In [43]:

```
import asyncio
import aiohttp
import pandas as pd
import nest_asyncio
import concurrent.futures
import time
import csv
import os
from datetime import datetime, timedelta

# Apply nest_asyncio for Jupyter Notebooks
nest_asyncio.apply()

# Constants and API endpoint
API_URL = "https://api.binance.com/api/v3/klines"
CURRENCIES = ["XRP", "SOL", "ETH", "BTC", "BCH"]  # Changed to base currency onl
YEARLY_FILE_2024 = "E:/Data/crypto_2024.csv"  # Only 2024 file for incremental l
AUDIT_FILE = "E:/Data/audit_table.csv"

# Function to load the audit table
def load_audit_table():
    if os.path.exists(AUDIT_FILE):
        audit_df = pd.read_csv(AUDIT_FILE)
    else:
        audit_df = pd.DataFrame(columns=["Symbol", "LastLoaded"])
    return audit_df

# Function to update the audit table
def update_audit_table(symbol, last_loaded):
    audit_df = load_audit_table()
    audit_df = audit_df[audit_df['Symbol'] != symbol]  # Remove old entry for th
    new_entry = pd.DataFrame({"Symbol": [symbol], "LastLoaded": [last_loaded]})
    audit_df = pd.concat([audit_df, new_entry], ignore_index=True)
    audit_df.to_csv(AUDIT_FILE, index=False)
    print(f"Audit table updated for {symbol}: LastLoaded set to {last_loaded}")

# Function to fetch historical data
async def fetch_data(session, symbol, start, end):
    params = {"symbol": f"{symbol}USDT", "interval": "1m", "startTime": start, "
    retries = 5
    for attempt in range(retries):
        try:
            async with session.get(API_URL, params=params) as response:
                if response.status == 200:
                    data = await response.json()
                    return [
                        {"Timestamp": datetime.fromtimestamp(item[0] / 1000), "P
                        for item in data
                    ]
                else:
                    print(f"Error {response.status} for {symbol}, attempt {attem
        except aiohttp.ClientError as e:
            print(f"Error fetching data for {symbol}, attempt {attempt + 1}: {e}
        await asyncio.sleep(2 ** attempt)  # Exponential backoff
    return []

# Function to append data to the 2024 file
def append_to_csv(data):
```

```
    file_exists = os.path.exists(YEARLY_FILE_2024)
    with open(YEARLY_FILE_2024, mode="a", newline="") as f:
        writer = csv.DictWriter(f, fieldnames=["Timestamp", "Price", "Symbol"])
        if not file_exists:
            writer.writeheader()
        writer.writerows(data)

# Function to fetch and save data incrementally based on the audit table
async def fetch_and_save_data(symbol, start_date, end_date):
    data = []
    async with aiohttp.ClientSession() as session:
        current = start_date
        while current < end_date:
            end = min(current + timedelta(minutes=1000), end_date)
            batch = await fetch_data(session, symbol, int(current.timestamp() *
            if batch:
                data.extend(batch)
                current = batch[-1]["Timestamp"] + timedelta(minutes=1)
                update_audit_table(symbol, current.strftime("%Y-%m-%d %H:%M:%S")
            else:
                current = end  # Move forward even if no data returned
    append_to_csv(data)
    print(f"Data saved for {symbol} from {start_date.year}")

# Main function with concurrency
async def main():
    audit_df = load_audit_table()
    end_date_2024 = datetime(2024, 12, 31, 23, 59)  # Adjust this to the current

    tasks = []
    start_time = time.time()

    # Using ThreadPoolExecutor to parallelize fetching for each symbol
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        for symbol in CURRENCIES:
            last_loaded_str = audit_df[audit_df["Symbol"] == symbol]["LastLoaded
            last_loaded = datetime.strptime(last_loaded_str, "%Y-%m-%d %H:%M:%S"

            # Start from last loaded timestamp if available, else from start of
            start_date = last_loaded + timedelta(minutes=1) if last_loaded else
            tasks.append(fetch_and_save_data(symbol, start_date, end_date_2024))

        # Execute all tasks concurrently
        await asyncio.gather(*tasks)

    end_time = time.time()
    print(f"Execution time: {end_time - start_time:.2f} seconds")

# Run the main function with asyncio
if __name__ == "__main__":
    asyncio.run(main())
```

```
Audit table updated for BCH: LastLoaded set to 2024-11-05 20:28:00
Audit table updated for BTC: LastLoaded set to 2024-11-05 20:28:00
Audit table updated for ETH: LastLoaded set to 2024-11-05 20:28:00
Audit table updated for SOL: LastLoaded set to 2024-11-05 20:28:00
Audit table updated for XRP: LastLoaded set to 2024-11-05 20:28:00
Data saved for BCH from 2024
Data saved for XRP from 2024
Data saved for ETH from 2024
Data saved for BTC from 2024
Data saved for SOL from 2024
Execution time: 17.55 seconds
```

In [ ]: