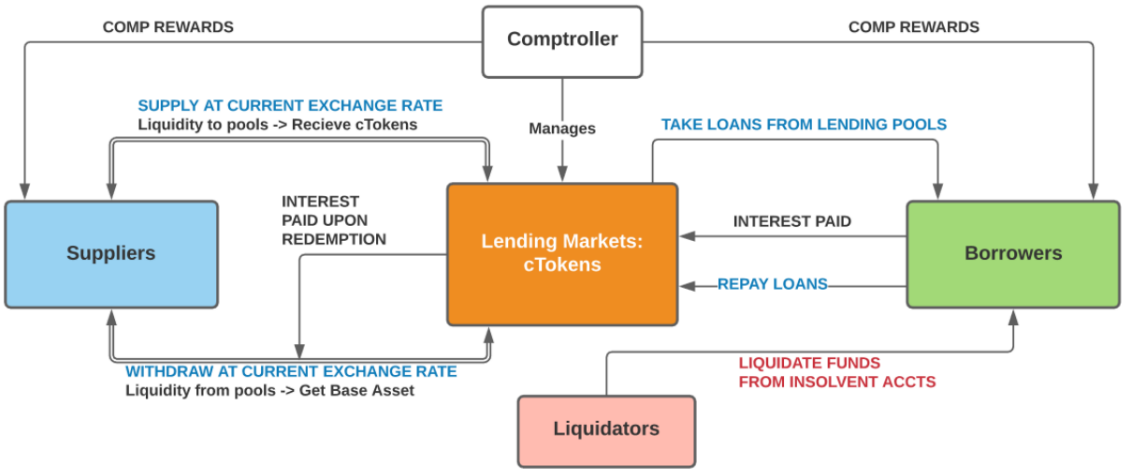
Compound

1. Description of the protocol

What is Compound?

Compound is a Decentralized lending protocol on Ethereum. It allows users to borrow or lend assets in pools of assets. In each pool, the interest rate is set algorithmically based on the supply & demand.

The protocol supports several assets, including Dai (DAI), Ether (ETH), USD Coin (USDC), Ox (ZRX), Tether (USDT), Wrapped BTC (WBTC), Basic Attention Token (BAT), Augur (REP), AAVE, LINK (Chainlink), MKR (Maker), SUSHI (SushiSwap), TUSD (TrueUSD).



Supplying Assets in Compound

Users can lend their crypto assets and get an equivalent amount of cTokens, the Compound's native token plus earn interest rate, also in cTokens.

The Compound protocol aggregates then the supply of each lender within a pool. At any time, they can withdraw their crypto assets and the earned interest by redeeming the cTokens they hold.

Note that the users earn interest rate through cToken's exchange rate, which can only increase in value relative to the underlying asset to incentivize liquidity providers to keep their funds in Compound longer.

Borrowing Assets in Compound

Users can borrow money from Compound by using any other supported assets (expressed in cToken) in the protocol as collateral and pay interest rate on every Ethereum block.

The maximum amount users can borrow is limited by the collateral factors of the assets they've supplied. Illiquid assets have low collateral factor while liquid assets have high collateral factor.

They cannot borrow more than their borrowing capacity, protecting the protocol from default risk

$$borrowing\ \ capacity = \sum_j User's\ cToken_j\ balance\ \ \times\ collateral\ factor_{cToken_j}$$

where $cToken_j$ balance is expressed in ETH.

Liquidation

If a user's borrowing balance exceeds their borrowing capacity, a portion of the borrowing may get liquidated by other users to eliminate the protocol's risk. "Liquidators" are incentivized to do so as they get in exchange the (equivalent amount of) user's cToken collateral at the current market price minus a liquidation discount (5%).
More specifically, when liquidating a position, the liquidator chooses:
- among accounts that are subject to liquidation, which outstanding borrow they want to repay and the portion they want to repay.
- The collateral cTokens they want to get in exchange.

Note that it is always more pertinent to liquidate the account that is more in debt, as gas fees are fixed and the **reward for a liquidator is a percentage of the collateral of the liquidated borrower.** Further, the liquidator cannot repay more than 50% of the outstanding amount in a single transaction. This is called the **close factor**. The close factor is set protocol wide through governance decisions and is the same for all assets. Then the liquidator can either keep the cTokens in the protocol and earn interest rate as their value increases over time or redeem for the underlying asset.

