameter:

$$1, 2, 5, 10, 15, 20, 50$$

We create buckets by applying a floor function to the amounts in given range, and assign the result of the floor function to the users that were assigned to the original amounts (inputs in the floor function). After that we group the users by the same amounts (the same process as in approach 1).

4.4.2 Approach 4a Gas costs

The gas costs can be calculated as:

$$gasCostDist = gasCostUserIdsDist + gasCostUniqueGroupedBalancesDist$$

$$gasCostUserIdsPerDist = numAirdropsDist * gasCostUserId$$

$$gasCostUniqueBalancesPerDist = numUniqueGroupedBalancesDist$$

$$*gasCostUserBalance$$

The table 6 displays the gas costs and savings by using this approach, for each value of the *groupingRange* parameter defined above:

gR value	Num. groups	G. compressed (raw)	G. compressed grouping	G. naive	G. saved over naive	G. saved % over naive
1	8742	27650910	27191710	5865783	23691584	46.56%
2	6948	27650910	26990782	5865783	23892512	46.95%
5	4662	27650910	26734750	5865783	24148544	47.45%
10	3225	27650910	26573806	5865783	24309488	47.77%
15	2591	27650910	26502798	5865783	24380496	47.91%
20	2200	27650910	26459006	5865783	24424288	48.00%
50	1235	27650910	26350926	5865783	24532368	48.21%

Table 6: Approach 4a Savings

4.4.3 Approach 4b Grouping by number of amounts

For this approach we also apply a floor function to the amounts, but instead using a given range we group the amounts into groups of *groupingRange* (gR) values.

For this approach the values of the *groupingRange* that we chose are:

$$2, 5, 10, 20, 50, 100$$

Each group contain the same amount of values, where the last group might contain less than *groupingRange* values, if *groupingRange* is not divisible by the number of airdrops in the distribution.

The *groupingRange* parameter represents the number of values in a group. The number of groups/grouped unique amounts can be calculated as:

$$uniqueGroupedBalancesDist = \begin{cases} \frac{numAirdropsDist}{gR}, & numAirdropsDist \bmod gR = 0 \\ \frac{numAirdropsDist}{gR} + 1, & numAirdropsDis \bmod gR \neq 0 \end{cases}$$

After applying the floor function we group the users by the same amounts (the same process as in approach 1).

Approach 4b gas costs

The gas costs can be calculated as:

$$gasCostDist = gasCostUserIdsDist + gasCostUniqueGroupedBalancesDist$$

$$gasCostUserIdsPerDist = numAirdropsDist * gasCostUserId$$

$$gasCostUniqueBalancesPerDist = numUniqueGroupedBalancesDist$$

$$*gasCostUserBalance$$

The following table displays the gas costs and savings by using this approach, for each value of the $groupingRange$ (gR) parameter defined above:

gR value	Num. groups	G. compressed (raw)	G. compressed grouping	G. cost naive	G. saved over naive	G. saved percent over naive
2	110141	27650910	38548398	5865783	12334896	24.24%
5	44058	27650910	31147102	5865783	19736192	38.78%
10	22031	27650910	28680078	5865783	22203216	43.63%
20	11018	27650910	27446622	5865783	23436672	46.05%
50	4408	27650910	26706302	5865783	24423616	47.51%
100	2206	27650910	26459678	5865783	24423616	47.99%

Table 7: Approach 4b Savings

4.5 Approach 5: Adding only new unique users and referencing the old ones, balance is represented as balance diff from the previous state

This approach is based on the approach 3, but here instead of publishing the full user balance for the repeating users in the distributions different than the first one, we can just publish the difference from the previous user amount. The cost for the difference in the amount from previous state can be calculated as: $userBalanceDiffBits + 1$. We need one bit for the sign, so that we know if the difference is positive or negative from the previous balance. To determine the value for $userBalanceDiffBits$, we can use the number of bits needed to represent the largest difference in the current dataset. If $userBalanceDiffBits + 1 < userBalanceBits$, this approach will give better results than the approach 3.

The worst case scenario for $userBalanceDiffBits$ is if this value is equal to the $userBalanceBits$. One scenario where this can happen is if the user was assigned the maximum balance value (global maximum for the dataset) in a previous distribution and the minimum balance value (global minimum for the dataset) in the current distribution.

Other thing we can do to optimize the costs is to introduce one more bit which will represent if the published balance is the original balance for the user within this distribution or the difference related to the state from the previous distributions. This might be more cost efficient solution if $userBalanceDiffBits = userBalanceBits$ or $userBalanceDiffBits + 1 = userBalanceBits$. This is true because the value of the amount that we will be publishing (original new balance or difference from the previous state) will always be less than or equal to $\frac{maxUserBalance}{2}$. If the difference is bigger than this number, it means it is more cost effective to publish the new original balance value, as the new original balance is less than the difference. If the opposite is true then the difference value is smaller than the original new balance.

Let's see what are the costs for these approaches applied on the r/FortNiteBR dataset.

4.5.1 Approach 5a: Balance difference

Gas cost for balance difference:

$$userBalanceDiffCost = (userBalanceDiffBits + 1) * totalGasPerBit$$

To determine the value for $userBalanceDiffBits$ we need to find the absolute value of the maximum difference of the balances from the repeating users and find the minimum number of bits needed to represent that value. First we need to obtain all the balances for each distribution for each user. Then we calculate the balance difference between each consecutive distribution that the user was part of for each user. After that we obtain the max balance difference across all users. For the r/FortniteBR airdrop dataset, this value is 38554, which needs 16 bits for representation. This is the same amount needed to represent the original balances.

