Comparing Chainlink & Maker update times

Tellor's Oracle will be used if an one of Chainlink or Maker does not update for 3 hours or longer

My goal is to look into the past histories of these oracles' update times and see what the time differences were, and basically how often we can expect Tellor's oracle to be used.

Web3 is a great way to connect to smart contracts, using the address and the contract's ABI, it handles the connection for you so you can directly call the contract's functions.

In [1]:
```
from web3 import Web3
infura_link = 'https://mainnet.infura.io/v3/3e5c6b2a48494d9a921a52ec1cc0a8ff'
w3 = Web3(Web3.HTTPProvider(infura_link))
```

The addresses for contracts we will be calling were found on the Chainlink and Maker websites for their ETH/USD oracle price update. Web3 only accepts addresses in checksum format, so I convert them here as well.

We ended up not needing the maker address as I used Dune for their historical update data, but it's nice to have in case we do decide to connect to their contract in the future

In [2]:
```
chainlink_add = "0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419"
chainlink_add_cs = w3.toChecksumAddress(chainlink_add)
maker_add = "0x81FE72B5A8d1A857d176C3E7d5Bd2679A9B85763"
maker_add_cs = w3.toChecksumAddress(maker_add)
```

The ABI's are needed to connect to the contract. These were sourced from etherscan. Then, using Web3, we can access the contracts easily through a single line of code. This creates a contract object, that we can communicate with through the contract's functions.

In [3]:
```
chainlink_abi = '[{"inputs":[{"internalType":"address","name":"_aggregator","type":"addr
maker_abi = '[{"inputs":[{"internalType":"address","name":"src_","type":"address"}],"pay

contract_cl = w3.eth.contract(address = chainlink_add_cs, abi = chainlink_abi)
contract_m = w3.eth.contract(address = maker_add_cs, abi = maker_abi)
```

Chainlink

Chainlink makes it easy to directly access their data. Using `latestRoundData()` , you can find out the time and ID of the latest round. It returns a list of mutliple data points: round ID, answer, started at, ended at, and answered in round. We are mainly looking at the "updated at" data point.

In [4]:
```
latestData = contract_cl.functions.latestRoundData().call()
print("chainlink: ", latestData)
```

```
chainlink:  [92233720368547768476, 334573380480, 1631624118, 1631624118, 9223372036854
7768476]
```

Using the returns from the function, we can create lists we will later add to. If we want to store values such as price, we can easily do that.

Want to change the structure of this so I start out at `round_id_0` , create an array going backwards up to the desired upper limit round ID, then apply the `getRoundData()` function to each roundID. This will be faster and more streamlined, and it's included below.

In [6]:
```
round_id_0 = latestData[0]
price_0 = latestData[1]
update_time_0 = latestData[3]

round_ids = []
prices = []
update_times_raw = []
update_times_utc = []

round_ids.append(round_id_0)
prices.append(price_0)
update_times_raw.append(update_time_0)
```

Going to be working with dates, so will be using the datetime module. This makes it easier to find differences between timestamps, which is what I am interested in recording.

The end date was tricky. I wanted to go back to a time that matched the earliest time I accessed of the Maker oracle (11/15/2019). However, the contract seems to not work past 5/11/2021, recording the timestamps as 1/1/1970 and the price of ETH as 0. I spoke to Chris about this and he said it could be they implemented this structure in May, so it can't go back any further than that. I wonder if I change the structure as I discussed above, if it would show more data.

The problem with incrementing in a loop is that if one datapoint is a bad apple, the rest can't follow. I think that structural change will help me see if it's just one data point that is tripping up the loop, or if there truly is no data to access earlier than that.

I am also going to look more into Chainlink's history and see if I am able to find it on Dune further back than May.

In [8]:
```
from datetime import datetime, timedelta
update_times_utc.append(datetime.utcfromtimestamp(update_time_0))

import numpy as np
import pandas as pd

current_date = str(datetime.utcfromtimestamp(update_time_0))
current_date = current_date.split()[0]
#end_date = '2019-11-15'
end_date = '2021-05-12'
```

I made a helper function, `time_convert()` , to convert times from raw data to utc for easier parsing/reading

In [9]:
```
def time_convert(raw_time):
    return datetime.utcfromtimestamp(raw_time)
```

The main function, `inc_rounds()` uses the current round ID as a starting point for backwards incrementing. Starts going backwards from current block, getting the previous blockId, and then getting update times from each block. This is the system Chris used in his Oracle Diffs project, but like I mentioned above, I am interested in structuring it in a different way that I think will be faster and less prone to error.

In [12]:
```
def inc_rounds(curr_round_data, end_date, time_arr):

    curr_round_id = curr_round_data[0]
    current_date = str(time_convert(curr_round_data[3])).split()[0]

    while current_date != end_date:
        curr_round_id = curr_round_id - 1
        historicalData = contract_cl.functions.getRoundData(curr_round_id).call()
        update_time_raw = historicalData[3]
        time_arr.append(time_convert(update_time_raw))

        #this iteration is only focused on time, but price and ID analysis will be usefu
        #round_ids.append(historicalData[0])
        #prices.append(historicalData[1])


        current_date = str(time_convert(update_time_raw)).split()[0]
        print("time: ",time_convert(update_time_raw), "price: ", historicalData[1], "rou
```

