Okay, let's redesign the Spring Batch solution focusing on external parameterization, performance, SOLID principles, design patterns, and better concurrency management.

We will break down responsibilities more granularly and leverage Spring Batch's features for dynamic configuration and parallel execution.

**Key Improvements & Principles Applied:**

1. **External Triggering & Parameterization:** A dedicated service (BatchTriggerService) accepts job parameters to launch specific job instances dynamically.
2. **SRP (Single Responsibility Principle):**
   * BatchTriggerService: Launches jobs.
   * JobConfiguration: Configures *a specific* job's beans (Step, Reader, Processor, Writer). Potentially multiple config classes for different *types* of jobs.
   * MappingService: Responsible *only* for retrieving the MappingInfo list based on an identifier.
   * ReaderFactory: Creates configured ItemReader instances based on parameters.
   * Processor: Handles only transformation/validation logic.
   * Writer: Handles only bulk database writing based on mapping.
3. **OCP (Open/Closed Principle):** Components are designed to allow extension (e.g., adding new mapping types, new validation rules) without modifying existing code, often through strategy patterns or configuration.
4. **DIP (Dependency Inversion Principle):** Components depend on abstractions (interfaces, Spring Batch components), not concrete implementations. Achieved via constructor injection and Spring's DI.
5. **Parameter Awareness (@StepScope, @JobScope):** Components like Readers and Writers are created within the scope of a step or job, allowing them to access JobParameters passed during the launch. This is crucial for dynamic behavior.
6. **Performance:**
   * **Multi-threaded Step:** Retained for parallel chunk processing within a job instance.
   * **Optimized TaskExecutor:** Configurable thread pool.
   * **Efficient I/O:** JdbcPagingItemReader and JdbcTemplate.batchUpdate.
   * **Dynamic SQL Caching:** SQL string is built once per step execution based on parameters/mapping.
7. **Concurrency:** Spring Batch handles concurrent job executions via the JobRepository. Ensure underlying resources (thread pool, DB connections) are adequately sized. Using unique JobParameters is essential.
8. **Resource Management:** Chunk processing keeps memory usage low. @StepScope ensures components are created/destroyed appropriately.

**1. Service Layer: Triggering the Job**

This service acts as the entry point, accepting parameters and launching the job.

Java

package com.yourapp.batch.service;  
  
import com.yourapp.batch.config.mapping.MappingInfo;  
import com.yourapp.batch.config.mapping.MappingService;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.batch.core.Job;  
import org.springframework.batch.core.JobParameters;  
import org.springframework.batch.core.JobParametersBuilder;  
import org.springframework.batch.core.JobParametersInvalidException;  
import org.springframework.batch.core.launch.JobLauncher;  
import org.springframework.batch.core.repository.JobExecutionAlreadyRunningException;  
import org.springframework.batch.core.repository.JobInstanceAlreadyCompleteException;  
import org.springframework.batch.core.repository.JobRestartException;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Service;  
  
import java.util.Date;  
import java.util.List;  
import java.util.Map;  
  
@Service  
public class BatchTriggerService {  
  
 private static final Logger log = LoggerFactory.getLogger(BatchTriggerService.class);  
  
 private final JobLauncher jobLauncher;  
 private final Job dynamicInsertJob; // Inject the specific Job bean  
 private final MappingService mappingService;  
  
 // Constants for Job Parameter Keys  
 public static final String PARAM\_TIMESTAMP = "timestamp";  
 public static final String PARAM\_TARGET\_TABLE = "targetTable";  
 public static final String PARAM\_MAPPING\_ID = "mappingId";  
 public static final String PARAM\_SOURCE\_FILTER = "sourceFilter"; // Example custom param  
  
  
 @Autowired  
 public BatchTriggerService(JobLauncher jobLauncher,  
 @Qualifier("dynamicInsertJob") Job dynamicInsertJob, // Qualify the specific job  
 MappingService mappingService) {  
 this.jobLauncher = jobLauncher;  
 this.dynamicInsertJob = dynamicInsertJob;  
 this.mappingService = mappingService;  
 }  
  
 /\*\*  
 \* Triggers the dynamic insertion batch job.  
 \*  
 \* @param targetTable The destination table name.  
 \* @param mappingId Identifier to fetch the correct field mapping configuration.  
 \* @param sourceFilter Optional filter criteria for the source query (e.g., "WHERE status='NEW'").  
 \* @param additionalParams Optional map for other custom parameters.  
 \* @return The JobExecution ID.  
 \* @throws JobExecutionAlreadyRunningException If the same job instance is running.  
 \* @throws JobRestartException If the job instance cannot be restarted.  
 \* @throws JobInstanceAlreadyCompleteException If the job instance is already complete.  
 \* @throws JobParametersInvalidException If parameters are invalid.  
 \* @throws IllegalArgumentException If mapping info is not found.  
 \*/  
 public Long runDynamicInsertJob(String targetTable, String mappingId, String sourceFilter, Map<String, Object> additionalParams)  
 throws JobExecutionAlreadyRunningException, JobRestartException,  
 JobInstanceAlreadyCompleteException, JobParametersInvalidException {  
  
 log.info("Attempting to launch job for table: {}, mappingId: {}", targetTable, mappingId);  
  
 // Validate mapping exists before launching (optional, but good practice)  
 List<MappingInfo> mapping = mappingService.getMappingInfo(mappingId);  
 if (mapping == null || mapping.isEmpty()) {  
 log.error("Mapping configuration not found for ID: {}", mappingId);  
 throw new IllegalArgumentException("Mapping configuration not found for ID: " + mappingId);  
 }  
  
 JobParametersBuilder paramsBuilder = new JobParametersBuilder();  
 paramsBuilder.addDate(PARAM\_TIMESTAMP, new Date()); // Ensure unique instance per run  
 paramsBuilder.addString(PARAM\_TARGET\_TABLE, targetTable);  
 paramsBuilder.addString(PARAM\_MAPPING\_ID, mappingId);  
 if (sourceFilter != null && !sourceFilter.isBlank()) {  
 paramsBuilder.addString(PARAM\_SOURCE\_FILTER, sourceFilter);  
 }  
  
