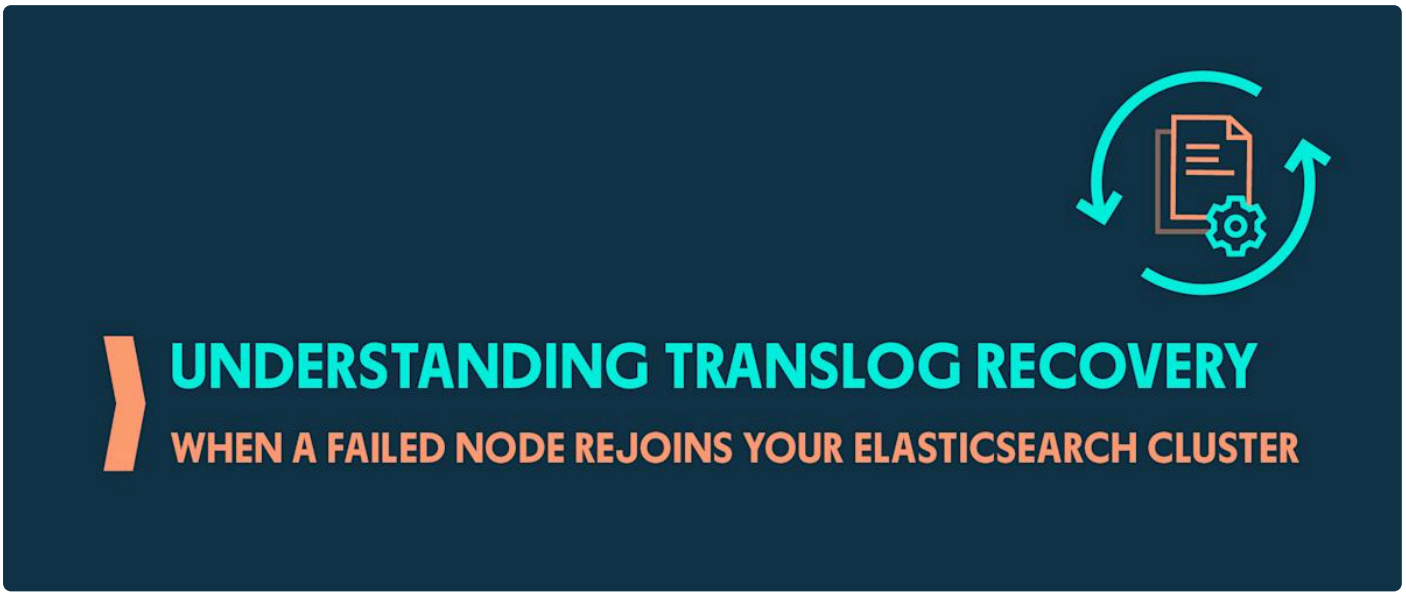
Understanding Translog Recovery When a Failed Node Rejoins Your Elasticsearch Cluster

Jochen Kressin · 2026-08-06



Introduction

Imagine you're running an Elasticsearch cluster, and suddenly one of your nodes goes down. The good news? Elasticsearch automatically promotes a replica shard to primary, keeping your cluster operational. But what happens when that failed node comes back online? More specifically, what happens to the translog, the append-only record of operations that haven't been committed to a Lucene segment yet, on the recovered node?

This scenario raises important questions about data consistency and safety. If the translog on the recovered node contains writes that were never replicated before the failure, could you end up with conflicting data? Let's explore how Elasticsearch elegantly handles this situation.

The Problem: Unreplicated Operations in the Translog

Picture this scenario:

1. You have a primary shard on Node A and a replica on Node B
2. Some write operations arrive at the primary on Node A
3. Before these operations can be replicated to Node B, Node A suddenly crashes
4. Elasticsearch promotes the replica on Node B to be the new primary
5. Node A comes back online and needs to rejoin the cluster

Now here's the concern: Node A's translog might contain operations that were never replicated to Node B. These operations were in-flight when the failure occurred. If Elasticsearch blindly used this translog during recovery, you could end up with:

- Data inconsistencies between the promoted primary and the recovered shard
- Conflicting writes that create an undefined state
- Potential data loss if the systems disagree on what data should exist

This is a legitimate concern, and understanding how Elasticsearch prevents this issue is key to trusting your cluster's data integrity.

How Elasticsearch Ensures Data Safety

Elasticsearch has a sophisticated mechanism to handle this exact scenario. The key insight is that Elasticsearch distinguishes between safe and unsafe operations in the translog.

What Makes an Operation "Safe"?

An operation is considered safe once it has been fully replicated to all in-sync replicas. That is exactly the condition that lets the primary acknowledge the write to the client: acknowledgment happens after the operation reaches the primary and every in-sync replica, so an acknowledged operation is always present on every in-sync copy.

It's worth being precise here, because it matters later: the global checkpoint is a separate marker that lags slightly behind acknowledgment—it is not on the synchronous acknowledgment path. An acknowledged write is safe because it physically exists on all in-sync copies, not because the global checkpoint has already advanced past it.

Operations that were never fully replicated are unsafe and can be discarded, because the client was never told that the write succeeded.

