بسمه تعالی

جزوه

مبانی بازیابی و جست‌جوی وب

نام استاد:

نرگس مرعشی

گردآورندگان:

محمدمعید جمالی‌مهر

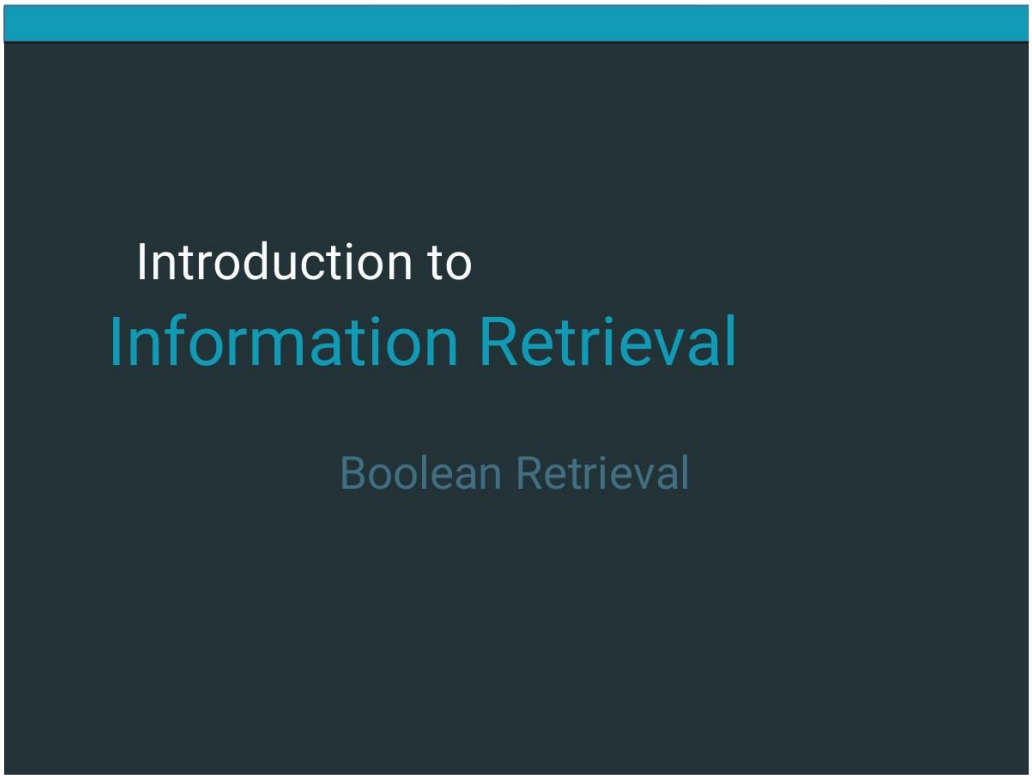
علیرضا نیکنام پیردهی

بهار ۱۴۰۲

مقدمه

در این درس قرار است شما با طرز کار موتورهای جست‌جو و الگوریتم‌های آن‌ها آشنا شوید. از گوگل (Google) می‌توانیم به عنوان یکی از شناخته شده‌ترین و قدرتمندترین موتور جست‌جو نام ببریم. با فرض آنکه تعداد صفحات وب در دنیا غیر قابل شمارش هستند، لازمه اینکه گوگل تشخیص دهد چه صفحه‌ای برای شما بهترین انتخاب محسوب می‌شود استفاده از چندین الگوریتم است.

هدف این درس آشنایی با گوشه‌ای از طرز کار موتور جست‌جوی گوگل است. گر چه به علت محرمانگی امکان دسترسی عمومی به الگوریتم‌های گوگل فراهم نیست اما می‌توانیم با تحلیل عملکرد به چند الگوریتم پایه برای جست‌جومیان صفحات وب (Document) دست یابیم. در ادامه به بررسی این الگوریتم‌ها و نحوه رتبه‌بندی صفحات وب خواهیم پرداخت.



جلسه اول

ماتریس سند – عبارت (Term - Document):

داده‌های ما یا حالت ساختاریافته (Structured) دارند و یا غیرساختاریافته (Unstructured) هستند. به عنوان مثال داده‌های ساختاریافته را می‌توان همانند ذخیره‌سازی نام و نام‌خانوادگی یا سن افراد در یک پایگاه‌داده نام برد. در این حالت داده‌های ما اسامی اشخاص یا سن آن‌ها خواهد بود. اما متنی را فرض کنید که تمام حروف کلمات آن از یکدیگر فاصله ندارند به طوری که نمی‌توان مفهوم جمله را بدون خواندن سطحی تشخیص داد. این متن را می‌توانیم یک داده غیرساختاریافته بنامیم چرا که باید آن را تحلیل و مفهوم آن را دریابیم. یکی از کارهایی که گوگل انجام می‌دهد تحلیل جملات و تشخیص مفهوم آن‌هاست.

بیاییم نگاهی به موتورهای جست‌جو بیاندازیم. فرض کنید می‌خواهیم کلمه Software را پیدا کنیم. سوال اینجاست که گوگل چطور تشخیص می‌هد که کلمه Software در کدام صفحه قرار دارد. اولین روش پیشنهادی برای یافتن یک کلمه در یک سند **ماتریس سند – عبارت** است. ماتریسی که مشخص می‌کند آیا عبارت در سند هست یا خیر.

«در نظر داشته باشید ماتریس سند – عبارت در بعد از عملیات جست‌جو ساخته می‌شود»

در ماتریس سند – عبارت به تعداد عبارت‌ها سطر و به تعداد سندها ستون خواهیم داشت. فرض کنید می‌خواهیم اسامی چند شخصیت داستانی را در نمایشنامه‌های شکسپیر جست‌جو کنیم و ببینیم آیا این شخصیت در این نمایشنامه حضور دارد یا خیر. در ماتریس سند – عبارت اگر عبارت مورد نظر (سطر) در سند مورد نظر (ستون) یافت شده باشد، در تقاطع سطر و ستون ماتریس مقدار 1 قرار می‌گیرد اما اگر عبارت در سند وجود نداشته باشد، مقدار 0 خواهد گرفت.

فرض کنید پس از جست‌جو ماتریس ما به صورت زیر باشد:

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

حال گوگل چطور از ماتریس سند – عبارت استفاده می‌کند؟

در پایان جست‌جو و تشکیل ماتریس سند – عبارت، گوگل محدوده (وکتور) اعداد بدست آمده در سطر سند مورد نظر را کنار هم قرار می‌دهد. به عنوان مثال در ماتریس بالا عبارت Brutus تنها در سه سند یافت شده است. در این صورت به جای عبارت Brutus از وکتور آن استفاده خواهد شد.

بیایم چند عبارت را با هم جست‌جو کنیم. فرض کنید می‌خواهیم عبارت Brutus و Caesar و Calpurnia را با هم جست‌جو کنیم. یعنی سندی باید وجود داشته باشد که هر سه این عبارات را در خودش داشته باشد. اگر بیاییم و از وکتور هر سه عبارت استفاده کنیم، می‌توانیم با عمل ضرب (And) به جواب برسیم:

Brutus → 110100
Caesar → 110111
Calpurnia → 101111

110100 And 110111 And 101111 = **100100**

می‌بینیم با اعمال ضرب پاسخ به یک وکتور از 0 و 1 تبدیل شده است. با بررسی حاصل بدست آمده می‌توانیم تشخیص دهیم این سه عبارت با هم در سند شماره ۲ یافت می‌شوند چرا که رقم دوم از سمت چپ مقدار 1 گرفته است.

حال اگر بخواهیم دو عبارت Brutus و Caesar با هم در یک سند باشند اما عبارت Calpurnia در آن نباشد می‌توانیم وکتور عبارت Calpurnia را نقیض کنیم و سپس عمل ضرب را انجام دهیم.

ماتریس سند – عبارت به نظر مفید می‌آید اما مشکل اصلی چیست؟

مجموعه صفحات وب (Collection):

در این درس از آنجایی که به صفحات وب سند (Document) گفته می‌شود، مجموعه‌ای از صفحات وب یا اسناد را کلکسیون (Collection) می‌نامیم.

فرض کنید ما کلکسیونی به تعداد $N = 1000000$ سند داریم که در هر سند ۱۰۰۰ کلمه وجود دارد. به طور میانگین هر کلمه با فرض فاصله ۶ بایت حافظه را اشتغال کرده است. در نتیجه ما 1000000×6000 بایت (در مجموع 6GB) داده خواهیم داشت. می‌توانیم با آزمون و خطا پی ببریم از یک میلیون سند می‌توانیم $M = 500000$ عبارت غیر تکراری پیدا کنیم (کلمات تکراری the و as و غیره می‌باشند).

حال فرض کنید برای این کلکسیون و این تعداد عبارت بخواهیم یک ماتریس سند – عبارت بسازیم. این ماتریس قرار است 1M ستون و 500K سطر داشته باشد که شامل 500 میلیارد 0 و 1 خواهد بود. با فرض اینکه تعداد 1ها به یک میلیارد برسد این ماتریس یک ماتریس خلوت (Sparse) خواهد بود چرا که تعداد بسیاری 0 در آن وجود دارد.

بهترین راه حل چیست؟ برای حل این مشکل می‌توانیم تنها 1 ها را ذخیره کنیم. یعنی به جای استفاده از 0 و 1 از شماره سطرو ستونی استفاده کنیم که مقدار 1 دارد. به عنوان مثال اگر مقدار 1 در سطر ۵ و ستون ۶ باشد عدد (5,6) ذخیره خواهد شد.

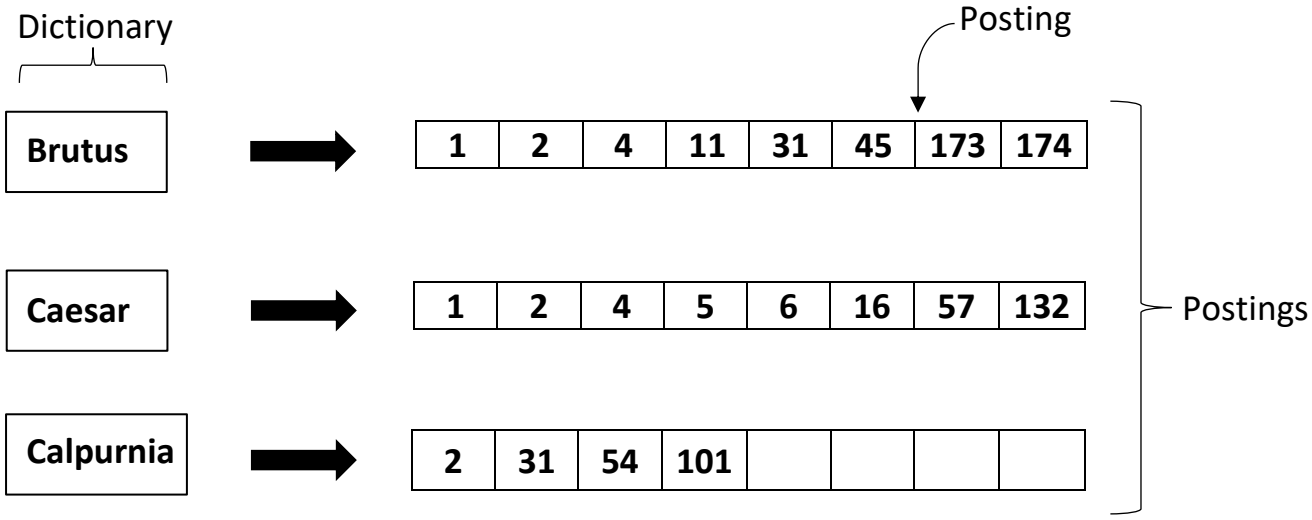
جلسه دوم

اندیس‌گذاری یا شاخص‌گذاری معکوس:

در جلسه قبل در خصوص ماتریس سند – عبارت صحبت کردیم. یکی از مشکلاتی که این ماتریس داشت عدم کارایی هنگام افزایش کلکسیون بود. بیان کردیم برای حل این مشکل می‌توانیم تنها مقدار 1 های ماتریس را ذخیره کنیم.

روش اندیس‌گذاری معکوس (Inverted Index) یکی از روش‌های نوینی است که جایگزین ماتریس سند – عبارت شد. این روش همچنان در حال استفاده است. این روش مشخص می‌کند عبارت شما در کدام سندها وجود دارد به جای استفاده از آدرس و کلمه خاص، از یک **شناسه سند** (DocID) و کلمه استفاده می‌کند. در حقیقت برای هر سند یک شماره مشخص می‌شود و برای یافتن آنکه چه کلماتی در سند وجود دارد از یک لیست استفاده می‌کند به طوری که با استفاده از یک اشاره‌گر میان عبارت و لیست ارتباط برقرار می‌کند.

به عنوان مثال:



در مثال بالا عبارت Brutus در سندهای ۱ و ۲ و ۴ و ۱۱ و ۳۱ و ۴۵ و ۱۷۳ و ۱۷۴ یافت شده است.

در اندیس‌گذاری معکوس به مجموعه عبارت‌ها به اصطلاح **دیکشنری** (Dictionary) و به شماره سندها **پُستینگ** (Posting) گفته می‌شود. در حقیقت در دیکشنری کلمات ذخیره می‌شوند و در پستینگ لیست شماره سندهایی که آن کلمه در آن وجود دارد ذخیره می‌گردد.

آیا می‌توان از یک آرایه با اندیس ثابت برای ذخیره‌سازی شناسه سند استفاده کرد؟

چه اتفاقی می‌افتد اگر کلمه Caesar به سند شماره ۱۴ اضافه شود؟

دلیل اصلی که نباید از یک آرایه با اندیس ثابت استفاده کرد محدود شدن در هنگام اضافه کردن کلمه است. زمانی که کلمه مورد نظر در چندین سند اضافه شود ممکن است به علت نبود فضا سرریز رخ داده و شماره سندها حذف شوند.

اما باید توجه داشت شماره اسناد در پستینگ لیست به صورت مرتب شده ذخیره شده‌اند. در هنگام اضافه کردن یک سند به لیست باید یک الگوریتم تا زمانی لیست را پیمایش کند که شماره سند فعلی بزرگتر از سند ذخیره شده در لیست می‌باشد.

«هم دیکشنری و هم پستینگ لیست به صورت مرتب شده هستند»

بنابراین برای پستینگ لیست‌ها باید از لیست‌های با طول متغیر استفاده کرد.

در فضاهای ذخیره‌سازی متفاوت از جمله دیسک‌های سخت اجرای مداوم پستینگ‌ها بهترین روش است. اما درفضای حافظه می‌توانیم از لیست‌های پیوندی و آرایه‌هایی با طول متغیر استفاده کنیم.

ساختار اندیس گذاری معکوس:

۱- بدست آوردن توکن‌ها (Tokenizer): در ابتدا به منظور شاخص‌گذاری سند توکن‌های آن سند را پیدا می‌کند. در حقیقت با حذف پیوستگی میان کلمات در جمله، کلمه توکن را پیدا می‌کند.

۲- استفاده از ماژول‌های زبانی: با استفاده از حذف قواعد و قوانین زبان‌ها، توکن‌ها را به منظور استفاده در شاخص‌گذاری ویرایش می‌کند (به عنوان مثال تبدیل جمع به مفرد: Friends ← friend).

۳- شاخص‌گذاری (Indexer): در نهایت با استفاده از توکن‌های ویرایش‌شده شاخص‌گذاری معکوس را انجام می‌دهد.