 // Add any other dynamic parameters needed by the job steps  
 if (additionalParams != null) {  
 additionalParams.forEach((key, value) -> {  
 if (value instanceof String) paramsBuilder.addString(key, (String) value);  
 else if (value instanceof Long) paramsBuilder.addLong(key, (Long) value);  
 else if (value instanceof Double) paramsBuilder.addDouble(key, (Double) value);  
 else if (value instanceof Date) paramsBuilder.addDate(key, (Date) value);  
 // Add more types as needed  
 });  
 }  
  
 JobParameters jobParameters = paramsBuilder.toJobParameters();  
  
 log.debug("Launching job with parameters: {}", jobParameters);  
 var jobExecution = jobLauncher.run(dynamicInsertJob, jobParameters);  
 log.info("Job launched successfully with execution ID: {}", jobExecution.getId());  
  
 return jobExecution.getId();  
 }  
}

**2. Mapping Configuration Service**

Decouples loading of mapping definitions.

Java

package com.yourapp.batch.config.mapping;  
  
import org.springframework.stereotype.Service;  
import java.util.List;  
import java.util.Map;  
import java.util.concurrent.ConcurrentHashMap;  
import java.sql.Types; // Use java.sql.Types  
  
// Represents the mapping configuration for a single field  
@lombok.Data // Using lombok for brevity  
@lombok.AllArgsConstructor  
@lombok.NoArgsConstructor  
public class MappingInfo {  
 private String sourceFieldName;  
 private String destFieldName;  
 // Store java.sql.Types constant for database independence  
 private int jdbcType;  
 // Optional: Add validation rules, default values etc.  
}  
  
  
@Service  
public class MappingService {  
  
 // Example: Cache mappings in memory. Load from DB/File in a real app.  
 private final Map<String, List<MappingInfo>> mappingCache = new ConcurrentHashMap<>();  
  
 // Example initialization (replace with actual loading logic)  
 @jakarta.annotation.PostConstruct // Use jakarta version with Spring Boot 3+  
 public void initializeMappings() {  
 // Load "productMapping"  
 mappingCache.put("productMapping", List.of(  
 new MappingInfo("prod\_id", "PRODUCT\_ID", Types.BIGINT),  
 new MappingInfo("prod\_name", "PRODUCT\_NAME", Types.VARCHAR),  
 new MappingInfo("price", "LIST\_PRICE", Types.DECIMAL),  
 new MappingInfo("create\_dt", "CREATION\_DATE", Types.TIMESTAMP)  
 ));  
 // Load "customerMapping"  
 mappingCache.put("customerMapping", List.of(  
 new MappingInfo("cust\_ref", "CUSTOMER\_ID", Types.VARCHAR),  
 new MappingInfo("c\_name", "NAME", Types.VARCHAR),  
 new MappingInfo("join\_date", "MEMBER\_SINCE", Types.DATE)  
 ));  
 // ... load other mappings  
 }  
  
 /\*\*  
 \* Retrieves mapping information based on an identifier.  
 \* @param mappingId The unique ID for the mapping configuration.  
 \* @return List of MappingInfo objects, or null if not found.  
 \*/  
 public List<MappingInfo> getMappingInfo(String mappingId) {  
 // In a real app: Load from database or config file if not in cache  
 return mappingCache.get(mappingId);  
 }  
}

**3. Batch Job Configuration (DynamicJobConfig.java)**

Uses @StepScope heavily to make components aware of job parameters.

Java

package com.yourapp.batch.config;  
  
import com.yourapp.batch.config.mapping.MappingInfo;  
import com.yourapp.batch.config.mapping.MappingService;  
import com.yourapp.batch.decider.StatusDecider; // Assuming a decider for conditional flow  
import com.yourapp.batch.listener.StepStatusListener; // Listener for logging/cleanup  
import com.yourapp.batch.model.RecordData; // Your data model class  
import com.yourapp.batch.processor.StatusTrackingProcessor;  
import com.yourapp.batch.reader.FetchRepoReaderFactory; // Factory for reader  
import com.yourapp.batch.writer.DynamicBulkInsertWriter;  
import org.springframework.batch.core.Job;  
import org.springframework.batch.core.Step;  
import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;  
import org.springframework.batch.core.configuration.annotation.JobBuilderFactory;  
import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;  
import org.springframework.batch.core.configuration.annotation.StepScope;  
import org.springframework.batch.core.job.builder.FlowBuilder;  
import org.springframework.batch.core.job.flow.Flow;  
import org.springframework.batch.core.job.flow.support.SimpleFlow;  
import org.springframework.batch.core.launch.support.RunIdIncrementer;  
import org.springframework.batch.item.ItemProcessor;  
import org.springframework.batch.item.ItemReader;  
import org.springframework.batch.item.ItemWriter;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.core.task.TaskExecutor;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;  
  
import javax.sql.DataSource;  
import java.util.List;  
  
@Configuration  
@EnableBatchProcessing // Redundant if using Spring Boot AutoConfig, but explicit  
public class DynamicJobConfig {  
  
 private final JobBuilderFactory jobBuilderFactory;  
 private final StepBuilderFactory stepBuilderFactory;  
 private final DataSource sourceDataSource;  
 private final DataSource targetDataSource;  
 private final MappingService mappingService;  
 private final FetchRepoReaderFactory readerFactory; // Inject the factory  
  
 // Constructor injection promotes DIP  
 public DynamicJobConfig(JobBuilderFactory jobBuilderFactory,  
 StepBuilderFactory stepBuilderFactory,  
 @Qualifier("sourceDataSource") DataSource sourceDataSource,  
 @Qualifier("targetDataSource") DataSource targetDataSource,  
 MappingService mappingService,  
 FetchRepoReaderFactory readerFactory) {  
 this.jobBuilderFactory = jobBuilderFactory;  
 this.stepBuilderFactory = stepBuilderFactory;  
 this.sourceDataSource = sourceDataSource;  
 this.targetDataSource = targetDataSource;  
 this.mappingService = mappingService;  
 this.readerFactory = readerFactory;  
 }  
  
