Elasticsearch vs Vespa Performance Comparison

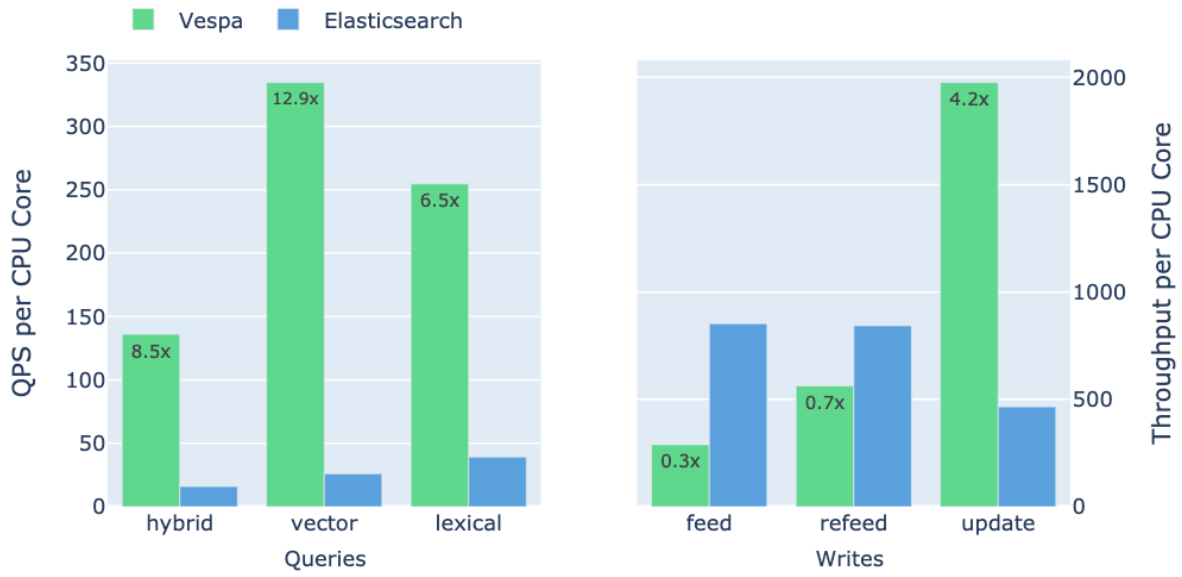
Geir Storli, Jo Kristian Bergum, Radu Gheorghe
November 2024

Vespa

1. Executive Summary

This report presents a reproducible and comprehensive performance comparison between Vespa (8.427.7) and Elasticsearch (8.15.2) for an e-commerce search application using a dataset of 1 million products. The benchmark evaluates both write operations (document ingestion and updates) and query performance across different search strategies: lexical matching, vector similarity, and hybrid approaches. All query types are configured to return equivalent results, ensuring a fair, apples-to-apples comparison.

Vespa demonstrates significant query performance, scalability, and update efficiency advantages.

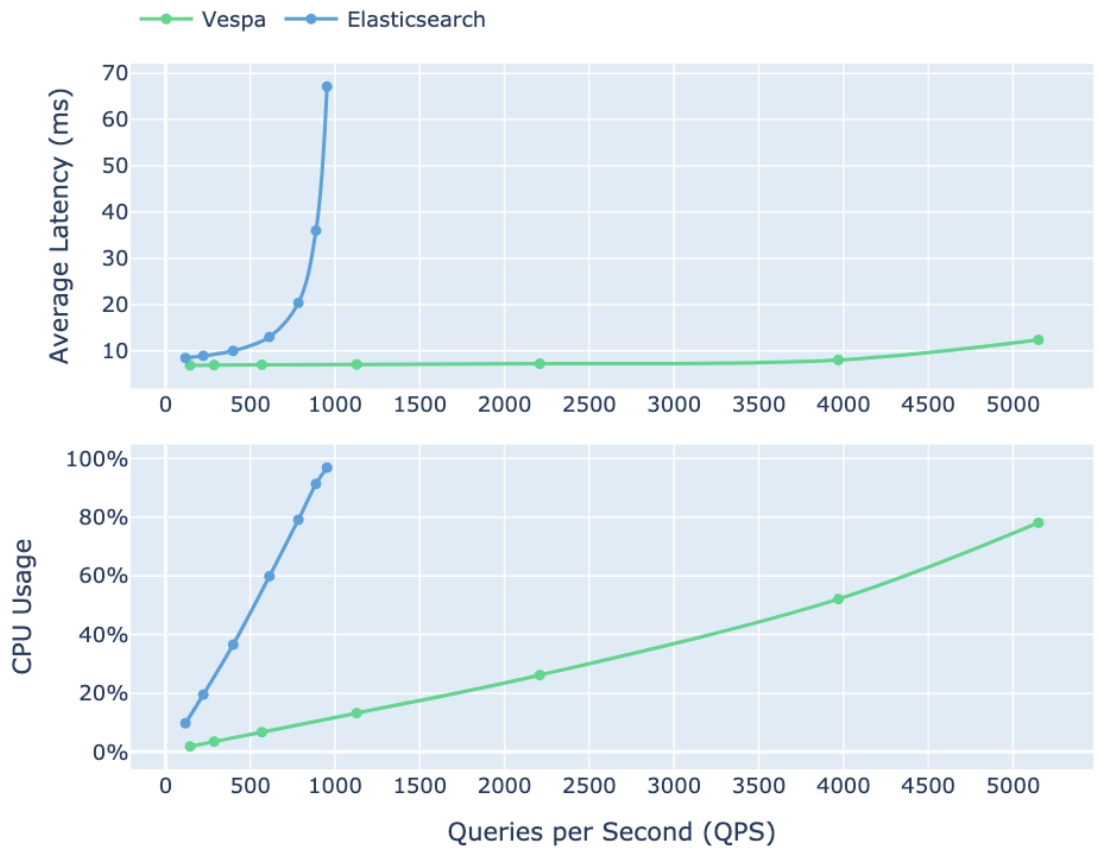


The illustration shows queries per second per CPU core for different query types and reveals Vespa's significant query efficiency advantages:

- **Hybrid Queries**: Vespa achieves 8.5x higher throughput per CPU core than Elasticsearch.
- **Vector Searches**: Vespa demonstrates up to 12.9x higher throughput per CPU core.
- **Lexical Searches**: Vespa yields 6.5x better throughput per CPU core.

Updates: Vespa is about 4x more efficient than Elasticsearch for in-place updates. While Elasticsearch demonstrates high efficiency during the initial write (bootstrap from 0 to 1M) phase, Vespa excels in steady-state operations, handling queries and updates more efficiently.

The following illustration compares how both systems handle hybrid queries, showing the relationship between latency, CPU usage, and query throughput as user concurrency increases.



Vespa shows higher CPU efficiency, demonstrated by a lower CPU usage gradient compared to Elasticsearch. This superior performance efficiency directly reduces infrastructure costs, as demonstrated in section 10, where the efficiency improvements yield **5x reduction in infrastructure costs** on a typical deployment (see section 10).

The findings of this report align with feedback from organizations that have migrated their e-commerce solutions from Elasticsearch to Vespa. In "Vinted Search Scaling Chapter 8: Goodbye Elasticsearch, Hello Vespa Search Engine", they summarize their migration experience with the following:

The migration was a roaring success. We managed to cut the number of servers we use in half (down to 60). The consistency of search results has improved since we're now using just one deployment (or cluster, in Vespa terms) to handle all traffic. Search latency has improved by 2.5x and indexing latency by 3x. The time it takes for a change to be visible in search has dropped from 300 seconds (Elasticsearch's refresh interval)

to just 5 seconds. Our search traffic is stable, the query load is deterministic, and we're ready to scale even further.

2. Table of Contents

3. Preface

We benchmark software to establish **performance expectations**. The usual reason is cost: deploying a more efficient technology will cost less. However, this cost reduction can have other implications, like unlocking otherwise prohibitive features or making clusters easier to scale and manage.

This benchmark compares Elasticsearch and Vespa under identical use case scenarios and workloads. Our analysis focuses on an e-commerce search application encompassing lexical, vector, and hybrid queries. In the context of e-commerce search, three performance factors (beyond relevance) are essential:

- Query Latency: How quickly does the search engine respond?
- Throughput: How many searches can the engine handle at once?
- Efficiency: How much computing power (CPU) does the engine use?

Efficiency is vital because, in both Elasticsearch and Vespa, you can increase throughput and reduce latency by using more resources. Search is a parallelizable workload that can be distributed across multiple CPU cores to achieve speedups or reduced query latency (***"faster"***).

A Comparative Example

Let's look at two hypothetical search engines running on identical hardware (16 CPU Cores):

- Engine A - 100ms latency using 2 CPU cores.
- Engine B - 50ms latency using 8 CPU cores.

We could say that B is *faster* (it is!) and leave it with that without mentioning that B uses 4x the resources compared to A. But what if we could make A serve at 50 ms using 4 CPU cores? Then, we could reduce CPU-related costs by 2x using A instead of B while maintaining the 50 ms latency.

This example shows how easy it is to create misleading benchmark reports. By omitting resource usage metrics, we might draw incorrect conclusions about which engine is truly more efficient.

Throughout this benchmark, we compare the engines' performance fairly, transparently reporting resource usage and employing realistic workloads. Both engines perform identical tasks (see 6.6), returning nearly the same results for equivalent queries (see 6.3.4). Whenever we report throughput, we also normalize with regard to the amount of CPU resources consumed to achieve it (see section 7). Our approach to benchmarking requires establishing a common baseline of features and capabilities across the two engines compared. We detail the configuration of both engines, the experimental setup, and the realistic use case workloads (see section 6).

Last but not least, **benchmarks must be reproducible**. This benchmark runs daily in the Vespa performance system test framework, and you can find everything you need to run it yourself in section 11.

Let's discuss everything mentioned above, starting with the use case: which dataset we used and what workloads we used. Then, in section 5, we'll take a deep dive into the architectures of both engines before looking at the test configuration (section 6) and results (sections 7 and 8).

4. Benchmark Use Case & Dataset

4.1 Dataset

In this benchmark, we create an e-commerce use case based on the Amazon Reviews 2023 dataset from McAuley-Lab. This dataset contains 571.54M user reviews and 48.19M products across 34 categories. We select 1M products by random from the dataset to form our e-commerce document collection and extract the following metadata for each product:

- Product Title
- Product Description
- Product Category
- Product Average rating
- Product Price

In addition, we generate a text embedding vector using a concatenation of the title and description. We use a small text embedding model for this benchmark: Snowflake's Arctic-embed-xs. This model has 22M parameters and produces embedding vectors of 384 dimensions. We do not include embedding model inference in the benchmark.

Duplicate products are removed, and descriptions are cleaned up before selecting 1M products randomly. See the e-commerce hybrid search data prep for details. We use a small sample of 1M products because the performance test runs daily as part of the standard CI/CD procedure. A larger sample size would delay the pipeline and developer feedback cycle. Here we show an example product document in Vespa JSON format so you can get a feel for the data:

```
Unset
{
  "put": "id:product:product::Cell_Phones_and_Accessories73",
  "fields": {
      "id": 73,
      "title": "Samsung Galaxy Note 9 Wallet Case, Jaorty Premium
Leather Folio Flip Case Cover Book Design with Kickstand Feature with
Card Slots/Cash Compartment for Samsung Galaxy Note 9 - Red",
      "category": "Cell_Phones_and_Accessories",
      "description": "Premium LEATHER MULTI-FUNCTIONAL VIEW FROM ANY
ANGLE CONVENIENT TO USE UPDATED SERVICE",
      "price": 1269,
      "average_rating": 5.0,
      "embedding": [...]
  }
}
```

We use a random sample of 10000 product titles as user queries. Only the first 8 words of the product title are used, to mimic users searching for specific products. You can find more information on how these queries look like in section 6.3.

4.2 Workloads

The following write and query workloads are most relevant for e-commerce search systems.

4.2.1 Write workloads

Initial writing of product documents

This represents the scenario where we have an empty index and populate it with data from scratch, bootstrapping the index from 0 to 1M products.

Updating the price of each product by rewriting all documents again

This represents the scenario where we rewrite from the source of truth while we have an existing index that is serving query traffic. In e-commerce search, there is usually a constant stream of inventory updates, plus a periodic (e.g. daily) refresh from the source of truth.

Updating the price of each product by partial update.

This represents the streaming scenario where only specific fields are updated. This could, for example, be price or inventory status (in stock) but also include fields that are used for ranking, e.g. based on click streams.

4.2.2 Query workloads

Users searching for relevant products by using different types of queries:

- **Lexical (keyword) search** over the title and description fields.
- **Vector search** using text embeddings. Powered by approximate nearest neighbor search over the dense vector representations of the products.
- **Hybrid search** - a combination of the above query types.

For all three query types, we also benchmark with a filter on category. Making it a total of six query type workloads. Each query type orders (ranks) products using a scoring function. Queries are benchmarked when the system is idle without writes and with ongoing writes (concurrent reads and writes).

Before diving into the benchmark results, let's examine the architectural differences between Elasticsearch and Vespa: specifically, how they handle data ingestion and execute queries.

5. Elasticsearch vs Vespa Architectural Differences

We'll focus only on this benchmark's areas: writing to and reading from (querying) an index. Both engines strive to strike a balance between query and write performance.

We think that an architecture deep dive is essential to understand how both engines handle the trade-offs between indexing and querying, ultimately influencing their performance characteristics for various workloads. Which is why this section is about ¼ of the whole report. That said, feel free to navigate to the benchmark details (section 6) and results (section 8) and return to this section to gain deeper insights into the architectural differences.

5.1 Write Path

Elasticsearch and Vespa write to a transaction log for persistence and durability while writing to internal buffers and data structures on disk. But there are significant differences in:

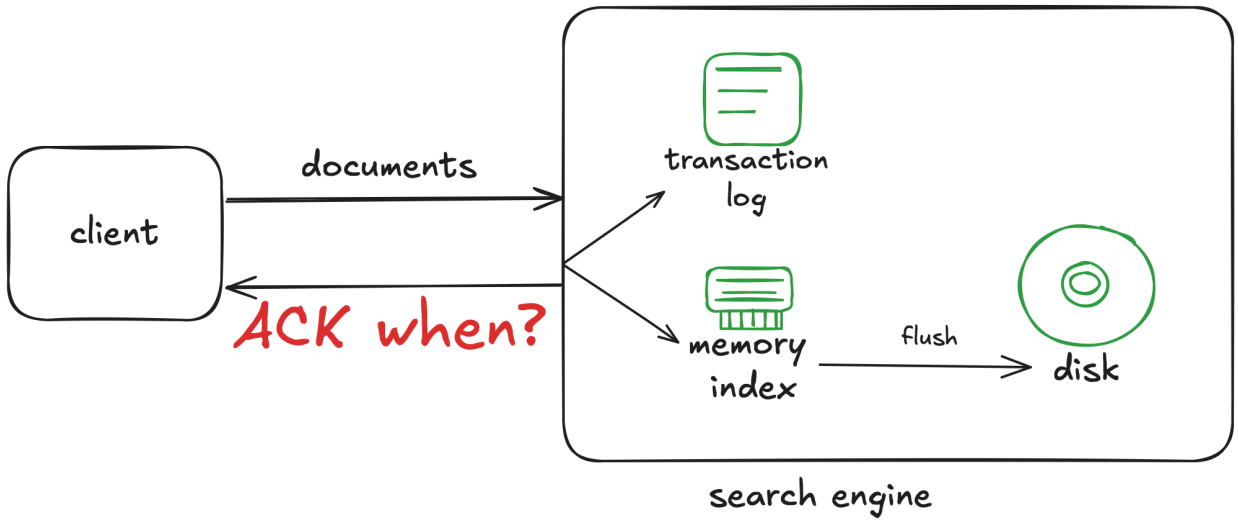
- How client applications write data over HTTP(S)
- How they approach eventual consistency
- Underlying data structures

5.1.1 Write APIs

Typically, clients write to Elasticsearch using the Bulk API. The payload is newline-delimited JSON (NDJSON). Each operation has a metadata line and a data line unless it's a delete action. The top-level response indicates if there's an error. If it is, the detailed part of the response gives error details, just like with individual indexing requests.

With Vespa, clients always write documents individually, there is no batch or Bulk API equivalent. Throughput can be achieved via HTTP/2 multiplexing, allowing multiple requests to be sent simultaneously over a single connection. The Vespa Feed Client library uses HTTP/2 and it's already powering integrations like the Logstash Output for Vespa and the Kafka Connect Vespa Sink.

5.1.2 Real-time

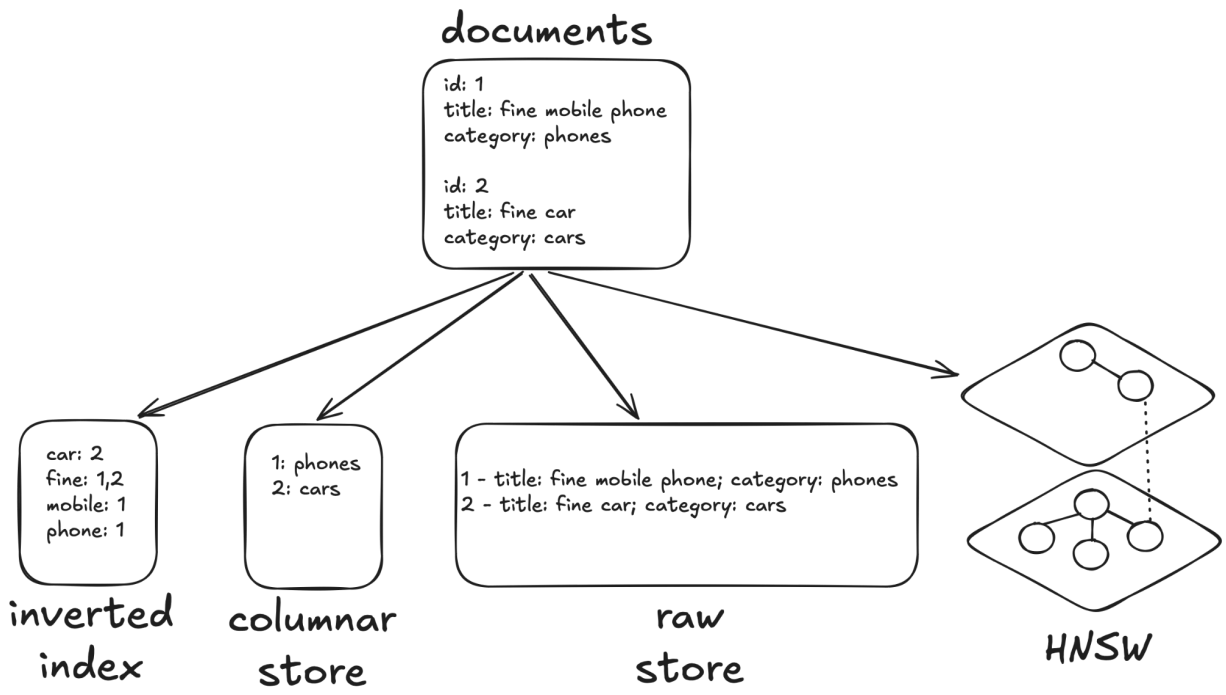


When Vespa returns an acknowledgment of a write operation to a client, the operation has been written to the transaction log (TLS), *and* the result is visible in searches. In contrast, Elasticsearch returns the acknowledgment when the document is written to the TLS. Documents are *eventually available* for search after the next refresh operation.

The Elasticsearch refresh mechanism allows batching of document operations but at the cost of up to a *refresh interval* in indexing latency. If the use case requires **serving data immediately** in real-time (e.g., I uploaded a new product to a marketplace and want to see it there), you then must use a short refresh interval.