From here we can determine that this approach is more expensive over the approach 3, because $userBalanceDiffBits + 1 > userBalanceBits$.

$userBalanceDiffBits$ = number of bits to represent the max difference in value for repeating user.

4.5.2 Approach 5b: Balance difference or original balance

Gas cost for new balance or balance difference:

$$userBalanceCost = (diffOrBalanceBits + 2) * totalGasPerBit$$

To determine the value for $diffOrBalanceBits$ we need to find the maximum user balance across all distributions and divide it by 2. After that we need to determine the number of bits to represent this value.

The maximum user balance across all distributions is: 54981. Half of this amount is 27490 (rounded). Number of bits needed to represent this value is 15.

This approach is also more expensive than the approach 3 because $diffOrBalanceBits+2 > userBalanceBits$.

For the dataset that we're exploring, both of the previous approaches don't make sense because of the nature of the data. If we make more assumptions and alter the dataset - such as ceiling and bucketing the user balances these approaches might give better results.

5 Summary of the current progress

So far the best results for the r/FortniteBR airdrop were obtained by approach 1 (Grouping data by amounts, per distribution). This approach is simplest in theory but returns the best results because of the removed dependencies of the underlying airdrop distribution (e.g. number of unique and repeated users). The second best approach implementable in practice is the approach 3 (Adding only new unique users and referencing the old ones). It is worse than the approach 1, but depending on the further data distributions it might give better results. The approach 4 gives the best results overall, but it is not implementable in practice as we make assumptions on the airdrop distribution process itself. The gas costs for all approaches is displayed at the end of Appendix B.

5.1 Things to do

There are few other approaches that needs to be explored which rely on the idea of combining groups made by balance grouping and address grouping. Theoretically from the approach 3, this approaches won't give better results than approach 1 because of the current data distribution. But what is important to have is the actual numbers and also more distribution data. To obtain more insights on the current approaches more comparisons can be done by using different airdrop data. The r/CryptoCurrency data is good fit for this. Also the data can be partitioned in multiple datasets and the current algorithms can be run on the different datasets - this will allow us to see how the algorithms that rely on previous data perform in a different environment.

