oren.e.nahum@gmail.com

אחזור מידע

מבוא

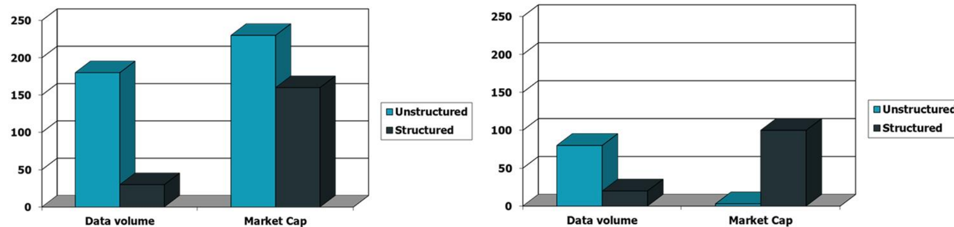
נניח שיש לנו אוסף גדול מאד של מסמכים השמורים על גבי מחשב אחד או יותר (נהוג לכנותם בשם קורפוס). כמו כן, יש נושא לגבי אני מעוניין במידע (צורך מסוים). חלק מהמסכמים בקורפוס מכילים את המידע בו אני מעוניין, כלומר אותם מסמכים הם בעלי עניין, או רלוונטיים, עבורי. אבל איך אני יכול לקבל את המסמכים הללו, או במילים אחרות, איך אני יודע מי מהמסמכים רלוונטי עבורי. בכדי לקבל את המסמכים הרלוונטיים אני צריכים לבטא את הצורך שלנו בצורה של שאילתא, שבעזרתה אנו מתשאלים את אותו מחשב/ים, שבהתאם לכך יודעים לחפש ולהביא את אותם המסמכים.

האופן בו השאילתא נכתבת תלוי באם המסמכים הם בעלי מבנה מוגדר או לא. דוגמא למסמך בעל מבנה מוגדר היא טבלה. בטבלה אנו יודעים בברור מה המשמעות של כל עמודה, ולעיתים, מה המשמעות של כל תא בטבלה. לדוגמא, בבסיס נתונים טבלאי, הנתונים נשמרים בטבלאות. אנו יודעים מה המשמעות של כל רשומה, ואף כל שדה ברשומה. כמו כן, עומדים לרשותנו כלים חזקים, כמו SQL, שמאפשרים לנו לתשאל את בסיס הנתונים ולקבל את המידע מהטבלאות השונות.

למסמך בעל מבנה לא מוגדר, אין, כמו שנרמז, מבנה מוגדר, ואנו לא יכולים לדעת במה ואיך עוסק המסמך. במקרים כאלו אנו לא יכולים לבצע שאילתות מורכבות כמו בבסיסי נתונים. אבל אנו יכולים לשאול אם ביטויים מסוימים נמצאים במסמך.

אחזור מידע (Information Retrieval – IR) הוא מציאת חומר (בדרך כלל מסמכים) בעלי מבנה לא מוגדר (בדרך כלל טקסט) המספק מידע נדרש מתוך אוספים גדולים (בדרך כלל מאוחסנים במחשבים).

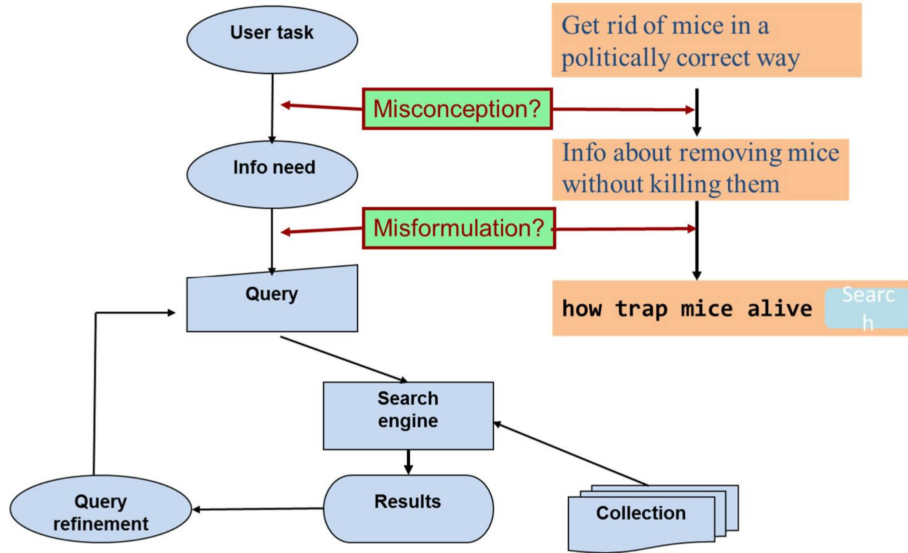
כאשר מדברים על אחזור מידע, הדבר הראשון העולה לראש הוא חיפוש באינטרנט, אבל יש הרבה מקרים אחרים הקשורים לאחזור מידע: (1) חיפוש ב-e-mails, (2) חיפוש במחשבים, (3) חיפוש במערכות אירגוניות ו-(4) חיפוש במסמכים משפטיים.



הגרף מימין מראה שכבר בשנת 1996 כמות המסמכים בעלי המבנה הלא מוגדר הייתה גדולה פי כמה מכמות המסמכים בעלי המבנה המוגדר. ולמרות זאת, מעט מאד חברות התעסקו בניתוח מסמכים בעלי מבנה לא מוגדר. בשנת 2009 לעומת זאת (הגרף משמאל), כמות החברות שהתעסקו בניתוח מסמכים בעלי מבנה לא מוגדר גדלה משמעותית ואף עברה את כמות החברות שהתעסקו עם מסמכים מובנים (אנו רואים פיתוח של הרבה מנועי חיפוש למיניהם).

במציאות, גם כאשר אנו מדברים על מסמכים בעלי מבנה לא מוגדר, הם לא באמת כאלו. לדוגמא, למסמך הזה יש כותרת, שיכולה לרמז על תוכן המסמך. מאמרים אקדמאים יש להם מבנה מסוים (כותרת, אבסטרקט, מילות מפתח, ביבליוגרפיה וכו'). אפשר להשתמש במידע החצי מובנה הזה לצורך שליפת מסמכים (למשל, הבא את המסמכים שבכותרת שלהם מופיעות המילים....).

אנו עובדים עם אוסף של מסמכים, כלומר קבוצה של מסמכים, בשלב זה קבוצה קבועה שלא משתנה. לאוסף מסמכים זה, כאמור, נהוג לקרוא קורפוס. המטרה שלנו היא אחזור מסמכים המכילים מידע הרלוונטי למשתמש, והעוזרים למשתמש בביצוע משימה.



תהליך אחזור המידע מתחיל עם בעיה מסוימת של המשתמש, שבעקבותיה נוצרת דרישה מסוימת של המשתמש למידע (צורך). הדרישה הזאת מתורגמת לשאילתא, אשר מועברת למנוע החיפוש. מנוע החיפוש בודק אלו מהמסמכים עונים לשאילתא, ומחזיר אותם למשתמש. המשתמש בודקת את התשובה המתקבלת, ובמידה והיא לא עונה על הצורך (נגיד, מס' רב של מסמכים לא רלוונטיים), המשתמש חוזר ומנסח את השאילתא שלו מחדש, בצורה מדויקת יותר.

בכדי לדעת עד כמה טוב הפתרון שלנו, כלומר עד כמה קבוצת המסמכים שהחזרנו עונה לצרכי המשתמש אנו נגדיר שני מדדים: בהינתן שאילתא, יהי R קבוצת המסמכים הרלוונטיים ו-A קבוצת המסמכים שאוחזרו. המדד הראשון הוא דיוק – קבוצת המסמכים המאוחזרים הרלוונטיים לצורכי המידע של המשתמש מתוך כלל המסמכים שאוחזרו - $\frac{R \cap A}{A}$. המדד השני הוא כיסוי – קבוצת המסמכים המאוחזרים הרלוונטיים לצורכי המידע של המשתמש מתוך כלל המסמכים הרלוונטיים למשתמש - $\frac{R \cap A}{R}$.

<u>מטריצת מפגשים לביטויים במסמכים</u>

נניח שאנו מעוניינים לדעת אלו מהמחזות של שייקספיר (הקורפוס שלנו) מכילים את המילים Brutus וגם Caesar אבל לא את Calpurnia. נתחיל עם פתרון נאיבי. בכדי להגיע לפתרון אפשר לחפש (למשל באמצעות grep), בכל המחזות, את השורות שמכילות Brutus ו/או Caesar, ומהתוצאה להסיר את כל השורות שמכילות את המילה Calpurnia. לצורת צורת חיפוש מספר חסרונות: (1) איטי – כל פעם שנבצע שאילתא אנו נצטרך לסרוק את הקבצים מחדש על מנת לדעת אם ביטוי מסוים נמצא בהם או לא (במקרה של קורפוס מאד גדול לא פתרון מעשי), (2) שיטה זו לא נותנת פתרון ל-"לא את Calpurnia", (3) אופרטורים אחרים (למשל המילה Romans נמצאת ליד המילה countrymen) אינם ניתנים למימוש בשיטה זו, ו-(4) אי אפשר לדרג את הפתרונות.

	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

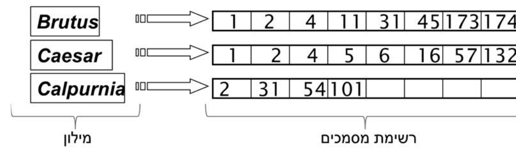
אנו יכולים להשתמש במטריצת מפגשים בכדי לציין אלו ביטויים נמצאים בכל מסמך. את מטריצת המפגשים אנו בונים מראש. לכל מסמך יש ווקטור מפגשים המתאים לו, בו כל ביטוי יכול לקבל את ערך 0 (כשהביטוי לא נמצא במסמך) או 1 (כשהביטוי נמצא במסמך). באופן דומה, לכל ביטוי יש ווקטור מפגשים, בו כל מסמך יכול לקבל את ערך 0 (כשהביטוי לא נמצא במסמך) או 1 (כשהביטוי נמצא במסמך). מטריצת המפגשים אינה שומרת את מס'

הפעמים שכל ביטוי מופיע בכל מסמך, וכן, אנו לא יודעים היכן במסמך (מיקום) נמצא כל ביטוי. בכדי לענות על השאילתא שלנו, אנו לוקחים את הווקטורים של הביטויים Brutus, Caesar והמשלים של Calpurnia (כי אנו רוצים שלא יכיל את Calpurnia), ונבצע עליהם פעולת bitwise AND. כלומר, 110100 and 110111 and 101111 = 100100, ומכאן שהתשובה היא Antony and Cleopatra ו-Hamlet.

נניח שיש לנו מיליון מסמכים, שבכל אחד מהם כ-1000 מילים, כשבממוצע מילה מורכבת משישה תווים (כולל רווחים ותווי פיסוק), כלומר 6GB של מידע. כמו כן, נניח שיש 500,000 מושגים שונים במסמכים האלו. במקרה כזו, אנו מדברים על מטריצת מפגשים בגודל $500K\times1M$ של אפסים ואחדות (אם אנו משתמשים בבית אחד עבור כל תא, גודל המטריצה בזיכרון הוא 500GB). ברוב במקרים המטריצות האלו הן ממש דלילות, כלומר, מספר האחדות הוא נמוך משמעותית מסך התאים שבמטריצה (במקרה שלנו $1K\times1M$, שזה 0.2% מסך התאים במטריצה). אם כך, אנו צריכים למצוא שיטה טובה יותר לייצג את המידע, שתשמור רק את המידע של האחדות.

Inverted Index – אינדקס מסמכים

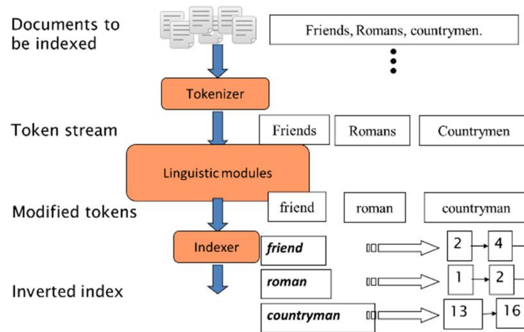
עבור כל מושג, t, אנו צריכים לשמור רשימה של כל המסמכים בהם הוא נמצא. נניח שכל מסמך מיוצג על-ידי מספר סידורי, docID, אנו יכולים לייצר את הרשימה הבאה.



למבנה הזה אנו קוראים Inverted Index, כאשר רשימת הביטויים נקראת מילון (dictionary) ורשימת המסמכים נקראת posting list.

מכוון שאנו רוצים לחסוך במקום / זיכרון, אנו נשתמש ברשימות דינמיות (כלומר, מבנה נתונים שהגודל שלו אינו קבוע, ויכול להשתנות בהתאם לכמות המידע הנשמר).

נתבונן בתהליך הכללי של בניית אינדקסים.



(1) Tokenization – עבור כל מסמך, אנו חותכים את רצף התווים למילים (tokens). כמו כן, אנו, בין היתר, מורידים סימני פיסוק. (2) Normalization - העברת מילים וביטויים הנכתבים בצורה שונה למצב זהה (למשל USA ו-U.S.A). (3) Stemming - מציאת השורש של המילים והביטויים השונים (למשל authorize, authorization). (4) Stop words - הסרה (או לא) של מילים נפוצות כגון the, a, to, of.

בכדי ליצור את ה-inverted index אנו (1) חותכים המסמכים למילים, כל מילה עוברת דרך ה-Linguistic modules, כשאת הביטוי המתקבל (term) משייכים למסמך (docID). בשלב הזה אנו מקבלים רשימה של זוגות, <ביטוי, מס' מסמך>. (2) מיון הרשימה. אנו ממיינים את רשימת הזוגות שלנו לפי הביטויים, בסדר עולה. (3) הורדת כפילויות. מכוון שאותו ביטוי יכול להופיע מס' פעמים באותו הטקסט, או במספר מסמכים שונים, אנו מבצעים פעולה של הורדת כפילויות. כתוצאה מפעולה זה אנו מקבלים שני תוצרים. האחד, רשימה של ביטויים ומספר המופעים שלהם באוסף המסמכים שלנו (מס' מופעים באותו מסמך נספרים פעם אחת בלבד), וכן, עבור כל ביטוי, רשימה מקושרת של מספרי המסמכים בהם הוא נמצא (posting list).

ביצוע שאילתות בעזרת Inverted Index

נבחן את השאילתא הבאה: Brutus AND Caesar. תחילה, יש לאתר את Brutus בקובץ המילון, ובהתאם לקבל את רשימת הקבצים בהם הוא נמצא. שנית, יש לאתר את Caesar בקובץ המילון, ובהתאם לקבל את רשימת הקבצים בהם הוא נמצא. בכדי לדעת באיזה מסמכים נמצאות המילים Brutus וגם Caesar, נעבור על שתי הרשימות ונחפש מסמכים המשותפים לשני הביטויים. אם ה-posting lists ממוינים, אפשר לעבור על שתיהן במקביל (בדומה ל-marge sort), ואז שסיבוכיות התהליך היא $\mathcal{O}(x+y)$, כאשר x הוא מספר האיברים ברשימה הראשונה, ו-y הוא מספר האיברים ברשימה השנייה.

```
INTERSECT(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 p1 ← next(p1)
 9              else p2 ← next(p2)
10  return answer
```

האלגוריתם הנ"ל מתאר את אופן המעבר על שתי הרשומות במקביל. אנו מתחילים עם תשובה ריקה (שורה 1). p1 ו-p2 הם מצביעים ל-docID בכל אחת מהרשימות. אם p1 או p2 שווים ל-NIL סיימנו. אם לא, אז אנו בודקים docID(p1) שווה ל-docID(p2), אם כן, הרי שמצאנו מסמך משותף, ואנו נוסיף אותו לתשובה שלנו (שורה 4). במקרה זה אנו גם נקדם את p1 ו-p2. אם הם לא שווים, הרי שלא מצאנו מסמך משותף. במקרה הזה אנו לא נקדם את שני המצביעים, אלא רק את המצביע שמצביע ל-docID קטן יותר.

שאילתות של OR, כגון Brutus OR Caesar, עובדות באופן דומה. אנו עוברים על שתי הרשימות, ומוסיפים את כל אחד מהאיברים. אם איבר מסוים נמצא בשתי הרשימות, נוסיף אותו פעם אחת בלבד.

<u>שאילתות בוליאניות</u>

שאילתא בוליאנית היא שאילתא הכוללת בתוכה אופרטורים בוליאניים כגון "וגם", "או" ו-"לא" על מנת לחבר תוצאות של שאילתות על ביטויים פשוטים (זאת לא צורת השאילתא במערכות IR חדשות כמו גוגל). לדוגמא: WestLaw, החברה הגדולה ביותר המספקת שרותי חיפוש בתיקים משפטיים שונים, עם בערך 700,000 משתמשים. השרות שלהם התחיל ב-1975. הם מספקים שרות חיפושים מבוסס שאילתות בוליאניות, שעדיין נמצא בשימוש ע"י מספר רב של משתמשים, וזאת למרות שכבר ב-1992 הם אפשרו שיטות חיפוש אחרות.

נניח שאנו רוצים לענות על השאלה הבאה:

What is the statute of limitations in cases involving the federal tort claims act?

השאילתא, בפורמט של WestLaw תהייה

LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM

כאשר עומדות ברשותנו מס' אפשרויות נוספות, כגון /3 = within 3 words, /S = in same sentence.

השאילתא הזאת אומרת, תביא את כל המסמכים שכוללים מילה המתחילה ב-Limit, וכן את המילים Statute, Action, Federal, Tort ו-Claim. כאשר המילים Limit ו-Statute צריכים להיות במרחק של לא יותר מ-3 מילים במסמך. המילים Federal ו-Tort צריכים להופיע במרחק של לא יותר מ-2 מילים במסמך. והמילים Tort ו-Claim צריכים להופיע במרחק של לא יותר מ-3 מילים במסמך. המילים Action ו-Federal צריכים להופיע באותו המשפט.

האם ניתן להשתמש -inverted index (עם שינויים קלים) על מנת לבצע את השאילתות הבאות: (1) Brutus AND NOT Caesar. פעולת NOT משמעותה על ה-docID שלא נמצאים ב-posting list. (2) Brutus OR NOT Caesar. האם זמן הסיבוכיות היא עדיין $\mathcal{O}(x+y)$? פעולת NOT, במקרה שה-posting list קצר והקורפוס מאד גדול לא ניתנת לביצוע בזמן ריצה לינארי. מה לגבי שאילתות בוליאניות כלליות, כגון: (Brutus OR Caesar) AND NOT (Antony OR Cleopatra). האם זמן הריצה הוא תמיד לינארי ?

מהו הסדר הטוב ביותר לעיבוד השאילתות. נניח שיש שאילתת AND של n ביטויים. עבור כל אחד מהביטויים צריך לקבל את רשימת המסמכים בו הוא נמצא, ולבצע AND על כל הרשימות. כיצד נבצע את הפעולה הזאת בצורה הטובה ביותר, כלומר, בזמן ריצה קצר ביותר וכמה שפחות זיכרון. בכדי לבצע כמה שפחות פעולות, יש להתחיל עם הביטויים שיש להם הכי פחות מופעים. בדוגמא שלנו נבצע תחילה Calpurnia AND Brutus, ועל התוצאה המתקבלת נבצע AND Caesar. עבור שאילתות שמכילות ביטויי OR, למשל (madding OR crowd) AND (ignoble OR strife), יש לקבל את מספר המופעים של כל ביטוי. להעריך את הגודל של כל תוצאת OR, ע"י סכימת המופעים שלהם. לבצע את השאילתות בהתאם לגודל המוערך שהתקבל. זאת הסיבה שאנו שומרים את מספר המופעים (ששווה לאורך ה-posting list) עבור כל ביטוי.

שאילתות של ביטויים

אנו מעוניינים לענות על שאילתות הכוללות ביטויים כגון "מכללה האקדמית תל-אביב – יפו". למשל, המשפט "אני לומד ב**מכללה האקדמית** אשר נמצאת ב**תל-אביב - יפו**", אינה עונה על השאילתא שלנו.

