IR U2

Components of an Index

Indexes are specialized **data structures** used in search engines and databases to improve the speed of retrieving relevant documents.
A typical index (like an *inverted index*) is made up of the following components:

1. Dictionary (Lexicon)

- **Definition:** Stores **all unique terms** (words) present in the document collection, much like a vocabulary.
- **Purpose:**
  - Acts as an **entry point** for searching.
  - Links each term to its **postings list**.
- **Example:** If documents contain "AI", "data", "science", the dictionary stores these terms once.

2. Postings Lists

- **Definition:** For each term in the dictionary, there is a **list of postings**, where each posting contains:
  - **Document ID (docID)** → The document where the term appears.
  - Optional: **Term Frequency (TF)**, **positions** of the term in the document.
- **Purpose:** Helps identify **exactly which documents** contain the term.
- **Example:** For "AI" → [Doc3, Doc5, Doc8]

3. Skip Pointers

- **Definition:** Special links inside postings lists that allow the search engine to **jump over sections** instead of scanning every entry sequentially.
- **Purpose:** Improves search speed, especially in large postings lists.
- **Example:** Instead of reading each docID one-by-one, skip pointers might jump from Doc3 → Doc10 → Doc17.

4. Metadata

- **Definition:** Additional data stored with terms or documents to support ranking and filtering.
- **Common Metadata Fields:**
  - **Term Frequency (TF):** How many times a term appears in a document.
  - **Document Frequency (DF):** Number of documents containing the term.
  - **Document Length:** Total words in the document.
- **Purpose:** Used in ranking algorithms like **TF-IDF** or **BM25**.

Index Life Cycle

The **Index Life Cycle** refers to the different stages an index goes through from its initial creation to its removal.
This process ensures that the index remains **accurate, efficient, and optimized** for fast data retrieval.

1. Creation:

- **Definition:** Building the index for the first time from a collection of documents.

- **Process:**
  - Extract terms from documents.
  - Create dictionary and postings lists.
  - Store index on disk.
- **Example:** First-time indexing of a new document collection.

2. Updating

- **Definition:** Modifying the index when documents are **added, updated, or deleted**.
- **Process:**
  - Incrementally update postings lists.
  - Adjust metadata like term/document frequency.
- **Example:** Adding new blog posts to a search engine index.

3. Merging

- **Definition:** Combining **smaller partial indexes** into a larger one to improve efficiency.
- **Purpose:** Reduces lookup time and storage overhead.
- **Example:** Merging daily indexes into a monthly index.

4. Optimization

- **Definition:** Reorganizing index structure to remove **fragmentation** and make queries faster.
- **Process:**
  - Reorder postings lists.
  - Remove deleted document entries.
- **Example:** Compacting an index to improve performance.

5. Deletion

- **Definition:** Removing outdated or unused indexes.
- **Purpose:** Saves storage space and avoids searching through obsolete data.
- **Example:** Removing old index files after a product catalog update.

Static Inverted Index

A **Static Inverted Index** is an index that is **built once** from a fixed document collection and **never updated** after creation.
It is most useful when the dataset is **unchanging** or **rarely updated**, so the index structure remains constant.

Key Characteristics

1. **Built Once** → Created from the complete document collection at a single point in time.
2. **Immutable** → Cannot be modified; if changes are required, the **entire index must be rebuilt**.
3. **Optimized for Search** → Since there are no update operations, the index can be stored in a highly compact and query-efficient format.

When to Use

- When the document collection is **static** (does not grow or change).
- Examples:
  - Digital archives of historical books.
  - Static research paper repositories.
  - Government records that are frozen after a certain date.

Advantages

- **Faster Query Processing:**
  No update overhead → structure can be fully optimized for search speed.
- **Compact Storage:**
  No need to keep extra data structures for updates.
- **Simplicity:**
  Easier to maintain since there are no incremental updates or merges.

Disadvantages

- **No Real-Time Updates:**
  Cannot handle newly added or modified documents without **rebuilding the index from scratch**.
- **Rebuild Cost:**
  For large collections, rebuilding the index is time-consuming and resource-intensive.
- **Not Suitable for Dynamic Content:**
  Poor choice for news sites, social media, or e-commerce where content changes frequently.

