<https://chatgpt.com/share/8388a0e5-649d-4991-b4e3-2e458432070e>

as per what i understand by this line "Cache Lookup: Before performing the computation, the cache is checked for an existing result." , i doubt this may not work as per expectation as here you are checking for the cache lookup , before applying logic on current data , as time passes there may be matching key atleast 1 for each of is\_bullish\_abandoned\_baby and is\_shooting\_star and is\_hammer in the allotted cache with time to live of previous 5 seconds.

ChatGPT

You're right. If the data changes frequently and the caching key is based solely on the last candle's index, the cache might not behave as expected because it could hold stale data that doesn't reflect the current state accurately.

To address this, we need a more granular approach to caching that ensures the cache is always relevant to the most recent data. One way to achieve this is by incorporating a hash of the last candle's data as part of the cache key. This way, even if the time is within the TTL, the cache will only return a result if the exact same data was processed recently.

Here's how you can adjust the implementation:

**Updated Implementation with Data Hashing**

python

Copy code

import pandas as pd

import cachetools

import time

from cachetools import TTLCache

import hashlib

# Initialize a TTL cache with a max size of 100 and a TTL of 5 seconds

cache = TTLCache(maxsize=100, ttl=5)

def hash\_candle(candle):

candle\_str = f"{candle['Open']}{candle['Close']}{candle['High']}{candle['Low']}"

return hashlib.md5(candle\_str.encode()).hexdigest()

def is\_hammer(data):

last\_candle = data.iloc[-1]

cache\_key = (hash\_candle(last\_candle), 'is\_hammer')

if cache\_key in cache:

return cache[cache\_key]

body\_size = abs(last\_candle['Close'] - last\_candle['Open'])

lower\_shadow = last\_candle['Low'] - min(last\_candle['Close'], last\_candle['Open'])

upper\_shadow = last\_candle['High'] - max(last\_candle['Close'], last\_candle['Open'])

result = lower\_shadow > 2 \* body\_size and upper\_shadow < 0.1 \* body\_size

cache[cache\_key] = result

return result

def is\_shooting\_star(data):

last\_candle = data.iloc[-1]

cache\_key = (hash\_candle(last\_candle), 'is\_shooting\_star')

if cache\_key in cache:

return cache[cache\_key]

body\_size = abs(last\_candle['Close'] - last\_candle['Open'])

upper\_shadow = last\_candle['High'] - max(last\_candle['Close'], last\_candle['Open'])

lower\_shadow = min(last\_candle['Close'], last\_candle['Open']) - last\_candle['Low']

result = upper\_shadow > 2 \* body\_size and lower\_shadow < 0.1 \* body\_size

cache[cache\_key] = result

return result

def is\_bullish\_abandoned\_baby(data):

if len(data) < 3:

return False

first\_candle = data.iloc[-3]

second\_candle = data.iloc[-2]

third\_candle = data.iloc[-1]

cache\_key = (hash\_candle(first\_candle), hash\_candle(second\_candle), hash\_candle(third\_candle), 'is\_bullish\_abandoned\_baby')

if cache\_key in cache:

return cache[cache\_key]

is\_bearish\_first = first\_candle['Close'] < first\_candle['Open']

is\_doji\_second = abs(second\_candle['Close'] - second\_candle['Open']) < (second\_candle['High'] - second\_candle['Low']) \* 0.1

is\_bullish\_third = third\_candle['Close'] > third\_candle['Open']

gap\_down = second\_candle['High'] < first\_candle['Low']

gap\_up = third\_candle['Low'] > second\_candle['High']

result = is\_bearish\_first and is\_doji\_second and is\_bullish\_third and gap\_down and gap\_up

cache[cache\_key] = result

return result

# Example usage

# Assuming 'data' is a pandas DataFrame with the required columns

# while True:

# print(is\_hammer(data))

# print(is\_shooting\_star(data))

# print(is\_bullish\_abandoned\_baby(data))

# time.sleep(1) # Simulate real-time data updates

**Explanation**

1. **Hashing Function**: hash\_candle function creates a hash based on the candle's open, close, high, and low values. This ensures that the cache key is unique to the specific data.
2. **Cache Key**: The cache key now includes the hash of the last candle's data, ensuring that the cached result is specific to the exact data being processed.
3. **is\_bullish\_abandoned\_baby**: This function now uses the hashes of the last three candles to create a unique cache key.