Appendices

A Data charts

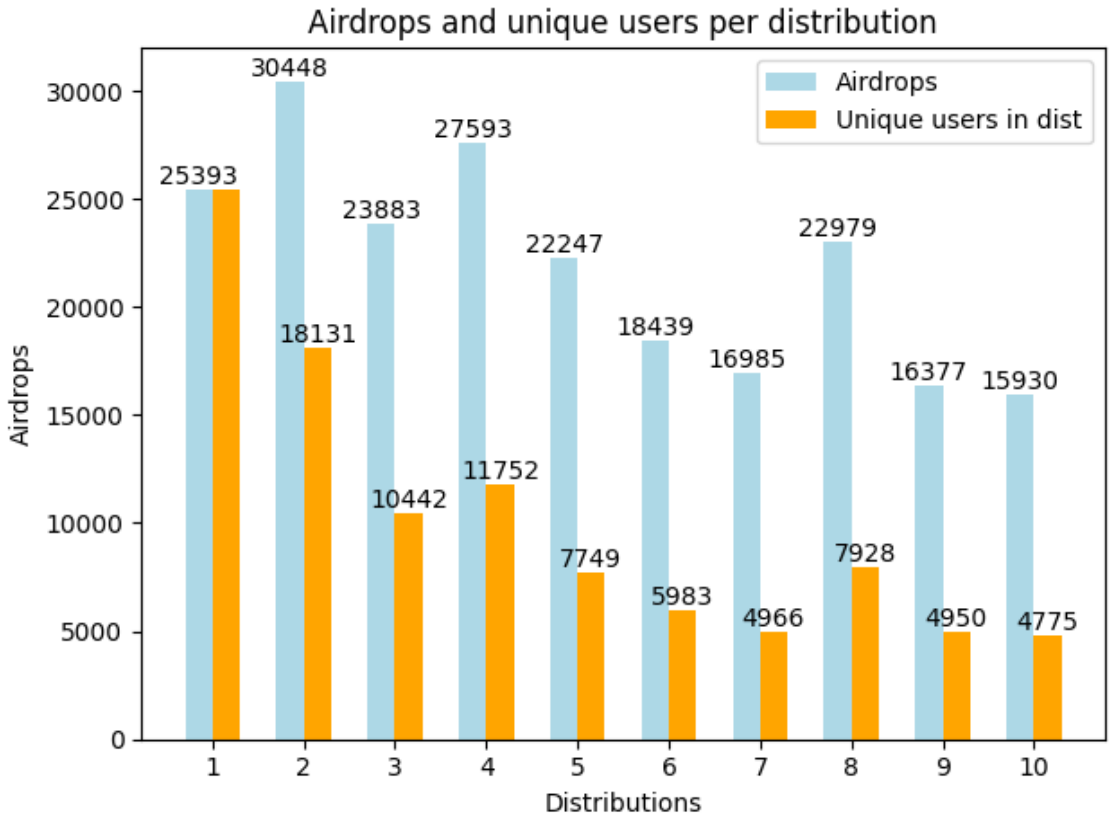

Figure 2: Airdrops and unique users per distribution

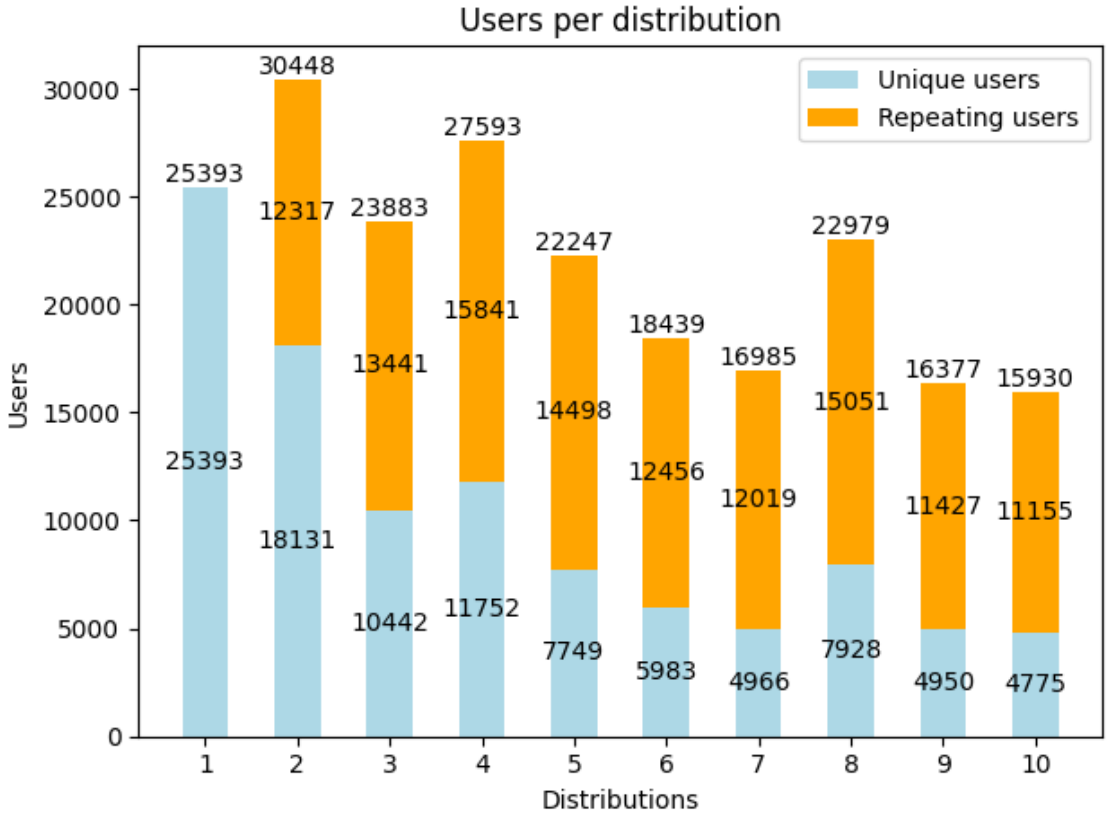

Figure 3: Repeating users per distribution

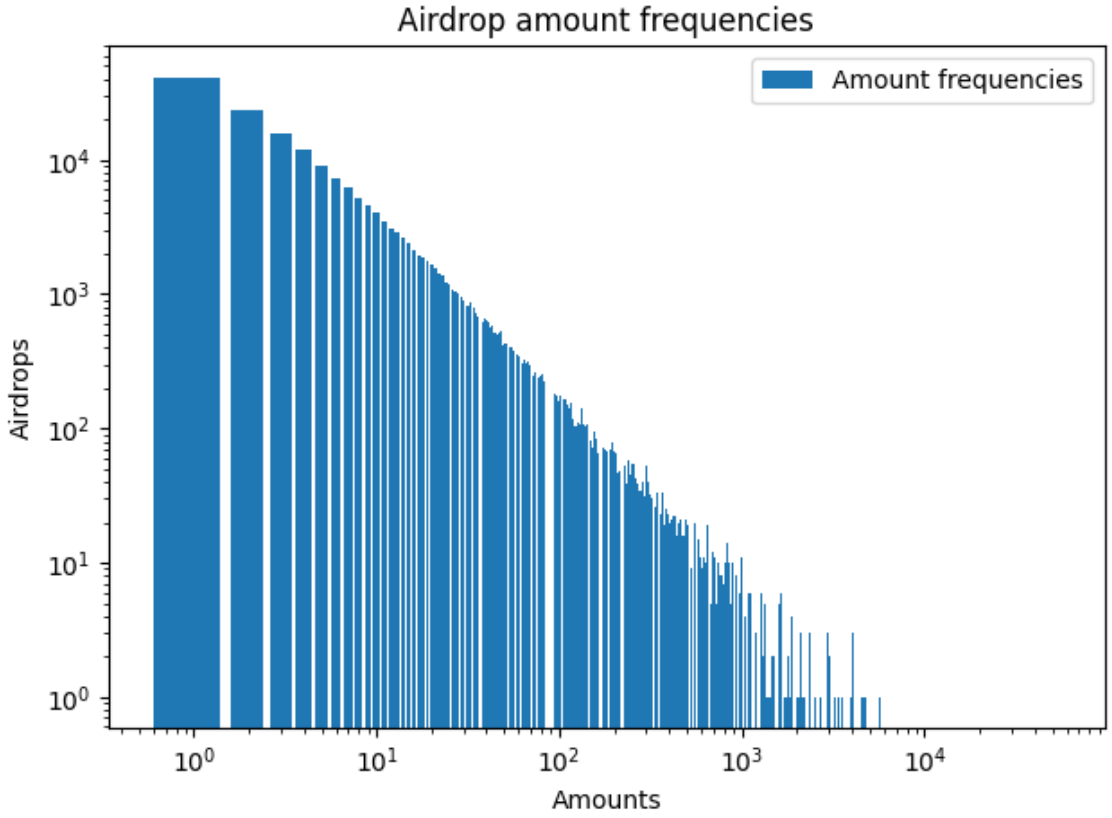