Liquidators are expected to look for liquidation opportunities themselves either by querying Compound's own API or The Graph-s Compound subgraph. Once they have found a liquidation opportunity they can trigger it by sending a call to a smart contract (see Appendix A. for more details).
How the liquidation is triggered:
https://medium.com/dragonfly-research/liquidators-the-secret-whales-helping-defi-function-acf132fbea5e

Exchange Rate between cTokens and the underlying asset

The exchange rate between cTokens and the underlying asset is equal to:

$$Exch.rate_a = \frac{underlyingBalance + totalBorrowBalance_a - Reserve_a}{cTokenSupply_a}$$

Where:
- $underlyingBalance$: total amount of the underlying asset owed by the contract
- $totalBorrowBalance$: amount of underlying currently loaned out by the market

- $Reserves_a$ portion of historical interest rate paid by borrowers set aside for protocol's governance, determined by the reserve factor
- $cTokenSupply$: total amount of the cToken

Interest Rates

Interest rates are set algorithmically based on the supply & demand of the corresponding asset. They should increase as a function of demand. More specifically, they are determined as a function of the utilisation ratio. The utilisation ratio U for each market a aggregates the information on the supply and the demand into one variable as follows:

$$U_a = \frac{Borrows_a}{Cash_a + Borrows_a - Reserves_a}$$

where:
- $Borrows_a$: the amount of a borrowed
- $Cash_a$: the amount of a left in the system

The utilisation ratio is thus the rate of money borrowed out of the total money available in the pool.

Borrow and supply rates

The borrowing interest rate depends on the interest rate model. The coins in Compound do not all have the same interest rate model .

Example: The USDC interest rate model (the standard one) takes in two parameters:
- Base rate per year, the minimum borrowing rate
- Multiplier per year, the rate of increase in interest rate with respect to utilisation

$$Borrowing\ Interest\ Rate_a = multiplier \times U_a + base\ rate$$

Then the Supply interest rate, which is different from the borrowing interest rate, depends on the later.

$$Supply\ interest\ rate_a = Borrowing\ Interest\ Rate_a \times U_a \times (1 - Reserve\ Factor_a)$$

2. Data description & representation

To understand the relationship among lenders, borrowers and market positions, we constructed two matrices for different periods of time: one for the borrowings and one for the lending. In both matrices, the columns correspond to pools and the rows to users. We also constructed one matrix with all the liquidations in each pool.

To construct those matrices, we used the Market and AccountCToken tables that can be extracted directly from the Graph API. Those tables report the current state of the market up to the specified block and the state of all accounts at the specified block (snapshot).
We extracted daily snapshots in 2020-2021 (the last block of each day).

1. **The market table** contains the daily information from the Market table on the total borrows, total supply, reserves, borrow rate, supply rate, reserve factor, exchange rate, and market price.
2. **The lending table** indicates the total amount that each account (user) deposits to each pool, including interests he/she has accumulated up to that time (i.e. cTokenBalance column in AccountCToken table). Each column indicates the asset while each row is an account.
3. **The borrowing table** indicates the total amount that each account owes to the pool at that time (i.e. borrowBalanceUnderlying in AccountCToken). Same as the underlying asset metric, each column indicates the asset while each row is an account.
4. **Liquidation table** mainly indicates the account being liquidated, the amount repaid by the liquidator, and the amount of collateral the latter receives in exchange.

See Appendix B. for the database schemas.

3. Network Analysis of Compound

3.1 Research ideas

While the protocol claims to protect lenders from default risks, lenders (still) get a high return. However, in classical Finance, a high return is associated with a high risk. So, is the protocol really **risk-free**? When a shock hits a user, does that propagate throughout the network i.e is there any systemic risk in the protocol? If so, how does it propagate?

On Compound, lenders and borrowers interact with pools of assets. They do not interact directly with each other. Similarly, pools are independent.
Nonetheless, users borrow in some pools and collateralize in other pools, which make the pools interconnected. Furthermore, some users may have the same portfolio, they may then be impacted by market dynamics and by exogenous shocks in a similar way even though they don't have a direct credit relationship.

