Abstract

We present a new algorithm for order book DEX “LOBSTER - Limit Order Book with Segment Tree for Efficient oRder-matching” that enables on-chain order matching and settlement on decentralized smart contract platforms. With LOBSTER, market participants can place limit and market orders in a fully decentralized, trustless way at a manageable cost.

Overview

Implementing an order book on the Ethereum Virtual Machine (EVM) is a non-trivial task. Unlike centralized exchanges (CEX) that can iterate over orders to settle trades or several price points to handle large market orders, on EVM, such operations can reach the network’s gas or time limitations and fail to go through. Any operations that iterate an arbitrary number of times must be constrained and optimized to save the user from excessive gas fees and allow orders to be made reliably.

When implementing Clober, we had to solve two significant challenges regarding arbitrary iteration.

1. Iterating over orders to settle.
2. Iterating over price points to fill a large market order.

Through the thoughtful use of data structures that we modified and rewrote from scratch to optimize gas fees, we overcame these two bottlenecks and successfully implemented a gas-feasible order book. Our rigorous optimizations, for the first time, allow an EVM order book with gas fees on par with what the state-of-the-art AMMs have to offer.

1. Iterating Over Orders to Settle

Manual Claiming

Settling an order involves distributing the proceeds of a trade to both the taker and the maker. Although there is only one taker involved in any trade, multiple makers could require payment. However, iterating over a lengthy list of makers would be problematic on the EVM as it may lead to the transaction timing out and failing. To resolve this issue, the maker’s share of the settlement must be handled separately, relieving the taker of the burden.

Knowing When to Claim via Claim Range

Without the takers alerting the makers that their orders are claimable, checking if an order is claimable is left to the claimer. This can be done by comparing the accumulative sum taken from an order queue with the “claim range” of an order, a value derived by adding the order sizes in an order queue. The key takeaway is that with the introduction of manual claiming, the burden of iterating over orders is simplified to a problem of finding the sum or partial sum of a constantly updating queue of orders.

(Finding the sum of an array or subarray of orders might seem like a problem that involves iteration. However, as discussed later, it can also be solved elegantly using a segment tree.)

Order Claim Range

The order claim range

is derived from the following equation.

$$Order \ Claim \ Range: [\alpha_n, \beta_n]$$

$$f(n)$$

= order size for n

th order

$$\alpha_0 = 0$$

$$\alpha_{n>1} = \sum_{x=0}^{n-1} \ f(x)$$

$$\beta_n = \alpha_n + f(n) = \sum_{x=0}^{n} \ f(x)$$

Total Claimable Amount

We will call the “accumulative sum that has been taken” mentioned earlier the total claimable amount. When someone takes an order from an order queue, the total claimable amount is increased by that amount.

Comparing Order Claim Range with Total Claimable Amount

Now Let’s put these to values to use. The equations below show if and by how much, an order can be claimed.

Let T

= total claimable amount

There are three different states an order claim interval [a, b]

can be concerning T

.

1. b \leq T

, the order is completely claimable.

1. a < T < b

, the order is partially claimable and the claimable amount is T - a

.

1. T \leq a

, the order is not claimable at all.

In other words, the claimable amount c(T, [a, b]) = min(max(0, T - a), b - a)

Example

For example, let’s assume that Alice, Bob, and Carol each bid for 10 ETH in sequence at a certain price point. If Dave comes in to sell 15 ETH at that price point, Alice can claim all 10 ETH, Bob can claim 5 ETH, and Carol none at all. While this is quite intuitively true, it can get confusing when claims, partial claims, and cancels start entering the picture. By comparing the order claim range of each user with the total claimable amount, it becomes easier to track.

[

example1

914×362 36.9 KB

](https://ethresear.ch/uploads/default/original/2X/a/a3ad51f07a88731c2311143325ef408f6af73288.png)

If Bob were to cancel his order (which would automatically claim the 5 ETH currently available to claim), Bob’s order size and Carol’s claim range would have to change to reflect this.

[

example2

914×354 36.6 KB

](https://ethresear.ch/uploads/default/original/2X/d/d432093636d17dd5c790bb37cdb650d94d0125bc.png)

As Bob’s order size decreases, Carol’s order claim range updates to keep things consistent. This is good news for Carol as she does not have to wait for Bob’s order to be filled anymore for her turn to come.

From this, we can understand that a solution that holds the order claim range in memory would not be sufficient, as canceling an order would cause the code to iterate over every subsequent order to update each claim range. There needs to be a better way of getting the sum of order sizes in a given range.

Sum of Given Range

Let’s look into three different ways of getting the sum of a range and compare their pros and cons to select the best solution.

