Retrieval Models (2)

- SI 650 / EECS 549 Information Retrieval
- September 10, 2025

Some slides based on slide deck by Dr. Jurgens

Today's plan

- Building IR for Scalability
- Thinking about what is relevance
  - New relevance functions
- Axiomatic Approaches to define good retrieval function

Building IR for Scalability

Indexing

- Indexing = Convert documents to data structures that enable fast search
- Inverted index is the dominating indexing method (used by all search engines)

Inverted index - example

Doc 1

This is a sample document with one sample sentence

Doc 2

This is another sample document

Dictionary

Term	# docs	Total freq
This	2	2
is	2	2
sample	2	3
another	1	1
…	…	…

Postings

Doc id	Freq	Pos
1	1	1
2	1	1
1	1	2
2	1	2
1	2	4, 8
2	1	4
2	1	3
…	…	…
…	…	…

- From ChengXiang Zhai's slides

Scalability Issues: Number of Postings

An Example: Reuters RCV1 Collection

- Number of docs = m = 800,000
  - Average tokens per doc: 200
- Number of distinct terms = n = 400K
- 160 million (non-positional) postings in the inverted index

Bottleneck(数据量大，慢）

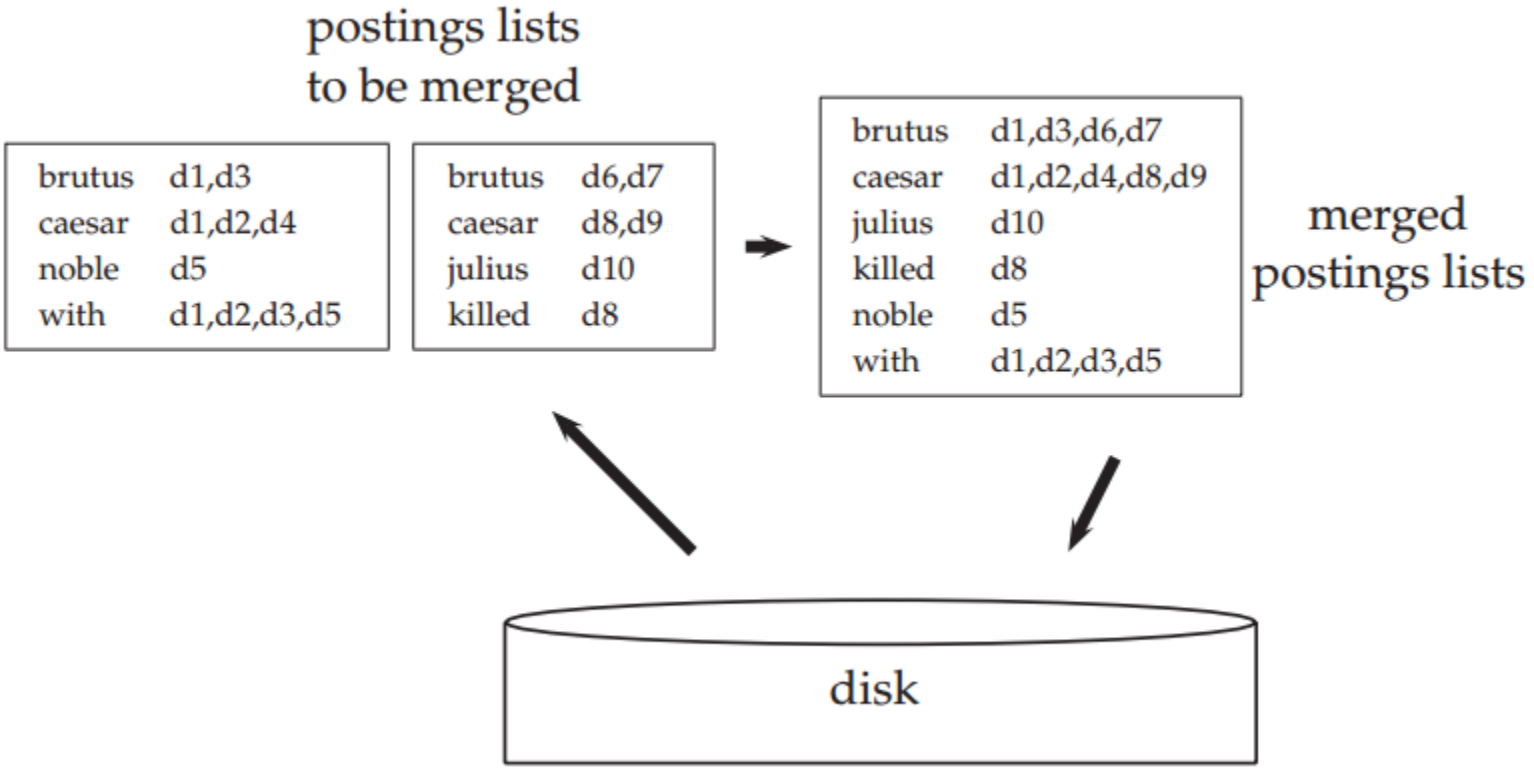
- Parse and build postings entries one doc at a time
- Sort postings entries by term (then by doc within each term)
- Doing this with random disk seeks would be too slow – must sort N=160M records

If every comparison took 2 disk seeks (10 milliseconds each), and N items could be sorted with N $\log_2$N comparisons, how long would this take?

Sorting with fewer disk seeks

- 12-byte (4+4+4) records *(term, doc, freq)*
- These are generated as we parse docs
- Must now sort these 12-byte records by *term*
- Define a Block (e.g., ~ 10M) records
  - Blocks defined such that each block can fit in memory
- Sort within blocks first (in memory) and write to disk, then merge the blocks into one long sorted order.
- Blocked Sort-Based Indexing (BSBI)

Blocked Sort-Based Indexing - Example



BSBI Example with two blocks: The two blocks ("postings lists to be merged") are loaded from disk into memory, merged in memory ("merged postings lists") and written back to disk. Terms are shown instead of termIDs for better readability.

Problem with Sort-Based Algorithm

- Assumption: we can keep the dictionary in memory.
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.
- Actually, we could work with term, docID postings instead of termID, docID postings . . .
  - What could go wrong here?
- . . . but then intermediate files become very large. (We would end up with a very slow index construction method.)
- 需要全局字典（dictionary）在内存中，但当词汇量很大时放不下。
- 必须生成 (termID, docID, freq) 记录再排序，中间文件体积巨大，磁盘 I/O 成本高。

SPIMI:
Single-pass in-memory indexing

- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.
- Key idea 2: Don't sort. Accumulate postings in postings lists as they occur.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.

SPIMI-Invert

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6        then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

Distributed indexing

- For web-scale indexing
  - must use a distributed computing cluster
- Individual machines are fault-prone
  - Can unpredictably slow down or fail
- How do we exploit such a pool of machines?
  - Maintain a master machine directing the indexing job – considered “safe”.
  - Break up indexing into sets of (parallel) tasks.
  - Master machine assigns each task to an idle machine from a pool.