Therefore, before answering this question, it will be first interesting:

- to study and understand the network structure of Compound using **network measures such as centrality measures**.
- Do agents have the same lending/borrowing strategy?
- What are the risk factors of the pool? (e.g. total borrow amount, interest rate, or total asset in the pool)
- Measure the contagion risk between pools when a user defaults in a given pool (what is the probability of default based on the probability of another user to default?) , when the price of one asset drop or any other exogenous shock.
- When and in which case liquidation can pose the risk of cascading failure in the *collateral → borrow assets network* ? (see part 3.2)

It will also be interesting to see:

- whether it evolves over time and depending on the market situation.
- Do we have sophisticated agents or “random” agents: do we have a random graph or not, do we have different patterns depending on market conditions?

3.2 The network representation

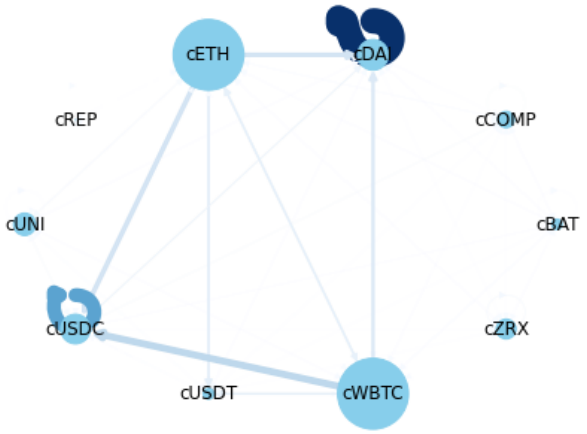
3.2.1 The lender-borrower network

For each snapshot, we construct the *collateral → borrow assets network* to model the dependency of borrowed assets on the collateral asset for each borrower. We defined the directed weight graph where:

- A *node* represents each pool (asset). The size of the node indicates the pool size in terms of cash (= the amount of underlying balance owned by a cToken contract) expressed in ETH.
- An *edge* measures the total amount in the current ETH price that draws from a collateral asset to the borrowed one. We calculate the amount between the collateral and borrowed assets as

$$amount(collateral \rightarrow borrow) = storedBorrowBalance \times \frac{cTokenBalance}{\Sigma\, cTokenBalance}$$

Example of the *collateral → borrow assets network* at the snapshot of 2021-02-24. The size of the circle indicates the total supply in ETH and the width of the line indicates the amount from collateral to borrowed assets.

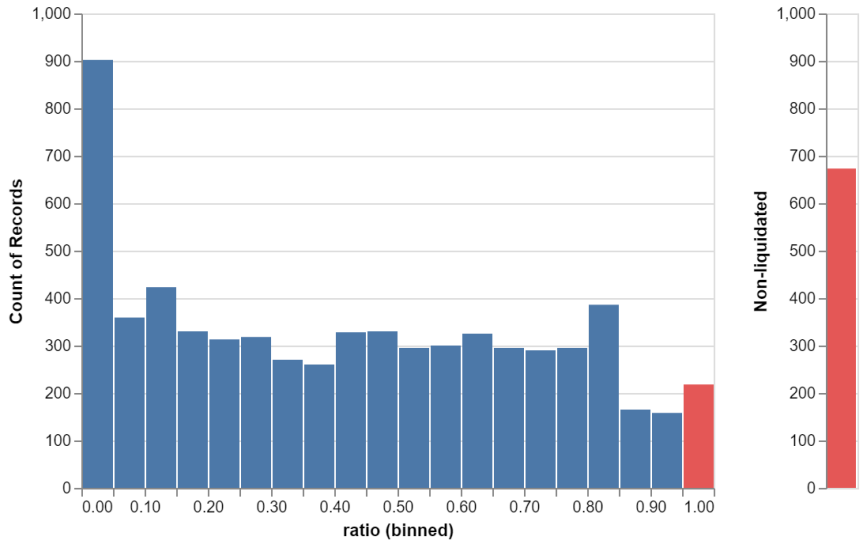


Note: We observed the self-loop on cDAI and cUSDC. It means that users deposit and borrow the same token at the same time. Those users were most likely farming the COMP reward token.

