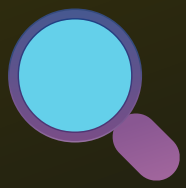

Elasticsearch / OpenSearch

A Complete 11-Chapter Databases Course

Topics covered:

Search-first architecture, the 2021 licensing fork, and the inverted index

Query DSL, BM25 relevance scoring, and aggregations/faceted search

Distributed sharding & replication, and an honest decision framework

Exercises: 33 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Single standalone course · search-first, not relational — a genuinely different center of gravity

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 11

What Elasticsearch/OpenSearch Actually Is — Search-First, Not Relational

Elasticsearch / OpenSearch

Chapter 1 · What Elasticsearch/OpenSearch Actually Is — Search-First, Not Relational

`postgres1-6` deferred a real, dedicated comparison against a purpose-built search engine to this exact course. This is the site's first genuinely search-first, distributed-by-design database — a different center of gravity from everything covered so far.

A Genuinely Different Center of Gravity

Every other database engine on this site treats search or text-matching as either absent entirely or a secondary feature layered onto a different primary data model. MySQL and Postgres are fundamentally relational, with full-text search as a real but genuinely secondary add-on (`postgres1-6`). MongoDB is fundamentally document-oriented. Redis is fundamentally an in-memory key-value/data-structure store. SQLite is fundamentally an embedded relational engine.

Elasticsearch and OpenSearch flip this arrangement. The inverted index (a term-to-document mapping, formally explained in `search1-4`) and relevance scoring (formally explained in `search1-6`) are the primary, founding data structure and query model — not a bolt-on feature. Structured, analytical querying (`search1-7`'s own aggregations) is built *on top of* that search-first foundation, the reverse of Postgres's own "relational engine with search bolted on" arrangement.

Distributed by Design From Day One

`postgres1-11`'s own replication and `mongodb2-6`'s own sharding are both real, mature capabilities — but both were added to engines whose original design center was a single-node system, later extended to support multiple nodes. Elasticsearch and OpenSearch were built distributed from the very beginning: running a "cluster" of one or more nodes is the normal way to run either engine even in development, with sharding and replication as foundational architectural concepts baked into the core design from day one, not layered onto a single-node-first architecture afterward. `search1-8` covers this in full.

What This Actually Looks Like

There's no SQL at all — everything is communicated as JSON documents over a REST/HTTP API, covered fully in `search1-3`. A "document" resembles MongoDB's own documents in shape, but is indexed specifically for full-text search retrieval and relevance ranking, not just storage and retrieval by key or simple query.

Contrasted Against Every Other Engine on This Site

ENGINE	PRIMARY MODEL	SEARCH'S ROLE
MySQL/Postgres	Relational, ACID	An add-on feature (postgres1-6)
MongoDB	Document, flexible schema	Not a design center
Redis	In-memory key-value/data structures	Not a focus at all
SQLite	Embedded relational	Minimal, via an extension
Elasticsearch/OpenSearch	Search + analytics-first, distributed by design	The founding, primary design center

This Course's Own Throughline

This course exists specifically to close the loop `postgres1-6` opened — showing, in real depth, what a genuinely dedicated search engine looks like, rather than simply naming it as a category to reach for elsewhere.

NOT A GENERAL-PURPOSE PRIMARY DATABASE

Elasticsearch/OpenSearch can store and retrieve structured data, but it is not a general-purpose replacement for a relational or document database — it doesn't offer the same transactional/consistency guarantees `mysql1`, `postgres1`, or `mongodb1` do. In real practice, it's most commonly used *alongside* a primary database — a "search index" kept in sync from a genuine source of truth — rather than as the sole system of record. `search1-10` covers this decision honestly in full.

POSTGRES1-6'S OWN DEFERRED PROMISE, NOW DELIVERED

Everything `postgres1-6` gestured at abstractly — a dedicated search engine's real strengths — gets a full, honest treatment across this entire course, closing that loop chapter by chapter.

Hands-On Exercises

EXERCISE 1

Explain the "different center of gravity" distinction this chapter draws — why is Elasticsearch/OpenSearch's own relationship between search and structured querying the reverse of Postgres's own arrangement?

View solution

EXERCISE 2

Explain what "distributed by design from day one" means, contrasting it with how mongodb2-6's sharding and postgres1-11's replication were added to their respective engines.

View solution

EXERCISE 3

Using this chapter's own warn-box, explain why Elasticsearch/OpenSearch is commonly used alongside a primary database rather than replacing one, and what the "same transactional/consistency guarantees" comparison specifically warns against assuming.

View solution

CHAPTER 1 QUICK REFERENCE

- The inverted index and relevance scoring are the PRIMARY design center, not a bolt-on — the reverse of Postgres's own relational-plus-search arrangement
- Distributed (sharding/replication) from day one — not a capability added later, unlike mongodb2-6's sharding or postgres1-11's replication
- No SQL — JSON documents over a REST/HTTP API
- Not a general-purpose primary database — commonly runs alongside a real source-of-truth database, not in place of one
- This course closes the loop postgres1-6 deliberately left open
- **Next chapter:** The Elasticsearch/OpenSearch Split — A Real Licensing Story

CHAPTER 2 OF 11

The Elasticsearch/OpenSearch Split — A Real Licensing Story

Elasticsearch / OpenSearch

Chapter 2 · The Elasticsearch/OpenSearch Split — A Real Licensing Story

`postgres1-1` covered MySQL's Oracle ownership against Postgres's community governance as a stable, long-standing contrast. This chapter covers something different in kind — an active, relatively recent fork, with real, ongoing consequences this course keeps returning to.

Elastic NV's Original Open-Source Release

Elasticsearch is built on Apache Lucene — an existing, mature, open-source search library that both Elasticsearch and OpenSearch still use as their actual underlying engine (previewed here, covered further in `search1-4`). Elasticsearch itself was originally released by Elastic NV under the permissive Apache 2.0 license — genuinely open source by any common definition.

The Cloud-Provider Tension

Elastic NV's own commercial business model centered on paid, hosted/managed Elasticsearch offerings and paid proprietary features layered on top of the open core. AWS began offering its own competing hosted "Amazon Elasticsearch Service," built directly on the same open-source, Apache-licensed code — competing with Elastic's own commercial offering without contributing revenue back to Elastic. This specific tension — a cloud provider running a competing hosted service on top of an open-source project's own code — isn't unique to Elastic; MongoDB made an almost identical move for the same reason a few years earlier, in 2018, introducing the Server Side Public License (SSPL) it created specifically for this situation. Elastic's own later move explicitly followed that real, direct precedent.

The 2021 License Change

In January 2021, Elastic NV announced that future Elasticsearch and Kibana versions would move away from pure Apache 2.0, to a dual-license model offering the Elastic License and SSPL — no longer purely open source in the Apache 2.0 sense. Elastic's own explicit stated reasoning: preventing cloud providers from offering the software as a competing managed service without paying back into the project.

AWS's Fork — OpenSearch Is Born

Rather than accept the new, more restrictive licensing, AWS — along with several other companies — forked the *last Apache 2.0-licensed version* of Elasticsearch and Kibana, creating OpenSearch. This detail matters: OpenSearch isn't a from-scratch rewrite, but a genuine continuation of the last fully open-source Elasticsearch codebase, developed independently from that point forward. Announced in 2021, OpenSearch is now maintained under its own governance structure, with a real, broader contributor base beyond AWS alone.

