Table of Contents

Appendix A: Batch Processing and Transactions

A.1. Simple Batching with No Retry

Consider the following simple example of a nested batch with no retries. It shows a common scenario for batch processing: An input source is processed until exhausted, and we commit periodically at the end of a "chunk" of processing.

```
1   |  REPEAT(until=exhausted) {
|
2   |    TX {
3   |      REPEAT(size=5) {
3.1 |        input;
3.2 |        output;
|      }
|    }
|
|  }
```

The input operation (3.1) could be a message-based receive (such as from JMS), or a file-based read, but to recover and continue processing with a chance of completing the whole job, it must be transactional. The same applies to the operation at 3.2. It must be either transactional or idempotent.

If the chunk at `REPEAT` (3) fails because of a database exception at 3.2, then `TX` (2) must roll back the whole chunk.

A.2. Simple Stateless Retry

It is also useful to use a retry for an operation which is not transactional, such as a call to a web-service or other remote resource, as shown in the following example:

```
0   |  TX {
1   |    input;
1.1 |    output;
2   |    RETRY {
2.1 |      remote access;
|    }
|  }
```

This is actually one of the most useful applications of a retry, since a remote call is much more likely to fail and be retryable than a database update. As long as the remote access (2.1) eventually succeeds, the transaction, `TX` (0), commits. If the remote access (2.1) eventually fails, then the transaction, `TX` (0), is guaranteed to roll back.

A.3. Typical Repeat-Retry Pattern

The most typical batch processing pattern is to add a retry to the inner block of the chunk, as shown in the following example:

```
1   |  REPEAT(until=exhausted, exception=not critical) {
|
2   |    TX {
3   |      REPEAT(size=5) {
|
4   |        RETRY(stateful, exception=deadlock loser) {
4.1 |          input;
5   |        } PROCESS {
5.1 |          output;
6   |        } SKIP and RECOVER {
|           notify;
|         }
|
|       }
|     }
|
|  }
```

The inner `RETRY` (4) block is marked as "stateful". See the typical use case for a description of a stateful retry. This means that if the retry `PROCESS` (5) block fails, the behavior of the `RETRY` (4) is as follows:

1. Throw an exception, rolling back the transaction, `TX` (2), at the chunk level, and allowing the item to be re-presented to the input queue.
2. When the item re-appears, it might be retried depending on the retry policy in place, executing `PROCESS` (5) again. The second and subsequent attempts might fail again and re-throw the exception.
3. Eventually, the item reappears for the final time. The retry policy disallows another attempt, so `PROCESS` (5) is never executed. In this case, we follow the `RECOVER` (6) path, effectively "skipping" the item that was received and is being processed.

Note that the notation used for the `RETRY` (4) in the plan above explicitly shows that the input step (4.1) is part of the retry. It also makes clear that there are two alternate paths for processing: the normal case, as denoted by `PROCESS` (5), and the recovery path, as denoted in a separate block by `RECOVER` (6). The two alternate paths are completely distinct. Only one is ever taken in normal circumstances.

In special cases (such as a special `TranscationValidException` type), the retry policy might be able to determine that the `RECOVER` (6) path can be taken on the last attempt after `PROCESS` (5) has just failed, instead of waiting for the item to be re-presented. This is not the default behavior, because it requires detailed knowledge of what has happened inside the `PROCESS` (5) block, which is not usually available. For example, if the output included write access before the failure, then the exception

should be re-thrown to ensure transactional integrity.

The completion policy in the outer `REPEAT` (1) is crucial to the success of the above plan. If the output (5.1) fails, it may throw an exception (it usually does, as described), in which case the transaction, `TX` (2), fails, and the exception could propagate up through the outer batch `REPEAT` (1). We do not want the whole batch to stop, because the `RETRY` (4) might still be successful if we try again, so we add `exception=not critical` to the outer `REPEAT` (1).

Note, however, that if the `TX` (2) fails and we *do* try again, by virtue of the outer completion policy, the item that is next processed in the inner `REPEAT` (3) is not guaranteed to be the one that just failed. It might be, but it depends on the implementation of the input (4.1). Thus, the output (5.1) might fail again on either a new item or the old one. The client of the batch should not assume that each `RETRY` (4) attempt is going to process the same items as the last one that failed. For example, if the termination policy for `REPEAT` (1) is to fail after 10 attempts, it fails after 10 consecutive attempts but not necessarily at the same item. This is consistent with the overall retry strategy. The inner `RETRY` (4) is aware of the history of each item and can decide whether or not to have another attempt at it.

