

Chapter 18: Search Systems

Introduction: Why Search is Hard

Search seems simple but becomes complex at scale.

Simple Search (Small Dataset):

100 products	
Search: "laptop"	
Method: SQL LIKE query	
Time: 10ms	
✓ Works fine	

Complex Search (Large Dataset):

100 million products	
Search: "laptop"	
SQL LIKE query: 30 seconds!	
User: Left the site	
✗ Too slow	

Search Engine (Optimized):

100 million products	
Search: "laptop"	
Inverted index: 50ms	
User: Happy	
✓ Fast enough	

1. Full-Text Search Fundamentals

What is Full-Text Search?

Definition: Finding documents that match a query in a large collection of text.

Simple String Matching (SQL):

```
sql
```

-- **X** SLOW for large datasets

```
SELECT * FROM products
WHERE name LIKE '%laptop%'
OR description LIKE '%laptop%';
```

Problems:

1. Full table scan (checks every row)
2. No ranking (all results equal)
3. No fuzzy matching ("laptp" won't match "laptop")
4. No relevance scoring
5. Can't handle synonyms ("computer" won't match "laptop")

Time: O(n) where n = number of products

For 100M products: 30+ seconds

Full-Text Search Engine:

- ✓ Inverted index (pre-computed)
- ✓ Relevance ranking (best matches first)
- ✓ Fuzzy matching (typo tolerance)
- ✓ Synonym support ("computer" matches "laptop")
- ✓ Phrase matching ("gaming laptop")
- ✓ Filters (price, category, rating)

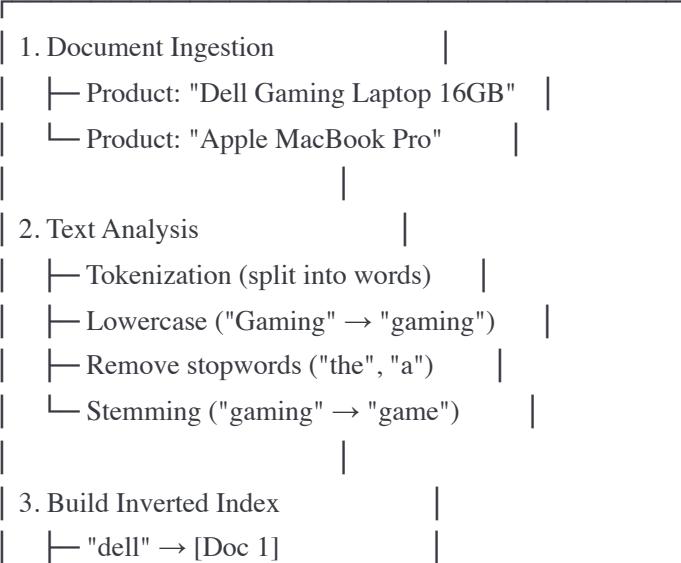
Time: O(log n) with index

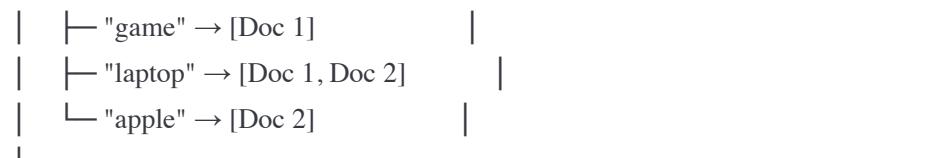
For 100M products: 50ms

600x faster!

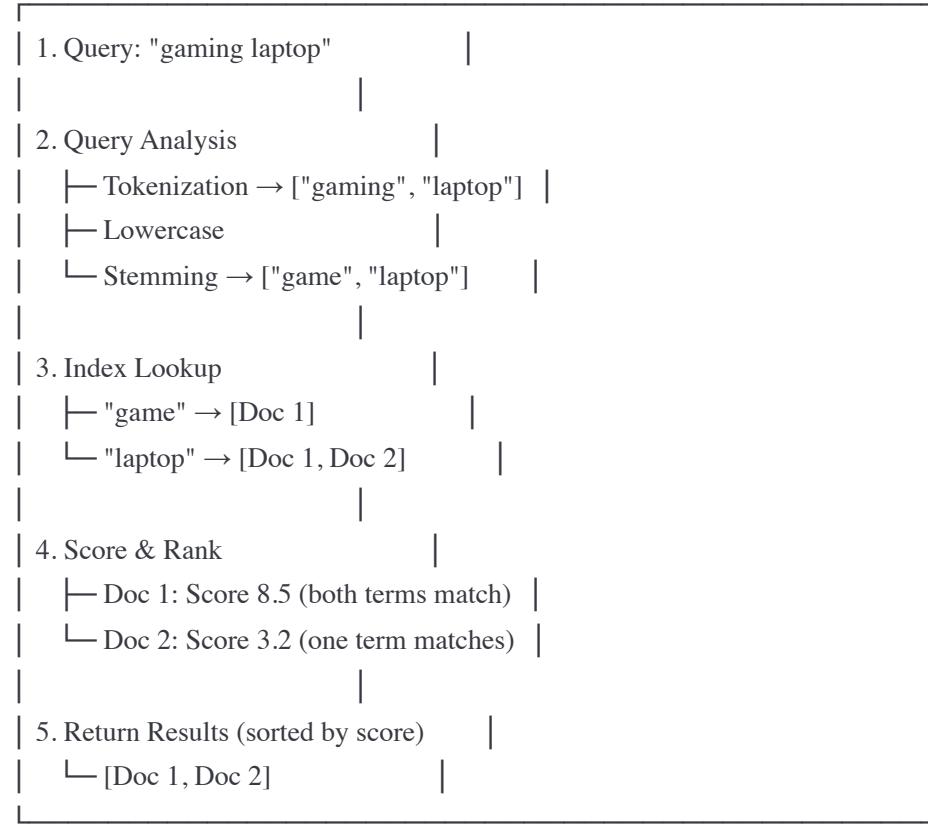
Search Process Flow

INDEXING (Offline):





SEARCHING (Online):



2. Inverted Index

What is an Inverted Index?

Forward Index (Document → Words):

Document 1: "The quick brown fox"

Document 2: "Quick brown dogs"

Document 3: "Lazy brown cats"

Forward Index:

Doc	Words
1	[the, quick, brown, fox]
2	[quick, brown, dogs]
3	[lazy, brown, cats]

Query: "brown"

Must scan all documents → $O(n)$ → SLOW!