A More Recent Twist — Elastic's 2024 Return to Open Source

Worth naming honestly rather than treating 2021 as a permanently frozen state of affairs: in 2024, Elastic added AGPL as a third licensing option for Elasticsearch specifically, alongside the existing Elastic License and SSPL options — a genuine, more recent development in an ongoing story, not a settled, static fact from several years ago.

Echoing postgres1-1's Own Governance Material, With a Real Difference

`postgres1-1`'s own MySQL-Oracle-vs-Postgres-community contrast described a stable, long-standing state of affairs — Oracle has owned MySQL for well over a decade, with no active dispute currently reshaping that arrangement. This story is different in kind: a genuine, still-evolving fork with real, ongoing consequences for compatibility and feature parity between the two now-separate codebases — `search1-9` covers exactly where that divergence stands in practice.

THIS CHAPTER DELIBERATELY DOESN'T DECLARE A WINNER

"Which one is better" has a real, current answer that can genuinely change over time as the two projects continue to diverge — or don't. This history chapter deliberately doesn't attempt a final verdict; that's exactly what `search1-9`'s own dedicated, practical comparison chapter exists to do, with specifics that actually need to stay current rather than frozen here.

SEARCH1-9 TURNS THIS HISTORY INTO A PRACTICAL COMPARISON

Everything in this chapter is background for a real decision — `search1-9` is where that decision actually gets made, covering API compatibility and real divergence since the fork.

Hands-On Exercises

EXERCISE 1

Explain what specifically triggered Elastic NV's 2021 license change, and explain the real, direct precedent (MongoDB's own 2018 SSPL move) this chapter names for that kind of decision.

View solution

EXERCISE 2

Explain what OpenSearch actually is in relation to Elasticsearch — what specific version did AWS fork, and why does that detail matter for understanding OpenSearch's own origins?

View solution

EXERCISE 3

Explain the key structural difference this chapter draws between this course's own Elastic/OpenSearch history and postgres1-1's own MySQL-Oracle-vs-Postgres-community material — why is one described as a stable state of affairs and the other as an active, still-evolving situation?

View solution

CHAPTER 2 QUICK REFERENCE

- Elasticsearch originally Apache 2.0, built on Apache Lucene; AWS's own competing hosted service, built on that same open code, created real commercial tension with Elastic NV
- MongoDB's own 2018 SSPL move was the real, direct precedent Elastic explicitly followed in 2021
- January 2021: Elastic moved future Elasticsearch/Kibana versions to a dual Elastic License/SSPL model
- AWS (and others) forked the LAST Apache 2.0-licensed version to create OpenSearch (2021) — a real continuation, not a rewrite
- 2024: Elastic added AGPL as a third licensing option for Elasticsearch — an ongoing, still-evolving story
- Unlike postgres1-1's own stable MySQL-Oracle contrast, this is an active fork with real, ongoing divergence — search1-9 covers the current practical state
- **Next chapter:** Installing & Basic Concepts — Indices, Documents & the REST API

CHAPTER 3 OF 11

Installing & Basic Concepts — Indices, Documents & the REST API

Elasticsearch / OpenSearch

Chapter 3 · Installing & Basic Concepts — Indices, Documents & the REST API

This chapter covers the practical basics both engines share — a real, quiet demonstration of `search1-2`'s own "shared ancestry" point, since installation and the REST API remain largely compatible at this level.

Installing — A Cluster, Even of One

Docker is the common modern way to run either engine locally. `search1-1`'s own "distributed by design" claim becomes concrete immediately: even a single-node local install starts up internally as a cluster, not a standalone server the way `mysql1` or `postgres1-2` install.

Everything Is JSON Over HTTP

No client library is strictly required — plain `curl` works:

```
curl -X GET "localhost:9200"
```

That returns cluster info as JSON, directly. This is a genuine architectural difference from every other engine already covered: `mysql1`'s own `mysql` client, `postgres1-2`'s own `psql`, `sqlite1-2`'s own `sqlite3`, and `mongodb1`'s own `mongosh` are all dedicated client programs speaking a specific, database-specific wire protocol. Elasticsearch/OpenSearch instead exposes a REST API directly — any HTTP client at all can talk to it natively, with no database-specific driver required at the most basic level, even though official client libraries exist for convenience.

Indices — Not Quite a Table, Not Quite a Database

```
curl -X PUT "localhost:9200/products"
```

An **index** is the closest analog to a "table," or arguably a whole "database." It's worth naming a real terminology collision immediately: this "index" is *not* the same concept as a MySQL/Postgres index (the B-

tree/GIN structure `postgres1-7` covered, speeding up queries on an existing table). Here, "index" *is* the primary container itself — analogous to a table or a MongoDB collection — not a secondary structure built on top of one. Getting this vocabulary distinction right immediately avoids real confusion later.

Documents — JSON, Like MongoDB, But Indexed Differently

```
curl -X PUT "localhost:9200/products/_doc/1" -H "Content-Type: application/json" -d '{
  "name": "Wireless Mouse",
  "price": 24.99,
  "in_stock": true
}'
```

A **document** is a single JSON object stored inside an index — structurally similar to `mongodb1-1`'s own document model. The key difference: every field of every document is, by default, automatically analyzed and added to the inverted index (previewed here, covered fully in `search1-4`) at write time — which is exactly why write throughput and query latency trade off differently here than in MongoDB, where storage and indexing remain more separable choices.

No Fixed Schema Required by Default — Dynamic Mapping

By default, both engines automatically infer a "mapping" (their own version of a schema) for a new field the first time they encounter it in a document — genuinely schema-flexible out of the box, in similar spirit to `mongodb1-1`'s own flexible schema. There's a real, important difference worth an honest one-line flag here: once a field's mapping is inferred, it's considerably harder to change afterward than a MongoDB document's own genuinely per-document flexibility.

A Basic Walkthrough

```
curl -X GET "localhost:9200/products/_doc/1"

curl -X GET "localhost:9200/products/_search?q=mouse"
```

That final query previews `search1-5`'s own full Query DSL chapter — this is just enough to confirm the round trip works end to end.

> **DYNAMIC MAPPING'S OWN REAL GOTCHA**
>
> Since a field's type is inferred from its first-ever value, a product SKU that happens to be all-digits in its first indexed document gets permanently mapped as a numeric type — and a later document with a genuinely alphanumeric SKU value for that same field will fail to index correctly, or behave unexpectedly. This is conceptually the same category of problem `sqlite1-3`'s own type affinity gotchas covered — "what happens when the engine has to guess a type" —

though the underlying mechanism differs: SQLite stores each value's own actual type flexibly per-row, while here the mapping is inferred *once* and then effectively locked in for that field.

THIS IS BASIC-CONCEPTS TERRITORY — DEPTH COMES NEXT

Everything introduced here at a basic level — the inverted index, real querying, mapping in depth — gets its own full treatment in `search1-4` and `search1-5`.

Hands-On Exercises

EXERCISE 1

Explain why Elasticsearch/OpenSearch's REST-API-first design is a genuine architectural difference from every other database engine already covered on this site, which all require a dedicated client program.