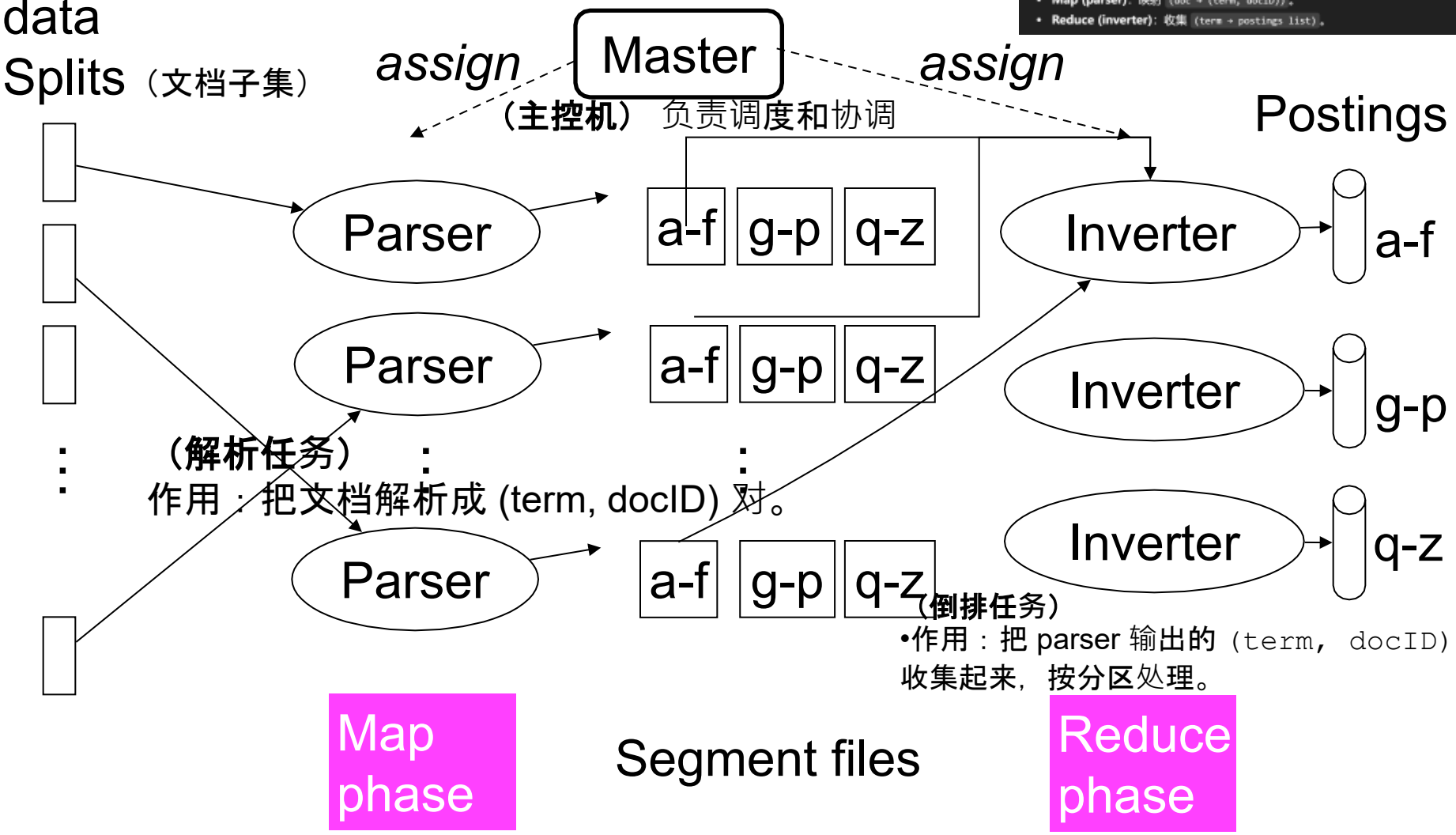
Parallel tasks

- Use two sets of parallel tasks
  - Parsers: Break the document in to (term, docID) tuples
  - Inverters: Turn the (term, docID) tuples into an inverted index
- Break the input document corpus into splits
  - Each split is a subset of documents
  - E.g., corresponding to blocks in BSBI
- Master assigns a split to an idle parser machine
- Parser reads a document at a time and emits (term, doc) pairs
  - writes pairs into j partitions
  - Each partition is for a range z) – here j = 3.
- Inverter collects all (term, doc) p o postings list

Data flow

4. 数据流过程（简化版）
1. 输入文档 → 按块（split）分配给多个 parser。
2. Parser：逐文档解析，输出 (term, docID)，写到分区文件（按 term 范围分组）。
3. Inverter：每个 inverter 负责一个分区（比如 a-f 的所有 terms），
   - 收集对应的 (term, docID) 对，
   - 排序并构建 postings list。
4. 最终合并所有分区，得到全局倒排索引。

👉 这个过程很像 MapReduce：
- Map (parser)：映射 (doc → (term, docID))。
- Reduce (inverter)：收集 (term → postings list)。



Dynamic indexing

- Problem with keeping a static index:
  - Docs come in over time
    - postings updates for terms already in dictionary
    - new terms added to dictionary
  - Docs get deleted

- Simplest Approach
  - Maintain a “big” main index
  - New docs go into a “small” auxiliary index
  - Search across both, merge results
  - Deletions
    - Invalidation bit-vector for deleted docs
    - Filter docs output on a search result by this invalidation bit-vector
  - Periodically, re-index into one main index

Index on disk vs. memory

- Most retrieval systems keep the dictionary in memory and the postings on disk
- Web search engines frequently keep both in memory
  - massive memory requirement
  - feasible for large web service installations
  - less so for commercial usage where query loads are lighter

Retrieval From Indexes

- Given the large indexes in IR applications, searching for keys in the dictionaries becomes a dominant cost
- Two main choices for dictionary data structures: Hashtables or Trees
  - Using Hashing
    - requires the derivation of a hash function mapping terms to locations
    - may require collision detection and resolution for non-unique hash values
  - Using Trees
    - Binary search trees
    - nice properties, easy to implement, and effective
    - enhancements such as B+ trees can improve search effectiveness
    - but, requires the storage of keys in each internal node