Refreshes are asynchronous, but they aren't free. **Getting the latest version of a document** requires either an index refresh or a search combined with a transaction log lookup. While we didn't benchmark Get performance, this impacts updates, as discussed later in this section.

5.1.3 Data structures



At a macro level, Elasticsearch and Vespa employ similar fundamental data structures for hybrid search functionality, though their technical implementations vary substantially. Let's explore the fundamental data structures with broad strokes.

Inverted indices

Inverted indices are the cornerstone of efficient text search in both Elasticsearch and Vespa. They power lexical searches, supporting retrieval (and scoring) of documents containing text tokens. An inverted index consists of two main components:

- Dictionary: A sorted list of unique tokens in the indexed documents.
- Postings: For each token in the dictionary, a list of documents containing that token. Possibly, other metadata (e.g. positions).

During feeding/indexing

- Documents are broken down into tokens.
- Each unique token is added to the dictionary.
- The document ID is added to the postings list for each unique token.

During search

- The search term(s) is looked up in the dictionary.
- The corresponding postings list provides quick access to all documents containing that term.

This structure allows for text searches, as the system can quickly identify which documents contain the queried tokens without scanning every document.

Columnar store

Both Elasticsearch and Vespa use a columnar store to support efficient retrieval and manipulation of field values across multiple documents. For example, during sorting, faceting or ranking. In Elasticsearch, this data structure is known as "doc values". In Vespa, it's called "attributes".

Raw document store

This component stores the original document data:

- In Elasticsearch: Known as the _source field or stored fields.
- In Vespa: Called document summaries.

Purpose:

- Allows retrieval of the full original document or specific fields.
- Supports highlighting of search terms in results.
- Enables reindexing and redistribution without reading from an external data source.

HNSW index for Approximate Nearest Neighbor Search (ANN)

Both Elasticsearch (via Apache Lucene), and Vespa implement the HNSW (Hierarchical Navigable Small World) algorithm for efficient ANN searches. HNSW is a graph-based ANN algorithm that creates a multi-layer structure to enable fast *approximate* nearest neighbor search.

- Hierarchical structure: Multiple layers of graphs, with the top layer being the sparsest and the bottom layer the densest.
- Navigable: Designed for efficient traversal to find nearest neighbors quickly.
- Small World: Exhibits the small-world property, where most nodes can be reached from every other node by a small number of hops.

During feeding/indexing

The HNSW indexing process is an operation that involves inserting vectors into a multi-layer graph structure. As each vector is added, it undergoes similarity comparisons with existing vectors to determine its position within the graph. This process, along with subsequent search operations, requires graph traversals that involve random access to vectors already in the graph. Due to the need for rapid and repeated access to vector data for similarity computations, **the entire dataset should reside in primary memory (RAM) to ensure efficient performance**. Many developers are surprised to learn that writing to the graph structure requires computational effort similar to performing a search operation, as both processes involve traversing and evaluating nearest neighbors.

During Search

A nearest neighbor search starts at the top layer and gradually traverses the graph and layers. At each layer, it navigates to the nearest known points to the query vector. The respective HNSW implementations allow both Elasticsearch and Vespa to perform efficient vector similarity searches.

Fundamental data structure differences

The core differences between Elasticsearch and Vespa are about how these fundamental data structures are created and accessed. Elasticsearch maintains all of them in self-contained immutable Lucene segments. In contrast, Vespa maintains a mix of both immutable and mutable data structures, depending on the schema field types and indexing parameters.

Mutable versus immutable data structures

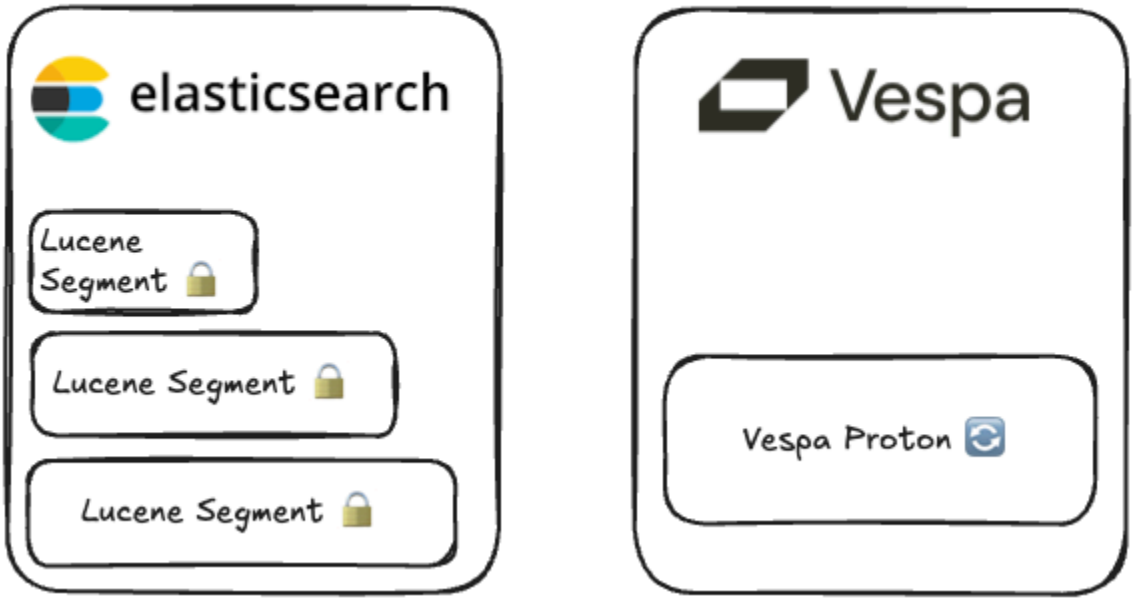
When building search structures like inverted indices or HNSW graphs, the choice between mutable and immutable approaches presents distinct tradeoffs. **Mutable structures allow for dynamic updates**, enabling the index or graph to evolve as new documents or vectors arrive. This approach supports real-time indexing but requires careful handling of concurrency to prevent race conditions, particularly in HNSW graphs where neighbor connections need to be updated atomically.

In contrast, immutable structures require batch processing to ingest a set of changes. While this approach offers advantages like known document counts and simpler parallel processing, it introduces latency between writes and their visibility. When new content is added, it won't be searchable until the next batch processing cycle completes and a new immutable structure is created. This write-to-visibility delay can be significant for

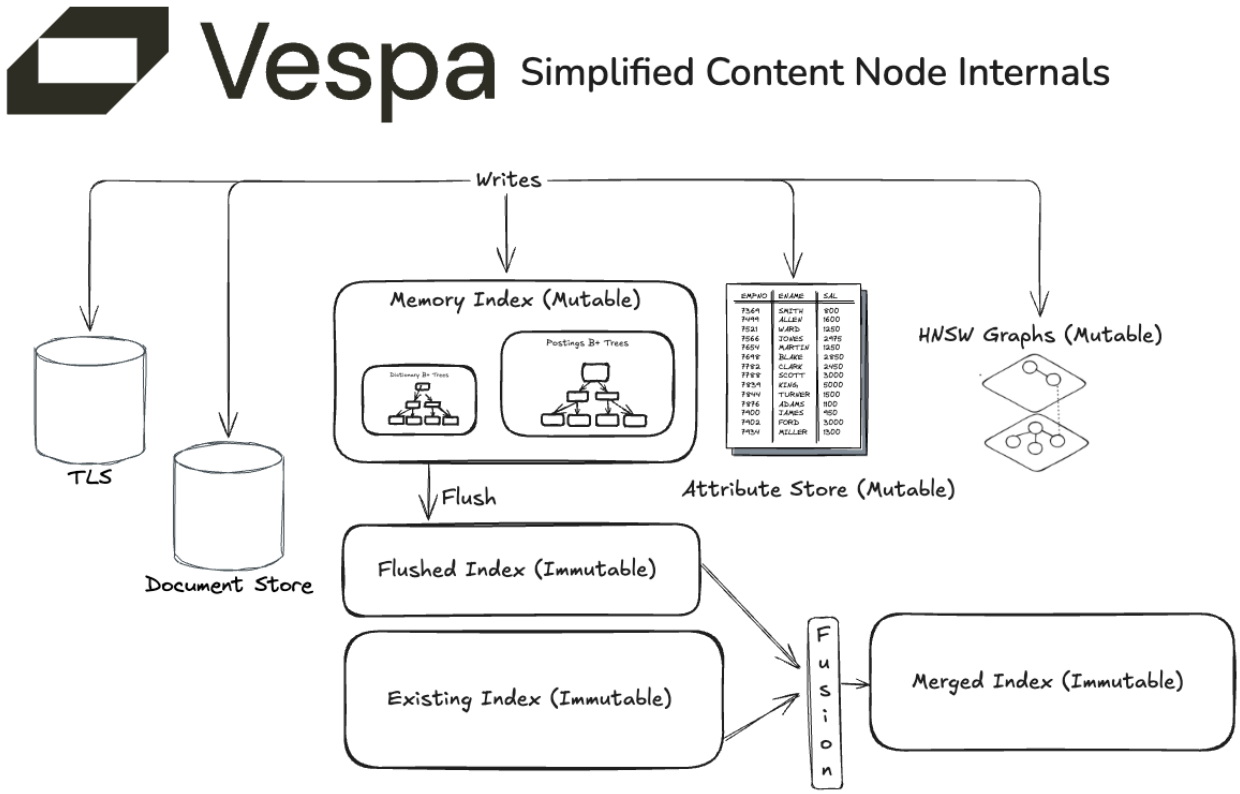
use cases requiring real-time search capabilities, such as news platforms, social media feeds, and e-commerce sites.

Query performance also differs significantly between these approaches. Immutable structures typically exist as multiple segments or partitions, requiring queries to search across all of them and merge results. While this can enable efficient parallel processing, the overhead of searching multiple segments and merging results can impact query performance compared to a single, mutable structure.

The fundamental difference lies in the approach: building a search structure for N items as a single batch versus incrementally constructing it as items arrive from 0 to N. This choice influences not just the write path, but also query performance and system architecture. We'll expand on both approaches below.



Elasticsearch is built on Apache Lucene, a powerful open-source search library that can also be used independently for custom search implementations. In contrast, Vespa uses its own search engine architecture that, while feature-rich, is designed as a monolithic system and cannot be decomposed into separate library components.



A simplified illustration of Vespa's content node (Proton) internals.

The content node in Vespa processes writes in a multi-layered approach. When a write operation occurs, the data is first recorded in the **Transaction Log Store** (TLS), which ensures durability of all operations, and the **Document Store**, which maintains the primary storage of documents. Tokenization and other document operations happen in the Vespa stateless layer.

From there, the write propagates to several (possibly) mutable components. The **Memory Index**, which uses B+ trees for efficient organization of both dictionaries and postings, temporarily stores these updates before they're flushed to disk.

The **Attribute Store**, which operates like mutable in-place tables, keeps structured data readily available for quick access and updates. Alongside these, the **HNSW Graphs** component, also mutable, maintains graph structures used for approximate nearest neighbor search. The vector data (or tensors, in general) are stored in the attribute store and not in the graph structure.

The system manages long-term inverted index storage through a process of flushing and merging indexes. When the Memory Index needs to be cleared, it flushes its contents to create what's called a Flushed Index. This **Flushed Index is immutable**, meaning it can't be changed after creation. It then goes through a **fusion process with the Existing Index** (previously flushed and merged indices) to create a new Merged Index, which is also immutable. Note that the Memory Index and Flush Index are only relevant for string fields with indexing enabled. The Attribute Store and HNSW graphs are maintained separately and they are mutable.

While this illustration focuses on write operations, queries can interact with multiple system components depending on their complexity. For example, a hybrid query might access the memory index, the flushed fused index, and both the HNSW graph and attribute store. The only component not involved in queries is the Transaction Log Store (TLS). See more in Vespa documentation (Proton).

Apache Lucene segments

Lucene-based engines, such as Elasticsearch, utilize immutable segments to write data structures to disk. An index comprises one or more such segments. Once written, a Lucene segment remains unalterable. To achieve near real-time indexing (add, remove, update), Elasticsearch uses the following procedure:

- A new segment is generated upon writing a batch of documents.
- Each segment incorporates all the data structures necessary for querying: inverted index, columnar store, raw document store, and HNSW graphs for dense_vector fields.
- During query execution, all the segments of an index (roughly equivalent to a document type in Vespa) are searched and the results are merged.

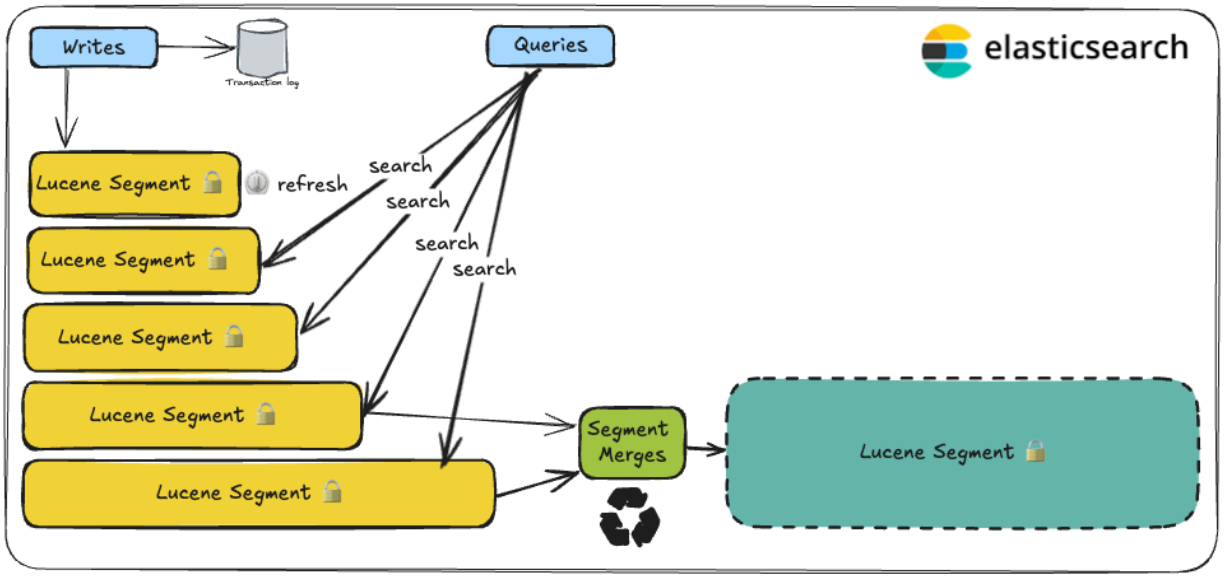
Segments maintain immutability. Consequently, document deletion from a segment results in a soft delete, where the document ID is appended to the segment's deletion list. Deleted documents persist in internal searches but are filtered using the deletion list.

Similarly, updating a document by partial update is implemented by soft-deleting the document that is updated and writing the updated version of the document to a new segment. Hence, partial updates follow a read-update-write pattern. In Vespa, only some partial updates require reading the latest version of the document, applying the update, and writing back the result. In Elasticsearch, every partial update operation follows this read-update-write pattern.

This read-update-write pattern can be a performance bottleneck for partial updates for Elasticsearch. Elasticsearch can't tell if a get request returns the last version of the document unless it looks up in the transaction log or forces a synchronous refresh which blocks the get request.

Segment merges

Maintaining an optimal number of segments in Elasticsearch is important to prevent search performance degradation and inefficient disk space usage. To reduce the number of active segments, background segment merges occur periodically. As described in this seminal blog post, this process involves combining N immutable segments to create a new segment. A typical index structure includes a mix of segment sizes, ranging from small to progressively larger, resembling the following pattern:



This illustration shows the architecture of Elasticsearch, depicting the flow of data through its core components. Document operations are written to a transaction log for durability and indexed into immutable Lucene segments. The diagram demonstrates how queries search across all segments, while a background process merges smaller segments into larger ones for improved query performance.

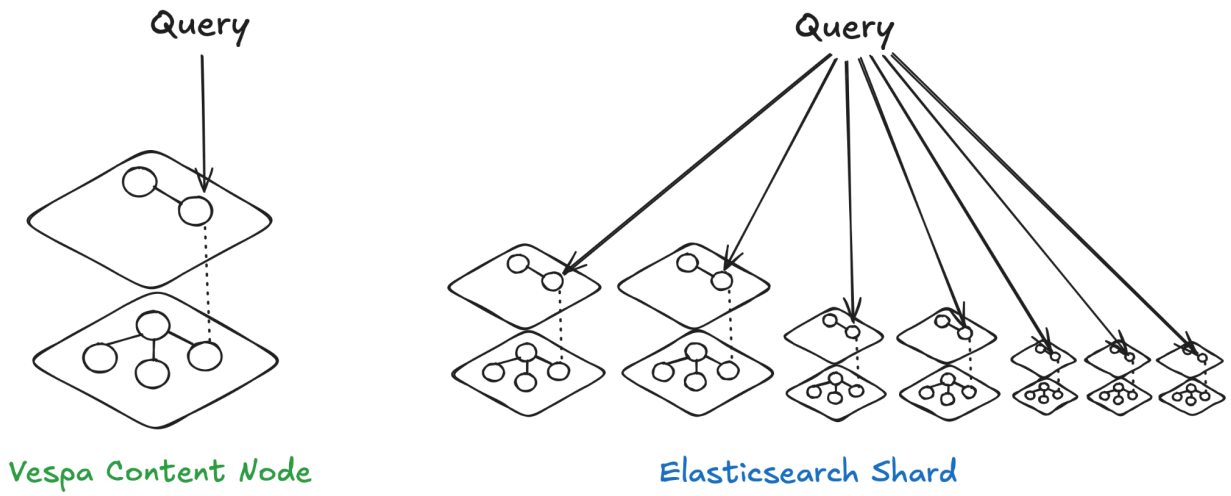
The benchmark covers typical workloads and configurations used in practice:

1) Continuous near real-time indexing using the **default merge policy**.
2) Index **force merged to one segment**. This fits the "nightly reindex" pattern. E-commerce applications often update their product catalogs nightly. This process typically involves creating a new index and force-merging it into a single segment before serving it to users. The transition to the latest index is usually accomplished by updating the alias used by queries. While force-merging is resource-intensive, consuming significant time, CPU, and I/O capacity, it generally proves beneficial for query performance in scenarios where the index remains effectively read-only throughout the day.

You can tweak the number of segments for case 1 by adjusting the merge policy. These knobs allow you to control the number of deletes, how much to favor large vs small segments, and the upper segment size limit. We only benchmark using the default merge policy because:

- **The optimal merge policy depends on the specific use case**. Factors such as the document updates (including the generation of deleted documents), the volume of updates and indexing latency.
- **Most don't adjust the merge policy**. The last Elasticsearch documentation for merge policy is about ten years old (version 1.4). The argument was that these settings were too trappy. But if you're curious, there are quite a few resources to help you understand and tweak the Lucene merge policy, especially around Solr (that can be translated to Elasticsearch). Benefits can be significant, as demonstrated in this Elastic blog post.

Why do we spend so much time on segments? Because they significantly impact query performance. To power vector and hybrid searches, Lucene maintains **one HNSW graph per segment** for each dense_vector field to serve approximate nearest neighbor searches. By contrast, **Vespa maintains a mutable HNSW graph per tensor field** (with index enabled) per content node, which means it behaves similarly to an Elasticsearch index with one segment (force-merged).



Vespa maintains one mutable HNSW graph per tensor field, while Elasticsearch maintains one HNSW graph per segment per dense_vector field.

