Accelerating Aggregations with Lucene-Aware Execution in OpenSearch

- Ankit Jain, AWS OpenSearch

About Me

@jain_ankitk
LinkedIn
Github

SPEAKER

Ankit Jain

AWS OpenSearch | Lucene Committer | OpenSearch Maintainer

Palo Alto, California, United States

Ankit Jain is a Software Engineer on the Amazon OpenSearch Service team, leading performance and scalability initiatives for search infrastructure. He is an active maintainer and committer for the Apache Lucene and OpenSearch projects, with hands-on experience operating large-scale OpenSearch deployments and solving complex production performance challenges. He holds a B.Tech. in Computer Science from IIT Delhi.

AREA OF EXPERTISE

Information & Communications Technology

Agenda

- The Problem: Per-Document Overhead in Aggregations
- Quick Recap: Prior OpenSearch Work
- Lucene's Bulk Collection Primitives
- Bulk Collection in OpenSearch
- Skip Index + Bulk Collection
- Performance Results & Trade-offs
- What's Next

"8 Seconds to return 12 values"

```json
{
  "size": 0,
  "query": {
    "range": {
      "dropoff_datetime": {
        "gte": "2015-01-01",
        "lt": "2016-01-01"
      }
    }
  },
  "aggs": {
    "dropoffs_over_time": {
      "date_histogram": {
        "field": "dropoff_datetime",
        "calendar_interval": "month"
      }
    }
  }
}
```

nyc_taxis dataset

Simple monthly date histogram

Returns only 12 count values

Takes 7-8 seconds ❌

7-8 sec for 12 numbers

Where does the Time go?

More than 90% of the time goes in document collection — the most expensive part of query execution.

org/apache/lucene/ut.. vtab..
org/apache/lucene/codecs/lucene90/L..
org/apache/lucene/codecs/lucene90/Lucen..
org/apache/lucene/index/SingletonSortedNumericDocValues... org/opensearch/search/aggregations/bucket/BucketsAggregator.collectExistingBucket
org/opensearch/search/aggregations/bucket/histogram/DateHistogramAggregator$1.collect
org/opensearch/search/aggregations/LeafBucketCollector.collect
org/apache/lucene/search/MultiCollector$MultiLeafCollector.collect
org/apache/lucene/util..
org/apache/lucene/codecs/lucene90/.. or.. org/opensearch/common/util/LongHash.set
org/apache/lucene/codecs/lucene90/Luc.. org/a.. org/opensearch/common/util/LongHash.add
org/apache/lucene/index/SingletonSortedNumericDocVa.. org/opensearch/search/aggregations/bucket/terms/LongKeyedBucketOrds$FromSingle.add
org/opensearch/search/aggregations/bucket/histogram/DateHistogramAggregator$1.collect
org/opensearch/search/aggregations/LeafBucketCollector.collect
org/apache/lucene/search/Weight$DefaultBulkScorer.scoreRange
org/apache/lucene/search/Weight$DefaultBulkScorer.score
org/opensearch/search/internal/CancellableBulkScorer.score
org/apache/lucene/search/BulkScorer.score
org/opensearch/search/internal/ContextIndexSearcher.searchLeaf
org/opensearch/search/internal/ContextIndexSearcher.search
org/apache/lucene/search/IndexSearcher.search
org/opensearch/search/query/QueryPhase.searchWithCollector
org/opensearch/search/query/QueryPhase$DefaultQueryPhaseSearcher.searchWithCollector
org/opensearch/search/query/QueryPhase$DefaultQueryPhaseSearcher.searchWith
org/opensearch/search/query/QueryPhase.executeInternal
org/opensearch/search/query/QueryPhase.execute
org/opensearch/search/SearchService.loadOrExecuteQueryPhase
org/opensearch/search/SearchService.executeQueryPhase
org/opensearch/search/SearchService$2.lambda$onResponse$0
org/opensearch/search/SearchService$2$$Lambda$5268.0x0000000801dc1ec0.get
org/opensearch/search/SearchService$$Lambda$5271.0x0000000801dc2538.get
org/opensearch/action/ActionRunnable.lambda$supply$0
org/opensearch/action/ActionRunnable$$Lambda$4504.0x0000000801c988f0.accept
org/opensearch/action/ActionRunnable$2.doRun
org/opensearch/common/util/concurrent/AbstractRunnable.run
org/opensearch/threadpool/TaskAwareRunnable.doRun
org/opensearch/common/util/concurrent/AbstractRunnable.run
org/opensearch/common/util/concurrent/TimedRunnable.doRun
org/opensearch/common/util/concurrent/ThreadContext$ContextPreservingAbstractRunnable.doRun
org/opensearch/common/util/concurrent/AbstractRunnable.run
java/util/concurrent/ThreadPoolExecutor.runWorker
java/util/concurrent/ThreadPoolExecutor$Worker.run
java/lang/Thread.run
all

