**📊 KURZORA TRADING PLATFORM - FINANCIAL DATA PROCESSING ANALYSIS**

**Executive Summary**

Complete financial algorithms, data structures, and implementation code for Kurzora's signal generation and trading systems. This document provides production-ready code for immediate Cursor development, covering market data integration, technical analysis, signal processing, and risk management.

**1. 📈 MARKET DATA INTEGRATION**

**Polygon.io API Integration Patterns**

// ===================================================================

// POLYGON.IO API INTEGRATION SYSTEM

// ===================================================================

// lib/market-data/polygonClient.ts

interface PolygonConfig {

apiKey: string;

baseUrl: string;

maxRequestsPerMinute: number;

enableRealTime: boolean;

}

interface MarketData {

ticker: string;

timestamp: number;

open: number;

high: number;

low: number;

close: number;

volume: number;

vwap?: number;

transactions?: number;

}

interface AggregateBar {

ticker: string;

timespan: 'minute' | 'hour' | 'day' | 'week';

multiplier: number;

from: string;

to: string;

results: MarketData[];

}

interface TickerSnapshot {

ticker: string;

value: number;

todaysChange: number;

todaysChangePerc: number;

min: {

av: number;

c: number;

h: number;

l: number;

o: number;

t: number;

v: number;

vw: number;

};

prevDay: {

c: number;

h: number;

l: number;

o: number;

v: number;

vw: number;

};

}

class PolygonApiClient {

private config: PolygonConfig;

private rateLimiter: RateLimiter;

private cache: Map<string, { data: any; timestamp: number }>;

private retryQueue: Set<string>;

constructor(config: PolygonConfig) {

this.config = config;

this.rateLimiter = new RateLimiter(config.maxRequestsPerMinute, 60000);

this.cache = new Map();

this.retryQueue = new Set();

}

// Real-time stock quotes

async getSnapshot(ticker: string): Promise<TickerSnapshot | null> {

const cacheKey = `snapshot\_${ticker}`;

const cached = this.getCachedData(cacheKey, 30000); // 30 seconds cache

if (cached) return cached;

try {

await this.rateLimiter.waitForToken();

const response = await fetch(

`${this.config.baseUrl}/v2/snapshot/locale/us/markets/stocks/tickers/${ticker}?apikey=${this.config.apiKey}`,

{

method: 'GET',

headers: {

'Accept': 'application/json',

'User-Agent': 'Kurzora/1.0'

}

}

);

if (!response.ok) {

if (response.status === 429) {

// Rate limited, add to retry queue

this.retryQueue.add(ticker);

await this.delay(1000);

return this.getSnapshot(ticker);

}

throw new Error(`Polygon API error: ${response.status}`);

}

const data = await response.json();

const snapshot = data.results?.[0];

if (snapshot) {

this.setCachedData(cacheKey, snapshot);

return snapshot;

}

return null;

} catch (error) {

console.error(`Failed to get snapshot for ${ticker}:`, error);

return null;

}

}

// Historical aggregates (OHLCV data)

async getAggregates(

ticker: string,

multiplier: number,

timespan: 'minute' | 'hour' | 'day' | 'week',

from: string,

to: string

): Promise<MarketData[]> {

const cacheKey = `agg\_${ticker}\_${timespan}\_${multiplier}\_${from}\_${to}`;

const cached = this.getCachedData(cacheKey, timespan === 'minute' ? 60000 : 300000);

if (cached) return cached;

try {

await this.rateLimiter.waitForToken();

const response = await fetch(

`${this.config.baseUrl}/v2/aggs/ticker/${ticker}/range/${multiplier}/${timespan}/${from}/${to}?adjusted=true&sort=asc&apikey=${this.config.apiKey}`,

{

method: 'GET',

headers: {

'Accept': 'application/json',

'User-Agent': 'Kurzora/1.0'

}

}

);

if (!response.ok) {

if (response.status === 429) {

await this.delay(2000);

return this.getAggregates(ticker, multiplier, timespan, from, to);

}

throw new Error(`Polygon API error: ${response.status}`);

}

const data: AggregateBar = await response.json();

const results = data.results || [];

// Normalize the data

const normalizedData = results.map(bar => ({

ticker,

timestamp: bar.t || 0,

open: bar.o || 0,

high: bar.h || 0,

low: bar.l || 0,

close: bar.c || 0,

volume: bar.v || 0,

vwap: bar.vw,

transactions: bar.n

}));

this.setCachedData(cacheKey, normalizedData);

return normalizedData;

} catch (error) {

console.error(`Failed to get aggregates for ${ticker}:`, error);

return [];

}

}

// Batch request for multiple tickers

async getMultipleSnapshots(tickers: string[]): Promise<Map<string, TickerSnapshot>> {

const results = new Map<string, TickerSnapshot>();

const batchSize = 20; // Polygon limits

for (let i = 0; i < tickers.length; i += batchSize) {

const batch = tickers.slice(i, i + batchSize);

const promises = batch.map(async (ticker) => {

const snapshot = await this.getSnapshot(ticker);

if (snapshot) {

results.set(ticker, snapshot);

}

});

await Promise.allSettled(promises);

// Rate limiting between batches

if (i + batchSize < tickers.length) {

await this.delay(1000);

}

}

return results;

}

// WebSocket for real-time data

initializeRealTimeStream(tickers: string[], onData: (data: any) => void): WebSocket | null {

if (!this.config.enableRealTime) return null;

try {

const ws = new WebSocket('wss://socket.polygon.io/stocks');

ws.onopen = () => {

// Authenticate

ws.send(JSON.stringify({

action: 'auth',

params: this.config.apiKey

}));

// Subscribe to tickers

ws.send(JSON.stringify({

action: 'subscribe',

params: tickers.map(ticker => `T.${ticker}`)

}));

};

ws.onmessage = (event) => {

try {

const data = JSON.parse(event.data);

onData(data);

} catch (error) {

console.error('Failed to parse WebSocket message:', error);

}

};

ws.onerror = (error) => {

console.error('WebSocket error:', error);

};

ws.onclose = () => {

console.log('WebSocket connection closed');

// Implement reconnection logic

setTimeout(() => {

this.initializeRealTimeStream(tickers, onData);

}, 5000);

};

return ws;

} catch (error) {

console.error('Failed to initialize WebSocket:', error);

return null;

}

}

// Helper methods

private getCachedData(key: string, maxAge: number): any | null {

const cached = this.cache.get(key);

if (cached && Date.now() - cached.timestamp < maxAge) {

return cached.data;

}

return null;

}

private setCachedData(key: string, data: any): void {

this.cache.set(key, {

data,

timestamp: Date.now()

});

// Cleanup old cache entries

if (this.cache.size > 10000) {

const entries = Array.from(this.cache.entries());

entries.sort((a, b) => a[1].timestamp - b[1].timestamp);

entries.slice(0, 1000).forEach(([key]) => this.cache.delete(key));

}

}

private delay(ms: number): Promise<void> {

return new Promise(resolve => setTimeout(resolve, ms));

}

}

// Rate limiter implementation

class RateLimiter {

private tokens: number;

private maxTokens: number;

private refillRate: number;

private lastRefill: number;

constructor(maxRequestsPerInterval: number, intervalMs: number) {

this.maxTokens = maxRequestsPerInterval;

this.tokens = maxRequestsPerInterval;

this.refillRate = maxRequestsPerInterval / intervalMs;

this.lastRefill = Date.now();

}

async waitForToken(): Promise<void> {

this.refillTokens();

if (this.tokens < 1) {

const waitTime = (1 - this.tokens) / this.refillRate;

await new Promise(resolve => setTimeout(resolve, waitTime));

this.refillTokens();

}

this.tokens--;

}

private refillTokens(): void {

const now = Date.now();

const elapsed = now - this.lastRefill;

const tokensToAdd = elapsed \* this.refillRate;

this.tokens = Math.min(this.maxTokens, this.tokens + tokensToAdd);

this.lastRefill = now;

}

}

**Data Normalization and Storage**

// ===================================================================

// DATA NORMALIZATION AND STORAGE SYSTEM

// ===================================================================

// lib/market-data/dataProcessor.ts

interface NormalizedMarketData {

ticker: string;

timestamp: Date;

timeframe: '1m' | '5m' | '15m' | '1h' | '4h' | '1d' | '1w';

ohlcv: {

open: number;

high: number;

low: number;

close: number;

volume: number;

};

indicators?: {

vwap?: number;

sma?: Record<number, number>; // period -> value

ema?: Record<number, number>;

rsi?: Record<number, number>;

macd?: {

macd: number;

signal: number;

histogram: number;

};

};

metadata: {

source: string;

quality: 'high' | 'medium' | 'low';

adjustedClose?: number;

splitRatio?: number;

dividendAmount?: number;

};

}

class MarketDataProcessor {

private polygonClient: PolygonApiClient;

private cache: Map<string, NormalizedMarketData[]>;

constructor(polygonClient: PolygonApiClient) {

this.polygonClient = polygonClient;

this.cache = new Map();

}

// Normalize raw market data from various sources

async normalizeAndStore(

ticker: string,

rawData: MarketData[],

timeframe: string,

source: string = 'polygon'

): Promise<NormalizedMarketData[]> {

const normalized = rawData.map(data => this.normalizeDataPoint(data, timeframe, source));

// Validate data quality

const validatedData = normalized.filter(data => this.validateDataPoint(data));

// Store in database

await this.storeNormalizedData(validatedData);

// Cache for immediate use

const cacheKey = `${ticker}\_${timeframe}`;

this.cache.set(cacheKey, validatedData);

return validatedData;

}

private normalizeDataPoint(

data: MarketData,

timeframe: string,

source: string

): NormalizedMarketData {

return {

ticker: data.ticker.toUpperCase(),

timestamp: new Date(data.timestamp),

timeframe: this.normalizeTimeframe(timeframe),

ohlcv: {

open: this.roundPrice(data.open),

high: this.roundPrice(data.high),

low: this.roundPrice(data.low),

close: this.roundPrice(data.close),

volume: Math.round(data.volume)

},

metadata: {

source,

quality: this.assessDataQuality(data),

...(data.vwap && { vwap: this.roundPrice(data.vwap) })

}

};

}

private normalizeTimeframe(timeframe: string): '1m' | '5m' | '15m' | '1h' | '4h' | '1d' | '1w' {

const mapping: Record<string, any> = {

'minute': '1m',

'1': '1m',

'5': '5m',

'15': '15m',

'hour': '1h',

'60': '1h',

'240': '4h',

'day': '1d',

'week': '1w'

};

return mapping[timeframe] || '1d';

}

private validateDataPoint(data: NormalizedMarketData): boolean {

const { ohlcv } = data;

// Basic OHLCV validation

if (ohlcv.open <= 0 || ohlcv.high <= 0 || ohlcv.low <= 0 || ohlcv.close <= 0) {

return false;

}

// High should be >= low

if (ohlcv.high < ohlcv.low) {

return false;

}

// OHLC relationships

if (ohlcv.open > ohlcv.high || ohlcv.open < ohlcv.low ||

ohlcv.close > ohlcv.high || ohlcv.close < ohlcv.low) {

return false;

}

// Volume should be non-negative

if (ohlcv.volume < 0) {

return false;

}

// Check for extreme price movements (>50% in a single period)

const maxChange = Math.max(

Math.abs(ohlcv.high - ohlcv.low) / ohlcv.low,

Math.abs(ohlcv.open - ohlcv.close) / ohlcv.open

);

if (maxChange > 0.5) {

console.warn(`Extreme price movement detected for ${data.ticker}: ${maxChange \* 100}%`);

// Don't reject, but mark as low quality

data.metadata.quality = 'low';

}

return true;

}

private assessDataQuality(data: MarketData): 'high' | 'medium' | 'low' {

let score = 100;

// Check for zero volume

if (data.volume === 0) score -= 30;

// Check for missing VWAP

if (!data.vwap) score -= 10;

// Check for round numbers (potential placeholder data)

if (data.close % 1 === 0) score -= 5;

// Check timestamp freshness

const age = Date.now() - data.timestamp;

if (age > 24 \* 60 \* 60 \* 1000) score -= 20; // Older than 1 day

if (score >= 80) return 'high';

if (score >= 60) return 'medium';

return 'low';

}

private roundPrice(price: number): number {

// Round to 4 decimal places for precision

return Math.round(price \* 10000) / 10000;

}

private async storeNormalizedData(data: NormalizedMarketData[]): Promise<void> {

if (data.length === 0) return;

try {

// Batch insert for performance

const batchSize = 1000;

for (let i = 0; i < data.length; i += batchSize) {

const batch = data.slice(i, i + batchSize);

const insertData = batch.map(point => ({

ticker: point.ticker,

timestamp: point.timestamp.toISOString(),

timeframe: point.timeframe,

open: point.ohlcv.open,

high: point.ohlcv.high,

low: point.ohlcv.low,

close: point.ohlcv.close,

volume: point.ohlcv.volume,

vwap: point.metadata.vwap,

source: point.metadata.source,

quality: point.metadata.quality,

created\_at: new Date().toISOString()

}));

const { error } = await supabaseClient

.from('market\_data')

.upsert(insertData, {

onConflict: 'ticker,timestamp,timeframe',

ignoreDuplicates: false

});

if (error) {

console.error('Failed to store market data batch:', error);

throw error;

}

}

} catch (error) {

console.error('Failed to store normalized data:', error);

throw error;

}

}

// Retrieve normalized data with caching

async getNormalizedData(

ticker: string,

timeframe: string,

from: Date,

to: Date

): Promise<NormalizedMarketData[]> {

const cacheKey = `${ticker}\_${timeframe}\_${from.getTime()}\_${to.getTime()}`;

const cached = this.cache.get(cacheKey);

if (cached) return cached;

try {

const { data, error } = await supabaseClient

.from('market\_data')

.select('\*')

.eq('ticker', ticker)

.eq('timeframe', timeframe)

.gte('timestamp', from.toISOString())

.lte('timestamp', to.toISOString())

