

Inverted Files for Text Search Engines

An inverted file maps words to the documents that contain them.

Mazen El Sedfy
CS848



Based on Zobel & Moffat, ACM Computing Surveys, 2006

About the paper

Justin Zobel and Alistair Moffat

Inverted Files for Text Search Engines
ACM Computing Surveys 38(2), Article 6, 2006

A tutorial and survey of efficient text-search implementation

It connects established indexing and retrieval techniques.

The problem the paper addresses

How can a search engine rank matches quickly across a large document collection?

Postings list: documents containing one term (indexed word).
Query: the user's search terms. Long lists increase query work.

Ranking orders matches by score; efficiency measures the work needed.

Main contribution and scope

Core explanation	Further techniques
Vocabulary and postings	Compression: fewer bits; construction: build index
Ranked query evaluation	Phrase query: ordered terms; maintenance: updates
Retrieval costs	Distributed index: split across machines

Vocabulary: all indexed terms.

Sparse vector: a vector with mostly zero coordinates.

One collection, one query

Document	Text
D1	database database database search
D2	database search
D3	search engine engine
D4	database systems systems systems
D5	engine systems
D6	music piano piano

Query: database search

Six documents. Six vocabulary terms. Every document contains text.

Retrieval structure and ranking model

View	Question it answers
Forward and inverted indexes	Forward: document $\rightarrow$ terms Inverted: term $\rightarrow$ documents
Vector retrieval alone	How similar is each document to the query?
Combined retrieval	How can postings compute the same scores?

Corpus: the document collection. All views use the same query.

Forward and inverted access

Document access

D1: database × 3, search × 1

Term access

database: (D1, 3), (D2, 1), (D4, 1)

Posting: one document entry in a term's list, often with a count.

Building a postings list

Stage	D1 example
Tokenize: split text into terms	database, database, database, search
Count within document	database: 3 search: 1
Append postings	database: (D1, 3) search: (D1, 1)
Update dictionary	Increment df once for each distinct term

$tf(d,t)$: term frequency, occurrences of term t in document d .

$df(t)$: document frequency, documents containing t .

D1's three database occurrences create one posting with $tf = 3$.

Documents as sparse term vectors

Document	database	search	engine	systems	music	piano
D1	3	1	0	0	0	0
D2	1	1	0	0	0	0
D3	0	1	2	0	0	0
D4	1	0	0	3	0	0
D5	0	0	1	1	0	0
D6	0	0	0	0	1	2

Query $q = [1, 1, 0, 0, 0, 0]$ (raw term counts)

A term vector has one coordinate per term; zero means absent.

The inverted index

Term	df(t)	Postings (document, tf)
database	3	(D1, 3) (D2, 1) (D4, 1)
search	3	(D1, 1) (D2, 1) (D3, 1)
engine	2	(D3, 2) (D5, 1)
systems	2	(D4, 3) (D5, 1)
music	1	(D6, 1)
piano	1	(D6, 2)

Each list stores one posting per document containing the term.

Inverted-index lookup: OR

database OR search

database: D1, D2, D4

search: D1, D2, D3

Result: D1, D2, D3, D4

OR: match at least one query term (the union of its lists).

Inverted-index lookup: AND

database AND search

database cursor	search cursor	Action
D1	D1	Emit D1, advance both
D2	D2	Emit D2, advance both
D4	D3	Advance search cursor
D4	End	Stop

Result: D1, D2

AND: match every term. Cursor: current position in a sorted list.

Query terms select the work

database: D1, D2, D4

search: D1, D2, D3

Candidate union: D1, D2, D3, D4

Candidate: a document sharing a query term.
Cosine similarity compares vector directions.
D5 and D6 share no query terms, so their cosine is zero.

Vector retrieval on its own

1. Represent the query and documents as term vectors
2. Compute a weighted similarity for every document
3. Return the documents with the highest scores

Dot product: sum of coordinate products.

Cosine: dot product divided by both vector lengths.

How we calculate query IDF weights

IDF (inverse document frequency) gives more weight to rarer terms.

$N = 6$ documents

$df(t)$ = documents containing term t

$\ln$ = natural logarithm

Term	$df(t)$	Calculation	IDF weight
database	3		1.0986
search	3		1.0986
engine	2		1.3863

df counts each document once, even when the term repeats within it.

Actual query: database search. Both query weights are 1.0986.

Engine is a comparison term. Its weight in this query is 0.

