Technical Indicators Calculator - Implementation Instructions

Overview

Create a Python function `calculate_technicals.py` that reads stock price data from a PostgreSQL table `daily_charts` and calculates technical indicators for each ticker, updating the table with today's indicator values.

Database Schema Assumptions

The `daily_charts` table should contain these core columns:

sql

```sql
CREATE TABLE daily_charts (
    id SERIAL PRIMARY KEY,
    ticker VARCHAR(20) NOT NULL,
    date DATE NOT NULL,
    open_price DECIMAL(10,4),
    high_price DECIMAL(10,4),
    low_price DECIMAL(10,4),
    close_price DECIMAL(10,4),
    volume BIGINT,
    adjusted_close DECIMAL(10,4),

    -- Technical Indicators (to be calculated/updated)
    rsi_14 DECIMAL(8,4),
    cci_20 DECIMAL(8,4),
    ema_20 DECIMAL(10,4),
    ema_50 DECIMAL(10,4),
    ema_100 DECIMAL(10,4),
    ema_200 DECIMAL(10,4),
    bb_upper DECIMAL(10,4),
    bb_middle DECIMAL(10,4),
    bb_lower DECIMAL(10,4),
    macd_line DECIMAL(8,4),
    macd_signal DECIMAL(8,4),
    macd_histogram DECIMAL(8,4),
    atr_14 DECIMAL(8,4),
    vwap DECIMAL(10,4),
    obv BIGINT,
    vpt DECIMAL(15,4),
    stoch_k DECIMAL(8,4),
    stoch_d DECIMAL(8,4),

    -- Support & Resistance Levels
    pivot_point DECIMAL(10,4),
    resistance_1 DECIMAL(10,4),
    resistance_2 DECIMAL(10,4),
    resistance_3 DECIMAL(10,4),
    support_1 DECIMAL(10,4),
    support_2 DECIMAL(10,4),
    support_3 DECIMAL(10,4),

    -- Swing Levels (Local Extrema)
    swing_high_5d DECIMAL(10,4),
    swing_low_5d DECIMAL(10,4),
    swing_high_10d DECIMAL(10,4),
    swing_low_10d DECIMAL(10,4),
    swing_high_20d DECIMAL(10,4),
```

```
    swing_low_20d DECIMAL(10,4),

    -- Key Levels
    week_high DECIMAL(10,4),
    week_low DECIMAL(10,4),
    month_high DECIMAL(10,4),
    month_low DECIMAL(10,4),
    nearest_support DECIMAL(10,4),
    nearest_resistance DECIMAL(10,4),
    support_strength INTEGER, -- 1-10 scale
    resistance_strength INTEGER, -- 1-10 scale

    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(ticker, date)
);
```

Required Python Libraries

Include these imports in your function:

python

```python
import pandas as pd
import numpy as np
import psycopg2
from sqlalchemy import create_engine, text
import talib
from datetime import datetime, timedelta
import logging
from typing import Dict, List, Optional
```

Core Function Structure

Create a main function with this signature:

python

```python
def calculate_technicals(
    connection_string: str,
    target_date: Optional[str] = None,
    tickers: Optional[List[str]] = None,
    lookback_days: int = 250
) -> Dict[str, int]:
    """
    Calculate technical indicators for stocks in daily_charts table.

    Args:
        connection_string: PostgreSQL connection string
        target_date: Date to calculate indicators for (default: today)
        tickers: List of specific tickers to process (default: all)
        lookback_days: Days of historical data to use (minimum 250 for 200-day indicators)

    Returns:
        Dict with processing results: {'processed': count, 'errors': count}
    """
```

Technical Indicator Calculations

1. RSI (Relative Strength Index)

python

```python
def calculate_rsi(prices: pd.Series, period: int = 14) -> pd.Series:
    """
    Calculate RSI using the standard formula.
    RSI = 100 - (100 / (1 + RS))
    where RS = Average Gain / Average Loss
    """
    # Use talib for accuracy: talib.RSI(prices.values, timeperiod=period)
    # OR implement manually:
    delta = prices.diff()
    gain = (delta.where(delta > 0, 0)).rolling(window=period).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=period).mean()
    rs = gain / loss
    rsi = 100 - (100 / (1 + rs))
    return rsi
```