مراحل اولیه پردازش متن:

در روش شاخص‌گذاری معکوس بدست آوردن توکن‌ها از سندها بسیار ضروری است. توکن‌ها باید قبل از استفاده در شاخص‌گذاری ویرایش شوند. در اینجا برخی از مراحل اولیه پردازش متن را مورد بررسی قرار می‌دهیم. به خاطر داشته باشید این مراحل به قواعد و قوانین زبان جست‌جو بستگی دارد:

- نشانه‌گذاری: یعنی حذف دنباله‌ای کارکترها و تبدیل آن‌ها به کلمه توکن (جداسازی کلمات)

- نرمال‌سازی: یعنی تبدیل متن یا عبارت به فرم خودش(به عنوان مثال تبدیل U.S.A به USA)

- ریشه‌یابی: یعنی بدست آوردن ریشه کلمات (به عنوان مثال تبدیل authorization به authorize)

- خالص‌سازی: یعنی حذف حروف مشترک (به عنوان مثال حذف the ، a ، to ، of)

مراحل شاخص‌گذاری:

پس از پردازش متن سند و ویرایش توکن‌ها، مراحل شاخص‌گذاری شروع می‌شود. بگذارید تا با یک مثال به شرح مراحل آن بپردازیم. فرض کنید ما برای شروع دو سند Doc1 و Doc2 را داریم که به ترتیب دارای شناسه 1 و 2 هستند. متن داخل این سندها به صورت زیر می‌باشد:

Doc1: I did enact Julius Caesar I was killed i’ the Capitol; Brutus killed me.

Doc2: So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious.

پس از خواندن متن‌ها و جداسازی توکن‌ها، موتور جست‌جو در مرحله **اول** یک ماتریس با دو ستون عبارت (Term) و شناسه سند (docID) ایجاد می‌کند (دنباله‌ای از توکن‌ها):

Doc 1

I did enact Julius Caesar I was killed i’ the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious.

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i’	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
Caesar	2
was	2
ambitious	2

در مرحله **دوم** (اصلی‌ترین مرحله شاخص‌گذاری) به منظور پیمایش و جست‌جو سریع‌تر ماتریس عبارت – شناسه سند را مرتب می‌کند (مرتب‌سازی). در ابتدا عبارت‌ها به ترتیب حروف الفبا و سپس شناسه سندها به ترتیب از کوچکتر به بزرگتر:

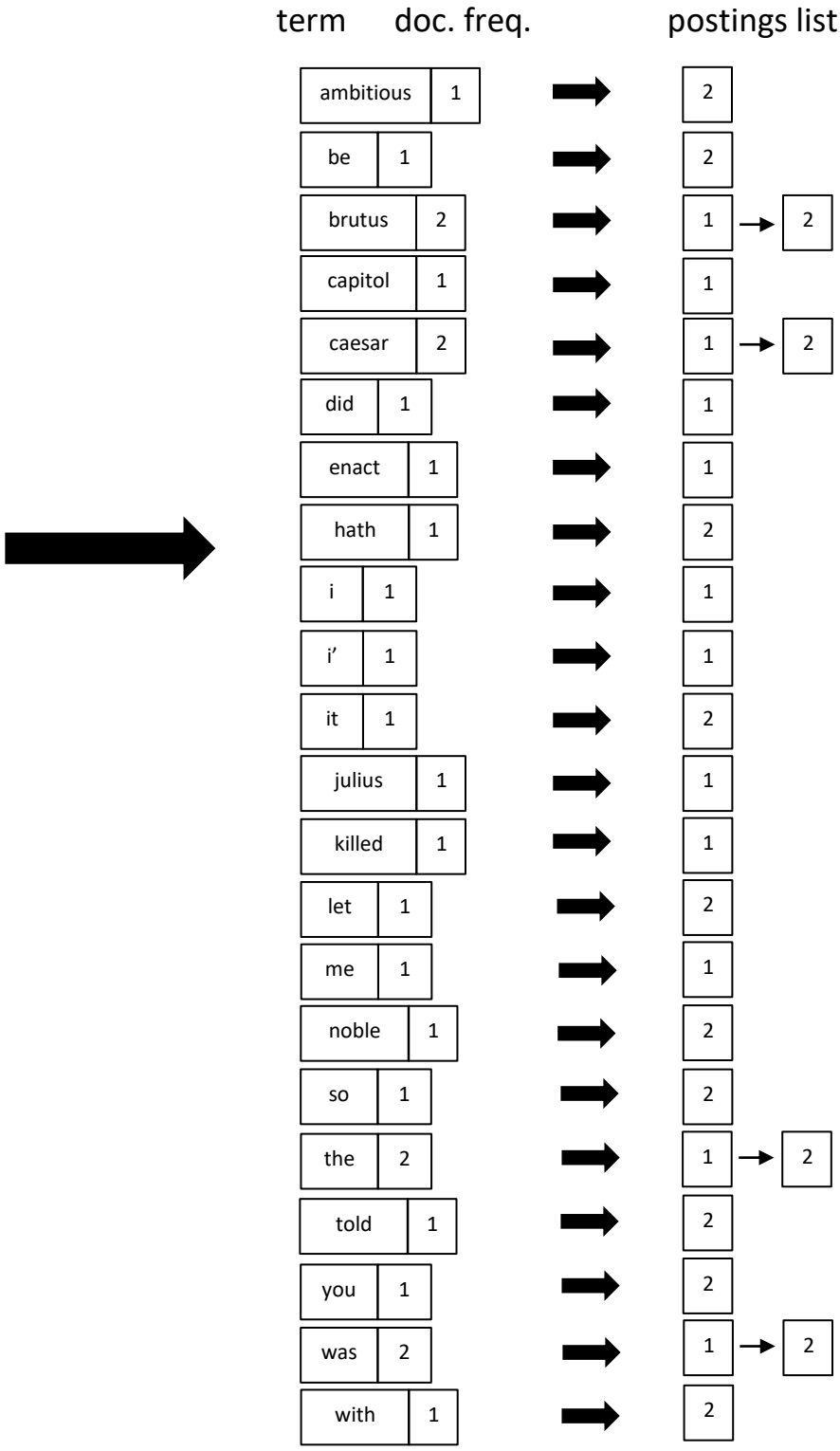
Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i’	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
Caesar	2
was	2
ambitious	2

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
I’	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

و در نهایت در مرحله **سوم** (مرحله آخر) ماتریس عبارت – شناسه سند مرتب شده به دو قسمت دیکشنری و پُستنگ‌ها تقسیم می‌شود. با این تفاوت که یک ستون دیگر به نام **فرکانس سند** (Doc. frequency) به دیکشنری اضافه خواهد شد.

«فرکانس سند یک کلمه مشخص می‌کند آن کلمه داخل چند سند وجود دارد»

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
I’	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2



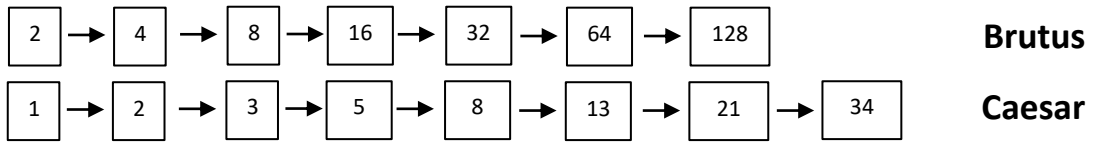
همانطور که مشاهده می‌کنید کلمه brutus هم درسند اول و هم در سند دوم وجود دارد بنابراین دارای فرکانس ۲ می‌باشد. اما موتورهای جست‌وجو چگونه از شاخص‌گذاری معکوس برای جست‌جو استفاده می‌کنند؟

پردازش جست‌جو با شاخص‌گذاری معکوس:

حال که با مراحل شاخص‌گذاری سندها آشنا شدید سوال اصلی اینجاست که چگونه موتورهای جست‌جو می‌توانند پس از شاخص‌گذاری معکوس به ما پاسخ جست‌جو را برگردانند. بگذارید با یک مثال بررسی کنیم.

فرض کنید می‌خواهیم ببنیم دو عبارت Brutus و Caesar با هم در چه سندهایی وجود دارند.

در اولین قدم عبارت Brutus را در دیکشنری پیدا و پستینگ آن را بازیابی می‌کنیم. در قدم دوم به دنبال عبارت Caesar در دیکشنری می‌گردیم و پستینگ آن را نیز بازیابی می‌کنیم. در نهایت عمل **ادغام** (Merge) را میان دو پستینگ انجام می‌دهیم (اشتراک‌گیری میان دو پستینگ). فرض کنید پستینگ لیست هر دو عبارت به صورت زیر باشد:



در چنین حالتی می‌توان با یک نگاه تشخیص داد در سند شماره ۲ و سند شماره ۸ دو عبارت Brutus و Caesar با هم وجود دارند. این عمل همان اشتراک‌گیری میان دو پستینگ (ادغام) است.

ادغام (Merge):

همانطور که در مثال قبل دیدید در پردازش جست‌جو با شاخص‌گذاری معکوس اگر بخواهیم دو یا جند عبارت را بیابیم که با یکدیگر در یک سند وجود دارند یا خیر، از عمل ادغام استفاده می‌کنیم. ادغام همان اشتراک‌گیری میان دو یا چند پستینگ لیست است. برای انجام ادغام تمام پستینگ‌ها را به صورت همزمان پیمایش می‌کنیم و به دنبال اشتراک‌ها می‌گردیم.

اگر طول لیست اول را X و طول لیست دوم را Y بنامیم، برای انجام عمل ادغام O(X+Y) عمل صورت خواهد گرفت (به تعداد عناصر آرایه بستگی دارد).

نکته مهمی که باید به خاطر داشت این است که پستینگ‌ها بر اساس شناسه سند مرتب شده هستند.

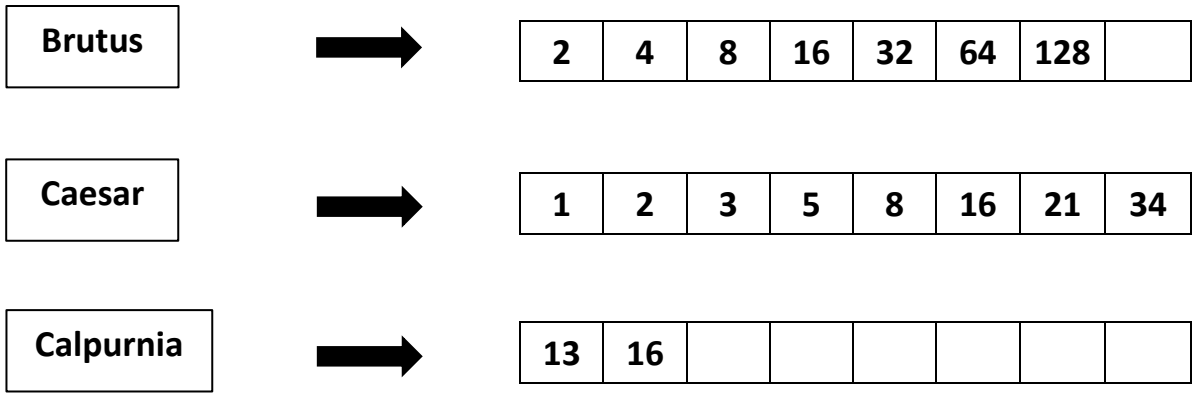
بهینه‌سازی جست‌جو (Query Optimization):

در بخش‌های قبلی مثال زدیم اگر بخواهیم ببینیم دو عبارت Brutus و Caesar آیا در یک سند وجود دارند یا خیر باید پس از بررسی دیکشنری و یافتن دو عبارت، پستینگ لیست آن‌ها را نیز بازیابی کنیم و عمل ادغام را انجام دهیم. حال فرض کنید ما جست‌جویی با تعداد $n > 2$ عبارت داریم که باید به تعداد عبارت‌ها (n) عمل ادغام را انجام دهیم.

بهترین ترتیب برای پردازش یک جست‌جو چیست؟

بگذارید با یک مثال به این مسئله بپردازیم. فرض کنید یک جست‌جو به صورت زیر داریم که هر عبارت در دیکشنری دارای پستینگ لیست می‌باشد:

Query: **Brutus** And **Caesar** And **Calpurnia**



روشی که به منظور بهینه‌سازی جست‌جو استفاده می‌شود مبتنی بر **فرکانس سند** است. اگر خاطرتان باشد در دیکشنری ما به همراه عبارت، فرکانس سند را نیز اضافه می‌کردیم که در بهینه‌سازی جست‌جو به ما کمک می‌کند. در حقیقت در عمل ادغام از کوچکترین مجموعه به بزرگترین مجموعه می‌رویم و اشتراک می‌گیریم.

«در حقیقت پردازش جست‌جو به ترتیب افزایش فرکانس خواهد بود»

بنابراین جست‌جو به صورت (**Calpurnia** And **Brutus**) And **Caesar** اجرا خواهد شد.

اما اگر در جست‌جو به جای استفاده از And ، از Or استفاده کنیم چه خواهد شد؟

به عنوان مثال فرض کنید جست‌جو ما به صورت زیر باشد:

Query: (**madding** Or **crowd**) And (**ignoble** Or **strife**)

در این صورت ابتدا فرکانس‌های سند هر یک از عبارت‌ها را بازیابی می‌کنیم. سپس فرکانس سند عباراتی که با هم OR شده‌اند را با هم جمع می‌کنیم. در نهایت همانند مثال قبل بر اساس افزایش فرکانس پردازش می‌کنیم و کمترین‌ها را با هم اشتراک می‌گیریم.

پردازش اسناد (Parsing Document):

اولین کاری که یک موتور جست‌جو انجام می‌دهد پردازش سند است. یعنی با بررسی کردن سند به سه سوال پاسخ می‌دهد:

۱- قالب یا فرمت استفاده شده در آن چیست؟ (pdf/word/excel/html)

۲- چه زبانی یا زبان‌هایی در سند استفاده شده؟ (فارسی، انگلیسی، آلمانی، ژاپنی و ...)

۳- از چه ترکیب کارکترهایی در سند استفاده شده؟ (بسته به زبان سیستم متفاوت است)

جلسه سوم

توکن‌بندی (Tokenization):

جلسه گذشته در خصوص روش شاخص‌گذاری معکوس صحبت کردیم. یکی از مراحل این روش پیدا کردن توکن‌ها در یک سند است. توکن در حقیقت دنباله‌ای از کارکتر در سند می‌باشد که پس از نرمال‌سازی در دیکشنری به عنوان یک عبارت (Term) ذخیره می‌شود. اما سوال اینجاست که چگونه کلمه‌های درون سند را به عنوان توکن برگردارنیم؟ شرایط تشخیص توکن به چه صورت است؟

بیایید تا با چند مثال مشکلاتی که ممکن است در تشخیص توکن پیش رو داشته باشیم را بررسی کنیم:

به عنوان مثال فرض کنید می‌خواهیم کلمه Finland’s capitol را بررسی کنیم. آیا می‌توانیم تنها عبارت Finland به عنوان توکن درنظر بگیریم؟ کلمه Finlands یا Finland’s ؟ کدام یک را انتخاب کنیم؟

یا کلمه Hewlett-Packard را در نظر بگیرید. آیا باید کلمه Hewlett و Packard را به صورت دو توکن مجزا درنظر بگیریم؟ یا باید با هم به عنوان یک توکن درنظر بگیریم؟

اگر بخواهیم اعداد را نیز بررسی کنیم باید از چه قالب یا فرمتی استفاده کنیم؟ به عنوان مثال ترتیب اعداد 15/3/2021 را به عنوان تاریخ بپذیریم یا 3/15/2021 یا *Mar. 15,2021* ؟

اما مسئله تنها به حروف و اعداد انگلیسی ختم نمی‌شود. اگر بخواهیم یک موتور جست‌جو را بین‌المللی کنیم باید بتوانیم کلمات و قواعد زبان‌های مختلف را نیز درنظر بگیریم. به عنوان مثال کلمه فرانسوی *L’ensemble* را درنظر بگیرید. آیا می‌توانیم عبارت *L’* را یک توکن درنظر بگیریم؟ *L* یا *Le* چطور؟

جالب است موتور جست‌جوی گوگل تا قبل از سال ۲۰۰۳ میلادی تنها زبان انگلیسی را تشخیص می‌داد و امکان پاسخ به زبان‌های دیگر را نداشت.