פתרון אפשרי הוא בעזרת Biwords. יש לאנדקס כל שתי מילים סמוכות בטקסט כביטוי. לדוגמא, עבור הטקסט "Friends, Romans, Countrymen" לקבל את ה-Biword-ים הבאים: (1) friends romans ו-(2) romans countrymen. כל אחד מה-Biword-ים היא ביטוי עבור המילון שלנו. באופן זה, אנו יכולים לעשות חיפושים מידיים על כל זוגות המילים בטקסט.

שאילתות על ביטויים ארוכים ניתן לבצע ע"י חלוקתם לביטויים קצרים יותר. את הביטוי "Stanford university Palo alto" ניתן לאחזר ע"י שימוש בשאילתא הבוליאנית הבאה: "Stanford university" AND "university Palo" AND "Palo alto". שאילתא מסוג זה עלולה להחזיר תוצאות שגויות. במקרה כזה, אנו צריכים את המסמכים עצמם בכדי לבדוק אם קיבלנו את התשובות הנכונות. חסרונות השיטה: (1) תוצאות שגויות. (2) קובץ אינדקס גדול כתוצאה מהגדלת המילון. (3) אינדקס Biword אינו הפתרון במקרים כגון אלו, אך יכול להוות חלק מפתרון כולל יותר.

ב-positional index, לכל ביטוי, אנו שומרים את מס' הקבצים שמכילים את הביטוי וכן את המיקום של הביטוי בכל קובץ.

<term, number of docs containing term;
doc1: position1, position2 … ;
doc2: position1, position2 … ;
etc.>

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

בכדי לבצע שאילתות על ביטויים, אני משתמשים באלגוריתם המיזוג באופן רקורסיבי על גבי המסמכים. אולם במקרים כאלו, אנו צריכים להתמודד לא רק עם שוויוניים.

נניח שאנו מחפשים את הביטוי "to be or not to be". עבור כל מילה אנו שולפים את רשימת המיקומים שלה בכל אחד מהמסמכים.

to: 2:1,17,74,222,551; 4:8,**16**,**190**,**429**,**433**; 7:13,23,191; ...

be: 1:17,19; 4:**17**,**191**,291,**430**,**434**; 5:14,19,101; ...

כעט, לאחר שזיהינו מסמכים משותפים, אנו מחפשים סמיכות בין שתי מילים (למשל 16 ו-17, 190 ו-191 ו-433 ו-434). אנו יכולים גם לפסול מסמכים מסוימים על סמך האינדקס הזה, באופן פשוט יחסית. למשל בביטוי שלנו, המילים to ו-be מופיעים פעמיים. המרחק בין to ל-to הוא 5 מילים, וכנ"ל לגבי be. מסמך מס' 2, למשל, מכיל שני מופעים בלבד של המילה be, שהמרחק בניהם הוא 146 מילים, הרבה מעבר ל-5 המילים שאנו דורשים, ולכן המסמך הזה הוא לא חלק מפתרון השאילתא שלנו.

oren.e.nahum@gmail.com

שימוש באינדקס מיקומים מגדיל באופן משמעותי את שטח האחסון הנדרש (למרות שניתן לכווץ את הקובץ הנ"ל). בכל זאת, למרות חיסרון זה, שימוש באינדקס מיקומים נמצא בשימוש באופן סטנדרטי בגלל הכוח והתועלת המתקבלים בעת ביצוע שאילתות על ביטוי. עבור כל ביטוי אנו צריכים לשמור את כל המיקומים שלו במסמך, בניגוד לאינדיקציה אם הוא נמצא או לא. לפיכך, גודלו של אינדקס מיקומים תלוי בגודלם של המסמכים שלנו. בהנחה שהשכיחות של כל מושג במסמך היא 0.1% מתקבל

Document size	Postings	Positional postings
1000	1	1
100,000	1	100

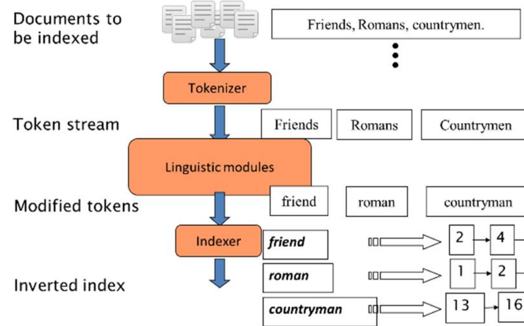
גודלו של אינדקס מיקומים הוא פי 2 עד פי 4 יותר מאשר אינדקס רגיל. עבור מסמך נתון, גודלו של אינדקס המיקומים מהווה בין 35% ל-50% מגודלו של המסמך המקורי.

oren.e.nahum@gmail.com

אחזור מידע

Document ingestion

כאמור, אנו עובדים עם מסמכים. אם נחזור ונסתכל על תהליך יציאת ה-Inverted Index, הרי שהתהליך מתחיל עם פרוק המסמכים למילים. ובכן, מהו בדיוק מסמך ?



עד עכשיו התייחסנו למסמך כאוסף של מילים. אולם בהרבה מקרים זה, לא כך. במסמכים רבים אוסף המילים שלנו "מוחבאים" בין נתונים נוספים. בחלק מהמסכים הם שמורים בצורה "מוצפנת" או "מעורבלת", ולכן יתכן ויהיה שלב מקדים והוא המרה של המסמך מפורמט אחד לפורמט שני.

אם כך, מהן הפעולות, או מהם האתגרים שאיתם אנו צריכים להתמודד בשלב המקדים.

- פורמט של המסמך אותו אנו מנתחים. כאמור, המסמכים מגיעים בפורמטים שונים, כגון, pdf, word, excel, html, וכו'. הפורמטים השונים "מקודדים" את המילים בצורות שונות, ואנו צריכים לחלץ את המילים מתוך הפורמט של המסמך.
- השפה בה כתוב המסמך. אנו צריכים לזהות את השפה בה כתוב המסמך, מכוון שהן ה-Tokenizer והן ה-Linguistic modules תלויים בה.
- אוסף התווים (קידוד) שבו נעשה שימוש במסמך. כידוע, ישנם סוגים שונים של אוספי תווים, למשל CP1525, CP1255, UTF8, וכדומה. אוסף התווים בו נעשה שימוש קובע למעשה איך אנו מתרגמים תו (או אוסף תווים) לאותיות, ולכן חשוב לנו לדעת מהו אוסף התווים שבנו נעשה שימוש, על מנת לקבל את האותיות והמילים הנכונות.

כל אחד מהבעיות הללו היא בעיה סיווג, אשר נלמד בהמשך הקורס. ברור המקרים, הטיפול בבעיות אלו נעשה ע"י שימוש בהיוריסטיקות שונות (למשל, ע"י שימוש בסיומת הקובץ, או בחינה של meta-data של הקובץ).

האתגרים ההלו יכולים להיות מסובכים או קשים ביותר במספר מקרים. להלן מס' דוגמאות.

- המסמכים אותם אנו מאנדקסים יכולים להיות בשפות שונות. המשמעות היא שקובץ האינדקס שלנו יכיל מושגים משפות שונות.
- יתכן שקובץ (או קובץ + מרכיבים נוספים, כגון מסמכי השלמה שונים וכו') יכיל שפות שונות ו/או פורמטים שונים. למשל, קובץ אי-מייל בעברית עם קובץ מצורף באנגלית. מסמך וורד בעברית המכיל ציטוט באנגלית.

קיימות היום ספריות קוד מוכנות, הן מסחריות והן חופשיות, המסוגלות להתמודד עם הבעיות הללו.

עד עתה השתמשנו במושג מסמך. אנו מאנדקסים מסמכים. אנו מחפשים מסמכים. מהו מסמך ?

- האם זה קובץ ?
- האם זה אי-מייל ? אם כן, איך נתייחס לאי-מייל אשר מצורפים אליו קבצים נוספים. תוכנות מסוימות שומרות ספרייה (למשל את ה-Inbox) עם כל המיילים שבו כקובץ אחד – איך להתייחס לכך ?
- האם זה אוסף של קבצים (למשל עבודה הכתובה בקובץ וורד ומחולקת למספר קבצים) ?
- ספר – האם נתייחס לספר כולו כמסמך אחד, או שאולי, נתייחס לכל פרק כמסמך אחד (נניח שיש לנו ספר על ההיסטוריה של אירופה בימי הביניים. הספר מדבר בין היתר בפרק אחד על הנצרות, ובפרק אחר על הקמת האוניברסיטאות. שאילתא כגון Christ university, יכולה להחזיר את הספר הזה מכוון ששני המושגים נמצאים

בו. אבל אם נתייחס לכל פרק כמסמך בנפרד, אז לא נקבל את הספר כתשובה לשאילתא). מצד שני, חלוקה קטנה מידי עלולה לגרום לאיבוד מידע.

Tokens

אחרי שהגדרנו מהו מסמך. עברנו על המסמכים שלנו, והוצאנו מהם את המילים, אנו יכולים להתחיל בתהליך ה-Tokenization.

Token הוא רצף של תווים במסמך שלנו (שיכול לכלול רווחים וסימני פיסוק). אם רצף תווים מסוים חוזר פעמיים במסמך, אנו נקבל את אותו ה-token פעמיים. במילון שלנו אנו לא מוצאים את ה-tokens שהוצאנו מהמסמכים, אלא את ה-terms (ביטויים / מושגים). Term הוא token אשר עבר עיבוד מסוים, ולרוב קיבל צורה חדשה כפי שנראה בהמשך.

כביכול, עבודתו של ה-tokenizer היא פשוטה, חלוקה של המסמך למילים, אבל לא תמיד זה כך. להלן מס' נושאים שיש להתייחס עליהם בעת הטוקניזצייה. נתון לנו הביטוי "Finland's capital". כיצד נתייחס למילה "Finland's" ? האם כשתי מילים, Finland ו-s, או Finlands או Finland's ? כלומר, איך נתייחס במקרה הזה לגרש שמופיע במילה. באופן דומה, איך נתייחס לגרש בטוקן aren't, הרי כאן המשמעות היא שונה, לקבל צורה מקוצרת ל-are not.

דוגמה נוספת, הביטוי "Hewlett-Packard". במקרה הזה, בין שתי המילים מפריד מקף, ואנו צריכים לקבוע איך נתייחס למקף הזה. האם נפרק אותו ל-Hewlett ו-Packard כשתי מילים נפרדות ? במקרה הזה, "Hewlett-Packard" הינו שם של חברה, המורכבת משמות שני המייסדים שלה. אם נפרק את הביטוי לשתי מילים, אנו עלולים להחזיר מסמכים שבהם שתי המילים מופיעות בנפרד, כשני שמות שונים, ולא כשם החברה (למשל, אם יש מסמך שמתאר את הביוגרפיה של Hewlett, כנראה שכתוב שם שהוא עבד או הקים חברה ביחד עם Packard). באופן דומה, מה יקרה עם המושגים lower-case או co-education. נניח ובשאילתא המשתמש מקליד "Packard Hewlett" (ללא הרווח), איך נדע למה התכוון ?

San Francisco, במקרה זה אין מקף בין שתי המילים. האם אנו צריכים להתייחס אליהן כטוקן אחד או שניים, ואם כשני טוקנים, איך נדע זאת. במקרה הזה, האות הראשונה בשני המילים היא אות גדולה. אולי אפשר להשתמש במידע הזה בכדי להחליט אם אנו רוצים לחלק להתייחס לביטוי כטוקן אחד או שניים (אות גדולה באמצע משפט יכולה לרמז שמדובר בשם).

מה עושים עם תאריכים. תאריכים אפשר לכתוב במספר אופנים, חלקם תלוי במדינה בה כתבו את המסמך. למשל, "3/20/91" או "Mar. 20, 1991" או "20/3/91". כאשר אנו נתקלים במקרים כאלו, אנו צריכים לזהות שמדובר בתאריכים. כמובן שהאפשרות לפורמטים שונים מגדיל את הסיבוכיות במקרה הזה. כמובן, שתאריכים לא מפצלים (למשל בהתאם ל-"/"). כמו כן, אנו רוצים לשמור את כל התארים בפורמט זהה, בכדי שהשאילתא שלנו לא תהייה מוגבלת לפורמט כלשהו.

במקרים רבים מספרים כוללים בתוכם רווחים, מקפים, ואף תווים אחרים. למשל, 55 B.C., B-52, My PGP key is 324a3df234cb23e או מס' הטלפון 234-2333 (800). במקרים מסוימים, כאשר המספר מייצג מטה-דאטה (למשל תאריך יצירה של הקובץ), אנו נאנדקס אותו באופן נפרד (ואז נוכל לבצע שאילתות חכמות יותר, למשל, הבא לי את כל המסמכים שנוצרו בטווח תאריכים מסויימים).

עד כה דיברנו על בעיות שאנו יכולים להיתקל בהם בשפה האנגלית. אבל גם בשפות אחרות אנו יכולים להיתקל בבעיות. למשל, בצרפתית, "ה" הידיעה היא המילה Le. כאשר היא מופיעה לפני אות "צליל", נוהגים לקצר ל-"'L". לדוגמא, L'ensemble. מה עושים במקרה זה, האם זה טוקן אחד או שניים ? (L ? L' ? Le ?).

בסינית ויפנית לא קיימים רווחים בין המילים ואף לא בין משפטים. לדוגמא,莎拉波娃现在居住在美国东南部

的佛罗里达. החוסר ברווחים מקשה על חלוקה לטוקנים. מעבר לכך, ביפנית קיימים סטים שונים של אותיות, שניתן לשלב יחדיו באותו המשפר. לדוגמא, フォーチュン500社は情報不足のため時間あた$500K(約6,000万円). כמו כן, יש ביפנית שימוש בפורמטים שונים, למשל מספרים, תאריכים, יחידות מידה (kg) וכו', שגם הם מקשים על החלוקה לטוקנים.

עברית וערבית נכתבים מימין לשמאל, אולם, הם יכולים להכיל חלקים (למשל מספרים) הנכתבים משמאל לימין. קבצים בפורמטים מסוימים פותר בעיות מסוג זה. למשל, בקובץ המקודד ב-UTF8, כל הטקסט נשמר בכיוון אחד (שמירה פשוטה), ועל התוכנה המציגה את הקובץ מוטלת המשימה של הצגה נכונה של הטקסט.

Terms

ישנן מילים רבות בעלות משמעות סמנטית מועטה (the, a, and, to, be) שמופיעות כמעט בכל מסמך, שניתן להסיר אותן מהמילון. אם ניקח את המילון שלנו, ונסיר ממנו את 30 ה-terms הנפוצים ביותר (לפי ה-collection frequency, שסופר את כל המופעים בכל המסמכים יחדיו, ולא document frequency, שסופר רק את מספר המסמכים שבהם מופיע כל term), נוכל להקטין את המילון בכ-30%. אולם הנטייה כיום היא לא לעשות זאת, וזאת ממס' סיבות:

- קיימים אלגוריתמים לכיווץ נתונים מאפשרים לשמור את המילון, הכולל את אותם מילים, בקבצים קטנים יחסית.
- שיטות אופטימיזציה מאפשרות לבצע שאילתות אחזור הכוללות את המילים הנ"ל ללא תוספת משמעותיות בעלויות הביצוע.
- לעיתים, אנו זקוקים למילים הנ"ל, מכוון שהם מהווים חלק ממושגים, "King of Denmark", שמות של שירים, "To be or not to be", "Let it be", וכן הם מציינים קשרים למיניהם, "flights to London" .

בשלב הבא אנו צריכים ל"נרמל" את המילים הן בטקסט המאונדקס והן בשאילתות בכדי להביא אותם לאותה הצורה. לדוגמא, USA ו-U.S.A. term הוא צורה "מנורמלת" של טוקן כפי שהיא מופיע במילון של מערכת האחזור שלנו. מילים המופיעות בצורה שונה במסמכים, אך ה-term (המנורמל) שלהם זהה, יהיו באותו posting list. צורה מנורמלת מושגת לרוב בשני אופנים: (1) הורדה של סימני פיסוק, U.S.A ← USA. (2) הורדה של מקפים בין מילים, anti-discriminatory ← antidiscriminatory.

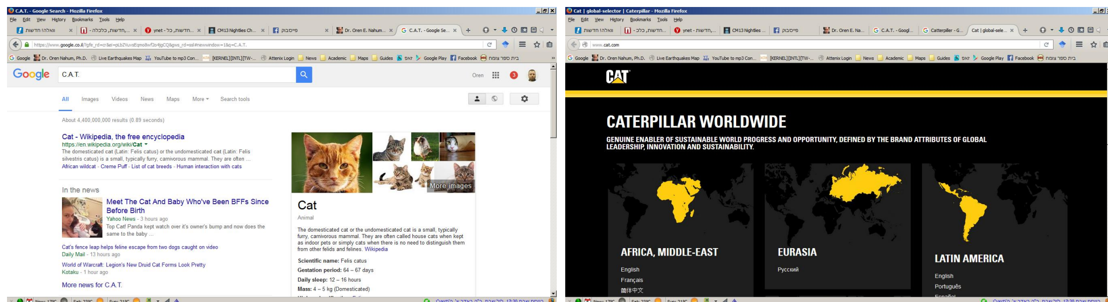
תהליך הנרמול תלוי בשפה שלנו, ולכן בשפות שונות, יתכן ותהליך הנרמול יהיה מעט שונה. בשפות כמו צרפתית אנו נסיר סימנים הקשורים למבטא, résumé ← resume (שימו לב שבאנגלית קיימת המילה resume, שמשמעות שונה, להמשיך, בעוד שבצרפתית משמעות המילה היא קורות חיים). בנרמול מושגים, הכלל הכי חשוב הוא איך המשתמשים נוהגים לכתוב את אותם המושגים, ובהתאם לכך נקבע את האופן שבו הם יופיע במילונים שלנו. לדוגמא, גם בשפות שבהן, הסטנדרט כולל סימנים הקשורים למבטא, המשתמשים נוהגים לכתוב ללא הסימנים הללו.

תהליך הנרמול אינו מוגבל רק למילים. למשל, יש צורך בנרמול של תאריכים, 7/30 vs. 7月30日. במקרה זה אנו רוצים ששני המסמכים יאוחזרו, מכוון שמדובר באותו התאריך. הערה: איך אנו יודעים במדובר בתאריך ולא ביטוי מתמטי ? ובכן, זה לא פשוט. אנו יכולים להתייחס למידע אחר במסמך בכדי לקבוע במה מדובר. למשל, במסמך שהוא מתמטי באופיו, כנראה שמדובר בביטוי מתמטי.

טוקניזצייה ונרמול תלויים בשפה שבה כתוב המסמך, ולכן לעיתים התהליך מקושר לתהליך של זיהוי שפה. יש לשים לב לכך שהן תהליך הטוקניזצייה, הנרמול והשאילתות יעשו בהתאם (לפי אותם מאפיינים).

בנוסף, בשפות כמו אנגלית, נהוג להפוך את כל האותיות לאותיות קטנות. במקרים מסוימים לא נעשה זאת, למשל אות גדולה באמצע משפט (שיכולה לציין שם). ברוב המקרים נהפוך את כל האותיות לאותיות קטנות מכוון שהמשתמשים נוהגים לכתובת באותיות קטנות מבלי להתייחס לאופן הנכון שבו יש לרשום את המילים.

תהליך הנורמליזציה יכול לגרום לאובדן מידע. לדוגמא, C.A.T, אשר מתייחס לחברה, הופך ל-cat. במקרה זה, גוגל, לדוגמא, לא יודע למה התכוונו ומחזיר (במקום הראשון) קישור לוויקיפדייה בנושא חתולים.



במקרים מסוימים, במקום לנרמל מושגים נעשה פעולה הפוכה (query expansion). לדוגמא: הקלט: window, החיפוש יתבצע עבור: window, windows. הקלט: windows, החיפוש יתבצע עבור: Windows, windows, window. הקלט: Windows, החיפוש יתבצע עבור: Windows. באופן כללי, אנו מאפשרים חיפוש טוב יותר, עם יותר אפשרויות. מצד שני, אנו מבצעים יותר חיפושים (זמן חיפוש) ומחזירים יותר תוצאות.

כיצד אנו יכולים לטפל במילים נרדפות ? נניח שהשאילתא היא car, יתכן שאנו רוצים לקבל מסמכים המכילים גם automobile. באופן דומה, נניח שהשאילתא היא color, אנו רוצים לקבל גם מסמכים בהן כתוב colour.

באופן ידני אנו יכולים ליצור רשימה של מילים נרדפות. במקרים כאלו יש לנו שתי אפשרויות. הראשונה, אנו מבנה שאילתא מקבילה, שבה נחפש עבור כל מילה את כל המילים הנרדפות שלה, למשל השאילתא color תהפוך ל- color or colour, ועבור השאילתא car נקבל את השאילתא car or automobile. אפשרות שניה, לאנדקס מילים כאלו מספר פעמים, בהתאם למילים הנרדפות שלהם.

שגיאות כתיב - אחת הדרכים להתמודד עם שגיאות כתיב היא ע"י שימוש בסאונדקס. סאונדקס הוא תחליף פונטי למילה, כך שלשתי מילים הנשמעות אותו הדבר יש את אותו הסאונדקס. אלגוריתם סאונדקס המופעל על string מסוים, בודק אותו, מזהה את העיצורים השונים שיש בו, ובונה string חדש שבו יש ייצוג פונטי של ה-string המקורי. עבור שני string-ים שנשמעים אותו הדבר, אלגוריתם הסאונדקס יצור string פונטי זהה. השימוש בסאונדקס רלוונטי בעיקר עבור שמות.

Lemmatization & Stemming

Lemmatization הינו התהליך של העברת מילים לצורת הבסיס שלהם. לדוגמא:

- am, are, is → be.
- car, cars, car's, cars' → car.
- the boy's cars are different colors → the boy car be different color.

בכדי לבצע lemmatization אנו צריכים מנגנון שמכיר את השפה שלנו.

Stemming גם הוא התהליך של העברת terms (מושגים/תארים) לצורת הבסיס. הפעולה נעשית ע"י הסרה של תחליות או סיומות מה-terms שלנו. לדוגמא:

- automate(s), automatic, automation → automat

פעולת ה-stemming אינה, בהכרח, מייצרת לנו מילים "חוקיות".

האלגוריתם הנפוץ ביותר ל-Stemming באנגלית הוא האלגוריתם של פורטר. תוצאות האלגוריתם לא נופלות באיכותן מהתוצאות המתקבלות מהאלגוריתמים האחרים (נבדק ע"י השוואת תוצאות). האלגוריתם מורכב מחמישה שלבים לקיצור המילים והעברתם לצורכת הבסיס, המבוצעים אחד אחר השני. כל שלב כולל מספר כללים לבחירת הפעולה של אותו שלב, לדוגמא, בחירת הפעולה שפועלת על הסיומת הארוכה ביותר.

- sses → ss
- ies → I
- ational → ate
- tional → tion

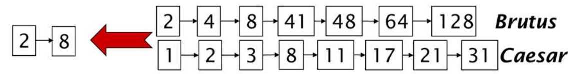
כללים אחרים משתמשים במושג של "משקל" או "אורך" (ביחס למס' ההברות), בכדי לקבוע אם ניתן להפעיל את הכלל על המילה או לא.

- (m>1) EMENT → _ (מילה שמסתיימת ב-ement ומה שמופיע לפניה הוא יותר מתו אחד)
  - replacement → replac
  - cement → cement

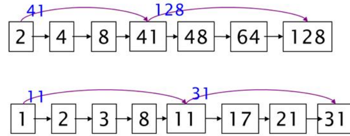
כל הכללים שדיברנו עליהם תלויים בשפה שאיתה אנו עובדים. Stemming לא תמיד עוזר לנו, לרוב זה תלוי גם בשפה שלנו.

Inverted Index & Skip pointers

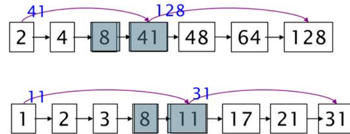
נניח שנתונה השאילתא Brutus and Caesar. בכדי לענות עליה, אנו עוברים על שני posting lists בו זמנית. זמן הריצה הוא לינארי, ותלוי במספר המסמכים בכל רשימה. נניח שמספר המסמכים ברשימה הראשונה הוא n, וברשימה השניה m, אז זמן הריצה הוא O(n+m).



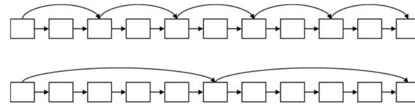
האם ניתן לעשות זאת מהר יותר ? כן. ניתן לשפר את זמן הריצה ע"י הוספת מצביעים ה"קופצים" (skip pointers) לנקודות שונות ברשימה שלנו. היכן עלינו למקם את המצביעים הללו ? לאן הם צריכים להצביע ? ואיך אנו משתמשים בהם ?



נניח ועברנו על הרשימות שלנו והגענו ל-8 בשתיהן. אם נמשיך לעבור על שתי הרשימות הרי שנקבל 41 עבור הרשימה הראשונה ו-11 עבור הרשימה השנייה. ל-11, הערך הנמוך, קיים מצביע "קופץ". אם נשתמש בו, נגיע ל-31, שעדיין נמוך מ-41. כלומר אנו יכולים לדלג ישירות ל-31, מבלי לפגוע בתהליך האיחוד.



ככל שיש יותר מצביעים, המצביעים "קופצים" למרחקים קצרים יותר, והסיכוי ל"קפיצות" גדל. פחות מצביעים, המצביעים "קופצים" למרחקים ארוכים יותר, והסיכוי ל"קפיצות" קטן.



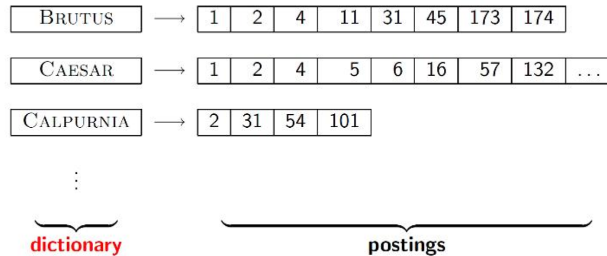
בכדי לקבוע כמה והיכן מצביעים "קופצים" לשים, נשתמש בהיוריסטיקה פשוטה: עבור רשימה בגודל L, נשתמש ב-$\sqrt{L}$, מצביעים "קופצים", הממוקמים במרחקים שווים. היוריסטיקה זו מתעלמת מהתפלגות המושגים בשאילתא. היוריסטיקה זו קלה למימוש אם אנו לא משנים את הרשימות שלנו. אם הרשימות משתנות בתדירות גבוהה (עדכונים), המימוש הוא בעייתי.

oren.e.nahum@gmail.com

אחזור מידע

Dictionaries and tolerant retrieval

האינדקס (Inverted Index) בנוי ממילון מכיל רשימת ביטויים, מספר ההופעות של אותם הביטויים וכן מצביע ל-postings list וכמובן את ה-postings lists עצמם. מהו מבנה הנתונים בו אנו משתמשים עבור האינדקס ? האם מבין האפשרויות השונות לייצוג האינדקס יש מבנה נתונים שהוא יעיל יותר ?



אנו נתמקד כרגע במבנה של המילון. את המילון אפשר לשמור כמערך של-struct ים בעל המבנה הבא:

term	document frequency	pointer to postings list
a	656,265	⟶
aachen	65	⟶
. . .	. . .	. . .
zulu	221	⟶

כל רשומה בנויה מה-term, מס' המופעים שלו ומצביע ל-posting list המתאים. הרשומות עצמן ממוינות בהתאם ל-terms השונים. אם נניח שגודל ה-term הוא 20 בתים, מספר המופעים של ה-term (document frequency) הוא מסוג int (כלומר 4 בתים) וגם המצביע ל-posting list הוא מסוג (כלומר עוד 4 בתים), הרי שסה"כ כל רשומה בנויה מ-28 בתים. אנו יכולים למצוא כל term ע"י חיפוש בינארי. צורת המימוש הזאת היא הפשוטה ביותר. אנו מעוניינים במבנה נתונים שיהיה יעיל יותר מבחינת צריכת הזיכרון, ושיאפשר לנו לבצע חיפוש יעיל (כלומר מהיר) במילון.

לצורך כך עומדות לרשותנו שתי אפשרויות עיקריות: (1) טבלאות Hash, ו-(2) עצי חיפוש.

טבלאות Hash

הערה: ההנחה היא שיש לכם ידע מוקדם בנושא ה-Hash וטבלאות Hash.

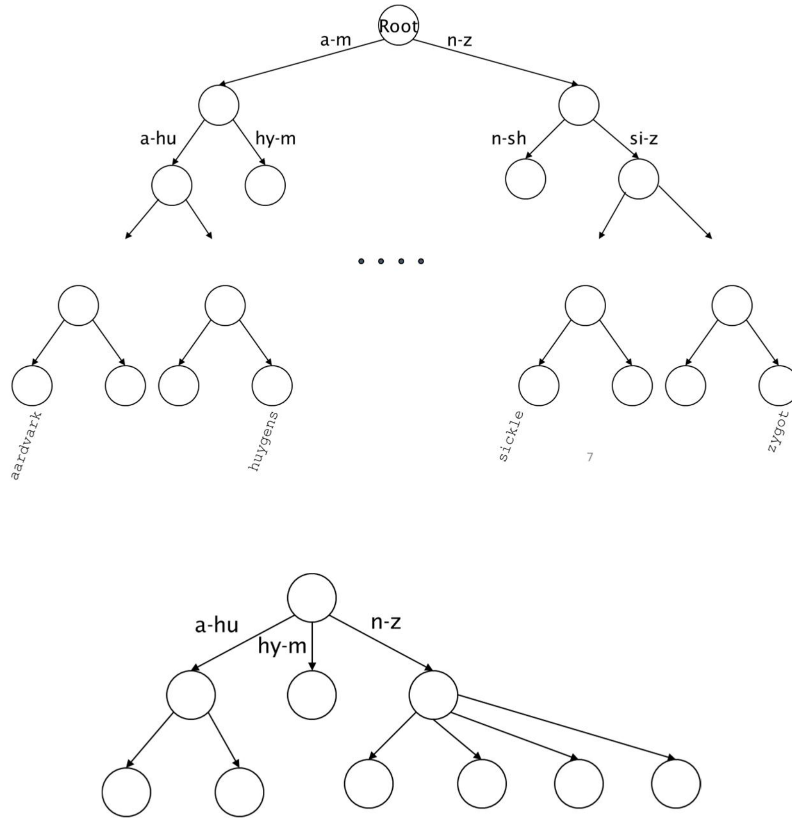
כל term במילון, ע"י שימוש בפונקציית ה-Hash מתורגם למספר (Integer). באופן זה אנו יכולים בצורה "מידית" לשמור כל רשומה (בהתאם ל-term) במקום ייחודי וידוע במערך של רשומות.

לשימוש בטבלאות Hash יתרון ברור. חיפוש של ביטוי בעזרת טבלאות ופונקציות Hash הינה פעולה בסדר גודל $\mathcal{O}(1)$ (או קרוב $\mathcal{O}(n)$, למשל בשימוש בדליים בגודל n).

אבל, ישנם גם מספר חסרונות לשימוש בטבלאות Hash. (1) יתכן ומס' terms ימופו לאותו הערך. במקרה כזה, יש לנו מספר שיטות לטפל בהתנגשויות הללו, אבל אז זמן החיפוש גדל במעט. (2) יתכן וביטויים דומים ימופו לערכים שונים. למשל judgment ו-judgement (כתיב אמריקאי/בריטי), לא בהכרח ימופו לאותו הערך ולכן קשה לקבל את שני המופעים (הם לא יופיעו אחד ליד השני בטבלה). (3) אי אפשר לבצע שאילתות כגון *judg. ו-(4) אם מספר הביטויים ממשיך לגדול פונקציית ה-Hash תתחיל ליצור הרבה התנגשויות, במקרים כאלו יש צורך להחליף את פונקציית ה-Hash בכזו שיוצרת פחות התנגשויות, ובהתאם יש צורך (קבוע) לבצע Hashing מחדש לכל הביטויים בטבלה.

עצים

הגדרה כללית: מבנה נתונים בצורת עץ ששומר מידע ממוין ומאפשר חיפוש, גישה סדרתית, הוספת אברים ומחיקתם בסיבוכיות לוגריתמית. עץ בינארי הוא עץ, שבו לכל צומת יכולים להיות עד שני בנים. עץ B הוא הכללה של עץ חיפוש בינארי בכך שלכל צומת יכולים להיות יותר מ-2 בנים ובנוסף כל העלים באותו עומק. להבדיל מעצי חיפוש מאוזנים, עץ B מיועד לעבודה יעילה במערכות שקוראות וכותבות בלוקים גדולים של מידע.



עצי חיפוש הם עצים מאוזנים שמאפשרים חיפוש מהיר של איברים. קיימים מס' סוגים של עצים. הפשוטים הם עצים בינאריים. "מתוחכמים יותר", וגם שימושיים יותר הם עצי B (קיימים סוגים שונים של עצי B). היתרון הברור בשימוש בעצים הוא שעצים פותרים את בעיית התחיליות.

אבל, בדומה לטבלאות Hash, גם לעצים יש חסרונות. כששני החסרונות העיקריים הם (1) חיפוש איטי יותר, $O(\log m)$, שתלוי בעומק העץ (ולכן, בכדי שיהיה יעיל דורש מאתנו עצים מאוזנים). ו-(2) תהליך האיזון מחדש, שנדרש כל פעם שמוסיפים term לעץ, הוא תהליך איטי. כאשר, בעצי B תהליך האיזון מחדש מתבצע באופן "אוטומטי" עקב המבנה המיוחד של העץ ואופי ביצוע פעולות ההוספה והמחיקה.

Wild-card queries

נניח שאנו רוצים לבצע את השאילתא הבאה: *mon, כלומר למצוא את כל המסמכים בהם ישנם terms המתחילים ב-mon.

אם אנו משתמשים בעצים בינאריים או עצי B, הפתרון הוא פשוט. נביא את כל ה-terms הנמצאים בטווח mon ≤ w < moo, ועבור כל term נביא את רשימת המסמכים שבו הוא נמצא מה-position list שלו.

נניח שעכשיו אנו רוצים את השאילתא הבאה: mon*, כלומר למצוא את כל המסמכים בהם ישנם terms המסתיימים ב-mon. אנו לא יכולים לעשות זאת בעזרת העץ שיש לנו כרגע, מכוון שהחיפוש בו מבוסס על התחלות המילים.

אולם ניתן לפתור בעיה זו ע"י בנייה של עץ בינארי או עץ B נוסף, שבו terms כתובים מהסוף להתחלה. אם נשתמש בעץ הזה, השאילתא עבורו תשנה מ-*mon ל-*nom (מכוון שכל term בו כתוב מהסוף להתחלה).

שאילתא כגון, הבא את כל המושגים הנמצאים בטווח nom ≤ w < non, דורשת עבודה עם שני העצים (הרגיל ו"ההפוך"), כשכל term שמתקבל, ונמצא בתוצאות של שני העצים, אנו נביא את רשימת המסמכים מה-position list שלו.

נניח ואנו רוצים למצוא את כל המסמכים בהם יש ביטויים המתחילים ב-co ומסתיימים ב-tion (co*tion). תחילה, נחפש את כל הביטויים המתחילים ב-co (*co) בעץ B, ועבור כל ביטוי ניקח את רשימת המסמכים שלו מה- posting index שלו. אח"כ, נחפש את כל הביטויים המסתיימים ב-tion (*tion) בעץ B "הפוך", ועבור כל ביטוי ניקח את רשימת המסמכים שלו מה-posting index שלו. לבסוף, נבצע פעולת AND בין התוצאות (כלומר אלו מסמכים מופעים גם בחלק הראשון וגם בחלק השני של הפתרון). הפתרון הזה הוא יקר. אנו מעוניינים למצוא דרך מהירה יותר לעשות פעולה זו.

אינדקס Permuterm

נניח שיש לנו את ה-hello term. בכדי לבנות את ה-Permuterm אינדקס של ה-term הזה, נוסיף בסוף ה-term את הסימן המיוחד $ (שמציין את סוף הביטוי), כלומר נקבל את ה-term החדש hello$. מה-term החדש אנו נבנה terms חדשים, כשכל term מבוסס על ה-term הקודם, והוא בנוי כך שלוקחים את התו הראשון של ה-term הקודם, ומעבירים אותו לסוף ה-term. כלומר, עבור ה-hello term, אנו מקבלים את ה-permuterms הבאים: hello$, ello$h, llohe, lohel, o$hell, $hello.

כל permuterm כזה נחשב כ-term עבור תהליך יצירת ה-inverted index (כשכולם למעשה מתייחסים לאותו term מקורי, ולכן מצביעים לאותו מסמך).

כאמור, המטרה שלנו היא לאפשר חיפוש עם wild-cards באופן מהיר. אנו עושים זאת באופן הבא:

- X, יש לבצע חיפוש עבור X$
- X*, יש לבצע חיפוש עבור X$*
- *X, יש לבצע חיפוש עבור $X*
- *X*, יש לבצע חיפוש עבור X*
- X*Y, יש לבצע חיפוש עבור Y$X*

בכדי לבצע את השאילתא hel*o, אנו רואים שהשאילתא היא למעשה בצורה X*Y, ולכן X=hel, Y=o, ובהתאם נחפש במילון את ה-term (או ה-permuterm) o$hel*.

שימו לב ששימוש ב-permuterm אינו דורש מאתנו להחזיק עץ "הפוך".

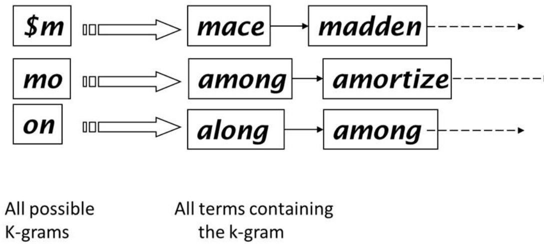
אינדקס Bigram (k-gram)

שיטה נוספת שמאפשרת לנו לבצע שאילתות עם wild-cards היא k-gram. k-gram הינו צרוף של k תווים במסמך שלנו, כאשר גם כאן אנו משתמשים בתו $ בכדי לציין הפרדה בין terms שונים במסמל. בכדי לעבוד עם k-gram אנו צריכים, תחילה, למנות את כל ה-k-gram-ים (צרופים של k אותיות) הקיימים בכל terms שלנו. לדוגמא, עבור הטקסט "April is the cruelest month", נקבל את ה-k (k=2)-gram-ים (bigram-ים) הבאים:

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

במקרה זה, אנו יוצרים inverter index מיוחד שהמילון שלו מורכב מ-k-grams, ואילו ה-posting list של כל ביטוי במילון, הוא רשימת terms המכילים את ה-k-gram הנ"ל.

הדוגמא הבאה מתארת אינדקס k-gram המתקבל עבור k=2 (bigram).



נניח ואנו רוצים לבצע את השאילתא mon*. את השאילתא הזאת אפשר לתרגם ל-$m AND mo AND on. השאילתא הנ"ל תחזיר לנו את התוצאות העונות לשאלה mon*, אבל היא גם עלולה להחזיר תוצאות שאינן מתאימות לשאילתא המקורית שלנו, כמו moon. לכן, כל תשובה שמתקבלת, יש לבדוק התאמה מול השאילתא המקורית. את התוצאות שתואמות את השאילתא המקורית יש לחפש ב-inverted index של המושגים-מסמכים.

שיטה זו מהירה יותר וחסכונית יותר בזיכרון בהשוואה ל-permuterm.

בשאילתות כגון pyth* AND prog*, אנו נתייחס לכל שאילתת wild-card בנפרד. ועבור התוצאות שנקבל אנו נבצע שאילתת AND. ריצה של שאילתות כאלו, עלולה לקחת זמן רב.

<u>Spelling correction</u>

לתיקון שגיאות כתיבת שני שימושים עיקריים. (1) תיקון שגיאות כתיב במסמכים אותם אנו מאנדקסים. (2) תיקון שגיאות כתיב בשאילתות שהמשתמש מזין (על מנת לקבל תשובות נכונות).

תיקון שגיאות נעשה בשני אופנים עיקריים. (1) תיקון שגיאות של מילים בודדות – תיקון שגיאות כתיב של מילה בודדת. במקרים כאלו, לא נזהה שגיאות אם האיות של המילה תקין (from במקום form). (2) תיקון שגיאות בהתאם להקשר – כל מילה נבדקת ביחס למילים שנמצאות לידה. לדוגמא, I flew <u>form</u> Heathrow to Narita.

תיקון שגיאות נדרש במיוחד עבור מסמכים שנסרקו (OCR). במקרים כאלו, האלגוריתמים שמזהים את האותיות, עלולים לטעות ולזהות אותיות (או צירופי אותיות) מסוימות כאותיות אחרות. לדוגמא, האותיות m ו-rn דומות אחת לשנייה. במידה והסריקה לא באיכות טובה, או שהאלגוריתם אינו מדויק/רגיש מספיק, יהיו מצבים שהאות m תזוהה כ-rn ולהפך. אלגוריתמים לתיקון שגיאות בנויים לזהות שגיאות מסוג זה. הם כוללים "ידע מיוחד" הקשור לבעיות סריקה, כגון רשימה שלא אותיות הנראות דומה, כמו m ו-rn או O ו-D (לעומת I ו-O, שהסיכוי לבלבול ביניהם נמוך).

גם במסמכים מסוגים אחרים, למשל מוקלדים, יש שגיאות כתיב. למרות שהמטרה שלנו, שבמילון יהיו כמה שפחות שגיאות כתיב, בהרבה מקרים אנו לא "מתקנים" את המסמכים עצמם, אלה דואגים שהמיפוי בין השאילתא למסמך יהיה תקין.

נניח שאנו מחפשים את הביטוי Alanis Morisett. אנו יכולים (1) להחזיר את המסמכים המכילים את הכתיב הנכון של הביטוי, או (2) להחזיר מספר אפשרויות לשאילתות חדשות, המכילות תיקוני שגיאות כתיב (Did you mean … ?)

כאשר אנו מבצעים תיקון של מילים בודדות, הנחת יסוד היא שיש לקסיקון שלפיו נקבע האיות הנכון של כל מילה. במקרה הזה, יש לנו שתי אפשרויות, (1) להשתמש בלקסיקון סטנדרטי, כדוגמת Webster's English Dictionary או (2) לבנות לקסיקון הבנוי במילים הנמצאות במסמכים An "industry-specific" lexicon – hand-maintained. המאונדקסים שלנו (הקורפוס). הלקסיקון הזה יכלול את כל המילים ב-web ואת כל השמות (כולל מילים עם שגיאות כתיב).

בדיקת האיות נעשית באופן הבא. כאמור, נתון לנו לקסיקון וסידרה של תווים, Q (המילה אותה אנו רוצים לבדוק). המטרה שלנו היא למצוא את המילה בלקסיקון הקרובה ביותר ל-Q. אם קיימת מילה בלקסיקון, ש"המרחק" בינה לבין סידרת התווים שלנו, Q, הוא 0, הרי שהמילה שלנו מאויתת נכון. אחרת, ישנה שגיאת כתיב, וכנראה שהמילה הקרובה ביותר במילון היא הכתיב הנכון.

ובכן, כיצד אנו מגדירים את המושג "קרובה ביותר" ? לשם כך נדבר על מספר אלטרנטיבות: (1) Edit distance (Levenshtein distance), (2) Weighted edit distance ו-(3) n-gram overlap.

Edit distance

נתונות שתי מחרוזות, S_1 ו-S_2, edit distance מוגדר כמספר הפעולות (דרמת התווים – הוספה, מחיקה ושינוי תווים) המינימאלי הנדרש על מנת להגיע מהמחרוזת הראשונה לשנייה.

לדוגמא:

- מ-dof ל-dog, זה 1
- מ-cat ל-act, זה 2
- מ-cat ל-dog, זה 3

החישוב של edit distance לרוב ממומש ע"י תכנות דינאמי (http://www.merriampark.com/ld.htm).

Weighted edit distance

בדומה ל-Edit distance, weighted edit distance מוגדר כמספר הפעולות המינימאלי הנדרש על מנת להגיע מהמחרוזת הראשונה לשנייה, אולם לכל פעולה יש משקל התלוי בתווים המשתתפים באותה פעולה. הרעיון הוא, שבכל סוג של מסמך יש טעויות שהן נפוצות יותר ונפוצות פחות. במסמכים סרוקים, הסיכוי להחליף בין m ל-rm גבוה יותר (משקל נמוך יותר) מאשר הסיכוי להחליף בין m ל-q (משקל גבוה יותר). באופן דומה, במסמכים מודפסים יש סיכוי גבוה יותר לשגיאות הנובעות מטעויות הקלדה (החלפת סדר אותיות, או הקלדה על אות "קרובה" במקלדת). בכדי לחשב את ה-weighted edit distance אנו צריכים שתהייה לנו מטריצת משקלים, שמציינת את המשקל של החלפת אות אחת באות שנייה בהתאם להסתברות לבצע טעות כזו.

החישוב של ה-weighted edit distance דורש שינוי (קל) באלגוריתם התכנות הדינאמי שלנו.

השימוש ב-edit distance או weighted edit distance מתבצע באופן הבא. בהינתן שאילתא, תחילה אנו מונים את כל צירופי התווים שיש להם (Weighted) Edit distance נתון (למשל 2). נבצע הצלבה בין התוצאות שקיבלנו לרשימה של מילים "נכונות", בהתאם יש לנו שלוש אפשרויות. (1) נציג למשתמש את המילים שנמצאו (הצעות). (2) נבצע שאילתא המכילה את כל המילים שמצאנו (איטי). (3)נשתמש באפשרות הסבירה ביותר במקום כל המילים (אלו שהשכיחות שלהם היא הגבוהה ביותר). בצורה זו אנו "חוסכים" את הצורך במשתמש (לא תמיד לטובה), ובכך מספקים תשובות מהר יותר.

כאמור, בהתחלה אנו מונים את כל צירופי התווים שיש להם (Weighted) Edit distance נתון. למעשה אנו יוצרים הרבה מאד צרופים לכל term בשאילתא. אפשרות אחרת, במקום ליצור צרופים חדשים, נחשב את ה- edit distances בין כל term בשאילתא לכל term במילון. התהליך הזה הוא מאד איטי, ולכן האם אנו באמת צריכים לחשב את ה- edit distances לכל מושג במילון ?

n-gram overlap

אנו יכולים להקטין את קבוצת ה-terms של המילון המעומדים לבדיקה (כלומר מולם נחשב את ה-edit distance), למשל ע"י שימוש ב-n-grams overlap. שיטה זו מאפשרת בדיקת איות גם עבור השאילתות עצמן.

שיטה זו פועלת באופן הבא. תחילה אנו מונים ה-n-grams במחרוזת השאילתא וכן בלקסיקון. אחר-כך אנו משתמשים באינדקס ה-n-grams (חיפוש עם wild-cards) בכדי להביא מהלקסיקון את כל המושגים התואמים ל--n grams שבמחרוזת השאילתא. אנו קובעים סף המבוסס על מספר ה-n-grams התואמים (אפשרויות נוספות, להשתמש במשקלים המותאמים לשגיאות הקלדה וכו'), שלפיו אנו מחליטים אם הכתיב של term הוא תקין או לא, ובמידה ולא, איזה terms מהמילון יכולים להוות אלטרנטיבות לתיקון הכתיב השגוי.

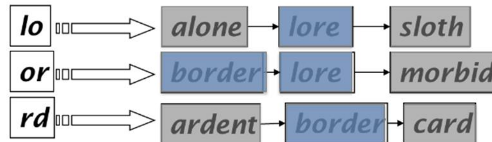
לדוגמא, נניח שהטקסט שלנו הוא November. בהתאם, אנו יכול לחשב את ה-3-grams הבאים: nov, ove, vem, emb, mbe, ber. נניח שהשאילתא שלנו היא December. עבורה ה-3-grams הם: dec, ece, cem, emb, mbe, ber. אנו רואים שיש התאמה של 3 3-grams. המספר הזה אינו עוזר לנו כי הוא יכול להיות שונה בין terms שונים, ואז תהייה לו משמעות שונה. צריכים להפוך את הערך הזה לערך מנורמל.

שיטה אחת לנירמול היא Jaccard coefficient. נניח ש-X ו-Y הן שתי קבוצות, J.C. מוגדר כ:

$$\frac{|X \cap Y|}{|X \cup Y|}$$

באופן זה, אנו מקבלים 1 כאשר X ו-Y (אשר אינם חייבים להיות באותו הגודל) מכילים את אותם הערכים, ו-0 כאשר הם שונים לגמרי. הערך המתקבל הוא תמיד בין 0 (כולל) ל-1 (כולל). באופן כזה אנו יכולים לקבוע סף, למשל, אם J.C גדול מ-0.8, אנו מתייחסים לזה כהתאמה.

נניח שהשאילתא שלנו היא lord, ואנו מעוינים במילים שיש בהם לפחות 2 מ-3 ה-bigrams שבביטוי (lo, or, rd). אנו נשתמש בערך מנורמל (.J.C או אחר) בכדי לקבוע התאמה.



<u>תיקון שגיאות בהתאם להקשר</u>

נתון הטקסט: I flew <u>from</u> Heathrow to Narita. וכן, נתונה השאילתא: Heathrow <u>form</u> flew. אנו רוצים לשאול את המשתמש אם הוא התכוון ל-flew from Heathrow, מכוון שאין מסמכים המכילים את השאילתא. מכוון שאין שגיאות במילים עצמן, ניתן להבין שהמילה form שגוייה רק לפי ההקשר (המילים שלידה).

בכדי לעשות זאת נפעל לפי השלבים הבאים. (1) לכל term בשאילתא נביא את ה-term-ים הקרובים לו (למשל ב-weighted edit distance). (2) נבדוק את התוצאות המתקבלות עבור כל השאילתות, כאשר כל שאילתא היא אחת מהפרמוטציות האפשריות.

אם נניח שבשאילתא יש רק מילה אחת שגוייה , מספר השאילתות הנבדקות יהיה קטן יותר. במקרה זה נבדוק את התוצאות המתקבלות עבור כל השאילתות, כאשר בכל שאילתא אנו מחליפים term אחד ב-term "מתוקן". למשל, (1) flew from Heathrow, (2) fled form Heathrow, (3) flea form Heathrow וכו'. אנו נציע למשתמש את השאילתא שמחזירה את מירב התוצאות.

כאמור, יש לנו מספר אפשרויות ל"האם התכוונת ל... ?", ואנו צריכים החליט אלו מהאפשרויות להציג למשתמש. בדוגמא הקודמת אמרנו שנציג למשתמש את השאילתא מחזירה את מירב התוצאות, אולם לפעמים אנו יכולים לבצע ניתוח של שאילתות עבר ולהחליט על סמך זה. באופן כללי יותר, אנו רוצים לדרג את ההסתברות לקבלת תשובה נכונה (כלומר, $argmax_{corr}P(Corr|Query)$ או בהתאם לחוק Bayes, $argmax_{corr}P(Query|Corr)\times P(Corr)$ – אנו נדבר על נושאים אלו בהרצאה אחרת).

<u>Soundex</u>

סאונדקס היא היוריסטיקה שמטרתה להפוך מילה או שאילתא למקבילה הפונטית שלה. כל token במסמך מועבר למצב "מוקטן" הבנוי מ-4 תווים, ומאונדקס לפיו. את אותה הפעולה אנו עושים לכל term בשאילתא. החיפוש נעשה בהתאם לצורות ה"מוקטנות" כאשר החיפוש הוא לפי סאונדקס.

<u>אלגוריתם סאונדקס כללי</u>

1. Retain the first letter of the word.
2. Change all occurrences of the following letters to '0' (zero):
   'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows:
   - B, F, P, V → 1
   - C, G, J, K, Q, S, X, Z → 2

- D,T → 3
- L → 4
- M, N → 5
- R → 6

4. Remove all pairs of consecutive digits.
5. Remove all zeros from the resulting string.
6. Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter> <digit> <digit> <digit>.

E.g., ***Herman*** becomes H655.

לדוגמא, נסתכל על המילה Herman. בשלב הראשון אנו שומרים את האות הראשונה של המילה, כלומר, Herman. בשלב השני, כל המופעים של האותיות a ו-e מוחלפים ב-0, בהתאם לסעיף 2 של האלגוריתם, ואנו מקבלים H0rm0n. בשלב השלישי אנו מחליפים את האות r ל-6, ואת האותיות m ו-n ל-5, ומקבלים H06505. לאחר מכן אנו צריכים להסיר תווים זהים רציפים, אולם במקרה שלנו אין לנו. בשלב הבא אנו מורידים את ה-0-ים שהופיעו, ומקבלים H655. בשלב האחרון אנו לוקחים רק את 4 האברים השמאליות ביותר. כלומר, התוצאה הסופית היא H655.

סאונדקס הוא אלגוריתם כללי, שקיים כמעט בכל בסיס נתונים (מיקרוסופט, אורקל...). עד כמה הוא יעיל ? לא הרבה בתחום של אחזור מידע. כן ניתן למצוא לו שימוש במקרים מיוחדים (למשל שמות). קיימים אלגוריתמים אחרים בתחום הפונטיקה שהם יותר ישימים בתחום של אחזור מידע (Zobel and Dart, 1996).

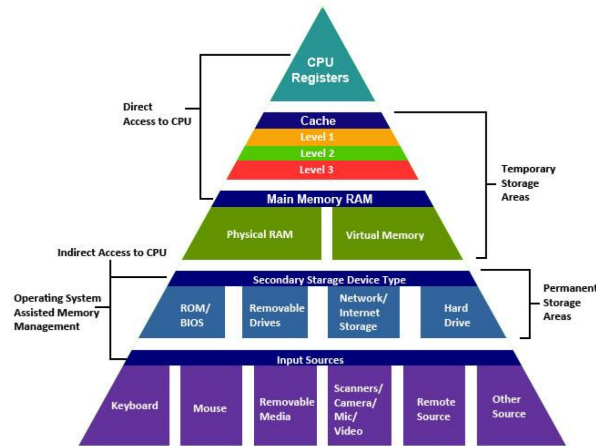
תרגיל: יש להפעיל את האלגוריתם על שתי המילים הבאות: Beijing ו-Peking.

אחזור מידע

Index Construction

הנושא המרכזי של הרצאה זו הוא תהליך בניית האינדקסים (ה-Indexing / אינדוקס) וכיצד הוא נעשה. אנו דיברנו על תהליך בניית האינדקסים בהרצאה הראשונה, אולם ההנחה שלנו הייתה שכמות המסמכים שלנו היא כזו שהן המילון שאנו יוצרים והן ה-posting lists הם קטנים דיים, כך שהם יכולים להישמר הזיכרון המחשב בשלמותם. כזכור, כחלק מפעולת האינדוקס, אנו צריכים למיין את ה-terms השונים, עד כה הנחנו שאין לנו בעיה, ואנו יכולים לבצע פעולה זו בשלמותה בזיכרון המחשב (ה-RAM). בהרצאה זו אנו נהייה יותר מציאותיים. אנו נזנח את ההנחה שלנו לגבי גודלם של המסמכים שלנו (הקורפוס), ונראה באלו אסטרטגיות אנו יכולים להשתמש כאשר הזיכרון הזמין לנו מוגבל ואינו מאפשר את ביצוע הפעולות השונות בזיכרון המחשב (מפאת כמות הנתונים וגודלם שאינם יכולים להישמר בשלמותם בזיכרון).

לפני שנדבר על האלגוריתמים עצמם אנו נתחיל עם סקירה בסיסית של חומרת המחשב, היות ובתחום אחזור נתונים הרבה החלטות הנוגעות לעיצוב המערכת מבוססות על מבנה החומרה שלנו. העיקרון הראשון שאליו אנו צריכים להתייחס הוא שגישה לנתונים הנמצאים בזיכרון מהירה **פי כמה** המאשר לנתונים המאוכסנים על גבי הדיסק הקשיח.



אחד מהגורמים לזמני הגישה הגבוהים לנתונים המאוכסנים על גבי הדיסק הקשיח הוא מה שנקרא Seek Time, שמורכב משלושה גורמים: (1) הזמן שלוקח למקם את הראשים של הדיסק הקשיח במקומם, (2) הזמן שלוקח לסובב את הדיסק ולהביא את הנקודה הנכונה אל מתחת לראשים, ו-(3) זמן הקריאה עצמו של הראשים (אנו נתייחס לשתי הפעולות הראשונות ב-Seek Time). מכוון שב-Seek Time, לא מתבצעת העברת נתונים, עדיף להעביר של גוש אחד גדול של נתונים, שהינה פעולה מהירה יותר, מאשר להעביר של מספר גושים קטנים של נתונים.

נקודה נוספת שדורשת התייחסות היא שפעולות I/O של הדיסק נעשות בגושים, הנקראים בלוקים. כלומר כתיבה או קריאה של נתונים לא נעשית ברמת הבית, אלה בבלוקים (גוש נתונים), המכילים מס' בתים כל אחד. מכוון שברוב המקרים, לאחר שאנו קוראים בית אחד, אנו מעוניינים בקריאה של הבית שנמצא אחריו בקובץ, קריאה של בלוק שלם תהייה יעילה יותר מבחינת זמני התגובה. גודלו של בלוק ממוצע הוא בין 8KB ל-256KB.

בשרתי IR ממוצעים, הזיכרון הוא של מספר GB, לפעמים עשרות. גודלו של הדיסק הקשיח הוא פי כמה (2-3) גדול מהזיכרון הקיים במחשב.

נקודה נוספת אליה נתייחס היא מניעת תקלות בשרתי IR. מניעת תקלות מאד יקרה, ולכן במקום מחשב אחד עמיד ויקר עדיף כמה (עשרות/מאות/אלפים) מחשבים פשוטים וזולים.

הטבלה הבאה מציגה מספר ערכים הקשורים לחומרה אשר ישמשו אותנו בהמשך בדוגמאות השונות שלנו. יש לקחת בחשבון שהערכים הללו הם לא ערכים מדויקים, ובהחלט יתכן שהם השתנו (והתקצרו) במהלך השנים. אולם, הם עדיין מהווים מדד לסדרי הגודל השונים בעבודה בזיכרון המחשבים ובעזרת הדיסק הקשיח.

סימון	משמעות	ערך
s	average seek time	5 ms = 5 x 10^{-3} s
b	transfer time per byte	0.02 μs = 2 x 10^{-8} s
	processor's clock rate	10^{9} s^{-1}
	low-level operation (e.g., compare & swap a word)	0.01 μs = 10^{-8} s
	size of main memory	several GB
	size of disk space	1 TB or more

נקודה נוספת שאליה אנו צריכים להתייחס היא אוסף היצירות של שייקספיר, שאינו גדול מידי בכדי להדגים את הסוגיות בהן עוסקת הרצאה זו. ולכן, לצורך ההדגמה אנו נשתמש באוסף מסמכים אחר, Reuters RCV1 collection, אף שגם הוא לא נחשב גדול, הוא מספיק לצורכי ההרצאה שלנו. אוסף מסמכים זה הוא אוסף של ידיעות חדשותיות שהופצו ע"י Reuters למערכות עיתונים שונות, בין השנים 1995-1996, במהלך 12 חודשים.

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

A Reuters RCV1 document

הטבלה הבאה מתארת מס' נתונים סטטיסטיים לגבי אוסף המסמכים RCV1.

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

מדוע יש הבדל בין מס' הבתים הממוצע של token ומספר הבתים הממוצע של term ? הסיבה לכך היא ש-term הוא ייצוג בודד של מס' tokens זהים. מכוון שיש הרבה tokens קצרים (stop words בעיקר), הרי שהממוצע של אורכי ה-tokens היה הנמוך מהממוצע של אורכי ה-terms. תכונה זו הינה תלויה בשפה, במקרה זה, אנגלית.

<u>מיון</u>

כזכור מההרצאה הראשונה, בתהליך בניית האינדקס אנו עוברים על כל המסמכים, אחד אחרי השני. כל מסמך עובר תהליך של tokenization שלאחריו הוא עובר דרך ה-linguistic module, שבעקבותיו אנו מקבלים רשימה של terms הנמצאים במסמך. בסוף התהליך, הרשימה המלאה של ה-terms (מכל המסמכים ביחד), ממוינת תחילה בסדר לקסיקוגרפי, ואח"כ, כמיון משני, לפי ה-docID. את תהליך המיון הנ"ל כולו ביצענו בזיכרון המחשב (ב-RAM).

