The Bloom Paradox: When not to Use a Bloom filter?

במאמר "$The\ Bloom\ Paradox$" ראינו כי הגישה המסורתית לעבודה עם מסנן בלום לא לוקחת בחשבון את הסיכוי הא-פריורי לשאילתת שייכות. אם יש לנו זיכרון גדול ו-$Cache$ קטן, ואנו מגרילים איבר מסוים בהסתברות אחידה מהזיכרון, הסיכוי שהאיבר אכן יהיה ב-$Cache$ הוא נמוך ורוב הסיכויים שמסנן הבלום יחזיר תשובה שהיא FP.

כתוצאה מכך, בהרבה שאילתות נצטרך לבדוק אם האלמנט נמצא ב-$Cache$, למרות שהוא אינו שם,

וייתכן שהשימוש ב-$Bloom\ Filter$ ייקר את עלות הבדיקה במקום למזער אותה.

נשים לב שהעלות הכוללת לשאילתה במקרה שה-$Bloom\ Filter$ החזיר תשובה שהיא FN,

היא קטנה יותר מעלות בדיקה במקרה של FP, מכיוון שבמקרה הראשון אנחנו נבדוק רק בזיכרון הראשי,

לעומת המקרה השני בו נבדוק ב-$Cache$, נבין שהאלמנט לא נמצא בו, ואז נבדוק בזיכרון הראשי.

נסמן את היחס הבא: $\alpha = W_{FN}/W_{FP}$, ולפי התיאור הנ"ל נעדיף ש-α יהיה גדול ככל הניתן.

אחת התרומות שכותבי המאמר סיפקו, הוא קריטריון הקובע מתי הפרדוקס מתקיים ומתי לא. הם הציגו חסם על ההסתברות האפריורית (שמהווה גבול תחתון) כך שכל הסתברות קטנה ממנו הופכת את מסנן הבלום ללא רלוונטי:

$$\Pr(x \in cache) < \frac{1}{1 + \alpha \cdot 2^{\ln(2)\cdot\frac{m}{n}}}$$

(כאשר n מייצג את מספר האיברים ב-$Cache$ ו-m את מספר הביטים)

נשים לב שככל ש-α גדל, הסיכוי לקיום הפרדוקס גדל גם הוא.

כמו כן, הפרדוקס תלוי גם ביחס הביטים לאלמנטים ב-$Bloom\ Filter$ – מה שמשפיע על כמות ה-FP.

בהתבסס על תובנות אלו, כותבי המאמר הציעו את השיפורים הבאים:

1. נחשב מראש את החסם על ההסתברות האפריורית והפרמטרים המייצגים את הפרדוקס, ובמידה ומבנה הנתונים מספק את הפרדוקס, נגדיר את מסנן הבלום להחזיר תשובה שלילית תמיד (או במקרה של הכנסה, לא נכניס). זה אכן ימעיט את כמות ה-FP, אבל יגדיל את כמות ה-FN שכן תמיד נחזיר $False$.
2. ננסה לחלק את המידע לקבוצות שונות, נגדיר עבור אילו קבוצות יש יותר סיכוי שיופיעו ב-$Cache$ ונבדוק רק עבורם (למשל חלוקה לפי ה-$Subset$ של פקטה שנשלחה).
3. $Counting\ Bloom\ Filter$ - ניתן לחשב את ההסתברות שאיבר נמצא ב-$Cache$ בהסתמך על המספר שמראים המונים שלו: אם כל המונים גבוהים, סביר להניח שהאיבר כבר הוכנס למאגר. פתרון זה עדיין מאפשר FP, מכיוון שיתכן שהמונים נדלקו כל אחד בנפרד עבור איברים אחרים. את הסיכוי נחשב באמצעות הנוסחה:

$$\Pr(x \in Cache|CBF) = \frac{m^k \cdot \left(\prod_{j=1}^{k} c_j\right) \cdot \Pr(x \in Cache)}{m^k \cdot \left(\prod_{j=1}^{k} c_j\right) \cdot \Pr(x \in Cache) + (n \cdot k)^k \cdot (1 - \Pr\,(x \in Cache)}$$