Execution Example

Query [Fast]

Gives doc IDs on which the aggregations will be performed →

Aggregation [Slow]

Status == 200 And OS == Android

Find docIds that matches the filter

200 → 1,2,5...700

404 → 4 ...

BKD index

Doc ID	Status
1	200
2	200
3	400
4	404
5	200
...	...
...	...
700	200
701	400

AND

Doc ID	OS
1	Android
2	iOS
3	iOS
4	Android
5	Android
...	...
...	...
700	Android
701	iOS

Find docIds that matches the filter

Android	1,4,5 ..700
iOS	2,3 .. 701

Inverted index

Find Avg size

Doc ID	Size
1	111
2	123
3	234
4	97
5	177
...	...
...	...
700	56
701	243

Doc values

Traverse through each matched document
And retrieve the values [SLOW]

Aggregation Execution

Key factors:
- Independence of query and aggregation
- Single traversal with docIds
- Document-at-a-time (DAAT)

Index Structure

Query

Matching
Doc Ids

- doc 1
- doc 2
- doc 5

Aggregator A

Agg B

Agg C

...

...

collect

Buckets

-A1
-[B1....]...
-[C1...]
-A2
-[B1....]..
-[C1...]
-....

Document-at-a-Time (DAAT)

Weight → Scorer → LeafCollector → collect(doc)

Per-document virtual calls:

collect(doc)

advanceExact(doc)

longValue()

× millions of docs

Why Per-Document Execution Is Slow

Deep Virtual Call Chains

advanceExact() + longValue() on every document. JIT can't inline deep chains.

Megamorphic Call Sites

Multiple aggregator implementations behind the same collect interface defeat JIT speculation.

No Bulk Counting

One doc at a time means no popcnt, no SIMD, no auto-vectorization opportunities.

Poor Cache Locality

Random doc value access pattern causes cache misses. No spatial or temporal locality.

Lucene has been addressing similar overhead on the query side — introducing bulk APIs like DocIdStream, intoBitSet(), and vectorized scoring. The same principles apply to aggregations.

Quick Recap: Prior OpenSearch Work

→ **Range Filter per bucket** → **BKD Tree count()**

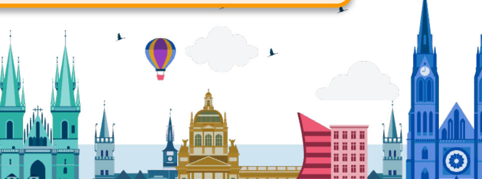
Filter Rewrite (OS 2.12, #11083):
- Rewrites date histogram into range filters
- Counts via BKD tree. 10-50x faster.

Multi-Range Traversal (OS 2.14, #13317):
- Two-pointer: all ranges in single BKD pass
- Up to 100x on minute-level aggs.

Limitation:

Filter field must equal aggregation field

No support for deleted documents
(in multi-range traversal)



Filter Rewrite

Range Query [351,771)

420

[1,800]
800 docs

50

370

[1,400]

[401,800]

0

50

200

170

[1,200]

[201,400]

[401,600]

[601,800]

100

70

50

1.....................100

101..............200

201..............300

301..............400

401..............500

501..............600

601..............700

701..............800

Multi Range Traversal

[1,800]
800 docs

[1,400]
[401,800]

[1,200]
[201,400]
[401,600]
[601,800]

1..................100
101............200
201............300
301............400
401...........500
501...........600
601...........700
701...........800

100
50
100
200
20
20

[-100,151)
150

[201, 301)
100

[401,601)
200

[681,721)
40

Lucene's Bulk Collection Primitives

Click and drag to zoom; shift + click and drag to scroll after zooming; hover over an annotation to see details