There are two aspects that impact query and write performance:

- Querying multiple HNSW graphs, one per segment per dense_vector field
  In Elasticsearch 8.10, the default search strategy changed from a single thread searching segments sequentially to concurrently, using multiple threads (see section 5.2.1). This reduces wall time and speeds up the search ("faster"), at the cost of CPU usage per query. A benefit of searching the segments concurrently compared to in sequence is that search threads can exchange information, as detailed in this blog post.
- Merging of multiple HNSW graphs during segment merges
  Merging inverted index data structures for lexical search, where dictionaries and posting lists are sorted, has low computational complexity. **Merging HNSW graphs from two or more segments into a single graph is more complex**. The current merge implementations will use the largest segment's HNSW graph as a seed graph to avoid building the merged graph from scratch.

In a Vespa content node, you have memory indexes flushed to disk occasionally (similar to Elasticsearch's flush). A flush triggers a disk index fusion with the existing index for that field on the disk. Meanwhile, Elasticsearch will have one instance of all data structures in every Lucene segment. Especially with the default merge policy, Vespa will generally touch fewer "partitions" of data structures during query execution than Elasticsearch.

5.1.4 Threading

Both Vespa and Elasticsearch have separate thread pools for write and read workloads. In Elasticsearch, the write thread pool can use all the available CPU cores by default.

In Vespa, the main task of writing (e.g., creating inverted indexes and dealing with their compactions, much like Lucene merges) is done by Feed Writer threads, which default to half the available CPU cores. We didn't increase it from there because other thread pools are also handling writes. For example, writing document summaries to disk (i.e., raw data, like _source in Elasticsearch). Generally, altering Vespa's mutable data structures per field requires synchronization, meaning that one thread per field can mutate the data structure.

Similarly, both engines have search-related thread pools. But there's more to the queries than just threads, so let's zoom in.

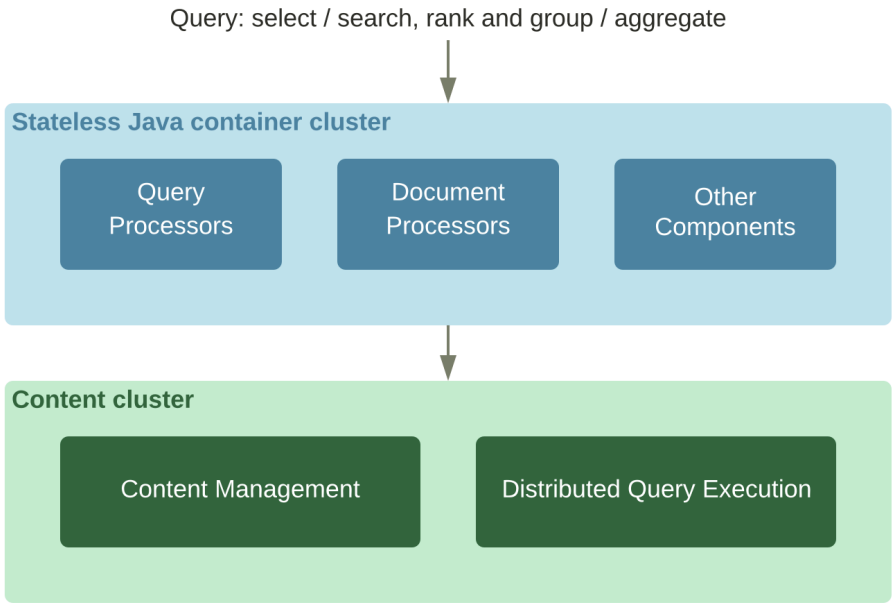
5.2 Query Path

As with indexing, search thread pools differ between the two search engines. Let us examine those before discussing how Elasticsearch and Vespa handle the query types this benchmark includes.

5.2.1 Threading

In Elasticsearch, with concurrent segment searching enabled (the default since 8.10.0 and which can only be disabled by an undocumented setting), it's the `search` thread pool's job to coordinate queries across different shards. Then the `search_worker` thread pool runs searches concurrently across active segments. The size of each search thread pool defaults to about 1.5 times the number of available CPU cores.

Vespa has a slightly different approach. Query coordination happens in a JVM process (stateless container node), which is separate from the native C++ process that deals with matching and most phases of ranking (content node). You can think of these as Elasticsearch coordinator and data nodes, respectively. You'll find more information on how concepts translate between Vespa and Elasticsearch in this glossary blog post, but here, we'll concentrate on the relationship between container and content nodes:



For queries, Vespa has these thread pools:

- The container process has a worker thread pool, similar to Elasticsearch's `search` thread pool. Its size defaults to about the number of CPU cores and is rarely a bottleneck. More commonly, heap sizing and GC are something to tune. It's a similar situation with Elasticsearch coordinator nodes.
- The content node has configurable search threads, similar to the `search_worker` thread pool in Elasticsearch. By default, there are 64 workers, but this number might need tweaking based on hardware and workload.
- In the content node, you can also configure the number of threads to be used per search, which rank profiles can override. Rank profiles define how to compute relevance scores. This allows for intra-query concurrency, where a single query can use multiple threads to retrieve and rank documents. While Elasticsearch supports concurrently searching multiple segments, an intra-segment search cannot currently use more than one thread (This is now supported in Apache Lucene 10).
- There's a separate summary thread pool in the content node. It deals with fetching raw summary data (after identifying the global top-k ranked documents). We don't use it in this benchmark because we only return document IDs, not the raw documents (in both Vespa and Elasticsearch), to reduce the number of variables.

Now that we've covered threading, let's examine the different query types run by this benchmark and how each engine handles them.

5.2.2 Lexical search (accelerated by Weak AND)

In an e-commerce setup, users search for multiple words (e.g., "Nokia 3310 blue"). The definition of a "word" is fuzzy, depending on how linguistics is set up (in Elasticsearch, analysis), so we'll refer to the final search terms as "tokens".

Consider a query for N tokens. If N is large, the query becomes expensive, as many documents will match at least one of these tokens. **If we use the OR operator, many documents must be scored to determine the most relevant**.

Vespa and Elasticsearch implement variants of a pruning algorithm called "Weak AND" (references here and here) to make OR searches more efficient. This algorithm produces the same top N list as a full OR evaluation using an additive scoring function over the terms, without computing most of the scores.

5.2.3 Vector search using Approximate Nearest Neighbor (ANN) search

"Semantic search" has become synonymous with ANN. Here's the rough idea:

- **Run text through a text embedding model that generates a dense vector representation** (e.g., using the product title and description as the input). If the model fits the use case, vectors should capture semantics well.
- **Index those vectors** along with the original text in Vespa or Elasticsearch.
- **Run the query string through the same embedding model** to produce a dense vector representation of the query text.
- **Documents with vectors closer to the query vector should be more relevant.**

Computing the distance between the query vector and all document vectors **(exact K nearest neighbor search) is expensive**. Exact KNN only makes sense if other parts of the query restrict matching documents to less than about 5% of the full corpus. To speed up nearest neighbor queries, Vespa and Elasticsearch both use HNSW graphs.

5.2.4 Hybrid search

Both vector and lexical search have their pros and cons. For example, vector search tends to have better recall, while lexical search tends to have better precision. This is why many deployments use both and combine their scores for better overall relevance.

Vespa is flexible when it comes to combining scores. The most popular one is Reciprocal Rank Fusion (RRF). Elasticsearch also implements RRF through retrievers, but at the time of writing, it's in technical preview. Since we're not focusing on relevance, but we want to benchmark identical functionality, **we're simply adding the lexical and vector search scores** in Vespa's hybrid rank profile. This is the equivalent of a boolean query in Elasticsearch.

5.2.5 Pre-filtering and post-filtering

Most e-commerce searches have filters (e.g., by product category). While using filters in lexical search is very straightforward (another query clause that doesn't count for ranking), combining it with ANN search is more challenging. That's because the HNSW graph doesn't store data about the product category (as in this use case). So, how can we extract only the nearest neighbors matching the filter criteria?

There are two solutions to this: pre- and post-filtering. In short, **pre-filtering is more precise, while post-filtering is faster**.

Pre-filtering implies that when a node is visited in the HNSW graph, **the filter to tell whether the node can be considered a hit has already been applied**. This is done for every node as the graph is traversed until the target number of hits is met. This makes the query functionally complete but expensive because the filter has to accumulate all matching documents, and there's a double-check step for every visited node in the HNSW graph.

Post-filtering **runs the ANN search first, then checks the top N results against the filter**. Which is efficient, but there's no guarantee that you'll get the requested number of hits or that you will get any at all. Depending on the use case, this might be acceptable, especially if you over-fetch the number of results from the ANN search.

Both Vespa and Elasticsearch support pre- and post-filtering of vector search. There are some implementation differences, though. Let's go over the main ones.

Vespa allows post-filtering to kick in based on a configurable hit ratio. For example, suppose the filter matches 80% of the whole dataset (or, even better, if we predict that it does before even running it). In that case, we can be confident that we get enough results, especially if we over-fetch. This behavior is disabled by default. We kept it disabled for this benchmark, as there's no equivalent in Elasticsearch.

If only a few documents match the filter, it's likely faster to simply iterate through them (i.e., do an exact nearest neighbor search). Vespa checks this in two steps:

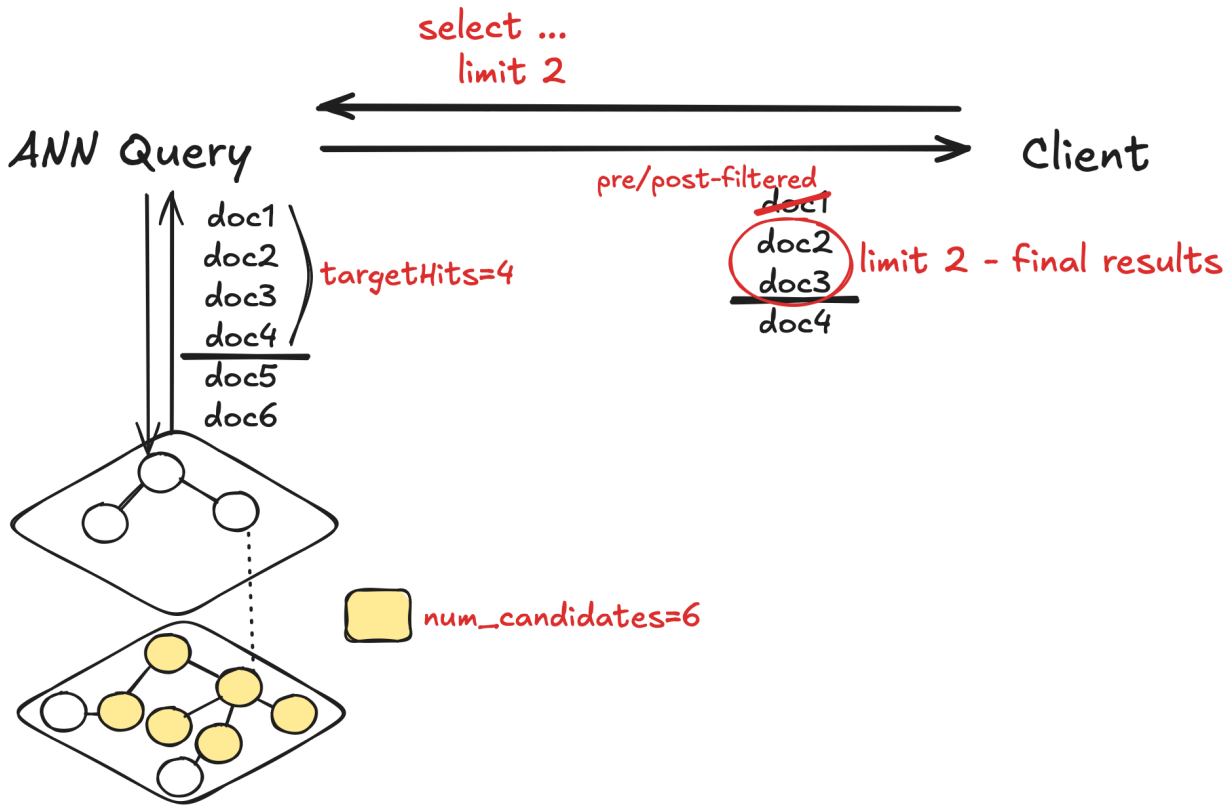
- As an estimation, before running the pre-filter.
- After running the pre-filter.

If there are fewer matching documents than a configurable threshold, Vespa runs an exact KNN search. You can find an extended explanation of Vespa's pre- and post-filtering behavior in this blog post. Meanwhile, Elasticsearch falls back to the exact KNN in two scenarios:

- When it has already visited more HNSW nodes than the number of documents matching the filter.
- When the number of documents from the filter is smaller than K.

Note that K (as in "K nearest neighbors") has a different meaning depending on the context. There are three such contexts:

1) **Top K vectors visited in the HNSW graph**. This is the K mentioned in the last bullet for Elasticsearch, defined by num_candidates in the Knn query. For Vespa, this is targetHits + hnsw.exploreAdditionalHits. Because the HNSW graph is inherently approximate, we can trade performance for precision by fetching more vectors.
2) **Top K vectors returned by the ANN query**. These come from computing the distance between the query vector and the vectors visited in the previous step. This is `k` in the Knn query in Elasticsearch and `targetHits` in Vespa.
3) **Top K final documents to return to the client**, considering other filters, post-filters, etc. That is the first page of the results. This is the `size` parameter in the Elasticsearch request body and the limit/hits in Vespa.



When running the benchmark, we've calibrated all those "K" parameters to be the same for both engines to level the playing field. For example, `k` in Elasticsearch is set to the same value as `targetHits` in Vespa, and so on. You'll find the exact queries in section 6.3.

5.3 Other Aspects

Even though there's quite a lot of information above, there's much more to Elasticsearch and Vespa architecture differences that we didn't cover here. It's not that they don't matter in practice, but considering them will increase the number of variables we can control during a benchmark. They may well be the subject of other benchmarks.

Both Vespa and Elasticsearch can scale horizontally by adding more nodes. The distributed models vary greatly, offering pros and cons for each side. We eliminated this variable here by running benchmarks on a single node.

In Elasticsearch, you can have multiple indices, which are discrete units of data and configuration. Vespa has document types that provide a similar separation. Each Elasticsearch index can be divided into **multiple shards that can spread out on multiple**

nodes. Vespa can distribute data more granularly, where the storage layer on each content node comprises **many buckets, which are also spread between nodes**. There are lots of pros and cons to discuss here as well. They're out of the scope of this report, but there are starting points in the Vespa and Elasticsearch documentation, respectively.

The data layout (e.g., dividing data into multiple indices or document types) is essential, and the ideal configuration depends on the use case. We use one document type, index, and shard per index to eliminate this variable. This way, as far as client applications are concerned, we only use the smallest search unit.

Let's zoom into those searches now: how they're done and on which configuration. Later on, we'll explore the benchmark results.

6. System Configuration

This benchmark is reproducible and runs multiple times daily in the Vespa system test environment. Which means we're using the same hardware as the other Vespa performance tests. As you'll see below, it's over-provisioned for the dataset of 1M documents. That's because the primary constraint is the time to run all the combinations (query types, number of clients).

6.1 Hardware Specifications and Software Versions

To keep the number of variables in check, we used one Elasticsearch index of one shard, while Vespa had one content node.

Everything runs on a single machine. The machine used is AWS c6id.metal with:

- 128 vCPUs. Only 62 are available for the performance test, which both Elasticsearch and Vespa correctly detect. The test uses 62 rather than 64 to accommodate system processes in Kubernetes. Yes, the test only uses half the machine, because the test framework normally runs two tests on the same machine concurrently.
- 256GB of RAM, out of which 150GB is available for the performance test.
- 4x1900GB NVMe SSD.

As with everything, your mileage might vary. Which is why you can always re-run the test on your own hardware (see section 11) and you can even see results from less capable hardware in Appendix A.3.

We use this machine type for Vespa performance tests because it's a "metal" machine, which gives it better stability between consecutive test runs. The instance type is clearly overprovisioned, given that we index only 1M products, but our performance test framework does not support allocating test-specific instance types.

All this is OK for this type of use-case and research: **e-commerce applications often care only about CPU efficiency**. By benchmarking, we can figure out how many CPUs we need to support the required traffic with acceptable search latency. Then we can go hunting for optimized instance types with the right balance between CPU cores and memory.

If the CPU is not your bottleneck, your setup is very likely sub-optimal cost-wise. You need enough RAM to store the HNSW graph plus vector data for ANN search; otherwise, queries will page from disk, slowing them down **a lot**.

As for versions, the test framework uses the latest Vespa version automatically. The Elasticsearch version can easily be changed. The results presented in this report are from **Vespa 8.427.7 and Elasticsearch 8.15.2**.

Below, we'll review the benchmark setup, including queries and schema, thread pools, and heap size. Then, we'll give a high-level overview of the test procedures.

6.2 Schema

Documents in the test dataset have the following fields:

- `id` - document identifiers that are stored.
- `title` - the title of the product used for lexical search.
- `description` - product description, also used for lexical search.
- `category` - string field used for filtering by exact match (no tokenization).
- `price` and `average_rating` - numeric fields (integer and float respectively) that can be used for scoring or faceting. We don't query them in this benchmark. We do change the price during the update test.
- `embedding` - vectors of 384 float dimensions generated from product title and description via Snowflake's Arctic-embed-xs.

The mapping in Elasticsearch is equivalent to the Vespa schema:

- `id`, `category`, `price`, and `average_rating` are stored in row-based and columnar storage. The row-based storage is the _source field in Elasticsearch and document summary in Vespa, while the columnar storage is doc_values in Elasticsearch and attribute in Vespa. You can find more Vespa-to-Elasticsearch equivalents in this blog post. Except for `category`, none of these fields have an inverted index (e.g., index=false in Elasticsearch) because we only filter by category.
- `title` and `description` are tokenized and indexed. When searched, they produce a BM25 score in both search engines. They are also stored in _source/summary.
- `embedding` is a dense_vector/tensor field of 384 floats. We generate an HNSW index using the same parameters (from the original implementation, `M=16`, `ef_construction=200`). The distance metric is an optimized version of cosine similarity: `dot_product` in Elasticsearch and prenormalized-angular in Vespa. We can use this optimization because document and query embeddings are normalized to unit length. We also store the original vector value but don't return it during queries.

There have been changes in Elasticsearch related to defaults for dense_vector fields. For example, 8.14.0 changes the default index type to `int8_hnsw` from `hnsw`. This means the index will automatically use scalar quantization of input float vectors to byte. We started this benchmarking effort with earlier versions and we decided to keep the Elasticsearch `index_options.type=hnsw` to compare apples versus apples using float precision, even if this is no longer the default setting in Elasticsearch.

The choice to maintain float precision rather than use the newly introduced int8 scalar quantization ensures a baseline comparison of raw algorithmic performance using the same precision type in both engines with the same HNSW settings. While int8_hnsw in Elasticsearch offers reduced memory footprint through scalar quantization of float vectors to bytes (4x) and accelerated dot products (2.5-3x speedups), this comes with an accuracy tradeoff. This type of optimization might be well worth exploring for users of both engines, where you would configure Vespa also with int8 (byte) precision to enjoy the same type of benefits.

Both Elasticsearch and Vespa continue to invest in memory footprint reduction and vector similarity acceleration techniques to reduce infrastructure costs, including binary quantization that converts float vectors to bit vectors. However, for meaningful engine comparisons, using configurations with equivalent accuracy levels provides the most reliable performance insights.

6.3 Queries

We run three types:

- **Lexical search** matching against the product title and description.
- **Vector search** (i.e., ANN) matching vectors derived from the keywords.
- **Hybrid search**, running both queries and adding up the relevance scores.