.order('timestamp', { ascending: true });

if (error) throw error;

const normalized = data.map(row => ({

ticker: row.ticker,

timestamp: new Date(row.timestamp),

timeframe: row.timeframe,

ohlcv: {

open: row.open,

high: row.high,

low: row.low,

close: row.close,

volume: row.volume

},

metadata: {

source: row.source,

quality: row.quality,

vwap: row.vwap

}

}));

this.cache.set(cacheKey, normalized);

return normalized;

} catch (error) {

console.error('Failed to retrieve normalized data:', error);

return [];

}

}

}

**2. 🔧 TECHNICAL INDICATOR CALCULATIONS**

**RSI Implementation**

// ===================================================================

// TECHNICAL INDICATOR CALCULATIONS

// ===================================================================

// lib/indicators/rsi.ts

interface RSIResult {

value: number;

timestamp: Date;

period: number;

signal: 'oversold' | 'overbought' | 'neutral';

}

class RSICalculator {

private period: number;

private prices: number[];

private gains: number[];

private losses: number[];

private avgGain: number;

private avgLoss: number;

constructor(period: number = 14) {

this.period = period;

this.prices = [];

this.gains = [];

this.losses = [];

this.avgGain = 0;

this.avgLoss = 0;

}

// Calculate RSI for a single data point

calculate(price: number): number | null {

this.prices.push(price);

if (this.prices.length < 2) return null;

// Calculate price change

const change = price - this.prices[this.prices.length - 2];

const gain = change > 0 ? change : 0;

const loss = change < 0 ? Math.abs(change) : 0;

this.gains.push(gain);

this.losses.push(loss);

// Need at least 'period' values to calculate RSI

if (this.gains.length < this.period) return null;

// Calculate average gain and loss

if (this.gains.length === this.period) {

// Initial calculation - simple average

this.avgGain = this.gains.reduce((a, b) => a + b, 0) / this.period;

this.avgLoss = this.losses.reduce((a, b) => a + b, 0) / this.period;

} else {

// Subsequent calculations - smoothed average (Wilder's smoothing)

this.avgGain = ((this.avgGain \* (this.period - 1)) + gain) / this.period;

this.avgLoss = ((this.avgLoss \* (this.period - 1)) + loss) / this.period;

}

// Prevent division by zero

if (this.avgLoss === 0) return 100;

// Calculate RSI

const rs = this.avgGain / this.avgLoss;

const rsi = 100 - (100 / (1 + rs));

return Math.round(rsi \* 100) / 100; // Round to 2 decimal places

}

// Calculate RSI for array of prices

static calculateSeries(prices: number[], period: number = 14): number[] {

const calculator = new RSICalculator(period);

return prices.map(price => calculator.calculate(price)).filter(rsi => rsi !== null) as number[];

}

// Calculate RSI with signal interpretation

static calculateWithSignal(prices: number[], period: number = 14): RSIResult[] {

const rsiValues = this.calculateSeries(prices, period);

return rsiValues.map((value, index) => ({

value,

timestamp: new Date(), // This would be the actual timestamp in real implementation

period,

signal: this.interpretRSI(value)

}));

}

private static interpretRSI(rsi: number): 'oversold' | 'overbought' | 'neutral' {

if (rsi <= 30) return 'oversold';

if (rsi >= 70) return 'overbought';

return 'neutral';

}

// Advanced RSI with divergence detection

static detectDivergence(

prices: number[],

rsiValues: number[],

lookback: number = 10

): { bullish: boolean; bearish: boolean } {

if (prices.length < lookback || rsiValues.length < lookback) {

return { bullish: false, bearish: false };

}

const recentPrices = prices.slice(-lookback);

const recentRSI = rsiValues.slice(-lookback);

// Find local highs and lows

const priceHighs = this.findLocalExtremes(recentPrices, 'high');

const priceLows = this.findLocalExtremes(recentPrices, 'low');

const rsiHighs = this.findLocalExtremes(recentRSI, 'high');

const rsiLows = this.findLocalExtremes(recentRSI, 'low');

// Bullish divergence: price makes lower low, RSI makes higher low

const bullishDivergence = priceLows.length >= 2 && rsiLows.length >= 2 &&

priceLows[priceLows.length - 1] < priceLows[priceLows.length - 2] &&

rsiLows[rsiLows.length - 1] > rsiLows[rsiLows.length - 2];

// Bearish divergence: price makes higher high, RSI makes lower high

const bearishDivergence = priceHighs.length >= 2 && rsiHighs.length >= 2 &&

priceHighs[priceHighs.length - 1] > priceHighs[priceHighs.length - 2] &&

rsiHighs[rsiHighs.length - 1] < rsiHighs[rsiHighs.length - 2];

return {

bullish: bullishDivergence,

bearish: bearishDivergence

};

}

private static findLocalExtremes(data: number[], type: 'high' | 'low'): number[] {

const extremes: number[] = [];

for (let i = 1; i < data.length - 1; i++) {

if (type === 'high') {

if (data[i] > data[i - 1] && data[i] > data[i + 1]) {

extremes.push(data[i]);

}

} else {

if (data[i] < data[i - 1] && data[i] < data[i + 1]) {

extremes.push(data[i]);

}

}

}

return extremes;

}

// Reset calculator state

reset(): void {

this.prices = [];

this.gains = [];

this.losses = [];

this.avgGain = 0;

this.avgLoss = 0;

}

}

**MACD Implementation**

// ===================================================================

// MACD (Moving Average Convergence Divergence) IMPLEMENTATION

// ===================================================================

// lib/indicators/macd.ts

interface MACDResult {

macd: number;

signal: number;

histogram: number;

timestamp: Date;

trend: 'bullish' | 'bearish' | 'neutral';

}

class MACDCalculator {

private fastPeriod: number;

private slowPeriod: number;

private signalPeriod: number;

private fastEMA: EMACalculator;

private slowEMA: EMACalculator;

private signalEMA: EMACalculator;

private macdHistory: number[];

constructor(fastPeriod: number = 12, slowPeriod: number = 26, signalPeriod: number = 9) {

this.fastPeriod = fastPeriod;

this.slowPeriod = slowPeriod;

this.signalPeriod = signalPeriod;

this.fastEMA = new EMACalculator(fastPeriod);

this.slowEMA = new EMACalculator(slowPeriod);

this.signalEMA = new EMACalculator(signalPeriod);

this.macdHistory = [];

}

calculate(price: number): MACDResult | null {

const fastEMA = this.fastEMA.calculate(price);

const slowEMA = this.slowEMA.calculate(price);

if (fastEMA === null || slowEMA === null) return null;

// MACD Line = Fast EMA - Slow EMA

const macd = fastEMA - slowEMA;

this.macdHistory.push(macd);

// Signal Line = EMA of MACD Line

const signal = this.signalEMA.calculate(macd);

if (signal === null) return null;

// Histogram = MACD - Signal

const histogram = macd - signal;

return {

macd: Math.round(macd \* 10000) / 10000,

signal: Math.round(signal \* 10000) / 10000,

histogram: Math.round(histogram \* 10000) / 10000,

timestamp: new Date(),

trend: this.determineTrend(macd, signal, histogram)

};

}

static calculateSeries(

prices: number[],

fastPeriod: number = 12,

slowPeriod: number = 26,

signalPeriod: number = 9

): MACDResult[] {

const calculator = new MACDCalculator(fastPeriod, slowPeriod, signalPeriod);

return prices.map(price => calculator.calculate(price))

.filter(result => result !== null) as MACDResult[];

}

private determineTrend(macd: number, signal: number, histogram: number): 'bullish' | 'bearish' | 'neutral' {

// Bullish when MACD > Signal and histogram is positive and increasing

if (macd > signal && histogram > 0) {

if (this.macdHistory.length >= 2) {

const prevMACD = this.macdHistory[this.macdHistory.length - 2];

const currentMACD = this.macdHistory[this.macdHistory.length - 1];

if (currentMACD > prevMACD) return 'bullish';

}

return 'bullish';

}

// Bearish when MACD < Signal and histogram is negative and decreasing

if (macd < signal && histogram < 0) {

if (this.macdHistory.length >= 2) {

const prevMACD = this.macdHistory[this.macdHistory.length - 2];

const currentMACD = this.macdHistory[this.macdHistory.length - 1];

if (currentMACD < prevMACD) return 'bearish';

}

return 'bearish';

}

return 'neutral';

}

// Detect MACD signal crossovers

static detectCrossovers(macdResults: MACDResult[]): {

bullishCrossovers: number[];

bearishCrossovers: number[];

} {

const bullishCrossovers: number[] = [];

const bearishCrossovers: number[] = [];

for (let i = 1; i < macdResults.length; i++) {

const prev = macdResults[i - 1];

const current = macdResults[i];

// Bullish crossover: MACD crosses above Signal

if (prev.macd <= prev.signal && current.macd > current.signal) {

bullishCrossovers.push(i);

}

// Bearish crossover: MACD crosses below Signal

if (prev.macd >= prev.signal && current.macd < current.signal) {

bearishCrossovers.push(i);

}

}

return { bullishCrossovers, bearishCrossovers };

}

reset(): void {

this.fastEMA.reset();

this.slowEMA.reset();

this.signalEMA.reset();

this.macdHistory = [];

}

}

// EMA Calculator helper class

class EMACalculator {

private period: number;

private multiplier: number;

private ema: number | null;

constructor(period: number) {

this.period = period;

this.multiplier = 2 / (period + 1);

this.ema = null;

}

calculate(price: number): number | null {

if (this.ema === null) {

this.ema = price; // First value is the price itself

} else {

this.ema = (price \* this.multiplier) + (this.ema \* (1 - this.multiplier));

}

return this.ema;

}

reset(): void {

this.ema = null;

}

}

**Bollinger Bands Implementation**

// ===================================================================

// BOLLINGER BANDS IMPLEMENTATION

// ===================================================================

// lib/indicators/bollingerBands.ts

interface BollingerBandsResult {

upperBand: number;

middleBand: number; // SMA

lowerBand: number;

bandwidth: number;

percentB: number;

squeeze: boolean;

timestamp: Date;

signal: 'buy' | 'sell' | 'neutral';

}

class BollingerBandsCalculator {

private period: number;

private standardDeviations: number;

private prices: number[];

private smaCalculator: SMACalculator;

constructor(period: number = 20, standardDeviations: number = 2) {

this.period = period;

this.standardDeviations = standardDeviations;

this.prices = [];

this.smaCalculator = new SMACalculator(period);

}

calculate(price: number): BollingerBandsResult | null {

this.prices.push(price);

// Keep only the required number of prices

if (this.prices.length > this.period) {

this.prices.shift();

}

if (this.prices.length < this.period) return null;

// Calculate middle band (SMA)

const middleBand = this.smaCalculator.calculate(price);

if (middleBand === null) return null;

// Calculate standard deviation

const variance = this.prices.reduce((sum, p) => {

return sum + Math.pow(p - middleBand, 2);

}, 0) / this.period;

const standardDeviation = Math.sqrt(variance);

// Calculate upper and lower bands

const upperBand = middleBand + (this.standardDeviations \* standardDeviation);

const lowerBand = middleBand - (this.standardDeviations \* standardDeviation);

// Calculate bandwidth (volatility measure)

const bandwidth = (upperBand - lowerBand) / middleBand;

// Calculate %B (position within bands)

const percentB = (price - lowerBand) / (upperBand - lowerBand);

// Detect squeeze (low volatility)

const squeeze = bandwidth < 0.1; // Configurable threshold

return {

upperBand: Math.round(upperBand \* 100) / 100,

middleBand: Math.round(middleBand \* 100) / 100,

lowerBand: Math.round(lowerBand \* 100) / 100,

bandwidth: Math.round(bandwidth \* 10000) / 10000,

percentB: Math.round(percentB \* 10000) / 10000,

squeeze,

timestamp: new Date(),

signal: this.generateSignal(price, upperBand, lowerBand, percentB)

};

}

static calculateSeries(

prices: number[],

period: number = 20,

standardDeviations: number = 2

): BollingerBandsResult[] {

const calculator = new BollingerBandsCalculator(period, standardDeviations);

return prices.map(price => calculator.calculate(price))

.filter(result => result !== null) as BollingerBandsResult[];

}

private generateSignal(

price: number,

upperBand: number,

lowerBand: number,

percentB: number

): 'buy' | 'sell' | 'neutral' {

// Buy signal when price touches lower band (oversold)

if (percentB <= 0.05) return 'buy';

// Sell signal when price touches upper band (overbought)

if (percentB >= 0.95) return 'sell';

return 'neutral';

}

// Detect Bollinger Band squeeze

static detectSqueeze(results: BollingerBandsResult[], lookback: number = 20): boolean {

if (results.length < lookback) return false;

const recentResults = results.slice(-lookback);

const avgBandwidth = recentResults.reduce((sum, r) => sum + r.bandwidth, 0) / lookback;

// Squeeze when bandwidth is below 10th percentile of recent values

const sortedBandwidths = recentResults.map(r => r.bandwidth).sort((a, b) => a - b);

const tenthPercentile = sortedBandwidths[Math.floor(sortedBandwidths.length \* 0.1)];

return avgBandwidth <= tenthPercentile;

}

// Detect band walk (trending condition)

static detectBandWalk(

results: BollingerBandsResult[],

type: 'upper' | 'lower',

minConsecutive: number = 3

): boolean {

if (results.length < minConsecutive) return false;

const recent = results.slice(-minConsecutive);

if (type === 'upper') {

return recent.every(r => r.percentB >= 0.8);

} else {

return recent.every(r => r.percentB <= 0.2);

}

}

reset(): void {

this.prices = [];

this.smaCalculator.reset();

}

}

// SMA Calculator helper class

class SMACalculator {

private period: number;

private prices: number[];

constructor(period: number) {

this.period = period;

this.prices = [];

}

calculate(price: number): number | null {

this.prices.push(price);

if (this.prices.length > this.period) {

this.prices.shift();

