Software Requirements Specification (SRS) for Spring Batch DB Cluster Partitioning

1. Introduction

1.1 Purpose

This Software Requirements Specification (SRS) details the functional and non-functional requirements for the "Spring Batch DB Cluster Partitioning" system. The primary purpose of this system is to enhance Spring Batch's partitioning capabilities by enabling distributed execution of batch jobs across a dynamic cluster of nodes, using a shared relational database for coordination and state management. This system aims to provide scalability, fault tolerance, and efficient workload distribution for large-scale batch processing.

1.2 Scope

The system focuses on the core components required for cluster-aware partitioning in Spring Batch. This includes:

- Dynamic discovery and management of active cluster nodes.
- Strategies for distributing batch job partitions (workloads) among available nodes.
- A mechanism for worker nodes to poll for and execute assigned partitions.
- Aggregation of results from distributed partitions and custom callback handling for overall job status.
- Support for configurable partition transferability in case of node failures.

This SRS does not cover:

- The implementation details of the underlying Spring Batch framework itself.
- Specific business logic of the batch jobs being partitioned.
- Detailed database schema design beyond what's necessary for cluster coordination.
- Deployment and infrastructure automation (e.g., Kubernetes, Docker orchestration).

1.3 Definitions, Acronyms, and Abbreviations

- **SRS**: Software Requirements Specification
- **Spring Batch**: An open-source framework for robust batch processing.
- **Partitioner**: A Spring Batch component responsible for dividing a step's execution into multiple {@link org.springframework.batch.item.ExecutionContext} instances.

- **Partition**: A distinct unit of work created by a Partitioner.
- **Cluster Node**: An individual instance of the Spring Batch application participating in the distributed execution.
- **ExecutionContext**: A Spring Batch object used to store and retrieve data during a step's execution.
- **JobRepository**: A Spring Batch component for persisting job and step execution metadata.
- **JobExplorer**: A Spring Batch component for reading job and step execution metadata.
- **DB**: Database.
- **Lombok**: A Java library that automatically plugs into your build process to 'lombok' your Java files, saving you boilerplate code.
- **Slf4j**: Simple Logging Facade for Java.

1.4 References

- Spring Batch Documentation: https://docs.spring.io/spring-batch/
- Provided GitHub Repository: https://github.com/jchejarla/spring-batch-db-cluster-partitioning/tree/main

1.5 Overview

This document is structured as follows:

- Section 1: Provides an introduction to the system, its purpose, scope, and key definitions.
- Section 2: Describes the overall system, including its perspective, functions, user characteristics, constraints, assumptions, and dependencies.
- Section 3: Details the specific functional and non-functional requirements.
- Section 4: Outlines external interface requirements.

2. Overall Description

2.1 Product Perspective

The Spring Batch DB Cluster Partitioning system is a component designed to integrate with existing Spring Batch applications. It extends Spring Batch's native partitioning capabilities by introducing a database-driven coordination layer, allowing for dynamic scaling and fault tolerance across multiple application instances. It acts as a specialized Partitioner and provides mechanisms for worker nodes to claim and execute assigned work.

2.2 Product Functions

The system provides the following key functions:

- **Cluster Node Management**: Discover and maintain a list of active nodes participating in the cluster via a shared database.
- **Partition Assignment**: Distribute job partitions to available cluster nodes based on configurable strategies.
- **Workload Splitting**: Allow for custom logic to define how the overall batch workload is split into individual partitions.
- **Partition Execution**: Enable worker nodes to poll for and execute partitions assigned to them.
- **Status Management**: Update and track the status of partitions (e.g., CLAIMED, STARTED, COMPLETED, FAILED) in the shared database.
- **Fault Tolerance (Configurable)**: Support for marking partitions as transferable, allowing them to be reassigned to other nodes if the original assigned node fails.
- **Results Aggregation**: Aggregate the results of individual partitioned step executions and provide callbacks for handling overall success or failure.

2.3 User Characteristics

The primary users of this system are:

- **Batch Developers**: Developers who configure and integrate this partitioning solution into their Spring Batch jobs. They need to understand how to implement the abstract methods for workload splitting and strategy building.
- **DevOps/Operations Teams**: Teams responsible for deploying, monitoring, and managing the Spring Batch applications in a clustered environment. They need to ensure database connectivity and monitor node health.

2.4 Constraints

- **Shared Relational Database**: The system relies heavily on a shared relational database for all cluster coordination and state management. This database must be accessible to all cluster nodes.
- **Spring Batch Framework**: The system is built on top of and specifically designed for the Spring Batch framework.
- **Java Environment**: The system is implemented in Java and requires a compatible Java Runtime Environment (JRE).
- **Lombok Dependency**: The project uses Lombok for boilerplate code reduction.
- **Concurrency**: Assumes the underlying database and Spring Batch's JobRepository can handle concurrent updates from multiple nodes.