View solution

EXERCISE 2

Explain the terminology collision between an Elasticsearch/OpenSearch "index" and a MySQL/Postgres "index," and why getting this vocabulary distinction right matters.

View solution

EXERCISE 3

Using this chapter's own warn-box, explain the dynamic mapping gotcha with a concrete example (a numeric-looking SKU), and explain how this is conceptually similar to, but mechanically different from, sqlite1-3's own type affinity gotchas.

View solution

CHAPTER 3 QUICK REFERENCE

- No dedicated client program required — plain HTTP/JSON, unlike every other engine on this site
- **Index** — the primary container (like a table/database), NOT the same concept as a MySQL/Postgres index structure
- **Document** — a JSON object, similar to MongoDB's own model, but every field is analyzed into the inverted index at write time
- Dynamic mapping — schema inferred automatically on first write, genuinely flexible but much harder to change once set than MongoDB's own per-document flexibility

- Dynamic mapping's own gotcha (a numeric-looking value locking in a numeric type) echoes sqlite1-3's own type-guessing problem, mechanically different underneath
- **Next chapter:** The Inverted Index — How Search Actually Works Underneath

CHAPTER 4 OF 11

The Inverted Index — How Search Actually Works Underneath

Elasticsearch / OpenSearch

Chapter 4 · The Inverted Index — How Search Actually Works Underneath

`search1-1` named the inverted index as this engine's own primary structure. This chapter formally explains what it actually is — and revisits a structure this site already covered once before, at a smaller scale.

What an Inverted Index Actually Is

A "forward" index maps a document to its own content — the way a book's table of contents maps a chapter to a page. An **inverted** index does the opposite: it maps each individual term to the list of documents containing it — exactly like a book's own back-of-book index, mapping a word to the page numbers it appears on.

A Concrete Worked Example

Indexing two tiny documents — `"the quick brown fox"` (doc 1) and `"the lazy dog"` (doc 2) — produces an inverted index roughly shaped like this:

```
the    -> [doc 1, doc 2]
quick  -> [doc 1]
brown  -> [doc 1]
fox    -> [doc 1]
lazy   -> [doc 2]
dog    -> [doc 2]
```

A search for "fox" becomes a direct, structured lookup of the term `fox` in this table — not a scan through every document's own raw text.

Analysis — Turning Text Into Terms

Before terms go into the inverted index, text passes through an **analyzer**: tokenization (splitting text into individual words), normalization (lowercasing), and often stemming (reducing words to a root form — "running" → "run") and stop-word removal. This directly echoes `postgres1-6`'s own `to_tsvector()` material

— lexemes, stop words, stemming — except here it's the actual foundation the entire engine is built on, not a bolt-on feature reached for occasionally.

Revisiting postgres1-7's Own GIN Material

`postgres1-7` explained GIN as storing "a mapping from each individual component of a value... to the list of rows containing it." That's genuinely, literally the same core concept — an inverted index — just applied at smaller scale. The real difference is architectural: in Postgres, GIN is one index type among several (per `postgres1-7`'s own comparison table — B-tree/GIN/GiST/BRIN/Hash), reached for specifically when JSONB containment or full-text search is needed, while the rest of the engine is built around ordinary structured relational data. Here, per `search1-1`'s own throughline, the inverted index isn't one option among several — it's the engine's own primary, foundational structure, used for every field by default unless deliberately configured otherwise.

Why This Explains Fast Full-Text Search at Scale

Because searching means directly looking up a term in the inverted index — a fast, structured operation — rather than scanning every document's own raw text, search stays fast even as the number of documents grows very large. This is the exact same underlying performance principle GIN itself relies on in Postgres (`postgres1-7`), just built here as the engine's own foundational structure rather than an add-on index type.

Postings Lists & What Else They Store

A postings list doesn't just record *which* documents contain a term — it typically also stores term **frequency** (how many times the term appears in that document) and **position** (where in the document). This extra information is exactly what `search1-6`'s own relevance-scoring chapter needs.

DOCUMENTS ARE EFFECTIVELY IMMUTABLE UNDERNEATH

Because a change to one document can require updating potentially many different postings lists across the inverted index — every term that document contained — Elasticsearch/OpenSearch documents are actually immutable at the underlying segment level once written. An "update" is really implemented as marking the old document deleted and indexing a brand-new one. This is conceptually reminiscent of `postgres1-9`'s own MVCC material (a new tuple per update), though for a genuinely different underlying reason and mechanism — worth naming honestly rather than glossing over.

TERM FREQUENCY AND POSITION FEED DIRECTLY INTO SEARCH1-6

The postings-list details introduced here — frequency and position — are exactly the raw material `search1-6`'s own relevance-scoring chapter (TF-IDF/BM25) is built on.

Hands-On Exercises

EXERCISE 1

Explain what an inverted index is, using this chapter's own book-index analogy, and walk through the worked example showing what the resulting structure looks like for the two sample documents.

View solution

EXERCISE 2

Explain the direct connection this chapter draws to postgres1-7's own GIN material — what's the same underlying concept, and what's the real structural difference in how each engine actually uses it?

View solution

EXERCISE 3

Using this chapter's own warn-box, explain why documents are effectively immutable at the underlying segment level, and explain the conceptual (not mechanical) echo this has with postgres1-9's own MVCC material.

View solution

CHAPTER 4 QUICK REFERENCE

- **Inverted index** — term → list of documents, exactly like a book's own back-of-book index
- Analysis: tokenization, normalization, stemming, stop-word removal — echoes postgres1-6's own to_tsvector() material, now foundational rather than a bolt-on
- Same core concept as postgres1-7's own GIN — the real difference is architectural: one index type among several in Postgres vs. the engine's own primary structure here
- Postings lists store term frequency and position, not just document membership — the raw material for search1-6's own relevance scoring
- Documents are effectively immutable at the segment level — an "update" is really delete-old-plus-index-new, echoing postgres1-9's own MVCC in spirit, not mechanism
- **Next chapter:** Querying — Query DSL vs. SQL

CHAPTER 5 OF 11

Querying — Query DSL vs. SQL

Elasticsearch / OpenSearch

Chapter 5 · Querying — Query DSL vs. SQL

`search1-3`'s own basic `_search?q=...` query was just a preview. This chapter covers the real query language — a genuinely different paradigm from SQL, not just different syntax for the same idea.

A Genuinely Different Query Paradigm

SQL (per `mysql2` / `mysql3` / `postgres1`) is a declarative, text-based language with its own dedicated syntax. The Query DSL is expressed as ordinary JSON, sent as the body of an HTTP POST request to a `_search` endpoint — not a special language at all, just structured data describing what to search for, sent over the exact same REST API `search1-3` already introduced.

match vs. term — The Most Important Distinction

A **match** query analyzes the search input the *same way* the field was analyzed at index time (`search1-4`'s own analyzer material) — meant for genuine full-text search against analyzed text fields. Searching `"quick fox"` can match a document containing `"The Quick Brown Fox"`, because both go through identical lowercasing and tokenization.