}

if (this.prices.length < this.period) return null;

const sum = this.prices.reduce((a, b) => a + b, 0);

return sum / this.period;

}

reset(): void {

this.prices = [];

}

}

**Volume Analysis and Multi-Timeframe Integration**

// ===================================================================

// VOLUME ANALYSIS AND MULTI-TIMEFRAME SYSTEM

// ===================================================================

// lib/indicators/volumeAnalysis.ts

interface VolumeIndicators {

volume: number;

volumeMA: number;

volumeRatio: number;

obv: number; // On-Balance Volume

cmf: number; // Chaikin Money Flow

vwap: number; // Volume Weighted Average Price

volumeProfile: VolumeProfileLevel[];

signal: 'accumulation' | 'distribution' | 'neutral';

}

interface VolumeProfileLevel {

price: number;

volume: number;

percentage: number;

}

interface MultiTimeframeIndicators {

timeframes: {

'1H': TimeframeIndicators;

'4H': TimeframeIndicators;

'1D': TimeframeIndicators;

'1W': TimeframeIndicators;

};

confluence: {

bullishCount: number;

bearishCount: number;

overallSignal: 'bullish' | 'bearish' | 'neutral';

confidence: number;

};

}

interface TimeframeIndicators {

rsi: number;

macd: MACDResult;

bollingerBands: BollingerBandsResult;

volume: VolumeIndicators;

score: number;

weight: number;

}

class VolumeAnalysisCalculator {

private period: number;

private volumes: number[];

private prices: number[];

private highs: number[];

private lows: number[];

private closes: number[];

private obvValue: number;

private volumeMACalculator: SMACalculator;

constructor(period: number = 20) {

this.period = period;

this.volumes = [];

this.prices = [];

this.highs = [];

this.lows = [];

this.closes = [];

this.obvValue = 0;

this.volumeMACalculator = new SMACalculator(period);

}

calculate(

high: number,

low: number,

close: number,

volume: number,

prevClose?: number

): VolumeIndicators | null {

this.highs.push(high);

this.lows.push(low);

this.closes.push(close);

this.volumes.push(volume);

// Maintain sliding window

if (this.volumes.length > this.period) {

this.volumes.shift();

this.highs.shift();

this.lows.shift();

this.closes.shift();

}

if (this.volumes.length < this.period) return null;

// Calculate Volume Moving Average

const volumeMA = this.volumeMACalculator.calculate(volume);

if (volumeMA === null) return null;

// Calculate Volume Ratio

const volumeRatio = volume / volumeMA;

// Calculate On-Balance Volume

if (prevClose !== undefined) {

if (close > prevClose) {

this.obvValue += volume;

} else if (close < prevClose) {

this.obvValue -= volume;

}

// If close === prevClose, OBV remains unchanged

}

// Calculate Chaikin Money Flow

const cmf = this.calculateCMF();

// Calculate VWAP

const vwap = this.calculateVWAP();

// Calculate Volume Profile

const volumeProfile = this.calculateVolumeProfile();

return {

volume,

volumeMA: Math.round(volumeMA),

volumeRatio: Math.round(volumeRatio \* 100) / 100,

obv: Math.round(this.obvValue),

cmf: Math.round(cmf \* 10000) / 10000,

vwap: Math.round(vwap \* 100) / 100,

volumeProfile,

signal: this.determineVolumeSignal(volumeRatio, cmf, this.obvValue)

};

}

private calculateCMF(): number {

if (this.highs.length === 0) return 0;

let moneyFlowVolume = 0;

let totalVolume = 0;

for (let i = 0; i < this.highs.length; i++) {

const high = this.highs[i];

const low = this.lows[i];

const close = this.closes[i];

const volume = this.volumes[i];

// Money Flow Multiplier

const mfMultiplier = ((close - low) - (high - close)) / (high - low);

// Money Flow Volume

const mfVolume = mfMultiplier \* volume;

moneyFlowVolume += mfVolume;

totalVolume += volume;

}

return totalVolume > 0 ? moneyFlowVolume / totalVolume : 0;

}

private calculateVWAP(): number {

let cumulativeTPV = 0; // Typical Price \* Volume

let cumulativeVolume = 0;

for (let i = 0; i < this.highs.length; i++) {

const typicalPrice = (this.highs[i] + this.lows[i] + this.closes[i]) / 3;

cumulativeTPV += typicalPrice \* this.volumes[i];

cumulativeVolume += this.volumes[i];

}

return cumulativeVolume > 0 ? cumulativeTPV / cumulativeVolume : 0;

}

private calculateVolumeProfile(): VolumeProfileLevel[] {

if (this.highs.length === 0) return [];

// Create price levels based on the price range

const minPrice = Math.min(...this.lows);

const maxPrice = Math.max(...this.highs);

const priceRange = maxPrice - minPrice;

const levels = 20; // Number of price levels

const levelSize = priceRange / levels;

const volumeAtLevel = new Array(levels).fill(0);

const priceLevels = Array.from({ length: levels }, (\_, i) =>

minPrice + (i + 0.5) \* levelSize

);

// Distribute volume across price levels

for (let i = 0; i < this.highs.length; i++) {

const high = this.highs[i];

const low = this.lows[i];

const volume = this.volumes[i];

// Distribute volume evenly across the bar's price range

const startLevel = Math.floor((low - minPrice) / levelSize);

const endLevel = Math.floor((high - minPrice) / levelSize);

for (let level = Math.max(0, startLevel); level <= Math.min(levels - 1, endLevel); level++) {

volumeAtLevel[level] += volume / (endLevel - startLevel + 1);

}

}

const totalVolume = volumeAtLevel.reduce((a, b) => a + b, 0);

return priceLevels.map((price, index) => ({

price: Math.round(price \* 100) / 100,

volume: Math.round(volumeAtLevel[index]),

percentage: totalVolume > 0 ? Math.round((volumeAtLevel[index] / totalVolume) \* 10000) / 100 : 0

}));

}

private determineVolumeSignal(

volumeRatio: number,

cmf: number,

obv: number

): 'accumulation' | 'distribution' | 'neutral' {

let bullishSignals = 0;

let bearishSignals = 0;

// High volume ratio suggests interest

if (volumeRatio > 1.5) bullishSignals++;

if (volumeRatio < 0.5) bearishSignals++;

// Positive CMF suggests accumulation

if (cmf > 0.1) bullishSignals++;

if (cmf < -0.1) bearishSignals++;

// Rising OBV suggests accumulation

// This would require historical OBV values to determine trend

// For now, we'll use the absolute value as a proxy

if (obv > 0) bullishSignals++;

if (obv < 0) bearishSignals++;

if (bullishSignals > bearishSignals) return 'accumulation';

if (bearishSignals > bullishSignals) return 'distribution';

return 'neutral';

}

reset(): void {

this.volumes = [];

this.prices = [];

this.highs = [];

this.lows = [];

this.closes = [];

this.obvValue = 0;

this.volumeMACalculator.reset();

}

}

// Multi-Timeframe Analysis System

class MultiTimeframeAnalyzer {

private timeframeWeights: Record<string, number> = {

'1H': 0.4, // 40%

'4H': 0.3, // 30%

'1D': 0.2, // 20%

'1W': 0.1 // 10%

};

async analyzeAllTimeframes(

ticker: string,

marketDataProcessor: MarketDataProcessor

): Promise<MultiTimeframeIndicators> {

const timeframes = ['1H', '4H', '1D', '1W'];

const results: Partial<MultiTimeframeIndicators['timeframes']> = {};

// Analyze each timeframe

for (const timeframe of timeframes) {

const indicators = await this.analyzeTimeframe(ticker, timeframe, marketDataProcessor);

if (indicators) {

results[timeframe as keyof MultiTimeframeIndicators['timeframes']] = indicators;

}

}

// Calculate confluence

const confluence = this.calculateConfluence(results as MultiTimeframeIndicators['timeframes']);

return {

timeframes: results as MultiTimeframeIndicators['timeframes'],

confluence

};

}

private async analyzeTimeframe(

ticker: string,

timeframe: string,

marketDataProcessor: MarketDataProcessor

): Promise<TimeframeIndicators | null> {

try {

// Get market data for the timeframe

const endDate = new Date();

const startDate = new Date(endDate.getTime() - (100 \* this.getTimeframeMs(timeframe))); // 100 periods

const marketData = await marketDataProcessor.getNormalizedData(

ticker,

timeframe,

startDate,

endDate

);

if (marketData.length < 50) return null; // Need enough data

// Extract price and volume arrays

const closes = marketData.map(d => d.ohlcv.close);

const highs = marketData.map(d => d.ohlcv.high);

const lows = marketData.map(d => d.ohlcv.low);

const volumes = marketData.map(d => d.ohlcv.volume);

// Calculate indicators

const rsiValues = RSICalculator.calculateSeries(closes);

const macdResults = MACDCalculator.calculateSeries(closes);

const bbResults = BollingerBandsCalculator.calculateSeries(closes);

const volumeCalculator = new VolumeAnalysisCalculator();

const volumeResults = marketData.map((d, i) =>

volumeCalculator.calculate(

d.ohlcv.high,

d.ohlcv.low,

d.ohlcv.close,

d.ohlcv.volume,

i > 0 ? marketData[i - 1].ohlcv.close : undefined

)

).filter(r => r !== null) as VolumeIndicators[];

// Get latest values

const rsi = rsiValues[rsiValues.length - 1] || 50;

const macd = macdResults[macdResults.length - 1] || { macd: 0, signal: 0, histogram: 0, timestamp: new Date(), trend: 'neutral' };

const bollingerBands = bbResults[bbResults.length - 1] || { upperBand: 0, middleBand: 0, lowerBand: 0, bandwidth: 0, percentB: 0.5, squeeze: false, timestamp: new Date(), signal: 'neutral' };

const volume = volumeResults[volumeResults.length - 1] || { volume: 0, volumeMA: 0, volumeRatio: 1, obv: 0, cmf: 0, vwap: 0, volumeProfile: [], signal: 'neutral' };

// Calculate timeframe score

const score = this.calculateTimeframeScore(rsi, macd, bollingerBands, volume);

return {

rsi,

macd,

bollingerBands,

volume,

score,

weight: this.timeframeWeights[timeframe] || 0

};

} catch (error) {

console.error(`Failed to analyze timeframe ${timeframe} for ${ticker}:`, error);

return null;

}

}

private calculateTimeframeScore(

rsi: number,

macd: MACDResult,

bb: BollingerBandsResult,

volume: VolumeIndicators

): number {

let score = 0;

// RSI scoring (30% weight)

if (rsi < 30) score += 30; // Oversold - bullish

else if (rsi > 70) score += 15; // Overbought - caution

else if (rsi >= 40 && rsi <= 60) score += 20; // Neutral zone

// MACD scoring (25% weight)

if (macd.trend === 'bullish') score += 25;

else if (macd.trend === 'bearish') score += 5;

else score += 15;

// Bollinger Bands scoring (20% weight)

if (bb.percentB < 0.2) score += 20; // Near lower band - bullish

else if (bb.percentB > 0.8) score += 10; // Near upper band - caution

else score += 15;

// Volume scoring (25% weight)

if (volume.signal === 'accumulation' && volume.volumeRatio > 1.2) score += 25;

else if (volume.signal === 'distribution') score += 5;

else score += 15;

return Math.min(100, Math.max(0, score));

}

private calculateConfluence(timeframes: MultiTimeframeIndicators['timeframes']): {

bullishCount: number;

bearishCount: number;

overallSignal: 'bullish' | 'bearish' | 'neutral';

confidence: number;

} {

let bullishCount = 0;

let bearishCount = 0;

let weightedScore = 0;

let totalWeight = 0;

Object.entries(timeframes).forEach(([tf, indicators]) => {

if (!indicators) return;

const { score, weight } = indicators;

if (score >= 70) bullishCount++;

else if (score <= 40) bearishCount++;

weightedScore += score \* weight;

totalWeight += weight;

});

const finalScore = totalWeight > 0 ? weightedScore / totalWeight : 50;

let overallSignal: 'bullish' | 'bearish' | 'neutral';

if (finalScore >= 70 && bullishCount >= 2) overallSignal = 'bullish';

else if (finalScore <= 40 && bearishCount >= 2) overallSignal = 'bearish';

else overallSignal = 'neutral';

// Confidence based on confluence across timeframes

const totalTimeframes = Object.keys(timeframes).length;

const confluenceRatio = Math.max(bullishCount, bearishCount) / totalTimeframes;

const confidence = Math.round(confluenceRatio \* 100);

return {

bullishCount,

bearishCount,

overallSignal,

confidence: Math.min(100, Math.max(0, confidence))

};

}

private getTimeframeMs(timeframe: string): number {

const mapping: Record<string, number> = {

'1H': 60 \* 60 \* 1000,

'4H': 4 \* 60 \* 60 \* 1000,

'1D': 24 \* 60 \* 60 \* 1000,

'1W': 7 \* 24 \* 60 \* 60 \* 1000

};

return mapping[timeframe] || 24 \* 60 \* 60 \* 1000;

}

}

export { VolumeAnalysisCalculator, MultiTimeframeAnalyzer };

**3. ⚡ SIGNAL PROCESSING ALGORITHM**

**Weighted Scoring Methodology**

// ===================================================================

// SIGNAL PROCESSING AND SCORING ALGORITHM

// ===================================================================

// lib/signals/signalProcessor.ts

interface SignalScoreBreakdown {

rsi: { score: number; weight: number; contribution: number };

macd: { score: number; weight: number; contribution: number };

bollingerBands: { score: number; weight: number; contribution: number };

volume: { score: number; weight: number; contribution: number };

momentum: { score: number; weight: number; contribution: number };

total: number;

confidence: number;

timeframe: string;

}

interface SignalResult {

ticker: string;

finalScore: number;

signalType: 'bullish' | 'bearish' | 'neutral';

strength: 'strong' | 'valid' | 'weak';

timeframeScores: Record<string, SignalScoreBreakdown>;

confluence: {

agreements: number;

disagreements: number;

neutrals: number;

overallConfidence: number;

};

riskReward: {

entryPrice: number;

stopLoss: number;

takeProfit: number;

riskRewardRatio: number;

};

metadata: {

timestamp: Date;

marketCondition: 'trending' | 'ranging' | 'volatile';

volatility: number;

volumeProfile: 'high' | 'medium' | 'low';

};

}

class SignalProcessor {

private indicatorWeights = {

rsi: 0.25, // 25%

macd: 0.25, // 25%

bollingerBands: 0.20, // 20%

volume: 0.20, // 20%

momentum: 0.10 // 10%

};

private timeframeWeights = {

'1H': 0.40, // 40%

'4H': 0.30, // 30%

'1D': 0.20, // 20%

'1W': 0.10 // 10%

};

async processSignal(

ticker: string,

multiTimeframeData: MultiTimeframeIndicators

): Promise<SignalResult> {

// Calculate scores for each timeframe

const timeframeScores: Record<string, SignalScoreBreakdown> = {};

for (const [timeframe, indicators] of Object.entries(multiTimeframeData.timeframes)) {

if (indicators) {

timeframeScores[timeframe] = this.calculateTimeframeScore(

indicators,

timeframe

);

}

}

// Calculate final weighted score

const finalScore = this.calculateFinalScore(timeframeScores);

// Determine signal type and strength

const signalType = this.determineSignalType(finalScore, timeframeScores);

const strength = this.determineSignalStrength(finalScore);

// Calculate confluence

const confluence = this.calculateConfluence(timeframeScores);

// Calculate risk/reward levels

const riskReward = this.calculateRiskReward(

ticker,

multiTimeframeData,

signalType

);

// Assess market conditions

const metadata = this.assessMarketConditions(multiTimeframeData);

return {

ticker,

finalScore: Math.round(finalScore \* 100) / 100,

signalType,

strength,

timeframeScores,

confluence,

riskReward,

metadata

};

}

private calculateTimeframeScore(

indicators: TimeframeIndicators,

timeframe: string

): SignalScoreBreakdown {

const { rsi, macd, bollingerBands, volume } = indicators;

// RSI Score (25% weight)

const rsiScore = this.scoreRSI(rsi);

// MACD Score (25% weight)

const macdScore = this.scoreMacd(macd);

// Bollinger Bands Score (20% weight)

const bbScore = this.scoreBollingerBands(bollingerBands);

// Volume Score (20% weight)

const volumeScore = this.scoreVolume(volume);

// Momentum Score (10% weight)

const momentumScore = this.scoreMomentum(rsi, macd);

// Calculate weighted contributions

const rsiContribution = rsiScore \* this.indicatorWeights.rsi;

const macdContribution = macdScore \* this.indicatorWeights.macd;

const bbContribution = bbScore \* this.indicatorWeights.bollingerBands;

const volumeContribution = volumeScore \* this.indicatorWeights.volume;

const momentumContribution = momentumScore \* this.indicatorWeights.momentum;

const total = rsiContribution + macdContribution + bbContribution +

volumeContribution + momentumContribution;

// Calculate confidence based on indicator agreement

const scores = [rsiScore, macdScore, bbScore, volumeScore, momentumScore];

const avgScore = scores.reduce((a, b) => a + b, 0) / scores.length;

const variance = scores.reduce((sum, score) => sum + Math.pow(score - avgScore, 2), 0) / scores.length;

const confidence = Math.max(0, 100 - Math.sqrt(variance));

return {

rsi: { score: rsiScore, weight: this.indicatorWeights.rsi, contribution: rsiContribution },

macd: { score: macdScore, weight: this.indicatorWeights.macd, contribution: macdContribution },

bollingerBands: { score: bbScore, weight: this.indicatorWeights.bollingerBands, contribution: bbContribution },

volume: { score: volumeScore, weight: this.indicatorWeights.volume, contribution: volumeContribution },

momentum: { score: momentumScore, weight: this.indicatorWeights.momentum, contribution: momentumContribution },

total: Math.round(total \* 100) / 100,

confidence: Math.round(confidence \* 100) / 100,

timeframe

};

}

private scoreRSI(rsi: number): number {

// RSI scoring logic

if (rsi <= 25) return 95; // Extremely oversold - very bullish

if (rsi <= 30) return 85; // Oversold - bullish

if (rsi <= 35) return 75; // Moderately oversold

if (rsi <= 45) return 65; // Slightly oversold

if (rsi <= 55) return 50; // Neutral

if (rsi <= 65) return 40; // Slightly overbought

if (rsi <= 70) return 30; // Moderately overbought

if (rsi <= 75) return 20; // Overbought - bearish

return 10; // Extremely overbought - very bearish

}

private scoreMacd(macd: MACDResult): number {

let score = 50; // Base neutral score

// MACD line position relative to signal line

if (macd.macd > macd.signal) {

score += 25; // Bullish when MACD > Signal

} else {

score -= 25; // Bearish when MACD < Signal

}

// Histogram momentum

if (macd.histogram > 0) {

score += 15; // Positive momentum

} else {

score -= 15; // Negative momentum

}

// Trend strength

if (macd.trend === 'bullish') {

score += 10;

} else if (macd.trend === 'bearish') {

score -= 10;

}

return Math.max(0, Math.min(100, score));

}

private scoreBollingerBands(bb: BollingerBandsResult): number {

let score = 50; // Base neutral score

// Position within bands (%B)

if (bb.percentB <= 0.1) {

score += 40; // Near lower band - very bullish

} else if (bb.percentB <= 0.2) {

score += 30; // Below lower region - bullish

} else if (bb.percentB <= 0.4) {

score += 10; // Lower half - slightly bullish

} else if (bb.percentB <= 0.6) {

score += 0; // Neutral zone

} else if (bb.percentB <= 0.8) {

score -= 10; // Upper half - slightly bearish

} else if (bb.percentB <= 0.9) {

score -= 30; // Above upper region - bearish

} else {

score -= 40; // Near upper band - very bearish

}

// Band squeeze (low volatility - potential breakout)

if (bb.squeeze) {

score += 10; // Squeeze adds bullish bias for breakout potential

}

// Bandwidth consideration

if (bb.bandwidth < 0.05) {

score += 5; // Very tight bands

} else if (bb.bandwidth > 0.2) {

score -= 5; // Very wide bands - high volatility

}

return Math.max(0, Math.min(100, score));

}

private scoreVolume(volume: VolumeIndicators): number {

let score = 50; // Base neutral score

// Volume ratio scoring

if (volume.volumeRatio > 2.0) {

score += 30; // Very high volume

} else if (volume.volumeRatio > 1.5) {

score += 20; // High volume

} else if (volume.volumeRatio > 1.2) {

score += 10; // Above average volume

} else if (volume.volumeRatio < 0.5) {

score -= 20; // Very low volume

} else if (volume.volumeRatio < 0.8) {

score -= 10; // Below average volume

}

// Chaikin Money Flow

if (volume.cmf > 0.2) {

score += 20; // Strong buying pressure

} else if (volume.cmf > 0.1) {

score += 10; // Moderate buying pressure

} else if (volume.cmf < -0.2) {

score -= 20; // Strong selling pressure

} else if (volume.cmf < -0.1) {

score -= 10; // Moderate selling pressure

}

// Volume signal

if (volume.signal === 'accumulation') {

score += 15;

} else if (volume.signal === 'distribution') {

score -= 15;

}

return Math.max(0, Math.min(100, score));

}

private scoreMomentum(rsi: number, macd: MACDResult): number {

let score = 50; // Base neutral score

// RSI momentum (rate of change)

if (rsi < 30 && macd.trend === 'bullish') {

score += 25; // Oversold with bullish MACD

} else if (rsi > 70 && macd.trend === 'bearish') {

score -= 25; // Overbought with bearish MACD

}

// MACD histogram momentum

if (macd.histogram > 0) {

score += 15;

} else {

score -= 15;

}

// Divergence would be checked here if historical data available

// For now, we'll use the current momentum indicators

return Math.max(0, Math.min(100, score));

}

private calculateFinalScore(timeframeScores: Record<string, SignalScoreBreakdown>): number {

let weightedSum = 0;

let totalWeight = 0;

Object.entries(timeframeScores).forEach(([timeframe, scoreData]) => {

const weight = this.timeframeWeights[timeframe] || 0;

weightedSum += scoreData.total \* weight;

totalWeight += weight;

});

return totalWeight > 0 ? weightedSum / totalWeight : 0;

}

private determineSignalType(

finalScore: number,

timeframeScores: Record<string, SignalScoreBreakdown>

): 'bullish' | 'bearish' | 'neutral' {

// Primary determination based on final score

if (finalScore >= 65) return 'bullish';

if (finalScore <= 35) return 'bearish';

// Secondary check: timeframe agreement

const timeframeSignals = Object.values(timeframeScores).map(score => {

if (score.total >= 65) return 'bullish';

if (score.total <= 35) return 'bearish';

return 'neutral';

});

const bullishCount = timeframeSignals.filter(s => s === 'bullish').length;

const bearishCount = timeframeSignals.filter(s => s === 'bearish').length;

if (bullishCount > bearishCount && bullishCount >= 2) return 'bullish';

if (bearishCount > bullishCount && bearishCount >= 2) return 'bearish';

return 'neutral';

}

private determineSignalStrength(finalScore: number): 'strong' | 'valid' | 'weak' {

if (finalScore >= 80) return 'strong';

if (finalScore >= 60) return 'valid';

return 'weak';

}

private calculateConfluence(timeframeScores: Record<string, SignalScoreBreakdown>): {

agreements: number;

disagreements: number;

neutrals: number;

overallConfidence: number;

} {

const signals = Object.values(timeframeScores).map(score => {

if (score.total >= 65) return 'bullish';

if (score.total <= 35) return 'bearish';

return 'neutral';

});

const bullishCount = signals.filter(s => s === 'bullish').length;

const bearishCount = signals.filter(s => s === 'bearish').length;

const neutralCount = signals.filter(s => s === 'neutral').length;

const maxCount = Math.max(bullishCount, bearishCount, neutralCount);

const totalSignals = signals.length;

const overallConfidence = totalSignals > 0 ? (maxCount / totalSignals) \* 100 : 0;

return {

agreements: maxCount,

disagreements: totalSignals - maxCount - neutralCount,

neutrals: neutralCount,

overallConfidence: Math.round(overallConfidence \* 100) / 100

};

}

private calculateRiskReward(

ticker: string,

multiTimeframeData: MultiTimeframeIndicators,

signalType: 'bullish' | 'bearish' | 'neutral'

): {

entryPrice: number;

stopLoss: number;

takeProfit: number;

riskRewardRatio: number;

} {

// Get current price from 1H timeframe (most recent)

const currentTimeframe = multiTimeframeData.timeframes['1H'];

const entryPrice = currentTimeframe?.bollingerBands.middleBand || 0;

if (signalType === 'neutral' || entryPrice === 0) {

return {

entryPrice,

stopLoss: entryPrice,

takeProfit: entryPrice,

riskRewardRatio: 0

};

}

let stopLoss: number;

let takeProfit: number;

if (signalType === 'bullish') {

// For bullish signals

const dailyBB = multiTimeframeData.timeframes['1D']?.bollingerBands;

stopLoss = dailyBB?.lowerBand || entryPrice \* 0.98; // 2% stop loss fallback

// Target 2:1 risk-reward ratio

const riskAmount = entryPrice - stopLoss;

takeProfit = entryPrice + (riskAmount \* 2);

} else {

// For bearish signals (short)

const dailyBB = multiTimeframeData.timeframes['1D']?.bollingerBands;

stopLoss = dailyBB?.upperBand || entryPrice \* 1.02; // 2% stop loss fallback

// Target 2:1 risk-reward ratio

const riskAmount = stopLoss - entryPrice;

takeProfit = entryPrice - (riskAmount \* 2);

}

const riskAmount = Math.abs(entryPrice - stopLoss);

const rewardAmount = Math.abs(takeProfit - entryPrice);

const riskRewardRatio = riskAmount > 0 ? rewardAmount / riskAmount : 0;

return {

entryPrice: Math.round(entryPrice \* 100) / 100,

stopLoss: Math.round(stopLoss \* 100) / 100,

takeProfit: Math.round(takeProfit \* 100) / 100,

riskRewardRatio: Math.round(riskRewardRatio \* 100) / 100

};

}

private assessMarketConditions(multiTimeframeData: MultiTimeframeIndicators): {

timestamp: Date;

marketCondition: 'trending' | 'ranging' | 'volatile';

volatility: number;

volumeProfile: 'high' | 'medium' | 'low';

} {

const dailyData = multiTimeframeData.timeframes['1D'];

// Assess market condition based on Bollinger Bands width

let marketCondition: 'trending' | 'ranging' | 'volatile' = 'ranging';

let volatility = 0;

if (dailyData?.bollingerBands) {

volatility = dailyData.bollingerBands.bandwidth;

if (volatility > 0.15) {

marketCondition = 'volatile';

} else if (volatility > 0.08) {

marketCondition = 'trending';

} else {

marketCondition = 'ranging';

}

}

// Assess volume profile

let volumeProfile: 'high' | 'medium' | 'low' = 'medium';

if (dailyData?.volume) {

if (dailyData.volume.volumeRatio > 1.5) {

volumeProfile = 'high';

} else if (dailyData.volume.volumeRatio < 0.8) {

volumeProfile = 'low';

}

}

return {

timestamp: new Date(),

marketCondition,

volatility: Math.round(volatility \* 10000) / 10000,

volumeProfile

};

}

}

export { SignalProcessor, SignalResult, SignalScoreBreakdown };

**4. 🔄 DATA PIPELINE ARCHITECTURE**

**Real-time Data Ingestion**

// ===================================================================

// DATA PIPELINE ARCHITECTURE

// ===================================================================

// lib/pipeline/dataPipeline.ts

interface PipelineConfig {

batchSize: number;

maxRetries: number;

retryDelay: number;

parallelProcessing: number;

enableRealTime: boolean;

qualityThreshold: number;