Those queries generally run after indexing is finished with different numbers of benchmark client threads (1 to 64 to simulate query concurrency) in the following setups:

- As they are (after indexing 1M products).
- After force-merging the Elasticsearch index to 1 segment.
- With a category filter.
- After force-merging with a category filter.

We also run the query types after indexing finishes while reindexing all data to simulate a mixed workload. We only run those queries with category filters to limit the benchmark time.

We didn't force-merge during these concurrent read and write workloads because it's an anti-pattern to do that during indexing: you'd rather change the merge policy.

As for the queries themselves, let's describe how they are represented in both engines.

6.3.1 Lexical search - accelerated by Weak AND

Here's an example of the weak AND query in Elasticsearch with the category filter:

```
Unset
{
  "size": 10,
  "_source": false,
  "query": {
    "bool": {
      "must": {
        "multi_match": {
          "query": "Createx 2 Ounce Wicked Pearl Black",
          "fields": [
            "title",
            "description"
          ],
          "type": "best_fields"
        }
      },
      "filter": [
        {
          "term": {
            "category": "Arts_Crafts_and_Sewing"
          }
        }
      ]
    }
  }
}
```

Notice that we target both the title and the description fields, but we take the maximum BM25 score between those fields (which is the default for multi_match).

We don't return the whole document (_source) to reduce the number of variables. In practice, different use cases will fetch different numbers of fields and fields of varying sizes. We eliminate the fetch phase from influencing the benchmark and focus on query-matching alone.

The equivalent weak AND Vespa query is:

```
Unset
{
  "yql": "select * from product where category contains
\"Arts_Crafts_and_Sewing\" and userQuery()",
  "ranking.profile": "bm25",
  "presentation.summary": "minimal",
  "query": "Createx 2 Ounce Wicked Pearl Black"
}
```

The referenced ranking profile (bm25) calculates the maximum score from the title and description fields to match the Elasticsearch behavior:

```
Unset
rank-profile bm25 {
  function best_bm25() {
    expression: max(bm25(title), bm25(description))
  }
  first-phase {
    expression: best_bm25
  }
}
```

6.3.2 Vector search

Here's a filtered ANN query for Elasticsearch:

```
Unset
"knn": {
    "field": "embedding",
    "query_vector": [
 … lots of floats go here …
    ],
    "k": 100,
    "num_candidates": 100,
    "filter": [
      {
        "term": {
          "category": "Arts_Crafts_and_Sewing"
        }
      }
    ]
  }
```

The `k` value is meant to match `targetHits` from the Vespa query. Meanwhile, `num_candidates` is kept at 100 to match Vespa's default hnsw.exploreAdditionalHits=0, which implies that 100 candidates are explored in both engines:

```
Unset
{
  "yql": "select * from product where category contains
\"Arts_Crafts_and_Sewing\" and
({targetHits:100}nearestNeighbor(embedding,q_embedding))",
  "ranking.profile": "closeness",
  "presentation.summary": "minimal",
  "query": "Createx 2 Ounce Wicked Pearl Black",
  "input.query(q_embedding)": [
... lots of floats go here ...
  ]
}
```

Note that the `closeness` rank profile simply computes the distance (defined in the schema) between the query vector and the document vector:

```
Unset
rank-profile closeness {
  inputs {
    query(q_embedding) tensor<float>(x[384])
  }
  first-phase {
    expression: closeness(field, embedding)
  }
}
```

6.3.3 Hybrid search

For Elasticsearch, we used the boolean query, which simply adds up the scores from lexical and vector searches. Note that we're not evaluating relevance in benchmarks here - which hybrid approach works best varies significantly by use case. Something like Reciprocal Rank Fusion might work better, but adding scores is good enough for measuring performance.

Unset

```
{
  "size": 10,
  "_source": false,
  "query": {
    "bool": {
      "must": {
        "multi_match": {
          "query": "Createx 2 Ounce Wicked Pearl Black",
          "fields": [
            "title",
            "description"
          ],
          "type": "best_fields"
        }
      },
      "filter": [
        {
          "term": {
            "category": "Arts_Crafts_and_Sewing"
          }
        }
      ]
    }
  },
  "knn": {
    "field": "embedding",
    "query_vector": [
... lots of floats ...
    ],
    "num_candidates": 100,
    "k": 100,
```

```
    "filter": [
      {
        "term": {
          "category": "Arts_Crafts_and_Sewing"
        }
      }
    ]
  }
}
```

Note that the `knn` query is outside the main query block. This implies that it does a pre-filter on its inner category `filter`, and then fetches the top 100 ANN results. The main query block is effectively another boolean `should` clause, performing the same filtered lexical search that we've seen earlier.

Another approach would have been to put the `knn` query inside the `query→bool` block, with it and the `multi_match` as two should clauses. But then the `filter` clause for the category would apply as a post-filter, which wouldn't be apples-to-apples for the Vespa query below. **That's why we need to duplicate the category filter: to be applied to both the lexical search and the ANN search as a pre-filter**.

As for the equivalent Vespa query:

Unset

```
{
  "yql": "select * from product where category contains
\"Arts_Crafts_and_Sewing\" and
(({targetHits:100}nearestNeighbor(embedding,q_embedding)) or
userQuery())",
  "ranking.profile": "hybrid",
  "presentation.summary": "minimal",
  "query": "Createx 2 Ounce Wicked Pearl Black",
  "input.query(q_embedding)": [
... lots of floats go here ...
  ]
}
```

Here, the category filter works for the weak AND (`userQuery()`) and a pre-filter for the ANN search (`nearestNeighbor`).

We still aren't precisely apples-to-apples regarding how the query runs because of how Elasticsearch/Lucene and Vespa fall back to exact brute-force KNN. See section 5.2.5 for more details.

6.3.4 Result set comparison

Our approach focuses on assessing the *set similarity* of results produced by both engines for identical queries. Specifically, we execute each query against both systems and then analyze the overlap among the top-10 results. With this approach, we can gauge the comparability of the engines' outputs, ensuring that performance differences observed are not due to fundamental disparities in the results returned but rather due to efficiency differences.

Query Type	**Average overlap@10 (Vespa versus Elasticsearch)**
Lexical (WAND)	0.79
Lexical (WAND) + filter	0.81
Vector (ANN)	0.94
Vector (ANN) + filter	0.97
Hybrid	0.79
Hybrid + filter	0.81

The divergence in lexical search results is mainly down to linguistics. Even though we configured Elasticsearch's analysis to be similar to Vespa's linguistics, there are still differences:

- **Tokenization**
  Vespa's SimpleTokenizer is similar, but not the same to Elasticsearch's Standard tokenizer. For example, Vespa always tokenizes on dots, while Elasticsearch sometimes doesn't (e.g. "vespa.ai" remains a single token).
- **Stemming algorithms**
  We checked a few input texts and most produced the same root words, but not in every case. For example, "dreams" gets indexed as "dream" in both search engines, but "hardwired" becomes "hardwire" in Vespa and "hardwir" in Elasticsearch. We tried all the built-in Elasticsearch stemming implementations and the porter2 stemmer seems the closest to Vespa.

Vector searches demonstrate a higher degree of overlap, suggesting consistency in vector search implementations between the two engines.

The hybrid query type, which combines both lexical and vector elements, shows a similar result overlap as the lexical. This alignment is due to the hybrid scoring, which favors the lexical component. The bias occurs because lexical scores (BM25) typically produce larger absolute values compared to the normalized dot product (cosine similarity) used in vector search, thus exerting a stronger influence on the final hybrid ranking.

6.4 Elasticsearch Refresh Interval

Another challenge in making a fair comparison is that Vespa is real-time: when you add, update, or delete a document, changes will be visible after the client gets the acknowledgment. Elasticsearch, however, is near-real-time: changes are visible after the next refresh.

We kept the default 1s refresh interval as a middle ground between a fair comparison and realistic Elasticsearch workloads:

- A shorter refresh interval makes the comparison more "fair" because it results in more similar functionality.
- A longer refresh interval is closer to what most Elasticsearch deployments use in production.

Elasticsearch also has a clever optimization: if you don't search an index for 30 seconds, refresh is off until you search it again.

Most benchmarks here write the data, do a manual refresh, then start queries. This way Elasticsearch can write without spending resources on making results visible. Unless we're talking about updates. More details on how updates work in both engines are in section 5.1.

There's also a benchmark of searching while refeeding, where the refresh interval becomes essential: make it shorter, and refresh takes more resources and invalidates more caches. Make it longer, and we'll be serving more stale data. Meanwhile, Vespa is real-time.

6.5 JVM and Thread Pool Settings

JVM settings are also complex to compare because Vespa has more processes than the stateless container JVM. Still, we increased it to 16GB maximum heap size and kept everything else default. Elasticsearch is effectively one JVM process, where we kept the default of 31GB maximum heap size. During benchmarks, we weren't constrained by heap size on either of the engines, and GC time was insignificant.

It's a similar story with the number of threads: while some thread pools are similar between Elasticsearch and Vespa, they execute these tasks differently. For example, both have write threads, but Vespa's write path uses more CPU, even though the default thread pool size is ½ the number of processors compared to the number of processors in Elasticsearch. Part of this could be because Vespa's writes are individual and real-time. Another part could be because Vespa has another thread pool (shared) that is used to search the HNSW index in parallel to locate the neighborhood in which to insert a new vector.

These differences mean we can't simply set the same thresholds for similar thread pools to make results comparable. Instead, we looked at whether thread pool settings are a good enough fit for the use case. And it seems like the defaults (for both Vespa and Elasticsearch) are decent. So we kept the defaults, which, at the time of writing this, on a machine with 62 cores available for the search engine, are:

- For Vespa
  - 64 match threads for searches
  - 31 for field writer and 31 shared threads used during indexing
- For Elasticsearch
  - 31 `search_coordination` and 94 `search_worker` threads for queries
  - 62 write threads for indexing

We can argue that the total number of threads for Elasticsearch is too large. Especially during a mixed workload, we'd have more than 3x more active threads than CPU cores. But this isn't enough to make us steer from the defaults for several reasons:

1) **Side-effects**. If we reduce thread pools, we might negatively affect performance for query-only or write-only workloads.
2) **Wait time**. Search threads sometimes have to wait. It's the reason why the default is 1.5 times the number of cores (plus one).
3) **Fairness**. Vespa's total number of threads is also larger than the number of CPUs.
4) **Stay realistic**. In our experience, few Elasticsearch users reduce thread pool sizes. Most hosted Elasticsearch providers don't even allow you to change that.
5) **Reproducibility**. We want to stick as close to the defaults as possible so that new runs of the benchmark reflect changes in the defaults. Defaults are generally sensible and often adjusted as the engine's internals change.

6.6 Test Procedure

The test procedure is best described through the Elasticsearch test procedure here. Each invoked function is described in the base file. Vespa test files are in the same directory but are more challenging to read due to deeper dependencies within the test framework. In essence, both tests perform the same steps:

1. **Preparation steps**. Start Elasticsearch/Vespa and create the index or deploy the application, respectively.
2. **Write the data**. This implies preparing the dataset in the Bulk API and Document V1 API formats. For actual writes, for Elasticsearch, we invoke cURL over 16 async threads with batches of 12000 lines (6K documents). We found that this concurrency and batch size gives good enough throughput to manage write efficiency (more on metrics below) while cURL used insignificant CPU compared to Elasticsearch. Similarly, we used Vespa's FeedClient over 4 connections. Like with cURL, FeedClient wasn't compressing, but unlike cURL, it used TLS. Still, the CPU usage of FeedClient was insignificant, so we could go ahead and measure efficiency.
3. In both engines, we **flush the memory index to disk**. This way, none of them "cheat" by using the indexing buffer for queries.
4. **Run queries**. There are multiple combinations here: with and without filters, with various client threads for each. Elasticsearch and Vespa queries are run via vespa-fbench, which can benchmark almost any HTTP request.
5. **Rewrite the data and perform in-place updates**. After the bulk of queries, two more write tests are done: re-writing all data again (i.e., a reindex test) and partial updates (i.e., updating the price field). Both are done similarly to the initial write: preparing the data and using cURL and Vespa FeedClient.
6. **Mixed workload**. Finally, another rewrite workload is performed (precisely the same as the previous rewrite), but this time, we re-run some of the queries. Specifically, filtered queries on 1, 16, and 64 client threads for only 9 seconds each. This ensures that queries are done by the time re-writing is complete. By contrast, queries at step 4 run for 20 seconds each because we don't have this constraint.

To explore the impact of the number of segments on Elasticsearch query performance, we have a separate test procedure where we write the data and then force-merge to one segment before running queries.

If you look at the code, you'll notice that we record metrics at each step above (except for preparation). We'll describe those metrics next.

7. Description of Sampled Metrics

To understand the detailed results in the next section, we must first examine the metrics sampled during the benchmarks and their definitions.

7.1 Writes

When benchmarking write workloads, we concentrate on throughput (as seen from the client used for writing) and the CPU usage on the machine running Vespa or Elasticsearch.

- **Throughput** is the number of operations the system handles per second. In the benchmark, this can be document puts per second or document updates per second. However, looking at this metric in isolation is often misleading, as it doesn't say anything about the resources used by the system when handling the workload.
- **CPU cores** - The average number of CPU cores used when handling the workload. This is a direct measure of the amount of resources used. We sometimes divide by the number of cores (62) and multiply by 100 to get the CPU usage (percentage).
- **Throughput per CPU core** - Dividing the absolute throughput by CPU core usage says something about the system's effectiveness in handling the write **workload.** This metric should be used for the most accurate and fair comparison between the two systems.

7.2 Queries

When benchmarking queries, we concentrate on latency, throughput, and CPU usage on the machine running Vespa or Elasticsearch.

- **Average latency** - The average latency in milliseconds among all query requests, measured by vespa-fbench.
- **99 percentile latency** - The maximum latency in milliseconds experienced by 99% of all query requests, with only 1% of requests taking longer.
- **QPS** - The average number of queries per second handled, measured among all query requests.
- **CPU cores** - The average number of CPU cores *used* when handling the query workload.
- **QPS per CPU core** - The result of dividing **QPS** by **CPU cores** and says something about the system's effectiveness while handling the query workload. This is similar to the **Throughput per CPU core** measured during the writing benchmarks.
- **CPU usage** - The CPU cores in use divided by the total number of CPU cores on the machine. This is used when we test scalability with increasing concurrency.

7.3 Ratios

As seen above, two broad categories of metrics are collected in the benchmarks:

- Throughput (count/sec): higher numbers are better.
- Latency (ms): lower numbers are better.

To compare the results between Vespa and Elasticsearch, we calculate the ratio between two samples by dividing one sample by the other. **A ratio larger than 1.0 is in favor of Vespa, and a ratio less than 1.0 is in favor of Elasticsearch**. The ratios are calculated as follows:

- **Throughput ratio** = Vespa sample / Elasticsearch sample
- **Latency ratio** = Elasticsearch sample / Vespa sample

When comparing the results from different workloads, the ratios show which of the two systems performs best. The ratio numbers for **Throughput per CPU core** (writing) and **QPS per CPU core** (queries) are the most relevant, including the resources used to achieve the performance.

8. Results

8.1 Writes

The following charts present a comparative analysis of write performance between Vespa and Elasticsearch across four distinct writing-related workloads:

- Write/feed, bootstrapping the index from 0 to 1M documents
- Rewrite/refeed
- Rewrite/refeed with concurrent queries
- Update

This analysis focuses on two metrics:

- **Throughput (operations/sec)**: Measuring the overall system capacity to process operations.
    - Elasticsearch shows a strong advantage in the initial write (feed) bootstrap workload.
    - Vespa's update throughput is more than 9 times higher than Elasticsearch's.
    - Vespa achieves higher absolute throughput than Elasticsearch in 3 out of 4 workloads (refeed, refeed_with_queries, and update). That said, none of the engines used 100% CPU in any of the scenarios, suggesting there are bottlenecks that would benefit from further investigation. They could be down to threading (in the search engine or in the client) or contention points in the search engines.
- **Throughput per CPU core**: Indicating the efficiency of resource utilization.
    - Elasticsearch demonstrates higher efficiency in the feed and refeed workloads. In other words, it gets more done with less CPU utilization.
    - Vespa shows better performance in the refeed_with_queries workload. This might be due to more efficient querying (see below) as the measured CPU usage includes queries.
    - Vespa outperforms Elasticsearch in the update workload, with about 4 times higher throughput per CPU core.

It's important to note that this analysis does not consider the latency of individual write operations, as the primary focus is on overall system throughput and efficiency.

In conclusion, while Elasticsearch shows advantages in bootstrap writing, Vespa gets close when it comes to rewriting, especially when we consider the query load, see below. Vespa is a clear winner when it comes to updates.

8.2 Queries

In the following illustrations, we present the results of our performance tests using a configuration of 16 concurrent client threads, with filtered queries. Later in this section, we perform a scalability analysis, examining how these engines perform across a spectrum of user concurrency levels.

8.2.1 Fixed number of clients

This setup represents a balanced approach within our broader experiment, which spans from low concurrency (1 client) to high concurrency (64 clients, slightly exceeding the

available CPU cores). We chose the **16-client configuration with query filters** as our primary focus for three reasons:

- It represents a middle-to-lower level of user concurrency, aligning well with real-world scenarios such as e-commerce search solutions where multiple users interact simultaneously.
- It provides a balanced view of system performance, avoiding the extremes.
- Filtered search is a common e-commerce use case.

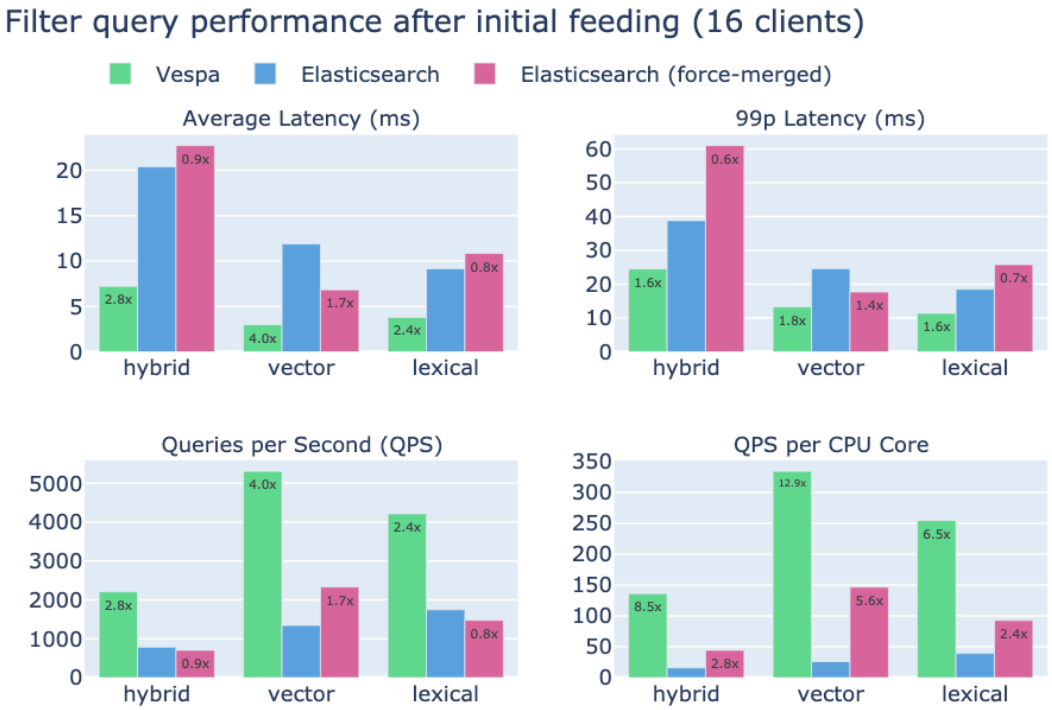
This configuration allows us to assess how the engines perform under conditions that closely mirror practical deployment scenarios. For snapshots of other configurations (e.g. with a single client or unfiltered) see Appendix A1.

