University of Kragujevac

Artificial Intelligence

Project for an Processing Large Volumes of Data (VI-23)

Cryptocurrency Data Processing and Analysis

Student:

Babaev Bogdan 0866511

Professor: Ana Kaplarević-Mališić

Kragujevac, September 2025

Contents

1. Project Overview

Objective This project, developed for the course *Processing Large Volumes of Data (VI-23)*, aims to build a scalable Big Data application to collect, process, and analyze historical price data for Bitcoin (BTC) and Ethereum (ETH). The goals are:

- Collect large-scale cryptocurrency data from an external API.
- Aggregate data using Apache Spark's RDDs and SparkSQL.
- Optimize processing with parallel computing techniques.
- Ensure scalability and data security for real-world applications.

Solution Summary The application fetches BTC and ETH price and volume data from January 1, 2023, to January 1, 2025, via the CryptoCompare API. It processes the data using Apache Spark's structured streaming with 5-minute window aggregations, stores results in a SQLite database, and applies AES encryption for security. The project demonstrates Big Data principles, focusing on scalability, performance, and financial data analysis.

2. Methodology and Implementation

2.1 Environment Setup

The project runs in Google Colab with Google Drive for persistent storage. Libraries such as pandas, pyspark, requests, and pycryptodome are installed to handle data processing, API calls, and encryption. Data is stored in /content/drive/MyDrive/crypto_analysis_sqlcube.

```python
# Block 1: Environment and Google Drive Setup
from google.colab import drive
import os

drive.mount('/content/drive')
PROJECT_DIR = "/content/drive/MyDrive/crypto_analysis_sqlcube"
os.makedirs(PROJECT_DIR, exist_ok=True)
```

2.2 Database Initialization

A SQLite database (crypto.db) is set up to store aggregated data in three tables: Cryptocurrency, Date, and PriceData. Write-Ahead Logging (WAL) is enabled for better concurrency and performance.

```python
# Block 2: Initialize SQLite Database
import sqlite3

DB_PATH = os.path.join(PROJECT_DIR, "crypto.db")
conn = sqlite3.connect(DB_PATH)
cursor = conn.cursor()

cursor.executescript("""
PRAGMA journal_mode=WAL;

CREATE TABLE IF NOT EXISTS Cryptocurrency (
    crypto_id INTEGER PRIMARY KEY,
    ticker TEXT NOT NULL UNIQUE,
    name TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS Date (
    date_id INTEGER PRIMARY KEY,
    year INTEGER NOT NULL,
    month INTEGER NOT NULL,
    day INTEGER NOT NULL,
    UNIQUE (year, month, day)
);

CREATE TABLE IF NOT EXISTS PriceData (
    price_id INTEGER PRIMARY KEY,
    crypto_id INTEGER NOT NULL,
    date_id INTEGER NOT NULL,
    avg_price REAL,
    volume REAL,
    FOREIGN KEY (crypto_id) REFERENCES Cryptocurrency(crypto_id),
    FOREIGN KEY (date_id) REFERENCES Date(date_id)
);
""")
conn.commit()
```

2.3 Data Collection

Historical price and volume data for BTC and ETH are fetched from the CryptoCompare API. The average price is calculated as (high + low) / 2. Data is saved as crypto_raw_data.csv for further processing.

```python
# Block 3: Fetch Historical Data from CryptoCompare
import pandas as pd
import requests
from datetime import datetime
import time

def get_crypto_data(symbol: str, start_date: datetime, end_date: datetime) -> pd.DataFrame:
    base_url = "https://min-api.cryptocompare.com/data/v2/histoday"
    total_days = (end_date - start_date).days
    params = {
        "fsym": symbol,
        "tsym": "USD",
        "limit": min(total_days, 2000),
        "toTs": int(end_date.timestamp())
    }
    r = requests.get(base_url, params=params, timeout=30)
    r.raise_for_status()
    data = r.json()
    if data.get("Response") != "Success":
        raise ValueError(f"API error for {symbol}: {data.get('Message')}")
    df = pd.DataFrame(data["Data"]["Data"])
    if df.empty:
        return pd.DataFrame(columns=["date", "avg_price", "volume", "ticker"])
    df["date"] = pd.to_datetime(df["time"], unit="s")
    df["avg_price"] = (df["high"] + df["low"]) / 2.0
    df["volume"] = df["volumeto"]
    df = df[["date", "avg_price", "volume"]]
    df = df[(df["date"] >= start_date) & df["avg_price"].gt(0) & df["date"].notna()]
    return df
```