Who will get liquidated — and why?

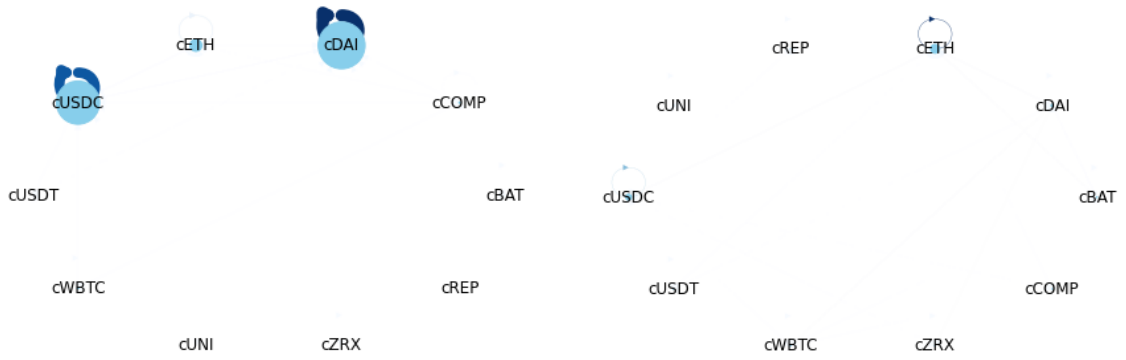
We calculate the *ratio* between borrow and collateral assets in ETH price to identify users (accounts) that almost get liquidated. Users who have the ratio close to 1 are risky to get liquidated. This measure is actually the inverse of the account *health* variable from the protocol. Ratio > 1 implies that borrowers that their assets are not yet liquidated by any liquidator.

The histograms shows the number of borrower with different ratios. The bar highlight in red are risky borrower wih ratio ≥ 0.95 and non-liquidated borrowers.



We focus on these two groups of borrows because they are risky users to liquidated and therefore we assume that they pose the risk to the protocol.

Here we plot node-link diagrams of two risky groups (left) borrower with the ratio ≥ 0.95 and (right) non-liquidated borrowers. The size of the circle indicates the total collateral assets (ETH) while the edge width indicates the amount between collateral and borrowed assets.



The most risky assets come from those who did yield farming. Notice that if this assets are liquidates then the collateral asserts (the size of the circle) is at risk.

The next challenge is to look into details on the behavior of borrowers and lenders.

3.2.2 Bipartite Graph

As pointed out previously, users are not connected directly to each other on Compound, neither are the pools. Nonetheless, users borrow in some pools and collateralize in other pools, which make the pools interconnected. Furthermore, some users may be impacted in a similar way by some market dynamics and some exogenous shocks even though they don't have a direct credit relationship as they have some portofolio similarities.

Therefore, to study the systemic risk of Compound, we want to create a bi-partite network model composed of **users as one node type and pools as another node type** and make simulations of cascading failure to describe and measure the risk propagation process and liquidations during crises within Compound.

It will also be interesting to project the bi-partite graph into one mode to see the relation structure within pools and also among users.

References

- Compound Market Risk Assessment https://gauntlet.network/reports/compound
- Compound Risk Management Dashboard https://gov.gauntlet.network/compound
- Default Cascades in Complex Networks: Topology and Systemic Risk https://www.nature.com/articles/srep02759

Appendix A : Liquidation in Compound

Executive Summary

1. Liquidator finds the unhealthy accounts from Compound API.
2. Liquidator calls LiquidateBorrow() with the sufficient amount of tokens to repay borrowers.
3. Compound uses liquidateBorrowInternal function to determine if this liquidation is valid.

To find accounts to liquidate, liquidators are expected to query the official Compound Finance API https://compound.finance/docs/api, https://api.compound.finance/api/v2/account). From here they can ask for accounts that are close to the liquidation boundary, by specifying the “max_health” variable. The official API is quite similar to the API provided by the Graph.

When a liquidator has found an account that they would like to liquidate, they can send a call to the liquidateBorrow(). This can be done in most programming languages through the help of a library that allows interacting with Web3 (such as Web3.py in Python).

Now to walk through, the function calls for liquidation. The only function that the end user can interact with and the first function called is the LiquidateBorrow() function which looks like the following:

```
function liquidateBorrow(address borrower, uint repayAmount, CTokenInterface cTokenCollateral)
external returns (uint) {
        (uint err,) = liquidateBorrowInternal(borrower, repayAmount, cTokenCollateral);
        return err;
}
```

There are two types of cTokens: CErc20 and CEther. The CErc20 wraps an underlying ERC-20 asset, while CEther simply wraps Ether itself. As such, the liquidation function is slightly different depending on the type, each of which is shown below.

From the CErc20 contract.
The liquidateBorrow function goes on to call the liquidateBorrowInternal function, which cannot be interacted directly by normal users:

```
function liquidateBorrowInternal
(address borrower, uint repayAmount, CTokenInterface cTokenCollateral) internal nonReentrant
returns (uint, uint) {
        uint error = accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
        // accrueInterest emits logs on errors, but we still want to log the fact  that  an
attempted liquidation failed
            return (fail(Error(error), FailureInfo.LIQUIDATE_ACCRUE_BORROW_INTEREST_FAILED), 0);
        }

        error = cTokenCollateral.accrueInterest();
        if (error != uint(Error.NO_ERROR)) {
            // accrueInterest emits logs on errors, but we still want to log the fact that an
attempted liquidation failed
            return (fail(Error(error), FailureInfo.LIQUIDATE_ACCRUE_COLLATERAL_INTEREST_FAILED),
0);
        }
```

```
        // liquidateBorrowFresh emits borrow-specific logs on errors, so we don't need to
        return liquidateBorrowFresh(msg.sender, borrower, repayAmount,
cTokenCollateral);
}
```

From the cToken contract.
The liquidation function has the following parameters:

- param borrower: The borrower of this cToken to be liquidated
- param cTokenCollateral: The market in which to seize collateral from the borrower
- param repayAmount: The amount of the underlying borrowed asset to repay
- return (uint, uint) An error code, which is 0 on success

The liquidator is the msg.sender account.

The accrue Interest function does the following: “Applies accrued interest to total borrows and reserves. This calculates interest accrued from the last checkpointed block up to the current block and writes new checkpoint to storage.”

When this accrue Interest function is called. It will check the last block when the interest rate was updated and add the new interest rate up to the current block.

1. `simpleInterestFactor = Δblocks × borrowRate`
2. `interestAccumulated = totalBorrows × simpleInterestFactor`
3. `totalBorrowsNew = totalBorrows + interestAccumulated` → This is the new borrow with the up-to-date interest rate.
4. `totalReservesNew = totalReserves + interestAccumulated × reserveFactor`

The next function which is called is “LiquidateBorrowFresh” first checks whether the liquidation is allowed to occur, which is the case if the following conditions are met:

1. The market for the token and the cToken is listed or the market is depreciated
2. That the liquidated account has indeed a shortfall of collateral
3. The liquidation amount is less than what is allowed by the closeFactor

Then it proceeds to verify that the markets for both the collateral and the borrowed assets are up to date by checking their accrual block numbers. Then it checks that the borrower is not the liquidator, that the repayment amount is larger than 0.

