Quiz — 2 — Answer
MIR — Spring 2026
Instructor: Dr. Amin Eskandari
Head Teaching Assistants: Hamid Namjoo, Amir Hossein Hemati
Teaching Assistants: Ali Mojahed, Ali Nikvan, Amir Javvi

پاسخ سوال 1

الف) تحلیل و انتخاب الگوریتم ایندکس‌سازی:

در مواجهه با ۱۰ میلیون سند و تنها ۲ گیگابایت RAM ، استفاده از روش‌های معمولی که کل فهرست را در حافظه می‌سازند، غیرممکن است. هر دو الگوریتم BSBI و SPIMI برای ساخت فهرست معکوس برای مجموعه‌داده‌های بزرگ با حافظه محدود طراحی شده‌اند، اما تفاوت‌های کلیدی دارند:

۱ .الگوریتم BSBI (مرتب‌سازی بلوکی):

- روش کار:

1. مجموعه اسناد به بلوک‌هایی تقسیم می‌شود که termID-docID آن‌ها در RAM جا شود.
2. در هر بلوک، ابتدا نگاشت (term به termID با استفاده از یک دیکشنری که به تدریج در RAM رشد می‌کند) انجام می‌شود.
3. سپس جفت‌های termID-docID در حافظه مرتب‌سازی می‌شوند.
4. فهرست معکوس آن بلوک روی دیسک نوشته می‌شود.
5. در نهایت، همه بلوک‌های ذخیره‌شده روی دیسک با یکدیگر ادغام می‌شوند.

- مشکل اصلی: نیاز به نگهداری دیکشنری term به termID در RAM، برای مجموعه‌داده‌ای با ۱۰ میلیون سند، تعداد واژه‌های یکتا (Types) ممکن است به چندین میلیون برسد. این دیکشنری خود می‌تواند حجم زیادی از RAM را اشغال کند و فرآیند را با شکست مواجه سازد. (فصل ۴، بخش 4.3 به این محدودیت اشاره دارد).

۲ .الگوریتم SPIMI (ایندکس‌سازی تک‌گذره در حافظه):

- روش کار:

1. به‌جای استفاده از termID، مستقیماً از خود term استفاده می‌شود. این یعنی نیازی به نگاشت term به termID در سطح کلی نیست.

2. هر بلوک به صورت مستقل پردازش می‌شود: جفت term-docID ها جمع‌آوری می‌شوند. وقتی RAM پر شد، لیست termها مرتب شده و برای هر term یک posting list ساخته می‌شود و سپس بلوک روی دیسک نوشته می‌شود.
3. این فرآیند بدون مرحله مرتب‌سازی جفت‌ها و مستقیماً با ساخت پویای posting listها انجام می‌شود که سرعت را افزایش می‌دهد.

- مزیت برتر :حذف نگاشت termبه termID باعث صرفه‌جویی قابل توجه در حافظه RAM می‌شود. این دقیقاً همان نقطه قوتی است که SPIMI را برای محیط‌های با حافظه فوق‌العاده محدود (مانند سناریوی ما) به گزینه برتر تبدیل می‌کند. (فصل ۴، بخش 4.3 به وضوح این برتری را بیان می‌کند).

نتیجه‌گیری برای بخش الف :با توجه به محدودیت شدید ۲ گیگابایتی RAM ، الگوریتم SPIMI بهینه‌ترین انتخاب است، زیرا با حذف دیکشنری سراسری termبه termID، حافظه مصرفی را به شدت کاهش می‌دهد و امکان پردازش مجموعه‌داده‌های عظیم را در یک ماشین با RAM کوچک فراهم می‌کند.

ب) پشتیبانی از پرسمان‌های عبارتی:

برای پاسخ به پرسمان‌هایی که ترتیب کلمات در آن‌ها مهم است مانند "هوش مصنوعی"، فهرست معکوس ساده (که فقط docID را ذخیره می‌کند) کافی نیست. ما نیازمند یک فهرست موقعیتی (Positional Index) هستیم. در این ساختار، برای هر واژه، علاوه بر شناسه سند(docID) ، موقعیت دقیق وقوع واژه در آن سند (شماره توکن) نیز ذخیره می‌شود. ساختار یک posting در فهرست موقعیتی به صورت زیر است:

term: docID: [pos1, pos2, pos3, ...] ; docID: [pos1, pos2, ...]