אם נניח שכל term, לפני בניית רשימת ה-terms עצמה, תופס 12 בתים, עבור רשימה גדולה מאד של מסמכים אנו מדברים על כמות עצומה של זיכרון. במקרה של RCV1 אנו מדברים על $T = 100,000,000$. מכוון שאת הרשימה המלאה של ה-terms למיון אנו יכולים לקבל רק בסוף תהליך הבנייה, הרי שבמקרה הזה המשמעות היא שאנו צריכים 1.12GB של זיכרון לצורך שמירת ה-terms עצמם. במחשבים של היום, במקרה הזה, עדיין ניתן לבצע את המיון בזיכרון.

למרות גודלו, RCV1 אינו נחשב לאוסף גדול של מסמכים. למשל, ה-New-York Time, מאפשר לחפש במסמכים שנכתבו במשך יותר מ-150 שנה. ולכן, אנו צריכים אלגוריתמים למיון, שעובדים בשלבים, ויכולים לשמור תוצאות ביניים על הדיסק.

האם ניתן להשתמש באותם אלגוריתמים למיון (שעובדים בזיכרון), אולם במקום להשתמש בזיכרון המחשב, נשתמש בדיסק ? נניח שיש לנו 100,000,000 רשומות שיושבות על הדיסק. נניח שלצורך פעולת השוואה אנו צריכים שתי פעולות seek, שהן, כאמור, פעולות איטיות מאד (5ms) בהשוואה לקריאה מהזיכרון. וכן, על מנת למיין N רשומות יש צורך ב-$NlogN$ השוואות, אזי במקרה שלנו אנו מדברים על $2\times100,000,000log100,000,000\times5 = 8000000000\ ms$, שזה 92.6 ימים. ולכן, התשובה היא לא. אנו לא יכולים להשתמש באלגוריתמים הרגילים למיון כאשר המידע יושב על הדיסק הקשיח.

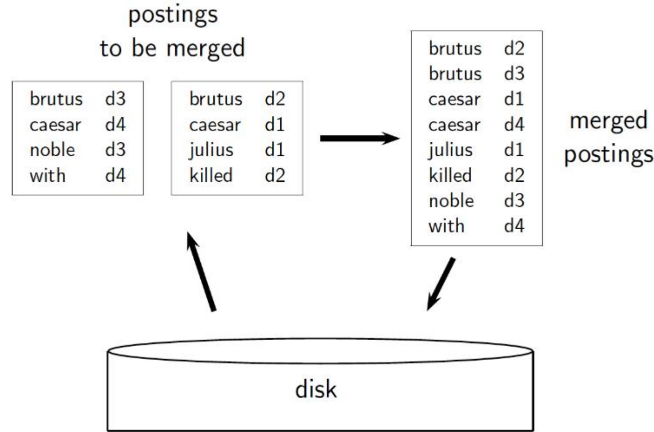
<u>BSBI: Blocked sort-based Indexing</u>

כאמור, יש לנו 100M רשומות בגודל של 12 בתים (term, docID, freq) שהתקבלו מעיבוד מסמכים. אנו צריכים למיין את הרשומות לפי השדה term, ומיון משני לפי השדה docID. מכוון שלא ניתן (מכוון שיש מגבלות זיכרון) לשמור את כל ה-100M רשומות בזיכרון המחשב (לצורך המיון), אנו צריכים למצוא דרך אחרת לבצע את המיון. לצורך כך, נחלק את 100M הרשומות שלנו לבלוקים קטנים יוצר, נניח בלוקים בגודל של ~10M, באופן כזה שניתן לשמור שני בלוקים בזיכרון המחשב.

אופן בניית הבלוקים הוא פשוט. אנו נעבור על קבצים שלנו, ועבור כל term שנקבל, נרשום אותו ואת ה-docID שלו לבלוק הנוכחי. אנו נבצע תהליך זה עד שהבלוק שלנו יתמלא, אז נשמור את הבלוק על הדיסק. נמשיך לעבור על הקבצים שלנו, כאשר אנו מתחילים למלא בלוק חדש. כך נעשה עד שנסיים לעבור על כל הקבצים.

כעט, אנו נמיין כל אחד מהבלוקים שלנו (בנפרד). את זה ניתן לבצע בזיכרון, מכוון שהבלוקים הינם קטנים יחסית. לאחר מכן, נאחד את כל הבלוקים (ששמורים על הדיסק) לרשימה ממוינת אחת.

את תאריך האיחוד, ניתן לעשות באופן זהה לאופן שבו איחדנו, או סרקנו רשומות קודם לכן. אנו נטען לזיכרון את ההתחלה של הבלוק הראשון ואת ההתחלה של הבלוק השני. אנו נתחיל לעבור על שתי הרשימות, ובצע תהליך הדומה ל-merge sort. כאשר נגיע לסוף של המידע שנמצא בזיכרון של אחד מהבלוקים, נטען חלק נוסף לזיכרון.



להלן הפסאודו-קוד של האלגוריתם הנ"ל. בשורה 1, n הוא מס' הבלוק, שמאותחל לערך 0. כל עוד לא סיימנו לעבור על כל המסמכים (שורה 2), אנו מגדילים את ערכו של n ב-1 (שורה 3), ממלאים את הבלוק הנוכחי ברשימה של הזוגות (term,docID) (שורה 4), ממיינים אותו (שורה 5) ושומרים אותו לדיסק (שורה 6). לאחר שעברנו על כל המסמכים, אנו ממזגים את הבלוקים הממוינים, ששמורים על הדיסק, לרשימה אחת גדולה וממוינת (שורה 7).

```
BSBIndexConstruction()
1  n ← 0
2  while (all documents have not been processed)
3  do n ← n + 1
4      block ← ParseNextBlock()
5      BSBI-Invert(block)
6      WriteBlockToDisk(block, f_n)
7  MergeBlocks(f_1, ..., f_n; f_merged)
```

עד עכשיו עסקנו במיון רשימת ה-(term,docID), כאשר השלב הבא הוא מעבר על הרשימה הממוינת הזאת, בניית המילון וה-posting lists. אנו יכולים לעבוד בצורה זו כל עוד שניתן לשמור את המילון בזיכרון. למעשה, בכדי לחסוך בזיכרון במקום להשתמש ב-term אנו יכולים להשתמש ב-termID. במקרה כזה, אנו צריכים את המילון (שגדל באופן דינאמי ככל שמספר ה-terms גדל כתוצאה ממעבר על מסמכים נוספים) בכדי לבצע את המיפוי של הביטויים (term) למס' המזהה שלו (termID). אמנם, ניתן להשתמש במיקומים מבוססים על הביטוי ומזהה המסמך (term,docID) במקום להשתמש ב-termID ו-docID, אולם אז אנו נצטרך לשמור קבצי ביניים גדולים.

SPIMI: Single-pass in-memory indexing

בכדי לפתור את בעיית הזיכרון של המילון, אנו יכולים להשתמש באלגוריתם SPIMI.

האלגוריתם דומה לאלגוריתם הקודם, עם מס' שינויים. (1) לכל בלוק ניצור "מילון" משלו שהמבנה שלו הוא (term, docID). כך אנו **לא צריכים** לשמור מיפוי של ביטוי למס' ביטוי (term-termID) בין הבלוקים. (2) לא נבצע מיון. נשמור את המיקומים של ה-terms ב"מילון", בהתאם לסדר שהם מתקבלים, בזמן המעבר על הקבצים. (3) בהתאם לשתי הפעולות האלו, אנו יכולים ליצור inverted index עבור כל בלוק (למעשה ניתן לבצע את שלבים 1 ו-2 במקביל). (4) בהמשך, ניתן למזג את האינדקסים הללו לאינדקס אחד גדול.

להלן הפסאודו-קוד של האלגוריתם SPIMI-Invert.

```
SPIMI-Invert(token_stream)
 1  output_file = NewFile()
 2  dictionary = NewHash()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6       then postings_list = AddToDictionary(dictionary, term(token))
 7       else postings_list = GetPostingsList(dictionary, term(token))
 8     if full(postings_list)
 9       then postings_list = DoublePostingsList(dictionary, term(token))
10     AddToPostingsList(postings_list, docID(token))
11  sorted_terms ← SortTerms(dictionary)
12  WriteBlockToDisk(sorted_terms, dictionary, output_file)
13  return output_file
```

output_file (שורה ראשונה) הוא הקובץ שבו ישמר כל ה-inverted index. אנו מתחילים עם "מילון" ריק (שורה 2), שבמקרה זה ממומש ע"י hash table. כל עוד יש לנו זיכרון פנוי (שורה 3), אנו מבצעים את השלבים הבאים: (1) נקרא את ה-token הבא מהקובץ (שורה 4). למעשה אנו מקבלים את הצרוף (term, docID). (2) אם ה-term לא נמצא במילון (שורה 5), אז ניצור posting list חדש, שיכיל את ה-docID הנוכחי (שורה 6). (3) אחרת, נקרא את ה-posting list של ה-term הקיים, ונוסיף לסופו את ה-docID הנוכחי (שורה 7). (4)במידה ורלוונטי, שורות 8 ו-9 מטפלות במקרים שבהם ה-posting list הוא מלא (למשל אם השתמשנו במערכים). (5) בשורה 10 אנו מעדכנים במילון את ה-posting list של ה-term הנוכחי (הוספה של חדש או עדכון posting list קיים). אחרי שלב זה, יש לנו inverted index עבור הבלוק הנ"ל. אנו צריכים למיין את ה-inverted index הזה (שורה 11), ואז לכתוב אותו לדיסק (שורה 12).

אנו מבצעים את התהליך הזה לכל המסמכים. במהלך התהליך נוצרים מספר קבצים, אותם אנו צרכים למזג בסוף התהליך.

ניתן ליעל את SPIMI ע"י שימוש בכיווץ. כאשר, (1) אפשר לכווץ את ה-terms, ו-(2) אפשר לכווץ את posting lists. נדבר על כך בהרצאה הבאה.

<u>אינדקסים מבוזרים</u>

בדוגמאות הקודמות פתרנו את בעיית הזיכרון – כיצד ניתן לבנות את ה-inverted index כאשר הזיכרון שלנו מוגבל. עשינו זאת ע"י ניצול הדיסק הקשיח. אולם במקרים מסוימים (למשל, כאשר אנו רוצים לבנות אינדקס של כל אתרי האינטרנט בעולם), גם גודלו של הדיסק הקשיח מהווה מגבלה.

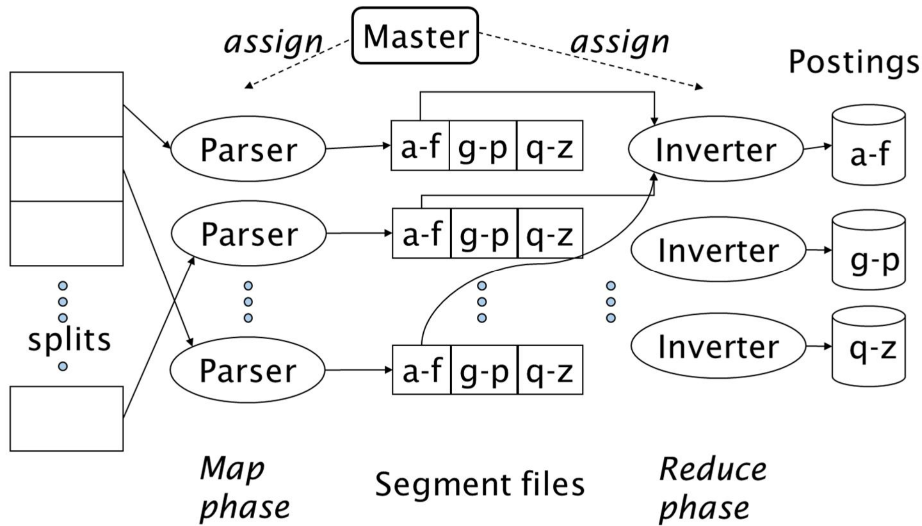
נניח שרוצים לבצע אינדוקס של דפים באינטרנט, מהסיבות שהזכרנו, אנו לא נשתמש במחשב בודד, אלא נשתמש באשכול מחשבים מבוזר. מעבר לכך, מחשבים יכולים באופן בלתי צפוי להתחיל ולעבוד לאט או להתקלקל. אנו נראה כיצד אשכול מחשבים מבוזר יכול לעזור לנו במקרים כאלו.

מרכזי הנתונים של מנועי חיפוש באינטרנט (כמו Google), אתרים שונים בהם מתבצעים תהליכי האינדוקס וחיפוש, מכילים מחשבים רבים. בנוסף, הם מפוזרים במקומות שונים ברחבי העולם. ההערכות (נכון ל-2007) הן שלגוגל יש בערך מיליון שרתים, ובהם, בסה"כ, כ-3 מיליון מעבדים/ליבות, מפוזרים במקומות שונים ברחבי העולם.

נניח שיש לנו אשכול עם 1000 מחשבים. לכל מחשב יש uptime של 99.9% (כלומר, כל מחשב עובד ב-99.9% אחוז מהזמן). מהו ה-uptime של המערכת כולה (כלומר, מהו אחוז הזמן שבו כל המחשבים עובדים ביחד) ? = 99.9^{1000} 36%. זהו אחוז נמוך, אם אנו דורשים שכל המחשבים יעבדו בו זמנית ביחד. למזלנו, אנו לא צריכים שכל המחשבים יעבדו בו זמנית.

נניח שיש לנו אשכול מחשבים שאנו רוצים לבצע את תהליך האינדוקס. לשם כך, נקצה מחשב אחד מאשכול המחשבים, מחשב ה"מאסטר", לניהול תהליך האינדוקס. כמו כן, נפרק את תהליך האינדוקס לקבוצה של משימות, אותן נבצע במקביל. מחשב ה"מאסטר" יקצה את כל אחת מהמשימות למחשב פנוי באשכול המחשבים שלנו. במידה ותתרחש תקלה באחד המחשבים, מחשב ה"מאסטר" יכול להקצות את אותה המשימה (שהוקצתה למחשב התקול) למחשב אחר.

לצורך תהליך האינדוקס אנו נשתמש בשתי קבוצות של משימות מקביליות: (1) Parsers ו-(2) Inverters. נחלק את אוסף המסמכים שלנו לקבוצות (Splits - מקבילים לבלוקים באלגוריתמים BSBI/SPIMI). ה"מאסטר" מקצה כל קבוצת מסמכים למחשב Parser פנוי. ה-Parser קורא את המסמכים ומייצר זוגות של term-docID. ה-Parser כותב את הזוגות הללו ב-j מחיצות שונות, כשכל מחיצה מכילה terms בטווח אותיות שונה. למשל, a-f, g-p, q-z, כשבמקרה זה j=3. השלב הבא הוא השלמת האינדקס, והוא נעשה ע"י ה-Inverters. תפקידו של ה-Inverter הוא לקחת את הזוגות של term-docID ממחיצה אחת בודדת, למיין אותם וליצור postings lists אותם הוא רושם לדיסק.



ההחלטה איזה מחשב יהיה Parser ואיזה יהיה Inverter היא באחריות מחשב ה"מאסטר", והיא אינה קבועה. כלומר, ה"מאסטר" יכול להחליף את התפקיד של מחשב מסוים בהתאם לאילוצים הקיימים במערכת.

MapReduce

האלגוריתם עליו דיברנו כרגע הוא גירסה של MapReduce. MapReduce (Dean and Ghemawat 2004) הוא framework קונספטואלי רובסטי לעבודה עם מחשבים מבוזרים. סכמטית, MapReduce מתאר שני שלבים: (1) map: input → list(k, v), (2) reduce: (k,list(v)) → output.

לבניית האינדקסים האלגוריתם פועל באופן הבא: (1) map: collection → list(termID, docID), (2) reduce: (<termID1, list(docID)>, <termID2, list(docID)>, …) → (postings list1, postings list2, …).

לדוגמא:

- Map:
  - d1 : C came, C c'ed.
  - d2 : C died.
  - → <C,d1>, <came,d1>, <C,d1>, <c'ed, d1>, <C, d2>, <died,d2>
- Reduce:
  - (<C,(d1,d2,d1)>, <died,(d2)>, <came,(d1)>, <c'ed,(d1)>) → (<C,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <c'ed,(d1:1)>)

oren.e.nahum@gmail.com

אינדקס דינאמי

עד עכשיו הנחנו שאוסף המסמכים שלנו הוא סטטי. במציאות, אוסף מסמכי קבוע הוא נדיר. מסמכים שונים מתווספים עם הזמן, מסמכים עוברים עדכונים ומסמכים נמחקים. המשמעות היא שהמילון וה-postings lists דורשים שינויים ועדכונים. הראשון, צריך לעדכן מיקומים של מושגים שכבר נמצאים במילון, והשני, צריך להוסיף מושגים חדשים למילון.

אחד מהדרכים להתמודד עם בעיה זו היא באופן הבא. אנו ננהל אינדקס ראשי גדול, כשמסמכים חדשים יכנסו לאינדקס חדש, קטן. בעת ביצוע חיפוש, הוא יתבצע מול שני האינדקסים והתוצאות יאוחדו לתוצאה אחת. בכדי להתמודד עם מחיקות באופן פשוט, אנו נשמור ווקטור בוליאני (bit-vector) שבו מצוין אם המסמך מחוק או לא (כך שאנו לא באמת צריכים למחוק את המסמך מהאינדקס). כל מסמך שמתקבל מתוצאת החיפוש, נבדק מול הווקטור הנ"ל בכדי לדעת אם הוא נמחק או לא, ואם יש להציג אותו כחלק מהתוצאה או לא.

מידי תקופה, כאשר האינדקס המשני, הקטן, גדל מעבר לגודל מסוים (כך, שלא ניתן לשמור אותו בשלמותו בזיכרון המחשב), אנו בונים את האינדקס הראשי מחדש (כולל את המסמכים הישנים והחדשים).

לשיטה זו מספר חסרונות. (1) בעיות כאשר יש מיזוגים בתדירות גבוהה. (2) ביצועים נמוכים בעת ביצוע פעולת המיזוג. למעשה, ניתן למזג את האינדקס המשני עם האינדקס הראשי ביעילות יחסית אם כל postings list נשמרת בקובץ אחד. אולם, במקרה כזה, פעולת המיזוג היא למעשה פעולת הוספה (Append), ואנו נזדקק להרבה מאד קבצים – לא יעיל מבחינת מערכת ההפעלה. בהמשך ההרצאה אנו מניחים שיש לנו קובץ אינדקס אחד גדול.

מיזוג לוגריתמי

מיזוג לוגריתמי הינה שיטה יעילה יותר להתמודדות עם אוספי מסמכים דינאמיים. המיזוג הלוגריתמי עובד באופן הבא. אנו שומרים מספר אינדקסים, כל אחד גדול פי 2 מקודמו. את האינדקס הקטן ביותר (Z_0) אנו שומרים בזיכרון. אינדקסים גדולים יותר ($I_0, I_1, \ldots$) נשמרים על הדיסק. אם Z_0 גדל מעבר לגודל מסוים, n, נשמור אותו על הדיסק כ-I_0. אם I_0 כבר קיים, אז נמזג אותו עם I_0 כ-Z_1. את Z_1 נשמור כ-I_1, במידה ו-I_1 לא קיים, אחרת נמזג עם I_1 בכדי לצור את Z_2.

```
LMergeAddToken(indexes, Z0, token)
 1  Z0 ← Merge(Z0, {token})
 2  if |Z0| = n
 3    then for i ← 0 to ∞
 4           do if Ii ∈ indexes
 5                then Zi+1 ← Merge(Ii, Zi)
 6                       (Zi+1 is a temporary index on disk.)
 7                     indexes ← indexes − {Ii}
 8                else Ii ← Zi     (Zi becomes the permanent index Ii.)
 9                     indexes ← indexes ∪ {Ii}
10                     Break
11         Z0 ← ∅

LogarithmicMerge()
 1  Z0 ← ∅     (Z0 is the in-memory index.)
 2  indexes ← ∅
 3  while true
 4  do LMergeAddToken(indexes, Z0, getNextToken())
```

בפסאודו-קוד, כל פעם שאנו מוסיפים token ל-Z_0 (שורה 1), אנו בודקים אם Z_0 הגיע לגודל המקסימאלי שלו (שורה 2). אם כן, אנו מבצעים את המיזוג הלוגריתמי (שורות 3 עד 10). אנו עוברים על כל האינדקסים האפשריים, ובודקים אם הם קיימים (שורות 3 ו-4). אם I_i קיים, אנו ממזגים את I_i עם Z_i ויוצרים את Z_{i+1} (כאינדקס כזמני על הדיסק), ובמקביל מוחקים את האינדקס I_i (שורות 4 עד 7). אחרת, אנו יוצרים אינדקס חדש I_i שהוא למעשה האינדקס Z_i (שורות 8 ו-9). בסוף התהליך אנו מאפסים את Z_0.

oren.e.nahum@gmail.com

בהנחה שמספר ה-tokens באוסף המסמכים שלנו (הקורפוס) הוא T. כאשר אנו עובדים בשיטת האינדקס הראשי והמשני, זמן בניית האינדקס הוא $O(T^2)$, מכוון שכל מיקום מטופל בכל מיזוג. כאשר אנו משתמשים במיזוג לוגריתמי, כל מיקום מטופל $O(\log T)$ פעמים, ולכן הזמן הכולל הוא $O(T \log T)$. לפיכך, מיזוג לוגריתמי הוא יעיל יותר לבניית האינדקסים.

