אחזור מידע

Index Compression

מדוע להשתמש בכיווץ (כללי) ? מידע כשהוא מכווץ תופס פחות מקום על הדיסק (וכתוצאה מתקבל חיסכון בכסף). כיווץ נתונים מאפשר לשמור יותר נתונים בזיכרון המחשב. כתוצאה מכך, אנו מקבלים שיפור ביצועים. נניח שאנו לא יכולים לשמור את כל המילון שלנו בזיכרון המחשב. במקרים כאלו, אנו צריכים, מידי פעם, לבצע פניות לדיסק, על מנת להביא את החלק הרלוונטי של המילון מהדיסק לזיכרון. כזכור, עבודה עם הדיסק הינה איטית. אם, כתוצאה מכיווץ הנתונים המילון כולו (או חלק גדול יותר שלו) נשמר בזיכרון, אנו לא צריכים לבצע גישות לדיסק (או מבצעים פחות גישות לדיסק. וכתוצאה, מתקבל שיפור בביצועים.

גם אם המילון, כשהוא לא מכווץ, נשמר כולו בזיכרון, עדיין כדאי לכווץ אותו. בזיכרון שנחסך, אנו יכולים, למשל, לשמור posting list של term-ים נפוצים, וכך לשפר את ביצועי המערכת.

כיווץ נתונים מקצר את זמן העברת הנתונים מהדיסק לזיכרון המחשב. נניח שיש לנו מידע ששמור באופן לא מכווץ בדיסק. הזמן שנדרש להביא אותו מהדיסק גדול יותר מהזמן שהיה נדרש להביא את אותו המידע או הוא היה מכווץ (מכוון שהוא תופס יותר מקום על הדיסק והוא יותר "ארוך"). אם אלגוריתמי הפריסה שלנו מהירים, הרי שזמן ההבאה של אותם נתונים לא מכווצים מתקזז עם הזמן שנדרש לפרוס את אותם הנתונים במידה והם היו מכווצים. ולכן, בפועל מה שמתקבל הוא שקריאת נתונים מכווצים מהדיסק ופריסתם מהירה יותר מקריאת נתונים לא מכווצים.

לסיכון, אם כך, מדוע אנו רוצים לכווץ את ה- inverted indexes ?

(1) כיווץ המילון - כיווץ הופך את המילון לקטן דיו, כך שאפשר לשמור אותו בזיכרון המחשב. במקרים מסוימים, המילון קטן מאד, ואז אנו יכולים לשמור, בנוסף, גם postings lists בזיכרון. (2) Postings file(s) - מקטין את נפח הדיסק הנדרש לשמירת כל posting list, ולכן גם מקטין את הזמן הדרוש על מנת לקרוא את ה-postings lists מהדיסק. בנוסף, הרבה מנועי חיפוש שומרים חלק מה-postings lists בזיכרון (על מנת לשפר ביצועים) – נוכל לשמור יותר.

מעט סטטיסטיקות

לצורך הדגמת הנושאים הבאים אנו נשתמש באוסף המסמכים RCV1. הטבלה הבאה מסכמת את הנתונים החשובים של אוסף מסמכים זה.

סימון	משמעות	ערך
N	documents	800,000
L	avg. # tokens per doc	200
M	terms (= word types)	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term	7.5
	non-positional postings	100,000,000

נניח שאנו עוברים על ה-800,000 מסמכים, מחלקים כל מסמך ל-token-ים שלו, מבלי לעשות איזשהו עיבוד לשוני. במקרה זה, כפי שניתן לראות בטבלה הבאה, גודל המילון שמתקבל הוא 484K, גודלו של ה-non-positional index הוא 109,971K ואילו גודלו של ה-positional index הוא 197,879K.

נניח שאנו מתעלמים מכל ה-token-ים שמתייחסים למספרים. במקרה זה, ניתן לראות, שמספר ה-term-ים במילון קטן בכ-10K (שינוי של כ-2%). גודלו של ה-non-positional index ירד ל-100,680K ואילו גודלו של ה- positional index ירד ל-179,158K (שינוי של כ-8/9% עבור שניהם).

שימוש ב-case folding יוצר שינוי משמעותי הרבה יותר בגודלו של המילון, שקטן בכ-17%. מבחינתן ה--non positional index, יש ירידה של כ-3% (שני מופעים ב-case-ים שונים, עכשיו מופעים כמופע יחיד). ואילו עבור ה-positional index, השינוי הוא אפסי (שינוי ה-case של term אינו משנה את מספר ההופעות שלו במסמך).

נניח שאנו מורידים 30 stop words, לא מתקבל שינוי בגודל המילון (הורדנו בסה"כ 30 term-ים). אולם, מכוון שאלו הם stop words, והשכיחות שלהם מאד גבוהה יש שינוי גבוה (ירידה של כ-14%) בגודלו של ה- non-positional postings, ושינוי גבוה אף יותר (כ-31%) בגודלו של ה-positional postings.