Brute Force

Simply iterating over the queue to query the sum every time is possible, but it would require a very hard limit on the maximum number of orders allowed on the queue. Some implementations of EVM order books that iterate over orders only allow 32 orders per queue. This implementation has an advantage over other methods in that updating an order size only requires one storage slot update. However, we considered the 32 orders per queue limit too strong of a constraint to be considered a fully-fledged order book.

Prefix Sum (Lookup Array)

A slightly more sophisticated approach (algorithm-wise) uses a lookup array with the sum of all the orders up to the n

th order as the n

th element of the array. This can make querying the sum a matter of O(1)

, but show horrible performance when an order is claimed or canceled since when an unclaimed order size is updated, it will require an iteration updating the lookup array. Considering how common canceling and claiming are, and how sstore

can be ten times more expensive than a sload

, this version has the worst overall performance.

Segment Tree

With a segment tree, querying the sum and updating an element both take O(log(n))

and the height of the tree is the biggest factor in how expensive a query and update will be. An update must be made at every level of the tree to update an element. Therefore with this method, we need to constrain the tree's height to ensure the fees don't get out of hand.

Segmented Segment Tree

With a naive implementation of a segment tree, to support the 32768 orders per queue currently supported, 16 storage updates must be made to create or update an order. This was not acceptable for our team, and we went back to the drawing board to lower the number of sstore

s from 16 to just 4. The Segmented Segment Tree was born.

There's Always Room for More

The first step to making the Segmented Segment Tree was to fit the maximum number of nodes in a single storage slot. By placing the following constraints on the way orders were stored, we could fit the order size in 64 bits, and since each storage slot is 256 bits long, four nodes fit per slot.

1. Have a unit amount for the order size (the quote unit).
2. Save the order size using the quote unit for both bids and asks.

When 0.000001 USDC is used as the quote unit, since the biggest number uint64

can hold is 18,446,744,073,709,551,615, as long as the depth of every price point is less than $18,446,744,073,709.55, there will be no overflow. Over half of the M2 money supply would have to be placed on a single price point for an overflow to happen.

Segmentation

The Segmented Segment Tree's core concept comes from splitting the segment tree into multiple segments, which are then captured and stored in storage slots. What makes the Segmented Segment Tree so remarkable is that not only can four nodes fit into one storage slot, but eight nodes stored in two storage slots can generate seven parent nodes, eliminating the need to store those seven in storage altogether! As a result, every two storage slots represent 15 nodes, with eight leaf nodes stored and seven parent nodes generated in memory when needed.

Below is a diagram of how a segment tree looks before and after segmentation.

[

tree

1985×836 20.8 KB

](https://ethresear.ch/uploads/default/original/2X/8/8bf561fc5a6b53f1425fc46aa7a99d2f246d1b3d.png)

Every tree level needs to be updated when updating a segment tree. By segmenting the segment tree, a tree with 32768 leaves would have 4 levels instead of 16. This translates to needing 4 sstore

s rather than 16, a significant improvement. Also, the updates would write on the same memory slots more often, which means even lower gas fees on average. For example, the first four leaves will write to the same four storage slots on updates. This aspect of the Segmented Segment Tree lowers the average gas fees by a substantial amount making the performance of this tree even more impressive.

# of Segmented Segment Tree Levels

# of Segment Tree Levels

# of Leaf Nodes

1

4

8

2

8

128

3

12

2048

4

16

32768

5

20

524288

Order Index

Whenever an order is placed, the order index is incremented to track how the order claim ranges should be queried and where to place following orders. The modulus of the order index is used, cycling through the order queue. If the leaf node that the new order is trying to utilize is not empty, the order books checks if the order placed 32768 orders ago is claimable or claimed. If it is, the new order happily replaces the value with its own. If not, the order will revert as the tree is filled. In other words, the 32768 order limit is on the number of open orders, not the number of orders accumulated.

2. Iterating Over Price Points to Fill a Large Market Order

Mind the Gap!

Lower liquidity in markets can create instances where the two lowest ask price points or the two highest bid price points are not adjacent, creating a gap. This gap can become problematic when iterating over price points to handle market orders filled across multiple price points. There is no reliable way to know how big these gaps are, and the worst case would merit a scan through the entire list of possible prices.

In the order book below, 10 ETH is sold at 1000 USDC, and 10 ETH is sold at 2000 USDC. If Alice were to make a market order of 30000 USDC, iterating over the gap would be extremely expensive gas-wise, with 1000 storage reads being called to check the depth at each price point. This transaction would probably not even go through. (If Alice were to make an order of 30001 USDC, it would have to iterate to the highest possible price to find that there wasn’t enough liquidity, to begin with.)

