OpenSearchCon EUROPE

Boosting OpenSearch Performance: Lucene Bulk Collection and gRPC Search in Action

Abdul Muneer, Carlos Rolo

April 17, 2026

NetApp Instaclustr

About us

- Abdul Muneer Kolarkunnu (Muneer) @akolarkunnu
- Open Source Developer @ NetApp Instaclustr
- Maintainer: OpenSearch MLCommons
- Committer: OpenJDK

- Carlos Rolo @cjrolo
- Open Source Developer @ NetApp Instaclustr
- OpenSearch Ambassador and Contributor in OpenSearch MLCommons
- Contributor: Cassandra

Problem framing

- Aggregations are CPU-heavy in analytics workloads.
- Traditional per-doc collection repeats expensive call patterns.
- The optimization theme: **do more work per call**, fewer dispatches.
- Collector and doc-value access are on hot path.
- Sub-aggregations multiply collection cost.
- 0-40% performance improvement

Types of Aggregations

- **Metric aggregations**: avg/sum/stats
- **Bucket aggregations**: range/histogram/date_histogram/filters
- **Pipeline aggregations:** max_bucket, min_bucket

JVM concepts

- Virtual dispatch is expensive at scale.
- Method inlining is easier when call sites are predictable.
- Batching reduces repeated dispatch and helps JIT optimize.

Small Java example: Method Inline


```java
public class InliningDemo {
    public static void main(String[] args) {
        Position pos = new Position(10);
        for (int i = 0; i < 1_000_000; i++) {
            int currentX = pos.getX(); // JIT will likely inline this call
            pos.setX(currentX + 1);    // JIT will likely inline this call
        }
        System.out.println("Final X: " + pos.getX());
    }
}

class Position {
    private int x;
    public Position(int x) { this.x = x; }

    // Small "accessor" methods are prime candidates for inlining
    public int getX() { return x; }
    public void setX(int x) { this.x = x; }
}
```


Small Java example: Method Inline


```java
public class InliningDemo {
    public static void main(String[] args) {
        Position pos = new Position(10);
        // Call the method many times to trigger JIT optimization
        for (int i = 0; i < 1_000_000; i++) {
            int currentX = pos.x; // JIT will likely inline this call
            pos.x = currentX + 1;    // JIT will likely inline this call
        }
        System.out.println("Final X: " + pos.getX());
    }
}

class Position {
    private int x;
    public Position(int x) { this.x = x; }

    // Small "accessor" methods are prime candidates for inlining
    public int getX() { return x; }
    public void setX(int x) { this.x = x; }
}
```


Small Java example: Method Inline


```
@ 17   java.lang.String::isLatin1 (19 bytes)   inline
@ 27   java.lang.StringLatin1::hashCode (9 bytes)   inline
  @ 5   jdk.internal.util.ArraysSupport::hashCodeOfUnsigned (55 bytes)   failed to
@ 37  java/lang/StringUTF16::hashCode (not loaded)   failed to inline: not inlineal
@ 6   java.lang.String::coder (15 bytes)   inline
@ 6   Position::<init> (10 bytes)   inline
  @ 1   java.lang.Object::<init> (1 bytes)   inline
@ 19   Position::getX (5 bytes)   inline
@ 27   Position::setX (6 bytes)   inline
@ 40   Position::getX (5 bytes)   inline
@ 43  java/lang/invoke/MethodHandle::invokeBasic (not loaded)   failed to inline: r
@ 48  java/io/PrintStream::println (not loaded)   failed to inline: not inlineable
@ 6   Position::<init> (10 bytes)   inline
  @ 1   java.lang.Object::<init> (1 bytes)   inline
@ 19   Position::getX (5 bytes)   inline
@ 27   Position::setX (6 bytes)   inline
@ 40   Position::getX (5 bytes)   inline
@ 43  java/lang/invoke/MethodHandle::invokeBasic (not loaded)   failed to inline: r
```


Small Java example: Virtual Inline


```java
class Animal {
    void speak() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void speak() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    void speak() {
        System.out.println("Cat meows");
    }
}
```


