ICU Done Wrong: How to Build a Polyglot Text Search Index

Sharzy Manaka, https://github.com/SharzyL

Apr 24, 2026

Story 2020

In 2020, I was still new to the Telegram world, finding it hard to search for Telegram messages.

So I built a bot!

Commit `6c5a8a1`

Sharzy Luo committed on Jul 10, 2020

```
first commit
```

`master` · v0.5.0 ••• v0.1.1

4 files changed

Story 2026

In 2026, I decided to rewrite it in 🚀🚀🚀 🦀🦀🦀 Rust.

- The search library whoosh has been unmaintained since 2022
- Unsatisfied with the current indexing mechanism
- Yes, we can vibe code in 2026

So I rewrote it in Rust + Tantivy, 3 months ago.

But why still not released?

Recall: How Full-Text Search (FTS) Works

$\text{doc}_0 =$ `武汉市长江大桥 Mayor of Wuhan, Jiang Daqiao`

$\text{doc}_1 =$ `南京市长江大桥 Nanjing Changjiang Bridge`

$\text{Jieba}(\text{doc}_0) =$ `武汉市`(0) `长江大桥`(1 2) | `Mayor`(3 4) | `of`(5 6) | `Wuhan`(7) `,`(8 9) | `Jiang`(10 11) | `Daqiao`(12)

$\text{Jieba}(\text{doc}_1) =$ `南京市`(0) `长江大桥`(1 2) | `Nanjing`(3 4) | `Changjiang`(5 6) | `Bridge`(7)

Inserted into the inverted index:

Term	doc	freq	positions
` `	doc_0	5	[2, 4, 6, 9, 11]
	doc_1	3	[2, 4, 6]
`,`	doc_0	1	[8]
`Bridge`	doc_1	1	[7]
`Changjiang`	doc_1	1	[5]
`Daqiao`	doc_0	1	[12]
.....			

Recall: How Full-Text Search (FTS) Works (Cont'd)

Term	doc	freq	positions
	doc_0	5	[2, 4, 6, 9, 11]
	doc_1	3	[2, 4, 6]
,	doc_0	1	[8]
Bridge	doc_1	1	[7]
Changjiang	doc_1	1	[5]
Daqiao	doc_0	1	[12]
Jiang	doc_0	1	[10]
Mayor	doc_0	1	[3]
Nanjing	doc_1	1	[3]
Wuhan	doc_0	1	[7]
of	doc_0	1	[5]
南京市	doc_1	1	[0]
武汉市	doc_0	1	[0]
长江大桥	doc_0	1	[1]
	doc_1	1	[1]

Query $Q =$ 武汉市 is parsed as $\mathrm{TermQuery}($武汉市$)$ and hits doc_0.

$$
\begin{aligned}
&\mathrm{Score}_{\mathrm{BM25}}(\mathrm{doc}_0, Q) \\
&= \mathrm{IDF} \cdot \frac{f \cdot (k_1 + 1)}{f + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\mathrm{avgdl}}\right)} \\
&= \ln\left(1 + \frac{N - n + 0.5}{n + 0.5}\right) \cdot \frac{1 \cdot 2.2}{1 + 1.2 \cdot \left(0.25 + 0.75 \cdot \frac{13}{10.5}\right)} \\
&= \ln 2 \cdot \frac{2.2}{2.41} \approx 0.63
\end{aligned}
$$

- $N = 2$: total number of documents
- $n = 1$: documents containing 武汉市
- $f = 1$: term frequency in doc_0
- $k_1 = 1.2, b = 0.75$: constant
- $|D| = 13$: length of doc_0 (in tokens)
- $\mathrm{avgdl} = 10.5$: average document length

CJK: From Jieba to Bigram

Jieba Fails for Japanese

$\text{doc}_2 =$ 天気がいいから、散歩しましょう

$\text{jieba}(\text{doc}_2) =$ 天 気 が い い か ら 、散 歩 し ま し ょ う

$\text{jieba}(\text{doc}_2, \text{HMM}) =$ 天 気 が い い か ら 、散歩 し ま し ょ う

$\text{ParseQuery}($天気$)$ is parsed as either

- $\text{TermQuery}($天$) \vee \text{TermQuery}($気$)$
- $\text{PhraseQuery}($天, 気$)$ (matches doc with 天 and 気 with adjacent positions)

Bad performance for both cases

Phrase Query Not Working As Expected

Jieba has a `cut_for_search` mode that generates overlapping tokens, to optimize recall

$$\text{JiebaForSearch}(\text{武汉市长江大桥}) = \underset{0}{\text{武汉}}\ \ \underset{0}{\text{武汉市}}\ \ \underset{1}{\text{长江}}\ \ \underset{1}{\text{长江大桥}}\ \ \underset{2}{\text{大桥}}$$

$\text{ParseQuery}(\text{武汉长江}) = \text{PhraseQuery}(\text{武汉}, \text{长江})$ **matches** 武汉市长江大桥.