אם במקום 30 stop words, אנו מורידים 150 stop words, הרי שגם עכשיו לא מתקבל שינוי בגודל המילון (הורדנו בסה"כ 150 term-ים). אולם, מאד גבוהה יש שינוי גבוה (ירידה של כ-30%) בגודלו של ה-non-positional postings, ושינוי גבוה אף יותר (כ-47%) בגודלו של ה-positional postings.

שימוש ב-stemming יקטין את גודלו של המילון בעוד כ-17%. גודלו של ה-non-positional postings ירד בעוד 4% ואילו גודלו של ה-positional postings של ישתנה.

size of	word types (terms)			non-positional postings			positional postings		
	dictionary			non-positional index			positional index		
	Size (K)	∆%	cumul %	Size (K)	∆ %	cumul %	Size (K)	∆ %	cumul %
Unfiltered	484			109,971			197,879		
No numbers	474	-2	-2	100,680	-8	-8	179,158	-9	-9
Case folding	392	-17	-19	96,969	-3	-12	179,158	0	-9
30 stopwords	391	-0	-19	83,390	-14	-24	121,858	-31	-38
150 stopwords	391	-0	-19	67,002	-30	-39	94,517	-47	-52
stemming	322	-17	-33	63,812	-4	-42	94,517	0	-52

בתחום של כיווץ נתונים אנו מבדילים בין שני סוגים של כיווץ. (1) Lossless compression – תהליך של כיווץ נתונים שבו כל המידע נשמר בצורה מדויקת רק מכווץ – בתחום אחזור הנתונים רוב הכיווץ נעשה בסוג זה של כיווץ. (2) Lossy compression – תהליך כיווץ נתונים בו חלק מהמידע אובד בתהליך הכיווץ. שימוש בסוגים כאלו של כיווץ נעשה בתחומים של עיבוד תמונות (למשל פורמט JPEG), עיבוד ווידאו (פורמט MP4) ועיבוד צלילים (MP3).

לחלק מתהליך הקדם עיבוד, כמו case folding, stop words, stemming, number elimination, ניתן להתייחס כ- Lossy compression, וזאת מכוון שלאחר ביצוע תהליכים אלו אין באפשרותנו לשחזר את המידע כפי שהופיע בטקסט המקורי.

<u>מס' ה-term-ים בהתאם לגודלו של אוסף המסמכים</u>

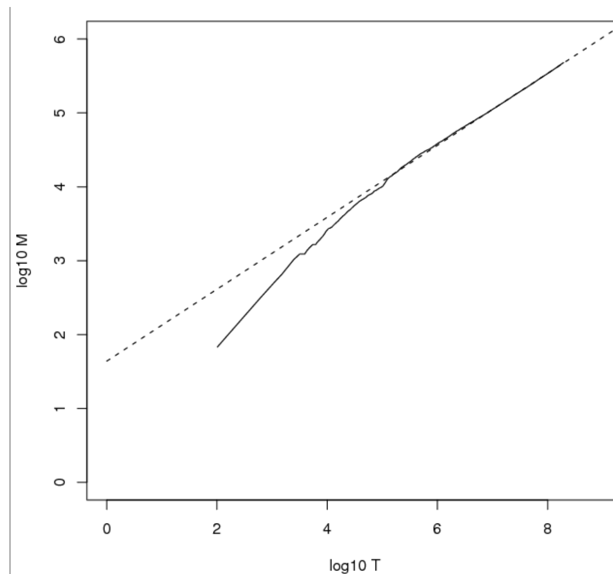
נניח שידוע לנו מהו גודלו של אוסף המסמכים שלנו. האם ניתן לדעת או להעריך מה יהיה גודלו של אוסף ה-terms שלנו (Vocabulary) ? האם ניתן לדעת כמה מילים שונות יש לנו ? (נניח לצורך תכנון מערכת אחזור המידע, בחירת החומרה וכו').

האם ניתן להניח שיש גבול עליון לגודל ה-vocabulary ? נניח שאנו רוצים לחשב גבול עליון לגודלו של ה-vocabulary ללא תלות באוסף המסמכים שלנו. אם נניח שיש לנו מילים בגודל מקסימלי של 20 תווים, הרי אנו יכולים לקבל 70^{20} מילים שונות, כלומר 10^{37} מילים. מספר זה הוא גבוה מאד, וברור שהוא כולל בתוכו צרופים שהם אינן מילים תקניות (סתם אוסף של תווים), ולכן אי אפשר להשתמש בערכה זו לצרכים שונים, כגון תכנון מערכת אחזור מידע.

