Ques 1: What is meant by an Information Retrieval System? Discuss how the process of searching influences the overall document collection.

Definition (high level)
An **Information Retrieval (IR) System** stores, indexes, and retrieves documents (text, audio, images, etc.) in response to users' queries. Unlike a database (which answers precise structured queries), IR systems handle *unstructured* or *semi-structured* content and return ranked lists of documents by estimated relevance.

Core goals

- retrieve documents that satisfy an information need (recall),
- rank the results so the most useful appear early (precision / ranking),
- do both efficiently at web or enterprise scale.

Main components (brief)

- Document acquisition (crawlers, ingestion pipelines)
- Text processing (tokenization, stopword removal, normalization, stemming/lemmatization)
- Indexer (usually inverted index: term → postings list)
- Query processor (parsing, expansion)
- Retrieval/ranking model (Boolean, Vector Space, BM25, neural rankers)
- UI & feedback (query logs, relevance feedback)
- Index maintenance (updates, deletions, merges)

How the *search process* influences the document collection
Search is not just a consumer of the collection — it also shapes it. Important influences:

1. **Indexing choices shape what's retrievable**
    - Tokenization, stopword lists, stemming/lemmatization, case-folding affect which terms are indexed. If you stem aggressively, morphological variants collapse (better recall, sometimes worse precision).
    - Which fields are indexed (title, body, metadata) affects matching.
2. **Query logs drive collection evolution**
    - Frequent query patterns signal which documents or topics are important → operators may crawl/ingest more documents on those topics or promote them.
    - Query logs fuel query expansion dictionaries, synonym lists, and machine-learned ranking features.
3. **Feedback & relevance judgments shape re-indexing and ranking**
    - Relevance feedback (explicit or implicit via clicks) can cause reweighting of documents, learning-to-rank model updates, personalized indexes, or curated content addition/removal.
4. **Curation, deduplication, and quality control**
    - Popular search patterns reveal low-quality or spammy documents; operators may block or downrank them, reducing their presence in the effective collection.
5. **Bias and "rich-get-richer" effects**
    - Document visibility in search influences subsequent traffic and linking, which can reinforce the prominence of already-popular content; system design (e.g., diversification) is used to mitigate this.

6. **Storage & partitioning decisions**

    ◦ Search workloads (hot queries, burst traffic) determine sharding/replication policies and which documents are kept in memory caches vs. cold storage.

7. **Legal/privacy and retention policies**

    ◦ Search requirements (e.g., GDPR right-to-be-forgotten) affect which documents are retained and how they are indexed (e.g., low-granularity vs. hashed personally-identifying fields).

8. **Update & incremental indexing needs**

    ◦ Frequently searched topics may require low-latency updates (news), pushing the system design toward dynamic/delta indexes instead of large batch rebuilds.

Practical example
A medical search engine that discovers many queries for a new drug will:

- prioritize crawling papers & official pages about the drug,
- add domain-specific synonyms (brand/generic name),
- choose to index abstracts and clinical-trial tables as fields,
- and expose freshness signals in ranking.

Summary
Search and the document collection are in a feedback loop: indexing and preprocessing choices determine what search can retrieve; search behavior (queries, clicks) drives collection growth, curation, ranking models, and storage policies.

Ques 2: What do you understand about Vocabulary Domains? Describe the functional architecture of an Information Retrieval System.

Vocabulary Domains — meaning & importance

- A **vocabulary domain** is the set of words, phrases, abbreviations, technical terms, and conventions used within a specific subject area or user community (e.g., medical, legal, social media slang).
- **Why it matters**:
    - *Synonymy*: multiple surface forms for same concept (e.g., "myocardial infarction" vs "heart attack").
    - *Polysemy*: same word different meanings (e.g., "bank").
    - *Morphology & orthography*: domain affects tokenization (chemical formulas, hyphens, word compounds).
    - *Special tokens*: codes, abbreviations, measurement units, gene names, legal citations.

Effects on IR design

- Need for domain-specific dictionaries, ontologies (UMLS for medicine), or term normalization.
- Query expansion and mapping of user vocabulary to document vocabulary (synonym maps).
- Different stop-word lists and stemming/lemmatization rules per domain.