Due to significant performance disparities between Elasticsearch configurations with a single merged segment and multiple segments, we report these as separate entries. The force-merged configuration represents a read-only index with a single segment, while the default setup reflects more typical real-world usage with multiple segments (here, around 50).

Note that these benchmarks were performed with Elasticsearch 8.15.2, where concurrent search of segments is enabled by default.

Queries with fixed concurrency and no feeding

In the following illustrations, the ratio number in the bars is calculated by comparison with Elasticsearch default (multi-segment). For example, for the hybrid query case, Vespa is 2.8x faster in average latency than Elasticsearch default, while Elasticsearch (force-merged) is 0.9x faster than Elasticsearch default (meaning it's slower).

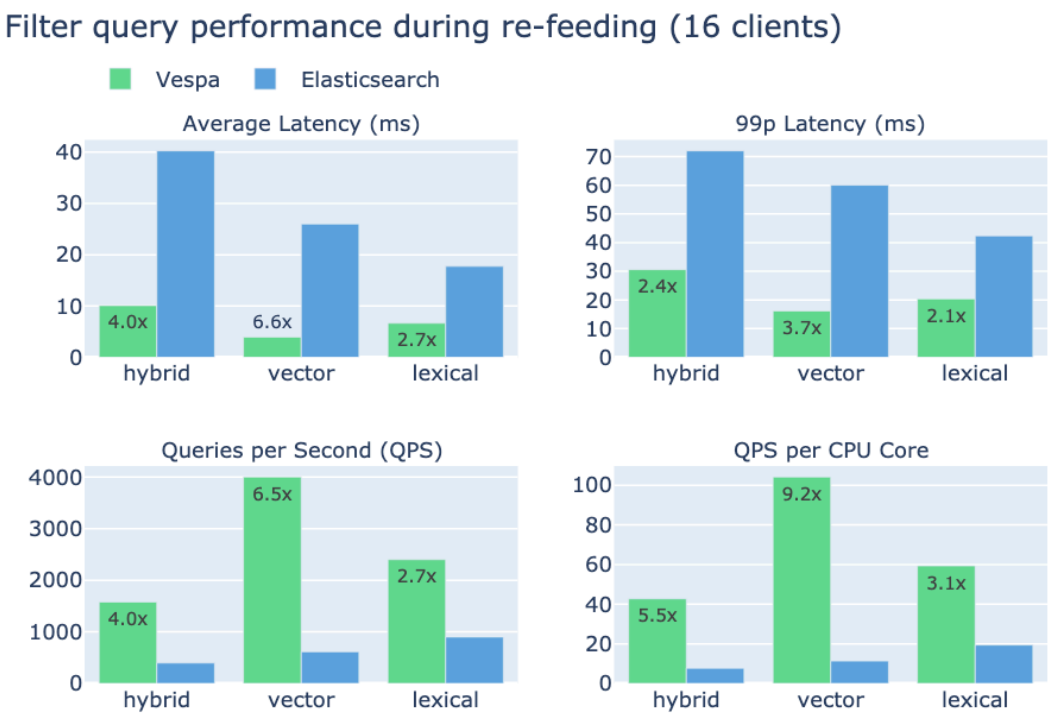


The latency and throughput metrics reveal Vespa's superiority across all query types. **Vespa outperforms both Elasticsearch variants in average latency**, with the gap most pronounced for hybrid queries. Notably, all systems exhibit higher latencies for hybrid queries than vector and lexical queries. This is natural as they combine the two other types.

The 99th percentile (99p) latency data paints a similar picture, but notice here how the multi-segment parallelization in Elasticsearch allows for a lower 99p latency than the force-merged configuration for hybrid and lexical search, but not for vector.

Throughput metrics demonstrate Vespa's performance advantage. **Vespa processes more queries per second per CPU core than both Elasticsearch configurations across all query types**, with the gap widening for vector queries (12.9x). Force-merged Elasticsearch shows improved QPS over default, especially for vector queries, but still lags behind Vespa. Note the low QPS per CPU core for standard multi-segment Elasticsearch compared to the force-merged Elasticsearch (and Vespa).

Queries with fixed concurrency during refeeding



In the concurrent query and feeding workload, there is no single-segment Elasticsearch variant, as that would be incompatible with near real-time indexing. **Vespa outperforms Elasticsearch on all metrics in this test**. Notably, Elasticsearch's 99th percentile latency nearly doubles for all query types compared to the previous workload without concurrent indexing. In the hybrid case, the 99th percentile latency increases from 40ms to 70ms for Elasticsearch, while Vespa's increases only from 25ms to 30ms.

It's important to note that in this scenario, we cannot differentiate between CPU usage related to queries versus writes. Consequently, the QPS per CPU core metric is **heavily biased against Vespa**: it also accounts for CPU usage from write operations, where **Vespa achieved 2.5x more throughput** (see section 8.1).

8.2.2 Scaling with increased load & concurrency

In this section, we paint a full picture of each system's performance characteristics, from low-load situations to high-stress scenarios. For example, 16 concurrent clients push Elasticsearch's concurrent segment search to nearly 80% CPU usage. This high usage results in higher latency for the multi-segment Elasticsearch variant compared

to scenarios with fewer clients. We've examined performance across various concurrency levels to show each system's capabilities and limitations.

The concurrency graphs below, often called **"saturation curves" or "load vs. performance curves"**, are used to assess the performance characteristics of systems under varying loads. They help evaluate how the system behaves as we increase load by increasing the client concurrency. We evaluate all query types using the filtered variant: the query type is combined with filters as this is the most complex query type and also a query type that reflects real-world e-commerce search use cases well.

The graphs plot three key metrics against Queries Per Second (QPS):

- **Average Latency**: The mean time to process a query.
- **99p Latency**: The 99th percentile of query processing times, indicating worst-case performance.
- **CPU Usage**: The amount of computational resources used. 100% here means that all 62 available CPU cores would be in use, while 10% would be about 6.

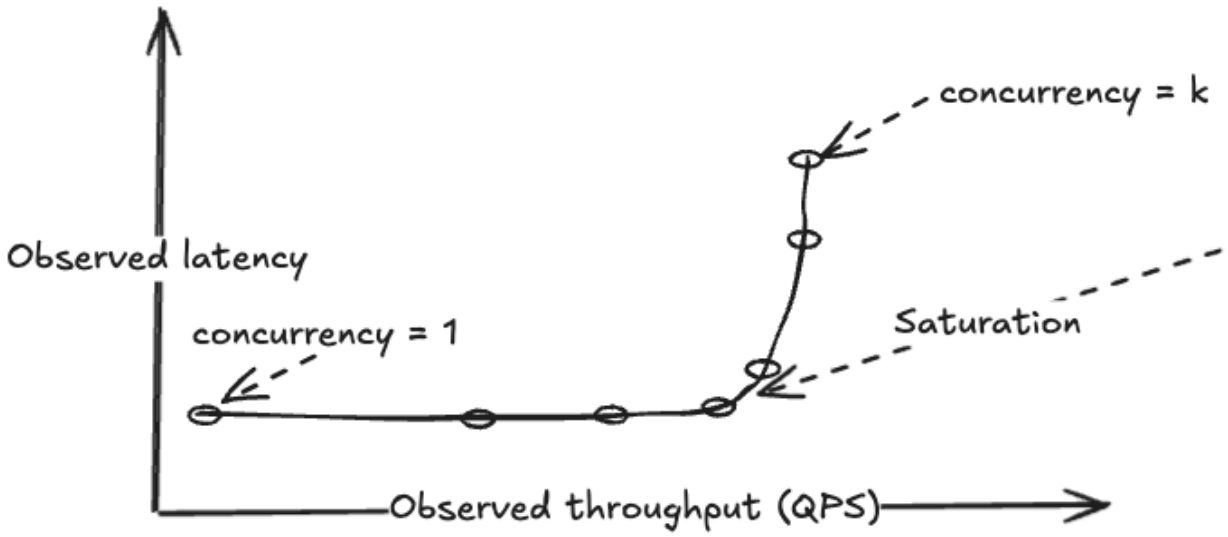
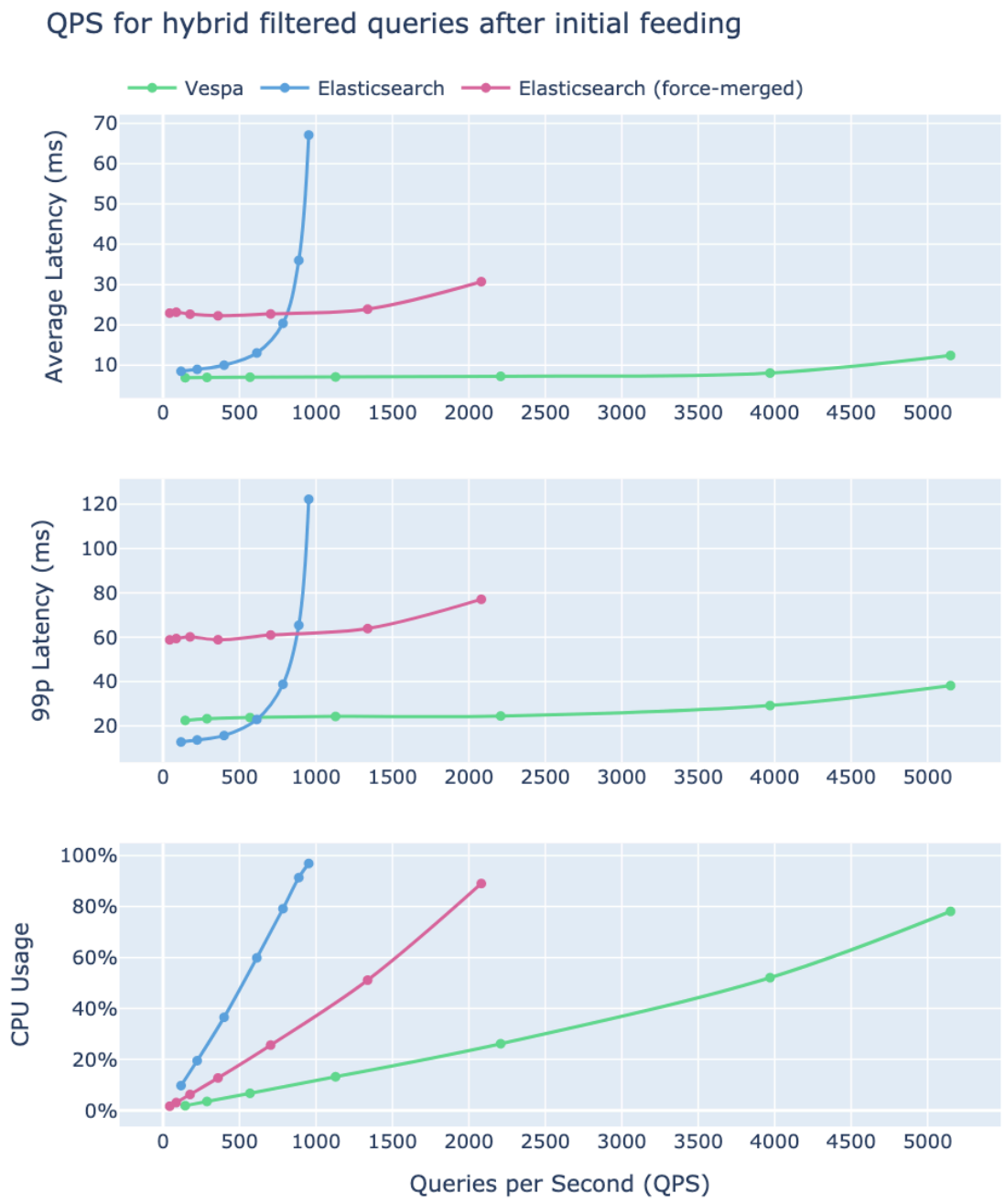


Illustration of expected system behavior. We start with one client (concurrency=1) and observe the latency and obtained throughput (QPS). Then we increase the number of clients (concurrency) up until k and continue to observe the throughput and latency. At a certain user concurrency, saturation occurs (or a software bottleneck emerges), causing latency to climb rapidly as queuing times become the dominant factor.

The ideal system maintains low and stable latency while efficiently utilizing resources before showing signs of saturation, where latency starts to climb. **Once saturation is reached, the systems will start queuing requests**, and as we push more concurrent clients **after that point, the time spent queueing will dominate the overall latency** (queue time + service time). Each point in these graphs is generated by a certain number of client threads.

While we could subject these engines to thousands of client threads to determine their throughput capacity, the resulting latency numbers would be arbitrarily large and would not generalize to lower user concurrency scenarios. This is a common mistake practitioners make when benchmarking software systems, **throwing lots of concurrency at once, and then measuring latency**. A more prudent approach is to start with a single client thread (no concurrency), carefully monitoring resource usage and latency, before progressing to higher loads. These types of saturation graphs are invaluable for capacity planning and understanding system bottlenecks. For more on this, see this practical engineering one-pager Little's Law: How Long is the Wait.



The graphs compare the performance of Vespa, Elasticsearch, and Elasticsearch (force-merged) for hybrid queries with increasing client concurrency (load). The queries are executed against "idle" systems, without any concurrent write operations. Each point per line represents a concurrency setting in (1, 2, 4, 8, 16, 32, 64).

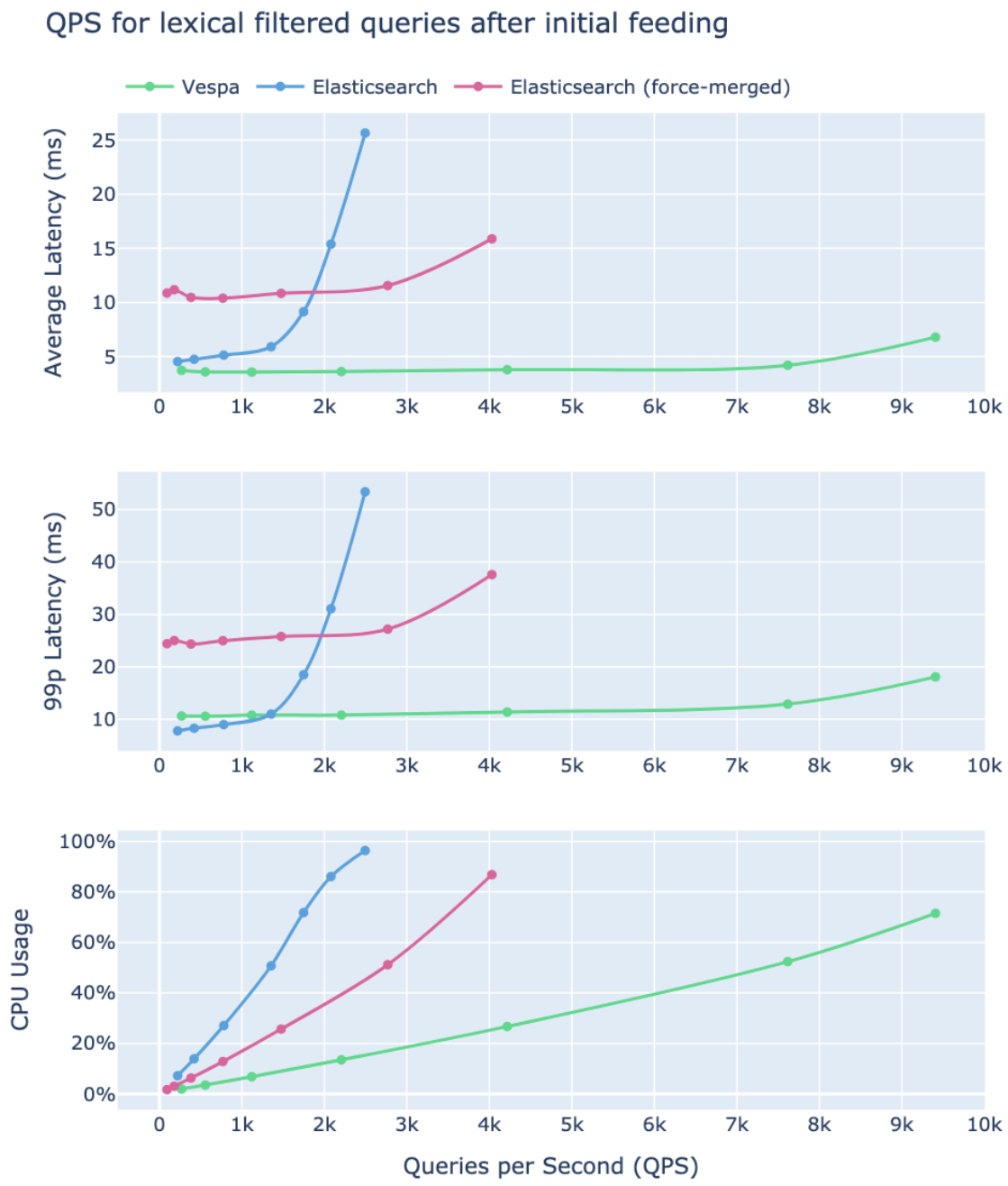
All three configurations scale CPU usage linearly with query throughput, but the slope of the CPU usage line is much steeper for the Elasticsearch variants. **A steeper slope indicates lower efficiency, resulting in fewer queries per second (QPS) per CPU core**.

Vespa maintains low latency up to about 5000 QPS with 64 concurrent clients where there are signs of latency increase. Vespa's CPU usage is lower than the Elasticsearch variants at peak reported QPS. This indicates that **it would be possible to reach higher throughput by increasing the number of concurrent clients** (e.g., pushing to 128 instead of stopping at 64).

The multi-segmented **Elasticsearch variant starts to climb the latency hockey stick at 600 QPS**. The latency skyrockets before CPU (100%) saturation. Observe that the **99p latency starts climbing early at relatively low CPU utilization** and concurrent clients. This latency increase indicates that the concurrent multi-segment search introduces contention, possibly related to synchronization between the concurrent threads.

The **multi-segment Elasticsearch variant achieves the lowest 99p latency at low load** (e.g. 1 client) but **uses significantly more CPU resources than Vespa and the force-merged Elasticsearch** variant to achieve lower latency. However, this low 99p latency doesn't translate to **average latency, where Vespa is better across the board**.