בפרקטיקה, ככול שנוסיף מסמכים למערכת שלנו, ה-vocabulary יגדל. אולם, ככל שמספר המסמכים שלנו גדל, הוספה של מסמכים חדשים לא תוסיף term-ים חדשים רבים למערכת. ובסופו של דבר, נגיע למצב שבו הוספה של מסמך חדש תגרום לתוספת של term-ים חדשים לעיתים רחוקות בלבד (אבל עדיין יתווספו, למשל שמות, שמות של חברות, מילים חדשות בשפה או בשפות זרות וכו').

חוק אמפירי, שנקרא חוק היפ, מתאר את הקשר בין מספר ה-token-ים באוסף מסמכים מסוים, למספר ה-term-ים שיתקבל עבור אותו אוסף מסמכים. נניח שיש אוסף מסמכים ובו T token-ים, חוק היפ טוען שאת מספר ה-term-ים במילון (M) אפשר לבטא בעזרת השיווין $M = kT^b$, כש-k ו-b הם קבועים כלשהם (לרוב $30 \le k \le 100$, $b \approx 0.5$).

אנו יכולים להשתמש בתרשים log-log, כשבציר ה-X מופיע T (מס' ה-token-ים באוסף המסמכים) ובציר ה-Y מופיע M (מס' ה-term-ים במילון), על מנת לתאר את שינוי ב-T כפונקציה של M (במקרה זה שיווין הוא $\log M = \log k + b \log T$. במקרה זה, חוק Heap (שכאמור מבוסס על נתונים אימפיריים), חוזה שנקבל קו ישר עם שיפוע של 0.5, ואנו מקבלים קשר פשוט בין שני המשתנים.



הגרף הנ"ל הוא עבור RCV1. בגרף זה הקו השחור מציין נתוני אמת, בעוד שהקו המקווקו מציין את פונקציית רגרסיה, כלומר $\log_{10}M = 0.49 \log_{10}T + 1.64$. מכאן, $M = 10^{1.64}T^{0.49}$, ו-$k = 10^{1.64} \approx 44$ & $b = 0.49$.

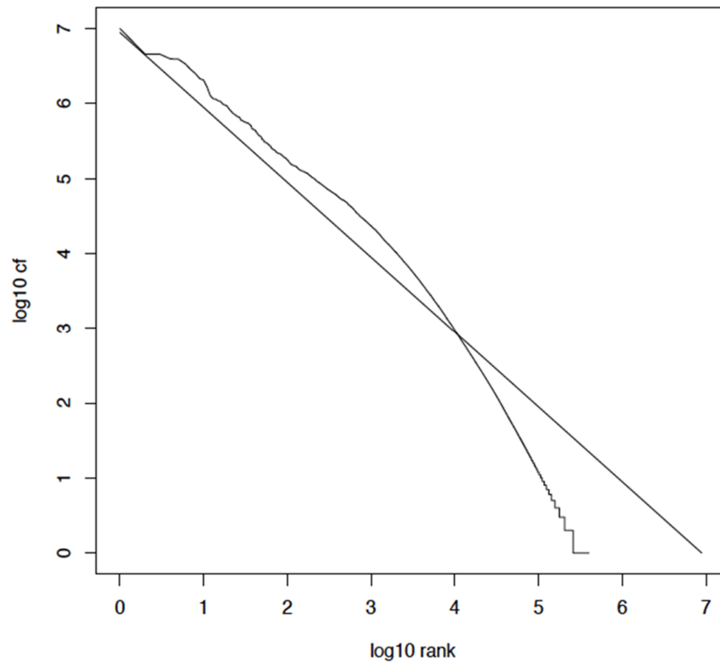
עבור ה-1,000,020 tokens הראשונים החוק חוזה שנקבל 38,323 terms שונים, בפועל מקבלים 38,365 – הבדל קטן מאד (החוק עובד).

דוגמא לשימוש בחוק Heap

נניח שידוע שבאוסף מסמכים שבו יש 1,000 מילים מתקבלים 400 term-ים שונים, וכן שבאוסף מסמכים שבו 10,000 מילים מתקבלים 600 term-ים שונים.

