Notes-Heaven

נושאים מתקדמים במסדי נתונים

הכנה למבחן

דינה זליגר

סמסטר ב'
תשס"ט

WWW.NOTES-HEAVEN.COM

תנאי שימוש

Please read the following important legal information before reading or using these notes. The use of these notes constitutes an agreement to abide by the terms and conditions below, just as if you had signed this agreement.

A. THE SERVICE.

The following notes ("The service") are provided by DinaZil's Notes-Heaven ("Notes-Heaven").

B. DISCLAIMER OF WARRANTIES; LIMITATION OF LIABILITY.

Notes-Heaven does not endorse content, nor warrant the accuracy, completeness, correctness, timeliness or usefulness of any opinions, advice, content, or services provided by the Service.

YOU AGREE THAT USE OF THE SERVICE IS ENTIRELY AT YOUR OWN RISK. THE SERVICE PROVIDED IS PROVIDED "AS IS," WITHOUT WARRANTY OF ANY KIND. NOTES-HEAVEN EXPRESSLY DISCLAIMS ALL WARRANTIES OF ANY KIND, WHETHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION: ANY WARRANTIES CONCERNING THE ACCURACY OR CONTENT OF INFORMATION OR SERVICES. NOTES-HEAVEN MAKES NO WARRANTY THAT THE SERVICE WILL MEET YOUR REQUIREMENTS, OR THAT THE SERVICE WILL BE ERROR FREE; NOR DOES NOTES-HEAVEN MAKE ANY WARRANTY AS TO THE RESULTS THAT MAY BE OBTAINED FROM THE USE OF THE SERVICE OR AS TO THE ACCURACY OR RELIABILITY OF ANY INFORMATION OBTAINED THROUGH THE SERVICE. YOU UNDERSTAND AND AGREE THAT ANY DATA OBTAINED THROUGH THE USE OF THE SERVICE IS DONE AT YOUR OWN DISCRETION AND RISK AND THAT YOU WILL BE SOLELY RESPONSIBLE FOR ANY DAMAGE TO YOUR GPA.

NEITHER NOTES-HEAVEN NOR ANY OF ITS PARTNERS, AGENTS, AFFILIATES OR CONTENT PROVIDERS SHALL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF USE OF THE SERVICE OR INABILITY TO GAIN ACCESS TO OR USE THE SERVICE OR OUT OF ANY BREACH OF ANY WARRANTY.

C. INDEMNIFICATION.

You agree to indemnify and hold Notes-Heaven, its partners, agents, affiliates and content partners harmless from any dispute which may arise from a breach of terms of this Agreement. You agree to hold Notes-Heaven harmless from any claims and expenses, including reasonable attorney's fees and court costs, related to your violation of this Agreement.

D. OWNERSHIP RIGHTS.

The materials provided by the Service may be downloaded or reprinted for personal use only. You acknowledge that the Service contains information that is protected by copyrights, trademarks, trade secrets or other proprietary rights, and that these rights are valid and protected in all forms, media and technologies existing now or hereafter developed. You may not modify, publish, transmit, participate in the transfer or sale, create derivative works, or in any way exploit, any of the Content, in whole or in part. You may not upload, post, reproduce or distribute Content protected by copyright, or other proprietary right, without obtaining permission of the owner of the copyright or other proprietary right.

E. NO COPYING OR DISTRIBUTION.

You may not reproduce, copy or redistribute the design or layout of this service, individual elements of the design, Notes-Heaven logos or other logos appearing on this service, without the express written permission of Notes-Heaven, Inc. Reproduction, copying or redistribution for commercial purposes of the service is strictly prohibited without the express written permission of Notes-Heaven, Inc.

If you have any questions about this statement or the practices of this service you can contact

Dina Zeliger

dinazil at notes-heaven.com

תוכן עניינים

תנאי שימוש ... 1

תוכן עניינים ... 1

חלק א: רקע ... 7

איך האינטרנט עובד? ... 7
תשתית ... 7
כתובות IP ו-port-ים ... 7
Domain Name Servers ... 7
התקשורת ... 8
משאבים ... 8
Universal Resource Locator ... 8
קישורים יחסיים ... 9
הערות ... 9
דפים דינאמיים ... 10
Web Server Side ... 10
Client Side ... 10
Hyper Text Transfer Protocol ... 10
שיחת HTTP טיפוסית ... 10
פורמט הודעות HTTP ... 11
בקשות HTTP ... 11
תגובות HTTP ... 12
מטמון ו-Proxy Servers ... 12
Cookies ... 13
מאפיינים של קוקיות ... 13
אנטומיה של קוקיות ... 14
שימוש בקוקיות לניהול שיח ... 14
חסרונות בשימוש בקוקיות ... 15
גניבת קוקיות ... 15
הרעלת קוקיות ... 15
קוקיות צד ג' ... 15
אבטחת מידע ביישומי רשת ... 16
אבטחה ב-HTTP ... 16
Basic Access Authentication ... 16
Digest Access Authentication ... 17

HTTP/1.1 על קצה המזלג 17

שימושיות האינטרנט 18
עקרונות השימושיות 18
כללי עיצוב 18
עיצוב דפים 18
עיצוב עמוד הבית 18
עיצוב האתר 18
נגישות 18

חלק ב: מנועי חיפוש 19

אתגרים בבניית מנוע חיפוש 19

מבנה של מנוע חיפוש 19

האינדקס - איך מאחסנים כמויות גדולות של מידע? 20
רעיונות גרועים 20
שמירה כקבצים 20
שמירה במסד נתונים רלציוני 20
Bitmaps 22
רעיונות טובים יותר 23
מבנה האינדקס 23
הלקסיקון 24
מחרוזות שוות ארוך 24
מחרוזות משורשרות 25
Front Coding 25
3-in-4 Front Coding 26
סיכום 27
האינדקס המהופך 27
רשימות הפרשים 27
דחיסה 27
שמירת מבנה הרשת 35
תכונות מעניינות של קישורים 35
ייצוג נאיבי 35
רשימת הפרשים 35
דחיסת הפניות 36
דחיסה דיפרנציאלית 36
דחיסת רשימת הצמתים החסרים 37
בניית האינדקס 37

אינדקסים עבור שאילתות עם תווי-כל 37
יצירת אינדקס בעזרת n-grams 38
יצירת אינדקס בעזרת לקסיקון מסובב 39
אינדקס לצירופי מילים 40

עיבוד מקדים 41
Case Folding 41
Stemming 41
Stop Words 41
תיקון שגיאות כתיב 42
תיקון מבודד 42
מרחק עריכה 42
חפיפת n-grams 43

עיבוד שאילתות 44
שאילתות גימום 44
גישה יעילה לרשימות משורשרות 45
נקודות סנכרון 45
שאילתות לא קוניוקטיביות 46

Crawling 46
אלגוריתם בסיסי 47
ניהול התור 47
מחסנית - חיפוש לעומק 47
תור - חיפוש לרוחב 48
תור קדימויות 48
ארכיטקטורה של רובוט חיפוש 49
רובוטים צריכים להיות ממושמעים 49
בעיית ה-DUST 50
מציאת שכפולים מדויקים 51

דירוג 52
הגדרת איכות תוצאות חיפוש 52
שיטות דירוג 53
שיטות מותאמות לשאילתא 53
TF-IDF ומודל המרחב הווקטורי 53
דירוג מבוסס HTML 55
שיטות דירוג גלובאליות - ניתוח קישורים 56
גישה נאיבית 56
גולש אקראי - PageRank 56

PageRank מותאם נושא ... 59
איך לרמות מנוע חיפוש ... 59
Click-Through Ranking: DirectHit ... 59

חלק ג: XML ... 61

מוטיבציה ... 61
הרשת הסמנטית ... 61
החלפת מידע ... 62
הפרדת תוכן מעיצוב ... 62

תחביר של XML ... 63
השוואה בין XML ל-HTML ... 63
כללי תחביר של XML ... 63
עץ XML ... 64
אטריבוטים ... 65
XML לעומת טבלאות ... 65
אכיפת מבנה על מסמכי XML ... 66
למה לאכוף מבנה? ... 66
הגדרה ודוגמאות ... 66
הגדרת אטריבוטים ... 67
1-חד-משמעיות ... 68
אוטומט גלושקוב ... 69
חסרונות של DTD ... 71
אזהרות ... 71

שליפת מידע מ-XML ... 72

אחסון ... 72

חישוב שאילתות ... 72

שאילתות מילות מפתח ... 72

נספח א - הבדלים בין HTTP/1.0 ו-HTTP/1.1 ... 73

נספח ב - TRIES ... 92

נספח ג - טיפול בבעיית ה-DUST ... 95

נספח ד – אלגוריתם הדירוג HILLTOP 96

ביבליוגרפיה... 98

חלק א: רקע

איך האינטרנט עובד?

תשתית

כתובות IP ו-port-ים

כדי להבין איך עובד האינטרנט נעקוב אחרי מה שקורה כאשר לוחצים על קישור בדפדפן. דפים באינטרנט כתובים ב-Hyper Text Markup Language (או HTML) אשר מגדירה איך הדף נראה. בעזרת HTML אפשר לשלוט על העיצוב של הדף – איך התוכן שלו יהיה מסודר על המסך. כאשר לוחצים על קישור, הקוד שמסתתר מאחוריו ב-HTML נראה בערך כך:

```
<a href="http://www.notes-heaven.com/
SemesterB/67830_AdvancedTopicsInDatabaseTheory/index.html">solutions
to the homework in ATDB</a>
```

הסינטקס לא מאוד חשוב כרגע אבל יש להבין כאן שני דברים בדף אנחנו רואים את הטקסט " solutions to the homework in ATDB " והוא למעשה מצביע לכתובת האינטרנט http://www.notes-heaven.com/SemesterB/67830_AdvancedTopicsInDatabaseTheory/index.html.
הדפדפן יודע שזו הכתובת שהוא צריך לגשת אליה. אז הדפדפן שולח בקשה לשרת. השרת בתגובה ניגש למערכת הקבצים שלו ושולח בחזרה את הדף המבוקש. הדפדפן אז מציג את הנתונים שהגיעו (למשל אם מדובר בדף HTML אז הוא מציג אותו לפי הסגנון שמוגדר בו).

אלא שהאתר נמצא על שרת מרוחק וכדי לגשת אליו יש לדעת את כתובת ה-IP שלו ואת ה-port שעליו הוא מאזין (במקרה של דפי אינטרנט מדובר בד"כ ב-port 80 וזוהי ברירת המחדל שהדפדפן משתמש בה). אבל הכתובת שנתונה בדף שלנו אינה נתונה באמצעות IP ו-port אלא באמצעות השם המילולי של האתר. החלק הראשון בכתובת www.notes-heaven.com נקרא Domain Name. אינטואיטיבית, Domain Name הוא כמו שם של בן אדם, IP הוא כמו כתובת הבניין שהוא גר בו ו-port הוא כמו מספר הדירה. לא מספיק לדעת את השם של הבן אדם כדי להגיע אליו הביתה – צריך לדעת גם את הכתובת המלאה שלו.

Domain Name Servers

שרתים מיוחדים שנקראים Domain Name Servers (או בקיצור DNS) יודעים לקבל כתובת מילולית ולקשר אותה לכתובת ה-IP של השרת הרלוונטי. לכל DNS יש רשימה של Domain Names וה-IP-ים המתאימים וכן הוא מכיר כתובות של DNS-ים אחרים שאליהם הוא יכול לגשת במקרה שהתקבלה בקשה שהוא לא יודע את התשובה אליה. בכל מקרה התהליך הזה שקוף מבחינת המשתמש. הוא מדבר רק עם DNS אחד אשר דואג לעשות את כל הבירורים האחרים בעצמו מבלי ליידע את המשתמש עלכך.

הדפדפן יודע כתובות IP של כמה שרתי DNS ולכן הוא יכול לגשת אליהם ולבקש את כתובת ה-IP של www.notes-heaven.com. לאחר שכתובת ה-IP ידועה הדפדפן יכול לשלוח בקשה לשרת שעליו יושב האתר ולבקש את הדף הרלוונטי.

התקשורת

כל המחשבים המעורבים מעבירים ביניהם מידע בחבילות (packets). כל חבילה מכילה את כתובת היעד (כדי שנתבים בדרך יידעו לאן להעביר אותה), כתובת המקור (כדי שניתן יהיה לדעת למי לענות במידת הצורך) ואת תוכן הבקשה (בשפה ששני המחשבים צריכים להבין – נדבר על זה בהמשך). הפרוטוקול שמתאר את העברת החבילות ברשת נקרא Internet Protocol (IP).

הפרוטוקול שמתאר את יצירת החבילות וקבלתן נקרא Transmission Control Protocol (TCP). הוא מבטיח העברה אמינה של נתונים (סדורים) בין שתי תחנות ברשת מחשבים. הוא מעביר את הנתונים שהועברו בעזרת IP, מוודא את נכונותם ומאשר את קבלת הנתונים במלואם או מבקש שליחה מחדש של נתונים שלא הגיעו בצורה תקינה.

משאבים

משאב (resource) הוא פיסת מידע שניתנת לזיהוי יחיד באמצעות Universal Resource Locator (URL) – אותן כתובות שמופיעות בקישורים בדפי HTML. משאב יכול להיות קובץ סטטי (HTML, טקסט, תמונה) או דף נוצר דינאמית.

מה שאנחנו רואים בדפדפן יכול להיות צירוף של כמה משאבים בו זמנים (למשל דף ובו תמונות או סרט).

כאשר דפדפן מקבל דף ובו תמונות, בקוד ה-HTML כתובים ה-URL-ים של כל המשאבים שיש להציג.

```
<td width="200" align="center" valign="top">
  <p>
    <img src="Pictures/IMG_1481.JPG" width="200" height="150" />
    <br />
    <em>
      <strong>Grilled chicken breast on baked apples with cranberry
sauce</strong>
    </em>
  </p>
  <p>
    <img src="Pictures/IMG_1490.JPG" width="200" height="150" />
    <br />
    <em>
      <strong>Strawberry tart</strong>
    </em>
  </p>
</td>
```

כאשר הדפדפן מפרש את ה-HTML הוא רואה שעליו להביא משאב נוסף. אז הוא מבצע את כל התהליך שדיברנו עליו קודם.

Universal Resource Locator

הפורמט הבסיסי של URL הוא

protocol://domain/path

למשל:

- http://www.notes-heaven.com/SemesterB/67830_AdvancedTopicsInDatabaseTheory/index.html
- ftp://ftp.netscape.com/robots.txt

ל-URL יכולים גם להיות עוגן (anchor) אשר מאפשר להגדיר קישור לאמצע העמוד (במקום חלקו העליון) ופרמטרים שהם ערכים נוספים שמועברים לשרת יחד עם ה-path ומאפשרים למשל ליצור דפים דינאמיים.

המבנה המלא של URL הוא אם כן

protocol:://domain/path#anchor?parameters

כאשר הפרמטרים מועברים כרשימה של זוגות. דוגמאות:

- http://en.wikipedia.org/wiki/Anchor_text#Overview
- http://www.google.com/search?hl=en&q=notes&aq=f&oq=&aqi=g10

נשים לב שבמקרה של עוגן המשאב המוחזר זהה לחלוטין למשאב שהיה מוחזר אילו לא היה שימוש בעוגן העוגן רק מאפשר לדפדפן להציג את העמוד החל מהאמצע שלו ואילו במקרה של שימוש בפרמטרים יכול להיות מוחזר משאב שונה – למשל השפה יכולה להשתנות, או במקרה של מנוע חיפוש – מוחזרות תוצאות שונות.

קישורים יחסיים

URL בדף HTML יכול להיות כתוב רק באמצעות ה-path וללא שימוש ב-Domain Name או הפרוטוקול. הדפדפן במקרה זה יודע להשלים את הפרטים הדרושים לפי הכתובת של הדף הנוכחי: הכתובת היא ביחס לתיקייה הנוכחית. לכן, אם נלחץ על הלינק `<a href="robes.html">buy robes</a>` נגיע למקומות שונים כתלות במקום שבו התחלנו:

- http://www.abc.com/shop/index.html → http://www.abc.com/shop/robes.html
- http://www.abc.com/shop → http://www.abc.com/shop/robes.html
- http://www.abc.com/ → http://www.abc.com/robes.html

הערות

נעיר לסיכום רק ש-URL קובע באופן יחיד משאב. בהינתן URL יש רק משאב אחד שמתאים לו. הכיוון ההפוך אינו נכון. יכולים להיות כמה ה-URL-ים שמצביעים לאותו המשאב. למשל, ישנם אתרי מראה שמכילים עותקים זהים של אתרים אחרים. אך יש גם דוגמאות פתולוגיות יותר:

- http://mail.google.com, http://www.google.com/mail, http://www.gmail.com
- http://www.notes-heaven.com, http://www.notes-heaven.com/, http://www.notes-heaven.com/index.html

דפים דינאמיים

יש שתי דרכים ליצור דף דינאמי. הראשונה באמצעות השרת שמגיב לבקשות של הלקוח, והשנייה באמצעות הדפדפן שמשנה את המשאב לפי שהוא מציג אותו.

Web Server Side

עד עכשיו הנחנו שהשרת מחזיר קובץ ממערכת הקבצים שלו. אבל זה לא תמיד המצב וגם ברור שלא כל דף באינטרנט יכול להיות ממומש בצורה זו. למשל, תוצאות חיפוש של מנוע חיפוש לא יכולות להיות שמורות בשרת. השרת הרי לא יכול לחזות מראש את כל השאילתות האפשריות ואף אילו זה היה אפשרי הרי לא היה מקום לשמור את כל הדפים השונים. שרת יכול להריץ תוכנה עם הפרמטרים שהועברו ב-URL ואז להחזיר תוצאה בהתאם לריצה. דפים כאלה הם דפים דינאמיים – הם נוצרים לפי בקשת הלקוח. אז ברצף הפעולות שבגישה למשאב כלשהו, במקום גישה למערכת הקבצים, השרת מריץ תוכנה ואז הוא מחזיר את התוצאה של הריצה.

כלים נפוצים לבניית דפים דינאמיים מצד השרת הם CGI, Java Servlets, ASP, PHP ועוד.

Client Side

חלקים מאפליקציות אינטרנט יכולים להתבצע מקומית בתוך הדפדפן. למשל, ניתן לבצע כל מיני אימותים מקומיים על הקלט לפני שהוא נשלח לשרת. למשל אפשר לשלב בתוך דף HTML שפות סקריפט כמו JavaScript, VBScript, AJAX ועוד.

הרצת קוד אצל הלקוח חוסכת זמן ומשאבים לשרת אך לא תמיד הדבר אפשרי. בין השאר, הדבר כרוך במתן התוכנית שצריכה לרוץ ללקוח, ובוודאי זה לא דבר שנותני השירות רוצים לעשות בהכרח. יתר על כן, לא תמיד יש אפשרות להריץ את התכונה בצד הלקוח. דוגמה טובה לכך היא מנוע חיפוש – ללקוח בוודאי אין גישה למאגרי המידע של המנוע ואילו הוא היה ניגש אליהם בכל זאת זה לא באמת היה חוסך משאבים לשרת

מאידך יש מקרים שבהם לא ניתן להריץ את התכונה בצד השרת. למשל, אם הלקוח מתבקש להכניס שם משתמש וסיסמה, הם בוודאי לא נשלחים כפי שהם ברשת. אז צד הלקוח אחראי על ההצפנה של הנתונים.

כמובן, ניתן גם לשלב ולבנות דפים דינאמיים גם מצד השרת וגם מצד הלקוח.

Hyper Text Transfer Protocol

כפי שכבר אמרנו קודם, כדי שהמחשבים ברשת יוכלו להעביר מידע ביניהם, הם צריכים לדבר בשפה משותפת – שפה ששניהם מבינים. השפה הזאת היא הפרוטוקול שמופיע ב-URL. אנחנו נדבר על Hyper Text Markup Language (או בקיצור HTTP) שהוא הפרוטוקול שנועד להעביר דפי HTML והאובייקטים שכם מכילים (תמונות, קובצי קול, סרטוני flash וכו').

הדפדפן מכיר את הפרוטוקול והוא משתמש בו כדי לשלוח בקשות לשרת אשר מאזין לבקשות ויודע לענות להן באותו הפרוטוקול. יש שני סטנדרטים של HTTP – HTTP/1.0 ו-HTTP/1.1. אנחנו נדון ב-HTTP/1.0.

שיחת HTTP טיפוסית

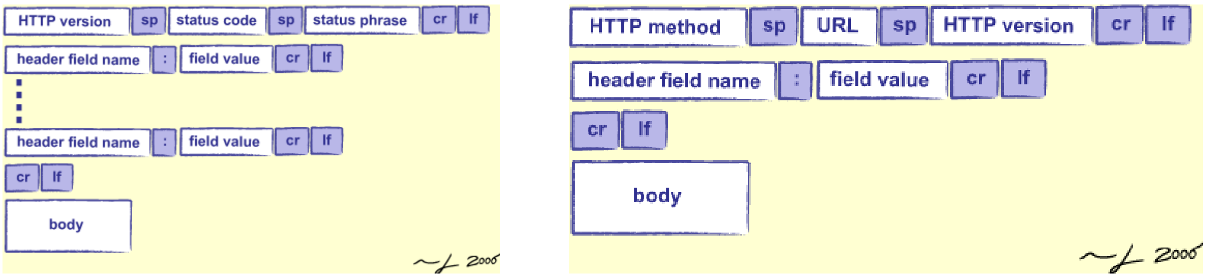
שיחת HTTP בסיסית כוללת ארבעה שלבים:

1. הלקוח פותח ערוץ תקשורת (TCP)
2. הלקוח שולח בקשה

3. השרת שולח תגובה
4. השרת סוגר את ערוץ התקשורת

HTTP הוא פרוטוקול חסר זיכרון (stateless). ברגע שהשרת העביר את המידע המבוקש הוא לא זוכר את כל מה שקרה לפני כן. אם יש צורך בזיכרון, יש להשתמש בכלי תכנות בצד השרת שידאגו לניהול מצב השרת

פורמט הודעות HTTP



איור 1 בקשת HTTP (מימין) ותגובת HTTP (משמאל)

בקשות HTTP

הבקשה מתחילה ב-HTTP method שלמעשה מצהירה מהו סוג הבקשה. בקשות נפוצות הן:

- GET – מיועדת לקבלת אובייקט שנמצא על השרת, בכתובת שניתנת בתחילת ההודעה. בקשות GET הן הנפוצות ביותר ברשת האינטרנט.
- HEAD – מבקשת מהשרת לשלוח את כל שדות הכותרת שהיו נשלחים לבקשת GET, אך בלי האובייקט עצמו. השיטה נועדה, בין השאר, לאפשר בדיקה של קישורים שבורים או זמני שינויים של אובייקטים מבלי לבקש את כל האובייקט.
- POST – בקשות המכילות גוף הודעה. בקשות POST משמשות בדרך כלל לשליחה של נתונים מטפסי HTML לשרת לשם עיבוד.
- PUT – מבקשת מהשרת לשמור את גוף ההודעה המצורף לבקשה בתור אובייקט, שכתובתו היא הכתובת שניתנה בתחילת הבקשה.
- DELETE – מבקשת מהשרת למחוק את האובייקט שכתובתו מצוינת בתחילת הבקשה.

שתי השיטות האחרונות נחסמות בד"כ ע' השרת.

דוגמאות ל-headers נפוצים:

- Accept-Encoding – אומר מהם הקידודים שהדפדפן מבין
- Cookie – קוקית שנשלחה קודם לכן ע"י השרת
- If-Modified-Since – מאפשר לשרת להודיע שהמשאב לא השתנה ולכן תוכנו לא נשלח מחדש
- User-Agent – שם התוכנה בצד הלקוח ששולחת את הבקשה
- Content-Length – מספר הבתים בגוף ההודעה (ארוך המשאב עצמו)
- Referer – הכתובת של האתר שממנו נלחץ הקישור

תגובות HTTP

המבנה של התגובה דומה לזה של הבקשה, אך הוא גם כולל קוד מצב שנותן לנו מידע על ביצוע הבקשה:

קוד המצב מורכב ממספר תלת ספרתי והסבר קצר (בן שורה אחת) על משמעות הקוד. המספרים מחולקים ל-5 מחלקות, בהתאם לספרה הראשונה שלהם:

- 1xx - ההודעה מכילה מידע אינפורמטיבי בלבד.
- 2xx - הבקשה ששלח הלקוח בוצעה בהצלחה על ידי השרת, והתשובה מכילה את מה שהשרת נתבקש לשלוח בהתאם לשיטת הבקשה.
- 3xx - הבקשה פוענחה בהצלחה, אך מסיבות כלשהן התשובה אינה כוללת את האובייקט המבוקש (למשל, משום שהעמוד מפנה אוטומטית לעמוד אחר).
- 4xx - נמצאה שגיאה כלשהי בבקשה עצמה.
- 5xx - השרת לא הצליח למלא אחר הבקשה כתוצאה מכשל פנימי.

מטמון ו-Proxy Servers

עד עכשיו תיארנו את תהליך הבאת משאב באופן הבא:

1. הלקוח שולח בקשה למשאב ישירות לשרת האינטרנט
2. השרת מחזיר את המשאב ללקוח

אבל אז פירוש הדבר הוא שבכל פעם שמישהו עושה חיפוש ב-Google, נשלחת בקשה ישירות לשרת שמבקשת את הלוגו שלהם. זה בזבוז גדול של זמן ומשאבים. התמונה לא משתנה בין חיפוש לחיפוש ולכן לא כדאי להביא אותה כל פעם מחדש. לא רק שמספיק לקבל מהשרת הודעה שהתמונה אל השתנתה ולחסוך ברוחב פס, אלא שלא כדאי אפילו לבקש אותה מלכתחילה. לכן, כדי לחסוך בתעבורה ולשפר זמני ריצה, אפשר לשמור משאבים במטמון (cache) בשני מקומות:

- מקומית אצל הלקוח (דפדפנים מאפשרים לשלוט על הגודל של המטמון)
- בנקודות ציון בדרך מהשרת אל הלקוח, בשרתים ייעודיים שנקראים Proxy Servers.

Proxy (בעברית בא-כוח) זהו שרת שתפקידו לספק גישה מהירה למשאבים חיצוניים ברשת מחשבים הדבר מתאפשר באמצעות התקנת תוכנה על השרת שתפקידה לשמור דפי אינטרנט בזיכרון המטמון ולאפשר בכך גישה מהירה אליהם מהמחשבים המחוברים לפרוקסי. בחלק מהמקרים הפרוקסי משמש נקודת הפרדה בין הארגון לעולם החיצוני ומאפשר מעקב, חסימה או שינוי לדפי HTML על פי קריטריונים שונים.

Proxy הוא תוכנה שיכולה עקרונית לעשות כל דבר. השימוש הנפוץ הוא קבלה ושמירת בקשות:

1. השרת מאזין ל-port מסוים
2. בלולאה אינסופית:
   a. יוצר ערוץ תקשורת עם מבקש הבקשה
   b. קריאת בקשה
   c. עיבוד הבקשה (בדיקה במטמון, בדיקת תוכן וכו')
   d. שליחת תשובה

הדפדפן אז לא ניגש ישירות לשרת שעליו נמצא המשאב אלא לתוכנת ה-Proxy Server שתפקידה לגשת לשרת במידת הצורך. מאחר שאותו Proxy נמצא בשימוש של הרבה לקוחות השימוש במטמון ברמה זו יכול להיות יעיל מאוד. ברגע שמשאב נמצא כבר במטמון ב-proxy, בפעם הבאה שלקוח (כלשהו) יבקש משאב זה, לא יהיה צורך לגשת לשרת עצמו ומספיק יהיה להחזר את המשאב ששמור ב-proxy. כמו ב-DNS, גם בשרתי proxy יש היררכיה. אם ה-proxy שניגשו אליו לא מכיל את המשאב הוא ניגש ל-proxy גדול יותר וכן הלאה עד שבלית ברירה ניגשים לשרת שעליו נמצא המשאב

שרתי proxy מורידים את ה-latency של דפדפן אם הוא מביא את המשאב מהמטמון שלו. כתוצאה מזה כמות התעבורה קטנה ויורד קצת עומס מהשרת שעליו נמצא המשאב ולכן, גם ה-latency של בקשות שבכל זאת נשלחות לשרת יורד.

אם כך, היתרונות בשימוש ב-Proxy Server הן ברורות. אבל יש גם חיסרון גדול בשימוש במטמון – המידע שאנחנו מקבלים עלול להיות לא מעודכן. כדי להתגבר על הבעיה אפשר להשתמש ב- If-Modified-Since Header ולדאוג שנקבל מידע עדכני.

Cookies

נזכור ש-HTTP הוא פרוטוקול חסר זיכרון. קוקיות[1] (cookies) יכולות לשמש כפיתרון פשוט לבעיה זו. הקוקיות משמשות את שרתי דפי האינטרנט, הן מאפשרות להבדיל בין המשתמשים באתר האינטרנט, ולאגירת מידע רלוונטי המתייחס למשתמש במהלך ביקורו באתר, ולעתים גם בביקורים עתידיים.

קוקיות מאפשרות לאתר קניות לשמור את עגלת הקניות של הלקוח בזמן שהוא מדפדף באתר. שימוש נוסף בעוגיות הוא האפשרות להתחבר לחשבון המשתמש באתר. העוגיות מאפשרות לשרת לברר האם המשתמש נרשם בעבר וכך לאפשר לו להירשם בצורה אוטומטית ולאפשר לו גישה לשירותים או פעולות שניתנות רק למשתמשים רשומים. דוגמה נפוצה לכך היא שירות דואר אלקטרוני כגון Gmail: לאחר שהמשתמש הזדהה בטופס הכניסה לשירות, נשמרים שם המשתמש שלו וסיסמתו בקוקית. מרגע זה, גם אם יעבור לאתר אחר ויחזור לשירות הדואר האלקטרוני במועד מאוחר יותר, לא יצטרך להירשם מחדש, שכן באמצעות הקוקית שנוצרה בטופס הכניסה לשירות יזהה השרת את המשתמש באופן מיידי.

מספר אתרי אינטרנט משתמשים בעוגיות למטרת התאמה אישית של תצוגת האתר על פי העדפות המשתמש. אתרים הדורשים אימות מרבים להשתמש בתכונה זו, אולם היא קיימת גם באתרים שאין בהם חיוב להזדהות. לדוגמה, אתר האינטרנט Wikipedia מאפשר למשתמשים שנרשמו לבחור את ממשק המשתמש (skin) האהוב עליהם. מנוע החיפוש Google מאפשר (גם למשתמשים לא רשומים) לבחור כמה תוצאות יוצגו בכל דף.

השימוש בעוגיות יעיל כאשר המשתמש נוהג לגשת לאתר מסוים רק ממחשב אחד. במצב כזה, במקום להזין מחדש את פרטי ההזדהות, פרטי כרטיס האשראי וכו' עליו לעשות זאת פעם אחת בלבד, ומאותו הרגע יודע הדפדפן לשלוח את המידע הזה לשרת באופן אוטומטי.

מאפיינים של קוקיות

לקוקית יכולים להיות כמה מאפיינים:

- שם וערך (אלה חייבים להיות מוגדרים, אחרת אין לקוקית משמעות)

[1] האקדמיה ללשון העברית, מילון: מידע – 35: רישות (תשס"ד)

- תאריך תפוגה (שאחריו אין להשתמש בקוקית). אם תאריך התפוגה אינו מופיע ברירת המחדל היא שהקוקית תקפה כל עוד הדפדפן פתוח.
- נתיב – הנתיבים שעבורם הקוקית תקפה. אם לא ניתן נתיב אז ברירת המחדל היא שימוש בנתיב של ה-URL המבוקש.
- מידע לגבי חיבורים מקודדים – אם מותר לשלוח את הקוקית רק דרך ערוצים מאובטחים

אנטומיה של קוקיות

כאשר שרת רוצה ליצור קוקית אצל הלקוח הוא משתמש ב-Set-Cookie header של הודעת ה-HTTP שלו.

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: name=VALUE; expires=DATE, path=Path; (secure)
```

בשורה זו השרת מבקש מהדפדפן לשמור את המחרוזת `name=VALUE`. אם הדפדפן מתיר את השימוש בקוקיות, הקוקית תישלח אוטומטית בכל פעם מחדש כאשר הדפדפן מבקש משרת זה דף חדש. למשל, אם הבקשה הבאה תהיה הדף http://www.w3.org/spec.html, אזי הבקשה הבאה תישלח לשרת www.w3.org:

```
GET /spec.html HTTP/1.0
Host: www.w3.org
Cookie: name=VALUE
Accept: */*
```

השרת יכול לשלוח למשתמש כמה קוקיות ביחד ואז המשתמש מצופה להחזיר את כולם לשרת בכל פנייה. השרת גם יכול לדרוס קוקיות קודמות ע"י שליחת קוקית אם אותו השם ואותו הנתיב. באופן זה שרת יכול למחוק קוקית ע"י שליחת קוקית עם שם ונתיב של קוקית קיימת אך תאריך שתוקפו פג

שימוש בקוקיות לניהול שיח

HTTP הוא פרוטוקול חסר זיכרון. כלומר, כל בקשה לשרת מטופלת באופן בלתי תלוי בקודמות. בלי תמיכה חיצונית לא ניתן לדעת אם בקשה מסוימת היא חלק משיחה מתמשכת בין השרת ללקוח. ובכל זאת, יש אפליקציות רשת שכן מנהלות מצבים – חנויות מקוונות ששומרות עגלת קניות, שירותי דוא"ל שזוכרים את המשתמש ועוד.

הפתרון לבעיה הוא פשוט: השאת והלקוח מעבירים ביניהם מידע ייחודי כלשהו לכל אורך השיח(session). התהליך כמובן צריך להיות שקוף מבחינת משתמשי הקצה ובוודאי היינו רוצים לדאוג שיהיה יעיל ככל הניתן (למשל, לא נראה סביר לשלוח את כל תוכן עגלת הקניות בכל הודעה שעוברת מהשרת ללקוח והפוך).

שרת שתומך בשיח ארוך טווח צריך לשמור במבנה נתונים פנימי כלשהו מידע ייחודי לכל שיח עם כל משתמש. למשל, השרת יכול לשמוק את עגלת הקניות של כל משתמש. כשמגיעה הבקשה הראשונה, השרת מאתחל את המידע של השיח ושולח למשתמש מזהה ייחודי למידע זה. במהלך השיח, הלקוח מוסיף מזהה זה לכל בקשה שהוא שולח לשרת. כך השרת יכול לזהות את הבקשות ולדעת שהן חלק משיח מתמשך.

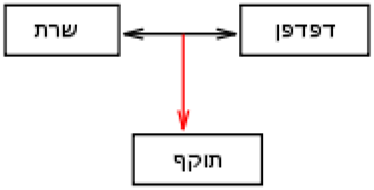
את המזהה של השיחה ניתן להעביר באמצעות קוקיות. בתגובה לבקשה הראשונה בשיחה, השרת שולח קוקית שמכילה את מזהה השיח. כל עוד השיח נמשך, כאשר הלקוח שולח בקשות, הוא שולח את הקוקית בחזרה. גם השרת בודק בצד שלו אם השיח עדיין נמשך ואם כן הוא מתייחס לקוקית כתקפה הקוקיות שמשמשות למטרה זו נקראות קוקיות שיח. הן למעשה קוקיות רגילות אלא שאין להן תאריך תפוגה ספציפי; אלא הן מיועדות לכל אורך השיח. השיח נמשך כל עוד אף אחד מהצדדים לא החליט שהוא נגמר. בצד השרת השיח יכול להיגמר בגלל שהלקוח סיים אותו באופן מפורש(למשל, logout מהחשבון) או בגלל שהשיח לא

היה פעיל זמן רב מדי (למשל, כמו שב-Moodle הדפדפן "שוכח" את המשתמש אחרי זמן מה). בצד הלקוח השיח נמשך כל עוד הדפדפן (באותו התהליך) פתוח.

המידע עצמו שקשור לשיח נשמר בשרת ואילו הקוקית רק שומרת את המזהה של השיח(כלומר, במקרה של חנות מקוונת, עגלת הקניות לא שמורה בקוקית ולא עוברת ברשת כל הזמן, אלא נמצאת רק במבנה נתונים של השרת).

חסרונות בשימוש בקוקיות

מאז שהוצגו לראשונה באינטרנט, הופצו באמצעות האינטרנט והתקשורת מספר תפיסות מוטעות בכל הנוגע לקוקיות. למעשה, קוקיות כוללות בתוכן רק נתונים, ולא קוד תוכנית, והן לא יכולות להסיר או לקרוא מידע שנמצא במחשב המשתמש.



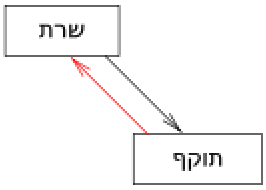
בכל זאת, יש בשימוש הקוקיות פגם אינהרנטי. לכל דפדפן יש אזור אחסון משלו, ולכן אם במחשב נתון יש יותר מדפדפן אחד, ייתכנו מספר קוקיות זהות – אחת עבור כל דפדפן. בנוסף, הקוקיות אינן מסוגלות להבדיל בין משתמשים שונים המשתמשים באותו מחשב ודפדפן, ובאותו חשבון משתמש.

מלבד זה, ניתן להשתמש לרעה בקוקיות בכמה דרכים.

גניבת קוקיות

הקוקיות נשלחות פעמים רבות במהלך אורך חייהן אל השרת וחזרה אל דפדפן המשתמש. הגישה אל קוקיות אלו אמורה להיות מוגבלת שכן קוקיות עשויות לכלול מידע אישי ורגיש, כגון שם משתמש וסימן המשמש לאימות. גניבת קוקיות מתרחשת כאשר צד לא מורשה מקבל לידיו קוקית.

דרך אחת לגנוב קוקית היא באמצעות רחרחן חבילות מידע (packet sniffer). ניתן לקרוא את התעבורה ברשת באמצעות מחשבים אחרים שנמצאים באותה רשת מחשבים, גם אם הם אינם מעורבים ישירות בשליחת וקבלת המידע. בתעבורה זו כלולות גם קוקיות שנשלחות לפרוטוקול HTTP. משתמשים במחשבים אלו יכולים לקרוא את המידע העובר ברשת, כולל את הקוקיות, באמצעות תוכנות הקרויות רחרחני חבילות מידע. ניתן להתגבר על הבעיה על ידי שימוש ב-URI ובפרוטוקול אבטחת שיגור היפרטקסט (HTTPS) שקוראות לפרוטוקול TLS על מנת שיצפין את קו התקשורת. בעת יצירת קוקית יכול השרת לציין דגלון אבטחה והדפדפן ישלח את הקוקית באמצעות ערוץ מאובטח, כגון פרוטוקול SSL.



הרעלת קוקיות

קוקית אינה אמורה לעבור שינויים נוספים, מעבר לאלו שנעשו בעת היווצרותה, בעת המעבר בין הדפדפן לשרת. עם זאת, תוקף יכול לשנות את תוכן הקוקית לפני שהיא נשלחת חזרה לשרת. לדוגמה, אם נתונה קוקית שבה כלול סך הסכום לתשלום עבור הפריטים שהמשתמש מחזיק בעגלת הקניות, שינוי ערך זה יכול לגרום לכך שהמשתמש ישלם פחות מהנדרש.

קוקיות צד ג'

לקוקיות משמעות רבה בהקשר לנושא הפרטיות והאלמוניות באינטרנט. הקוקיות נשלחות אך ורק לשרת שיצר אותן או אל שרת אחר תחת אותו מתחם. אולם, דף האינטרנט עשוי לכלול תמונות או רכיבים אחרים שמקורם בשרת של domain אחר. קוקיות שנוצרות במהלך האחזור של רכיבים אלו קרויות בשם קוקיות צד ג'.

חברות פרסום משתמשות בקוקיות צד ג' כדי לעקוב אחר המשתמש בעודו עובר בין האתרים השונים חברת פרסום עשויה, למשל, לעקוב אחר משתמש שעבר דרך דפי אינטרנט שכוללים תמונות פרסומיות בעזרת ידע זה עשויה חברת הפרסום לדעת מהן העדפות המשתמש וכך להציג בפניו את הפרסומות שמתאימות לו

אבטחת מידע ביישומי רשת

יש דפי אינטרנט שהיינו רוצים להגביל את הגישה אליהם. לכן יש צורך במנגנוני אימות שמוודאים שגופים מורשים בלבד ניגשים לדפים אלה.

יש שתי גישות שניתן לנקוט בהן לצורך מטרה זו:

- אבטחה תכנונית – מעצב האתר צריך לתכנת את הלוגיקה של הגבלת הגישה. זו גישה מאוד גמישה אך דורשת תכנות רב.
- אבטחה הצהרתית – מצהירים מי יכול לגשת לאיזה משאב ומשתמשים במנגנונים שקיימים ב-HTTP כדי לשלוט על הגישה. זו גישה לא גמישה מאוד אך קלה לשימוש ולא דורשת כתיבת הרבה קוד חדש. HTTP מאפשר שתי סכמות אבטחה הצהרתית – Basic ו-Digest.

אבטחה ב-HTTP

ממלכה (Realm) היא קבוצת מסמכים בשרת. אם יש קבוצת מסמכים ולהם אותן זכויות גישה נגדיר אותם להיות באותה הממלכה. לכל ממלכה יש רשימה של משתמשים והסיסמאות שלהן. השרת בוחר כיצד להגדיר וכיצד לשמור נתונים אלה.

הרעיון הוא שכאשר משתמש רוצה לקבל גישה למשאב כלשהו בממלכה כלשהי הוא צריך לשלוח את שם המשתמש והסיסמה שלו. אם הם לא נשלחו יחד עם הבקשה, השרת יבקש אותם. אם יש למשתמש הרשאות גישה הוא יקבל את המשאב ואחרת השרת יסרב לשרת את הבקשה.

ההבדל בין שתי הסכמות של HTTP הוא בפרטים של הפרוטוקול שבו השיחה הזאת בין השרת למשתמש מתבצעת.

Basic Access Authentication

בפרוטוקול הבסיסי ארבעה שלבים:

1. הלקוח מבקש משאב מוגבל גישה
2. אם בבקשה לא מופיע ה-header עם שם המשתמש והסיסמה
   `Authorization: Basic username:password`
   השרת מבקש אותם מהלקוח עם תשובה עם קוד מצב 401[2] ובו מופיע ה-header
   `WWW-Authenticate: Basic realm = REALM_NAME`
3. הלקוח שולח את שם המשתמש והסיסמה שלו לשרת (בבסיס 64, **לא** מוצפנים)
4. אם הלקוח ברשימת המורשים השרת שולח את המשאב בחזרה

במהלך השיח הדפדפן שומר את שם המשתמש והסיסמה ושולח אותם אוטומטית לשרת אם המשאב המבוקש נמצא באותה ספריה כמו המשאב שכבר אומת, או אם השרת ביקש לאמת גישה באמצעות שליחת הודעה עם קוד 401 ו-WWW-Authenticate header עם אותה הממלכה כמו קודם.

נשים לב ששם המשתמש והסיסמה אינם מוצפנים בעת השליחה וכל מאזין לערוץ התקשורת יכול לגלות אותם ואז להשתמש בהם כרצונו.

---

[2] שפירושו שללקוח אין הרשאות לגשת למשאב

Digest Access Authentication

שיטת ה-Digest פותרת פגמים רבים של השיטה הבסיסית. כמו קודם, שם המשתמש והסיסמה צריכים להישלח בכל בקשה, אלא שכאן הסיסמאות לא נשלחות כטקסט רגיל. במקום לשלוח את הסיסמה עצמה נשלחת פונקצית ערבול חד-כיוונית שלה[3]. השרת שומר אצלו את הסיסמה של המשתמש וכאשר מגיעה הבקשה של הלקוח הוא יכול לאמת אותה באמצעות חישוב פונקצית הערבול בעצמו והשוואה לנתון שהגיע מהמשתמש. פונקצית ערבול פופולארית היא MD5 אשר מעבירה כל רצף שרירותי של בתים לתקציר (digest) בן 128 ביט. זה נותן כ-10^{39} תקצירים שונים אפשריים[4]. הנה כמה דוגמאות לפעולת הפונקציה:

קלט	תקציר MD5
Notes Heaven	f48d1634c67d754a2d91be5cef3a5579
Notes-Heaven	a8aed58473758718d2f3308310998749
http://notes-heaven.com	bbfb82cee1ace64dd17fdf71a84e8625
Pi	f1e85d1e676d4394d24e066127125672
3.14159265358979323846264338327950	b03c4a1689bef7f0528f91874fcff5bd
22/7	c62bc352613b07fbacda7bc543df0e73

בחזרה לעניינו. בשיטה זו אכן הסיסמה לא נשלחת ברשת באופן גלוי. אלא ששיטה זו עדיין לא עמידה בפני התקפת replay. מאזין לערוץ התקשורת אמנם לא יכול לגלות את הסיסמה עצמה אך ברגע שהוא יודע את ערך הערבול שלה, זה טוב באותה מידה, מאחר שהוא פשוט יכול לשלוח אותו מבלי לדעת את הסיסמה עצמה.

כדי לפתור בעיה זו "מקודדים" לא רק את הסיסמה אלא גם ערכים כלשהם שהשרת שולח ללקות בכל פעם שהלקוח מבקש משאב השרת יוצר nonce – מספר שלא אמור לחזור על עצמו אחרי בקשה זו. הלקוח מחשב את ערך הערבול של user:password:nonce והוא זה שנשלח לשרת לצורך אימות. זה מונע התקפת replay משום שאותו nonce לא אמור לחזור על עצמו עוד ולכן אם מתקיף ינסה להשתמש בהודעה שכבר ראה בעבר היא לא תעבור את תהליך האימות משום שהיא לא מכילה את ה-nonce הנכון.

כעת אף אחד לא יכול להתחזות ללקוח, שהרי הסיסמה שלו לא ידועה, היא לא עוברת באופן גלוי ברשת והשימוש ב-nonce-ים מונע התקפת replay. אלא שבינתיים ללקוח אין שום דרך לדעת שהוא אכן מדבר עם השרת. לכן ניתן להשתמש בשיטת ה-nonce גם מהצד של הלקוח. הלקוח שולח לשרת nonce והשרת בתורו מחזיר את ערך הערבול של user:password:client-nonce:resource. באופן זה גם הלקוח יכול לאמת שהוא אכן מדבר עם השרת.

HTTP/1.1 על קצה המזלג

עד עכשיו דיברנו רק על HTTP/1.0. ב-HTTP/1.1 נעשו הרבה שינויים ושיפורים. בנספח א' ניתן לראות תקציר של המאמר *Key Differences between HTTP/1.0 and HTTP/1.1* מאת **Balachander Krishnamurthy**, **Jeffrey C. Mogul** and **David M. Kristol** שדן בהבדלים בין הגרסאות.

---

[3] זוהי פונקציה חד-כיוונית H היא פונקציה על מחרוזות כך שבהינתן מחרוזת x, קל לחשב את התמונה $H(x)$ אך בהינתן $y = H(x)$ קשה לחשב את המקור x.

[4] פחות ממספר האטומים ביקום, אך עדיין די הרבה...

שימושיות האינטרנט

השימושיות של אתרי אינטרנט ושירותי רשת חשובה עוד יותר מהשימושיות של מוצרים מסורתיים כאשר לקוח קונה טלוויזיה הוא פונה למדריך שלה כדי לקבל הסבר על ההפעלה. אבל אם גולש אינטרנט מגיע לאתר ולא מצליח למצוא בו את מבוקשו או לא מצליח להשתמש בשירותיו הוא יוצא ממנו מחפש מקום אחר לקבל את רצונו. השימושיות של אתרי אינטרנט היא חיונית להישרדות שלהם!

ובכל זאת מחקרים מראים ש-90% מכלל האתרים לא לוקחים בחשבון שימושיות כאשר מעצבים את האתר.

עקרונות השימושיות

ג'ייקוב נילסון הגדיר ששה עקרונות שימושיות

1. **יכולת ללמוד** (learnability): האם משתמשים חדשים מצליחים לבצע משימות בסיסיות בפעם הראשונה שהם נתקלים במערכת?
2. **יעילות** (efficiency): ברגע שהמשתמשים למדו את העיצוב, כמה מהר הם יכולים לבצע משימות?
3. **יכולת לזכור** (memorability): אם משתמשים חוזרים למערכת אחרי הפסקה ארוכה שבה לא השתמשו בה, האם הם יכולים לחזור בקלות לאותה רמת שליטה שהייתה להם קודם במערכת?
4. **שגיאות** (errors): כמה טעויות משתמשם עושים וכמה מהר הם מתקנים אותן?
5. **שביעות רצון** (satisfaction): האם נעים להשתמש במערכת?
6. **שימושיות** (utility): האם המערכת עושה את מה שהמשתמשים צריכים?

כללי עיצוב

עיצוב דפים

- אחידות היא דבר חשוב. ככלל, כדאי ליצור template שיגדיר את המערך של כל הדפים. זה מונע את הצורך להחליט החלטות לכל דף בנפרד ובעיקר למשתמש קל יותר לזכור איפה כל דבר נמצא
- ניגודיות (contrast) בצבעים חיונית כדי שהמשתמש יוכל לשים לב לסוגים שונים של אובייקטים בדף: התפריט צריך להיות בצבע אחר, הכותרות צריכות לבלוט וכו'.
- לא להשתמש ברקעים רועשים - זה מקשה על הקריאה ועושה כאב ראש.
- להשתמש בגופנים סטנדרטיים.
- משתמשים לא אוהבים לגלול אנכית אבל גלילה אופקית זה אפילו יותר גרוע כדאי לעצב את האתר שיתאים לגדלים סטנדרטיים של מזכים ורזולוציות. במידת האפשר כדאי לעצב את האתר כך שיתכווץ או יתרחב לפי גודל החלון.
-

עיצוב עמוד הבית

עיצוב האתר

נגישות

חלק ב: מנועי חיפוש

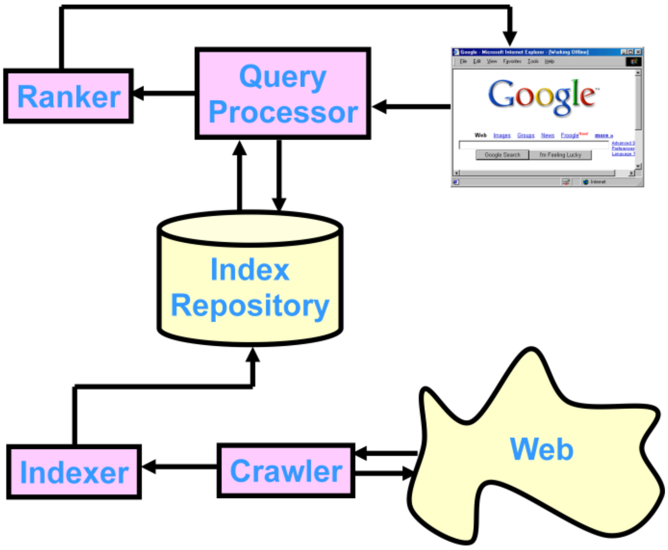
אתגרים בבניית מנוע חיפוש

כשבונים מנוע חיפוש חדש צריך לפתור כמה שאלות מעניינות ולהתחשב בכמה אתגרים שהרשת מציבה בפנינו:

- מהירות – הרשת מכילה **המון** מידע ויש לה הרבה מאוד משתמשים. חשוב להחזיר את תוצאת החיפוש הכי מהר שאפשר.
- דירוג – כדאי שמנוע החיפוש יחזיר את התוצאות מדורגות מהטוב אל הפחות טוב בעניין יש לזכור שההחלטה מהי תוצאה טובה היא סובייקטיבית ותלוית-משתמש.
- כיסוי – מנוע החיפוש צריך לדאוג שהוא מכסה חלקים נרחבים של הרשת
- אחסון – מידע על הדפים ברשת מאוחסן מקומית במנוע החיפוש. צריך למצוא דרך לאחסן את כל המידע בשרתי המנוע בצורה קומפקטית.

כל השאלות האלה קלות למענה כשמדובר בכמות קטנה של נתונים. בחיפוש שערכתי ב-Google מצאתי מישהו שטוען שכל האינטרנט יכול לתפוס בקלות עשרות או אפילו מאוד Exabytes[5].

מבנה של מנוע חיפוש



מנוע חיפוש מכיל ששה רכיבים עיקריים:

- אינדקס – מכיל דפי רשת ומידע נוסף
- Crawler – תוכנה שרצה ברקע כל הזמן, מטיילת ברשת ומחפשת דפים חדשים
- מאנדקס – תוכנה שרצה ברקע יחד עם ה-crawler ולוקחת דפים חדשים שנמצאו ומכניסה אותם ומידע עליהם לאינדקס
- מעבד שאילתות – תוכנה שמקבלת שאילתות ממשק המשתמש ומחזירה את התוצאות הרלוונטיות מהאינדקס
- מדרג – תוכנה שמדרגת את תוצאות השאילתה לפי טיב ההתאמה שלהן

[5] $1\ EB = 10^{18}$ bytes $= 10^{9}$ GB = הרבה מאוד מידע!!

- ממשק משתמש – הממשק שהמשתמש רואה ומתקשר דרכו עם מנוע החיפוש

האתגר העיקרי הוא לנהל את כל הפעילויות השונות שקורות במקביל. משתמשים מבקשים שאילתות, דפים חדשים נוספים לרשת ועוד.

האינדקס – איך מאחסנים כמויות גדולות של מידע?

רעיונות גרועים

המטרה היא לשמור מידע על הרבה (מאוד) דפים מהאינטרנט. השאילתות מגיעות בצורה של רשימה של מילים ואנחנו נרצה למצוא את קבוצת המסמכים שמכילים את המילים האלה. בשלב זה לא נדרג את התוצאות עדיין. כרגיל, יש trade off בין המהירות לגודל הזיכרון לסוג השאילתות שניתן לענות עליהן.

שמירה כקבצים

נשמור את הדפים כפי שהם בקבצים. אז ניתן למצוא מילים ע"י מעבר פשוט על הקובץ (בעזרת כלים כמו grep).

נניח שהמערכת שלנו מאופיינת ע"י הערכים הבאים:

זמן ממוצע ל-seek של הדיסק	5msec
זמן מעבר בית לזיכרון הראשי	0.02usec
פעולות בסיסיות של המעבד	0.01usec
גודל הזיכרון הראשי	כמה GB
גודל הדיסק	1TB

נניח שיש 1GB של טקסט ששמור ברצף בדיסק. אז כדי להעביר אותו לזיכרון הראשי צריך לעשות פעולת seek אחת ואז לקרוא ברצף. לכן הזמן שזה ייקח הוא $5msec + 0.02usec \cdot 109 = 20.005sec$.

כעת, אם יש לנו 1GB של מידע מחולק ל-100 חלקים רציפים בדיסק נצטרך לעשות 100 פעולות seek ולכן הזמן שזה ייקח הוא $5msec \cdot 100 + 0.02usec \cdot 109 = 20.5s$.

בדוגמה זו ההבדל לא מאוד ניכר אבל אם הנתונים פזורים במקומות שונים בדיסק הקריאה הזו יכולה להיות מאוד לא יעילה. לכן שיטת השמירה בקצבים יכולה להיות מהירה אבל זה תלוי בכמות המידע שיש ובכמות הזיכרון שיש ובעיקר בסידור הקבצים על הדיסק.

סיבוכיות הזיכרון של השיטה הזו היא רחוקה מאוד מאופטימאלית. מילים יכולות לחזור על עצמן פעמים רבות במסמכים והן כולם נשמרות בכל זאת. לכל הפחות ניתן היה לתמצת כל מסמך להכיל כל מילה פעם אחת בלבד...

שמירה במסד נתונים רלציוני

ניתן לחשוב על כמה מודלים לשמירת האינדקס במסד נתונים.

מודל א': טבלה עם מספר המסמכים והתוכן שלהם:

Documents	
Document ID	Document

1	Hurricane Ioke was the strongest hurricane ever recorded in the Central Pacific...
2	Pi or π is a mathematical constant whose value is the ratio of any circle's circumference to its diameter in Euclidean space...
...	

כדי למצוא מסמכים שמכילים אוסף של מילים ניתן להשתמש בשאילתותSQL רגילות:

```
select docID
from Documenets
where document conatins word1 && ... && document contains wordN
```

השיטה הזו למעשה זהה לשיטה של השמירה בקבצים פרט אולי לאופטימיזציות שמסד הנתונים יכול לתת במה שקשור לקריאה של הנתונים מהדיסק. בסופו של עניין עבור שאילתא המסד הולך לעבור על כל המסמכים ולבדוק באילו מהם קיימות המילים המבוקשות ולכן לא שיפרנו את מצבנו כלל.

מודל ב': טבלת מסמכים והמילים שמופיעות בהם:

Appears	
Document ID	**Word**
1	hurricane
1	Ioke
1	was
1	the
...	
2	pi
2	or
2	π
...	

השיטה הזו דומה מאוד לקודמת אבל יש לה חיסרון ויתרון. החיסרון הוא שכדי לענות על שאילתא שכוללת כמה מילים נצטרך להשתמש בפקודת INTERSECT שהיא פעולה יקרה (כי היא דורשת למשל מיון של הנתונים):

```
select DocID
from Appears
where Word = word1

INTERSECT
...
INTERSECT

select DocID
```

```
from Appears
where Word = wordN
```

אבל היתרון הגדול של השיטה הוא שמאגר הנתונים כאן תופס פחות מקום כל מילה בכל מסמך מופיעה רק פעם אחת בניגוד לשיטה הקודמת שפשוט כללה עותק של המסמכים כפי שהם.

מודל ג': נשמור שתי טבלאות – האחת עם המילים עצמן והשנייה כמו במודל ג'.

Word Index	
Word	**Word ID**
hurricane	1
loke	2
Pi	3
...	

Appears	
Doc ID	**Word ID**
1	1
1	2
2	3
...	

כאן חישוב השאילתא מעט יותר מסובך ודורש פעולת join של הטבלאות:

```
select DocID
from Appears, WordIndex
where Word = word1 and Appears.WordId = WordIndex.WordId
```

מאידך, שיטה זו עוד יותר חסכונית באחסון משום שמילים עצמן נשמרות רק פעם אחת ואילו רק המספר המזהה שלהן (אשר סביר להניח שתופס הרבה פחות מקום מהמילה עצמה) משמש לקישור בין המסמכים למילים.

גם כאן כדי לחשב שאילתות של כמה מילים יש להשתמש בפעולת INTERSECT.

ככלל, לשימוש במסד נתונים רלציוני יש כמה חסרונות שמשותפים לכל המודלים שדיברנו עליהם למשל, לא ניתן לחשב בקלות שאילתות שדורשות שמילה מסוימת **לא** תופיע במסמך. כמו כן, מילים שמופיעות במסמכים שונים יירשמו במסד כמספר המסמכים השונים וזה בזבוז מקום. כמו כן, אם מילה מופיעה בכמה מסמכים, כדי למצוא אותם צריך לעבור על מספר שורות כמספר המסמכים (לפחות) וזה יכול לבזבז הרבה זמן בגלל גישות מרובות לדיסק. ולבסוף, כפי שכבר ציינו, שאילתות שדורשות כמה מילים הרבה פחות יעילות לחישוב אז השיטה לא מתרחבת באופן טבעי לשאילתות מרובות מילים.

Bitmaps

נשמור טבלה גדולה של ביטים שאומרת לכל זוג של מילה ומסמך אם המילה מופיעה במסמך:

Word	Doc_1	Doc_2	...	Doc_M
$Word_1$				

$Word_2$				
...			Is $Word_i$ in Doc_j?	
$Word_N$				

אם יש N מילים ו-M מסמכים אז בטבלה יהיו MN תאים. זה יכול להיות די בזבזני. נניח שיש בסה"כ 500 אלף מילים שונות, 10^6 קבצים ובכל אחד 1000 מילים. אזי בטבלה יש $500000*10^6=5*10^{11}$ תאים ובהם בסה"כ $1000*10^6=10^9$ תאים שבהן 1. פירוש הדבר ש-$(5*10^{11}-10^9)/5*10^{11}=0.998$ מהמקום שבטבלה כלל לא מנוצל! זה בזבוז עצום!!

מאידך, חישוב השאילות נוח, קל ויעיל. לכל מילה נסתכל על השורה המתאימה לה כווקטור מציין של המסמכים שהיא מופיעה בהם. אז על השאילתות קל מאוד לענות בעזרת פעולות על ביטים וזה בתורו יכול להיות ממומש בצורה יעילה בחומרה.

נסמן ב-V_w את הווקטור המציין של מילה w. אז התשובה על השאילה w_N and ... and w_1 היא פשוט פעולת and על הווקטורים המתאימים – V_{w1} and ... and V_{wN}. באופן דומה כל שאילתא בוליאנית על אוסף מילים מיתרגמת לאותה פעולה בוליאנית על הווקטורים המציינם המתאימים.

רעיונות טובים יותר

בחלק זה נציג כמה רעיונות מוצלחים יותר לתחזוק האינדקס.

מבנה האינדקס

האינדקס מורכב משני מבני נתונים:

- לקסיקון – רשימת כל המילים שמופיעות במסמכים. לכל מילה נשמור מצביע לרשימה שלה באינדקס המהופך.
- אינדקס מהופך (inverted index) – לכל מילה בלקסיקון נשמור רשימה של המסמכים שהיא מופיעה בהם (כמובן, עדיף לשמור מזהה של המסך ולא את המסמך עצמו). לפעמים שומרים גם את המיקומים של המילה בתוך המסמכים (זה יכול לעזור כאשר מעוניינים לתמוך גם בשאילתות שמתייחסות למיקום של המילים בתוך המסמכים).

דוגמה:

נניח שיש שלושה מסמכים:

Doc1: a b c Doc2: e b d Doc3: a b d f

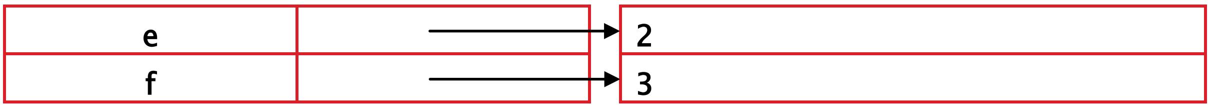
אז מבנה הנתונים מאורגן כך:

Lexicon

Word	Pointer to list
a	→
b	→
c	→
d	→

Inverted Index

1,3
1,2,3
1
2,3



כדי למצוא את רשימת המסמכים שתי מילים צריך למצוא את המסמכים שנמצאים בשתי הרשימות המתאימות. זה נשמע פשוט אך יש פרטים שצריך לשים לב אליהם: צריך למצוא את זוג המילים בלקסיקון ואז צריך לחשב את חיתוך הרשימות. צריך לתכנן את מבני הנתונים כך שהפעולות האלה יהיו יעילות כמה שניתן! אך גם חסכוניות במקום...

הלקסיקון

בשלב הראשון נניח שאנחנו רוצים לתמוך רק בשאילתות של מילה אחת אשר נתונה במלואה ללא תווי-כל (wildcards). המטרה שלנו תהיה לתכנן מבנה נתונים יעיל בזמן החיפוש (למשל גישה בזמן לוגריתמי) שלא תופס הרבה מקום (ונכנס בזיכרון הראשי).

נרצה לשמור בלקסיקון שלשות של מילים, מצביע לרשימות שלהן ואורך הרשימות (כפי שנראה בהמשך, הרשימות יופיע בזיכרון באופן רציף ולכן צריך לדעת את האורך כדי לדעת מתי להפסיק לקרוא את הרשימה).

בדוגמאות שיופיעו בהמשך נניח את הפרמטרים הבאים:

מספר המילים השונות הקיימות	1000000
אורך מילה ארוכה ביותר	20 תווים
אורך מילה ממוצעת	8 תווים
תו	1 בית
מצביע	4 בתים
ערך תדירות	4 בתים

נבחן כמה אופציות לשמירת הלקסיקון.

מחרוזות שוות ארוך

נשמור שלשות שוות אורך שמכילות את המילה, מספר המסמכים שהיא מופיעה בהם ומצביע לרשימת המסמכים. השלשות תהיינה ממוינות לפי א"ב לפי המילים:

Word	Frequency	Pointer to list
jezebel	20	
jezer	3	
jezerit	1	
jeziah	1	
jeziel	1	
jezliah	1	
jezoar	1	
jezrahiah	1	
jezreel	39	

מאחר שהמילה הארוכה היותר היא באורך 20 תווים בטבלה הנ"ל צריך להיות מקום לפחות ל-20 תווים בשדה המילה. לאחר מכן נשמרת התדירות שלה אשר תופסת 4 בתים ולבסוף נשמר מצביע שגם הוא באורך

4 בתים. לכן בסה"כ כל שורה תהיה באורך $20*1bytes+4bytes+4bytes=28bytes$. בטבלה יש שורה לכל מילה ולכן בסה"כ הטבלה תופסת $1000000*28bytes=28MB$.

מאחר שהרשומות שוות ארוך וממוינות לפי המילה, ניתן לגשת לרשומות נפרדות ע"י הצורך ולכן מתאפשר חיפוש בינארי בלקסיקון – כלומר זמן הגישה ללקסיקון הוא לוגריתמי.

מחרוזות משורשרות

השיטה הקודמת של שמירת הלקסיקון בזבזנית במקום שהרי כל שורה תופסת בדיוק אותו מספר בתים למרות שלמעשה יש שורות שיכלו להיות הרבה יותר קצרות.. במקום זה ניתן לשרשר את כל המילים למחרוזת אחת ארוכה ולשמור מצביעים לתוך המחרוזת. אז בלקסיקון יש לנו שני מבני נתונים: מחרוזת ארוכה וטבלה דומה לטבלה מקודם, אלא שמקום מילים יש בה מצביעים:

...jezebeljezerjezeritjeziahjezieljezliahjezoarjezrahiahjezreel...

Pointer to word	Frequency	Pointer to list
	20	
	3	
	1	
	1	
	1	
	1	
	1	
	1	
	39	

כמה מקום תופס הלקסיקון כעת?

במחרוזת הארוכה יש 1000000 מילים שאורכן הממוצע 8 תווים. לכן המחרוזת הארוכה תופסת $1000000*8*1bytes=8MB$. הטבלה תופסת כעת פחות מקום משום שבכל שורה יש רק $(4+4+4)bytes=12bytes$ ולכן בסה"כ $1000000*12bytes=12MB$. אז בסה"כ כל מבנה הלקסיקון כולו תופס $8MB+12MB=20MB$. זה שיפור ניכר לעומת השיטה הקודמת!

נשים לב שלא צריך לשמור את אורך כל מילה משום שהמצביע בשורה הבאה מצביעלמקום אחד אחרי שהמילה נגמרת. כמו כן, משום שהמצביעים עדיין ממוינים לפי המילים עצמן עדיין ניתן לבצע חיפוש בינארי בלקסיקון!

Front Coding

נשים לב שבמחרוזת הארוכה מהדוגמה שלנו התחילית jez חוזרת על עצמה הרבה מאוד פעמים. נראה חבל לשמור אותה שוב ושוב ושוב. נרצה לנצל את העובדה שלמילים עוקבות יש תחיליות משותפות אז נשמור את הלקסיקון בעזרת קידוד קדמי (Front Coding): נסיר את התחיליות המשותפות, נשמור את אורך התחיליות שהסרנו ונשמור מצביעים לתוך המחרוזת המשורשרת.

ניח בדוגמה שלנו שהמילה jezebel היא המילה הראשונה בלקסיקון. אז נקבל:

jezebelritiahelliahoarrahiaheel...

For clarity...	Pointer to word	Prefix Size	Frequency	Pointer to list
(jezebel)		0	20	
(jezer)		4	3	
(jezerit)		5	1	
(jeziah)		3	1	
(jeziel)		4	1	
(jezliah)		3	1	
(jezoar)		3	1	
(jezrahiah)		3	1	
(jezreel)		4	39	

המילה הכי ארוכה היא באורך 20 תווים ולכן התחילית המשותפת הארוכה ביותר שיכול להיות היא באורך 19 תווים. לכן כדי לייצג את אורך התחילית המשותפת אנחנו צריכים לכל היותר חמישה ביטים.

נניח שבממוצע יש שלושה תווים בתחילית משותפת אזי במחרוזת המשותפת תופענה 1000000 מילים והפעם אורכן הממוצע יהיה 5 תווים (כי אנחנו מורידים את התחילית המשותפת). לכן גודל המחרוזת יהיה $1000000*5*1bytes=5MB$. למעשה החישוב הזה לא מדויק כי יש לקחת בחשבון שאת המילה הראשונה אנו שומרים במלואה אך בוודאי החישוב שעשינו מראה את סדר הגודל שמדובר פה ושונה מהחישוב האמתי בכמה בתים לכל היותר. כעת, כל שורה בטבלה תופסת $5bit+(4+4+4)byte=101bit$ ויש 1000000 שורות, לכן הטבלה תופסת $1000000*101bit=12.625MB$. אז כל המבנה כולו תופס כעת כ-18MB. זה אמנם שיפור ביחס לשיטה הקודמת אבל איבדנו את היכולת לבצע חיפוש בינארי שטבלה אמנם כל שורה בטבלה היא שוות אורך ולכן ניתן לגשת לכל שורה באופן אקראי אך לא ניתן ב-$O(1)$ לגלות מה המילה שמתאימה למצביע. צריך לחזור עד תחילת המחרוזת הארוכה ולעקוב אחרי כל התחיליות החסרות כדי לגלות מהי המילה שמסתתרת שם וזה לוקח זמן ליניארי במספר המילים שיש בלקסיקון.

3-in-4 Front Coding

שיטת הקידוד הקדמי הייתה מאוד חסכונית במקום אך איבדנו את היכולת לבצע חיפוש בינארי וזה חבל. שיטת קידוד קדמי 3 ב-4 מאפשרת גם לחסוך במקום וגם לשמור את האפשרות לבצע חיפוש בינארי.

השיטה דומה לשיטת הקידוד הקדמי. למעשה, מחלקים את כל המילים לקבוצות בנות ארבע מילים וכל קבוצה מקודדים בקידוד קדמי. כלומר, המילה הראשונה נשמרת בשלמותה ואילו שלוש האחרות מקודדות בקידוד קדמי. כך כאשר ניגשים למצביע ורוצים לדעת מה המילה שהוא מצביע אליה יש לכל היותר ארבע מילים שצריך לעבור עליהן. לכן הזמן לגלות את המילה הוא $O(1)$ וניתן לבצע חיפוש בינארי.

גודל הלקסיקון כעת מעט גדול יותר. כמו קודם, נניח שתחילית משותפת ממוצעת היא באורך שלושה תווים כעת, רבע מהמילים מצויות במחרוזת בשלמותן ושאר המילים מחוסרות תחיליות. לשם הנוחות נניח שעדיין האורך הממוצע של המילים המופיעות בשלמותן הוא 8 תווים. לכן במחרוזת הארוכה יהיו $1000000/4*8*1bytes+1000000*3/4*5*1bytes=5.75MB$. ואילו הטבלה תופסת כמו קודם 12.625MB ולכן בסה"כ הלקסיקון תופס 18.375MB. אז במחיר לא מאוד גדול הצלחנו גם לחסוך במקום וגם לאפשר גישה יעילה!

סיכום
הלקסיקון אמור לאפשר גישה מהירה למילים. אחרי שמוצאים מילה ניתן לגשת לרשימת המסמכים שהיא מופיעה בהם ולענות על השאילתא שקיבלנו. ראינו ארבע שיטות לשמור את הלקסיקון אשר נבדלות זו מזו בזמן הגישה ובמקום שהן תופסות.

ישנה שיטה נוספת שלא נדון בה בהרחבה אשר משתמשת במבנה נתונים שנקרא Trie. פרטים נוספים מופיעים בנספח ב'.

האינדקס המהופך
האינדקס המהופך הוא אוסף רשימות שנקראות Posting Lists – אחת לכל מילה בלקסיקון. לדוגמה, אם נתונים שלושה מסמכים

Doc1: a c f Doc2: b e d b Doc3: a b d f

אז הרשימה של b תהיה 2,3. אם נרצה לשמור גם את המיקומים של המילים בכל מסמך הרשימה תהיה

(2: 1,4),(3: 2)

למעשה הרשימה תישמר בצורה 2,2,1,4,3,1,2 כאשר לכל מסמך לאחר המספר שלו מופיע מספר המופעים של המילה ואז רשימת המיקומים שלה. זה מאפשר לשחזר מהרשימה את המסמכים והמיקומים באופן חד-ערכי. שמירת המיקומים כמובן מגדילה את אורך הרשימות, גם בגלל שצריך לשמור את המיקומים עצמם וגם בגלל שצריך לשמור את מספר המיקומים בכל מסמך.

לעת עתה נניח שהרשימות מכילות רק מספרי מסמכים ולא את המיקומים של המילים את מספרי המסמכים נשמור בצורה ממוינת. אז posting list היא למעשה פשוט רשימה ממוינת של מספרים.

המבנה מאוד פשוט אך יש הרבה מאוד רשימות כאלה והן יכולות להיות ארוכות מאוד. לכן צריך למצוא דרך לשמור אותן בצורה מכווצת.

רשימות הפרשים
במקום לשמור את הרשימה עצמה ניתן לשמור רק את האיבר הראשון בה ואחריו את ההפרשים. כלומר, אם יש לנו רשימה $x_1,x_2,\ldots,x_n$ נשמור במקומה את $x_1,x_2-x_1,\ldots,x_n-x_{n-1}$. הרעיון הוא שההפרשים הם מספרים קטנים יותר ולכן ניתן לייצג אותם במספר קטן יותר של ביטים אבל למעשה, ההפרש הגדול ביותר האפשרי הוא מאותו סדר גודל כמו מספר המסמכים הקיימים ולכן אם משתמשים בקידוד עם אורך קבוע כדי לייצג את ההפרשים, לא נחסוך מקום בכלל. בלאו הכי לכל הפרש נצטרך לשמור אותו מספר ביטים כמו המספר הנחוץ כדי לייצג מספר של מסמך. אז נצטרך למצוא דרך אחרת לייצג את רשימת ההפרשים.

דחיסה
ישנם שני סודי דחיסה: דחיסה פרמטרית ודחיסה לא פרמטרית. בדחיסה פרמטרית הדחיסה מותאמת באופן אישי למידע שיש לכווץ. ואילו בדחיסה לא פרמטרית הכיווץ אינו תלוי באף פרמטר חיצוני; ובפרט אינו תלוי בקלט. כמו כן, הדחיסה יכולה להיות גלובאלית, כלומר כל רשימה נדחסת ע"י אותה טכניקת כיווץ והיא יכולה להיות מקומית, כלומר כל רשימה נדחסת באופן אחר.

שיטות ללא פרמטרים
אנחנו רוצים לדחוס רשימות הפרשים. על אף העובדה שלא ניתן להניח כי כל ההפרשים יהיו קטנים ממספר המסמכים, בכל זאת סביר להניח שהפרשים קטנים יהיו נפוצים יותר מאשר הפרשים גדולים מילים נפוצות מופיעות בהרבה מסמכים ולכן יוצרות הפרשים קטנים ואילו גם מילים לא נפוצות יכולות ליצור הפרשים קטנים

אם המסמכים נטענים וממוספרים באיזשהו סדר לוגי של הקשר. לכן נרצה לקודד מספרים קטנים בפחות ביטים מאשר מספרים גדולים.

שנון הוכיח חסם תחתון על הדחיסה האפשרית. נניח שמספר התווים הוא m ויש c תווים שונים אפשריים. אם תו i מופיע $f(i)$ פעמים אז התדירות של i היא $p_i = \frac{f(i)}{m}$ והאנטרופיה המוגדרת ע"י $H = -\sum_{i=1}^{c} p_i \log p_i$ היא חסם תחתון על מספר הביטים הממוצע שצריך כדי לייצג כל תו במסמך שלנו בצורה מכווצת. אז כדי לכווץ את כל המסמך כולו צריך לכל הפחות mH ביטים. כמו כן, ניתן לומר שאם לסימן i יש תדירות p_i אז יש צורך בלפחות $-\log p_i$ ביטים כדי לייצג אותו.

באופן אידיאלי, היינו רוצים למצוא שיטת כיווץ שמממשת את החסם התחתון.

קידוד אונארי

כל מספר $0 < x$ (בשיטה זו אי אפשר לקודד את המספר 0!) נייצג ע"י $x - 1$ מופעים של 1 ואז מופע אחד של 0. למשל:

מספר	קידוד אונארי של
1	0
2	10
3	110
10	1111111110

הפענוח כמובן טריוויאלי. כל מופע של 0 מייצג מספר חדש ומספר המופעים של 1 מאפשר לגלות מהו המספר המקודד.

נניח שהקידוד הזה אופטימאלי, כלומר הוא משיג את החסם התחתון של שנון. נשים לב שמספר הביטים בייצוג של x הוא למעשה x. כמו שאמרנו, אם ייצוג זה אופטימלי אז מספר הביטים הנחוצים לייצוג של אות שתדירותה p_i הוא $-\log p_i$ ולכן נקבל שעבור מספר x מתקיים $x = -\log p_x$, כלומר $2^x = 2^{-\log p_x} = \frac{1}{p_x}$ ולכן $p_x = 2^{-x}$. אז התדירות של מספר יורדת מעריכית אם הגודל שלו. זה לא נשמע סביר במיוחד.

קידוד גמא

עבור מספר כלשהו $x > 1$, נקודד אותו באופן הבא: ראשית, נכתוב את הקידוד האונארי של $1 + \lfloor \log x \rfloor$ ולאחר מכן נכתוב את הקידוד הבינארי באורך $\lfloor \log x \rfloor$ של $x - 2^{\lfloor \log x \rfloor}$. למשל:

מספר	$\lfloor \log x \rfloor$	קידוד אונארי של $1 + \lfloor \log x \rfloor$	$x - 2^{\lfloor \log x \rfloor}$	קידוד בינארי של $x - 2^{\lfloor \log x \rfloor}$	קידוד גמא של x
1	0	0	0	-	-
2	1	10	0	0	100
3	1	10	1	1	101
5	2	110	1	01	11001
9	3	1110	1	001	1110001
11	3	1110	3	011	1110011

פענוח של מספר יחיד הוא לא מסובך. בתחילת על מספר מופיע רצף של y 1-ים (y יכול גם להיות 0) ואז מופע של 0. אז לפי שיטת הקידוד נובע ש-$\lfloor \log x \rfloor + 1 = y + 1$. כלומר, $y = \lfloor \log x \rfloor$. וזה אורך הקידוד הבינארי של $x - 2^{\lfloor \log x \rfloor} = x - 2^y$. לכן ניתן לחלץ את x. לדוגמה: נתבונן במילת הקוד 1110010. בתחילתה יש רצף של שלוש אחדות ואפס אשר מייצגים את המספר 4. לכן אורך הקידוד של החלק הבינארי הוא $3 = 1 - 4$. החלק הבינארי הוא 010, כלומר 2. אז $x - 2^3 = 2$. מכאן ש-$x = 10$.

אבל מכאן גם ברור איך לפענח רצף של מילות קוד, שהרי אחרי שקראנו את הרצף הראשון של אחדות אנחנו כבר יודעים איפה נגמר הקידוד של המספר הראשון. ועכשיו הבעיה חוזרת על עצמה.

למשל, ניתן לפרק את הרצף 1111010011011111010 כך: 111101001,101,11010.

כאן, אורך הקידוד של x הוא $1 + 2\lfloor \log x \rfloor = \lfloor \log x \rfloor + \lfloor \log x \rfloor + 1$. לכן אילו קידוד גאמא היה אופטימלי היינו מקבלים ש-$-\log p_x = 1 + 2\lfloor \log x \rfloor$, כלומר $\frac{1}{p_x} = 2^{1+2\lfloor \log x \rfloor} \approx 2^{1+2\log x} = 2x^2$ ומכאן ש-$p_x \approx \frac{1}{2}x^{-2}$. יכול להיות שזה יותר סביר מהמקרה האונארי אך בכל אופן מאחר שהקידוד של כל מספר הוא באורך שונה אנחנו מאבדים את היכולת לבצע חיפוש בינארי. יתר על כן, אם ניגש למקום אקראי ברשימה המקודדת לא נוכל לדעת אם המקום הזה הוא תחילה של מספר או אמצעו כדי לפענח את הרשימה חייבים לקרוא את כולה מההתחלה.

קידוד פשוט עם אורך משתנה

בכל בית יש שמונה ביטים. נשתמש בביט הראשון כדי לציין אם זה הבית האחרון של הקידוד או לא באשר הביטים נשתמש כדי לייצג את המספר בקידוד בינארי. השיטה הזו יכולה להיות מאוד מהירה כי עיבוד של בתים שלמים ממומש בצורה יעילה בחומרה.

למשל, הייצוג הבינארי של 256 הוא 100000000, כלומר צריך 9 ביטים כדי לייצג את המספר. לכן נזדקק לשני בתים ומספר הביטים שמיועדים לקידוד יהיה 14. אז למעשה הקידוד הבינארי של 256 שנשתמש בו הוא 00000100000000 ונקבל את הייצוג 10000000 00000010. הפענוח טריוויאלי.

באופן כללי, כדי לייצג מספר x בבסיס בינארי יש צורך ב-$\lfloor \log_2 x \rfloor + 1$ ביטים. את הייצוג הזה אנחנו צריכים לחלק לחלקים של שבעה ביטים. לכן יש $\left\lceil \frac{\lfloor \log_2 x \rfloor + 1}{7} \right\rceil$ חלקים, כלומר $\left\lceil \frac{\lfloor \log_2 x \rfloor + 1}{7} \right\rceil$ בתים או $8 \cdot \left\lceil \frac{\lfloor \log_2 x \rfloor + 1}{7} \right\rceil$ ביטים.

שיטות עם פרמטרים

נדון בשתי שיטות חשובות לקידוד פרמטרי.

קודי האפמן קנוניים

ראשית, נזכיר מהו קוד האפמן. קוד האפמן הוא קוד רישא אופטימלי. בקוד רישא מקודדים כל אות בנפרד ואין מילת קוד שהיא רישא של מילת קוד אחרת. כך ניתן לפענח את הקוד באופן חד-ערכי.

בהינתן קבוצה של סימנים ותדירויות בונים את קוד האפמן שלהן בעזרת האלגוריתם החמדן הבא

1. צור עלה לכל אות וכתוב את תדירות האות בעלה
2. כל עוד יש שני צמתים ללא הורה
   a. בחר שני צמתים בעלי הערך הנמוך ביותר
   b. צור צומת חדש ובו סכום הערכים של הצמתים מהסעיף הקודם
   c. חבר את הצומת החדש לשני הצמתים המינימאליים

בסוף האלגוריתם נוצר עץ שבאמצעותו ניתן לקבל את הקידוד של כל אות עוברים במסלול מהשורש לעלה שמייצג את האות. כל פנייה ימינה נותנת מופע של 1 וכל פנייה שמאלה נותנת מופע של 0.

ניתן להוכיח שהקוד הזה הוא אכן קוד רישא אופטימלי.

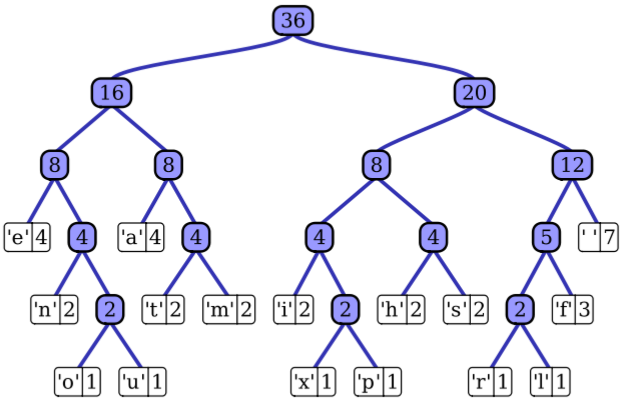
נשים לב שקוד האפמן אינו יחיד משום שבכל שלב זוג הצמתים שנבחר לא בהכרח נקבע ביחידות בפרט אם לכל האותיות יש מלכתחילה אותה תדירות אז כבר בשלב הראשון יש $n(n-1)$ דרכים לבחור זוג אותיות ראשון.

לדוגמה:

נניח שנתונה טבלת סימנים:

Character	Space	A	E	F	H	I	M	N	S	T	L	O	P	R	U	X
Frequency	7	4	4	3	2	2	2	2	2	2	1	1	1	1	1	1

אז יכול למשל להתקבל עץ הקידוד הבא:



על אף האופטימאליות של הקוד הזה, יש לו כמה חסרון שמונע ממנו להיות מתאים למקרה שיש הרבה מאוד סימנים לקודד. כדי שנוכל לפענח את המידע המקודד יש צורך לשמור את העץ ולכל מילת קוד יש לעבור בעץ במסלול שעובר מהשורש ומגיע עד לעלה. כדי שהתהליך הזה יהיה יעיל כדאי מאוד שהעץ כולו ייכנס בזיכרון הראשי. אבל אם כמות הסימנים שיש לקודד גדולה מאוד יכול להיות שהזיכרון לא יספיק וזה יאט משמעותית את פעולת הפענוח.

הפתרון לבעיה זו הוא קודי האפמן קנוניים. הרעיון מאחורי הקוד הוא שמספיק לדעת את רשימת הסימנים ואורכי הקידוד שלהם כדי לפענח טקסט. מידע זה כמובן תופס הרבה פחות מקום מעץ קידוד שלם.

לקוד האפמן קנוני יש כמה מאפיינים:

- מילות קוד באותו אורך הן מספרים בינאריים עוקבים
- בהינתן שני סימנים s,s' עם קידוד שווה $c(s),c(s')$ מתקיים $c(s) < c(s') \Leftrightarrow s < s'$
- מילת הקוד הקצרה ביותר הראשונה היא מחרוזת של 0
- מילת הקוד הארוכה ביותר האחרונה היא מחרוזת של 1
- אם d מילת הקוד האחרונה באורך i ומילת הקוד הבאה c היא באורך j אז $c = 2^{j-i}(d+1)$ (כלומר מוסיפים ל-d 1 ואז מרפדים ב-$j-i$ אפסים כך שיתקבל האורך j)

דוגמה: נניח שנתונה טבלת הסימנים הבאה:

Character	A	B	C	D	E	F	G
Code length	3	2	4	4	2	3	3

ראשית, נמיין את הסימנים לפי הסדר שבו יהיה מסודר הקוד שלהן. האותיות עם קידוד קצר באות לפני אותיות עם קידוד ארוך ובתוך קבוצה של אותיות שוות ערך האותיות ממוינות לפי א'ב':

- סימנים באורך 2: B,E

- סימנים באורך 3: A,F,G
- סימנים באורך 4: C,D

אז B היא המילה המקודדת הראשונה ולכן הקוד שלה הוא 00. אחריה מגיעה E והיא אמורה להיות מספר בינארי עוקב של B ולכן הקוד שלה הוא 01. כעת A היא האות הראשונה שהקידוד שלה הוא באורך 3. לכן נחבר לקידוד של E 1 ונקבל 10 וכעת נרפד באפסים מימין עד שנקבל את הקידוד באורך 3 והוא 100. לכן F מקודדת ע"י 101 ו-G מקודדת ע"י 110. עבור C נוסיף 1 ל-110 ונקבל 111. כעת נרפד ונקבל 1110, ולבסוף הקידוד של D הוא 1111.

הפענוח של קוד האפמן קנוני הוא לא מסובך. נתונה לנו טבלה של סימנים ואורכי הקידוד שלהם. נסמן ב-$l_1, \dots, l_n$ את אורכי מילות הקוד בקוד הקנוני. נסמן:

- c_i היא מילת הקוד של הסימן הראשון שהקידוד שלו באורך l_i (זה המספר הבינארי המינימלי בקידוד באורך l_i)
- n_i הוא מספר מילות הקוד באורך l_i

את שני אלה קל לחשב בהינתן המידע שיש לנו לגבי אורכי הקידודים. כעת נפעל לפי האלגוריתם הבא:

1. כל עוד יש סימנים לקרוא:
   a. נאתחל $i = 0$
   b. חזור על:
      i. i = i + 1
      ii. תהי d המילה המתקבלת ע"י קריאת l_i סימנים

   עד ש-$d \leq c_i + n_i - 1$ ($c_i + n_i - 1$ היא מילת הקוד המקסימלית באורך l_i. אם d גדולה ממילת קוד זו היא לא יכולה להיות מילת קוד חוקית ואילו התנאי מתקיים אז בגלל שמילות הקוד באורך נתון הן מספרים בינאריים עוקבים נובע שזו בהכרח מילת קוד. מאחר שאין מילת קוד שהוא רישא של מילת קוד אחרת, d היא בהכרח מילת הקוד שאנחנו מחפשים).
   c. החזר את הסימן ה-$d - c_i + 1$ באורך l_i לפי סדר הא"ב

דוגמה: עם הטבלה מהדוגמה הקודמת, נפענח את 10001110.

- אתחול: $i = 0$
- איטרציה ראשונה: $i = 1$ ואז $n_1 = 2, c_1 = 00, d = 10, l_1 = 2$. לכן $c_1 + n_1 - 1 = 0 + 2 - 1 = 1$ ולא מתקיים $1 \leq d = 2$. לכן ממשיכים ל-
- איטרציה שנייה: $i = 2$ ואז $n_2 = 3, c_2 = 100, d = 100, l_2 = 3$. לכן $c_2 + n_2 - 1 = 4 + 3 - 1 = 6$ ומתקיים $6 \leq d = 4$ ולכן לא ממשיכים לאיטרציה הבאה ומחזירים את הסימן ה-$4 - 4 + 1 = 1$ באורך 3 שהוא A.

נותר לפענח את 01110. שוב:

- אתחול: $i = 0$
- איטרציה ראשונה: $i = 1$ ואז $n_1 = 2, c_1 = 111, d = 1110, l_1 = 4$. לכן $c_3 + n_3 - 1 = 7 + 2 - 8 = 1$ ואכן מתקיים $8 \leq d = 1$. לכן לא ממשיכים לאיטרציה הבאה ומחזירים את הסימן ה-$2 = 1 + 0 - 1$ באורך 2 שהוא E.

נותר לפענח את 110. כמו קודם:

- אתחול: $i = 0$
- איטרציה ראשונה: $i = 1$ ואז $n_1 = 2$, $c_1 = 00$, $d = 11$, $l_1 = 2$. לכן $c_1 + n_1 - 1 = 0 + 2 - 1 = 1$ ולא מתקיים $1 \leq d = 3$. לכן ממשיכים ל-
- איטרציה שנייה: $i = 2$ ואז $n_2 = 3$, $c_2 = 100$, $d = 110$, $l_2 = 3$. לכן $c_2 + n_2 - 1 = 4 + 3 - 1 = 6$ ואכן מתקיים $6 \leq d = 6$ לכן לא ממשיכים לאיטרציה הבאה ומחזירים את הסימן ה-$6 - 4 + 1 = 3$ באורך 3 שהוא G.

סה"כ קיבלנו שהפענוח של 10001110 הוא AEG. זה אכן תואם לקוד שחישבנו קודם.

אז למדנו לקודד ולפענח קוד האפמן קנוני. אבל כדי לעשות זאת אנחנו צריכים לדעת את אורכי הקידודים לא כל אוסף אורכים שרירותי שנבחר יתאים לייצוא קוד האפמן קנוני. כדי לבחור אורכים בצורה נכונה מייצרים בד"כ קוד האפמן כלשהו ואז משתמשים באורכים שהתקבלו ממנו

קוד האפמן לא יכול לממש את החסם של שנון מאחר שהוא מקודד כל סימן נפרד ובפרא כל סימן חייב להיות מקודד ע"י מספר שלם של ביטים (בפרט מספר הביטים הוא לכל הפחות 1). זה יכול להיות לא יעיל כאשר יש סימנים נפוצים או נדירים באופן מיוחד. למשל, נניח ש-$p_A = 0.99$ ו-$p_B = 0.01$. אז הקידוד של A יהיה נניח 0 והקידוד של B יהיה 1. ואילו החסם של שנון אומר שצריך לפחות $-\log 0.99 \approx 0.015$ לכל סימן A... כלומר מספיק הרבה פחות מאשר ביט אחד!

בכל זאת, את חוסר היעילות של קוד האפמן ניתן לחסום ע"י $p_{s_m} + \log\frac{2\log e}{e} \approx p_{s_m} + 0.086 \leq 1.086$ כאשר s_m הסימן הנפוץ ביותר.

קידוד אריתמטי

כדי להגיע קרוב יותר לחסם של שנון נצטרך לקודד כמה סימנים ביחד. קידוד אריתמטי מקבל קבוצה של סימנים עם תדירויות ומקודד אותם באמצעות מספר (בינארי) ב-$[0,1)$.

האלגוריתם פועל באופן הבא:

<u>ArithmeticEncoding($s_1, \ldots, s_n$)</u>

1. low = 0
2. high = 1
3. for i = 1 to n do
   a. (low, high) = Restrict(low, high, s_i)
4. Return any number in (low, high)

<u>Restrict(low, high, s_i)</u>

1. low_bound = $\sum_{\substack{s\in S \\ s<s_i}} p_s$
2. high_bound = low_bound + p_{s_i}
3. range = high - low
4. new_low = low + range*low_bound
5. new_high = low + range*high_bound

6. return (new_low, new–high)

מה שהפרוצדורה restrict עושה הוא להוציא מהמקטע הנתון את החלק היחסי שמתאים להסתברות של s_i.

נדגים את האלגוריתם עם ייצוג עשרוני אך למעשה יש לבצע את כל החישובים בייצוג בינארי: נניח שיש שלושה סימנים:

Character	A	B	C
Frequency	0.2	0.3	0.5

נקודד את ACCB.

- האות הראשונה היא A שהסתברותה 0.2 ולכן היא מצמצמת את התחום שלנו ל-$(0,0.2)$.
- האות השנייה היא C שהסתברותה 0.5. זה פשוט מצמצם אותנו לחצי הימני של הקטע ומתקבל $(0.1,0.2)$.
- כעת שוב מגיעה C. שוב אנחנו מצטמצמים לחצי הימני של הקטע ומקבלים $(0.15,0.2)$
- לבסוף מגיעה B שהסתברותה 0.3. נחלק את הקטע שלנו לעשרה קטעים שווים באורך 0.005. אז B מצמצמת אותנו לקטע מרכזי באורך 0.015 אשר מתחיל ב-$0.15 + 2 \cdot 0.005 = 0.16$. אז הקטע הסופי הוא $(0.16,0.175)$.

הקוד של ACCB יכול להיות כל מספר בקטע הסופי. למשל, 0.17.

כדי לפענח קידוד אריתמטי יש לדעת את מספר התווים המקודדים הרעיון הוא שבהינתן מספר $k \in [0,1)$ נחפש את רצף הקטעים המקוננים שיכלו להביא אותנו למספר הזה. מאחר שבכל פעם מצטמצמים לאחד מכמה קטעים זרים הרצף הזה (אם קיים) הוא יחיד והוא מהווה את הפענוח של הקוד.

ArithmeticDecoding(k,n)

1. low = 0
2. high = 1
3. for i = 1 to n do
   a. for each $s \in S$ do
      i. (new_low, new_high) = Restrict(low, high, s)
      ii. If new_low $\leq$ k $<$ new_high then
         1. Output s
         2. low = new_low
         3. high = new_high
         4. break

דוגמה: נתונה מילת הקוד 0.34 אשר מקודדת מילה באורך 3.

פשוט נבדוק באופן סידרתי אילו מהאותיות יכלה להביא אותנו למצב הזה.

- באיטרציה הראשונה בקידוד האותיות השונות מחלקות את $[0,1)$ לקטעים $[0,0.2), [0.2,0.5), [0.5,1)$. מילת הקוד נמצאת בקטע האמצעי ולכן באות הראשונה הייתה בהכרח B.
- באיטרציה השנייה הקטע $[0.2,0.5)$ התחלק לקטעים $[0.2,0.26), [0.26,0.35), [0.35,0.5)$. מילת הקוד שוב נמצאת בקטע האמצעי ולכן גם האות השנייה הייתה בהכרח B.
- באיטרציה השלישית הקטע $[0.26,0.35)$ מתחלק לקטעים $[0.26,0.278), [0.278,0.305), [0.305,0.35)$. מילת הקוד נמצאת במקטע האחרון ולכן האות האחרונה שאנחנו מעוניינים בה היא C.

לסיכום, נחשב מהו אורך הקידוד נוצר בשיטה זו. כדי לשמור מספר בקטע בגודל $high - low$ צריך $-\log(high - low)$ ביטים. במקרה שלנו הגודל של הקטע הסופי הוא $\Pi_{i=1}^{n} p_{s_i}$ ולכן כדי לשמור מספר בקטע זה אשר מהווה את הקידוד צריך $-\log \Pi_{i=1}^{n} p_{s_i} = \sum_{i=1}^{n} -\log p_{s_i}$. זה כבר יותר קרוב לחסם של שנון..

קידוד אריתמטי אדפטיבי

הקידוד אריתמטי צריך לדעת את הסתברות המופע של כל תו. את ההסתברויות האלה צריך לחשב ולשמור, מה שמבזבז זמן ומקום. יתר על כן, ההסתברויות יכולות להשתנות לאורך הטקסט. למצב כזה כלל לא נתנו מענה עד כה.

בקידוד אריתמטי אדפטיבי העת הקידוד ובעת הפענוח מחשבים את ההסתברויות on-the-fly ככל שמתקדמים בטקסט. החישוב מתבצע ע"י ספירת מופעי התווים.

במהלך הקידוד החישוב נעשה אופן הבא: לפני תחילת הקידוד לכל אחד מהתווים השונים $a_1, \ldots, a_m$ יש הסתברות זהה $\frac{1}{m}$. כעת נניח שקראנו סה"כ n תווים ולכל תו a_i קיבלנו עד כה הסתברות $p_i = \frac{f_i}{n}$. אם התו ה-$n+1$ הנקרא הוא a_j אז נעדכן $p_j = \frac{f_j+1}{n+1}$ ו-$p_i = \frac{f_i}{n+1}$ לכל $i \neq j$. ההסתברויות האלה משמשות בכל שלב לחלוקת הקטע שאנחנו נמצאים בו בדיוק כמו בקידוד האריתמטי הרגיל.

לדוגמה, נקודד את bccb שלקוח מתוך הא"ב {a,b,c}:

- בשלב הראשון ההסתברות של כל אחת מהאותיות היא $\frac{1}{3}$. לכן כאשר קוראים את b הראשונה אנחנו צריכים להצטמצם לקטע $\left[\frac{1}{3}, \frac{2}{3}\right)$.
- כעת מאחר שקראנו מופע אחד של b ההסתברויות מתעדכנות להיות $p_a = \frac{1}{4}, p_b = \frac{2}{4}, p_c = \frac{1}{4}$ (נשים לב שסכום ההסתברות הוא עדיין 1). לכן כאשר נקרא את האות הבאה c נצטרך להצטמצם לרבע האחרון של הקטע שלנו, כלומר ל-$\left[\frac{7}{12}, \frac{2}{3}\right)$.
- אחרי שקראנו את c ההסתברויות מתעדכנות להיות $p_a = \frac{1}{5}, p_b = \frac{2}{5}, p_c = \frac{2}{5}$. ובגלל שקוראים עוד c הפעם צריך להצטמצם ל-$\frac{2}{5}$ מסוף הקטע שלנו. לכן נצטמצם ל-$\left[\frac{19}{30}, \frac{2}{3}\right)$.
- ההסתברויות מתעדכנות להיות $p_a = \frac{1}{6}, p_b = \frac{2}{6}, p_c = \frac{3}{6}$ ולכן אחרי קריאת b נצטמצם לבסוף לקטע $\left[\frac{23}{36}, \frac{13}{20}\right)$. הקידוד הסופי יהיה מספר כלשהו בטווח הזה, למשל 0.64.

הפענוח נעשה בדיוק באותו אופן כמו בפענוח של קידוד אריתמטי רגיל ועדכון ההסתברויות נעשה כמו שתיארנו למעלה. פשוט בכל שלב סודקים מהי האות הבאה האפשרית. מאחר שכל שלב מחלק את הקטע תת-קטעים זרים יש רק אפשרות אחת נכונה.

שמירת מבנה הרשת

היינו רוצים לתכנן מבנה נתונים שיאפשר לענות באופן יעיל את שאילתות שקשורות למבנה הרשת– איזה אתרים מצביעים לאתר נתון ולאיזה אתרים מצביע אתר נתון. ליכולת כזאת יכולים להיות כמה שימושים.

למשל, ניהול crawling, ניתוח גרף האינטרנט, ניתוח של קישורים ועוד. בהמשך נראה שזה גם יכול לשמש אותנו לקביעת דירוג של תוצאות שאילתא.

אנחנו נלמד את המאמר *The Webgraph Framework I: Compression Techniques* של **Boldi** ו- **Vigna**. המטרה שלהם הייתה לשמור רשימת שכנויות של הצמתים בגרף האינטרנט בזיכרון הראשי. הרשימות האלה יכולות להיות ארוכות מאוד ולכן דחיסה יעילה היא קריטית בהקשר זה.

באופן נאיבי ניתן לשמור רשימה ארוכה של כל זוגות הצמתים השכנים. אם כל URL מיוצג ע"י מספר שלם אז אם יש 118M דפים באינטרנט נצטרך 27 ביטים כדי לייצג כל URL. לכן כל זוג שכנים, כלומר כל קישור יתפוס 54 ביטים. אפילו אם כל דף מצביע רק לדף אחד נקבל שצריך $118 \cdot 10^6 \cdot 54 = 6372 \cdot 10^6$ ביטים שהם למעשה כ-800MB. זה די הרבה. יתר על כן, ברור שיש הרבה יותר שכנים שצריך לשמור כי דפים בד"כ מצביעים ליותר מדף אחד.

במקום לשמור זוגות אפשר לשמור לכל דף רשימת שכנים. זה מצמצם את הזיכרון הנחוץ פי שניים לכל קישור, שהרי צד אחד של הקישור נשמר רק פעם אחת כעת

השיטה שמוצגת במאמר מאפשרת לשמור קישור בשימוש בשלושה ביטים בממוצע!

תכונות מעניינות של קישורים

לקישורים יש כמה תכונות שניתן להשתמש בהם כדי לייעל את אופן השמירה.

1. לוקאליות: רוב הקישורים מובילים את המשתמש לדפים באותו השרת לכן אם ה-URL-ים שמורים לפי סדר א"ב אז האינדקס של המקור והמטרה יהיו קרובים.
2. דמיון: ל-URL-ים אשר דומים לקסיקוגרפית יש הרבה עוקבים משותפים.
3. רציפות: הרבה פעמים לעוקבים רבים של דף נתון יש כתובות עוקבות.

ייצוג נאיבי

נשמור טבלה עם מספר המשאב, מספר הקישורים שלו ורשימת העוקבים:

Node	Outdegree	Successors
...	...	...
15	11	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041
17	0	
18	5	13, 15, 16, 17, 50
...	...	...

הטבלה הזאת לא באמת נשמרת בזיכרון כטבלה עם שורות שניתן לגשת אליהם אקראית אלא באופן רציף בזיכרון. לכן, כמו במקרה של האינדקס המהופך צריך לשמור את דרגת היציאה של כל קודקוד. זו אומרת לנו מתי להפסיק לקרוא את רשימת השכנים. ללא דרגת היציאה לא נוכל לקרוא את ה"טבלה" בצורה נכונה.

רשימת הפרשים

כבר דיברנו בעבר שיכול להיות יתרון בשמירת הפרשים, שהרי המספרים שם נוטים להיות קטנים יותר. לכן, עדיף למעשה לשמור את הנתונים כרשימת הפרשים. אלא שברשימת הפרשים רגילה המספר הראשון נשמר כמו שהוא. נראה לחסוך גם בשמירת מספר זה. אז עבור קודקוד x, המספר הראשון ברשימה שלו יהיה $s_1 - x$ כאשר s_i הוא השכן ה-i. אבל $s_1 - x$ יכול להיות מספר שלילי. לכן נפעיל עליו את הפונקציה

$$v(y) = \begin{cases} 2y, & y \geq 0 \\ 2|y| - 1, & y < 0 \end{cases}$$

אז כדי לפענח את המספר הראשון ברשימה יש לבדוק את הזוגיות שלו ואז לבודד את s_1 בהתאם.

כדי לחסוך אפילו יותר, נשים לב שברשימה אין מסמך שמופיע פעמיים. לכן כל ההפרשים הם לכל הפחות 1. לכן נוכל להחסיר 1 מכל ההפרשים, להישאר עם ערכים אי שליליים ועדיין נוכל לפענח את הרשימה. אז למעשה עבור קודקוד x נשמור את הרשימה $S(x) = \{v(s_1 - x), s_2 - s_1 - 1, \dots, s_k - s_{k-1} - 1\}$:

Node	Outdegree	Successors
...	...	...
15	11	3, 1, 0, 0, 0, 0, 3, 0, 178, 111, 718
16	10	1, 0, 0, 4, 0, 0, 290, 0, 0, 2723
17	0	
18	5	9, 1, 0, 0, 32
...	...	...

דחיסת הפניות

במקום לשמור את $S(x)$ ניתן לייצג אותו כגירסה מעודכנת של $S(y)$ עבור $x < y$ כלשהו. אז $x - y$ הוא מספר ההפניה שמאפשר לנו למצוא את הרשימה הקודמת. Copy List היא רצף של ביטים באורך $S(y)$ שאומר איזה מהערכים של $S(y)$ נמצאים גם ב-$S(x)$. חוץ מה copy list-צריך לשמור גם את אוסף השכנים החסרים $S(x) \setminus S(y)$. כדי לציין שאין שימוש בהפניה נקבע את מספר ההפניה להיות 0.

כך, רשימת ההפרשים הופכת להיות:

Node	Outd.	Ref.	Copy list	Extra nodes
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

כאן עבור צומת 15 לא השתמשנו ברשימת הפניות ולכן היא מופיעה בשלמותה. השאר כבר משתמשים בהפניות לצמתים קודמים.

דחיסה דיפרנציאלית

אפשר לחסוך ולא לשמור את רשימת ההפניות כרצף ביטים באורך $S(y)$ אלא להסתכל עליה כסדרת מתחלפת של בלוקים עם 1 ובלוקים עם 0. נניח שהבלוק הראשון הוא תמיד בלוק של 1 (אולי באורך אפס). אז נשמור את מספר הבלוקים (כרגיל כדי שנדע מתי להפסיק לקרוא), את אורך הבלוק הראשון ואת אורכי שאר הבלוקים פחות 1 (מאחר שהבלוקים האחרים הם באורך לכל הפחות 1, ניתן לחסוך ולשמור את אורכים פחות 1 ועדיין לקבל מספרים אי-שליליים). גם ממספר הבלוקים אפשר להחסיר 1 כי האורך הוא כמובן לכל הפחות 1. כמו כן, ניתן לא לשמור את אורך הבלוק האחרון משום שניתן להסיק אותו מהאורך של$S(y)$ וסכום אורכי הבלוקים הקודמים. כך נקבל את הייצוג הבא:

Node	Outd.	Ref.	# blocks	Copy blocks	Extra nodes
...	...	...	...	...	...
15	11	0			13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	7	0, 0, 2, 1, 1, 0, 0	22, 316, 317, 3041
17	0				
18	5	3	1	4	50
...	...	...	...	...	...

דחיסת רשימת הצמתים החסרים

נשתמש ברציפות כדי לדחוס את רשימת הצמתים החסרים. נמצא רצפים של מספרים עוקבים שאורכם לפחות חסם כלשהו שנבחר L_{min}. אז לכל רצף כזה אפשר לשמור את המספר הכי קטן שלו ואת אורך הרצף. את הערכים המינימאליים של הרצפים שומרים ברשימת הפרשים ומחסירים 2 מכל הפרש (שהרי ההפרש בין כל שני ערכים כאלה הוא לפחות 2). את אורכי הרצפים שומרים אחרי החסרה של L_{min}. לבסוף, את השאריות נשמור ברשימת הפרשים. למשל, אם $L_{min} = 2$ נקבל בדוגמה שלנו:

Node	Outd.	Ref.	# blocks	Copy blocks	# intervals	Left extremes	Length	Residuals
...	...	...	...	...	...	...	...	...
15	11	0			2	0, 2	3, 0	5, 189, 111, 718
16	10	1	7	0, 0, 2, 1, 1, 0, 0	1	600	0	12, 3018
17	0							
18	5	3	1	4	0			50
...	...	...	...	...	...	...	...	...

נעיר לסיכום, שאת מספר הצומת לא באמת צריך לשמור. השורה הראשונה מתאימה לצומת הראשון ובגלל שאנחנו שומרים את כל אורכי הרשימות ידוע מתי שורה נגמרת כך אנחנו חוסכים גם בשמירת מספרי הצמתים.

במאמר עצמו ניתן לראות את החישוב של מספר הביטים לכל קישור.

בניית האינדקס

עד כה דנו בכמה מבני נתונים שיכולים להחזיק את כל המידע שאנחנו מעוניינים בו השווינו זמני ריצה וסיבוכיות מקום. הנחנו שהמבנה קיים ואנחנו משתמשים פה. אבל עולה גם השאלה הטבעית של בניית המבנה מלכתחילה. יצירת מבני הנתונים האלה אינה משימה פשוטה. קודם כל, יכול להיות שיש צורך בהרבה מאוד מקום אחסון והמבנה כולו לא נכנס בזיכרון הראשי. אז אנחנו עלולים להיתקל ב-page faults מרובים ולבזבז הרבה מאוד זמן על בניית המבנה. יתר על כן, כל הזמן הסתמכנו על כך שמבני הנתונים שמורים בזיכרון בצורה רציפה. אבל למשל בעת הבנייה של האינדקס המהופך אנחנו לא יכולים לדעת כמה זיכרון צריך להקצות לכל posting list. אכן הבעיה העיקרית היא בבניית האינדקס המהופך. בניית הלקסיקון לא מאוד קשה כי הוא לא מאוד גדול ואפשר להסתפק בקריאה בודדת של כל המסמכים

ננסה להציע אינטואיציה לפיתרון יחסית יעיל של בניית האינדקס המהופך. כדי ליצור את האינדקס צריך לקרוא את הקבצים ולפרק (parse) אותם. בזמן הקריאה אפשר לכתוב את כל הזוגות מהצורה (wordID, docID). זה יעיל כי אנחנו כל הזמן כותבים לסוף של קובץ (לא בודקים חזרות, פשוט כותבים כל מילה שנתקלים בה).

כעת צריך למיין את הזוגות לפי wordID ואז לפי docID ואז אפשר ליצור את הרשימות במעבר בודד על הרשימה הממוינת. נשאלת אז השאלה כיצד למיין באופן יעיל כל כך הרבה נתונים (הבעיה היא שלא כל הזוגות נכנסים לזיכרון הראשי בבת אחת). לא נדון בנושא זה בהרחבה כאן אבל נציין שלמדנו איך עושים את זה בקורס הראשון במסדי נתונים...

אינדקסים עבור שאילתות עם תווי-כל

ראינו איך בהינתן מילה אנחנו יכולים למצוא אותה בלקסיקון בעזרת חיפוש בינארי! אבל זה אפשרי רק אם המילה ידועה במלואה. אנחנו נרצה להוסיף יכולת לתמוך בשאילתות עם תווי-כל (wildcards). זה כמובן אפשרי במבנה הנוכחי של האינדקס אבל זה לא מאוד יעיל:

- כדי למצוא את המילים שמתאימות ל-*lab ניתן למצוא בחיפוש בינארי את המילה הראשונה שמתחילה ב-lab ואז לקרוא ברצף אחריה את כל המילים המתאימות. זה עדיין יעיל.
- כדי למצוא את המילים שמתאימות ל-or* אין ברירה אלא לעבור ליניארית על כל הלקסיקון. אפשר גם לשמור שתי גרסאות של הלקסיקון: לקסיקון ישר ולקסיקון הפוך. אבל זה תופס מקום יקר ויותר

חשוב מזה, זה לא באמת פותר את הבעיה הכללית של שאילתות עם תווי-כל, אלא מותאם באופן אישי לשאילתות שבהם ידוע רק הסוף של המילה.

- כדי למצוא את המילים שמתאימות ל-lab*r צריך לעשות חיתוך בין שתי השאילתות הקודמות

ברור שהיינו רוצים למצוא מנגנון כללי יותר ויעיל שיאפשר לטפל בשאילתות מסוג זה

יצירת אינדקס בעזרת n-grams

n-gram הוא רצף של n אותיות במילה. נסמן ב-$ את ההתחלה והסוף של מילה. אז למשל ה-digrams של labor הם l,la,ab,bo,or,r.

נשמור מבנה נוסף – אינדקס מהופך של digrams שלכל digram מכיל רשימה של המילים שהוא מופיע בהן. למעשה, זוהי רשימה של מצביעים לתוך הלקסיקון. רשימת ה-digrams ממוינת לקסיקוגרפית. למשל:

Word ID	Word
1	abhor
2	bear
3	laaber
4	labor

Digram	Word IDs
$a	1
$b	2
$l	3,4
aa	3
ab	1,3,4
ar	2
be	2,3
…	
la	3,4
…	
r$	1,2,3,4

כעת, כדי לענות על השאילתה lab*r נרצה למצוא את המילים שמתאימות לכל ה-digrams של השאילתא - l,la,ab,r. בגלל שרשימת ה-digrams ממוינת לפי א"ב ניתן בקלות למצוא בחיפוש בינארי את כל הרשימות שמעניינות אותנו:

Digram	Word IDs
$a	1
$b	2
$l	3,4
aa	3
ab	1,3,4
ar	2
be	2,3
…	
la	3,4

...	
r$	1,2,3,4

הרשימות שמורות ממוינות ולכן בזמן ליניארי ניתן למצוא את החיתוך שלהן. במקרה שלנו המילים המתאימות הן 3 ו-4. לבסוף צריך לוודא שהמילים אכן מתאימות לשאילתא, שהרי השיטה הזו לא מגבילה את סדר ה-digrams במילים וגם לא מתייחסת לאותיות נוספות שאולי נמצאות שם אז המועמדות שלנו הן laaber ו-labor. ברור שרק האחרונה מתאימה לנו.

יצירת אינדקס בעזרת לקסיקון מסובב

אם ידוע ששאילות עם תווי-כל הן נפוצות אולי נרצה לחסוך זמן על חשבון מקום אחסון. בעזרת אינדקס מסובב ניתן למצוא את המילים שמתאימות לכל שאילתא עם תו-כל יחיד בעזרת חיפוש בינארי בודד.

לכל אות (כולל הסימן $ שמציין תחילה של מילה) של הלקסיקון ניצור רשומה באינדקס המסובב הרשומה תתאים לאותה מילה אך מסובבת. לכל מילה מסובבת כזאת נשמור את הכתובת שלה – מספר המילה שלה וגודל הסיבוב. שני המספרים האלה מאפשרים לחשב את המילה המסובבת ולכן אין צורך לשמור אותה. למשל, במקרה שלנו:

Word ID	Word
1	abhor
2	bear
4	labor

Rotated Form (for clarity)	Address
$abhor	(1,0)
$bear	(2,0)
$labor	(4,0)
abhor$	(1,1)
abor$l	(4,2)
ar$be	(2,3)
bear$	(2,1)
bhor$a	(1,2)
bor$la	(4,3)
...	
r$abho	(1,5)
r$bea	(2,4)
r$labo	(4,5)

כעת ניתן למצוא את המילים המתאימות לשאילתא בקלות

- כדי למצוא את המילים שמתאימות ל-lab* נחפש באינדקס המסובב את המילה המסובבת הראשונה שמתחילה ב-lab$. אז ברצף אחריה יופיעו כל המילים המתאימות
- כדי למצוא את המילים שמתאימות ל-*or נשים לב שלמעשה מספיק למצוא את המילים שמתאימות לשאילתא $*or או *or$ לכן בדומה לשאילתא הקודמת מספיק למצוא באינדקס המסובב את המילה המסובבת הראשונה שמתחילה ב-or$ ואז שאר המילים יופיעו ברצף אחריה.
- כדי למצוא את המילים שמתאימות ל-r*l נשים לב שלמעשה מספיק למצוא מילים שמתאימות לשאילתא *l$r ונמשיך כמו קודם.

- כדי לענות על שאילתות שבהן יש יותר מתו-כל אחד ניתן לחלק אותה לחלקים שבהם רק תו-כל אחד, לבצע חישוב בנפרד לכל חלק ואז לחתוך את התוצאות

אינדקס לצירופי מילים

עד כה דיברנו על אינדקסים שתומכים בשאילתות שבהן אין הגבלה על המקום של המילים השונות במסמך. היינו רוצים לתמוך גם בשאילתות של צירופי מילים, כלומר בשאילתות שבהן יש דרישה שהמילים מופיעות בסדר מסוים וצמודות אחת לשנייה.

הערנו בתחילת הדיון על אינדקסים שניתן לשמור לא רק את רשימת המסמכים שכל מילה מופיעה בהםאלא גם את המיקומים של המילה בתוך המסמך. זה דורש הרבה יותר מקום אחסון אך זה מאפשר לענות על שאילתות מסוג זה ע"י מציאת התוצאה לשאילתא כאילו המילים לא מהוות צירוף בודד ואז בעיבוד נוסף לבדוק את המיקומים של המילים. זה יקר גם מבחינת מקום אחסון וגם מבחינת זמני ריצה.

נציע מבנה נתונים Biword Index אשר תומך ביעילות במציאת צירופים בני שתי מילים. הרעיון הוא להרחיב את הפתרון שאנחנו כבר מכירים. במקום להתייחס לכל מילה בנפרד פשוט נתייחס לזוגות מילים סדורים כמונחים בלקסיקון. לכל זוג סדור נשמור את רשימות המסמכים שהוא מופיע בהם. הפרטים זהים לפרטים של כל מבני האינדקס שדיברנו עליהם.

האפשרות הכי פשוטה היא שהלקסיקון יכיל זוגות מילים ומצביע לרשימות ההפרשים שלהם למשל, נניח שנתונים המסמכים הבאים:

- מסמך 1: בהצלחה רבה במבחן
- מסמך 2: עידו עבר בהצלחה רבה
- מסמך 3: קל במבחן קל

אז הלקסיקון יכיל את כל זוגות המילים הסדורים שמופיעים במסמכים ממוינים לפי א"ב:

Lexicon: Biword	Lexicon: Pointer to list	Inverted Index
בהצלחה רבה	→	1,2
במבחן קל	→	3
עבר בהצלחה	→	2
עידו עבר	→	2
קל במבחן	→	3
רבה במבחן	→	1

כל מבנה זה ניתן להפעיל את כל האופטימיזציות שדיברנו עליהם קודם כמו שירת רשימת הפרשים, קידוד קדמי וכו'.

ניתוח המקום שהאינדקס לוקח נעשה בצורה זהה לניתוח שעשינו במקרה של מילה בודדת אלא שעכשיו יש לספור את מספר זוכות המילים ואורכי זוגות מילים. החיפוש בלקסיקון הוא כמובן בזמן לוגריתמי במספר הזוגות. נשים לב שאם יש בסה"כ n מילים שונות אז מספר הזוגות הסדורים השונים הוא לכל היותר n^2 ולכן החיפוש בלקסיקון לוקח $O(\log n^2) = O(2\log n) = O(\log n)$. אז אמנם הלקסיקון תופס יותר מקום עכשיו אבל זמן החיפוש נשאר לוגריתמי במספר המילים המקורי. כמו כן, לא סביר להניח שאכן כל זוג מילים הוא צירוף אפשרי. יש הרבה צירופי מילים שבמסמך אמיתי לא יכולים להופיע ביחד (למשל "דלת מעופפת" או "האיש הלכה"). לכן יכול להיות שגם סיבוכיות המקום לא תממש את המקרה הגרוע ביותר..

לסיכום נעיר שהשיטה הזו טובה למקרה שאנחנו מעוניינים רק בצירופים של שתי מילים, אך לא ניתן להרחיב אותה בקלות למקרה שנרצה לעשות חיפוש של יותר משתי מילים או מילים שאינן צמודות דווקא הפיתרון ששומר גם את המסמכים וגם את המיקומים של המילים במסמכים ניתן להרחבה לתמיכה בסוגי שאילתות מרובים יותר.

עיבוד מקדים

עד כה הנחנו שבאורח פלא נתונות לנו כל המילים שצריך להכניס לאינדקס. אבל לא המצב בד"כ. ישנן כמה החלטות שלצריך לעשות בנוגע למילים. האופן שבו המידע שמור בלקסיקון ישפיע על תוצאות השאילתא כמו כן, גם כשמתקבלת שאילתא אולי נרצה לעשות עליה עיבוד מקדים כלשהו

Case Folding

הדוגמה הכי פשוטה לפעולה מקדימה שלרוב עושים על המילים בלקסיקון ועל המילים בשאילתא היא case folding – אם המילים בשפה שיש בה אותיות גדולות ואותיות קטנות, לפני ההכנסה של מילים ללקסיקון הופכים את כל האותיות לקטנות. גם לפני שמחשבים את השאילתא הופכים את כל האותיות בה לקטנות זה מקטין את גודל הלקסיקון ויוצר אחדות מסוימת הרי אם מישהו החליט לכתוב בעמוד שלו nOtEs-HeAvEn לא נרצה לשמור צורה זו בלקסיקון. מאידך, זה יכול ליצור בלבול בעיקר בחיפוש של קיצורים וראשי תיבות למשל המילה us (אנחנו) לעומת הקיצור US (ארצות הברית). לא ניתן להבחין בין שתי מילים אלה כאשר עושים case folding.

Stemming

עוד פעולה שיכולה לעזור במציאת תוצאות רלוונטיות היא חישוב שורש של המילה. אם למשל השאילתא היא "running shoe" יכול להיות שנרצה גם להחזיר דפים שמופיעים בהן המילים run, runs, shoe. כדי לתמוך במטרה זו אפשר להשתמש ב-Stemmer שבהינתן מילה מחזיר את השורש שלה. שימוש במנגנון כזה גורם לזה שמוחזרות יותר תוצאות רלוונטיות אך גם יותר תוצאות לא רלוונטיות Stemmer שנהוג להשתמש בו נקרא Porter Stemmer. הוא מבצע אוסף מניפולציות על מילה שבסופו נשאר "שורש" שמייצג את המילה. אנחנו לא נתעמק בכללי הפעולה שלו. ניתן למצוא אותם באינטרנט.

Stop Words

Stop Words הן מילים מאוד נפוצות שלרוב אין להן חשיבות גדולה והן מופיעות כמעט בכל מסמך נתון למשל a, the, to וכו'. מילים כאלה תופסות המון מקום באינדקס כי רשימות המסמכים שלה ארוכות מאוד (כנראה מכילות את כל מיליוני המסמכים שקיימים במנוע החיפוש) וגורמות להאטה בחישוב שאילתות (גם כי צריך לחפש יותר מילים וגם כי צריך לחתוך את הרשימות המאוד ארוכות שלהן עם הרשימות של שאר המילים). יתר על כן, שוב, מאחר שהן מופיעות כמעט בכל המסמכים הן לא משפרות בד"כ את איכות התוצאות. לכן ישנן מנועי חיפוש שכלל לא שומרים מילים אלה ומורידים אותם מהשאילתות. למשל, כשביצעתי ב-Google חיפוש של "I went to the park", המילים שהוא למעשה חיפש באינדקסים שלו היו רק I, went ו-park:

Results **1** - **10** of about **108,000,000** for I went **to the** park. (**0.22** seconds)

מאידך, יכול להיות שנרצה למצוא שאילתא שרוב המילים שלה הן דווקא מהסוג הזה למשל, נתבונן בשאילתא "to be or not to be". כאן לכאורה Google חישב את השאילתא

Results **1** - **10** of about **1,010,000,000** for to be **or not** to be. (**0.12** seconds)

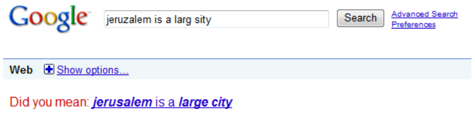
אבל מעניין לציין שהשאילתות "to be" ו-"to be to be" לא הניבו את התוצאות הרצויות. אז כנראה ש-Google בכל זאת מתייחסים איכשהו ל-Stop words.

תיקון שגיאות כתיב

לתיקון שגיאות כתיב יש שני שימושים עיקריים:

- תיקון המסמכים שנכנסים לאינדקס
    a. שגיאות יכולות להגיע בגלל שימוש במנגנוני OCR[6]. שם יכול לקרות בלבול בין אותיות דומות כמו O ו-D.
    b. שגיאות יכולות גם להגיע מטעויות הקלדה שבהם יש בלבול בין אותיות סמוכות במקלדת למשל במקלדות QWERTY ניתן להקליד בטעות I במקום O.

  כדאי להתאים את מנגנון תיקון השגיאות למקור שממנו הן מגיעות. כשמדובר בקובץ שהגיעה מ-OCR מאוד לא סביר שיקרה בלבול בין I ל-O ואילו במקרה של שגיאות הקלדה לא סביר שיקרה בלבול בין D ל-O.
- תיקון שגיאות כתיב בשאילתות שמגיעות. יש שתי אפשרויות לטיפול בשגיאות בשאילתות
    a. ניתן פשוט לתקן את השגיאה ולהחזיר את תוצאת השאילתא המתאימה
    b. ניתן להציע למשתמש מספר חלופות לבחור מתוכן. כמו למשל ב-Google:



ישנם שני סוגי תיקונים שניתן להפעיל:

- תיקון שפועל על כל מילה באופן מבודד ללא קשר למילים האחרות
- תיקון בהתאם להקשר שהמילה מופיעה בו (למשל במשפט I flew *form* Tel-Aviv to Rome רק תיקון שמשתמש בהקשר יעזור)

תיקון מבודד

נדון בהרחבה בתיקון שגיאות במילים מבודדות. הנחת היסוד היא שיש לקסיקון שממנו לקוחות המילים מאויתות כהלכה. הלקסיקון הזה יכול להגיע משני מקורות: לקסיקון סטנדרטי (Webster או לקסיקון של תחום מסוים) או לקסיקון שנבנה מהמסמכים שבאינדקס (כולל המילים שלא איותו כהלכה).

בבעיית תיקון השגיאות יש לקסיקון ובהינתן רצף תווים נרצה למצוא את המילה בלקסיקון הקרובה ביותר לרצף התווים הזה. כדי שנוכל לפתור את הבעיה יש להגדיר ראשית מה המשמעות של קרבה בין רצפי תווים בהקשר זה. אנחנו נדון בכמה מדדי קרבה.

מרחק עריכה

בהינתן שתי מחרוזות s_1, s_2 מרחק העריכה (Edit Distance) ביניהן מוגדר להיות המספר המינימאלי של פעולות בסיסיות (הכנסת אות, מחיקת אות, החלפת אות) שיש לבצע על s_1 כדי לקבל את s_2. לדוגמה מרחק העריכה בין cat ל-dog הוא 3 ואילו בין dog ל-dot המרחק הוא 1 (יש רק להחליף את g ב-t).

---

[6] Optical Character Recognition - התהליך כרוך בסריקה או בצילום של טקסט המקור לקובץ תמונה, והפעלה של תוכנה מתאימה שמזהה בתמונה את התווים השונים המרכיבים את הטקסט, וממירה כל אחד מהם לתו יחיד בקובץ טקסט.

את מרחק העריכה ניתן לחשב באמצעות תכנון דינאמי. נניח שנרצה לחשב את המרחק בין s ל-t. נסמן ב-s_j את הרישא באורך j של s וב-t_i את הרישא באורך i של t. ניצור טבלה בגודל $(1+|s|)\times(1+|t|)$ ובתא ה-i,j שלה יופיע מרחק העריכה בין t_i ל-s_j. אז המרחק בין s ל-t יופיע בתא האחרון בטבלה. נתחיל למלא את הטבלה מהרישות הקצרות אל הארוכות. ביתר פירוט:

1. $n = |s|, m = |t|$
2. If $n = 0$ return m and exit
3. If $m = 0$ return n and exit
4. Construct a matrix D containing $m + 1$ rows and $n + 1$ columns
5. Initialize the first row to $0, \dots, n$
6. Initialize the first column to $0, \dots, m$
7. For $i = 1, \dots, n$ (for each character of s)
   a. For $j = 1, \dots, m$ (for each character of t)
      i. If $s(i) = t(j)$ then
         1. $c = 0$
      ii. Else
         1. $c = 1$
      iii. $d_{i,j} = \min\{d_{i-1,j} + 1, d_{i,j-1} + 1, d_{i-1,j-1} + c\}$
8. Return d_{nm}

דוגמה: נניח שאנחנו רוצים למצוא את המרחק בין gumbo ל-gambol (אינטואיטיבית נראה שהמרחק הוא 2):

		G	U	M	B	O
	0	1	2	3	4	5
G	1	$\min\{1+1,1+1,0+0\}=0$	$\min\{2+1,0+1,1+1\}=1$	$\min\{3+1,1+1,2+1\}=2$	$\min\{4+1,2+1,3+1\}=3$	$\min\{5+1,3+1,4+1\}=4$
A	2	$\min\{0+1,2+1,1+1\}=1$	$\min\{1+1,1+1,0+1\}=1$	$\min\{2+1,1+1,1+1\}=2$	$\min\{3+1,2+1,2+1\}=3$	$\min\{4+1,3+1,3+1\}=4$
M	3	$\min\{1+1,3+1,2+1\}=2$	$\min\{1+1,2+1,1+1\}=2$	$\min\{2+1,2+1,1+0\}=1$	$\min\{3+1,1+1,2+1\}=2$	$\min\{4+1,2+1,3+1\}=3$
B	4	$\min\{2+1,4+1,3+1\}=3$	$\min\{2+1,3+1,2+1\}=3$	$\min\{1+1,3+1,2+1\}=2$	$\min\{2+1,2+1,1+0\}=1$	$\min\{3+1,1+1,2+1\}=2$
O	5	$\min\{3+1,5+1,4+1\}=4$	$\min\{3+1,4+1,3+1\}=4$	$\min\{2+1,4+1,3+1\}=3$	$\min\{1+1,3+1,2+1\}=2$	$\min\{2+1,2+1,1+0\}=1$
L	6	$\min\{4+1,6+1,5+1\}=5$	$\min\{4+1,5+1,4+1\}=5$	$\min\{3+1,5+1,4+1\}=4$	$\min\{2+1,4+1,3+1\}=3$	$\min\{1+1,3+1,2+1\}=2$

ניתן להעדיף פעולות בסיסיות מסוימות ע"י מתן משקלים לפעולות. למשל אם המסמך מגיע מ-OCR ניתן לתת משקל נמוך יותר לחילוף של O ו-D ובכך להעדיף תיקון כזה מאשר למשל תיקון של O ב-L. אז צריך לתת לאלגוריתם מטריצת משקלים שמגדירה מה המשקל של כל פעולה – הוספת אות בודדת, מחיקת אות בודדת והחלפת זוג אותיות.

חפיפת n-grams

חישוב מרחק העריכה בין שתי מילים באורך n ו-m לוקח זמן $O(nm)$. אם כדי לתקן מילה נצטרך לחשב את המרחק שלה אל כל מילות הלקסיקון זה ייקח יותר מדי זמן היינו רוצים למצוא דרך לצמצם את רשימת המועמדים.

דרך פשוטה לצמצם את המועמדים היא לבדוק את מספר ה-n-grams המשותפים למילת השאילתא ולמילים בלקסיקון. כמו בשאילתות עם תווי-כל, נשמרו אינדקס של n-grams. בהינתן מילה, נחלק אותה ל-n-grams ונמצא את כל המילים בלקסיקון שמתאימות ל-n-grams של המילה. ניתן לקבוע חסם שדורש שתהיה חפיפה לפחות בגודל חסם זה. את המילים שמצאנו נבדוק בצורה פרטנית בעזרת האלגוריתם הדינאמי.

לדוגמה, נסתכל ב-trigrams של december ושל november:

december: dec, ece, cem, emb, mbe, ber

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

יש לשתי המילים שני trigrams משותפים.

היתרון של שיטת החפיפה הוא שהיא דורשת מעט מאוד זמן חישוב – זמן ליניארי באורכי המילים ולא מכפלת האורכים!

הבעיה עם השיטה הזאת היא שהתוצאה שהיא נותנת תלויה באורך המילים. היינו רוצים למצוא מדד מנורמל כלשהו של החפיפה. מקדם Jaccard מהווה מדד נפוץ לחפיפה. יהיו X ו-Y שתי קבוצות. אזי מקדם Jaccard שלהן מוגדר להיות היחס $\frac{|X\cap Y|}{|X\cup Y|}$. המקדם שווה ל-1 כאשר הקבוצות זהות והוא מתאפס כאשר הן זרות. היתרון של השיטה הזו הוא שהיא תמיד נותנת תוצאה בין 0 ל-1. אז ניתן לבחור חסם קבוע שיוכל לעבוד לכל זוגות המילים ללא תלות באורך שלהן.

למרות שמצאנו דרך קצת יותר יעילה לתיקון שגיאות מאשר מעבר על כל המילים, התהליך הזה בכל זאת יקר ואולי כדאי להימנע מלבצע אותו שלא לצורך. למשל, ניתן להריץ אותו רק על שאילתות שהתאימו למספר קטן מאוד של מסמכים...

עיבוד שאילתות

שאילתא היא צירוף בוליאני של מילים בשימוש בפעולות הלוגיות AND, OR ו-NOT.

שאילתות גימום

שאילתת גימום משתמשת רק בפעולת AND. ניתן לענות על שאילתות גימום באמצעות מציאת המסמכים שמתאימים לכל מילה בנפרד ולאחר מכן חיתוך הרשימות (כבר ציינו זאת מספר פעמים). אבל לסדר שבו חותכים את הרשימות יש השפעה על זמני הריצה.

<u>Conjunctive Query</u>

1. For each query term t
   a. Stem t
   b. Search the lexicon for the stem
   c. Record frequency f_t and pointer to list l_t
2. Sort terms in ascending order of frequency
3. Let t be the least frequent term
4. Read list l_t and set $C = l_t$

5. For each remaining term t' in the ordered list of term
    a. Read the list $l_{t'}$
    b. Set $C = C \cap l_{t'}$
6. For each document in C
    a. Find the address of the document and return it to the user

מה שצריך לשים לב אליו הוא שחישוב חיתוך הרשימות מעשה מהרשימה הקצרה אל הארוכה. זה חוסך זמנים משום שידוע שהתוצאה תהיה לכל היותר באורך הרשימה הקצרה ביותר. מאחר שחישוב חיתוך שתי רשימות ממוינות נעשה בזמן ליניארי בסכום האורכים עדיף לנו לשמור על אורכי הרשימות קצרים ככל הניתן לכל אורך החישוב. נוסף על החיסכון בזמן אנחנו חוסכים גם במקום משום שתוצאות הביניים תופסות פחות מקום.

עד כה הנחנו שיש לנו שתי רשימות ממוינות שצריך לחתוך. כן, אם הרשימות ממוינות ניתן לחתוך אותן בזמן ליניארי בסכום האורכים. אם אחת הרשימות l_2 מאפשרת חיפוש בינארי אז ניתן לחתוך אותן בזמן $O(|l_1|\log|l_2|)$ ע"י חיפוש של כל איברי הרשימה הראשונה ברשימה השנייה. ההחלטה איזה אלגוריתם יעיל יותר תלויה האורכי הרשימות.

במקרה שלנו, אף אחת מהאפשרויות אינה ישימה באופן ישיר. נזכור שאמרנו שאת רשימות המסמכים אנחנו שומרים כרשימת הפרשים. לכן אין לנו למעשה שתי רשימות ממוינות. אבל בזמן ליניארי ניתן להחזיר אותם לצורתן המקורית ואז לחשב את החיתוך באמצעות האלגוריתם הראשון, ואילו את האלגוריתם השני כלל אי אפשר ליישם כאן. ראשית, לא ניתן לבצע חיפוש בינארי משום שהרשימה נשמרת בקידוד באורך משתנה ולכן אי אפשר לגשת אקראית לערכים שונים ברשימה. אבל אפילו אם הדבר היה אפשרי אז בגלל שמדובר ברשימת הפרשים אפילו אם ניגשנו להפרש מסוים צריך לחזור אחורה לתחילת הרשימה כדי לגלות איזה מסמך ההפרש הזה מציין.

ובכל זאת, אם הרשימה השנייה ארוכה והראשונה קצרה דווקא האפשרות השנייה תהיה יעילה יותר. נרצה למצוא דרך לגשת לרשימות משורשרות בצורה אקראית.

גישה יעילה לרשימות משורשרות

נראה איך ניתן לאפשר חיפוש מהיר ברשימות שאי אפשר לגשת לאיבריהן רנדומאלית. בהינתן רשימה משורשרת נוסיף לה מצביעים באופן הבא: לכל צומת מהצורה 2^i נוסיף מצביע לצומת ה-2^i הבא. כלומר, יהיו מצביעים שמקשרים את כל הצמתים הזוגיים ויהיו צמתים שמקשרים את כל הצמתים שמספרם מתחלק ב-4 וכן הלאה. זה מאפשר לבצע חיפוש בינארי ברשימה: בכל שלב באלגוריתם יש לנו אפשרות לגשת אקראית למקום האמצעי של תת הרשימה שהצטמצמנו אליה. אז הגישה לרשימה יעילה מאוד עכשיו אבל הכנס של איברים חדשים או מחיקתם מסובכת מאוד. יש המון מצביעים שיש לעדכן.

יש מבנה שנקרא Skip List אשר מבטיח בהסתברות גבוהה התנהגות יעילה והוא פחות מסובך ממה שתיארנו כעת. לא נתעמק בו.

נקודות סנכרון

נקודת סנכרון ב-posting list היא זוג מהצורה $(docID, bitAddress)$ כאשר

- $docID$ הוא המסמך הנוכחי ברשימה
- $bitAddress$ הוא מצביע לנקודת הסנכרון הבאה

כמובן, את נקודות הסנכרון אפשר לשמור כהפרשים מנקודות הסנכרון הקודמות

לדוגמה: נניח שרשימת המסמכים היא 5,8,12,13,15,18,23,28, 29. אז רשימת ההפרשים היא 5,3,4,1,2,3,5,5,1. אם נרצה לשמור נקודת סנכרון בכל מקום שלישי נקבל את הרשימה

$$(5, a_1), 3,4, (8, a_2 - a_1), 2,3, (10, a_3 - a_2), 5,1$$

נקודות סנכרון מאפשרות להתקדם ברשימה בצורה קצת יעילה. נשאלת השאלה כמה נקודות סנכרון צריך להוסיף. ככל שיש יותר נקודות הגישה אולי נוחה יותר אבל זה תופס יותר זיכרון כרגיל, כדי לקבל שיפור צריך לשלם עליו. בכל מקרה, ננסה לנקוט בשיטה אנליטית כדי להחליט מהו גודל המרווח המיטבי. נסמן ב-x את מספר האיברים שכל מצביע קופץ מעליהם. יהי k אורך הרשימה כולה. אז יש $\frac{k}{x}$ נקודות סינכרון. מספר ההשוואות שמתבצע כדי למצוא איבר כלשהו הוא $\frac{k}{x} + x - 1$: $\frac{k}{x} - 1$ כדי לקבוע באילו מהרשימות נמצא האיבר ואז עוד x השוואות עם האיברים שבתת הרשימה. לכן נרצה למזער את הפונקציה $c(x) = \frac{k}{x} + x - 1$. ניתוח פשוט מראה שהפונקציה מקבלת מינימום עבור $x = \sqrt{k}$. אכן, זו "חכמת הרחוב" שנוקטים בה לרוב – מוסיפים שכבה אחת של $\sqrt{k}$ מצביעים.

לסיכום נעיר שצריך לזכור שאפשר להשתמש בשיטה הזאת רק אם משתמשים בשיטת דחיסה שדוחסת כל אות בנפרד, כמו קידוד האפמן או קידוד גאמא. בקידוד אריתמטי למשל מתייחסים לכל הרשימה כולה כאובייקט אחד ואין אפשרות לפענח חלק ממנה. לכן אם הרשימה שלנו שמורה בקידוד אריתמטי נקודות הסנכרון לא יעזרו כלל.

שאילתות לא קוניוקטיביות

שאילתא שאינה שאילתת גימום משתמשת גם בפעולות OR ו-NOT. הטיפול בשאילתות מהצורה

$$(*)\ (\text{word}_{11}\ \text{OR} \ldots \text{OR word}_{1n})\ \text{AND} \ldots \text{AND}\ (\text{word}_{m1}\ \text{OR} \ldots \text{OR word}_{mn})$$

הוא יחסית פשוט. נשערך את הגודל של כל שאילתת איווי כסכום גדלי השאילתות של המילים הנפרדות ואז נמשיך כמקודם, כאשר מתייחסים לכל הערכים של שאילתת איווי ביניים בו-זמנית.

עוד מקרה יחסית פשוט הוא שאילתת גימום שבכמה מילים מה יש תווי-כל. אז הניתוח של השאילתא הוא למעשה ניתוח של שאילתא מהצורה (*), שהרי כל מילה למעשה מסמל כמה אפשרויות חלופיות

עד כה זמן ניתוח השאילתא היה ליניארי בסכום גדלי השאילתות של כל מילה בנפרד. אבל יש שאילתות שלא ניתן לענות עליהם בזמן ליניארי (בשימוש במנגנונים שדיברנו עליהם במסגרת זו). למשל, שאילתות מהצורה word1 AND (not word2) דורשות זמן ריבועי...

Crawling

בבניית האינדקס יש שתי פעולות עיקריות: מציאת המסמכים ובניית מבני הנתונים. על מבני הנתונים כבר דיברנו בהרחבה. הפרק הזה נדון בתהליך שמוצא את הדפים החדשים ברשת ומעדכן את האינדקסים – תהליך ה-crawling.

לרובוט חיפוש (Crawler) יש כמה מטרות:

1. להוריד קבוצה גדולה של דפים ברשת
2. לעדכן דפים שכבר הורדו מהרשת
3. למצוא דפים חדשים שלא קיימים במאגר עדיין

4. לוודא שיש כמה שיותר דפים "טובים"

פרט לכך, נרצה שהרובוט לא ישלח יותר מדי בקשות לאותו שרת (כדי לא ליצור עליו עומס יתר) ויתעדכן עם חלק מהדפים בתדירות גבוהה יותר מדפים אחרים. שתי הדרישות האלה כמובן עלולות להתנגש אחת עם השנייה.

אלגוריתם בסיסי

הרובוט הכי בסיסי פועל באופן הבא:

1. מתחילים עם גרעין של URL-ים ידועים מראש (אולי כדאי לבחור אותם להיות URL-ים "טובים"...)
2. מכניסים אותם לתור
3. כל עוד התור לא ריק
   a. מוציאים URL מהתור ומבקשים את המשאב מהשרת
   b. מפרקים את המשאב ומוצאים את כל ה-URL-ים שהוא מצביע אליהם
   c. מכניסים כל URL שעוד לא ביקרנו בו לתור

תיאורטית, מאחר שמספר הדפים ברשת הוא סופי התהליך הזה אמור להסתיים מתישהו. אבל בפועל דפים חדשים מתווספים בקצב כל כך גבוה ודפים קיימים מתעדכנים כל הזמן. בפועל עבודתו של הרובוט לא מסתיימת לעולם...

בפועל סביר שיש כמה רובוטים שעובדים בו-זמנית. במקרה זה ניתן לחלק את הדפים בין הרובוטים לפי ערכי פונקצית ערבול על ה-URL-ים המתאימים.

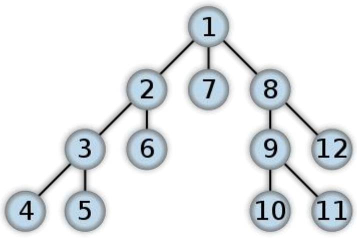
יש כמה פרטים שהשמטנו מהתיאור הבסיסי הנ"ל. למשל, לא ברור לפי איזה קריטריון ממוין התור ואיך מכניסים ומוציאים ממנו URL-ים. הסדר שבו מוציאים URL-ים משפיע ישירות על הנתיב שבו אנחנו מתקדמים, על קבוצת המשאבים שאנחנו מגיעים אליה.

ניהול התור

יש שלוש אסטרטגיות ניהול תור בסיסיות: חיפוש לעומק וחיפוש לרוחב או תור קדימויות (צריך אז איכשהו להגדיר ציונים לאיברים בתור). יש הרבה מאוד אפשרויות אחרות וכל אחד יכול לבחור אסטרטגיה שמתאימה לצרכיו. אנחנו נפרט קצת על האסטרטגיות הבסיסיות שהזכרנו.

מחסנית – חיפוש לעומק

אם נרצה לעשות חיפוש לעומק נכניס תמיד כתובות חדשות שמתגלות לתחילת התור ונוציא כתובות מתחילתו. זהו מודל של מחסנית LIFO. אז הסדר שמבקרים בקודקודים הוא למשל כזה:

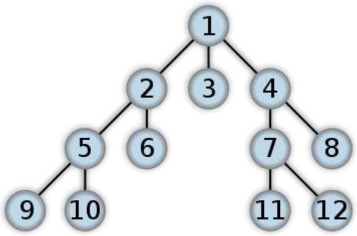


בצורה זו, אם למשל נתקלנו בעמוד שיש בו קישור לאתר של CNN וקישור לאתר של BBC אז קודם כל נבקר בכל העמודים של האתר של CNN ורק לאחר מכן נגיע לבקר באתר של BBC. יש לזה יתרונות מבחינת מטמון של DNS אבל יכול להיות שלא נקבל כך כיסוי מספיק נרחב. אולי כלל לא נצליח להגיע חזרה לאתר של BBC.

יתר על כן יש לשיטה זו חיסרון טכני. הרובוט שולח הרבה מאוד בקשות ובתדירות מאוד גבוהה. השרתים של CNN יכולים להחליט לחסום אותו משום שייראה כאילו הוא עושה להם מתקפת denial of service.

תור – חיפוש לרוחב

אם נרצה לעשות חיפוש לעומק נכניס כתובות חדשות לסוף התור ונוציא כתובות מתחילתו, כלומר נעבוד בשיטת FIFO. אז בדוגמה הקודמת סדר המעבר על הדפים יהיה כזה:



בהמשך לדוגמה של CNN ו-BBC, כאן ניכנס לעמוד הראשי של CNN ואז לעמוד הראשי של BBC ורק אז תחיל להתקדם הלאה בתוכם. כאן אנחנו מאבדים את תכונת הלוקליות שעוזרת לנו ב-DNS אבל אנחנו פותרים את בעיית העומס על השרתים וכן מגיעים לדפים הרדודים קודם בד"כ הדפים העמוקים יותר פחות חשובים ולכן זה יכול להוות יתרון.

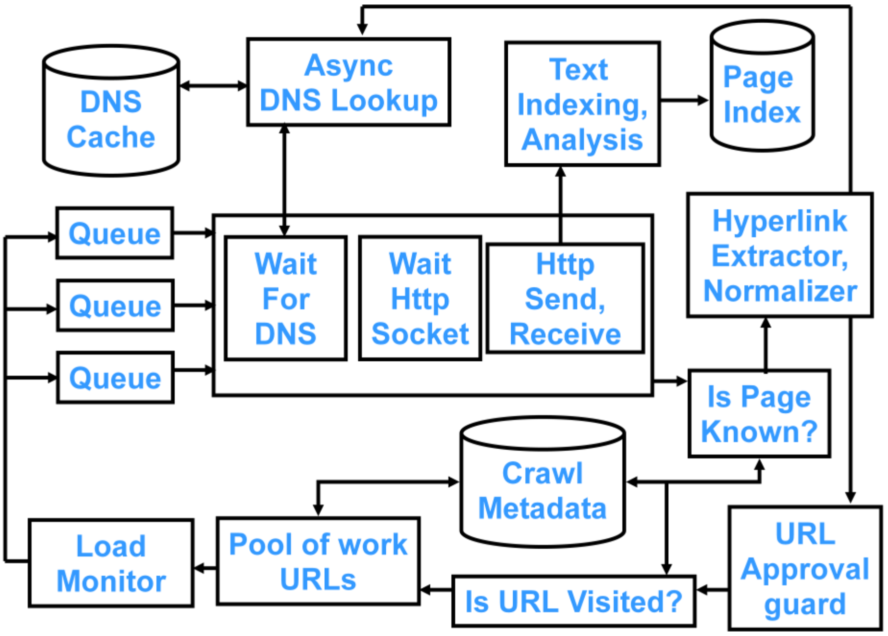
תור קדימויות

כדי להשתמש בתור קדימויות לכל URL צריך לתת ציון שקובע כמה הוא חשוב. בהינתן ציונים תמיד נוציא מהתור את ה-URL עם הציון הגבוה ביותר. יש כל מיני דרכים לתת ציונים לדפים. למשל:

- לפי נושא עניין (יכול להיות טוב לרובוט שמחפש דפים בנושא מסוים):
ניצור רשימה של מילים שמתארות את הנושא שמעניין אותנו. אז נגדיר את החשיבות של דף לפי רמת הדמיון שלו לקבוצת מילים זו. יותר מילים משותפות פירושו שהדף חשוב יותר. אפשר להמציא כל מיני נוסחאות מתוחכמות שמשלבות את מספר המופעים של המילים בדף ו/או תדירות המילה בכל רשת האינטרנט (למשל אם יש מילה שככלל היא מאוד נדירה והיא דווקא מופיעה במסמך מסוים אז כנראה שהוא שכדאי לנו לבחון).
הבעיה עם השיטה הזאת היא שאנחנו צריכים להחליט מה הציון של דף מבלי לקרוא את כולו ומבלי לדעת את תדירות המילים (החישוב של הפרטים האלה כמובן לוקח זמן רב ויקר). אז אפשר לשערך את טיב הדף ע"פ ההקשר של הקישור או הדף כולו שמכיל את הקישור.
- לפי פופולאריות:
החשיבות של דף פרופורציונאלית למספר הדפים שיש להם קישור אליו. צריך למצוא דרך לשערך את מספר הדפים האלה. בעזרת שיטה שנקראת PageRank ניתן לבצע שיערוך זה. אנחנו נדון בזה בהמשך.
- לפי מיקום:
יכול להיות שנחליט שהחשיבות של דף נקבעת לפי ה-URL שלו. למשל, אולי דפים של ה-CNN יותר חשובים מבחינתנו מאשר הדפים של BBC. או אולי דפים מישראל יותר חשובים לנו מדפים מאוזבקיסטן. המדד הזה קל מאוד לחישוב ולא דורש שום מידע נוסף מחיפושים קודמים של הרובוט

כמובן, אפשר גם לשלב את הגישות האלה...

ארכיטקטורה של רובוט חיפוש



נניח שרובוט רוצה להוריד משאב מסוים. אז הוא פועל בדיוק כמו דפדפן: ניגש ל-DNS, מקבל את ה-IP, ניגש לשרת וכו'. אחרי שהוא מקבל את המשאב הוא צריך לפרק אותו, להוציא ממנו את כל הקישורים ולהמשיך באותו אופן. את התהליך הזה הוא צריך לעשות הרבה מאוד פעמים וכדאי שגם שיעשה את זה מהר. אבל בדרך הוא יכול להיתקל בכל מיני מכשולים ועיכובים שצריך להתגבר עליהם:

- מציאת כתובת IP באמצעות DNS

כדי להתגבר על מכשול זה אפשר לשמור כתובות IP במטמון. זה במיוחד יעיל אם נוקטים בגישת החיפוש לעומק. עוד אפשרות היא לעשות pre-fetching ולקבל כתובות IP מה-DNS מיד כשמגיע URL (ולא לחכות עד שממש צריך לגשת למשאב). כך הכתובת תהיה מוכנה ברגע שצריך אותה.

- התקשרות לשרתים ושליחת בקשות
- ציפייה לתשובות

רובוטים צריכים להיות ממושמעים

לפעמים מנהלי אתרים רוצים לתת לרובוטים שמבקרים אותם הוראות לא לבקר בקבצים, לא להכניס את הקבצים לאינדקס, לאפשר גישה רק לרובוטים מסוימים, להפנות את הרובוט לקישורים חשובים, להודיע לרובוט על שינויים ועוד. את ההוראות האלה ניתן לתת באמצעות Robots Exclusion Protocol.

ההוראות נכתבות בקובץ טקסט robots.txt בספרייה השורש של השרת. למשל, http://www.cnn.com/robots.txt, או http://www.google.com/robots.txt. הרובוטים מצופים להישמע להוראות, אבל אין שום מנגנון אכיפה.

הקובץ robots.txt מורכב מאוסף רשומות מהצורה

```
User-agent: <Names of crawlers>
Allow: <Allowed URLs for these crawlers>
Disallow: <Disallowed URLs for these crawlers>
```

לדוגמה, אם נרצה לאסור על כל רובוטים לעבור על הקבצים באתר שלנו נכתוב בקובץ

```
User-agent: *
```

```
Disallow: /
```

ההוראות מיושמות לפי סדר ההופעה שלהם. אז אם נרצה לאפשר רק לרובוט אחד גישה לקבצים שלנו נכתוב בקובץ:

```
User-agent: My Favorite Robot
Disallow:

User-agent: *
Disallow: /
```

הוראות לרובוטים אפשר להעביר גם דרך דפי ה-HTML עצמם. כך ניתן לומר לרובוט אם להכניס את הדף לאינדקס או לא ואם להמשיך לעבור בקישורים שלו או לא ההוראות נכתבות באמצעות meta tag בשם robots:

```
<meta name = "robots" content = "options"/>
```

האפשרויות הקיימות הן nofollow,follow,noindex,index. כמו קודם, הרובוט מצופה להקשיב להוראות שניתנו לו דרך ה-meta tag אך אין שום מנגנון אכיפה שמבטיח התנהגות ראויה. אם רוצים להבטיח התנהגות נימוסית מצד הרובוט צריך לנקוט באמצעים יותר חמורים כמו שמות משתמש, סיסמאות וחיבורים מאובטחים.

עוד דרך להעביר לרובוט הוראות היא באמצעות meta tag שאומר לרובוט מתי לחזור. זה יכול לייעל את פעולת הרובוט. אם כותב האתר יודע שהוא הולך לעדכן אותו בעוד שבוע אז כדאי שיגיד את זה לרובוט:

```
<meta name = "revisit-after" content = "7 days"/>
```

עוד נעיר שרובוט יכול להתחזות לרובוטים אחרים ולכן לא באמת ניתן לדעת אילו רובוטים ניגשו לאתר שלנו..

בעיית ה-DUST

ברשת יש הרבה שכפול מידע – אתרי מראה, domain names שונים שמצביעים לאותה כתובת IP, URL-ים שונים שמצביעים לאותו משאב ברשת. יש גם מידע "כמעט משוכפל". למשל עמוד הבית של Google מחליף תמונה כל יום אבל הטקסט נותר זהה. עוד דוגמה היא אתר עם טקסט כמעט קבוע שבחלקו התחתון מופיע מונה מבקרים או שהוא מציג את התאריך.

תופעה זו נקראת DUST – קיצור של Different URL Similar Text.

תופעה זו מציבה אתגר רציני בפני רובוט החיפוש. האידיאל הוא להימנע מלהוריד מידע משוכפל. זה גם מבזבז זמן ומקום וגם פוגע באיכות התוצאות. יש כמה דרכים שמנסות להתגבר על בעיית השכפולים. למשל, ניתן לשמור גרסאות קנוניות כלשהם של ה-URL-ים. זה פותר בעיות כמו הדוגמה שכבר צויינו בתחילת המסמך עם http://www.notes-heaven.com/, http://www.notes-heaven.com, http://www.notes-heaven.com/index.html. כל ה-URL-ים האלה מצביעים בדיוק לאותו המשאב אז נורמליזציה של כתובות לא יכול לפתור את בעיית אתרי המראה. אפשרות אחרת היא שמירת ערך ערבול של הדפים. כשמגיע דף חדש נשווה את הערך שלו עם ערכים של דפים שכבר קיימים אצלנו. זה פותר את בעיית אתרי המראה אך לא מתמודד עם "כמעט שכפולים".

מציאת שכפולים מדויקים

כדי למצוא שכפולים מדויקים ניתן להשוות את הדפים עם מדדי מרחק ידועים כמו מרחק העריכה הבעיה היא שזה לוקח המון זמן, גם כי הדפים יכולים להיות ארוכים וגם כי יש מיליוני דפים שצריך להשוות אז נרצה למצוא מדד קצר יותר לחישוב. נרצה ליצור תקציר של הדף שהרבה יותר קצר אבל הכול זאת מאפשר לנו למצוא שכפולים. נניח שהדף שלנו מיוצג כרצף של אסימונים (tokens), ללא סימני פיסוק, ללא HTML וכו'. נסתכל רק על התוכן של דף.

בהינתן מסמך D $w-$shingle שלו הוא רצף של w של אסימונים עוקבים במסמך[7]. אז $w-$shingling של המסמך הוא אוסף כל ה-$w-$shingls שלו. נסנן אוסף זה ב-$S(D,w)$. לדוגמה, אם המסמך הוא

A rose is a rose is a rose is a rose is a rise is a rose is a rose

אז $S(D,4) = \{(\text{a,rose,is,a}), (\text{rose,is,a,rose}), (\text{is,a,rose,is})\}$.

נגדיר את הדמיון של שני מסמכים A,B ע"י נוסחה דומה לזאת של מקדם Jaccard:

$$r_w(A,B) = \frac{|S(A,w) \cap S(B,w)|}{|S(A,w) \cup S(B,w)|}$$

גם כאן מתקיים $0 \leq r_w(A,B) \leq 1$ ואכן $r_w(A,A) = 1$ אבל אם $r_w(A,B) = 1$ לא נובע בהכרח ש-$A = B$ (ולא כי מספר המופעים של כל shingle לא משחק תפקיד בנוסחה לדמיון. למשל בדוגמה לעיל יכולנו לשרשר את המסמך עם עצמו כמה פעמים ועדיין ה-$w-$shingling שלו לא היה משתנה).

קבוצת כל ה-shingles של מסמך היא גדולה, אפילו יותר גדולה מהמסמך עצמו. לכן נרצה ליצור תקציר של המסמך ע"י דגימה של מספר קטן של shingles. הדרישה היא כמובן שדמיון של התקצירים מהווה שיערוך לדמיון בין המסמכים המקוריים.

מעתה נניח ש-w נתון וקבוע ולכן נשמיט אותו מהסימונים.

בחירה אקראית מתוך $S(A),S(B)$ אינה מקיימת את הדרישות שלנו. אפילו אם המסמכים זהים, הסיכוי שנקבל $r(A,B) = 1$ הוא קטן. נניח שיש במסמכים בסה"כ n shingles. יהי m_A shingle מ-A. אם נבחר shingle m_B בהתפלגות אחידה מ-B אז $\Pr(m_A = m_B) = \frac{1}{n}$. אבל הרי $r(A,B) = 1$. אז הסיכוי שנקבל את המדד הנכון הוא די נמוך.

נראה דרך טובה יותר ליצור תקציר. נניח שעל קבוצת כל ה-shingles מוגדר סדר מלא (כלומר אפשר להשוות בין כל שני איברים). נגדיר את m_A להיות ה-shingle המינימאלי של A ואת m_B להיות ה-shingle המינימאלי של B. ונגדיר $r'(A,B) = |m_A \cap m_B|$. אם המסמכים A,B זהים אז בוודאי מתקיים שוויון $r(A,B) = 1 = r'(A,B)$.

כעת הרעיון הוא שנתבונן בהרבה סידורים מלאים אקראיים של קבוצת כל ה-shingles. לכל אחד מהם נשווה את ה-shingles המינימליים. ההסתברות שהמסמכים זהים אך כל ההשוואות נכשלות היא קטנה.

---

[7] זה כמו $n-$grams של מילה.

שיטה נוספת לטיפול בעיית ה-DUST הוצגה ע"י צבי בר-יוסף, עידית קידר ואורי שונפלד. הם מציגים שיטה שבה ניתן ללמוד מהסתכלות ברשימות URL-ים את הכללים שלפיהם ניתן לקבוע אם שני URL-ים מצביעים לאותו המשאב. לפרטים נוספים עיינו בנספח ג.

דירוג

עד עכשיו הנחנו שסיפוק היא פונקציה בוליאנית של השאילתא: או שתוצאה מספקת שאילתא או שלא. רצינו שכל התוצאות המספקות יוחזרו למשתמש ללא משמעות מיוחדת לסדר החזרה. זה דומה לגישה של עיבוד שאילתות במסדי נתונים. אבל במציאות המודל הזה לא תמיד מתאים. במציאות משתמשים יאהבו חלק מהתוצאות יותר וחלק פחות. יתר על כן, מסמך יכול לכלול את כל מילות השאילתא ולא להיות רלוונטי ומסמך אחר יכול לא לכלול אף אחת ממילות השאילתא ובכל זאת להיות רלוונטי.

תוצאות השאילתא צריכות לקבל ציונים ולחזור למשתמש מסודרות לפי הדירוג הבעיה היא שרלוונטיות היא אמת מידה סובייקטיבית ורק המשתמש יכול לומר לנו מהו הדירוג הנכון. בכל זאת נרצה למצוא דרכים להעריך את הדירוג שהמשתמש היה נותן לתוצאות.

אז מטרת העל שלנו תהיה להחזיר את כל התוצאות הרלוונטיות ולא להחזיר אף תוצאה לא רלוונטית את התוצאות הרלוונטיות נרצה להחזיר לפי סדר הרלוונטיות.

נעיר רק שהדירוג לא אמור להיות קשור לכסף ומנוע חיפוש לא צריך לקבל כסף תמורת העלאת דירוג של אתרים.

הגדרת איכות תוצאות חיפוש

יש הרבה מדדים של איכות אבל אנחנו נתמקד בשלושה:

- דיוק (Precision) – אחוז התוצאות הרלוונטיות מכלל התוצאות שהוחזרו: $\frac{\text{relevant}\cap\text{retrieved}}{\text{retrieved}}$
- כיסוי (Recall) – אחוז התוצאות הרלוונטיות שהוחזרו מכלל המסמכים הרלוונטיים: $\frac{\text{relevant}\cap\text{retrieved}}{\text{relevant}}$
- דיוק ב-k – אחוז התוצאות הרלוונטיות מתוך k התוצאות הראשונות שהחוזרו

דוגמה: נניח שיש 1000 מסמכים ומתוכן 50 רלוונטיים לשאילתא, 30 מסמכים מוחזרים ומתוכם 20 מסמכים רלוונטיים. אז

$$\#\text{relevant} = 50 \qquad \#\text{retrieved} = 30 \qquad \#(\text{relevant} \cap \text{retrieved}) = 20$$

ולכן מתקבלים המדדים הבאים:

$$\text{precision} = \frac{\text{relevant} \cap \text{retrieved}}{\text{retrieved}} = \frac{20}{30} \approx 67\%$$
$$\text{recall} = \frac{\text{relevant} \cap \text{retrieved}}{\text{relevant}} = \frac{20}{50} = 40\%$$

דיוק מושלם יכול להיות מושג אם אף פעם לא נחזיר את תוצאה.[8] כיסוי מושלם מתקבל אם מחזירים תמיד את כל המסמכים.

כדי לבדוק איכות של מנוע חיפוש אפשר להריץ אותו על נתונים סינתטיים שבהם תוצאות השאילתא ידועות. אז ניתן לבדוק את הדיוק והכיסוי של המנוע באופן אוטומטי.

---

[8] טכנית אי אפשר להגדיר דיוק של תוצאה כזאת כי אי אפשר לחלק ב- 0 אבל הרעיון הוא שאם לא מחזירים אף תוצאה אז בפרט לא מחזירים אף תוצאה שאינה נכונה.

שיטות דירוג

בגדול יש שתי שיטות דירוג: דירוג תלוי שאילתא ודירוג בלתי תלוי בשאילתא היתרון בדירוג תלוי שאילתא הוא ברור. אך זה אומר שבכל פעם שהמנוע מקבל שאילתא הוא צריך לבזבז זמן ולדרג את התוצאות בדירוג בלתי תלוי בשאילתא לכל דף יש דירוג גלובאלי שכבר מחושב ושמור בשרת זה מקצר את זמני הריצה מאידך, אנחנו מאבדים את היתרון של קבלת דירוג שאולי רלוונטי יותרלשאילתא הספציפית שביקשנו.

לדוגמה, נניח שיש לנו שני מסמכים שבהם הראשון מדורג גבוה יותר מהשני:

Doc1: John Locke was a famous British philosopher... His name was used in the popular TV show *Lost*...

Doc2: John Locke is a character in the Popular TV show *Lost*. Jonathan "John" Locke is one of the middle section survivors of Oceanic Flight 815. His paralysis was immediately healed...

אם נחשב את השאילתא "John Locke" נקבל את המסמך הראשון במקום הראשון. זו כנראה תוצאה סבירה. אבל אם נחשב את השאילתא "John Locke Lost" היינו מצפים דווקא לקבל את המסמך השני במקום הראשון. אם הדירוג הוא גלובאלי זה בלתי אפשרי.

שיטות מותאמות לשאילתא

נניח שהשאילתות נתונות בצורה של אוסף מילים ומסמך עלול להיות רלוונטי לשאלתא אם הוא מכיל לפחות מילה אחת ממנה. בהינתן שאילתא ומסמך נרצה למצוא ציון שמתאר כמה המסמך מתאים לשאילתא

TF-IDF ומודל המרחב הווקטורי

השיטה מבוססת על כמה עקרונות:

- לכל מילה t במסמך d ניתן משקל $w_{t,d}$
- לכל מילה t בשאילתא ניתן משקל w_t
- הציון של מסמך בהינתן שאילתא יהיה פונקציה של משקלות מילות השאילתא במסמך

נשאלות כמובן השאלות אין מגדירים את המשקלות ומהי הפונקציה שמשתמשים בה לחישוב הציון

אם t לא מופיעה ב-d נגדיר $w_{t,d} = 0$. שאלה יותר קשה היא איך להגדיר את המשקל של מילה שדווקא כן מופיעה במסמך. הגדרה פשוטה יכולה להיות $w_{t,d} = 1$ לכל מילה שמופיעה במסמך. זאת כמובן שיטה מאוד קלה ומהירה לחישוב. השיטה הזו לא מתחשבת במספר המופעים של מילה במסמך. זה יכול להיות גם יתרון וגם חיסרון כי לא ברור אם תמיד תדירות מופע גדולה מסמנת חשיבות. זה גם חיסרון כי אולי נרצה להתייחס למילים נדירות בצורה שונה (למשל אם יש מילה מאוד נדירה שמופיעה במסמך כלשהו נרצה לתת לה דווקא משקל גבוה).

אפשרות חלופית מסתמכת על האינטואיציה שבכל זאת אם מילה כלשהי מופיעה הרבה פעמים במסמך אז יש קשר הדוק בין המילה למסמך. אז ניתן להגדיר $w_{t,d} = f_{t,d}$ כמספר הפעמים שהמילה t מופיעה במסמך t. זוהי התדירות של המילה במסמך. השיטה השאת אולי אינטואיטיבית יותר אך החיסרון הבולט שלה הוא שהיא לא מתחשבת באורך המסמך. אם מילה מופיעה פעם אחת במסמך של מאות עמודים אולי המעמד שלה שונה מאשר מילה שמופיעה במסמך של שורה בודדת..

לכן נרצה לנרמל איכשהו את תדירות המופע. הדרך המקובלת לעשות זאת היא ע"י ההגדרה

$$w_{t,d} = \begin{cases} 1 + \log f_{t,d}\,, & f_{t,d} > 0 \\ 0, & f_{t,d} = 0 \end{cases}$$

אבל לא כל המילים נולדו שוות. מופע של מילה מאוד נדירה הוא בד"כ מעניין יותר ממופע של מילה מאוד נפוצה למשל, נתבונן בשאילתא Winnie the Pooh. אף אם במסמך אחד יש 300 מופעים של המילה the ובמסמך אחר יש מופע אחד של המילה Pooh סביר שדווקא המסמך השני הוא זה שמעניין אותנו.

ניתן להגדיר את תדירות המסמכים (Document Frequency) של מילה t. זהו מספר המסמכים שמכילים את המילה t. נסמן תדירות זו ב-f_t. אז נגדיר את המשקל של מילה בודדת להיות $w_t = \log\left(1 + \frac{n}{f_t}\right)$ כאשר n הוא המספר הכולל של המסמכים. אז w_t מייצג את תדירות המסמכים ההפוכה (Inverse Document Frequency) של המילה t.

אנחנו נשתמש בסכמות המשקול

$$w_{t,d} = \begin{cases} 1 + \log f_{t,d}\,, & f_{t,d} > 0 \\ 0, & f_{t,d} = 0 \end{cases}$$

$$w_t = \log\left(1 + \frac{n}{f_t}\right)$$

סכמה זו נקראת TF-IDF משום שהיא לוקחת בחשבון את ה-term frequency ואת ה- inverse document frequency.

יש הרבה אפשרויות אחרות להגדיר את הערכים האלה, אבל כל השיטות מקיימות שני תנאים של מונוטוניות:

- מילה שמופיעה בהרבה מסמכים לא צריכה להיות חשובה יותר ממילה שמופיעה במעט מסמכים
- מסמך שבו מופעים רבים של מילה לא יכול להיות חשוב פחות ממסמך ובו מעט מופעים של אותה מילה

בסכמת המשקול TF-IDF משתמשים במודל המרחב הווקטורי כדי לתת ציונים לתוצאות שאילתא במודל זה מסתכלים על מסמכים ועל שאילתות כווקטורים במרחב k-ממדי כאשר k הוא מספר המילים בלקסיקון. וקטור של מסמך מכיל את המשקלים כל המילים במסמך, כלומר מסמך d מיוצג ע"י הווקטור $\left(w_{t_1,d}, \dots, w_{t_k,d}\right)$ ואילו שאילתא מיוצגת ע"י הווקטור המציין שלה – וקטור שאומר אילו מילים מהלקסיקון מופיעות בשאילתא ואילו לא. במקומות שמתאימים למילים שמופיעות בשאילתא מופיע המשקל שלהם

דמיון בין שני וקטורים נמדד ע"י הזווית ביניהם. ככל שהזווית קטנה יותר שני הווקטורים דומים יותר. מדד לגודל הזווית הוא הקוסינוס שלה. במרחב מכפלה פנימית (כמו $\mathbb{R}^n$) מגדירים את הזווית בין שני וקטורים ע"י

$$\cos\theta = \frac{\langle x, y\rangle}{\|x\|\|y\|} = \frac{\sum_{i=1}^{k} x_i y_i}{\sqrt{\sum_{i=1}^{k} x_i^2} \cdot \sqrt{\sum_{i=1}^{k} y_i^2}}$$

ככל ש-$\cos\theta$ גדול יותר כך הזווית θ קטנה יותר והווקטורים דומים יותר. כלומר ככל ש-$\cos\theta$ גדול יותר כך המסמך מתאים יותר לשאילתא ודירוגו גבוה יותר.

$$r(Q, D) = \frac{\langle Q, D\rangle}{\|Q\|\|D\|} = \frac{1}{\|Q\|\|D\|}\sum_{i=1}^{k} w_{t_i} w_{t_i,d}$$

מאחר ש-$\|Q\|$ מופיע בכל החישובים עבור שאילתא ספציפית, ניתן להוריד אותו מהחישוב ונישאר עם פונקציית הציון $\frac{\sum_{i=1}^{k} w_{t_i} w_{t_i,d}}{\|D\|} = \frac{\langle Q,D \rangle}{\|D\|} = r(Q,D)$.

דוגמה: נניח את הנתונים הבאים:

$$n = 210, k = 4, f_a = 30, f_b = 210, f_c = 70$$

ונניח שהמסמכים הם:

Doc1: aabbd　　　　Doc2: bcccc

והשאילתא היא abc

אז הווקטורים המתאימים הם

$$D_1 = (w_{a,d_1}, w_{b,d_1}, w_{c,d_1}, w_{d,d_1}) = (1 + \log f_{a,d_1}, 1 + \log f_{b,d_1}, 0, 1 + \log f_{d,d_1}) =$$
$$= (1 + \log 2, 1 + \log 2, 0, 1 + \log 1) = (2,2,0,1)$$
$$D_2 = (w_{a,d_2}, w_{b,d_2}, w_{c,d_2}, w_{d,d_2}) = (0, 1 + \log f_{b,d_2}, 1 + \log f_{c,d_2}, 0) =$$
$$= (0, 1 + \log 1, 1 + \log 4, 0) = (0,1,3,0)$$
$$Q = \left(\log\left(1 + \frac{n}{f_a}\right), \log\left(1 + \frac{n}{f_b}\right), \log\left(1 + \frac{n}{f_c}\right), 0\right) =$$
$$= (\log(1+7), \log(1+1), \log(1+3), 0) = (3,1,2,0)$$

כדי לדעת איזה מהמסמכים מדורג גבוה יותר צריך לחשב את מדד הדימיון:

$$r(Q, D_1) = \frac{\langle Q, D_1 \rangle}{\|D_1\|} = \frac{\langle (3,1,2,0), (2,2,0,1) \rangle}{\|(2,2,0,1)\|} = \frac{3 \cdot 2 + 1 \cdot 2 + 2 \cdot 0 + 0 \cdot 1}{\sqrt{2^2 + 2^2 + 1^2}} = \frac{8}{3}$$
$$r(Q, D_2) = \frac{\langle Q, D_2 \rangle}{\|D_2\|} = \frac{\langle (3,1,2,0), (0,1,3,0) \rangle}{\|(0,1,3,0)\|} = \frac{3 \cdot 0 + 1 \cdot 1 + 2 \cdot 3 + 0 \cdot 0}{\sqrt{1^2 + 3^2}} = \frac{7}{\sqrt{10}}$$

לכן המסמך הראשון מדורג גבוה יותר.

דירוג מבוסס HTML

קוד HTML של עמוד מגדיר את העיצוב שלו. בין השאר למשל, הוא אומר איזה חלקים הם כותרות, איזה חלקים מודגשים וכן הלאה. הרעיון הוא לתת ציון גבוה יותר למילים שנמצאות בחלקים חשובים של ה-HTML. אפשר להכניס שיקולים אלה למודל המרחב הווקטורי. כל מה שצריך לעשות הוא להחליט מה ההשפעה של כל חלק ב-HTML על חשיבות המילים המופיעות בו.

בהקשר זה נעיר שהטקסט שמופיע על קישור (anchor text) יכול להיות לעזר רב בדירוג. הרבה פעמים הטקסט הזה מתאר את מהות הקישור. הרבה פעמים דפים לא מכילים מילים שמתארות את התוכן שלהם (בעמוד הבית של Google לא מופיע הטקסט "מנוע חיפוש"). ואז הטקסט של הקישור הוא הרמז היחיד שיש לנו לגבי נושא הדף. זה כמובן נכון רק אם כותבים את הקישורים בצורה נכונה.

I started working on notes for the exam in ATDB

vs.

Click here for more information about our products and services

שיטות דירוג גלובאליות – ניתוח קישורים

בשיטות דירוג גלובאליות עמודים מקבלים דירוג מלכתחילה והוא אינו תלוי בשאילתא שמתקבלת. יש בדיוק סדר אחד שבו הדפים מדורגים וכל תוצאת חיפוש היא תת סדרה של הסדר הזה.

גישה נאיבית

את הרשת ניתן לייצג כגרף: דפים הם צמתים ויש קשת בין P_1 ל-P_2 אמ"מ ב-P_1 יש קישור שמצביע ל-P_2. אפשר לומר שאם דף מצביע לדף אחר הוא סומך עליו במידה מסוימת אז ככל שיותר דפים סומכים על דף כלשהו כנראה שהוא יותר טוב. כעת ניתן להגדיר שני מדדים:

- פופולאריות מכוונת – הציון של דף הוא מספר הקישורים אליו (דרגת הכניסה של הצומת בגרף מכוון)
- פופולאריות לא מכוונת – הציון של דף הוא סכום מספר הקישורים אליו ומספר הקישורים ממנו (דרגת הכניסה של הצומת בגרף לא מכוון)

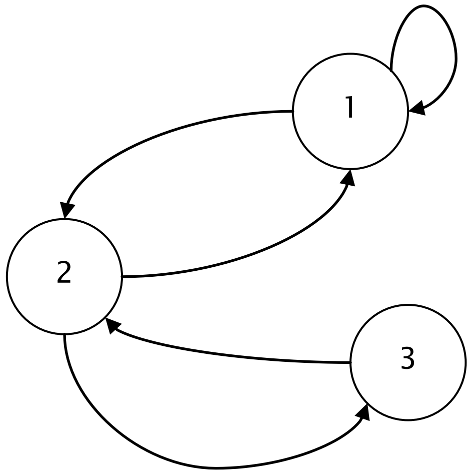
בשיטות הדירוג האלה יש כמה בעיות. ראשית, בשיטה זו נותנים חשיבות זהה לכל הדפים. אבל במציאות היינו רוצים לסמוך רק על דפים שאנחנו כבר יודעים שהם חשובים. שנית, קל לנצל את השיטה לרעה. אם בבעלותנו אתר, ניתן ליצור דף אחר שיש בו מלא קישורים לאתר שלנו זה מעלה את הדירוג של האתר בצורה לא "חוקית".

גולש אקראי – PageRank

נדמיין גולש אינטרנט שמבצע הילוך מקראי על גרף הרשת. הוא מתחיל בקודקוד אקראי ובכל צעד יכול ללכת על אחת מהצלעות היוצאות של הקודקוד בהסתברות שווה. אז ההסתברות שהגולש ימצא את עצמו בסופו של דבר בדף מסוים יכולה לשמש כציון של אותו הדף. את ההילוך המקרי ניתן לתאר בעזרת מטריצת מעברים.
מטריצת המעברים היא מטריצה בגודל $n \times n$ כאשר n הוא מספר העמודים הקיימים. במקום ה-i, j של המטריצה מופיעה ההסתברות לעבור מ-P_j ל-P_i בצד אחד. כלומר,

$$m_{ij} = \begin{cases} \dfrac{1}{\text{outdegree}(P_j)}, & P_j \text{ links to } P_i \\ 0, & \text{otherwise} \end{cases}$$

לדוגמה, נתבונן בגרף הבא:



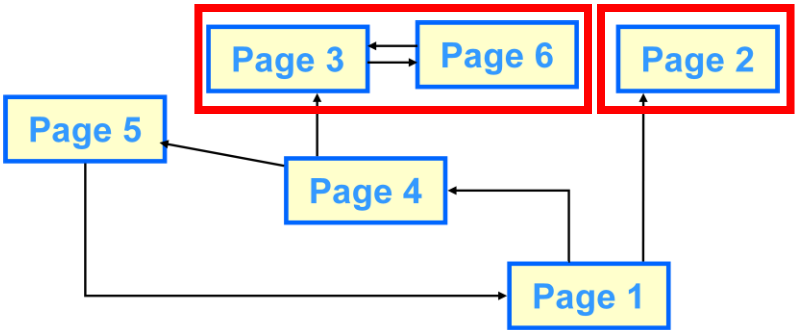
אז נקבל דרגות היציאה של הקודקודים הן

קודקוד	1	2	3
דרגת יציאה	2	2	1

ולכן נקבל שמטריצת המעברים היא

$$M=\begin{pmatrix}\frac{1}{2} & \frac{1}{2} & 0\\ \frac{1}{2} & 0 & 1\\ 0 & \frac{1}{2} & 0\end{pmatrix}$$

הסיכוי להתחיל בקודקוד P_j ולסיים אחרי k צעדים אקראיים בקודקוד P_i הוא $(M^k)_{ij}$. אז נשאיף את k לאינסוף ונקבל את מה שצריך. אבל הרשת מלאה בדרכים ללא מוצא ומלכודות עכבישים וההילוך המקרי עלול להיתקע שם:



זה פוגע באיכות הפתרון שכן אם בטעות הגענו שמבוי סתום זה לא בהכרח אומר שהדף הזה הכי חשוב והשאר לא... פיתרון לבעיה זו הוא טלפורטציה: בהסתברות d נמשיך בהילוך המקרי הקלאסי על גבי הצלעות ובהסתברות $1-d$ נקפוץ לדף אקראי כלשהו (אולי אותו הדף שאנחנו נמצאים בו). ההסתברות d נקראת damping factor והרבה פעמים בוחרים $d=0.85$. הטלפורטציה מבטיחה שמטריצת המעברים היא שרשרת מרקוב ארגודית. זה לא מאוד חשוב מה זה בדיוק אומר אבל מבחינתנו זה אומר שלכל דף יש הסתברות שנתחיל ממקום אקראי ואחרי זמן רב נגיע אליו.

שיטת דירוג זו נקראת PageRank והיא מוגדרת ע"י הנוסחה

$$\Pr P=\frac{1-d}{n}+d\left(\frac{\Pr P_1}{O_1}+\cdots+\frac{\Pr P_k}{O_k}\right)$$

כאשר $P_1,\dots,P_k$ הם דפים שמצביעים ל-P (כלומר דרגת הכניסה של P היא k), $O_1,\dots,O_k$ הם דרגות היציאה של דפים אלה ו-n מספר הדפים הכללי שקיים.

אם נכתוב את הנוסחה הזאת לכל הדפים שקיימים נקבל למעשה את המשוואה

$$\Pr P=dMP+\frac{1-d}{n}\vec{1}$$

כאשר P וקטור הציונים של כל הדפים ו-M מטריצת המעבר של הרשת.

בדוגמה הקודמת שלנו נקבל את המשוואה

$$\begin{pmatrix}p_1\\ p_2\\ p_3\end{pmatrix}=d\begin{pmatrix}\frac{1}{2} & \frac{1}{2} & 0\\ \frac{1}{2} & 0 & 1\\ 0 & \frac{1}{2} & 0\end{pmatrix}\begin{pmatrix}p_1\\ p_2\\ p_3\end{pmatrix}+(1-d)\begin{pmatrix}\frac{1}{3}\\ \frac{1}{3}\\ \frac{1}{3}\end{pmatrix}$$

את המשוואות ניתן לפתור בשיטה הרגילה עם תהליך גאוס אבל ברשת יש מיליוני דפים וזה יכול להיות קצת מסובך. אבל למזלנו התכונות של המערכת שלנו מבטיחות ששיטה אחרת תעבוד: נבחר ערך התחלתי שרירותי לכל אחד. הנוסחה למעלה למעשה מגדירה נוסחת רקורסיה. אז נמשיך לחשב את הציונים איטרטיבית. התהליך הזה מובטח להתכנס.

בדוגמה שלנו, נניח ש-$d = 0.85$ ונתחיל מווקטור ציון התחלתי: $\begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix}$

$$\begin{pmatrix} p_1^1 \\ p_2^1 \\ p_3^1 \end{pmatrix} = 0.85 \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix} + 0.15 \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix} = \begin{pmatrix} \frac{1}{3} \\ \frac{19}{40} \\ \frac{23}{120} \end{pmatrix} = \begin{pmatrix} 0.333 \\ 0.475 \\ 0.191 \end{pmatrix}$$

$$\begin{pmatrix} p_1^2 \\ p_2^2 \\ p_3^2 \end{pmatrix} = 0.85 \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{3} \\ \frac{19}{40} \\ \frac{23}{120} \end{pmatrix} + 0.15 \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix} = \begin{pmatrix} \frac{1889}{4800} \\ \frac{851}{2400} \\ \frac{403}{1600} \end{pmatrix} = \begin{pmatrix} 0.393 \\ 0.354 \\ 0.251 \end{pmatrix}$$

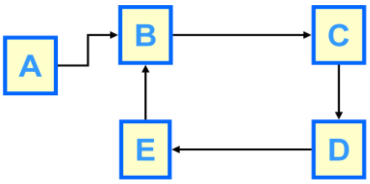
$$\begin{pmatrix} p_1^3 \\ p_2^3 \\ p_3^3 \end{pmatrix} = 0.85 \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} \frac{1889}{4800} \\ \frac{851}{2400} \\ \frac{403}{1600} \end{pmatrix} + 0.15 \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix} = \begin{pmatrix} \frac{23549}{64000} \\ \frac{82819}{192000} \\ \frac{19267}{96000} \end{pmatrix} = \begin{pmatrix} 0.367 \\ 0.431 \\ 0.2 \end{pmatrix}$$

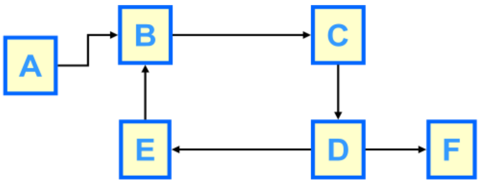
$$\begin{pmatrix} p_1^4 \\ p_2^4 \\ p_3^4 \end{pmatrix} = 0.85 \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} \frac{23549}{64000} \\ \frac{82819}{192000} \\ \frac{19267}{96000} \end{pmatrix} + 0.15 \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix} = \begin{pmatrix} \frac{1496461}{3840000} \\ \frac{579031}{1536000} \\ \frac{1791923}{7680000} \end{pmatrix} = \begin{pmatrix} 0.389 \\ 0.376 \\ 0.233 \end{pmatrix}$$

לא נמשיך מטעמי זמן ומקום אבל זה אמור להתכנס בסופו של דבר.

נראה עוד כמה דוגמאות אינטואיטיביות:

- בגרף מלא, הציונים של כל הקודקודים זהים, שהרי יש סימטריה מלאה במערכת.
- נתבונן בשני הגרפים:





הגרפים זהים פרט לקודקוד F שנוסף לגרף הימני. הקודקוד הזה גורם לכך שהסיכוי לעבור מ-D ל-E בגרף הימני קטן יותר. לכן הציון של E בגרף הימני יהיה נמוך יותר, כי הסיכוי להגיע אליו נמוך יותר.

נעים לסיכום שבעיית ה-DUST מהווה מכשול לשיטת דירוג זו. מבחינת מנוע החיפוש של URL שונה מצביע למשאב אחר ולכן אם יש כמה URL-ים שונים שמצביעים לאותו משאב הציון שלו יהיה למעשה מחולק בין כמה קודקודים.

PageRank מותאם נושא

אם אנחנו רוצים לבנות מנוע חיפוש שמותאם לנושא מסוים צריך למצוא דרך לדרג את הדפים בהקשר לנושא זה. למשל, אם אנחנו מעוניינים במנוע חיפוש שמתמחה בבישול כאשר אנחנו מחפשים את המילה batter נעדיף למצוא דפים עם מתכונים לעוגיות ולא תמונות של גברים מזיעים משחקים בייסבול. אבל ה-PageRank הסטנדרטי לא יודע איזה סוג batter אנחנו מחפשים והדירוג של כל התוצאות הרי כבר ידוע מראש[9]...

ניתן לעשות שינוי קל באלגוריתם הבסיסי ולשפר את המצב לקבוצת הדפים שניתן לקפוץ אליהם בטלפורטציה נקרא קבוצת הטלפורטציה. עד כה קבוצת הטלפורטציה כללה את כל הדפים ברשת אבל אם נרצה חיפוש מוכוון נושא נבחר את קבוצת הטלפורטציה להיות קבוצת דפים שאנחנו יודעים שהם קשורים לנושא העניין שלנו. בהנחה שדפים אלה מצביעים לעוד דפים על הנושא והאחרים גם הם מצביעים לדפים על הנושא וכן הלאה זה ייצור הטיה לטובת דפים על הנושא שלנו בחישוב הציונים.

איך לרמות מנוע חיפוש

כמו ב-PageRank מותאם נושא ניתן להטות מנוע חיפוש ע"י יצירת חוות קישורים – דפים שכל ייעודם הוא להצביע לדפים אחרים. אתר שמעוניין בהעלאת הדירוג שלו בסה"כ צריך לדאוג שיהיו הרבה קישורים אליו. תופעה זו נוגדת את העיקרון והמטרות שעומדות מאחורי הדירוג

אלגוריתם הדירוג Hilltop מנסה להתמודד עם הבעיה ש-PageRank אינו יודע להבדיל בין נושאים שונים. בתוצר לוואי האלגוריתם גם חסין יותר לרמאות מסוג זה. יותר פרטים ניתן למצוא בנספח ד.

אגב, גם שיטת ה-TF-IDF ניתן לרמות אבל לא בעזרת חוות קישורים. השיטה מתבססת של ספירת מופעים של מילים. אז אתר שמכיל הרבה מאוד מופעים שלהרבה מילים שקשורות לנושא שלו יקבל דירוג גבוה יותר. פעם היו מרמים ע"י כתיבת חזרות רבות של מילים בתחתית העמוד בצבע הרקע המשתמש לא היה רואה אותם אך מנוע החיפוש היה סופר אותם כמופעים לגיטימיים של טקסט.

Click-Through Ranking: DirectHit

שיטה נוספת לקבוע פופולאריות של עמוד היא לפי מספר המבקרים שבוחרים בו מתוך תוצאות של מנוע חיפוש. למשל, נניח שאנחנו מסתכלים על השאילתא "Scrubs". אם 20 משתמשים חיפשו שאילתא ואחרי שבחנו את עשר התוצאות הראשונות של מנוע החיפוש כולם בחרו את הדף מ-IMDb סימן שהאתר הזה נחשב יותר רלוונטי מאתרים אחרים. בפעם הבאה שמישהו יחפש את אותה שאילתא הדף מ-IMDb יופיע בדירוג גבוה יותר. השיטה הזאת לא בדיוק תלוית שאילתא משום שהדירוג לא מחושב מחדש לכל שאילתא, אך הוא מתעדכן אחרי שהשאילתא חושבה ומשתמשים בחרו את הקישורים האהובים עליהם

שיטת דירוג דומה וספרת את הזמן שמשתמש מבלה באתר. ככל שמשתמש מבלה יותר זמן באתר סביר שהאתר טוב יותר. כאן חשוב לזכור שלמנוע החיפוש איך דרך לדעת אם המשתמש הלך לשתות קפה מיד אחרי שנכנס לאתר או שבאמת ישב וקרא את כל תוכנו בתשומת לב.. זאת גם דרך מעולה לרמות מנוע

---

[9] אגב, מעניין לשים לב שאם מחפשים batter ב-Google הדף הראשון שנמצא הוא קישור ל-Wikipedia שמבקש לדעת באיזה הקשר אנחנו מעוניינים: http://en.wikipedia.org/wiki/Batter. ההקשר הראשון הוא דווקא בישול אבל אני לא בטוחה אם יש לזה משמעות מיוחדת...

חיפוש שמשתמש בדירוג כזה. מי שרוצה להעלות את דירוג האתר שלו פשוט ייכנס אליו וישאיר את העמוד פתוח...

שתי השיטות שתיארנו כאן סובלות מהתופעה the rich get richer. ככל שמשתמשים בוחרים דפים מסוימים או מבלים בהם יותר זמן הם מקבלים דירוגים גבוהים יותר. אבל אז המשתמשים רואים אותם במקומות הראשונים והסבירות שיבחרו בהם עולה. אז דפים יכולים ממש לקבל מונופול על שאילתות מסוימות ואז לדפים חדשים קשה מאוד להתקדם בדירוג.

חלק ג: XML

מוטיבציה

הרשת הסמנטית

דפים ברשת כתובים ב-HTML שמגדירה את המראה שלהם. היינו רוצים לקבל מהדפים לא רק את התוכן והעיצוב אלא גם את המשמעות של התוכן. היינו רוצים שתהליך אוטומטי יוכל "להבין" את התוכן של הדפים. למשל, אולי היינו רוצים למצוא באופן אוטומטי את כתובת הדוא"ל של איזשהו בן אדם. אז אנחנו צריכים שאיפשהו בדף יהיה מסומן איפה הכתובת הזאת רשומה ולמי היא שייכת ב-XML ניתן לכתוב למשל

```
<person>
  <name>Jane Doe</name>
  <email>anonymous@mail-client.com</email>
</person>
```

כאן מופיע לא רק טקסט אלא גם tags שגם מתארים את המשמעות שלהם. תוכנה שיודעת את המשמעות של התגיות שמופיעות בקובץ יכולה "להבין" את התוכן שלו.

להשוואה, הנה קוד HTML שמכיל את אותו המידע:

```
<p>
  <strong>
    <em>Person:</em>
  </strong>
</p>
<p>
  <strong>Name</strong>: Jane Doe
</p>
<p>
  <strong>E-Mail</strong>: anonymous@mail-client.com
</p>
```

אמנם אותו המידע נמצא שם אך המטא-מידע לא מאפשר לקורא הלא האנושי להבין את השדות שמופיעים

עוד דוגמה ליכולת שהיינו מעוניינים בה היא התאמה אישית לאתרים למשל, אם ידוע שלאדם מסוים יש תחומי עניין כלשהו אז כשהוא גולש ל-Amazon היינו רוצים להציג לו ספרים על תחום העניין שלו, או כשהוא נכנס לאתר של שופרסל היינו רוצים מיד להראות לו מבצעים על מוצרים שהוא אוהב ניתן להשיג מטרה זו בעזרת קוקיות אך השיטה הזו לא מאוד גמישה. בפרט, אם כל 5 דקות תחומי העניין שלי או האוכל האהוב עלי משתנים, לקוקיות אין דרך לדעת את זה. יתר על כן, קוקיות שייכות לאתר ספציפי יחיד ואנחנו היינו רוצים להגדיר את הפרופיל של המשתמש פעם אחת בלבד...

כדי לבצע את המשימות הנ"ל צריך לדעת למצוא דפים שמתארים אנשים ואז להוציא מהם את המידע הרלוונטי. וכדי לעשות את זה צריך למצוא דרך לבדוק שדף מתאר בן אדם, להבין איזה מידע צריך להוציא ממנו (הרי ב-HTML כל אחד יכול לעצב את העמוד שלו כרצונו) ולבסוף גם להבין מה בדיוק המידע הזה אומר.

רשת סמנטית (Semantic web) היא בסיס נתונים רשתי שמכיל את הקשרים הסמנטיים בין מרכיביו. בפרט המושג מתייחס לרשת אינטרנט שמכילה מידע סמנטי לגבי הקבצים המצויים ברשת. רשת כזו תכיל מידע שיאפשר שימוש יעיל יותר בתוכן המצוי באינטרנט הן על ידי משתמשים אנושיים והן על ידי מחשבים

המידע הסמנטי שיתווסף אמור להיות נוח יותר לשימוש מהמידע שניתן לכלול בקטגוריות או בתגיות שאפשר לשייך לדפי אינטרנט. בפרט הוא יקל לקבל פרטים על האובייקטים המקושרים לדף האינטרנט שבו מעיין המשתמש, עוד בטרם הוא בוחר לעבור לדפים המקושרים.

רעיון הרשת הסמנטית בהקשר של יישומי אינטרנט הועלה לראשונה על ידי טים ברנרס-לי, ממציא האינטרנט. לדבריו "הרשת הסמנטית תהווה מבנה לתוכן בעל משמעות של דפי האינטרנט, ובכך תיצור סביבה שבה תוכנות שמשוטטות בין דפי אינטרנט יוכלו בקלות לבצע משימות מתוחכמות לטובת המשתמשים" (The Semantic Web will bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users).

יש כמה גישות בסיסיות לבעיה:

- אילו **כל** המידע באינטרנט היה מאורגן באופן מסודר במסד נתונים אחד ענק הבעיה הייתה נפתרת בקלות, אבל זה בוודאי לא המצב.
- גישות של בלשנות חישובית – שיטות מתוחכמות לזיהוי תבניות ומידע.
- גישת DB: נהפוך את הרשת לכאילו מסד נתונים ע"י שימוש ב-XML. זו הגישה שאנחנו נתמקד בה כמובן.

כדי לממש את הרעיון של רשת סמנטית יש צורך בכמה טכנולוגיות

- Extensible Markup Language (XML) – השפה שבאמצעותה מקשרים בין טקסט למשמעותו
- Resource Description Framework (RDF) – מגדיר אובייקטים והיחסים ביניהם
- Web Ontology Language (OWL) – מגדיר אונטולוגיות שמקשרות בין רעיונות שונים (למשל, מכונית היא סוג של כי רכב וכו')
- שירותי רשת – מאפשרים לשירותים שניתנים בצורה מקוונת להיות נגישים באמצעות תוכנות

בימינו כבר יש XML ברשת האינטרנט. למשל, אתרים יכולים להפיץ מידע ב-RSS. RSS (Really Simple Syndication) הוא אוסף תקני XML שמאפשר לאתר אינטרנט לספק תוכן מעודכן ללקוחות, ובפרט לאתרים אחרים, לשם הצגתו בהם. כל קובץ RSS הוא ערוץ תוכן שמורכב מפריטים הכוללים תקציר וקישור לתוכן המלא. הרעיון הוא לאפשר ללקוח לרכז ולשלוט בכמות גדולה של תכנים שמגיעים ממספר אתרים אתרי חדשות מפיצים מידע ב-RSS, אתרי שיתוף מפיצים כך מידע. ניתן להירשם לערוץ מידע כזה למשל ב-Outlook ובכל פעם שנוסף מידע הוא פשוט מגיע באורח קסום למחשב שלנו

החלפת מידע

עם ריבוי מקורות המידע החלפתו ושיתופו הופכת לקשה. לכל מאגר מידע יש סכמה אחרת מסוג אחר. לכן קשה לאחד מידע ממקורות שונים. למשל, אם שתי חברות מחליטות להתאחד הם יצטרכו לעבוד קשה כדי לאחד את מסדי הנתונים שלהם. פתרון פשוט הוא לעבור לשכבת ביניים. כל אחד מהצדדים יעביר את המידע שלו לסכמת ביניים מוסכמת (למשל ל-XML) ואז שני הצדדים יוכלו לדבר אחד עם השני.

הפרדת תוכן מעיצוב

אתרי אינטרנט מתפתחים עם הזמן. אופנת העיצוב והטכנולוגיות הקיימות משתנים אבל התוכן אולי נשאר דומה. לחילופין אולי דווקא העיצוב לא משתנה והתוכן מתחלף כל יום (אתר חדשות למשל). לכן היינו רוצים להפריד את תוכן הדף מהעיצוב שלו, כדי שנוכל לשנות כל אחד מהם בנפרד מבלי להשפיע על השני. יש פה

גם שיקולים ניהוליים – האנשים שעובדים על העיצוב של אתר הם בד"כ לא אותם האנשים שכותבים את התוכן שלו. הם צריכים להיות מסוגלים לעבוד בנפרד.

קיים כבר מנגנון כזה ב-HTML שנקרא CSS. אבל המנגנון הזה מאוד מוגבל ולא מאפשר את כל היכולות שהיינו רוצים. בעזרת XML ניתן לתאר תוכן בלבד ובעזרת XSL (EXtensible Stylesheet Language) ניתן לתרגם את המידע הזה לקובצי HTML. יתר על כן, את המידע ניתן לתרגם בכל מיני דרכים שיתאימו לתצוגה במערכות שונות. למשל, באמצעות XSL ניתן לתרגם XML לפורמט שקריא במכשירים ניידים, או לפורמט קריא בדפדפנים או פורמט טקסט שניתן לפתוח ב-Excel למשל. האפשרויות הן עצומות!

תחביר של XML

השוואה בין XML ל-HTML

XML ו-HTML הן שפות אחיות. שתיהן סוג של SGML (Standard Generalized Markup Language) – שפה שתוכננה במקור לאפשר למחשבים של הממשלה לקרוא ולחלוק מסמכים

ב-HTML יש אוסף קבוע של תגיות ואטריבוטים ולכל אחד מהם משמעות מסוימת דפדפנים מתוכנתים להבין את התגיות האלה וכך הם יודעים איך להציג את המסמכים. לעומת זאת, ב-XML הכותבים יכולים להמציא תגיות ואטריבוטים כרצונם ואין להם משמעות מיוחדת עבור תוכנות כלשהן

כללי תחביר של XML

התחביר של XML הוא מאוד פשוט. המבנה הבסיסי של מסמך XML הוא האלמנט. קטע המסמך שבין תגית פתיחה לתגית הסגירה המתאימה לה נקרא אלמנט. למשל, החלקים המוקפים בקטע הקוד הבא עם כל האלמנטים שלו:

```
<person>
  <name>Jane Doe</name>
  <email>anonymous@mail-client.com</email>
</person>
```

ואילו

```
<name>Jane Doe</name>
<email>anonymous@mail-client.com</email>
```

אינו אלמנט. אלה שני אלמנטים... ☺

נתאר את כללי התחביר של XML:

1. לכל אלמנט של XML חייבת להיות תגית סוגרת. זה בניגוד ל-HTML שם ניתן למצוא אלמנטים מהצורה

```
<p>This is a paragraph
<p>This is another paragraph
```

2. תגיות של XML הן case-sensitive. בפרט התגית הפותחת והתגית הסוגרת צריכות להתאים אחת לשנייה ב-case.
3. קינון אלמנטים: תגית סוגרת של אלמנט y לא יכולה להופיע באמצע אלמנט x. כלומר, קטע הקוד

```
<b>
  <i>This text is bold and italic
  </b>
</i>
```

אינו חוקי.

4. למסמכי XML חייב להיות אלמנט שורש יחיד – אלמנט שהוא אבא של כל האלמנטים האחרים. כלומר, קטע הקוד

```
<name>Jane Doe</name>
<email>anonymous@mail-client.com</email>
```

אינו חוקי.

5. ערכי האטריבוטים באלמנט חייבים להופיע בתוך מרכאות כפולות למשל,

```
<note date=12/11/2007>
  <to>Tove</to>
  <from>Jani</from>
</note>
```

לעומת

```
<note date="12/11/2007">
  <to>Tove</to>
  <from>Jani</from>
</note>
```

6. Entity References: ישנם תווים שיש להם משמעות מיוחדת ב-XML ולא ניתן להשתמש בהם באמצע אלמנט. למשל, כאשר מופיע הסימן < ההנחה היא שמדובר בתגית פתיחה. במקום להשתמש בתווים עצמם יש להשתמש ב-entity references שלהם. יש חמישה entity references כאלה המוגדרים מראש ב-XML:

Entity Reference	Character	Meaning
<	<	less than
>	>	greater than
&	&	ampersand
'	'	apostrophe
"	"	quotation mark

7. הערות: `<!-- This is a comment -->`

8. מסמכי HTML לא מאפשרים לכתוב כמה רווחים ברצף. כלומר, אם למשל ב-HTML כתוב

This   is an   HTML   document

אז הקובץ בדפדפן ייראה כך:

This is an HTML document

ב-XML ניתן לכתוב כמה רווחים שרוצים והם נשמרים.

כל מסמך XML חייב להתחיל בשורה `<?xml version="1.0" encoding="ISO-8859-1"?>`. היא לא מהווה חלק מה-XML. אחריה מגיע אלמנט השורה של המסמך.

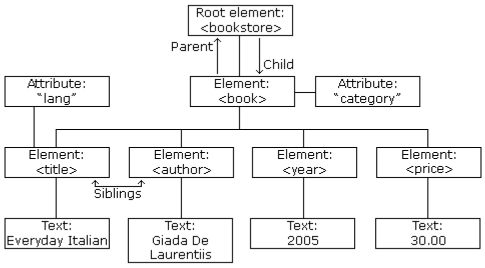
עץ XML

מסמך XML מגדיר גרף מכוון ללא מעגלים. כל אלמנט מיוצג ע"י קודקוד. תתי אלמנטים של אלמנט הם ילדים של הקודקוד המתאים. השורש של העץ הוא אלמנט השורש של המסמך. למשל, מסמך ה-XML הבא

```
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
```

```
    </book>
    <book category="CHILDREN">
      <title lang="en">Harry Potter</title>
      <author>J K. Rowling</author>
      <year>2005</year>
      <price>29.99</price>
    </book>
    <book category="WEB">
      <title lang="en">Learning XML</title>
      <author>Erik T. Ray</author>
      <year>2003</year>
      <price>39.95</price>
    </book>
  </bookstore>
```

מגדיר את העץ



אטריבוטים

לאלמנטים יכולים להיות אטריבוטים שמופיעים בתגית הפתיחה ומתארים את האלמנט. אטריבוטים הרבה פעמים מספקים מידע נוסף שאינו חלק מהנתונים עצמם אבל חשוב לתוכנה שתרצה להשתמש באלמנט כל מה שמיוצג באמצעות אטריבוטים ניתן לייצג גם באמצעות אלמנטים למשל,

```
<person sex="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

לעומת

```
<person>
  <sex>female</sex>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

יכולות להיות כמה בעיות שנובעות משימוש באטריבוטים:

- אטריבוטים לא יכולים להכיל ערכים מרובים (בעוד שאלמנטים יכולים)
- אטריבוטים לא יכולים להכיל מבנים מקוננים) בעוד שאלמנטים יכולים)
- לא ניתן להרחיב בקלות אטריבוטים לשימושים עתידיים

לכן ההמלצה היא לא להשתמש בד"כ באטריבוטים כשכותבים מסמכי XML אם כי זה עניין של טעם...

XML לעומת טבלאות

תוכן של טבלה קל להציג באמצעות מסמך XML. כל שורה יכולה להיות אלמנט ובו מקוננים אלמנטים שמתאימים לעמודות. מאידך המעבר ממסמך XML לטבלאות הוא לא טריוויאלי. קודם כל, לא כל האלמנטים מכילים בהכרח את אותם התת אלמנטים. אז לא ברור איך לחלק את המסמך לטבלאות ועמודות שנית, ב-XML יכול להיות קינון פנימי בלתי מוגבל בעוד שבטבלאות יש רק שני ממדים שלישית, ב-XML תת אלמנט יכול לחזור על עצמו כמה פעמים. לא ברור כיצד לתרגם זאת לטבלה. בהמשך נראה מה בכל זאת ניתן לעשות.

אכיפת מבנה על מסמכי XML

למה לאכוף מבנה?

כדי שנוכל לקחת מסמכי XML ולהתאים להם סכמה של מסד נתונים רלציוני צריך להיות להם מבנה מוגדר.

בעזרת DTD (Document Type Descriptor) נוכל לקבוע מה מסמך תקין צריך להכיל. DTD לא נועד לוודא שהתחביר של המסמך נכון, אלא שיש לו מבנה מסוים של אלמנטים (למשל, לבן אדם יש לכל היותר מספר זהות אחד אבל יכולים להיות הרבה מספרי טלפון). המטרה היא להגדיר את אבני הבניין החוקיות של המסמך הן עבור האלמנטים והן עבור האטריבוטים.

הגדרה ודוגמאות

ה-DTD משתמש בביטויים רגולריים כדי להגדיר את המבנה החוקי של מסמך. לדוגמה, DTD שמתאר עיתון יכול להיראות כך:

```
<!DOCTYPE NEWSPAPER [
  <!ELEMENT NEWSPAPER (ARTICLE+)>
  <!ELEMENT ARTICLE (HEADLINE,BYLINE,LEAD,BODY,NOTES)>
  <!ELEMENT HEADLINE (#PCDATA)>
  <!ELEMENT BYLINE (#PCDATA)>
  <!ELEMENT LEAD (#PCDATA)>
  <!ELEMENT BODY (#PCDATA)>
  <!ELEMENT NOTES (#PCDATA)>
]>
```

כל שורה ב-DTD אומרת איך אמור להיראות אלמנט מסוים. כללי התחביר של הביטויים הרגולריים:

Expression	Meaning
a	Element a
e?	0 or 1 occurrences of expression e
e*	0 or more occurrences of expression e
e+	1 or more occurrences of expression e
e,f	Expression e followed by expression f
e|f	Expression e or expression f (but **not** both)
(e)	Grouping
#PCDATA	Parsed character data – text that WILL be parsed by a parser. The text will be examined by the parser for entities and markup.
#CDATA	Character data – text that will NOT be parsed by a parser.
EMPTY	No content
ANY	Any content
(#PCDATA|a|…|z)*	Mixed content

לדוגמה, הביטוי name,greet?,addr*,(tel|fax)*,email+ פירושו שהאלמנטים הבאים מופיעים לפי הסדר:

- בדיוק אלמנט אחד של name
- לכל היותר אלמנט אחד של greet
- מספר כלשהו של אלמנטים של addr
- מספר כלשהו של אלמנטים של tel או fax בלי חשיבות לסדר
- אלמנט אחד לכל הפחות של email

נתבונן בכמה ביטויים נוספים:

- name,greet?,addr*,tel*,fax*,email+

ביטוי זה שונה מהביטוי בדוגמה משום שכאן קודם כל חייבים להופיע כל האלמנטים של tel (אם יש כאלה) ורק אז מופיעים האלמנטים של fax (אם יש כאלה), ואילו בביטוי המקורי הסדר שלהם לא משנה.

- name,greet?,addr*,(fax|tel)*,email+

ביטוי זה זהה לביטוי המקורי משום שסדר הביטויים בסוגריים שמציינים בחירה אינו משנה

- name,greet?,addr*,(tel|fax)*,email,email*

גם ביטוי זה זהה לביטוי המקורי משום שגם הוא מבטיח שיש לכל הפחות אלמנט email אחד.

- name,greet?,addr*,(tel|fax)*,email*,email

וגם ביטוי זה זהה לביטוי המקורי שהרי הביטויים email,email* ו-email*,email מציינים את אותה השפה.

דוגמה: יש רשימת דרישות עבור אלמנט של מדינה:

- לכל מדינה יש שם בתור האיבר הראשון
- לכל מדינה יש עיר בירה בתור האיבר השני
- למדינה יכול להיות מלך
- למדינה יכולה להיות מלכה

תלמידי הציע DTD עבור דרישות אלה:

```
<!ELEMENT country (name, capital?, king*, queen)>
```

ה-DTD הזה לא מתאים לדרישות מכמה סיבות:

1. למדינה חייבת להיות עיר בירה ואילו ה-DTD מאפשר מצב שבו אין עיר בירה
2. למדינה יכול להיות לכל היותר מלך אחד ואילו ה-DTD מאפשר כל מספר של מלכים
3. יכול להיות שלמדינה אין מלכה ואילו ה-DTD כופה על כל מדינה שתהיה לה בדיוק מלכה אחת

הגדרת אטריבוטים

התחביר הכללי להגדרת אטריבוטים הוא:

```
<!ATTLIST element-name
          attribute-name1 type1 default-value1
          ...
          attribute-nameN typeN default-valueN>
```

יש הרבה types אפשריים אבל אנחנו נדון בחמישה:

CDATA	The value is character data
(en1|en2|..)	The value must be one from an enumerated list
ID	The value is a unique id (unique in the entire document and amongst all elements)
IDREF	The value is the id of any other existing element
IDREFS	The value is a list of other existing ids

ברירת המחדל יכולה להיות אחד מהבאים:

value	The default value of the attribute
#REQUIRED	The attribute value is required
#IMPLIED	The attribute value is not required

לדוגמה:

```
<!ELEMENT height (#PCDATA)>
<!ATTLIST height
    dimension (cm|in) #REQUIRED
    accuracy  CDATA   #IMPLIED
    resiable  CDATA   "yes"
>
```

באמצעות IDREF ניתן לוודא שאלמנטים שמצביעים אליהם אכן קיימים. אך המנגנון הזה לא מאוד מפותח והוא לא מאפשר בדיקות של טיפוסים. למשל, נתבונן ב-DTD הבא:

```
<!DOCTYPE family [
  <!ELEMENT family (person)*>
  <!ELEMENT person (name)>
  <!ELEMENT name (#PCDATA)>

  <!ATTLIST person
        id        ID      #REQUIRED
        mother    IDREF   #IMPLIED
        father    IDREF   #IMPLIED
        children  IDREFS  #IMPLIED>
]>
```

שום דבר לא מבטיח שהמצביע לילדים אכן מצביע לבני אדם ושום דבר לא מבטיח שהמצביע לאמא מצביע לאישה...

1-חד-משמעיות

כדי להקל על השימוש ב-DTD נדרוש שהוא יהיה 1-חד-משמעי (1-unambigious). פירוש הדבר שבזמן הפירוק של המסמך בכל רגע נתון צריך להיות ברור איזה חלק של הביטוי הרגולרי אנחנו נמצאים למשל, נתבונן בביטוי (a,b)|(a,c) אם המסמך שלנו הוא ab אז אחרי קריאת התו הראשון אנחנו לא יודעים אם אנחנו בביטוי הראשון (a,b) או בביטוי השני (a,c). אולי זה נראה חסר משמעות אבל אנחנו רוצים למכן ניתוח מסמכים נרצה שהפורמט יהיה כה שיותר פשוט וחד-משמעי ללא מקום לניחושים. זה גם מאפשר לנו לפרק את הביטוי במעבר אחד מבלי צורך לנחש באיזה ביטוי להתקדם ואז לחזור אחורה במידה שטעינו

אוטומט גלושקוב

ביטויים רגולריים מגדירים שפה – קבוצת כל הביטויים שמתאימים לביטוי. לדוגמה, נתבונן בביטוי הרגולרי a+,(b|c),a*. אזי המחרוזות aba,aaacaaa מתאימות לו ואילו המחרוזות abca ו-aab אינן בשפה שלו. הראשונה אינה בשפה משום שמופיעים מה שני התווים b,c ואילו הביטוי הרגולרי מאפשר רק אחד מהם, והשנייה אינה בשפה משום שהיא אינה נגמרת באות a.

שפה ניתנת לתיאור גם ע"י אוטומט. אוטומט סופי דטרמיניסטי (להלן אס"ד) הוא חמישייה $\langle Q,\Sigma,\delta,q_o,F\rangle$ כאשר:

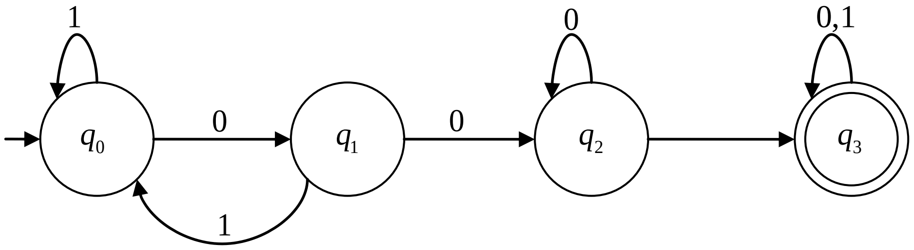
1. Q קבוצה סופית של מצבים
2. Σ א"ב
3. $\delta: Q\times\Sigma\to Q$ פונקציית מעברים
4. $q_0\in Q$ מצב התחלתי
5. $F\subset Q$ קבוצת מצבים מקבלים

בהינתן מילה $w=w_1\dots w_n$, ריצה r של אס"ד M על w היא סדרת מצבים $r=r_0r_1\dots r_n$ ב-Q כך שמתקיימות התכונות הבאות:

- $r_0=q_0$ (הריצה מתחילה במצב ההתחלתי)
- לכל $1\le i<n$ מתקיים $r_{i+1}=\delta(r_i,w_{i+1})$ (הריצה מתקדמת בהתאם לפונקציית המעברים)

ריצה $r=r_0r_1\dots r_n$ היא מקבלת אם $r_n\in F$, אחרת r דוחה. אס"ד M מקבל מילה w אם הריצה של M על w היא מקבלת, אחרת M דוחה את w. שפה של אס"ד היא קבוצת כל המילים שהוא מקבל– $L(M)=\{w\in\Sigma^*: M \text{ accepts } w\}$.

לדוגמה, נתבונן באס"ד



השפה שלו היא שפת כל המילים מעל $\{0,1\}$ שמכילות את הרצף 001.

אוטומט לא דטרמיניסטי הוא בדיוק כמו אוטומט דטרמיניסטי אלא שפונקצית המעברים שלו היא לא דטרמיניסטית. כלומר בהינתן מצב ואות קלט, יש כמה מצבים שניתן לעבור אליהם. עבור אוטומט לא דטרמיניסטי, נאמר שהוא מקבל מילה אם קיימת ריצה כלשהי של האוטומט שמקבלת אותה.

לביטויים רגולריים ניתן להתאים אוטומט מיוחד שנקרא אוטומט גלושקוב ובעזרתו ניתן לקבוע אם הביטוי הוא 1-חד-משמעי או לא.

שלבים ביצירת אוטומט גלושקוב

1. מנרמלים את הביטוי הרגולרי ע"י החלפת כל המופעים של e+ ב-e,e* (שימו לב: **לא** e*,e!)
2. ממספרים את כל המופעים של כל תו עם אינדקס
3. יוצרים מצב התחלתי ומצב לכל תו עם אינדקס

4. בוחרים את המצבים המקבלים להיות המצבים שמתאימים לאותיות שבהן יכולה להיגמר מילה חוקית בשפה
5. יוצרים מעבר ממצב l_i למצב k_j אם יש מילה חוקית בשפה שבה מופיע הצירוף l_ik_j. המעבר הזה מתאים לתו k.

לדוגמה, ניצור אוטומט גלושקוב עבור הביטוי +a*,(b|c),a:

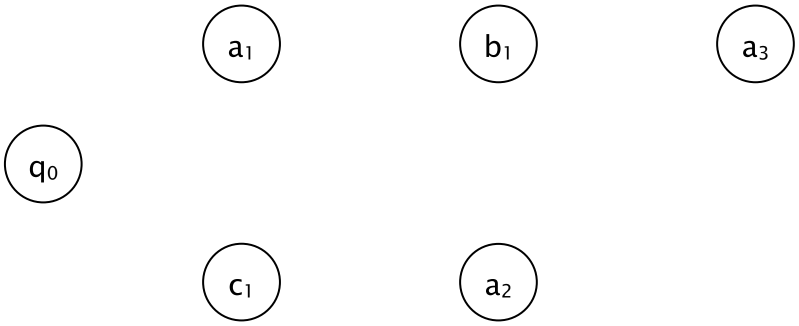
צעד 1: מנרמלים את הביטוי:

$$a^*,(b|c),a+ \rightarrow a^*,(b|c),a,a^*$$

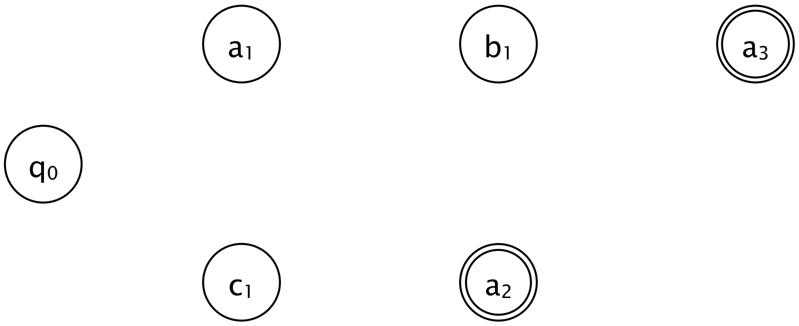
צעד 2: מאנדקסים את כל האותיות:

$$a_1{}^*,(b_1|c_1),a_2,a_3{}^*$$

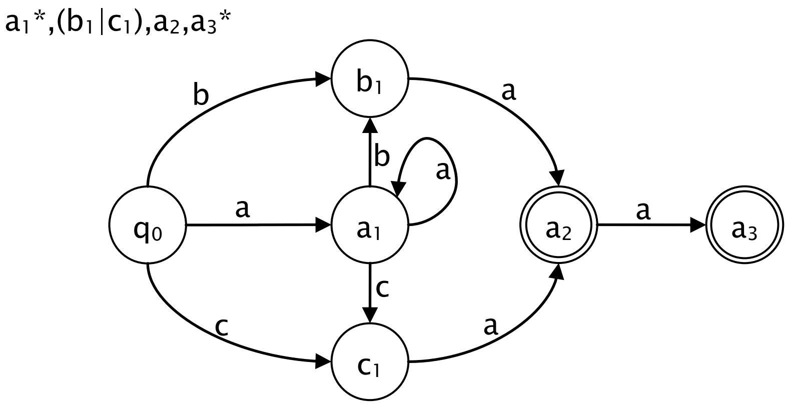
צעד 3: יוצרים מצב התחלתי ומצב לכל אות מאונדקסת:



צעד 4: המצבים המקבצים הם המצבים שמתאימים לאותיות שמילים חוקיות יכולות להיגמר בהן:



צעד 5: נוסיף מעברים לכל זוגות האותיות שיכולות להופיע ברצף במילה חוקית



תכונה חשובה של אוטומט גלושקוב היא שהביטוי הרגולרי שהוא מתאים לו הוא1-חד-משמעי אמ"מ האוטומט שלו הוא דטרמיניסטי. זה מאפשר לנו לבדוק בקלות יחסית אם הביטויים הם אכן 1-חד-משמעיים.

חסרונות של DTD

DTD-ים אמנם יכולים להועיל אבל בכל זאת יש להם כמה מגבלות. יכולת הביטוי של DTD היא מאוד חלשה ביחס לסטנדרטים של מסדי נתונים ושפות תכנות:

- יש רק סוג אחד של אלמנט – PCDATA
- אין אבסטרקציות מועילות (כמו קבוצות וכו')
- ל-IDREF אין טיפוס
- הגדרות התגיות גלובאליות ולא תלויות ב-scope
- לא קל לפרק DTD (בניגוד ל-XML)
- היעילות של תיאור ה-DTD חסומה ע"י יעילות התיאור כביטוי רגולרי
  למשל, אין דרך פשוטה לומר שלאלמנט כלשהו יש בדיוק שלושה ילדים בסדר כלשהו אין ברירה אלא לכתוב את כל שש התמורות של הילדים.

אזהרות

לסיכום ניתן כמה מילות אזהרה. חשוב לא ליצור ב-DTD רקורסיה אינסופית. למשל, נתבונן בהגדרה הבאה:

```
<!DOCTYPE geneology [
  <!ELEMENT geneology (person)*>
  <!ELEMENT person (name,
                    dateOfBirth,
                    person,        <!--mother-->
                    person)        <!--father-->
  >
]>
```

כאן בתוך אלמנט של בן אדם חייבים להופיע בדיוק שני אלמנטים(לא IDREF) של בני אדם. זה יוצר רקורסיה אינסופית ואף מסמך סופי לא יכול להיות תקין לפי הגדרה זו

ננסה לתקן אותה באופן הבא:

```
<!DOCTYPE geneology [
  <!ELEMENT geneology (person)*>
  <!ELEMENT person (name,
                    dateOfBirth,
                    person?,        <!--mother-->
```