מצד שני, כאשר אנו מבצעים שאילתא, אנו צריכים למזג תוצאות של $O(\log T)$ אינדקסים. מעבר לכך, איסוף סטטיסטיקות במקרה של מיון לוגריתמי קשה יותר. לדוגמא, לצורך תיקון שגיאות, אחת השיטות היא להציג את האלטרנטיבה עם מירב התוצאות. איך משיגים את התוצאה הזאת כאשר יש לנו מספר רב של אינדקסים (אולי להתייחס רק לאינדקס הגדול ביותר).

oren.e.nahum@gmail.com

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 (מכווץ 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

 oren.e.nahum@gmail.com

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>

oren.e.nahum@gmail.com

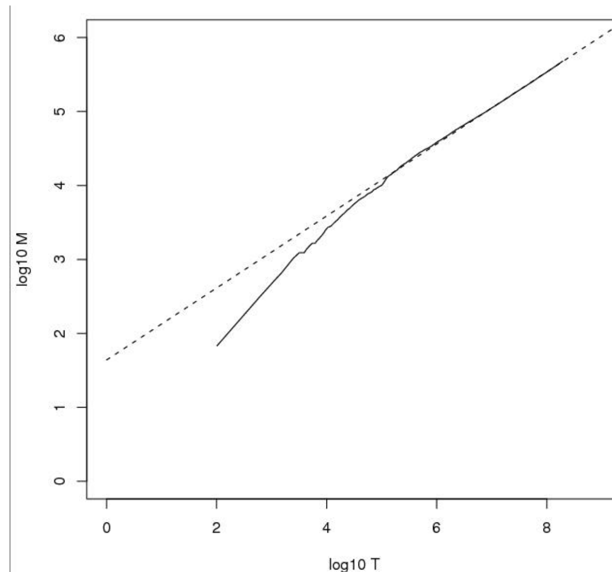
נניח שידוע לנו מהו גודלו של אוסף המסמכים שלנו. האם ניתן לדעת או להעריך מה יהיה גודלו של אוסף ה-terms שלנו (Vocabulary) ? האם ניתן לדעת כמה מילים שונות יש לנו ? (נניח לצורך תכנון מערכת אחזור המידע, בחירת החומרה וכו').

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 שיוויון 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-$b = 0.49$ & $k = 10^{1.64} \approx 44$.

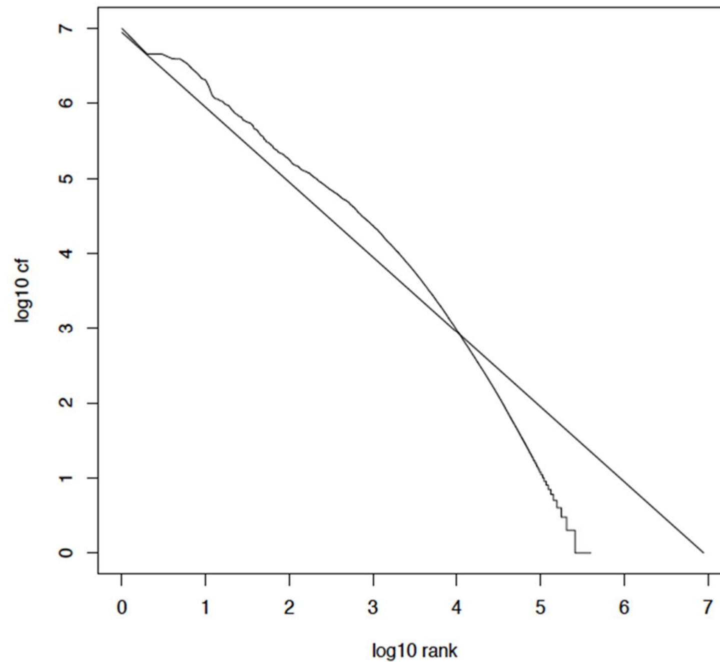
עבור ה-1,000,020 tokens הראשונים החוק חוזה שנקבל 38,323 terms שונים, בפועל מקבלים 38,365 – הבדל קטן מאד (החוק עובד).

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

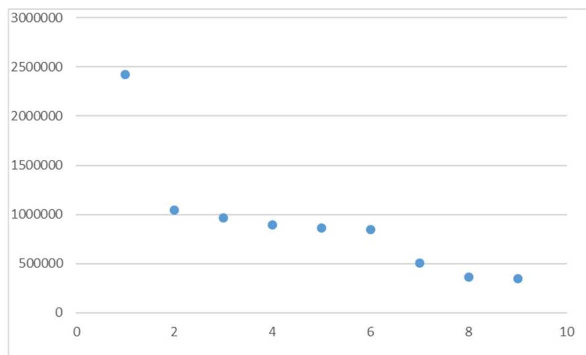
הטבלה הבאה מתארת את מס' ההופעות של תשעת ה-term-ים הנפוצים ביותר במסמך מסויים.

דרוג	מס' הופעות
1	2420778
2	1045733

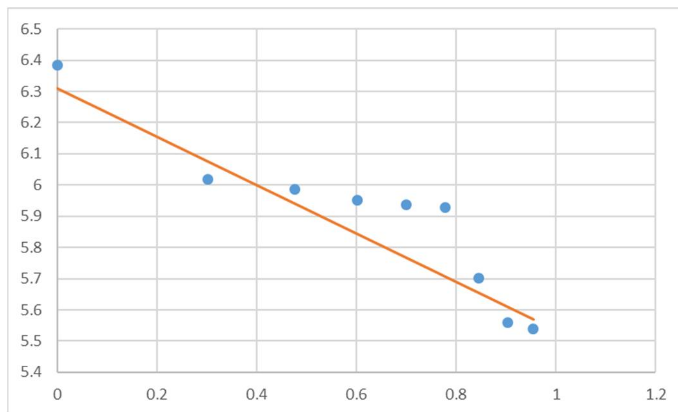
oren.e.nahum@gmail.com

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

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.

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

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

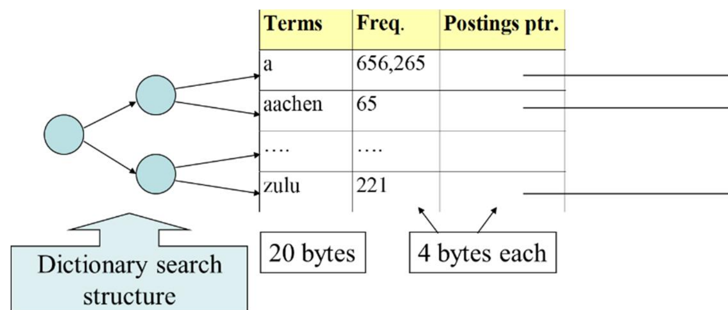
דוגמא נוספת. נניח שידוע שה-term הנפוץ ביותר מופיע 1000 פעמים. כמו כן, ה-term השני הנפוץ ביותר מופיע 250 פעמים. כמה פעמים מופיע ה-term השלישי הנפוץ ביותר ?

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 רגרסיה לוגריתמית, ועל סמך הרגרסיה לחשב את Cf_3. במקרה שלנו, פונקצית הריגרסייה הלוגריתמית יוצא $\log Cf_i = 3 - 2\log i$. בהתאם לכך, $Cf_3 = 111.11$.

<u>כיווץ המילון</u>

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.



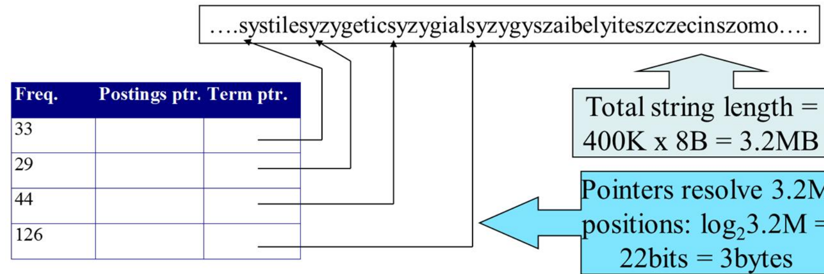
נניח שהמילון שלנו שמור כמערך שבו רשומות באורך קבוע. מבנה הרשומה הוא באופן הבא: (1) ה-term, אשר תופס 20 תווים, (2) מספר המסמכים שבו מופיע ה-term, מיוצג ע"י integer, שתופס 4 תווים, ו-(3) מצביע ל- posting list, שגם הוא מיוצג ע"י integer, ותופס 4 תווים. סה"כ גודלה של רשומה הוא 28 תווים.

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

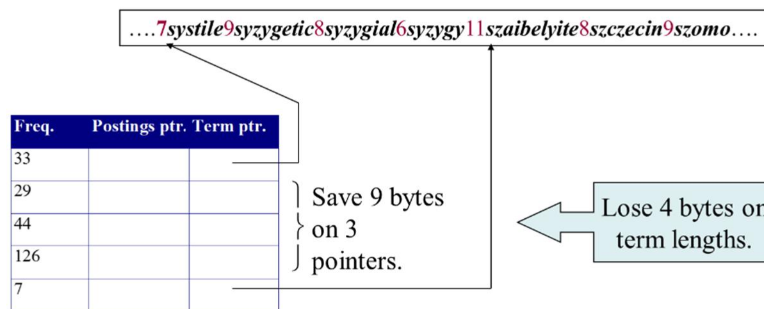
בכדי לחסוך בזיכרון, נעבוד באופן הבא. נשמור את כל ה-terms של המילון כרשימה (string) אחת ארוכה. במקום לשמור את ה-term בכל רשומה, אנו שומרים ברשומה של ה-term מצביע לתחילת ה-term ברשימה שלנו (ה-string). בכדי לדעת מה אורכו של ה-term (בכדי שנוכל "לחלץ" אותו מתוך ה-string הארוך שלנו), אנו נשתמש במצביע לתחילת 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.



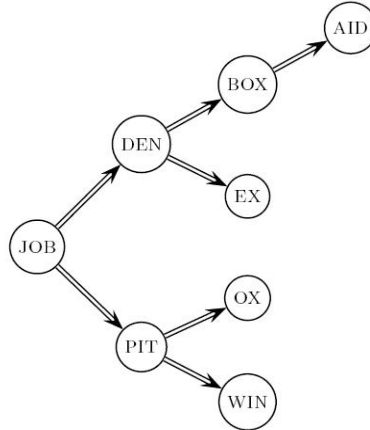
בכדי שנוכל לדעת איפה מתחיל ומסתיים כל term ב-string (יש לנו את המיקום של תחילת כל k=4, אבל אנו לא יודעים מה קורה בתוך החלק הזה), עבור כל term אנו צריכים לשמור את האורך שלו (תוספת של בית אחד, שיכול להכיל ערכים בין 0 ל-255, שמתווסף בהתחלה, לפני ה-term עצמו).

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

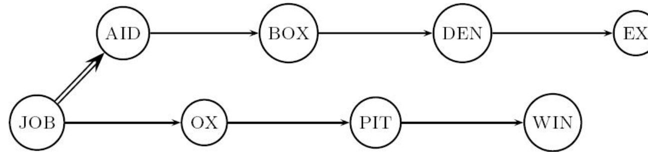
ניתן לחסוך מקום נוסף אם נשתמש ב-k גדול יותר, אך האם כדאי להשתמש ב-k גדול ? לא מכוון שאז זמן החיפוש של term המתאים יהיה לינארי.

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****a1◊***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 הבא 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.

oren.e.nahum@gmail.com

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 → 1101 → 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

 oren.e.nahum@gmail.com

אחזור מידע

Ranked retrieval

עד כה, השאילתות שלנו היו בוליאניות, כלומר, או שהמסמכים הכילו את התשובה לשאילתא שלנו או שלא. שיטת עבודה זו מתאימה למשתמשים "מומחים", בעלי הבנה מדויקת של צרכיהם ושל אוסף המסמכים, וכן עבור אפליקציות, שיכולות לעבד באופן עצמאי את התוצאות המתקבלות. אולם, עבור רוב המשתמשים צורת עבודה זו לא מתאימה משתי סיבות עיקריות: (1) רוב המשתמשים אינם בקיאים ו/או יכולים לכתוב שאילתות בוליאניות (או שאילתות מדוייקות), ו-(2) לרוב, כמות התשובות החוזרת משאילתא בוליאנית הינה גדולה מאד, ורוב המשתמשים אינם יכולים להתמודד כמויות גדולות של מסמכים.

ליתר דיוק, ברוב המקרים, שאילתא בוליאנית מחזירה או מעט מידי תוצאות (0) או מספר רב של תוצאות (1000). לדוגמא:

- שאילתא 1: "standard user dlink 650" → 200,000 hits
- שאילתא 2: "standard user dlink 650 no card found" → 0 hits

לכן, דרושה מיומנות גבוהה בכדי לנסח שאילתא שתחזיר מס' תוצאות שאפשר להתמודד איתו.

במקרה של ranked retrieval, במקום להחזיר קבוצת מסמכים העונה לתוצאות השאילתא, המערכת מחזירה קבוצת מסמכים, כשכל מסמך מקבל "ציון". לרוב, כאשר אנו מתייחסים ל-ranked retrieval, אנו גם מתכוונים לכך שהשאילתות שלנו כתובות כטקסט חופשי. כלומר, במקום להשתמש בשפה מיוחדת לכתיבת השאילתא (בדומה לשימוש ב-And, or ו-not בשאילתות בוליאניות), השאילתא נכתבת כמשפט טבעי בשפה.

כאמור, שאילתא בוליאנית לרוב מחזירה מעט מידי תוצאות (0) או מספר רב של תוצאות (1000). גם במקרה של ranked retrieval המערכת עלולה להחזיר קבוצת מסמכים גדולה (מאד), שרובם לא יהיו בשימוש. אולם כעט יש באפשרותנו להציג למשתמש רק את k (≈ 10) התוצאות הטובות ביותר, במקום "להפציץ" אותו בתוצאות שכנראה פחות רלוונטיות עבורו.

בכדי שנוכל להחזיר למשתמש, בסדר עולה, מסמכים שהם רלוונטיים ושימושיים למשתמש, אנו צריכים לדרג/למיין את המסמכים שלנו ביחס לשאילתא שלנו. בכדי שנוכל לעשות זאת לכל מסמך אנו ניתן ציון, נניח בין 0 ל-1, המבטא עד כמה המסמך עונה בצורה טובה לשאילתא שלנו.

נניח שיש לנו שאילתא שמורכבת מ-term אחד בלבד. אנו מעוניינים במנגנון למתן ציון הפועל באופן הבא: אם ה-term לא נמצא במסמך, המסמך מקבל ציון 0. ככל שמספר המופעים של ה-term מסמך גדול יותר, המסמך מקבל ציון גבוה יותר.

נבחן מספר אפשרויות לאופן מתן הציון.

Jaccard coefficient

דיברנו על מדד זה בהרצאה 3. JC הינו שיטת נירמול המציינת כמה איברים (terms) משותפים בין שני מסמכים A ו-B (או מסמך ושאילתא), מבין כלל האיברים, ללא כפילויות, שנמצאים בשני המסמכים.

JC מוגדר באופן הבא: $JC(A,B) = |A \cap B| / |A \cup B|$

מהגדרה זו, כאשר משווים מסמך עם עצמו, אז $JC(A,A) = 1$, ואילו כאשר אין איברים משותפים כלל בין שני המסמכים, כלומר $A \cap B = 0$, אז $JC(A,B) = 0$. כזכור, A ו-B אינם צריכים להיות באותו הגודל, והמדד תמיד מחזיר תוצאה בין 0 ל-1.

לדוגמא:

- <u>Query</u>: *ides of march*
- <u>Document</u> 1: *caesar died in march* , JC=1/6
- <u>Document</u> 2: *the long march* , JC=1/5

הבעייתיות עם JC, היא שהוא אינו מתייחס לתדירות הופעת ה-term במסמך. ישנם terms שתדירות ההופעה שלהם נמוכה (נדירים), ולכן term כזה אם הוא מופיע בשאילתא ובמסמך יש לו יותר משמעות מאשר term שכיח. JC אינו מתייחס לנושא זה, ולכן, אנו צריכים מדד טוב יותר, שישקף את הנקודות הנ"ל.

במטריצת מפגשים אנו השתמשנו בערכים בוליאניים בכדי לציין אם term מסוים מופיע במסמך או לא. אנו יכולים להרחיב או לשנות את אופן הייצוג של המטריצה כך, שבמקום להשתמש בערך בוליאני, שמציין אם term קיים או לא, נרשום את מספר המופעים של אותו term במסמך (למטריצה זו אנו קוראים count matrix).

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	0
Brutus	4	157	0	1	0	0
Caesar	232	227	0	2	1	1
Calpurnia	0	10	0	0	0	0
Cleopatra	57	0	0	0	0	0
mercy	2	0	3	5	5	1
worser	2	0	1	1	1	0

באופן זה, אנו יכולים לדעת כמה פעמים מופיע כל term במסמך, ולהתייחס לכך בהתאם. אולם, השימוש בווקטורים אינו משמר את מיקום ה-terms במסמך. לדוגמא, המסמכים John is quicker than Mary ו- Mary is quicker than John הם בעלי אותו הווקטור. אנו קוראים למצב הזה bag of words model. יש לנו מסמכים שהם בעלי אותו הווקטור, כלומר, עבוד שאילתא מסוימת, אם מסמך אחד עונה עליה, אז גם השני עונה עליה באותה המידה. נראה, אם כך, שחזרנו שלב אחד אחורנית. היה לנו פתרון לבעיה זו והוא השימוש ב-positional index, אפשר לנו להבדיל בין שני המסמכים. אנו נראה פתרון לכך בהמשך.

Term frequency tf

ה-term frequency, $tf_{t,d}$, של term t במסמך d, מוגדר כמספר הפעמים ש-t מופיע ב-d. מסמך שבו term מסוים מופיע 10 פעמים יותר רלוונטי ממסמך שבו ה-term מופיע פעם אחת בלבד, אבל, הוא לא רלוונטי פי 10. רלוונטיות של מסמך היא לא פרופרציונאלית למס' הפעמים ש-term נתון מופיע בו. בהתאם לכך, אנו מעוניינים להשתמש במדד ה-term frequency בעת חישוב ציון המסמך. כיצד נעשה זאת ?

אנו נגדיר מדד חדש המבוסס על ה-ft (מבוסס על תצפיות אימפריות), מדד זה הוא ה-Log-frequency weighting של term t במסמך d, הוא מוגדר כ:

$$w_{t,d} = \begin{cases} 1 + \log_{10} tf_{t,d} & tf_{t,d} > 0 \\ 0 & otherwise \end{cases}$$

בהתאם למדד זה, אם term מופיע 0 פעמים, אז $w_{t,d} = 0$, אם term מופיע פעם אחת, אז $w_{t,d} = 1$, באופן דומה אם הוא מופיע פעמיים אז $w_{t,d} = 1.3$, אם הוא מופיע 10 פעמים אז $w_{t,d} = 2$, אם הוא מופיע 1000 פעמים אז $w_{t,d} = 4$, וכך הלאה.

את הציון הכללי של המסמך נחשב ביחס לשאילתא באופן הבא: נסכום את ה-Log-frequency weighting של כל ה-terms שמופיעים גם בשאילתא וגם במסמך.

$$Score = \sum_{t \in q \cap d} \left(1 + \log_{10} tf_{t,d}\right)$$

במידה ואף אחד מה-terms שמופיעים בשאילתא לא מופיע במסמך, אנו מקבלים ציון 0.