برای مثال، اگر واژه "هوش" در سند شماره ۵ در موقعیت‌های ۱۲، ۴۵ و ۸۹ آمده باشد، و واژه "مصنوعی" در همان سند در موقعیت‌های ۱۳ و ۴۶ آمده باشد، سیستم می‌تواند با بررسی این موقعیت‌ها تشخیص دهد که آیا این دو واژه با فاصله ۱ از هم قرار دارند (یعنی یک عبارت را تشکیل می‌دهند) یا خیر. (فصل ۲، بخش 2.4.2 به‌طور کامل به این موضوع پرداخته است).

ج) تحلیل استراتژی پردازش پرسمان ترکیبی:

پرسمان ("هوش مصنوعی" AND "یادگیری ماشین") OR ("شبکه عصبی" AND NOT "پرسپترون ") شامل دو بخش اصلی است که با OR ترکیب شده‌اند.

استراتژی اول (بهینه) :هر زیرعبارت عبارتی "هوش مصنوعی "و "یادگیری ماشین " ابتدا با استفاده از فهرست موقعیتی پردازش می‌شوند. نتیجه هر کدام، مجموعه‌ای از docIDها است که آن عبارت دقیق را شامل می‌شوند. سپس این مجموعه‌ها با عملگرهای بولیAND ، OR، NOT که روی مجموعه‌ها عمل می‌کنند، ترکیب می‌شوند. این روش، معنای دقیق پرسمان را حفظ می‌کند.

استراتژی دوم (اشتباه و فاجعه‌بار) :اگر ابتدا کل پرسمان را به‌عنوان یک پرسمان بولی ساده در نظر بگیریم مثلاً هوش AND مصنوعی AND یادگیری AND ماشین و سپس نتایج را فیلتر کنیم، یک خطای مفهومی بنیادین رخ می‌دهد .پرسمان بولی ساده هوش AND مصنوعی هر سندی را که هر دو واژه "هوش" و "مصنوعی" را در هر جای متن و با هر ترتیبی داشته باشد، برمی‌گرداند (مثلاً جمله "مصنوعی بودن هوش او، کاملاً مشهود بود"). سپس اگر با یادگیری AND ماشین اشتراک بگیرد، مجموعه نتایج بسیار بزرگ و نادقیقی تولید می‌شود .فیلتر کردن این مجموعه بزرگ با فهرست موقعیتی در مرحله آخر، بسیار پرهزینه و از نظر محاسباتی غیربهینه است. زیرا شما حجم عظیمی از داده‌های نامرتبط را تا مراحل پایانی پردازش حمل کرده‌اید.

نتیجه‌گیری :استراتژی اول، یعنی پردازش زودهنگام قیدهای عبارتی با استفاده از فهرست موقعیتی، همواره بهینه‌تر است. این استراتژی با کاهش فضای جستجو در همان ابتدای کار، از پردازش‌های بیهوده جلوگیری کرده و دقت و سرعت را به طور هم‌زمان افزایش می‌دهد. اشتباه رایج دانشجویان این است که عملگر AND بولی را با مفهوم "عبارت" یکی می‌گیرند، در حالی که AND بولی نسبت به ترتیب و proximity واژه‌ها کور است.

نکته: همیشه ابتدا محدودکننده‌ترین بخش پرسمان (Selective Terms) را پردازش کنید .در پرسمان‌های ترکیبی، قیود عبارتی (" ") و عملگر NOT معمولاً بیشترین قدرت فیلترینگ را دارند و باید در اولویت پردازش قرار گیرند.

