🔄 Chunked Historical Processing Flow

This document outlines the architecture, control flow, and technical implementation of chunked historical data processing in EventTrader.

1. Overall System Architecture

The chunked historical processing system uses a sequential, multi-stage approach to reliably process large historical date ranges while managing system resources effectively.

High-Level Flow

1. **Bash Script**: Entry point via `event_trader.sh chunked-historical`
2. **Date Chunking**: Breaks date range into smaller chunks (default: 5 days per chunk)
3. **Sequential Processing**: Launches separate Python processes for each chunk
4. **Completion Monitoring**: Python monitors Redis for processing completion
5. **Clean State**: Ensures clean system termination between chunks
6. **Embedding Generation**: Triggers vector embedding creation after processing
7. **Reconciliation**: Handles edge cases where items remain stuck

```
chunked-historical {from_date} {to_date}
    │
    ├── Chunk 1: Process days 1-5
    │   └── Monitor completion via Redis state
    │
    ├── Chunk 2: Process days 6-10
    │   └── Monitor completion via Redis state
    │
    └── Chunk N: Process remaining days
        └── Monitor completion via Redis state
```

2. Entry Point: event_trader.sh in Detail

- **event_trader.sh**: Bash script control interface
  - `chunked-historical` command
    - **Command validation**:
      - Checks that required FROM_DATE is provided
      - Uses default TO_DATE (today) if not specified
      - Verifies date format (YYYY-MM-DD)
    - **`process_chunked_historical()` function execution**:
      - **Configuration loading**:
        - Calls `detect_python()` to find Python interpreter

- Loads `HISTORICAL_CHUNK_DAYS` and `HISTORICAL_STABILITY_WAIT_SECONDS` from `config/feature_flags.py`
- Validates configuration values are positive integers

- **Logging setup**:
  - Creates unique folder `logs/ChunkHist_{FROM_DATE}_to_{TO_DATE}_{TIMESTAMP}/`
  - Creates separate log files:
    - Combined shell log file for tracking overall process
    - Individual log file for each chunk's Python process
  - Defines `shell_log()` function to write to shell log
  - Shell logs are written to combined file while Python logs go to chunk-specific files
- **System checks**:
  - Verifies Redis connectivity with `redis-cli ping`
  - Logs available data sources (news, reports, transcripts)
  - Records total start time for duration tracking
- **Date chunking**:
  - Converts date strings to Unix timestamps (OS-specific compatibility)
  - Creates chunks based on `HISTORICAL_CHUNK_DAYS` configuration
  - Initializes chunk counter and monitoring variables
- **Processing loop** (for each chunk):

  - Records chunk start time

  - Calculates chunk start/end dates

  - Creates chunk-specific log file

  - Executes `stop-all` to terminate previous instances

  - Executes Python processor with parameters:

```
$PYTHON_CMD "$SCRIPT_PATH" \
  --from-date "$chunk_start" \
  --to-date "$chunk_end" \
  -historical \
  --ensure-neo4j-initialized \
  --log-file "$CHUNK_LOG_FILE"
```

  - **Process monitoring and PID tracking**:

    - Captures and stores Python PID: `EVENTTRADER_PID=$!`
    - Writes PID to file for external tracking
    - Monitors process with timeout controls:
      - Watches for completion messages in log files
      - Periodically checks if process is still running via PID
      - Times out after 2 hours maximum per chunk
    - Handles process termination if needed:
      - First attempts graceful shutdown with SIGTERM
      - Waits 5 seconds for clean exit
      - Forces termination with SIGKILL if process remains

- Captures Python exit code
- Handles success/failure scenarios
- **Log summary extraction**:
  - Extracts key events (errors, warnings, completions) from chunk log
  - Appends summary to combined log file
- Executes `stop-all` to ensure clean state between chunks
- Calculates and logs chunk duration
- Advances to next chunk start date

- **Finalization**:
  - Calculates and logs total process duration
  - Creates summary file with statistics
  - Logs completion message with full range processed