Figure 4: Amount frequencies

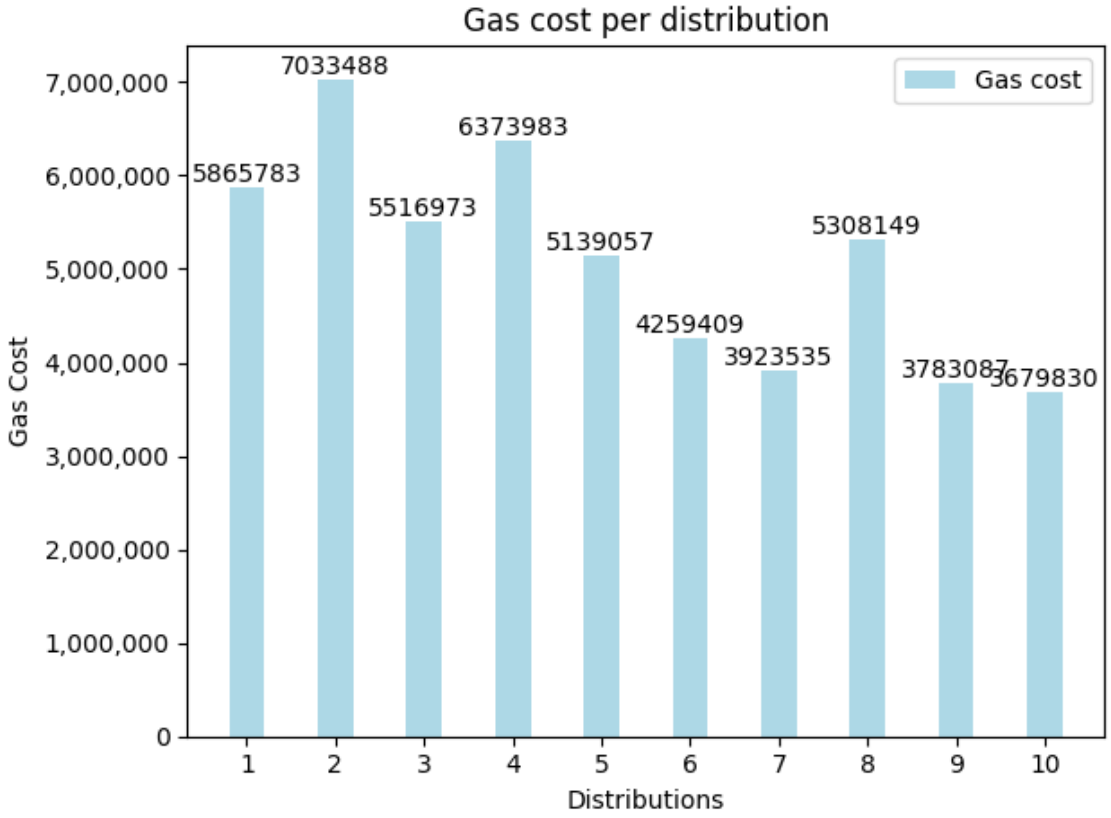

Figure 5: Gas costs per distribution

B Gas cost comparison for different compression approaches

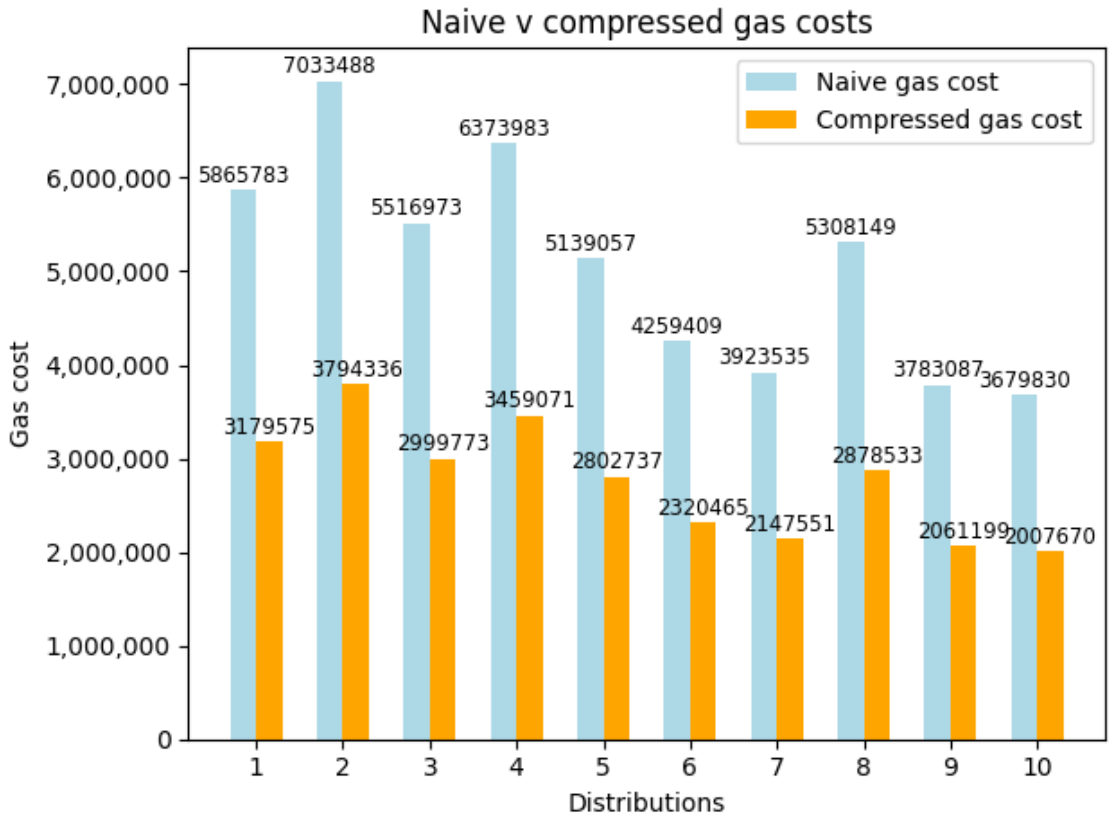

Figure 6: Gas costs naive vs compressed (approach 1)

