D.A.T.A

Dynamic Analysis & Trading Alerts

Project Documentation

GitHub Repository: https://github.com/Daniel6278/marketwatch-db

December 8, 2025

Contents

1 Data Source & Collection Strategy

Our database infrastructure mirrors real-time market data from the **S&P 500 NYSE listings**, encompassing all 503 ticker symbols actively traded on the exchange. This comprehensive approach ensures broad market coverage and diverse investment opportunities for our users.

1.1 Data Acquisition Pipeline

We leverage the powerful **YahooFinance.com** platform as our primary data source, utilizing the `yfinance` Python library to programmatically extract market information. This robust solution provides reliable, up-to-date financial data with minimal latency.

Installation:

```
pip install yfinance
```

1.2 Data Collection Process

Our automated data collection script retrieves historical market data within a configurable date range (`date_start` to `date_end`), systematically processing each ticker symbol to build a comprehensive historical database.

Data Schema Structure:

Each record in our database follows this standardized format:

Ticker Symbol	Date	Open Price	High Price	Low Price	Close Price	Volume
AAPL	2024-01-15	182.50	185.20	181.80	184.90	52,340,000
MSFT	2024-01-15	375.25	378.60	374.10	377.80	28,120,000
GOOGL	2024-01-15	140.15	142.35	139.90	141.75	31,450,000

Table 1: Database Schema Example - Daily Price History Records

Field Descriptions:

- **ticker_symbol**: The unique identifier for each stock (e.g., AAPL, MSFT, GOOGL)
- **date**: Trading date in standardized format
- **open_price**: The price at which the stock opened trading for that day
- **high_price**: The highest price reached during the trading session
- **low_price**: The lowest price reached during the trading session
- **close_price**: The final price at market close
- **volume**: The total number of shares traded during the session

1.3 Database Infrastructure

Our data is hosted on **Amazon RDS (Relational Database Service)**, providing scalability, reliability, and automated backups. Each retrieved row is systematically injected into the database, maintaining data integrity and enabling efficient querying for real-time analysis.

Storage Optimization:

Due to database size constraints and performance considerations, we have strategically chosen to utilize **daily price histories** rather than intraday data. This approach balances data granularity with storage efficiency, allowing us to maintain extensive historical records while keeping query performance optimal.

2 Technical Indicators & Alert System

Our platform calculates and monitors several sophisticated technical indicators, enabling users to make informed trading decisions based on mathematical analysis of price movements and market behavior.

2.1 Supported Technical Indicators

2.1.1 1. Average True Range (ATR)

The Average True Range measures market volatility by calculating the average range between high and low prices over a specified period.

Calculation:

$$\text{True Range} = \max[(\text{High} - \text{Low}), |\text{High} - \text{Previous Close}|, |\text{Low} - \text{Previous Close}|]$$

$$\text{ATR} = \text{Moving Average of True Range over } N \text{ periods (typically 14 days)}$$

Use Case: Helps traders assess volatility and set appropriate stop-loss levels based on normal price fluctuations.

SQL Implementation:

```sql
WITH true_range AS (
    SELECT
        ticker_symbol,
        date,
        close_price,
        GREATEST(
            high_price - low_price,
            ABS(high_price - LAG(close_price) OVER (PARTITION BY ticker_symbol ORDER BY date)),
            ABS(low_price - LAG(close_price) OVER (PARTITION BY ticker_symbol ORDER BY date))
        ) AS tr
    FROM Ticker
)
SELECT
    ticker_symbol,
    date,
    AVG(tr) OVER (
        PARTITION BY ticker_symbol
        ORDER BY date
        ROWS BETWEEN 13 PRECEDING AND CURRENT ROW
    ) AS atr_14
FROM true_range
ORDER BY ticker_symbol, date;
```

Listing 1: ATR Calculation Query

2.1.2 2. Bollinger Bands

Bollinger Bands consist of three lines that create a volatility-based envelope around price movements.

Components:

- **Middle Band**: Simple Moving Average (typically 20-day SMA)
- **Upper Band**: Middle Band $+(2 \times$ Standard Deviation$)$
- **Lower Band**: Middle Band $-(2 \times$ Standard Deviation$)$

Use Case: Identifies overbought conditions (price near upper band) and oversold conditions (price near lower band), useful for mean reversion strategies.