Small Java example: Virtual Inline


```java
// Create a large list of mixed objects
List<Animal> zoo = new ArrayList<>();
for (int i = 0; i < 1000; i++) {
    if (i % 3 == 0) {
        zoo.add(new Dog());
    } else if (i % 3 == 1) {
        zoo.add(new Cat());
    } else {
        zoo.add(new Animal());
    }
}

// Traditional per-object invocation (virtual dispatch for each element)
System.out.println("=== Individual Invocation ===");
for (Animal a : zoo) {
    a.speak();  // JVM decides at runtime which speak() to call
}
```


Small Java example: Virtual Inline


```java
// Grouping similar elements together and invoking in bulk
System.out.println("\n=== Bulk Grouped Invocation ===");
List<Dog> dogs = new ArrayList<>();
List<Cat> cats = new ArrayList<>();
List<Animal> animals = new ArrayList<>();

for (Animal a : zoo) {
    if (a instanceof Dog) dogs.add((Dog) a);
    else if (a instanceof Cat) cats.add((Cat) a);
    else animals.add(a);
}

// Bulk invocation per group
System.out.println("Dogs:");
dogs.forEach(Dog::speak);  // tight loop, fewer virtual dispatches

System.out.println("Cats:");
cats.forEach(Cat::speak);

System.out.println("Animals:");
animals.forEach(Animal::speak);
```


Key Lucene change #1: bulk doc-values retrieval

- Important code idea introduced

```
// Before: advanceExact + longValue per doc
// After: bulk retrieval
values.longValues(count, docBuffer, valueBuffer, missingSentinel);
```

- **Why this helps:**
- Fewer API calls for same number of docs
- Better cache-local processing over arrays
- Lower per-doc overhead

Key Lucene change #1: bulk doc-values retrieval

OpenSearchCon EUROPE

```
81  +   public void longValues(int size, int[] docs, long[] values, long defaultValue)
82  +       throws IOException {
83  +     for (int i = 0; i < size; ++i) {
84  +       int doc = docs[i];
85  +       long value;
86  +       if (advanceExact(doc)) {
87  +         value = longValue();
88  +       } else {
89  +         value = defaultValue;
90  +       }
91  +       values[i] = value;
92  +     }
93  +   }
```

```
163 -       for (int i = 0; i < size; ++i) {
164 -         if (norms.advanceExact(buffer.docs[i])) {
165 -           normValues[i] = norms.longValue();
166 -         } else {
167 -           normValues[i] = 1L;
168 -         }
169 -       }
```

NetApp Instaclustr

Key Lucene change #2: batched collector input

- Important code idea introduced

```
for (int count = stream.intoArray(docBuffer);
   count != 0;
   count = stream.intoArray(docBuffer)) {

  values.longValues(count, docBuffer, valueBuffer, Long.MIN_VALUE);
  // local tight loop for aggregation
}
```

- **Why this helps:**
- Replaces repeated single-doc collection with chunked processing
- Less virtual dispatch pressure in hot loop
- Better inlining opportunities