Weights and cosine similarity

q: query; d: document; t: term; N: number of documents

V: all indexed terms; Q: distinct query terms in V; $\ln$: natural logarithm

$w(d,t)$: document term weight

For $tf(d,t) > 0$; otherwise $w(d,t) = 0$.

W_d : document length (full L2 norm)

$w(q,t)$: query term weight

For $t \in Q$; otherwise $w(q,t) = 0$.

W_q : query vector length (L2 norm)

Cosine similarity

$N = 6$; $df(\text{database}) = df(\text{search}) = 3$.

Each query weight is $\ln(3) \approx 1.0986$.

Zero-length query: no positive matches.

Zero-length document: score 0.

Exhaustive vector scoring

```
for each document d:  
    dot = 0  
    for each term t in Q:  
        dot += w(q,t) × w(d,t)  
    score[d] = dot / (W_q × W_d)  
return highest-scoring documents
```

dot: running sum of coordinate products.
Use precomputed full-vector lengths.
If $W_d = 0$, score 0; if $W_q = 0$, no positive matches.

Score every document, including zeros.

Definitions used by the loop

Present terms; absent terms have weight 0.

Distinct terms in Q; other weights are 0.

Wd: all terms present in document d

Wq sums distinct query terms in Q

A document norm uses the whole vector

D3: search engine engine

Weighted document vector

Engine contributes to the norm even though the query contains no engine term.

Indexed vector scoring

```
A = empty map
for each query term t:
    for each (d, tf) in postings[t]:
         $A[d] += w(q,t) \times (1 + \ln(tf))$ 
for each document d in A:
     $score[d] = A[d] / (W_q \times W_d)$ 
return highest-scoring candidates
```

Accumulator $A[d]$: running weighted overlap for document d.
A sparse map stores only touched documents.

Postings update accumulators

Document	After database	After search
D1	2.3056	3.4042
D2	1.0986	2.1972
D3	0.0000	1.0986
D4	1.0986	1.0986

D1 receives a contribution from both postings lists.

Six posting visits

Visit	Term	Document	Increment	New accumulator
1	database	D1	2.3056	2.3056
2	database	D2	1.0986	1.0986
3	database	D4	1.0986	1.0986
4	search	D1	1.0986	3.4042
5	search	D2	1.0986	2.1972
6	search	D3	1.0986	1.0986

Every accumulator equals one partial dot product.

Normalization changes the ranking

Document	Dot product	Document norm	Cosine
D2	2.1972	1.4142	1.0000
D1	3.4042	2.3247	0.9425
D3	1.0986	1.9664	0.3596
D4	1.0986	2.3247	0.3042

Top 3: D2, D1, D3

D2 matches the query direction exactly, giving cosine 1.0000.

Why both routes return the same scores

The vector route sums query-coordinate products.

The indexed route visits every nonzero product in those same coordinates.

Both routes use the same document and query norms.

Same sums and denominators give the same cosine scores.

Demo: query to ranked results

Query: database search $K = \text{requested result count} = 3$

Demo view	What the audience sees
Build	Forward entries and inverted postings
Step	One posting updates one accumulator
Normalize	Full document norms change the order
Compare	Indexed scores match exhaustive scores

**Audience prediction: D1 has more term occurrences.
Will D1 or D2 have the highest cosine?**

Three demo views

Mode	Visible execution	Default result
Index comparison	Scan documents vs traverse postings	OR: D1, D2, D3, D4
Vectors alone	Score D1 through D6	Top 3: D2, D1, D3
Combined	6 posting visits, 4 candidates	Top 3: D2, D1, D3

The equality view compares all six document scores.

Query cost

N: documents; m: query terms; P: posting visits; A: candidate count

$O(\cdot)$: asymptotic work growth. A size-K heap retains the best K results.

Route	Scoring	Result selection
Exhaustive	$O(Nm)$	$O(N \log(K+1))$
Indexed, sparse map	$O(m + P + A)$	$O(A \log(K+1))$

This example: 12 coordinate checks versus 6 posting visits

Counts describe different operations, not a measured speedup.
Common query terms can still make P and A large.

Queries that reveal different behavior

Query or change	What to inspect
music	One list and one candidate
engine systems	D5 receives contributions from both terms
unknownterm	No in-vocabulary query terms
database database search	Same result under the set-query policy
Edit a document	Rebuild postings, df and norms together

A changed corpus can change scores throughout the collection.