مثال از زبان آلمانی (عدم وجود فاصله میان کلمات):

- Lebensversicherungsgesellschaftsangestellter

‘life insurance company employee’

مثال از زبان چینی (عدم امکان تعیین توکنی متمایز):

莎拉波娃现在居住在美国东南部的佛罗里达。

مثال از زبان فارسی، عربی یا عبری (حروف از راست به چپ اما عداد از چپ به راست):

استقلت الجزائر في سنة 1962 بعد 132 عاما من الاحتلال الفرنسي.

کلمات اضافی (Stop Words):

کلمات اضافی کلماتی هستند که به خودی خود دارای معنی نیستند و برای ارتباط برقرار کردن در جملات استفاده می‌شوند (مانند the ، a ، and ، to ، be و ...). کلمات اضافی معنای کم و فراوانی بالایی (تقریبا ۳۰ درصد پستینگ‌ها در ۳۰ کلمه اول) دارند. در ابتدا تصمیم به حذف کلمات اضافی گرفته شد اما با بالا رفتن سطح فشرده‌سازی و همچنین بهینه‌سازی خوب جست‌جوها از حذف آن‌ها صرف نظر شد. اگرچه نگهداری کلمات اضافی باعث کمبود فضای ذخیره‌سازی می‌شود اما در مواقعی به آن‌ها نیاز داریم و مفید واقع می‌شوند. از جمله:

- جست‌جوهای عبارتی (Phrase queries) : به عنوان مثال "King of Denmark"
- عنوان‌ها و عبارت‌های خاص (اشعار و ...) : به عنوان مثال "To be or not to be"
- جست‌جوهای رابطه‌ای (Relational queries) : به عنوان مثال "flights to London"

«پس اولین مرحله‌ای که موتور جست‌جو بعد از پردازش سندانجام می‌دهد، توکن‌بندی و جدا کردن کلمات بر اساس زبان استفاده شده در سند می‌باشد»

نرمال‌سازی کلمات (Normalization):

پس از مرحله توکن‌بندی مرحله نرمال‌سازی آغاز می‌شود. نرمال‌سازی یعنی تبدیل یک کلمه به قالب یا فرمتی مشخص (قالب بقیه کلمات). **عبارت** (Term) که در موتور جست‌جو به عنوان یک ورودی برای دیکشنری به شمار می‌آید در حقیقت یک نوع کلمه نرمال‌سازی شده است. معمولا برای تعریف و طبقه‌بندی هم‌ارزی از علامت کاما (,) استفاده می‌کنیم. به عنوان مثال:

- حذف نقطه برای ایجاد قالب عبارت: *U.S.A , USA* ⟵ *USA*
- حذف خط فاصله (-) برای ایجاد قالب عبارت: anti-discriminatory , antidiscriminatory ⟵ antidiscrimantory

نرمال‌سازی در دیگر زبان‌ها:

به عنوان مثال در زبان فرانسوی با علامت‌گذاری بر روی حروف، گویش آن حرف متفاوت خواهد شد. گویش کلمه ***résumé*** با کلمه ***resume*** متفاوت است. بهترین راه این است تا عبارت بدون گویش و خالص ذخیره شود. در این صورت موتور جست‌جو علامت‌ها را حذف کرده و کلمه ***resume*** را به عنوان عبارت در نظر می‌گیرد.

تبدیل به حروف کوچک (Case folding):

یکی دیگر از روش‌های نرمال‌سازی تبدیل حروف کلمه به حروف کوچک (Lower Case) است. به عنوان مثال تبدیل *Fed* به *fed* یا تبدیل *SAIL* به *sail* و غیره. این عمل مفید است چرا که تمام کلمات به یک فرم یا قالب ذخیره می‌شوند اما به عنوان مثال فرض کنید می‌خواهیم کلمه C.A.T که بیانگر شرکت **کاترپیلار** می‌باشد را در گوگل جست‌جو کنیم. در ابتدا موتور جست‌جو برای ذخیره این کلمه نقاط آن را حذف کرده و حروف را به حروف کوچک تبدیل می‌کند (کلمه C.A.T به cat تبدیل خواهد شد). طبیعی است اولین نتیجه جست‌جو برای عبارت **گربه** (cat) خواهد بود چرا که پس از نرمال‌سازی کلمه جست‌جو به cat تبدیل شد.

بنابراین با اینکه نرمال‌سازی باعث تغییر ناخواسته در نتیجه جست‌جوخواهد شد، روش بسیار مفیدی به منظور بهینه کردن اطلاعات خواهد بود.

روش جایگزین:

یک راه جایگزین برای طبقه‌بندی هم‌ارزی را می‌توانیم **امتداد نامقارن** (asymmetric expansion) بنامیم. به عنوان مثال:

- کلمه وارد شده: window جست‌جو: windows, window
- کلمه وارد شده: windows جست‌جو: window, windows, Windows
- کلمه وارد شده: Windows جست‌جو: Windows

درست است که این روش، روش قدرمتندی است اما نمی‌تواند کارآمد باشد.

اصطلاحنامه (Thesauri):

آیا ما می‌توانیم مترادف‌ها و همنام‌ها را نیز مدیریت کنیم؟

به عنوان مثال فرض کنید ما دو طبقه هم‌ارزی به صورت زیر داریم:

car = automobile *color = colour*

در چنین شرایطی می‌توانیم دو عمل را انجام دهیم:

۱- می‌توانیم به صورت مجدد به قالب طبقه‌بندی هم‌ارزی تبدیل کنیم (به عنوان مثال اگر سند شامل کلمه automobile بود در پایینِ عبارت car-automobile شاخص‌گذاری کنیم یا برعکس).

۲- می‌توانیم جست‌جو را امتداد دهیم. (به عنوان مثال اگر سند شامل کلمه automobile بود به دنبال کلمه car نیز بگردیم).

تبدیل نوع (Lemmatization):

یکی دیگر از مرحله‌ها برای نرمال‌سازی، تبدیل نوع کلمات است. تبدیل نوع یعنی کاهش انواع فرم‌ها و نوع‌های یک کلمه به یک فرم پایه. به عنوان مثال کلمات am و is و are به کلمه be تبدیل خواهند شد. یا کلمات car ، cars ، car's و cars' به کلمه car تبدیل خواهند شد.

مثال:

- the boy's cars are different colors → the boy car be different color

ریشه‌یابی (Stemming):

ریشه‌یابی یعنی تبدیل عبارت‌ها (Term) به ریشه کلماتشان قبل از شاخص‌گذاری که به نوع زبان بستگی دارد.

به عنوان مثال کلمات automate(s) ، automatic و automation به کلمه automat تبدیل می‌شوند.

مثال:

for example compressed and compression are both accepted as equivalent to compress.

Google

for exampl compress and compress ar both accept as equival to compress

الگوریتم پورتر (Porter's algorithm):

یکی از الگوریتم‌های معروف برای پیدا کردن ریشه کلمات انگلیسی، الگوریتم پورتر می‌باشد. خروجی الگوریتم به اندازه کافی نسبت به بقیه روش‌های ریشه‌یابی خوب هست. در اینجا نمی‌خواهیم الگوریتم را کامل توضیح دهیم بلکه با چند مثال می‌خواهیم عملکرد آن را بررسی کنیم.

بخشی از قوانین الگوریتم پورتر به صورت زیر است:

- *sses* → *ss* (تبدیل جمع به مفرد)
- *ies* → *i* (تبدیل جمع به مفرد)
- *ational* → *ate* (تبدیل کلمه به مصدر خودش)
- *tional* → *tion* (تبدیل کلمه به مصدر خودش)

نکته‌ای که باید به خاطر داشته باشیم حساسیت قوانین الگوریتم به طول کلمات است. به عنوان مثال اگر ما طول کلمه را m بنامیم، با تغییر مقدار m قانون زیر دو خروجی متفاوت خواهد داشت. اگر طول کلمه از 1 بیشتر باشد و در آخر کلمه ement وجود داشته باشد، ement را حذف می‌کند اما اگر طول کلمه برابر 1 باشد آن را تغییر نمی‌دهد:

- *(m > 1) EMENT*
  - *replacement* → *replac*
  - *cement* → *cement*

آیا ریشه‌یابی و بقیه روش‌های نرمال‌سازی مفیداند؟

برای پاسخ به این سوال باید گفت بله. درست است که ریشه‌یابی و دیگر روش‌های نرمال‌سازی باعث کاهش دقت در کار خواهند شد اما در نهایت موجب کم شدن حجم اطلاعات برای ذخیره‌سازی می‌شوند. در دنیای امروزی سرعت و فضای ذخیره‌سازی از مهمترین عوامل هستند. با افزایش سرعت و دقت فضای به ذخیره‌سازی بیشتری احتیاج داریم و برعکس با مدیریت کردن و بهینه کردن حافظه، سرعت و دقت را پایین می‌آوریم.

همچنین ممکن است در زبان انگلیسی شاهد تغییرات چشمگیری نباشیم اما تجربه نشان می‌دهد این روش‌ها برای زبان‌های دیگر ازجمله اسپانیایی، آلمانی، فنلاندی و غیره، باعث افزایش ۳۰ درصدی کارآیی خواهد شد.

جلسه چهارم

اشاره‌گرهای فرار یا اشاره‌گرهای لیست (Skip pointers/skip lists):

در جلسات گذشته در خصوص روش شاخص‌گذاری معکوس صحبت کردیم. گفتیم برای پیدا کردن دو یا چند عبارت از عمل ادغام (Merge) استفاده می‌کنیم. در حقیقت میان پستینگ هر عبارت به طور همزمان اشتراک می‌گیریم که مدت این اشتراک به اندازه لیست‌های ما (پستینگ‌ها) بستگی داشت.

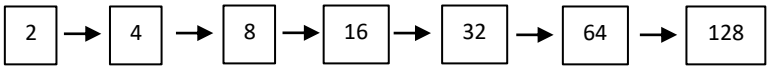
اما سوال اینجاست آیا می‌توانیم این عمل را بهتر انجام دهیم؟

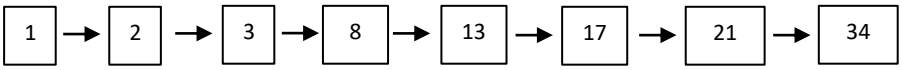
باید گفت بله اما اگر شاخص‌ها خیلی سریع تغییر نکنند.

تقویت پستینگ‌ها با اشاره‌گرهای فرار در هنگام شاخص‌گذاری:

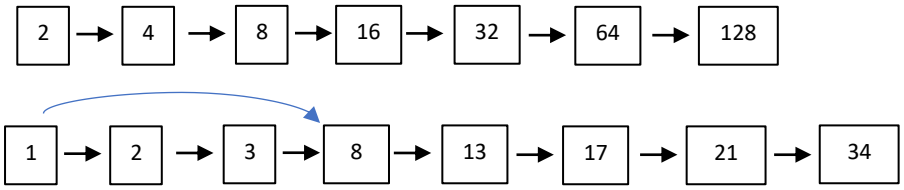
می‌توانیم برای سریعتر کردن عمل اشتراک‌گیری از **اشاره‌گرهای فرار** (Skip pointers) استفاده کنیم. دلیل اصلی استفاده از اشاره‌گرهای فرار، پرش از روی شناسه سندهایی است که در جست‌جو ما تاثیری نخواهند داشت.

بگذارید تا با یک مثال با عملکرد آن‌ها آشنا شویم. فرض کنید دو پستینگ زیر را داریم.

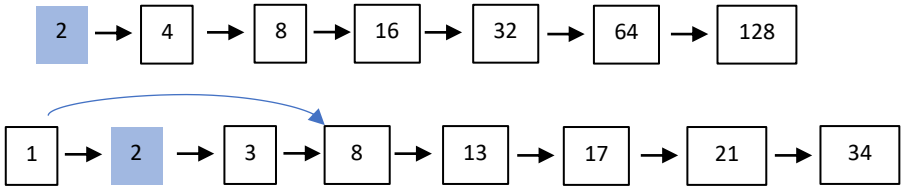




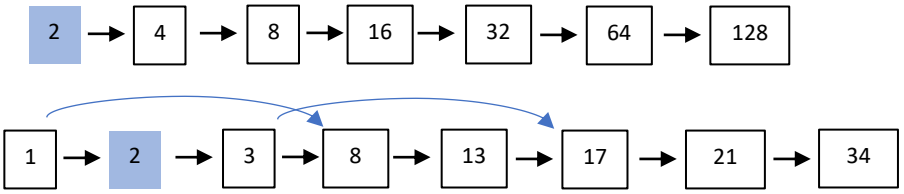
در ابتدا دو خانه اول مورد بررسی قرار می‌گیرند (1 و 2). از آنجایی مقدار این دو خانه با یکدیگر برابر نیست، پرش از لیستی که مقدار خانه اولش کمتر بوده آغاز می‌شود. سپس به اندازه تعدادی که برای پرش تعیین شده، پرش را انجام می‌دهد. فرض کنید مقدار پرش را ۲ در نظر گرفته ایم. با پرش به اندازه ۲ خانه از عدد 1 به عدد 8 خواهیم رسید.



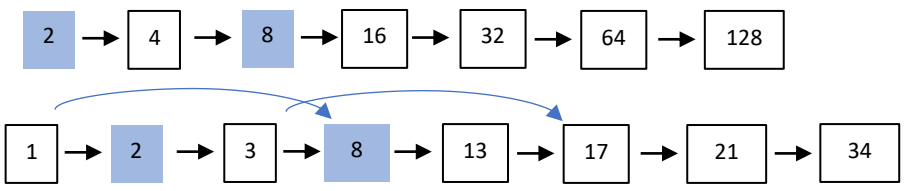
حال مقدار خانه اول در لیست اول (2) را با مقدار خانه‌ای که در لیست دوم با پرش به آن رسیده (8) بررسی می‌کند. از آنجایی که مقدار 8 از 2 بزرگتر است عمل پرش موفقیت‌آمیز نمی‌باشد. زیرا احتمال آنکه مقدار 2 در لیست دوم قبل از خانه‌ای با مقدار 8 باشد وجود دارد. در این حالت در لیستی که پرش صورت گرفته باید تمام مقادیر قبل از پرش (قبل از 8) با مقدار 2 بررسی شوند بنابراین در لیست دوم یک خانه به جلو می‌رویم. حال مقدارخانه دوم در لیست دوم (2) با مقدار خانه اول در لیست اول برابر است که در این صورت اولین اشتراک را تشخیص می‌دهد.



با هر اشتراک بدست آمده در هردو لیست یک خانه به جلو می‌رود. بنابراین در مرحله بعدی خانه‌های 3 و 4 را مورد بررسی قرار می‌دهد. از آنجایی که مقدار 4 از 3 بزرگتر است، پس مجددا در لیست به اندازه مقدار پرش به جلو می‌رود که از مقدار 3 به مقدار 17 می‌رسد. مجدد مقدار 17 را با مقدار 4 بررسی می‌کند.



از آنجایی که مقدار 17 از 4 بزرگتر است پس پرش موفقیت آمیز نبوده و مجددا خانه‌های قبلی مورد بررسی قرار می‌گیرند. از خانه 3 یک خانه به سمت جلو می‌آید (خانه 8). حال چون مقدار 8 از 4 نیز بزرگتر پس ممکن نیست خانه‌ای با مقدار 4 وجود داشته باشد. بنابراین لیست اول را از خانه 4 به خانه 8 انتقال می‌دهد که در این صورت اشتراک دوم را پیدا می‌کند.



این عملیات تا زمانی که به انتهای لیست اول برسیم ادامه می‌یابد.