Not necessarily problematic, but semantically incorrect

CJK Bigram: Simple but Working

- Jieba and cang-jie work for Chinese but fail for Japanese
- Lindera works for Chinese, Japanese and Korean, but is weak for Chinese
- No single segmenter works well for all
- Really?

$\text{doc}_0 =$ `武汉市长江大桥 Mayor of Wuhan, Jiang Daqiao`

$\text{CJKBigram}(\text{doc}_0) =$ `武汉`$_0$ `汉市`$_1$ `市长`$_2$ `长江`$_3$ `江大`$_4$ `大桥`$_5$ `Mayor`$_6$ `of`$_7$ `Wuhan`$_8$ `Jiang`$_9$ `Daqiao`$_{10}$

$\text{ParseQuery}($`武汉`$) = \text{TermQuery}($`武汉`$)$, **matches** doc_0

$\text{ParseQuery}($`武汉市`$) = \text{PhraseQuery}($`武汉`, `汉市`$)$, **matches** doc_0

Drawback: `市长` matches `武汉市长江大桥`, not necessarily bad

Japanese Bigram: More Than Kanji=漢字

Japanese is a mix of Kanji=漢字 and Kana=仮名

CJK Bigram in Lucene only generates inner-script bigrams

$$\mathrm{CJKBigram}_{\mathrm{Lucene}}(\text{可愛くてごめん}) = \text{可愛}\ \ \text{くて}\ \ \text{てご}\ \ \text{ごめ}\ \ \text{めん}$$

Consider the query 可愛く

- $\mathrm{TermQuery}(\text{可愛}) \vee \mathrm{TermQuery}(\text{く})$: matches too many
- $\mathrm{PhraseQuery}(\text{可愛}, \text{く})$: does not match 可愛くてごめん

Japanese Bigram: More Than 漢字 Cont'd

We choose to generate kanji-kana bigrams

$\mathrm{CJKBigram}_{\mathrm{ours}}(\text{可愛くてごめん}) = \text{可愛}\ \text{愛く}\ \text{くて}\ \text{てご}\ \text{ごめ}\ \text{めん}$

Consider the query 可愛く

- $\mathrm{PhraseQuery}(\text{可愛}, \text{愛く})$: **matches** 可愛くてごめん

Drawback: producing undesired bigrams

Unigram Desired

What if we want to search for a single CJK Unified Ideograph (汉字/漢字/.....)?

$\text{doc}_0 =$ ふたりの時間、言い訳の五千円

Double-field index:

$$\texttt{bigram}\ (\text{doc}_0) = \text{ふた}\ \ \text{たり}\ \ \text{りの}\ \ \text{の時}\ \ \text{時間}\ \ \text{言い}\ \ \text{い訳}\ \ \text{訳の}\ \ \text{の五}\ \ \text{五千}\ \ \text{千円}$$

$$\texttt{unigram}\ (\text{doc}_0) = \text{時}\ \ \text{間}\ \ \text{言}\ \ \text{訳}\ \ \text{五}\ \ \text{千}\ \ \text{円}$$

Only route unigram query to `unigram` index

$$\text{ParseQuery}(\text{ふたり　訳}) = \text{PhraseQuery}_{\text{bigram}}(\text{ふた}, \text{たり}) \vee \text{TermQuery}_{\text{unigram}}(\text{訳})$$

ICU: Segmentation, Normalization and More

Why ICU?

Unicode is full of weird things:

- 17 kinds of spaces (Space_Separator, Zs)
- (E202+U `RIGHT-TO-LEFT OVERRIDE`) tes retcarahc lortnoC
- Composable (`⿲一⿰冂⿰冂丶⿰冂丶` ⇒ `丽`) and decomposable (㍿=`株式会社`) things
- Same codepoint for different scripts (U+201C `“`)
- Different codepoints for the same ideograph (`神` U+FA19, `神` U+795E, `神` U+795E U+FE00)[1]

ICU is a library suite to handle Unicode.