}

interface DataQualityMetrics {

totalRecords: number;

validRecords: number;

invalidRecords: number;

qualityScore: number;

errors: string[];

warnings: string[];

}

interface PipelineStatus {

isRunning: boolean;

lastRun: Date;

recordsProcessed: number;

errorsCount: number;

averageProcessingTime: number;

throughput: number; // records per second

}

class MarketDataPipeline {

private config: PipelineConfig;

private status: PipelineStatus;

private polygonClient: PolygonApiClient;

private dataProcessor: MarketDataProcessor;

private signalProcessor: SignalProcessor;

private processingQueue: Queue<ProcessingTask>;

private metrics: Map<string, DataQualityMetrics>;

constructor(

config: PipelineConfig,

polygonClient: PolygonApiClient,

dataProcessor: MarketDataProcessor,

signalProcessor: SignalProcessor

) {

this.config = config;

this.polygonClient = polygonClient;

this.dataProcessor = dataProcessor;

this.signalProcessor = signalProcessor;

this.processingQueue = new Queue<ProcessingTask>();

this.metrics = new Map();

this.status = {

isRunning: false,

lastRun: new Date(),

recordsProcessed: 0,

errorsCount: 0,

averageProcessingTime: 0,

throughput: 0

};

}

// Start the real-time data pipeline

async startRealTimePipeline(tickers: string[]): Promise<void> {

console.log(`Starting real-time pipeline for ${tickers.length} tickers`);

this.status.isRunning = true;

this.status.lastRun = new Date();

try {

// Initialize WebSocket connection for real-time data

if (this.config.enableRealTime) {

await this.initializeRealTimeStream(tickers);

}

// Start periodic batch processing

await this.startBatchProcessing(tickers);

// Start signal generation

await this.startSignalGeneration(tickers);

} catch (error) {

console.error('Failed to start real-time pipeline:', error);

this.status.isRunning = false;

throw error;

}

}

// Initialize real-time WebSocket stream

private async initializeRealTimeStream(tickers: string[]): Promise<void> {

const ws = this.polygonClient.initializeRealTimeStream(

tickers,

(data) => this.handleRealTimeData(data)

);

if (!ws) {

throw new Error('Failed to initialize WebSocket connection');

}

console.log('Real-time WebSocket stream initialized');

}

// Handle incoming real-time data

private async handleRealTimeData(data: any): Promise<void> {

try {

// Parse WebSocket message

const parsedData = this.parseWebSocketMessage(data);

if (!parsedData) return;

// Add to processing queue

this.processingQueue.enqueue({

type: 'realtime',

data: parsedData,

timestamp: new Date(),

priority: 'high'

});

// Process immediately if queue is not too full

if (this.processingQueue.size() < this.config.batchSize) {

await this.processQueueBatch();

}

} catch (error) {

console.error('Error handling real-time data:', error);

this.status.errorsCount++;

}

}

// Start batch processing for historical data

private async startBatchProcessing(tickers: string[]): Promise<void> {

const batchSize = 50; // Process 50 tickers at a time

for (let i = 0; i < tickers.length; i += batchSize) {

const batch = tickers.slice(i, i + batchSize);

// Add batch to processing queue

this.processingQueue.enqueue({

type: 'batch',

data: { tickers: batch },

timestamp: new Date(),

priority: 'medium'

});

}

// Start processing queue

this.processQueue();

}

// Process tasks from the queue

private async processQueue(): Promise<void> {

while (this.status.isRunning) {

if (this.processingQueue.size() > 0) {

await this.processQueueBatch();

} else {

// Wait before checking again

await this.delay(1000);

}

}

}

// Process a batch of tasks from the queue

private async processQueueBatch(): Promise<void> {

const tasks: ProcessingTask[] = [];

const batchSize = Math.min(this.config.batchSize, this.processingQueue.size());

// Dequeue tasks

for (let i = 0; i < batchSize; i++) {

const task = this.processingQueue.dequeue();

if (task) tasks.push(task);

}

if (tasks.length === 0) return;

console.log(`Processing batch of ${tasks.length} tasks`);

const startTime = Date.now();

try {

// Process tasks in parallel

const promises = tasks.map(task => this.processTask(task));

await Promise.allSettled(promises);

// Update metrics

const processingTime = Date.now() - startTime;

this.updateMetrics(tasks.length, processingTime);

} catch (error) {

console.error('Error processing queue batch:', error);

this.status.errorsCount++;

}

}

// Process individual task

private async processTask(task: ProcessingTask): Promise<void> {

try {

switch (task.type) {

case 'realtime':

await this.processRealTimeTask(task);

break;

case 'batch':

await this.processBatchTask(task);

break;

case 'signal\_generation':

await this.processSignalGenerationTask(task);

break;

default:

console.warn(`Unknown task type: ${task.type}`);

}

} catch (error) {

console.error(`Error processing task ${task.type}:`, error);

// Retry logic

if (task.retries < this.config.maxRetries) {

task.retries = (task.retries || 0) + 1;

task.priority = 'low'; // Lower priority for retries

// Add back to queue with delay

setTimeout(() => {

this.processingQueue.enqueue(task);

}, this.config.retryDelay \* task.retries);

}

}

}

// Process real-time data task

private async processRealTimeTask(task: ProcessingTask): Promise<void> {

const { ticker, price, volume, timestamp } = task.data;

// Validate data quality

const qualityCheck = this.validateRealTimeData(task.data);

if (!qualityCheck.isValid) {

console.warn(`Low quality real-time data for ${ticker}:`, qualityCheck.errors);

return;

}

// Store in database

await this.storeRealTimeData(ticker, task.data);

// Update running calculations

await this.updateRunningCalculations(ticker, task.data);

// Check if signal generation is needed

if (this.shouldGenerateSignal(ticker)) {

this.processingQueue.enqueue({

type: 'signal\_generation',

data: { ticker },

timestamp: new Date(),

priority: 'high'

});

}

}

// Process batch data task

private async processBatchTask(task: ProcessingTask): Promise<void> {

const { tickers } = task.data;

const timeframes = ['1H', '4H', '1D', '1W'];

for (const ticker of tickers) {

for (const timeframe of timeframes) {

try {

// Get historical data

const endDate = new Date();

const startDate = new Date(endDate.getTime() - this.getTimeframeLookback(timeframe));

const marketData = await this.polygonClient.getAggregates(

ticker,

this.getTimeframeMultiplier(timeframe),

this.getTimeframeUnit(timeframe),

startDate.toISOString().split('T')[0],

endDate.toISOString().split('T')[0]

);

if (marketData.length > 0) {

// Normalize and store data

await this.dataProcessor.normalizeAndStore(

ticker,

marketData,

timeframe,

'polygon'

);

// Update quality metrics

this.updateQualityMetrics(ticker, timeframe, marketData);

}

} catch (error) {

console.error(`Error processing batch data for ${ticker} ${timeframe}:`, error);

}

}

}

}

// Process signal generation task

private async processSignalGenerationTask(task: ProcessingTask): Promise<void> {

const { ticker } = task.data;

try {

// Get multi-timeframe analysis

const multiTimeframeAnalyzer = new MultiTimeframeAnalyzer();

const analysis = await multiTimeframeAnalyzer.analyzeAllTimeframes(

ticker,

this.dataProcessor

);

// Generate signal

const signal = await this.signalProcessor.processSignal(ticker, analysis);

// Store signal if it meets criteria

if (signal.strength !== 'weak' && signal.finalScore >= 60) {

await this.storeSignal(signal);

// Trigger alerts for strong signals

if (signal.strength === 'strong') {

await this.triggerSignalAlert(signal);

}

}

} catch (error) {

console.error(`Error generating signal for ${ticker}:`, error);

}

}

// Data validation methods

private validateRealTimeData(data: any): { isValid: boolean; errors: string[] } {

const errors: string[] = [];

if (!data.ticker || typeof data.ticker !== 'string') {

errors.push('Invalid ticker');

}

if (!data.price || typeof data.price !== 'number' || data.price <= 0) {

errors.push('Invalid price');

}

if (!data.volume || typeof data.volume !== 'number' || data.volume < 0) {

errors.push('Invalid volume');

}

if (!data.timestamp || isNaN(new Date(data.timestamp).getTime())) {

errors.push('Invalid timestamp');

}

// Check for extreme price movements

if (data.price && data.prevPrice) {

const changePercent = Math.abs((data.price - data.prevPrice) / data.prevPrice);

if (changePercent > 0.2) { // 20% change

errors.push('Extreme price movement detected');

}

}

return {

isValid: errors.length === 0,

errors

};

}

// Data quality assessment

private updateQualityMetrics(

ticker: string,

timeframe: string,

marketData: MarketData[]

): void {

const key = `${ticker}\_${timeframe}`;

let validRecords = 0;

let invalidRecords = 0;

const errors: string[] = [];

const warnings: string[] = [];

marketData.forEach(data => {

// Validate OHLCV data

if (this.isValidOHLCV(data)) {

validRecords++;

} else {

invalidRecords++;

errors.push(`Invalid OHLCV data for ${ticker} at ${data.timestamp}`);

}

// Check for data gaps

if (data.volume === 0) {

warnings.push(`Zero volume for ${ticker} at ${data.timestamp}`);

}

});

const totalRecords = validRecords + invalidRecords;

const qualityScore = totalRecords > 0 ? (validRecords / totalRecords) \* 100 : 0;

this.metrics.set(key, {

totalRecords,

validRecords,

invalidRecords,

qualityScore,

errors,

warnings

});

}

private isValidOHLCV(data: MarketData): boolean {

return data.open > 0 &&

data.high > 0 &&

data.low > 0 &&

data.close > 0 &&

data.high >= data.low &&

data.open <= data.high &&

data.open >= data.low &&

data.close <= data.high &&

data.close >= data.low &&

data.volume >= 0;

}

// Helper methods

private parseWebSocketMessage(data: any): any | null {

try {

if (Array.isArray(data)) {

return data.find(item => item.ev === 'T'); // Trade event

}

return data.ev === 'T' ? data : null;

} catch {

return null;

}

}

private shouldGenerateSignal(ticker: string): boolean {

// Generate signals every 5 minutes for real-time data

const lastSignalTime = this.getLastSignalTime(ticker);

const now = Date.now();

return now - lastSignalTime > 5 \* 60 \* 1000; // 5 minutes

}

private getLastSignalTime(ticker: string): number {

// This would be stored in cache or database

return 0; // Placeholder

}

private async storeRealTimeData(ticker: string, data: any): Promise<void> {

// Store in time-series optimized table

const { error } = await supabaseClient

.from('realtime\_quotes')

.insert([{

ticker,

price: data.price,

volume: data.volume,

timestamp: new Date(data.timestamp).toISOString(),

created\_at: new Date().toISOString()

}]);

if (error) {

console.error('Failed to store real-time data:', error);

}

}

private async updateRunningCalculations(ticker: string, data: any): Promise<void> {

// Update running averages, indicators, etc.

// This would use efficient sliding window calculations

}

private async storeSignal(signal: SignalResult): Promise<void> {

const { error } = await supabaseClient

.from('trading\_signals')

.insert([{

ticker: signal.ticker,

final\_score: signal.finalScore,

signal\_type: signal.signalType,

strength: signal.strength,

entry\_price: signal.riskReward.entryPrice,

stop\_loss: signal.riskReward.stopLoss,

take\_profit: signal.riskReward.takeProfit,

confidence: signal.confluence.overallConfidence,

timeframe\_scores: signal.timeframeScores,

metadata: signal.metadata,

triggered\_at: new Date().toISOString()

}]);

if (error) {

console.error('Failed to store signal:', error);

}

}

private async triggerSignalAlert(signal: SignalResult): Promise<void> {

// Trigger Make.com webhook for alerts

try {

await fetch(process.env.MAKE\_WEBHOOK\_URL!, {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({

type: 'signal\_alert',

signal: {

ticker: signal.ticker,

score: signal.finalScore,

type: signal.signalType,

strength: signal.strength,

entryPrice: signal.riskReward.entryPrice

}

})

});

} catch (error) {

console.error('Failed to trigger signal alert:', error);

}

}

private updateMetrics(recordsProcessed: number, processingTime: number): void {

this.status.recordsProcessed += recordsProcessed;

this.status.averageProcessingTime =

(this.status.averageProcessingTime + processingTime) / 2;

this.status.throughput = recordsProcessed / (processingTime / 1000);

}

private getTimeframeLookback(timeframe: string): number {

const lookbacks: Record<string, number> = {

'1H': 7 \* 24 \* 60 \* 60 \* 1000, // 7 days

'4H': 30 \* 24 \* 60 \* 60 \* 1000, // 30 days

'1D': 365 \* 24 \* 60 \* 60 \* 1000, // 1 year

'1W': 2 \* 365 \* 24 \* 60 \* 60 \* 1000 // 2 years

};

return lookbacks[timeframe] || 30 \* 24 \* 60 \* 60 \* 1000;

}

private getTimeframeMultiplier(timeframe: string): number {

const multipliers: Record<string, number> = {

'1H': 1,

'4H': 4,

'1D': 1,

'1W': 1

};

return multipliers[timeframe] || 1;

}

private getTimeframeUnit(timeframe: string): 'minute' | 'hour' | 'day' | 'week' {

const units: Record<string, any> = {

'1H': 'hour',

'4H': 'hour',

'1D': 'day',

'1W': 'week'

};

return units[timeframe] || 'day';

}

private delay(ms: number): Promise<void> {

return new Promise(resolve => setTimeout(resolve, ms));

}

// Public methods for monitoring

getStatus(): PipelineStatus {

return { ...this.status };

}

getQualityMetrics(): Map<string, DataQualityMetrics> {

return new Map(this.metrics);

}

async stopPipeline(): Promise<void> {

console.log('Stopping data pipeline...');

this.status.isRunning = false;

}

}

// Queue implementation

class Queue<T> {

private items: T[] = [];

enqueue(item: T): void {

this.items.push(item);