2. CCI (Commodity Channel Index)

python

```python
def calculate_cci(high: pd.Series, low: pd.Series, close: pd.Series, period: int = 20) -> pd.Se
    """
    CCI = (Typical Price - SMA of Typical Price) / (0.015 × Mean Deviation)
    Typical Price = (High + Low + Close) / 3
    """
    typical_price = (high + low + close) / 3
    sma_tp = typical_price.rolling(window=period).mean()
    mean_deviation = typical_price.rolling(window=period).apply(
        lambda x: np.mean(np.abs(x - x.mean()))
    )
    cci = (typical_price - sma_tp) / (0.015 * mean_deviation)
    return cci
```

3. Exponential Moving Averages

python

```python
def calculate_ema(prices: pd.Series, period: int) -> pd.Series:
    """
    EMA calculation with proper initialization.
    """
    return prices.ewm(span=period, adjust=False).mean()
```

4. Bollinger Bands

python

```python
def calculate_bollinger_bands(prices: pd.Series, period: int = 20, std_dev: float = 2) -> Dict[
    """
    Calculate Bollinger Bands: Middle (SMA), Upper (SMA + 2*STD), Lower (SMA - 2*STD)
    """
    middle = prices.rolling(window=period).mean()
    std = prices.rolling(window=period).std()
    upper = middle + (std * std_dev)
    lower = middle - (std * std_dev)

    return {
        'bb_upper': upper,
        'bb_middle': middle,
        'bb_lower': lower
    }
```

5. MACD

python

```python
def calculate_macd(prices: pd.Series, fast: int = 12, slow: int = 26, signal: int = 9) -> Dict[
    """
    MACD Line = EMA(12) - EMA(26)
    Signal Line = EMA(9) of MACD Line
    Histogram = MACD Line - Signal Line
    """
    ema_fast = calculate_ema(prices, fast)
    ema_slow = calculate_ema(prices, slow)
    macd_line = ema_fast - ema_slow
    signal_line = calculate_ema(macd_line, signal)
    histogram = macd_line - signal_line

    return {
        'macd_line': macd_line,
        'macd_signal': signal_line,
        'macd_histogram': histogram
    }
```

6. ATR (Average True Range)

python

```python
def calculate_atr(high: pd.Series, low: pd.Series, close: pd.Series, period: int = 14) -> pd.Se
    """
    True Range = max(high-low, abs(high-prev_close), abs(low-prev_close))
    ATR = EMA of True Range
    """
    prev_close = close.shift(1)
    tr1 = high - low
    tr2 = np.abs(high - prev_close)
    tr3 = np.abs(low - prev_close)
    true_range = np.maximum(tr1, np.maximum(tr2, tr3))
    atr = true_range.ewm(span=period, adjust=False).mean()
    return atr
```

7. VWAP (Volume Weighted Average Price)

python

```python
def calculate_vwap(high: pd.Series, low: pd.Series, close: pd.Series, volume: pd.Series) -> pd.
    """
    VWAP = Sum(Typical Price * Volume) / Sum(Volume)
    Calculated on daily basis (reset each day)
    """
    typical_price = (high + low + close) / 3
    # For daily VWAP, this would be a single value per day
    # For intraday, you'd need to reset the cumulative calculation each day
    vwap = (typical_price * volume).cumsum() / volume.cumsum()
    return vwap
```

8. Volume Indicators

python

```python
def calculate_obv(close: pd.Series, volume: pd.Series) -> pd.Series:
    """
    On-Balance Volume: Add volume on up days, subtract on down days
    """
    price_change = close.diff()
    obv_change = np.where(price_change > 0, volume,
                 np.where(price_change < 0, -volume, 0))
    obv = obv_change.cumsum()
    return obv

def calculate_vpt(close: pd.Series, volume: pd.Series) -> pd.Series:
    """
    Volume Price Trend: OBV weighted by price change percentage
    """
    price_change_pct = close.pct_change()
    vpt_change = volume * price_change_pct
    vpt = vpt_change.cumsum()
    return vpt
```