כאשר נחזיר תשובה חיוביות אם ורק אם מתקיים:

$$\Pr(x \in cache|CBF) \geq \frac{1}{\alpha + 1} = P1$$

4. $Selective\ Counting\ Bloom\ Filter$ – מבנה נתונים שבודק לפני כל שאילתת הכנסה האם כתוצאה מההכנסה נגדיל את ההסתברות האפריורית, מה שיגרום לפרדוקס להתקיים. אם כן, לא נוסיף את האיבר ל-BF. ואילו, כאשר נקבל ערך לבדיקת שאילתת שייכות, נשתמש בטכניקה שראינו קודם עבור CBF.

בעת הבדיקה נסתכל גם על המונים שפונקציות הגיבוב מיפו אליהן, ונוודא שאכן כולם גדולים מספיק.

סימולציות:

על מנת לבצע סימולציות, כותבי המאמר מצאו גבול תחתון למספר ה-FP ו-FN. לאחר מכן הם ביצעו סימולציות ובדקו סוגים שונים של BFs. את הבדיקות ביצעו פעמיים: פעם אחת עם יחס שגיאה $\alpha = 5$, ופעם שניה עם יחס שגיאה $\alpha = 100$. במהלך הבדיקות הם גילו שהתרומה של ההסתברות האפריורית משמעותית יותר בשאילתת שייכות מאשר בשאילתת הכנסה, והסבירו זאת בכך שבניסוי שערכו גודל הזיכרון הראשי היה קטן בהרבה מגודל ה $Cache$. לכן, הימנעות מ-FP עבור איברים עם הסתברות אפריורית קטנה הייתה יעילה יותר בשאילתת שייכות מאשר מניעת ההכנסה של איברים עם אותה הסתברות.

לאחר מכן, החוקרים הריצו סימולציה גם על CBF, מה שלא הניב שיפור משמעותי. לעומת זאת, כאשר אימצו את הבדיקה של $SCBF$ לפני שאילתת שייכות, אחוז ה-FP והעלות הכוללת היו נמוכים יותר, והראו שיפור של 11.43%.

החוקרים בדקו גם את ה-$Tradeoff$ בין FP ל-FN וההבדלים בין CBF לבין $SCBF$. הם גילו שב-CBF אם משתמשים בפחות פונקציות גיבוב, אחוז ה-FP גדל, הסיכוי שנקבל תשובות חיוביות קטן, והשיפור של ה-$SCBF$ הופך למשמעותי הרבה יותר. אבל בכל המקרים העלות של $SCBF$ הייתה נמוכה מהעלות של CBF.

הסרת ההנחה הראשית

אחת ההנחות הבולטות שראינו במאמר הייתה שהאיברים שנשלחים לשאילתה נבחרים באופן אקראי ובהסתברות אחידה, כלומר בכל פעם יוגרל איבר כלשהו ועליו תישאל שאילתה במבנה הנתונים שלנו. אחד השימושים העיקרים של $Bloom\ Filter$ הוא למפות, לזהות וכתוצאה מכך לחסום בקלות כתובות IP זדוניות.

ייתכן שיהיו מספר כתובות IP שיחזרו על עצמן בתדירות גבוהה ולכן סביר להניח שכתובות אלה יהיו ב-$Cache$.

כלומר, ההנחה שה-$Data$ מגיע בהסתברות אחידה לא תקפה במקרה זה. נקודה זו העלתה את השאלה האם נוכל לזהות בעלות נמוכה של זמן ריצה ובשימוש מועט של מקום נוסף איברים שחוזרים על עצמם, ורק עליהם להריץ שאילתות ב-$Bloom\ Filter$ לפני שנבדוק אם הם נמצאים ב-$Cache$ או לא.

השינוי בהנחות:

הסרנו מהמאמר הראשי את ההנחה שה-$Data$ יגיע באופן אקראי וייבחר מתוך הסתברות אחידה. ההנחה כעת הינה שה-$Data$ הוא סדרת איברים שנבחרה בהסתברות אחידה ובאופן אקראי אבל בתוכו יש מספר איברים שיחזרו על עצמם בתדירות גבוהה.

