The Journal of Open Source Software

spring-batch-db-cluster-partitioning: Database-driven clustering with heartbeats and failover for Spring Batch

Janardhan Reddy Chejarla [1]

1 Independent Researcher, United States

DOI: 10.21105/joss.09460

Software

- Review
- Repository
- Archive

Editor: Mark A. Jensen

Reviewers:

- @david-guzman
- @majensen

Submitted: 23 August 2025
Published: 08 June 2026

Summary

Batch processing of large datasets — transforming, validating, or migrating millions of records in a single run — is a routine requirement across scientific and enterprise computing, with well-documented use in bioinformatics and clinical informatics (Wolstencroft et al., 2013), in the Earth and climate sciences (Cinquini et al., 2014), and in financial services (Cogoluègnes et al., 2011). Spring Batch (*Spring Batch Project*, 2025) is a widely used Java framework for these workloads, providing transactional chunk processing, restart semantics, and a rich programming model. When a single machine cannot process the data fast enough, Spring Batch supports *remote partitioning*: the work is split into partitions and dispatched to worker nodes through a messaging layer such as RabbitMQ or Apache Kafka (*Remote Partitioning with Spring Batch and Spring Integration*, 2025).

However, operating a highly available message broker solely for partition coordination adds significant infrastructure overhead — deployment, monitoring, and failure handling for a component that is incidental to the batch workload itself. More critically, the message-based model provides no built-in mechanism for the master node to discover how many workers are actually available, or to detect and recover from a worker crash after a partition has been dispatched.

`spring-batch-db-cluster-partitioning` is an open-source Java extension that replaces the messaging layer with the relational database that Spring Batch already requires. It introduces three capabilities absent from the standard remote partitioning model: (1) **proactive node awareness**, allowing the master to query exactly how many workers are active before distributing work; (2) **explicit partition lifecycle tracking**, recording each partition's state (`PENDING`, `CLAIMED`, `COMPLETED`, `FAILED`) transactionally so that no work is silently lost; and (3) **automatic failover**, detecting unresponsive nodes via heartbeat timeouts and reassigning their incomplete partitions to healthy workers without human intervention. By persisting all coordination state in standard SQL tables, the extension eliminates the operational burden of a message broker while simultaneously providing real-time visibility into job progress through ordinary SQL queries.

Statement of Need

Spring Batch is used in research-adjacent and enterprise contexts where reproducible, auditable processing of large datasets is essential. In bioinformatics, workflow-based pipelines process sequencing data and clinical datasets at scale (Wolstencroft et al., 2013). In the Earth and climate sciences, federated infrastructures coordinate the ingestion and processing of model output across petabyte-scale archives (Cinquini et al., 2014). In financial services, batch processing underpins end-of-day reconciliation, payment processing, and regulatory reporting (Cogoluègnes et al., 2011). In each case, horizontal scale-out — distributing partitions across

multiple worker nodes — is a common strategy for meeting processing-time targets.

Spring Batch offers two parallelism models (*Spring Batch*, 2025): local multi-threaded partitioning within a single JVM, and remote partitioning across multiple JVMs via a messaging middleware (*Remote Partitioning with Spring Batch and Spring Integration*, 2025). Local partitioning is bounded by the resources of a single machine. Remote partitioning removes this ceiling but introduces a critical operational challenge: the master node dispatches partitions through a message queue and has no direct knowledge of how many workers are active, or whether a worker that received a partition has completed, failed, or silently died. If a worker crashes after claiming a partition but before acknowledging completion, the master has no reliable mechanism to detect this and reassign the work. Timeout-based recovery requires careful tuning and still introduces indefinite blocking during failure scenarios (Tanenbaum & Van Steen, 2007).

This extension addresses these gaps by using the relational database as the coordination plane. The design produces several concrete benefits. First, topology visibility: the master queries a `BATCH_NODES` table before partitioning, so it knows the exact number of active workers and can distribute work proportionally — enabling partition counts that scale with the available cluster size at runtime. Second, guaranteed state durability: every partition state transition is committed transactionally, so a worker crash leaves a recoverable record rather than a lost message. Third, simplified architecture: teams already operating a production database incur no additional infrastructure to coordinate batch workers. Fourth, operational observability: standard SQL queries expose live job status, partition progress, and node health without specialised tooling.

The performance benefit is direct: a job that takes 60 minutes on a single node can complete in approximately 15 minutes across four workers using round-robin distribution, with near-linear speedup for I/O-bound workloads. The extension adds only lightweight SQL operations (heartbeat writes, partition claim updates) whose overhead is negligible relative to the batch workload itself.