3. Python Application: run_event_trader.py

- **run_event_trader.py**: Main Python entry point
  - **`main()` function**:
    - **Error handling setup**:
      - Comprehensive try-except block for entire application
      - Graceful shutdown on errors with detailed logging
    - **Command-line processing**:
      - `parse_args()`: Processes command-line arguments
        - Parses `--from-date` and `--to-date` (required)
        - Handles `-historical` flag to disable live data
        - Processes `--ensure-neo4j-initialized` flag
        - Configures `--log-file` path (receives chunk-specific log file)
    - **Feature flag configuration**:
      - Sets `ENABLE_HISTORICAL_DATA=True` (for chunked-historical mode)
      - Sets `ENABLE_LIVE_DATA=False` (historical only)
    - **Logging initialization**:
      - Sets up logging framework with file and console handlers
      - Writes to the chunk-specific log file provided by shell script
    - **Signal handlers**:
      - Registers handlers for SIGINT/SIGTERM for clean shutdown
    - **DataManager creation**:
      - Initializes manager with date range: `manager = DataManager(date_from, date_to)`
    - **Neo4j validation**:
      - Verifies Neo4j connection with `manager.has_neo4j()`
      - Proceeds if initialized, exits with error if failed
    - **System startup**:
      - Calls `manager.start()` to begin processing
      - Enters monitoring loop for completion in historical-only mode

        - **Completion monitoring** (historical mode):
            - Helper functions:
                - `check_initial_processing()`: Checks fetch completion and queue states
                - `only_withreturns_remain()`: Detects special case where only items in withreturns remain
            - Obtains Redis connection from manager
            - Monitors multiple Redis indicators for each source:
                1. Batch fetch completion flags: `batch:{source}:{from}-{to}:fetch_complete`
                2. Raw queue emptiness: `{source}:queues:raw`
                3. Historical namespace emptiness: `{source}:hist:{raw|processed}:*`
                4. Pending returns sets: `{source}:pending_returns`
                5. WithReturns namespace: `{source}:withreturns:*`
                6. WithoutReturns namespace: `{source}:withoutreturns:*`
            - Periodically checks status every 30 seconds
            - Implements timeout/retry mechanism (`WITHRETURNS_MAX_RETRIES`)
            - Triggers reconciliation if processing stalls with only withreturns items
            - Logs completion status for each source
        - **Embedding generation**:
            - After all processing completes, calls `neo4j_processor.batch_process_qaexchange_embeddings()`
            - Uses batch size defined in `feature_flags.QAEXCHANGE_EMBEDDING_BATCH_SIZE`
            - Embeds QA pairs to enable semantic search
        - **Shutdown and cleanup**:
            - Calls `manager.stop()` to shut down cleanly
            - Logs "Shutdown complete. Exiting Python process" (triggers shell script detection)
            - Exits Python process with success code (0)
            - Returns control to bash script for next chunk

4. Data Manager Initialization and Source Management

- **DataManager.__init__(date_from, date_to)**:
    - **Core initialization**:
        - Sets up logging and signal handlers
        - Stores date range in `historical_range` dictionary
        - Creates empty `sources` dictionary
    - **Source initialization**:
        - `initialize_sources()`: Creates source manager instances
            - `BenzingaNewsManager(historical_range)`: News data source
                - Initializes Redis connections (separate for live/historical)
                - Creates NewsProcessor for raw→processed conversion
                - Creates ReturnsProcessor for market impact calculation
            - `ReportsManager(historical_range)`: SEC filings source
                - Initializes Redis connections and processors

            - `TranscriptsManager(historical_range)`: Earnings calls source
                - Initializes Redis connections and processors
    - **Neo4j initialization**:
        - `initialize_neo4j()`: Sets up graph database connection
            - Creates Neo4jProcessor instance
            - Initializes database schema (creates constraints, indexes)
            - Creates date nodes for the processing date range
            - Validates connectivity
            - Starts background processing thread via `process_with_pubsub()`

5. Source Manager Start Processes

- **manager.start()**: Initiates all data processing
    - Calls `start()` on each source manager, which:
        - **BenzingaNewsManager.start()**:
            - `rest_client.get_historical_data()`: Fetches data directly in the main execution path (not in a separate thread)
            - Creates and starts daemon threads:
                - `processor_thread`: Runs `processor.process_all_news()`
                - `returns_thread`: Runs `returns_processor.process_all_returns()`
        - **ReportsManager.start()**:
            - Creates and starts daemon threads:
                - `processor_thread`: Runs `processor.process_all_reports()`
                - `returns_thread`: Runs `returns_processor.process_all_returns()`
                - `historical_thread`: Runs `rest_client.get_historical_data()` (unlike News, Reports data is fetched in a thread)
        - **TranscriptsManager.start()**:
            - `_initialize_transcript_schedule()`: Sets up retrieval plan
            - Creates and starts daemon threads:
                - `processor_thread`: Runs `processor.process_all_transcripts()`
                - `returns_thread`: Runs `returns_processor.process_all_returns()`
                - `historical_thread`: Runs `_fetch_historical_data()`
    - Returns dictionary of status results from all source starts