ניסיון ראשון לשיפור

הרעיון: ניסינו למצוא אלגוריתם שיבדוק בזריזות על כל איבר שנשאלת עליו שאילתה האם הוא חוזר על עצמו בתדירות גבוהה או לא.

5. אם כן, נעביר אותו ל-$Bloom\ Filter$.
6. אחרת, נדלג על ה-$Bloom\ Filter$ ונבדוק ישירות בזיכרון הראשי.

כל זה נבע מהמחשבה שאולי נוכל להריץ בדיקה קצרה בעלות קטנה מאוד שתמפה לנו על אילו איברים כן נשתמש ב-$Bloom\ Filter$ ועל אילו איברים לא, ובכך נמנע מהפרדוקס.

כך, גם כשהזיכרון הראשי גדול משמעותית מה-$Cache$, לא נבצע יותר מידי בדיקות מיותרות.

<u>האלגוריתם</u>: בהנחה שיש X איברים שחוזרים על עצמם, נגדיר טווח כלשהו ונחלק אותו ל-X תתי טווחים.

לאחר מכן נגדיר פונקציית גיבוב הממפה כל איבר לאחד מבין X הטווחים ולכל טווח נגדיר איבר מוביל.

לפני כל שאילתה, נריץ קודם כל את פונקציית הגיבוב ונגיע לאחד מהטווחים הנ"ל.

כעת נבדוק כמה התוצאה שקיבלנו רחוקה או קרובה לאיבר המוביל באותו טווח.

אם היא קרובה – נמשיך את התהליך עם ה-BF, אחרת נעבור לבדוק ישר בזיכרון הראשי.

בסוף הבדיקה נעדכן את האיבר המוביל להיות קצת יותר קרוב לערך שקיבלנו, כדי שאם נישאל עליו שוב בעתיד אולי כן נבדוק אותו ב-BF, ונקבל את הערך שלו בזריזות. ניתן לעשות זאת למשל על ידי שימוש בנוסחה הבאה:

$$new\ leader = 0.7 \cdot old\_leader\ +\ 0.3\ \cdot current\_value$$

הבעיה העיקרית ברעיון זה היא שאין משמעות לקרבה לערך שפונקציית הגיבוב ממפה אליו (למשל אם פונקציית הגיבוב מיפתה לערך 24 כי הוא באמת איבר מוביל, זה לא אומר שום דבר על איברים 23 ו-25), כמו כן שיטה זו חשופה עדיין להרבה מקרים של FP ו-FN, ולא בטוח שתשפר בהרבה את התוצאות, אם בכלל.

ניסיון שני לשיפור

הפתרון הקודם שהצענו הוביל אותנו לבעיית ה-$Heavy\ Hitters$ – כלומר מציאת האיברים המשפיעים ביותר, אלה שמופיעים הכי הרבה מבין זרם/רשימה של קלטים. בעיה זו דומה לבעיה שלנו, ולכן החלטנו לבדוק האם נוכל להשתמש בפתרונות של בעיית ה-$Heavy\ Hitters$ גם אצלנו.

במהלך החיפושים נתקלנו במבנה הנתונים $Min - Cut$ עליו נרחיב בעמודים הבאים.

הצגת הפתרון החדש לפי מאמר [2]

כפי שראינו, אם ההסתברות האפריורית של אלמנט מסוים מספקת את ה-$Bloom\ Paradox$, תוצאת השאילתה על אלמנט זה ב-$Bloom\ Filter$ ככל הנראה תהיה FP. המאמר מציע מספר פתרונות אפשריים להתמודדות עם בעיה זו והקטנת מספר ה-FP שנקבל:

1. לא להכניס אלמנטים כאלה ל-$Bloom\ Filter$ מלכתחילה, מכיוון שאין ערך לתשובות שנקבל עליהם.
2. לא לבצע שאילתות ל-$Bloom\ Filter$ על אלמנטים כאלה, מאותה הסיבה.
3. שימוש ב $Counting\ Bloom\ Filter$ המאפשר לראות כמה פעמים פונקציות הגיבוב מיפו לאיבר מסוים.
4. שימוש ב-$Selective\ Counting\ Bloom\ Filter$

בפתרונות 1-2 אנחנו מאבדים את היתרון של קיצור זמני תגובה לשאילתות על אלמנטים עם הסתברות אפריורית.
השימוש בפתרון 3 הביא לתוצאות טובות, והשימוש בפתרון 4 הביא לתוצאות טובות עוד יותר.
נציג כעת פתרון נוסף, שיתמודד עם בעיית השייכות של אלמנט למבנה נתונים כלשהו ללא תלות בהסתברות האפריורית שלו.