[1] Typst still lacks complete IVS/SVS support

Normalization

UAX #15 defines 4 normalization forms:

- **Canonical** vs **Compatibility**: canonical decomposition preserves formatting distinctions; compatibility decomposition erases them.
- **Decomposed** vs **Composed**: whether to recompose sequences back into precomposed codepoints after decomposition.

	Decomposed	**Composed**
Canonical	**NFD** — break apart only ü → u + ◌̈, 神 (U+FA19) → 神 (U+795E)	**NFC** — break apart, then recombine Å (Angstrom) → A + ◌̊ → Å
Compatibility	**NFKD** — also flatten formatting ㍿ → 株式会社, ⑨ → 9, ﷺ → ا ل ل ه	**NFKC** — flatten, then recombine ﬁ́ (ligature fi + acute) → fi◌́ → fí

CaseFold

Unicode defines 4 case folds

- C (Common) `A` → `a`, `ς` → `σ`
- S (Simple) German `ß` → `ß`, Greek `Aι` → `ᾳ`
- F (Full) German `ß` → `ss`, Greek `Aι` → `αι`
- T (Turkic) Turkic `I` → `ı`, because uppercase `i` in Turkic is `İ`

Possible combinations: C + F, C + S, C + F + T, C + S + T

$\mathrm{CaseFold}_{\mathrm{C+F}}(\texttt{İ}) = ?$

CaseFold

Unicode defines 4 case folds

- C (Common) `A` → `a`, `ς` → `σ`
- S (Simple) German `ß` → `ß`, Greek `Aι` → `ᾳ`
- F (Full) German `ß` → `ss`, Greek `Aι` → `αι`
- T (Turkic) Turkic `I` → `ı`, because uppercase `i` in Turkic is `İ`

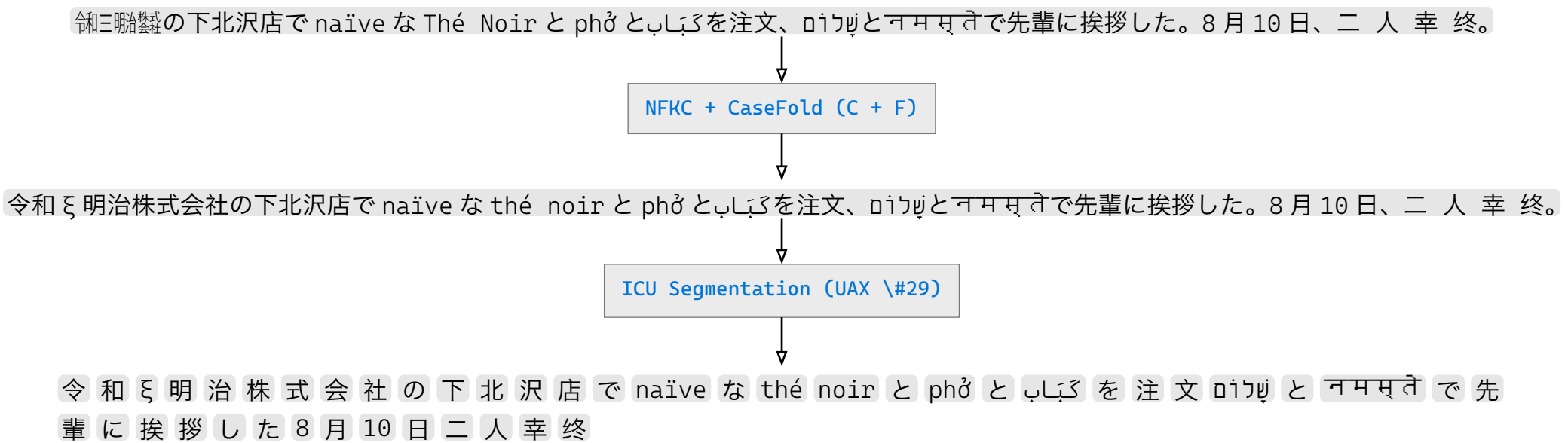
Possible combinations: C + F, C + S, C + F + T, C + S + T

$\mathrm{CaseFold}_{\mathrm{C+F}}(\texttt{İ}) = ?$

$\mathrm{CaseFold}_{\mathrm{C+F}}(\texttt{İ}) = \texttt{i}\ \texttt{◌̇}$

Segmentation (UAX #29)

ICU word break (UAX #29): rule-based segmentation using character properties.