در بعضی اوقات عمل پرش باعث می‌شود تا برخی عملیات انجام نشود اما از طرفی اگر پرش موفقیت‌آمیز نباشد باید به عقب برگردیم و بررسی کنیم. این روش می‌تواند تا حدودی سرعت را بهبود بدهد.

حال سوال اینجاست مقدار پرش را باید چقدر درنظر بگیریم؟

اگر مقدار پرش را **زیاد** انتخاب کنیم احتمال آنکه از خانه‌هایی که ممکن است اشتراک داشته باشند رد شویم و مجبور شویم برگردیم زیاد است اما اگر مقدار پرش را **کم** انتخاب کنیم تعداد مقایسه‌هایمان افزایش می‌یابد. بنابراین بهتر است تا حد **وسطی** را برای مقدار پرش درنظر بگیریم.

برای محاسبه حد وسط کافیست تا جذر تعداد خانه‌های پستینگ را محاسبه کنیم.

«پس تعداد پرش به تعداد خانه‌ها (L) بستگی دارد که می‌شود از $\sqrt{L}$ آن را بدست آوریم»

اشتراک‌گیری پستینگ‌ها با اشاره‌گرهای فرار:

الگوریتم زیر الگوریتم اشتراک‌گیری با استفاده از روش اشاره‌گرهای فرار می‌باشد که در بالا با یک مثال عملکرد آن را توضیح دادیم:

```
INTERSECTWITHSKIPS(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then ADD(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8                then if hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
 9                        then while hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
10                                do p1 ← skip(p1)
11                        else p1 ← next(p1)
12                else if hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
13                        then while hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
14                                do p2 ← skip(p2)
15                        else p2 ← next(p2)
16  return answer
```

در این الگوریتم اشتراک میان پستینگ لیست P1 و P2 داخل آرایه‌ای به نام answer ذخیره می‌شود. مطابق الگوریتم تا زمانی که به پایان دو پستینگ نرسیدیم، شناسه‌های سند دو پستینگ با هم مقایسه خواهند شد. زمانی که شناسه سند مشترک پیدا شود جواب را داخل answer قرار می‌دهد.

جست‌جوهای عبارتی (Phrase queries):

تا به اینجا ما در خصوص جست‌جوهای تک کلمه‌ای بسیار صحبت کردیم. اگر به دنبال یک یا چند کلمه در چندین سند باشیم از روش‌های مختلفی استفاده می‌کنیم اما اگر جست‌جوی ما تک کلمه‌ای نباشد و ما بخواهیم یک عبارت مانند "stanford university" را پیدا کنیم باید از چه روشی استفاده کنیم؟ دقت کنید ما به دنبال همین ترکیب دو کلمه‌ای هستیم. یعنی جمله "I went to university at Stanford" پاسخ ما نخواهد بود.

اندیس‌های دوکلمه‌ای (Biword indexes):

اولین روشی که برای حل مشکل جست‌جوهای عبارتی پیشنهاد شد استفاده از اندیس‌های دوکلمه‌ای بود. یعنی داخل دیکشنری به جای ذخیره شدن یک کلمه، دو کلمه ذخیره خواهد شد. به عنوان مثال متن "Friends, Romans, Countrymen" به صورت friends romans و romans countrymen دوکلمه‌ای خواهد بود. هر کدام از این دوکلمه‌ها به یک عبارتی (Term) از دیکشنری تبدیل شده‌اند. حال پردازش جست‌جوهای دوکلمه‌ای بسیار ساده‌تر خواهد شد. اما با بالا رفتن تعداد کلمات جست‌جو این روش کارآمد نیست.

«استفاده از ایندکس دو کلمه‌ای فقط در جست‌جوهای دوکلمه‌ای کاربرد دارد»

مثال:

- *stanford university palo alto*

پس از شکست متن به جست‌جو دوکلمه‌ای

stanford usiversity *AND* ***university palo*** *AND* ***palo alto***

پاسخ این جست‌جو ممکن است درست و یا غلط باشد.

دوکلمه‌های گسترش‌یافته (Extended biwords):

روشی که برای جست‌جوهای عبارتی پیشنهاد شد که روش بهبود یافته اندیس‌های دوکلمه‌ای است، استفاده از دوکلمه‌های گسترش‌یافته بود. در این روش عبارت‌ها را به دو قسمت اسم (Noun) یا N و حروف اضافه (prepositions) یا X تقسیم کردند. در این صورت هر رشته‌ای از عبارات که به فرم NX*N باشد (* یعنی چه N باشد و چه X) را یک دوکلمه‌ای گسترش‌یافته می‌نامیم که به صورت یک عبارت در دیکشنری ذخیره می‌شود.

مثال:

- ***catcher in the rye***

N X X N

پردازش جست‌جو این متن را به N ها و X ها تبدیل می‌کند. سپس عنصر اصلی (اسامی) را به دوکلمه‌ای تبدیل می‌کند و به صورت catcher rye جست‌جو خواهد شد.

از اشکلات این روش می‌توان به برگرداندن پاسخ نامناسب و غلط و همچنین اشغال فضای بیشتر حافظه نام برد. در حقیقت اندیس‌های دوکلمه‌ای راه‌حل استانداردی نیستند اما می‌توانند به عنوان قسمتی از یک راه‌حل استفاده شوند.

اندیس‌های مکانی(Positional indexes):

دومین روشی که پیشنهاد شد استفاده از اندیس‌های مکانی بود. در اندیس‌های مکانی غیر از کلمه و سندی که کلمه در آن هست، مکان آن کلمه در آن سند را نیز مشخص می‌کند. قالب ذخیره شدن آن به صورت زیر است:

<**عبارت**, تعداد سندهایی که **عبارت** را در خود دارند (فرکانس);

doc1: جایگاه اول , جایگاه دوم , ... ;

doc2: جایگاه اول , جایگاه دوم , ... ;

.

.

.

>

مثال: در کدام سند ۱ و ۲ و ۴ و ۵ جمله "to be or not to be" وجود دارد؟

<***be***: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>

در این مثال عبارت be به در 993427 سند وجود دارد. در سند اول ۶ بار تکرار شده است در سند ۲ نیز ۲ بار و به همین ترتیب. اگر کلمه be اول را شماره ۱ بنامیم این کلمه تا be دوم ۴ کلمه فاصله دارد. در سند ۴ اندیس مکانی 430 و 434 دقیقا ۴ تا کلمه با یکدیگر فاصله دارند. همچنین در سند شماره ۵ نیز اندیس مکانی 363 و 367 نیز به همین صورت است. ممکن است این جمله در سندهای شماره ۴ و ۵ وجود داشته باشند.

«الگوریتم اندیس مکانی دقت بالایی دارد اما حافظه را بیشتر اشغال می‌کند»

فرض کنید یک صفحه وب به طور میانگین (تعداد کلمات ما در یک سند) دارای ۱۰۰۰ عبارت باشد و فرکانس یک عبارت داخل سند برابر 0.1% باشد (یعنی به ازای هر ۱۰۰۰ کلمه‌ای که وجود دارد این کلمه نیز وجود دارد). حال فرض کنید یک صفحه دیگر به راحتی به ۱۰۰/۰۰۰ عبارت می‌رسد. بنابراین این کلمه با فرکانس 0.1% به تعداد ۱۰۰ بار در این صفحه تکرار شده است.

در پستینگ ما تنها به یک فضا برای یک عبارت احتیاج داشتیم اما در اندیس معکوس با ۱۰۰ بار تکرار کلمه در یک صفحه به ۱۰۰ تا حافظه برای ذخیره کردن آن عبارت نیاز داریم.

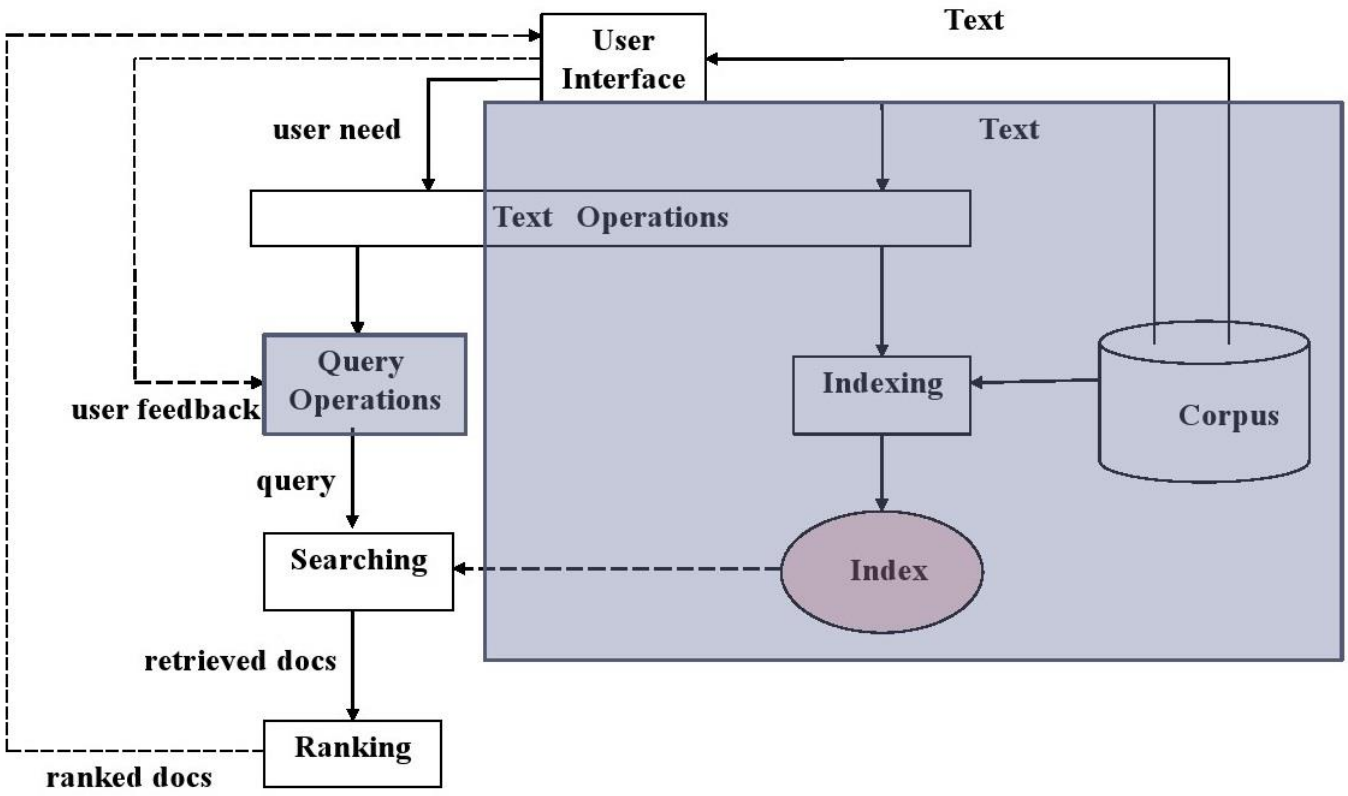
بنابراین:

- اندیس مکانی ۲ الی ۴ برابر بیشتر از اندیس معمولی فضا می‌گیرد.
- نسبت به ذخیره سند اصلی ۳۵ الی ۵۰ درصد بیشتر نیاز به حافظه دارد.
- این اعداد و محاسبات تنها برای زبان‌های مشابه انگلیسی خواهد بود.

«موتورهای جست‌جو معمولا ترکیبی از روش‌های اندیس دوکلمه‌ای، اندیس مکانی و غیره استفاده می‌کنند»

جلسه پنجم

معماری یک سیستم IR (Information Retrieval):



یک دید کلی از معماری یک موتور جست‌جو در شکل بالا را مشاهده می‌کنید. قسمت User Interface همان صفحات وب است. عملیاتی روی متن موجود در صفحات وب انجام می‌شود و صفحه وب اندیس‌گذاری می‌شود. قسمت Text Operations تمام عملیات‌هایی از جمله ریشه‌یابی کلمات و تمام مراحل نرمال‌سازی آن‌ها را شامل می‌شود. پس از توکن‌بندی کلمات این عملیات انجام خواهد شد. سپس با استفاده ازCorpus (دیکشنری موتور جست‌جو) اندیس عبارت را بیرون کشیده و هر زمان کاربر یک پرسش انجام دهد، با استفاده از اندیس به دنبال آن می‌گردد.

قسمت Ranking به منظور رتبه‌دادن به صفحات وب است. الگورتیم‌هایی تعریف شده است تا به صفحات وب بسته به تقاضای کاربران رتبه داده شود. اینکار باعث می‌شود تا صفحات وب بر اساس رتبه به کاربران نمایش داده شود.

باید توجه داشت که در اینجا اندیس ما دارای دو قسمت است. یک قسمت دیکشنری (Corpus) و یک قسمت لیست پستینگ (Posting list) می‌باشد.

حال به علت بالا بودن حجم اطلاعات نیاز داریم تا از یک ساختار داده استفاده کنیم. موتورهای جست‌جو از دو روش برای ذخیره‌سازی اندیس‌هایی که ساخته‌اند استفاده می‌کنند. یک روش استفاده از جدول‌های هَش (Hashtables) و روش دیگر استفاده از درخت‌ها (Trees) می‌باشد.

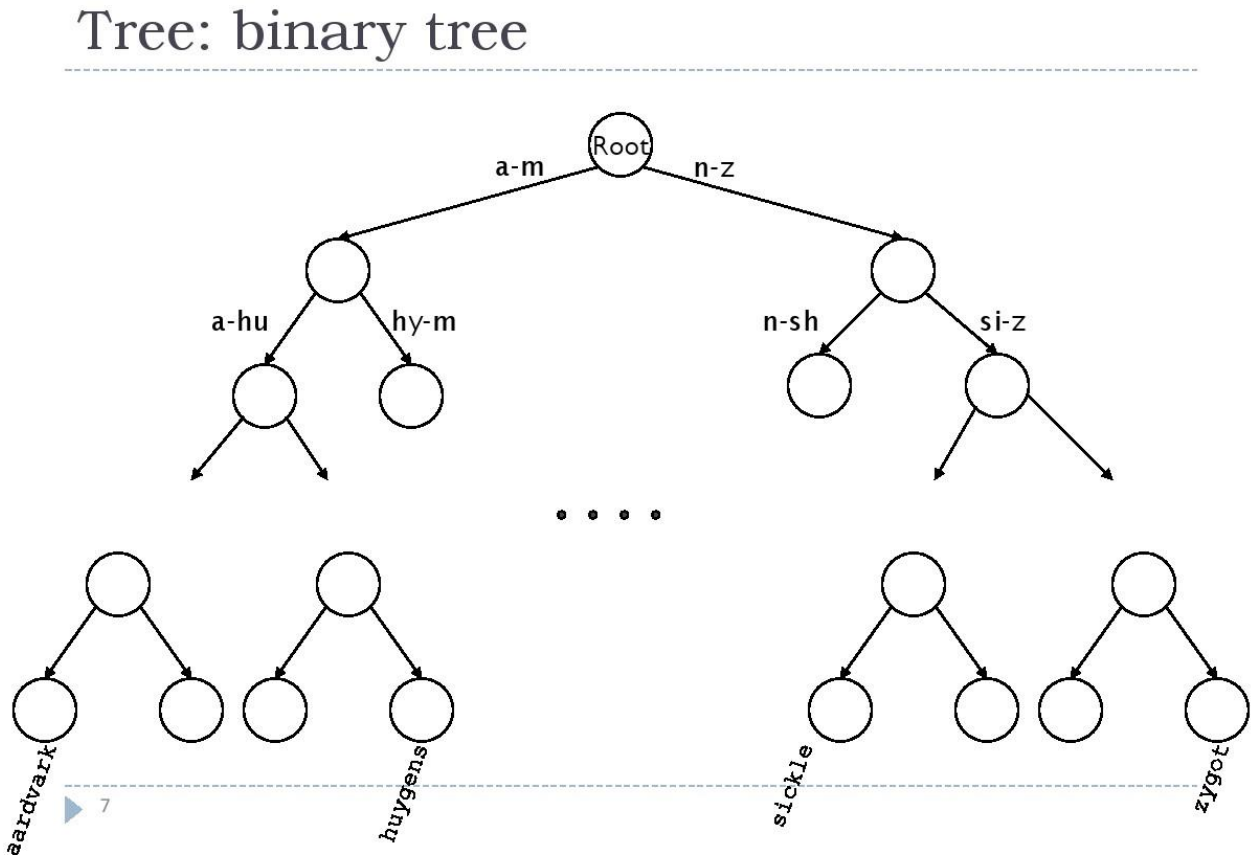
در ادامه به بررسی نحوه ذخیره‌سازی دیکشنری با استفاده از درخت دودویی خواهیم پرداخت.