A **term** query does not analyze the input at all — it looks for an exact match against the raw indexed term, meant for exact-value fields (an ID, a status enum, a keyword-mapped field). This is probably the single most important practical query-writing gotcha in the entire engine: using `term` against an analyzed text field routinely returns zero results, because the raw, unanalyzed search string doesn't match any of the lowercased, tokenized terms actually stored in the inverted index.

bool Queries — Combining Conditions

A **bool** query combines multiple conditions using `must`, `should`, `must_not`, and `filter` clauses — roughly analogous in spirit to SQL's own AND/OR/NOT, but structurally very different (nested JSON objects rather than infix operators).

The `must`-vs-`filter` distinction specifically has no direct SQL equivalent: `must` clauses contribute to relevance scoring (`search1-6`'s own material); `filter` clauses express yes/no criteria that don't affect

scoring at all — and, as a result, are more cacheable and efficient. Ordinary SQL's own `WHERE` clause has no built-in concept of "this condition should also feed a relevance score" at all — this distinction is genuinely new territory.

A Worked Example

```
POST /products/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "description": "wireless mouse" } }
      ],
      "filter": [
        { "range": { "price": { "lte": 50 } } },
        { "term": { "category": "electronics" } }
      ]
    }
  }
}
```

The `match` clause does real, relevance-affecting full-text search; the `filter` clauses narrow the results by exact, non-scoring criteria (price range, category) — a realistic e-commerce-style query, directly previewing `search1-11`'s own capstone.

Comparing the Two Paradigms Side by Side

	SQL	QUERY DSL
Surface form	Text, dedicated syntax	JSON, sent as an HTTP request body
Combining conditions	Infix AND/OR/NOT	Nested must/should/must_not/filter
Execution model	Query planner optimizing a relational plan	Directly invokes the engine's own search/scoring machinery

TERM AGAINST AN ANALYZED FIELD IS THE CLASSIC BEGINNER MISTAKE

A `term` query for `"Quick Fox"` against a text field will typically return zero results, since the inverted index actually stores lowercased, tokenized terms like `quick` and `fox`, not the raw string `"Quick Fox"`. The exact same search, run as a `match` query instead, works correctly — because `match` analyzes the input identically to how the field was analyzed at index time, per this chapter's own earlier explanation.

MUST VS. FILTER SETS UP SEARCH1-6 DIRECTLY

The scoring-vs-non-scoring distinction introduced here is exactly what `search1-6`'s own relevance-scoring chapter builds on next.

Hands-On Exercises

EXERCISE 1

Explain the difference between a match query and a term query, and explain the classic beginner mistake of using term against an analyzed text field, using a concrete before/after example.

View solution

EXERCISE 2

Explain the difference between must and filter clauses inside a bool query — specifically, what does "contributes to relevance scoring" actually mean, and why does SQL's own WHERE clause have no direct equivalent to this distinction?

View solution

EXERCISE 3

Using this chapter's own worked example, explain why combining a match clause and a filter clause in the same bool query is a genuinely common, realistic pattern — what real-world scenario does each clause serve?

View solution

CHAPTER 5 QUICK REFERENCE

- Query DSL — JSON in an HTTP POST body, not a text-based dedicated language like SQL
- **match** — analyzed, for full-text search · **term** — exact, unanalyzed, the classic beginner trap against text fields
- **bool** — must/should/must_not/filter, nested JSON rather than infix operators
- **must** — affects relevance score · **filter** — yes/no, non-scoring, more cacheable — no direct SQL WHERE equivalent
- SQL uses a query-planner-optimized relational plan; Query DSL directly invokes the engine's own search/scoring machinery
- **Next chapter:** Relevance Scoring & Full-Text Search Done Right

CHAPTER 6 OF 11

Relevance Scoring & Full-Text Search Done Right

Elasticsearch / OpenSearch

Chapter 6 · Relevance Scoring & Full-Text Search Done Right

This is the chapter that formally closes the loop `postgres1-6` deliberately left open — a real, in-depth comparison against a purpose-built relevance-ranking algorithm, not just a named category.

The Question Every Search Engine Has to Answer

Given multiple documents that all match a query, how does the engine decide which are the *best* matches, and in what order to return them? This is exactly the question `search1-4`'s own postings-list frequency and position data was collected for, and exactly what `search1-5`'s own must-vs-filter scoring distinction was building toward.

TF-IDF — The Classic Foundation

Term Frequency (TF): how many times does the search term appear in *this* document — a document mentioning "wireless" five times is, all else equal, probably more relevant to a "wireless" search than one mentioning it once.

Inverse Document Frequency (IDF): how *rare* is this term across the whole index — a term appearing in nearly every document tells you less about relevance than a term appearing in only a few. A document matching a rare term is weighted more heavily than one matching a common term.

Combining the two: a TF-IDF score weights a document's own term frequency down for common terms and up for rare ones — the classic, foundational formula underlying modern information retrieval.

BM25 — The Modern Default

BM25 (Best Match 25) has been Elasticsearch/OpenSearch's own actual default scoring algorithm since version 5 — built on the same underlying TF/IDF ideas, with two real, meaningful refinements:

- **Term frequency saturation** — repeating a term more and more within a document gives diminishing returns to the score, rather than scaling linearly forever. A document that says "wireless" 50 times isn't 50 times as relevant as one that says it once — BM25 accounts for this.

- **Field length normalization** — a term match in a genuinely short document or field is weighted more heavily than the identical match in a very long one, since a short document mentioning a term is proportionally more "about" that term.

Formally Delivering on postgres1-6's Own Deferred Comparison

`postgres1-6` covered `ts_rank()` as Postgres's own relevance-ranking function, and explicitly deferred a full comparison to this exact course. Here it is, honestly: `ts_rank()` does incorporate real signals — term-frequency-adjusted weighting via `setweight()`, some proximity awareness — but it's a genuinely simpler, less sophisticated ranking function than BM25. It doesn't implement the same level of length normalization or frequency saturation refinement, and Postgres's own full-text search generally wasn't designed to be tuned to the same degree. This isn't "Postgres's own version is broken" — it's a genuine, honest "smaller-scale approximation vs. purpose-built, heavily-refined algorithm" difference, matching this course's own consistent pattern of fair, not dismissive, comparisons.

Explaining a Score — The explain API

A genuinely useful, real practical tool: an `_explain` endpoint shows exactly how a document's own relevance score was computed, term by term — useful for debugging "why did this result rank above that one" in a real, concrete, inspectable way, rather than treating scoring as an opaque black box.

Boosting — Deliberately Tuning Relevance

Fields or query clauses can be given an explicit "boost" multiplier, deliberately increasing their own contribution to the final score — weighting a product's `name` field higher than its `description` field, for instance. This is directly comparable in spirit to `postgres1-6`'s own `setweight()` material (title vs. body weighting).

SCORES AREN'T COMPARABLE ACROSS DIFFERENT QUERIES

Relevance scores are not stable, portable numbers — a score of 5.2 from one query means nothing compared to a score of 3.8 from a genuinely different query. Scores are only meaningful for ranking results *within* the same single query's own result set, not as an absolute, general "how good" a match is. This is a genuinely common, real source of confusion worth flagging explicitly.

POSTGRES1-6'S OWN DEFERRED PROMISE, CLOSED

This formally closes the loop `postgres1-6` left open. `search1-7` covers aggregations next, resolving that same chapter's own "faceted search as a first-class feature" callout.

Hands-On Exercises

EXERCISE 1

Explain TF-IDF's own two components (term frequency and inverse document frequency) and explain why combining them produces a better relevance signal than term frequency alone.