Inverted Index (Word → Documents):

Inverted Index:

Term	Documents
brown	[Doc 1, Doc 2, Doc 3]
cats	[Doc 3]
dogs	[Doc 2]
fox	[Doc 1]
lazy	[Doc 3]
quick	[Doc 1, Doc 2]
the	[Doc 1]

Query: "brown"

Direct lookup → [Doc 1, Doc 2, Doc 3] → $O(1)$ → FAST!

Inverted Index with Positions

Enhanced index stores word positions for phrase matching.

Document 1: "The quick brown fox jumps"

Position: 0 1 2 3 4

Inverted Index with Positions:

Term	Postings (Doc ID, Positions)
brown	(Doc1, [2]), (Doc2, [1])
fox	(Doc1, [3])
jumps	(Doc1, [4])
quick	(Doc1, [1]), (Doc2, [0])
the	(Doc1, [0])

Phrase Query: "quick brown"

1. Find "quick": Doc1[1], Doc2[0]
2. Find "brown": Doc1[2], Doc2[1]
3. Check if positions adjacent:
 - Doc1: positions 1,2 ✓ Adjacent!
 - Doc2: positions 0,1 ✓ Adjacent!
4. Results: [Doc1, Doc2]

Phrase Query: "brown jumps"

1. Find "brown": Doc1[2]
2. Find "jumps": Doc1[4]
3. Check if adjacent:
 - Doc1: positions 2,4 ✗ Not adjacent (gap of 1)
4. Results: []

Building an Inverted Index

python

```
class InvertedIndex:
    def __init__(self):
        self.index = {} # term -> [(doc_id, positions)]
        self.documents = {} # doc_id -> original text

    def add_document(self, doc_id, text):
        """Add document to index"""
        self.documents[doc_id] = text

        # Tokenize and analyze
        tokens = self.analyze(text)

        # Build index
        for position, token in enumerate(tokens):
            if token not in self.index:
                self.index[token] = []

            # Find or create posting for this document
            posting = next((p for p in self.index[token] if p['doc_id'] == doc_id), None)

            if posting:
                posting['positions'].append(position)
            else:
                self.index[token].append({
                    'doc_id': doc_id,
                    'positions': [position]
                })

    print(f"Indexed document {doc_id}: {len(tokens)} tokens")

def analyze(self, text):
    """Text analysis pipeline"""

    # 1. Lowercase
    text = text.lower()

    # 2. Tokenization
    import re
    tokens = re.findall(r'\w+', text)

    # 3. Remove stopwords
    stopwords = {'the', 'a', 'an', 'and', 'or', 'but', 'in', 'on', 'at'}
    tokens = [t for t in tokens if t not in stopwords]

    # 4. Stemming (simplified)
    tokens = [self.stem(t) for t in tokens]
```

```

return tokens

def stem(self, word):
    """Simple stemming (remove common suffixes)"""
    suffixes = ['ing', 'ed', 's', 'es']
    for suffix in suffixes:
        if word.endswith(suffix):
            return word[:-len(suffix)]
    return word

def search(self, query):
    """Search for documents matching query"""
    # Analyze query same way as documents
    query_tokens = self.analyze(query)

    if not query_tokens:
        return []

    # Get postings for each term
    results = {}

    for token in query_tokens:
        if token in self.index:
            for posting in self.index[token]:
                doc_id = posting['doc_id']

                if doc_id not in results:
                    results[doc_id] = {
                        'doc_id': doc_id,
                        'score': 0,
                        'matched_terms': []
                    }

                results[doc_id]['score'] += 1
                results[doc_id]['matched_terms'].append(token)

    # Sort by score (simple relevance)
    sorted_results = sorted(
        results.values(),
        key=lambda x: x['score'],
        reverse=True
    )

    return sorted_results

def search_phrase(self, phrase):
    """Search for exact phrase"""

```

```

tokens = self.analyze(phrase)

if len(tokens) < 2:
    return self.search(phrase)

# Find documents containing all terms
doc_sets = []
for token in tokens:
    if token in self.index:
        docs = [p['doc_id'] for p in self.index[token]]
        doc_sets.append(set(docs))
    else:
        return [] # Term not found

# Intersection (documents with all terms)
common_docs = set.intersection(*doc_sets)

# Check if terms are adjacent
phrase_matches = []

for doc_id in common_docs:
    # Get positions for each term in this document
    term_positions = []
    for token in tokens:
        posting = next(p for p in self.index[token] if p['doc_id'] == doc_id)
        term_positions.append(posting['positions'])

    # Check for consecutive positions
    if self.has_consecutive_positions(term_positions):
        phrase_matches.append({
            'doc_id': doc_id,
            'score': 10 # Higher score for phrase match
        })

return phrase_matches

def has_consecutive_positions(self, term_positions):
    """Check if terms appear consecutively"""
    if not term_positions:
        return False

    # Check if first term position + 1 is in second term positions
    for pos1 in term_positions[0]:
        expected = pos1
        match = True

        for positions in term_positions[1:]:
            if expected not in positions:
                match = False
                break
            expected += 1
        if not match:
            break
    return match

```

```

    if expected not in positions:
        match = False
        break
    expected += 1

if match:
    return True

return False

def get_index_stats(self):
    """Get index statistics"""
    return {
        'total_documents': len(self.documents),
        'total_terms': len(self.index),
        'index_size': sum(len(postings) for postings in self.index.values())
    }

# Usage Example
index = InvertedIndex()

# Add documents
index.add_document('doc1', 'The quick brown fox jumps over the lazy dog')
index.add_document('doc2', 'Quick brown dogs are friendly')
index.add_document('doc3', 'The lazy cat sleeps')
index.add_document('doc4', 'Gaming laptop with 16GB RAM and fast processor')

# Search
print("\n==== Search: 'brown' ====")
results = index.search('brown')
for result in results:
    print(f"Doc {result['doc_id']}: Score {result['score']}")
    print(f" Text: {index.documents[result['doc_id']]}\n")

# Phrase search
print("\n==== Phrase Search: 'quick brown' ====")
results = index.search_phrase('quick brown')
for result in results:
    print(f"Doc {result['doc_id']}: Score {result['score']}")
    print(f" Text: {index.documents[result['doc_id']]}\n")

# Statistics
print("\n==== Index Statistics ====")
print(index.get_index_stats())

# Output:
# Indexed document doc1: 7 tokens