2.5 Assumptions and Dependencies

- **Database Availability**: It is assumed that the shared relational database is highly

available and performs reliably.
- **Network Connectivity**: All cluster nodes must have stable network connectivity to the shared database.
- **Unique Node Identifiers**: Each cluster node is assumed to have a unique identifier.
- **Spring Context**: Spring application context is properly configured for dependency injection and component scanning.
- **ConditionalOnClusterEnabled**: The system's components are conditionally enabled, implying a configuration property to activate clustering.
- **TaskScheduler and TaskExecutor**: Appropriate Spring task scheduling and execution beans are configured for polling and asynchronous task execution.

3. Specific Requirements

3.1 Functional Requirements

3.1.1 FR1: Cluster-Aware Partitioning

- **Description**: The system shall provide a custom Spring Batch Partitioner (ClusterAwarePartitioner) that replaces the standard partitioning logic.
- **Input**: gridSize (integer, ignored), available cluster nodes from BatchDatabaseClusterService, custom workload chunks from splitIntoChunksForDistribution(), and partitioning strategy from buildPartitionStrategy().
- **Output**: A Map<String, ExecutionContext> where keys are partition identifiers and values are ExecutionContext instances containing the assigned node ID and partition transferability status.
- **Processing**:
  1. Fetch a list of active cluster nodes from the BatchDatabaseClusterService.
  2. If no active nodes are found, throw a BatchConfigurationException.
  3. Call the abstract splitIntoChunksForDistribution() method to get a list of ExecutionContext representing the workload chunks.
  4. Obtain the desired PartitionStrategy using PartitionStrategyFactory based on the PartitionBuilder returned by buildPartitionStrategy().
  5. If buildPartitionStrategy() returns null, default to ROUND_ROBIN partitioning mode.
  6. If FIXED_NODE_COUNT is chosen and fixedNodeCount is 0, default fixedNodeCount to 1.
  7. Assign each ExecutionContext to an available node using the selected PartitionStrategy.
  8. Embed the assigned node ID

(ClusterPartitioningConstants.CLUSTER_NODE_IDENTIFIER) and partition transferability status (ClusterPartitioningConstants.IS_TRANSFERABLE_IDENTIFIER) into each partition's ExecutionContext.

3.1.2 FR2: Configurable Partitioning Strategies

- **Description**: The system shall support different strategies for assigning partitions to nodes.
- **Input**: PartitionBuilder configuration (partitioning mode, fixed node count).
- **Output**: Partitions assigned according to the chosen strategy.
- **Processing**:
  - **FR2.1: Round-Robin Partitioning**:
    - **Description**: Distribute partitions sequentially to available nodes, cycling back to the first node after reaching the last.
    - **Component**: RoundRobinPartitionStrategy.
  - **FR2.2: Fixed Node Count Partitioning**:
    - **Description**: Distribute partitions among a predefined fixed number of available nodes. If fewer nodes are available than the fixed count, use all available nodes.
    - **Component**: FixedNodeCountPartitionStrategy.
  - **FR2.3: Scale-Up Partitioning**:
    - **Description**: Distribute partitions across all currently available nodes, effectively scaling the number of nodes used with the number of partitions. (Note: Current implementation ScaleUpPartitionStrategy appears to be a basic round-robin across *all* available nodes).
    - **Component**: ScaleUpPartitionStrategy.

3.1.3 FR3: Dynamic Workload Splitting

- **Description**: The system shall allow developers to define how the batch job's overall workload is split into individual ExecutionContext chunks.
- **Component**: Abstract method splitIntoChunksForDistribution(int availableNodeCount) in ClusterAwarePartitioner.
- **Processing**: Subclasses must implement this method to return a List<ExecutionContext> representing the work units, typically based on the availableNodeCount or other domain-specific logic.

3.1.4 FR4: Partition Task Polling and Execution

- **Description**: Worker nodes shall periodically poll a shared database for partition tasks assigned to them and execute the corresponding Spring Batch steps.
- **Component**: PartitionWorkerTasksRunner.

- **Processing**:
    1. Upon ApplicationReadyEvent, schedule pollAndExecute() to run at a fixed rate (configured by batchClusterProperties.getPollForPartitionTasks()).
    2. pollAndExecute() shall fetch PartitionAssignmentTask objects assigned to the current node from BatchDatabaseClusterService.
    3. If tasks are found, update their status to "CLAIMED" in the database.
    4. For each claimed task, execute it asynchronously using a TaskExecutor.
    5. During execution:
        - Retrieve JobExecution and StepExecution from JobExplorer.
        - Set StepExecution status to STARTED and update JobRepository.
        - Load the actual Spring Batch Step bean from the ApplicationContext using the master step name.
        - Execute the Step.
        - Update StepExecution status to COMPLETED or FAILED based on execution outcome and update JobRepository.
        - Update the partition task status in the database (BatchDatabaseClusterService) to "COMPLETED" or "FAILED".
    6. Log errors if task execution fails.