After all these checks are done, the contract tries to repay the collateral. If it succeeds it proceeds with seizing the liquidation incentive. Making sure that the amount of tokens seized is less than the amount of cTokens the liquidated user has, and runs an added check for re-entrancy. If no errors are encountered during the way the SeizeInternal() function handles the transfer of tokens from the liquidated user to the liquidator.

```
function liquidateBorrowFresh(address liquidator, address borrower, uint repayAmount,
CTokenInterface cTokenCollateral) internal returns (uint, uint) {
        /* Fail if liquidate not allowed */
        uint allowed = comptroller.liquidateBorrowAllowed(address(this), address(cTokenCollateral),
liquidator, borrower, repayAmount);
        if (allowed != 0) {
```

```
            return (failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.LIQUIDATE_COMPTROLLER_REJECTION, allowed), 0);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.LIQUIDATE_FRESHNESS_CHECK), 0);
        }

        /* Verify cTokenCollateral market's block number equals current block number */
        if (cTokenCollateral.accrualBlockNumber() != getBlockNumber()) {
            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.LIQUIDATE_COLLATERAL_FRESHNESS_CHECK),
0);
        }

        /* Fail if borrower = liquidator */
        if (borrower == liquidator) {
            return (fail(Error.INVALID_ACCOUNT_PAIR, FailureInfo.LIQUIDATE_LIQUIDATOR_IS_BORROWER),
0);
        }

        /* Fail if repayAmount = 0 */
        if (repayAmount == 0) {
            return (fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED,
FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_ZERO), 0);
        }

        /* Fail if repayAmount = -1 */
        if (repayAmount == uint(-1)) {
            return (fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED,
FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX), 0);
        }


        /* Fail if repayBorrow fails */
        (uint repayBorrowError, uint actualRepayAmount) = repayBorrowFresh(liquidator, borrower,
repayAmount);
        if (repayBorrowError != uint(Error.NO_ERROR)) {
            return (fail(Error(repayBorrowError), FailureInfo.LIQUIDATE_REPAY_BORROW_FRESH_FAILED),
0);
        }

        ////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /* We calculate the number of collateral tokens that will be seized */
        (uint amountSeizeError, uint seizeTokens) =
comptroller.liquidateCalculateSeizeTokens(address(this), address(cTokenCollateral),
actualRepayAmount);
        require(amountSeizeError == uint(Error.NO_ERROR),
"LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED");

        /* Revert if borrower collateral token balance < seizeTokens */
        require(cTokenCollateral.balanceOf(borrower) >= seizeTokens, "LIQUIDATE_SEIZE_TOO_MUCH");

        // If this is also the collateral, run seizeInternal to avoid re-entrancy, otherwise make an
external call
        uint seizeError;
        if (address(cTokenCollateral) == address(this)) {
            seizeError = seizeInternal(address(this), liquidator, borrower, seizeTokens);
        } else {
            seizeError = cTokenCollateral.seize(liquidator, borrower, seizeTokens);
        }

        /* Revert if seize tokens fails (since we cannot be sure of side effects) */
        require(seizeError == uint(Error.NO_ERROR), "token seizure failed");

        /* We emit a LiquidateBorrow event */
        emit LiquidateBorrow(liquidator, borrower, actualRepayAmount, address(cTokenCollateral),
seizeTokens);

        /* We call the defense hook */
```

```
        // unused function
        // comptroller.liquidateBorrowVerify(address(this), address(cTokenCollateral), liquidator,
borrower, actualRepayAmount, seizeTokens);

        return (uint(Error.NO_ERROR), actualRepayAmount);
    }
```

Liquidation Incentive

The additional collateral given to liquidators is an incentive to perform liquidation of underwater accounts. For example, if the liquidation incentive is 1.1, liquidators receive an extra 10% of the borrowers collateral for every unit they close.
Gas prices may erode the liquidation profits, further for each dollar or loan repaid, for example to liquidate a $1million loan the liquidator needs $1million to do it. This capital requirement can be overcome with the use of flash loans by first borrowing the necessary capital to liquidate, liquidating the position and then immediately reselling the cTokens to close the flash loan.
Further it is always more pertinent to liquidate the account that is more in debt, as gas fees are fixed and the reward for a liquidator is a percentage of the collateral of the liquidated borrower. So the profit is a linear function of the amount of debt with a negative intercept.

```
function liquidationIncentiveMantissa() view returns (uint)
```

`RETURN`: The liquidationIncentive, scaled by 1e18, is multiplied by the closed borrow amount from the liquidator to determine how much collateral can be seized.