Hashtables for Storing Indices

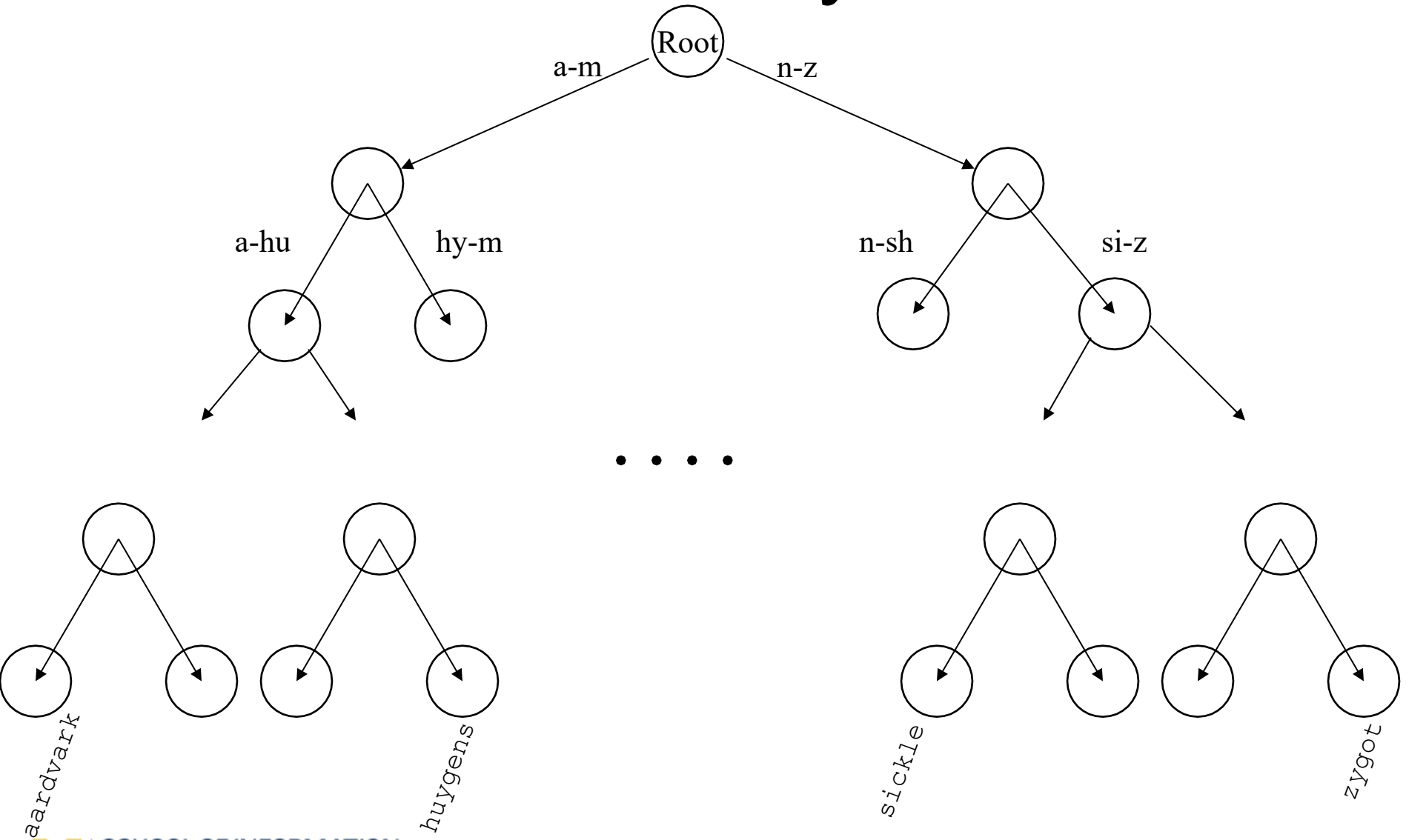
（index的复数）

- Each vocabulary term is hashed to an integer
  - (We assume you've seen hashtables before)
- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing everything

Trees for Storing Indices

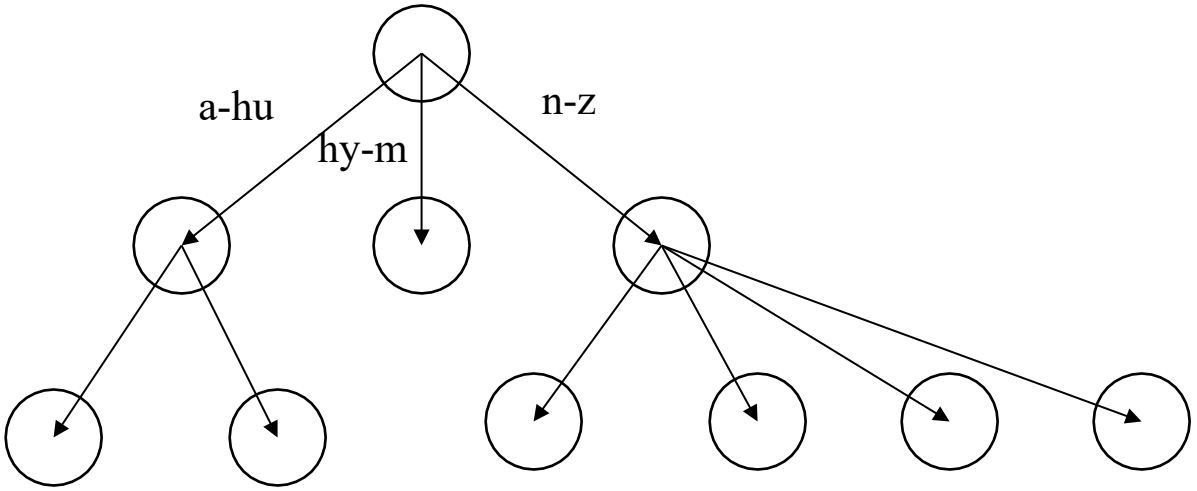
- Simplest: binary tree
- More usual: B-trees
- Trees require a standard ordering of characters and hence strings … but we typically have one
- Pros:
  - Solves the prefix problem (e.g., terms starting with hyp)
- Cons:
  - Slower: O(log M) [and this requires balanced tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

Tree: binary tree



Tree: B-tree

- Definition: Every internal node has a number of children in the interval [a,b] where a, b are appropriate natural numbers, e.g., [2,4].



Phrase based Queries

Phrase queries

- We want to answer a query such as [university of michigan] – as a phrase.
- The concept of phrase query has proven easily understood by users.
- About 10% of web queries are phrase queries.
- Consequence for inverted index: it no longer suffices to store docIDs in postings lists for terms.
- Two ways of extending the inverted index:
  - biword index
  - positional index

Biword indexes

- Index every consecutive pair of terms in the text as a phrase.
- For example, ”university of michigan” would generate two biwords: “university of” and “of michigan”
- Each of these biwords is now a vocabulary term! How big is your vocab now? （n^2）
- Two-word phrases can now easily be answered.

Longer phrase queries

- A long phrase like “university michigan ann arbor” can be represented as the Boolean query “university michigan” AND “michigan ann” AND “ann arbor”
- Does this always guarantee the correct match?
  - We need to do post-filtering of hits to identify subset that actually contains the 4-word phrase.
- What about phrases like, “university of michigan”?

Extended biwords

- Parse each document and perform part-of-speech tagging
- Bucket the terms into (say) nouns (N) and articles/prepositions (X)
- Now deem any string of terms of the form NX*N to be an extended biword
- Examples: catcher in the rye
  N X X N
  Queen of Denmark
  N X N
- Include extended biwords in the term vocabulary
- Queries are processed accordingly

Issues with biword indexes

- Why are biword indexes rarely used?
  - False positives, as noted earlier
  - Index blowup in size due to very large term vocabulary
- What can be an alternative?

Positional indexes

每个 postings 不仅包含 docID，还包含该词在文档中出现的 **位置列表**

- Positional indexes are a more efficient alternative to biword indexes.
- Postings lists in a nonpositional index: each posting is just a docID
- Postings lists in a positional index: each posting is a docID and a list of positions

Positional indexes: Example

通过相对位置判断

Query: "to$_1$ be$_2$ or$_3$ not$_4$ to$_5$ be$_6$"

TO, 993427:

‹ 1: ‹7, 18, 33, 72, 86, 231›;
2: ‹1, 17, 74, 222, 255›;
4: ‹8, 16, 190, 429, 433›;
5: ‹363, 367›;
7: ‹13, 23, 191›; . . . ›

BE, 178239:

‹ 1: ‹17, 25›;
4: ‹17, 191, 291, 430, 434›;
5: ‹14, 19, 101›; . . . ›

OR, 693228:

‹ 1: ‹238, 301›;
2: ‹5, 23, 79, 200, 230›;
4: ‹8, 16, 190, 431, 433›;
5: ‹363, 367›;
8: ‹13, 23, 191›; . . . ›

NOT, 602034:

‹ 1: ‹29,100›;
4: ‹21, 200, 432, 500›;
7: ‹14, 19, 101›; . . . ›

Doc 4 is a match!

Proximity search (临近搜索，两个词相距不超过n)

- We just saw how to use a positional index for phrase searches.
- Can we also use it for proximity search?
- For example: employment /4 place
  - Find all documents that contain EMPLOYMENT and PLACE within 4 words of each other.
- "Employment agencies that place healthcare workers are seeing growth" is a hit.
- "Employment agencies that have learned to adapt now place healthcare workers" is not a hit.

Proximity search

- Use the positional index
- Simplest algorithm: look at cross-product of positions of (i) EMPLOYMENT in document and (ii) PLACE in document
- Very inefficient for frequent words, especially stop words（比如of the and）

Combination scheme

- Biword indexes and positional indexes can be profitably combined.
- Many biwords are extremely frequent: Harry Styles, etc
- For these biwords, increased speed compared to positional postings intersection is substantial.
- Combination scheme: Include frequent biwords as vocabulary terms in the index. Do all other phrases by positional intersection.