3.1.5 FR5: Configurable Partition Transferability

- **Description**: The system shall allow developers to specify whether partitions assigned to a failed node can be transferred and reassigned to other active nodes.
- **Component**: Abstract method arePartitionsTransferableWhenNodeFailed() in ClusterAwarePartitioner.
- **Processing**: Subclasses must implement this method to return a PartitionTransferableProp enum value, which is then stored in the partition's ExecutionContext.

3.1.6 FR6: Aggregation with Custom Callbacks

- **Description**: The system shall aggregate the results of partitioned step executions and provide custom callbacks for handling the overall success or failure of the partitioned job.
- **Component**: ClusterAwareAggregator and ClusterAwareAggregatorCallback.
- **Processing**:
    1. ClusterAwareAggregator extends RemoteStepExecutionAggregator and uses an internal DefaultStepExecutionAggregator for core aggregation.
    2. After aggregation, it checks the BatchStatus of the aggregated StepExecution.

3. If the status is FAILED, it invokes clusterAwareAggregatorCallback.onFailure(executions).
4. Otherwise (e.g., COMPLETED), it invokes clusterAwareAggregatorCallback.onSuccess(executions).

- **Interface**: ClusterAwareAggregatorCallback defines two methods:
  - onSuccess(Collection<StepExecution> executions): Called when all partitions succeed.
  - onFailure(Collection<StepExecution> executions): Called when at least one partition fails.

3.2 Non-Functional Requirements

3.2.1 Performance

- **NFR1.1**: The polling mechanism for partition tasks shall be configurable to avoid excessive database load while ensuring timely task execution. (Addressed by batchClusterProperties.getPollForPartitionTasks()).
- **NFR1.2**: Partition assignment and execution should not introduce significant overhead compared to single-node partitioning for similar workloads.
- **NFR1.3**: Database operations for cluster coordination (node registration, task claiming, status updates) should be optimized for performance.

3.2.2 Scalability

- **NFR2.1**: The system shall support scaling out by adding more cluster nodes to handle increased workload.
- **NFR2.2**: The partitioning strategies should effectively distribute workload across an increasing number of nodes.
- **NFR2.3**: The shared database must be able to handle the increased load from multiple cluster nodes performing read/write operations for coordination.

3.2.3 Reliability

- **NFR3.1**: The system shall gracefully handle scenarios where active nodes are not found in the database (e.g., during initial startup or database connectivity issues).
- **NFR3.2**: Partition execution failures on a worker node should be properly captured and reflected in the database, leading to the overall job status being marked as FAILED.
- **NFR3.3**: If partitions are marked as transferable, the system should ideally support reassigning failed or unexecuted partitions from a crashed node to another active node (though the re-assignment logic itself is not explicitly shown in the provided code, the IS_TRANSFERABLE_IDENTIFIER suggests this capability).

3.2.4 Maintainability

- **NFR4.1**: The codebase shall be well-commented and follow standard Java coding conventions. (Addressed by Javadoc generation).
- **NFR4.2**: The system's components should be modular and loosely coupled to facilitate future enhancements and bug fixes.
- **NFR4.3**: Configuration for clustering parameters (e.g., polling interval) should be externalized and easily manageable.

3.2.5 Portability

- **NFR5.1**: The system shall be portable across different relational databases supported by Spring Batch's JobRepository (e.g., H2, MySQL, PostgreSQL, Oracle) as it relies on standard JDBC operations via BatchDatabaseClusterService.

4. External Interface Requirements

4.1 User Interfaces

- There is no direct end-user graphical interface for this system.
- Interaction is primarily through Spring Batch job configurations and custom Java code implementations.
- Logging (Slf4j) provides visibility into the system's operations and status for developers and operations teams.

4.2 Hardware Interfaces

- No specific hardware interfaces are required beyond a standard server environment capable of running Java applications and connecting to a relational database.

4.3 Software Interfaces

- **Spring Batch Framework**: Core dependency for partitioning, job execution, and metadata management.
- **Spring Framework**: For dependency injection, event handling, and task scheduling.
- **Relational Database**: Any RDBMS compatible with Spring Batch's JobRepository for cluster coordination.
- **Lombok**: Compile-time dependency for code generation.
- **Logging Framework**: Slf4j API with an underlying implementation (e.g., Logback, Log4j2).

4.4 Communications Interfaces

- **JDBC**: Standard Java Database Connectivity for communication between cluster nodes and the shared relational database.
- **Internal JVM Communication**: For TaskExecutor and TaskScheduler within a single node.