6. Processor Data Flow

6.1 Per-Source Processing Flow

Each data source follows a similar processing pattern:

1. **Fetch Stage** - Historical data retrieval
    - Sets `batch:{source}:{from}-{to}:fetch_complete` when fetch completes

    - Stores raw items in Redis: `{source}:hist:raw:{id}`, adds to `{source}:queues:raw`
2. **Base Processing** - Raw to structured data conversion
    - `BaseProcessor.process_all_items()` consumes items from `{source}:queues:raw`
    - Converts raw data to structured format (standardize, clean, add metadata)
    - Stores processed items: `{source}:hist:processed:{id}`
    - Adds to `{source}:queues:processed`
3. **Returns Processing** - Market impact analysis
    - `ReturnsProcessor.process_all_returns()` handles processed items
    - Calculates hourly, session, and daily returns
    - Uses event metadata to determine return timing
    - Stores as `{source}:withreturns:{id}` (complete) or `{source}:withoutreturns:{id}` (pending)
    - Adds incomplete items to `{source}:pending_returns` ZSET with due timestamp
4. **Neo4j Integration** - Graph database storage
    - `Neo4jProcessor.process_with_pubsub()` processes withreturns/withoutreturns entries
    - Creates nodes and relationships in Neo4j database
    - Deletes items from Redis once successfully stored in Neo4j

6.2 Embedding Generation

After all processing completes, the system triggers embedding generation:

- **Embedding Generation Process**:
    - `batch_process_qaexchange_embeddings()`: Processes all question-answer pairs
    - Creates vector embeddings using OpenAI API
    - Stores embeddings as node properties in Neo4j
    - Enables semantic similarity search on text content

7. Completion Monitoring and Error Handling

7.1 Completion Monitoring

The system employs a multi-stage approach to determine when processing is complete:

- **Fetch Completion**:
    - Checks `batch:{source}:{from}-{to}:fetch_complete` flags
    - Verifies `{source}:queues:raw` is empty
    - Ensures no items remain in historical raw namespace
- **Processing Completion**:
    - Verifies historical processed namespace is empty
    - Checks pending returns sets are empty
- **Final Completion**:
    - Ensures withreturns and withoutreturns namespaces are empty
    - Indicates all items have been successfully moved to Neo4j

7.2 Error Handling and Recovery

The system implements several error recovery mechanisms:

- **Timeout Controls**:
  - Maximum 2-hour timeout per chunk in shell script
  - Monitoring cycle with retries in Python process
- **Reconciliation**:
  - After `WITHRETURNS_MAX_RETRIES` cycles (default: 3) monitoring cycles
  - Checks if only withreturns items remain (`only_withreturns_remain()`)
  - Triggers `reconcile_missing_items()` to force reload from Redis to Neo4j
- **Clean Shutdown**:
  - Signal handling for graceful termination
  - `manager.stop()` ensures proper cleanup
  - `stop-all` command between chunks

8. Thread Execution By Mode

The following table shows which threads are started (✅) or not started (🚫) in chunked-historical mode compared to live mode:

Thread	Live Mode (-live)	Chunked-Historical (-historical)
processor_thread (News)	✅	✅
returns_thread (News)	✅	✅
ws_thread (News WebSocket)	✅	🚫
historical_thread (News Historical Fetch)	🚫	🚫
processor_thread (Reports)	✅	✅
returns_thread (Reports)	✅	✅
ws_thread (Reports WebSocket)	✅	🚫
historical_thread (Reports Historical Fetch)	🚫	✅
processor_thread (Transcripts)	✅	✅
returns_thread (Transcripts)	✅	✅
ws_thread (Transcripts)	🚫	🚫
historical_thread (Transcripts Historical Fetch)	🚫	✅
neo4j_thread (PubSub Event Processor)	✅	✅

9. Detailed Redis Flow and Completion States

9.1 Fetch Completion Indicators

For each source, the system sets completion flags when fetching is complete:

```
batch:news:{from_date}-{to_date}:fetch_complete = "1"
batch:reports:{from_date}-{to_date}:fetch_complete = "1"
batch:transcripts:{from_date}-{to_date}:fetch_complete = "1"
```

9.2 Queue and Namespace Emptiness Checks

The system verifies multiple Redis structures for emptiness:

1. **Raw Queues** - Must be empty to indicate all raw data has been processed:

   ```
   news:queues:raw
   reports:queues:raw
   transcripts:queues:raw
   ```

2. **Historical Namespaces** - Must be empty to ensure all items were processed:

   ```
   news:hist:raw:*
   news:hist:processed:*
   reports:hist:raw:*
   reports:hist:processed:*
   transcripts:hist:raw:*
   transcripts:hist:processed:*
   ```

3. **Return Storage** - All items must be processed by Neo4j:

   ```
   news:withreturns:*
   news:withoutreturns:*
   reports:withreturns:*
   reports:withoutreturns:*
   transcripts:withreturns:*
   transcripts:withoutreturns:*
   ```

4. **Pending Returns** - ZSET must be empty to indicate all returns are calculated:

   ```
   news:pending_returns
   reports:pending_returns
   ```

9.3 Reconciliation Logic

When only withreturns items remain:

1. System detects withreturns keys but all other conditions are met
2. After `WITHRETURNS_MAX_RETRIES` cycles (default: 3)
3. Triggers `reconcile_missing_items()`
4. Forces check and reload of Redis keys into Neo4j
5. Handles potential race conditions if Neo4j connection issues occurred

10. Performance and Scalability Considerations

10.1 Resource Management

- **Memory Efficiency**:
  - Processing in chunks prevents memory exhaustion
  - Clean termination between chunks releases memory
- **CPU Utilization**:
  - Limit of one chunk processed at a time
  - Parallel processing within each chunk via daemon threads
- **API Rate Limiting**:
  - Smaller chunks reduce burst API usage
  - Configurable chunk size via `HISTORICAL_CHUNK_DAYS`

10.2 Failure Isolation

- **Chunk Isolation**:
  - Failure in one chunk doesn't affect others
  - Shell script tracks per-chunk success/failure
  - Detailed logs for troubleshooting each chunk

10.3 Configuration Parameters

- **`HISTORICAL_CHUNK_DAYS`** (default: 5):
  - Controls chunk size in days
  - Lower values reduce memory usage but increase overhead
  - Higher values increase efficiency but require more resources
- **`HISTORICAL_STABILITY_WAIT_SECONDS`** (default: 60):
  - Optional wait time between chunks
  - Allows system stability before starting new chunk
- **`WITHRETURNS_MAX_RETRIES`** (default: 3):
  - Controls reconciliation trigger threshold
  - Number of monitoring cycles before forcing reconciliation

11. Logging and Monitoring

11.1 Log Structure

- **Shell Script Logs**:
  - Main shell operations log: `logs/ChunkHist_{FROM_DATE}_to_{TO_DATE}_{TIMESTAMP}/combined_{FROM_DATE}_to_{TO_DATE}.log`
    - Contains shell script operations and commands
    - Includes summaries extracted from chunk logs
    - Tracks overall progress across all chunks
  - Per-chunk logs: `logs/ChunkHist_{FROM_DATE}_to_{TO_DATE}_{TIMESTAMP}/chunk_{start}_to_{end}.log`
    - Contains detailed Python process logs for each chunk
    - Detailed error messages and stack traces

        - Full Redis completion monitoring status
    - Summary file: `logs/ChunkHist_{FROM_DATE}_to_{TO_DATE}_{TIMESTAMP}/summary.txt`
        - Provides overview of complete run
- **Log Flow**:
    - Shell script writes directly to combined log via `shell_log()`
    - Python processes write to their individual chunk logs
    - Shell script extracts important events from chunk logs and appends to combined log
    - This two-tier approach keeps detailed logs separate while maintaining overall visibility

11.2 Monitoring Points

- **Critical Checkpoints**:
    1. Fetch completion for each source
    2. Queue emptiness
    3. Returns calculation completion
    4. Neo4j ingestion completion
    5. Embedding generation success
- **Performance Metrics**:
    - Per-chunk processing time
    - Overall job duration
    - Items processed per source

12. Conclusion

The chunked historical processing system provides a robust, scalable approach to processing large historical data ranges. Its key advantages include:

1. **Resource Efficiency**: Controlled memory and CPU usage
2. **Fault Tolerance**: Chunk isolation prevents cascading failures
3. **Comprehensive Monitoring**: Multi-stage completion checks
4. **Error Recovery**: Automatic reconciliation mechanisms
5. **Scalability**: Configurable parameters for different environments

These capabilities enable reliable processing of extensive historical data while maintaining system stability and performance.