[

orderbook

415×584 5.11 KB

](https://ethresear.ch/uploads/default/original/2X/2/2df43f91f87fed8a4d2eefd6ca0c1cf953b2abcd.png)

Skipping the Gap. Heap Helping Heaps

Instead of iterating through the price points in the gap, the order book uses a heap data structure to keep tabs on all the valid price points, so it can effectively skip the gaps. A heap is a tree-based data structure where if it's a max heap, each node is bigger than or equal to all of its children, and if it's a min heap, each node is smaller than or equal to all of its children. We used a max heap to handle the highest current bid and a min heap to handle the lowest current ask.

By using a highly customized heap, the Octopus Heap, we successfully created a heap that can handle most cases using just one storage read or storage write.

Octopus Heap

Similar to how the Segmented Segment Tree benefited immensely by packing several nodes into a single slot, we compressed the heap while taking advantage of the fact that non-empty price points tend to be in proximity. Another innovation was born, the Octopus Heap.

CPI (Compressing the Price to an Index)

For any optimization to occur, we need to minimize the number of bits used to store the price point. We found that the price could be expressed in 16 bits without losing data by using an index instead of storing the actual price. Storing the actual price would be wasting precious bits to support integers that are not viable price points. For example, let's say that the viable price points are 10010, 10020, 10030, and so on. The bits used to express 10010 in two's complement can also be used to express 10011, 10012, and 10013, which is unnecessary as those prices are not supported. By saving 10010, 10020, and 10030 as 0, 1, and 2, we can save on wasted space and use fewer bits.

Currently, there are two strategies for mapping prices to an index. These strategies are called price books; we have arithmetic and geometric price books. As the name suggests, the arithmetic price book uses an arithmetic progression, and the geometric price book uses a geometric progression. The initial term and common difference/ratio are set by the market creator.

Splitting the Bill (8 bits + 8 bits = 16 bits)

The Octopus Heap consists of a compressed heap

and a leaf bitmap,

and instead of storing the whole 16-bit price index on the heap, it stores the first 8 bits on the compressed heap and the remaining 8 bits on the leaf bitmap.

Compressed Heap

A compressed heap

is a heap where several nodes are stored on a single storage slot, but unlike the Segmented Segment Tree, we cannot omit any nodes in this process, meaning the parent nodes must be saved as well. By only storing 8-bit values on the heap, 32 nodes can be packed into a single storage slot.

As the first 8 bits of our price index have 256 different values, a heap with nine levels is enough to capture each value. We stored this 9-level high heap by storing the first five levels in a single slot called the head and the remaining in slots called the arms. Every leaf node of the head has two child trees that are three levels high (except for the first leaf node with a child that is four levels high). These 3-level high trees have seven nodes each, so up to 4 of these trees can be stored in a single arm. Since there are 32 smaller trees, we need eight arms to hold them, hence the name Octopus Heap.

The result is a heap that is nine levels high but only requires 2 sstore

s at most for popping or pushing, which is incredibly gas efficient. Even though operations regarding the heap are very cheap, they will rarely be used and, in most cases, replaced by an even cheaper operation on the leaf bitmap.

Leaf Bitmap

The leaf bitmap

stores the entire price index, using the first 8 bits as the key and the last 8 bits as a position on a storage slot using mapping(uint8 => uint256)

. When saving the last 8 bits of the price index, we fetch the storage slot mapped to the first 8 bits and mark a bit on the 256-bit storage slot to represent that it exists. For example, when saving 0b00000011, we would mark the fourth bit on a uint256

, as shown below. (Coincidentally, 8 bits have exactly 256 integer values.)

8 bit: 00000011 (int value 3)

256-bit storage: 00000000 00001000

Since, in most cases, the price points with open orders will be close to each other, many bits will be marked on the same 256-bit leaf storage slot. This makes finding the next valid price point for most cases possible without using additional sload

s. This would also make updates to the compressed heap very rare, as only price indices in different groups of 256 need to update the heap. If each price point is a multiple of 1.001 away from the other, the highest price point in a slot would be 1.29(1.001^{256})

times the lowest price point. So when the market price is 1000, a limit order of 1290, a price 29% higher, can still be stored in the same storage slot, and the heap does not need to be updated.

Conclusion

Order books on Clober, with its design choice to remove the burden of claiming from the taker and rigorous gas optimizations by building novel data structures from scratch, achieved what was once deemed impossible, a Fully On-chain Order Book on EVM that can scale. On-chain order books are arguably the holy grail of DeFi, a fair, trustless, and permissionless way to facilitate trades without sacrificing the composability smart contracts offer.

As DeFi continues to grow and attract trading volume away from CeFi, order book DEXs will play an increasingly critical role in the pricing of assets. Order books will also enable financial instruments with expiration dates, such as bonds, options, and prediction markets, to find a natural home in DeFi, with liquidity supplied in a manner that AMMs cannot match. Ultimately, the shift towards order book DEXs will help to improve the overall efficiency and stability of the DeFi ecosystem.