 // Define the central Task Executor for parallel steps  
 // Give it a specific qualifier if multiple executors are needed  
 @Bean  
 @Qualifier("batchTaskExecutor")  
 public TaskExecutor batchTaskExecutor() {  
 ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();  
 // CORE\_POOL\_SIZE: Threads always alive. Start smaller, e.g., number of cores / 2.  
 executor.setCorePoolSize(4);  
 // MAX\_POOL\_SIZE: Upper limit. Depends on I/O vs CPU nature. For I/O bound, can be higher than cores.  
 executor.setMaxPoolSize(10);  
 // QUEUE\_CAPACITY: Tasks waiting if all threads busy. Avoid unbound queues. Sized based on expected burst load.  
 executor.setQueueCapacity(20);  
 // KeepAlive: Time idle threads wait before terminating (down to corePoolSize).  
 executor.setKeepAliveSeconds(60);  
 executor.setThreadNamePrefix("batch-exec-");  
 // REJECTION\_POLICY: What happens if queue is full & max pool reached. CallerRunsPolicy runs task in calling thread (can slow down job submission). AbortPolicy throws exception.  
 executor.setRejectedExecutionHandler(new java.util.concurrent.ThreadPoolExecutor.CallerRunsPolicy());  
 executor.initialize();  
 return executor;  
 }  
  
  
 // --- Job Definition ---  
 @Bean  
 public Job dynamicInsertJob(@Qualifier("mainProcessingStep") Step mainProcessingStep,  
 @Qualifier("completionFlow") Flow completionFlow, // Example conditional flow  
 StatusDecider statusDecider) { // Example decider  
 return jobBuilderFactory.get("dynamicInsertJob")  
 .incrementer(new RunIdIncrementer())  
 .start(mainProcessingStep)  
 // Example of conditional flow based on step outcome  
 .on("COMPLETED").to(completionFlow) // Go to completionFlow if step completes normally  
 .from(mainProcessingStep).on("FAILED").fail() // Fail job if step fails  
 .from(mainProcessingStep).on("\*").stop() // Stop for any other status (optional)  
 // Alternative: Use a Decider for more complex logic  
 // .start(mainProcessingStep)  
 // .next(statusDecider).on("SEND\_NOTIFICATION").to(notificationStep)  
 // .from(statusDecider).on("CONTINUE").to(anotherStep)  
 // .end()  
 .end() // End the main flow path  
 .build();  
 }  
  
 // --- Step Definitions ---  
 @Bean  
 @Qualifier("mainProcessingStep")  
 public Step mainProcessingStep(ItemReader<RecordData> itemReader, // Resolved via @StepScope bean below  
 ItemProcessor<RecordData, RecordData> itemProcessor, // Resolved via @StepScope bean below  
 ItemWriter<RecordData> itemWriter, // Resolved via @StepScope bean below  
 @Qualifier("batchTaskExecutor") TaskExecutor taskExecutor,  
 StepStatusListener stepListener) {  
 return stepBuilderFactory.get("mainProcessingStep")  
 .<RecordData, RecordData>chunk(200) // TUNABLE: Adjust chunk size (e.g., 100-1000)  
 .reader(itemReader)  
 .processor(itemProcessor)  
 .writer(itemWriter)  
 .faultTolerant() // Enable skip/retry logic if needed  
 // .skipLimit(10).skip(SomeSkippableException.class)  
 // .retryLimit(3).retry(SomeRetryableException.class)  
 .listener(stepListener) // Add listeners for logging/notifications  
 .taskExecutor(taskExecutor)  
 // THROTTLE\_LIMIT: Max concurrent chunks being processed. Should align with maxPoolSize.  
 .throttleLimit(10)  
 .build();  
 }  
  
 // Example of another step potentially run conditionally  
 @Bean  
 public Step notificationStep(/\* dependencies \*/) {  
 return stepBuilderFactory.get("notificationStep")  
 .tasklet((contribution, chunkContext) -> {  
 System.out.println("Sending completion notification...");  
 // Add notification logic (email, message queue, etc.)  
 return org.springframework.batch.repeat.RepeatStatus.FINISHED;  
 })  
 .build();  
 }  
  
 // --- Flow Definition (for conditional paths) ---  
 @Bean  
 @Qualifier("completionFlow")  
 public Flow completionFlow(Step notificationStep) { // Example flow  
 FlowBuilder<SimpleFlow> flowBuilder = new FlowBuilder<>("completionFlow");  
 flowBuilder.start(notificationStep)  
 // .next(anotherCleanupStep) // Chain more steps if needed  
 .end();  
 return flowBuilder.build();  
 }  
  
  
 // --- ItemReader Bean (Step Scoped) ---  
 @Bean  
 @StepScope // Creates a new reader instance for each step execution  
 public ItemReader<RecordData> itemReader(  
 @Value("#{jobParameters['" + BatchTriggerService.PARAM\_SOURCE\_FILTER + "']}") String sourceFilter // Inject parameter  
 /\* Inject other needed params \*/) {  
 // Use the factory to create a configured reader instance  
 return readerFactory.createJdbcPagingReader(sourceDataSource, sourceFilter);  
 }  
  
 // --- ItemProcessor Bean (Step Scoped) ---  
 @Bean  
 @StepScope // Create per step execution if it needs Job/Step context or params  
 // Can be singleton if stateless and doesn't need params directly  
 public ItemProcessor<RecordData, RecordData> itemProcessor(  
 @Value("#{jobParameters['" + BatchTriggerService.PARAM\_MAPPING\_ID + "']}") String mappingId) {  
 List<MappingInfo> mappingInfo = mappingService.getMappingInfo(mappingId);  
 if (mappingInfo == null) {  
 throw new IllegalArgumentException("Processor: Mapping configuration not found for ID: " + mappingId);  
 }  
 // Pass the dynamically retrieved mapping to the processor instance  
 return new StatusTrackingProcessor(mappingInfo);  
 }  
  