1. 背景

- **Biword Index**：把连续的两个词当作一个"词条"建索引，比如 `"Harry Styles"` → `("harry styles")`。
  - 优点：短语查询很快，因为直接查这个 biword 即可。
  - 缺点：索引体积大，且对不常见的短语没有太多价值。
- **Positional Index**：记录每个词在文档中的具体位置，用交叉匹配来判断短语或邻近匹配。
  - 优点：支持任意短语或 proximity 查询。
  - 缺点：计算量大（需要交叉比较很多位置）。

---

2. Combination Scheme 思路

- **核心思想**：取二者的优势。
  - 对**非常高频的短语**（比如 `"Harry Styles"`、地名、人名、常见 bigram）直接建 biword 索引，提高查询速度。
  - 对其他短语，使用 positional index 做交叉匹配，节省空间。

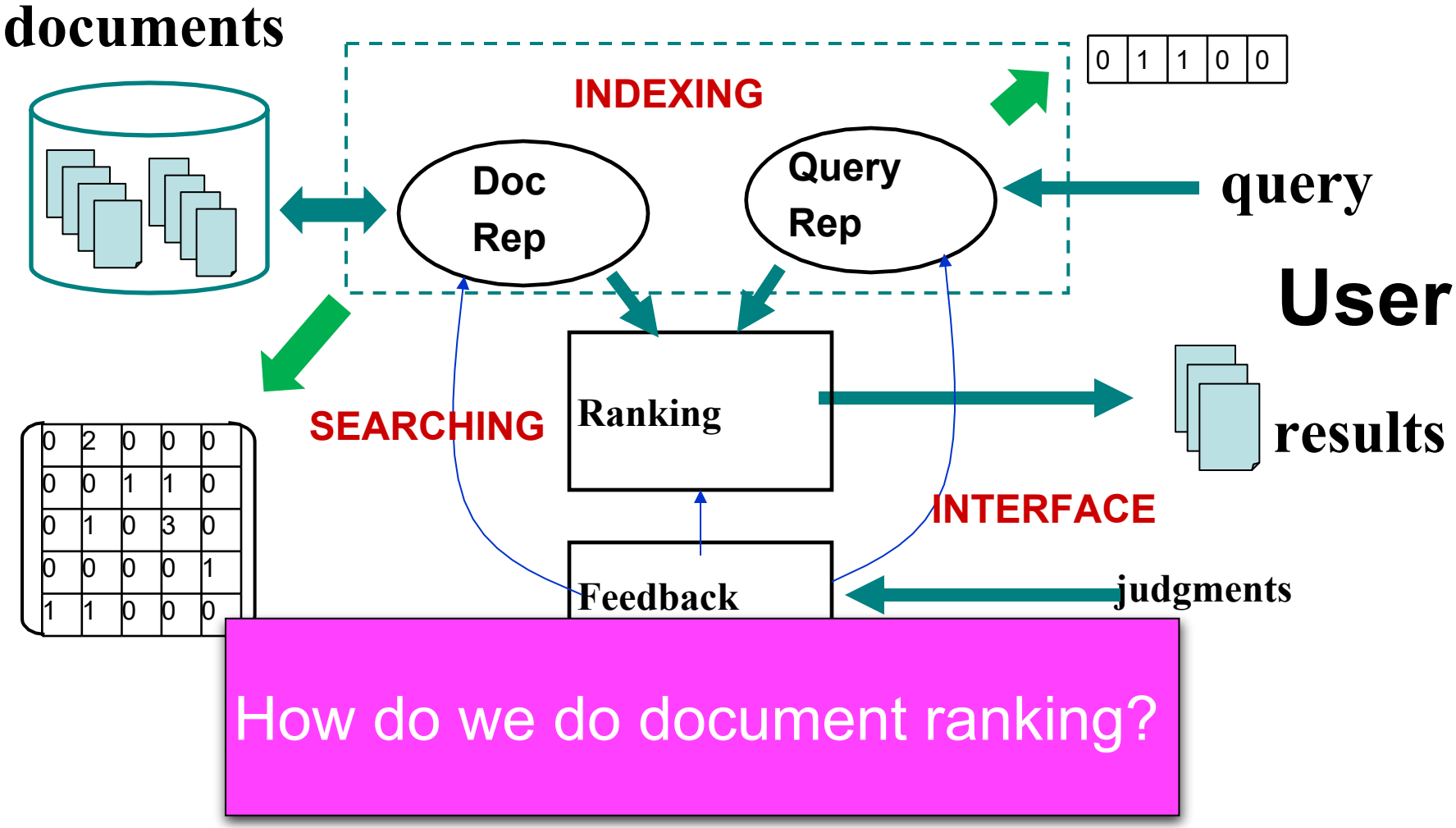
---

3. 好处

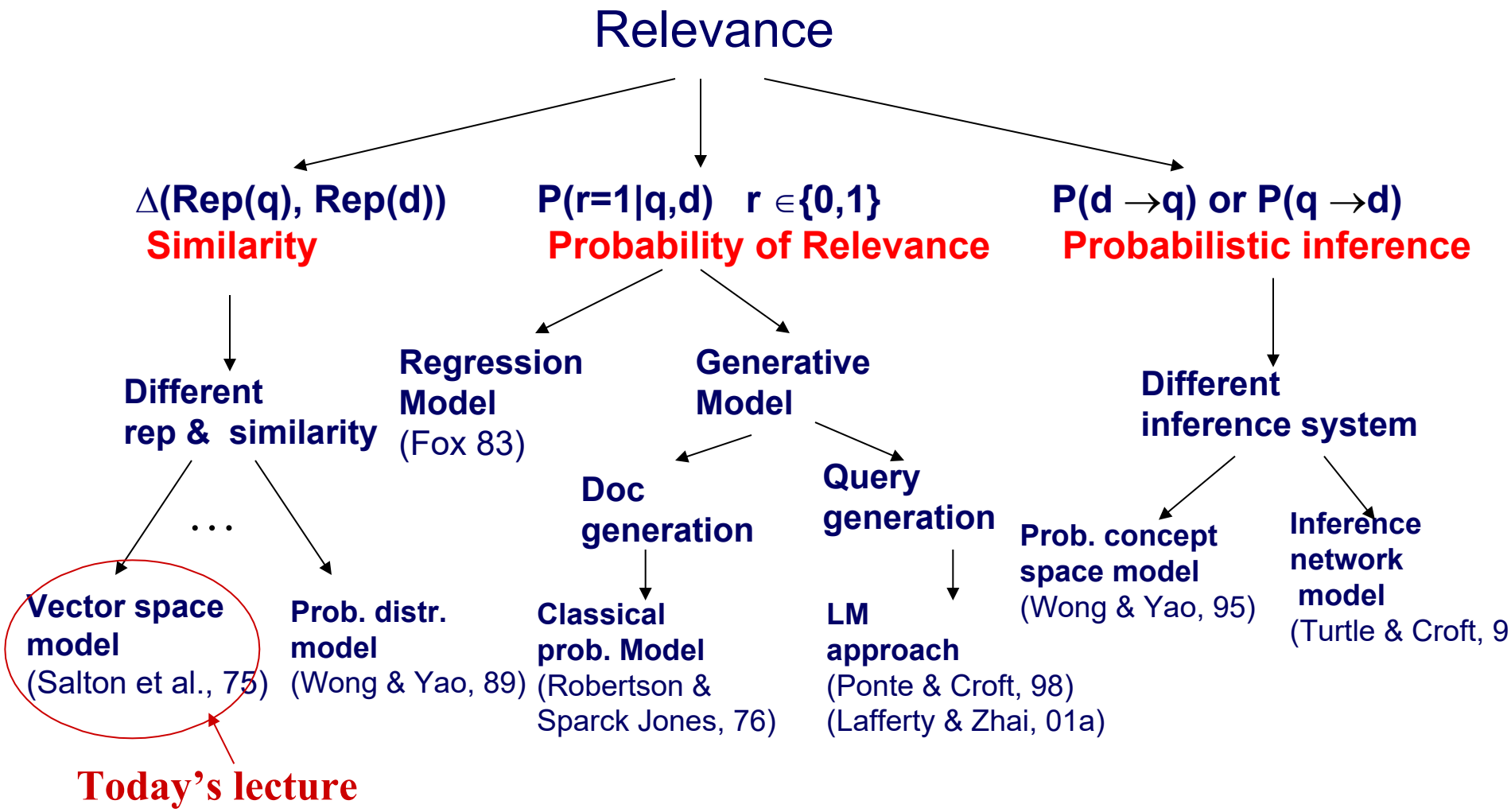
- **查询性能**：常用短语直接查 biword，避免大量位置交叉计算 → 快。
- **索引大小**：只存常用 biwords，控制索引膨胀 → 省空间。
- **灵活性**：仍可处理任意短语查询，因为 fallback 到 positional index。