הרעיון: נתחזק מבנה נתונים (במקום ה-$Bloom\ Filter$) שמחזיק בכל רגע נתון עד K אלמנטים שהופיעו הכי הרבה עד כה. במילים אחרות, נשתמש בפתרון לבעיית ה-$Heavy\ Hitters$ הידועה כפתרון לבעיה שלנו.

הבעיה: לא קיים אלגוריתם שפותר את בעיית ה-$Heavy\ Hitters$ במעבר בודד על הקלט תוך שימוש בכמות תת-לינארית של זיכרון. כדי שהפתרון שאנחנו מציעים יהווה תחליף ראוי ל-$Bloom\ Filter$, נצטרך למצוא אלגוריתם שפותר אותו בזמן טוב.

הפתרון: נשתמש באלגוריתם חלש יותר, $\epsilon - Heavy\ Hitters$, הפותר בעיה חלשה יותר, אך מספקת, לבעיה.
קלט: זרם של n אלמנטים ופרמטרים k, ϵ.
רעיון האלגוריתם: ניהול ערימה שמכילה בכל רגע נתון ערכים המקיימים את התנאים הבאים:

7. כל ערך שהופיע לפחות $\frac{n}{k}$ פעמים עד כה, מופיע בערימה.
8. כל ערך בערימה בהכרח הופיע לפחות $\frac{n}{k} - \epsilon n$ פעמים עד כה.

k מסמן את מספר האיברים שאנחנו מעוניינים לשמור לכל היותר בכל רגע נתון במבנה הנתונים החדש שלנו.
נשים לב שיש $Trade - off$ בין מספר האלמנטים שאנחנו מעוניינים לשמור במבנה הנתונים לבין אפקטיביות האלגוריתם, שכן ככל ש-k גדל כך גדל הסיכוי להחזיק אלמנטים שהתדירות שלהם נמוכה יותר.

ϵ מסמן את הסטייה שאנחנו מוכנים לקבל בתוצאות. ככל ש-ϵ קטן, כך גם הסטייה.
נשים לב שהמקום הנדרש לאלגוריתם זה גדל בקצב של $\frac{1}{\epsilon}$,
כלומר יש $Trade - off$ בין מרווח הסטייה שאנחנו מוכנים לקבל לבין המקום הנדרש על ידי אלגוריתם זה.

מבנה הנתונים $Count - Min$

נציג כעת מבנה נתונים הסתברותי שישמש אותנו לפתרון הבעיה.

מבנה הנתונים $Count - Min$ משתמש ב-l פונקציות $Hash$ שונות הממפות קלטים לערכים בין 0 ל-b כלשהו.

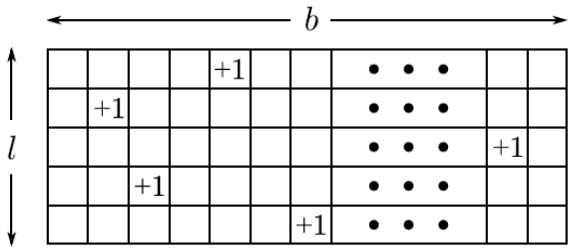
כלומר למבנה נתונים זה יש 2 פרמטרים:

9. הערך b קובע עד כמה אנחנו מכווצים את העולם שלנו (מספר האלמנטים n המקורי) כאשר $b \ll n$.
10. הערך l קובע את מספר הניסיונות הבלתי תלויים למיפוי אלמנט x ל-$Buckets$, בדומה ל-$Bloom\ Filter$ ששם אנחנו ממפים את x באופן בלתי תלוי לביטים שונים, כמספר פונקציות ה-$Hash$ שלנו.