CountOrHighHigh

Queries/sec

0
50
100
150
200
250
300
350
400
450
500
550
600
650

Oct 2023
Jan 2024
Apr 2024
Jul 2024
Oct 2024
Jan 2025
Apr 2025
Jul 2025
Oct 2025
Jan 2026
Apr 2026

Date

collect(DocIdStream) — Bulk Collection

TRADITIONAL:

```
void collect(int doc)
    throws IOException;
```

NEW:

```
void collect(DocIdStream stream)
    throws IOException;
```

- DocIdStream provides count() — uses hardware popcnt on backing bitset
- DocIdStream provides forEach() — iterate docs when needed
- BooleanScorer creates DocIdStream backed by its internal bitset
- Default impl: stream.forEach(this::collect) — opt-in optimization

Result: 60% speedup on disjunction counting

intoArray() — Doc IDs into Buffers

```
DocIdStream#intoArray(int[] buffer)

  Materializes doc IDs from a stream directly
  into an int[] buffer.

  No per-doc virtual calls on the stream.
```

This is the key API that OpenSearch's LeafBucketCollector uses to batch doc IDs before flushing to aggregators.

DocIdStream → int[] buffer → batch flush to collect(int[], int, long)

longValues() — Bulk Doc Value Retrieval

```
NumericDocValues#longValues(
    int count,
    int[] docs,        // input: doc IDs
    long[] values,     // output: corresponding values
    long sentinel      // marker for missing values
)
```

- Bulk retrieval: given array of doc IDs, fills corresponding array of long values in one call
- Eliminates per-document advanceExact() + longValue() virtual calls
- This is the API that enables the true fast path in OpenSearch aggregators

The key enabler: batch doc values loading for DateHistogram, Sum, Avg, Min, Max, NumericTerms

DenseConjunctionBulkScorer

Path 1: collectRange — all clauses match, no deletes

```
if (acceptDocs == null
    && windowApproximations.isEmpty()) {
  // All clauses fully match this window
  collector.collectRange(min, maxDocID);
}
```

Path 2: collect(DocIdStream) — bitset intersection

```
// After AND-ing clause bitsets:
collector.collect(
  new BitSetDocIdStream(
    windowMatches, windowBase));
```

- Used for dense conjunctions (most docs match) including match_all — very common in aggregation workloads
- collectRange: entire segment passed as a range — no doc ID enumeration at all
- collect(DocIdStream): intersection result wrapped as BitSetDocIdStream — enables popcount and intoArray
- This is the bridge — without this scorer, neither bulk path on LeafBucketCollector would fire

Key Insight for Aggregations

These primitives were built for search collectors (counting, faceting).

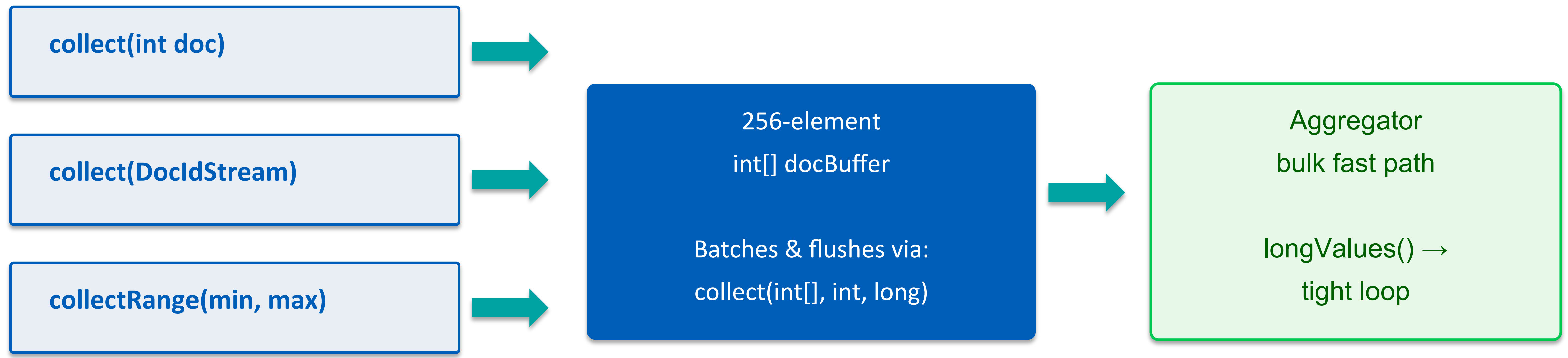
OpenSearch extends them into the aggregation framework — the same batch-oriented philosophy, applied to:

- Bucket assignment (which bucket does this doc go in?)
- Metric computation (sum, avg, min, max over values)
- Sub-aggregation propagation (pass batches downstream)

Bulk Collection in OpenSearch

- LeafBucketCollector
- collect(int doc) → abstract collect(int doc, long bucket (0))
- collect(DocIdStream) ??
- collectRange(min, max) ??

All Entry Points → Single Batch Path



Aggregator implementations only need to override collect(int[], int, long) to get bulk behavior from all three entry points. The framework handles batching.

Key PRs: OpenSearch#19933 • OpenSearch#20009

collectStream & collectRange

collect(DocIdStream, long)

Uses Lucene 10.4's intoArray() to batch doc IDs directly into the buffer, then flushes via the batch path.
No per-doc virtual calls on the stream.

collectRange(int min, int max, long)

Fills buffer from contiguous range and flushes in batches.
Enables bulk doc value retrieval even when scorer provides a range (skip index).

Why this matters:

collect(DocIdStream) → uses intoArray() → batch into buffer → flush
collectRange(min, max) → fill buffer from range → flush
collect(int doc) → buffer single doc → flush when full

All three paths converge to collect(int[], int, long)

Bulk Doc Value Retrieval — The Fast Path

TRADITIONAL — per-doc virtual calls:

```
public void collect(int doc, long bucket)
    throws IOException {
  if (singleton.advanceExact(doc)) {
    long value = singleton.longValue();
    collectValue(sub, doc, bucket,
      preparedRounding.round(value));
  }
}
```

BULK — single call loads all values:

```
public void collect(int[] docs, int count,
                    long bucket)
    throws IOException {
  singleton.longValues(count, docs,
    valueBuffer, Long.MIN_VALUE);
  for (int i = 0; i < count; i++) {
    if (valueBuffer[i] != Long.MIN_VALUE)
      collectValue(sub, docs[i], bucket,
        preparedRounding.round(
          valueBuffer[i]));
  }
}
```

The tight inner loop over primitive arrays is JIT-friendly:
no virtual calls, predictable branching, good cache locality, eligible for auto-vectorization.

Aggregators using this fast path: DateHistogram, NumericTerms, Sum, Avg, Min, Max

Metric Aggregators with Bulk Collection

Aggregator	Bulk Strategy
Sum	longValues bulk load → Kahan summation over buffer
Avg	longValues bulk load → sum + count in tight loop
Min	longValues bulk load → Math.min scan over buffer
Max	longValues bulk load → Math.max scan over buffer
Stats	Per-doc advanceExact in batch + collectRange
ExtendedStats	Per-doc advanceExact in batch + collectRange
ValueCount	Per-doc advanceExact in batch + collectRange
Cardinality	Per-doc ordinal collection in batch + collectRange

