Deduplicating Events in Elasticsearch: The Upsert Strategy

Jochen Kressin · 2026-09-15

Introduction

Imagine you're ingesting events into Elasticsearch through Logstash, and you notice something peculiar: the same event (identified by a unique ID) keeps arriving multiple times with slightly different field values. Your business stakeholders don't want to see duplicate rows in Kibana—they want a single, consolidated view showing the most recent state of each event.

You consider using Elasticsearch transforms to group these events, but quickly realize this approach would double your storage requirements. You're storing both the raw events and the transformed results, which isn't sustainable. What's the solution?

The answer lies in treating Elasticsearch like a database with replace-on-write operations—overwriting existing documents instead of always inserting new ones. Let's explore how to implement this strategy efficiently.

Problem Description

You have an ELK (Elasticsearch, Logstash, Kibana) stack where:

- Events arrive through Logstash
- Multiple events can share the same ID but have different field values
- Each arrival represents an updated state of the same logical entity
- Business requirements dictate showing only one row per ID in Kibana
- Storage is limited, preventing you from keeping both raw and transformed data

Your initial approach was to use Elasticsearch transforms to create a separate index with deduplicated data. However, this doubles your storage footprint:

1. **Source index**: Contains all raw events (duplicates included)
2. **Destination index**: Contains transformed, deduplicated events

With limited storage, this isn't viable. You need a way to consolidate events at ingestion time, not after the fact.

Root Cause Analysis

The core issue is a mismatch between your data model and your ingestion strategy:

Event-Based vs. Entity-Based Data Models

Elasticsearch is often used as an event store where each document represents an immutable event. This works well for log data, where you want to preserve every occurrence. However, your use case is different—you're tracking entities that change over time, not independent events.

When you ingest data without specifying a document ID, Elasticsearch generates a random ID for each document. This means:

```
POST /events/_doc
{
  "event_id": "12345",
  "status": "pending",
  "timestamp": "2025-05-27T10:00:00Z"
}
```

And later:

```
POST /events/_doc
{
  "event_id": "12345",
  "status": "completed",
  "timestamp": "2025-05-27T11:00:00Z"
}
```

Both get stored as separate documents, even though they represent the same logical entity (`event_id: 12345`).

Why Transforms Aren't the Right Solution

Elasticsearch transforms create derived indices by continuously aggregating source data. For deduplication, you'd configure a transform that groups by ID and keeps the latest values:

```
{
  "source": { "index": "events" },
  "dest": { "index": "events-deduplicated" },
  "pivot": {
    "group_by": { "event_id": { "terms": { "field": "event_id" } } },
    "aggregations": { /* latest values */ }
  }
}
```

This works, but you're now storing:

- Every raw event in the source index
- The deduplicated result in the destination index

With tight storage constraints, this duplication is a deal-breaker.

Solution

The solution is to implement **replace-on-write logic at ingestion time** using Logstash. This ensures that only the latest version of each event is stored in Elasticsearch.

Strategy: Use Document IDs for Deterministic Overwrites

Instead of letting Elasticsearch generate random document IDs, specify the `event_id` as the document ID. When you index a document with an existing ID, Elasticsearch replaces the old document entirely with the new one—every field from the previous version that isn't present in the new document will be lost.

Implementation in Logstash

Configure your Logstash pipeline to use the event's unique identifier as the Elasticsearch document ID:

```
output {
  elasticsearch {
    hosts => ["http://localhost:9200"]
    index => "events"
    document_id => "%{event_id}"
    action => "index"
  }
}
```

Here's what each parameter does:

Parameter	Purpose
`document_id`	Sets the Elasticsearch `_id` to the value of the `event_id` field from your event
`action`	Set to `"index"` (default), which creates a new document or replaces an existing one entirely

How It Works

When Logstash processes events:

First event arrives:

```
{
  "event_id": "12345",
  "status": "pending",
  "timestamp": "2025-05-27T10:00:00Z"
}
```

Logstash indexes this as:

```
PUT /events/_doc/12345
{
  "event_id": "12345",
  "status": "pending",
  "timestamp": "2025-05-27T10:00:00Z"
}
```

Second event arrives with the same ID:

```
{
  "event_id": "12345",
  "status": "completed",
  "timestamp": "2025-05-27T11:00:00Z"
}
```

Logstash indexes this as:

```
PUT /events/_doc/12345
{
  "event_id": "12345",
  "status": "completed",
  "timestamp": "2025-05-27T11:00:00Z"
}
```

Elasticsearch replaces the first document entirely. Only the latest version remains in the index.