מבנה הנתונים תומך בפעולות הבאות:

$Inc(x)$ – הגדלת ה-$Counters$ של x ב-1.

$Count(x)$ – סכום ערכי ה-$Counters$ של x.



תיאור אלגוריתם לפתרון הבעיה המקורית של המאמר

1. נשמור במשתנה m את מספר האלמנטים שעיבדנו עד לנקודת הזמן הנוכחית.
2. נשמור את ה-$Heavy\ Hitters$ הפוטנציאליים שלנו בערימה בעזרת $Count - Min$ כפי שתיארנו לעיל.
3. בהינתן אלמנט x מהזרם, נריץ עליו $Inc(x)$ ולאחר מכן $Count(x)$ ונפעל באופן הבא:
   - אם $Count(x) \geq \frac{m}{k}$, נוסיף את x לערימה כ-$Tuple$ מהצורה $(x, Count(x))$

     (אם הוא נמצא כבר, נשנה את $Count(x)$ לערך העדכני ונמשיך לאלמנט הבא)
   - אם קיים אלמנט y בערימה כך ש-$Count(y) < \frac{m}{k}$, נוציא אותו מהערימה.

   (החיפוש מתבצע ב-$O(1)$ בעזרת $Find - Min$, והמחיקה מתבצעת בעזרת $Extract - Min$)
4. אם x נמצא בערימה (כלומר $Count(x) \geq \frac{m}{k}$) נחזיר תשובה חיובית לשאילתה, אחרת תשובה שלילית.

<u>הערה</u>: נשים לב שבכל רגע נתון יהיו לנו לכל היותר k אלמנטים בערימה.

(אם כל האלמנטים הופיעו לפחות $\frac{m}{k}$ פעמים, ויש יותר מ-k כאלה, סימן שראינו יותר מ-m אלמנטים עד כה בסתירה)

השוואת מבני הנתונים

נרצה לבצע השוואה בין הפתרון הטוב ביותר שהוצע במאמר המקורי (שימוש במבנה הנתונים $Selective\ Counting\ Bloom\ Filter$), לבין הפתרון שאנחנו הצענו (שימוש במבנה הנתונים $Count - Min$).

לשם כך ביצענו סימולציה ב-$Python$ לפי השלבים הבאים:

1. הגרלנו 2 רשימות של כתובות IP שנבחרו בהסתברות אחידה למעט 20% מגודל הרשימה.
   איברים אלו מדמים את כתובות ה-IP שחוזרים על עצמם בתדירות גבוהה.
2. מימשנו מחלקות המייצגות את $Selective\ Counting\ Bloom\ Filter$ ואת $Count - Min$
3. כל אחד ממבני הנתונים קיבל זרם של כתובות IP, ועיבד את כולן
4. הרצנו מספר רב של שאילתות שייכות בעזרת רשימה נוספת של כתובות IP המכילה כתובות שונות מהכתובות שעובדו על ידי מבנה הנתונים, למעט ה-$Heavy\ Hitters$ עצמם).
5. לבסוף סכמנו את מספר ה-FP, FN, TP, TN של כל אחד ממבני הנתונים.

התוכנית שכתבנו נמצאת ב-$Github$ הבא: https://github.cs.huji.ac.il/idan356/article

הרצות והשוואת תוצאות

<u>סבב ראשון</u>

נבצע מספר הרצות עם ערכי α שונים, ורשימות של 30,000 כתובות IP.

```
Min-Cut Results:

False positive amount: 28
False positive rate: 0.0011656952539550376
True positive amount: 5500
True positive rate: 0.919732441471572
False negative amount: 480
False negative rate: 0.0802675585284281
True negative amount: 23992
True negative rate: 0.998834304746045


Selective Counting Bloom Filter Results:

False positive amount: 24
False positive rate: 0.0009991673605328892
True positive amount: 4320
True positive rate: 0.7224080267558528
False negative amount: 1660
False negative rate: 0.27759197324414714
True negative amount: 23996
True negative rate: 0.9990008326394672
```

