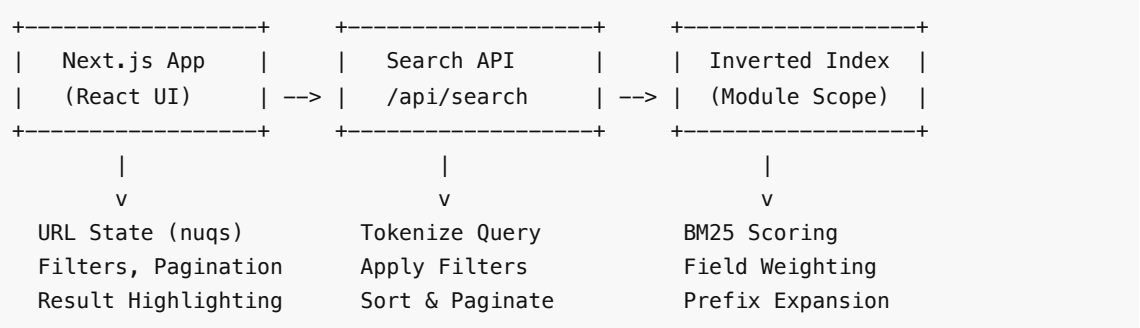
IMAGO Media Search - Architecture Documentation

A full-text search system for media metadata with BM25 relevance scoring, preprocessing for dirty German data, and a scalable architecture designed for the IMAGO coding challenge.

Live Demo: https://imago-coding-challenge-i3f9.vercel.app/

Architecture Overview

The system follows a simple but effective architecture: an in-memory inverted index built at server startup, queried through a Next.js API route, with results rendered in a React frontend.



Key Design Decisions

In-memory index: With 10k items, the entire index fits comfortably in memory (~50MB). This eliminates database round-trips and enables sub-millisecond query times.

Module-scope singleton: The index is initialized once per serverless instance and persists across requests. On Vercel, this means cold starts (~500ms) are rare after initial traffic.

```
// src/lib/search/index.ts
let index: InvertedIndex | null = null;

export const getIndex = (): InvertedIndex => {
  if (!index) {
    index = new InvertedIndex();
    const items = loadMediaItems();
    items.forEach((item, i) => index.addDocument(i, preprocessItem(item)));
    index.finalizeIndex();
  }
  return index;
}
```

Separate field indices: Each searchable field (suchtext, fotografen, bildnummer) has its own inverted index. This enables independent BM25 scoring and field-specific weighting.

Directory Structure

```
src/
  app/
    api/
```

```
      search/route.ts    # Search endpoint with filtering/pagination
      filters/route.ts   # Filter options (photographers, restrictions)
    page.tsx             # Main search UI
  lib/
    search/
      inverted-index.ts  # Core index implementation
      bm25.ts            # BM25 scoring functions
      tokenizer.ts       # Text tokenization
      searcher.ts        # Query orchestration
    data/
      preprocess.ts      # Data cleaning (dates, umlauts, restrictions)
    api/
      highlight.ts       # Keyword highlighting
```

Search Algorithm (BM25)

BM25 (Best Matching 25) is the industry-standard ranking function used by Elasticsearch, Lucene, and most modern search systems. It improves upon TF-IDF by adding term frequency saturation and document length normalization.

The Formula

```
score = IDF * (tf * (k1 + 1)) / (tf + k1 * (1 - b + b * (docLen / avgDocLen)))
```

Where:

- **IDF** (Inverse Document Frequency): How rare the term is across documents
- **tf**: Term frequency in this document
- **k1=1.2**: Controls term frequency saturation (standard value)
- **b=0.75**: Controls document length normalization

Lucene IDF Variant

We use the Lucene/Elasticsearch variant of IDF which is always non-negative:

```
// src/lib/search/bm25.ts
export const computeIDF = (docFreq: number, totalDocs: number): number => {
  if (totalDocs === 0 || docFreq === 0) return 0;
  return Math.log(1 + (totalDocs - docFreq + 0.5) / (docFreq + 0.5));
}
```

This avoids negative IDF values for common terms (appearing in >50% of documents) which can cause counter-intuitive ranking.

Field Weighting Strategy

Different fields have different importance for search relevance:

Field	Weight	Rationale
suchtext	3.0	Primary content - descriptions, keywords

fotografen	1.5	Photographer name matches are relevant but secondary
bildnummer	1.0	Optional ID lookups, lower priority than content

The weights multiply the BM25 score for each field, so a match in suchtext contributes 3x more than the same match in fotografen.