Arabic and Hebrew

Arabic and Hebrew use **diacritical marks** for vowels that are often omitted in everyday writing. The same word may appear with or without them:

- `كِتَابٌ` (with harakat) = `كتاب` (without) = “kitāb” (book)
- `שָׁלוֹם` (with niqqud) = `שלום` (without) = “shalom”

Input	Normalized	Rule
ٱلله	الله	Alif wasla → Alif
مدرسة	مدرسه	Ta marbuta → Ha
ی / ک	ي / ك	Farsi → Arabic form
الـــله	الله / ﷲ	Tatweel (kashida) removed
٢٠٢٦ / ۲۰۲۶	2026	Arabic / Persian digits → ASCII
ت	ت	Don’t touch this

Not accurate for Uyghur Arabic alphabet

More Than Normalization: Diacritic Folding

- NFKC Casefold normalizes `A p p l e` → `apple` and `Ω` (U+2126) → `ω` (U+03C9)
- But precomposed accented characters survive: `café` stays `café`.

For search, we want `cafe` to match `café`. Solution: **diacritic folding**.

Algorithm: NFD decompose → strip foldable combining marks → NFC recompose

$$\texttt{café} \xrightarrow{\text{NFD}} \texttt{cafe◌́} \xrightarrow{\text{strip}} \texttt{cafe} \xrightarrow{\text{NFC}} \texttt{cafe}$$

Naïve idea: remove all combining marks (ccc > 0)

Combining Marks (CC) ≠ Diacritic Marks

Language	CC Usage	CC in Daily Writing	CC Stripping Acceptable
European Languages (?)	Accents, Variants	Yes	Mostly Yes
Japanese	Consonant Variants	Yes	Yes/No
Arabic, Hebrew	Vowels	No	Yes
Brahmic scripts	Vowels	Yes	No

Foldable ranges (Latin/Greek/Cyrillic/Vietnamese accents):

- U+0300–036F: Combining Diacritical Marks
- U+1AB0–1AFF: Combining Diacritical Marks Extended
- U+1DC0–1DFF: Combining Diacritical Marks Supplement

Preserved (structurally significant):

- Japanese dakuten (U+3099): `で` = て + ◌゙ — folding would destroy meaning
- Devanagari virama (U+094D): `क्ष` — consonant conjuncts
- Arabic harakat, Hebrew niqqud — handled by `SemiticNormalizationFilter`

Vietnamese: Diacritic Folding Unacceptable

Unmarked	Grave	Hook	Perispomeni	Acute	Dot
	◌̀(U+0300)	◌̉(U+0309)	◌̃(U+0303)	◌́(U+0301)	◌̣(U+0323)
Uppercase letters					
A	À (U+00C0)	Ả (U+1EA2)	Ã (U+00C3)	Á (U+00C1)	Ạ (U+1EA0)
Ă (U+0102)	Ằ (U+1EB0)	Ẳ (U+1EB2)	Ẵ (U+1EB4)	Ắ (U+1EAE)	Ặ (U+1EB6)
Â (U+00C2)	Ầ (U+1EA6)	Ẩ (U+1EA8)	Ẫ (U+1EAA)	Ấ (U+1EA4)	Ậ (U+1EAC)

[1]