Functional architecture of an IR system (detailed data flow)

I'll describe a classical pipeline from document ingestion through query to ranking. For each block I include its responsibilities and common data structures.

1. **Document acquisition**

- Crawlers, file ingestion, connectors (APIs) collect documents.
- Responsibilities: fetch, deduplicate, respect robots/policies.

2. **Document parsing & preprocessing**
   - Parse raw content (HTML → text, PDF → text).
   - Extract fields (title, author, body, metadata).
   - Language identification, charset normalization.
3. **Text processing (term extraction pipeline)**
   - **Tokenization** (split text into tokens/words).
   - **Stop-word removal** (optional).
   - **Normalization** (case-folding, accent removal, punctuation handling).
   - **Stemming/Lemmatization** (reduce morphological variants).
   - **Named-entity recognition** / phrase detection.
   - Output: list of normalized tokens with positions (for positional indexes).
4. **Index construction**
   - **Lexicon (dictionary)**: mapping term → term ID, document frequency, pointers to postings.
   - **Postings lists**: for each term, list of posting entries (docID, term frequency, positions, field info).
   - **Storage choices**: compressed postings (variable-byte, gamma), skip pointers, positional info.
   - Writes to disk as inverted index; may be partitioned/sharded.
5. **Index maintenance**
   - Merging blocks, delta indexes for dynamic updates, rebuilding/refreshing.
6. **Query processing**
   - Query parsing & normalization (apply same tokenization and normalization as docs).
   - Query expansion (synonyms, spelling correction, query logs).
   - Translate query to operations on index (boolean, term weight vector).
7. **Retrieval & ranking**
   - **Candidate generation**: fetch postings for query terms, union/intersect/posting merges.
   - **Scoring**: compute relevance scores (TF-IDF, BM25, probabilistic models, or neural ranker).
   - **Re-ranking**: apply secondary features (click data, freshness, personalization).
   - **Output**: ranked document list with snippets.
8. **User interface & feedback**
   - Display results, facets, snippets, query suggestions.
   - Collect implicit feedback (clicks/time-on-page) and explicit relevance labels.

9. **Analytics & learning**

    - Use query logs, A/B tests, learning-to-rank training, and model updating.

10. **Scale & distributed architecture**

    - **Sharding**: split index across machines (by document or by term).
    - **Replication**: copies for availability.
    - **Coordination**: distributed search front-ends, aggregator nodes.

Data structures of note

- Inverted index (term → postings)
- Blocked/segment files (for merges)
- Lexicon + vocabulary statistics (df, nt)
- Document store (for retrieving original doc and snippets)
- Forward index or document vectors (sometimes used for re-ranking)

Wrap-up
The IR architecture is intentionally layered so linguistic processing and indexing are decoupled from ranking and UI. Vocabulary domains feed into preprocessing and query-expansion stages: the closer the preprocessing matches domain usage, the better the retrieval effectiveness.

Ques 3: What is Boolean Retrieval? Describe the procedure for finding the common elements between two postings lists p1 and p2.

Boolean retrieval — quick definition

- **Boolean retrieval** uses Boolean logic (AND, OR, NOT) to retrieve documents that either *satisfy* or *don't* satisfy exact Boolean expressions over terms.
- Documents are either *relevant* or *not* (no ranking by degree of relevance in pure Boolean model).

Characteristics

- Fast, exact matching.
- Typically implemented with inverted indexes; boolean operators correspond to set operations (intersection, union, complement).
- No ranked ordering (unless augmented with heuristics or later ranking stage).

Finding common elements between two postings lists (intersection) — merge algorithm

Assume postings lists are sorted in increasing order of docID.

Input: p1 (length m), p2 (length n), both sorted.
Goal: compute `p1 ∩ p2` (docs appearing in both).

Two-pointer merge algorithm (standard)

Pseudocode:

i ← 0, j ← 0

answer ← []

```
while i < len(p1) and j < len(p2):

    if p1[i] == p2[j]:

        answer.append(p1[i])

        i ← i + 1

        j ← j + 1

    else if p1[i] < p2[j]:

        i ← i + 1

    else:

        j ← j + 1

return answer
```

Complexity: O(m + n) comparisons in worst case. Works best when lists are of similar length.