جدول‌های هَش (Hashtables):

در جدول هش هر عبارت داخل دیکشنری را به یک عدد صحیح (Integer) تبدیل می‌شود و به همان صورت ذخیره می‌گردد که در این حالت سرعت بررسی بالا می‌رود.

اما به علت اینکه کلمات به عدد تبدیل می‌شوند در صورت اشتباه نوشتن کلمات امکان اصلاح آن‌ها وجود ندارد. برخی از کلمات نیز دارای پیشوند (prefix) هستند که در این صورت امکان تشخیص آن برای ما فراهم نمی‌باشد. همچنین به علت وابستگی عملیات هش به همدیگر در صورت اضافه شدن کلمه‌ای به دیکشنری باید مجددا عملیات هش را انجام دهیم.

درخت دودویی (binary tree):



درخت دودویی به منظور ذخیره‌سازی دیکشنری استفاده می‌شود. دارای یک ریشه (Root) می‌باشد که هر ریشه دارای دو شاخه است. این شاخه‌ها تا زمانی ادامه می‌یابند که به برگ‌ها برسند. داخل برگ‌ها کلمات ذخیره می‌شوند. اگر درخت را از سمت راست پیمایش کنیم کلماتی که اولین حرف آن‌ها در بازه n – z خواهد بود را می‌یابیم و اگر از سمت چپ پیمایش کنیم در بازه a – m را خواهیم یافت.

راحت‌ترین حالت استفاده از درخت دودویی برای ذخیره‌سازی است اما روش کارآمدتر که اغلب موتورهای جست‌جو از آن استفاده می‌کنند درخت متعادل یا متوازن (B-tree) می‌باشد.

استفاده از درخت‌ها باعث می‌شود مشکل پیشوندها که در جدول هش به آن برخوردیم حل شود چرا که تمام کلمه در درخت ذخیره شده و با داشتن ابتدای کلمه می‌توان کلمه را پیدا کرد. اما درخت‌ها نسبت به جدول هش کندتر عمل می‌کنند چون باید در درخت به دنبال کلمات بگردیم. همچنین متعادل کردن درخت‌ها (برابری سمت چپ و راست) هزینه‌بر خواهد بود. البته متعادل کردن درخت در درخت‌های متوازن راحت‌تر می‌باشد.

جست‌جوهای Wild-card (* ستاره‌دار):

علامت آستریکس یا ستاره (*) در کامپیوتر به معنای **همه چیز** یا **هر چیزی** است. به عنوان مثال فرض کنید عبارت *mon را میخواهید جست‌جو کنید. کامپیتر متوجه می‌شود شما به دنبال تمام کلماتی هستید که با mon شروع بشود (مانند monday). معمولا از این روش زمانی استفاده می‌کنیم که ابتدای کلمه را می‌دانیم اما آخر آن را نمی‌دانیم. در این صورت پس از وارد کردن عبارت موتور جست‌جو در دیکشنری به دنبال آن می‌گردد سپس اندیس‌سازی می‌کند و به دنبال آن در صفحات وب می‌گردد.

اما سوال اینجاست اگر کاربر چنین عبارتی را وارد کند ما چگونه می‌توانیم آن را پیدا کنیم؟

در درخت دودویی (یا درخت متوازن) اینکار به سادگی امکان‌پذیر است چرا که باید به دنبال شاخه‌ای بگردیم که محدوده آن mon <= w < moo باشد. به منظور تعیین محدوده مطابق حروف الفبا حرف بعدی آخرین حرف کلمه mon (یعنی n) را یک واحد به جلو می‌برد(بعد از n حرف o می‌باشد) حال که محدوده را داریم باید کلمه با mon شروع شود و به moo نرسد.

حال اگر کاربر به دنبال کلماتی باشد که به mon ختم شود چطور؟ (*mon)

در این صورت باید به دنبال کلماتی میان nom و non باشیم. یعنی nom <= w < non .

بگذارید با یک مثال آن را بررسی کنیم. فرض کنید ما به دنبال کلمه lemon میگردیم اما قبل از آن را نمی‌دانیم. برای جست‌جو در این حالت باید یک درخت اضافه ساخته شود که اطلاعات در آن برعکس ذخیره شده. در این

صورت کلمه lemon در این درخت به صورت nomel ذخیره شده است. حال موتور جست‌جو باید به دنبال شاخه‌ای بگردد که با nom شروع می‌شود.

حال فرض کنید ورودی کاربر به صورت co*tion باشد. در این حالت باید چکار کرد؟

می‌توانیم این عبارت رو به دو بخش تقسیم کنیم: co* AND *tion . در این حالت در درخت عادی به دنبال عبارتی با پیشوندی co میگردیم ($co \le w < cp$) و در درخت معکوس به دنبال عبارتی با پسوندی tion (noit $\le w <$ noiu). در نهایت اشتراک این دو مجموعه کلمات بدست آمده پاسخ ما خواهد بود.

«موتور جست‌جو با هر بار ساخت دیکشنری دو درخت می‌سازد درخت عادی و درخت معکوس»

گرچه این روش کارآمد است اما روشی هزینه‌بر است (هزینه زمانی). برای حل این مشکل دو روش Permuterm index و k-gram index پیشنهاد می‌شود.

روش اول (Permuterm index):

به منظور ایجاد اندیس پرمیوترم از علامت خاص $ استفاده می‌کنیم. بگذارید با یک مثال با نحوه عملکرد آن آشنا شویم. فرض کنید می‌خواهیم اندیس‌های کلمه hello را بدست بیاوریم. در ابتدا علامت $ را به انتهای کلمه اضافه می‌کنیم. سپس با چرخش حروف کلمات(از ابتدا به انتها)، انواع حالت‌های احتمالی آن را ذخیره می‌کنیم. به عنوان مثال پرمیوترم کلمه hello به صورت زیر خواهد بود:

- *hello$, elloh , llohe , lohel , ohell*

با اینکار تمام جایگشت‌های کلمه hello در دیکشنری ذخیره خواهد شد.

حال فرض کنید کلمه moon در دیکشنری ما وجود دارد. در روش اندیس پرمیوترم باید سعی شود تا علامت (*) در جست‌جوهای wild-card در انتهای عبارت قرار گیرند. بدین منظور موتور جست‌جو به صورت زیر عمل می‌کند:

m*n → m*n$ → *n$m → n$m*

حال که موتور جست‌جو با جابه‌جایی علامت (*) را در انتهای عبارت قرار داد، در دیکشنری تمام کلماتی که پرمیوترم آن به صورت n$m* می‌باشد را بررسی می‌کند. فرض کنید پرمیوترم کلمه moon در حافظه به صورت زیر ذخیره شده است:

moon → moon$ → oon$m → on$mo → n$moo

از آنجایی که معنای علامت (*) در کامپیوتر به معنای هر چیزی می‌باشد. n$moo همان n$m* است ، بنابراین کلمه moon نیز میتواند در مجموعه خروجی ما قرار بگیرد.

«مشکل روش اندیس پرمیوترم استفاده بیش از اندازه از حافظه است»

روش دوم (Bigram (k-gram) indexes):

در کامپیوتر عبارت Bi به معنای دو است (همانند باینری به معنای دودویی). در این روش همانند اندیس پرمیترم از علامت $ استفاده می‌کنیم با این تفاوت که در یک جمله با دنباله‌ای از k حرف، در ابتدا حرف اول کلمات علامت $ گذاشته و به صورت دوتایی از هم جدا می‌کنیم (اگر حرف آخر کلمه به صورت تکی می‌ماند علامت $ نیز به آخر آن اضافه می‌شود). به عنوان مثال:

- ***"April is the cruelest month"***

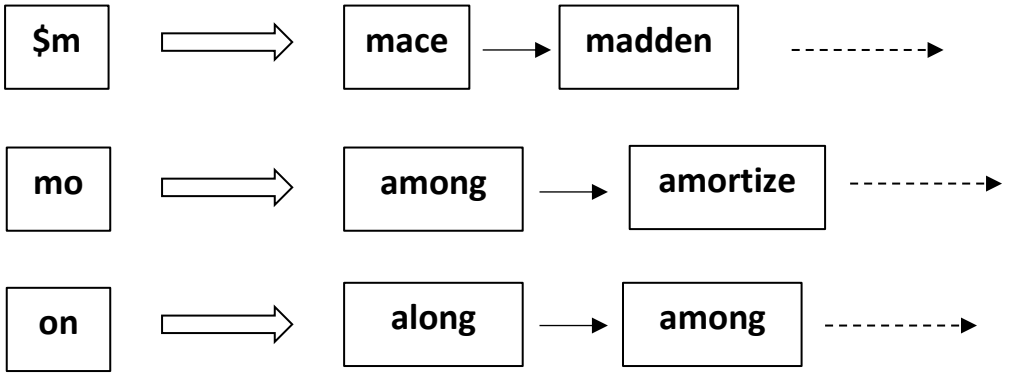
a, ap, pr, ri, il, l, i, is, s, t, th, he, e, $c, cr, ru,
ue, el, le, es, st, t$, m, mo, on, nt, h

در نهایت یک اندیس معکوس ثانویه ایجاد می‌کند و عبارات دوتایی را به صورت دیکشنری در حافظه ذخیره می‌کند.

جلسه ششم

مثال از روش bigram index :

در جلسه گذشته راجب الگوریتم bigram (k-gram) صحبت کردیم. فرض کنید کاربر کلمه moon را جست‌جو می‌کند. مطابق الگوریتم k-gram اگر مقدار k=2 باشد (bigram) و بخواهیم کلمه moon را پیدا کنیم، کلمه به صورت $m و mo و on تبدیل می‌شود. سپس برای هر مجموعه دو حرفی لیستی از کلماتی را تعریف می‌کند که شامل آن دو حرف می‌باشند:



در اینجا تمام کلماتی که داخل آن‌ها $m ، mo و on هست را در لیستی مرتب شده مشخص می‌کند و جلوی آن می‌آورد.

حال بگذارید مثالی دیگر بزنیم. فرض کنید جست‌جویی به صورت *mon داریم. در روش bigram در ابتدا موتور جست‌جو این کلمه را به صورت مجموعه دوحرفی تبدیل می‌کند. بنابراین *mon به $m ، mo و on تبدیل می‌شود. سپس موتور جست‌جو در دیکشنری به دنبال کلماتی همانند $m (در اینجا mace و madden و ...) و کلمات mo و on می‌گردد و در نهایت از لیست سه کلمه اشتراک می‌گیرد ($m AND mo AND on). در نهایت می‌تواند پاسخ ما monday یا money باشد که در لیست هر سه کلمه یافت می‌شود (اشتراک سه کلمه monday و money می‌تواند باشد).

اما ممکن است کلمه moon را نیز به عنوان اشتراک هر سه کلمه به ما به عنوان نتیجه نمایش دهد (کلمه moon در هر سه کلمه وجود دارد). درحالی که ما به دنبال کلمه‌ای بودیم که با mon شروع شود.

تصحیح نوشتار (Spell correction):

گاهی ممکن است در حین جست‌جو نوع صحیح نوشتار کلمه‌ای را فراموش کنیم و یا اشتباه وارد کنیم. معمولا موتورهای جست‌جو همانند گوگل در صورت وجود ایرادات کوچک نوشتاری، متن را تصحیح می‌کند. این عملیات از طریق احرا الگوریتم‌ها امکان‌پذیر است. این الگوریتم‌ها تقریبا همان الگوریتم‌های تشخیص و پردازش جست‌جوهای ستاره‌دار (wild-card) هستند که برای تصحیح املا کلمات استفاده می‌شوند.

برای حل مشکلات نوشتاری می‌توان دو پیشنهاد مطرح کرد:

۱- کلمه‌ای را که کاربر وارد می‌کند با دیکشنری مقایسه کند و نزدیک‌ترین کلمه را برگرداند
(مثال *form → from*)

۲- گاهی اوقات کاربر کلمه را اشتباه نمی‌نویسد بلکه مکان کلمه درست نیست
(مثال *i flew form Heathrpw to Narita*)

در پیشنهاد دوم کلمه form نیز یک کلمه با نگارش صحیح است اما باید به جای آن کلمه from باشد که لازم است تا به اطراف کلمه نگاه شود تا مکان کلمه را تشخیص دهیم.

تصحیح سند (Document correction):

زمانی که در حال نوشتن متنی هستیم بعضی از حروف از نظر ویژگی‌های ظاهری تقریبا با هم اشتباه گرفته می‌شوند. به عنوان مثال در سیستم‌های OCR (Optical Character Reposition) سیستم نمی‌تواند تشخیص دهد منظور کاربرحرف O بوده یا حرف D . در تایپ کردن نیز به عنوان مثال دو حرف M و N هم از نظر ظاهری شبیه به هم هستند و هم به علت نزدیک بودنشان در صفحه کیبورد احتمال اشتباه نوشتن آن‌ها را بالا می‌برد.

این مسائل به موتورهای جست‌جو در جهت بهینه‌سازی جست‌جو بسیار مفید هستند. اما نکته‌ای که وجود دارد این است موتورهای جست‌جو اشتباهاتی که در سندها وجود دارند را تغییر نمی‌دهند بلکه جست‌جوها را تصحیح می‌کنند.

حال گوگل چطور می‌تواند اشتباهات ما را درست کند؟

مشخص است باید عبارت درست را از دیکشنری خودش نگاه کند و با کلمه کاربر بررسی کند و ببیند کلمه کاربر به کدام کلمه در دیکشنری نزدیک است. در این حالت پیشنهاد تصحیح می‌دهد.

به عنوان مثال فرض کنید کاربر کلمه grnt را وارد کرده است. موتور جست‌جو پس از بررسی دیکشنری متوجه می‌شود این کلمه اصلا وجود ندارد. صحیح این کلمه نیز دو حالت grunt و grant خواهد بود. حال کلمه‌ای را انتخاب می‌کند که بیشتر مورد انتخاب بوده (معقول‌تر است).

در ادامه با روش‌های تشخیص و تصحیح کلمات بیشتر آشنا خواهیم شد.

جلسه هفتم

تصحیح کلمات جست‌جو:

در جلسه قبل در مورد فرآیندهای تصحیح نوشتار توسط موتور جست‌جو صحبت کردیم. فهمیدیم موتورهای جست‌جو تنها کلمات جست‌جو را تصحیح می‌کنند و به اشتباهات داخل سندها کاری ندارند. برای تصحیح کلمات نیز در هنگام جست‌جو سعی می‌کند تا مشابه آن کلمه را در دیکشنری پیدا کند.