```

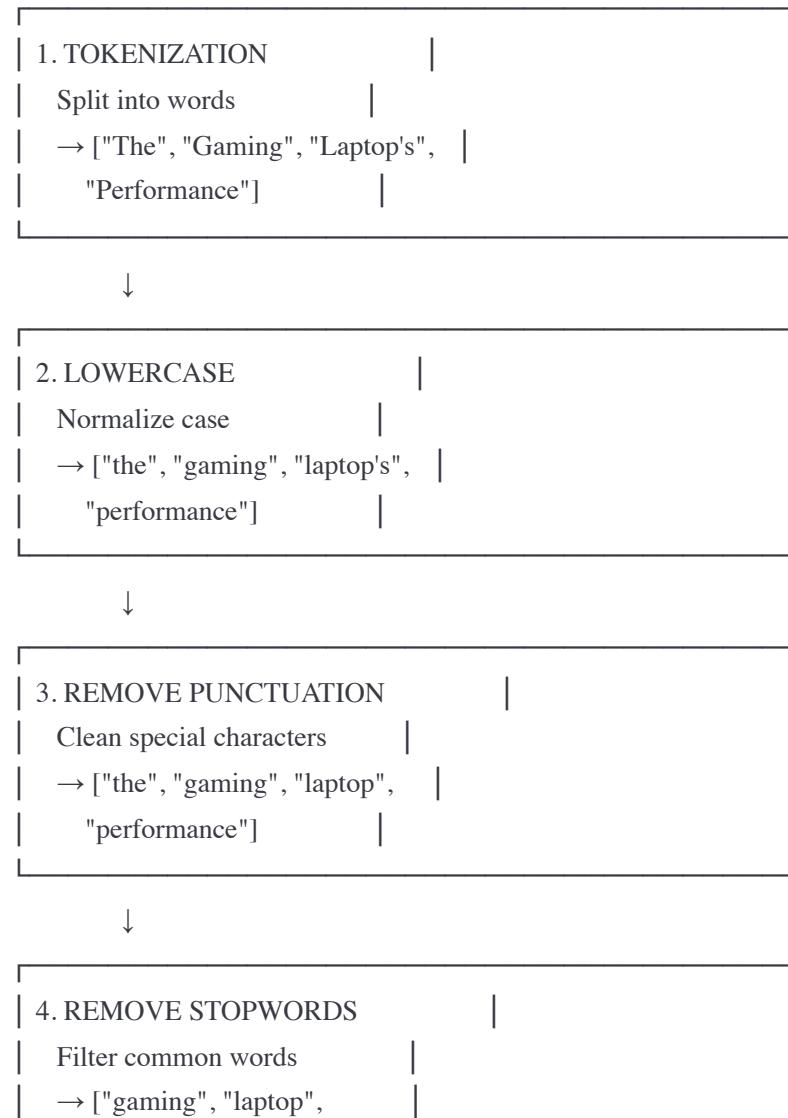
```

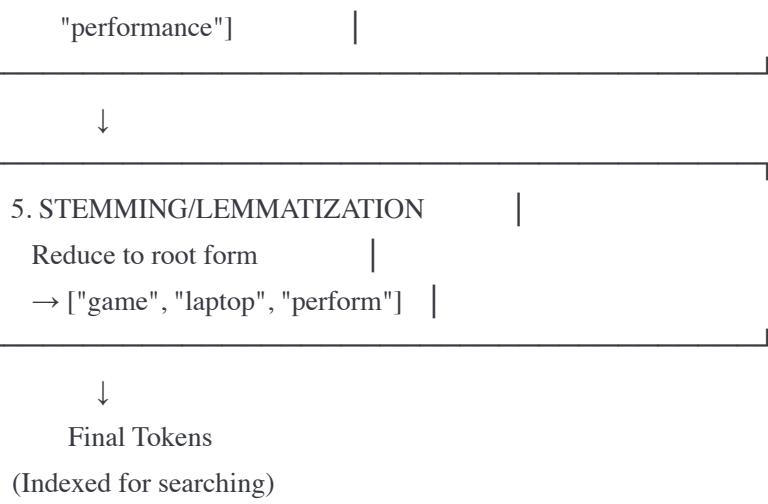
# Indexed document doc2: 4 tokens
# Indexed document doc3: 4 tokens
# Indexed document doc4: 7 tokens
#
# === Search: 'brown' ===
# Doc doc1: Score 1
# Text: The quick brown fox jumps over the lazy dog
# Doc doc2: Score 1
# Text: Quick brown dogs are friendly
#
# === Phrase Search: 'quick brown' ===
# Doc doc1: Score 10
# Text: The quick brown fox jumps over the lazy dog
# Doc doc2: Score 10
# Text: Quick brown dogs are friendly

```

Text Analysis Pipeline

Input: "The Gaming Laptop's Performance"





Implementation:

python

```
import re
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords

class TextAnalyzer:
    def __init__(self):
        self.stemmer = PorterStemmer()
        self.stopwords = set(stopwords.words('english'))

    def analyze(self, text):
        """Complete text analysis pipeline"""
        # 1. Lowercase
        text = text.lower()

        # 2. Tokenization
        tokens = re.findall(r'\b\w+\b', text)

        # 3. Remove stopwords
        tokens = [t for t in tokens if t not in self.stopwords]

        # 4. Stemming
        tokens = [self.stemmer.stem(t) for t in tokens]

    return tokens

def analyze_with_positions(self, text):
    """Analysis preserving positions"""
    text = text.lower()
    tokens = re.findall(r'\b\w+\b', text)

    analyzed = []
    position = 0

    for token in tokens:
        if token not in self.stopwords:
            stemmed = self.stemmer.stem(token)
            analyzed.append({
                'token': stemmed,
                'original': token,
                'position': position
            })
            position += 1

    return analyzed

# Example
```

```
analyzer = TextAnalyzer()

text = "The Gaming Laptop's Performance is amazing"
tokens = analyzer.analyze(text)
print("Analyzed:", tokens)
# Output: ['game', 'laptop', 'perform', 'amaz']

# With positions
detailed = analyzer.analyze_with_positions(text)
for item in detailed:
    print(f"Position {item['position']}: {item['original']} → {item['token']}")

# Output:
# Position 1: gaming → game
# Position 2: laptop → laptop
# Position 3: performance → perform
# Position 5: amazing → amaz
```

3. Ranking and Relevance

TF-IDF (Term Frequency - Inverse Document Frequency)

Most common relevance algorithm.

$$\text{TF-IDF} = \text{TF} \times \text{IDF}$$

TF (Term Frequency):

How often does term appear in document?

More occurrences = more relevant

$$\text{TF} = (\text{Number of times term appears in doc}) / (\text{Total terms in doc})$$

IDF (Inverse Document Frequency):

How rare is the term across all documents?