A.4. Asynchronous Chunk Processing

The inner batches or chunks in the typical example can be executed concurrently by configuring the outer batch to use an `AsyncTaskExecutor`. The outer batch waits for all the chunks to complete before completing. The following example shows asynchronous chunk processing:

```
1   |  REPEAT(until=exhausted, concurrent, exception=not critical) {
|
2   |    TX {
3   |      REPEAT(size=5) {
|
4   |        RETRY(stateful, exception=deadlock loser) {
4.1 |          input;
5   |        } PROCESS {
|          output;
6   |        } RECOVER {
|          recover;
|        }
|
|      }
|    }
|
|  }
```

A.5. Asynchronous Item Processing

The individual items in chunks in the typical example can also, in principle, be processed concurrently. In this case, the transaction boundary has to move to the level of the individual item, so that each transaction is on a single thread, as shown in the following example:

```
1   |  REPEAT(until=exhausted, exception=not critical) {
|
2   |    REPEAT(size=5, concurrent) {
|
3   |      TX {
4   |        RETRY(stateful, exception=deadlock loser) {
4.1 |          input;
5   |        } PROCESS {
|           output;
6   |        } RECOVER {
|           recover;
|         }
|       }
|
|     }
|
|  }
```

This plan sacrifices the optimization benefit, which the simple plan had, of having all the transactional resources chunked together. It is only useful if the cost of the processing (5) is much higher than the cost of transaction management (3).

A.6. Interactions Between Batching and Transaction Propagation

There is a tighter coupling between batch-retry and transaction management than we would ideally like. In particular, a stateless retry cannot be used to retry database operations with a transaction manager that does not support NESTED propagation.

The following example uses retry without repeat:

```
1   |  TX {
|
1.1 |    input;
2.2 |    database access;
2   |    RETRY {
3   |      TX {
3.1 |        database access;
|       }
|     }
|
|  }
```

Again, and for the same reason, the inner transaction, `TX` (3), can cause the outer transaction, `TX` (1), to fail, even if the `RETRY` (2) is eventually successful.

Unfortunately, the same effect percolates from the retry block up to the surrounding repeat batch if

there is one, as shown in the following example:

```
1   |  TX {
|
2   |    REPEAT(size=5) {
2.1 |      input;
2.2 |      database access;
3   |      RETRY {
4   |        TX {
4.1 |          database access;
|         }
|       }
|     }
|
|  }
```

Now, if TX (3) rolls back, it can pollute the whole batch at TX (1) and force it to roll back at the end.

What about non-default propagation?

- In the preceding example, `PROPAGATION_REQUIRES_NEW` at `TX` (3) prevents the outer `TX` (1) from being polluted if both transactions are eventually successful. But if `TX` (3) commits and `TX` (1) rolls back, then `TX` (3) stays committed, so we violate the transaction contract for `TX` (1). If `TX` (3) rolls back, `TX` (1) does not necessarily (but it probably does in practice, because the retry throws a roll back exception).
- `PROPAGATION_NESTED` at `TX` (3) works as we require in the retry case (and for a batch with skips): `TX` (3) can commit but subsequently be rolled back by the outer transaction, `TX` (1). If `TX` (3) rolls back, `TX` (1) rolls back in practice. This option is only available on some platforms, not including Hibernate or JTA, but it is the only one that consistently works.

Consequently, the `NESTED` pattern is best if the retry block contains any database access.

A.7. Special Case: Transactions with Orthogonal Resources

Default propagation is always OK for simple cases where there are no nested database transactions. Consider the following example, where the `SESSION` and `TX` are not global `XA` resources, so their resources are orthogonal:

```
0   |  SESSION {
1   |    input;
2   |    RETRY {
3   |      TX {
3.1 |        database access;
|       }
|     }
|  }
```

Here there is a transactional message `SESSION` (0), but it does not participate in other transactions with `PlatformTransactionManager`, so it does not propagate when `TX` (3) starts. There is no database access outside the `RETRY` (2) block. If `TX` (3) fails and then eventually succeeds on a retry, `SESSION` (0) can commit (independently of a `TX` block). This is similar to the vanilla "best-efforts-one-phase-commit" scenario. The worst that can happen is a duplicate message when the `RETRY` (2) succeeds and the `SESSION` (0) cannot commit (for example, because the message system is unavailable).

A.8. Stateless Retry Cannot Recover

The distinction between a stateless and a stateful retry in the typical example above is important. It is actually ultimately a transactional constraint that forces the distinction, and this constraint also makes it obvious why the distinction exists.

We start with the observation that there is no way to skip an item that failed and successfully commit the rest of the chunk unless we wrap the item processing in a transaction. Consequently, we simplify the typical batch execution plan to be as follows:

```
0   |  REPEAT(until=exhausted) {
|
1   |    TX {
2   |      REPEAT(size=5) {
|
3   |        RETRY(stateless) {
4   |          TX {
4.1 |            input;
4.2 |            database access;
|           }
5   |        } RECOVER {
5.1 |          skip;
|         }
|
|       }
|     }
|
|  }
```

The preceding example shows a stateless `RETRY` (3) with a `RECOVER` (5) path that kicks in after the final attempt fails. The `stateless` label means that the block is repeated without re-throwing any

exception up to some limit. This only works if the transaction, `TX` (4), has propagation NESTED.

If the inner `TX` (4) has default propagation properties and rolls back, it pollutes the outer `TX` (1). The inner transaction is assumed by the transaction manager to have corrupted the transactional resource, so it cannot be used again.

Support for NESTED propagation is sufficiently rare that we choose not to support recovery with stateless retries in the current versions of Spring Batch. The same effect can always be achieved (at the expense of repeating more processing) by using the typical pattern above.