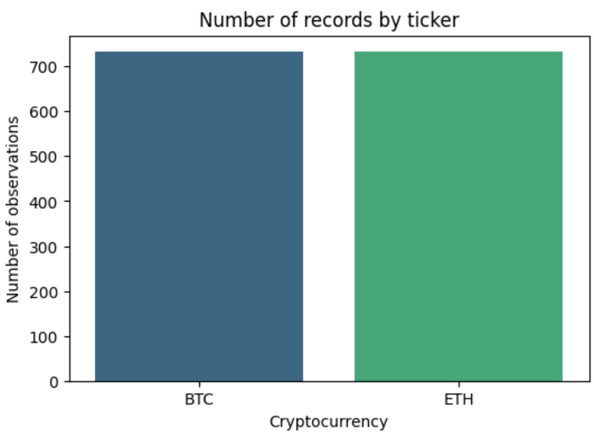
2.4 Exploratory Data Analysis (EDA)

EDA is performed to understand the dataset’s structure and characteristics. Visualizations include record counts, time coverage, price distributions, and correlations, which help validate data quality before processing.

```
# Block 4: Exploratory Data Analysis
import seaborn as sns
import matplotlib.pyplot as plt

df = raw_df.copy()

# Dataset size and sample
print("Dataset shape:", df.shape)
print("\nFirst rows:")
display(df.head())
```

```
Dataset shape: (1464, 4)

First rows:
```

	date	avg_price	volume	ticker
0	2023-01-01	16561.335	3.352209e+08	BTC
1	2023-01-02	16656.110	5.033604e+08	BTC
2	2023-01-03	16685.855	6.285822e+08	BTC
3	2023-01-04	16811.820	9.906893e+08	BTC
4	2023-01-05	16817.240	5.245115e+08	BTC



2.5 Spark Initialization and DataFrame Preparation

Apache Spark is initialized with a custom configuration to optimize performance for processing large datasets. The raw data is loaded into a Spark DataFrame with a defined schema for efficient querying.

```
# Block 5: Spark Initialization and DataFrame Prep
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, DoubleType, TimestampType
from pyspark.sql.functions import col

spark = (
    SparkSession.builder
    .appName("CryptoPipelineAdvancedSQL")
    .config("spark.sql.shuffle.partitions", "4")
    .getOrCreate()
)
spark.sparkContext.setLogLevel("WARN")

schema = StructType([
    StructField("date", TimestampType(), True),
    StructField("avg_price", DoubleType(), True),
    StructField("volume", DoubleType(), True),
    StructField("ticker", StringType(), True),
])

sdf = spark.createDataFrame(raw_df, schema=schema)
sdf.createOrReplaceTempView("crypto")
```

