I am developing spring batch using version 5. Deployed 7 services in azure kubernetes cluster. Backed dB is azure sql managed instance.

When running jobs concurrently, I am getting deadloclloseraccessexception. While inserting record to batch\_job\_instance

Please list all changes needed to fix the issue

To address the DeadlockLoserDataAccessException when inserting records into the BATCH\_JOB\_INSTANCE table in Spring Batch 5, deployed on an Azure Kubernetes cluster with Azure SQL Managed Instance as the backend database, you need to implement a combination of configuration changes, database optimizations, and application-level adjustments. Below is a comprehensive list of changes to fix the deadlock issue, based on best practices and insights from similar issues reported in Spring Batch with SQL Server databases.

1. Use SQL Server Sequences Instead of Identity Columns Issue: Spring Batch’s default SqlServerMaxValueIncrementer for SQL Server uses identity columns or manual increment logic for sequence tables (BATCH\_JOB\_SEQ, etc.), which can cause deadlocks due to concurrent inserts and deletes on these tables. SQL Server supports native sequences since version 2012, which are more efficient for high-concurrency scenarios. Change: Configure Spring Batch to use SQL Server sequences for generating IDs in the metadata tables. Step 1: Create Sequences in the Database Modify the schema to use sequences instead of identity columns for BATCH\_JOB\_SEQ, BATCH\_JOB\_EXECUTION\_SEQ, and BATCH\_STEP\_EXECUTION\_SEQ. Run the following SQL on your Azure SQL Managed Instance: sql

Collapse

Wrap

Copy CREATE SEQUENCE BATCH\_JOB\_SEQ START WITH 1 INCREMENT BY 1; CREATE SEQUENCE BATCH\_JOB\_EXECUTION\_SEQ START WITH 1 INCREMENT BY 1; CREATE SEQUENCE BATCH\_STEP\_EXECUTION\_SEQ START WITH 1 INCREMENT BY 1; Ensure the schema aligns with Spring Batch 5’s expectations (refer to the official schema scripts for SQL Server in Spring Batch documentation). Step 2: Configure Custom Incrementer Override the default incrementer factory to use a sequence-based incrementer for SQL Server. Create a custom DataFieldMaxValueIncrementerFactory: java

Collapse

Wrap

Copy import org.springframework.batch.core.repository.dao.JdbcJobInstanceDao; import org.springframework.jdbc.support.incrementer.DataFieldMaxValueIncrementer; import org.springframework.jdbc.support.incrementer.SqlServerMaxValueIncrementer; import javax.sql.DataSource;

public class CustomDataFieldMaxValueIncrementerFactory extends DefaultDataFieldMaxValueIncrementerFactory { private final DataSource dataSource;

public CustomDataFieldMaxValueIncrementerFactory(DataSource dataSource) { super(dataSource); this.dataSource = dataSource; }

@Override public DataFieldMaxValueIncrementer getIncrementer(String incrementerType, String incrementerName) { if (DatabaseType.SQLSERVER.name().equalsIgnoreCase(incrementerType)) { // Use sequence instead of identity return new SqlServerMaxValueIncrementer(dataSource, incrementerName) { @Override protected String getIncrementStatement() { return "SELECT NEXT VALUE FOR " + getIncrementerName(); } }; } return super.getIncrementer(incrementerType, incrementerName); } } Step 3: Configure JobRepository Update your Spring Batch configuration to use the custom incrementer factory: java

Collapse

Wrap

Copy import org.springframework.batch.core.configuration.annotation.DefaultBatchConfigurer; import org.springframework.batch.core.repository.JobRepository; import org.springframework.batch.core.repository.support.JobRepositoryFactoryBean; import org.springframework.beans.factory.annotation.Value; import org.springframework.context.annotation.Configuration; import javax.sql.DataSource;

@Configuration public class BatchConfiguration extends DefaultBatchConfigurer { private final DataSource dataSource; @Value("${spring.batch.table-prefix:}") private String tablePrefix;

public BatchConfiguration(DataSource dataSource) { this.dataSource = dataSource; }