Video Google: visual-word retrieval

Video Google: A Text Retrieval Approach to Object Matching in Videos

Josef Sivic and Andrew Zisserman
International Conference on
Computer Vision (ICCV), 2003

Frame: one video image

**Descriptor: a patch's numeric
features**

**Visual word: an identifier (ID) for
similar descriptors**

An earlier application of inverted retrieval

Query region



Another view



One object across different views

Photos: Oxford Visual Geometry Group demo,
Groundhog Day

1–2. Learn a vocabulary of visual words

128D descriptor = 128 numbers describing one local patch

SIFT: Scale-Invariant Feature Transform; a 128D local descriptor

k-means: group around k centers	What changes
Assign descriptors	Nearest centroid in descriptor space
Update centroids	Mean of each assigned cluster
Repeat until stable	Codebook: shared set of centroids

x : descriptor; C_j : cluster j ; μ_j : mean (centroid) of C_j
 $|C_j|$: cluster size; k : number of clusters

Visual-word IDs: $V_1, V_2, \dots, V_k$

3. Quantize each descriptor to a word ID

Centroid	Squared distance
V1	0.0008 ← nearest
V2	1.3750
V3	1.7468
V4	1.7525
V5	1.7562
V6	1.7550

F2-P1 becomes V1

F2-P1: patch 1 in frame F2.
128D descriptor; k-dimensional
word vector

Squared distance: sum of squared
coordinate differences.

$c(x)$: word ID. argmin : ID giving the smallest distance.

4. Build visual-word inverted files

Frame	V1	V2	V3	V4	V5	V6
F1	3	1	0	0	0	0
F2	1	1	0	0	0	0
F3	0	1	2	0	0	0
F4	1	0	0	3	0	0
F5	0	0	1	1	0	0
F6	0	0	0	0	1	2

V1: (F1, 3), (F2, 1), (F4, 1)

V2: (F1, 1), (F2, 1), (F3, 1)

5. Query postings and rank frame vectors

j : word ID; n_{jd} : word j 's count in frame d ; n_d : descriptors in d ; df_j : frames containing j ; N : frames
 TF-IDF: term frequency $\times$ inverse document frequency; $v_j(d)$: weighted visual-word coordinate
 v_q, v_d : weighted query/frame vectors; $\|v\|_2$: L2 length; $v_q^T v_d$: dot product

Query patches select V1 and V2

Frame	Exhaustive cosine	Indexed cosine
F2	1.0000	1.0000
F1	0.8944	0.8944
F3	0.2127	0.2127
F4	0.1455	0.1455

6 posting visits, 4 candidates. Exhaustive route: 12 probes.

Photo demo: query patches to ranked frames

SIFT = Scale-Invariant Feature Transform (128D patch descriptor)

Query: eight descriptors from the outlined tie



F1 · query view

Cosine 0.9382



F2

Cosine 0.8193



F4

Cosine 0.7924

24 words, 144 SIFT descriptors. 3 posting visits; 6 exhaustive probes.

What this model captures

Each dimension corresponds to a vocabulary term.

Shared terms contribute to the dot product.
Word order does not affect this bag-of-words score.

A relevant document may use different words and receive a zero lexical score.

Bag-of-words ignores order; lexical matching uses exact term overlap.

Limitations and scope boundaries

Limited treatment of relevance effectiveness

Hardware-specific performance examples reflect 2006.

Neural retrieval uses learned representations and lies beyond the paper's historical scope.

Effectiveness: relevance of results. Efficiency: resources used to obtain them.

Representative later developments

Work	Development	Relationship
Block-Max WAND Ding & Suel, 2011	Skip blocks using score bounds	Cites the survey
Partitioned Elias-Fano Ottaviano & Venturini, 2014	Compress locally clustered lists	Cites the survey
SPLADE Formal et al., 2021	Learn sparse weights and expansion	Same index principle

Score bound: maximum possible score. Pruning skips work that cannot change top K.

Citation documents a link. It does not establish invention.

WAND: score-bound skipping. SPLADE: Sparse Lexical and Expansion Model.

Future directions

Exact top-K pruning with justified score bounds

Compression tuned to current hardware and workloads

Learned sparse representations with inverted access

Present-day directions, rather than the authors' stated agenda

Conclusion

Inverted lists expose the nonzero coordinates needed for the query's dot products.

**Accumulation computes the numerator.
Normalization completes cosine scoring.**

Primary reading

Zobel & Moffat. Inverted Files for Text Search Engines.

ACM Computing Surveys 38(2), Article 6, 2006. Sections 2–3.