 // --- ItemWriter Bean (Step Scoped) ---  
 @Bean  
 @StepScope // Crucial for dynamic SQL and target table based on parameters  
 public ItemWriter<RecordData> itemWriter(  
 @Qualifier("targetJdbcTemplate") JdbcTemplate targetJdbcTemplate,  
 @Value("#{jobParameters['" + BatchTriggerService.PARAM\_MAPPING\_ID + "']}") String mappingId,  
 @Value("#{jobParameters['" + BatchTriggerService.PARAM\_TARGET\_TABLE + "']}") String targetTable) {  
 List<MappingInfo> mappingInfo = mappingService.getMappingInfo(mappingId);  
 if (mappingInfo == null || mappingInfo.isEmpty()) {  
 throw new IllegalArgumentException("Writer: Mapping configuration not found or empty for ID: " + mappingId);  
 }  
 if (targetTable == null || targetTable.isBlank()) {  
 throw new IllegalArgumentException("Writer: Target table name parameter is missing or empty.");  
 }  
 // Pass dynamic parameters to the writer instance  
 return new DynamicBulkInsertWriter(targetJdbcTemplate, mappingInfo, targetTable);  
 }  
  
 // --- Supporting Beans ---  
 @Bean  
 @Qualifier("targetJdbcTemplate")  
 public JdbcTemplate targetJdbcTemplate(@Qualifier("targetDataSource") DataSource targetDataSource) {  
 return new JdbcTemplate(targetDataSource);  
 }  
  
 // Example Decider bean  
 @Bean  
 public StatusDecider statusDecider() {  
 return new StatusDecider();  
 }  
  
 // Example Listener bean  
 @Bean  
 public StepStatusListener stepListener() {  
 return new StepStatusListener();  
 }  
}

**4. ItemReader Factory (WorkspaceRepoReaderFactory.java)**

Encapsulates the creation logic for the reader, making the config cleaner.

Java

package com.yourapp.batch.reader;  
  
import com.yourapp.batch.model.RecordData;  
import com.yourapp.batch.model.ProcessingStatus;  
import org.springframework.batch.item.database.JdbcPagingItemReader;  
import org.springframework.batch.item.database.PagingQueryProvider;  
import org.springframework.batch.item.database.builder.JdbcPagingItemReaderBuilder;  
import org.springframework.batch.item.database.support.SqlPagingQueryProviderFactoryBean;  
import org.springframework.jdbc.core.RowMapper;  
import org.springframework.stereotype.Component;  
  
import javax.sql.DataSource;  
import java.sql.ResultSet;  
import java.sql.ResultSetMetaData;  
import java.sql.SQLException;  
import java.util.HashMap;  
import java.util.Map;  
  
@Component // Make it a Spring bean so it can be injected  
public class FetchRepoReaderFactory {  
  
 // TUNABLE: Page size for reader query  
 private static final int PAGE\_SIZE = 500;  
  
 public JdbcPagingItemReader<RecordData> createJdbcPagingReader(DataSource dataSource, String sourceFilter) {  
 try {  
 return new JdbcPagingItemReaderBuilder<RecordData>()  
 .name("fetchRepoReader") // Unique name for the reader bean instance  
 .dataSource(dataSource)  
 .queryProvider(createQueryProvider(dataSource, sourceFilter))  
 .pageSize(PAGE\_SIZE) // How many records to fetch per DB query  
 .rowMapper(new RecordDataRowMapper()) // Use the custom row mapper  
 .saveState(false) // Essential for multi-threaded/partitioned steps  
 .build();  
 } catch (Exception e) {  
 throw new RuntimeException("Failed to create JDBC Paging Reader", e);  
 }  
 }  
  
 private PagingQueryProvider createQueryProvider(DataSource dataSource, String sourceFilter) throws Exception {  
 SqlPagingQueryProviderFactoryBean factory = new SqlPagingQueryProviderFactoryBean();  
 factory.setDataSource(dataSource);  
  
 // --- !!! IMPORTANT DYNAMIC PART !!! ---  
 // Base query structure - Adapt SELECT and FROM to your actual source table  
 String selectClause = "SELECT \* "; // Or specify columns explicitly: "SELECT ID, COL1, COL2 "  
 String fromClause = "FROM YOUR\_SOURCE\_TABLE"; // \*\*\* Replace with actual table \*\*\*  
 String whereClause = (sourceFilter != null && !sourceFilter.isBlank()) ? sourceFilter : ""; // Append dynamic filter  
  
 factory.setSelectClause(selectClause);  
 factory.setFromClause(fromClause);  
 if (!whereClause.isEmpty()) {  
 // Ensure sourceFilter starts with "WHERE" or "AND" appropriately if needed  
 // Basic example assumes sourceFilter IS the complete WHERE clause if provided  
 factory.setWhereClause(whereClause);  
 }  
 // \*\*\* Replace with a guaranteed unique sort key from YOUR\_SOURCE\_TABLE \*\*\*  
 factory.setSortKey("UNIQUE\_SOURCE\_ID\_COLUMN");  
 // The factoryBean attempts to auto-detect DB type for pagination syntax (ROWNUM vs OFFSET/FETCH)  
 // May need specific PagingQueryProvider impls for complex cases or unsupported DBs  
  
 return factory.getObject();  
 }  
  
 // Inner class or separate class for RowMapper - SRP  
 private static class RecordDataRowMapper implements RowMapper<RecordData> {  
 @Override  
 public RecordData mapRow(ResultSet rs, int rowNum) throws SQLException {  
 ResultSetMetaData metaData = rs.getMetaData();  
 int columnCount = metaData.getColumnCount();  
 Map<String, Object> sourceFields = new HashMap<>();  
 Long sourceId = null; // Assuming one column holds the unique ID  
  