@Override protected JobRepository createJobRepository() throws Exception { JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean(); factory.setDatabaseType("SQLSERVER"); factory.setDataSource(dataSource); factory.setTransactionManager(getTransactionManager()); factory.setTablePrefix(tablePrefix); factory.setIncrementerFactory(new CustomDataFieldMaxValueIncrementerFactory(dataSource)); factory.setIsolationLevelForCreate("ISOLATION\_REPEATABLE\_READ"); factory.afterPropertiesSet(); return factory.getObject(); } } Why This Helps: Sequences are designed for high-concurrency environments and reduce contention compared to identity column increments, which involve table locks. 2. Adjust Transaction Isolation Level Issue: The default transaction isolation level (SERIALIZABLE) in Spring Batch for job creation can lead to excessive locking, causing deadlocks when multiple jobs try to insert into BATCH\_JOB\_INSTANCE concurrently. Change: Set the isolation level to REPEATABLE\_READ for job creation operations. Implementation: As shown in the JobRepository configuration above, set: java

Collapse

Wrap

Copy factory.setIsolationLevelForCreate("ISOLATION\_REPEATABLE\_READ"); Why This Helps: REPEATABLE\_READ reduces the scope of locks compared to SERIALIZABLE, allowing concurrent reads while preventing dirty reads and non-repeatable reads, thus reducing deadlock likelihood. 3. Add Primary Key Constraints to Sequence Tables Issue: Missing or improper indexes on sequence tables (BATCH\_JOB\_SEQ, etc.) can lead to table scans or inefficient locking, contributing to deadlocks. Change: Ensure primary key constraints are defined on the sequence tables. Implementation: Run the following SQL to add primary keys: sql

Collapse

Wrap

Copy ALTER TABLE BATCH\_JOB\_SEQ ADD CONSTRAINT PK\_BATCH\_JOB\_SEQ PRIMARY KEY CLUSTERED (ID ASC); ALTER TABLE BATCH\_JOB\_EXECUTION\_SEQ ADD CONSTRAINT PK\_BATCH\_JOB\_EXECUTION\_SEQ PRIMARY KEY CLUSTERED (ID ASC); ALTER TABLE BATCH\_STEP\_EXECUTION\_SEQ ADD CONSTRAINT PK\_BATCH\_STEP\_EXECUTION\_SEQ PRIMARY KEY CLUSTERED (ID ASC); Why This Helps: Primary keys ensure efficient index seeks instead of scans, reducing lock contention. 4. Implement Optimistic Locking Issue: Concurrent updates to the same metadata tables can cause deadlocks if pessimistic locking is used. Change: Enable optimistic locking for Spring Batch metadata tables by adding a version column and configuring Spring Batch to use it. Step 1: Add Version Columns Modify the schema to include a VERSION column in BATCH\_JOB\_INSTANCE, BATCH\_JOB\_EXECUTION, and BATCH\_STEP\_EXECUTION if not already present: sql

Collapse

Wrap

Copy ALTER TABLE BATCH\_JOB\_INSTANCE ADD VERSION BIGINT DEFAULT 0; ALTER TABLE BATCH\_JOB\_EXECUTION ADD VERSION BIGINT DEFAULT 0; ALTER TABLE BATCH\_STEP\_EXECUTION ADD VERSION BIGINT DEFAULT 0; Step 2: Configure Spring Batch Spring Batch 5 supports optimistic locking by default if the VERSION column exists. Ensure your JobRepository is configured to use the default JdbcJobInstanceDao, which handles versioning. Step 3: Add Retry Logic Use Spring Retry to automatically retry transactions that fail due to optimistic locking exceptions: java

Collapse

Wrap

Copy import org.springframework.retry.annotation.Retryable; import org.springframework.stereotype.Component; import org.springframework.batch.core.Job; import org.springframework.batch.core.launch.JobLauncher; import org.springframework.dao.OptimisticLockingFailureException;