SQL Implementation:

```sql
WITH sma_stddev AS (
    SELECT
        ticker_symbol,
        date,
        close_price,
        AVG(close_price) OVER (
            PARTITION BY ticker_symbol
            ORDER BY date
            ROWS BETWEEN 19 PRECEDING AND CURRENT ROW
        ) AS sma_20,
        STDDEV(close_price) OVER (
            PARTITION BY ticker_symbol
            ORDER BY date
            ROWS BETWEEN 19 PRECEDING AND CURRENT ROW
        ) AS stddev_20
    FROM Ticker
)
SELECT
    ticker_symbol,
    date,
    close_price,
    sma_20 AS middle_band,
    sma_20 + (2 * stddev_20) AS upper_band,
    sma_20 - (2 * stddev_20) AS lower_band
FROM sma_stddev
ORDER BY ticker_symbol, date;
```

Listing 2: Bollinger Bands Calculation Query

2.1.3 3. Moving Average Crossover

This indicator tracks the intersection points of two moving averages with different time periods to identify trend changes.

Common Configurations:

- Short-term MA: 50-day moving average
- Long-term MA: 200-day moving average

Signals:

- **Golden Cross**: Short-term MA crosses above long-term MA (bullish signal)
- **Death Cross**: Short-term MA crosses below long-term MA (bearish signal)

Use Case: Identifies potential trend reversals and entry/exit points for position trading.

SQL Implementation:

```sql
WITH moving_averages AS (
    SELECT
        ticker_symbol,
        date,
        close_price,
        AVG(close_price) OVER (
            PARTITION BY ticker_symbol
            ORDER BY date
            ROWS BETWEEN 49 PRECEDING AND CURRENT ROW
        ) AS ma_50,
        AVG(close_price) OVER (
            PARTITION BY ticker_symbol
            ORDER BY date
            ROWS BETWEEN 199 PRECEDING AND CURRENT ROW
        ) AS ma_200,
        LAG(AVG(close_price) OVER (
            PARTITION BY ticker_symbol
            ORDER BY date
            ROWS BETWEEN 49 PRECEDING AND CURRENT ROW
        )) OVER (PARTITION BY ticker_symbol ORDER BY date) AS prev_ma_50,
        LAG(AVG(close_price) OVER (
            PARTITION BY ticker_symbol
            ORDER BY date
            ROWS BETWEEN 199 PRECEDING AND CURRENT ROW
        )) OVER (PARTITION BY ticker_symbol ORDER BY date) AS prev_ma_200
    FROM Ticker
)
SELECT
    ticker_symbol,
    date,
    close_price,
    ma_50,
    ma_200,
    CASE
        WHEN ma_50 > ma_200 AND prev_ma_50 <= prev_ma_200 THEN 'Golden Cross'
        WHEN ma_50 < ma_200 AND prev_ma_50 >= prev_ma_200 THEN 'Death Cross'
        ELSE 'No Signal'
    END AS crossover_signal
FROM moving_averages
WHERE ma_50 IS NOT NULL AND ma_200 IS NOT NULL
ORDER BY ticker_symbol, date;
```

Listing 3: Moving Average Crossover Detection Query

2.1.4 4. Moving Average Convergence Divergence (MACD)

MACD is a momentum indicator that shows the relationship between two exponential moving averages of prices.

Calculation:

$$\text{MACD Line} = \text{12-period EMA} - \text{26-period EMA}$$
$$\text{Signal Line} = \text{9-period EMA of MACD Line}$$
$$\text{Histogram} = \text{MACD Line} - \text{Signal Line}$$

Signals:

- MACD crossing above Signal Line: Bullish
- MACD crossing below Signal Line: Bearish
- Divergence between MACD and price: Potential reversal

Use Case: Identifies momentum shifts, trend strength, and potential reversal points.

SQL Implementation:

```sql
WITH ema_12 AS (
    SELECT
        ticker_symbol,
        date,
        close_price,
        AVG(close_price) OVER (
            PARTITION BY ticker_symbol
            ORDER BY date
            ROWS BETWEEN 11 PRECEDING AND CURRENT ROW
        ) AS ema_12
    FROM Ticker
),
ema_26 AS (
    SELECT
        ticker_symbol,
        date,
        close_price,
        AVG(close_price) OVER (
            PARTITION BY ticker_symbol
            ORDER BY date
            ROWS BETWEEN 25 PRECEDING AND CURRENT ROW
        ) AS ema_26
    FROM Ticker
),
macd_line AS (
    SELECT
        e12.ticker_symbol,
        e12.date,
        e12.close_price,
        e12.ema_12 - e26.ema_26 AS macd
    FROM ema_12 e12
    JOIN ema_26 e26
        ON e12.ticker_symbol = e26.ticker_symbol
        AND e12.date = e26.date
)
SELECT
    ticker_symbol,
```

```sql
    date,
    close_price,
    macd AS macd_line,
    AVG(macd) OVER (
        PARTITION BY ticker_symbol
        ORDER BY date
        ROWS BETWEEN 8 PRECEDING AND CURRENT ROW
    ) AS signal_line,
    macd - AVG(macd) OVER (
        PARTITION BY ticker_symbol
        ORDER BY date
        ROWS BETWEEN 8 PRECEDING AND CURRENT ROW
    ) AS histogram
FROM macd_line
ORDER BY ticker_symbol, date;
```