Prefix Matching with Penalty

For better UX, partial word matches (prefix matching) are supported. Typing "ber" will match "berlin", "beruf", etc.

```
// Binary search for O(log n) prefix lookup
getPrefixTerms = (prefix: string, field: string, limit: number = 50): string[] => {
  const terms = this.getFieldTermsSorted(field);

  // Binary search for first term >= prefix
  let lo = 0, hi = terms.length;
  while (lo < hi) {
    const mid = (lo + hi) >>> 1;
    if (terms[mid] < prefix) lo = mid + 1;
    else hi = mid;
  }

  // Collect matching terms
  const matches: string[] = [];
  for (let i = lo; i < terms.length && matches.length < limit; i++) {
    if (terms[i].startsWith(prefix)) matches.push(terms[i]);
    else break;
  }
  return matches;
}
```

Prefix matches receive a **0.8 penalty** (configurable) to rank exact matches higher:

```
const DEFAULT_SEARCH_CONFIG = {
  prefixPenalty: 0.8,  // 20% score reduction for prefix matches
  minPrefixLength: 3,   // Avoid explosion from short prefixes
  maxPrefixExpansion: 50 // Limit terms per prefix
};
```

Multi-Word Query Behavior

Queries with multiple words use **OR logic** with BM25 relevance ranking:

- `"berlin portrait"` finds documents containing "berlin" OR "portrait"
- Documents matching BOTH terms rank higher (scores accumulate)
- BM25 naturally handles this: multi-term matches get additive scores
- Tie-breaker for equal scores: newest date first

```
Query: "berlin portrait"

Results (highest score first):
1. "Berlin portrait photography..." (matches both, highest score)
2. "Portrait session in Berlin..."  (matches both)
3. "Berlin cityscape at dusk..."    (matches "berlin" only)
4. "Portrait studio lighting..."    (matches "portrait" only)
```

This approach maximizes recall (finds anything relevant) while BM25 ensures precision (best matches rank first).

IDF Caching

IDF values only depend on corpus statistics, not the query. We cache them after first computation:

```
const idfCache = new Map<string, Map<string, number>>();

const getIDF = (term: string, field: string, fieldIndex: FieldIndex): number => {
  let fieldCache = idfCache.get(field);
  if (!fieldCache) {
    fieldCache = new Map();
    idfCache.set(field, fieldCache);
  }

  const cached = fieldCache.get(term);
  if (cached !== undefined) return cached;

  const postings = fieldIndex.termPostings.get(term);
  const idf = computeIDF(postings?.length || 0, fieldIndex.totalDocs);
  fieldCache.set(term, idf);
  return idf;
}
```

Preprocessing Strategy

Raw media metadata is messy: European date formats, German umlauts, and restriction tokens embedded in text. Preprocessing normalizes this for consistent indexing and search.

Data Flow

```
Raw Item                    Processed Item
+-------------------+       +------------------------+
| suchtext: "..."   |  -->  | searchableText: "..."  |
| fotografen: "..." |       | photographerNormalized |
| datum: "DD.MM.YY" |       | dateISO: "YYYY-MM-DD"  |
+-------------------+       | restrictions: [...]    |
                            +------------------------+
```

Restriction Token Extraction

Restriction tokens (e.g., `PUBLICATIONxINxGERxONLY`) are uppercase words joined by 'x'. They must be extracted **before** tokenization, or the 'x' delimiters fragment them:

```
// src/lib/data/preprocess.ts
const RESTRICTION_PATTERN = /[A-Z]+(?:x[A-Z]+)+/g;

export const extractRestrictions = (text: string): {
  restrictions: string[];
  cleanText: string;
} {
  const restrictions = text.match(RESTRICTION_PATTERN) || [];
  const cleanText = text
    .replace(RESTRICTION_PATTERN, ' ')
    .replace(/\s+/g, ' ')
    .trim();
  return { restrictions, cleanText };
}
```

Example:

```
"Berlin PUBLICATIONxINxGERxONLY portrait"
  -> restrictions: ["PUBLICATIONxINxGERxONLY"]
  -> cleanText: "Berlin portrait"
```

Date Normalization

German dates use DD.MM.YYYY format. We normalize to ISO for consistent sorting:

```
export const parseDate = (dateStr: string): string | null => {
  // DD.MM.YYYY (European with dots)
  const dotMatch = dateStr.match(/^(\d{1,2})\.(\d{1,2})\.(\d{4})$/);
  if (dotMatch) {
    const [, day, month, year] = dotMatch;
    return `${year}-${month.padStart(2, '0')}-${day.padStart(2, '0')}`;
  }
  // Already ISO
  if (/^\d{4}-\d{2}-\d{2}$/.test(dateStr)) return dateStr;
  return null;
}
```

Using explicit regex instead of `Date()` constructor because JS date parsing of non-ISO formats is browser-dependent and unreliable.