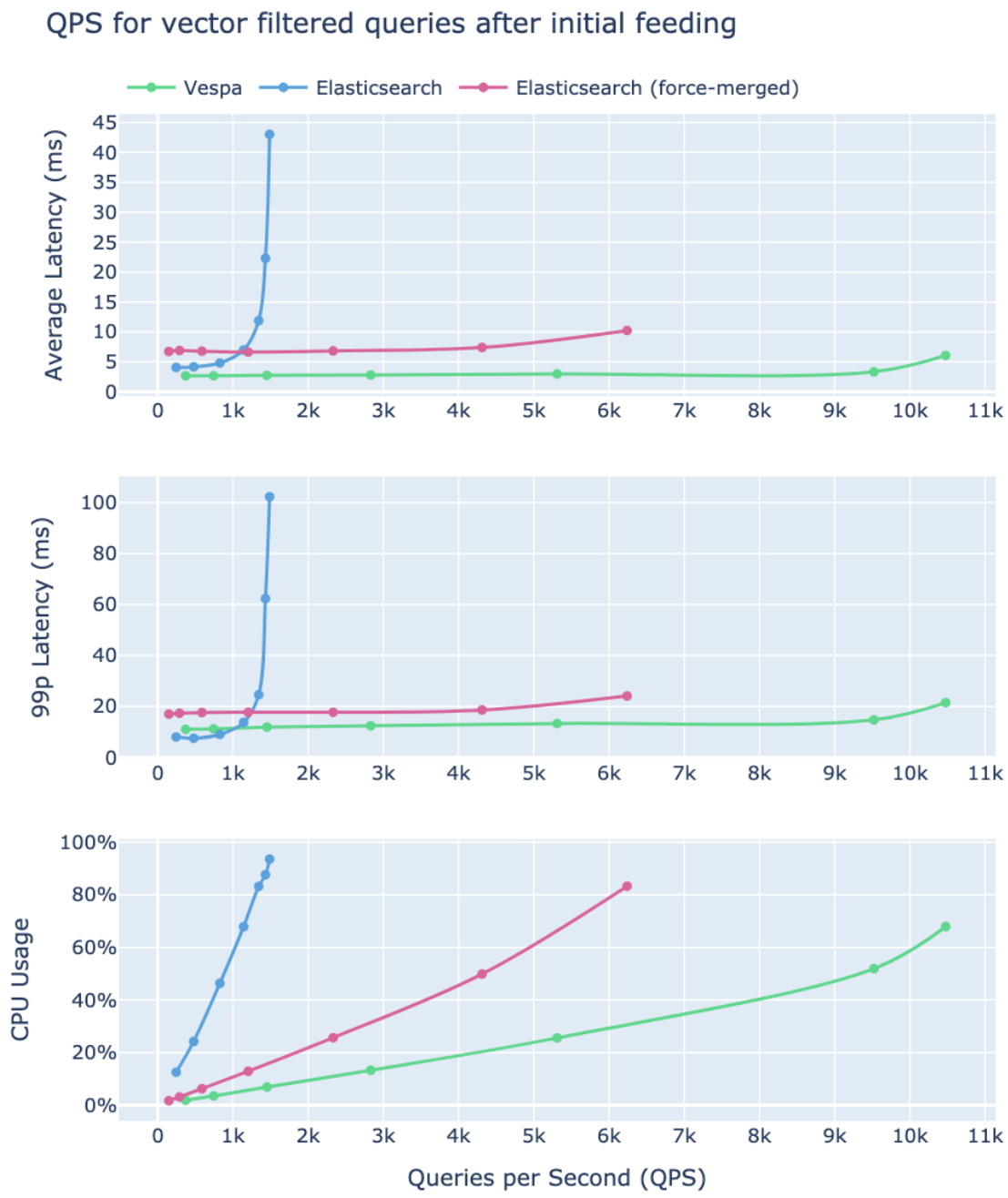
The average latency and the 99p latency of the multi-segment Elasticsearch variant are lower than the force-merged Elasticsearch variant, up to 16 clients. This exemplifies that reporting latency numbers at low concurrency without resource utilization metrics doesn't paint the full picture.



The graphs compare Vespa, Elasticsearch, and Elasticsearch (force-merged) performance for lexical queries (accelerated by WAND) with increasing client concurrency (load). The queries are executed against "idle" systems, without any concurrent write or indexing operations. Each point per line represents a concurrency setting in (1, 2, 4, 8, 16, 32, 64).

Vespa maintains low and stable latency up to 8000 QPS, with only a slight increase at the highest concurrency at 9500 QPS. Elasticsearch's latency rises sharply after about 1500 QPS, saturating the CPU due to concurrent segment search. The force-merged Elasticsearch variant shows better scaling with increased concurrency, maintaining low latency up to about 3500 QPS.

Once again, Vespa's average latency is the lowest across the board, while the multi-segment Elasticsearch variant has the lowest 99p latency at low concurrency, at the expense of higher CPU usage. The CPU usage lines show similar differences in slope as for the hybrid case.



The graphs compare Vespa, Elasticsearch, and Elasticsearch (force-merged) performance for filtered vector queries with increasing client concurrency (load). The queries are executed against "idle" systems, without any concurrent write or indexing operations. Each point per line represents a concurrency setting in (1, 2, 4, 8, 16, 32, 64).

Vespa demonstrates exceptional scaling, maintaining low latency (below 5ms) up to 10,000 QPS, with only a slight increase at the highest loads. Elasticsearch with concurrent segment search reaches saturation at around 1,300 QPS with 8 concurrent clients where latency climbs the hockey stick. The force-merged Elasticsearch exhibits better efficiency, maintaining stable and low latency up to approximately 6,000 queries per second (QPS). The CPU usage line slopes are similar to the other query types.

8.3 Results Summary

The results show Vespa's superior query performance and efficiency across all query load types. **Vespa consistently delivers low latency and efficient resource usage over a significantly broader range of queries per second (QPS) compared to both Elasticsearch configurations**. Notably, when examining CPU usage in relation to throughput, Vespa exhibits a much gentler incline than both Elasticsearch variants. This flatter slope is evident across all three query types, indicating that Vespa's CPU usage increases more gradually as the query load intensifies. You'll see the same patterns for un-filtered queries in Appendix A.2.

The low QPS per CPU Core metric reveals an intriguing aspect of Elasticsearch's concurrent segment search feature, which can be attributed to the non-linear scaling of parallel segment query processing. While increasing CPU resources aims to reduce latency, it results in reduced efficiency. The force-merged Elasticsearch configuration achieves significantly higher throughput and throughput *per CPU core* than the multi-segment variant.

This phenomenon occurs because the parallel processing of queries across multiple segments doesn't yield proportional performance gains. For instance, utilizing eight threads to search eight segments concurrently doesn't result in a latency reduction to ⅛ of what a single thread searching sequentially would achieve.

This relationship between CPU resource usage and lack of performance improvement stems from increased coordination overhead, potential resource contention, and the inherent limits of parallelizing ***top-k search operators with sub-linear complexity***. Consequently, as more CPU cores are employed to enhance latency, the efficiency of queries processed per core declines.

Top-k search operators, such as approximate nearest neighbor search or weak `AND` variants, aim to reduce computational complexity by efficiently finding the best

matches *without exhaustively scoring every document (linear complexity)*. **These algorithms offer sub-linear complexity relative to the total number of indexed documents, meaning they scale well as the corpus grows.**

Let's consider a theoretical example with a corpus of 1 million (1M) documents and compare a single segment (force-merged) versus 50 equally sized segments:

Single Segment Scenario:
If all 1M documents were in a single segment, an HNSW graph search approaches logarithmic complexity. The search might take approximately log(1,000,000) = 6 time units.

Multi-Segment Scenario:
With 50 segments, each segment contains roughly 20,000 documents (1,000,000 / 50). For each segment, the search takes about log(20,000) ≈ 4 time units.

However, we need to search all 50 segments, so the total complexity becomes 50 * log(20,000) ≈ 200 time units. The multi-segment approach results in much higher overall complexity (200 >> 6). Although these calculations are theoretical, they paint a fairly accurate picture.

This is an interesting property of algorithms with logarithmic scaling with regard to the number of documents. Let's compare the above with a hypothetical linear search algorithm where the time complexity scales directly with the number of documents (N). When splitting such a dataset into M partitions, each partition would contain N/M documents. Since these partitions can be searched independently and concurrently using M threads, we would achieve nearly linear speedup.

9. Summary

With this benchmark, we aim to paint a comprehensive picture of the trade-offs made by the two search engines, as well as their performance implications. While Elasticsearch excels in initial writes, Vespa demonstrates superior query performance and scalability across various load types. Vespa also has faster writes during the mixed workload and during the update test.

Use cases with frequent updates and complex, high-volume query requirements might find Vespa more suitable, whereas those prioritizing fast initial bootstrap could lean towards Elasticsearch. The benchmark underscores the importance of evaluating specific workload characteristics, including the balance between write and read operations, query complexity, and scalability needs when choosing a search engine solution. Ultimately, the ”best” choice will depend on the particular use case, performance priorities, and operational constraints.

9.1 Key Performance Differences

If the bottleneck for your use-case is CPU while **writing new documents, Elasticsearch is the clear winner**. When it comes to **queries, Vespa is more efficient in almost all cases**, the question becomes “by how much?”. The biggest speedup was with **vector search: 10x-20x**. There is a **2x-8x speedup in lexical search**, while hybrid searches are somewhere in the middle.

The intuition: **most of the performance differences are related to the architectural differences** explained in section 5. Vespa has fewer data structures to process at query time but pays for them with some write performance.

9.2 Implications of Performance Differences

How does this translate to our example e-commerce use-case? Let's say we have this 62-core machine and want to install Elasticsearch or Vespa to serve our product search. Here are the differences that you can expect:

- Initial write throughput: about **3x higher per CPU core with Elasticsearch** compared to Vespa.
- Updating prices and stocks: **4x more throughput per CPU core with Vespa** compared to Elasticsearch. Those updates will be instantly visible to search, as will all the write operations.
- If you're doing a **nightly rewrite** to the new index and that index remains static during the day:
    - Elasticsearch's indexing speedup will be offset by the force-merge. That's because we measured force-merge time and divided it by the number of documents, resulting in 900 docs/s, more than 10x less than Vespa's measured write throughput. Still, force-merge is usually worth it because of the query performance gains (see section 8).
    - You can expect Vespa to support 2.3-3.2x the query throughput of Elasticsearch across all query types tested here. Similar numbers apply to average latency and p99 latency.
- If you **write and query concurrently**, Elasticsearch will have more segments, so differences depend more on the query type:
    - Vespa will support 4x more query throughput for hybrid searches than Elasticsearch.
    - For vector searches, that number grows past 6x.
    - For lexical searches, the number shrinks to 2.7x.
    - All this while Vespa returns 2x-6x lower average latency.

The numbers above are conservative in Elasticsearch's favor. For example, during the mixed workload query tests, Vespa indexed at 2.5x the throughput compared to Elasticsearch, leaving less CPU available for queries.

But what do the numbers above translate to in terms of cost? Let's do a simulation with a cloud provider to find out.

10. Cost Example

Say that you have this e-commerce data, and the SLAs are to run hybrid searches on it with:

- Under **50ms query times**. Quite common for e-commerce applications, especially as some will have multiple underlying queries per user action.
- At least **700QPS**. For the same reason, a site with enough traffic will quickly generate these numbers.

And support at least **300 docs/second write** speed. This is more than most 1M docs datasets need daily, but it's enough to reindex the whole dataset in 1 hour.

To support this use-case, **Vespa needs less than ⅕ the hardware Elasticsearch needs**. Here's how we got to this conclusion.

In this benchmark, filtered hybrid queries were running on Elasticsearch with 8 clients at 613 QPS and with 16 clients at 784 QPS. The average latency ranged from 13ms to 20ms, both within the SLA. In both situations, Elasticsearch achieved around 17QPS per core, so to meet our SLA we need **41 CPU cores**.

Looking at the same type of queries for Vespa, we had 569 QPS with 4 clients and 1128 QPS with 8 clients, using about 4 and 8 cores respectively. To meet our SLA, we need **5.1 CPU cores**. All this while providing a lower average latency of 7ms.

When looking at the throughput-per-CPU values for writes, about **one CPU core** should be enough whether we're talking about new documents, rewrites, or updates.

In AWS EC2, Elasticsearch could use a **48-core c8g.12xl** machine to do the job. At the time of writing, the on-demand price for this instance is **$1.9/hour**. Meanwhile, Vespa could meet the same SLA with an 8-core c8g.2xl machine. But to be absolutely sure that we have enough RAM (maybe for future growth), we can use an **m8g.2xl at $0.36/hour**. Still **more than 5x cheaper**.

Caveats of the math above:

- We took the query and write numbers separately, instead of the “query during refeed” numbers. That’s in order to make the comparison fair: the read+write test does a lot more indexing throughput (and therefore uses a lot more CPU) for Vespa than for Elasticsearch. This skews numbers in the raw results, especially if we look at fewer clients: most of the CPU is used by indexing, making the throughput-per-CPU numbers there meaningless.
- Other use-cases will behave differently. Even with this e-commerce dataset, pure lexical and vector searches will have different requirements. For filtered lexical search on 16 clients, Vespa’s throughput-per-CPU values are still more than 6x better than Elasticsearch’s, but for filtered vector search it’s more than 12x better on Vespa.

11. How to Reproduce

This guide shows how to reproduce the benchmark results in this report using a single machine. Vespa 8.427.7 and Elasticsearch 8.15.2 are used.

11.1 Prerequisites

You need a machine running Linux or macOS (x86_64 or arm64) with the following installed:

- Podman
- git
- python3

To reproduce the benchmarks results the Podman machine should have 64 vCPUs as we run query benchmarks with up to 64 clients.

If you haven't used Podman before, you need to run the following (adjust accordingly to fit your environment). The following sets 8 CPU cores and 28 GiB of memory available. This fits well on a machine with 32 GiB of RAM.

```
Unset
podman machine init --memory=28672 --cpus 8
podman machine start
```

NOTE: The test configuration specifies 16 GiB for the Vespa stateless container so going below this will cause the container to fail to start.

11.2 Run Performance Tests

Checkout the system-test repo that contains the performance tests:

```
Unset
cd $HOME/git
git clone https://github.com/vespa-engine/system-test.git
```

Enter the directory with the performance tests:

```
Unset
cd system-test/tests/performance/ecommerce_hybrid_search/
```

Run the performance test for Vespa. The results are placed in `perf_results/8.427.7/vespa.json`.

```
Unset
./run-perf-test.sh vespa
```

Run the performance test for Elasticsearch. The results are placed in `perf_results/8.427.7/elasticsearch.json`.

```
Unset
./run-perf-test.sh elasticsearch
```

Run the performance test for Elasticsearch that force-merges to one index segment. The results are placed in `perf_results/8.427.7/elasticsearch-force-merged.json`.

```
Unset
./run-perf-test.sh elasticsearch-force-merged
```

11.3 Create Report

Ensure you are still in the directory with the performance tests:

```
Unset
cd system-test/tests/performance/ecommerce_hybrid_search/
```

Create a virtual Python environment and install dependencies:

```
Unset
python3 -m venv myenv
source myenv/bin/activate
pip install -r requirements.txt
```

Create all figures and save them in `report_output/`.
NOTE: Remember to change the values for `--machine_cpus` and `--test_cpus` to match the number of vCPUs on the machine used.

```
Unset
python3 create_report.py --machine_cpus 128 --test_cpus 62 --output
report_output perf_results/8.427.7/vespa.json
perf_results/8.427.7/elasticsearch.json
perf_results/8.427.7/elasticsearch-force-merged.json figure
```

NOTE: If the performance tests are executed in a virtual machine, CPU usage sampling might not be available. If so, all "*per CPU core*" metrics are calculated as if one CPU core was used.

See Appendix A.3 for sample reports generated by running the benchmark on a laptop with fewer resources.

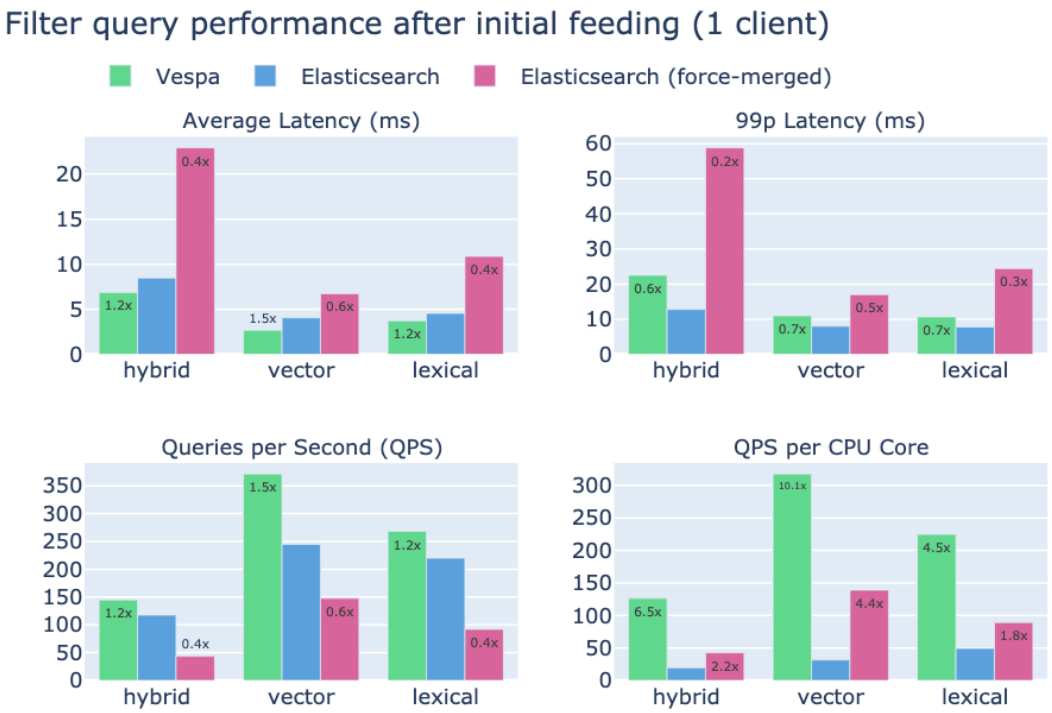
Appendix A: Other Test Results

These graphs complement those in section 8.2 by showing how results vary with:

- Different number of clients.
- Removing filters.
- Less capable hardware.

A.1 Fixed number of clients

A.1.1 Queries with fixed concurrency and no feeding

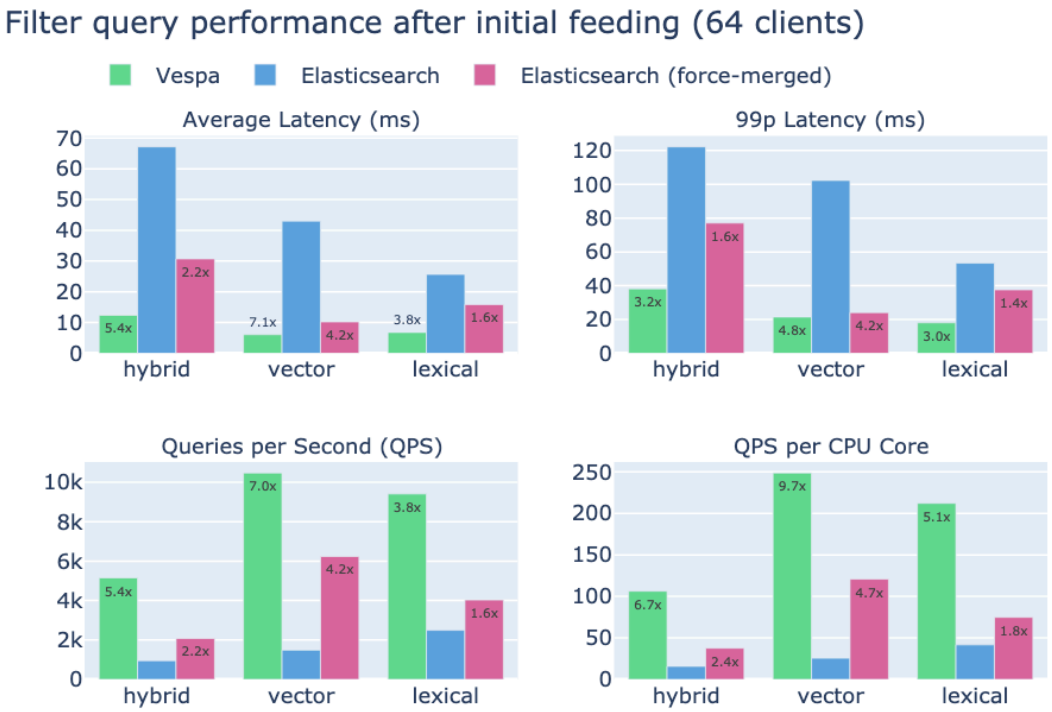


The above illustrates the observed performance with one benchmark client, meaning there is no user concurrency. A single user has the entire system available.