9. Stochastic Oscillator

python

```python
def calculate_stochastic(high: pd.Series, low: pd.Series, close: pd.Series,
                         k_period: int = 14, d_period: int = 3) -> Dict[str, pd.Series]:
    """
    %K = ((Close - LowestLow) / (HighestHigh - LowestLow)) * 100
    %D = SMA of %K
    """
    lowest_low = low.rolling(window=k_period).min()
    highest_high = high.rolling(window=k_period).max()

    k_percent = ((close - lowest_low) / (highest_high - lowest_low)) * 100
    d_percent = k_percent.rolling(window=d_period).mean()

    return {
        'stoch_k': k_percent,
        'stoch_d': d_percent
    }
```

Support & Resistance Level Calculations

1. Pivot Points (Traditional Method)

python

```python
def calculate_pivot_points(high: pd.Series, low: pd.Series, close: pd.Series) -> Dict[str, pd.S
    """
    Calculate traditional pivot points for intraday and swing trading.
    Uses previous day's High, Low, Close for calculation.
    """
    # Shift by 1 to use previous day's data
    prev_high = high.shift(1)
    prev_low = low.shift(1)
    prev_close = close.shift(1)

    # Calculate pivot point
    pivot = (prev_high + prev_low + prev_close) / 3

    # Calculate resistance levels
    r1 = (2 * pivot) - prev_low
    r2 = pivot + (prev_high - prev_low)
    r3 = prev_high + 2 * (pivot - prev_low)

    # Calculate support levels
    s1 = (2 * pivot) - prev_high
    s2 = pivot - (prev_high - prev_low)
    s3 = prev_low - 2 * (prev_high - pivot)

    return {
        'pivot_point': pivot,
        'resistance_1': r1,
        'resistance_2': r2,
        'resistance_3': r3,
        'support_1': s1,
        'support_2': s2,
        'support_3': s3
    }

def calculate_fibonacci_pivots(high: pd.Series, low: pd.Series, close: pd.Series) -> Dict[str,
    """
    Calculate Fibonacci-based pivot points for more accurate levels.
    """
    prev_high = high.shift(1)
    prev_low = low.shift(1)
    prev_close = close.shift(1)

    pivot = (prev_high + prev_low + prev_close) / 3
    range_hl = prev_high - prev_low

    # Fibonacci ratios
    fib_382 = 0.382
```

```
    fib_618 = 0.618
    fib_1000 = 1.000

    # Fibonacci resistance levels
    r1 = pivot + (fib_382 * range_hl)
    r2 = pivot + (fib_618 * range_hl)
    r3 = pivot + (fib_1000 * range_hl)

    # Fibonacci support levels
    s1 = pivot - (fib_382 * range_hl)
    s2 = pivot - (fib_618 * range_hl)
    s3 = pivot - (fib_1000 * range_hl)

    return {
        'fib_pivot': pivot,
        'fib_r1': r1,
        'fib_r2': r2,
        'fib_r3': r3,
        'fib_s1': s1,
        'fib_s2': s2,
        'fib_s3': s3
    }
```