במקרה זה, מכוון שמדובר בשתי מדידות בלבד, אנו יכולים, במקום ברגרסיה לוגריתמים, לעבוד באופן הבא. עביר את הנתונים שקיבלנו לסקאלה לוגריתמית. כלומר, 1000 מילים שווים ל-$\log 1000 = 3$, ו-400 term-ים שווים ל-$\log 400 = 2.6$. באופן דומה, 10,000 מילים שווים ל-$\log 10000 = 4$, ו-600 term-ים שווים ל-$\log 600 = 2.78$. כעט אנו יכולים לבנות משוואת ישר. שיפוע הישר הוא $m = \frac{y_2 - y_1}{x_2 - x_1}$, ובמקרה שלנו הוא $m = \frac{2.78-2.6}{4-3} = 0.18$. בהתאם לך, משוואת הישר היא $y = m(x - x_1) + y_1$, כלומר $y = 0.18(x - 3) + 2.6 = 0.18x + 2.834$. על סמך ישר זה, אנו יכולים לחזות מהו מס' ה-term-ים הקיימים באוסף מסמכים בעל 1,000,000 מילים. במקרה זה, $x = \log 1000000 = 6$. נציב אותו במשוואת קו הישר ונקבל $y = 0.18 \times 6 + 2.834 = 3.914$. מכוון שמדובר בתוצאה שהועברה לסקאלה לוגריתמית, נשאר לנו לבצע את החישוב $10^{3.914} = 8203.5$, שזה מס' ה-term-ים הצפוי.

חוק Heap מתייחס לגודלו של ה-vocabulary ביחס לגודל אוסף המסמכים שלנו (מספר ה-tokens). אנו יודעים שלכל term יש התפלגות יחסית שמתייחסת למס' המופעים שלו באוסף המסמכים. בכל שפה טבעית, יש term-ים שהם יותר שכיחים (מעטים) וכאלו שפחות (רבים). חוק זיפף (Zipf), שגם הוא חוק אמפירי) טוען שה-term ה-i הכי נפוץ, ההתפלגות שלו פרופורציונאלית ל-$1/i$. כלומר, $cf_i \propto 1/i = K/i$, כאשר K הוא קבוע מנורמל, ו-cf_i הוא collection frequency, מספר המופעים של ה-term t_j באוסף המסמכים. נניח שה-term הכי נפוץ הוא t_1, ה-term השני הכי נפוץ הוא t_2, ה-term השלישי הכי הנפוץ הוא t_3, וכך הלאה. נניח שמספר המופעים של t_1 הוא Cf_1, בהתאם לחוק zipf, מספר המופעים של t_2 הוא $Cf_1/2$, כלומר, מחצית ממספר המופעים של Cf_1. מספר המופעים של t_3 הוא $Cf_1/3$, כלומר, שליש ממספר המופעים של Cf_1, וכך הלאה. בכללי, $Cf_i = K/i$, כאשרר k הוא גורם מנורמל, ומכאן $\log Cf_i = \log K - \log i$, ואנו, שוב, מקבלים קשר לינארי בין $\log Cf_i$ ובין $\log i$.



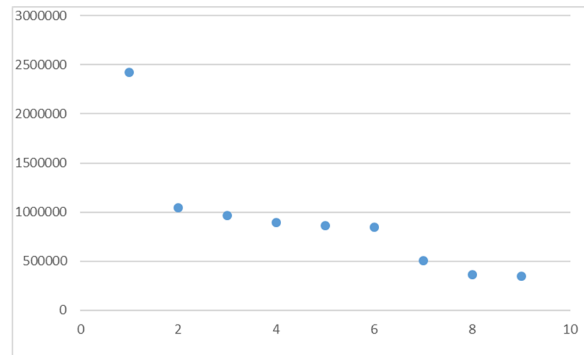
דוגמא לשימוש בחוק Zipf

הטבלה הבאה מתארת את מס' ההופעות של תשעת ה-term-ים הנפוצים ביותר במסמך מסויים.

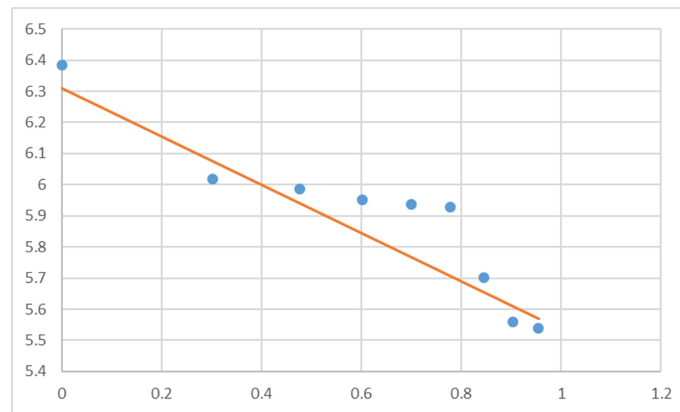
דרוג	מס' הופעות
1	2420778
2	1045733

דרוג	מס' הופעות
3	968882
4	892429
5	865644
6	847825
7	504593
8	363865
9	347072

אנו, כמובן, יכולים להציג את התפלגויות ה-term-ים השונים על גבי גרף, כמו הגרף הבא:



אולם, אם נציג את התפלגויות ה-term-ים השונים על גבי גרף לוג-לוג, וכן, אם נבצע רגרסיה לוגריתמית על הנתונים, נקבל את הגרף הבא:



מהגרף קל לראות, שקיימת מגמה מסוימת באופן פיזור התפלגויות ה-term-ים השונים. חוק זיפף מתאר מגמה זאת באופן פשוט: מס' המופעים של ה-term המדורג במקום ה-i שווה למס' המופעים של ה-term המדורג במקום הראשון לחלק ל-i. הטבלה הבאה מתארת את התוצאות הצפויות במקרה שלנו.