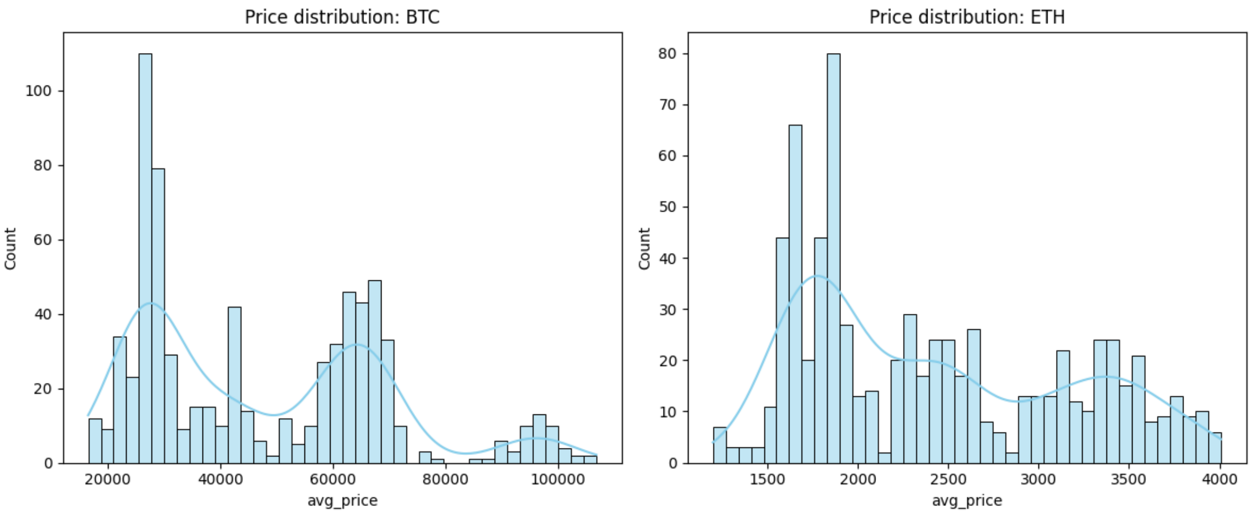
2.6 Data Aggregation with Spark SQL

Spark SQL is used to compute monthly aggregates of average price and volume, mimicking OLAP-style analytics for summarizing trends over time.

```
# Block 6: Spark SQL Monthly Aggregates
monthly_sql = spark.sql("""
SELECT
    ticker,
    YEAR(date) AS y,
    MONTH(date) AS m,
    AVG(avg_price) AS monthly_avg_price,
    AVG(volume) AS monthly_avg_volume
FROM crypto
GROUP BY ticker, YEAR(date), MONTH(date)
ORDER BY ticker, y, m
""")
```

Monthly Average Prices This chart visualizes the monthly average prices for BTC and ETH, highlighting trends over time.

```
[Spark SQL] Monthly aggregates (first rows):
+------+----+---+------------------+--------------------+
|ticker|y   |m  |monthly_avg_price |monthly_avg_volume  |
+------+----+---+------------------+--------------------+
|BTC   |2023|1  |20160.385967741935|1.2571720268654842E9|
|BTC   |2023|2  |23307.10178571429 |1.1308345866660712E9|
|BTC   |2023|3  |25050.094193548386|1.3795680694480646E9|
|BTC   |2023|4  |28851.107166666665|9.432897673156666E8 |
|BTC   |2023|5  |27512.852903225805|6.968809180993547E8 |
|BTC   |2023|6  |27723.187166666667|6.987233601986668E8 |
|BTC   |2023|7  |30099.816774193554|4.7316168831935495E8|
|BTC   |2023|8  |27855.754838709676|4.7491610116870964E8|
|BTC   |2023|9  |26308.72816666666 |4.400226051206666E8 |
|BTC   |2023|10 |29713.217580645156|8.120941657125806E8 |
|BTC   |2023|11 |36524.017166666665|9.518301370643333E8 |
|BTC   |2023|12 |42429.411129032254|1.1961175915025804E9|
+------+----+---+------------------+--------------------+
only showing top 12 rows
```



2.7 Window Functions for Moving Averages

A 7-day moving average is calculated using Spark's window functions to smooth price trends and identify patterns.

```python
# Block 7: Window Functions for 7-Day Moving Average
from pyspark.sql.window import Window
from pyspark.sql.functions import avg as favg, row_number

w = Window.partitionBy("ticker").orderBy(col("date").cast("long")).rowsBetween(-7, 0)
with_ma = sdf.withColumn("ma_7d", favg("avg_price").over(w))

w2 = Window.partitionBy("ticker").orderBy(col("date").desc())
with_rank = with_ma.withColumn("rn", row_number().over(w2))
ma_tail = with_rank.filter(col("rn") <= 30).drop("rn")
```

```
[Spark] 7-day moving average (last rows per ticker):
+-------------------+-----------------+---------------+------+-----------------+
|date               |avg_price        |volume         |ticker|ma_7d            |
+-------------------+-----------------+---------------+------+-----------------+
|2025-01-01 00:00:00|93841.155        |1.79816146781E9|BTC   |95136.035625     |
|2024-12-31 00:00:00|94017.33499999999|3.65999486824E9|BTC   |95460.73125      |
|2024-12-30 00:00:00|93110.38         |5.2874984625E9 |BTC   |95508.98062500001|
|2024-12-29 00:00:00|94013.055        |1.68413473583E9|BTC   |95843.49875      |
|2024-12-28 00:00:00|94775.39         |1.33974354648E9|BTC   |96336.17375      |
|2024-12-27 00:00:00|95310.755        |4.47965469505E9|BTC   |96381.49375000001|
|2024-12-26 00:00:00|97493.41500000001|3.70400759319E9|BTC   |96862.86125000002|
|2024-12-25 00:00:00|98526.8          |2.14704999869E9|BTC   |97579.200625     |
+-------------------+-----------------+---------------+------+-----------------+
only showing top 8 rows
```

