**LRT Token Integration Strategies with GammaSwap**

**Preface**

In some of the examples below I’m using a pool made of ARB and USDC.e. Hence an ARB/USDC.e GammaSwap pool. However, one can replace ARB with the LRT token of choice. The logic would be the same. In other places I’m using an example of wstETH as the LRT token paired with USDC but one can choose any other LRT token to replace wstETH, just as any other token can replace USDC.

The strategies described below are to calculate the balance amounts in GammaSwap of a user’s specific token (e.g. LRT token) using smart contract calls or subgraph (theGraph.com) calls. At the end of the document I describe a way to find the total balance of a token (e.g. LRT Token) in our platform.

GammaPool refers to the pool that allows the borrowing of liquidity. I’ll use GammaPool and pool interchangeably unless specififed that I’m talking about the CFMM of the GammaPool

**Two types of integrations:**

1) **Providing liquidity** (LRT token and some other token) into a full range x\*y=k AMM. Users would earn points from the LRT token in the AMM.

Example:

user1 provides 100 wstETH and 100 USDC as liquidity into the AMM. His wstETH balance is 100 wstETH. Total wstETH in the platform is 100 wstETH.

2) **Borrowing liquidity** from the above mentioned AMM in order to turn impermanent loss into impermanent gain. Users would earn points on wstETH held as collateral against the AMM liquidity loan.

Example:

After user1 above provides liquidity, user2 borrows 90 wstETH and 90 USDC from the AMM as a liquidity loan, holding 90wstETH and 90 USDC as collateral. User2 wstETH balance is 90 wstETH.

However, user1’s balance has now changed to 10 wstETH because user2 borrowed 90 wstETH from it. Total wstETH in the platform is still 100 wstETH because wstETH never leaves the platform. All loans are held inside the GammaSwap platform (either the AMM as liquidity or in the GammaPool as collateral for a loan). User1 of course would be compensated in interest payments from user2 for lending his wstETH which he can use to earn LRT point rewards.

**Integration 1: Providing Liquidity into GammaSwap’s x\*y=k AMM (DeltaSwap)**

For the first type of integration (providing liquidity), we can calculate the total balance of wstETH of a user with address x using the following smart contract call on the LPViewer contract

(address token0, address token1, uint256 token0Balance, uint256 token1Balance,) = ILPViewer(lpViewer).lpBalanceByPool(x,gammaPool)

The above function will return the balance of token0 and token1 in the GammaPool belonging to the user (including balances deposited in our staking contracts). For example if in the wstETH/USDC pool token0 is wstETH and token1 is USDC, then the balance of wstETH is given by token0Balance

If we are looking for balances held across multiple pools we can use the following function

(address[] memory tokens, uint256[] memory tokenBalances, uint256 size) = ILPViewer(lpViewer).tokenBalancesInPools(x,[pool1,pool2,pool3])

This function will return the balances for each token in the array of pools provided. The variable “size” is the number of token addresses found in the array of pools. The index of the “tokenBalances” array refers to the token address given by the index of the “tokens” array. For example, if wstETH is in pool1 and pool2 and it’s in index 1 of “tokens” array, then the balance of wstETH held by the user in pool1 and pool2 is given by index 1 in the “tokenBalances” array.

Note:

DeltaSwap is a fork of UniswapV2, therefore emits the same events as the UniswapV2 AMM

The following events emitted by the AMM (DeltaSwap) imply the wstETH balance of a user may have changed

Smart Contract: DeltaSwapPair.sol (github: https://github.com/gammaswap/v1-deltaswap)

Events:

event Mint(address indexed sender, uint256 amount0, uint256 amount1);

event Burn(address indexed sender, uint256 amount0, uint256 amount1, address indexed to);

event Swap(address indexed sender, uint256 amount0In, uint256 amount1In, uint256 amount0Out, uint256 amount1Out, address indexed to);

event Sync(uint112 reserve0, uint112 reserve1);

event Transfer(address indexed from, address indexed to, uint256 value);

**How do you get a list of users that have LP positions in a pool?**

Run the following request against the subgraph. The query picks up LP positions deposited through the GammaPool as LP tokens and that have and have not been staked in the staking contracts (RewardTracker)

poolAndStakedBalances(where:

{

isRewardTracker:false,

pool:"0x16ce9d64490b2f63f88f67e58eb3ce1063c9db10"  
 balance\_gt: 0}) {

id

pool {

id

}

account{

id

}

balance

}

The response looks like  
  
"poolAndStakedBalances": [

{

"id": "0x16ce9d64490b2f63f88f67e58eb3ce1063c9db10-0xa829c1d4542f70714b35ffe95a247373329131df",

"pool": {

"id": "0x16ce9d64490b2f63f88f67e58eb3ce1063c9db10"

},

"account": {

"id": "0xa829c1d4542f70714b35ffe95a247373329131df"

},

"balance": "3162001162"

},  
 …  
]

balance is the quantity of LP tokens of the GammaPool identified by address 0x16ce9d64490b2f63f88f67e58eb3ce1063c9db10. We specify rewardTracker as false to ensure we are not picking up addresses that are staking contracts. The address of the user that owns the LP tokens is in account.id (e.g. address 0xa829c1d4542f70714b35ffe95a247373329131df has 3162001162 LP tokens in pool 0x16ce9d64490b2f63f88f67e58eb3ce1063c9db10, either deposited directly in the GS pool or through a staking contract)  
  
After getting this list, you can request the reserve quantities the user has through the LPViewer

**Integration 2: Borrowing Liquidity through GammaSwap from GammaSwap’s x\*y=k AMM (DeltaSwap)**

For the second type of integration (borrowing liquidity), we will use subgraph calls to calculate the total amount of a token held by a user that has borrowed liquidity in a pool and across all of GammaSwap. The end point to call is a subgraph service with the following url

https://api.thegraph.com/subgraphs/name/gammaswap/gammaswap-v1-arbitrum

To calculate the amount of a token whose address is “0xff970a61a04b1ca14834a43f5de4533ebddb5cc8” (this is the address of USDC.e in arbitrum) held by a user with address “0x749f90c2f42f7184a018ce3743c413bfe4f316ac” that has borrowed liquidity in a pool with address “0xfedf53e144a4f0071fe66b79bf6ccc908f9ebe4e” (this is the address of ARB/USDC.e gammapool in arbitrum” is given by the following subgraph call.

collateralTokenBalances(

where:{

account:"0x749f90c2f42f7184a018ce3743c413bfe4f316ac",

token:" 0xff970a61a04b1ca14834a43f5de4533ebddb5cc8"

pool:"0xfedf53e144a4f0071fe66b79bf6ccc908f9ebe4e"

}) {

Pool {

id

}

balance

}

The above request will return the following response

"collateralTokenBalances": [

{

“pool” {

“id”: “0xfedf53e144a4f0071fe66b79bf6ccc908f9ebe4e”

}

"balance": "656880132"

}

]

Where the balance is the big integer form of the token balance for the user in solidity. In this example, the token address is for USDC.e, which is 6 decimals. Therefore, the result has to be divided by 10^6 to get its decimal representation.

If we would like to get the balance of the same token above held as collateral by the same liquidity borrower across all pools in GammaSwap, we can do so with the following subgraph call

totalCollateralTokenBalances(

where:{

account:"0x749f90c2f42f7184a018ce3743c413bfe4f316ac",

token:" 0xff970a61a04b1ca14834a43f5de4533ebddb5cc8"

}) {

balance

}

The above request will return the following response

"totalCollateralTokenBalances": [

{

"balance": "3331306978"

}

]

The response value is a greater number because this user has multiple loans in different pools holding USDC.e as collateral.

If one wishes to get a list of all the addresses in the integration holding the balance of a specific token at a given pool perform the following subgraph request, where the pool has address 0xfedf53e144a4f0071fe66b79bf6ccc908f9ebe4e and the token in question is USDC.e, which has address 0xff970a61a04b1ca14834a43f5de4533ebddb5cc8 in arbitrum. We use the field balance\_gt set to 0 to specify that we are only looking for positive balances.

collateralTokenBalances(

where:{

pool:"0xfedf53e144a4f0071fe66b79bf6ccc908f9ebe4e",

token: “0xff970a61a04b1ca14834a43f5de4533ebddb5cc8”

balance\_gt: 0

}) {

account {

id

}

balance

}

The balance of the token will be in the balance field, which is in big integer form, and the address of the users holding the token will be in the account.id variable.

The result you get would look as follows

"collateralTokenBalances": [

{

"account": {

"id": "0x5243524a18aee9d8a13159203eb453a322609a7a"

},

"balance": "1767512"

},

{

"account": {

"id": "0x720b575265c62d1ac21a41fcfdaca57069048caf"

},

"balance": "3080836"

},

…

,{

"account": {

"id": "0x749f90c2f42f7184a018ce3743c413bfe4f316ac"

},

"balance": "656880132"

}

]

If one wishes to get a list of all addresses holding a token in any GammaSwap pool at the given chain of the subgraph perform the following request

totalCollateralTokenBalances(

where:{

token:"0xff970a61a04b1ca14834a43f5de4533ebddb5cc8"

balance\_gt:0

}) {

account {

id

}

balance

}

We use as an example the same token address as before. That’s the address for USDC.e in arbitrum. Just as in the previous example, the addresses holding a positive balance of USDC.e will be in the account.id field

The result would look as follows

"totalCollateralTokenBalances": [

{

"balance": "458484530",

"account": {

"id": "0x049e08f8b00a4a3d7abfe3f2e7ab4e8f407e1248"

}

},

{

"balance": "108278793",

"account": {

"id": "0x05325ef5a6e146b475289c3a8ffe0a3713265a3f"