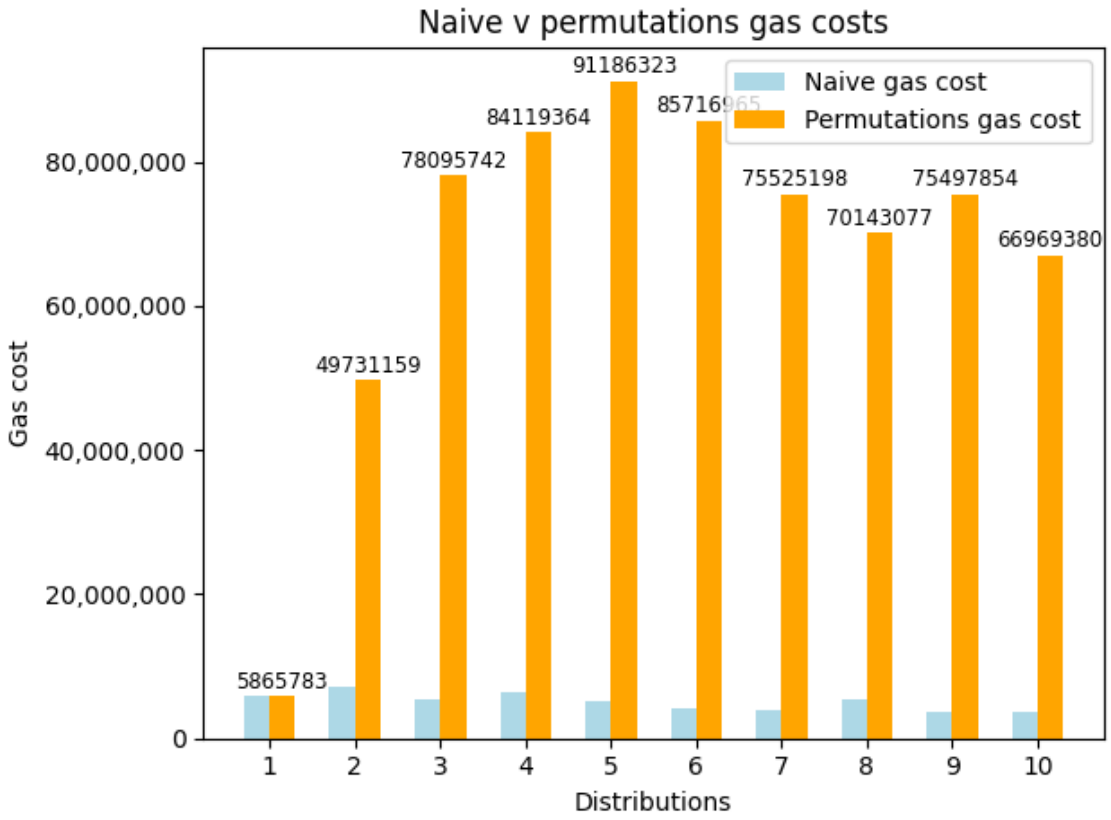

Figure 7: Gas costs naive vs compressed (approach 2)