The Recovery Process

When the failed node rejoins the cluster, Elasticsearch follows a careful recovery process:

Step 1: Roll Back to the Last Safe Commit

The recovering shard first rolls back to its last "safe" commit point. A safe commit point is a Lucene commit that contains only fully-replicated writes (its operations are all at or below the global checkpoint). This ensures that the shard starts from a known-good state that's consistent with what other nodes in the cluster have.

Step 2: Replay Only Safe Operations

Next, Elasticsearch replays operations from the local translog, but only up to the global checkpoint. The operations it knows were fully replicated before the failure. Any operations above that point, those that were never acknowledged to the client, are discarded.

This approach is both safe and correct because:

- Clients never received acknowledgment for the discarded operations
- From the client's perspective, those writes never succeeded
- The client's application logic should handle the lack of acknowledgment (for example, by retrying)

Why This Works

The beauty of this approach is that it maintains consistency without sacrificing availability. Here's why:

- Data Consistency: By discarding unreplicated operations, Elasticsearch ensures that the recovered shard matches the current primary's state. There are no conflicts or divergent data.
- Client Expectations: Since unreplicated operations were never acknowledged to clients, discarding them doesn't violate any guarantees. The client already knows these operations might have failed.
- Automatic Recovery: The process happens automatically. You don't need to manually intervene or fix inconsistencies.

Understanding Translog Commits and Replication

To fully appreciate this mechanism, it helps to understand how Elasticsearch tracks replication:

Concept	Description
Global Checkpoint	The sequence number up to which all operations have been processed on all in-sync shards. Operations below this point are safe.
Local Checkpoint	The sequence number up to which all operations have been processed on a specific shard. This can be ahead of the global checkpoint.
Translog	An append-only log of operations. Used for crash recovery and replication.
Safe Commit	A Lucene commit point that only includes operations up to the global checkpoint.

When a primary shard processes a write operation, it:

1. Writes the operation locally—assigning it a sequence number and appending it to its own Lucene index and translog
2. Forwards the operation to all in-sync replica shards
3. Waits for those replicas to write it locally and acknowledge; each reply also carries the replica's updated local checkpoint
4. Acknowledges the write to the client, once the primary and all in-sync replicas hold the operation

The global checkpoint advances separately, after the fact. Using the local checkpoints reported by the replicas, the primary moves the global checkpoint forward and propagates the new value to the shard copies on the next indexing operation (or via a periodic global-checkpoint sync). In other words, acknowledgment does not wait for the global checkpoint to advance.

This sequence ensures that any acknowledged write is present on every in-sync copy, so it survives even if nodes fail. An acknowledged operation may briefly sit above the global checkpoint, but because it lives on all in-sync copies, the promoted primary still has it and resynchronizes it to the others during the primary/replica resync.

Solution: Trust Elasticsearch's Recovery Process

You don't need to do anything special to handle this scenario. Elasticsearch's built-in recovery process automatically:

1. Identifies the last safe commit point on the recovered shard
2. Rolls back to that commit point, discarding any unsafe state
3. Replays the safe operations from the translog (up to the global checkpoint)
4. Synchronizes with the current primary to catch up on any operations that occurred while the node was offline

In current Elasticsearch versions, that catch-up from the primary is driven by the shard's soft-delete history together with retention leases—rather than the translog—which lets the primary choose between an operation-based recovery (replaying just the missing operations) and a file-based recovery (copying Lucene segments) when the needed history is no longer retained. The local rollback in steps 1–3, however, is exactly the translog-based mechanism described above.

The recovered shard seamlessly rejoins the cluster with the correct data, and your cluster continues operating normally.

What About the Discarded Operations?

You might wonder: "What happens to those unreplicated operations that get discarded?"

The answer is straightforward: Nothing bad happens. Here's why:

- The client that sent those write requests never received an acknowledgment
- From the client's perspective, those operations failed
- Well-designed client applications already handle failures by implementing retry logic
- The client will retry the operation, and it will succeed against the current primary

This is actually the correct behavior. Elasticsearch's consistency model guarantees that once a write is acknowledged, it's safe. But if a write isn't acknowledged, the client knows it might not have succeeded and can take appropriate action.

Takeaways

- Elasticsearch automatically handles node recovery safely. When a failed primary node rejoins the cluster, it discards any unreplicated operations from its translog and synchronizes with the current primary.
- Unreplicated operations can be safely discarded because they were never acknowledged to clients. Clients already know these operations might have failed and can retry them.
- The recovery process maintains consistency by rolling back to the last safe commit point and replaying only fully-replicated operations. This prevents conflicts and data inconsistencies.
- Global and local checkpoints track replication. Elasticsearch uses these checkpoints to determine which operations are safe and which can be discarded during recovery. Note that the global

checkpoint lags slightly behind client acknowledgment—an acknowledged write is safe because it sits on every in-sync copy, not because the global checkpoint has already moved past it.

- Client retry logic is essential. Your application should always implement retry logic for write operations, since Elasticsearch's consistency guarantees only apply to acknowledged writes.
- Trust the built-in recovery mechanism. You don't need to manually intervene when nodes rejoin the cluster. Elasticsearch's recovery process is designed to handle these scenarios automatically and safely.

Understanding how Elasticsearch handles translog recovery helps you trust your cluster's data integrity, even when nodes fail and recover. The system is designed to prioritize consistency while maintaining high availability, giving you the best of both worlds.

This article was published on the Search Guard blog: https://search-guard.com/blog/translog-recovery-failed-node-rejoins-elasticsearch/