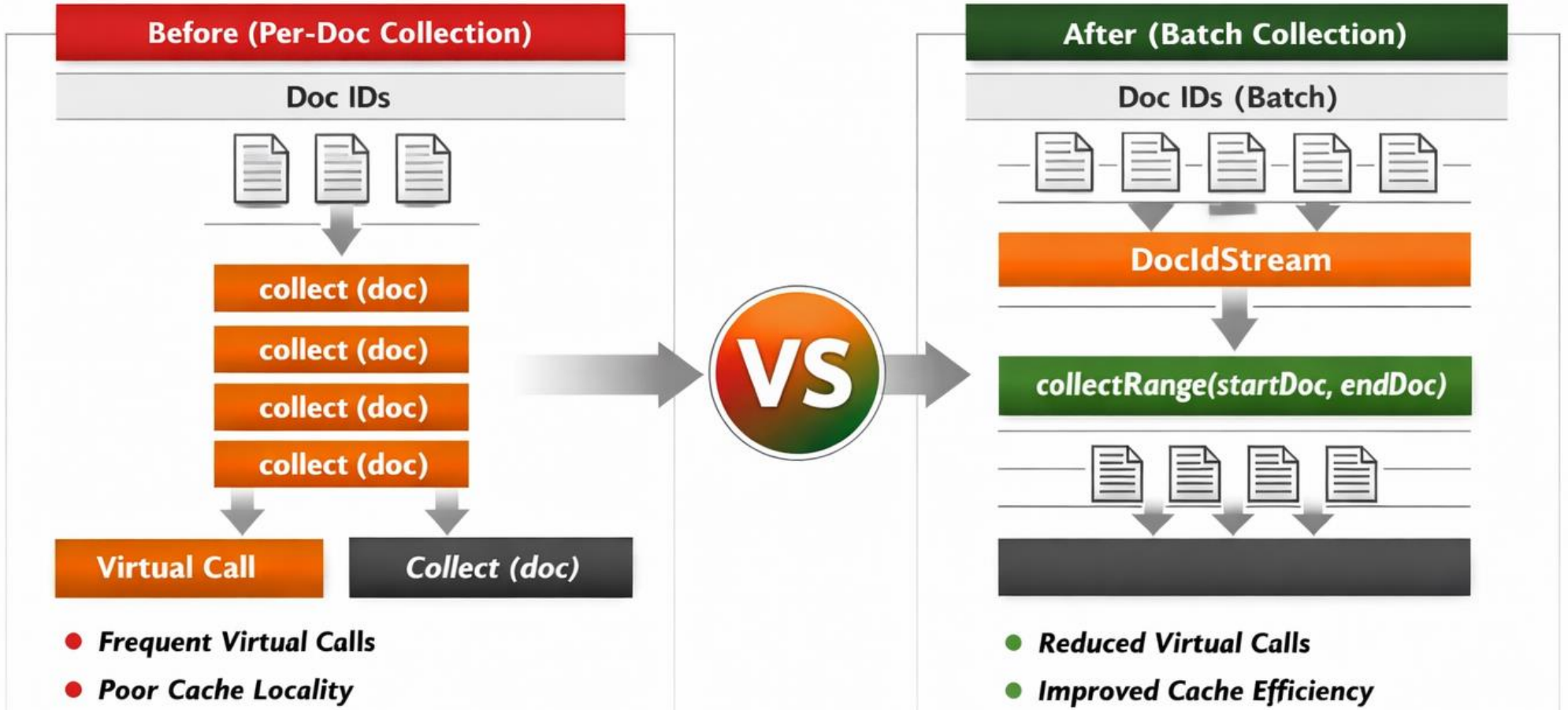
Key Lucene change #3: range-oriented collection path

- Important code idea introduced

```
collector.collectRange(minDoc, maxDoc);
```

- **Why this helps:**
- Reduces Virtual Dispatch Overhead
- Improves CPU Efficiency
- Enhances JIT Optimization

Key Lucene change #3: range-oriented collection path



Why gRPC for OpenSearch?

- REST/JSON is flexible, but parsing text payloads adds overhead.
- gRPC uses protocol buffers, giving a binary, typed transport layer.
- This is especially attractive for low-latency, high-throughput applications.
- The goal is not to replace REST everywhere, but to offer a faster path for the right workloads.

Where gRPC Stands in OpenSearch


```yaml
aux.transport.types: [transport-grpc]
aux.transport.transport-grpc.port: '9400-9500'
```

- gRPC support was introduced in OpenSearch **3.0**.
- The **Bulk** and **k-NN** gRPC APIs became generally available in **3.2**.
- The **Search** gRPC API supports selected query types and is still marked experimental.
- To use it, clients need the OpenSearch protobuf definitions and gRPC transport enabled.

What OpenSearch 3.4 Added

- OpenSearch **3.4** expanded gRPC Search with more query types.
- It also improved gRPC Bulk with better request handling and more document format support.
- The key message is: `broader query coverage + more practical bulk ingestion`.
- This makes gRPC more usable in real application flows, not just niche demos.
- **Search**: more supported query types
- **Bulk**: better format support and fixes

New Query Support in 3.4

- Added query support includes ***ConstantScoreQuery***, ***FuzzyQuery***, ***MatchBoolPrefixQuery***, ***MatchPhrasePrefix***, ***PrefixQuery***, and ***MatchQuery***.
  - **MatchQuery** helps with normal full-text search.
  - **FuzzyQuery** helps with typo tolerance.
  - **MatchBoolPrefixQuery** and **MatchPhrasePrefix** help with search-as-you-type and autocomplete-like experiences.
  - **PrefixQuery** helps when matching structured prefixes.
  - **ConstantScoreQuery** is useful when filtering matters more than score differences.