Variants / optimizations

- If one list (say p_short) is much shorter than the other, for each element of p_short do a binary search in the longer list: cost O(|short| · log|long|). This can be faster than O(m+n) when |short| ≪ |long|.
- Use **skip pointers** in long postings to jump ahead in larger steps — reduces comparisons when elements are far apart.

Worked small example
p1 = [2, 6, 7, 15], p2 = [1, 2, 7, 10]

- Compare 2 vs 1 → 2 > 1 → advance p2
- Compare 2 vs 2 → equal → add 2; advance both
- Compare 6 vs 7 → 6 < 7 → advance p1
- Compare 7 vs 7 → equal → add 7; advance both
- Done. Answer = [2,7].

Binary-search variant pseudocode

```
for each doc d in p_short:

    if binary_search(p_long, d) is true:

        answer.append(d)
```

Complexity: O(|short| · log |long|).

Ques 4: Given the 4 documents — create (a) term–document incidence matrix and (b) inverted index.

Documents:

- Doc 1: `breakthrough drug for schizophrenia`
- Doc 2: `new schizophrenia drug`
- Doc 3: `new approach for treatment of schizophrenia`
- Doc 4: `new hopes for schizophrenia patients`

(I'll treat tokens as lowercased single words, and keep small function words like **for**, **of** as terms unless asked to remove stopwords.)

Step 1 — Unique terms (sorted)

Let's list unique terms found across the four docs:

1. approach
2. breakthrough
3. drug
4. for
5. hopes
6. new
7. of
8. patients
9. schizophrenia
10. treatment

(a) Term–document incidence matrix

Rows = terms (above order). Columns = Doc1, Doc2, Doc3, Doc4. Put 1 if term appears in document.

Term \ Doc	Doc 1	Doc 2	Doc 3	Doc 4
approach	0	0	1	0
breakthrough	1	0	0	0
drug	1	1	0	0
for	1	0	1	1
hopes	0	0	0	1
new	0	1	1	1
of	0	0	1	0
patients	0	0	0	1
schizophrenia	1	1	1	1
treatment	0	0	1	0

(You can reorder terms any way — above is alphabetical. If you remove stopwords like *for* / *of*, drop those rows accordingly.)

(b) Inverted index (term → posting list of docIDs)

Sorted docIDs in lists:

- approach: {3}
- breakthrough: {1}

- drug: {1, 2}
- for: {1, 3, 4}
- hopes: {4}
- new: {2, 3, 4}
- of: {3}
- patients: {4}
- schizophrenia: {1, 2, 3, 4}
- treatment: {3}

(If you include term frequencies or positions, you'd store `(docID, tf)` or `(docID, [positions])` in the postings.)

Ques 5: Describe the following key terms used in Information Retrieval

(a) Tokenization

Definition: Breaking raw text into basic units (tokens) — typically words, numbers, punctuation tokens or multiword expressions.

Issues & choices:

- Whitespace splitting vs language-specific rules.
- Handling punctuation, hyphens, apostrophes (e.g., "don't"), URLs, email addresses, dates, numbers, contractions.
- Unicode normalization for diacritics.
- Multiword tokens (named entities, phrases) sometimes kept as single token ("New York").
- Tokenization affects indexing and matching, so it must match query processing.

Example:
Text: `"Dr. Smith's e-mail: dr.smith@example.com — costs $10.50."`
Tokens might be: `["dr", "smith", "s", "e-mail", "dr.smith@example.com", "costs", "10.50"]` — choices vary with tokenizer.

(b) Stop words

Definition: Very common words (a, the, of, in, to, etc.) often removed from the index because they offer little discriminatory power.

Pros:

- Smaller index, faster searches.
- Less noisy matching.

Cons:

- May break phrase queries ("to be or not to be") and cause incorrect results.
- In some domains (legal, biomedical), common words may be important.

Modern practice: For many systems, minimal or no stopword removal is done because storage is cheaper and phrase matching requires keeping stopwords; instead, treat them specially in ranking.

(c) Normalization

Definition: Converting tokens to a canonical form — includes lowercasing, removing diacritics, mapping punctuation, canonical Unicode normalization, expanding contractions, canonicalizing numbers (e.g., "ten" ↔ "10"), and mapping synonyms/abbreviations.