پاسخ سوال 2

الف) تحلیل پیچیدگی و انتخاب روش بهینه:

روش اول (محاسبه فاصله ویرایشی با تمام دیکشنری):

- الگوریتم :برای یک کلمه اشتباه q با طول m، و دیکشنری با N = 5,000,000 کلمه با طول متوسط n، محاسبه فاصله ویرایشی با برنامه‌نویسی پویا مانند الگوریتم Levenshtein پیچیدگی زمانی O(m × n) برای هر مقایسه دارد.
- تحلیل :زمان کل برای یک جستجو برابر خواهد بود با O(N × m × n) و با فرض m=7 و n=8، این یعنی حدود 280,000,000 ≈ 56 × 5,000,000 عملیات برای فقط یک کلمه اشتبا. این کار در یک محیط واقعی با میلیون‌ها کاربر در ثانیه، کاملاً غیرممکن و فاجعه‌بار است. (فصل ۳، بخش 3.3.3 به این هزینه بالا اشاره دارد).

روش دوم (استفاده از ایندکس k-gram و ضریب Jaccard):

- استراتژی دو مرحله‌ای:

1. مرحله فیلتر سریع :ابتدا کلمه اشتباه q به مجموعه ۳gram-هایش شکسته می‌شود (مثلاً $sa, sam, ams, msu, sun, ung, ng$). سپس با استفاده از ایندکس ۳gram-، فقط کلماتی از دیکشنری که حداقل تعداد مشخصی ۳ gram- مشترک با q دارند، به عنوان کاندیدا انتخاب می‌شوند. این انتخاب بر اساس ضریب Jaccard انجام می‌شود.
2. مرحله محاسبه دقیق :فاصله ویرایشی دقیق فقط برای این مجموعه کوچک از کاندیداها محاسبه می‌شود.

- فرمول ضریب Jaccard بر اساس k-gram:

$$J(q,t) = \frac{| G_q \cap G_t |}{| G_q \cup G_t |}$$

که در آن G_q مجموعه k-gramهای کلمه اشتباه و G_t مجموعه k-gramهای کلمه دیکشنری است. در عمل می‌توان از فرمول معادل و سریع‌تر زیر استفاده کرد:

$$J(q,t) = \frac{h}{|\,q\,| + |\,t\,| - h}$$

که h تعداد k-gram های مشترک و |q| و |t| تعداد k-gramهای هر کلمه است (فصل ۳، بخش 3.3.4).

- تحلیل کارایی :ایندکس k-gram فضای جستجو را از ۵ میلیون به شاید ۳۰-۲۰ کاندیدا کاهش می‌دهد. بنابراین زمان کلی به شدت کاهش یافته و عملی می‌شود. این دقیقاً روش استاندارد صنعتی است که در فصل ۳، بخش 3.3.4 به عنوان راه‌حل کارآمد معرفی شده است.

ب) شبیه‌سازی فرآیند اصلاح برای Samsang:

۱ .تولید ۳gram-ها برای Samsang: با اضافه کردن نماد $ به ابتدا و انتها :

$sa, sam, ams, msa, san, ang, ng$

۲ .جستجو در ایندکس ۳ gram-: برای هر ۳gram-، لیست کلمات دیکشنری که حاوی آن هستند از ایندکس بازیابی می‌شود.

۳ .محاسبه ضریبJaccard : برای کلمه دیکشنری Samsung، ۳gram-ها عبارتند از

$sa, sam, ams, msu, sun, ung, ng$

اشتراک این دو مجموعه ۶ مورد است ($sa, sam, ams, sun)

توجه: در مثال دقیق نیست، اما مفهوم را می‌رساند.

۴ .مقایسه با Samsoon: ۳gram-های Samsoon: $sa, sam, ams, mso, soo, oon, on$. اشتراک با Samsang کمتر است. فقط $sa, sam, ams بنابراین ضریب Jaccard پایین‌تری دارد و ممکن است از آستانه فیلتر عبور نکند یا در رتبه پایین‌تری قرار گیرد.