Appendix B: Database schemas

Market

Database Schema	Description
markets(block: {number: 'xxxxxxxx'}) {	
borrowRate	Yearly borrow rate. With 2102400 blocks per year
cash	The cToken contract balance of ERC20 or ETH
collateralFactor	Collateral factor determining how much one can borrow
exchangeRate	Exchange rate of tokens / cTokens
interestRateModelAddress	Address of the interest rate model
name	Name of the cToken
reserves	Reserves stored in the contract
supplyRate	Yearly supply rate. With 2104400 blocks per year
symbol	CToken symbol
id	CToken address
totalBorrows	Borrows in the market
totalSupply	CToken supply. CTokens have 8 decimals
underlyingAddress	Underlying token address
underlyingName	Underlying token name
underlyingPrice	

underlyingSymbol	Underlying price of token in ETH (ex. 0.007 DAI)
accrualBlockNumber	Underlying token symbol
blockTimestamp	Block the market is updated to
borrowIndex	Timestamp the market was most recently updated
reserveFactor	The history of the markets borrow index return (Think S&P 500)
underlyingPriceUSD	The factor determining interest that goes to reserves
underlyingDecimals	Underlying token price in USD
}	Underlying token decimal length

Example table: market_2020-01-01.csv

	borrowRate	cash	collateralFactor	exchangeRate	interestRateModelAddress	name	reserves	supplyRate	symbol	
0	0.021170	1.516208e+06	0.50	0.020036	0xbae04cbf96391086dc643e842b517734e214d698	Compound Augur	10.786920	0.000074	cREP	0x
1	0.089366	1.815307e+07	0.75	0.020817	0xc64c4cba055efa614ce01f4bad8a9f519c4f8fab	Compound USD Coin	83890.143560	0.036031	cUSDC	0x
2	0.022070	3.486976e+05	0.75	0.020011	0xbae04cbf96391086dc643e842b517734e214d698	Compound Ether	6.360946	0.000137	cETH	0x
3	0.041549	1.036304e+07	0.75	0.020051	0xec163986cc9a6593d6addcbff5509430d348030f	Compound Dai	151638.770109	0.039646	cDAI	0x5
4	0.042792	4.076504e+06	0.60	0.020208	0xbae04cbf96391086dc643e842b517734e214d698	Compound Basic Attention Token	743.504409	0.002926	cBAT	0x
5	0.031739	6.024811e+06	0.60	0.020047	0xbae04cbf96391086dc643e842b517734e214d698	Compound 0x	1650.652844	0.001118	cZRX	0x
6	0.061946	1.290757e+02	0.00	0.020062	0xbae04cbf96391086dc643e842b517734e214d698	Compound Wrapped BTC	0.046021	0.007797	cWBTC	0x
7	0.093230	3.726216e+06	0.75	0.021133	0xa1046abfc2598f48c44fb320d281d3f3c0733c9a	Compound Dai	26086.483951	0.032051	cDAI	0

8 rows × 22 columns

Account & AccountCToken - We select the highlighted variable to construct the lending and borrowing tables

Databased Schema	Description
accounts(block: {number: 'xxxxxxxx'}) {	
id	User ETH address
countLiquidated	Count user has been liquidated
countLiquidator	Count user has liquidated others
hasBorrowed	True if user has ever borrowed
health	
totalBorrowValueInEth	
totalCollateralValueInEth	
tokens {	
symbol	Symbol of the cToken
accrualBlockNumber	Block number this asset was updated at in the contract
enteredMarket	True if user is entered, false if they are exited
cTokenBalance	CToken balance of the user → Lending amount
totalUnderlyingSupplied	Total amount of underlying supplied
totalUnderlyingRedeemed	Total amount of underling redeemed

accountBorrowIndex totalUnderlyingBorrowed totalUnderlyingRepaid storedBorrowBalance supplyBalanceUnderlying lifetimeSupplyInterestAccrued borrowBalanceUnderlying lifetimeBorrowInterestAccrued } }	The value of the borrow index upon users last interaction Total amount underlying borrowed, exclusive of interest Total amount underlying repaid Current borrow balance stored in contract → Borrowing amount

We convert the amount of cTokenBalance and storedBorrowBalance to ETH currency with the current market price.