 for (int i = 1; i <= columnCount; i++) {  
 String columnName = metaData.getColumnLabel(i); // Use label in case of aliases  
 Object value = rs.getObject(i);  
 sourceFields.put(columnName, value);  
 // \*\*\* Adapt this logic to identify your unique source ID column \*\*\*  
 if ("UNIQUE\_SOURCE\_ID\_COLUMN".equalsIgnoreCase(columnName) && value instanceof Number) {  
 sourceId = ((Number) value).longValue();  
 } else if ("ID".equalsIgnoreCase(columnName) && sourceId == null && value instanceof Number) {  
 // Fallback guess if specific ID column name not matched  
 sourceId = ((Number) value).longValue();  
 }  
 }  
  
 if (sourceId == null) {  
 // Handle cases where ID couldn't be determined - maybe throw exception or log warning  
 // Depending on requirements, could use rowNum or hash of fields as fallback temporary ID  
 // throw new SQLException("Could not determine unique source ID for row " + rowNum);  
 System.err.println("Warning: Could not determine unique source ID for row " + rowNum);  
 }  
  
  
 return RecordData.builder()  
 .sourceId(sourceId) // Store the identified ID  
 .sourceFields(sourceFields)  
 .status(ProcessingStatus.PENDING) // Initial status  
 .build();  
 }  
 }  
}

**5. ItemProcessor (StatusTrackingProcessor.java)**

Remains similar, but now receives MappingInfo via constructor injection (facilitated by @StepScope bean definition).

Java

package com.yourapp.batch.processor;  
  
import com.yourapp.batch.config.mapping.MappingInfo;  
import com.yourapp.batch.model.ProcessingStatus;  
import com.yourapp.batch.model.RecordData;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.batch.item.ItemProcessor;  
import java.sql.Types; // Use java.sql.Types  
import java.math.BigDecimal;  
import java.util.Date;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
// This class can often be a singleton bean unless it needs step-specific state  
// But defining as @StepScope in config allows easy access to JobParameters if needed later  
public class StatusTrackingProcessor implements ItemProcessor<RecordData, RecordData> {  
  
 private static final Logger log = LoggerFactory.getLogger(StatusTrackingProcessor.class);  
  
 private final List<MappingInfo> mappingModelList;  
  
 // Constructor injection - Receives mapping specific to this job run  
 public StatusTrackingProcessor(List<MappingInfo> mappingModelList) {  
 this.mappingModelList = mappingModelList;  
 log.debug("Processor initialized with {} mapping rules.", mappingModelList.size());  
 }  
  
 @Override  
 public RecordData process(RecordData item) throws Exception {  
 log.trace("Processing item with source ID: {}", item.getSourceId());  
 Map<String, Object> mappedData = new HashMap<>();  
  
 try {  
 for (MappingInfo mapping : mappingModelList) {  
 Object sourceValue = item.getSourceFields().get(mapping.getSourceFieldName());  
  
 // 1. Basic Validation (Example: Check non-null if required by mapping?)  
 // You could add flags to MappingInfo for 'required', 'maxLength', etc.  
 // if (mapping.isRequired() && sourceValue == null) {  
 // throw new ValidationException("Field '" + mapping.getSourceFieldName() + "' is required but was null.");  
 // }  
  
 // 2. Type Conversion & Mapping  
 Object targetValue = convertAndValidateType(sourceValue, mapping.getJdbcType(), mapping.getDestFieldName());  
 mappedData.put(mapping.getDestFieldName(), targetValue);  
 }  
  
 item.setMappedData(mappedData);  
 item.setStatus(ProcessingStatus.VALIDATED); // Mark as ready for writing  
 return item;  
  
 } catch (Exception e) { // Catch validation or conversion errors  
 log.warn("Failed to process item with source ID {}: {}", item.getSourceId(), e.getMessage());  
 item.setStatus(ProcessingStatus.INVALID);  
 item.setErrorMessage(e.getMessage().substring(0, Math.min(e.getMessage().length(), 255))); // Truncate error msg  
 // Return null to filter out invalid records (they won't be written)  
 // If you need to write failed records elsewhere, return the item with INVALID status  
 // and handle it in the writer or a separate step.  
 return null;  
 }  
 }  
  
 // Enhanced Type Conversion & Basic Validation Logic  
 private Object convertAndValidateType(Object sourceValue, int targetJdbcType, String fieldName) throws Exception {  
 if (sourceValue == null) {  
 // TODO: Add check if null is allowed for this field based on MappingInfo or DB constraints  
 return null;  
 }  
  
 try {  
 switch (targetJdbcType) {  
 case Types.VARCHAR:  
 case Types.NVARCHAR:  
 case Types.LONGVARCHAR:  
 case Types.CHAR:  
 case Types.NCHAR:  
 // TODO: Add length validation if needed based on MappingInfo  
 return sourceValue.toString();  
  
 case Types.INTEGER:  
 if (sourceValue instanceof Number) return ((Number) sourceValue).intValue();  
 return Integer.parseInt(sourceValue.toString());  
 case Types.BIGINT:  
 if (sourceValue instanceof Number) return ((Number) sourceValue).longValue();  
 return Long.parseLong(sourceValue.toString());  
 case Types.SMALLINT:  
 case Types.TINYINT:  
 if (sourceValue instanceof Number) return ((Number) sourceValue).shortValue();  
 return Short.parseShort(sourceValue.toString());  
  
 case Types.DECIMAL:  
 case Types.NUMERIC:  
 if (sourceValue instanceof BigDecimal) return sourceValue;  
 // TODO: Add precision/scale validation if needed  
 return new BigDecimal(sourceValue.toString());  
 case Types.FLOAT: // Represents FLOAT (approximate) - often maps to double  
 case Types.DOUBLE: // Represents DOUBLE PRECISION (approximate)  
 if (sourceValue instanceof Number) return ((Number) sourceValue).doubleValue();  
 return Double.parseDouble(sourceValue.toString());  
 case Types.REAL: // Represents REAL (approximate) - often maps to float  
 if (sourceValue instanceof Number) return ((Number) sourceValue).floatValue();  
 return Float.parseFloat(sourceValue.toString());  
  