۵ .فیلتر نهایی :(Post-Filtering) حتی اگر کلمه‌ای از فیلتر Jaccard عبور کند، در مرحله بعد با محاسبه فاصله ویرایشی با Samsang، امتیاز نهایی آن مشخص می‌شود Samsung .با فاصله ویرایشی ۲ جایگزینی uبا aو حذف یک s انتخاب می‌شود. این فرآیند دو مرحله‌ای دقیقاً مطابق با الگوریتم شرح داده شده در فصل ۳ (بخش 3.3.4) است.

ج) اصلاح حساس به بافت:

اصلاح تک‌کلمه‌ای برای عبارت "Samsng Glaxy" ممکن است به "Samson Glaze" منجر شود، زیرا هر دو کلمه به تنهایی اصلاحات معتبری هستند. برای حل این مشکل، باید بافت عبارت را در نظر بگیریم:

۱ .تولید کاندیداهای مستقل :برای Samsng لیستی از کاندیداها مثل Samsung, Samson, Saming و برای Glaxy نیز لیستی مثل Galaxy, Glaze, Glary تولید می‌شود.
۲ .ایجاد ترکیب‌های ممکن :تمام ترکیب‌های ممکن از این دو لیست ساخته می‌شود مثلاً "Samsung Galaxy", "Samsung Glaze", "Samson Galaxy", ...
۳ .استفاده از آمار دوواژه‌ای: سیستم یک دیکشنری از دوواژه‌های رایج (که از لاگ جستجوهای قبلی کاربران یا خود مجموعه اسناد محصولات ساخته شده) دارد. برای هر ترکیب، فراوانی آن دوواژه در این دیکشنری بررسی می‌شود. دوواژه "Samsung Galaxy"احتمالاً فراوانی بسیار بالایی دارد (چون یک محصول محبوب است)، در حالی که "Samson Glaze"یا "Samsung Glaze"فراوانی صفر یا بسیار کمی دارند.
۴ .انتخاب بهترین ترکیب :سیستمی که در فصل ۳، بخش 3.3.5 Context Sensitive Spelling Correction توضیح داده شده است، دقیقاً به همین روش عمل می‌کند. این روش از مدل بولی گسترش‌یافته با پرسمان نزدیکی ضمنی استفاده می‌کند. به جای جستجوی Samsung AND Galaxy، سیستم تشخیص می‌دهد که این دو کلمه معمولاً با هم (در کنار هم) ظاهر می‌شوند و بنابراین ترکیب آن‌ها امتیاز بالاتری دارد. در نهایت، ترکیبی که بالاترین امتیاز را بر اساس آمار دوواژه‌ای داشته باشد، به عنوان پیشنهاد نهایی ("Samsung Galaxy") به کاربر نمایش داده می‌شود.

نکته: یک اشتباه رایج در طراحی سیستم‌های اصلاح املایی، نادیده گرفتن Context است. همیشه به یاد داشته باشید که یک سیستم جستجوی خوب، فقط کلمات را تصحیح نمی‌کند، بلکه قصد کاربر (User Intent) را تشخیص می‌دهد. استفاده از آمار n-gram مخصوصاً byword و triword بر روی لاگ جستجوها، یک روش قدرتمند و استاندارد برای فهم Context و ارائه پیشنهادهای هوشمندانه است.

پاسخ سوال 3

الف) ضرورت ایندکس‌سازی پویا و رویکردهای آن:

بازسازی کامل ایندکس هر شب (یا حتی هر هفته) برای یک خبرگزاری غیرقابل قبول است، زیرا اخبار جدید تا ۱۲ ساعت (زمان بازسازی) قابل جستجو نخواهند بود. در دنیای خبر، «نو» بودن محتوا حیاتی‌ترین عامل است.

برای حل این مشکل، فصل ۴ دو رویکرد اصلی برای ایندکس‌سازی پویا (Dynamic Indexing) معرفی می‌کند:

1. ایندکس کمکی:
   - روش :یک ایندکس اصلی بزرگ (روی دیسک) و یک ایندکس کمکی کوچک (درRAM) نگهداری می‌شود. اسناد جدید به ایندکس RAM اضافه می‌شوند. جستجو روی هر دو انجام و نتایج ادغام می‌شوند. حذف با یک bit vector مدیریت می‌شود. هر از گاهی، ایندکس RAM با ایندکس اصلی ادغام می‌شود.
   - عیب اصلی :هزینه ادغام ایندکس RAM با ایندکس اصلی بزرگ بسیار بالاست و در طول این ادغام، سیستم ممکن است با افت شدید کارایی مواجه شود.
2. ادغام لگاریتمی:
   - روش :نگهداری چندین ایندکس با اندازه‌های نمایی مثلاً I0, I1, I2, ... به ترتیب با ظرفیت‌های n, 2n, 4n, ... اسناد جدید ابتدا به ایندکس کوچک Z0 در RAM اضافه می‌شوند. وقتی Z0 پر شد، با ایندکس‌های روی دیسک طبق الگوریتمی مشابه یک binomial heap ادغام می‌شود. (فصل ۴، بخش 4.5).
   - عیب اصلی :پیچیدگی پیاده‌سازی و نیاز به جستجو روی چندین ایندکس مجزا.

ب) تشریح ساختار و فرآیند ادغام لگاریتمی با n=2:

در این روش، یک ایندکس درون‌حافظه‌ای به نام Z_0 داریم. ایندکس‌های روی دیسک با نام $I_0, I_1, I_2, \ldots$ وجود دارند. ظرفیت Z_0 برابر با n=2 سند است. اندازه ایندکس I_k برابر با $2^k \times n$ است (یعنی I_0 ظرفیت ۲، I_1 ظرفیت ۴، I_2 ظرفیت ۸ و....)

شبیه‌سازی برای ۸ سند اول (هر سند یک posting یا token در نظر گرفته می‌شود):

توضیح رویداد	وضعیت ایندکس‌های دیسک	وضعیت Z_0	سند جدید
سند ۱ به Z_0 اضافه می‌شود.	I0: -, I1: -	[1]	۱
سند ۲ اضافه می‌شود و Z_0 پر می‌شود.	I0: -, I1: -	[1, 2]	۲
Z_0 به I_0 (که خالی است) منتقل می‌شود و خودش خالی می‌شود. سند ۳ به Z_0 جدید اضافه می‌شود.	I0: [1, 2], I1: -	[3]	۳
Z_0 پر می‌شود.	I0: [1, 2], I1: -	[3, 4]	۴
Z_0 با I_0 (که پر است) ادغام می‌شود و I_1 جدید (با ۴ سند) ساخته می‌شود .و Z_0 خالی می‌شوند. سند ۵ به Z_0 اضافه می‌شود.	I0: -, I1: [1, 2, 3, 4]	[5]	۵
Z_0 پر می‌شود.	I0: -, I1: [1, 2, 3, 4]	[5, 6]	۶
Z_0 به I_0 منتقل می‌شود.	I0: [5, 6], I1: [1, 2, 3, 4]	[7]	۷
Z_0 پر می‌شود.	I0: [5, 6], I1: [1, 2, 3, 4]	[7, 8]	۸
Z_0 با I_0 و سپس I_1 ادغام شده و یک I_2 جدید می‌سازد. تمام ایندکس‌های قبلی خالی می‌شوند.	I0: -, I1: -, I2: [1..8]	[9]	۹

این فرآیند دقیقاً مطابق با الگوریتم LMERGEADDTOKEN در شکل 4.7 از فصل ۴ است.

ج) تأثیر مرتب‌سازی نتایج بر اساس زمان بر ساختار posting list:

پیش‌فرض در اکثر مباحث فهرست معکوس، مرتب‌سازی posting list بر اساس docID است. این کار به دلایل زیر انجام می‌شود:

- امکان ادغام سریع posting listها (با الگوریتم O(x+y))
- امکان فشرده‌سازی بهتر با استفاده از تکنیک‌های gap encoding (چون فواصل docID ها کوچک می‌شوند.)

اما در سناریوی خبری که «جدیدترین‌ها اول» اهمیت دارد، اگر posting listها بر اساس docID مرتب باشند، برای بازیابی ۱۰ خبر جدید باید کل posting list اسکن شود تا docIDهای بزرگ (جدیدتر) پیدا شوند. این کار برای واژه‌های پرتکرار بسیار کند است.