2. Swing Highs and Lows (Local Extrema)

python

```python
def identify_swing_levels(high: pd.Series, low: pd.Series,
                          lookback_periods: List[int] = [5, 10, 20]) -> Dict[str, pd.Series]:
    """
    Identify swing highs and lows over different time periods.
    Critical for swing traders to identify key support/resistance levels.
    """
    swing_levels = {}

    for period in lookback_periods:
        # Swing highs: highest high in the lookback period
        swing_high = high.rolling(window=period, center=True).max()

        # Swing lows: lowest low in the lookback period
        swing_low = low.rolling(window=period, center=True).min()

        # Only mark as swing point if it's actually the highest/lowest in the window
        swing_high = swing_high.where(high == swing_high)
        swing_low = swing_low.where(low == swing_low)

        swing_levels[f'swing_high_{period}d'] = swing_high
        swing_levels[f'swing_low_{period}d'] = swing_low

    return swing_levels

def calculate_swing_strength(high: pd.Series, low: pd.Series, close: pd.Series,
                             swing_highs: pd.Series, swing_lows: pd.Series,
                             volume: pd.Series) -> Dict[str, pd.Series]:
    """
    Calculate the strength of swing levels based on:
    1. Number of times level was tested
    2. Volume at the level
    3. Time since level was established
    4. Price reaction magnitude
    """

    def calculate_level_strength(levels: pd.Series, prices: pd.Series,
                                 volumes: pd.Series, tolerance: float = 0.02) -> pd.Series:
        """Calculate strength score for support/resistance levels."""
        strength_scores = pd.Series(index=levels.index, dtype=float)

        for i, level in levels.items():
            if pd.isna(level):
                continue

            # Count how many times price approached this level (within tolerance)
            approaches = prices[(prices >= level * (1 - tolerance)) &
```

```
                                        (prices <= level * (1 + tolerance))]

            # Base strength on number of approaches
            base_strength = min(len(approaches), 10)  # Cap at 10

            # Bonus for high volume at level
            if len(approaches) > 0:
                avg_volume_at_level = volumes[approaches.index].mean()
                avg_volume_overall = volumes.mean()
                volume_multiplier = min(avg_volume_at_level / avg_volume_overall, 2.0)
                base_strength *= volume_multiplier

            strength_scores[i] = min(base_strength, 10)  # Cap at 10

        return strength_scores

    resistance_strength = calculate_level_strength(swing_highs, close, volume)
    support_strength = calculate_level_strength(swing_lows, close, volume)

    return {
        'resistance_strength': resistance_strength,
        'support_strength': support_strength
    }
```

3. Key Time-Based Levels

```
python
```

```python
def calculate_key_levels(high: pd.Series, low: pd.Series, close: pd.Series,
                        df_with_dates: pd.DataFrame) -> Dict[str, pd.Series]:
    """
    Calculate important time-based support/resistance levels.
    """
    # Ensure we have a datetime index
    if not isinstance(df_with_dates.index, pd.DatetimeIndex):
        df_with_dates = df_with_dates.copy()
        df_with_dates.index = pd.to_datetime(df_with_dates.index)

    # Weekly levels (Monday to Friday)
    weekly_high = high.resample('W').max().reindex(df_with_dates.index, method='ffill')
    weekly_low = low.resample('W').min().reindex(df_with_dates.index, method='ffill')

    # Monthly levels
    monthly_high = high.resample('M').max().reindex(df_with_dates.index, method='ffill')
    monthly_low = low.resample('M').min().reindex(df_with_dates.index, method='ffill')

    # Previous day's high/low (important for day/swing traders)
    prev_day_high = high.shift(1)
    prev_day_low = low.shift(1)

    return {
        'week_high': weekly_high,
        'week_low': weekly_low,
        'month_high': monthly_high,
        'month_low': monthly_low,
        'prev_day_high': prev_day_high,
        'prev_day_low': prev_day_low
    }

def calculate_psychological_levels(close: pd.Series) -> Dict[str, pd.Series]:
    """
    Calculate psychological support/resistance levels (round numbers).
    These often act as significant levels due to human psychology.
    """
    current_price = close.iloc[-1] if len(close) > 0 else 0

    # Find nearest round numbers
    if current_price > 100:
        # For stocks > $100, use $10 increments
        round_factor = 10
    elif current_price > 50:
        # For stocks $50-$100, use $5 increments
        round_factor = 5
    elif current_price > 10:
```

```python
        # For stocks $10-$50, use $1 increments
        round_factor = 1
    else:
        # For stocks < $10, use $0.50 increments
        round_factor = 0.5

    # Calculate nearest psychological levels
    nearest_round_below = (int(current_price / round_factor)) * round_factor
    nearest_round_above = nearest_round_below + round_factor

    # Create series with these levels
    psych_support = pd.Series(nearest_round_below, index=close.index)
    psych_resistance = pd.Series(nearest_round_above, index=close.index)

    return {
        'psychological_support': psych_support,
        'psychological_resistance': psych_resistance
    }
```