Example

Imagine an **online archive** of Shakespeare's complete works:

- The text is fixed and never changes.
- A static inverted index is created to store every unique word and the list of plays/pages it appears in.
- The index will never need to update because the text is permanent.

Dictionaries

A **dictionary** in information retrieval is the part of the index that **maps each term to its postings list** (list of documents where the term appears).
It is the first step in retrieving information — a search begins by finding the term in the dictionary.

a) Sort-Based Dictionary

- **Structure:** Terms stored in **sorted order** (often lexicographical order: A → Z).
- **Search:**
  - Use **binary search** to find the term quickly in O(log n) time.
  - Good for finding **exact matches** and **prefix matches**.
- **Advantages:**
  - Supports **prefix queries** (e.g., "comp*" matches "computer", "competition").
  - Easy to compress dictionary since terms are stored in a predictable order.

- **Disadvantages:**
    - **Slow updates**: Inserting new terms requires shifting entries to maintain order.
    - Not suitable for highly dynamic collections.

b) Hash-Based Dictionary

- **Structure:** Terms stored in a **hash table** with term as key and postings list as value.
- **Search:** Constant time lookup on average **O(1)**.
- **Advantages:**
    - Very fast lookups for exact matches.
    - Works well for large vocabularies where quick access is important.
- **Disadvantages:**
    - No inherent ordering → cannot easily perform prefix searches or range queries.
    - Hash collisions require extra handling.

c) Interleaving

- **Structure:** Stores multiple term lists **interleaved** in memory to save space and improve **cache locality**.
- **Purpose:**
    - Reduces **memory fragmentation**.
    - Improves **CPU cache performance** during lookups.
- **Advantages:**
    - Space-efficient.
    - Can speed up search operations due to better memory access patterns.
- **Disadvantages:**
    - More complex to implement.
    - May require decoding steps during lookup.

d) Posting Lists

- **Definition:** The **core mapping**: Term → List of documents (docIDs) containing that term.
- **Contents of a Posting:**
    - **Document ID (docID)**
    - **Term Frequency (TF)** – How many times the term appears in that document.
    - **Positions** – For phrase queries (exact position of term in text).
    - **Weights** – Used in ranked retrieval (TF-IDF, BM25 scores).
- **Advantages:**
    - Stores all necessary information for retrieval.
- **Disadvantages:**
    - Can become very large — needs compression techniques (e.g., delta encoding, variable-byte encoding).

Index Construction Methods

Index construction is the process of **creating an index** from a set of documents to allow fast and efficient information retrieval. Different construction methods are used depending on **collection size**, **memory availability**, and **update frequency**.

1. In-Memory Index Construction

Process:

1. Read the **entire document collection** into RAM.
2. Tokenize documents to extract terms.
3. Build the **dictionary** and **postings lists** in memory.
4. Write the complete index to disk once construction is done.

Advantages:

- **Fastest method** since all operations are in RAM.
- Simple to implement; fewer I/O operations.

Disadvantages:

- **Memory-bound** — cannot handle collections larger than available RAM.
- Not scalable for very large or continuously growing datasets.

Use Case:

- Small academic projects or local search tools with a limited number of documents (<1–2 GB).

2. Sort-Based Index Construction

Process:

1. Extract all **(term, docID)** pairs from the document collection.
2. **Sort** these pairs first by term, then by document ID.
3. Merge duplicates to form postings lists.
4. Store the sorted dictionary and postings lists on disk.

Advantages:

- Works well for **large collections** that do not fit entirely in memory.
- Sorted order simplifies **compression** and **prefix searches**.

Disadvantages:

- Sorting is **computationally expensive** for very large datasets.
- Not ideal for highly dynamic collections because frequent updates require repeated sorting.

Use Case:

- Batch processing of static collections such as archives, digital libraries, or offline document repositories.

3. Merge-Based Index Construction

Process:

1. Build **small partial indexes** in memory from subsets of documents.
2. When memory is full, write partial indexes to disk.
3. Periodically **merge partial indexes** into a single large index using an efficient multi-way merge.
4. Repeat this process as new documents arrive.

Advantages:

- Can handle **massive datasets** larger than available RAM.
- Supports **dynamic updates** efficiently by merging small indexes instead of rebuilding the entire index.

Disadvantages:

- Merge operations require **extra CPU and disk I/O**, which can be time-consuming.
- Implementation is **more complex** than sort-based or in-memory methods.