Preserving Historical Data (Optional)

If you need to track changes over time while still presenting a single "current" view, consider this hybrid approach:

Option 1: Separate hot and cold indices

```
output {
  # Current state index (deduplicated)
  elasticsearch {
    hosts => ["http://localhost:9200"]
    index => "events-current"
    document_id => "%{event_id}"
    action => "index"
  }

  # Historical archive (all events)
  elasticsearch {
    hosts => ["http://localhost:9200"]
    index => "events-history-%{+YYYY.MM.dd}"
    action => "index"
  }
}
```

The `events-current` index has only the latest state (for Kibana dashboards), while `events-history` keeps everything (for auditing). You can apply different lifecycle policies—for example, keeping `events-current` in hot storage and moving `events-history` to cold storage after 30 days.

Option 2: Use the update action for field-level merging

If you want to merge new fields into existing documents rather than replacing them entirely, use Elasticsearch's update API. This is a genuine upsert operation—it creates the document if it doesn't yet exist, and merges the fields otherwise:

```
output {
  elasticsearch {
    hosts => ["http://localhost:9200"]
    index => "events"
    document_id => "%{event_id}"
    action => "update"
    doc_as_upsert => true
  }
}
```

This merges new fields into existing documents. The `doc_as_upsert => true` parameter is the key: it tells Elasticsearch to use the incoming Logstash event as both the partial update document and the upsert document if no document with the given ID exists yet. No Painless script is needed for this pattern—the merge happens natively.

For most use cases, though, the simpler replacement strategy (using `action => "index"`) is sufficient, faster, and more predictable.

Managing Storage with ILM

To further optimize storage, configure Index Lifecycle Management (ILM) to automatically move old data to cheaper storage tiers.

Important: Do not use the `rollover` action for your deduplicated index when writing through an alias! When ILM triggers a rollover, a new backing index is created and the write alias points to it. Any subsequent write for an existing document ID will create a new document in the new backing index rather than overwriting the original—breaking your deduplication. Instead, use a purely age-based policy without rollover:

```
PUT _ilm/policy/events-policy
{
  "policy": {
    "phases": {
      "hot": {
        "min_age": "0ms",
        "actions": {}
      },
      "warm": {
        "min_age": "30d",
        "actions": {
          "forcemerge": { "max_num_segments": 1 }
        }
      },
      "cold": {
        "min_age": "60d",
        "actions": {
          "searchable_snapshot": { "snapshot_repository": "my_repository" }
        }
      },
      "delete": {
        "min_age": "90d",
        "actions": {
          "delete": {}
        }
      }
    }
  }
}
```

Apply this policy to your index template:

```
PUT _index_template/events-template
{
  "index_patterns": ["events-*"],
  "template": {
    "settings": {
      "index.lifecycle.name": "events-policy"
    }
  }
}
```

Verification

After implementing the strategy, verify it's working:

1. **Check document count:**

```
GET /events/_count
```

The count should equal the number of unique event IDs, not the total number of ingested events.

2. **Inspect a specific document:**

```
GET /events/_doc/12345
```

You should see the latest version of the event.

3. **Monitor version numbers:**

```
GET /events/_search
{
  "query": { "term": { "event_id.keyword": "12345" } },
  "_source": false,
  "fields": ["event_id"],
  "version": true
}
```

The `_version` field increments with each update, confirming that documents are being replaced rather than duplicated.

Takeaways

1. **Choose the right data model for your use case**: If you're tracking entities that change over time (not immutable events), use the entity's ID as the Elasticsearch document ID to enable deterministic overwrites.
2. **Replacing at ingestion is more efficient than post-processing**: Using Logstash to specify document IDs prevents duplication at the source, eliminating the need for transforms and reducing storage by half.
3. **Transforms are powerful but have storage costs**: Elasticsearch transforms are ideal for aggregations and analytics, but they create separate indices. For simple deduplication, the replace-on-write approach is more efficient.
4. **Hybrid approaches balance current state and history**: If you need both a current view (for dashboards) and historical data (for auditing), use two separate indices with different lifecycle policies.
5. **ILM optimizes long-term storage, but beware of rollover with aliases**: Implement ILM to automatically move older data to cold storage, but remember that rollover actions will break document ID-based deduplication when you write through an alias pointing to multiple backing indices.
6. **Understand the difference between `action => "index"` and a true upsert**: Using `action => "index"` with a specified `document_id` replaces the entire document if it already exists. This is not the same as

a merge-based upsert—any fields missing from the new document will be lost. For field-level merging, use `action => "update"` with `doc_as_upsert => true` instead.

Need advanced data management and security for your Elasticsearch deployment? Search Guard provides comprehensive access control and audit logging to help you manage complex ingestion pipelines while maintaining compliance and data integrity.

This article was published on the Search Guard blog: https://search-guard.com/blog/deduplicating-events-elasticsearch-upsert-strategy/