4. Nearest Support/Resistance Calculator

python

```python
def find_nearest_levels(current_price: float, all_levels: Dict[str, pd.Series],
                        tolerance: float = 0.15) -> Dict[str, float]:
    """
    Find the nearest significant support and resistance levels.
    Used for setting stop-losses and targets for swing trades.
    """
    # Collect all potential levels
    resistance_levels = []
    support_levels = []

    for level_name, level_series in all_levels.items():
        if level_series is None or len(level_series) == 0:
            continue

        latest_level = level_series.iloc[-1]
        if pd.isna(latest_level):
            continue

        # Categorize as support or resistance based on current price
        price_diff_pct = (latest_level - current_price) / current_price

        if abs(price_diff_pct) <= tolerance:  # Within tolerance range
            if latest_level > current_price:
                resistance_levels.append(latest_level)
            elif latest_level < current_price:
                support_levels.append(latest_level)

    # Find nearest levels
    nearest_resistance = min(resistance_levels) if resistance_levels else None
    nearest_support = max(support_levels) if support_levels else None

    return {
        'nearest_resistance': nearest_resistance,
        'nearest_support': nearest_support,
        'resistance_distance_pct': ((nearest_resistance - current_price) / current_price * 100)
        'support_distance_pct': ((current_price - nearest_support) / current_price * 100) if ne
    }

def calculate_level_confluences(all_levels: Dict[str, pd.Series],
                                tolerance: float = 0.02) -> Dict[str, pd.Series]:
    """
    Identify confluence zones where multiple support/resistance levels cluster.
    These are typically stronger levels for swing trading.
    """
    # Implementation for finding confluence zones
    # This is advanced and would require clustering algorithm
```

```python
    # For now, return placeholder
    return {
        'confluence_resistance': pd.Series(dtype=float),
        'confluence_support': pd.Series(dtype=float)
    }
```

5. Volume-Based Support/Resistance

python

```python
def calculate_volume_profile_levels(high: pd.Series, low: pd.Series,
                                   close: pd.Series, volume: pd.Series,
                                   lookback_days: int = 20) -> Dict[str, pd.Series]:
    """
    Calculate support/resistance based on volume profile.
    Areas with high volume often act as strong support/resistance.
    """
    # Use rolling window to calculate volume at price levels
    volume_at_price = {}

    for i in range(len(close)):
        if i < lookback_days:
            continue

        # Get data for lookback period
        period_data = {
            'high': high.iloc[i-lookback_days:i+1],
            'low': low.iloc[i-lookback_days:i+1],
            'close': close.iloc[i-lookback_days:i+1],
            'volume': volume.iloc[i-lookback_days:i+1]
        }

        # Create price bins and sum volume
        price_range = period_data['high'].max() - period_data['low'].min()
        num_bins = min(20, int(price_range / (price_range * 0.01)))  # 1% increments

        if num_bins > 0:
            price_bins = np.linspace(period_data['low'].min(),
                                   period_data['high'].max(), num_bins)

            # This is a simplified version - full implementation would require
            # more sophisticated volume profile calculation

    # Return placeholder for now - full volume profile is complex
    return {
        'volume_resistance': pd.Series(dtype=float, index=close.index),
        'volume_support': pd.Series(dtype=float, index=close.index),
        'high_volume_price': close.rolling(window=lookback_days).median()
    }
```