What stands out here is the latency difference between Elasticsearch and Elasticsearch (force-merged) for all query types. By searching the segments concurrently using

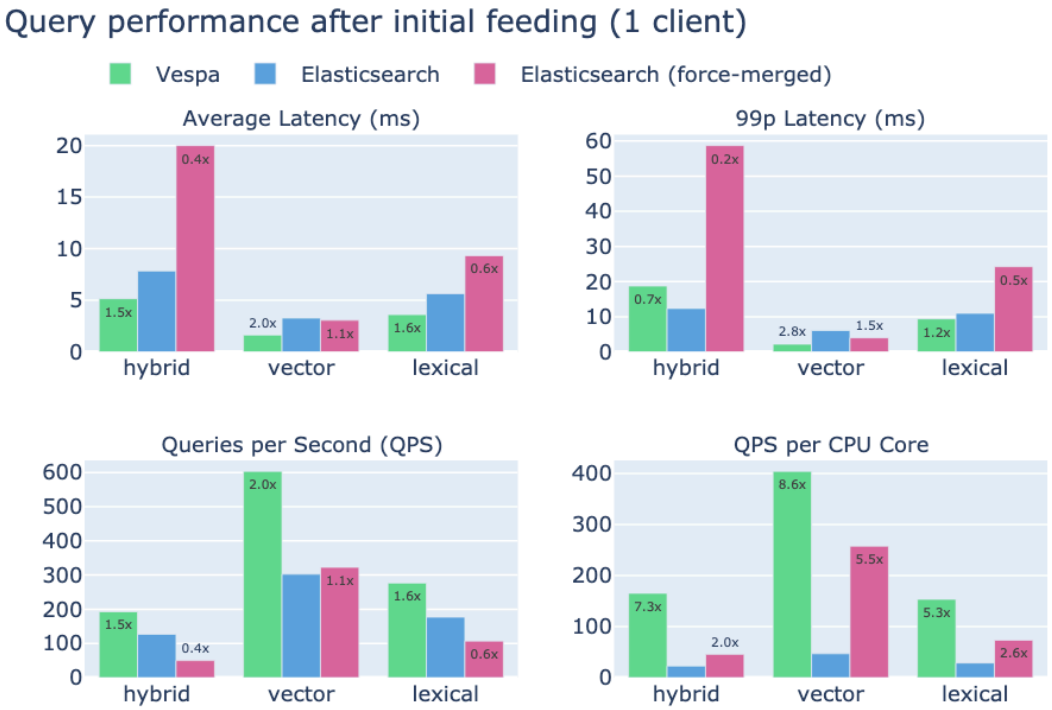
multiple threads, Elasticsearch is able to achieve lower latency than the variant with a single segment.

We can also see the same in the absolute Queries per Second. But, when we look at efficiency, the relative QPS per CPU core, the force-merged variant has the upper hand. The reason is simple, the multi-segment variant uses more CPU resources to achieve the lower latency. This demonstrates that just reporting latency at low concurrency doesn't draw the full picture. As we have already seen in the result section, the picture changes dramatically once we start increasing user concurrency.

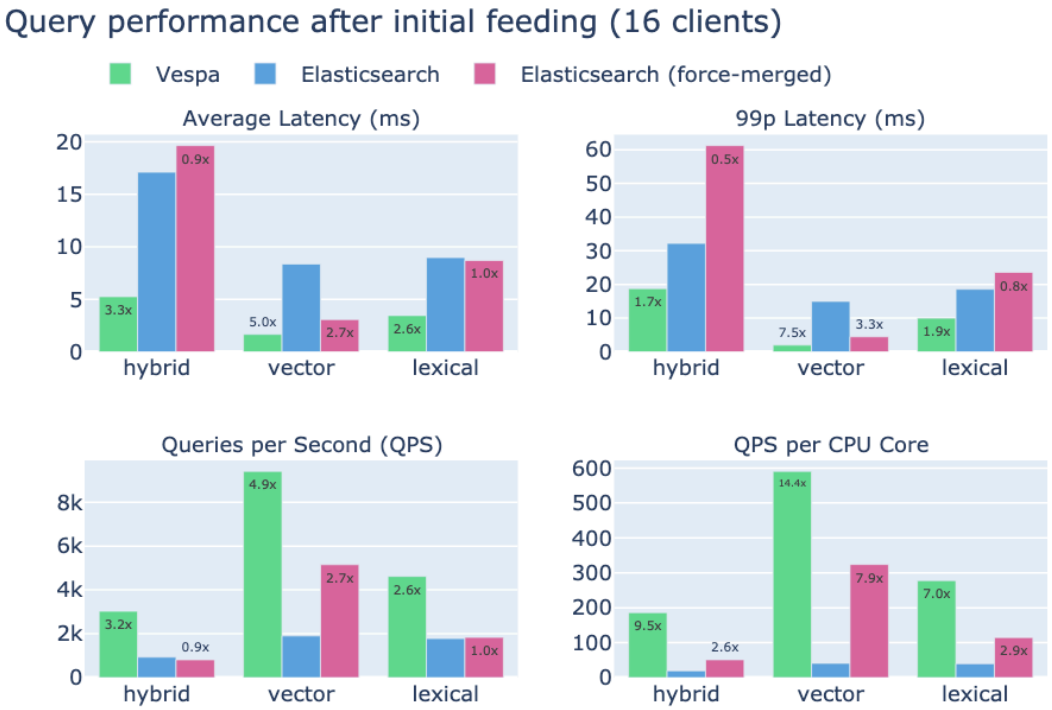


The above demonstrates what happens with much higher concurrency. In this case, the default variant with multi-segments has been pushed far above the knee of the hockey-stick-shaped latency versus utilization graph. Now, we don't want to pay too much attention to the high latency at this point, but notice the throughput differences between Elasticsearch and Elasticsearch (force-merged) where at this level of concurrency, the force-merged variant has the upper hand.

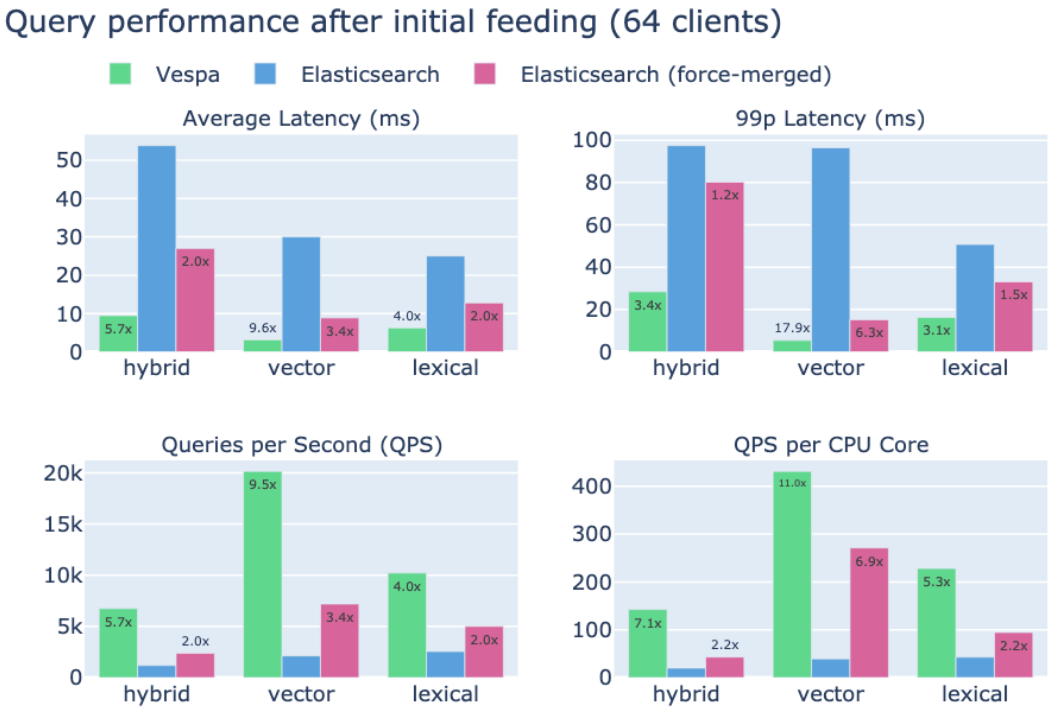
These effects also demonstrate that it is difficult to find the sweet spot for Elasticsearch configuration. The number of segments has a dramatic impact on the performance and the tradeoff between latency, throughput, and efficiency.



This is similar to the 1-client picture from the beginning of this section, but it's for queries without filters.

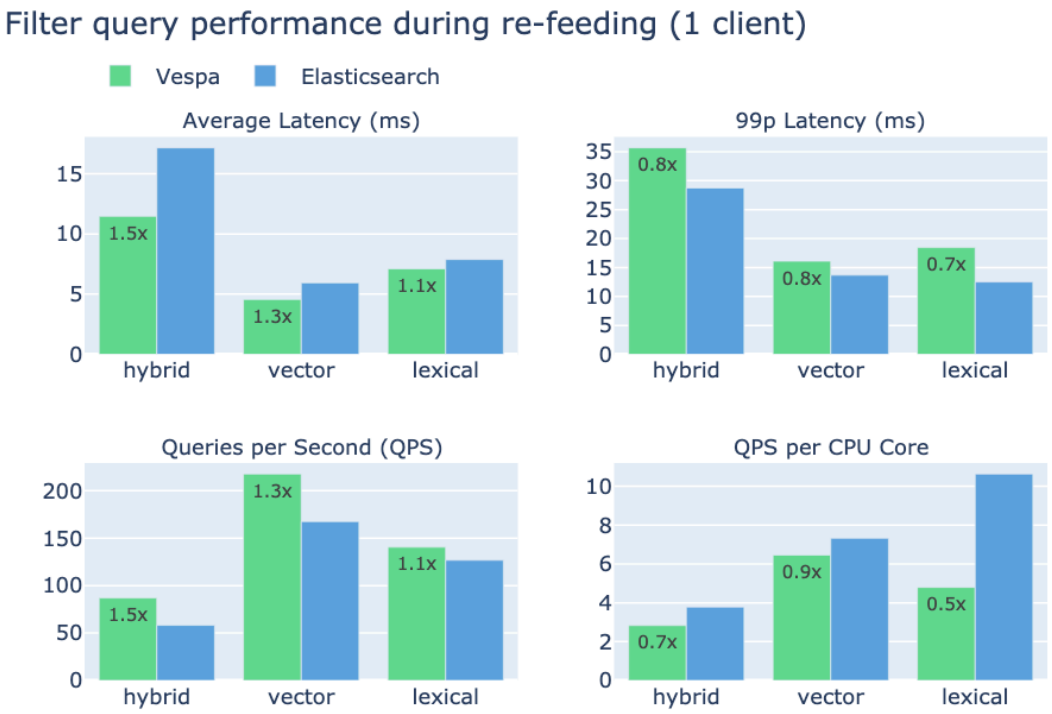


Same here: queries without filters on 16 clients. You'll find the equivalent with filtered queries in section 8.2.1.

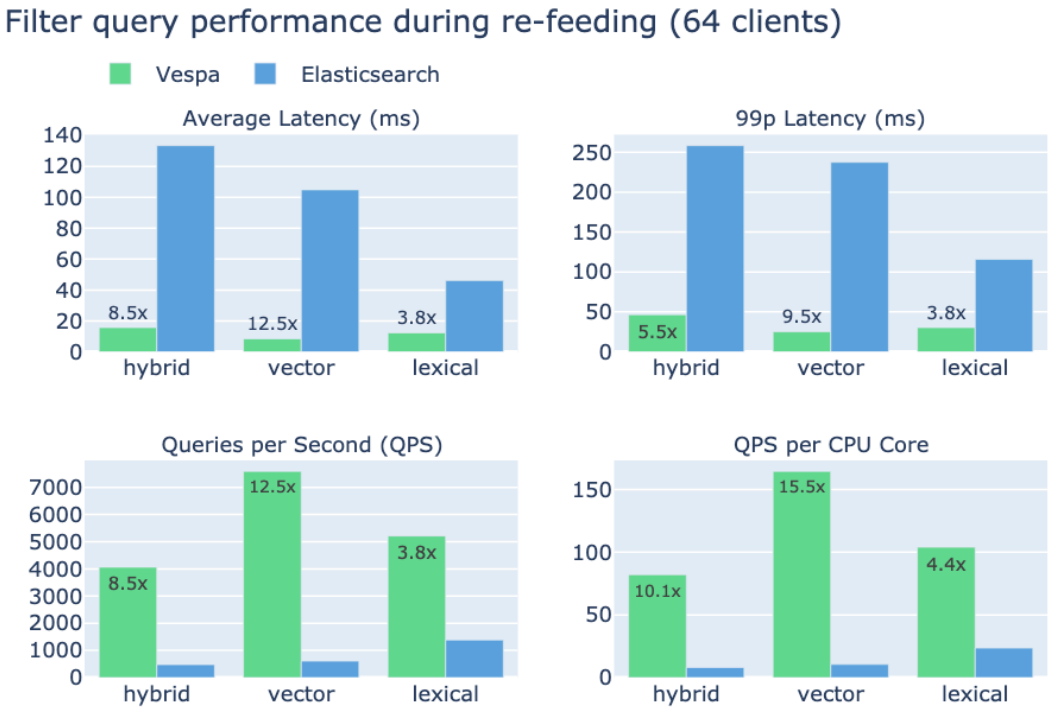


Similarly, unfiltered queries with 64 clients.

A.1.2 Queries with fixed concurrency during refeeding



You'll find the 16-client equivalent, along with our interpretation of it, in section 8.2.1. This single-client variant behaves similarly to the equivalent test at the beginning of Appendix A.1.1, with one important difference: **QPS per CPU Core values are irrelevant, as they include the CPU usage for writes**. And Vespa writes 2.5x faster in this test (see section 8.1 for details).

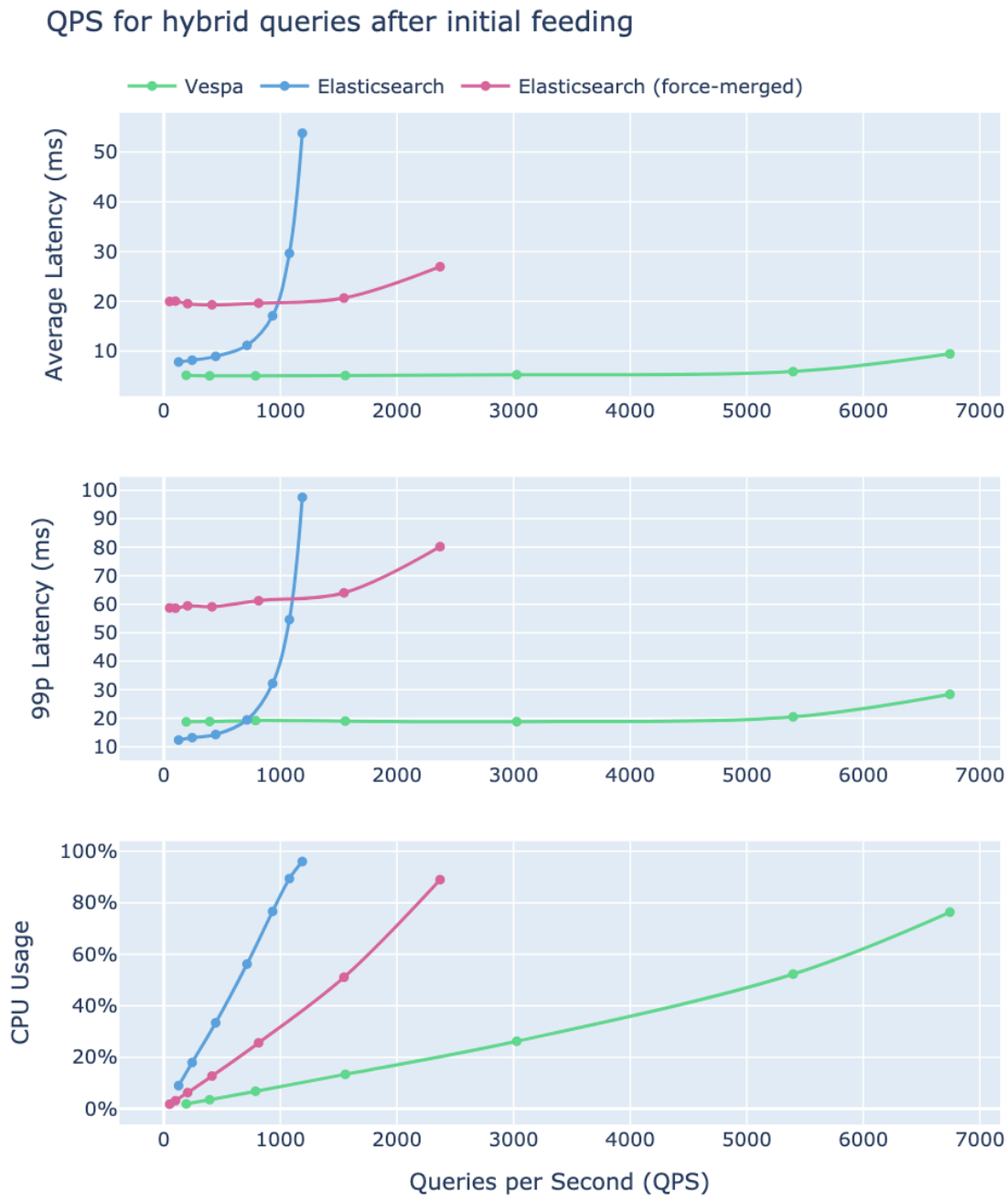


At the other end of the spectrum, when we increase the query load, the CPU is predominantly used by queries. But here **latency numbers become irrelevant for Elasticsearch because it's pushed beyond saturation** and we're effectively only measuring throughput. This is where the hockey-stick graphs paint a clearer picture - see section 8.2.2 and Appendix A.2 below.