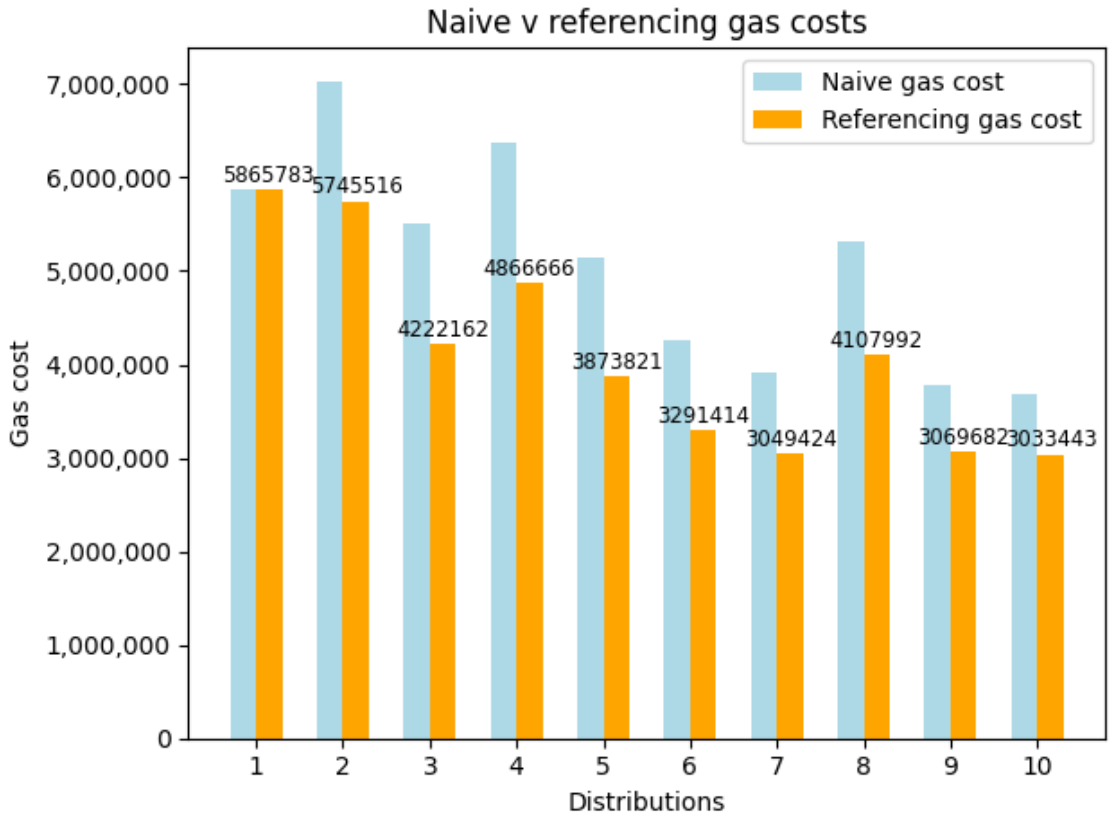

Figure 8: Gas costs naive vs referencing (approach 3)

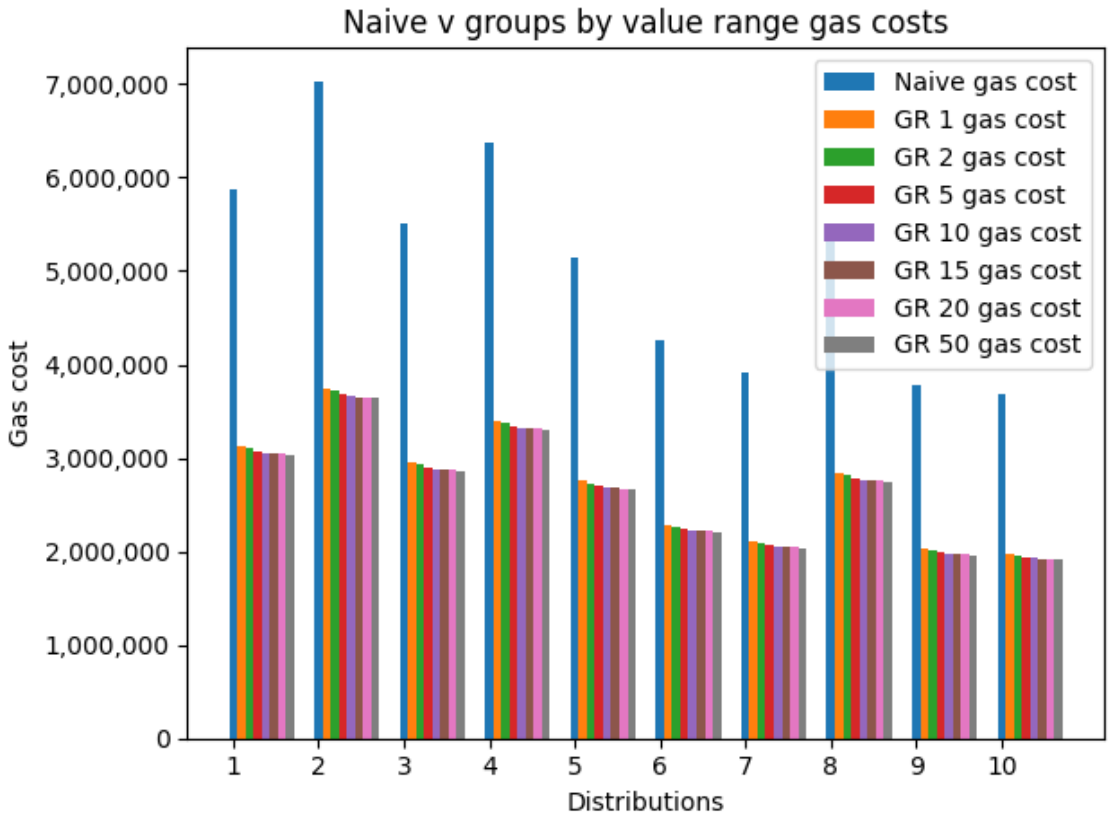

Figure 9: Gas costs naive vs grouping by value similarity (approach 4a)