Listing 4: MACD Calculation Query

2.1.5 5. Relative Strength Index (RSI)

RSI is a momentum oscillator that measures the speed and magnitude of price changes, ranging from 0 to 100.

Calculation:

$$\text{RS} = \frac{\text{Average Gain}}{\text{Average Loss}} \text{ (over 14 periods)}$$

$$\text{RSI} = 100 - \frac{100}{(1 + \text{RS})}$$

Interpretation:

- RSI > 70: Overbought condition (potential sell signal)
- RSI < 30: Oversold condition (potential buy signal)
- RSI $= 50$: Neutral momentum

Use Case: Identifies overbought/oversold conditions and potential reversal points through divergence analysis.

SQL Implementation:

```sql
WITH price_changes AS (
    SELECT
        ticker_symbol,
        date,
        close_price,
        close_price - LAG(close_price) OVER (
            PARTITION BY ticker_symbol ORDER BY date
        ) AS price_change
    FROM Ticker
),
gains_losses AS (
    SELECT
        ticker_symbol,
        date,
        close_price,
```

```sql
        CASE WHEN price_change > 0 THEN price_change ELSE 0 END AS gain,
        CASE WHEN price_change < 0 THEN ABS(price_change) ELSE 0 END AS loss
    FROM price_changes
),
avg_gain_loss AS (
    SELECT
        ticker_symbol,
        date,
        close_price,
        AVG(gain) OVER (
            PARTITION BY ticker_symbol
            ORDER BY date
            ROWS BETWEEN 13 PRECEDING AND CURRENT ROW
        ) AS avg_gain,
        AVG(loss) OVER (
            PARTITION BY ticker_symbol
            ORDER BY date
            ROWS BETWEEN 13 PRECEDING AND CURRENT ROW
        ) AS avg_loss
    FROM gains_losses
)
SELECT
    ticker_symbol,
    date,
    close_price,
    CASE
        WHEN avg_loss = 0 THEN 100
        ELSE 100 - (100 / (1 + (avg_gain / avg_loss)))
    END AS rsi_14
FROM avg_gain_loss
WHERE avg_gain IS NOT NULL AND avg_loss IS NOT NULL
ORDER BY ticker_symbol, date;
```

Listing 5: RSI Calculation Query

2.1.6 6. Stochastic Oscillator

The Stochastic Oscillator compares a stock's closing price to its price range over a specific period, generating values between 0 and 100.

Calculation:

$$\%K = \frac{(\text{Current Close} - \text{Lowest Low})}{(\text{Highest High} - \text{Lowest Low})} \times 100$$

$$\%D = \text{3-period moving average of } \%K$$

Interpretation:

- Values > 80: Overbought territory
- Values < 20: Oversold territory
- $\%K$ crossing above $\%D$: Bullish signal
- $\%K$ crossing below $\%D$: Bearish signal

Use Case: Identifies momentum changes and potential reversal points in trending or ranging markets.

SQL Implementation:

```sql
WITH price_ranges AS (
    SELECT
        ticker_symbol,
        date,
        close_price,
        high_price,
        low_price,
        MIN(low_price) OVER (
            PARTITION BY ticker_symbol
            ORDER BY date
            ROWS BETWEEN 13 PRECEDING AND CURRENT ROW
        ) AS lowest_low_14,
        MAX(high_price) OVER (
            PARTITION BY ticker_symbol
            ORDER BY date
            ROWS BETWEEN 13 PRECEDING AND CURRENT ROW
        ) AS highest_high_14
    FROM Ticker
),
percent_k AS (
    SELECT
        ticker_symbol,
        date,
        close_price,
        CASE
            WHEN (highest_high_14 - lowest_low_14) = 0 THEN 0
            ELSE ((close_price - lowest_low_14) / (highest_high_14 - lowest_low_14)) * 100
        END AS k_value
    FROM price_ranges
)
SELECT
    ticker_symbol,
    date,
    close_price,
    k_value AS percent_k,
    AVG(k_value) OVER (
        PARTITION BY ticker_symbol
        ORDER BY date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS percent_d,
    CASE
        WHEN k_value > 80 THEN 'Overbought'
        WHEN k_value < 20 THEN 'Oversold'
        ELSE 'Neutral'
    END AS signal
FROM percent_k
ORDER BY ticker_symbol, date;
```