Use Case:

- Web search engines or online platforms where documents are constantly added, modified, or deleted (e.g., Google, Bing).

4. Disk-Based Index Construction

Process:

1. Build the index **directly on disk** without loading the entire collection into memory.
2. Postings are incrementally written to disk, often using buffers to optimize I/O.
3. No in-memory consolidation is needed.

Advantages:

- Can handle **very large datasets** (terabytes or petabytes).
- Low memory requirements — can work on systems with limited RAM.

Disadvantages:

- **Slower** due to frequent disk reads/writes.
- More complex to manage, especially when performing merges or updates.

Use Case:

- Archival systems, government or scientific datasets where scale is prioritized over indexing speed.

Dynamic Indexing

Dynamic indexing is a way to keep an index up-to-date while documents are added, changed, or removed — **without rebuilding the whole index** every time.

Why we need it

- Real-world collections (news sites, blogs, web pages) change constantly.
- Rebuilding the entire index for each change would be **too slow and expensive**.
- Dynamic indexing lets the system accept updates quickly and keep searches correct.

How it works (simple steps)

1. **New or updated documents arrive.**
2. The system writes them to a **small in-memory index** (fast to update).

3. Queries check both the **in-memory index** and the **main disk index** so search results include recent documents.
4. When the in-memory index grows past a threshold, it is **merged** into the large disk-based index (background process).
5. During merge, deleted documents are permanently removed and space is reclaimed.

Common techniques / components

- **In-memory buffer / index:** Fast structure (RAM) that stores recent changes (insertions/updates/deletes).
- **Main disk index:** Large, optimized index stored on disk for long-term storage and fast reads.
- **Merge (compaction):** Periodic process that combines small in-memory index(es) with the main index to keep structure efficient.
- **Tombstones (delete markers):** When a document is deleted, a "tombstone" is recorded so the system knows to ignore it until merge removes it fully.
- **Log-structured approach / LSM-style:** Many systems use log-structured ideas (write new entries sequentially, merge later) to make writes cheap and merges efficient.

How queries are answered

- A search first looks in the **in-memory index** (to find newest docs), then in the **disk index**.
- Results are merged and ranked together, so users see up-to-date results immediately (even before a merge).

Handling deletes and updates

- **Update = delete + add:** A document update can be recorded as a tombstone for the old version plus a new posting for the updated version.
- **Tombstones** ensure deleted items are not returned in queries until a merge permanently removes them.

Advantages

- **Fast updates and inserts** (writes go to RAM, cheap).
- **Low query latency for new data** (recent documents are searchable immediately).
- **Scales to large, changing collections** — you avoid frequent full rebuilds.

Disadvantages

- **Merge/compaction cost:** background merges use CPU and disk I/O and can be expensive.
- **Temporary storage overhead:** until merged, there are multiple index fragments and tombstones which use space.
- **Complexity:** implementation is more complex than a static index (concurrency, consistency, merging logic).

Query Processing for Ranked Retrieval

Ranked retrieval is a method in information retrieval where **documents are returned not just as matching or not matching**, but in **ranked order of relevance** to the user query. Modern search engines (like Google, Bing) use this approach to show the most useful results first.

Steps in Ranked Retrieval

1. Retrieve Postings Lists for Query Terms

- **Postings list:** For each term in the query, there is a list of documents containing that term.
- **Step:**
  - For query Q = {term1, term2, term3}, fetch postings lists for each term from the index.
  - Example:
  - term1 → [Doc1, Doc3, Doc5]
  - term2 → [Doc2, Doc3, Doc6]
  - term3 → [Doc1, Doc4, Doc5]
- **Purpose:** Identify which documents contain the query terms.

2. Compute Relevance Score

- Each document is assigned a **score** representing how relevant it is to the query.
- **Common scoring methods:**

1. **TF-IDF (Term Frequency–Inverse Document Frequency):**

        - Measures importance of a term in a document relative to the entire collection.
        - Formula (simplified):
        - score(D, Q) = Σ (TF(term, D) * IDF(term))

2. **BM25:**
    - Advanced scoring model that considers term frequency, document length, and collection statistics.
    - More accurate for real-world search engines.

3. **Vector Space Model:**
    - Represents documents and queries as vectors in a multi-dimensional space.
    - Score = cosine similarity between query and document vectors.