Purpose: Reduce sparsity of lexical variants so matches succeed (e.g., "Color" vs "colour").

Example:
Normalization steps: `"USA"` → `"usa"`, `"Café"` → `"cafe"`, `"e-mail"` → `"email"`.

(d) Stemming and Lemmatization

Both reduce inflectional variants to a canonical form.

Stemming

- Heuristic rules that chop word endings (e.g., Porter stemmer).
- Fast, language-specific rules; result may not be a valid word ("comput" from "computing").
- Good for recall at cost of occasional precision loss.

Example:

- `running → run` (Porter), `studies → studi` (not always pretty).

Lemmatization

- Use vocabulary and morphological analysis (and usually POS) to map word to its dictionary base form (lemma).
- Requires more linguistic resources (POS tagger, wordnet-like resources).
- Produces valid words: `better → good` (with POS info).

Example:

- `was` → lemma `be`; `children` → lemma `child`.

When to use which?

- Stemming is lighter-weight and widely used when speed & simplicity required.
- Lemmatization is preferred where correctness is important (NLP pipelines, when semantics matter).

Ques 6: Two-word query — postings lists:

- Long postings list (16 entries):
  `[4,6,10,12,14,16,18,20,22,32,47,81,120,122,157,180]`
- Short postings list (1 entry): `[47]`

Work out comparisons for intersecting them with:

a) Using standard postings lists (merge algorithm)

We'll simulate the two-pointer merge, always comparing the current element of the long list with 47 (the only element of the short list), starting at the beginning of the long list:

Long list values in order: 4, 6, 10, 12, 14, 16, 18, 20, 22, 32, **47**, 81, ...

We compare until we reach 47.

Comparisons (value vs 47):

1. compare 4 vs 47 → advance long
2. compare 6 vs 47 → advance
3. compare 10 vs 47 → advance
4. compare 12 vs 47 → advance
5. compare 14 vs 47 → advance
6. compare 16 vs 47 → advance
7. compare 18 vs 47 → advance
8. compare 20 vs 47 → advance
9. compare 22 vs 47 → advance
10. compare 32 vs 47 → advance
11. compare 47 vs 47 → match; finish

Total comparisons = 11.

(After matching, both pointers would advance; since the short list is exhausted, intersection ends.)

Justification: The merge algorithm compares each element of the long list up to the match once; with the match at the 11th position, we needed 11 comparisons.

b) Using postings lists with skip pointers, skip length = √P

- Here P = length of the long postings list = 16. √P = 4 → skip every 4 entries.
- Skip pointers allow jumping from index i to i+4 (if available) and comparing the *skip target's* docID to 47 to decide whether to skip.

Simulate (showing comparisons between a long-list value or a skip-target value and 47):

1. Compare `p1[1]=4` with 47 → (4 < 47). Check skip target `p1[5]=14`.
2. Compare skip value `14` with 47 → (14 < 47) → jump to index 5.
3. At index 5 (`14`), check skip target `p1[9]=22`. Compare `22` with 47 → (22 < 47) → jump to index 9.
4. At index 9 (`22`), check skip target `p1[13]=120`. Compare `120` with 47 → (120 > 47) → **cannot** skip to index 13; instead scan linearly:
5. Compare `p1[10]=32` with 47 → (32 < 47).
6. Compare `p1[11]=47` with 47 → match.

Total comparisons = 6.

General note: Using skip pointers reduces the number of element-to-query comparisons by allowing block-wise jumps; ideal skip length often √P giving expected cost ≈ 2√P in balanced cases. But exact cost depends on the target's position.

Ques 7: Explain Bi-word indexing and Positional indexing with a suitable example.

Bi-word indexing (bi-gram / bigram of adjacent words)

Definition: Index every *adjacent pair of words* (bi-word) as a single token. The bi-word index maps each adjacent pair → list of documents where that pair occurs.

How it answers phrase queries:

- A phrase query with words `w1 w2 ... wk` can be answered by looking up bi-words `w1 w2`, `w2 w3`, ..., `w(k-1) wk` and intersecting their posting lists.
- Since bi-words capture adjacency, intersecting the bi-word postings ensures the phrase occurs exactly (assuming no gaps).