The target users are Java development teams running Spring Batch workloads that require horizontal scale-out or fault-tolerant execution, particularly in environments where introducing a message broker is undesirable due to operational constraints, compliance requirements, or infrastructure simplicity goals. A Spring-centric implementation was chosen deliberately: this extension uses Spring Batch's native partitioning API, requiring no changes to existing job or step definitions. Spring Batch has been a complete implementation of JSR-352 (Jakarta Batch) (Java Community Process, 2013) since version 3.0, and the JSR-352 specification defines closely analogous partitioning artifacts (`PartitionMapper`, `PartitionAnalyzer`, `PartitionReducer`) that mirror Spring Batch's `Partitioner` and aggregator. The coordination mechanism (message broker vs. shared database) is orthogonal to the framework API surface, and the database-driven approach described here could be re-implemented against the JSR-352 partitioning artifacts with the same semantics, in line with the broader Java ecosystem's emphasis on specification-driven interoperability.

Software Design

The extension is organised around three coordination tables that augment Spring Batch's existing `JobRepository` schema. The `BATCH_NODES` table records every active cluster node alongside a periodically refreshed heartbeat timestamp. The `BATCH_PARTITIONS` table tracks each partition's lifecycle — from `PENDING` through `CLAIMED` to `COMPLETED` or `FAILED` — with transactional state transitions that prevent double execution. The `BATCH_JOB_COORDINATION` table binds a specific job execution to its master node and partitioned step, providing the metadata needed for aggregation and completion detection.

When a partitioned step starts, the master node queries `BATCH_NODES` to discover currently active workers. It then invokes the user-supplied `ClusterAwarePartitioner`, passing the

live worker count so that the number of partitions can be tailored to the available cluster size. Partitions are assigned to workers using a configurable strategy — round-robin for even distribution, fixed-node-count for controlled parallelism, or scale-up for dynamic workloads — and persisted to `BATCH_PARTITIONS` with a `PENDING` status. Each worker node runs a polling loop that claims its assigned partitions transactionally, executes the corresponding Spring Batch step, and updates the partition status upon completion or failure.

Failure detection uses a two-phase protocol. Nodes that miss heartbeat updates beyond a configurable threshold are first marked `UNREACHABLE` — a transient state that accommodates temporary network hiccups or garbage-collection pauses. Only after a longer cleanup threshold does the system remove the node entry and make its incomplete partitions eligible for reassignment. This separation reduces false positives and avoids unnecessary partition transfers. Partitions marked as transferable are reassigned to healthy nodes, while non-transferable partitions are left in their failed state for manual investigation — giving operators control over the recovery policy.

The architecture diagram below illustrates the coordination flow:

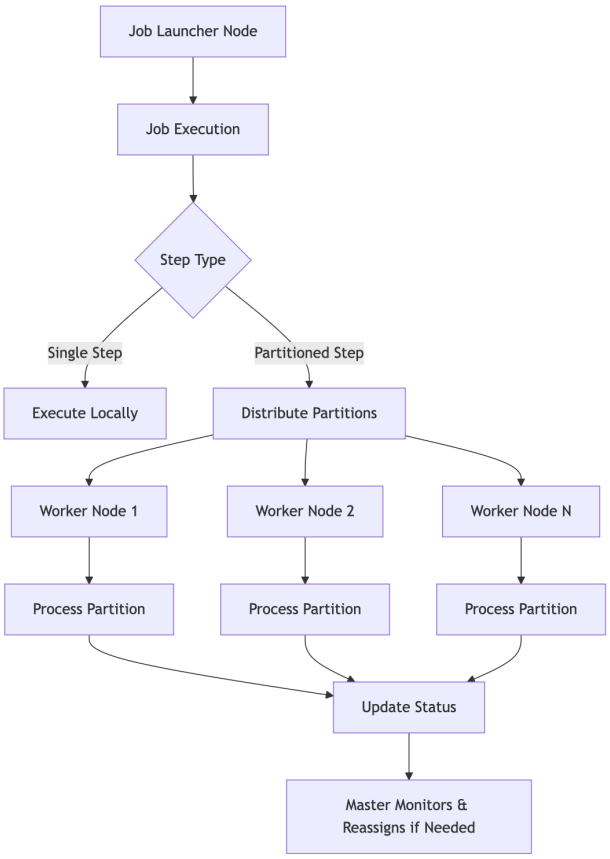

Figure 1: Coordination flow between the job launcher, shared database, and worker nodes.

State of the Field

Scientific workflow systems such as Pegasus (Deelman et al., 2015), Kepler (Ludäscher et al., 2006), and Taverna (Wolstencroft et al., 2013) provide sophisticated scheduling and provenance tracking for computational pipelines, emphasising reproducibility and durable state. These systems typically operate at the workflow level — orchestrating entire pipelines of heterogeneous tasks — rather than at the intra-step level of a single batch job. Spring Batch

occupies a different niche: it is a lightweight, embeddable framework for transactional chunk processing within Java applications (*Spring Batch Project*, 2025). External orchestrators such as Spring Cloud Data Flow (*Spring Cloud Data Flow*, 2025) can manage Spring Batch jobs at the deployment level, but they do not address intra-job partition coordination or worker-level failover.

Distributed coordination services like ZooKeeper (Hunt et al., 2010), Chubby (Burrows, 2006), and consensus protocols like Raft (Ongaro & Ousterhout, 2014) provide general-purpose primitives (leader election, distributed locks, configuration management) that could theoretically underpin a partitioning solution. However, adopting these services introduces the same operational complexity that this extension seeks to avoid: additional infrastructure to deploy, monitor, and secure. By embedding coordination directly in the relational database — a component already present in every Spring Batch deployment — this extension provides a pragmatic middle ground between single-node execution and full distributed-systems infrastructure.