- **Purpose:** Determine which documents are most relevant to the user's query.

3. Rank Documents by Score

- Once scores are computed for all documents, sort them **from highest to lowest**.
- Documents with higher scores are shown first in the search results.
- **Example Table:**

Document	Score
Doc3	0.95
Doc1	0.87
Doc5	0.75
Doc4	0.60

- Here, **Doc3** is considered the most relevant, so it appears at the top.

4. Return Results to User

- The top k documents are returned to the user.
- Many search engines may also apply **additional ranking signals**: user behavior, personalization, freshness, etc.

Additional Notes

- **Document-at-a-Time vs Term-at-a-Time:**
    - **Document-at-a-Time:** Score each document across all query terms before moving to the next document.
    - **Term-at-a-Time:** Process all documents for a single term, then combine scores across terms.
- **Pre-computing Scores:** Frequently accessed queries may have pre-computed ranking to improve speed.
- **Impact Ordering:** Process terms with the largest potential contribution to relevance first to reduce computation.

Advantages of Ranked Retrieval

- Users get **more relevant results first**, improving satisfaction.
- Can handle **long queries** and **partial matches**.
- Allows advanced ranking using TF-IDF, BM25, or learning-to-rank models.

Disadvantages / Challenges

- Requires **scoring all candidate documents**, which can be computationally expensive for large collections.
- Accuracy depends on **quality of scoring model** and **index metadata**.
- Ranking can be biased if scores don't consider all factors (like user intent).

Document-at-a-Time (DAAT) Query Processing

Document-at-a-Time (DAAT) is a query processing strategy where the search engine **processes one document at a time** across all postings lists of the query terms.
It is commonly used in **ranked retrieval systems** to compute document scores efficiently.

Key Idea

- Instead of processing one term at a time (Term-at-a-Time), DAAT **focuses on documents**.
- For each document, the system **combines contributions from all query terms** to compute the final relevance score before moving to the next document.

- Helps in **early pruning** when lists are sorted by impact (high-scoring documents first).

Step-by-Step Process

1. **Retrieve postings lists** for all query terms.
    - Each postings list contains **document IDs** where the term appears, optionally with **term frequency**, positions, or weights.
2. **Identify the smallest document ID** among the current pointers of all lists (or the next document in order).
    - This is the next document to process.
3. **Compute the full score** for this document by combining contributions from all query terms.
    - Example: Using TF-IDF or BM25, sum the individual term scores for this document.
4. **Move pointers forward** in the postings lists for the document just processed.
5. Repeat steps 2–4 until **all postings lists are exhausted** or the top-k results are obtained.

Example

Query: "machine learning"

Postings lists:

machine → [Doc1, Doc3, Doc5]

learning → [Doc2, Doc3, Doc4, Doc5]

Processing Steps (DAAT):

1. Next smallest docID across lists = Doc1
    - Score = contribution from “machine” (present) + “learning” (absent) = partial score
    - Move pointer for "machine" to next docID
2. Next smallest docID = Doc2
    - Score = contribution from “machine” (absent) + “learning” (present)
    - Move pointer for "learning"
3. Next smallest docID = Doc3
    - Score = contributions from both terms
    - Move both pointers
4. Continue until all documents are processed

Finally, documents are **ranked by total scores**.

Pros of DAAT

- **Full-score computation per document:** Ensures accurate ranking.
- **Works well with impact-ordered postings:** High-scoring documents can be processed first, enabling early termination in some algorithms.
- **Memory-efficient:** Can process large postings lists sequentially.

Cons of DAAT

- **May require scanning all postings lists** even if we only need top-k results.
- **Slower if lists are very large** and there is no effective early pruning.
- More complex implementation compared to term-at-a-time approaches.

Key Points to Remember

- DAAT processes **documents in order**, not terms.
- Score of a document = **sum of contributions from all query terms**.
- Ideal for **impact-sorted indexes** where higher-scoring documents appear early.
- Often used in **modern search engines** combined with optimization techniques to reduce the number of documents scored.

Term-at-a-Time (TAAT) Query Processing

Term-at-a-Time (TAAT) is a query processing strategy where the search engine **processes one query term at a time**, updating document scores for all documents containing that term. After all terms are processed, documents are ranked by their accumulated scores.