@Component public class JobRunner { private final JobLauncher jobLauncher; private final Job job;

public JobRunner(JobLauncher jobLauncher, Job job) { this.jobLauncher = jobLauncher; this.job = job; }

@Retryable(value = {OptimisticLockingFailureException.class}, maxAttempts = 3) public void runJob(JobParameters params) throws Exception { jobLauncher.run(job, params); } } Add the @EnableRetry annotation to your configuration: java

Collapse

Wrap

Copy import org.springframework.retry.annotation.EnableRetry; import org.springframework.context.annotation.Configuration;

@Configuration @EnableRetry public class RetryConfiguration { } Why This Helps: Optimistic locking reduces lock contention by checking for version conflicts at commit time, and retry logic handles transient failures gracefully. 5. Ensure Unique Job Parameters Issue: Running multiple instances of the same job without unique parameters can cause contention on the BATCH\_JOB\_INSTANCE table, as Spring Batch checks for existing job instances based on job name and parameters. Change: Use a RunIdIncrementer to append a unique run ID to each job execution. Implementation: Configure your job with a JobParametersIncrementer: java

Collapse

Wrap

Copy import org.springframework.batch.core.Job; import org.springframework.batch.core.configuration.annotation.JobBuilderFactory; import org.springframework.batch.core.launch.support.RunIdIncrementer; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration;

@Configuration public class JobConfiguration { private final JobBuilderFactory jobBuilderFactory;

public JobConfiguration(JobBuilderFactory jobBuilderFactory) { this.jobBuilderFactory = jobBuilderFactory; }

@Bean public Job myJob() { return jobBuilderFactory.get("myJob") .incrementer(new RunIdIncrementer()) .flow(step1()) .end() .build(); } } Why This Helps: Unique job parameters prevent duplicate job instance checks from causing contention, reducing the likelihood of deadlocks. 6. Optimize Database Connection Pool Issue: Improperly configured connection pools can exacerbate deadlocks by holding connections longer than necessary, increasing transaction duration. Change: Use a connection pool like HikariCP and tune its settings. Implementation: Configure HikariCP in your application.properties: properties

Collapse

Wrap

Copy spring.datasource.hikari.maximum-pool-size=10 spring.datasource.hikari.minimum-idle=5 spring.datasource.hikari.idle-timeout=30000 spring.datasource.hikari.connection-timeout=20000 spring.datasource.hikari.max-lifetime=1800000 Why This Helps: Efficient connection management reduces transaction duration, minimizing lock hold times and deadlock risks. 7. Enable Read-Committed Snapshot Isolation (Optional) Issue: Azure SQL Managed Instance defaults to READ COMMITTED, but enabling READ\_COMMITTED\_SNAPSHOT can reduce locking conflicts by using row versioning. Change: Enable READ\_COMMITTED\_SNAPSHOT on the database. Implementation: Run the following SQL as an admin: sql

Collapse

Wrap

Copy ALTER DATABASE YourDatabaseName SET READ\_COMMITTED\_SNAPSHOT ON; Considerations: This increases tempdb usage due to row versioning. Monitor tempdb performance and size. Test thoroughly, as it may not be necessary if other changes resolve the issue. Why This Helps: Row versioning reduces blocking for read operations, potentially alleviating deadlock scenarios. 8. Monitor and Analyze Deadlocks Issue: Without proper monitoring, it’s hard to confirm the root cause or verify fixes. Change: Enable deadlock monitoring using Azure SQL Managed Instance’s extended events. Implementation: Create an extended event session to capture deadlock graphs: sql

Collapse

Wrap

Copy CREATE EVENT SESSION DeadlockCapture ON SERVER ADD EVENT sqlserver.xml\_deadlock\_report ADD TARGET package0.event\_file ( SET filename = 'DeadlockCapture.xel', max\_file\_size = 10, max\_rollover\_files = 5 ) WITH ( MAX\_MEMORY = 4096 KB, EVENT\_RETENTION\_MODE = ALLOW\_SINGLE\_EVENT\_LOSS, MAX\_DISPATCH\_LATENCY = 30 SECONDS, TRACK\_CAUSALITY = ON ); ALTER EVENT SESSION DeadlockCapture ON SERVER STATE = START; Query the deadlock events: sql