Project Maturity

The extension has been published to Maven Central since version 1.0.0 and is currently at version 2.0.0, which introduced the scale-up partition strategy, actuator health endpoints, and multi-database support (PostgreSQL, MySQL, Oracle, H2). The repository includes a comprehensive unit and integration test suite executed on every push and pull request via GitHub Actions CI. A companion `examples/` module provides a runnable demonstration application with Docker-based database setup, enabling users to validate the core claims — partition distribution, failover, and restart — on a local machine within minutes. Documentation, issue templates, and a contributor guide are maintained in the repository to facilitate external participation. The author is committed to long-term maintenance and actively encourages community contributions through the issue tracker and pull requests.

Research Impact

By removing the requirement for messaging infrastructure, this extension lowers the barrier to running distributed batch workloads — particularly in academic and research settings where infrastructure resources are constrained. Research teams processing large datasets — for example, genomic analysis pipelines that motivate scientific workflow systems such as Taverna (Wolstencroft et al., 2013) — can achieve horizontal scale-out using only a relational database they already operate, without provisioning additional middleware. The transparent, SQL-queryable coordination state also supports reproducibility: every partition assignment, state transition, and failover event is durably recorded and can be audited after the fact.

Limitations

The extension targets small-to-medium clusters — typically two to approximately twenty nodes — where database throughput is sufficient for heartbeat and partition-tracking queries. Deployments at this scale represent the vast majority of Spring Batch production use cases. For clusters exceeding this range, or workloads requiring sub-second coordination latency, a dedicated coordination service (e.g., ZooKeeper) or a message-broker-based architecture may be more appropriate. The extension also relies on the availability of the shared database; production deployments should use a replicated database with appropriate backup and failover configuration to avoid a single point of failure.

AI Usage Disclosure

The author confirms that no generative AI or AI-assisted technologies (such as LLMs, automated coding assistants, or AI-based image generators) were used in the development of the underlying software code or the preparation of this manuscript. # Acknowledgements

This work builds on the Spring Batch and Spring Boot frameworks. The author thanks the JOSS reviewers for their constructive feedback.

Source code: Chejarla (2025).

References

Burrows, M. (2006). The Chubby lock service for loosely-coupled distributed systems. *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://research.google/pubs/pub27897/

Chejarla, J. R. (2025). *Spring-batch-db-cluster-partitioning*. https://github.com/jchejarla/spring-batch-db-cluster-partitioning.

Cinquini, L., Crichton, D., Mattmann, C., Harney, J., Shipman, G., Wang, F., Ananthakrishnan, R., Miller, N., Denvil, S., Morgan, M., Pobre, Z., Bell, G. M., Doutriaux, C., Drach, R., Williams, D., Kershaw, P., Pascoe, S., Gonzalez, E., Fiore, S., & Schweitzer, R. (2014). The Earth System Grid Federation: An open infrastructure for access to distributed geospatial data. *Future Generation Computer Systems*, *36*, 400–417. https://doi.org/10.1016/j.future.2013.07.002

Cogoluègnes, A., Templier, T., Bazoud, G., & Gregory, O. (2011). *Spring Batch in action*. Manning Publications. ISBN: 978-1935182955

Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., Ferreira da Silva, R., Livny, M., & Wenger, K. (2015). Pegasus: A workflow management system for science automation. *Future Generation Computer Systems*, *46*, 17–35. https://doi.org/10.1016/j.future.2014.10.008

Hunt, P., Konar, M., Junqueira, F. P., & Reed, B. (2010). ZooKeeper: Wait-free coordination for internet-scale systems. *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/legacy/event/atc10/tech/full_papers/Hunt.pdf

Java Community Process. (2013). *JSR-352: Batch applications for the Java platform*. https://jcp.org/en/jsr/detail?id=352.

Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., & Zhao, Y. (2006). Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, *18*(10), 1039–1065. https://doi.org/10.1002/cpe.994

Ongaro, D., & Ousterhout, J. (2014). In search of an understandable consensus algorithm (Raft). *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf

Remote partitioning with spring batch and spring integration. (2025). https://docs.spring.io/spring-batch/reference/partitioning.html.

Spring batch project. (2025). https://spring.io/projects/spring-batch.

Spring batch: Scaling and parallel processing. (2025). https://docs.spring.io/spring-batch/reference/scalability.html.

Spring cloud data flow. (2025). https://dataflow.spring.io/.

Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed systems: Principles and paradigms* (2nd ed.). Pearson Prentice Hall. ISBN: 978-0132392273

Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P., Bhagat, J., Belhajjame, K., Hardisty, A., Hidalga, A. N. de la, Vargas, M. R., & Goble, C. (2013). The Taverna workflow suite: Designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, *41*(W1), W557–W561. https://doi.org/10.1093/nar/gkt328