Deadlocks during concurrent job instance creation in Spring Batch with a SQL Server backend are a common issue, but you can definitely fix it.

The root cause is typically related to how SQL Server handles locking during the check-and-insert operation within the JobRepository. When multiple jobs start at the same time, they all check if a JobInstance with the same parameters exists. If not, they all try to insert a new record, leading to competing locks and a deadlock.

Here’s how to fix it by adjusting the transaction isolation level for the JobRepository.

### **Solution: Set the Isolation Level to ISOLATION\_READ\_COMMITTED\_SNAPSHOT**

For SQL Server, the recommended approach is to change the transaction isolation level for the JobRepository to READ\_COMMITTED\_SNAPSHOT. This is a non-standard isolation level specific to SQL Server that avoids holding shared locks on read operations, preventing them from blocking write operations.

You can achieve this by customizing the JobRepository bean in your Spring Batch configuration.

#### **1. Enable Snapshot Isolation in SQL Server**

First, ensure that READ\_COMMITTED\_SNAPSHOT is enabled on your SQL Server database. You'll need a database administrator to run this command if you don't have the permissions:

ALTER DATABASE [your\_database\_name] SET READ\_COMMITTED\_SNAPSHOT ON;

SQL

ALTER DATABASE [your\_database\_name] SET READ\_COMMITTED\_SNAPSHOT ON;

#### **2. Customize the JobRepository Configuration**

Next, you need to tell Spring Batch to use this specific isolation level for the JobRepository's transactions. The way you do this depends on whether you are using a Java-based configuration with DefaultBatchConfiguration or a custom one.

Here is the recommended approach using Java configuration in Spring Batch 5:

Java

import org.springframework.batch.core.repository.support.JobRepositoryFactoryBean;

import org.springframework.batch.core.repository.JobRepository;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.jdbc.support.JdbcTransactionManager;

import javax.sql.DataSource;

@Configuration

public class BatchConfig {

private final DataSource dataSource;

public BatchConfig(DataSource dataSource) {

this.dataSource = dataSource;

}

@Bean

public JobRepository jobRepository() throws Exception {

JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();

factory.setDataSource(dataSource);

// This is the key change for SQL Server

factory.setTransactionManager(new JdbcTransactionManager(dataSource));

factory.setIsolationLevelForCreate("ISOLATION\_READ\_COMMITTED\_SNAPSHOT");

factory.setTablePrefix("BATCH\_"); // Ensure this matches your setup

factory.afterPropertiesSet();

return factory.getObject();

}

}

**Key Points in the Code:**

* **@Configuration**: This class defines your Spring Batch configuration beans.
* **JobRepositoryFactoryBean**: This factory bean is the standard way to create a configured JobRepository.
* **setDataSource(...)**: We provide the DataSource for your batch metadata.
* **setTransactionManager(...)**: We explicitly set a JdbcTransactionManager.
* **setIsolationLevelForCreate("ISOLATION\_READ\_COMMITTED\_SNAPSHOT")**: This is the crucial line. It tells Spring Batch to use the READ\_COMMITTED\_SNAPSHOT isolation level only for the transaction that creates the job instance. This efficiently prevents the race condition that causes the deadlock.
* **getObject()**: This returns the configured JobRepository instance that the rest of your batch application will use.

By applying this configuration, you specifically target the operation known to cause deadlocks without affecting the transaction settings for your entire application. Your concurrent jobs should now be able to create their respective JobInstance records without deadlocking

The default transaction isolation level in SQL Server is **READ COMMITTED**.

Yes, it is generally safe to run the command ALTER DATABASE [your\_database\_name] SET READ\_COMMITTED\_SNAPSHOT ON;. This command enables a feature that changes how the default READ COMMITTED isolation level works. Instead of using shared locks for reading data, it uses a row versioning mechanism.

Here are the pros and cons of enabling READ\_COMMITTED\_SNAPSHOT.

### **Pros (Advantages) ✅**

* **Reduces Blocking and Deadlocks**: This is the biggest advantage. Read operations no longer acquire shared locks on data. This means readers don't block writers, and writers don't block readers. This significantly improves concurrency, especially in high-traffic applications like the Spring Batch scenario you mentioned.
* **Improved Performance**: By minimizing lock contention, overall database throughput and application responsiveness can increase dramatically.
* **Data Consistency**: Readers get the last committed version of the data that existed when their transaction started, ensuring they work with a consistent snapshot without blocking other processes.

### **Cons (Disadvantages) ⚠️**

* **Increased tempdb Usage**: Row versioning requires storing the old versions of rows in the tempdb database. If you have many long-running transactions or frequent updates, your tempdb can grow significantly. You must monitor tempdb size and performance.
* **Slight Overhead on Updates**: When a row is updated, its previous version must be copied to tempdb. This adds a small amount of overhead to all data modification operations (UPDATE, DELETE).
* **Potential for Optimistic Concurrency Errors**: Since readers don't lock data, it's possible for a transaction to read a value, and then have another transaction change that value before the first transaction completes. If the first transaction then tries to update the row based on the value it initially read, it may fail with an update conflict error. Your application logic must be prepared to handle these types of errors, though it's often preferable to handling deadlocks.

Yes, this deadlock can happen with other databases, but you are right that it's far less common with PostgreSQL. The reason is rooted in how different databases handle concurrency by default.

### **Why It's Rare in PostgreSQL**

PostgreSQL uses a concurrency model called **Multiversion Concurrency Control (MVCC)** by default.