$\alpha = 1$

```
Min-Cut Results:

False positive amount: 67
False positive rate: 0.0027847049044056526
True positive amount: 5460
True positive rate: 0.9191919191919192
False negative amount: 480
False negative rate: 0.08080808080808081
True negative amount: 23993
True negative rate: 0.9972152950955944


Selective Counting Bloom Filter Results:

False positive amount: 76
False positive rate: 0.0031587697423108895
True positive amount: 4400
True positive rate: 0.7407407407407407
False negative amount: 1540
False negative rate: 0.25925925925925924
True negative amount: 23984
True negative rate: 0.9968412302576891
```

$\alpha = 5$

```
Min-Cut Results:

False positive amount: 83
False positive rate: 0.003446843853820598
True positive amount: 5460
True positive rate: 0.9222972972972973
False negative amount: 460
False negative rate: 0.0777027027027027
True negative amount: 23997
True negative rate: 0.9965531561461795


Selective Counting Bloom Filter Results:

False positive amount: 143
False positive rate: 0.0059385382059800665
True positive amount: 4560
True positive rate: 0.7702702702702703
False negative amount: 1360
False negative rate: 0.22972972972972974
True negative amount: 23937
True negative rate: 0.9940614617940199
```

$\alpha = 100$

ניתן לראות כי ככל ש α גדל, אחוז ה-TP עולה (באופן יחסי), אך באותה מידה גם אחוז ה-FP גדל.

בנוסף, ניתן לראות ש-$Min - Cut$ מספק תוצאות קצת יותר טובות מבחינת אחוז ה-TP.

שאר הערכים נשארו זהים פחות או יותר.

<u>סבב שני</u>

נבצע שתי הרצות נוספות, הפעם עם ערך $\alpha = 100$, ורשימות של 100,000 כתובות IP.

```
Min-Cut Results:

False positive amount: 19164
False positive rate: 0.22134442134442134
True positive amount: 10400
True positive rate: 0.7749627421758569
False negative amount: 3020
False negative rate: 0.22503725782414308
True negative amount: 67416
True negative rate: 0.7786555786555787


Selective Counting Bloom Filter Results:

False positive amount: 3025
False positive rate: 0.03493878493878494
True positive amount: 4940
True positive rate: 0.368107302533532
False negative amount: 8480
False negative rate: 0.6318926974664679
True negative amount: 83555
True negative rate: 0.9650612150612151
```

ניתן לראות שה-$Bloom\ Filter$ כבר לא מחזיר תוצאות כל-כך טובות, יש עליה משמעותית בכמות ה-FN.

לעומת זאת, אחוזי ההצלחה של $Min - Cut$ לא ירדו באופן משמעותי (סביב ה-0.7).

כלומר למרות שהגדלנו משמעותית את כמות ה-IPs,

$Min - Cut$ הצליח להתמודד עם השינוי בהצלחה ללא שימוש במקום נוסף, וזה יתרון משמעותי.

השוואת כמות זיכרון

<u>$Selective\ Counting\ Bloom\ Filter$</u>

מבנה נתונים זה מחזיק מערך בודד של $Counters$, המתעדכנים לפי תוצאות המיפוי של פונקציות הגיבוב.

מספר הביטים הנדרש לכל תא תלוי בכמות הזיכרון שנרצה להקצות.

ככל שנקצה יותר ביטים לכל $Counter$, כך הוא יוכל להתמודד טוב יותר עם ערכים רבים שממופים לאותו תא.

בסימולציה שלנו בחרנו להשתמש ב-$Bloom\ Filter$ המכיל 33,280 מונים, בדומה לסימולציה מהמאמר המקורי.

לסדרי הגודל שלנו, 4 ביטים לכל מונה מספיקים, ולכן סה"כ מספר הביטים הנדרש על ידי $SCBF$ הינו:

$$num\ of\ couters\ in\ the\ bloom\ filter \cdot 4 = 33{,}280 \cdot 4 = 133{,}120\ bits$$

<u>$Min - Cut$</u>

מבנה נתונים זה מחזיק מטריצה של l שורות (מספר ה-$Hash\ Functions$) ו-b עמודות (מספר ה-$Buckets$).