Textual Relevance

Revisit: Typical IR system architecture



The Notion of Relevance



VSM + TF-IDF Revisited

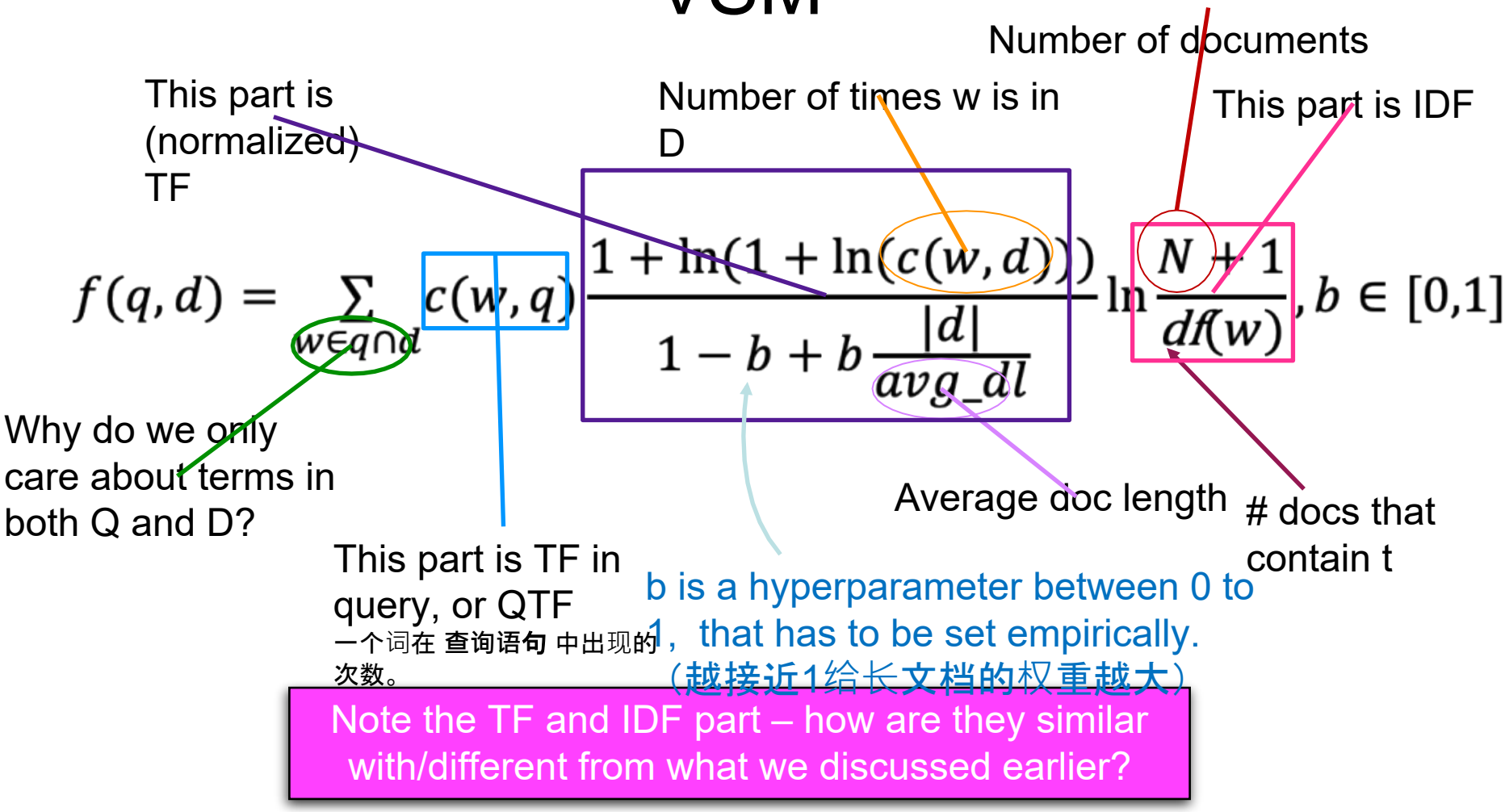
VSM：向量空间模型

- Under VSM, Relevance(d, q) $\propto$ similarity(d, q)
- We covered various ways to compute similarity/distance: Jaccard, Euclidean, and Cosine similarity
- But it is important to set the right weights to compute distance/similarity in ways to increase performance.
- One method: tf-idf”
  - TF (Term Frequency) = Within-doc-frequency
  - IDF (Inverse Document Frequency)

Why do we need to normalize TF?

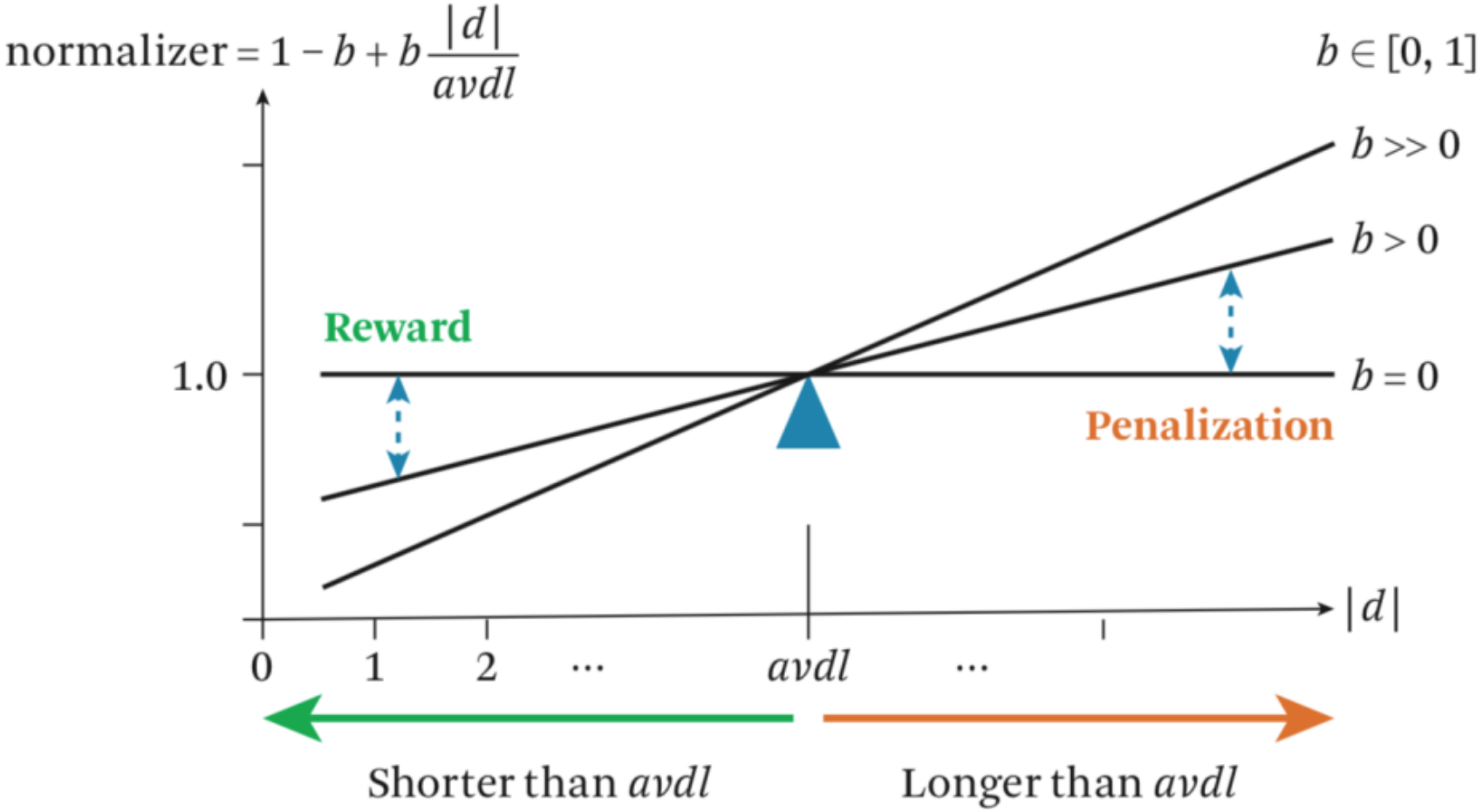
- "Repeated occurrences" are less informative than the "first occurrence"
- Why?
- One reason: Document length variation
- Two views of document length
  - A doc is long because it uses more words
  - A doc is long because it has more contents
- Generally we want to penalize long documents, but avoid over-penalizing
  - E.g. pivoted normalization