View solution

EXERCISE 2

Explain BM25's two real refinements over plain TF-IDF (term frequency saturation and field length normalization), each with a concrete example of the problem it solves.

View solution

EXERCISE 3

Using this chapter's own honest comparison, explain specifically what postgres1-6's own ts_rank() function does provide, and what it doesn't provide compared to BM25 — why is this framed as "smaller-scale approximation" rather than "broken"?

View solution

CHAPTER 6 QUICK REFERENCE

- **TF** — how often a term appears in THIS document · **IDF** — how rare the term is across the whole index
- **BM25** — the modern default; adds term-frequency saturation (diminishing returns) and field-length normalization over plain TF-IDF
- Honest comparison: postgres1-6's own ts_rank() provides real signals but is a smaller-scale approximation, not a broken version, of BM25
- The `_explain` API shows exactly how a score was computed, term by term
- Boosting echoes postgres1-6's own setweight() — deliberately weighting one field's contribution over another
- Scores are only meaningful for ranking WITHIN one query's own results — never comparable across different queries
- **Next chapter:** Aggregations — Analytics Built In

CHAPTER 7 OF 11

Aggregations — Analytics Built In

Elasticsearch / OpenSearch

Chapter 7 · Aggregations — Analytics Built In

This chapter resolves `postgres1-6`'s own explicit "faceted search as a first-class feature" callout — a capability genuinely different in kind from SQL's own `GROUP BY`, not just a different syntax for the same idea.

What Aggregations Actually Are

An aggregation computes summary statistics or groupings over the same document set a search query matches — *in the same request* as the search itself. This is a genuinely combined "search + analyze" operation, not two separate steps.

Metric Aggregations — The Simple Case

`avg`, `sum`, `min`, `max`, and `stats` (all of these at once) are conceptually similar to SQL's own aggregate functions (`SUM`, `AVG`, `COUNT`). This part alone isn't dramatically different from SQL.

Bucket Aggregations — Where It Diverges From GROUP BY

A **terms** aggregation groups documents by the distinct values of a field — conceptually the closest analog to SQL's own `GROUP BY` ("how many products are in each category"). The real divergence starts here: aggregations can be nested arbitrarily deep — a `terms` aggregation on `category`, with a nested `terms` aggregation on `brand` within each category, with a nested `avg` aggregation on `price` within each of those — producing a genuinely multi-level breakdown in a single request that would require either a complex multi-level `GROUP BY`/`ROLLUP` construction in SQL, or several separate queries entirely.

range/**histogram** aggregations bucket numeric values into ranges ($0–25, $25–50, $50+) or fixed-width histograms — directly usable for the classic "price range" faceted-search filter UI. **date_histogram** buckets by time interval (documents per day/week/month) — genuinely common in log/analytics use cases.

Combining Search and Aggregations in One Request

```
POST /products/_search
{
  "query": { "match": { "description": "wireless mouse" } },
  "aggs": {
    "by_category": { "terms": { "field": "category" } },
    "price_ranges": {
      "range": {
        "field": "price",
        "ranges": [
          { "to": 25 }, { "from": 25, "to": 50 }, { "from": 50 }
        ]
      }
    }
  }
}
```

This is exactly what powers a real e-commerce faceted search sidebar — search results, category counts ("12 in Electronics, 8 in Home Goods"), and price-range checkboxes, all computed over the same filtered result set, in one round trip. This directly, explicitly resolves `postgres1-6`'s own "faceted search as a first-class feature" callout.

Why This Is a Genuinely Different Capability Class From GROUP BY

SQL's `GROUP BY` operates on stored, structured rows — a natural fit for relational data, but with no built-in way to combine "full-text relevance-ranked search results" and "a faceted breakdown of those same results" in one single, efficient operation, because SQL's own execution model was never built around search-then-facet as a first-class combined pattern. Aggregations here also operate at genuinely large scale efficiently, across distributed shards (previewed here, covered fully in `search1-8`) — a real analytical, OLAP-style workload built into the same engine handling the search itself, rather than requiring a separate analytics system.

AGGREGATING ON AN ANALYZED TEXT FIELD DOESN'T DO WHAT YOU'D EXPECT

Aggregating directly on an analyzed text field creates buckets per individual token, not per whole value — a `category` field containing "Wireless Mouse" would bucket into separate "wireless" and "mouse" buckets, rather than one distinct "Wireless Mouse" category. The fix is a "keyword" sub-field — an exact, unanalyzed mapping alongside the analyzed text field — used specifically for aggregation and exact-matching purposes. This is genuinely the same underlying analyzed-vs-exact tension already seen twice: `search1-3`'s own mapping material, and `search1-5`'s own `match`-vs-`term` distinction — now showing up a third time, in a third context.

POSTGRES1-6'S OWN CALLOUT, RESOLVED

This closes the "faceted search as a first-class feature" item `postgres1-6` named explicitly. `search1-8` covers the distributed sharding/replication architecture that makes aggregations like this fast at real scale.

Hands-On Exercises

EXERCISE 1

Explain the difference between a metric aggregation and a bucket aggregation, and explain how the terms aggregation is the closest conceptual analog to SQL's own GROUP BY.

View solution

EXERCISE 2

Using this chapter's own worked example, explain how combining a search query and aggregations in one request is what actually powers a real e-commerce faceted search sidebar, and explain why this resolves postgres1-6's own "faceted search as a first-class feature" callout specifically.

View solution

EXERCISE 3

Using this chapter's own warn-box, explain the analyzed-field aggregation gotcha with a concrete example, and explain how this is the same underlying analyzed-vs-exact tension already seen in search1-3's mapping material and search1-5's match-vs-term distinction.

View solution

CHAPTER 7 QUICK REFERENCE

- **Metric aggregations** (avg/sum/min/max/stats) — close to SQL's own aggregate functions
- **Bucket aggregations** (terms/range/histogram/date_histogram) — arbitrarily nestable in one request, genuinely beyond a single SQL GROUP BY
- Search + aggregations combined in one request is what powers a real faceted-search sidebar — resolving postgres1-6's own callout
- Aggregations run efficiently across distributed shards — a real OLAP-style workload built into the same engine (search1-8)
- Aggregating on an analyzed field buckets by TOKEN, not by whole value — use a keyword sub-field; the same analyzed-vs-exact tension as search1-3's mapping and search1-5's match-vs-term
- **Next chapter:** Sharding & Replication — Distributed by Design

CHAPTER 8 OF 11

Sharding & Replication — Distributed by Design

Elasticsearch / OpenSearch

Chapter 8 · Sharding & Replication — Distributed by Design

`search1-1` claimed this engine is distributed by design from day one. This chapter delivers the actual mechanics, contrasted directly against two real prior chapters on this site.

Shards — Splitting an Index Across Nodes

A single index is split into multiple **shards** — and each shard is itself a complete, independent Apache Lucene index, the real payoff of `search1-2`'s own namecheck of Lucene as the underlying engine both Elasticsearch and OpenSearch are built on. The number of primary shards for an index is set at index-creation time, and in most real-world default configurations isn't changeable afterward without reindexing — worth naming honestly, though newer versions have added split/shrink APIs to help with this in some cases.