Listing 6: Stochastic Oscillator Calculation Query

3 Business Rules & System Architecture

Our platform operates under a well-defined set of business rules that govern user interactions, data relationships, and system behavior. These rules ensure data integrity, security, and optimal user experience.

3.1 User Management Rules

3.1.1 Registration & Authentication

- Every User **must** provide a unique email address during registration
- Passwords are required and must meet security standards (hashed and salted in database)
- Email uniqueness is enforced at the database level to prevent duplicate accounts
- Users authenticate via email/password combination for secure access

3.2 Portfolio Management Rules

3.2.1 User-Portfolio Relationship

- A User **can create and own** one or many Portfolios
  - Example: A user might maintain separate portfolios for "Long-term Investments," "Day Trading," and "Cryptocurrency"
- Each Portfolio **belongs to exactly one User**
  - Ensures clear ownership and prevents unauthorized access
  - Portfolios are not shareable between users (maintains data privacy)

3.2.2 Portfolio-Ticker Relationship

- A Portfolio **can contain** zero or many Tickers
  - New portfolios start empty (zero tickers)
  - Users can add multiple stocks to track
  - No upper limit on the number of tickers per portfolio
- Users have full CRUD (Create, Read, Update, Delete) capabilities:
  - **Add**: Insert new tickers to their watch-list
  - **Remove**: Delete tickers they no longer wish to track
  - **Update**: Modify ticker-specific settings or notes

3.3 Data Integrity Rules

3.3.1 Stock-Price History Relationship

- A Stock (Ticker) **has** zero or many Price Histories
  - Newly listed stocks may have limited historical data
  - Mature stocks have extensive price history spanning years
  - Each price history record represents one trading day
- Historical data is immutable once recorded (maintains data integrity)
- Daily updates append new records without modifying existing ones

3.4 Alert System Rules

3.4.1 User-Alert Relationship

- A User **can set** zero or many Alerts
  - Users are not required to set alerts (optional feature)
  - Power users may configure multiple alerts across different indicators
  - Examples: "Alert me when AAPL RSI drops below 30" or "Notify when TSLA crosses 50-day MA"

3.4.2 Alert-User Relationship

- An Alert **belongs to** one or many Users
  - Allows for future features like shared alerts or community signals
  - Currently implemented as one-to-one (one alert per user)
  - Architecture supports future expansion to alert subscriptions

3.5 Data Access & Permissions

3.5.1 Authorization Rules

- Users can only access their own portfolios and alerts
- Administrative users have read-only access to aggregate data (no PII access)
- API rate limiting prevents abuse and ensures fair resource allocation

3.5.2 Data Retention Rules

- Price history data is retained indefinitely for analysis
- User accounts remain active until explicitly deleted by the user
- Deleted portfolios are soft-deleted (archived) for 30 days before permanent removal

4 System Architecture Summary

4.1 Source Code Repository

The complete source code for this project is available on GitHub:

https://github.com/Daniel6278/marketwatch-db

The repository contains:

- Data collection scripts using yfinance
- Database schema and migration files
- Technical indicator calculation implementations
- Alert system logic and configurations
- API endpoints and backend services
- Documentation and setup instructions

4.2 Technology Stack

- **Backend**: Python with yfinance for data acquisition
- **Database**: Amazon RDS (Relational Database)
- **Data Frequency**: Daily granularity for optimal storage/performance balance
- **Coverage**: 503 tickers from S&P 500 NYSE listings

4.3 Key Features

- Automated daily data collection from Yahoo Finance
- Real-time calculation of six major technical indicators
- Customizable alert system based on indicator thresholds
- Multi-portfolio support for diverse investment strategies
- Secure user authentication and data isolation

4.4 Performance Considerations

- Daily data updates reduce API load and storage requirements
- Indexed database queries for fast portfolio and alert lookups
- Efficient calculation algorithms for technical indicators
- Scalable architecture supporting additional tickers and users