**Benefits**

* **Data Specificity**: The cache keys are specific to the actual data being processed, reducing the risk of stale data.
* **Performance**: The TTL cache still prevents redundant computations within the TTL period, but only for identical data.
* **Flexibility**: You can adjust the TTL and cache size as needed to balance performance and memory usage.

This approach ensures that your cache remains relevant and accurate even as the data changes frequently.

4o

apply same for below code "def calculate\_fibonacci\_levels(highest\_close, lowest\_close): fibonacci\_levels = [0, 0.236, 0.382, 0.5, 0.618, 0.786, 1.0] retracement\_range = highest\_close - lowest\_close fibonacci\_values = [lowest\_close + level \* retracement\_range for level in fibonacci\_levels] return fibonacci\_values def calculate\_fibonacci\_levels\_for\_downtrend(highest\_close, lowest\_close): fibonacci\_levels = [0, 0.236, 0.382, 0.5, 0.618, 0.786, 1.0] retracement\_range = highest\_close - lowest\_close fibonacci\_values = [highest\_close - level \* retracement\_range for level in fibonacci\_levels] return fibonacci\_values"

ChatGPT

To apply caching to the functions calculate\_fibonacci\_levels and calculate\_fibonacci\_levels\_for\_downtrend, we'll use a similar approach as before. We'll create a hash key based on the input parameters to ensure the cache is specific to the exact inputs.

Here's the updated implementation using the cachetools library with TTL caching:

**Updated Implementation**

python

Copy code

import cachetools

import hashlib

from cachetools import TTLCache

# Initialize a TTL cache with a max size of 100 and a TTL of 5 seconds

cache = TTLCache(maxsize=100, ttl=5)

def hash\_parameters(\*args):

hash\_input = ''.join(map(str, args))

return hashlib.md5(hash\_input.encode()).hexdigest()

def calculate\_fibonacci\_levels(highest\_close, lowest\_close):

cache\_key = (hash\_parameters(highest\_close, lowest\_close), 'calculate\_fibonacci\_levels')

if cache\_key in cache:

return cache[cache\_key]

fibonacci\_levels = [0, 0.236, 0.382, 0.5, 0.618, 0.786, 1.0]

retracement\_range = highest\_close - lowest\_close

fibonacci\_values = [lowest\_close + level \* retracement\_range for level in fibonacci\_levels]

cache[cache\_key] = fibonacci\_values

return fibonacci\_values

def calculate\_fibonacci\_levels\_for\_downtrend(highest\_close, lowest\_close):

cache\_key = (hash\_parameters(highest\_close, lowest\_close), 'calculate\_fibonacci\_levels\_for\_downtrend')

if cache\_key in cache:

return cache[cache\_key]

fibonacci\_levels = [0, 0.236, 0.382, 0.5, 0.618, 0.786, 1.0]

retracement\_range = highest\_close - lowest\_close

fibonacci\_values = [highest\_close - level \* retracement\_range for level in fibonacci\_levels]

cache[cache\_key] = fibonacci\_values

return fibonacci\_values

# Example usage

# Assuming highest\_close and lowest\_close are defined

# highest\_close = 150.0

# lowest\_close = 100.0

# print(calculate\_fibonacci\_levels(highest\_close, lowest\_close))

# print(calculate\_fibonacci\_levels\_for\_downtrend(highest\_close, lowest\_close))

**Explanation**

1. **Hashing Function**: hash\_parameters function creates a hash based on the input parameters (highest\_close and lowest\_close). This ensures that the cache key is unique to the specific inputs.
2. **Cache Key**: The cache key now includes the hash of the input parameters and the function name, ensuring that the cached result is specific to the exact inputs being processed.
3. **Cache Lookup**: Before performing the computation, the cache is checked for an existing result.
4. **Cache Insertion**: If the result is not cached, the computation is performed, and the result is stored in the cache with the appropriate key.

**Benefits**

* **Data Specificity**: The cache keys are specific to the actual inputs being processed, reducing the risk of stale data.
* **Performance**: The TTL cache prevents redundant computations within the TTL period but only for identical inputs.
* **Flexibility**: You can adjust the TTL and cache size as needed to balance performance and memory usage.

This approach ensures that your cache remains relevant and accurate even as the inputs change frequently