Pivoted Normalization – the Typical VSM



Note: the formula introduced in the textbook (Zhai and Massung 2016 page 108) is incorrect.

What's the effect of the normalizer in Pivoted Normalization



Okapi/BM25

$$S(Q,D)=\sum_{t\in Q\cap D}\ln\frac{N-df(t)+0.5}{df(t)+0.5}\cdot\frac{(k1+1)\cdot c(t,D)}{k1(1-b+b\frac{|D|}{avdl})+c(t,D)}\cdot\frac{(k3+1)\cdot c(t,Q)}{k3+c(t,Q)}$$

A variant form of IDF

Variant form of (normalized) TF

Normalized QTF

k1, k3, and b are all parameters that have to be set empirically.

The best performer in TREC, the core feature of Bing. More on this during discussion section

What Works the Best?

tf	is the term's frequency in document
qtf	is the term's frequency in query
N	is the total number of documents in the collection
df	is the number of documents that contain the term
dl	is the document length (in bytes), and
$avdl$	is the average document length

Okapi weighting based document score: [23]

$$\sum_{t \in Q,D} ln \frac{N - df + 0.5}{df + 0.5} \cdot \frac{(k_1 + 1)tf}{(k_1[(1 - b) + b\frac{dl}{avdl}]) + tf} \cdot \frac{(k_3 + 1)qtf}{k_3 + qtf}$$

Error in book

k_1 (between 1.0–2.0), b (usually 0.75), and k_3 (between 0–1000) are constants.

Pivoted normalization weighting based document score: [30]

$$\sum_{t \in Q,D} \frac{1 + ln(1 + ln(tf))}{(1 - s) + s\frac{dl}{avdl}} \cdot qtf \cdot ln \frac{N + 1}{df}$$

s is a constant (usually 0.20).

(Singhal 2001)

- Use single words
- Use stat. phrases
- Remove stop words
- Stemming
- Others(?)

Beyond the Vector Space Model

Advantages of VS Model

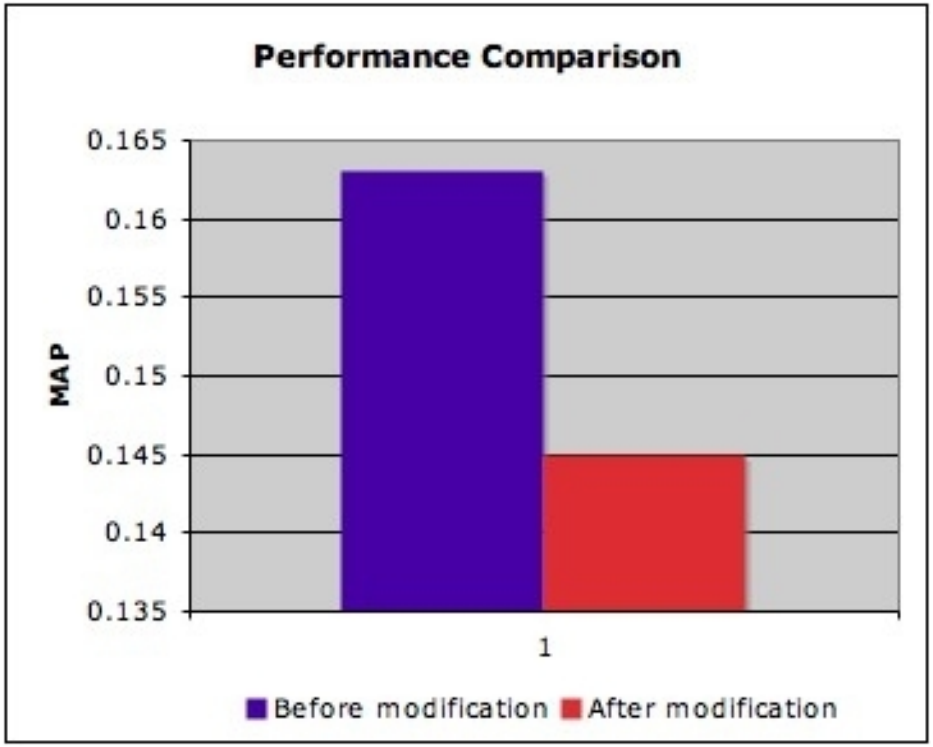
- Empirically effective! (Top TREC performance)
- Intuitive
- Easy to implement
- Well-studied/Most evaluated
- The Smart system
  - Developed at Cornell: 1960-1999
  - Still widely used
- Warning: Many variants of TF-IDF!

Disadvantages of VS Model

- Assume term independence
  - Which is never true!
- Assume query and document to be the same
- Lack of "predictive adequacy"
  - Arbitrary term weighting
  - Arbitrary similarity measure
- Very sensitive to the selection of weighting.
- Lots of parameter tuning!

No Way to Predict Performance

$$S(Q,D) = \sum_{t \in D \cap Q} c(t,Q) \times \log \frac{N+1}{df(t)} \times \frac{1+\log(1+\log(c(t,D)))}{(1-s)+s \times \frac{|D|}{avdl}}$$