Collapse

Wrap

Copy SELECT event\_data.value('(event/@name)[1]', 'varchar(50)') AS event\_name, event\_data.value('(event/@timestamp)[1]', 'datetime') AS event\_time, event\_data.query('(event/data/value/deadlock)[1]') AS deadlock\_graph FROM sys.fn\_xe\_file\_target\_read\_file('DeadlockCapture\*.xel', NULL, NULL, NULL) CROSS APPLY (SELECT CAST(event\_data AS XML) AS event\_data) AS event\_data; Why This Helps: Deadlock graphs provide detailed insights into which queries and resources are involved, helping validate fixes and identify any remaining issues. 9. Partition Job Execution (Optional) Issue: Running seven concurrent jobs may overwhelm the database if they all access metadata tables simultaneously. Change: Use Spring Batch partitioning to split jobs into smaller, independent chunks. Implementation: Configure a Partitioner to divide job data: java

Collapse

Wrap

Copy import org.springframework.batch.core.partition.support.Partitioner; import org.springframework.batch.item.ExecutionContext; import java.util.HashMap; import java.util.Map;

public class SimplePartitioner implements Partitioner { @Override public Map partition(int gridSize) { Map partitions = new HashMap<>(); for (int i = 0; i < gridSize; i++) { ExecutionContext context = new ExecutionContext(); context.putInt("partitionId", i); partitions.put("partition" + i, context); } return partitions; } } Configure the job to use partitioning: java

Collapse

Wrap

Copy @Bean public Job partitionedJob() { return jobBuilderFactory.get("partitionedJob") .incrementer(new RunIdIncrementer()) .flow(masterStep()) .end() .build(); }

@Bean public Step masterStep() { return stepBuilderFactory.get("masterStep") .partitioner(slaveStep().getName(), partitioner()) .step(slaveStep()) .gridSize(4) .taskExecutor(taskExecutor()) .build(); }

@Bean public Partitioner partitioner() { return new SimplePartitioner(); }

@Bean public TaskExecutor taskExecutor() { ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor(); executor.setCorePoolSize(4); executor.setMaxPoolSize(4); executor.setThreadNamePrefix("partition-thread-"); executor.initialize(); return executor; } Why This Helps: Partitioning reduces contention by spreading job execution across smaller, independent tasks, minimizing simultaneous access to metadata tables. 10. Test and Validate Change: After applying the above changes, simulate concurrent job execution in a staging environment using tools like Gatling or JMeter to replicate the production load. Implementation: Trigger all seven jobs simultaneously with unique parameters. Monitor deadlock occurrences using the extended event session. Check application logs for DeadlockLoserDataAccessException or OptimisticLockingFailureException. Why This Helps: Validates that the changes resolve the deadlock issue and ensures no new issues are introduced. Notes: Spring Batch 5 Specifics: Spring Batch 5 includes improvements for concurrency, but the deadlock issue with SQL Server persists from earlier versions due to the default incrementer behavior. The sequence-based approach is the most reliable fix. Azure SQL Managed Instance: Ensure the database is on a Business Critical tier for better performance under concurrent workloads. Monitor tempdb if enabling READ\_COMMITTED\_SNAPSHOT. Prioritize Changes: Start with steps 1–5, as they address the core issues (sequences, isolation level, indexes, optimistic locking, and unique parameters). Steps 6–9 are optimizations or situational fixes. Documentation: Refer to the Spring Batch 5 documentation for schema details and configuration options: https://docs.spring.io/spring-batch/docs/5.0.x/reference/html/index.html By implementing these changes, you should eliminate or significantly reduce deadlocks when running concurrent Spring Batch jobs on Azure SQL Managed Instance. If issues persist, analyze the deadlock graphs from the extended event session to identify any application-specific query patterns causing contention.