Example (using Doc2 from earlier)
Doc2: `new schizophrenia drug`

Bigrams:

- `new_schizophrenia` → doc2
- `schizophrenia_drug` → doc2

To answer phrase query `"new schizophrenia drug"`:

- Lookup `new_schizophrenia` → docs {2,3,4?} (only doc2 actually for that bigram)
- Lookup `schizophrenia_drug` → docs {2}
- Intersect → {2} → doc2 contains the exact phrase.

Pros:

- Simple and fast for exact phrase queries.
- No positional lists needed for basic adjacency.

Cons:

- Index size grows (many bigrams).
- Cannot handle queries with intervening words or proximity queries easily.
- If phrase length large, intersecting many bigram lists can be costly.
- Fails on queries that require matching across sentence boundaries or non-contiguous phrases.

Positional indexing

Definition: For each term, the postings list stores not only docIDs but also the *positions* (word offsets) where the term appears in each document.

Format (per term): term → list of `(docID, [pos1, pos2, ...])`

How to answer a phrase query (e.g., `"fools rush in"`):

- Fetch positional lists for `fools`, `rush`, `in`.
- For each doc that appears in all three lists, check if there exists a position p for `fools` such that `rush` occurs at p+1 and `in` occurs at p+2. That is, check offsets with appropriate offsets.

Example using a small made-up doc
DocX: `"the fools rush in to the garden"`

Positions (0-based or 1-based consistently). Suppose 1-based:

- `fools`: [2]

- `rush`: [3]
- `in`: [4]

Check: `2+1=3` matches `rush`, and `2+2=4` matches `in` → phrase found.

Pros:

- Handles exact phrase queries and arbitrary proximity queries (e.g., within k words).
- More general and accurate than bi-word indexing.

Cons:

- More storage (positions stored).
- Slightly slower or larger index, although compression mitigates this.

Which to choose?

- For general-purpose IR supporting phrases and proximity, **positional indexing** is the standard choice. Bi-word indexing is a lighter-weight alternative when phrase queries are common and index size is less of a concern.

Ques 8: Given positional index snippet — which document(s) match phrase queries?

You provided a positional index excerpt (terms with doc: positions):

(abridged)

```
angels: 2: <36,174,252,651>; 4: <12,22,102,432>; 7: <17>;
fools:  2: <1,17,74,222>; 4: <8,78,108,458>; 7: <3,13,23,193>;
fear:   2: <87,704,722,901>; 4: <13,43,113,433>; 7: <18,328,528>;
in:     2: <3,37,76,444,851>; 4: <10,20,110,470,500>; 7: <5,15,25,195>;
rush:   2: <2,66,194,321,702>; 4: <9,69,149,429,569>; 7: <4,14,404>;
to:     2: <47,86,234,999>; 4: <14,24,774,944>; 7: <199,319,599,709>;
tread:  2: <57,94,333>; 4: <15,35,155>; 7: <20,320>;
where:  2: <67,124,393,1001>; 4: <11,41,101,421,431>; 7: <16,36,736>;
```

We test the phrase queries:

a. "fools rush in"

We need documents where `fools` at position p, `rush` at p+1, and `in` at p+2.

Check **Doc 2**:

- `fools`: 1,17,74,222
- `rush`: 2,66,194,321,702
- `in`: 3,37,76,444,851
  Look for p such that p+1 is in `rush` and p+2 in `in`:
- p = 1 → p+1 = 2 ∈ rush, p+2 = 3 ∈ in → **match in Doc 2**.

Check **Doc 4**:

- `fools`: 8,78,108,458
- `rush`: 9,69,149,429,569
- `in`: 10,20,110,470,500

- p = 8 → 9 ∈ rush, 10 ∈ in → **match in Doc 4**.

Check **Doc 7**:

- `fools`: 3,13,23,193
- `rush`: 4,14,404
- `in`: 5,15,25,195
- p = 3 → 4 ∈ rush, 5 ∈ in → **match in Doc 7**.