موتور جست‌جو برای تصحیح کلمات از سه الگوریتم استفاده می‌کند:

۱- الگوریتم Edit distance

۲- الگوریتم n-gram

۳-

الگوریتم Edit distance :

به منظور مشخص کردن فاصله استفاده می‌شود. یعنی فاصله میان کلمه‌ای که کاربر وارد کرده با کلمه درستی که در دیکشنری وجود دارد را پیدا می‌کند. به عنوان مثال در جملات زیر به دو کلمه cat و act نگاه می‌کند تا ببیند چند حرف آن تغییر کرده است. مشخص است که تفاوت میان این دو، در دو حرف است پس فاصله میان آن‌ها برابر ۲ می‌باشد. یا کلمه dog با کلمه cat در سه حرف متفاوت هستند پس فاصله میان آن‌ها برابر ۳ خواهد بود.

- dof to dog distance = 1
- cat to act distance = 2
- cat to dog distance = 3

«هرچه فاصله میان دو کلمه کمتر باشد احتمال آنکه کلمه جابه‌جا نوشته شده باشد بیشتر است»

الگوریتم Weighted edit distance :

در جلسه قبل در خصوص OCR صحبت کردیم. این الگوریتم برای فاصله میان کلمات، وزنی هم برای آن درنظر می‌گیرد. این الگوریتم از لحاظ **نگارش** کلمات را بررسی نمی‌کند بلکه از نظر **دکمه‌های کیبورد** کلمات را بررسی می‌کند (چقدر یک دکمه به دکمه دیگر نزدیک است که باعث می‌شود شما اشتباه کنید). بر اساس دکمه‌ها وزنی را به منظور فاصله گذاشتن تعریف می‌کند. هر چه فاصله میان دکمه‌های کیبورد بیشتر باشد وزن آن نیز بیشتر خواهد بود.

استفاده از Edit distance ها برای تصحیح:

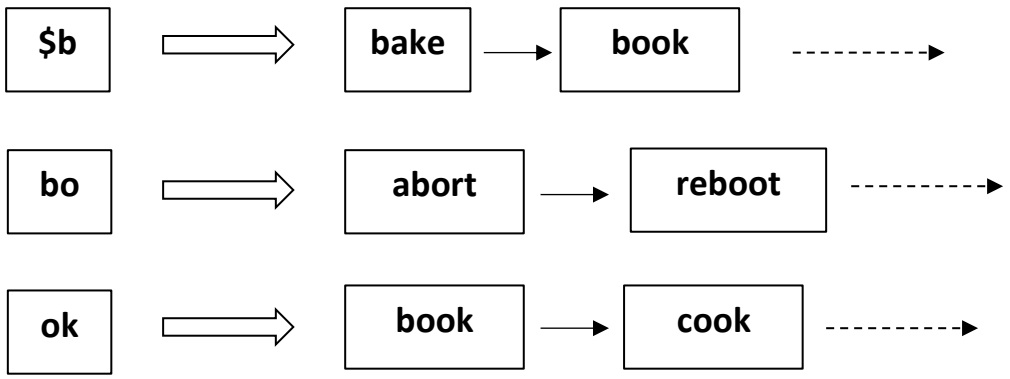
یک راه این است که می‌توانیم برای جست‌جوی داده شده تمام کلماتش را بررسی کنیم و یک مجموعه از Edit distance برای آن بسازیم.کلمه‌ای که فاصله کمتری دارد مسلما بهتر است اما برای آن کلماتی که فاصله یکسانی دارند، از Weighted edit distance استفاده کنیم. حال این مجموعه را با مجموعه کلمات درست در دیکشنری **اشتراک** می‌گیریم و بر اساس عبارت‌های پیدا شده به کاربر پیشنهاد می‌دهیم.

راه جایگزین، پیدا کردن تمام موارد درست در اندیس‌گذاری معکوسمان (دیکشنری) که ممکن است برای جست‌جو درست باشند و ارسال تمام سندهایی که با این موارد درست یکی می‌باشد است. گرچه این راه کندتر است اما بر خلاف راه قبلی در آن مرحله اشتراک‌گیری حذف شده است.

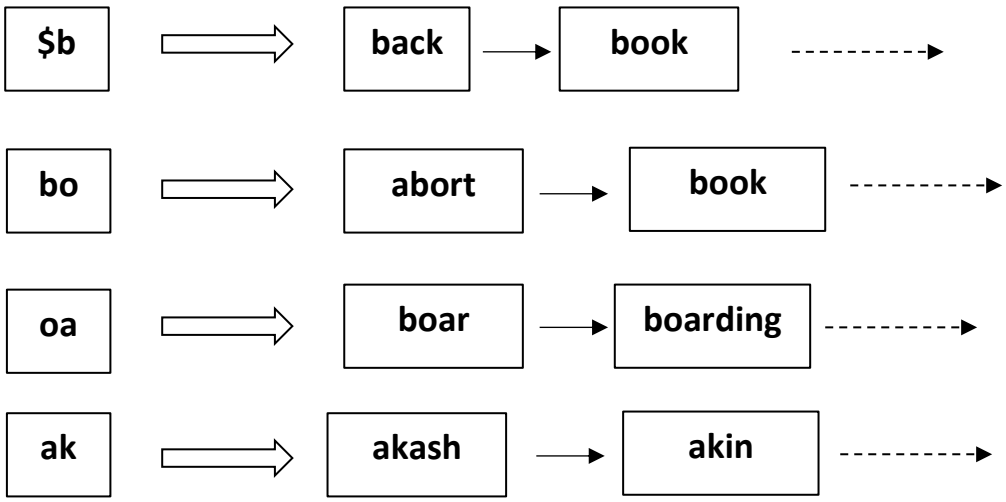
استفاده از n-gram overlap برای تصحیح:

مشابه bigram که در جست‌جوهای wild-card مورد استفاده بود یعنی تبدیل کلمات به کلمات دوحرفی به منظور تشخیص جست‌جوهای ستاره‌دار، این الگوریتم در تصحیح نوشتار نیز استفاده می‌شود. در الگوریتم n-gram از اندیس آن (لیست‌های کلمات) استفاده می‌کنیم. نکته‌ای که باید به خاطر داشت این است n-gram یعنی چند حرفی. به عنوان مثال اگر بخواهیم $n = 2$ باشد در حقیقت داریم از الگوریتم دوحرفی (2-gram = bigram) استفاده می‌کنیم.

فرض کنید می‌خواهیم دیکشنری با الگوریتم bigram داشته باشیم که در آن تصحیح کلمات وجود داشته باشد. بدین منظور برای کلمه book مجموعه دو حرفی زیر را داریم:



حال فرض کنید کلمه book را به صورت اشتباه boak نوشته‌ایم. در این صورت موتور جست‌جو کلمه را به مجموعه‌های دو حرفی $b ، bo ، oa و ak تبدیل می‌کند:



مشخص است مجموعه دو حرفی $b و bo در دو کلمه boak و book مشترک است. سپس دو مجموعه $b و bo را با یکدیگر اشتراک می‌گیرد که در این صورت اشتراک آن‌ها کلمه book خواهد بود. در نهایت به جای کلمه boak به ما کلمه book را پیشنهاد می‌دهد.

«برای تمام کلمات در همان ابتدای ساخت دیکشنری مجموعه حروف دوتایی آن نیز ساخته می‌شود»

حال فرض کنید می‌خواهیم از n = 3 باشد که در این صورت از الگوریتم trigram استفاده خواهیم کرد. در این الگوریتم به جای تقسیم کلمه به مجموعه‌های دو حرفی، کلمه را به مجموعه‌های سه حرفی تقسیم می‌کنیم.

به عنوان مثال کلمه November با کلمه December را فرض کنید. trigram هر یک از آن‌ها به صورت زیر خواهد بود (در trigram استفاده نکردن از علامت $ بلامانع است):

november : nov , ove , vem , emb , mbe , ber .

December : dec , ece , cem , emb , mbe , ber .

همانطور که مشاهده می‌کنید این دو کلمه با هم دارای ۳ اشتراک هستند بنابراین احتمال آنکه با یکدیگر به اشتباه جابه‌جا شوند وجود دارد.

«استفاده از روش n-gram به صورت bigram یا trigram و یا مراتب بالاتر به موتور جست‌جو بستگی دارد»

حال بهترین روش برای اینکه بفهمیم کدام کلمه به کلمه ما نزدیک‌تر است چیست؟

ضریب جَکارد (Jaccard coefficient):

ضریب جکارد میان دو مجموعه انتخابی برای تشخیص میزان همپوشانی (نزدیک بودن کلمات به یکدیگر) استفاده می‌شود. در ضریب جکارد اندازه اشتراک دو مجموعه بر اندازه اجتماع دو مجموعه تقسیم خواهد شد. فرمول ضریب جکارد (JC) به صورت زیر می‌باشد:

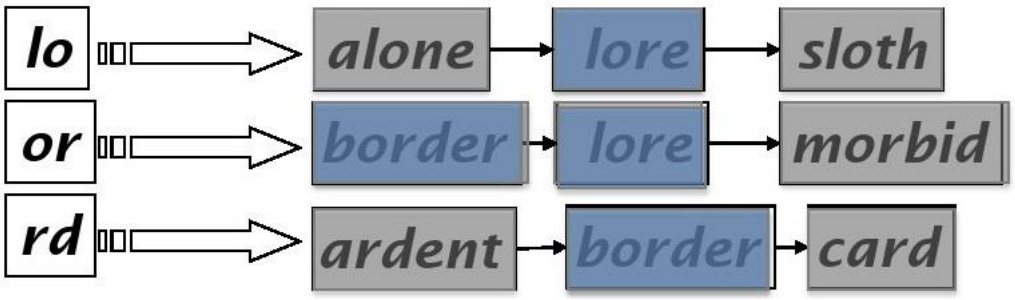
$$JC\,(X,Y\,) = \frac{|X \cap Y|}{|X \cup Y|}$$

در ضریب جکارد دو مجموعه X و Y می‌توانند اندازه‌های مختلفی داشته باشند. اگر عناصر دو مجموعه با هم برابر باشند در این صورت ضریب جکارد برابر 1 خواهد شد و اگر دو مجموعه هیچ اشتراکی با هم نداشته باشند ضریب جکارد برابر 0 خواهد شد.

«ضریب جکارد همیشه عددی میان 0 و 1 خواهد بود»

به عنوان مثال اشتراک دو کلمه november و devember برابر ۳ و اجتماع دو کلمه برابر ۹ خواهد بود. ضریب جکارد برابر 0.3 می‌باشد.

مثال: فرض کنید می‌خواهیم کلمه lord را جست‌جو کنیم. در این حالت می‌خواهیم دو کلمه شبیه به هم در bigram کلمه lord را بیابیم (lo, or, rd).



به منظور یافتن کلمات مشترک دو به دو مجموعه‌ها را با هم اشتراک می‌گیریم. لیست lo با or در کلمه lore مشترک‌اند و لیست or با rd در کلمه border . لیست lo با لیست rd نیز اشتراک ندارند. حال برای شباهت اشتراک‌ها با کلمه lord از ضریب جکارد استفاده می‌کنیم.

اشتراک دو کلمه lore و lord برابر ۳ است و اجتماع آن‌ها برابر ۵ می‌باشد (JC = 0.6) و اشتراک دو کلمه border و lord برابر ۳ است و اجتماع آن‌ها برابر ۷ می‌باشد (JC = 0.4). مشخص است کلمه lore به کلمه lord شباهت بیشتری دارد.

تصحیح کلمات با حساسیت متن (Context-sensitive):

اگر بتوانیم با نگاه به جمله به کلمه اشتباه پی ببریم در این صورت از روش حساسیت متن استفاده کرده‌ایم. فرض کنید ما جمله ***"flew form Heathrow"*** را جست‌جو می‌کنیم در حالی که متن ما ***I flew from Heathrow to Narita*** می‌باشد. ما می‌خواهیم موتور جست‌جو به ما پیغام زیر را نمایش دهد:

Did you mean "***flew from Heathrow***"?

در این حالت باید چکار کنیم؟

اولین پیشنهاد بازیابی نزدیک‌ترین عبارت‌های دیکشنری به کلمه مورد نظر بود. به عنوان مثال:

- ***flew from heathrow***
- ***fled form heathrow***
- ***flea form heathrow***

الگوریتم Soundex :

این الگوریتم بر اساس تلفظ کلمات می‌باشد. به عنوان مثال در زبان فارسی گویش کلمه «خواهر» با نوشتن آن متفاوت است. در انگلیسی نیز کلماتی وجود دارد که تلفظ آن با نوشتن آن متفاوت است.

در الگوریتم Soundex هر توکن به یک توکن ۴ حرفی تبدیل می‌شود و این عمل را با بقیه کلمات جست‌جو انجام می‌دهد. حال این الگوریتم چطور کار می‌کند؟

در ابتدا اولین حرف کلمه را بازیابی می‌کند و نگه می‌دارد. سپس برای هر حرف یک عدد پیشنهاد می‌دهد:

'A', E', 'I', 'O', 'U', 'H', 'W', 'Y' → 0

و حروف را به اعداد تبدیل می‌کند:

B, F, P, V → 1
C, G, J, K, Q, S, X, Z → 2
D, T → 3
L → 4
M, N → 5
R → 6

سپس اعداد 0 را حذف کرد و رشته خروجی را به فرم زیر نمایش می‌دهد.

<uppercase letter> <digit> <digit> <digit>

به عنوان مثال ***Herman*** → H655 حال حتی اگر کلمه به hermann تغییر پیدا کند نیز همان کد به خروجی خواهد رفت.

به خاطر داشته باشید تنها ۴ حرف اول کلمات مهم هستند.

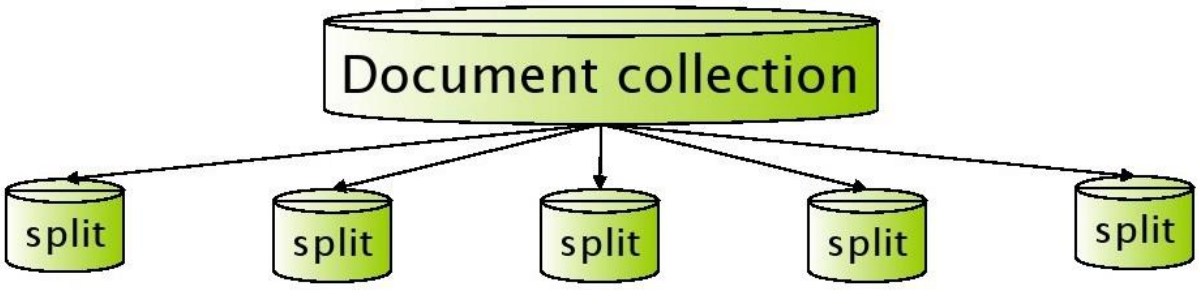
«حروف انگلیسی موجود در الگوریتم Soundex بر اساس آوای حروف شماره‌گذاری شده‌اند»

جلسه هشتم

سرورهای گوگل:

سرورهای گوگل همیشه در حال بررسی و اسکن صفحات وب جدید هستند. زمانی که کلکسیونی از صفحات وب در اختیار سرورهای گوگل قرار می‌گیرد این اطلاعات در بقیه سرورها پخش می‌شود. به منظور اندیس‌گذاری توسعه یافته یک سرور (ماشین) با نام Master به عملیات اندیس‌گذاری جهت می‌دهد. هر اندیس‌گذاری را به مجموعه‌ای از عملیات‌های موازی (Parallel tasks) تقسیم می‌کند و هر عملیات را به یک سرور بیکار (idle) اختصاص می‌دهد.