Main Processing Logic

Structure your main function like this:

python

```python
def calculate_technicals(connection_string: str, target_date: Optional[str] = None,
                        tickers: Optional[List[str]] = None, lookback_days: int = 250) -> Dict[

    # 1. Setup logging and database connection
    logging.basicConfig(level=logging.INFO)
    logger = logging.getLogger(__name__)
    engine = create_engine(connection_string)

    # 2. Determine target date (default to today)
    if target_date is None:
        target_date = datetime.now().date()
    else:
        target_date = datetime.strptime(target_date, '%Y-%m-%d').date()

    # 3. Get list of tickers to process
    if tickers is None:
        tickers = get_active_tickers(engine, target_date)

    results = {'processed': 0, 'errors': 0}

    # 4. Process each ticker
    for ticker in tickers:
        try:
            # Get historical data (ensure minimum lookback for 200-day indicators)
            start_date = target_date - timedelta(days=lookback_days)
            df = get_ticker_data(engine, ticker, start_date, target_date)

            if len(df) < 50:  # Minimum data requirement
                logger.warning(f"Insufficient data for {ticker}: {len(df)} rows")
                continue

            # Calculate all indicators
            indicators = calculate_all_indicators(df)

            # Update database with today's values
            update_ticker_indicators(engine, ticker, target_date, indicators)

            results['processed'] += 1
            logger.info(f"Processed indicators for {ticker}")

        except Exception as e:
            logger.error(f"Error processing {ticker}: {str(e)}")
            results['errors'] += 1

    return results
```

Database Helper Functions

python

```python
def get_active_tickers(engine, target_date) -> List[str]:
    """Get list of tickers that have data for target date."""
    query = """
    SELECT DISTINCT ticker
    FROM daily_charts
    WHERE date = %s
    ORDER BY ticker
    """
    with engine.connect() as conn:
        result = conn.execute(text(query), (target_date,))
        return [row[0] for row in result]

def get_ticker_data(engine, ticker: str, start_date, end_date) -> pd.DataFrame:
    """Retrieve historical data for a ticker."""
    query = """
    SELECT date, open_price, high_price, low_price, close_price,
           volume, adjusted_close
    FROM daily_charts
    WHERE ticker = %s AND date BETWEEN %s AND %s
    ORDER BY date
    """
    with engine.connect() as conn:
        df = pd.read_sql_query(query, conn, params=(ticker, start_date, end_date))
        df['date'] = pd.to_datetime(df['date'])
        df.set_index('date', inplace=True)
        return df

def update_ticker_indicators(engine, ticker: str, target_date, indicators: Dict):
    """Update the daily_charts table with calculated indicators."""

    # Get the latest values (today's indicators)
    latest_indicators = {}
    for key, series in indicators.items():
        if len(series) > 0 and not pd.isna(series.iloc[-1]):
            latest_indicators[key] = float(series.iloc[-1])

    if not latest_indicators:
        return

    # Build UPDATE query
    set_clause = ", ".join([f"{key} = %s" for key in latest_indicators.keys()])
    values = list(latest_indicators.values()) + [ticker, target_date]

    query = f"""
    UPDATE daily_charts
    SET {set_clause}, updated_at = CURRENT_TIMESTAMP
```

```python
    WHERE ticker = %s AND date = %s
    """

    with engine.connect() as conn:
        conn.execute(text(query), values)
        conn.commit()
```