Rare terms are more important

$$\text{IDF} = \log(\text{Total documents} / \text{Documents containing term})$$

Example:

Documents:

Doc 1: "laptop laptop computer" (3 words)

Doc 2: "laptop phone" (2 words)

Doc 3: "phone tablet" (2 words)

Query: "laptop"

TF for "laptop":

- Doc 1: $2/3 = 0.67$ (appears twice)
- Doc 2: $1/2 = 0.50$ (appears once)
- Doc 3: $0/2 = 0$ (doesn't appear)

IDF for "laptop":

$$\text{IDF} = \log(3 / 2) = \log(1.5) = 0.176$$

(appears in 2 out of 3 documents)

TF-IDF Scores:

- Doc 1: $0.67 \times 0.176 = 0.118$
- Doc 2: $0.50 \times 0.176 = 0.088$
- Doc 3: $0 \times 0.176 = 0$

Ranking: Doc 1, Doc 2, Doc 3

TF-IDF Implementation

python

```

import math
from collections import Counter

class TFIDF:
    def __init__(self):
        self.documents = {}
        self.df = {} # Document frequency: term -> count
        self.num_docs = 0

    def add_document(self, doc_id, text):
        """Add document and update statistics"""
        tokens = self.tokenize(text)
        self.documents[doc_id] = tokens
        self.num_docs += 1

        # Update document frequency
        unique_tokens = set(tokens)
        for token in unique_tokens:
            self.df[token] = self.df.get(token, 0) + 1

    def tokenize(self, text):
        """Simple tokenization"""
        return text.lower().split()

    def tf(self, term, doc_tokens):
        """Calculate term frequency"""
        if not doc_tokens:
            return 0
        return doc_tokens.count(term) / len(doc_tokens)

    def idf(self, term):
        """Calculate inverse document frequency"""
        if term not in self.df:
            return 0

        #  $IDF = \log(N / df)$ 
        return math.log(self.num_docs / self.df[term])

    def tfidf(self, term, doc_tokens):
        """Calculate TF-IDF score"""
        return self.tf(term, doc_tokens) * self.idf(term)

    def search(self, query):
        """Search and rank by TF-IDF"""
        query_terms = self.tokenize(query)
        scores = {}

        for term in query_terms:
            if term in self.df:
                scores[term] = self.tfidf(term, self.documents[term])

```

```

for doc_id, doc_tokens in self.documents.items():
    score = 0

    for term in query_terms:
        score += self.tfidf(term, doc_tokens)

    if score > 0:
        scores[doc_id] = score

# Sort by score
ranked = sorted(scores.items(), key=lambda x: x[1], reverse=True)

return [
{
    'doc_id': doc_id,
    'score': score,
    'text': ''.join(self.documents[doc_id])
}
for doc_id, score in ranked
]

def explain_score(self, query, doc_id):
    """Explain why document ranked as it did"""
    query_terms = self.tokenize(query)
    doc_tokens = self.documents[doc_id]

    print(f"\nScore breakdown for '{query}' in {doc_id}:")
    print(f"Document: {' '.join(doc_tokens)}\n")

    total_score = 0

    for term in query_terms:
        tf = self.tf(term, doc_tokens)
        idf = self.idf(term)
        tfidf = tf * idf
        total_score += tfidf

        print(f"Term: '{term}'")
        print(f" TF: {tf:.4f} (appears {doc_tokens.count(term)} times in {len(doc_tokens)} words)")
        print(f" IDF: {idf:.4f} (appears in {self.df.get(term, 0)}/{self.num_docs} documents)")
        print(f" TF-IDF: {tfidf:.4f}")

    print(f"\nTotal Score: {total_score:.4f}")

# Example Usage
tfidf = TFIDF()

```

```

# Add documents
tfidf.add_document('doc1', 'laptop laptop computer gaming')
tfidf.add_document('doc2', 'laptop phone mobile')
tfidf.add_document('doc3', 'phone tablet mobile device')
tfidf.add_document('doc4', 'gaming laptop laptop laptop powerful')

# Search
print("==== Search Results for 'laptop gaming' ====")
results = tfidf.search('laptop gaming')

for i, result in enumerate(results, 1):
    print(f" {i}. {result['doc_id']}: {result['score']:.4f}")
    print(f"  {result['text']}")

# Explain score
tfidf.explain_score('laptop gaming', 'doc4')

# Output:
# === Search Results for 'laptop gaming' ===
# 1. doc4: 1.2877 (3x laptop + gaming)
#   gaming laptop laptop laptop powerful
# 2. doc1: 0.6027 (2x laptop + gaming)
#   laptop laptop computer gaming
# 3. doc2: 0.1438 (1x laptop)
#   laptop phone mobile

```

BM25 (Better than TF-IDF)

Modern ranking algorithm used by Elasticsearch.

BM25 improves TF-IDF:

1. Diminishing returns for term frequency

TF-IDF: 10 occurrences = 10x better than 1

BM25: 10 occurrences \approx 2x better than 1 (saturates)

2. Document length normalization

Short documents with term = more relevant

Long documents with term = less relevant

3. Tunable parameters

k1: Controls TF saturation (default: 1.2)

b: Controls length normalization (default: 0.75)

Formula:

$$\text{BM25}(\text{term}, \text{doc}) = \text{IDF}(\text{term}) \times \frac{(\text{TF} \times (\text{k1} + 1))}{(\text{TF} + \text{k1} \times (1 - \text{b} + \text{b} \times (\text{docLength} / \text{avgDocLength})))}$$

Implementation:

python

```

class BM25:
    def __init__(self, k1=1.2, b=0.75):
        self.k1 = k1
        self.b = b
        self.documents = {}
        self.df = {}
        self.num_docs = 0
        self.avg_doc_length = 0

    def add_document(self, doc_id, text):
        tokens = text.lower().split()
        self.documents[doc_id] = tokens
        self.num_docs += 1

        # Update document frequency
        for token in set(tokens):
            self.df[token] = self.df.get(token, 0) + 1

        # Update average document length
        total_length = sum(len(doc) for doc in self.documents.values())
        self.avg_doc_length = total_length / self.num_docs

    def idf(self, term):
        """Calculate IDF"""
        df = self.df.get(term, 0)
        if df == 0:
            return 0

        # BM25 IDF formula
        return math.log((self.num_docs - df + 0.5) / (df + 0.5) + 1)

    def bm25(self, term, doc_tokens):
        """Calculate BM25 score for term in document"""
        if term not in doc_tokens:
            return 0

        # Term frequency in document
        tf = doc_tokens.count(term)

        # Document length
        doc_length = len(doc_tokens)

        # IDF
        idf = self.idf(term)

        # BM25 formula