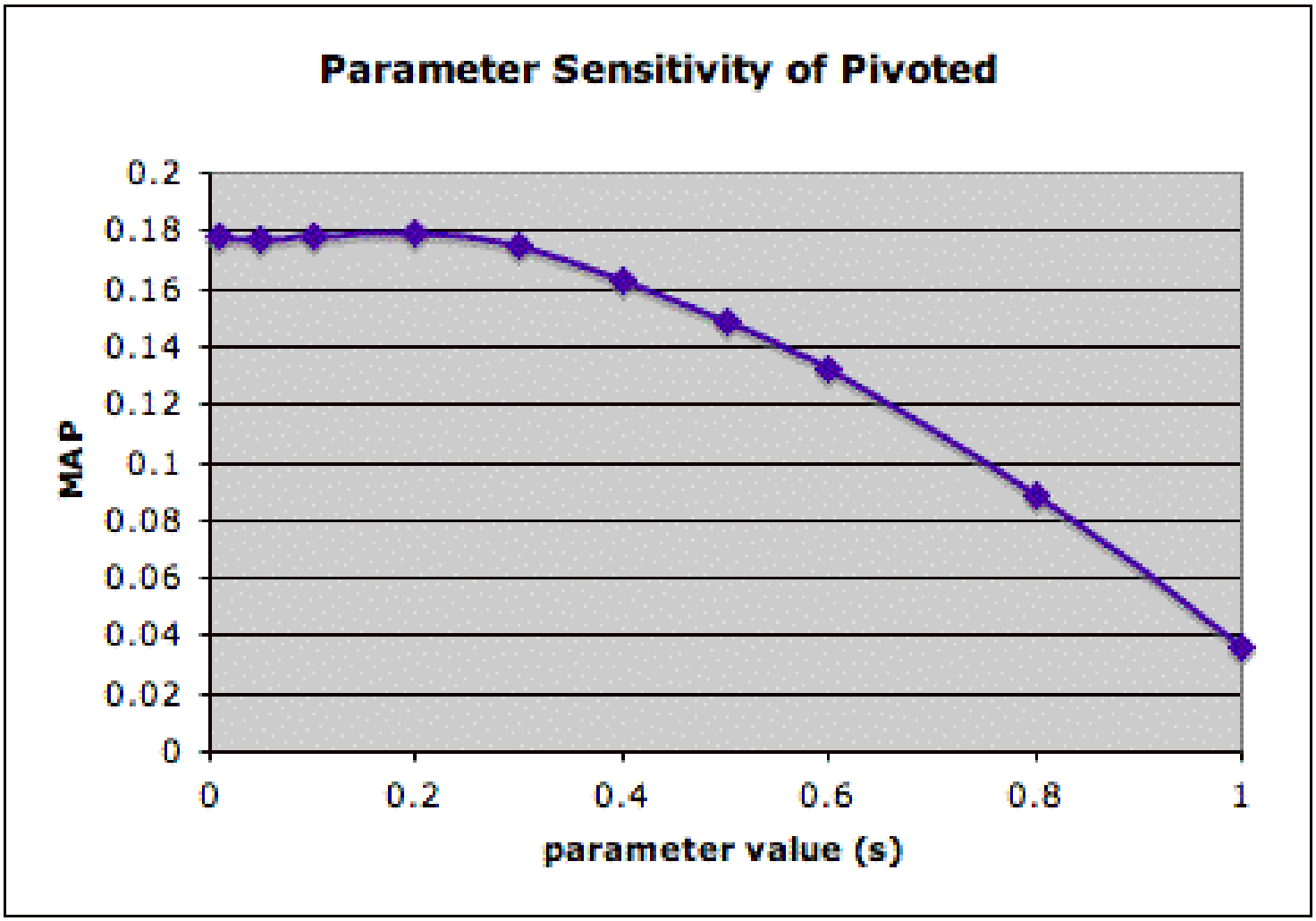
Sophisticated Parameter Tuning

$$S(Q,D) = \sum_{t \in Q \cap D} \log \frac{N - df(t) + 0.5}{df(t)} \cdot \frac{(k_1 + 1) \cdot c(t,D)}{c(t,D) + k_1((1 - b) + b \cdot \frac{|D|}{avdl})} \cdot \frac{(k_3 + 1) \cdot c(t,Q)}{k_3 + c(t,Q)}$$

"k_1, b and k_3 are parameters which depend on the nature of the queries and possibly on the database;
k_1 and b default to 1.2 and 0.75 respectively,
but smaller values of b are sometimes advantageous;
in long queries k_3 is often set to 7 or 1000."

[Robertson et al. 1999]

High Parameter Sensitivity



Measuring Retrieval Quality

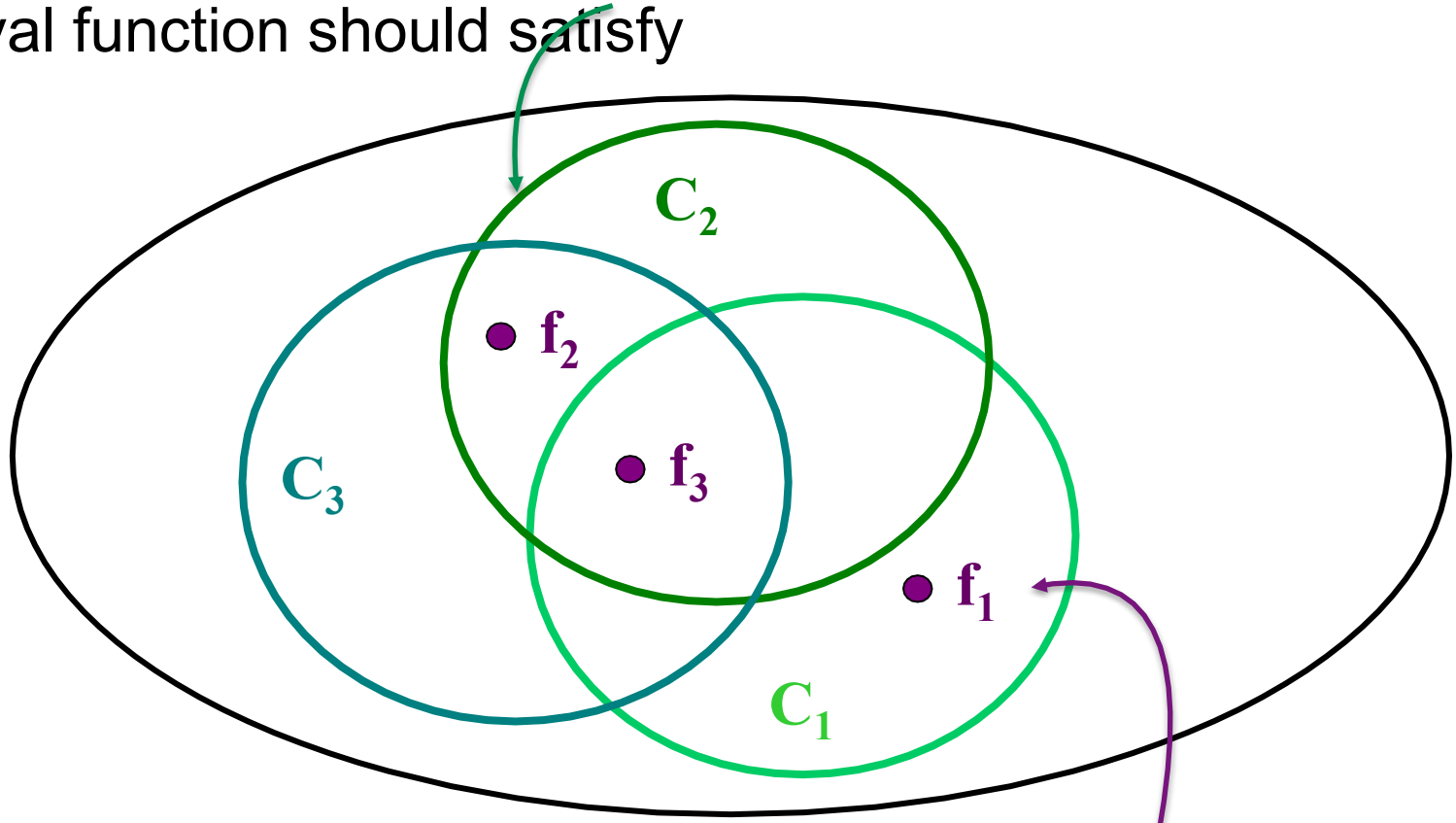
How to Find Well Performing Retrieval Functions?

- Simple approach: try all the variations, and evaluate their performance
  - Time consuming; can we really try “all” variations?