Master Indicator Calculator

python

```python
def calculate_all_indicators(df: pd.DataFrame) -> Dict[str, pd.Series]:
    """Calculate all technical indicators for a ticker's data."""

    indicators = {}

    # Ensure we have required columns
    required_cols = ['open_price', 'high_price', 'low_price', 'close_price', 'volume']
    for col in required_cols:
        if col not in df.columns:
            raise ValueError(f"Missing required column: {col}")

    # Extract price and volume data
    high = df['high_price']
    low = df['low_price']
    close = df['close_price']
    volume = df['volume']

    # Calculate trend indicators
    indicators['ema_20'] = calculate_ema(close, 20)
    indicators['ema_50'] = calculate_ema(close, 50)
    indicators['ema_100'] = calculate_ema(close, 100)
    indicators['ema_200'] = calculate_ema(close, 200)

    # Calculate momentum indicators
    indicators['rsi_14'] = calculate_rsi(close, 14)
    indicators['cci_20'] = calculate_cci(high, low, close, 20)

    # Calculate MACD
    macd_data = calculate_macd(close)
    indicators.update(macd_data)

    # Calculate volatility indicators
    bb_data = calculate_bollinger_bands(close)
    indicators.update(bb_data)
    indicators['atr_14'] = calculate_atr(high, low, close, 14)

    # Calculate volume indicators
    indicators['vwap'] = calculate_vwap(high, low, close, volume)
    indicators['obv'] = calculate_obv(close, volume)
    indicators['vpt'] = calculate_vpt(close, volume)

    # Calculate stochastic
    stoch_data = calculate_stochastic(high, low, close)
    indicators.update(stoch_data)

    # Calculate support and resistance levels
```

```python
    # 1. Traditional pivot points
    pivot_data = calculate_pivot_points(high, low, close)
    indicators.update(pivot_data)

    # 2. Swing levels (local extrema)
    swing_data = calculate_swing_levels(high, low, [5, 10, 20])
    indicators.update(swing_data)

    # 3. Key time-based levels
    key_levels = calculate_key_levels(high, low, close, df)
    indicators.update(key_levels)

    # 4. Psychological levels
    psych_levels = calculate_psychological_levels(close)
    indicators.update(psych_levels)

    # 5. Volume-based levels
    volume_levels = calculate_volume_profile_levels(high, low, close, volume)
    indicators.update(volume_levels)

    # 6. Calculate strength scores for swing levels
    if 'swing_high_20d' in indicators and 'swing_low_20d' in indicators:
        strength_data = calculate_swing_strength(
            high, low, close,
            indicators['swing_high_20d'],
            indicators['swing_low_20d'],
            volume
        )
        indicators.update(strength_data)

    # 7. Find nearest support/resistance levels
    current_price = close.iloc[-1] if len(close) > 0 else 0
    if current_price > 0:
        nearest_levels = find_nearest_levels(current_price, indicators)

        # Convert to series for database storage
        for key, value in nearest_levels.items():
            if value is not None:
                indicators[key] = pd.Series([value] * len(close), index=close.index)

    return indicators
```

Error Handling and Validation

python

```python
def validate_data(df: pd.DataFrame, ticker: str) -> bool:
    """Validate data quality before processing."""

    # Check for required columns
    required_cols = ['high_price', 'low_price', 'close_price', 'volume']
    missing_cols = [col for col in required_cols if col not in df.columns]
    if missing_cols:
        logger.error(f"{ticker}: Missing columns: {missing_cols}")
        return False

    # Check for sufficient data
    if len(df) < 200:
        logger.warning(f"{ticker}: Insufficient data ({len(df)} rows)")
        return False

    # Check for data quality issues
    if df[required_cols].isnull().any().any():
        logger.warning(f"{ticker}: Contains null values")
        return False

    # Check for negative prices or volumes
    if (df[['high_price', 'low_price', 'close_price']] <= 0).any().any():
        logger.error(f"{ticker}: Contains negative or zero prices")
        return False

    if (df['volume'] < 0).any():
        logger.error(f"{ticker}: Contains negative volume")
        return False

    return True
```

Usage Example

python

```python
if __name__ == "__main__":
    # Database connection string
    connection_string = "postgresql://user:password@localhost:5432/stock_db"

    # Calculate indicators for today
    results = calculate_technicals(
        connection_string=connection_string,
        target_date=None,  # Today
        tickers=None,      # All tickers
        lookback_days=250  # Minimum for 200-day indicators
    )

    print(f"Processing complete: {results}")
```

Performance Considerations

1. **Batch Processing**: Process multiple tickers in batches to manage memory
2. **Caching**: Cache unchanged historical calculations
3. **Indexing**: Ensure database indexes on (ticker, date) for fast queries
4. **Parallel Processing**: Use multiprocessing for large ticker lists
5. **Data Validation**: Validate data quality before expensive calculations

Key Requirements Summary

- Use 250+ days of lookback data for reliable 200-day indicators
- Handle missing data gracefully (skip or interpolate based on indicator)
- Update only today's values in the database
- Log all processing steps and errors
- Validate data quality before processing
- Use proper null handling for new tickers with insufficient history
- Consider using TA-Lib library for proven indicator implementations
- Implement proper error recovery and retry logic
- Add data quality checks for outliers and anomalies

This implementation provides a robust foundation for calculating technical indicators that integrates with your existing PostgreSQL database structure while maintaining the flexibility to add new indicators as needed.