דרוג	מס' מופעים	מס' מופעים צפוי
1	2420778	2420778
2	1045733	1210389
3	968882	806926
4	892429	605195
5	865644	484156

דרוג	מס' מופעים	מס' מופעים צפוי
6	847825	403463
7	504593	345825
8	363865	302597
9	347072	268975

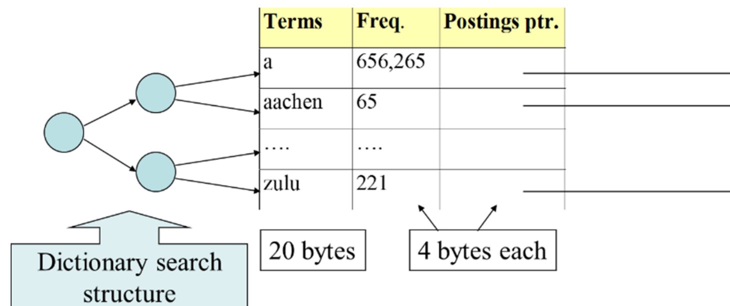
דוגמא נוספת. נניח שידוע שה-term הנפוץ ביותר מופיע 1000 פעמים. כמו כן, ה-term השני הנפוץ ביותר מופיע 250 פעמים. כמה פעמים מופיע ה-term השלישי הנפוץ ביותר ?

בהתאם לשוויון $Cf_i = K/i$, הרי $K/1 = 1000$, ומכאן במקרה זה $K = 1000$. לגבי ה-term השני, מתקיים $250 = K/2$, ומכאן שבמקרה זה $K = 500$. אפשר לראות שבמקרה זה, ערכו של K אינו נשאר זהה עבור שני ה-term-ים. במקרה זה, יש לנו מס' אפשרויות:

1. להשתמש ב-$K = 1000$, ובהתאם לבצע את החישוב. במקרה זה, $Cf_3 = \frac{1000}{3} = 333.33$. די ברור שתוצאה זו אינה אפשרית, מכוון שערכו של Cf_2 נמוך יותר.
2. להשתמש ב-$K = 500$, ובהתאם לערך זה לבצע את החישוב. במקרה זה, $Cf_3 = \frac{500}{3} = 166.67$.
3. לבצע ריגרסיה לוגריתמית, ועל סמך הריגרסייה לחשב את Cf_3. במקרה שלנו, פונקצית הריגרסייה הלוגריתמית יוצא $\log Cf_i = 3 - 2\log i$. בהתאם לכך, $Cf_3 = 111.11$.

כיווץ המילון

מדוע אנו מעוניינים בכיווץ המילון ? תהליך החיפוש מתחיל במילון. בהתאם ל-term-ים שמופיעים בשאילתא, אנו פונים למילון, על מנת לקבל את ה-posting lists של כל אחד מה-term-ים של השאילתא. אם המילון נמצא על הדיסק, אנו צריכים לבצע פניות לדיסק על מנת לקרוא ולחפש את אותם ה-term-ים. וכאמור, פניות לדיסק הינן איטיות, ולכן, בכדי לשפר ביצועים אנו רוצים לשמור את המילון בזיכרון המחשב. במקרה זה, במידה והמילון, כמות שהוא, גדול מזיכרון המחשב, כיווץ מקטין את גודלו, ומאפשר לנו לשמור אותו בזיכרון המחשב. ישנן סיבות נוספות שבגללן נרצה לכווץ את המילון. לדוגמא, (1) זיכרון המחשב לא פנוי רק עבורנו, ישנם תהליכים נוספים שרצים ודורשים זיכרון, (2) ישנם מכשירים (Embedded/mobile devices) שהם יש מעט זיכרון, ו-(3) גם אם לא ניתן לשמור את המילון בזיכרון, אנו רוצים שהוא יהיה קטן בכדי שנוכל לחפש בו מהר.