Answer (a): Documents **2, 4, and 7** match the phrase `"fools rush in"`.

b. "fools rush in" AND "angels fear to tread"

We already know which docs match the first phrase: {2, 4, 7}. Now check which of these also contain the second phrase `"angels fear to tread"` — requires `angels` at p, `fear` at p+1, `to` at p+2, `tread` at p+3.

Check **Doc 2**:

- `angels`: 36,174,252,651
- `fear`: 87,704,722,901
- `to`: 47,86,234,999
- `tread`: 57,94,333
  We need p so that p+1 ∈ fear, p+2 ∈ to, p+3 ∈ tread. None of the `angels` positions (36,174,252,651) are followed by such immediate consecutive positions in the other lists (e.g., angels=36 → fear would need 37 but fear has 87 etc.). **No match in Doc 2**.

Check **Doc 4**:

- `angels`: 12,22,102,432
- `fear`: 13,43,113,433
- `to`: 14,24,774,944
- `tread`: 15,35,155
  Look at p = 12 → 13 ∈ fear, 14 ∈ to, 15 ∈ tread → **match in Doc 4**.

Check **Doc 7**:

- `angels`: 17
- `fear`: 18,328,528
- `to`: 199,319,599,709
- `tread`: 20,320
  For p = 17 → fear has 18 (OK), to would need 19 but `to` does not have 19 (it has 199,319,...). So **no match in Doc 7**.

Answer (b): Only **Doc 4** contains both phrases `"fools rush in"` and `"angels fear to tread"`.

(Method note: phrase queries on positional indexes are answered by merging the postings for the phrase terms and checking offsets for consecutive positions — exactly what we did.)

Ques 9: What are wildcard queries? Explain spelling corrections and Edit distance in it with an example.

Wildcard queries

- A **wildcard query** uses special wildcard characters (e.g., `*`, `?`) to match multiple possible strings. Common forms:
  - `compu*` → `computer`, `computing`, `computation`
  - `te?t` → `test`, `text` (where `?` matches a single character)
  - `*ology` → `biology`, `technology` (if leading `*` supported)

Implementation approaches

1. **Query-time expansion**: enumerate all index terms matching the wildcard pattern (using a lexicon/trie) then OR their postings.
2. **Index-time k-gram (n-gram) index**: build an index of k-character substrings (`$comput` → `^c`, etc.). To answer `compu*` find all terms that contain `compu` as prefix by intersecting k-gram candidates.
3. **Permuterm index**: rotate terms with special end-marker for handling leading wildcards; allows match of `*ology` by transforming queries into prefix lookups on rotated terms.
4. **Tries / prefix trees**: fast prefix wildcard lookups.

Trade-offs: wildcard queries can be expensive (many candidate expansions and large postings unions); k-gram or permuterm indexes precompute substrings to speed this up at cost of index size.

Spelling correction & edit distance

Goal: Map a possibly misspelled query term to the intended correct term(s) to improve retrieval.

Edit distance (Levenshtein distance)

- Minimum number of single-character operations (insertions, deletions, substitutions) required to transform one string into another.
- Classic dynamic programming algorithm computes distance in O(mn) time where m and n are the string lengths.

Example

- Misspelled word: `"speling"`
- Correct target: `"spelling"`

Compute edit operations: insert one `'l'` between `l` and `i` → 1 insertion → **edit distance = 1**.

Another classic example: `"kitten"` → `"sitting"` has distance 3 (substitute `k→s`, `e→i`, insert `g`).

How to use edit distance for spelling correction

1. **Candidate generation**
   - Generate all dictionary terms within edit distance ≤ k (k usually 1 or 2). Generating all candidates naïvely is expensive.
   - Use **k-gram index**, **BK-tree** (metric tree for strings using edit distance), or **hashing techniques** to retrieve close candidates quickly.
2. **Candidate ranking**
   - Use noisy-channel model: choose candidate `c` maximizing $P(c) \cdot P(q \mid c)$ where $P(c)$ is prior (word frequency) and $P(q \mid c)$ the error model (related to edit distance and specific error probabilities).

    - Simpler: choose lowest edit distance; break ties by corpus frequency.

Example pipeline

- Query: `"recieve"`
- Candidate generation yields `"receive"`, `"recipe"`, etc.
- Edit distance for `"receive"` is 1 (swap `ie` to `ei` can be considered substitution/transposition in extended model). Choose `"receive"` if it has higher corpus frequency.