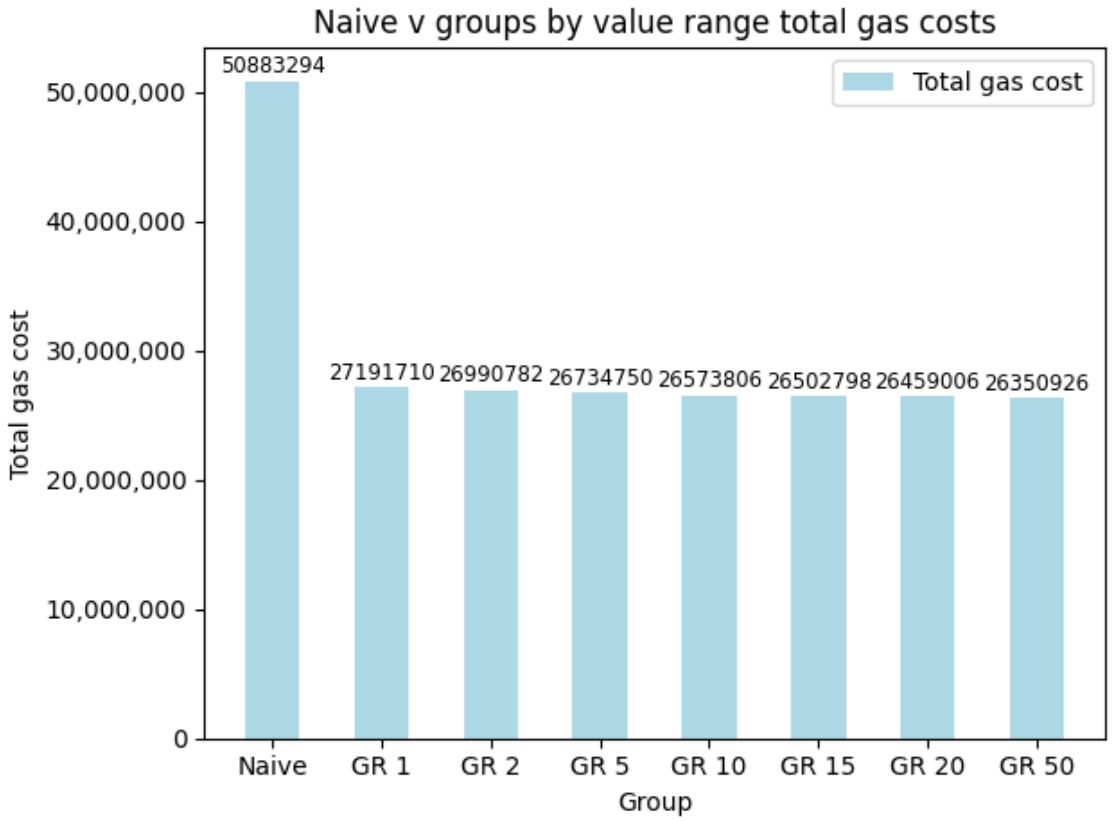

Figure 10: Gas costs naive vs grouping by value similarity total (approach 4a)

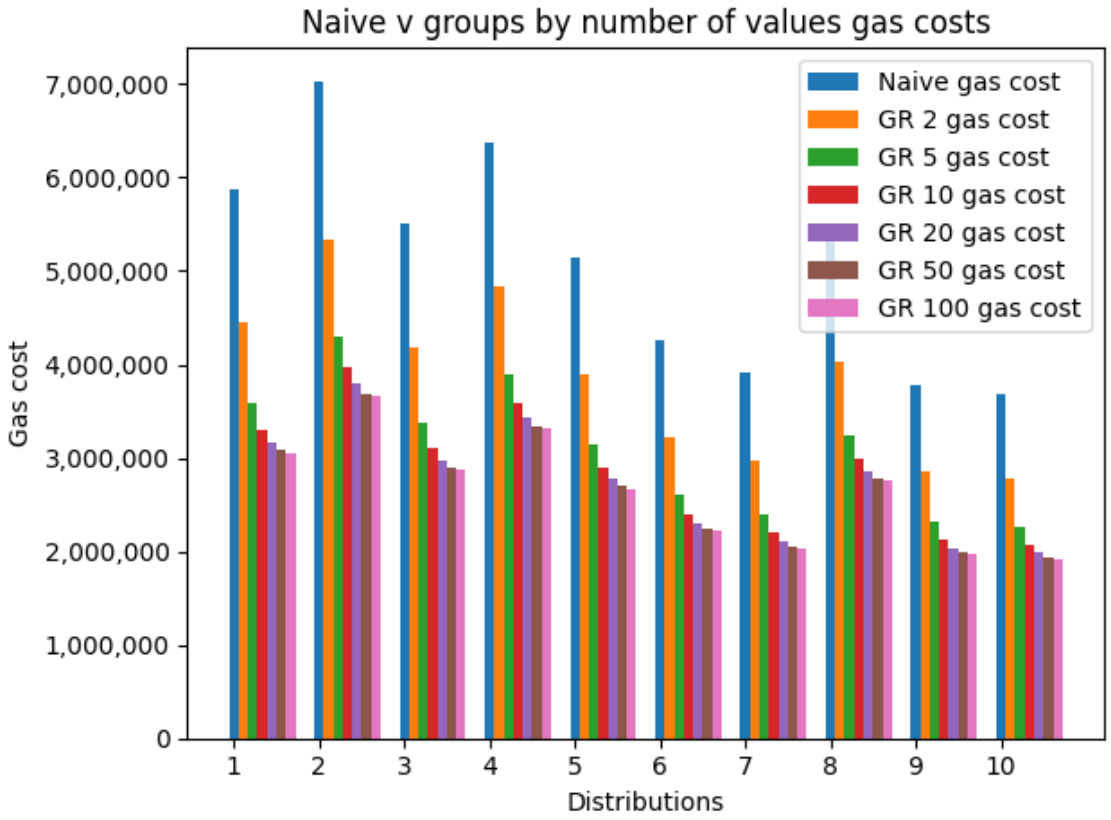

Figure 11: Gas costs naive vs grouping by number of values (approach 4b)