```

```

numerator = tf * (self.k1 + 1)
denominator = tf + self.k1 * (1 - self.b + self.b * (doc_length / self.avg_doc_length))

return idf * (numerator / denominator)

def search(self, query):
    """Search with BM25 ranking"""
    query_terms = query.lower().split()
    scores = {}

    for doc_id, doc_tokens in self.documents.items():
        score = 0

        for term in query_terms:
            score += self.bm25(term, doc_tokens)

        if score > 0:
            scores[doc_id] = score

    # Sort by score
    ranked = sorted(scores.items(), key=lambda x: x[1], reverse=True)

    return ranked

# Example
bm25 = BM25()

bm25.add_document('doc1', 'laptop laptop laptop') # Repeated term
bm25.add_document('doc2', 'laptop computer')
bm25.add_document('doc3', 'phone tablet')
bm25.add_document('doc4', 'laptop gaming powerful fast laptop')

results = bm25.search('laptop')

print("BM25 Rankings:")
for doc_id, score in results:
    print(f'{doc_id}: {score:.4f}')

# Output:
# doc1: 0.8754 (3 occurrences, short doc)
# doc4: 0.7234 (2 occurrences, longer doc)
# doc2: 0.4532 (1 occurrence)

# Note: doc1 and doc4 both have "laptop" multiple times
# but doc1 ranks higher (shorter = more relevant)

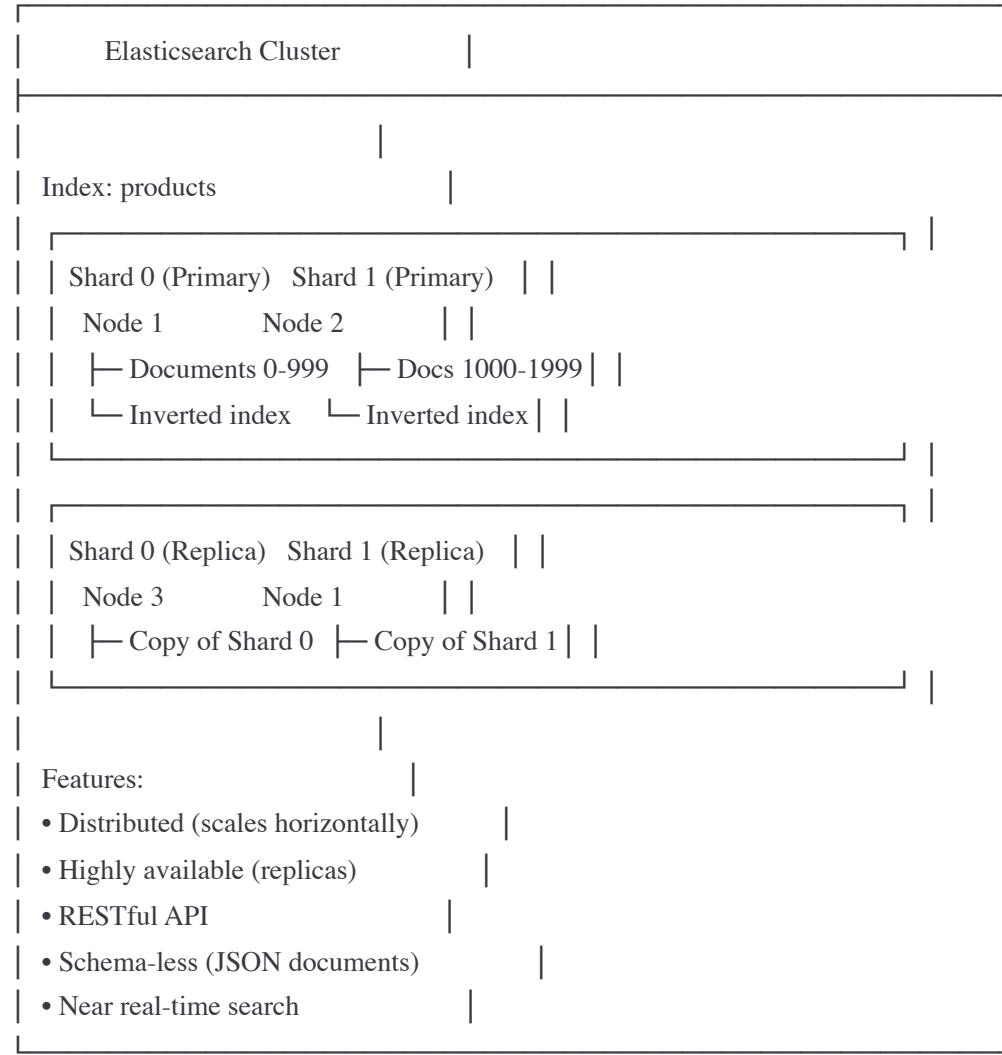
```

4. Elasticsearch

What is Elasticsearch?

Distributed search and analytics engine built on Apache Lucene.

Elasticsearch Architecture:



Elasticsearch Basics

Create Index:

bash

```
# Create index with settings
PUT /products
{
  "settings": {
    "number_of_shards": 2,
    "number_of_replicas": 1,
    "analysis": {
      "analyzer": {
        "custom_analyzer": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": ["lowercase", "stop", "snowball"]
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "name": {
        "type": "text",
        "analyzer": "custom_analyzer"
      },
      "description": {
        "type": "text",
        "analyzer": "custom_analyzer"
      },
      "price": {
        "type": "float"
      },
      "category": {
        "type": "keyword"
      },
      "rating": {
        "type": "float"
      },
      "in_stock": {
        "type": "boolean"
      }
    }
  }
}
```

Index Documents:

```
javascript
```

```
const { Client } = require('@elastic/elasticsearch');
const client = new Client({ node: 'http://localhost:9200' });

// Index a single document
await client.index({
  index: 'products',
  id: '1',
  document: {
    name: 'Dell Gaming Laptop',
    description: 'High-performance gaming laptop with 16GB RAM',
    price: 1299.99,
    category: 'Electronics',
    rating: 4.5,
    in_stock: true
  }
});

// Bulk indexing (efficient for large datasets)
const products = [
  { id: '2', name: 'Apple MacBook Pro', price: 1999, category: 'Electronics' },
  { id: '3', name: 'Logitech Mouse', price: 29.99, category: 'Accessories' },
  { id: '4', name: 'Mechanical Keyboard', price: 89.99, category: 'Accessories' },
  // ... millions more
];

const body = products.flatMap(product => [
  { index: { _index: 'products', _id: product.id } },
  product
]);

await client.bulk({ body });

console.log('Indexed products');
```