Simple Full-Text Search with *MatchQuery*

OpenSearchCon EUROPE

```
grpcurl -plaintext \
  -d '{
    "index": ["products"],
    "search_request_body": {
      "size": 3,
      "query": {
        "match": {
          "title": {
            "query": "wireless noise cancelling headphones"
          }
        }
      }
    }
  }' \
  localhost:9400 \
  org.opensearch.protobufs.services.SearchService/Search
```

- Product search
  - gRPC Search is becoming practical for common application search patterns
- Query the **title** field for something like `***wireless noise cancelling headphones***`.

NetApp Instaclustr

Autocomplete and Typo Tolerance


```
{
  "query": {
    "match_bool_prefix": {
      "title": {
        "query": "wireless hea"
      }
    }
  }
}
```

```
{
  "query": {
    "fuzzy": {
      "brand": {
        "value": "applw"
      }
    }
  }
}
```

- **MatchBoolPrefixQuery** is great for search-as-you-type.
- **FuzzyQuery** is great when users misspell a term.

Bulk Improvements in 3.4

- gRPC Bulk now supports **CBOR**, **SMILE**, and **YAML** document formats in addition to JSON.
- gRPC can auto-detect document format from bytes.
- Unlike REST Bulk, gRPC is not constrained by NDJSON line-delimiter parsing in the same way.
- Bulk fixes also improved request behavior, including better update handling and more efficient byte processing.

Mixed-Format Bulk Ingestion


```
{
  "index": "movies",
  "bulk_request_body": [
    {
      "operation_container": {
        "create": { "x_index": "movies", "x_id": "json-doc-1" }
      },
      "object": "<base64-json-bytes>"
    },
    {
      "operation_container": {
        "create": { "x_index": "movies", "x_id": "smile-doc-1" }
      },
      "object": "<base64-smile-bytes>"
    },
    {
      "operation_container": {
        "create": { "x_index": "movies", "x_id": "yaml-doc-1" }
      },
      "object": "<base64-yaml-bytes>"
    }
  ]
}
```

The important concept is that protobuf gives explicit message boundaries, so document bytes can be interpreted per request item.

That lets gRPC handle cases that are awkward or impossible in traditional NDJSON bulk flows.

Demo

- Quick Demo

Why gRPC for OpenSearch?

- gRPC gives OpenSearch a lower-overhead, typed transport option for high-throughput use cases.
- OpenSearch **3.4** expands gRPC Search with practical new query types.
- OpenSearch **3.4** also improves gRPC Bulk with support for multiple document formats and request-handling fixes.
- The result is better developer ergonomics and broader applicability for gRPC-based clients.
- One important caveat: gRPC Search is still experimental for select query types, so teams should validate it against their workloads.

"Taken together, these changes show how OpenSearch is improving performance at multiple layers, from Lucene aggregation internals to client-to-cluster transport efficiency."

Q&A

Questions? Let’s chat

Want to learn more? Let's connect!

OpenSearchCon EUROPE

NetApp Instaclustr

https://www.linkedin.com/in/muneer-kolarkunnu/

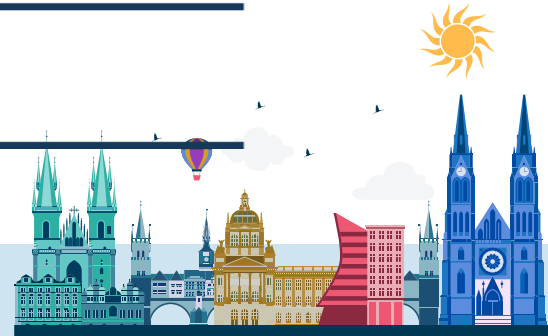
www.instaclustr.com

Muneer.kolarkunnu@netapp.com

https://www.linkedin.com/in/carlosjuzarterolo/

www.instaclustr.com

Carlos.Rolo@netapp.com




OpenSearchCon
EUROPE