}

dequeue(): T | undefined {

return this.items.shift();

}

size(): number {

return this.items.length;

}

}

interface ProcessingTask {

type: 'realtime' | 'batch' | 'signal\_generation';

data: any;

timestamp: Date;

priority: 'high' | 'medium' | 'low';

retries?: number;

}

export { MarketDataPipeline, PipelineConfig, DataQualityMetrics, PipelineStatus };

// \*\*5. 📊 PORTFOLIO & TRADING LOGIC\*\*

// ===================================================================

// PAPER TRADING EXECUTION ENGINE

// ===================================================================

interface PaperTrade {

id: string;

userId: string;

ticker: string;

signalId?: string;

tradeType: 'buy' | 'sell' | 'short' | 'cover';

orderType: 'market' | 'limit' | 'stop\_loss' | 'take\_profit';

shares: number;

entryPrice: number;

currentPrice: number;

stopLoss?: number;

takeProfit?: number;

status: 'open' | 'closed' | 'cancelled';

entryTime: Date;

exitTime?: Date;

exitPrice?: number;

pnl: number;

pnlPercent: number;

commission: number;

positionValue: number;

metadata: {

signalScore?: number;

riskRewardRatio?: number;

holdingPeriod?: number;

maxDrawdown?: number;

maxProfit?: number;

};

}

interface Portfolio {

id: string;

userId: string;

name: string;

balance: number;

equity: number;

buyingPower: number;

totalPnL: number;

totalPnLPercent: number;

dayPnL: number;

dayPnLPercent: number;

positions: Position[];

trades: PaperTrade[];

performance: PerformanceMetrics;

riskMetrics: RiskMetrics;

createdAt: Date;

updatedAt: Date;

}

interface Position {

ticker: string;

shares: number;

avgCost: number;

currentPrice: number;

marketValue: number;

unrealizedPnL: number;

unrealizedPnLPercent: number;

dayChange: number;

dayChangePercent: number;

weight: number; // percentage of portfolio

riskAmount: number;

stopLoss?: number;

takeProfit?: number;

}

class PaperTradingEngine {

private portfolio: Portfolio;

private marketDataProcessor: MarketDataProcessor;

private riskManager: RiskManager;

private commissionRate: number = 0.001; // 0.1% commission

constructor(

portfolio: Portfolio,

marketDataProcessor: MarketDataProcessor,

riskManager: RiskManager

) {

this.portfolio = portfolio;

this.marketDataProcessor = marketDataProcessor;

this.riskManager = riskManager;

}

// Execute a paper trade order

async executeTrade(orderRequest: TradeOrderRequest): Promise<PaperTrade | null> {

try {

// Validate order

const validation = await this.validateOrder(orderRequest);

if (!validation.isValid) {

throw new Error(`Order validation failed: ${validation.errors.join(', ')}`);

}

// Get current market price

const currentPrice = await this.getCurrentPrice(orderRequest.ticker);

if (!currentPrice) {

throw new Error(`Unable to get current price for ${orderRequest.ticker}`);

}

// Calculate position size if not specified

let shares = orderRequest.shares;

if (orderRequest.useRiskBasedSizing) {

shares = this.calculatePositionSize(

currentPrice,

orderRequest.stopLoss,

orderRequest.riskPercent || 2

);

}

// Create trade record

const trade: PaperTrade = {

id: this.generateTradeId(),

userId: this.portfolio.userId,

ticker: orderRequest.ticker,

signalId: orderRequest.signalId,

tradeType: orderRequest.tradeType,

orderType: orderRequest.orderType,

shares,

entryPrice: currentPrice,

currentPrice,

stopLoss: orderRequest.stopLoss,

takeProfit: orderRequest.takeProfit,

status: 'open',

entryTime: new Date(),

pnl: 0,

pnlPercent: 0,

commission: this.calculateCommission(shares, currentPrice),

positionValue: shares \* currentPrice,

metadata: {

signalScore: orderRequest.signalScore,

riskRewardRatio: orderRequest.riskRewardRatio,

}

};

// Update portfolio

await this.updatePortfolioForTrade(trade);

// Store trade

await this.storeTrade(trade);

console.log(`Executed ${trade.tradeType} order: ${trade.shares} shares of ${trade.ticker} at $${trade.entryPrice}`);

return trade;

} catch (error) {

console.error('Failed to execute trade:', error);

return null;

}

}

// Position sizing based on risk percentage

private calculatePositionSize(

entryPrice: number,

stopLoss: number | undefined,

riskPercent: number

): number {

if (!stopLoss || stopLoss >= entryPrice) {

// Default to 1% of portfolio if no stop loss

return Math.floor((this.portfolio.balance \* 0.01) / entryPrice);

}

const riskPerShare = entryPrice - stopLoss;

const totalRiskAmount = (this.portfolio.balance \* riskPercent) / 100;

const shares = Math.floor(totalRiskAmount / riskPerShare);

// Ensure position doesn't exceed maximum position size (20% of portfolio)

const maxPositionValue = this.portfolio.balance \* 0.2;

const maxShares = Math.floor(maxPositionValue / entryPrice);

return Math.min(shares, maxShares);

}

// Update current prices and P&L for all positions

async updatePortfolio(): Promise<void> {

for (const position of this.portfolio.positions) {

const currentPrice = await this.getCurrentPrice(position.ticker);

if (currentPrice) {

this.updatePositionPnL(position, currentPrice);

}

}

// Update open trades

for (const trade of this.portfolio.trades.filter(t => t.status === 'open')) {

const currentPrice = await this.getCurrentPrice(trade.ticker);

if (currentPrice) {

trade.currentPrice = currentPrice;

this.updateTradePnL(trade);

// Check for stop loss or take profit triggers

await this.checkExitConditions(trade);

}

}

// Recalculate portfolio metrics

this.calculatePortfolioMetrics();

// Store updated portfolio

await this.storePortfolio();

}

private updatePositionPnL(position: Position, currentPrice: number): void {

position.currentPrice = currentPrice;

position.marketValue = position.shares \* currentPrice;

position.unrealizedPnL = (currentPrice - position.avgCost) \* position.shares;

position.unrealizedPnLPercent = (position.unrealizedPnL / (position.avgCost \* position.shares)) \* 100;

position.weight = (position.marketValue / this.portfolio.equity) \* 100;

}

private updateTradePnL(trade: PaperTrade): void {

const priceDiff = trade.currentPrice - trade.entryPrice;

const multiplier = trade.tradeType === 'buy' ? 1 : -1; // Short positions inverse

trade.pnl = (priceDiff \* trade.shares \* multiplier) - trade.commission;

trade.pnlPercent = (trade.pnl / trade.positionValue) \* 100;

// Update metadata

if (!trade.metadata.maxProfit || trade.pnl > trade.metadata.maxProfit) {

trade.metadata.maxProfit = trade.pnl;

}

if (!trade.metadata.maxDrawdown || trade.pnl < trade.metadata.maxDrawdown) {

trade.metadata.maxDrawdown = trade.pnl;

}

}

// Check stop loss and take profit conditions

private async checkExitConditions(trade: PaperTrade): Promise<void> {

let shouldExit = false;

let exitReason = '';

if (trade.tradeType === 'buy') {

// Long position checks

if (trade.stopLoss && trade.currentPrice <= trade.stopLoss) {

shouldExit = true;

exitReason = 'Stop Loss Triggered';

} else if (trade.takeProfit && trade.currentPrice >= trade.takeProfit) {

shouldExit = true;

exitReason = 'Take Profit Triggered';

}

} else {

// Short position checks

if (trade.stopLoss && trade.currentPrice >= trade.stopLoss) {

shouldExit = true;

exitReason = 'Stop Loss Triggered';

} else if (trade.takeProfit && trade.currentPrice <= trade.takeProfit) {

shouldExit = true;

exitReason = 'Take Profit Triggered';

}

}

if (shouldExit) {

await this.closeTrade(trade.id, exitReason);

}

}

// Close a trade

async closeTrade(tradeId: string, reason: string = 'Manual Close'): Promise<boolean> {

const trade = this.portfolio.trades.find(t => t.id === tradeId && t.status === 'open');

if (!trade) return false;

try {

const currentPrice = await this.getCurrentPrice(trade.ticker);

if (!currentPrice) return false;

// Update trade record

trade.status = 'closed';

trade.exitTime = new Date();

trade.exitPrice = currentPrice;

trade.currentPrice = currentPrice;

// Calculate final P&L

this.updateTradePnL(trade);

// Calculate holding period

trade.metadata.holdingPeriod = Math.round(

(trade.exitTime!.getTime() - trade.entryTime.getTime()) / (1000 \* 60 \* 60 \* 24)

);

// Update portfolio balance and positions

this.portfolio.balance += trade.pnl;

await this.updatePortfolioPosition(trade);

console.log(`Closed trade: ${trade.ticker} | P&L: $${trade.pnl.toFixed(2)} | Reason: ${reason}`);

return true;

} catch (error) {

console.error('Failed to close trade:', error);

return false;

}

}

private async getCurrentPrice(ticker: string): Promise<number | null> {

try {

const snapshot = await this.marketDataProcessor.polygonClient.getSnapshot(ticker);

return snapshot?.value || null;

} catch (error) {

console.error(`Failed to get current price for ${ticker}:`, error);

return null;

}

}

private generateTradeId(): string {

return `trade\_${Date.now()}\_${Math.random().toString(36).substr(2, 9)}`;

}

private calculateCommission(shares: number, price: number): number {

return shares \* price \* this.commissionRate;

}

private async validateOrder(order: TradeOrderRequest): Promise<ValidationResult> {

const errors: string[] = [];

// Check buying power

const requiredAmount = order.shares \* (await this.getCurrentPrice(order.ticker) || 0);

if (requiredAmount > this.portfolio.buyingPower) {

errors.push('Insufficient buying power');

}

// Check position limits

const currentPosition = this.portfolio.positions.find(p => p.ticker === order.ticker);

const newWeight = ((currentPosition?.marketValue || 0) + requiredAmount) / this.portfolio.equity;

if (newWeight > 0.2) { // 20% max position size

errors.push('Position size exceeds 20% limit');

}

// Risk management checks

const riskCheck = await this.riskManager.validateTrade(order, this.portfolio);

if (!riskCheck.isValid) {

errors.push(...riskCheck.errors);

}

return {

isValid: errors.length === 0,

errors

};

}

}

// ===================================================================

// PERFORMANCE METRICS COMPUTATION

// ===================================================================

interface PerformanceMetrics {

totalReturn: number;

totalReturnPercent: number;

annualizedReturn: number;

volatility: number;

sharpeRatio: number;

maxDrawdown: number;

winRate: number;

profitFactor: number;

averageWin: number;

averageLoss: number;

largestWin: number;

largestLoss: number;

totalTrades: number;

winningTrades: number;

losingTrades: number;

averageHoldingPeriod: number;

monthlyReturns: MonthlyReturn[];

yearlyReturns: YearlyReturn[];

}

interface MonthlyReturn {

year: number;

month: number;

return: number;

returnPercent: number;

}

interface YearlyReturn {

year: number;

return: number;

returnPercent: number;

trades: number;

}

class PerformanceCalculator {

static calculateMetrics(portfolio: Portfolio, trades: PaperTrade[]): PerformanceMetrics {

const closedTrades = trades.filter(t => t.status === 'closed');

return {

totalReturn: this.calculateTotalReturn(portfolio),

totalReturnPercent: this.calculateTotalReturnPercent(portfolio),

annualizedReturn: this.calculateAnnualizedReturn(portfolio),

volatility: this.calculateVolatility(portfolio),

sharpeRatio: this.calculateSharpeRatio(portfolio),

maxDrawdown: this.calculateMaxDrawdown(portfolio),

winRate: this.calculateWinRate(closedTrades),

profitFactor: this.calculateProfitFactor(closedTrades),

averageWin: this.calculateAverageWin(closedTrades),

averageLoss: this.calculateAverageLoss(closedTrades),

largestWin: this.calculateLargestWin(closedTrades),

largestLoss: this.calculateLargestLoss(closedTrades),

totalTrades: closedTrades.length,

winningTrades: closedTrades.filter(t => t.pnl > 0).length,

losingTrades: closedTrades.filter(t => t.pnl < 0).length,

averageHoldingPeriod: this.calculateAverageHoldingPeriod(closedTrades),

monthlyReturns: this.calculateMonthlyReturns(closedTrades),

yearlyReturns: this.calculateYearlyReturns(closedTrades)

};

}

private static calculateWinRate(trades: PaperTrade[]): number {

if (trades.length === 0) return 0;

const winningTrades = trades.filter(t => t.pnl > 0).length;

return (winningTrades / trades.length) \* 100;

}

private static calculateProfitFactor(trades: PaperTrade[]): number {

const wins = trades.filter(t => t.pnl > 0);

const losses = trades.filter(t => t.pnl < 0);

const totalWins = wins.reduce((sum, t) => sum + t.pnl, 0);

const totalLosses = Math.abs(losses.reduce((sum, t) => sum + t.pnl, 0));

return totalLosses > 0 ? totalWins / totalLosses : 0;

}

private static calculateSharpeRatio(portfolio: Portfolio): number {

// Simplified Sharpe ratio calculation

const riskFreeRate = 0.02; // 2% annual risk-free rate

const excessReturn = portfolio.totalPnLPercent - riskFreeRate;

const volatility = this.calculateVolatility(portfolio);

return volatility > 0 ? excessReturn / volatility : 0;

}

private static calculateMaxDrawdown(portfolio: Portfolio): number {

// This would require historical equity curve data

// Simplified calculation based on current metrics

return portfolio.totalPnLPercent < 0 ? Math.abs(portfolio.totalPnLPercent) : 0;