Search Queries:

```
javascript
```

```
// 1. Simple match query
const result1 = await client.search({
  index: 'products',
  query: {
    match: {
      name: 'gaming laptop'
    }
  }
});
```

```
// 2. Multi-field search
const result2 = await client.search({
  index: 'products',
  query: {
    multi_match: {
      query: 'gaming laptop',
      fields: ['name^2', 'description'], // Boost name field 2x
      type: 'best_fields'
    }
  }
});
```

```
// 3. Boolean query (combine conditions)
const result3 = await client.search({
  index: 'products',
  query: {
    bool: {
      must: [
        { match: { name: 'laptop' } }
      ],
      filter: [
        { range: { price: { gte: 500, lte: 1500 } } },
        { term: { category: 'Electronics' } },
        { term: { in_stock: true } }
      ],
      should: [
        { match: { description: 'gaming' } } // Boosts score if matches
      ],
      must_not: [
        { match: { name: 'refurbished' } }
      ]
    }
  }
});
```

```
// 4. Fuzzy search (typo tolerance)
```

```
const result4 = await client.search({
  index: 'products',
  query: {
    match: {
      name: {
        query: 'laptop', // Typo!
        fuzziness: 'AUTO' // Allows 1-2 character edits
      }
    }
  }
});
```

// 5. Phrase search

```
const result5 = await client.search({
  index: 'products',
  query: {
    match_phrase: {
      description: 'high performance gaming'
    }
  }
});
```

// 6. Aggregations (faceted search)

```
const result6 = await client.search({
  index: 'products',
  size: 0, // Don't return documents, just aggregations
  query: {
    match: { name: 'laptop' }
  },
  aggs: {
    by_category: {
      terms: { field: 'category' }
    },
    price_ranges: {
      range: {
        field: 'price',
        ranges: [
          { to: 500 },
          { from: 500, to: 1000 },
          { from: 1000, to: 2000 },
          { from: 2000 }
        ]
      }
    },
    avg_price: {
      avg: { field: 'price' }
    }
  }
});
```

```
avg_rating: {  
    avg: { field: 'rating' }  
}  
}  
});  
  
// Result includes facets:  
// {  
//   "aggregations": {  
//     "by_category": {  
//       "buckets": [  
//         { "key": "Electronics", "doc_count": 45 },  
//         { "key": "Computers", "doc_count": 23 }  
//       ]  
//     },  
//     "price_ranges": {  
//       "buckets": [  
//         { "to": 500, "doc_count": 12 },  
//         { "from": 500, "to": 1000, "doc_count": 28 },  
//         { "from": 1000, "to": 2000, "doc_count": 15 }  
//       ]  
//     }  
//   }  
// }
```

Elasticsearch Advanced Features

Autocomplete (Search-as-you-type):

```
bash
```

```
# Create index with autocomplete field
```

```
PUT /products
```

```
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "name_suggest": {
        "type": "search_as_you_type"
      }
    }
  }
}
```

```
# Index document
```

```
POST /products/_doc/1
```

```
{
  "name": "Dell Gaming Laptop",
  "name_suggest": "Dell Gaming Laptop"
}
```

```
# Autocomplete query (user types "gam")
```

```
GET /products/_search
```

```
{
  "query": {
    "multi_match": {
      "query": "gam",
      "type": "bool_prefix",
      "fields": [
        "name_suggest",
        "name_suggest._2gram",
        "name_suggest._3gram"
      ]
    }
  }
}
```

```
# Returns: "Dell Gaming Laptop" (matches "gam" prefix)
```

Highlighting:

```
javascript
```

```
// Highlight matching terms in results
const result = await client.search({
  index: 'products',
  query: {
    match: { description: 'gaming laptop' }
  },
  highlight: {
    fields: {
      description: {
        pre_tags: ['<strong>'],
        post_tags: ['</strong>']
      }
    }
  }
});

// Result:
//{
//  "_source": {
//    "description": "High performance gaming laptop"
//  },
//  "highlight": {
//    "description": [
//      "High performance <strong>gaming</strong> <strong>laptop</strong>"
//    ]
//  }
//}
```

Synonyms:

```
bash
```

```

# Create index with synonyms
PUT /products
{
  "settings": {
    "analysis": {
      "filter": {
        "synonym_filter": {
          "type": "synonym",
          "synonyms": [
            "laptop, notebook, computer",
            "phone, mobile, cell",
            "fast, quick, speedy"
          ]
        }
      },
      "analyzer": {
        "synonym_analyzer": {
          "tokenizer": "standard",
          "filter": ["lowercase", "synonym_filter"]
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "name": {
        "type": "text",
        "analyzer": "synonym_analyzer"
      }
    }
  }
}

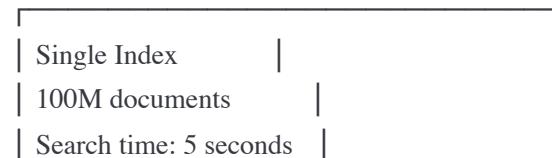
```

Search for "notebook" will also match "laptop"!

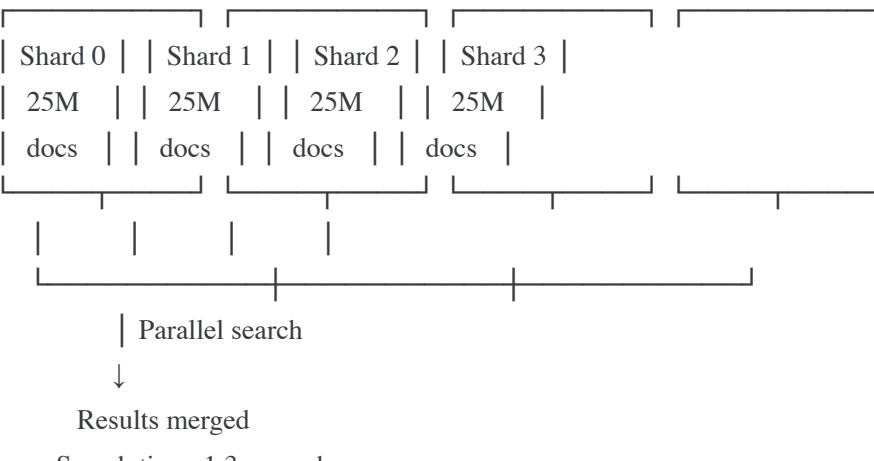
5. Search Optimization Techniques

Technique 1: Sharding (Horizontal Scaling)

Without Sharding:



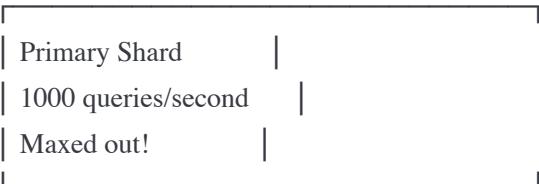
With Sharding (4 shards):



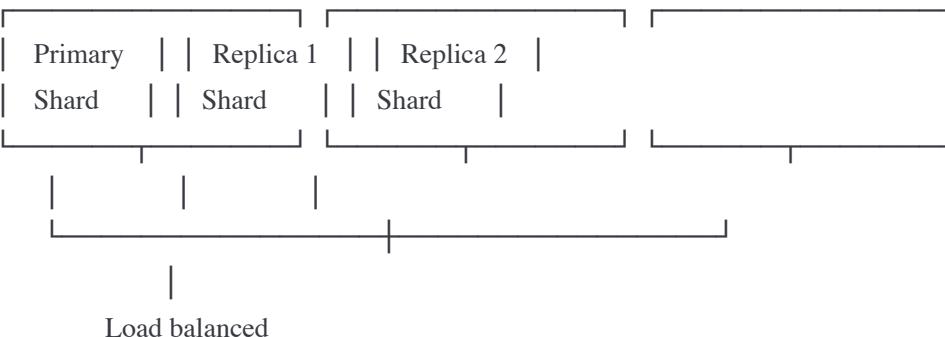
4x faster with parallel search!

Technique 2: Replication (Performance + Availability)

Without Replication:



With Replication (2 replicas):



Each handles 333 queries/second

Total: 1000 queries/second

Plus: Fault tolerance!

Technique 3: Query Optimization

Filter vs Query:

```
javascript
```

```
// ✗ SLOW: Scoring every document
const slowSearch = await client.search({
  index: 'products',
  query: {
    bool: {
      must: [
        { match: { name: 'laptop' } },
        { range: { price: { gte: 500, lte: 1500 } } }, // Scored!
        { term: { in_stock: true } } // Scored!
      ]
    }
  }
});
```

```
// ✓ FAST: Filter doesn't score (just yes/no)
```

```
const fastSearch = await client.search({
  index: 'products',
  query: {
    bool: {
      must: [
        { match: { name: 'laptop' } } // Only this is scored
      ],
      filter: [ // These are cached and fast!
        { range: { price: { gte: 500, lte: 1500 } } },
        { term: { in_stock: true } },
        { term: { category: 'Electronics' } }
      ]
    }
  }
});
```

```
// Fast search is 3-5x faster!
```

```
// Filter results are cached
```

Technique 4: Index Optimization

Refresh Interval:

```
javascript
```

```
// Default: Refresh every 1 second (make changes visible)
```

```
// For bulk indexing: Disable refresh
```

```
// Before bulk import
```

```
await client.indices.putSettings({
```

```
  index: 'products',
```

```
  settings: {
```

```
    refresh_interval: '-1' // Disable
```

```
}
```

```
});
```

```
// Bulk index millions of documents
```

```
await bulkIndexDocuments();
```

```
// Re-enable refresh
```

```
await client.indices.putSettings({
```

```
  index: 'products',
```

```
  settings: {
```

```
    refresh_interval: '1s'
```

```
}
```

```
});
```

```
// Force refresh
```

```
await client.indices.refresh({ index: 'products' });
```

```
// Improvement: 10x faster bulk indexing!
```

Technique 5: Caching

Elasticsearch Caching Layers:

1. QUERY CACHE

Cache: Entire query result

When: Exact same query repeated

2. FILTER CACHE

Cache: Filter results (which docs match)

When: Same filter used in multiple queries

3. FIELD DATA CACHE

Cache: Field values for sorting/aggregation

When: Sorting or aggregating on field

4. REQUEST CACHE (Shard level)

Cache: Results from each shard

When: Query hits same shards

Configuration:

bash

```
# Configure caching
PUT /products/_settings
{
  "index": {
    "queries": {
      "cache": {
        "enabled": true
      }
    },
    "max_result_window": 10000 # Limit deep pagination
  }
}
```

```
# Use filters for caching
```

```
GET /products/_search
{
  "query": {
    "bool": {
      "must": { "match": { "name": "laptop" } },
      "filter": [
        { "term": { "category": "Electronics" } }, # Cached!
        { "range": { "price": { "gte": 500 } } } # Cached!
      ]
    }
  }
}
```

```
# Subsequent searches with same filters are faster!
```

Technique 6: Denormalization

✖ Normalized (requires joins - not efficient in search):

Products Index:

```
{
  "product_id": 1,
  "name": "Laptop",
  "category_id": 5
```

```
}
```

Categories Index:

```
{
  "category_id": 5,
  "name": "Electronics"
}
```

Need to search products by category name → Must join!

✓ Denormalized (embedded for fast search):

Products Index:

```
{
  "product_id": 1,
  "name": "Laptop",
  "category": {
    "id": 5,
    "name": "Electronics" # Embedded!
  }
}
```

Can search directly! No joins needed!