Each shard can live on a different node in the cluster — this is what lets a single index's own data spread across many machines, and what lets a single search query execute in *parallel* across multiple shards and nodes simultaneously. This parallelism is part of why `search1-7`'s own aggregations can stay fast even over huge datasets.

Replicas — Redundancy Built the Same Way

Each primary shard can have one or more **replica** shards — exact copies, kept in sync, living on different nodes than their own primary. If a node holding a primary shard goes down, one of its replicas is automatically promoted to primary, and the cluster keeps serving both reads and writes with no manual intervention required.

This is a real, direct contrast with `postgres1-11`'s own replication material: Postgres's own streaming replication requires deliberate setup — a tool like Patroni or repmgr for automatic failover, per that chapter's own honest material. Here, replica-based redundancy and automatic failover are built into the cluster's own core coordination logic from the start, not something layered on afterward with a separate tool.

Contrasted Against mongodb2-6's Own Sharding

`mongodb2-6` covered sharding as a real, mature, but genuinely *optional* capability, requiring deliberate shard-key design decisions, added specifically when a deployment outgrows a single replica set — a real, substantial engineering decision layered onto an already-complete single-node-capable system.

Here, per `search1-1`'s own throughline, there's no equivalent "un-sharded" mode to fall back to as the default, normal way of running the system. Every index is already conceptually shard-based from the moment it's created — even a tiny single-node dev setup with one shard and no replicas is still, structurally, a one-shard cluster, not an un-sharded system. The same "distributed as the default assumption, not an optional upgrade" pattern `search1-1` named abstractly, now shown concretely against a specific, real comparison point.

Cluster Coordination — How Nodes Agree

A cluster needs a way to elect and maintain a "cluster manager" node, responsible for coordinating cluster-wide state — which shards live where, overall cluster health, and so on. Modern versions use a Raft-like consensus protocol for this. The kind of problem being solved here — getting multiple independent nodes to agree on shared state without a single point of failure — is conceptually related to the same class of problem `postgres1-11`'s own replica-promotion material and `mongodb2-5`'s own replica-set election material both deal with.

A Practical Consequence — Cluster Health

A real, checkable operational signal: **green** (all primary and replica shards allocated), **yellow** (all primaries allocated, but some replicas aren't — a genuinely single-node dev cluster is permanently yellow, since there's nowhere to place a replica), **red** (some primary shards themselves are unallocated — real data may be unavailable).

PRIMARY SHARD COUNT IS A REAL, UPFRONT CAPACITY-PLANNING DECISION

Since primary shard count is typically fixed at index-creation time, choosing too *few* shards limits future horizontal scalability, while choosing too *many* shards for a genuinely small dataset adds real, unnecessary per-shard overhead — each shard has its own real resource cost (file handles, memory, and more). This is a genuine, non-obvious decision that has to be made reasonably well up front, unlike Postgres's own comparatively low-commitment ability to simply add a read replica later. This is a real trade-off of the distributed-by-default design, not a free lunch.

SEARCH1-1'S OWN ROADMAP ENTRY, DELIVERED

This formally closes the "sharding & replication" item from `search1-1`'s own roadmap. `search1-9` turns to a genuinely practical comparison of Elasticsearch and OpenSearch next.

Hands-On Exercises

EXERCISE 1

Explain what a shard is and how splitting an index into shards enables both horizontal scaling and parallel query execution, tying your answer to search1-2's own Apache Lucene namecheck.

View solution

EXERCISE 2

Explain the contrast this chapter draws between automatic replica-based failover here and postgres1-11's own replication material — what's the key structural difference in how "built in" automatic failover actually is?

View solution

EXERCISE 3

Using this chapter's own warn-box, explain the primary-shard-count capacity-planning gotcha — why is this a real, upfront trade-off that doesn't have as direct an equivalent in Postgres's own "add a read replica later" flexibility?

View solution

CHAPTER 8 QUICK REFERENCE

- **Shard** — a complete, independent Lucene index; primary count fixed at index-creation time in most real setups
- Shards enable both horizontal scaling and parallel query execution across nodes
- **Replicas** — automatic redundancy and failover, built into the cluster's own core coordination, unlike postgres1-11's own deliberately-tooled Patroni/repmgr setup
- Distributed by default from index creation — no "un-sharded" fallback mode, unlike mongodb2-6's own optional, later-added sharding
- Cluster manager election via Raft-like consensus — conceptually related to postgres1-11's and mongodb2-5's own replica-election material
- Cluster health: green (fully allocated) / yellow (primaries only) / red (primary shards missing)
- Shard count is a real, upfront capacity-planning trade-off — too few limits scale, too many wastes resources
- **Next chapter:** Elasticsearch vs. OpenSearch in Practice

CHAPTER 9 OF 11

Elasticsearch vs. OpenSearch in Practice

Elasticsearch / OpenSearch

Chapter 9 · Elasticsearch vs. OpenSearch in Practice

`search1-2` deliberately deferred a real verdict to this chapter. Here it is — as honestly as the still-evolving nature of the situation allows.

Where the Two Projects Still Agree

The core REST API surface remains largely compatible for basic operations — indexing documents, basic search, basic aggregations — a real, practical consequence of both projects sharing the same starting codebase (`search1-2`'s own "last Apache-licensed version" fork point). Many client libraries and tools can work against either engine for common, basic operations with little or no modification.

Where They've Genuinely Diverged

Since the fork, each project has developed its own features independently, on separate release cycles, without the other — over time, this means some newer functionality genuinely exists in only one of the two, not both. Newer, added products have diverged in naming and branding too: OpenSearch Dashboards is a separate, independently-developed product built specifically for OpenSearch, not a shared or interchangeable tool with Kibana, even though the two started from a shared ancestor per `search1-2`. Version numbering has diverged independently as well — Elasticsearch and OpenSearch version numbers no longer correspond to each other at all; they're two separate projects with their own independent numbering, not the same underlying version wearing two labels.

This chapter deliberately avoids listing specific, dated feature comparisons as if they were permanently true — matching `search1-2`'s own "still-evolving" honesty. The durable guidance is to check current, up-to-date documentation for whichever specific feature actually matters to a real project.

Licensing Implications for a Real Deployment Decision

This is a direct callback to `search1-2`'s own full licensing history: Elasticsearch's own licensing (Elastic License/SSPL, with AGPL added back as an option in 2024) versus OpenSearch's own licensing — OpenSearch has remained genuinely, fully open-source under Apache 2.0 since its own creation. This is a real, concrete

decision point: does a specific deployment genuinely require or prefer a permissive, unambiguous open-source license (favoring OpenSearch), or is Elastic's own dual/triple-licensed model — with its own specific proprietary features and official commercial support options — actually the better practical fit?

Worth noting honestly: hosted/managed offerings exist for both — Elastic's own official Elastic Cloud, and AWS's own managed OpenSearch Service alongside other third-party managed OpenSearch offerings. This isn't simply "self-hosted vs. vendor-hosted" — both projects have real commercial hosting ecosystems around them.

A Practical Approach to Choosing

Rather than a static "X is better at Y" table that risks going stale quickly (an honest acknowledgment this course itself makes, given how recently and rapidly this situation has evolved), the durable guidance is:

1. Check the specific licensing terms against the deployment's own real constraints — an open-source requirement, a commercial-support need, or a specific proprietary feature dependency.
2. Check current, up-to-date documentation for whichever specific feature the project genuinely needs, since API/feature parity is a moving target.
3. Consider existing team/organizational familiarity and existing tooling compatibility, since both are now genuinely mature, capable options for most common use cases.