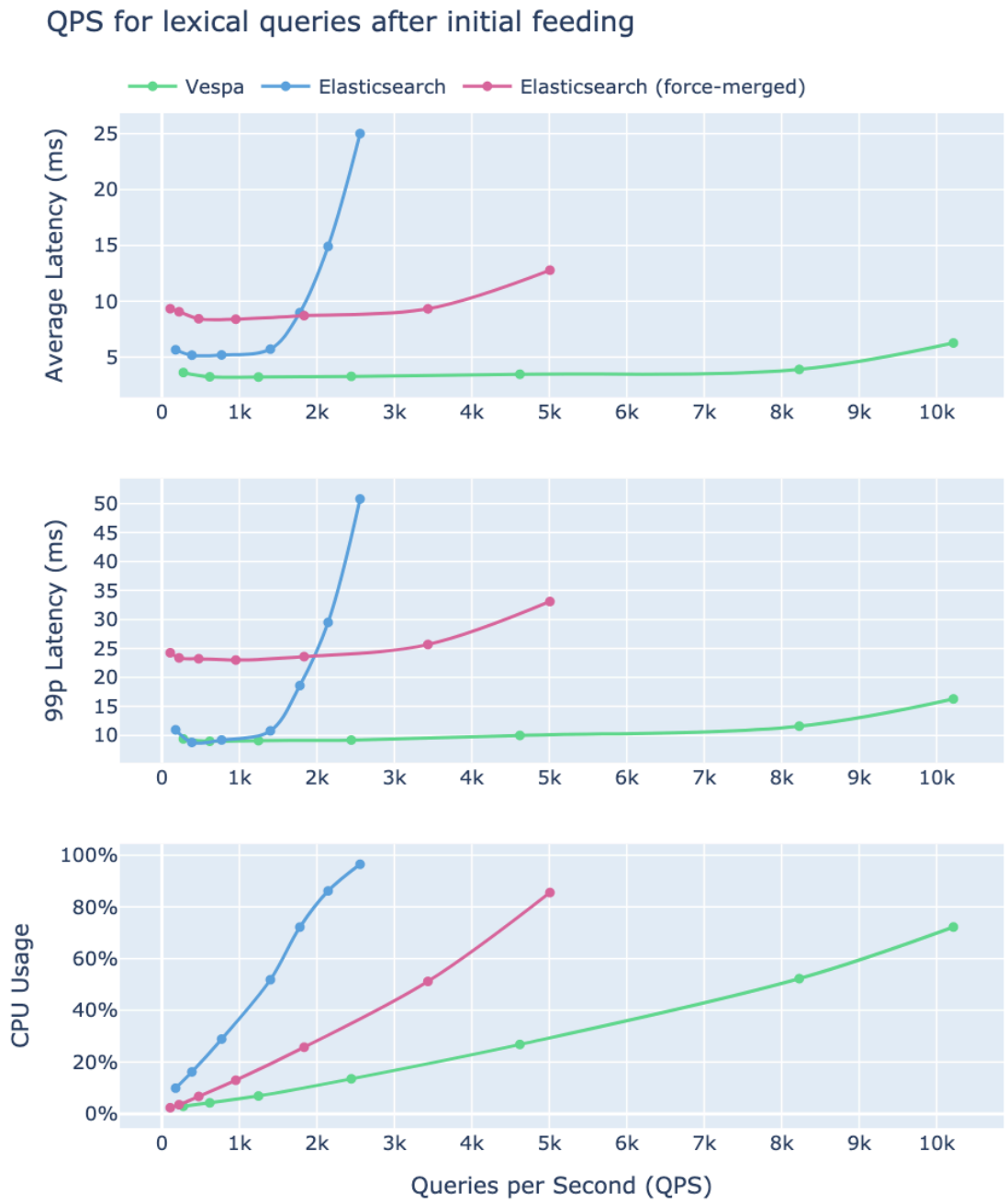
A.2 Scaling with increased load & concurrency

Note that these are queries without filters. Results for queries with filters are in section 8.2.2.

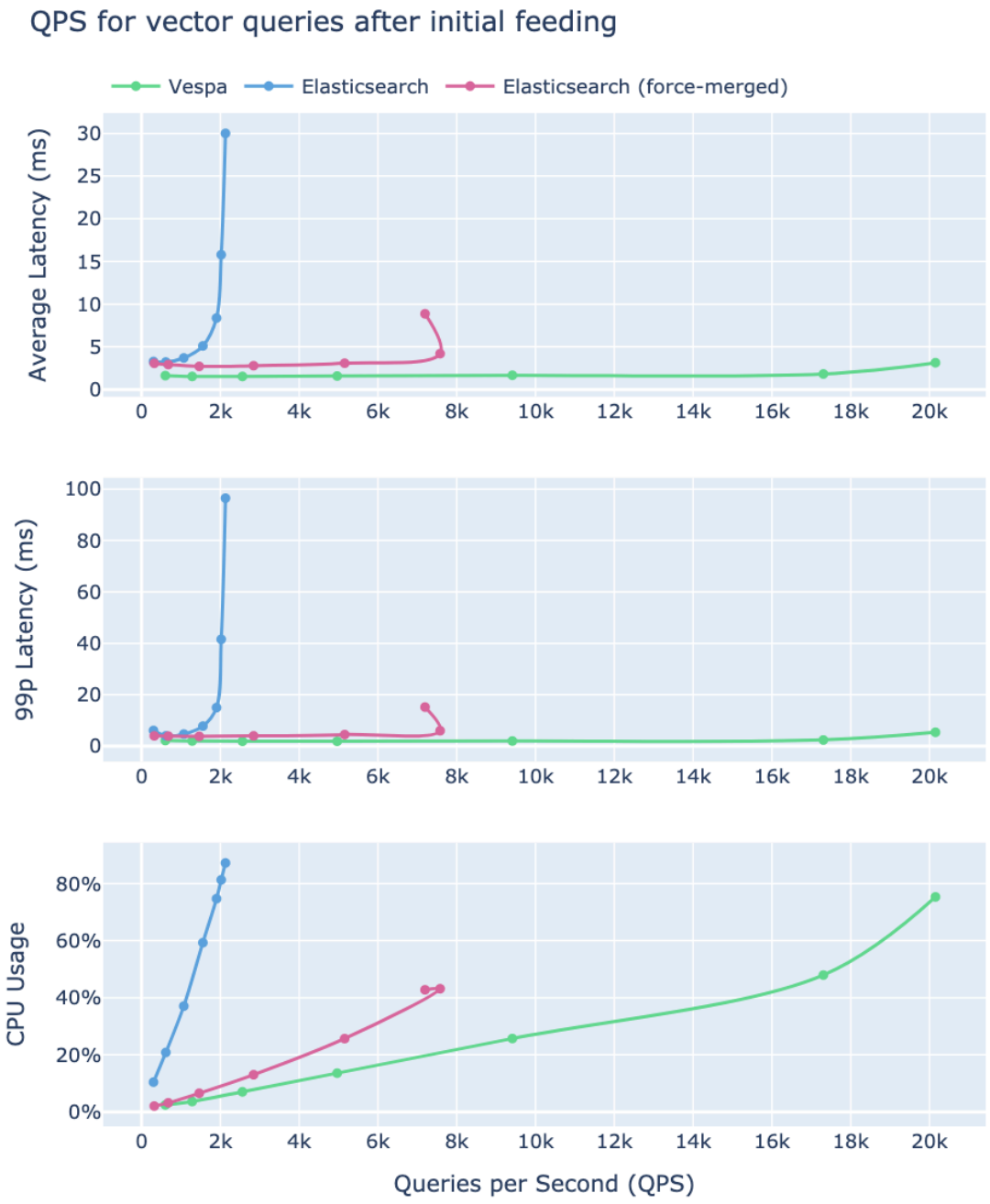
A.2.1 Hybrid search



A.2.2 Lexical search



A.2.3 Vector search



In our analysis of vector search without filters, we observe an unexpected sharp performance degradation in the force-segment variant when CPU usage is low (40%) with 64 clients. While other advanced query types show linear CPU scaling, vector search on a single segment encounters a bottleneck that cannot be explained by CPU

usage alone. The curve bends backward because we achieve lower query throughput with 64 clients than with 32 clients. We did not analyze the root cause of this behavior.

The Elasticsearch multi-segment variant shows a notable difference in CPU usage patterns between lexical and vector search. Vector search exhibits a steeper gradient, indicating poorer scaling performance. This difference can be attributed to their distinct acceleration methods: lexical queries use BM-WAND pruning, while vector search relies on HNSW graph search. The steeper gradient for vector search demonstrates that HNSW, being more sub-linear, is less responsive to increased CPU resources with concurrent segment search. Although Lucene's implementation attempts to share information across intra-query threads for both algorithms, this optimization proves significantly less effective for vector search compared to lexical search.

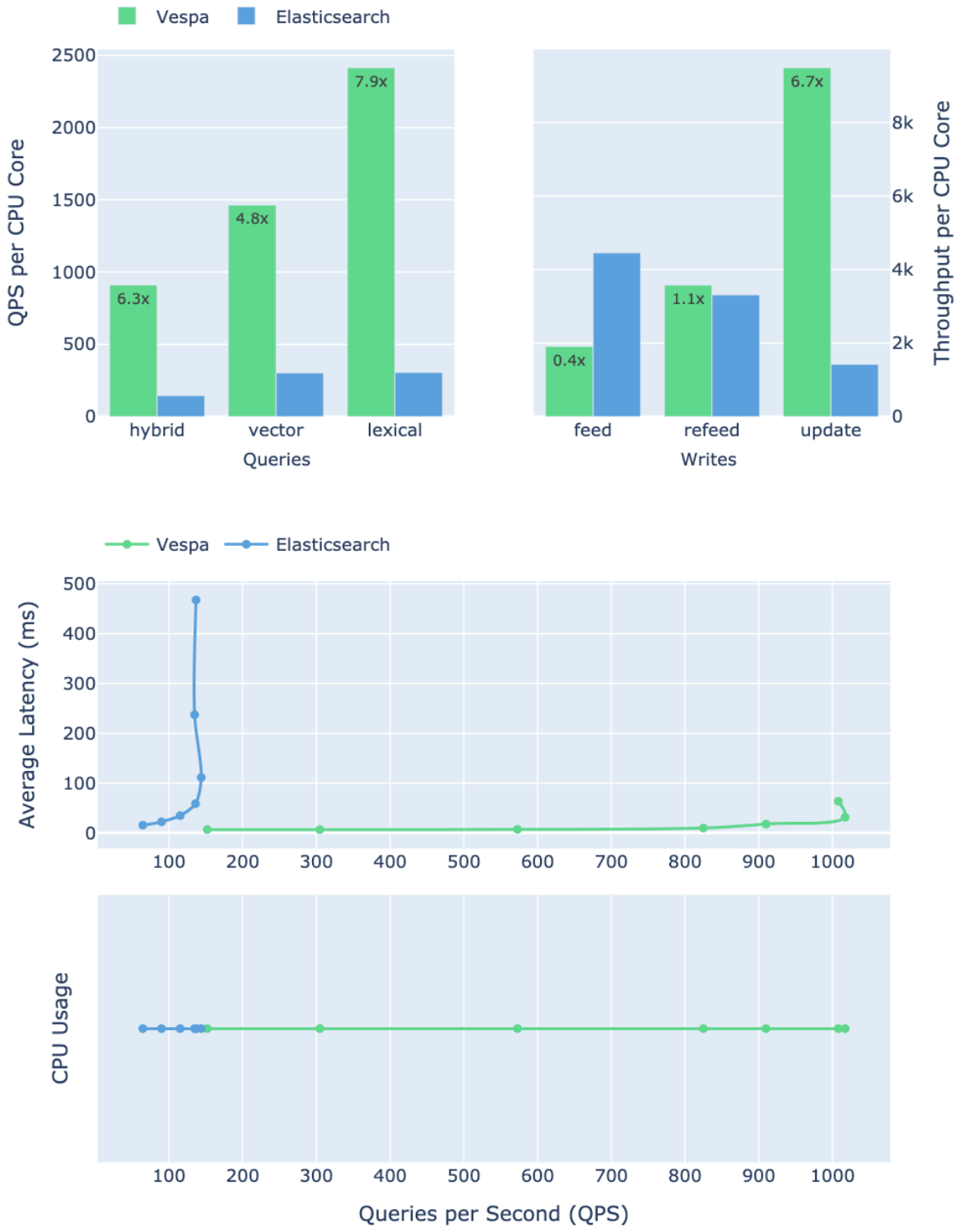
A.3 Impact of less capable hardware

The following presents results from running the same performance test on a Laptop with considerably less CPU-resources than used in the Vespa performance test framework.

Hardware:
Macbook Pro (2021) with Apple M1 pro chip and 32GiB of memory. We set aside 8 CPU threads for the test and 28GiB memory as detailed in section 11.

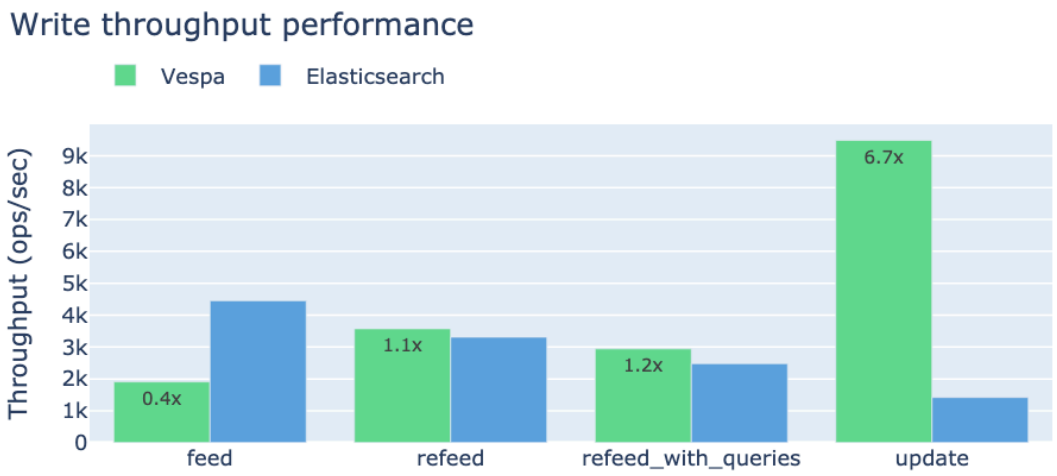
A.3.1 Overall

The following presents results as done in section 1 with two overall illustrations. These are filtered hybrid queries.



Note that since these results are obtained without CPU profiling, CPU usage is flat (1) and the throughput-per-core numbers assume 1 CPU core was used (in reality the test was given 8).

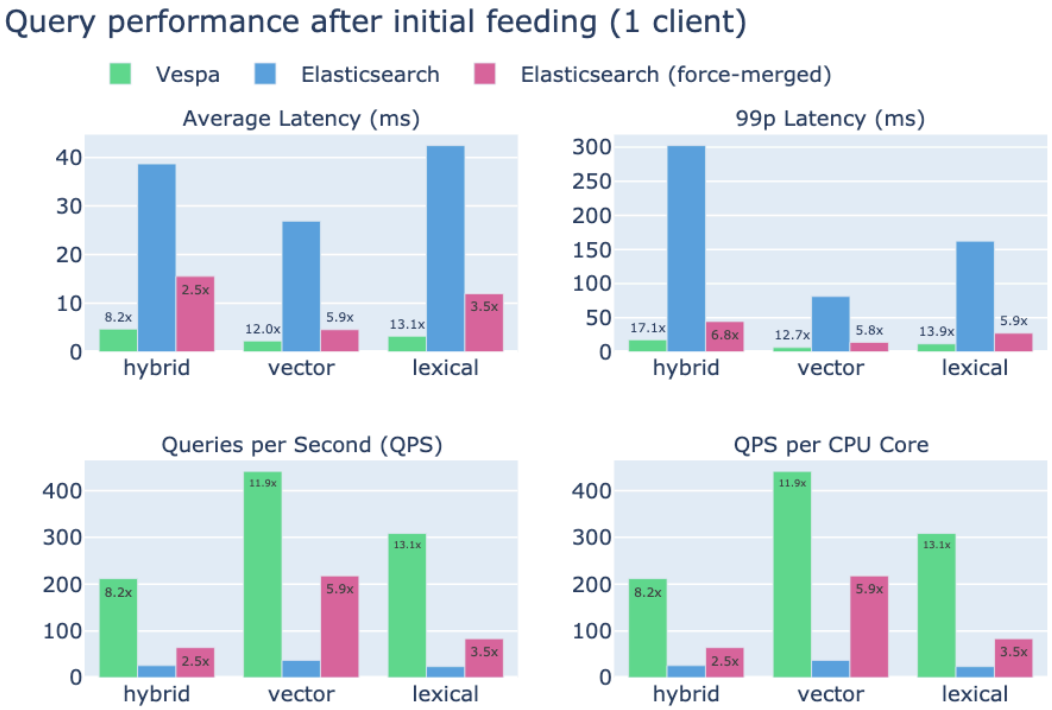
A.3.2 Feed Performance



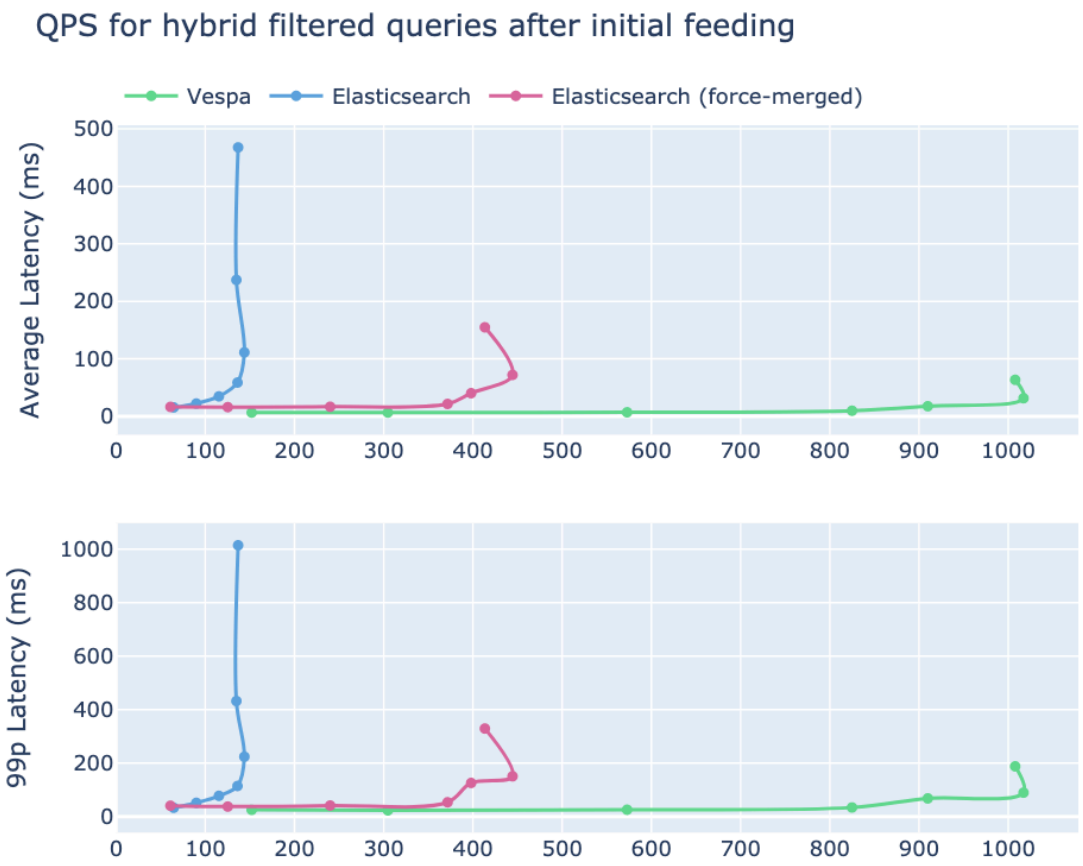
When compared to the results in section 8.1, Elasticsearch shows improved throughput ratios relative to Vespa in this CPU-constrained environment. Specifically, while the more powerful machine showed a ratio of 0.7, this environment achieves a ratio of 0.4 for feed operations. This indicates that Elasticsearch more than doubles its absolute feed throughput advantage over Vespa under CPU-constrained conditions. We don't show throughput-per-core here because it's the same as absolute throughput: in a VM, the test wrongly assumes it uses only one CPU core.

A.3.3 Query Performance

We only include two illustrations because, as demonstrated in the overall section, the picture doesn't change much. Vespa outperforms Elasticsearch also with lower resources available.



Our analysis with a single client reveals an interesting contrast with our earlier findings. While section A.1.1 shows the multi-segment Elasticsearch variant outperforming the force-merged variant in latency tests with one client, these tests show the opposite: the multi-segment variant exhibits worse latency. This discrepancy appears to be resource-dependent. On our less powerful node, the initial feed operation creates 90 segments, compared to approximately 50 segments on the more powerful machine used in the performance test. We attribute this performance reversal to CPU saturation: searching across 90 segments appears to overwhelm the less powerful node even with just a single client.



The image above illustrates system performance with increased concurrency and load for the filtered hybrid query type. As noted earlier, in these we don't see lower latency for the multi-segment variant versus the force-merged variant. Presumably the low CPU resources available and the many segments to search cause overload already at 1 client.