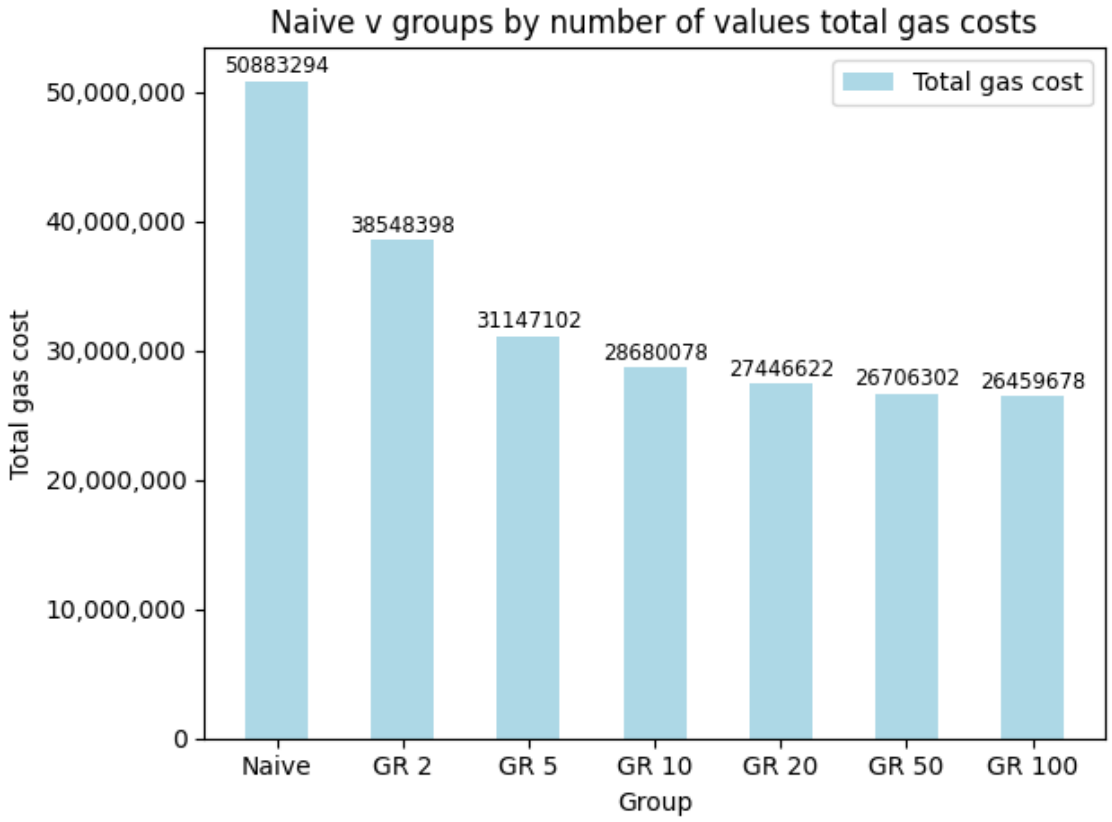

Figure 12: Gas costs naive vs grouping by number of values total (approach 4b)

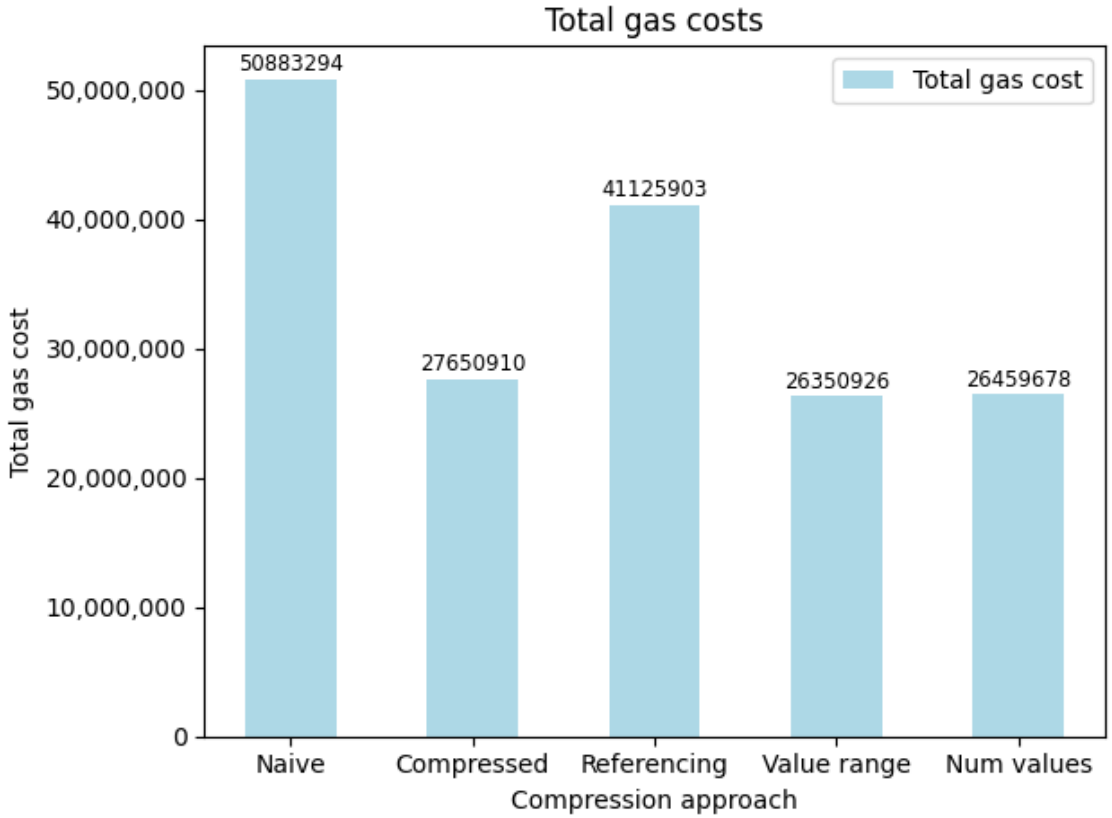

Figure 13: Gas cost comparison across all approaches

References

[1] B. WhiteHat. Rollup diff compression. Available at `https://ethresear.ch/t/rollup-diff-compression/7933`.

[2] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. Available at `https://ethereum.github.io/yellowpaper/paper.pdf`.