The Ecosystem Around Each

Both have real, growing plugin/extension ecosystems, and both integrate with common ingestion and dashboarding tooling. Whatever "ecosystem gap" exists at any given moment is exactly the kind of detail that needs current verification rather than being settled once and treated as permanent.

"COMPATIBLE TODAY" IS NOT "INTERCHANGEABLE FOREVER"

Because of the ongoing, independent divergence this chapter covers, a client library, plugin, or piece of tooling written or tested against one of the two engines is not guaranteed to keep working correctly against the other, even if it happens to work today. Genuinely testing against the specific engine and version actually being deployed — rather than assuming interchangeability based on shared ancestry — is the responsible practice.

SEARCH1-2'S OWN HISTORY, NOW MADE PRACTICAL

Every fact from `search1-2`'s own history chapter now has a real, concrete practical consequence here. `search1-10`'s own decision-framework chapter turns the whole course's own material — not just this specific Elasticsearch-vs-OpenSearch question — into one honest, usable framework.

Hands-On Exercises

EXERCISE 1

Explain why the core REST API remains largely compatible between the two engines for basic operations, tying your answer back to search1-2's own fork material, and explain why this compatibility isn't guaranteed to remain complete or permanent.

View solution

EXERCISE 2

Explain the real licensing implications this chapter draws for an actual deployment decision — what's the concrete difference between Elasticsearch's own licensing model and OpenSearch's, and what kind of deployment constraint would favor each?

View solution

EXERCISE 3

Using this chapter's own warn-box, explain why "compatible today" isn't the same as "interchangeable forever," and explain the responsible practice this chapter recommends instead of assuming interchangeability.

View solution

CHAPTER 9 QUICK REFERENCE

- Basic REST API operations remain largely compatible — a direct consequence of the shared fork point (search1-2)
- Independent development since the fork means genuine, growing divergence in features, product naming (Kibana vs. OpenSearch Dashboards), and version numbering
- Licensing: Elasticsearch's own Elastic License/SSPL/AGPL vs. OpenSearch's own consistent Apache 2.0 — a real, concrete deployment constraint to check
- Both have real commercial hosting ecosystems — not simply "self-hosted vs. vendor-hosted"
- Durable guidance: check licensing against real constraints, check current docs for specific features, weigh team familiarity — not a static comparison table
- Compatible today ≠ interchangeable forever — test against the specific engine/version actually deployed
- **Next chapter:** When to Reach for a Dedicated Search Engine

CHAPTER 10 OF 11

When to Reach for a Dedicated Search Engine

Elasticsearch / OpenSearch

Chapter 10 · When to Reach for a Dedicated Search Engine

This is this course's own central chapter — where nine chapters of architecture, indexing, querying, scoring, aggregations, and distribution stop being separate facts and become one honest, usable decision framework, directly answering the question `postgres1-6` deliberately left open.

When Postgres's Own Full-Text Search Is Enough

`postgres1-6`'s own guidance remains accurate, and this chapter isn't here to overturn it: Postgres's own built-in full-text search is the right choice "when search is a secondary feature of a primarily-relational application at moderate scale." Concrete signals: search is one feature among several in an otherwise-relational application; the dataset doesn't genuinely require horizontal distribution; the team doesn't want to operate an entirely separate system alongside the primary database; and `ts_rank()`'s own simpler relevance model (`search1-6`) is genuinely sufficient for the actual use case — a "does this roughly match" search, not sophisticated relevance tuning.

When a Dedicated Search Engine Is Genuinely Worth It

- **Search is the primary product**, not a secondary feature — a job-search site, a documentation search product, an e-commerce search-and-discovery experience where search quality directly drives revenue.
- **Genuinely large scale** requiring real horizontal distribution (`search1-8`) that a single Postgres server or read-replica setup can't practically match.
- **Faceted search/aggregations** (`search1-7`) are a first-class, heavily-used product requirement, not an occasional nice-to-have.
- **Sophisticated relevance tuning** (`search1-6`'s own BM25/boosting material) genuinely matters to the product's own success, not just "good enough" matching.
- **Log/observability-style analytics workloads** — a real, common, adjacent use case this course didn't go deep on, worth an honest namecheck.

The Real Operational Cost — Not Free

A direct callback to `search1-1`'s own warn-box and `search1-8`'s own distributed-architecture material: running a dedicated search engine means operating an entirely separate distributed system — real infrastructure, real operational expertise, and real ongoing complexity keeping two systems in sync. The primary database remains the source of truth, per `search1-1`'s own warn-box, and the search index has to be kept up to date *from* that source of truth — real, ongoing engineering work, not a one-time setup cost. This has to be weighed honestly against the real benefits — "search is genuinely important to us" alone doesn't automatically justify this cost if `postgres1-6`'s own built-in capability would actually suffice.

A Practical Decision Framework

1. Is search a *primary* product feature, or a secondary convenience? (Secondary → `postgres1-6` is probably enough.)
2. Does the dataset/query volume genuinely require horizontal distribution beyond what a single well-resourced Postgres server can handle?
3. Is faceted search/real-time aggregation a genuine, heavily-used product requirement?
4. Does relevance *quality* itself materially affect the product's own success — not just "does the search work at all"?
5. Is the team prepared to operate and maintain an entirely separate distributed system, keeping it in sync with a real source of truth?

If most of these point toward a genuine "yes," a dedicated search engine is likely worth its real cost. If most point toward "no" or "not really," `postgres1-6`'s own built-in full-text search is very likely the better, simpler choice.

The Honest Middle Ground

Not every real decision has a clean answer — some cases are genuinely ambiguous, and reasonable people disagree. This framework's own goal, matching `sqlite1-7`'s own precedent, isn't a rigid flowchart that removes judgment — it's making sure that judgment is informed by the real trade-offs covered across this entire course.

> **"SEARCH IS IMPORTANT TO OUR BUSINESS" ISN'T THE SAME AS "WE NEED A DEDICATED ENGINE"**
>
> A common, real, flawed pattern: reasoning that because search is important to a business, it must require Elasticsearch or OpenSearch. "Important" doesn't automatically mean "requires a dedicated distributed system" — the actual determining factor, per this chapter's own framework, is scale and sophistication requirements, not importance alone. A lot of genuinely important search features are served perfectly well by `postgres1-6`'s own built-in capability, and reaching for a separate system prematurely adds real operational cost, per this chapter's own "Real Operational Cost" section, for no actual corresponding benefit.

POSTGRES1-6'S OWN DEFERRED QUESTION, FINALLY ANSWERED

This chapter formally closes the question `postgres1-6` deliberately left open. `search1-11`'s own capstone is the final synthesis, combining faceted filtering, relevance ranking, and aggregations into one real feature.

Hands-On Exercises

EXERCISE 1

An internal company wiki has a basic search box used occasionally by 50 employees. Using this chapter's own decision framework, decide whether postgres1-6's own built-in search or a dedicated engine is the better fit.

View solution

EXERCISE 2

Explain the real operational cost this chapter names for choosing a dedicated search engine, and why "search is important" alone doesn't automatically justify that cost.