 case Types.DATE:  
 if (sourceValue instanceof java.sql.Date) return sourceValue;  
 if (sourceValue instanceof java.util.Date) return new java.sql.Date(((java.util.Date) sourceValue).getTime());  
 // TODO: Add parsing from String if needed (e.g., using SimpleDateFormat - careful with thread safety!)  
 throw new IllegalArgumentException("Cannot convert to java.sql.Date: " + sourceValue.getClass());  
  
 case Types.TIMESTAMP:  
 case Types.TIMESTAMP\_WITH\_TIMEZONE: // Handle TZ if necessary  
 if (sourceValue instanceof java.sql.Timestamp) return sourceValue;  
 if (sourceValue instanceof java.util.Date) return new java.sql.Timestamp(((java.util.Date) sourceValue).getTime());  
 // TODO: Add parsing from String  
 throw new IllegalArgumentException("Cannot convert to java.sql.Timestamp: " + sourceValue.getClass());  
  
 case Types.TIME:  
 case Types.TIME\_WITH\_TIMEZONE: // Handle TZ if necessary  
 if (sourceValue instanceof java.sql.Time) return sourceValue;  
 if (sourceValue instanceof java.util.Date) return new java.sql.Time(((java.util.Date) sourceValue).getTime());  
 // TODO: Add parsing from String  
 throw new IllegalArgumentException("Cannot convert to java.sql.Time: " + sourceValue.getClass());  
  
 case Types.BOOLEAN:  
 case Types.BIT: // JDBC BIT often maps to boolean  
 if (sourceValue instanceof Boolean) return sourceValue;  
 // Handle common string/number representations of boolean  
 String sVal = sourceValue.toString().toLowerCase();  
 if ("true".equals(sVal) || "1".equals(sVal) || "yes".equals(sVal) || "y".equals(sVal)) return Boolean.TRUE;  
 if ("false".equals(sVal) || "0".equals(sVal) || "no".equals(sVal) || "n".equals(sVal)) return Boolean.FALSE;  
 throw new IllegalArgumentException("Cannot convert to Boolean: " + sourceValue);  
  
 // Add cases for BLOB, CLOB, BINARY, VARBINARY, ARRAY etc. if needed  
  
 default:  
 log.warn("Unsupported JDBC type ({}) for field '{}'. Using setObject.", targetJdbcType, fieldName);  
 return sourceValue; // Fallback or throw error  
 }  
 } catch (NumberFormatException | ClassCastException e) {  
 throw new Exception(String.format("Type conversion failed for field '%s' (target type %d): Value '%s', Error: %s",  
 fieldName, targetJdbcType, sourceValue, e.getMessage()), e);  
 } catch (Exception e) {  
 throw new Exception(String.format("Error processing field '%s': %s", fieldName, e.getMessage()), e);  
 }  
 }  
}

**6. ItemWriter (DynamicBulkInsertWriter.java)**

Receives JdbcTemplate, MappingInfo, and targetTable dynamically via constructor injection, facilitated by @StepScope.

Java

package com.yourapp.batch.writer;  
  
import com.yourapp.batch.config.mapping.MappingInfo;  
import com.yourapp.batch.model.ProcessingStatus;  
import com.yourapp.batch.model.RecordData;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.batch.item.ItemWriter;  
import org.springframework.beans.factory.InitializingBean;  
import org.springframework.dao.DataAccessException; // Catch Spring JDBC exceptions  
import org.springframework.jdbc.core.BatchPreparedStatementSetter;  
import org.springframework.jdbc.core.JdbcTemplate;  
import java.sql.PreparedStatement;  
import java.sql.SQLException;  
import java.sql.Types;  
import java.util.List;  
import java.util.Map;  
import java.util.stream.Collectors;  
  
// This component \*must\* be Step Scoped because its behavior (SQL query, target table)  
// depends on job parameters which change per execution.  
public class DynamicBulkInsertWriter implements ItemWriter<RecordData>, InitializingBean {  
  
 private static final Logger log = LoggerFactory.getLogger(DynamicBulkInsertWriter.class);  
  
 private final JdbcTemplate jdbcTemplate;  
 private final List<MappingInfo> mappingModelList;  
 private final String targetTableName;  
  
 // State specific to this step instance  
 private String insertSql;  
 private List<String> orderedDestFields;  
 private Map<String, Integer> fieldTypeMap; // Cache field types for faster lookup  
  
 public DynamicBulkInsertWriter(JdbcTemplate targetJdbcTemplate,  
 List<MappingInfo> mappingModelList,  
 String targetTableName) {  
 this.jdbcTemplate = targetJdbcTemplate;  
 this.mappingModelList = mappingModelList; // This list is specific to the job run  
 this.targetTableName = targetTableName; // Specific to the job run  
 log.debug("Writer initialized for table '{}' with {} mapping rules.", targetTableName, mappingModelList.size());  
 }  
  
 // InitializingBean's method: Called after properties are set.  
 // Perfect place to build the dynamic SQL \*once\* per step execution.  
 @Override  
 public void afterPropertiesSet() throws Exception {  
 buildInsertSqlAndMetadata();  
 }  
  
 private void buildInsertSqlAndMetadata() {  
 if (mappingModelList == null || mappingModelList.isEmpty()) {  
 throw new IllegalStateException("Mapping model list cannot be empty for dynamic writer.");  
 }  
 if (targetTableName == null || targetTableName.trim().isEmpty()) {  
 throw new IllegalStateException("Target table name cannot be empty.");  
 }  
  
 // Ensure consistent order and cache types  
 orderedDestFields = mappingModelList.stream()  
 .map(MappingInfo::getDestFieldName)  
 .collect(Collectors.toList());  
  
 fieldTypeMap = mappingModelList.stream()  
 .collect(Collectors.toMap(MappingInfo::getDestFieldName, MappingInfo::getJdbcType));  
  
 String columns = String.join(", ", orderedDestFields);  
 String placeholders = orderedDestFields.stream().map(f -> "?").collect(Collectors.joining(", "));  
  