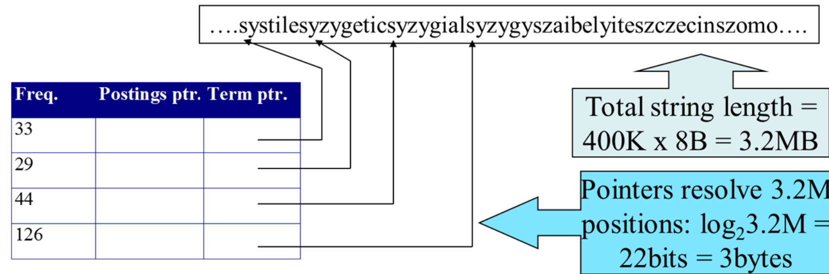
נניח שהמילון שלנו שמור כמערך שבו רשומות באורך קבוע. מבנה הרשומה הוא באופן הבא: (1) ה-term, אשר תופס 20 תווים, (2) מספר המסמכים שבו מופיע ה-term, מיוצג ע"י integer, שתופס 4 תווים, ו-(3) מצביע ל-posting list, שגם הוא מיוצג ע"י integer, ותופס 4 תווים. סה"כ גודלה של רשומה הוא 28 תווים.

במקרה של RCV1, יש לנו כ-400,000 term-ים שונים. מכוון שגודלה של כל רשומה הוא 28 תווים, גודלו של המילון, במקרה זה הוא 11.2 MB.

בדוגמא, הקצאנו 20 תווים עבור כל term. מכוון שיש הרבה terms שהם קטנים מ-20 תווים (אורכה של המילה הממוצעת באנגלים הוא 4.5 תווים ו אורכה של המילה הממוצעת במילון הוא 8 תווים), אנו מבזבזים באופן זה הרבה זיכרון.

Dictionary as a string

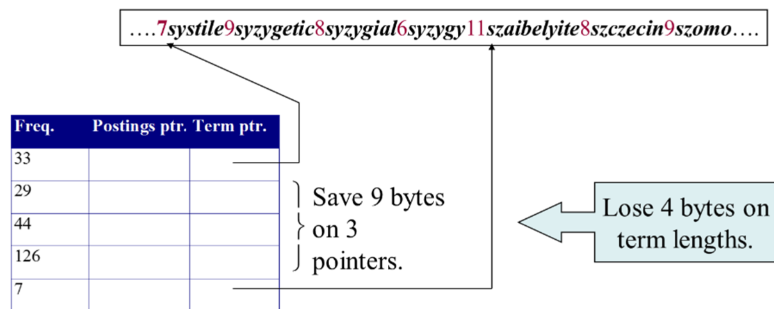
בכדי לחסוך בזיכרון, נעבוד באופן הבא. נשמור את כל ה-terms של המילון כרשימה (string) אחת ארוכה. במקום לשמור את ה-term בכל רשומה, אנו שומרים ברשומה של ה-term מצביע לתחילת ה-term ברשימה שלנו (ה-string). בכדי לדעת מה אורכו של ה-term (בכדי שנוכל "לחלץ" אותו מתוך ה-string הארוך שלנו), אנו נשתמש במצביע לתחילת ה-term שנמצא ברשומה הבאה. באופן זה, אורכו של ה-term שווה ל-NextTermIndex-CurrentTermIndex. באופן זה אנו יכולים לחסוך עד 60% מנפח המילון.



באופן זה אנו משתמשים ב-4 בתים עבור ה-Freq, 4 בתים עבור המצביע ל-Postings ו-3 בתים עבור המצביע ל-term. מכוון שגודלו הממוצע של ה-term הוא 8 בתים, עבור RCV1, גודלו של המילון קטן מ-11.2 MB ל-7.6 MB.

Blocking

שיטה אחרת לכיווץ הנתונים נקראת blocking. בשיטה זו, במקום לשמור את הכתובת של כל term ברשימה, אנו נשמור את הכתובת לכל k term ברשימה (string). נניח ש-k=4, זה אומר שעבור ה-term הראשון, אנו נשמור את המצביע אליו ב-string הארוך. ה-term הבא שאנו נשמור את המצביע אליו ב-string הארוך, הוא ה-term החמישי וכך הלאה. באופן זה, אנו חוסכים (לא שומרים) 3 מצביעים למיקומים ב-string של 3 term-ים, ובכך חוסכים 9 בתים.



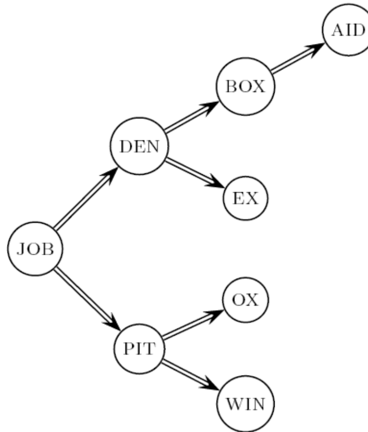
בכדי שנוכל לדעת איפה מתחיל ומסתיים כל term ב-string (יש לנו את המיקום של תחילת כל k=4, אבל אנו לא יודעים מה קורה בתוך החלק הזה), עבור כל term אנו צריכים לשמור את האורך שלו (תוספת של בית אחד, שיכול להכיל ערכים בין 0 ל-255, שמתווסף בהתחלה, לפני ה-term עצמו).