Key Idea

- Instead of computing the score for one document at a time (like DAAT), TAAT **focuses on one term at a time**.

- For each term, all documents containing that term are **updated in a score accumulator**.
- At the end, the score accumulators contain the **total relevance scores** for all documents.

Step-by-Step Process

1. **Retrieve postings list** for the first query term.
   - Each posting contains **document ID** and optionally **term frequency** or weights.
2. **Update score accumulators** for each document in the postings list.
   - Add the contribution of this term to the document's total score.
3. **Repeat steps 1–2** for all remaining query terms.
   - Each document’s score is updated cumulatively across all terms.
4. **Rank documents** by their final accumulated scores.
   - Return the top-k results to the user.

Example

Query: "machine learning"

Postings lists:

machine → [Doc1, Doc3, Doc5]

learning → [Doc2, Doc3, Doc4, Doc5]

Processing Steps (TAAT):

1. Process machine postings:
   - Doc1: score += contribution of “machine”
   - Doc3: score += contribution of “machine”
   - Doc5: score += contribution of “machine”
2. Process learning postings:
   - Doc2: score += contribution of “learning”
   - Doc3: score += contribution of “learning”
   - Doc4: score += contribution of “learning”
   - Doc5: score += contribution of “learning”
3. Score accumulator after all terms:

   Doc1 → 0.8

   Doc2 → 0.6

   Doc3 → 1.5

   Doc4 → 0.7

   Doc5 → 1.3
4. **Rank documents**: Doc3, Doc5, Doc1, Doc4, Doc2

Pros of TAAT

- **Simple to implement** and understand.
- Easy to combine **weights or term contributions** incrementally.
- Can handle complex scoring models where each term contributes independently.

Cons of TAAT

- **May process unnecessary postings** for documents that eventually get very low scores.
- Less efficient than DAAT for top-k queries because it can touch many low-relevance documents.
- Not ideal for very large collections without optimizations.

Pre-computing Score Contributions

Pre-computing score contributions is an optimization technique in ranked retrieval where **partial or complete scores for terms in documents are computed in advance** and stored in the index. This reduces the computation needed at query time and speeds up search results.

Key Idea

- Instead of calculating term scores (e.g., TF-IDF, BM25) for each document **during query processing**, we **pre-calculate** them and store in the index.
- During query execution, the search engine **simply retrieves pre-computed scores** from the index and combines them to rank documents.

How It Works

1. During index construction, compute **score contribution** of each term for each document.
   - Example: TF-IDF weight of term t in document D.
2. Store this **score in the postings list** along with the document ID.
   - Postings entry format: (DocID, TF-IDF_weight, positions, etc.)
3. At query time, retrieve postings lists for all query terms and **sum pre-computed scores** instead of calculating from scratch.

Example

Suppose query: "machine learning"

- Pre-computed TF-IDF weights in postings:

machine → Doc1: 0.8, Doc3: 0.7, Doc5: 0.9

learning → Doc2: 0.6, Doc3: 0.8, Doc4: 0.5, Doc5: 0.7

- At query time, score for Doc3 = 0.7 (machine) + 0.8 (learning) = **1.5**
- No need to calculate TF-IDF from raw term frequencies during the query.

Advantages

- **Faster query processing:** Scores are ready to use.
- Reduces CPU usage at query time.
- Useful for **high-traffic search engines** with millions of queries.

Disadvantages

- **Increased index size:** Storing scores for all term-doc pairs consumes more space.
- **Static scores:** If scoring parameters change (e.g., normalization, new ranking function), scores must be recomputed and the index rebuilt.

Impact Ordering

Impact ordering is a technique in information retrieval where **postings lists are sorted by the "impact" (importance) of the term in the document** rather than by document ID. The goal is to **process the most relevant documents first**, improving efficiency, especially in top-k retrieval.

Key Idea

- In traditional inverted indexes, postings are **ordered by document ID**.
- In impact-ordered indexes, postings are **sorted by term contribution to document relevance**, e.g., TF-IDF weight or BM25 score.
- This allows the system to **focus on documents with higher potential relevance first**, often skipping less relevant documents.