- If only we can find some formal guidance on how to find good retrieval functions!
  - Axiomatic approach helps to do that
  - A good retrieval function must satisfy a series of axioms;
  - And if it doesn’t …

Basic Idea of Axiomatic Approach

Define a set of retrieval constraints that any reasonable retrieval function should satisfy



The search space for all possible retrieval functions

An Axiomatic Framework for Retrieval Functions

- Component 1: Constraints to be satisfied by an effective retrieval function (Fang et al. 2004)
- Component 2: Function space that allows us to efficiently search for an effective function
  - We won't cover in this lecture (Fang and Zhai. 2005)

1. 问题背景

- 常见的检索函数（如 VSM、BM25）都依赖很多参数。
- **问题**：我们并不知道哪些函数一定表现好，往往只能不停尝试和调参 → 很耗时、不可预测。
- **目标**：是否有一套"原则"可以帮助我们判断一个检索函数是否合理?

---

2. 公理化方法的基本思想

- 定义一组 **检索约束 (retrieval constraints)**：任何"合理"的检索函数必须满足这些约束。
- 然后用这些约束来评估或设计函数。

换句话说，公理化方法是"**先规定规则，再让函数去符合规则**"。

---

3. 框架组成

课件引用了 Fang et al. (2004, 2005) 的研究，把框架分为两个部分 SI650-Week-03-IR-Models ：

1. **Component 1: 约束条件 (Constraints)**
   - 定义一组好的检索函数必须满足的性质，例如：
     - **TF 约束** (更多匹配的词 → 得分更高；但增长要递减)。
     - **长度归一化约束** (长文档要惩罚，但不能惩罚过度)。
     - **TF-长度交互约束** (词频和文档长度要合理结合)。
   - 这些就是"公理"。
2. **Component 2: 函数空间 (Function space)**
   - 给出一个候选函数空间，帮助我们高效地找到符合约束的函数。
   - 这部分在课件里没细讲。

Implementation of Component 1:

Question:
How do we define retrieval constraints?

What Should a Good Retrieval Function Actually Retrieve?

- A document that matches more query words?
- A document that matches more discriminative query words?
- A document that is longer?
- A document that covers more query words vs. a document that covers more unique query words?
- …

Let's think about what kinds of behaviors we want to see in an *idealized* retrieval scoring function

Another Way to Think About Scoring

- What do the good performers have in common?

1+ln(c(w,d))

Term Frequency

$$S(Q,D)=\sum_{t\in Q\cap D}\ln\frac{N+1}{df(t)}\cdot\frac{1+\ln(1+\ln(c(t,d)))}{1-s+s\frac{|D|}{avdl}}\cdot c(t,Q)$$

Alternative TF transformation

$$S(Q,D)=\sum_{t\in Q\cap D}\ln\frac{N-df(t)+0.5}{df(t)+0.5}\cdot\frac{(k1+1)\cdot c(t,D)}{k1(1-b+b\frac{|D|}{avdl})+c(t,D)}\cdot\frac{(k3+1)\cdot c(t,Q)}{k3+c(t,Q)}$$

Inverted Document Frequency

Document length normalization

How can we represent what we want to achieve with these?

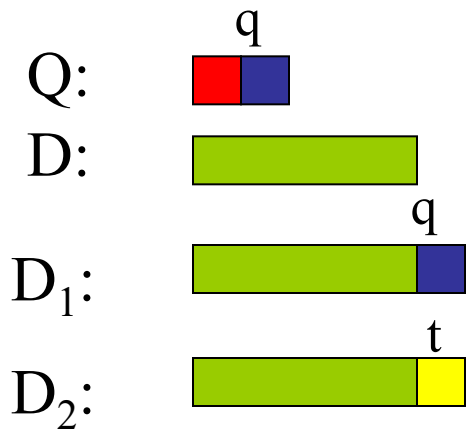
Term Frequency Constraints (TFC1)

- **TF weighting heuristic I:**
  Give a higher score to a document with more occurrences of a query term.

Let Q be a query and D be a document.

If $q \in Q$ and $t \notin Q$,

then $S(Q, D \cup \{q\}) > S(Q, D \cup \{t\})$



$$S(Q, D_1) > S(Q, D_2)$$

1. TFC1: 单调递增约束

- **含义**：当某个词在文档中的出现次数增加时，文档对该查询词的相关性分数应该 **单调增加。**
- **直观理解**：如果文档里"retrieval"出现 10 次，那肯定比只出现 1 次更相关。
- **公式化描述**：

$$tf_1 < tf_2 \quad \Rightarrow \quad score(tf_1) < score(tf_2)$$

- **问题**：如果无限增加词频，分数也无限增加，就会导致"刷词"作弊。
  👉 所以需要结合 TFC2。

Term Frequency Constraints (TFC2)

TF weighting heuristic II:

The amount of increase in the score due to adding a query term must decrease as we add more terms.

- *Let D_1, D_2, D_3 be documents of the same length.*
- *If*

2. TFC2: 递减收益约束 (Diminishing Returns)

- **含义**：随着词频增加，分数增长的幅度应该 **递减**，而不是线性增长。
- **直观理解**：
  - 第一次看到 "retrieval" → 很有用（强信号）。
  - 第 5 次出现 → 仍有帮助，但贡献小一点。
  - 第 50 次出现 → 几乎没有额外信息量了。
- **公式化描述**：

$$score(tf+1) - score(tf) \quad \text{递减，且趋向于 } 0$$

- **常见实现方式**：
  - 使用 **对数缩放**：$\log(1+tf)$
  - 或者 **平方根缩放**：$\sqrt{tf}$

$$c(q, D_2) - c(q, D_1) = c(q, D_3) - c(q, D_2) = 1$$

$$S(Q, D_2) - S(Q, D_1) > S(Q, D_3) - S(Q, D2)$$

Length Normalization Constraints (LNCs)

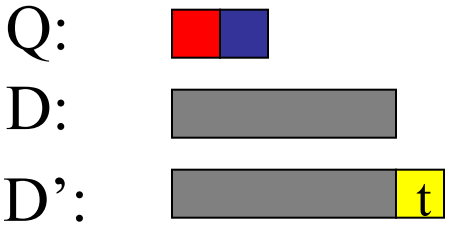
Document length normalization heuristic:
Penalize long documents (LNC1);
Avoid over-penalizing long documents (LNC2) .

- ***LNC1***

  Let Q be a query and D be a document.

  If t is a non-query term,

  then $S(D \cup \{t\}, Q) < S(D, Q)$

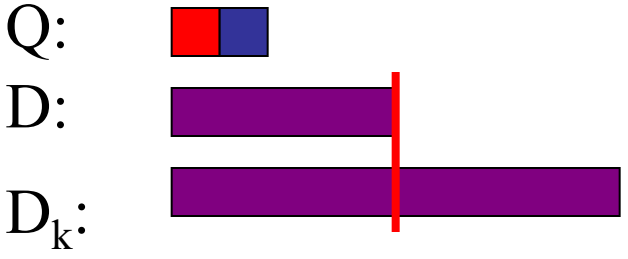


- ***LNC2***

  Let Q be a query and D be a document.

  If $D \cap Q \neq \varnothing$, and D_k is constructed by concatenating D with itself k times,

  then $S(D_k, Q) \geq S(D, Q)$



TF-LENGTH Constraint (TF-LNC)

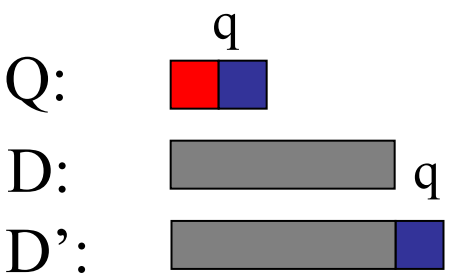
TF-LEN heuristic:

Regularize the interaction of TF and document length.

Let Q be a query and D be a document.

If q is a query term,

then $S(D \cup \{q\}, Q) > S(D, Q)$.



$$S(Q, D') > S(Q, D)$$

What You Should Know

- Vector space model is a typical (and well-performing) way to define retrieval functions
- TF-IDF weighting
- Formula of Pivoted normalization
- Advantage and disadvantage of VSM
- How to use retrieval constraints to analyze retrieval functions