Extensions

- **Damerau–Levenshtein** includes transpositions (important for common typos).
- **Phonetic algorithms** (Soundex, Metaphone) help with errors that preserve pronunciation.
- **Context-aware corrections**: use surrounding query words or language models to prefer `"New York"` vs `"new yrok"` → correct to `"new york"` based on bigram frequency.

Ques 10: Define Index construction and define the following terms

Index construction — definition

Index construction is the process of transforming the document collection into an index (usually an inverted index) that supports efficient retrieval. Steps include tokenization, normalization, generating term-document pairs (or term-document-position), sorting/merging, compressing, and writing index files to disk. Index construction must consider memory constraints, disk I/O efficiency, and support for incremental updates.

Now the specific methods:

a. Blocked sort-based indexing (BSBI)

Idea:

- Break the collection into blocks that fit into memory.
- For each block:
    1. Parse documents and produce list of (termID, docID) pairs.
    2. Sort those pairs in memory (typically by termID then docID).
    3. Write sorted block to disk as a partial inverted index.
- After processing all blocks, **merge** sorted block files in a multi-way merge to produce the final global inverted index.

Properties

- Disk I/O efficient because we write a set of sorted runs then merge.
- Sorting cost dominates; memory limited by the block size.
- Good for large collections where entire index doesn't fit in memory.

Complexity

- Sorting each block O(B log B), and merging K blocks using a K-way merge (using heap) efficient; total cost dominated by disk I/O.

b. Single-pass in-memory indexing (SPIMI)

Idea:

- Process documents in a streaming way and build in-memory dictionary → postings incrementally.
- When memory threshold reached, write out the current dictionary + postings as a block (term-sorted), then clear memory and continue.
- Blocks written are *already in inverted list form* (term → postings) so you do not sort all (term, doc) pairs first — this saves memory & CPU.

Advantages over BSBI

- Lower memory overhead for temporary arrays of term-doc pairs.
- Faster in practice because it avoids building huge arrays to sort; uses hash table to accumulate postings.

SPIMI algorithm (sketched):

- For each token in stream:
    - Insert token into in-memory hash map mapping term → postings list.
- When memory full:
    - Sort terms, write block (term → postings) to disk.
    - Clear in-memory map and continue.

Use: Widely used in modern indexers due to simplicity and performance.

c. Distributed indexing

Idea:

- Use parallel/distributed storage & computation to scale indexing across many machines (MapReduce/Hadoop or other distributed frameworks).

Typical MapReduce pattern

- **Map**: Each mapper reads a subset of docs and outputs (term, docID) pairs.
- **Shuffle/Sort**: Key-based redistribution groups the same term's pairs to the same reducers.
- **Reduce**: Aggregates postings for each term into a single postings list (possibly compressed), writes index shards.

Why distributed?

- Large web-scale collections require many machines for parsing, sorting, and storing huge indexes.
- Enables fault tolerance and parallelization.
- Also used to create *sharded* search architectures: each shard is a subset of docs and has its own index.

Extra details

- Postings can be partitioned by term hashing or document partitioning.
- Requires careful merging, load balancing, and fault tolerance.

d. Dynamic indexing

Definition: Techniques to allow updates (additions/deletions/edits) to an index without rebuilding the entire index from scratch.

Approaches

1. **In-memory delta index + periodic merge**
    - Maintain a small in-memory index for recent documents (fast inserts).
    - Periodically merge the delta (small) index into the main on-disk index (batch merge).
2. **Log-structured Merge Trees (LSM-tree)**
    - Maintain series of components (C0, C1, C2...) with increasing sizes; writes are appended to C0 then merged downstream.
3. **Document-level deletion markers**
    - Mark documents deleted logically (tombstones); garbage collect during merges.
4. **Incremental merging / real-time indexes**
    - Keep a set of indexes (main + incremental) and search across them at query time; merge when needed.

Trade-offs

- Fast updates vs search efficiency. Searching multiple segments increases query-time cost, but merges amortize it.
- Complexity in maintaining skip pointers, positions, compressed postings across merges.