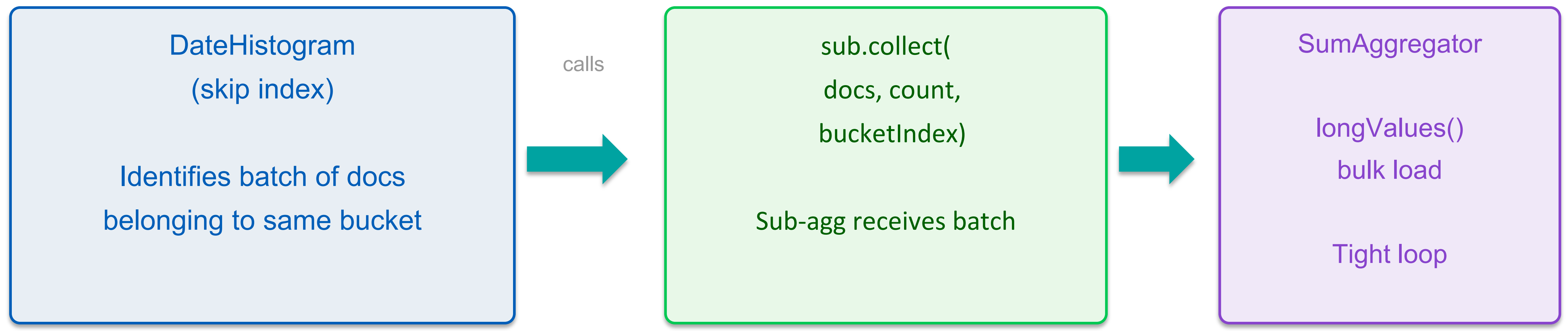
Bucket Aggregators with Bulk Collection

Aggregator	Bulk Strategy
DateHistogram	longValues bulk load → batch rounding → bucket assignment
AutoDateHistogram	longValues bulk load → dynamic rounding
NumericTerms	longValues bulk load → hash lookup bucket
NumericHistogram	Per-doc advanceExact in batch → floor/interval
Filter / Filters	Bits check in batch
Range	Per-doc advanceExact in batch → range matching
GlobalOrdinalsStringTerms	Per-doc advanceExact → ordinal collection
Composite	Per-doc in batch → queue-based collection

Skip Indexes

- Lucene 10's DocValuesSkipper: min/max metadata over 4,096-doc intervals on doc values
- Three decisions per block: skip (outside bucket), bulk-count via popcount (inside single bucket), or fall back to per-doc
- 28x faster on hourly_agg_with_filter benchmark
- Works with any query pattern — not limited to filter field = agg field
- Auto-enabled on @timestamp in OS 3.3

Skip Index + Bulk Collection



The multiplicative benefit:

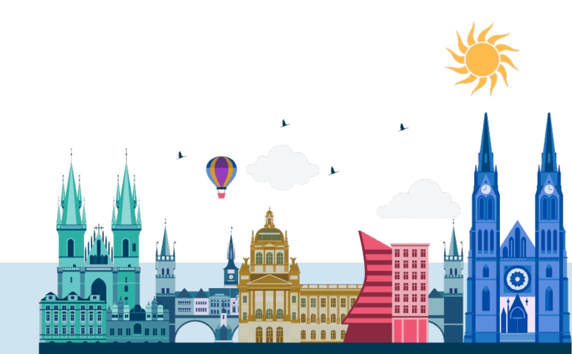
When a parent aggregator (DateHistogram using skip index) identifies a batch of docs belonging to the same bucket, it calls sub.collect(docs, count, bucketIndex). The sub-aggregator (Sum, Avg) receives the batch and uses its own bulk path.

Two levels of batching compound.

Performance Results

Optimization	Version	Benchmark	Improvement
Filter rewrite	OS 2.12	nyc_taxis date histogram	10-50x
Multi-range traversal	OS 2.14	minute-level aggs	up to 100x
Sub-aggregation support	**OS 3.4**	**big5 workload**	**30-40%**
Skip index (date hist)	OS 3.2-3.3	hourly_agg_with_filter	28x
Skip index (auto date)	OS 3.4	range-auto-date-histo	6.5x
PointTree bulk collection	Lucene 10.3	big5 histogram	32x

Storage overhead: ~0.1% for @timestamp only • ~1% for all numeric fields



When It Doesn't Help

❌ Unsorted data

Skip index intervals span many buckets → most blocks fall back to per-doc

❌ Complex nested sub-aggregations

Requiring per-document metric computation on non-numeric fields

❌ High cardinality with sparse data

Many buckets, few docs per bucket

❌ Multi-valued fields

Bulk longValues path requires singleton NumericDocValues

How It All Fits Together

Technique	When It Applies
Star tree	Pre-computed aggregations available
Filter rewrite (BKD tree)	Filter field = agg field, counting only
Multi-range traversal	Same, with many buckets
Skip index (doc values)	Any query pattern, any agg field. Best on sorted data
Bulk collection APIs	General-purpose, amortizes virtual call overhead across all aggregators

OpenSearch automatically selects the best strategy:
star tree → filter rewrite → skip index → bulk collection (always active as baseline)