راه‌حل بهینه که در بخش 4.6 (انواع دیگر ایندکس‌ها) اشاره شده، استفاده از posting list مرتب‌شده بر اساس impact score (که می‌تواند زمان انتشار باشد) است. در این ساختار:

- برای هر ایندکس I_k، posting listها بر اساس زمان انتشار (جدیدترین در ابتدا) مرتب می‌شوند.
- مزیت: در هنگام جستجو، می‌توان پردازش posting list را پس از یافتن تعداد کافی سند جدید متوقف کرد(Early Termination) و این کار سرعت پاسخ‌گویی را به شدت افزایش می‌دهد.
- معایب: ادغام ایندکس‌ها در روش لگاریتمی پیچیده‌تر می‌شود، زیرا نمی‌توان از ادغام ساده O(x+y) استفاده کرد و نیاز به مرتب‌سازی مجدد است. همچنین، درج سند جدید در ایندکس‌های بزرگ (چون جدیدترین است) همیشه در ابتدای posting listاتفاق می‌افتد و نه در انتهای آن، که این نیز عملیات سنگین‌تری است.

نکته: در طراحی سیستم‌های واقعی، یک ساختار داده واحد مانند posting listمرتب بر اساس docID برای همه کاربردها ایده‌آل نیست. شما باید بر اساس الگوی دسترسی به داده (Access Pattern) غالب در جستجوهای کاربران خود تصمیم بگیرید. اگر ۹۰٪ جستجوها به دنبال محتوای جدید هستند، بهینه‌سازی posting list برای بازیابی جدیدترین اسناد، یک تصمیم استراتژیک صحیح است، حتی اگر فرآیند ایندکس‌سازی را اندکی پیچیده‌تر کند.

پاسخ سوال 4

بخش (الف): فاجعه هش در دیکشنری

تحلیل مشکل:
جدول هش یک ساختار داده فوق‌العاده برای جستجوی دقیق (Exact Match) است. پیچیدگی متوسط آن O(1) است. اما مشکل بزرگ آن این است که هیچ ترتیب و ساختار منطقی بین کلیدها وجود ندارد. در یک جدول هش، کلید "economy" و "economics" ممکن است در دو باکت کاملاً متفاوت و دور از هم قرار بگیرند که هیچ ارتباطی به هم ندارند.

- برای پرسمان*econ : موتور جستجو باید تمام کلماتی که با "econ" شروع می‌شوند را پیدا کند. با ساختار هش، تنها راه این است که تک‌تک کلیدهای دیکشنری (که ممکن است میلیون‌ها عدد باشند) را از جدول هش واکشی کرده و بررسی کنیم که آیا با "econ" شروع می‌شوند یا خیر. این کار یک پیمایش خطی O(N) روی کل دیکشنری است که برای یک پرسمان آنلاین کاملاً غیرقابل قبول است.
- برای تکمیل خودکار: مشکل دقیقاً مشابه بالاست. برای یافتن کلماتی که با "comp" شروع می‌شوند، باید کل دیکشنری جستجو شود.

راه‌حل صحیح:
فصل ۳، بخش 3.1 به وضوح درختان جستجو (Search Trees) مانند B-Tree را به عنوان ساختار استاندارد دیکشنری در موتورهای جستجو معرفی می‌کند. در یک B-Tree ، تمام کلیدها به صورت مرتب شده بر اساس حروف الفبا ذخیره می‌شوند.

- برای پرسمان *econ، موتور جستجو در درخت به دنبال کلید "econ" می‌گردد. از آن نقطه به بعد، تمام کلیدهایی که در زیردرخت سمت راست آن قرار دارند (و از نظر لغوی بزرگتر هستند) تا جایی که پیشوند "econ" دیگر برقرار نباشد، به صورت ترتیبی و بسیار سریع با پیچیدگی O(log N + K) که K تعداد نتایج است، بازیابی می‌شوند.