נניח ש-k=4. לפני ה-Blocking שמרנו 4 מצביעים בגודל 3, סה"כ 12 בתים. אחרי ה-Blocking שמרנו מצביע אחד בגודל 3, וכן ארבעה בתים לסמן את הגודל של כל term, סה"כ 7 בתים. באופן זה חסכנו ~0.5MB, והגודל של המילון שלנו קטן מ- 7.6 MB ל-7.1 MB

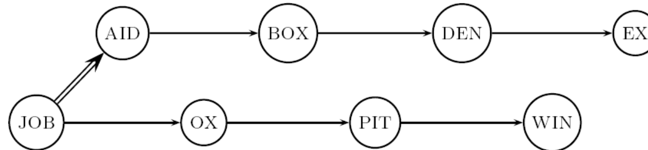
ניתן לחסוך מקום נוסף אם נשתמש ב-k גדול יותר, אך האם כדאי להשתמש ב-k גדול ? לא מכוון שאז זמן החיפוש של term המתאים יהיה לינארי.

מהו מספר ההשוואות שאנו מבצעים בכל אחת מהשיטות ?

נניח שאנו לא משתמשים ב-blocking, וכן כל term במילון יכול להופיע בשאילתא בהסתברות זהה (לא קורא במציאות). במקרה כזה, מספר ההשוואות הממוצע שנעשה הוא $(1+2\cdot2+4\cdot3+4)/8 \sim 2.6$.



במידה ואנו כן משתמשים ב-blocking, אז אנו מבצעים חיפוש בינארי עם, נניח, k=4 terms בכל בלוק (כשבכל בלוק נעשה חיפוש לינארי), ולכן מספר ההשוואות הממוצע שנעשה הוא $(1+2\cdot2+2\cdot3+2\cdot4+5)/8 = 3$



Front Coding

אם נסתכל במילים ממוינות, אנו נראה שיש הרבה מילים שיש להם תחיליות זהות וסיומות שונות.

8automata8automate9automatic10automation

אם כך, למה לשמור את כל התחיליות ?

→8***automat****a*1◊***e***2◊***ic***3◊***ion***

Encodes ***automat***

Extra length beyond ***automat.***

בדוגמא, יש לנו בלוק בגודל 4, שבו כל המילים מתחילים ב-automat. המילה הראשונה מכילה גם את המילה עצמה וגם את התחילית. המילה הראשונה היא בגודל 8. כאשר 7 התווים הראשונה מהווים את התחילית, והתו הנוסף הוא לצורך השלמת המילה הראשונה. המיקום של התחילית במילה הראשונה מסומן ע"י התו "*", שמשמעו שעד לתו זה, רצף התווים מהווה את התחילית המשותפת. כמו כן, עבור שאר המילים, אנו שומרים את אורך התוספת ואת הסופית שיש לצרף לתחילית.

הטבלה הבאה מסכמת את גודל המילון שמתקבל בכל אחת מהשיטות שהזכרנו.

Technique	Size in MB
Fixed width	11.2

Dictionary-as-String with pointers to every term	7.6
Also, blocking $k = 4$	7.1
Also, Blocking + front coding	5.9

כיווץ ה-posting lists

קובץ ה-postings גדול לפחות פי 10 בערך מקובץ המילון. מפאת גודלו של הקובץ, אנו רוצים לשמור כל posting בשלמותו בקובץ מכווץ. לצרכים שלנו, posting זה docID (כלומר מספר), ואנו לא מתייחסים למיקומים של הביטוי בתוך המסמך עצמו. עבור RCVQ (800,000 מסמכים) אנו נשתמש ב-int (4 בתים, 32 ביטים) לשמור את ה-docID. 32 ביט מאפשר לנו לעבוד עם הרבה יותר מ-800,000 מסמכים, למעשה, אנו יכולים להסתפק ב- $\log_2 800{,}000$ שזה 20 ביטים. המטרה שלנו היא להשתמש בפחות מ-20 ביטים לכל docID.

נניח שיש term כמו arachnocentric, שהשכיחות שלו מאד נמוכה. הוא יכול להופיע במסמך אחד מתוך מיליון. במקרה כזה נרצה לשמור את ה-posting שלו בעזרת $\log_2 1M \sim 20$ ביטים. מנגד, term כמו the נמצא כמעט בכל מסמך. במקרה הזה שימוש ב-20 ביטים הינו יקר מאד. ולכן, עדיף ווקטור של 0/1 במקרה זה

כל רשומה מכילה רשימה ממוינת של docID בסדר עולה

computer: 33,47,154,159,202 …

מכוון שהרשימה ממוינת, אפשר לשמור את הפער בין שני docIDs. באופן כזה, נוכל (אנו מקווים) לשמור את הרשימה כשאנו משתמשים בפחות מ-20 ביטים