```
person?)        <!--father-->
    >
  ]>
```

כאן לא חייבת להיווצר רקורסיה אינסופית כי המסמך יכול להיגמר באלמנט שבו לא מופיעים אלמנטים אחרים של בני אדם. אבל עכשיו ה-DTD שלנו לא 1-חד-משמעי.

הלקח מהדוגמה הזו הוא שזה לא לגמרי טריוויאלי לכתוב DTD-ים וצריך לחשוב היטב מה הדרך הכי טובה ונכונה לעשות זאת.

שליפת מידע מ-XML

אחסון

חישוב שאילתות

שאילתות מילות מפתח

נספח א – הבדלים בין HTTP/1.0 ו-HTTP/1.1

Based on a the paper *Key Differences between HTTP/1.0 and HTTP/1.1* by **Balachander Krishnamurthy**, **Jeffrey C. Mogul** and **David M. Kristol** (can be found at http://www.research.att.com/~bala/papers/h0vh1.html).

Basically there are quite a lot of changes and the paper discusses them thoroughly. I am going to write about the differences that sound more interesting to me.

Introduction

By any reasonable standard, the HTTP/1.0 protocol has been stunningly successful. As a measure of its popularity, HTTP accounted for about 75% of Internet backbone traffic in a (somewhat) recent study.

HTTP/1.0 evolved from the original "0.9" version of HTTP. The process leading to HTTP/1.0 involved significant debate, but never produced a formal specification. The HTTP Working Group (HTTP-WG) of the Internet Engineering Task Force (IETF) produced a document (RFC1945) that described the "common usage" of HTTP/1.0, but did not attempt to create a formal standard out of the many variant implementations. Instead, over a period of roughly four years, the HTTP-WG developed an improved protocol, known as HTTP/1.1.

The HTTP/1.1 specification is almost three times as long as RFC1945, reflecting an increase in complexity, clarity, and specificity. Even so, numerous rules are implied by the HTTP/1.1 specification, rather than being explicitly stated.

We'll now discuss some areas in which there have been significant changes.

Extensibility

The HTTP/1.1 effort assumed, from the outset, that compatibility with the installed base of HTTP implementations was mandatory. Because the HTTP/1.1 effort took over four years, and generated numerous interim draft documents, many implementers deployed systems using the "HTTP/1.1" protocol version before the

final version of the specification was finished. This created another compatibility problem: the final version had to be substantially compatible with these pseudo-HTTP/1.1 versions, even if the interim drafts turned out to have errors in them.

The compatibility issue also underlined the need to include, in HTTP/1.1, as much support as possible for future extensibility. That is, if a future version of HTTP were to be designed, it should not be hamstrung by any additional compatibility problems.

Note that HTTP has always specified that if an implementation receives a header that it does not understand, it must ignore the header. This rule allows a multitude of extensions without any change to the protocol version, although it does not by itself support all possible extensions.

Version numbers

In many cases the version number in an HTTP message can be used to deduce the capabilities of the sender. A companion document to the HTTP specification clearly specified the ground rules for the use and interpretation of HTTP version numbers.

The version number in an HTTP message refers to the hop-by-hop sender of the message, not the end-to-end sender. Thus the message's version number is directly useful in determining hop-by-hop message-level capabilities, but not very useful in determining end-to-end capabilities. For this reason, as well as to support debugging, HTTP/1.1 defines a `Via` header that describes the path followed by a forwarded message. The path information includes the HTTP version numbers of all senders along the path and is recorded by each successive recipient. Note that only the last of multiple consecutive HTTP/1.0 senders will be listed, because HTTP/1.0 proxies will not add information to the `Via` header.)

Upgrading to other protocols

In order to ease the deployment of incompatible future protocols, HTTP/1.1 includes the new `Upgrade request` header. By sending the `Upgrade` header, a client can inform a server of the set of protocols it supports as an alternate means of communication. The server may choose to switch protocols, but this is not mandatory.

Caching

Caching is effective because a few resources are requested often by many users, or repeatedly by a given user. Caches are employed in most Web browsers and in many proxy servers. Caching improves user-perceived latency by eliminating the network communication with the origin server. Caching also reduces bandwidth consumption, by avoiding the transmission of unnecessary network packets. Reduced bandwidth consumption also indirectly reduces latency for un-cached interactions, by reducing network congestion. Finally, caching can reduce the load on origin servers (and on intermediate proxies), further improving latency for un-cached interactions.

One risk with caching is that the caching mechanism might not be “semantically transparent”: that is, it might return a response different from what would be returned by direct communication with the origin server. While some applications can tolerate non-transparent responses, many Web applications (electronic commerce, for example) cannot.

Caching in HTTP/1.0

HTTP/1.0 provided a simple caching mechanism. An origin server may mark a response, using the `Expires` header, with a time until which a cache could return the response without violating semantic transparency. Further, a cache may check the current validity of a response using what is known as a *conditional request*: it may include an `If-Modified-Since` header in a request for the resource, specifying the value given in the cached response's `Last-Modified` header. The server may then either respond with a 304 (Not Modified) status code, implying that the cache entry is valid, or it may send a normal 200 (OK) response to replace the cache entry.

HTTP/1.0 also included a mechanism, the `Pragma: no-cache` header, for the client to indicate that a request should not be satisfied from a cache.

The HTTP/1.0 caching mechanism worked moderately well, but it had many conceptual shortcomings. It did not allow either origin servers or clients to give full and explicit instructions to caches; therefore, it depended on a body of heuristics that were not well-specified. This led to two problems: incorrect caching of some responses that should not have been cached, and failure to cache some responses that could have been cached. The former causes semantic problems; the latter causes performance problems.

Caching in HTTP/1.1

In HTTP/1.1 terminology, a cache entry is *fresh* until it reaches its expiration time, at which point it becomes *stale*. A cache need not discard a stale entry, but it normally must revalidate it with the origin server before returning it in response to a subsequent request. However, the protocol allows both origin servers and end-user clients to override this basic rule.

In HTTP/1.0, a cache revalidated an entry using the `If-Modified-Since` header. This header uses absolute timestamps with one-second resolution, which could lead to caching errors either because of clock synchronization errors, or because of lack of resolution. Therefore, HTTP/1.1 introduces the more general concept of an opaque cache validator string, known as an *entity tag*. If two responses for the same resource have the same entity tag, then they must (by specification) be identical. Because an entity tag is opaque, the origin server may use any information it deems necessary to construct it (such as a fine-grained timestamp or an internal database pointer), as long as it meets the uniqueness requirement. Clients may compare entity tags for equality, but cannot otherwise manipulate them. HTTP/1.1 servers attach entity tags to responses using the `ETag` header.

HTTP/1.1 includes a number of new conditional request-headers, in addition to `If-Modified-Since`. The most basic is `If-None-Match`, which allows a client to present one or more entity tags from its cache entries for a resource. If none of these matches the resource's current entity tag value, the server returns a normal response; otherwise, it may return a 304 (Not Modified) response with an `ETag` header that indicates which cache entry is currently valid. Note that this mechanism allows the server to cycle through a set of possible responses, while the `If-Modified-Since` mechanism only generates a cache hit if the most recent response is valid.

HTTP/1.1 also adds new conditional headers called `If-Unmodified-Since` and `If-Match`, creating other forms of preconditions on requests.

The Cache-Control header

In order to make caching requirements more explicit, HTTP/1.1 adds the new `Cache-Control` header, allowing an extensible set of cache-control directives to be transmitted in both requests and responses. The set defined by HTTP/1.1 is quite large, so we concentrate on several notable members.

Because the absolute timestamps in the HTTP/1.0 `Expires` header can lead to failures in the presence of clock skew, HTTP/1.1 can use relative expiration times, via the `max-age` directive. (It also introduces an `Age` header, so that caches can indicate how long a response has been sitting in caches along the way.)

Because some users have privacy requirements that limit caching beyond the need for semantic transparency, the `private` and `no-store` directives allow servers and clients to prevent the storage of some or all of a response. However, this does not guarantee privacy; only cryptographic mechanisms can provide true privacy.

Some proxies transform responses (for example, to reduce image complexity before transmission over a slow link), but because some responses cannot be blindly transformed without losing information, the `no-transform` directive may be used to prevent transformations.

Bandwidth optimization

Network bandwidth is almost always limited. HTTP/1.0 wastes bandwidth in several ways that HTTP/1.1 addresses. A typical example is a server's sending an entire (large) resource when the client only needs a small part of it. There was no way in HTTP/1.0 to request partial objects. Also, it is possible for bandwidth to be wasted in the forward direction: if a HTTP/1.0 server could not accept large requests, it would return an error code after bandwidth had already been consumed. What was missing was the ability to negotiate with a server and to ensure its ability to handle such requests before sending them.

Range requests

A client may need only part of a resource. For example, it may want to display just the beginning of a long document, or it may want to continue downloading a file after a transfer was terminated in mid-stream. HTTP/1.1 *range requests* allow a client to request portions of a resource. While the range mechanism is extensible to other units (such as chapters of a document, or frames of a movie), HTTP/1.1 supports only ranges of bytes. A client makes a range request by including the `Range` header in its request, specifying one or more contiguous ranges of bytes. The server can either ignore the `Range` header, or it can return one or more ranges in the response.

If a response contains a range, rather than the entire resource, it carries the 206 (Partial Content) status code. This code prevents HTTP/1.0 proxy caches from accidentally treating the response as a full one, and then using it as a cached response to a subsequent request. In a range response, the `Content-Range` header indicates the offset and length of the returned range.

Range requests can be used in a variety of ways:

1. To read the initial part of an image, to determine its geometry and therefore do page layout without loading the entire image
2. To complete a response transfer that was interrupted (either by the user or by a network failure)
3. To read the tail of a growing object.

Expect and 100 (Continue)

Some HTTP requests (for example, the `PUT` or `POST` methods) carry request bodies, which may be arbitrarily long. If, the server is not willing to accept the request, perhaps because of an authentication failure, it would be a waste of bandwidth to transmit such a large request body.

HTTP/1.1 includes a new status code, 100 (Continue), to inform the client that the request body should be transmitted. When this mechanism is used, the client first sends its request headers, and then waits for a response. If the response is an error code, such as 401 (Unauthorized), indicating that the server does not need to read the request body, the request is terminated. If the response is 100 (Continue), the client can then send the request body, knowing that the server will accept it.

However, HTTP/1.0 clients do not understand the 100 (Continue) response. Therefore, in order to trigger the use of this mechanism, the client sends the new `Expect` header, with a value of `100-continue`.

Because not all servers use this mechanism (the `Expect` header is a relatively late addition to HTTP/1.1, and early "HTTP/1.1" servers did not implement it), the client must not wait indefinitely for a 100 (Continue) response before sending its request body. HTTP/1.1 specifies a number of somewhat complex rules to avoid either infinite waits or wasted bandwidth.

Compression

One well-known way to conserve bandwidth is through the use of data compression. While most image formats (GIF, JPEG, MPEG) are pre-compressed, many other data types used in the Web are not. One study showed that aggressive use of additional compression could save almost 40% of the bytes sent via HTTP. While HTTP/1.0 included some support for compression, it did not provide adequate mechanisms for negotiating the use of compression, or for distinguishing between end-to-end and hop-by-hop compression.

HTTP/1.1 makes a distinction between content-codings, which are end-to-end encodings that might be inherent in the native format of a resource, and transfer-codings, which are always hop-by-hop. Compression can be done either as a content-coding or as a transfer-coding.

HTTP/1.0 includes the `Content-Encoding` header, which indicates the end-to-end content-coding(s) used for a message; HTTP/1.1 adds the `Transfer-Encoding` header, which indicates the hop-by-hop transfer-coding(s) used for a message.

HTTP/1.1 (unlike HTTP/1.0) carefully specifies the `Accept-Encoding` header, used by a client to indicate what content-codings it can handle, and which ones it prefers. HTTP/1.1 also includes the `TE` header, which allows the clients to indicate which transfer-codings are acceptable, and which are preferred.

Network connection management

HTTP almost always uses TCP as its transport protocol. The original HTTP design used a new TCP connection for each request, so each request incurred the cost of setting up a new TCP connection. Since most Web interactions are short this was highly inefficient.

Web pages frequently have embedded images, sometimes many of them, and each image is retrieved via a separate HTTP request. The use of a new TCP connection for each image retrieval serializes the display of the entire page on the connection-setup latencies for all of the requests.

To resolve these problems, Padmanabhan and Mogul recommended the use of *persistent connections* and the *pipelining* of requests on a persistent connection.

The `Connection` header

Given the use of intermediate proxies, HTTP makes a distinction between the end-to-end path taken by a message, and the actual hop-by-hop connection between two HTTP implementations.

HTTP/1.1 introduces the concept of `hop-by-hop` headers: message headers that apply only to a given connection, and not to the entire path. The use of `hop-by-hop` headers creates a potential problem: if such a header were to be forwarded by a naive proxy, it might mislead the recipient.

Therefore, HTTP/1.1 includes the `Connection` header. This header lists all of the `hop-by-hop` headers in a message, telling the recipient that these headers must be removed from that message before it is forwarded. This extensible mechanism allows the future introduction of new `hop-by-hop` headers; the sender need not know whether the recipient understands a new header in order to prevent the recipient from forwarding the header.

Because HTTP/1.0 proxies do not understand the `Connection` header, however, HTTP/1.1 imposes an additional rule. If a `Connection` header is received in an HTTP/1.0 message, then it must have been incorrectly forwarded by an HTTP/1.0 proxy. Therefore, all of the headers it lists were also incorrectly forwarded, and must be ignored.

The `Connection` header may also list *connection-tokens*, which are not headers but rather per-connection Boolean flags. For example, HTTP/1.1 defines the token `close` to permit the peer to indicate that it does not want to use a persistent connection. Again, the `Connection` header mechanism prevents these tokens from being forwarded.

Persistent Connections

HTTP/1.0, in its documented form, made no provision for persistent connections. Some HTTP/1.0 implementations, however, use a `Keep-Alive` header to request that a connection persist. This design did not interoperate with intermediate proxies; HTTP/1.1 specifies a more general solution.

In recognition of their desirable properties, HTTP/1.1 makes persistent connections the default. HTTP/1.1 clients, servers, and proxies assume that a connection will be kept open after the transmission of a request and its response. The protocol does

allow an implementation to close a connection at any time, in order to manage its resources, although it is best to do so only after the end of a response.

Because an implementation may prefer not to use persistent connections if it cannot efficiently scale to large numbers of connections or may want to cleanly terminate one for resource-management reasons, the protocol permits it to send a `Connection: close` header to inform the recipient that the connection will not be reused.

Pipelining

Although HTTP/1.1 encourages the transmission of multiple requests over a single TCP connection, each request must still be sent in one contiguous message, and a server must send responses (on a given connection) in the order that it received the corresponding requests. However, a client need not wait to receive the response for one request before sending another request on the same connection. In fact, a client could send an arbitrarily large number of requests over a TCP connection before receiving any of the responses. This practice, known as pipelining, can greatly improve performance. It avoids the need to wait for network round-trips, and it makes the best possible use of the TCP protocol.

Message transmission

HTTP messages may carry a body of arbitrary length. The recipient of a message needs to know where the message ends. The sender can use the `Content-Length` header, which gives the length of the body. However, many responses are generated dynamically. Without buffering the entire response (which would add latency), the server cannot know how long it will be and cannot send a `Content-Length` header.

When not using persistent connections, the solution is simple: the server closes the connection. This option is available in HTTP/1.1, but it defeats the performance advantages of persistent connections.

The Chunked transfer-coding

HTTP/1.1 resolves the problem of delimiting message bodies by introducing the `Chunked` transfer-coding. The sender breaks the message body into chunks of arbitrary length, and each chunk is sent with its length prepended; it marks the end

of the message with a zero-length chunk. The sender uses the `Transfer-Encoding: chunked` header to signal the use of chunking.

This mechanism allows the sender to buffer small pieces of the message, instead of the entire message, without adding much complexity or overhead. All HTTP/1.1 implementations must be able to receive chunked messages.

The Chunked transfer-coding solves another problem, not related to performance. In HTTP/1.0, if the sender does not include a `Content-Length` header, the recipient cannot tell if the message has been truncated due to transmission problems. This ambiguity leads to errors, especially when truncated responses are stored in caches.

Trailers

Chunking solves another problem related to sender-side message buffering. Some header fields, such as `Content-MD5` (a cryptographic checksum over the message body), cannot be computed until after the message body is generated. In HTTP/1.0, the use of such header fields required the sender to buffer the entire message.

In HTTP/1.1, a chunked message may include a *trailer* after the final chunk. A trailer is simply a set of one or more header fields. By placing them at the end of the message, the sender allows itself to compute them after generating the message body.

The sender alerts the recipient to the presence of message trailers by including a `Trailer` header, which lists the set of headers deferred until the trailer. This alert, for example, allows a browser to avoid displaying a prefix of the response before it has received authentication information carried in a trailer.

HTTP/1.1 imposes certain conditions on the use of trailers, to prevent certain kinds of interoperability failure. For example, if a server sends a lengthy message with a trailer to an HTTP/1.1 proxy that is forwarding the response to an HTTP/1.0 client, the proxy must either buffer the entire message or drop the trailer. Rather than insist that proxies buffer arbitrarily long messages, which would be infeasible, the protocol sets rules that should prevent any critical information in the trailer (such as authentication information) from being lost because of this problem. Specifically, a server cannot send a trailer unless either the information it contains is purely optional, or the client has sent a `TE: trailers header`, indicating that it is willing

to receive trailers (and, implicitly, to buffer the entire response if it is forwarding the message to an HTTP/1.0 client).

Internet address conservation

Companies and organizations use URLs to advertise themselves and their products and services. When a URL appears in a medium other than the Web itself, people seem to prefer “pure hostname” URLs; i.e., URLs without any path syntax following the hostname. These are often known as “vanity URLs”, but in spite of the implied disparagement, it's unlikely that non-purist users will abandon this practice, which has led to the continuing creation of huge numbers of hostnames.

IP addresses are widely perceived as a scarce resource. The Domain Name System (DNS) allows multiple host names to be bound to the same IP address. Unfortunately, because the original designers of HTTP did not anticipate the “success disaster” they were enabling, HTTP/1.0 requests do not pass the hostname part of the request URL. For example, if a user makes a request for the resource at URL http://example1.org/home.html, the browser sends a message with the *Request-Line*

```
GET /home.html HTTP/1.0
```

to the server at example1.org. This prevents the binding of another HTTP server hostname, such as exampleB.org to the same IP address, because the server receiving such a message cannot tell which server the message is meant for. Thus, the proliferation of vanity URLs causes a proliferation of IP address allocations.

The Internet Engineering Steering Group (IESG), which manages the IETF process, insisted that HTTP/1.1 take steps to improve conservation of IP addresses. Since HTTP/1.1 had to interoperate with HTTP/1.0, it could not change the format of the Request-Line to include the server hostname. Instead, HTTP/1.1 requires requests to include a `Host` header that carries the hostname. This converts the example above to:

```
GET /home.html HTTP/1.1
Host: example1.org
```

If the URL references a port other than the default (TCP port 80), this is also given in the Host header.

Clearly, since HTTP/1.0 clients will not send Host headers, HTTP/1.1 servers cannot simply reject all messages without them. However, the HTTP/1.1 specification requires that an HTTP/1.1 server must reject any HTTP/1.1 message that does not contain a Host header.

Error notification

HTTP/1.0 defined a relatively small set of sixteen status codes, including the normal 200 (OK) code. Experience revealed the need for finer resolution in error reporting.

The Warning header

HTTP status codes indicate the success or failure of a request. For a successful response, the status code cannot provide additional advisory information, in part because the placement of the status code in the *Status-Line*, instead of in a header field, prevents the use of multiple status codes.

HTTP/1.1 introduces a `Warning` header, which may carry any number of subsidiary status indications. The intent is to allow a sender to advise the recipient that something may be unsatisfactory about an ostensibly successful response.

HTTP/1.1 defines an initial set of `Warning` codes, mostly related to the actions of caches along the response path. For example, a `Warning` can mark a response as having been returned by a cache during disconnected operation, when it is not possible to validate the cache entry with the origin server.

The `Warning` codes are divided into two classes, based on the first digit of the 3-digit code. One class of warnings must be deleted after a successful revalidation of a cache entry; the other class must be retained with a revalidated cache entry. Because this distinction is made based on the first digit of the code, rather than through an exhaustive listing of the codes, it is extensible to `Warning` codes defined in the future.

Other new status codes

There are 24 new status codes in HTTP/1.1. Two of the more notable additions include:

- 409 (Conflict), returned when a request would conflict with the current state of the resource. For example, a PUT request might violate a versioning policy.

- 410 (Gone), used when a resource has been removed permanently from a server, and to aid in the deletion of any links to the resource.

Most of the other new status codes are minor extensions.

Security, integrity, and authentication

In recent years, the IETF has heightened its sensitivity to issues of privacy and security. One special concern has been the elimination of passwords transmitted "in the clear". This increased emphasis has manifested itself in the HTTP/1.1 specification.

Digest access authentication

HTTP/1.0 provides a challenge-response access control mechanism, *Basic authentication*. The origin server responds to a request for which it needs authentication with a `WWW-Authenticate` header that identifies the authentication *scheme* (in this case, "Basic") and *realm*. (The realm value allows a server to partition sets of resources into "protection spaces", each with its own authorization database.)

The client (user agent) typically queries the user for a username and password for the realm, and then repeats the original request, this time including an `Authorization` header that contains the username and password. Assuming these credentials are acceptable to it, the origin server responds by sending the expected content. A client may continue to send the same credentials for other resources in the same realm on the same server, thus eliminating the extra overhead of the challenge and response.

A serious flaw in Basic authentication is that the username and password in the credentials are unencrypted and therefore vulnerable to network snooping. The credentials also have no time dependency, so they could be collected at leisure and used long after they were collected. *Digest access authentication* provides a simple mechanism that uses the same framework as Basic authentication while eliminating many of its flaws.

The message flow in Digest access authentication mirrors that of Basic and uses the same headers, but with a scheme of "Digest". The server's challenge in Digest access authentication uses a nonce (one-time) value, among other information. To

respond successfully, a client must compute a checksum (MD5, by default) of the username, password, nonce, HTTP method of the request, and the requested URI. Not only is the password no longer unencrypted, but the given response is correct only for a single resource and method. Thus, an attacker that can snoop on the network could only replay the request, the response for which he has already seen. Unlike with Basic authentication, obtaining these credentials does not provide access to other resources.

As with Basic authentication, the client may make further requests to the same realm and include Digest credentials, computed with the appropriate request method and request-URI. However, the origin server's nonce value may be time-dependent. The server can reject the credentials by saying the response used a stale nonce and by providing a new one. The client can then recompute its credentials without needing to ask the user for username and password again.

In addition to the straightforward authentication capability, Digest access authentication offers two other features: support for third-party authentication servers, and a limited message integrity feature (through the `Authentication-Info` header).

Proxy authentication

Some proxy servers provide service only to properly authenticated clients. This prevents, for example, other clients from stealing bandwidth from an unsuspecting proxy.

To support proxy authentication, HTTP/1.1 introduces the `Proxy-Authenticate` and `Proxy-Authorization` headers. They play the same role as the `WWW-Authenticate` and `Authorization` headers in HTTP/1.0, except that the new headers are hop-by-hop, rather than end-to-end. Proxy authentication may use either of the Digest or Basic authentication schemes, but the former is preferred.

A proxy server sends the client a `Proxy-Authenticate` header, containing a challenge, in a 407 (Proxy Authentication Required) response. The client then repeats the initial request, but adds a `Proxy-Authorization` header that contains credentials appropriate to the challenge. After successful proxy authentication, a client typically sends the same `Proxy-Authorization` header to the proxy with each subsequent request, rather than wait to be challenged again.

Protecting the privacy of URIs

The URI of a resource often represents information that some users may view as private. Users may prefer not to have it widely known that they have visited certain sites.

The `Referer` header in a request provides the server with the URI of the resource from which the request-URI was obtained. This gives the server information about the user's previous page-view. To protect against unexpected privacy violations, the HTTP/1.1 specification takes pains to discourage sending the `Referer` header inappropriately; for example, when a user enters a URL from the keyboard, the application should not send a `Referer` header describing the currently-visible page, nor should a client send the `Referer` header in an insecure request if the referring page had been transferred securely.

State management

HTTP requests are stateless. That is, from a server's perspective, each request can ordinarily be treated as independent of any other. For Web applications, however, state can sometimes be useful.

Netscape introduced "cookies" in version 1.1 of their browser as a state management mechanism. The IETF subsequently standardized cookies in RFC2109. The basic cookie mechanism is simple. An origin server sends an arbitrary piece of (state) information to the client in its response. The client is responsible for saving the information and returning it with its next request to the origin server. RFC2109 and Netscape's original specification relax this model so that a cookie can be returned to any of a collection of related servers, rather than just to one. The specifications also restricts for which URIs on a given server the cookie may be returned. A server may assign a lifetime to a cookie, after which it is no longer used.

Cookies have both privacy and security implications. Because their content is arbitrary, cookies may contain sensitive application-dependent information. For example, they could contain credit card numbers, user names and passwords, or other personal information. Applications that send such information over unencrypted connections leave it vulnerable to snooping, and cookies stored at a client system might reveal sensitive information to another user of (or intruder into) that client.

RFC2109 proved to be controversial, primarily because of restrictions that were introduced to protect privacy. Probably the most controversial of these has to do with "unverifiable transactions" and "third-party cookies". Consider this scenario.

1. The user visits `http://www.example1.com/home.html`
2. The returned page contains an IMG (image) tag with a reference to `http://ad.example.com/adv1.gif`, an advertisement
3. The user's browser automatically requests the image. The response includes a cookie from `ad.example.com`.
4. The user visits `http://www.exampleB.com/home.html`
5. The returned page contains an IMG tag with a reference to `http://ad.example.com/adv2.gif`.
6. The user's browser automatically requests the image, sending the previously received cookie to `ad.example.com` in the process. The response includes a new cookie from `ad.example.com`

Privacy advocates, and others, worried that:

- The user receives, in step 3, a ("third-party") cookie from `ad.example.com`, a site she didn't even know she was going to visit (an "unverifiable transaction")
- The first cookie gets returned to `ad.example.com` in the second image request

If a `Referer` header is sent with each of the image requests to `ad.example.com`, then that site can begin to accumulate a profile of the user's interests from the sites she visited, here `http://www.example1.com/home.html` and `http://www.exampleB.com/home.html`. Such an advertising site could potentially select advertisements that are likely to be interesting to her. While that profiling process is relatively benign in isolation, it could become more personal if the profile can also be tied to a specific real person, not just a persona. For example, this might happen if the user goes through some kind of registration at `www.example1.com`.

RFC2109 sought to limit the possible pernicious effects of cookies by requiring user agents to reject cookies that arrive from the responses to unverifiable transactions. RFC2109 further stated that user agents could be configured to accept such cookies, provided that the default was *not* to accept them. This default setting was

a source of concern for advertising networks (companies that run sites like `ad.example.com` in the example) whose business model depended on cookies, and whose business blossomed in the interval between when the specification was essentially complete (July, 1996) and the time it appeared as an RFC (February, 1997). RFC2109 has undergone further refinement in response to comments, both political and technical.

Content Negotiation

Web users speak many languages and use many character sets. Some Web resources are available in several *variants* to satisfy this multiplicity. HTTP/1.0 included the notion of *content negotiation*, a mechanism by which a client can inform the server which language(s) and/or character set(s) are acceptable to the user.

Content negotiation has proved to be a contentious and confusing area. Some aspects that appeared simple at first turned out to be quite difficult to resolve. For example, although current IETF practice is to insist on explicit character set labeling in all relevant contexts, the existing HTTP practice has been to use a default character set in most contexts, but not all implementations chose the same default. The use of unlabeled defaults greatly complicates the problem of internationalizing the Web.

HTTP/1.0 provided a few features to support content negotiation, but the RFC never uses that term and devotes less than a page to the relevant protocol features. The HTTP/1.1 specification specifies these features with far greater care, and introduces a number of new concepts.

The goal of the content negotiation mechanism is to choose the best available representation of a resource. HTTP/1.1 provides two orthogonal forms of content negotiation, differing in where the choice is made:

1. In *server-driven* negotiation, the more mature form, the client sends hints about the user's preferences to the server, using headers such as `Accept-Language`, `Accept-Charset`, etc. The server then chooses the representation that best matches the preferences expressed in these headers.
2. In *agent-driven* negotiation, when the client requests a varying resource, the server replies with a 300 (Multiple Choices) response that contains a list of

> the available representations and a description of each representation's properties (such as its language and character set). The client (agent) then chooses one representation, either automatically or with user intervention, and resubmits the request, specifying the chosen variant.

Although the HTTP/1.1 specification reserves the `Alternates` header name for use in agent-driven negotiation, the HTTP working group never completed a specification of this header, and server-driven negotiation remains the only usable form.

Some users may speak multiple languages, but with varying degrees of fluency. Similarly, a Web resource might be available in its original language, and in several translations of varying faithfulness. HTTP introduces the use of *quality values* to express the importance or degree of acceptability of various negotiable parameters. A quality value (or *qvalue*) is a fixed-point number between 0.0 and 1.0. For example, a native speaker of English with some fluency in French, and who can impose on a Danish-speaking office-mate, might configure a browser to generate requests including

```
Accept-Language: en, fr;q=0.5, da;q=0.1
```

Because the content-negotiation mechanism allows qvalues and wildcards, and expresses variation across many dimensions (language, character-set, content-type, and content-encoding) the automated choice of the “best available” variant can be complex and might generate unexpected outcomes

Content negotiation promises to be a fertile area for additional protocol evolution. For example, the HTTP working group recognized the utility of automatic negotiation regarding client implementation features, such as screen size, resolution, and color depth. The IETF has created the Content Negotiation working group to carry forward with work in the area.

Conclusion

HTTP/1.1 differs from HTTP/1.0 in numerous ways, both large and small. While many of these changes are clearly for the better, the protocol description has tripled in length, and many of the new features were introduced without any real experimental evaluation to back them up. The HTTP/1.1 specification also includes

numerous irregularities for compatibility with the installed base of HTTP/1.0 implementations.

This increase in complexity complicates the job of client, server, and especially proxy cache implementers. It has already led to unexpected interactions between features, and will probably lead to others. Fortunately, the numerous provisions in HTTP/1.1 for extensibility should simplify the introduction of future modifications.

Tries – נספח ב

What is a Trie?

A **trie**, or **prefix tree**, is an ordered tree data structure that is used to store an associative array where they keys are usually strings. Unlike a binary search tree, no node in a trie stores the key associated with that node. Instead, its position in the tree shows what key is associated with it. All the descendents of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Strings are usually considered null-terminated, and thus no string can be the prefix of another. Note that without the null-termination, a string CAN be the suffix of another string (i.e. "the girl" and "the girl ate").

Though it is most common, tries need not be keyed by character strings. The same algorithms can be easily adapted to serve similar functions of ordered lists of any construct.

Advantages and Disadvantages

- Generally, when the total set of stored keys is very sparse within their representation space, the trie data structure is quite wasteful in storage.
- Comparing to a regular BST, tries have several advantages:
    - Looking up a key of length m takes $O(m)$ time at worst case. A balanced BST performs $O(\log n)$ comparisons where n is the number of elements in the tree. Thus, in the worst case, a BST takes $O(m \log n)$ time.
    - Tries can require less space when they contain a large number of short strings, because the keys are not stored explicitly and nodes are shared between keys with common initial subsequences.
    - Tries help with longest-prefix matching[10], where we wish to find the key sharing the longest possible prefix of characters all unique.

[10] **Longest prefix match** is an algorithm used by routers in Internet Protocol networking to select an entry from a routing table. Because each entry in a routing table may specify a network, one destination address may match more than one routing table entry. The most specific table entry – the one with the highest subnet mask – is called the longest prefix match. It is called this because it is also the entry where the largest number of leading address bits in the table entry match those of the destination address.

- Comparing to hash tables:
  - Advantages:
    - Looking up data in a trie is faster in the worst case, $O(m)$ time, compared to an imperfect hash table. An imperfect hash table can have key collisions. The worst-case lookup speed in an imperfect hash table is $O(n)$ time, but far more typically is O(1), with O(m) time spent evaluating the hash.
    - There are no collisions of different keys in a trie.
    - Buckets in a trie which are analogous to hash table buckets that store key collisions are only necessary if a single key is associated with more than one value.
    - There is no need to provide a hash function or to change hash functions as more keys are added to a trie.
    - A trie can provide an alphabetical ordering of the entries by key.
  - Disadvantages:
    - Tries can be slower in some cases than hash tables for looking up data, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random access time is high compared to main memory.
    - It is not easy to represent all keys as strings, such as floating point numbers, which can have multiple string representations for the same floating point number, e.g. 1, 1.0, 1.00, +1.0, etc.

Overcoming the Disadvantages – Compressed Tries

When the trie is mostly static, i.e. all insertions or deletions of keys from a prefilled trie are disabled and only lookups are needed, and when the trie nodes are not keyed by node specific data (or if the node's data is common) it is possible to compress the trie representation. This application is typically used for compressing lookup tables when the total set of stored keys is very sparse within their representation space. Another option is to do updating in batches, whenever the batch size exceeds a threshold.

Whereas each field in a trie has to be large enough to hold a pointer, each field in a C-trie is represented by a single bit. If the k-th bit of a node N on level l is set, it

indicates that one or more keys pass through this node N and have as $(l+1)$-th substring $x_{l+1} = k$.

The structure of a node N in level l of a C-trie is as follows:

- U – 1-bit field. If $U = 0$, then the node is internal and B, K, C are as follows. Otherwise, the node is a leaf and B, K, C contain the suffix $x_{l+1} \dots x_k$ of the key $x = x_1 \dots x_k$. Note that the prefix does not have to be stored since it is implicitly defined by the path from the root of the C-trie to the leaf.
- B – 1-bit field. B is set to 1 if one of the keys passing through the node terminates at level l and thus corresponds to a blank field in a trie which contains a key.
- K – $(m-1)$-bit field, each corresponding to a field in a node of a trie. m is the number of characters plus the terminating sign.
- C – at most $\lceil \log_2 n \rceil$ bits where n is the number of keys. C is equal to the number of nonzero bits of the K fields in the nodes on level l to the left of the node N. This number equals the number of nodes on level $l+1$ preceding the successor of the first nonzero bit in the field K of node N.

All the nodes are stored as a continuous bit string. First is the root node, which is followed by all the nodes in level 1 from left to right, and so on.

נספח ג – טיפול בבעיית ה- DUST

נספח ד – אלגוריתם הדירוג HillTop

The following is based on the paper *Hilltop: A Search Engine based on Expert Documents* by **Krishna Bharat** and **George A. Mihaila**

PageRank is based on connectivity. It assumes that authoritative pages tend to point to other authoritative pages. Moreover, it is query-independent. It cannot by itself distinguish between pages that are authoritative in general and pages that are authoritative on the query topic.

This is the problem that the Hilltop algorithm is trying to solve. Hilltop is very similar to PageRank, but the benefit of Hilltop over PageRank is that it is topic-sensitive, and thus it is harder to manipulate (it is easy to mislead PageRank by building link farms).

Hilltop relies on the same assumption that the number and quality of the sources referring to a page are a good measure of the page's quality. However, Hilltop only considers "expert" sources - pages that have been created with the sole purpose of directing people towards resources. In response to a query, it first computes a list of the most relevant experts on the query topic. Then in continues in a similar manner to PageRank. However, it also requires that it can easily locate at least two expert documents voting for the same Web page. Thus if a pool of experts is not available or if Hilltop cannot find a minimum of expert pages for the query topic, the result it will return will be absolute zero. So Hilltop is tuned for result precision rather than coverage. This is the main disadvantage of the algorithm.

Since the ranking phase is similar to that of PageRank (using connectivity analysis) we'll focus on the expert lookup. The authors felt that expert pages need to be objective and diverse. Therefore, in order to find the experts one needs to detect when the sited belong to the same organization.

Two hosts are said to be affiliated if at least one of the following holds:

- They share the same three octets of their IP address
- The rightmost non-generic token in their hostnames are the same.

In a preprocessing step it is possible to construct a host affiliation lookup - groups of hosts that are considered affiliated. Then to find the experts, Hilltop crawls a

search engine's database. Considering all pages with out-degree greater than some threshold k, it tests to see if these URLs point to k distinct non-affiliated hosts. Such pages are considered experts.

To locate expert pages that match queries there's an inverted index that maps keywords to experts. But this is done only on indexed text contained in "key phrases" – pieces of text that qualify one or more URLs in the page.

In response to a user query, Hilltop first computes a list of N experts that are most relevant for the query. Then it ranks results by selectively following the relevant links from these experts and assigning an authority score to each such page.

ביבליוגרפיה