- cTokenBalanceETH = cTokenBalance * exchangeRate * underlyingPrice
- storedBorrowBalanceETH = storedBorrowBalance * underlyingPrice

We provide the accounts and tokens data into two separated tables:

Example table: accounts_2020-01-01.csv

	id	countLiquidated	countLiquidator	hasBorrowed	health	totalBorrowValueInEth	totalCollateralValueInEth
0	0x0000000000000000000000000000000000000000	0	0	False	NaN	0.000000	1.710899e-03
1	0x00000000af5a61acaf76190794e3fdf1289288a1	0	49	False	NaN	0.000000	5.764875e-03
2	0x000000aaee6a496aaf7b7452518781786313400f	5	22	True	0.007845	0.000000	7.844641e-03
3	0x000000f54395c554346bfd24e6a1ccd90b881a4e	0	0	False	NaN	0.000000	2.705669e-03
4	0x000000fe6d4bc2de2d0b0e6fe47f08a28ed52f91	0	0	False	NaN	0.000000	8.554493e-04
...	...	...	...	...	...	...	...
13992	0xffec5ea11fa72dcfac5ae54da1778d9db46fcade	0	0	True	1.097033	0.156371	1.715443e-01
13993	0xfff7e32d50e2dfc9764190ff7b6b4cce878b1311	0	0	False	NaN	0.000000	9.517614e-02
13994	0xfff95dea424c0d7a25471982610a2485f302fb54	0	0	False	NaN	0.000000	3.094308e-09
13995	0xffff2c1d5fa3f7dc16902c3f4dfc56b138474d3e	0	0	True	2.638746	1.680757	4.435091e+00
13996	0xffffa2c37e4aa15eb223d2a3cf9f3021cde1d68a	0	0	False	NaN	0.000000	1.140189e-01

13997 rows × 7 columns

Example table: tokens_2020-01-01.csv

	id	symbol	accrualBlockNumber	cTokenBalance	storedBorrowBalance	cTokenBalanceETH	storedBorrowBala
0	0x0000000000000000000000000000000000000000	cREP	8671787	0.244000	0.00	3.255208e-04	
1	0x0000000000000000000000000000000000000000	cUSDC	8671787	2.500000	0.00	3.954174e-04	
2	0x0000000000000000000000000000000000000000	cETH	8671787	0.012000	0.00	2.401314e-04	
3	0x0000000000000000000000000000000000000000	cBAT	8671787	15.000000	0.00	4.525939e-04	
4	0x0000000000000000000000000000000000000000	cZRX	8671787	13.380000	0.00	3.759509e-04	
...	...	...	...	...	...	...	
25218	0xfff95dea424c0d7a25471982610a2485f302fb54	cDAI	9087046	0.000026	0.00	3.914383e-09	
25219	0xffff2c1d5fa3f7dc16902c3f4dfc56b138474d3e	cETH	8451053	209.908215	0.00	4.200463e+00	
25220	0xffff2c1d5fa3f7dc16902c3f4dfc56b138474d3e	cWBTC	8451084	0.000000	0.03	0.000000e+00	
25221	0xffff2c1d5fa3f7dc16902c3f4dfc56b138474d3e	cDAI	8451048	10736.957691	0.00	1.625234e+00	
25222	0xffffa2c37e4aa15eb223d2a3cf9f3021cde1d68a	cDAI	8842826	952.888162	0.00	1.442370e-01	

25223 rows × 7 columns

The data tables (market, accounts, and tokens) can be found in this shared folder: https://drive.google.com/drive/folders/1bEu0T5PQgPk19ULdg6L3GNS0u7B2WJAz?usp=sharing

Liquidation Event

Database Schema	Description
id	Transaction hash concatenated with log index
account	Account being liquidated (borrower)
blockNumber	Block number
blockTime	Block time
symbol	cToken that was seized as collateral
amount	cTokens seized
amountETH	cTokens seized in ETH
amountUSD	cTokens seized in USD
liquidator	Liquidator receiving tokens
liquidationSymbol	Symbol of the underlying asset repaid through liquidation
liquidationRepay	Underlying cToken amount that was repaid by liquidator
liquidationRepayETH	Repaid cToken amount in ETH
liquidationRepayUSD	Repaid cToken amount in USD

Liquidation event table is available in this Google Sheet: https://docs.google.com/spreadsheets/d/1v_l73uCkNYCWzCxgkFuFjLtoV5Yb7ztrUJhecwbtig4/edit?usp=sharing