}

},

…

,{

"balance": "868062412",

"account": {

"id": "0x1334fdd8ae0694168785aca024768f1d99b65ff9"

}

},

]

**How to Calculate total amount of wstETH (LRT Token) in GammaSwap?**

The total amount of a token in GammaSwap across all pools can be obtained with the following subgraph request

tokens(

where: {

id:"0xff970a61a04b1ca14834a43f5de4533ebddb5cc8"

}) {

balanceBN,

lpBalanceBN,

dsBalanceBN,

gsBalanceBN

}

The request above was for USDC.e in arbitrum.

balanceBN is the total amount of USDC.e held by all liquidity providers and all liquidity borrowers as collateral in all GammaSwap pools combined.

lpBalanceBN is the total amount of USDC.e held by all liquidity providers in all GammaPools combined (DeltaSwap and non DeltaSwap CFMMs).

dsBalanceBN is the total amount of USDC.e held by all liquidity providers in all DeltaSwap pools combined.

gsBalanceBN is the total amount of USDC.e held as collateral by all liquidity borrowers in all GammaPools combined.

The amounts are given in big integer form.

The result would look as follows

"tokens": [

{

"balanceBN": "618107199019",

"lpBalanceBN": "249137622574",

"dsBalanceBN": "100000000000”,

"gsBalanceBN": "368969576445"

}

]

The additions of lpBalanceBN and gsBalanceBN equal balanceBN. If there is a discrepancy in this relationship, it’s due to synchronization problems in the subgraph. The field lpBalance contains dsBalance, therefore, lpBalance is always greater or equal to dsBalance.

It is also possible to get the total amount of tokens held as collateral per pool using the subgraph. This can be done with the following query

gammaPools(where: {

id:"0xfedf53e144a4f0071fe66b79bf6ccc908f9ebe4e"

}) {

token0Balance

token1Balance

})

In the example above we get the total amount of collateral in token0 and token1 held in gammaPool with address 0xfedf53e144a4f0071fe66b79bf6ccc908f9ebe4e.

The balances in big integer form and therefore would have to be adjusted using their decimal expansion to get the decimal form.

The result of the above query in arbitrum network would look like this.

"gammaPools": [

{

"token0Balance": "1585953545781071012064",

"token1Balance": "2806972844"

}

]

The pool above was for ARB/USDC.e. Token0 is ARB therefore it’s 18 decimals, while token1 is USDC.e which is 6 decimals.

The same type of query can be accomplished using the subgraph for liquidity providers. However, it must be performed using the DeltaSwap address used by the GammaPool. The DeltaSwap address is the CFMM address of the GamamPool and it doesn’t change. It only needs to be queried once with the following query

gammaPools(where: {

id:"0xfedf53e144a4f0071fe66b79bf6ccc908f9ebe4e"

}) {

cfmm

})

The DeltaSwapPair address of the GammaPool is in the cfmm field. The result looks as follows

"gammaPools": [

{

"cfmm": "0xf1c03b8174dc1dd8bc48efc62b9c95434f524c8c"

}

]

Note, that a GammaPool can be made for any type of CFMM (e.g. Sushiswap, UniswapV2, etc.) Therefore, the logic explained here for obtaining the total token balances of all liquidity providers in a GammaPool only applies to DeltaSwap. Using the same logic for GammaPools created for Sushiswap, UniswapV2, etc. would count the balances held by SushiSwap and UniswapV2 as part of the GammaSwap protocol. Therefore, it would be double counting for the purposes of an LRT integration.

All DeltaSwap balances however, are part of GammaSwap’s balances since DeltaSwap is a GammaSwap product and liquidity providers deposit into DeltaSwap through the GammaPool.

Also, the logic above does not affect the total sum of tokens held in GammaSwap given by the

Query in the tokens entity. Since for GammaPools created for pools other than DeltaSwap (e.g. UniswapV2, Sushiswap, etc.) the dsBalanceBN field will always be zero.

Once we have the cfmm address (DeltaSwap pool address) we can use this to query for the reserves in the DeltaSwap pool with the following query

deltaSwapPairs(where: {id:"0xf1c03b8174dc1dd8bc48efc62b9c95434f524c8c"}) {

token0 {

id

symbol

}

token1 {

id

symbol

}

reserve0

reserve1

}

Where token0.id and token1.id are the addresses of the tokens in question and reserve0 and reserve1 is their corresponding balances of liquidity providers in the DeltaSwap pool in big integer form.

The result for the above query looks as follows

"deltaSwapPairs": [

{

"token0": {

"id": "0x912ce59144191c1204e64559fe8253a0e49e6548",

"symbol": "ARB"

},

"token1": {

"id": "0xff970a61a04b1ca14834a43f5de4533ebddb5cc8",

"symbol": "USDC.e"

},

"reserve0": "11670207194844393435491",

"reserve1": "18161380262"

}

]

These numbers are always up to date as per the events that our subgraph has processed.