Compounding Decreasing LQTY Deposit and Corresponding ETH Gain

Scalability of Ethereum dApps is constrained by the gas costs of writing to storage.

Here we derive formulas for a scalable implementation of a compounding, decreasing Stability Pool deposit and its corresponding ETH gain. Thus, Liquity tracks the deposits and rewards of every depositor with only O(1) complexity for each liquidation.

In previous work, Batog[1] and Solmaz[2] solved the problem of scalable reward distribution in a staking pool, distributing rewards in proportion to a staker's share of total stakes with a "pull" based approach. Only a single running sum need be updated at each reward, sidestepping the need for an expensive O(n) storage update for *n* stakers. However, their work stopped short of solving the problem for a compounding stake.

For a pull-based implementation in Liquity's Stability Pool, we must factor out the initial deposit, and track reward terms that are independent from the *individual* deposit values. We derive formulas for a compounding LQTY deposit and corresponding ETH gain as functions of the initial deposit, and involving a product P and a sum S, which are in turn purely functions of the LQTY losses, ETH rewards and the *total* deposits.

Terms

$\mathbf{d_i}$: A given user's LQTY deposit at liquidation event *i*
$\mathbf{q_i}$: The LQTY debt absorbed by the Stability Pool from liquidation *i*
$\mathbf{e_i}$**:** The ETH sent to the Stability Pool from liquidation *i*
$\mathbf{D_i}$: Total LQTY deposits at liquidation *i*

Let $\mathbf{d_0}$ represent the user's initial Stability Pool deposit. After the first liquidation, their new deposit $\mathbf{d_1}$ is given by:

1) $\mathbf{d_1 = d_0 - LQTYLoss}$

Liquidation "earns" a LQTY loss for the deposit proportional to the prior share of total deposits:

2) $\mathbf{d_1 = d_0 - (q_1 * d_0/D_0)}$

3) $\mathbf{d_1 = d_0(1 - q_1/D_0)}$

Since the deposit compounds, the deposit value after liquidation 2 is a function of the previous deposit value:

4) $d_2 = d_1 * (1- q_2/D_1)$

and substituting equation 2):

5) $d_2 = d_0 * (1-q_2/D_1) * (1-q_1/D_0)$

Similarly:

6) $d_3 = d_2 * (1-q_2/D_1)$

7) $d_3 = d_0 * (1-q_3/D_2) * (1-q_2/D_1) * (1-q_1/D_0)$

And the general case for a compounded deposit:

8) $d_n = d_0 * PROD^{n}_{i=1} (1-q_i / D_{i-1})$

Deposits made at t > 0

For a deposit made between liquidations [*t, t+1*], which is withdrawn between liquidations [*n, n +1*], rewards are earned from liquidations *t+1*, *t+2*, …, *n*.

We take snapshot of the product at liquidation *t*, and calculate the compounded stake at liquidation *n* as:

9) $d_n = d_t * PROD^{n}_{i=1} (1-q_i / D_{i-1}) / PROD^{t}_{i=1} (1-q_i / D_{i-1})$

i.e. labelling the product term ‘P’:

10) $d_n = d_t * P / P_0$

Where **P** is the current product, and P_0 is the snapshot of the product at the time when the user made the deposit.

Corresponding ETH Gain

The LQTY deposit earns an ETH gain at each liquidation.

At each liquidation *i*, the user's LQTY deposit effectively decreases. Thus we can write the user's corresponding cumulative ETH gain $\mathbf{E_n}$, from a series of *n* liquidations, as:

11) $$E_n = d_0(e_1 / D_0) + d_1(e_2 / D_1) + d_2(e_3 / D_2) + ... + d_{n-1}(e_n / D_{n-1})$$

Then, using our expression for the deposit from equation 8):

12) $$\begin{aligned} E_n = \; & d_0 * (e_1 / D_0) + \\ & d_0 * (e_2 / D_1) * PROD^{1}_{i=1} (1 - q_i / D_{i-1}) + \\ & d_0 * (e_3 / D_2) * PROD^{2}_{i=1} (1 - q_i / D_{i-1}) + \ldots + \\ & d_0 * (e_n / D_{n-1}) * PROD^{3}_{i=1} (1 - q_i / D_{i-1}) \end{aligned}$$

And factoring out the initial deposit, $\mathbf{d_0}$:

13) $$E_n = d_0 * SUM^{n}_{k=1} [(e_k / D_{k-1}) * PROD^{k-1}_{i=0} (1 - q_k / D_{k-1})]$$

Where the initial product term $\mathbf{PROD^{0}_{i=0} = 1.}$

Thus, we have $\mathbf{E_n}$ as a function of the initial deposit $\mathbf{d_0}$, and the rewards and total deposits at each liquidation.

Let the summation $\mathbf{SUM^{n}_{k=1} [\ldots]}$ be denoted **S**.

Deposits made at t > 0

For a deposit made between liquidations [*t, t+1*], which is withdrawn between liquidations [*n, n +1*], rewards are earned from liquidations *t+1*, *t+2*, …, *n*.

To account for the deposit entering after liquidations have already begun, we correct the product terms in 13) to $\mathbf{P^{k}_{i=1} / P_0}$, as per 10).