What's next

- OpenSearch#20031 • OpenSearch#19324

📋	**Extending skip index metadata**	Store sum, value count per block → stats/avg without visiting docs
📋	**Min/max via skip index**	Instant results from skip metadata, zero document visits (#20174)
🔄	**Panama Vector API**	Primitive array processing → natural fit for SIMD
🔄	**Multi-valued field optimization**	Treat as separate types to leverage bulk collection
📋	**Better benchmarking**	Call site pollution for realistic JVM behavior (#709)
📋	**Percentile aggregation efficiency**	#19622

Coordinated Multi-Field Skip Evaluation

```
Query: price:[10,50] AND rating:[4,5] AND date:[2024-01, 2024-06]

Block 0 skip metadata:
  price:  min=70, max=95   → NO  (all prices above 50)
  rating: min=2,  max=5    → MAYBE (overlaps [4,5])
  date:   min=Jan, max=Mar → MAYBE (overlaps [Jan, Jun])
```

Without coordination (today):

rating does per-doc evaluation, date confirms,
then price's skip says NO → wasted work

With coordination (proposed):

Check all fields' skip metadata at block level.
Price says NO → short-circuit. Zero wasted work.

Proposal: Sort fields by selectivity, evaluate skip metadata field-by-field, short-circuit on first NO. For YES blocks, skip per-doc evaluation entirely. No indexing changes required.

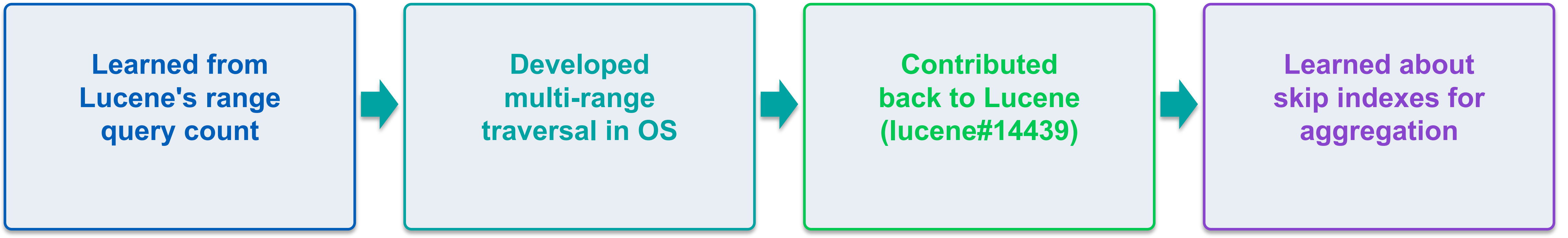
Key Takeaway

Think in BATCHES, not documents.

Think in METADATA, not values.

Filter rewrite showed skipping beats visiting → Skip indexes generalized to any query → Bulk collection ensures even visited docs are processed efficiently → Coordinated skip evaluation extends across fields.

Open Source Cross-Pollination



Now: Coordinated multi-field skip evaluation (lucene#15770) — contributing back to Lucene again.

This demonstrates how strong collaboration across communities can drive shared growth and success.

Notable Links

Blog Posts:

- "Transforming bucket aggregations: 100x"
- "Beyond filter rewrite: Doc value skip indexes"

Further Reading (Lucene internals):

- jpountz — "Why is Tantivy faster than Lucene?"
- jpountz — "Compilation vs. vectorization"
- jpountz — "Why you should configure an index sort"

Lucene Contributions:

- lucene#14439 — PointTreeBulkCollector
- lucene#15737 — Pre-aggregated sum
- lucene#15770 — Coordinated multi-field skip

Key Meta Issues:

- OpenSearch#20031 — Advanced Query Engine Opts
- OpenSearch#19324 — Bulk collection for aggs
- OpenSearch#19384 — Skip list optimization
- OpenSearch#18882 — Skip index plan for 3.x

Previous Talks:

- OpenSearchCon EU 2025: "Transforming Data Aggregation: Achieving 100x Faster Performance"
- OpenSearchCon US 2025: ”From Bottlenecks to Breakthroughs: Optimizing Aggregations in OpenSearch for Lightning-Fast Analytics
- OpenSearchCon CN 2026: "Leveraging Lucene Skip Indexes for Faster Aggregations in OpenSearch”

Thank You!

Questions?