Under MVCC, when a transaction reads data, it sees a "snapshot" of the data as it existed at that moment. Readers don't block writers, and writers don't block readers. This effectively prevents the specific locking conflict that causes the deadlock during the "check-then-insert" operation in the JobRepository. Your experience of not seeing this issue with PostgreSQL is the expected behavior. 👍

### **Why It's Common in SQL Server**

SQL Server's default **READ COMMITTED** isolation level uses a locking mechanism. When multiple transactions start concurrently, they can acquire shared locks while checking for the job instance's existence. When they all find nothing and try to promote their locks to exclusive locks for the INSERT operation, they end up in a deadlock, each waiting for the other to release its lock.

Enabling READ\_COMMITTED\_SNAPSHOT in SQL Server makes it behave more like PostgreSQL's MVCC, which is why it's the recommended fix.

So, while the potential for a race condition exists in the application logic, the database's underlying concurrency model is the primary reason

Map<Class<? extends Throwable>, Boolean> retryableExceptions = new HashMap<>();

retryableExceptions.put(DeadlockLoserDataAccessException.class, true);

retryableExceptions.put(UnexpectedJobExecutionException.class, true);

SimpleRetryPolicy policy = new SimpleRetryPolicy(3, retryableExceptions, true);

retryTemplate.setRetryPolicy(policy);

### **Best Fix for Spring Batch 5 + SQL Server**

Since sequence tables are out of the picture, the **most effective and cleanest solution** is the one Gemini suggested:

#### **🔹 Enable READ\_COMMITTED\_SNAPSHOT on your database**

sql

ALTER DATABASE [your\_database\_name] SET READ\_COMMITTED\_SNAPSHOT ON;

This makes SQL Server behave more like PostgreSQL’s MVCC model:

* Readers don’t block writers.
* Writers don’t block readers.
* No shared locks on reads = fewer deadlocks.

#### **🔹 Set the isolation level in your Spring Batch config**

java

factory.setIsolationLevelForCreate("ISOLATION\_READ\_COMMITTED\_SNAPSHOT");

This tells Spring Batch to use snapshot-based reads when checking for existing JobInstances.

### **🧪 Bonus: Add Retry Logic**

Even with snapshot isolation, transient deadlocks can still happen. Consider wrapping job launches in a retry mechanism using Spring Retry or a simple loop with exponential backoff.

### **🧵 TL;DR**

| **Feature** | **Status in Spring Batch 5** |
| --- | --- |
| Sequence tables | ❌ Deprecated |
| Identity columns | ✅ Used by default |
| Deadlock risk | ⚠️ Still possible on BATCH\_JOB\_INSTANCE |
| Best fix | ✅ Enable READ\_COMMITTED\_SNAPSHOT |

Let me know if you want help wiring this into your config or checking if your DB is already using snapshot isolation. We’ll get this humming smoothly.

RetryTemplate retryTemplate = new RetryTemplate();

retryTemplate.setRetryPolicy(new SimpleRetryPolicy(3)); // Retry up to 3 times

retryTemplate.setBackOffPolicy(new ExponentialBackOffPolicy());

retryTemplate.execute(context -> {

jobLauncher.run(myJob, new JobParametersBuilder()

.addLong("run.id", System.currentTimeMillis())

.toJobParameters());

return null;

});

retryTemplate.setBackOffPolicy(new ExponentialBackOffPolicy());

ExponentialBackOffPolicy backOff = new ExponentialBackOffPolicy();

backOff.setInitialInterval(200);

backOff.setMultiplier(2.0);

backOff.setMaxInterval(2000);

Map<Class<? extends Throwable>, Boolean> retryableExceptions = new HashMap<>();

retryableExceptions.put(DeadlockLoserDataAccessException.class, true);

retryableExceptions.put(UnexpectedJobExecutionException.class, true);

SimpleRetryPolicy policy = new SimpleRetryPolicy(3, retryableExceptions, true);

retryTemplate.setRetryPolicy(policy);

Check the Isolation Level for Your Session & DB

-- 🧠 Check current session's isolation level

SELECT CASE transaction\_isolation\_level

WHEN 0 THEN 'Unspecified'

WHEN 1 THEN 'Read Uncommitted'

WHEN 2 THEN 'Read Committed'

WHEN 3 THEN 'Repeatable Read'

WHEN 4 THEN 'Serializable'

WHEN 5 THEN 'Snapshot'

END AS CURRENT\_SESSION\_ISOLATION\_LEVEL

FROM sys.dm\_exec\_sessions

WHERE session\_id = @@SPID;

-- ⚙️ Check if READ\_COMMITTED\_SNAPSHOT is enabled at DB level

SELECT name AS DATABASE\_NAME,

snapshot\_isolation\_state\_desc,

is\_read\_committed\_snapshot\_on

FROM sys.databases

WHERE name = DB\_NAME();

### **Output Explained**

| **Column** | **Meaning** |
| --- | --- |
| CURRENT\_SESSION\_ISOLATION\_LEVEL | The transaction isolation level for **your current session** |
| snapshot\_isolation\_state\_desc | Whether **snapshot isolation** is enabled (ON, OFF, or IN\_TRANSITION) |
| is\_read\_committed\_snapshot\_on | Whether READ\_COMMITTED\_SNAPSHOT is on for your **entire DB** (0 = off, 1 = on) |