German Text Normalization

German umlauts are normalized to ASCII equivalents for search:

```
export const normalizeGerman = (text: string): string => {
  return text
    .toLowerCase()
    .replace(/ä/g, 'ae')  // ä -> ae
```

```
    .replace(/ö/g, 'oe')  // ö -> oe
    .replace(/ü/g, 'ue')  // ü -> ue
    .replace(/ß/g, 'ss'); // ß -> ss
}
```

This allows users to search "muenchen" or "münchen" and find "München".

Tokenization

The tokenizer handles hyphenated words by indexing both the whole word and its parts:

```
// src/lib/search/tokenizer.ts
export const tokenize = (text: string): string[] => {
  const normalized = normalizeGerman(text);
  const tokens: string[] = [];

  for (const word of normalized.split(/[\s,.;:!?"'()[\]{}]+/)) {
    if (word.includes('-')) {
      const parts = word.split('-').filter(p => p.length >= 2);
      if (parts.length >= 2) {
        tokens.push(word);         // "baden-wuerttemberg"
        tokens.push(...parts);     // "baden", "wuerttemberg"
      }
    } else if (word.length >= 2) {
      tokens.push(word);
    }
  }

  return tokens.filter(t => !GERMAN_STOPWORDS.has(t));
}
```

Stopwords (~50 common German function words) are removed to reduce index size and improve relevance.

Dual-Field Approach

We preserve raw values for display while adding normalized values for search:

- `suchtext` (raw) -> `searchableText` (normalized)
- `fotografen` (raw) -> `photographerNormalized` (normalized)
- `datum` (raw) -> `dateISO` (normalized)

This ensures the UI shows "München" while search matches "muenchen".

Filter Combination Logic

Filters combine with **AND logic between categories**, **OR within restriction selections**:

- **Photographer + Date + Restrictions:** Item must match ALL three filter types
- **Multiple restrictions selected:** Item matches if it has ANY of the selected restrictions
- **'None' restriction:** Matches items with empty restrictions array

Example: Selecting photographer "IMAGO" + restrictions "PUBLICATIONxINxGER", "NOxMODELxRELEASE" returns items where:

- Photographer is "IMAGO", AND
- Item has "PUBLICATIONxINxGER" OR "NOxMODELxRELEASE"

Scaling Approach

The current architecture works well for 10k items but would need changes at millions of items.

Current Architecture Performance

At 10k items:

- Index build time: ~100-200ms (cold start)
- Query latency: <5ms (typically 1-2ms)
- Memory usage: ~50MB for index
- Complexity: O(n) index build, O(1) term lookup + O(k) scoring where k = matching docs

What Changes at Millions of Items

Concern	Current	At Scale
Memory	50MB in-process	5-50GB, can't fit in single process
Build time	200ms	20+ minutes, unacceptable for serverless
Cold starts	Acceptable	Too slow, need warm instances
Horizontal scaling	N/A	Must share index across instances

Scaling Strategy: Three Tiers

Tier 1: Thousands of items (current)

- In-memory inverted index
- Module-scope singleton
- Works well on serverless

Tier 2: Hundreds of thousands

- Persistent index in Redis
- Build index offline, load on startup
- Shared across serverless instances
- Still sub-10ms queries

```
// Conceptual Redis-backed index
class RedisBackedIndex {
  async getPostings(term: string, field: string): Promise<Posting[]> {
    const key = `idx:${field}:${term}`;
    const data = await redis.get(key);
    return data ? JSON.parse(data) : [];
  }
}
```

Tier 3: Millions to billions

- Dedicated search engine (Elasticsearch, Typesense, Meilisearch)

- Built-in sharding and replication
- Optimized on-disk data structures
- Sub-100ms at any scale