Document frequency

כאמור, terms שהינם נדירים יש להם יותר משמעות מ-terms שכיחים (למשל stop words). נניח שיש לנו term נדיר בשאילתא. מסמך שמכיל את ה-term הזה הוא, ככל הנראה, מסמך רלוונטי עבורנו. במקרים כגון אנו, אנו רוצים שה-terms הנדירים הללו יהיו בעלי משקל גבוה יותר מ-terms שכיחים. כמו כן, נניח שיש לנו שאילתא שמכילה

terms שהינם שכיחים. מסמכים שמכילים את ה-terms הללו הינם יותר רלוונטיים ממסמכים שאינם מכילים את ה-terms כלל. אנו רוצים שגם terms כאלו יהיו בעלי משקל גבוהה יחסית, אך נמוך מאשר של terms נדירים.

נגדיר מדד חדש, df_t, שהוא document frequency של t, כלומר מספר המסמכים המכילים את t (והוא זהה לאורכו של ה-posting list של אותו ה-term כאשר אנו משתמשים ב-non-positional index). ערכו של df_t אינו גבוה ממספר המסמכים שיש לנו, $df_t \leq N$. למעשה, ככל ש-term יותר נדיר, כך הוא נמצא בפחות מסמכים, ובהתאם ערכו של ה-df_t נמוך יותר. בהתאם לכך, אנו מגדירים את idf (inverse document frequency) של t באופן הבא:

$$idf_t = \log_{10}\left(\frac{N}{df_t}\right)$$

כאשר אנו משתמשים ב-log בכדי לדכא את האפקט של idf. באופן זה, ככל שה-term t נמצא בפחות מסמכים, כך ערכו של idf_t עולה, כפי שרואים בדוגמא הבאה (שבה N=1,000,000).

term	df_t	idf_t
calpurnia	1	6
animal	100	4
sunday	1,000	3
fly	10,000	2
under	100,000	1
the	1,000,000	0

האם ל-idf יש השפעה על הציון שמקבל מסמך כשהשאילתא בנוייה מ-term אחד ? לא, ל-idf אין השפעה על הציון שמקבל מסמך כאשר השאילתא מורכבת מ-term אחד. idf משפיע על ציון המסמך כאשר השאילתא מורכבת מלפחות שני terms, מכוון שהמטרה של מדד זה היא להביא לידי ביטוי את השכיחות של כל אחד מה-terms של השאילתא במסמך, ולכן, אם השאילתא בנוייה מ-term אחד, אז אין למדד זה משמעות. לדוגמא, עבור השאילתא capricious person, idf מתייחס למספר המופעים של capricious בתוצאה הסופית, יותר מאשר מספר המופעים של person.

ה-collection frequency של t הוא מספר המופעים של t באוסף המסמכים, כולל חזרות, בניגוד ל-document frequency של t, כלומר מספר המסמכים המכילים את t. מדוע השתמשנו ב-inverse document frequency ולא ב-inverse collection frequency:

לצורך ההסבר נשתמש בדוגמא הבאה:

Word	Collection frequency	Document frequency
insurance	10440	3997
try	10422	8760

בדוגמא זו, ה-cf של insurance הוא 10440 ואילו ה-cf של try הוא 10422. אם נסתכל על ה-df, אז ה-df של insurance הוא 3997, ואילו של try הוא 8760. בהתאם לכך, איזה term יותר טוב בחיפוש, ויקבל ציון גבוה יותר ? אם אנו מסתכלים על ה-cf אנו רואים שהערכים של שני ה-terms דיי דומים, ולכן קשה להעריך מי מהם עדיף. אבל אם מסתכלים על ה-df, רואים ש-insurance הינו נדיר יותר מ-try, ולכן הוא עדיף בחיפוש, ויקבל ציון גבוה יותר.

tf.idf weighting

ה-tf.idf של term שווה למכפלה של ה-tf שלו ב-idf שלו. זהו מדד המשלב את מספר המופעים של term מסוים באוסף המסמכים ואת הנדירות של אותו ה-term.

$$w_{t,d} = \log(1 + tf_{t,d}) \times \log_{10}\left(\frac{N}{df_t}\right)$$

מדד זה הוא המדד הטוב ביותר בתחום אחזור המידע על מנת לתת ציונים למסמכים בהתאם לשאילות. ערכו של מדד זה עולה ככל שמספר ההופעות של term גדל במסמך, כמו כן, הוא גדל ככה שה-term נדיר יותר באוסף המסמכים שלנו.

בהתאם לכך, ציון של מסמך בהינתן שאילתא מוגדר כ:

$$Score(q,d) = \sum_{t \in q \cap d} tf.idf_{t,d}$$

כשקיימות הרבה גרסאות לחישוב זה. למשל, איך אנו מחשבים את tf (עם או בלי log), האם ל-terms בשאילתא יש משקלים, וכו'.

אנו יכולים לחזור למטריצת מפגשים, כאשר הפעם נרשום את ערך ה-tf.idf של כל term במסמך.

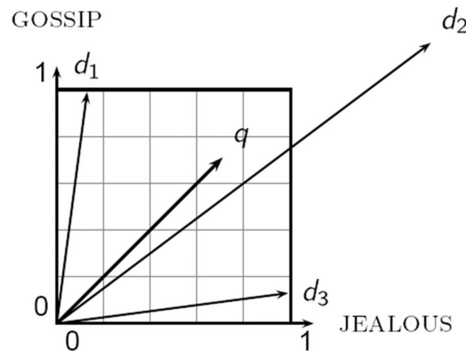
	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	5.25	3.18	0	0	0	0.35
Brutus	1.21	6.1	0	1	0	0
Caesar	8.59	2.54	0	1.51	0.25	0
Calpurnia	0	1.54	0	0	0	0
Cleopatra	2.85	0	0	0	0	0
mercy	1.51	0	1.9	0.12	5.25	0.88
worser	1.37	0	0.11	4.15	0.25	1.95

מסמכים כווקטורים

אנו יכולים להתייחס לאוסף המסמכים שלנו כמרחב ווקטורי בעל |V| מיימדים, כאשר ה-terms הם צירים באותו המרחב ואילו המסמכים הם נקודות או ווקטורים במרחב. כאשר מדובר על חיפוש ב-web, אז המרחב הווקטורי בעל מס' מיימדים מאד גדול. כמו כן, הווקטורים הללו מאד דלילים.

בכדי לבצע שאילתות, אנו תחילה נייצג שאילתות כווקטורים במרחב (אפשר להתייחס לשאילתא עצמה כאל מסמך מאד קטן), ואח"כ, נדרג כל מסמך בהתאם לקרבתו לווקטור השאילתא במרחב. כאשר קירבה, המרחק ההופכי, מייצגת את הדמיון בין הווקטורים. באופן זה אנו יכולים להימנע משאילתות בוליאניות, וכן אנו נותנים ציון גבוה יותר למסמכים שיותר רלוונטיים לנו, וציון נמוך למסמכים שלא רלוונטיים לנו.

בהתאם לכך, כיצד נגדיר מרחק ? המרחק האוקלידי, כפי שניתן לראות מהדוגמא, בין $\vec{q}$ ובין $\vec{d_2}$ הוא גדול, למרות שהתפלגות ה-terms בשאילתא $\vec{q}$ והתפלגות ה-terms ב-$\vec{d_2}$ מאוד דומות.



נבצע ניסוי תאורטי. ניקח מסמך d ונוסיף אותו לעצמו. נקרא למסמך החדש d'. "סמנטית", ל-d ול-d' יש את אותו התוכן, אולם המרחק האוקלידי בין שני המסמכים הוא יחסית גדול. אם נשתמש בייצוג פולארי, אנו נראה שהווקטור d' גדול פי 2 מהווקטור d, אולם הזווית בניהם היא 0, והיא מתאימה לזהות מקסימאלית בין המסמכים. אם כך,

נדרג את המסמכים בהתאם לזווית שביניהם. ומכאן, שבכדי לדרג את המסמכים בסדר יורד של הזווית בין השאילתא והמסמך אנו יכולים לדרג את המסמכים בסדר עולה של ערך הקוסינוס של הזווית בין וקטור השאילתא ווקטור המסמך.

מתמטית, ניתן לנרמל את אורכו של ווקטור ע"י חלוקת כל אחד ממרכיביו באורכו. לשם כך נגדיר את L_2 באופן הבא:

$$\|\vec{x}\|_2 = \sqrt{\sum_i x_i^2}$$

אם נחלק את הווקטור ב-L_2 שלו, נקבל את ווקטור היחידה. אם נסתכל על d ו-d', נראה שיש להם את אותו ווקטור לאחר הנירמול.

<u>cosine(query,document)</u>

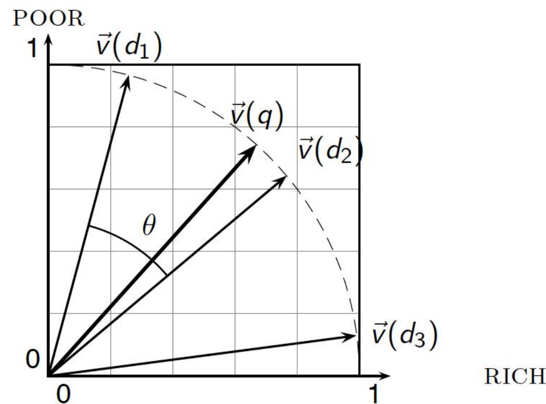
את ערך הקוסינוס בין השאילתא q והמסמך d אנו מחשבים באופן הבא:

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q}\cdot\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\vec{q}}{|\vec{q}|}\cdot\frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

וזאת כאשר q_i הוא tf.idf של term i בשאילתא ו-d_i הוא td.idf של term i במסמך.

עבור הווקטור המנורמל, הערך קוסינוס הדימיון הוא ה-dot product (תזכורת אם A ו-B הם ווקטורים אז $A\cdot B = A_1B_1 + A_2B_2 + \cdots + A_nB_n$, כמו כן, אם $\|A\|$ מציין את אורכו של הווקטור A, ו-$\|B\|$ מציין את אורכו של הווקטור B, אז $A\cdot B = \|A\|\|B\|cos(\theta)$, כאשר θ היא הזווית בין A ל-B).

$$\cos(\vec{q},\vec{d}) = \vec{q}\cdot\vec{d} = \sum_{i=1}^{|V|} q_i d_i$$



דוגמא:

עד כמה דומים המסמכים ? (לצורך הפשטות אנו לא נחשב את ה-idf).

term	SaS	PaP	WH
affection	115	58	20
jealous	10	7	11
gossip	2	0	6
wuthering	0	0	38

Term frequencies (counts)

יש לנו שלושה ספרים, SaS: *Sense and Sensibility*, PaP: *Pride and Prejudice* ו-.?WH: *Wuthering Heights*

לצורך הדוגמא אנו נתייחס לארבעה term-ים בספרים אלו. בטבלה מוצגים ה-term frequency עבור כל אחד מה-term-ים בארבעת הספרים שלנו. אנו צריכים לחשב את ה-tf וכן את ה-idf (בדוגמא אנו נניח שהערכי ה-idf הם 1 עבור כל ה-term-ים), ולהכפיל בין התוצאות. לדוגמא, עבור ה-term affection בספר SaS, ערכו של ה-tf הוא $ft = 1 + \log_{10} 115 = 3.06$. הערכים של tf עבור שאר ה-term-ים בספרים השונים מובאים בטבלה Log frequency weighting. בשלב הבא אנו נחשב עבור כל ווקטור (עמודה בטבלה) את ערכו המנורמל, ובהתאם את ערכי ה-term-ים המנורמלים. למשל, עבור הווקטור הראשון, SaS, אורכו הוא $\sqrt{3.06^2 + 2^2 + 1.3^2 + 0^2} = \sqrt{15.0536} = 3.88$. בהתאם לכך, ערכו המנורמל של ה-term affection בספר SaS הוא $\frac{3.06}{3.88} = 0.789$. שאר הערכים המנורמלים מובאים בטבלה After length normalization.

Log frequency weighting

term	SaS	PaP	WH
affection	3.06	2.76	2.30
jealous	2.00	1.85	2.04
gossip	1.30	0	1.78
wuthering	0	0	2.58

After length normalization

term	SaS	PaP	WH
affection	0.789	0.832	0.524
jealous	0.515	0.555	0.465
gossip	0.335	0	0.405
wuthering	0	0	0.588

נשתמש בנוסחא $\cos(\vec{q}, \vec{d}) = \vec{q} \cdot \vec{d} = \sum_{i=1}^{|V|} q_i d_i$ בכדי לחשב את הדמיון בין המסמכים השונים.

$\cos(\text{SaS,PaP}) \approx 0.789 \times 0.832 + 0.515 \times 0.555 + 0.335 \times 0.0 + 0.0 \times 0.0 \approx 0.94$

$\cos(\text{SaS,WH}) \approx 0.79$

$\cos(\text{PaP,WH}) \approx 0.69$

בהתאם לתוצאות, SaS ו-PaP הוא הספרים הדומים ביותר (או הקרובים ביותר) אחד לשני.

לסיכום, אופן ביצוע השאילתא מתואר באלגוריתם הבא:

```
CosineScore(q)
 1  float Scores[N] = 0
 2  float Length[N]
 3  for each query term t
 4  do calculate w_{t,q} and fetch postings list for t
 5      for each pair(d, tf_{t,d}) in postings list
 6      do Scores[d] += w_{t,d} × w_{t,q}
 7  Read the array Length
 8  for each d
 9  do Scores[d] = Scores[d]/Length[d]
10  return Top K components of Scores[]
```

הטבלה הבאה מסכמת מס' אפשרויות לחישוב ה-Term frequency, ה-Document frequency וביצוע הנירמול.

Term frequency		Document frequency		Normalization	
n (natural)	$\mathrm{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1+\log(\mathrm{tf}_{t,d})$	t (idf)	$\log\frac{N}{\mathrm{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2+w_2^2+\ldots+w_M^2}}$
a (augmented)	$0.5+\frac{0.5\times\mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$	p (prob idf)	$\max\{0,\log\frac{N-\mathrm{df}_t}{\mathrm{df}_t}\}$	u (pivoted unique)	$1/u$
b (boolean)	$\begin{cases}1 & \text{if } \mathrm{tf}_{t,d}>0\\0 & \text{otherwise}\end{cases}$			b (byte size)	$1/CharLength^{\alpha}$, $\alpha<1$
L (log ave)	$\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t\in d}(\mathrm{tf}_{t,d}))}$				

הרבה מנועי חיפוש מאפשרים שימוש של משקלים שונים בין השאילתא למסמכים. SMART – מייצג את הקומבינציה שבשימוש ע"י מנוע החיפוש, כשהסימול הוא בפורמט ddd.qqq (כל שלשה היא בהתאם לאותיות המופיעות בטבלה למעלה). לרוב נעשה שימוש ב-lnc.ltc, שמשמעו, עבור המסמך logarithmic term frequency, no document frequency ו-cosine normalization. ועבור השאילתא logarithmic term frequency, idf document frequency ו-cosine normalization.

לדוגמא:

- המסמך: car insurance auto insurance
- השאילתא: best car insurance

Term	Query						Document				Prod
	tf-raw	tf-wt	df	idf	wt	n'lize	tf-raw	tf-wt	wt	n'lize	
auto	0	0	5000	2.3	0	0	1	1	1	0.52	0
best	1	1	50000	1.3	1.3	0.34	0	0	0	0	0
car	1	1	10000	2.0	2.0	0.52	1	1	1	0.52	0.27
insurance	1	1	1000	3.0	3.0	0.78	2	1.3	1.3	0.68	0.53

Doc length = $\sqrt{1^2 + 0^2 + 1^2 + 1.3^2} \approx 1.92$

Score = 0+0+0.27+0.53 = 0.8

אחזור מידע

Scoring and results assembly

את ההרצאה הקודמת סיימנו עם האלגורית הבא לחישוב ציוני מסמכים המתבסס על ערכי הקוסינוס.

```
COSINESCORE(q)
 1  float Scores[N] = 0
 2  float Length[N]
 3  for each query term t
 4  do calculate w_{t,q} and fetch postings list for t
 5      for each pair(d, tf_{t,d}) in postings list
 6      do Scores[d] += w_{t,d} × w_{t,q}
 7  Read the array Length
 8  for each d
 9  do Scores[d] = Scores[d]/Length[d]
10  return Top K components of Scores[]
```

כאשר אנו שואלים שאילתא, אנו למעשה רוצים למצוא את k המסמכים באוסף המסמכים "הקרובים ביותר" לשאילתא שלנו. כלומר את k המסמכים עם ערך ה-cosine הגבוה ביותר.

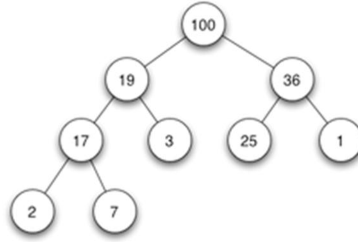
בכדי לבצע פעולה זו באופן יעיל אנו צריכים למצוא דרך לחשב את ערך ה-cosine לכל מסמך בצורה יעילה. ובהתאם, לבחור את k המסמכים עם ערך ה-cosine הגבוה ביותר, גם כן באופן יעיל.

השאלה היא, האם ניתן לעשות זאת מבלי לחשב את כל N ערכי ה-cosine (של כל המסמכים בקורפוס) ? (כן, אנו צריכים להתייחס רק למסמכים שנמצאים ב-posting lists של כל אחד מה-terms שמופיעים בשאילתא. אבל אנו רוצים משהו יעיל יותר מזה).

מה למעשה אנו עושים ? אנו פותרים את בעיית k השכנים הקרובים ביותר עבור ווקטור השאילתא. באופן כללי, אנו לא יודעים איך לפתור בעיה זו באופן יעיל כאשר יש לנו הרבה מסמכים ושאילתא ארוכה. אבל, בעיה זו פתירה עבור שאילתות קצרות (עם מספר מועט של terms), ואינדקסים סטנדרטיים תומכים בה בצורה טובה.

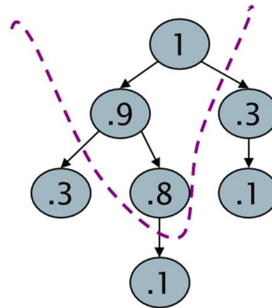
נסתכל על מקרה מיוחד, שהו אין משקלים לביטויים בשאילתא (אנו מניחים שכל ביטוי בשאילתא מופיע פעם אחת בלבד). במקרה כזה, לצורך הדרוג, אנו לא צריכים לנרמל את ווקטור השאילתא. ובהתאם האלגוריתם הוא מעט פשוט יותר מזה שהוצג בהרצאה הקודמת (בשורה 6 של האלגוריתם $w_{t,q}$ הוא 1).

ברור במקרים אנו רוצים לקבל את k המסמכים בעלי הדרוג הגבוה ביותר (לפי cosine ranking), ואנו רוצים לעשות זאת מבלי לחשב את ה-cosine ranking לכל מסמך ומסמך ומבלי למיין את כל המסמכים. נניח ש-J מציין את המסמכים להם ה-cosine ranking שונה מ-0 (מס' זה יכול להיות גבוה מאד), אנו רוצים לקבל את K המסמכים הטובים ביותר מתוך J. בכדי לעשות זאת ביעילות טובה יותר מ-$j \log j$ (היעילות של מיון), אנו יכולים להשתמש במבנה נתונים שנקרא heap. Heap הוא מבנה נתונים בצורת עץ בינארי המקיים תכונה בסיסית: המפתח של כל צומת בעץ קטן ממפתח אביו (בערימת מקסימום). כתוצאה מדרישה זו, הצומת בעל המפתח הגדול ביותר הוא תמיד השורש של העץ, וניתן למצוא אותו בסיבוכיות זמן $O(1)$ (כלומר במספר פעולות קבוע, שאיננו תלוי במספר האיברים בערימה).



בכדי לבנות את העץ אנו צריכים לבצע $2\times J$ פעולות. ואח"כ, בשביל לבחור את k המסמכים הטובים ביותר $\log(2\times J)$ פעולות לכל מסמך.

עבור J=1M ו-K=100, התהליך הזה לוקח בערך 10% מהעלות של מיון.