تله مهندسا:
مهندسان تیم در دام یک دیدگاه عمومی و غیرتخصصی به طراحی سیستم افتاده‌اند. آنها فقط به دنبال سریع‌ترین ساختار داده برای عملیات درج و جستجوی نقطه‌ای (Point Query) بوده‌اند (O(1) هش). اما فراموش کرده‌اند که در بازیابی اطلاعات، جستجوی دامنه‌ای (Range Query) و جستجوی

پیشوندی (Prefix Search) به اندازه جستجوی دقیق اهمیت دارند. آنها بهینه‌سازی محلی کرده‌اند اما معماری کلی سیستم را نابود کرده‌اند.

نکته مهم: همیشه قبل از انتخاب ساختار داده، الگوی پرسمان‌های (Query Patterns) کاربران خود را تحلیل کنید. اگر نیازمندی‌ها شامل wildcard، prefix search، یا spelling correction (که در فصل ۳ مفصل بحث شده) است، هرگز از هش استفاده نکنید.

بخش (ب): محدودیت مهلک ایندکس دوواژه‌ای در برابر ایندکس موقعیتی

تحلیل مشکل:
ایندکس دوواژه‌ای (Biword Index) هر جفت کلمه متوالی را به عنوان یک term جدید در دیکشنری ذخیره می‌کند (مثلاً "fast car" یک کلید می‌شود). این روش برای پرسمان "fast car" عالی است، زیرا فقط یک جستجوی دیکشنری انجام می‌شود.

اما پرسمان نزدیکی "breach" /3 "contract" را در نظر بگیرید. این پرسمان به دنبال اسنادی است که دو کلمه contract و breach با حداکثر ۳ کلمه فاصله از هم ظاهر شده‌اند. این دو کلمه می‌توانند به اشکال زیر ظاهر شوند:

- contract ... breach فاصله ۳
- breach ... contract فاصله ۳
- contract of the breach فاصله ۳

در ایندکس دوواژه‌ای، برای پوشش این حالت، باید تمامی ترکیب‌های ممکن از عبارات با طول ۲ تا ۵ کلمه که شامل این دو واژه هستند، از قبل ایندکس شوند. این کار انفجار نمایی در تعداد کلیدهای دیکشنری را به دنبال دارد و عملاً غیرممکن است. (فصل ۲، بخش 2.4.1 به معایب ایندکس دوواژه‌ای اشاره دارد).

راه‌حل صحیح:
ایندکس موقعیتی (Positional Index) که در فصل ۲، بخش 2.4.2 توضیح داده شده، راه‌حل استاندارد است. در این ساختار، برای هر term، یک posting list داریم که هر posting شامل docID و لیست موقعیت‌های (Positions) وقوع آن term در سند است.

برای پاسخ به "breach" /3 "contract":

1. لیست ارسال contract و breach خوانده می‌شود.
2. برای هر docID مشترک، لیست موقعیت‌های P1 برای contract و P2 برای breach مقایسه می‌شود.
3. الگوریتم بررسی می‌کند که آیا دو موقعیت pos1 از P1 و pos2 از P2 وجود دارند که |pos1 - pos2| <= 3 باشد. (الگوریتم دقیق آن در شکل 2.12 آمده است).

این روش انعطاف‌پذیری کامل برای هر فاصله k و هر ترتیبی از کلمات را فراهم می‌کند، بدون آنکه نیاز به پیش‌پردازش و ایندکس‌سازی جداگانه برای هر ترکیب باشد.

بخش (ج): فاجعه عملکردی در پردازش پرسمان ترکیبی

تحلیل و محاسبه ریاضی:
استراتژی تیم فنی:

1. خواندن posting list عظیم artificial (۱۰ میلیون سند) و intelligence (۸ میلیون سند)
2. اشتراک‌گیری این دو لیست (که نتیجه آن حدود ۱ میلیون سند است، چون همه اسناد حاوی artificialو intelligenceبا هم نیستند)
3. سپس در انتها، فیلتر کردن این ۱ میلیون سند برای یافتن آنهایی که artificial و intelligence دقیقاً کنار هم هستند.