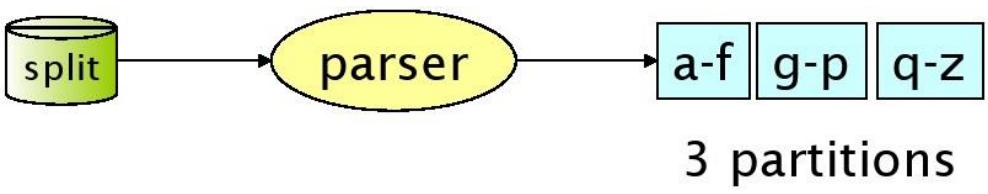
برای انجام عملیات‌های موازی یکسری از سرورهای گوگل نقش Parser و یکسری دیگر نقش Inverter را دارند. در این حالت کلکسیون سندهای ورودی به چندین بخش تقسیم می‌شوند (split) که هر بخش زیرمجموعه‌ای از سندها می‌باشد.



سرور Parser و Inverter :

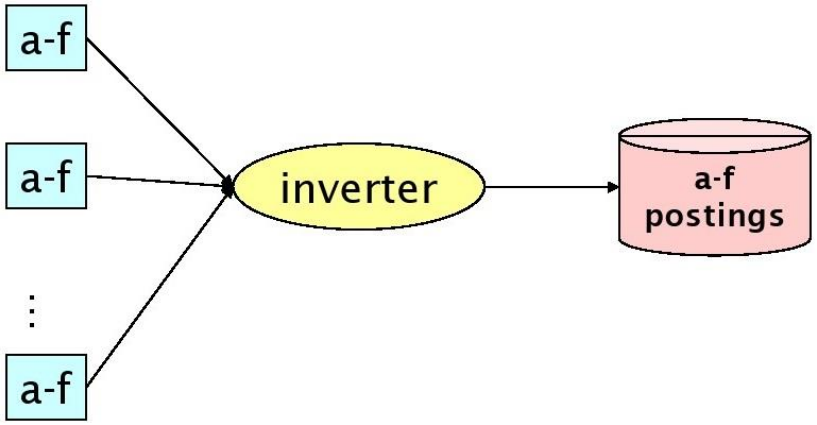
پس از بخش‌بندی کلکسیون سندها، سرور Master یک بخش را به یک سرور پارسر (Parser) بیکار اختصاص می‌دهد. پارسر سند را در هر زمان یک سند را می‌خواند و شناسه عبارت (Term-ID) و شناسه سند (Doc-ID) را با هم بیرون می‌کشد. یعنی می‌گوید این عبارت با این شناسه در این سند با این شناسه وجود دارد.

برای سرور پارسر باید در ابتدا پارتیشن تعریف نمود. پارسر تمام جفت (term-id , doc-id) ها را در j پارتیشن ذخیره می‌کند. هر پارتیشن یک محدوده از حروف اول عبارات می‌باشد (به عوان مثال اگر جفت‌ها q-z, g-p, a-f باشند j = 3 می‌باشد یعنی ۳ پارتیشن).

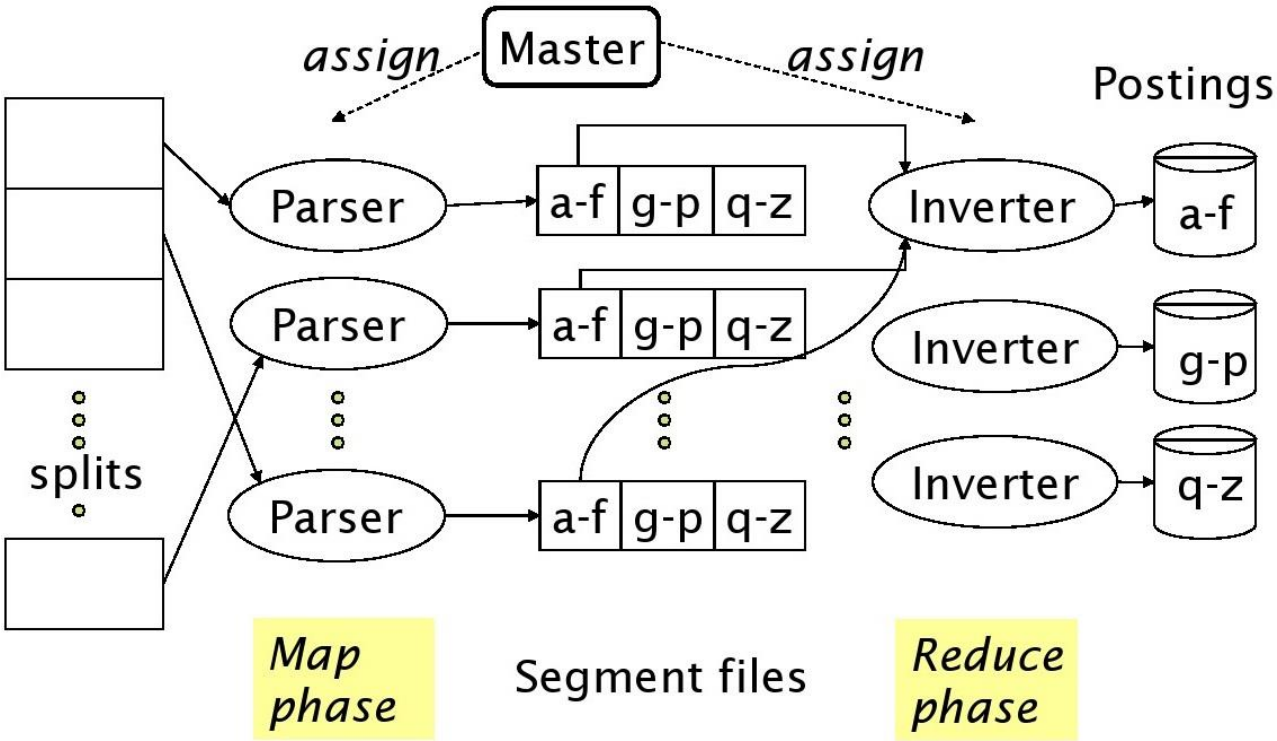


حال برای تکمیل اندیس‌گذاری معکوس از سرورهای Inverter استفاده می‌کنیم.

سرور Inverte تمامی جفت (term-id , doc-id) را برای یک عبارت-پارتیشن جمع‌آوری می‌کند (پستینگ‌ها). یعنی هر سرور پارتیشن مخصوص به خودش را می‌بیند. سپس آن‌ها را مرتب می‌کند و در لیست‌های پستینگ ذخیره می‌کند.



شکل زیر یک نمای کلی از عملکرد سرورهای گوگل (MapReduce) را نمایش می‌دهد:



فشرده‌سازی اطلاعات (Compression):

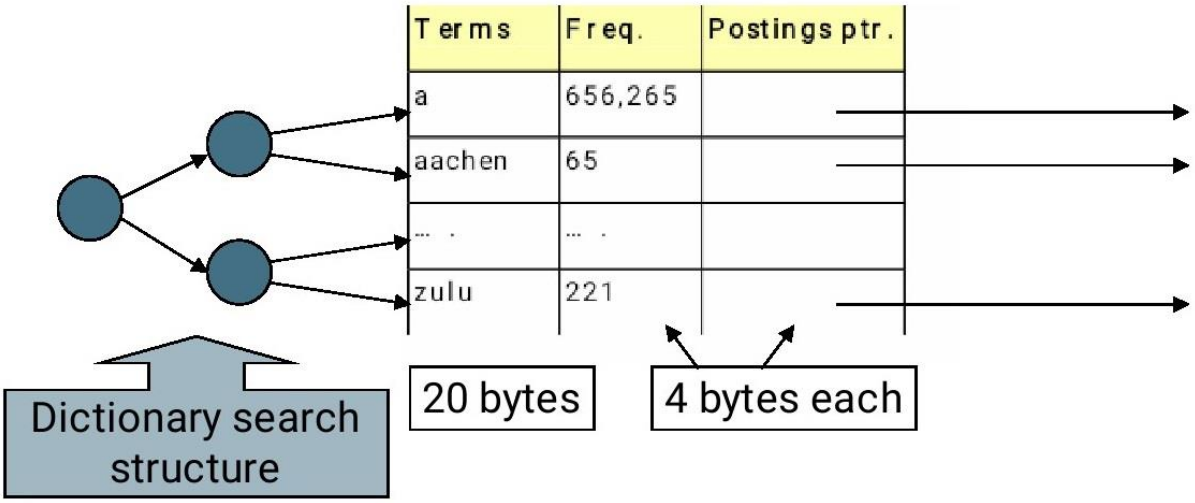
یکی از دلایل مهم فشرده‌سازی افزایش فضای ذخیره‌سازی و بالا بردن سرعت عملیات بر روی اطلاعات است. فشرده‌سازی تنها تغییر حالت اطلاعات نیست بلکه شما می‌توانید اطلاعات غیر ضروری را حذف کنید تا از حافظه کمتر استفاده کنید.

در فشرده‌سازی ما دو نوع روش Lossless و Lossy را داریم. در روش Lossless بدون حذف کردن و تنها با تغییر حالت اطلاعات، اطلاعات را فشرده می‌کنیم اما در روش Loosy با حذف اطلاعات، اطلاعات را فشرده می‌کنیم.

فضای ذخیره‌سازی دیکشنری - بدون تغییر:

همانطور که در جلسات قبل گفتیم برای ذخیره‌سازی دیکشنری از درخت دودویی یا درخت متوازن استفاده می‌کنیم به طوری که در برگ‌های درخت عبارت‌ها، فرکانس و اشاره‌گر به لیست پستینگ ذخیره شده‌اند.

- Array of fixed-width entries
  - ~400,000 terms; 28 bytes/term = 11.2 MB.



مطابق شکل بالا برای عبارات حداکثر 20 بایت درنظر گرفته شده است. زیرا بزرگترین کلمه انگلیسی ۲۰ حرف دارد و برای ذخیره این حرف احتیاج به 20 بایت حافظه داریم.

با در نظر گرفتن حداکثر 4 بایت برای فرکانس ما می‌توانیم تا 2^{32} فرکانس تعریف کنیم (هر بایت 8 بیت می‌باشد). حداکثر اندازه قابل قبول برای فرکانس به تعداد اندازه سندها می‌باشد.

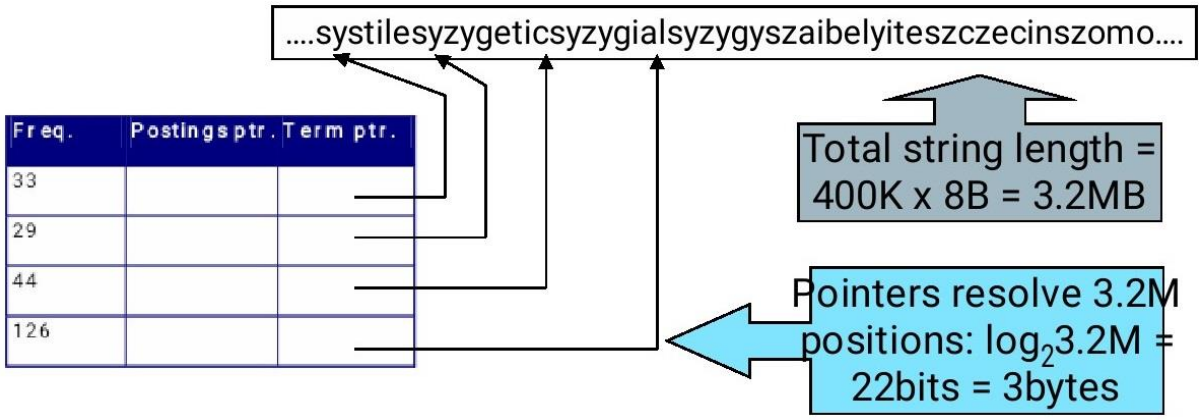
اشاره‌گر به پستینگ آدرس یک حافظه را ذخیره کرده است. یعنی هر وقت احتیاج به شماره سندها باشد از طریق اشاره‌گر به پستینگ به آن حافظه دسترسی پیدا می‌کند. برای اشاره‌گر نیز حداکثر 4 بایت درنظر گرفته شده است.

حال فرض کنید تقریبا ۴۰۰ هزار عبارت داریم که باید برای هر عبارت 28B فضا اختصاص دهیم. در نتیجه برای ذخیره ۴۰۰ هزار عبارت احتیاج به 11.2MB داریم.

در ادامه به روش‌های فشرده‌سازی دیکشنری خواهیم پرداخت.

فشرده‌سازی لیست عبارت (تبدیل دیکشنری به رشته):

اولین روش پیشنهادی فشرده‌سازی ، تبدیل دیکشنری به رشته بود به طوری که به جای ذخیره فضای 20B برای هر کلمه (حتی کلمه a) تمام کلمات را در یک رشته بلند (Long string) پشت سر هم ذخیره کنیم. به جای ذخیره خود کلمه در حافظه، یک اشاره‌گر به کلمه ذخیره کنیم (یک اشاره‌گر به 4 بایت حافظه احتیاج دارد). اشاره‌گر به کلمه بعدی پایان کلمه فعلی را مشخص می‌کند.



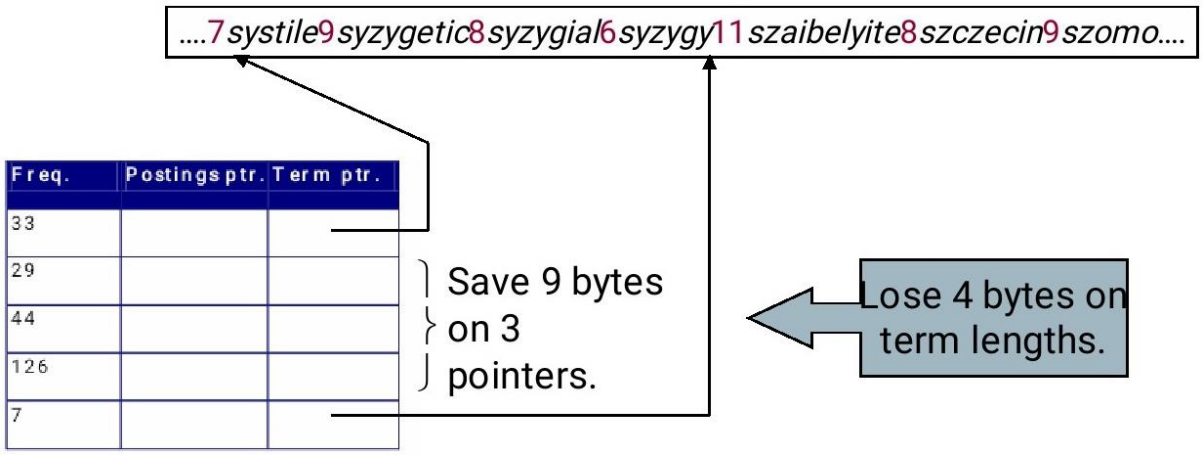
در زبان انگلیسی به طور میانگین برای ذخیره یک عبارت نیاز به 8 بایت حافظه داریم. اگر ما ۴۰۰ هزار کلمه داشته باشیم در نتیجه 3.2MB حافظه برای رشته کلمات نیاز خواهیم داشت. حافظه مورد نیاز اشاره‌گرها نیز به اندازه رشته بستگی دارد. اگر حافظه رشته 3.2MB باشد، در نتیجه به اندازه لگاریتم پایه ۲ آن احتیاج به فضا داریم (3 بایت).

چرا لگاریتم پایه ۲ ؟ فرض کنید ما 4 بایت داریم. برای مشخص کردن آدرس هر خانه احتیاج به ۲ بیت داریم. می‌توانیم خانه اول را 00 خانه دوم را 01 خانه سوم را 10 و خانه چهارم را 11 آدرس دهیم. در این صورت اگر ما از 4 لگاریتم پایه ۲ بگیریم به عدد 2 خواهیم رسید یعنی 2 بیت برای آدرس‌دهی 4 بایت کافی می‌باشد.