 // Consider using dialect-specific quotes if table/column names need it  
 // e.g., for SQL Server: "[" + targetTableName + "]"  
 this.insertSql = String.format("INSERT INTO %s (%s) VALUES (%s)",  
 targetTableName, columns, placeholders);  
  
 log.info("Built dynamic INSERT SQL for step execution: {}", this.insertSql);  
 }  
  
  
 @Override  
 public void write(List<? extends RecordData> items) throws Exception {  
 if (items.isEmpty()) {  
 log.debug("Writer received empty list, skipping write.");  
 return;  
 }  
  
 log.debug("Attempting to write chunk of {} items to table {}", items.size(), targetTableName);  
  
 try {  
 jdbcTemplate.batchUpdate(this.insertSql, new BatchPreparedStatementSetter() {  
 @Override  
 public void setValues(PreparedStatement ps, int i) throws SQLException {  
 RecordData item = items.get(i);  
 Map<String, Object> data = item.getMappedData();  
  
 for (int j = 0; j < orderedDestFields.size(); j++) {  
 String fieldName = orderedDestFields.get(j);  
 Object value = data.get(fieldName);  
 int sqlType = fieldTypeMap.getOrDefault(fieldName, Types.OTHER); // Get cached type  
  
 try {  
 if (value == null) {  
 ps.setNull(j + 1, sqlType);  
 } else {  
 // Use setObject with explicit SQL type for better driver compatibility  
 ps.setObject(j + 1, value, sqlType);  
 // Note: For specific types like DATE, TIMESTAMP, you might still prefer  
 // ps.setDate, ps.setTimestamp if 'value' is guaranteed to be the correct Java type.  
 // setObject is generally safer if types from processor are varied.  
 }  
 } catch (SQLException sqle) {  
 log.error("SQLException setting parameter #{} ('{}', type {}) for item source ID {}: {}",  
 j+1, fieldName, sqlType, item.getSourceId(), sqle.getMessage());  
 // Re-throw to potentially fail the batch, or handle more gracefully if possible  
 throw sqle;  
 }  
 }  
 }  
  
 @Override  
 public int getBatchSize() {  
 return items.size();  
 }  
 });  
  
 // Success Case: Update status in memory (Persistence needs separate strategy)  
 for (RecordData item : items) {  
 item.setStatus(ProcessingStatus.SUCCESS);  
 log.trace("Marked item source ID {} as SUCCESS (in memory)", item.getSourceId());  
 }  
 log.debug("Successfully wrote chunk of {} items.", items.size());  
  
 } catch (DataAccessException e) { // Catch Spring's wrapper  
 log.error("Batch update failed for table {} for chunk starting with source ID {}: {}",  
 targetTableName, items.get(0).getSourceId(), e.getMessage(), e);  
  
 // Failure Case: Mark all items in this chunk as FAILED (in memory)  
 for (RecordData item : items) {  
 item.setStatus(ProcessingStatus.FAILED);  
 item.setErrorMessage("Batch insert failed: " + e.getMessage().substring(0, Math.min(e.getMessage().length(), 255)));  
 log.warn("Marked item source ID {} as FAILED due to batch error (in memory)", item.getSourceId());  
 }  
 // Re-throw the exception to signal failure to the Step execution  
 throw new RuntimeException("Batch update failed for chunk.", e);  
 }  
 // --- IMPORTANT ---  
 // Status is updated IN MEMORY only here. Add a listener or separate step  
 // to persist these SUCCESS/FAILED statuses back to source or a tracking table.  
 }  
}

**7. Supporting Components (Listeners, Deciders - Examples)**

Java

package com.yourapp.batch.listener;  
  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.batch.core.ExitStatus;  
import org.springframework.batch.core.StepExecution;  
import org.springframework.batch.core.StepExecutionListener;  
import org.springframework.stereotype.Component;  
  
// Can be defined as bean in config or use @Component  
public class StepStatusListener implements StepExecutionListener {  
  
 private static final Logger log = LoggerFactory.getLogger(StepStatusListener.class);  
  
 @Override  
 public void beforeStep(StepExecution stepExecution) {  
 log.info("{} starting.", stepExecution.getStepName());  
 // Can initialize resources here  
 }  
  
 @Override  
 public ExitStatus afterStep(StepExecution stepExecution) {  
 log.info("{} finished with status: {}. Summary: {}",  
 stepExecution.getStepName(),  
 stepExecution.getExitStatus().getExitCode(),  
 stepExecution.getSummary()); // Read/Write/Filter/Commit/Rollback counts  
  
 if (stepExecution.getExitStatus().equals(ExitStatus.COMPLETED)) {  
 log.info("Step completed successfully.");  
 // Perform cleanup or success actions  
 } else {  
 log.error("Step failed!");  
 // Log failures, trigger alerts  
 stepExecution.getFailureExceptions().forEach(ex -> log.error("Failure Cause: ", ex));  
 }  
 // Return null to use the original exit status, or return a new one  
 return stepExecution.getExitStatus();  
 }  
}  
  
package com.yourapp.batch.decider;  
  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.batch.core.JobExecution;  
import org.springframework.batch.core.StepExecution;  
import org.springframework.batch.core.job.flow.FlowExecutionStatus;  
import org.springframework.batch.core.job.flow.JobExecutionDecider;  
  
// Example decider bean  
public class StatusDecider implements JobExecutionDecider {  
  
 private static final Logger log = LoggerFactory.getLogger(StatusDecider.class);  
  
 @Override  
 public FlowExecutionStatus decide(JobExecution jobExecution, StepExecution stepExecution) {  
 // Example logic: Check if any items were written in the previous step  
 long writeCount = stepExecution != null ? stepExecution.getWriteCount() : 0;  
 log.info("Decider checking status after step: {}. Write count: {}", stepExecution != null ? stepExecution.getStepName() : "N/A", writeCount);  
  
 if (writeCount > 0) {  
 return new FlowExecutionStatus("SEND\_NOTIFICATION"); // Custom status matching flow definition  
 } else {  
 return new FlowExecutionStatus("CONTINUE"); // Another custom status  
 }  
 // Can also return FlowExecutionStatus.COMPLETED, FAILED, STOPPED etc.  
 }  
}

**8. Data Model (RecordData.java)**

Java

package com.yourapp.batch.model;  
  
import lombok.Builder;  
import lombok.Data;  
import java.util.Map;  
  
@Data  
@Builder  
public class RecordData {  
 private Long sourceId; // Unique identifier from the source  
 private Map<String, Object> sourceFields; // Raw data from source  
 private ProcessingStatus status;  
 private String errorMessage;  
 private Map<String, Object> mappedData; // Data ready for target table  
}

Java

package com.yourapp.batch.model;  
public enum ProcessingStatus {  
 PENDING, VALIDATED, INVALID, SUCCESS, FAILED  
}

**9. Configuration (application.properties / .yml)**

Ensure DataSources (sourceDataSource, targetDataSource) are configured correctly for Oracle/MSSQL.