How It Works

1. **During index construction:**
   - Compute the **impact score** of each term in each document (e.g., TF-IDF or BM25 weight).
   - Sort the postings list for each term in **descending order of impact score**.
2. **During query processing:**
   - Start scoring documents from the **highest impact postings**.
   - Stop processing once the **top-k documents are guaranteed to be found**, without scanning the entire list.

Example

Query: "machine learning"

Impact-ordered postings list for "machine":

Doc5: 0.9 → Doc1: 0.8 → Doc3: 0.7

- For top-2 retrieval, the system can focus on **Doc5 and Doc1** first, and may skip Doc3 if it's guaranteed not to enter top results.

Impact vs DocID ordering:

DocID Order	**Doc5**	**Doc1**	**Doc3**
Impact Order	0.9	0.8	0.7

- **Top-k processing** becomes more efficient because high-impact documents are considered first.

Advantages

- **Speeds up top-k queries:** Only high-impact postings may need to be scanned.
- **Reduces unnecessary computation:** Low-impact documents can be ignored if they won't enter top results.
- **Works well with DAAT and TAAT:** Especially when combined with early termination techniques.

Disadvantages

- **Index sorting overhead:** Postings must be sorted by impact score during index construction.
- **Dynamic updates are harder:** Adding new documents may require re-sorting postings.
- **Best for ranked retrieval:** Less useful for exact-match or boolean queries.

Query Optimization
Query optimization refers to the **techniques used in information retrieval systems to make query processing faster and more efficient**, without changing the correctness of the results. It is crucial for **large-scale search engines** that handle millions of queries per day.

Why Query Optimization is Needed

- Searching large document collections can be **computationally expensive**.
- Naively processing all postings lists for every query can waste time and resources.
- Optimization techniques **reduce the number of postings processed** and speed up top-k retrieval.

Common Techniques for Query Optimization

1. Term Ordering

- **Definition:** Process query terms in an **optimal order** to reduce computation.
- **How it works:**
    - Rare terms (terms that appear in fewer documents) are processed **first**.
    - Why? Intersecting rare-term postings with other lists quickly reduces candidate documents.
- **Example:**
  Query: "machine learning AI"
    - Suppose document frequencies: machine = 5000, learning = 2000, AI = 1000
    - Process order: AI → learning → machine
    - Fewer documents are scored in early steps → faster query execution.

2. Skip Pointers

- **Definition:** Extra pointers in postings lists that allow **jumping over irrelevant documents** without scanning each one sequentially.
- **How it works:**
    - Postings lists are sorted by document ID.
    - Skip pointers allow the algorithm to jump ahead when the current docID is smaller than needed.
- **Benefit:** Reduces the number of comparisons during intersections.
- **Example:**
    - Postings: [1, 2, 3, 10, 15, 20]
    - Skip pointer from 3 → 10 allows **jumping over 4–9** instead of checking each docID.

3. Caching

- **Definition:** Store **results of frequent queries** or partial computations to avoid recomputation.
- **How it works:**
    - When a common query is executed, the result (or top-k docs) is stored in cache.
    - Subsequent identical queries fetch results **directly from cache**, saving time.
- **Example:**
    - Query "COVID-19 vaccine" is searched thousands of times.

    - Cache stores top-k results → immediate response for repeated queries.

4. Early Termination

- **Definition:** Stop processing postings lists once **top-k documents are guaranteed**, rather than scoring all candidates.
- **How it works:**
    - Use **impact-ordered lists** or **threshold-based scoring**.
    - Once the top-k scores are unlikely to change by processing remaining postings, stop computation.
- **Benefit:** Significantly reduces runtime for large postings lists.
- **Example:**
    - Top-10 documents requested.
    - High-impact documents processed first; low-impact documents cannot enter top-10 → stop early.

Combination of Techniques

- These techniques are often combined in modern search engines:
    - **Impact ordering + early termination:** Process most relevant postings first and stop when top-k results are stable.
    - **Skip pointers + term ordering:** Quickly reduce candidate documents.
    - **Caching:** Reduces repeated work for popular queries.

Advantages of Query Optimization

- Reduces **query response time**.
- Minimizes **CPU and memory usage**.
- Improves **user experience** with faster search results.
- Essential for **large-scale, real-time search systems**.

Disadvantages / Challenges

- Additional **index maintenance** may be required (e.g., skip pointers, pre-computation).
- Cache may consume extra **memory**.
- Complexity increases with **dynamic or frequently updated collections**.