پس از فشرده‌سازی برای ذخیره‌سازی فرکانس احتیاج به 4 بایت، برای اشاره‌گر پستینگ 4 بایت و برای اشاره‌گر عبارت نیز 3 بایت و برای ذخیره هر عبارت در رشته به طور میانگین به 8 بایت احتیاج داریم. با انجام این فشرده‌سازی ۶۰ درصد حافظه کمتری استفاده خواهیم کرد. در حالت عادی 24B برای دیکشنری احتیاج داشتیم اما در حال حاضر به 19B احتیاج داریم.

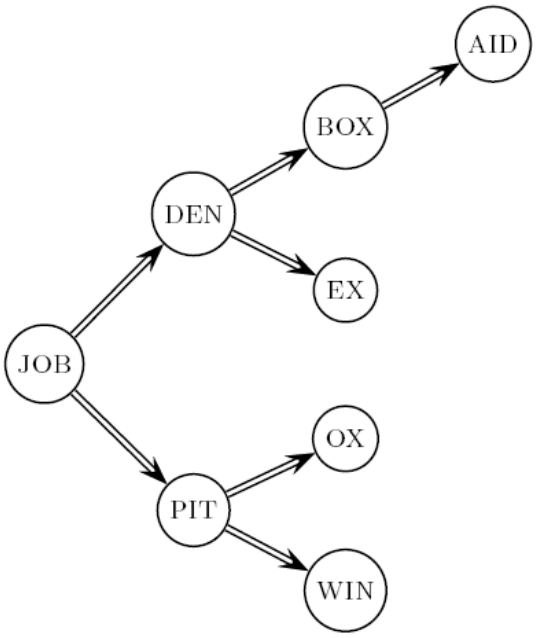
فشرده‌سازی به روش بلاک‌بندی:

دومین روش پیشنهادی پس از تبدیل دیکشنری به رشته، فشرده‌سازی به روش بلاک‌بندی (Blocking) بود به طوری که به جای ایجاد اشاره‌گر عبارت برای هر کلمه، با ساخت یک بلاک و ریختن k عبارت در آن به اولین خونه بلاک اشاره کنیم. همچنین در بلاک تعداد حروف هر عبارت را قبل از خودش اضافه می‌کنیم تا ابتدا و انتهای عبارات در بلوک مشخص شود.



همانطور که در شکل بالا مشاهده می‌کنید به ازای هر ۴ کلمه (k = 4) یک اشاره‌گر داریم. بدین ترتیب ۳ خانه 3B از حافظه استفاده نکرده‌ایم که در مجموع 9B را فشرده کردیم اما از طرفی در بلاک با اضافه کردن تعداد حروف عبارات، به تعداد هر عبارت 1B از حافظه را اشغال کرده‌ایم (در اینجا 4B). بنابراین به ازای هر بلاک ما 5B از حافظه را آزاد کرده‌ایم.

جست‌جو در دیکشنری - بلاک‌بندی نشده:



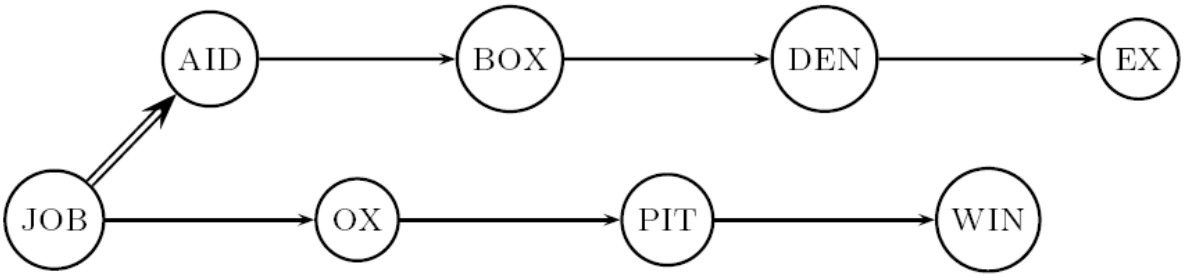
فرض کنید دیکشنری به صورت شکل مقابل داریم. حال اگر بخواهیم این درخت را پیمایش کنیم واطلاعاتی را پیدا کنیم، نیاز به چند بارجست‌جو داریم؟

در درخت دودویی اولین مقایسه با ریشه درخت صورت خواهد گرفت. فرض کنید ما می‌خواهیم کلمه JOB را در درخت پیدا کنیم. این عمل را می‌توانیم تنها با یکبار مقایسه انجام دهیم. اما برای پیدا کردن DEN باید دوبار مقایسه انجام شود یعنی یکبار با JOB و بار دیگر با DEN . به همین ترتیب می‌توانیم تعداد مقایسات لازم برای بقیه کلمات درخت را بیابیم. حال اگر بخواهیم میانگین تعداد مقایسات در این درخت را بیابیم کافیست تعداد کل مقایسات لازم را تقسیم بر تعداد عناصر داخل درخت کنیم.

میانگین = [1 + (2 + 2) + (3 + 3 + 3 + 3) + 4] ÷ 8 = 2.6

جست‌جو در دیکشنری - بلاک‌بندی شده:

حال اگر دیکشنری بالا را بلاک‌بندی کنیم به علت مرتب کردن اطلاعات داخل بلاک، شکل درخت و ترتیب عناصر آن عوض خواهد شد (در هر بلاک اطلاعات بر اساس حروف الفبا مرتب شده است).



برای مقایسه و پیمایش ابتدا جست‌جو دودویی به منظور یافتن بلاک با طول ۴ انجام داده و سپس در داخل بلاک جست‌جو خطی انجام می‌دهیم. میانگین مقایسات لازم در این درخت به صورت زیر محاسبه می‌شود:

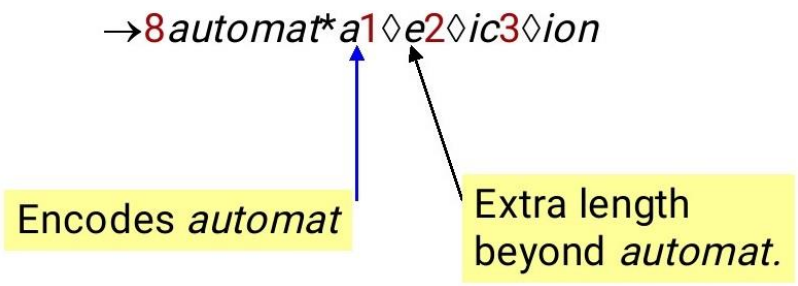
میانگین = [1 + (2 + 2) + (3 + 3) + (4 + 4) + 5] ÷ 8 = 2.6

فشرده‌سازی به روش Front coding :

آخرین روش پیشنهادی برای فشرده‌سازی دیکشنری، استفاده از روش Front coding می‌باشد. اطلاعات داخل دیکشنری بسیار به یکدیگر شبیه است. بدین منظور سعی می‌شود تا تنها تفاوت‌های کلمات ذخیره شوند. به عنوان مثال بلاک زیر را در نظر بگیرید:

8automata8automate9automatic10automation

در ابتدا تعداد حروف کلمه automata نوشته خواهد شد. سپس این کلمه با کلماتی که وجه مشترک دارد مقایسه می‌شود. مشخص است عبارت automat در همه این ۴ عبارت مشترک است و پس از حرف t کلمات متفاوت می‌شوند. بنابراین در انتهای آخرین حرف مشترک علامت (*) قرار می‌دهد و سپس تعداد تفاوت کارکتری آن را با کلمه بعدی جلوی آن می‌نویسد. سپس علامت لوزی را می‌گذارد و مشخص می‌کند به جای آن کلمه باید چه کلمه‌ای بگذارد.



جلسه نهم (آخر):

فشرده‌سازی پستینگ‌ها (Postings Compression):

در جلسات قبلی در خصوص لیست پستینگ صحبت کردیم. در لیست پستینگ شماره سندها به صورت مرتب ذخیره می‌شوند. به عنوان مثال فرض کنید پستینگ ما 33, 47, 154, 159, 202 باشد:

computer. 33, 47, 154, 159, 202…

یعنی کلمه کامپیوتر داخل سندهایی با شماره‌های ۳۳ و ۴۷ و ۱۵۴ و ۱۵۹ و ۲۰۲ وجود دارد.

یکی از روش‌های فشرده‌سازی پستینگ‌ها این است که ابتدا شماره اولین سندی که عبارت در آن وجود دارد ذخیره شود (33) و سپس فاصله (gap) میان سند اول و سند دوم ذخیره شود (14 = 33 - 47) و فاصله میان سند دوم و سند سوم (107 = 47 - 154) و به همین ترتیب تا آخرین سند. یعنی:

computer. 33, 14, 107, 5, 43…

مثال:

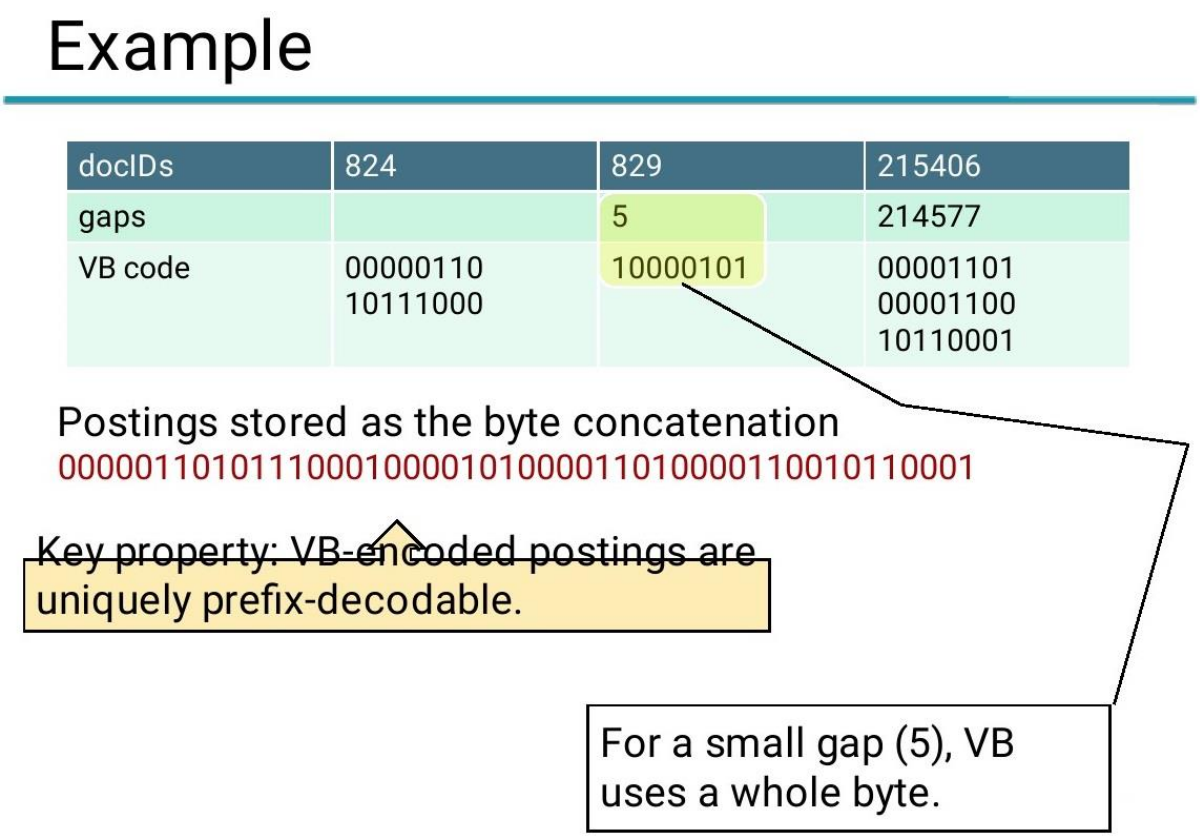
	encoding	postings list								
THE	docIDs	...	283042		283043		283044		283045	...
	gaps			1		1		1		...
COMPUTER	docIDs	...	283047		283154		283159		283202	...
	gaps			107		5		43		...
ARACHNOCENTRIC	docIDs	252000		500100						
	gaps	252000	248100							

اما این روش زمانی کارآمد است که فرکانس عبارت زیاد باشد یعنی داخل چند سند تکرار شده باشد و فاصله میان شماره سندها کم باشد.در ادامه دیگر روش‌های فشرده‌سازی پستینگ‌ها را بررسی خواهیم کرد.

فشرده‌سازی به روش Variable Byte (VB) :

در این روش از تعداد بایت‌های متفاوتی به منظور ذخیره عدد استفاده می‌کنیم. در حال حاضر موتور جست‌جو گوگل از این طریق برای ذخیره فاصله میان سندها استفاده می‌کند. در این روش خود عدد ذخیره نمی‌شود بلکه Variable Byte آن عدد ذخیره خواهد شد.

بگذارید با یک مثال نحوه عملکرد آن را بررسی کنیم:



در اینجا شناسه سند اول برابر 824 می‌باشد و شناسه سند بعدی نیز 829 است. فاصله (gap) میان این دو سند برابر 5 خواهد بود که باید داخل پستینگ عدد 5 ذخیره شود. همانطور که گفتیم در این روش خود عدد ذخیره نخواهد شد. اگر عدد 824 را به کد باینری (دودویی) تبدیل کنیم برابر 1100111000 خواهد شد. در روش VB کد باینری عدد را از سمت راست با ضریب ۷ جدا می‌کند. برای ذخیره VB از یابت استفاده می‌شود. یک بایت برابر 8 بیت می‌باشد. اگر عددی بیشتر از 7 بیت فضا نیاز داشته باشد، رقم هشتم را عدد 1 می‌گذارد و سپس در 8 بیت دیگر ادامه آن عدد را ذخیره می‌کند. اگر عدد پایان یافته باشد رقم هشتم را عدد 0 می‌گذارد. در این صورت می‌داند عدد از کجا شروع و به کجا ختم می‌شود (00000110 , 10111000 → 824).

بگذارید مثالی دیگر بزنیم. فرض کنید می‌خواهیم VB عدد 16 را ذخیره کنیم. عدد 16 به صورت باینری برابر 10000 می‌باشد. اگر از سمت راست عدد به صورت ضریبی از ۷ جدا کنیم، VB عدد 16 برابر 00010000 خواهد شد. نکته‌ای که باید به آن توجه داشت این است برای عدد 16 رقم هشتم 0 خواهد بود چراکه عدد ادامه نداشته و در 7 بیت پایان یافت.

«در روش Variable Byte اگر عدد بزرگتر از 127 باشد باید دو بایت کنار گذاشته شود»

کد یونری (Unary Code):

کدر یونری در حقیقت یک نوع نمایش اعداد است. در این روش ابتدا به تعداد عدد (به عنوان مثال عدد ۳) عدد 1 می‌گذاریم و در نهایت در انتها عدد 0 را قرار می‌دهیم. به عنوان مثال عدد ۳ برابر 1110 و عدد ۱۰ برابر 11111111110 می‌باشد.

اما این روش روشی کارآمد نمی‌باشد مگر در **کد گاما** (Gamma Code) استفاده شود.

کد گاما (Gamma Code):

یکی از بهترین روش‌ها برای فشرده‌سازی کدهای بیتی استفاده از کد گاما می‌باشد. در روش کد گاما اولین 1 از سمت چپ عدد حذف می‌شود. سپس بیت‌های باقی‌مانده (Offset) را محاسبه می‌کند و تعداد آن‌ها را به کد یونری تبدیل می‌کند. در نهایت به مقدار آفست را با کد یونری ذخیره می‌کند.

کد گاما برخی اعداد به شرح زیر می‌باشد:

number	length	offset	γ-code
0			none
1	0		0
2	10	0	10,0
3	10	1	10,1
4	110	00	110,00
9	1110	001	1110,001
13	1110	101	1110,101
24	11110	1000	11110,1000
511	111111110	11111111	111111110,11111111
1025	11111111110	0000000001	11111111110,0000000001

«در روش کد گاما به تعداد نیاز میتوان بیت استفاده کرد»