2.8 OLAP Cube and Rollup

Spark SQL's CUBE and ROLLUP operations provide multidimensional aggregates for flexible analysis of price data.

```python
# Block 8: OLAP Cube/Rollup via SQL
cube_sql = spark.sql("""
SELECT
    ticker,
    YEAR(date) AS y,
    MONTH(date) AS m,
    AVG(avg_price) AS avg_price_cube
FROM crypto
GROUP BY CUBE(ticker, YEAR(date), MONTH(date))
ORDER BY ticker, y, m
""")

rollup_sql = spark.sql("""
SELECT
    YEAR(date) AS y,
    MONTH(date) AS m,
    AVG(avg_price) AS avg_price_rollup
FROM crypto
GROUP BY ROLLUP(YEAR(date), MONTH(date))
ORDER BY y, m
""")
```

```
[Spark SQL] CUBE by (ticker, y, m) — sample:
+------+----+----+------------------+
|ticker|y   |m   |avg_price_cube    |
+------+----+----+------------------+
|NULL  |NULL|NULL|24923.078425546435|
|NULL  |NULL|1   |17215.997182539686|
|NULL  |NULL|2   |19418.818201754395|
|NULL  |NULL|3   |24434.932459677424|
|NULL  |NULL|4   |24957.089458333332|
|NULL  |NULL|5   |24451.492459677418|
|NULL  |NULL|6   |24780.023999999998|
|NULL  |NULL|7   |24499.91983870968 |
|NULL  |NULL|8   |23063.083185483865|
|NULL  |NULL|9   |22658.999791666658|
|NULL  |NULL|10  |24844.76745967742 |
|NULL  |NULL|11  |31957.66833333334 |
+------+----+----+------------------+
only showing top 12 rows
```

```
[Spark SQL] ROLLUP by (y, m) — sample:
+----+----+------------------+
|y   |m   |avg_price_rollup  |
+----+----+------------------+
|NULL|NULL|24923.078425546435|
|2023|NULL|15310.829157534246|
|2023|1   |10810.501129032256|
|2023|2   |12465.613750000002|
|2023|3   |13362.024758064515|
|2023|4   |15383.986166666666|
|2023|5   |14678.264112903225|
|2023|6   |14770.850916666668|
|2023|7   |15997.553709677422|
|2023|8   |14805.892741935482|
|2023|9   |13965.090999999997|
|2023|10  |15686.17564516129 |
+----+----+------------------+
only showing top 12 rows
```

2.9 Cleanup

Resources are properly released to ensure efficient use of the environment.

```
# Block 10: Cleanup
spark.stop()
conn.close()
drive.flush_and_unmount()
```

3. Technologies Used

- **Apache Spark**: Distributed processing, streaming, and aggregation (RDDs, SparkSQL).
- **Python**: Scripting, API interaction, and data manipulation.
- **Pandas**: Data preprocessing and structuring.
- **SQLite**: Relational storage for aggregated data.
- **CryptoCompare API**: Source of cryptocurrency data.
- **PyCryptoDome**: AES encryption for securing outputs.
- **Google Colab & Google Drive**: Cloud-based execution and storage.

4. Results

- Built a scalable Big Data application for processing BTC and ETH price data.
- Performed 5-minute window aggregations using Spark's streaming capabilities.
- Stored aggregated data in a SQLite database with AES encryption for security.
- Demonstrated proficiency in Big Data frameworks, parallel processing, and optimization.
- Visualized key insights through charts, enhancing data interpretation.

5. Conclusion

This project successfully addresses Big Data challenges, applying distributed computing, streaming, and optimization techniques. The solution is scalable and extensible to real-time processing with tools like Kafka. Source code, output files, and the database are available at: GitHub Repository