محاسبه تعداد مقایسه‌ها:
الگوریتم اشتراک‌گیری دو لیست مرتب (با روش دو اشاره‌گر) O(x+y) مقایسه docID انجام می‌دهد.

- اشتراک اولیه :O(10,000,000 + 8,000,000) = 18,000,000 مقایسه.
- سپس باید برای هر سند در نتیجه (۱ میلیون سند)، فاصله موقعیت‌ها بررسی شود (که خود O(P1+P2)مقایسه موقعیت دارد

استراتژی بهینه (پردازش زودهنگام قیود):

1. ابتدا پرسمان عبارتی "artificial intelligence" با استفاده از ایندکس موقعیتی پردازش می‌شود. این کار مستقیماً و با استفاده از الگوریتم POSITIONALINTERSECT، آن ۵۰,۰۰۰ سندی که این عبارت در آن‌ها وجود دارد را پیدا می‌کند. تعداد مقایسه‌های انجام شده متناسب با اندازه posting listهای این دو واژه و تعداد موقعیت‌های آن‌هاست، اما نتیجه نهایی بسیار کوچک است (۵۰,۰۰۰)
2. سپس این مجموعه کوچک ۵۰,۰۰۰ تایی با نتایج سایر بخش‌های پرسمان ترکیب می‌شود.

مقایسه عددی:

- روش تیم فنی (اشتباه) :حداقل ۱۸ میلیون مقایسه docID فقط برای مرحله اول، به علاوه هزینه سنگین فیلتر موقعیتی روی ۱ میلیون سند.
- روش بهینه (صحیح) :پردازش موقعیتی اولیه که روی لیست‌های بزرگ انجام می‌شود ولی به دلیل الگوریتم کارآمد POSITIONALINTERSECT با استفاده از skip pointers (در صورت وجود) و مرتب بودن داده‌ها، بسیار سریع‌تر از آن چیزی است که به نظر می‌رسد. اما نکته اصلی این است که مجموعه خروجی این مرحله (۵۰,۰۰۰) به جای ۱,۰۰۰,۰۰۰، بار پردازشی مراحل بعدی را به شدت کاهش می‌دهد.

تله تفکر الگوریتمیشون:

اشتباهشون و احتمالا اشتباهتون، عدم درک تفاوت معنایی و عملیاتی بین AND بولی و قید عبارت (Phase)، بود.

- عملگر AND بولی: صرفاً اشتراک مجموعه‌ای docIDهاست. ترتیب و proximity کلمات را نادیده می‌گیرد. نتیجه آن یک مجموعه بزرگ و نادقیق است.
- قید عبارت: یک فیلتر هندسی/موقعیتی است که نیاز به داده‌های اضافی (موقعیت) دارد و مجموعه نتایج را به شدت محدود می‌کند.

اونا (و احتمالا شما) به اشتباه فکر می‌کنند/می‌کردید که "A B" یعنی A AND B این در حالی است که "A B" یک زیرمجموعه بسیار کوچک از A AND B است. در بهینه‌سازی پرسمان، قانون طلایی این است: محدودکننده‌ترین بخش پرسمان را اول پردازش کنید. در اینجا،

بخش "artificial intelligence" به مراتب محدودکننده‌تر از artificial AND intelligence است و باید در اولویت پردازش قرار گیرد. این اصل در انتهای فصل ۱ (بخش 1.3) تحت عنوان استراتژی بهینه برای پرسمان‌های AND توضیح داده شده است: واژه‌ها را به ترتیب افزایش فراوانی سند پردازش کنید. در اینجا، «فراوانی سند» برای یک عبارت، به مراتب کمتر از تک‌کلمات تشکیل‌دهنده آن است.

نکته مهم: همیشه پرسمان را از منظر قدرت فیلترینگ (Selectivity) آنالیز کنید. بخشی از پرسمان که کمترین تعداد سند را برمی‌گرداند، گلوگاه(bottleneck) بهینه‌سازی شماست. تمام تلاش خود را بکنید تا آن بخش را اول از همه، با دقیق‌ترین ایندکس موجود، پردازش کنید.