}

private static calculateVolatility(portfolio: Portfolio): number {

// Simplified volatility calculation

// In production, this would use daily returns

return Math.abs(portfolio.dayPnLPercent) \* Math.sqrt(252); // Annualized

}

}

// ===================================================================

// \*\*6. 🔍 MARKET SCANNING ENGINE\*\*

// ===================================================================

interface ScanCriteria {

minPrice: number;

maxPrice: number;

minVolume: number;

minMarketCap: number;

maxMarketCap: number;

sectors: string[];

exchanges: string[];

minScore: number;

indicators: IndicatorCriteria[];

}

interface IndicatorCriteria {

type: 'rsi' | 'macd' | 'volume' | 'price\_change';

operator: 'greater\_than' | 'less\_than' | 'between' | 'crosses\_above' | 'crosses\_below';

value: number | number[];

timeframe: string;

}

interface ScanResult {

ticker: string;

companyName: string;

currentPrice: number;

volume: number;

marketCap: number;

sector: string;

exchange: string;

score: number;

signals: SignalMatch[];

indicators: IndicatorValues;

lastUpdated: Date;

}

interface SignalMatch {

type: string;

description: string;

strength: 'strong' | 'moderate' | 'weak';

confidence: number;

}

class MarketScanningEngine {

private scanQueue: Queue<ScanTask>;

private isScanning: boolean = false;

private stockUniverse: StockInfo[] = [];

private scanResults: Map<string, ScanResult> = new Map();

private lastScanTime: Date = new Date(0);

constructor(

private marketDataProcessor: MarketDataProcessor,

private signalProcessor: SignalProcessor

) {

this.scanQueue = new Queue<ScanTask>();

this.initializeStockUniverse();

}

// Initialize stock universe (6000+ stocks)

private async initializeStockUniverse(): Promise<void> {

try {

// Load from database or external source

const { data, error } = await supabaseClient

.from('stock\_universe')

.select('\*')

.eq('is\_active', true)

.gte('avg\_volume', 500000) // Minimum 500k average volume

.order('market\_cap', { ascending: false });

if (error) throw error;

this.stockUniverse = data.map(row => ({

ticker: row.ticker,

companyName: row.company\_name,

sector: row.sector,

marketCap: row.market\_cap,

avgVolume: row.avg\_volume,

exchange: row.exchange,

isActive: row.is\_active

}));

console.log(`Loaded ${this.stockUniverse.length} stocks for scanning`);

} catch (error) {

console.error('Failed to initialize stock universe:', error);

// Fallback to hardcoded list of major stocks

this.stockUniverse = this.getDefaultStockList();

}

}

// Start continuous market scanning

async startContinuousScanning(): Promise<void> {

if (this.isScanning) return;

this.isScanning = true;

console.log('Starting continuous market scanning...');

// Scan every 5 minutes during market hours

const scanInterval = 5 \* 60 \* 1000; // 5 minutes

while (this.isScanning) {

if (this.isMarketOpen()) {

await this.performFullMarketScan();

}

await this.delay(scanInterval);

}

}

// Perform full market scan

async performFullMarketScan(): Promise<ScanResult[]> {

const startTime = Date.now();

console.log(`Starting full market scan of ${this.stockUniverse.length} stocks...`);

const results: ScanResult[] = [];

const batchSize = 100; // Process 100 stocks at a time

const batches = this.chunkArray(this.stockUniverse, batchSize);

for (const batch of batches) {

const batchResults = await this.scanBatch(batch);

results.push(...batchResults);

// Rate limiting - pause between batches

await this.delay(1000);

}

// Filter results by minimum score threshold

const filteredResults = results.filter(r => r.score >= 60);

// Sort by score (highest first)

filteredResults.sort((a, b) => b.score - a.score);

// Store results

await this.storeScanResults(filteredResults);

const scanTime = Date.now() - startTime;

console.log(`Scan completed: ${filteredResults.length} signals found in ${scanTime}ms`);

this.lastScanTime = new Date();

return filteredResults;

}

// Scan a batch of stocks

private async scanBatch(stocks: StockInfo[]): Promise<ScanResult[]> {

const promises = stocks.map(stock => this.scanStock(stock));

const results = await Promise.allSettled(promises);

return results

.filter(result => result.status === 'fulfilled' && result.value !== null)

.map(result => (result as PromiseFulfilledResult<ScanResult>).value);

}

// Scan individual stock

private async scanStock(stock: StockInfo): Promise<ScanResult | null> {

try {

// Get current market data

const snapshot = await this.marketDataProcessor.polygonClient.getSnapshot(stock.ticker);

if (!snapshot) return null;

// Skip if price or volume filters not met

if (snapshot.value < 1 || snapshot.min.v < 100000) return null;

// Get multi-timeframe analysis

const multiTimeframeAnalyzer = new MultiTimeframeAnalyzer();

const analysis = await multiTimeframeAnalyzer.analyzeAllTimeframes(

stock.ticker,

this.marketDataProcessor

);

// Generate signal

const signal = await this.signalProcessor.processSignal(stock.ticker, analysis);

// Create scan result

const scanResult: ScanResult = {

ticker: stock.ticker,

companyName: stock.companyName,

currentPrice: snapshot.value,

volume: snapshot.min.v,

marketCap: stock.marketCap,

sector: stock.sector,

exchange: stock.exchange,

score: signal.finalScore,

signals: this.extractSignalMatches(signal),

indicators: this.extractIndicatorValues(analysis),

lastUpdated: new Date()

};

return scanResult;

} catch (error) {

console.error(`Error scanning ${stock.ticker}:`, error);

return null;

}

}

// Custom scan with specific criteria

async performCustomScan(criteria: ScanCriteria): Promise<ScanResult[]> {

console.log('Performing custom scan with criteria:', criteria);

// Filter stock universe based on criteria

const filteredStocks = this.stockUniverse.filter(stock =>

this.matchesCriteria(stock, criteria)

);

console.log(`Scanning ${filteredStocks.length} stocks matching criteria...`);

const results: ScanResult[] = [];

const batchSize = 50;

const batches = this.chunkArray(filteredStocks, batchSize);

for (const batch of batches) {

const batchResults = await this.scanBatch(batch);

// Apply indicator-specific filters

const filteredBatch = batchResults.filter(result =>

this.matchesIndicatorCriteria(result, criteria.indicators)

);

results.push(...filteredBatch);

await this.delay(500); // Rate limiting

}

// Filter by minimum score

const finalResults = results.filter(r => r.score >= criteria.minScore);

finalResults.sort((a, b) => b.score - a.score);

return finalResults;

}

// Real-time alert scanning for high-score signals

async scanForAlerts(): Promise<ScanResult[]> {

const alertCriteria: ScanCriteria = {

minPrice: 5,

maxPrice: 1000,

minVolume: 1000000, // 1M+ volume

minMarketCap: 100000000, // $100M+ market cap

maxMarketCap: 100000000000, // $100B max

sectors: [], // All sectors

exchanges: ['NYSE', 'NASDAQ'],

minScore: 80, // Strong signals only

indicators: []

};

const results = await this.performCustomScan(alertCriteria);

// Trigger alerts for new strong signals

for (const result of results) {

if (!this.wasAlertSent(result.ticker, result.score)) {

await this.triggerAlert(result);

this.markAlertSent(result.ticker, result.score);

}

}

return results;

}

private matchesCriteria(stock: StockInfo, criteria: ScanCriteria): boolean {

return (

stock.avgVolume >= criteria.minVolume &&

stock.marketCap >= criteria.minMarketCap &&

stock.marketCap <= criteria.maxMarketCap &&

(criteria.sectors.length === 0 || criteria.sectors.includes(stock.sector)) &&

(criteria.exchanges.length === 0 || criteria.exchanges.includes(stock.exchange))

);

}

private matchesIndicatorCriteria(result: ScanResult, criteriaList: IndicatorCriteria[]): boolean {

return criteriaList.every(criteria => {

const indicatorValue = this.getIndicatorValue(result.indicators, criteria.type, criteria.timeframe);

return this.evaluateCriteria(indicatorValue, criteria);

});

}

private getIndicatorValue(indicators: IndicatorValues, type: string, timeframe: string): number {

// Extract indicator value based on type and timeframe

const timeframeData = indicators[timeframe];

if (!timeframeData) return 0;

switch (type) {

case 'rsi':

return timeframeData.rsi || 0;

case 'macd':

return timeframeData.macd?.macd || 0;

case 'volume':

return timeframeData.volume?.volumeRatio || 0;

default:

return 0;

}

}

private evaluateCriteria(value: number, criteria: IndicatorCriteria): boolean {

switch (criteria.operator) {

case 'greater\_than':

return value > (criteria.value as number);

case 'less\_than':

return value < (criteria.value as number);

case 'between':

const range = criteria.value as number[];

return value >= range[0] && value <= range[1];

default:

return false;

}

}

}

// ===================================================================

// \*\*7. 🛡️ RISK MANAGEMENT SYSTEMS\*\*

// ===================================================================

interface RiskMetrics {

portfolioRisk: number;

maxRiskPerTrade: number;

totalExposure: number;

sectorExposure: Record<string, number>;

correlationRisk: number;

volatilityAdjustedRisk: number;

marginUtilization: number;

riskAdjustedReturn: number;

}

interface RiskLimits {

maxPositionSize: number; // 20% of portfolio

maxSectorExposure: number; // 30% per sector

maxTotalRisk: number; // 10% portfolio at risk

maxDrawdown: number; // 15% maximum drawdown

maxCorrelation: number; // 0.7 maximum correlation between positions

minLiquidity: number; // Minimum daily volume

}

class RiskManager {

private riskLimits: RiskLimits = {

maxPositionSize: 0.20, // 20%

maxSectorExposure: 0.30, // 30%

maxTotalRisk: 0.10, // 10%

maxDrawdown: 0.15, // 15%

maxCorrelation: 0.70, // 70%

minLiquidity: 500000 // 500k shares daily volume

};

constructor(private marketDataProcessor: MarketDataProcessor) {}

// Validate trade against risk limits

async validateTrade(orderRequest: TradeOrderRequest, portfolio: Portfolio): Promise<ValidationResult> {

const errors: string[] = [];

try {

// Position size check

const positionSizeCheck = this.checkPositionSize(orderRequest, portfolio);

if (!positionSizeCheck.isValid) {

errors.push(...positionSizeCheck.errors);

}

// Sector exposure check

const sectorCheck = await this.checkSectorExposure(orderRequest, portfolio);

if (!sectorCheck.isValid) {

errors.push(...sectorCheck.errors);

}

// Risk percentage check

const riskCheck = this.checkRiskPercentage(orderRequest, portfolio);

if (!riskCheck.isValid) {

errors.push(...riskCheck.errors);

}

// Correlation check

const correlationCheck = await this.checkCorrelation(orderRequest, portfolio);

if (!correlationCheck.isValid) {

errors.push(...correlationCheck.errors);

}

// Liquidity check

const liquidityCheck = await this.checkLiquidity(orderRequest.ticker);

if (!liquidityCheck.isValid) {

errors.push(...liquidityCheck.errors);

}

} catch (error) {

errors.push(`Risk validation error: ${error.message}`);

}

return {

isValid: errors.length === 0,

errors

};

}

// Calculate optimal position size based on 2% risk rule

calculateOptimalPositionSize(

accountBalance: number,

entryPrice: number,

stopLossPrice: number,

riskPercent: number = 2

): PositionSizeResult {

const riskAmount = (accountBalance \* riskPercent) / 100;

const riskPerShare = Math.abs(entryPrice - stopLossPrice);

if (riskPerShare === 0) {

return {

shares: 0,

positionValue: 0,

riskAmount: 0,

riskPercent: 0,

errors: ['Stop loss price cannot equal entry price']

};

}

const optimalShares = Math.floor(riskAmount / riskPerShare);

const positionValue = optimalShares \* entryPrice;

const actualRiskAmount = optimalShares \* riskPerShare;

const actualRiskPercent = (actualRiskAmount / accountBalance) \* 100;

// Position size limits

const maxPositionValue = accountBalance \* this.riskLimits.maxPositionSize;

const maxShares = Math.floor(maxPositionValue / entryPrice);

const finalShares = Math.min(optimalShares, maxShares);

const finalPositionValue = finalShares \* entryPrice;

const finalRiskAmount = finalShares \* riskPerShare;

const finalRiskPercent = (finalRiskAmount / accountBalance) \* 100;

return {

shares: finalShares,

positionValue: finalPositionValue,

riskAmount: finalRiskAmount,

riskPercent: finalRiskPercent,

errors: []

};

}

// Calculate stop loss and take profit levels

calculateRiskRewardLevels(

entryPrice: number,

direction: 'long' | 'short',

atr: number, // Average True Range for volatility-based stops

riskRewardRatio: number = 2

): RiskRewardLevels {

let stopLoss: number;

let takeProfit: number;

if (direction === 'long') {

// Long position

stopLoss = entryPrice - (2 \* atr); // 2 ATR stop loss

const riskAmount = entryPrice - stopLoss;

takeProfit = entryPrice + (riskAmount \* riskRewardRatio);

} else {

// Short position

stopLoss = entryPrice + (2 \* atr);

const riskAmount = stopLoss - entryPrice;

takeProfit = entryPrice - (riskAmount \* riskRewardRatio);

}

return {

entryPrice,

stopLoss: Math.round(stopLoss \* 100) / 100,

takeProfit: Math.round(takeProfit \* 100) / 100,

riskAmount: Math.abs(entryPrice - stopLoss),

rewardAmount: Math.abs(takeProfit - entryPrice),

riskRewardRatio

};

}

// Monitor portfolio risk in real-time

async calculatePortfolioRisk(portfolio: Portfolio): Promise<RiskMetrics> {

const totalPortfolioValue = portfolio.equity;

let totalRisk = 0;

const sectorExposure: Record<string, number> = {};

// Calculate position-level risks

for (const position of portfolio.positions) {

const positionRisk = await this.calculatePositionRisk(position, totalPortfolioValue);

totalRisk += positionRisk;

// Sector exposure

const sector = await this.getStockSector(position.ticker);

sectorExposure[sector] = (sectorExposure[sector] || 0) + position.weight;

}

// Calculate correlation risk

const correlationRisk = await this.calculateCorrelationRisk(portfolio.positions);

return {

portfolioRisk: totalRisk,

maxRiskPerTrade: this.riskLimits.maxTotalRisk \* 100,

totalExposure: portfolio.positions.reduce((sum, p) => sum + p.weight, 0),

sectorExposure,

correlationRisk,

volatilityAdjustedRisk: totalRisk \* 1.2, // Adjust for volatility

marginUtilization: (portfolio.equity / portfolio.balance) \* 100,

riskAdjustedReturn: portfolio.totalPnLPercent / Math.max(totalRisk, 1)

};

}

// Position-specific risk calculation

private async calculatePositionRisk(position: Position, portfolioValue: number): Promise<number> {

const stopLossDistance = position.stopLoss ?