Complete Search System Example

javascript

```
class SearchService {
  constructor() {
    this.client = new Client({ node: 'http://localhost:9200' });
    this.indexName = 'products';
  }

  async initialize() {
    // Create index with optimized settings
    await this.client.indices.create({
      index: this.indexName,
      body: {
        settings: {
          number_of_shards: 3,
          number_of_replicas: 2,
          analysis: {
            analyzer: {
              product_analyzer: {
                type: 'custom',
                tokenizer: 'standard',
                filter: [
                  'lowercase',
                  'stop',
                  'snowball',
                  'synonym_filter'
                ]
              }
            },
            filter: {
              synonym_filter: {
                type: 'synonym',
                synonyms: [
                  'laptop, notebook, computer',
                  'phone, mobile, smartphone'
                ]
              }
            }
          }
        },
        mappings: {
          properties: {
            name: {
              type: 'text',
              analyzer: 'product_analyzer',
              fields: {
                keyword: { type: 'keyword' } // For exact match
              }
            }
          }
        }
      }
    });
  }
}
```

```
        },
        description: {
          type: 'text',
          analyzer: 'product_analyzer'
        },
        category: { type: 'keyword' },
        price: { type: 'float' },
        rating: { type: 'float' },
        review_count: { type: 'integer' },
        in_stock: { type: 'boolean' },
        tags: { type: 'keyword' }
      }
    }
  }
});
```

```
async search(query, options = {}) {
  const {
    category = null,
    minPrice = null,
    maxPrice = null,
    minRating = null,
    inStock = null,
    page = 1,
    pageSize = 20,
    sortBy = '_score'
  } = options;
```

```
// Build Elasticsearch query
const esQuery = {
  bool: {
    must: [],
    filter: [],
    should: []
  }
};
```

```
// Main search query
if (query) {
  esQuery.bool.must.push({
    multi_match: {
      query: query,
      fields: ['name^3', 'description', 'tags^2'],
      type: 'best_fields',
      fuzziness: 'AUTO'
    }
  })
}
```

```
    });

}

// Filters (fast, cached)
if (category) {
  esQuery.bool.filter.push({ term: { category } });
}

if (minPrice !== null || maxPrice !== null) {
  const range = {};
  if (minPrice !== null) range.gte = minPrice;
  if (maxPrice !== null) range.lte = maxPrice;
  esQuery.bool.filter.push({ range: { price: range } });
}

if (minRating !== null) {
  esQuery.bool.filter.push({
    range: { rating: { gte: minRating } }
  });
}

if (inStock !== null) {
  esQuery.bool.filter.push({ term: { in_stock: inStock } });
}

// Execute search
const result = await this.client.search({
  index: this.indexName,
  from: (page - 1) * pageSize,
  size: pageSize,
  query: esQuery,
  sort: [
    sortBy === 'price_asc' ? { price: 'asc' } :
    sortBy === 'price_desc' ? { price: 'desc' } :
    sortBy === 'rating' ? { rating: 'desc' } :
    '_score'
  ],
  highlight: {
    fields: {
      name: {},
      description: {}
    }
  },
  aggs: {
    categories: {
      terms: { field: 'category', size: 10 }
    }
  }
});
```

```
    price_ranges: {
      range: {
        field: 'price',
        ranges: [
          { to: 100, key: 'Under $100' },
          { from: 100, to: 500, key: '$100-$500' },
          { from: 500, to: 1000, key: '$500-$1000' },
          { from: 1000, key: 'Over $1000' }
        ]
      }
    },
    avg_rating: {
      avg: { field: 'rating' }
    }
  });
}

return {
  total: result.hits.total.value,
  results: result.hits.hits.map(hit => ({
    id: hit._id,
    score: hit._score,
    ...hit._source,
    highlights: hit.highlight
  })),
  facets: result.aggregations,
  page: page,
  pageSize: pageSize,
  totalPages: Math.ceil(result.hits.total.value / pageSize)
};

}

async autocomplete(prefix) {
  """Autocomplete suggestions"""
  const result = await this.client.search({
    index: this.indexName,
    size: 5,
    query: {
      match_phrase_prefix: {
        name: {
          query: prefix,
          max_expansions: 10
        }
      }
    },
    _source: ['name']
  });
}
```

```
return result.hits.hits.map(hit => hit._source.name);
}

async moreLikeThis(productId) {
    """Find similar products"""
    const result = await this.client.search({
        index: this.indexName,
        query: {
            more_like_this: {
                fields: ['name', 'description', 'tags'],
                like: [
                    {
                        _index: this.indexName,
                        _id: productId
                    }
                ],
                min_term_freq: 1,
                max_query_terms: 12
            }
        },
        size: 10
    });

    return result.hits.hits.map(hit => ({
        id: hit._id,
        score: hit._score,
        ...hit._source
    }));
}

// Usage
const searchService = new SearchService();
await searchService.initialize();

// Search with filters
const results = await searchService.search('gaming laptop', {
    category: 'Electronics',
    minPrice: 500,
    maxPrice: 2000,
    minRating: 4.0,
    inStock: true,
    page: 1,
    pageSize: 20,
    sortBy: '_score'
});
```

```

console.log(`Found ${results.total} products`);
console.log(`Showing page ${results.page} of ${results.totalPages}`);
console.log(`\nTop results:`);
results.results.slice(0, 5).forEach((product, i) => {
  console.log(`${i+1}. ${product.name} - ${product.price} (score: ${product.score.toFixed(2)})`);
});

console.log(`\nFacets:`);
console.log('Categories:', results.facets.categories.buckets);
console.log('Price ranges:', results.facets.price_ranges.buckets);

// Autocomplete
const suggestions = await searchService.autocomplete('gam');
console.log('Autocomplete suggestions:', suggestions);
// → ["Gaming Laptop", "Gaming Mouse", "Gaming Chair"]

// Similar products
const similar = await searchService.moreLikeThis('product-123');
console.log('Similar products:', similar.map(p => p.name));

```

Technique 7: Boosting

Boost important fields or conditions:

javascript

```
const result = await client.search({
  index: 'products',
  query: {
    bool: {
      should: [
        {
          match: {
            name: {
              query: 'laptop',
              boost: 3.0 // Name match worth 3x more
            }
          }
        },
        {
          match: {
            description: {
              query: 'laptop',
              boost: 1.0 // Description match normal weight
            }
          }
        },
        {
          term: {
            tags: {
              value: 'featured',
              boost: 2.0 // Featured products boosted
            }
          }
        }
      ]
    }
  },
  // Function score for custom boosting
  functions: [
    {
      // Boost popular products
      field_value_factor: {
        field: 'review_count',
        factor: 0.1,
        modifier: 'log1p'
      }
    },
    {
      // Boost highly rated products
      filter: { range: { rating: { gte: 4.5 } } },
      weight: 1.5
    }
  ]
})
```

```
    }  
],  
  score_mode: 'sum',  
  boost_mode: 'multiply'  
});
```

Key Takeaways

1. Full-Text Search:

- Beyond simple LIKE queries
- Text analysis (tokenization, stemming)
- Fuzzy matching, synonyms
- Relevance ranking

2. Inverted Index:

- Term → Document mapping
- O(1) lookup vs O(n) scan
- With positions for phrase matching
- Foundation of search engines

3. Ranking:

- TF-IDF: Classic algorithm
- BM25: Modern, better
- Custom boosting
- Relevance scoring

4. Elasticsearch:

- Distributed search engine
- RESTful API
- Near real-time
- Rich query DSL
- Aggregations (facets)

5. Optimization:

- Sharding for parallelism
- Replication for performance

- Filters over queries
- Caching
- Denormalization

Practice Problems

1. Design a search system for Amazon (100M products). How would you shard? What ranking factors?
2. Implement autocomplete for Google-like search. What data structures?
3. Calculate: 10M documents, 100 queries/sec. How many shards and replicas?

Next Steps

You've completed 18 chapters covering most system design fundamentals!

Would you like to:

1. Continue with more chapters?
2. Practice complete system design problems?
3. Deep dive into specific topics?

What interests you most?