View solution

EXERCISE 3

Using this chapter's own material and postgres1-6's own original guidance, explain specifically what determines the answer to "when is a dedicated search engine actually worth it" — name at least three of this chapter's own concrete determining factors.

View solution

CHAPTER 10 QUICK REFERENCE

- postgres1-6's own guidance still holds — secondary search feature at moderate scale, ts_rank() likely sufficient
- A dedicated engine earns its cost when search IS the product, scale genuinely demands distribution, faceting/aggregations are first-class, or relevance quality itself drives success
- Real operational cost: an entirely separate distributed system, kept in sync with a real source of truth — not a one-time setup
- Five-question framework: primary feature? genuine distribution need? first-class facets? relevance-quality-driven success? team ready to operate a separate system?
- Some cases are genuinely ambiguous — the framework informs judgment, it doesn't replace it
- "Important" ≠ "needs a dedicated engine" — scale/sophistication, not importance alone, is the real determining factor

- **Next chapter:** Capstone — Building a Real Search & Analytics Feature

CHAPTER 11 OF 11

Capstone: Building a Real Search & Analytics Feature

Elasticsearch / OpenSearch

Chapter 11 · Capstone: Building a Real Search & Analytics Feature

Ten chapters covered what makes this engine genuinely different, closing two real loops (`postgres1-6`'s own deferred search comparison, and its own faceted-search callout) along the way. This capstone builds one real, working product search feature — but only after confirming, using `search1-10`'s own framework, that building it here is actually justified.

The Scenario

An e-commerce company with a genuinely large product catalog is deciding whether to build a dedicated product search feature. Applying `search1-10`'s own five-question framework directly: search *is* the storefront's own primary feature, not a secondary convenience (question 1 → yes); the catalog is genuinely large enough to benefit from real horizontal distribution (question 2 → yes); faceted filtering (category, brand, price) is a first-class, heavily-used requirement for this storefront (question 3 → yes); relevance quality genuinely affects conversion and revenue (question 4 → yes); and the team is prepared to operate a separate system and keep it synced with the real product database (question 5 → assumed yes). Unlike `search1-10`'s own two exercise scenarios — both genuine "no" cases — this is a real, complete "yes" case, closing that chapter's own framework from the other direction.

The Mapping

```
PUT /products
{
  "mappings": {
    "properties": {
      "name": { "type": "text", "fields": { "keyword": { "type": "keyword" } } },
      "description": { "type": "text" },
      "category": { "type": "text", "fields": { "keyword": { "type": "keyword" } } },
      "brand": { "type": "text", "fields": { "keyword": { "type": "keyword" } } },
      "price": { "type": "float" }
    }
  }
}
```

This mapping is explicit and deliberate, not left to dynamic inference — avoiding both `search1-3`'s own dynamic-mapping gotcha and `search1-7`'s own analyzed-field aggregation gotcha directly, by giving `category` and `brand` a `keyword` sub-field specifically for exact filtering and aggregation, alongside their own analyzed text version for full-text search.

The Combined Query

```
POST /products/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "multi_match": {
            "query": "wireless mouse",
            "fields": ["name^3", "description"]
          }
        }
      ],
      "filter": [
        { "term": { "category.keyword": "Electronics" } },
        { "term": { "brand.keyword": "Logitech" } },
        { "range": { "price": { "gte": 10, "lte": 60 } } }
      ]
    }
  },
  "aggs": {
    "by_category": { "terms": { "field": "category.keyword" } },
    "by_brand": { "terms": { "field": "brand.keyword" } },
    "price_ranges": {
      "range": {
        "field": "price",
        "ranges": [ { "to": 25 }, { "from": 25, "to": 50 }, { "from": 50 } ]
      }
    }
  }
}
```

`name^3` boosts matches in the product name three times higher than matches in the description — `search1-6`'s own boosting material, applied for real. The `must` clause drives relevance-ranked results; the `filter` clauses narrow by exact, non-scoring criteria against the `keyword` sub-fields; the `aggs` block computes category, brand, and price-range facet counts over that same filtered result set, in the same single request.

Why This Works — Tracing the Mechanism

The ranked results come from BM25 scoring (`search1-6`) computed via the inverted index (`search1-4`). The facet counts come from aggregations (`search1-7`) computed in parallel across shards (`search1-8`). The

whole request executes as fast as it does specifically because of the distributed, parallel architecture `search1-8` covered — every piece of this capstone traces back to a specific, real mechanism this course actually explained, not just a feature list.

Chapter Attribution

CAPSTONE ELEMENT	CHAPTER
Confirming a dedicated engine is justified for this use case	search1-10
Explicit mapping, keyword sub-fields avoiding two prior gotchas	search1-3, search1-7
multi_match, bool query with must/filter	search1-5
Field boosting (name^3)	search1-6
Nested terms/range aggregations for facets	search1-7
Parallel execution across shards making it fast at scale	search1-8
Underlying inverted index and BM25 scoring mechanism	search1-4, search1-6

HONEST SCOPE NOTE

This capstone does not build an actual data-sync pipeline from a real source-of-truth database — `search1-1`'s own warn-box named this as real, necessary, ongoing engineering work, and it remains a genuinely separate project of its own, out of scope here. It also doesn't build autocomplete/suggest functionality, typo-tolerance/fuzzy-matching tuning, or a real cluster deployment/production-hardening walkthrough — `search1-8`'s own sharding/replication material stayed conceptual throughout this course. This capstone demonstrates exactly what `search1-10`'s own framework confirmed was justified for this specific scenario — not a claim that every possible feature of a production search system is "done."

THE THROUGHLINE, CLOSED

`search1-1` opened this course by naming the inverted index and relevance scoring as this engine's own primary, founding design center — not a bolt-on feature. This capstone is the proof: one real request, genuinely combining search, filtering, relevance, and analytics, built entirely on the mechanisms this course actually explained.

Hands-On Exercises

EXERCISE 1

Walk through this chapter's own application of search1-10's five-question framework to the e-commerce scenario, and explain why this capstone represents a genuine "yes" case, complementing search1-10's own two "no" exercise scenarios.

View solution

EXERCISE 2

Explain why the capstone's own mapping gives category and brand a keyword sub-field, tying your answer to both search1-3's dynamic-mapping gotcha and search1-7's analyzed-field aggregation gotcha, with a concrete example of what would go wrong without it.

View solution

EXERCISE 3

Using this chapter's own "Why This Works" section, trace the combined query's own results back to the specific underlying mechanisms (inverted index, BM25, sharding) covered earlier in this course.

View solution

CHAPTER 11 QUICK REFERENCE — COURSE COMPLETE

- search1-10's own framework applied first, confirming a dedicated engine is genuinely justified for this scenario — a real "yes" case
- Explicit mapping with keyword sub-fields deliberately avoids search1-3's and search1-7's own gotchas
- One combined request: relevance-ranked, boosted, faceted, and filtered — search1-5/6/7 working together
- Fast at scale because of search1-8's own distributed, parallel shard execution
- Honest scope note: no real data-sync pipeline, no autocomplete, no fuzzy matching, no production deployment walkthrough
- This closes the full 11-chapter Elasticsearch/OpenSearch course, and the entire Databases subject's own bucket list