We take a snapshot of the sum at liquidation *t*, labelling it $\mathbf{S_0}$**.** Then, the ETH gain earned by the deposit at liquidation *n* is:

14) $$E_n = d_t * (S - S_0) / P_0$$

Basic Implementation

Making a deposit:
Record deposit: deposit[user] = $\mathbf{d_0}$
Update total deposits: $\mathbf{D = D + d_0}$
Record product snapshot: $\mathbf{P_0 = P}$
Record sum snapshot: $\mathbf{S_0 = S}$

Upon each liquidation yielding LQTY debt ‘q’ offset with the Stability Pool and ETH gain ‘e’:
Update S: $\mathbf{S = S + (e / D) * P}$ *(intuition: a deposit’s marginal ETH gain is equal to the deposit * ETH per unit staked * current correction factor)*
Update P: $\mathbf{P = P * (1- (q / D))}$
Update total deposits: $\mathbf{D = D - q}$

Withdrawing the deposit and ETH gain:
Compute final compounded LQTY deposit d: $\mathbf{d = d_0 * P / P_0}$
Compute final corresponding cumulative ETH gain E: $\mathbf{E = d_0 * (S - S_0) / P_0}$
Send **d** and **E** to user
Update deposit: deposit[user] = 0
Update total deposits: $\mathbf{D = D - d}$

Practical Implementation in Liquity

Two further considerations are needed for our implementation:

1) Liquidations that completely empty the Stability Pool
2) How to handle the eternally decreasing product **P**, without truncating to 0

1) Liquidations that completely empty the Pool

Problem: Pool-emptying should reduce all deposits to 0, but we should not set **P** to 0. Doing so would break deposit computations for all future deposits: as **P** is a running product, all snapshots and future values would be 0.

Solution: Complete pool-emptying is handled by tracking a “current epoch” variable. The ETH gain reward sum **S** for each epoch is stored in a mapping.

Upon a pool-emptying liquidation, **S** is first updated as usual, for the current epoch. Then, the current epoch is incremented by 1, and **P** and **S** terms are reset for the new epoch. By definition, deposits that were made in past epochs will have been completely cancelled with liquidated debt, and so reduced to 0.

Making a deposit:
Record a snapshot of the current epoch for the deposit

Liquidation that empties the Pool:
Compute the latest value of **S,** and store it for the current epoch
Increase the epoch by 1, and reset the product and sum (**P** = 1, **S** = 0).

Withdrawing Deposit:
When users withdraw, check their epoch snapshot against the current epoch: If equal, deposit and ETH gain are computed as normal.

If the current epoch is greater than the deposit's epoch snapshot, then the deposit was made before the pool-emptying liquidation. Therefore:

- The **compounded deposit** is 0, as the deposit has been fully used to absorb debt.
- The user's **ETH gain** is computed using the **S** sum that corresponds to the epoch in which their deposit was made.

2) Eternally decreasing the product P, without truncating to 0

At liquidation *i*, P is multiplied by some new term $0 < p_i < 1$. Thus in theory, P is always decreasing, but should never reach 0.

Problem: We cannot represent an arbitrarily small value in Solidity. Eventually, division will truncate a small value to 0. If so, all future values of P would be 0, and deposit computation would break.

Solution: use a "current scale" that allows P to decrease indefinitely, but never reach 0. Each scale represents a division by 1e18.

Upon a liquidation that would otherwise truncate P to 0, **S** is first updated as usual, for the current scale. Then, the current scale is incremented by 1, and **P** is updated and scaled by 1e18.

Making a deposit:
Record the current scale snapshot on the deposit

Liquidation:
Compute the latest value of **S,** and store it for the current scale
Liquidation $\mathbf{L_i}$ causes the product **P** to be multiplied by some new product factor $\mathbf{p_i}$. If $\mathbf{p_i}$ * **P** would be truncated to 0, instead do: **P** = (**P** * $\mathbf{p_i}$ * 1e18), and increment the current scale by 1.

Withdrawing a deposit:
First, compute the number of scale changes made during the deposit's lifetime, i.e. (currentScale - scaleSnapshot).

If the number of scale changes is 0, compute the deposit and ETH gain as normal.

Otherwise:

Compounded deposit: If a scale change in **P** was made during the deposit's lifetime, account for it in the deposit computation. If the deposit has decreased by a factor of < 1e-18 (i.e. if more than one scale change was made) just return 0.

ETH Gain: Since the deposit may span up to one scale change, so too does the reward. In this case, obtain the ETH gain using the **S** sums from the two consecutive scales that the deposit spans. The reward from the second sum is scaled by 1e-18. The rewards from both scales are added to make the final ETH gain.

Putting it all together

The implementation relies on a nested mapping: epochToScaleToSum.

The inner mapping stores the sum **S** at different scales, for a given epoch. The outer mapping stores the (scale => sum) mappings for each epoch.

This allows us to track the ETH reward terms at each scale and epoch.Thus we can correctly compute the ETH rewards for a compounding decreasing stake, taking account of periodic Pool-emptying, and getting around the limitations of Solidity arithmetic.

All implementation logic above is combined in *PoolManager.sol*, in the functions:

-getCompoundedDeposit()
_getCurrentETHGain()
_updateRewardSumAndProduct()

References

[1] B. Batog, L. Boca, N. Johnson, *"Scalable Reward Distribution on the Ethereum Blockchain",* 2018. http://batog.info/papers/scalable-reward-distribution.pdf

[2] O. Solmaz, *"Scalable Reward Distribution with Changing Stake Sizes",* 2019. *https://solmaz.io/2019/02/24/scalable-reward-changing/*