Properties

# === Batch Job Configuration ===  
spring.batch.job.enabled=false # Usually run jobs explicitly, not on startup  
# Use 'always' or 'embedded' for dev/testing, 'never' if DB schema managed externally  
spring.batch.initialize-schema=embedded  
  
# === Source Database (Example: Oracle) ===  
spring.datasource.source.jdbc-url=jdbc:oracle:thin:@//your\_oracle\_host:1521/YOUR\_SERVICE  
spring.datasource.source.username=source\_user  
spring.datasource.source.password=source\_password  
spring.datasource.source.driver-class-name=oracle.jdbc.OracleDriver  
# Source Connection Pool Tuning (HikariCP example)  
spring.datasource.source.hikari.maximum-pool-size=15 # Size based on reader threads + overhead  
spring.datasource.source.hikari.minimum-idle=5  
  
# === Target Database (Example: SQL Server) ===  
spring.datasource.target.jdbc-url=jdbc:sqlserver://your\_sql\_server\_host:1433;databaseName=YOUR\_DB;encrypt=true;trustServerCertificate=true; # Adjust security as needed  
spring.datasource.target.username=target\_user  
spring.datasource.target.password=target\_password  
spring.datasource.target.driver-class-name=com.microsoft.sqlserver.jdbc.SQLServerDriver  
# Target Connection Pool Tuning (HikariCP example)  
spring.datasource.target.hikari.maximum-pool-size=20 # Size based on writer threads (throttleLimit) + overhead  
spring.datasource.target.hikari.minimum-idle=5  
  
# === Logging ===  
logging.level.org.springframework.batch=INFO # Set to DEBUG for detailed batch logging  
logging.level.com.yourapp.batch=DEBUG # Set your app's package level  
  
# === Actuator (Optional, for monitoring) ===  
management.endpoints.web.exposure.include=health,info,metrics,batchjobs,batchexecutions,batchsteps  
management.endpoint.health.show-details=always

**Summary of Improvements:**

* **Clear Service Layer:** BatchTriggerService provides a clean API to run jobs with parameters.
* **Dynamic Configuration:** @StepScope allows Reader, Writer, and Processor to be configured dynamically based on JobParameters.
* **Factories/Services for Dependencies:** WorkspaceRepoReaderFactory and MappingService encapsulate specific concerns (reader creation, mapping retrieval), improving SRP and testability.
* **Optimized Parallelism:** Configurable ThreadPoolTaskExecutor and throttleLimit allow fine-tuning concurrent chunk processing.
* **Robustness:** Includes basic error handling, type conversion, and examples of listeners/deciders for more complex flows.
* **SOLID Adherence:** Responsibilities are better separated across classes. Dependencies rely on abstractions.
* **Performance Focus:** Bulk operations, paging reader, parallel execution are maintained and made configurable.
* **Resource Awareness:** @StepScope helps manage component lifecycles; chunking limits memory per thread.

**Next Steps / Further Considerations:**

* **Status Persistence:** Implement a robust strategy (listener updating source/tracking table, or separate step) to persist the SUCCESS/FAILED status currently held only in memory.
* **Dead Letter Queue/Error Handling:** For records failing processing (INVALID) or writing (FAILED), implement a mechanism to log them to a separate table or file for later analysis or reprocessing. Use Spring Batch's SkipListener or RetryListener if using those features.
* **Advanced Partitioning:** If CPU becomes a bottleneck or datasets are enormous, investigate Spring Batch Partitioning (Partitioner, StepExecutionSplitter, worker steps) for greater scalability, potentially across multiple JVMs.
* **Monitoring:** Integrate with monitoring tools (e.g., Prometheus/Grafana via Spring Boot Actuator metrics) to observe job performance, throughput, thread pool usage, and error rates.
* **Testing:** Implement unit tests for Processor, Writer logic and integration tests for the job flow using spring-batch-test.

**Sources**

1. <https://github.com/marecica2/dwh>

2. <https://github.com/Madhurya713/Demo>

3. <https://github.com/prostars/SpringBatchMultiThreadedPartitions>

4. <https://github.com/EBIvariation/eva-tools> subject to licence (Apache - 2.0)

5. <https://github.com/deadlocker8/BudgetMaster> subject to licence (MIT)

6. <https://github.com/spring-projects/spring-batch> subject to licence (Apache - 2.0)

7. <https://github.com/aBreaking/easyjpa>

8. <https://github.com/AdBhati/SpringBatchLatestRelease>

9. <https://github.com/AltheaZeta/Flooring_Mastery>

10. <https://jar-download.com/explore-java-source-code-detail.php?file=./javaSource/org.springframework.batch/spring-batch-infrastructure/3.0.7.RELEASE/org/springframework/batch/item/database/JdbcBatchItemWriter.java&key=3b8cde511b3df55b9433c7c01c9e20ca>

11. <https://github.com/Instagram-Clone-Coding/Spring_instagram-clone> subject to licence (MIT)

12. <https://github.com/openanalytics/phaedra>

13. <https://velog.io/@sdsd0908/Spring-JPA-save-vs-saveAll-%EA%B7%B8%EB%A6%AC%EA%B3%A0-JDBC-Template>