Math.abs(position.currentPrice - position.stopLoss) :

position.currentPrice \* 0.05; // 5% default if no stop loss

const riskAmount = (stopLossDistance / position.currentPrice) \* position.marketValue;

return (riskAmount / portfolioValue) \* 100;

}

// Check if position size exceeds limits

private checkPositionSize(orderRequest: TradeOrderRequest, portfolio: Portfolio): ValidationResult {

const positionValue = orderRequest.shares \* orderRequest.entryPrice;

const positionWeight = positionValue / portfolio.equity;

if (positionWeight > this.riskLimits.maxPositionSize) {

return {

isValid: false,

errors: [`Position size ${(positionWeight \* 100).toFixed(1)}% exceeds maximum ${(this.riskLimits.maxPositionSize \* 100)}%`]

};

}

return { isValid: true, errors: [] };

}

// Check sector exposure limits

private async checkSectorExposure(orderRequest: TradeOrderRequest, portfolio: Portfolio): Promise<ValidationResult> {

const sector = await this.getStockSector(orderRequest.ticker);

const currentSectorExposure = portfolio.positions

.filter(p => this.getStockSector(p.ticker) === sector)

.reduce((sum, p) => sum + p.weight, 0);

const newPositionWeight = (orderRequest.shares \* orderRequest.entryPrice) / portfolio.equity;

const totalSectorExposure = currentSectorExposure + newPositionWeight;

if (totalSectorExposure > this.riskLimits.maxSectorExposure \* 100) {

return {

isValid: false,

errors: [`Sector exposure ${totalSectorExposure.toFixed(1)}% exceeds maximum ${(this.riskLimits.maxSectorExposure \* 100)}%`]

};

}

return { isValid: true, errors: [] };

}

// Automated risk monitoring and alerts

async monitorRiskLimits(portfolio: Portfolio): Promise<RiskAlert[]> {

const alerts: RiskAlert[] = [];

const riskMetrics = await this.calculatePortfolioRisk(portfolio);

// Portfolio risk alert

if (riskMetrics.portfolioRisk > this.riskLimits.maxTotalRisk \* 100) {

alerts.push({

type: 'portfolio\_risk',

severity: 'high',

message: `Portfolio risk ${riskMetrics.portfolioRisk.toFixed(1)}% exceeds limit ${(this.riskLimits.maxTotalRisk \* 100)}%`,

recommendation: 'Reduce position sizes or close some positions'

});

}

// Sector concentration alerts

Object.entries(riskMetrics.sectorExposure).forEach(([sector, exposure]) => {

if (exposure > this.riskLimits.maxSectorExposure \* 100) {

alerts.push({

type: 'sector\_concentration',

severity: 'medium',

message: `${sector} sector exposure ${exposure.toFixed(1)}% exceeds limit ${(this.riskLimits.maxSectorExposure \* 100)}%`,

recommendation: `Diversify away from ${sector} sector`

});

}

});

// Correlation risk alert

if (riskMetrics.correlationRisk > this.riskLimits.maxCorrelation) {

alerts.push({

type: 'correlation\_risk',

severity: 'medium',

message: `Portfolio correlation ${riskMetrics.correlationRisk.toFixed(2)} is too high`,

recommendation: 'Add uncorrelated positions to diversify'

});

}

return alerts;

}

}

// ===================================================================

// \*\*8. 💾 DATA STORAGE OPTIMIZATION\*\*

// ===================================================================

// Time-series data partitioning strategy

class TimeSeriesDataManager {

private partitionStrategies = {

realtime\_quotes: 'daily', // Partition by day

market\_data: 'monthly', // Partition by month

trading\_signals: 'weekly', // Partition by week

user\_trades: 'yearly' // Partition by year

};

// Create optimized table structures

async createOptimizedTables(): Promise<void> {

const tableDefinitions = [

// Market data with partitioning

`

CREATE TABLE IF NOT EXISTS market\_data\_partitioned (

id UUID DEFAULT gen\_random\_uuid(),

ticker VARCHAR(10) NOT NULL,

timestamp TIMESTAMPTZ NOT NULL,

timeframe VARCHAR(5) NOT NULL,

open DECIMAL(12,4) NOT NULL,

high DECIMAL(12,4) NOT NULL,

low DECIMAL(12,4) NOT NULL,

close DECIMAL(12,4) NOT NULL,

volume BIGINT NOT NULL,

vwap DECIMAL(12,4),

indicators JSONB,

quality\_score INTEGER DEFAULT 100,

created\_at TIMESTAMPTZ DEFAULT NOW(),

PRIMARY KEY (ticker, timestamp, timeframe)

) PARTITION BY RANGE (timestamp);

`,

// Trading signals with compression

`

CREATE TABLE IF NOT EXISTS trading\_signals\_optimized (

id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),

ticker VARCHAR(10) NOT NULL,

signal\_type VARCHAR(20) NOT NULL,

final\_score DECIMAL(5,2) NOT NULL,

strength VARCHAR(10) NOT NULL,

timeframe\_scores JSONB COMPRESSED,

risk\_reward JSONB,

metadata JSONB,

triggered\_at TIMESTAMPTZ NOT NULL,

expires\_at TIMESTAMPTZ,

created\_at TIMESTAMPTZ DEFAULT NOW()

);

`,

// User trades with archival strategy

`

CREATE TABLE IF NOT EXISTS user\_trades\_archived (

id UUID PRIMARY KEY,

user\_id UUID NOT NULL,

ticker VARCHAR(10) NOT NULL,

trade\_type VARCHAR(10) NOT NULL,

shares INTEGER NOT NULL,

entry\_price DECIMAL(12,4) NOT NULL,

exit\_price DECIMAL(12,4),

pnl DECIMAL(15,4),

status VARCHAR(10) NOT NULL,

entry\_time TIMESTAMPTZ NOT NULL,

exit\_time TIMESTAMPTZ,

archived\_at TIMESTAMPTZ DEFAULT NOW()

) PARTITION BY RANGE (entry\_time);

`

];

for (const sql of tableDefinitions) {

try {

await supabaseClient.rpc('execute\_sql', { sql\_query: sql });

console.log('Table created successfully');

} catch (error) {

console.error('Failed to create table:', error);

}

}

}

// Create monthly partitions automatically

async createMonthlyPartitions(tableName: string, startDate: Date, months: number = 12): Promise<void> {

for (let i = 0; i < months; i++) {

const partitionDate = new Date(startDate);

partitionDate.setMonth(partitionDate.getMonth() + i);

const partitionName = `${tableName}\_${partitionDate.getFullYear()}\_${String(partitionDate.getMonth() + 1).padStart(2, '0')}`;

const startOfMonth = new Date(partitionDate.getFullYear(), partitionDate.getMonth(), 1);

const endOfMonth = new Date(partitionDate.getFullYear(), partitionDate.getMonth() + 1, 1);

const sql = `

CREATE TABLE IF NOT EXISTS ${partitionName}

PARTITION OF ${tableName}

FOR VALUES FROM ('${startOfMonth.toISOString()}') TO ('${endOfMonth.toISOString()}');

`;

try {

await supabaseClient.rpc('execute\_sql', { sql\_query: sql });

console.log(`Created partition: ${partitionName}`);

} catch (error) {

console.error(`Failed to create partition ${partitionName}:`, error);

}

}

}

// Optimize indexes for financial queries

async createOptimizedIndexes(): Promise<void> {

const indexes = [

// Market data indexes

`CREATE INDEX CONCURRENTLY IF NOT EXISTS idx\_market\_data\_ticker\_time

ON market\_data\_partitioned (ticker, timestamp DESC);`,

`CREATE INDEX CONCURRENTLY IF NOT EXISTS idx\_market\_data\_timeframe\_score

ON market\_data\_partitioned (timeframe, quality\_score) WHERE quality\_score >= 80;`,

// Trading signals indexes

`CREATE INDEX CONCURRENTLY IF NOT EXISTS idx\_signals\_score\_time

ON trading\_signals\_optimized (final\_score DESC, triggered\_at DESC)

WHERE final\_score >= 70;`,

`CREATE INDEX CONCURRENTLY IF NOT EXISTS idx\_signals\_ticker\_active

ON trading\_signals\_optimized (ticker, triggered\_at)

WHERE expires\_at > NOW();`,

// User trades indexes

`CREATE INDEX CONCURRENTLY IF NOT EXISTS idx\_trades\_user\_time

ON user\_trades\_archived (user\_id, entry\_time DESC);`,

`CREATE INDEX CONCURRENTLY IF NOT EXISTS idx\_trades\_performance

ON user\_trades\_archived (pnl, entry\_time) WHERE status = 'closed';`,

// Composite indexes for common queries

`CREATE INDEX CONCURRENTLY IF NOT EXISTS idx\_market\_data\_scan

ON market\_data\_partitioned (timestamp, volume, close)

WHERE timeframe = '1D' AND volume > 500000;`

];

for (const indexSql of indexes) {

try {

await supabaseClient.rpc('execute\_sql', { sql\_query: indexSql });

console.log('Index created successfully');

} catch (error) {

console.error('Failed to create index:', error);

}

}

}

// Data compression and archival

async compressOldData(tableName: string, olderThanDays: number = 90): Promise<void> {

const cutoffDate = new Date();

cutoffDate.setDate(cutoffDate.getDate() - olderThanDays);

// Move old data to compressed archive table

const archiveTableName = `${tableName}\_archive`;

const sql = `

WITH moved\_data AS (

DELETE FROM ${tableName}

WHERE created\_at < '${cutoffDate.toISOString()}'

RETURNING \*

)

INSERT INTO ${archiveTableName}

SELECT \* FROM moved\_data;

`;

try {

const result = await supabaseClient.rpc('execute\_sql', { sql\_query: sql });

console.log(`Archived data older than ${olderThanDays} days from ${tableName}`);

return result;

} catch (error) {

console.error('Failed to archive data:', error);

throw error;

}

}

// Query optimization for real-time data

async getLatestMarketData(

ticker: string,

timeframe: string,

limit: number = 100

): Promise<any[]> {

// Optimized query using indexes

const { data, error } = await supabaseClient

.from('market\_data\_partitioned')

.select('\*')

.eq('ticker', ticker)

.eq('timeframe', timeframe)

.gte('quality\_score', 80) // Only high-quality data

.order('timestamp', { ascending: false })

.limit(limit);

if (error) throw error;

return data || [];

}

// Bulk insert optimization

async bulkInsertMarketData(data: any[], batchSize: number = 1000): Promise<void> {

const batches = this.chunkArray(data, batchSize);

for (const batch of batches) {

try {

const { error } = await supabaseClient

.from('market\_data\_partitioned')

.insert(batch);

if (error) throw error;

} catch (error) {

console.error('Bulk insert failed for batch:', error);

// Retry with smaller batch size

if (batchSize > 100) {

await this.bulkInsertMarketData(batch, Math.floor(batchSize / 2));

}

}

}

}

// Database maintenance and optimization

async performMaintenance(): Promise<void> {

const maintenanceTasks = [

// Update table statistics

'ANALYZE market\_data\_partitioned;',

'ANALYZE trading\_signals\_optimized;',

'ANALYZE user\_trades\_archived;',

// Vacuum old partitions

'VACUUM (ANALYZE) market\_data\_partitioned;',

// Reindex if fragmentation is high

'REINDEX INDEX CONCURRENTLY idx\_market\_data\_ticker\_time;'

];

for (const task of maintenanceTasks) {

try {

await supabaseClient.rpc('execute\_sql', { sql\_query: task });

console.log(`Maintenance task completed: ${task}`);

} catch (error) {

console.error(`Maintenance task failed: ${task}`, error);

}

}

}

// Data retention policy enforcement

async enforceRetentionPolicy(): Promise<void> {

const retentionPolicies = [

{ table: 'realtime\_quotes', days: 7 },

{ table: 'market\_data\_partitioned', days: 365 },

{ table: 'trading\_signals\_optimized', days: 180 },

{ table: 'user\_activity\_logs', days: 90 }

];

for (const policy of retentionPolicies) {

await this.deleteOldData(policy.table, policy.days);

}

}

private async deleteOldData(tableName: string, retentionDays: number): Promise<void> {

const cutoffDate = new Date();

cutoffDate.setDate(cutoffDate.getDate() - retentionDays);

const sql = `

DELETE FROM ${tableName}

WHERE created\_at < '${cutoffDate.toISOString()}';

`;

try {

await supabaseClient.rpc('execute\_sql', { sql\_query: sql });

console.log(`Deleted data older than ${retentionDays} days from ${tableName}`);

} catch (error) {

console.error(`Failed to delete old data from ${tableName}:`, error);

}

}

private chunkArray<T>(array: T[], chunkSize: number): T[][] {

const chunks: T[][] = [];

for (let i = 0; i < array.length; i += chunkSize) {

chunks.push(array.slice(i, i + chunkSize));

}

return chunks;

}

}

// ===================================================================

// EXPORT INTERFACES AND CLASSES

// ===================================================================

export {

PaperTradingEngine,

PerformanceCalculator,

MarketScanningEngine,

RiskManager,

TimeSeriesDataManager,

type Portfolio,

type PaperTrade,

type PerformanceMetrics,

type ScanResult,

type RiskMetrics,

type RiskLimits

};