

# Mapper Module Parallel Processing (MPP) Design & Implementation Plan

**Date:** December 23, 2025

**Subject:** Technical Design & Implementation Strategy

**Status:** Draft for Review

## 1. Executive Summary

This document outlines the design and implementation plan for introducing parallel processing (MPP) capabilities to the mapper module. The primary objective is to significantly improve performance when handling large data volumes (1M to 100M+ rows). The design emphasizes a "Chunk-Based" approach to extraction, transformation, and loading while preserving existing UI/UX and maintaining backward compatibility.

## 2. Table of Contents

1. Current Architecture Analysis
2. Design Goals & Requirements
3. Proposed Architecture
4. Implementation Strategy
5. Technical Design
6. Configuration & Tuning
7. Error Handling & Recovery
8. Testing Strategy
9. Performance Expectations
10. Implementation Phases
11. Risks & Mitigation

## 3. Current Architecture Analysis

### 3.1 Current Data Flow

1. **Trigger:** User initiates job execution.
2. **Loading:** Execution Engine loads job flow configuration.
3. **Extraction:** Single SQL query executed via `fetchall`.

4. **Memory:** Entire dataset loaded into memory (DataFrame/cursor).
5. **Transformation:** Sequential processing (row-by-row or vectorized).
6. **Loading:** Sequential batch inserts (typically 1000 rows).

### 3.2 Key Limitations

- **Single-threaded extraction:** Memory-bound and limited by single-stream I/O.
- **CPU Underutilization:** Sequential logic fails to leverage multi-core processors.
- **Scalability:** Large datasets frequently encounter Out-of-Memory (OOM) errors.

## 4. Design Goals & Requirements

### 4.1 Functional Requirements

- **Hybrid Support:** Must handle both relational database sources and file-based sources.
- **Backward Compatibility:** Existing jobs must run without modification.
- **Transparency:** No UI changes required; maintain existing progress reporting.

### 4.2 Performance Requirements

- **Scalability:** Handle up to 100M+ rows efficiently.
- **Memory Efficiency:** Process data in discrete chunks to maintain a low memory footprint.
- **Fault Tolerance:** Capability to report partial success and handle individual chunk failures.

## 5. Proposed Architecture

The architecture introduces a **Parallel Processing Coordinator** that orchestrates the workflow.

### 5.1 Core Components

1. **Coordinator:** Manages the worker thread pool and aggregates results.
2. **Chunk Manager:** Logic to split source data (e.g., `ROW_NUMBER()` for SQL).
3. **Worker Pool:** ThreadPoolExecutor managing concurrent Extract-Transform-Load tasks.
4. **Result Aggregator:** Consolidates statistics and logs for the final job status.

## 6. Technical Design

### 6.1 Chunking Strategy

For database sources, the system will wrap existing queries to facilitate range-based extraction.

**Example (PostgreSQL):**

```

SELECT * FROM (
    SELECT ..., ROW_NUMBER() OVER (ORDER BY <key_column>) as rn
    FROM (<original_query>) subq
) WHERE rn BETWEEN :start_row AND :end_row

```

## 6.2 Implementation Snippet (Coordinator)

```

class ParallelProcessor:
    def process_mapper_job(self, source_conn, target_conn, source_sql, ...):
        # Calculate total chunks based on estimate
        total_rows = self._estimate_total_rows(source_conn, source_sql)
        num_chunks = (total_rows + self.chunk_size - 1) // self.chunk_size

        with ThreadPoolExecutor(max_workers=self.max_workers) as executor:
            # Dispatch chunks to workers
            futures = [executor.submit(self._process_chunk, i, ...) for i in range(num_chunks)]
        return self._aggregate_results(futures)

```

## 7. Configuration & Tuning

### 7.1 Recommended Settings

Dataset Size	Workers	Chunk Size
< 100K Rows	1 (Sequential)	N/A
100K - 1M Rows	2 - 4	50K
1M - 10M Rows	4 - 8	100K
> 10M Rows	8 - 16	100K+

## 8. Performance Expectations

Metric	Expected Improvement
<b>Small Datasets</b>	~1.2x Speedup (Minimal gain due to overhead)
<b>Medium Datasets</b>	~3.0x Speedup

Very Large Datasets

~4.5x Speedup

## 9. Risks & Mitigation

1. **Connection Exhaustion:** Mitigation involves setting `max_workers` relative to the DB connection pool size.
2. **Memory Overload:** Mitigation includes strict limits on concurrent active chunks.
3. **Transaction Deadlocks:** Each chunk will operate in its own independent transaction.

## 10. Conclusion

Introducing MPP to the Mapper module provides a robust path to high-performance data engineering within the existing framework. By utilizing chunk-based processing, we achieve significant throughput gains while remaining transparent to the end-user.