הבעיה העיקרית שלנו או במילים אחרות, צוואר הבקבוק העיקרי שלנו הוא חישוב ה-cosine. זה חישוב שלוקח הרבה זמן. ולכן, האם אנו יכולים להימנע מכל החישובים האלו ? כן, ניתן להשתמש בהיוריסטיקות שונות שיתנו לנו ערכים קרובים לחישוב שלנו (באופן מהיר ופשוט יותר), אבל אז אנו עלולים לקבל תשובות שגויות. במקרים כאלו, מסמך שלא נמצא ברשימת k המסמכים הטובים ביותר יכול להופיע ככזה. אבל, האם זה כל כך נורא ?

למשתמש יש משימה ממנה מגזרת השאילתא. ערך ה-cosine מוחשב למסמכים בהתאם לשאילתא, ולפיכך, ערך ה-cosine מתאר עד כמה המשתמש "שמח" עם התוצאה (המסמך) שקיבל. אבל, אנו מניחים שאם מסמך מכיל את ה-terms שמופיעים בשאילתא (בהתאם לתדירות והנדירות) אז הוא מסמך רלוונטי – אבל יתכן שאנו טועים והוא לא רלוונטי. ולכן, אם נקבל רשימה של k מסמכים "קרובים" ל-k המסמכים הטובים ביותר לפי ה-cosine, אז אנחנו במצב טוב. ואין זה נורא עם יהיו כמה תשובות שגויות ברשימת k המסמכים הטובים ביותר.

אם כך, אופן הפעולה שלנו יהיה באופן הבא: נמצא קבוצה A של מועמדים, כש-K < |A| << N (A לא חייב לכלול את K המסמכים הטובים ביותר, אבל הוא מכיל הרבה מ-K המסמכים הטובים ביותר). וכתשובה נחזיר את K המסמכים הטובים ביותר מ-A. ניתן ליישם גישה זה אם פונקציות דרוג שונות (לא בהכרח cosine). בהמשך נציג מספר סכמות שעובדות בצורה זו.

Index elimination

אלגוריתם חישוב ה-cosine הבסיסי מתייחס רק למסמכים שבהם יש לפחות term אחד מהשאילתא. ניתן להרחיב גישה זו. אפשרות אחת היא שנתייחס רק ל-terms בשאילתא עם high-idf. אפשרות נוספת היא שנתייחס רק למסמכים המכילים מספר terms מהשאילתא.

נניח שנתונה השאילתא: catcher in the rye. אינטואיטיבית, in ו-the תורמים מעט לציון (הציון שלהם קרוב ל-0), ולכן הם כמעט ולא משנים את הדרוג. ולכן, אנו צריכים לסכום רק את הציונים המתקבלים מ-catcher ומ-rye.

oren.e.nahum@gmail.com

מכוון שביטויים בעלי low-idf נמצאים בהרבה מסמכים, אם נתעלם מביטויים אלו, המסמכים הללו לא יופיעו בקבוצה A.

כל מסמך המכיל לפחות ביטוי אחד מהביטויים שבשאילתא הוא מועמד להיות חלק מ-K המסמכים הטובים ביותר. אם השאילתא מכילה מספר ביטויים, אנו נחשב את הציון של מסמכים המכילים יותר מביטוי אחד מכלל הביטויים של השאילתא.

Antony → 3 | 4 | 8 | 16 | 32 | 64 | 128

Brutus → 2 | 4 | 8 | 16 | 32 | 64 | 128

Caesar → 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34

Calpurnia → 13 | 16 | 32

בדוגמא, אם נניח שאנו מתייחסים לפחות ל-3 מתוך 4 terms, אנו נחשב את הציון רק עבור מסמכים מס' 8, 16 ו-32. שיטה זו קלה ליישום.

Champion lists

בשיטה זו, לכל term במילון אנו נחשב/נמצא את r (אנו צריכים לבחור את r בזמן בניית האינדקס) המסמכים עם הציון (tf.idf) הגבוה ביותר. למסמכים האלו אנו קוראים Champion lists (או fancy list או top docs). במקרה זה, יתכן ו-r יהיה קטן מ-K. בעת ביצוע השאילתא, נחשב את הציון רק עבור מסמכים שנמצאים ב- Champion lists(מספיק שעבור term מסוים, מסמך כלשהו יהיה ב-champion list בשביל שנתייחס אליו, אין צורך שאותו מסמך יהיה ב-champion list של כל ה-terms), בהתאם לביטויים שבשאילתא. K המסמכים הטובים ביותר יהיו מבין המסמכים ששייכים ל-Champion list .

Static quality scores

עד עכשיו התייחסנו לכל המסמכים כזהים בחשיבות. אבל אין זה כך. נניח שישנם שני מסמכים בנושא מסוים, אחד נכתב ע"י תלמיד בבית ספר, ואילו השני נכתב ע"י מרצה באוניברסיטה. במקרה זה, שני המסכים רלוונטיים בהקשר לנושא, ואולם אנו רוצים להתייחס גם לסמכותיות של המסמך. במקרה זה, המסמך שנכתב ע"י מרצה האוניברסיטה הוא יותר סמכותי. אפשר לסכם זאת כך: אנו רוצים שהמסמכים עם הדרוג הגבוה יהיו גם רוולטים וגם סמכותיים. את הרלוונטיות של המסמך אנו מודדים ע"י ציון ה-cosine. אבל, סמכותיותו של מסמך היא תכונה שאינה תלויה בשאילתא. להלן מס' דוגמאות שיכולות לציין שמסמך הוא מוסמך: דפי וויקיפדיה, מאמר בעיתונים מסוימים, מאמר עם מספר רב של ציטוטים, הרבה הפניות לדף ווב מסוים וכו'.

בכדי שנוכל להתייחס לסמכותיותו של מסמך, לכל מסמך אנו נשייך מדד, g(d) (goodness), שמציין את סמכותיות המסמך כערך בין 0 ל-1, ללא תלות בשאילת. למשל, מספר הציטוטים של מסמך מסויים יתורגם לערך בין 0 ל-1. אנו רוצים לאחד את ציון ה-cosine וציון הסמכותיות לערך אחד. אפשרות אחת היא ע"י סכימת שני הערכים: net-score(q,d) = g(d) + cosine(q,d), או בעזרת כל בקומבינציה לינארי אחרת. בהתאם, אנו נחפש את K המסמכים בעלי ה-net score הגבוה ביותר.

K המסמכים הטובים ביותר – שיטה מהירה

נמיין את כל ה-postings לפי g(d) (ולא לפי ה-docID). אם כל מסמך מקבל ערך g(d) ייחודי (כלומר, אין שני מסמכים עם אותו הערך), אנו יכולים לעבור על רשימות ה-postings של הביטויים בשאילתא ובמקביל לבדוק חיתוכים בין הרשימות השונות או לבצע את חישוב ערך ה-cosine. גם אם הערכים של g(d) הם לא ייחודיים, אנו יכולים לבצע את העבודה במקביל, רק שעכשיו, במידה ואנו נתקלים בשני ערכי g(d) זהים, אנו צריכים גם לבדוק אם יש להם docID זהה. אבל למה לעשות זאת ? מה היתרון בצורת עבודה זו ? אם נבצע את המיון בהתאם ל-

oren.e.nahum@gmail.com

g(d), אז סביר שמסמכים בעלי דרוג גבוה יופיעו בתחילת ה-posting list. במקרים שזמן החיפוש מוגבל, זה מבטיח לנו שאנו עוברים על המסמכים בעלי הדרוג הגבוה ראשונים.

אנו יכולים למזג את ה-champion lists עם ערך ה-g(d). במקרים אלו, לכל ביטוי אנו שומרים champion lists של r המסמכים עם ערך tf-idftd+ g(d) הגבוה ביותר. וכן, אנו נחפש את K המסמכים הטובים ביותר רק ברשימת ה-champion list.

High and low lists

לכל term אנו שומרים שתי רשימות (או posting lists) הנקראות High ו-Low. הרשימה הראשונה מכילה מסמכים בעלי חשיבות גבוהה, ואילו הרשימה השנייה מכילה מסמכים בעלי חשיבות נמוכה. ניתן לחשוב על High כ-champion list. כאשר אנו עוברים על ה-posting list במהלך ביצוע השאילתא, אנו נעבור תחילה על רשימת ה-high. אם נקבל מרשימת ה-high K או יותר מסמכים, ניקח את K הטובים ביותר, אחרת, נמשיך עם רשימת ה-low. לבניית הרשימות הללו אפשר להשתמש רק עם ערך ה-cosine, ואין צורך ב-g(d).

Impact-ordered postings

אנו רוצים לחשב את הציון רק עבור מסמכים שיש להם ערך $wf_{t,d}$ גבוה מספיק. אם אנו נמיין את ה-postings בהתאם לערך ה-$wf_{t,d}$, בעת מעבר על ה-postings נוכל לעצור כשנגיע למסמך שערך ה- ה-$wf_{t,d}$ לא גבוה מספיק. אולם יש לנו בעיה בשיטה זו, רשימות ה-postings השונות לא ממוינות באותו הסדר, ולכן לא ניתן לבצע עבודה במקביל. אם כך, איך נחשב את הציונים בכדי לבחור את K המסמכים הטובים ביותר. יש לנו שני רעיונות לשם כך.

כאשר אנו עוברים על רשימת ה-postings של הביטוי t (שממויין לפי $wf_{t,d}$), אנו נעצור (1) כאשר הגענו למספר r, מוגדר מראש, של מסמכים או (2) ערכו של $wf_{t,d}$ יורד מתחת לסף קבוע מראש. אנו לוקחים את האיחוד של קבוצות המסמכים (קבוצה עבור כל ביטוי) ומחשבים את הציון עבור המסמכים באיחוד הנ"ל.

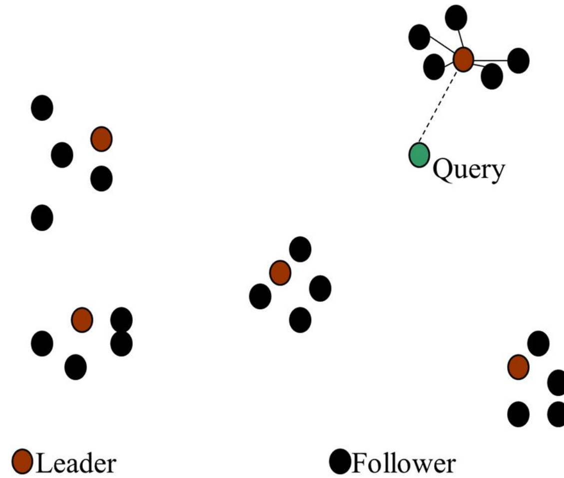
כאשר אנו עוברים על רשימת ה-postings נעשה זה בהתאם לערך ה-idf, מהגבוה לנמוך, כשערך idf גבוה כנראה תורם הרבה לחישוב הציון (למעשה אנו מתחילים עם הביטויים הנדירים יותר). כשאנו מעדכנים את ערך הציון של מסמך, בהתאם לביטויים השונים של השאילתא, אנו נעצור אם השינוי בערך החדש הוא זניח. במקרה זה, אפשר להשתמש בערך ה-cosine או בערך net score אחר.

Cluster pruning

נבחר $\sqrt{N}$ מסמכים באופן אקראי, ונקרא להם מנהיגים (leaders). לכל מסמך אחר, נחשב מראש מיהו המנהיג הקרוב ביותר אליו (למשל ע"י חישוב cosine). מסמך המשויך למנהיג הוא העוקב שלו. לכל מנהיג יש כ-$\sqrt{N}$ עוקבים.

ביצוע השאילתא יתבצע באופן הבא: (1) עבור שאילתא Q, נמצא את המנהיג L הקרוב ביותר אליה. (2) ניקח את K המסמכים הקרובים ביותר ל-L מבין העוקבים שלו.

oren.e.nahum@gmail.com



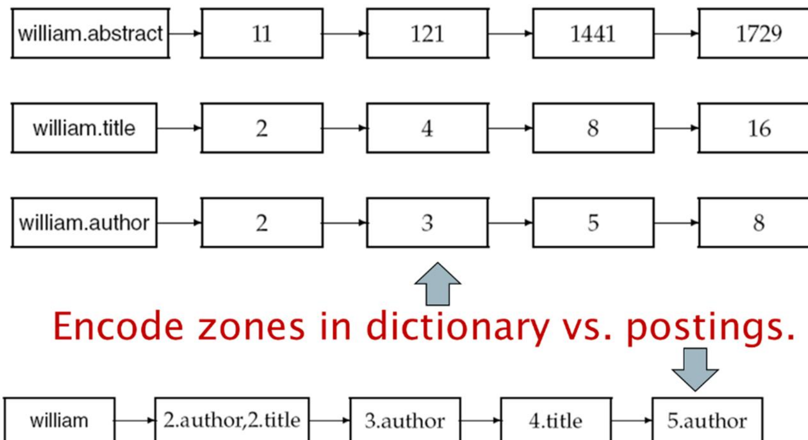
למה אנו בוחרים את המנהיגים באופן אקראי ? יש לכך שתי סיבות: (1) שיטה זו מהירה. (2) באופן זה, המנהיגים משקפים את התפלגות המידע.

אפשר להכליל שיטה זו. לדוגמא, כל מסמך משויך ישויך ל-b1=3 (נניח) מנהיגים קרובים ביותר (כלומר, לכל מסמך אין מנהיג אחד בלבד). באופן דומה, בעת ביצוע שאילתא, עבור השאילתא עצמה, נמצא את b2=4 (נניח) המנהיגים הקרובים ביותר ואת המסמכים המשויכים אליהם. באופן זה, כמות המסמכים שאנו צריכים לבדוק גדולה יותר, אבל ככל הנראה, אנו נגדיל את ה-recall של התוצאה המתקבלת.

<u>אינדקסים פרמטריים ואזוריים</u>

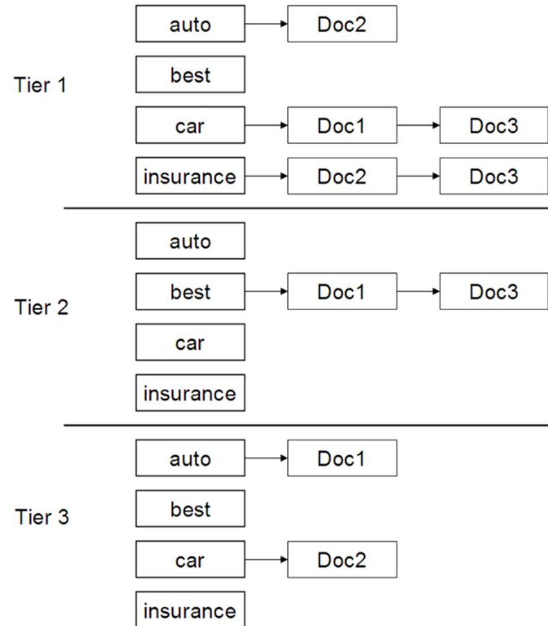
עד כה, התייחסנו למסמך כאוסף של terms. אולם, במציאות, למסמך יש מספר חלקים, להם משמעות סמנטית שונה, כגון שם המחבר, כותרת, תאריך פרסום, שפה, פורמט וכו'. נתונים אלו מרכיבים את המטה-דאטה של המסמך. ישנם מקרים שהם אנו מעוניינים לחפש בהתאם למטה-דאטה. לדוגמא, מצא מסמכים שנכתבו ע"י וויליאם שייקספיר בשנת 1601, המכילים alas poor Yorick. בדוגמא זו, Year=1601 הוא שדה, וכך גם שם משפחת המחבר, שייקספיר, הוא שדה. בכדי לבצע שאילתות על אותו מטה-דאטה, אנו יוצרים אינדקסים על שדות או פרמטרים שבו יש posting עבור כל ערך בשדה.

Zone (אזור) הוא קטע במסמך, כגון כותרת, אבסטרקט או בביליוגרפיה, המכיל כמות כלשהי של טקסט (בניגוד לשדה שבו מס' מילים מועט). אנו נבנה inverted index גם עבור ה-terms הנמצאים ב-zones השונים, על מנת לאפשר חיפוש בהם. לדוגמא, "find docs with merchant in the title zone and matching the query gentle rain"



Tiered indexes

במקום שיהיה לנו posting list אחד, נחלק את ה-postings להירכיה של רשימות, מהחשובה ביותר עד להכי פחות חשובה. בכדי לבנות את ההיררכיה הזאת ניתן להשתמש ב-g(d) או בכל מדד אחר. את ה-inverted index אנו מחלקים לקבוצות בעלי חשיבות פוחתת. בזמן השאילתא, אנו נשתמש בקבוצה ברמה הגבוהה ביותר, אך אם לא נקבל K מסמכים, נרד לרמה נמוכה יותר.

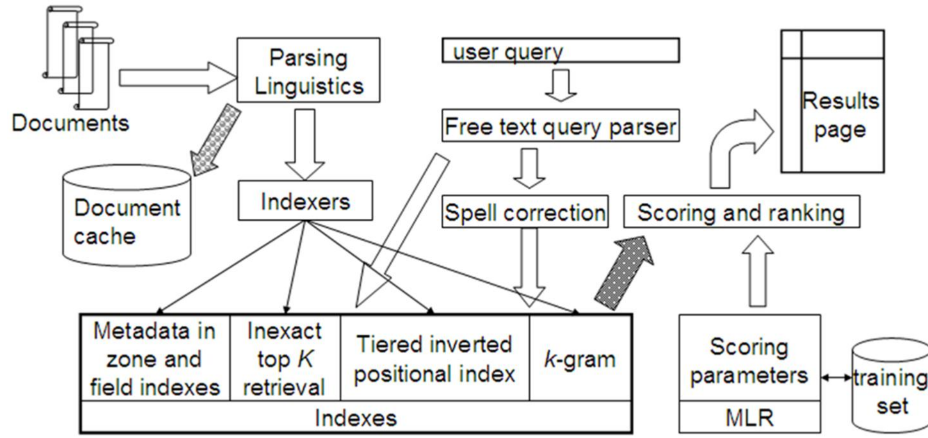


Query Term Proximity and Query Parsing

שאילתות טקסט חופשי הן אוסף של terms המוזנים למנוע החיפוש (נפוץ בחיפוש באינטרנט). המשתמשים מעדיפים מסמכים בהם ה-terms מופעים בקרבה יחסית אחת לשני. נגדיר כ-w את החלון (מס' ה-terms הרצופים) הקטן ביותר במסמך המכיל את כל ביטויי השאילתא. לדוגמא, עבור השאילתא "strained mercy", החלון הקטן ביותר במסמך "The quality of mercy is not strained" הוא 4 (מילים). אנו מעוניינים בפונקציית דרוג שתיקח ערך זה בחשבון.

שאילתות טקסט חופשי יכולות להתפרש כמספר שאילתות. נניח שיש לנו את השאילתא "rising interest rates". בשלב הראשון נבצע את השאילתא כ-phrase query. אם כמות המסמכים המוחזרת קטנה מ-k, נפצל את השאילתא לשתי שאילתות נוספות, לדוגמא, "rising interest" ו-"interest rates". אם עדיין כמות המסמכים המוחזרים קטנה מ-k, נבצע 3 שאילתות נוספות, אחת עבור כל term. נדרג את המסמכים שהתקבלו מהשאילתות לפי vector space scoring.

ראינו שפונקציית הדרוג יכולה לחבר ציונים שונים כגון cosine, static quality, קירבה וכו' (לתת לכל שיטה אחוז כלשהו מהציון הסופי). אולם, כיצד אנו יכולים לדעת מה הצרוף הטוב ביותר ? ובכן, בחלק מהמקרים, המערכת מכוונת ע"י אנשים (כלומר, אנשים עוברים על המערכת, מנסים שיטות שונות, רואים את התוצאות, ובהתאם בוחרים את השיטה שבעזרת מתקבלות התוצאות הטובות ביותר). אפשרות נוספת, שימוש ב-machine learning, שהולך ונהיה נפוץ יותר. אלו הן שיטות ממוחשבות שאפשר ליישם, שבודקות אפשרויות שונות, ובאופן אוטומטי יכולות לבחור את השיטה שתיתן את התוצאה הטובה ביותר.


Documents
Parsing
Linguistics
Document
cache
Indexers
user query
Free text query parser
Spell correction
Scoring and ranking
Results
page
Metadata in
zone and
field indexes
Inexact
top K
retrieval
Tiered inverted
positional index
k-gram
Indexes
Scoring
parameters
MLR
training
set