Migration Path

1. **Extract index interface** - Current code already uses `InvertedIndex` class
2. **Add Redis adapter** - Implement same interface with Redis backend
3. **Background indexing** - Move index build to async job
4. **Incremental updates** - Add/update documents without full rebuild

```
interface SearchIndex {
  getPostings(term: string, field: string): Posting[];
  getDocument(docId: number): ProcessedMediaItem;
  search(query: string): SearchResult[];
}

// Current: InMemoryIndex implements SearchIndex
// Tier 2: RedisIndex implements SearchIndex
// Tier 3: ElasticsearchIndex implements SearchIndex
```

Big-O Complexity Analysis

Current implementation:

Operation	Complexity	Notes
Index build	O(n * m)	n=docs, m=avg tokens/doc
Term lookup	O(1)	HashMap lookup
Prefix lookup	O(log v + k)	v=vocab size, k=results
BM25 scoring	O(k * t)	k=matching docs, t=query terms
Full query	O(t * (1 + k))	Dominated by scoring

At scale (with proper infrastructure):

Operation	Elasticsearch	Notes
Index build	O(n log n)	Segment merging
Term lookup	O(log n)	B-tree on disk
Query	O(k log n)	k=result size

Incremental Indexing

For continuous data ingestion, we'd need incremental indexing instead of full rebuilds:

```
// Conceptual incremental indexer
class IncrementalIndex {
  // Add single document without rebuild
```

```
  async addDocument(item: ProcessedMediaItem): Promise<void> {
    const tokens = tokenize(item.suchtext);
    for (const token of tokens) {
      await this.appendToPostings(token, item.id);
    }
    await this.updateStats();
  }

  // Background merge to optimize read performance
  async compactIndex(): Promise<void> {
    // Merge small posting lists into larger ones
  }
}
```

Testing Approach and Trade-offs

What's Tested

The search system prioritizes correctness in core algorithms:

1. **BM25 scoring** - Unit tests verify IDF computation and term scoring against known values
2. **Tokenization** - Tests for German umlauts, hyphenated words, stopword removal
3. **Preprocessing** - Date parsing, restriction extraction
4. **Integration** - API endpoints return expected results for test queries

What's Not Tested (Time Constraints)

- **Component tests** - React components don't have unit tests
- **E2E tests** - No Playwright/Cypress for full user flows
- **Performance regression** - No automated benchmarking
- **Edge cases** - Limited coverage for malformed data

Trade-offs Made

1. **Focus on search correctness over UI coverage** - The search algorithm is the core differentiator; UI is standard React patterns
2. **Manual verification over automated E2E** - Faster iteration during development
3. **Type safety as a testing substitute** - TypeScript catches many errors at compile time

What Would Be Added With More Time

1. **Property-based testing** for BM25 scoring (e.g., "score always >= 0")
2. **Snapshot tests** for search result ordering
3. **Load testing** to verify performance at scale
4. **E2E tests** for critical user flows (search, filter, pagination)

Analytics Tracking

The system tracks search queries for analytics purposes using in-memory storage.

Data Model

Each query logs:

- Query string

- Result count
- Response time (ms)
- Timestamp

```
interface QueryLog {
  query: string;
  resultCount: number;
  responseTimeMs: number;
  timestamp: Date;
}
```

Endpoint

`GET /api/analytics` returns aggregated statistics:

Metric	Description
totalSearches	Number of queries tracked
avgResultsPerQuery	Mean result count
avgResponseTimeMs	Mean response time
topSearchTerms	Top 10 search terms with counts

Implementation

Uses module-scope singleton array (same pattern as search index) with 10,000 entry cap to prevent unbounded memory growth. Query logs persist across requests but reset on serverless cold starts.

Future Work

Potential Improvements

1. **Query parsing** - Support operators like `"exact phrase"`, `-exclude`, `field:value`
2. **Synonyms** - "photo" should match "photograph", "Foto"
3. **Fuzzy matching** - Typo tolerance using Levenshtein distance
4. **Faceted search** - Count results by photographer/restriction without fetching all
5. **Analytics** - Track popular queries, zero-result queries

Technical Debt

1. **IDF cache never invalidates** - Fine for static data, but would need TTL for dynamic content
2. **No query result caching** - Could add LRU cache for repeated queries
3. **Monolithic index** - Could split by date range for time-based queries

Performance Optimizations

1. **Web Workers** - Move search off main thread in browser (for client-side search)
2. **Streaming results** - Return results as they're scored instead of all at once
3. **Query plan optimization** - Skip low-IDF terms first to reduce candidate set

---

Built for the IMAGO coding challenge. Source code at the repository root.