In [13]:
```
inc_rounds(latestData, end_date, update_times_utc)
```

```
time:  2021-09-14 12:34:21 price:  333652362260 round ID:  92233720368547768475
time:  2021-09-14 12:20:26 price:  331372820353 round ID:  92233720368547768474
time:  2021-09-14 12:07:38 price:  333193160053 round ID:  92233720368547768473
time:  2021-09-14 11:57:21 price:  333138780785 round ID:  92233720368547768472
time:  2021-09-14 11:44:46 price:  332339478953 round ID:  92233720368547768471
time:  2021-09-14 10:55:08 price:  334196000000 round ID:  92233720368547768470
time:  2021-09-14 09:55:11 price:  332552224273 round ID:  92233720368547768469
time:  2021-09-14 09:37:52 price:  333448508578 round ID:  92233720368547768468
time:  2021-09-14 08:55:26 price:  335236000000 round ID:  92233720368547768467
time:  2021-09-14 08:54:11 price:  335187802433 round ID:  92233720368547768466
time:  2021-09-14 08:41:47 price:  333498264313 round ID:  92233720368547768465
time:  2021-09-14 08:02:07 price:  331660523714 round ID:  92233720368547768464
time:  2021-09-14 07:55:28 price:  330879153198 round ID:  92233720368547768463
time:  2021-09-14 06:57:50 price:  330805025959 round ID:  92233720368547768462
time:  2021-09-14 06:55:50 price:  330804284732 round ID:  92233720368547768461
time:  2021-09-14 06:46:07 price:  330554691289 round ID:  92233720368547768460
time:  2021-09-14 05:56:56 price:  328824521251 round ID:  92233720368547768459
time:  2021-09-14 05:55:56 price:  328824521251 round ID:  92233720368547768458
time:  2021-09-14 04:55:33 price:  329012000000 round ID:  92233720368547768457
```

Testing the other method of applying the function to each element in an array via a simple list comprehension, it was a lot faster than I had hoped but still couldn't access any data before may 11th, said there was no data there to get. Makes sense with Chris' hypothesis. Apply function to entire set of roundIDs and then put into pandas dataframe where later analysis would be streamlined and simple. Going to use parts of this system for managing our monitoring data.

In [14]:
```
##test of other method

import numpy as np
round_id_arr = np.arange(round_id_0 - 8000, round_id_0)
```

In [15]:
```
historicalData_arr = [contract_cl.functions.getRoundData(i).call() for i in round_id_arr
df = pd.DataFrame(historicalData_arr, columns =["roundID", "answer", "started", "ended"
In [16]: df
```

Out[16]:

	roundID	answer	started	ended	answeredRound
0	92233720368547760476	227782002350	1621844970	1621844970	92233720368547760476
1	92233720368547760477	226613587152	1621845141	1621845141	92233720368547760477
2	92233720368547760478	227940195595	1621845984	1621845984	92233720368547760478
3	92233720368547760479	228502802012	1621846530	1621846530	92233720368547760479
4	92233720368547760480	227304794425	1621847604	1621847604	92233720368547760480
...	...	...	...	...	...
7995	92233720368547768471	332339478953	1631619886	1631619886	92233720368547768471
7996	92233720368547768472	333138780785	1631620641	1631620641	92233720368547768472
7997	92233720368547768473	333193160053	1631621258	1631621258	92233720368547768473
7998	92233720368547768474	331372820353	1631622026	1631622026	92233720368547768474
7999	92233720368547768475	333652362260	1631622861	1631622861	92233720368547768475

8000 rows × 5 columns

By going through the time arrays and using the Datetime module to calculate differences, we can see how many times since our "start date" the oracle has had a difference of 3 hours or more in between udpates.

```
In [18]: import math

def aggregate_diff(arr):
    count = 0
    for i in range(1, len(arr) - 1):
        first = arr[i - 1]
        last = arr[i]
        delta = first- last
        diff = float((delta.total_seconds()) / 3600)
        if diff >= 3:
            count+=1
    return count

print("amount of times chainlink diff was greater than 3 hours since ", str(end_date), '
```

```
amount of times chainlink diff was greater than 3 hours since  2021-05-12 : 0
```

Maker Data

Maker doesn't have a function that directs you to the next "round" or block like Chainlink does. Because of this, I had to approach their data a little differently.

Dune Analytics is a service that runs crypto databases available for query. Luckily, Maker's oracle contract, and all the data regarding when it has been called, are up there. Originally, I thought the `zzz()` function would give the correct answer, but that function only gives timestamps, and the data is on when the function was called. So, the `poke()` function is the one that updates the oracle, and I wrote a quick query getting all of the call data for that function, making sure to filter it for successful calls that actually updated the oracle. According to their docs, their oracle is supposed to be updated every hour. They keep it from updating with higher frequency than that, but sometimes it dips into lower frequency.

Pandas is a python data analysis library originally used for financial data that is now applied to a bunch of fields. I used it to import the csv. When we expand our data analysis to include more than just time stamps (prices, further analysis, etc.) pandas will make it easy to create dataframes ready for analysis right from a csv or database source.

In [19]:

```
import pandas as pd

data = pd.read_csv('/Users/laure/Downloads/fd7ef889-2907-4438-b7bf-9c75b574561c.csv')
s = data['call_block_time']
new_dates = [datetime.fromisoformat(i) for i in s]

print("maker oracle time difference since 11/15/2019: ", aggregate_diff(new_dates))
```

```
maker oracle time difference since 11/15/2019:  537
```

There was large gap in data from 5/5/21 to 5/21/21, so I started from there onward to give us a side by side comparison of what

In [20]:

```
print("maker oracle time difference since", new_dates[32452], ": ", aggregate_diff(new_(
```

```
maker oracle time difference since 2021-05-21 15:05:05+00:00 :  10
```

Conclusion

What I found was that since May, in a system where Tellor is being used as an oracle when Chainlink or Maker don't contribute to Reflexer's median price for 3 hours, Tellor would have been brought in as backup a total of 10 times, 0 for Chainlink and 10 for Maker.

This was between mid may and 9/7/21, when the Dune data was grabbed.

In [ ]:

```
```