	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							

כאמור, המטרה שלנו היא, שעבור term כמו arachnocentric, אנו נשתמש ב-20 ביט בשביל לשמור את הפער בין ה-docID. ואילו עבור term כמו the, נשתמש ב-1 ביט בשביל לשמור את הפער בין docID. אם הפער הממוצע בין ה-docIDs של term נתון הוא G, אנו מעוניינים להשתמש ב- $\sim\log_2 G$ בכדי לשמור את הפער. במילים אחרות, אנו מעוניינים לשמור את הפערים עבור על term בעזרת כמה שפחות ביטים. לצורך כך, אנו צריכים קידוד עם אורך משתנה. קידוד עם אורך משתנה מקצה קוד קצר למספרים קטנים (הפרשים קטנים), וקידוד ארוך למספרים גדולים (הפרשים גדולים). עבור כל פער G, אנו נשתמש במספר הבתים הקטן ביותר שנדרש על מנת לשמור $\log_2 G$ ביטים. אנו נתחיל עם בית אחד בשביל לשמור את G. בבית הזה נקצה ביט אחד לצורך המשכיות, הביט c. כל עוד $G \leq 127$ אנו יכולים להשתמש ב-7 ביטים הנותרים בבית. במקרה זה c=1. אחרת, נשתמש בבית נוסף (שני בתים שבכל אחד אפשר להשתמש ב-7 ביטים). ביט ההמשכיות של הבית הראשון הוא c=0, ושל הבית השני הוא c=1.

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Postings stored as the byte concatenation
000001101011100010000101000011010000110010110001

Key property: VB-encoded postings are uniquely prefix-decodable.

For a small gap (5), VB uses a whole byte.

במקום להשתמש בבתים אפשר להשתמש ביחידות אחרות, למשל 32 ביט (word), 16 ביט, 4 ביט (nibbles). שימוש באורך משתנה הינו בזבזני אם יש לנו הרבה פערים קטנים. אולם, הוא נמצא בשימוש בהרבה מערכות מסחריות / מחקריות.

בכדי לבצע כיווץ טוב יותר, אנו צריכים להשתמש ב-Unary Code. ב-Unary Code, הייצוג של המספר n הוא ע"י n אחדות, ובסוף הספרה 0. לדוגמא, Unary code של 3 הוא 1110. Unary code של 40 הוא 11111111111111111111111111111111111111110. Unary code של 80 הוא 111111111111111111111111111111111111111111111111111111111111111111111111111111110.
כמובן, ששימוש ב-Unary Code, כשעצמו אינו יעיל, מכוון שהוא "בזבזני" מבחינת הביטים בהם הוא משתמש לייצוג מספרים שונים.

נוכל לבצע כיווץ טוב יותר,Gamma Code, אם נעשה זאת ברמת הביטים. כלומר, נייצג פער G כזוג length ו-*offset*. Offset שווה ל-G בייצוג בינארי, שאת הביט במוביל (השמאלי) אנו מסירים, לדוגמא $13 \rightarrow 1101 \rightarrow 101$. Length הוא אורכו של ה-offset, לדוגמא עבור 13 (101), זה 3. אנו מקודדים את length בעזרת unary code (ומקבלים 1110). Gamma code של 13 הוא החיבור של length ו-offset (1110101).

להלן מס' דוגמאות:

number	length	offset	g-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

ב-Gamma Code, מספר G מקודד ע"י $2\lfloor \log G \rfloor + 1$ ביטים, כשהאורך של offset הוא $\lfloor \log G \rfloor$ ביטים, והאורך של length הוא $\lfloor \log G \rfloor + 1$ ביטים. בהתאם לכך, כל Gamma code מקודד ע"י מספר זוגי של ביטים. שימוש ב-Gamma code הוא פי 2 מהאפשרות הטובה ביותר, $\log_2 G$

למחשבים יש גודל קבוע של מילים, 8, 16, 32 או 64 ביטים. פעולות שנעשות על מעבר למילה אחת, איטיות יותר. ובהתאם לכך, גם פעולת הכיווץ תהייה איטית יותר. מכוון ש-Gamma Code הוא לא byte aligned, הוא איטי, ולכן לא נמצא בשימוש פרקטי. כיווץ Variable byte הוא יעיל יותר, מכוון שעובד עם בתים שלמים. מעבר לכך, כיווץ Variable byte הוא פשוט יותר, ולרוב תוספת הנפח שלו היא זניחה.

הטבלה הבאה מסכמת את הגדלים השונים של מילון וה-posting list שמתקבלים לאחר השימוש בשיטות הכיווץ השונות.

Data structure	Size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
with blocking, k = 4	7.1
with blocking & front coding	5.9
collection (text, xml markup etc)	3,600.0
collection (text)	960.0
Term-doc incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, g-encoded	101.0