Support & Resistance Trading Applications

For Swing Traders - Key Usage Patterns

Entry Strategies:

python

```python
def identify_swing_entry_opportunities(indicators: Dict, current_price: float) -> Dict:
    """
    Identify high-probability swing trading entries using support/resistance.
    """
    opportunities = {
        'long_setups': [],
        'short_setups': [],
        'risk_reward_ratios': {}
    }

    # Long setup: Price near strong support
    nearest_support = indicators.get('nearest_support')
    nearest_resistance = indicators.get('nearest_resistance')

    if nearest_support and nearest_resistance:
        support_distance = (current_price - nearest_support) / current_price
        resistance_distance = (nearest_resistance - current_price) / current_price

        # Long opportunity if close to support
        if 0 <= support_distance <= 0.02:  # Within 2% of support
            risk = current_price - nearest_support
            reward = nearest_resistance - current_price
            risk_reward = reward / risk if risk > 0 else 0

            opportunities['long_setups'].append({
                'entry_price': current_price,
                'stop_loss': nearest_support * 0.98,  # 2% below support
                'target': nearest_resistance * 0.98,  # Conservative target
                'risk_reward': risk_reward
            })

        # Short opportunity if close to resistance
        if 0 <= resistance_distance <= 0.02:  # Within 2% of resistance
            risk = nearest_resistance - current_price
            reward = current_price - nearest_support
            risk_reward = reward / risk if risk > 0 else 0

            opportunities['short_setups'].append({
                'entry_price': current_price,
                'stop_loss': nearest_resistance * 1.02,  # 2% above resistance
                'target': nearest_support * 1.02,  # Conservative target
                'risk_reward': risk_reward
            })

    return opportunities
```

Level Strength Assessment:

- **Strength 8-10**: Very strong levels, ideal for major position sizing
- **Strength 5-7**: Moderate levels, good for partial positions
- **Strength 1-4**: Weak levels, use with caution

Time Frame Considerations:

- **5-day swings**: Short-term swing trades (2-5 days)
- **10-day swings**: Medium-term swing trades (1-2 weeks)
- **20-day swings**: Longer swing trades (2-4 weeks)
- **Monthly levels**: Position trading (1-3 months)

Risk Management with S/R Levels:

python

```python
def calculate_position_size_with_sr(account_size: float, risk_per_trade: float,
                                   entry_price: float, stop_loss: float) -> int:
    """
    Calculate position size based on support/resistance stop loss.
    """
    risk_amount = account_size * risk_per_trade  # e.g., 2% of account
    price_risk = abs(entry_price - stop_loss)
    shares = int(risk_amount / price_risk)
    return shares

# Example usage:
# shares = calculate_position_size_with_sr(100000, 0.02, 150.00, 145.00)
# This risks 2% of $100k account with stop at support level
```

Alert System for Support/Resistance

python

```python
def generate_sr_alerts(ticker: str, indicators: Dict, current_price: float) -> List[str]:
    """
    Generate alerts when price approaches key support/resistance levels.
    """
    alerts = []

    # Check proximity to key levels
    for level_name, level_value in indicators.items():
        if 'support' in level_name or 'resistance' in level_name:
            if level_value is None or pd.isna(level_value):
                continue

            latest_level = level_value.iloc[-1] if hasattr(level_value, 'iloc') else level_valu
            distance_pct = abs(current_price - latest_level) / current_price

            if distance_pct <= 0.015:  # Within 1.5% of level
                direction = "approaching resistance" if latest_level > current_price else "appr
                alerts.append(f"{ticker}: {direction} at ${latest_level:.2f} (current: ${curren

    return alerts
```

This comprehensive support and resistance framework provides swing traders with the essential tools for identifying high-probability entry and exit points, managing risk effectively, and maximizing profit potential through proper level analysis.