[1] https://en.wikipedia.org/wiki/Vietnamese_alphabet

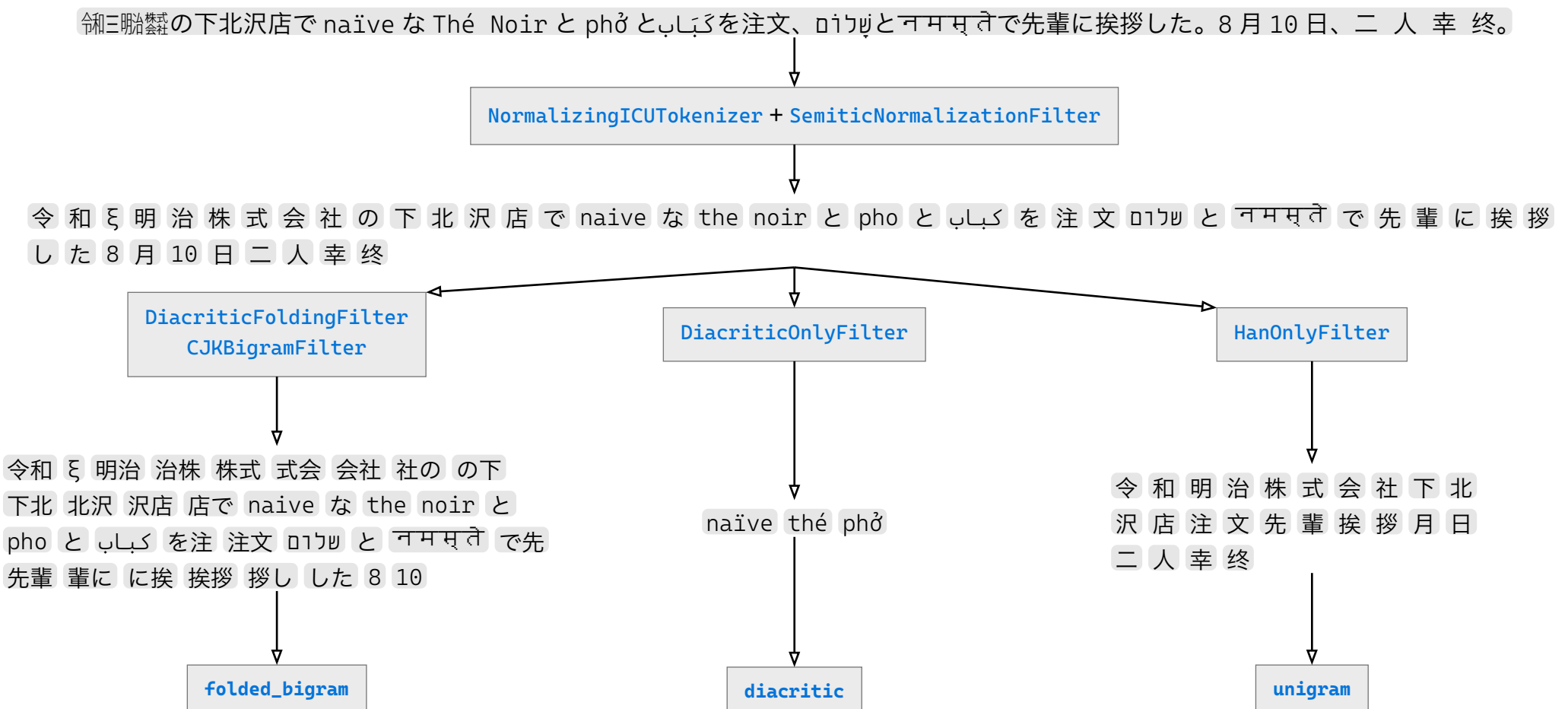
Solution: Triple-Field Indexing

Field	Content	Purpose
`folded_bigram`	diacritics folded + CJK bigrams	primary recall (broad match)
`unigram`	isolated Han unigrams	single-char CJK fallback
`diacritic`	only tokens **with** foldable diacritics, pre-fold form	sparse, exact-accent matching

Query routing:

- `cafe` (no diacritics) → $\mathrm{TermQuery}_{\mathrm{bigram}}(\texttt{cafe})$ — matches both
- `café` (has diacritics) → $\mathrm{TermQuery}_{\mathrm{bigram}}(\texttt{cafe}) \vee \mathrm{TermQuery}_{\mathrm{diacritic}}(\texttt{café})$
  - So `café` match is scored higher than `cafe` match

Solution: Triple-Field Indexing (Cont'd)



Scoring: Is BM25 the Silver Bullet?

Recall the BM25 formula:

$$\text{score} = \text{IDF} \cdot \frac{f \cdot (k_1 + 1)}{f + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

The $b \cdot \frac{|D|}{\text{avgdl}}$ term penalizes long documents. Makes sense for article search (a 10-page match is less relevant than a short article match).

But Telegram messages are short (5–50 tokens).

Matching a long message is **not necessarily less relevant** than matching a short message.

Scoring: Maybe IDF is Enough

Our approach: replace BM25 with `ConstScoreQuery`, scored by IDF only

$$\mathrm{IDF}(\mathrm{term}) = \ln\left(\frac{N - n(\mathrm{term}) + 0.5}{n(\mathrm{term}) + 0.5} + 1\right)$$

- IDF can be understood as information entropy (with numerical corrections)
- Rare terms get higher scores

Tradeoff: we lose term frequency signal entirely

Project Status

- `tantivy-analyzer-icu` is almost complete, will be a standalone repo later
- `tg-searcher` still lacks some features

Remarks:

- Prefer recall
- Achieve reasonable language-agnostic accuracy
- No language-specific features
  - Synonym
  - Stemmer
  - Stop word filter
  - OpenCC

今天信什么主义

Internationalism(国际主义)

Thanks & Questions