כל תא במטריצה הוא מונה של 4 ביטים (בדומה ל-$SCBF$). בנוסף, מבנה נתונים זה מחזיק גם עד ל-K $Heavy\ Hitters$. בסימולציה שלנו בחרנו להשתמש ב-30 פונקציות $Hash$ שונות, 350 $Buckets$,

ועד ל-350 $Heavy\ Hitters$. כל $Heavy\ Hitter$ הוא בפועל מחרוזת של IP, ולכן מספיק 32 ביטים לייצוג של כל אחד מהם. סה"כ מספר הביטים הנדרש על ידי $Min - Cut$ הינו:

$$l \cdot b \cdot 4 + k \cdot sizeof(HeavyHitter) = 30 \cdot 350 + 350 \cdot 32 = 21{,}700\ bits$$

ניתן לראות כי מספר הביטים הנדרש למימוש $Min - Cut$ קטן באופן משמעותי מאשר ב-$SCBF$!

מחיקת איברים ממבני הנתונים

בקורס "ארכיטקטורת נתבים ומתגים" למדנו על $Bloom\ Filter$ ועל $Counting\ Bloom\ Filter$.

באחד התרגילים התבקשנו לדון בסוגיה מה קורה כאשר נרצה למחוק איברים ממבנה הנתונים,

והצענו את הפתרונות הבאים:

1. מחיקת האיבר מבלי לעדכן את התאים הרלוונטיים ב-BF, ובכך לאפשר החזרת תשובה שהיא FP – מה שלא היה קיים קודם. ייתכן ובשלב מסוים כל הביטים ידלקו ותשובת ה-BF תהיה חסרת משמעות.
לכן פעם בכמה זמן נצטרך לאתחל את ה-BF ולהוסיף אליו את כל האיברים שנמצאים ב-$Cache$ מחדש.
2. שימוש ב-BF נוסף המציין איברים שנמחקו, ואז על כל שאילתה נצטרך לבדוק גם שהאיבר קיים ב-BF הראשון, וגם שהאיבר לא נמחק ב-BF השני. גם פתרון זה לא מספיק טוב, מכיוון שאם נוסיף איבר שמחקנו, שוב נצטרך להחזיר FN כי ה-BF השני יציין שהאיבר כבר נמחק.
3. שימוש ב-$Counting\ Bloom\ Filter$. בכל פעם שנרצה למחוק את האיבר, נחסר 1 מכל מונה שאליו פונקציית הגיבוב מיפתה. פתרון זה טוב יותר מהפתרונות הנ"ל, אמנם יש לקחת בחשבון שגם לו יש בעיות.
בהנחה שכל מונה מורכב מ-x ביטים, אם נוסיף יותר מ-2^x איברים המונה יתאפס ונאבד את הספירה.

נוכל להעלות את השאלה, מה יקרה אם בניסוי שלנו נרצה לאפשר גם מחיקת איברים.

המאמר המקורי הוביל אותנו לשימוש ב-$Counting\ Bloom\ Filter$, לכן נוכל להשתמש בפתרון מס' 3.

לגבי חריגה של המונים נפעל באופן הבא: אם מקום לא מהווה בעיה, נגדיר שמספר הביטים של כל מונה יהיה לפי גודל ה-$Cache$. אם ה-$Cache$ הוא בגודל 2^8 אז נשתמש ב-8 ביטים לכל מונה. אם שיקולי מקום כן מהווים בעיה, נוכל להגדיר כמות קטנה של ביטים לכל מונה, לספוג את האפשרות שיחזרו FN מעת לעת, ופעם בכמה זמן לאתחל מחדש את מבנה הנתונים שלנו רק עם האיברים שב-$Cache$.

עבור מבנה הנתונים $Min - Cut$, נוכל בדומה ל-CBF לחסר 1 מכל מונה במטריצת המונים שלנו.

לסיכום

ניתן לראות בתוצאות הסימולציה שביצענו, שיש ל-$Min - Cut$ שני יתרונות עיקריים על $SCBF$:

1. הוא דורש פחות מקום.
2. הוא נותן תוצאות טובות גם עבור קלטים גדולים מאוד.

על כן נסיק שלעיתים כדאי להשתמש במבנה הנתונים $Min - Cut$ במקום ב-$Bloom\ Filter$, בהתאם לסיטואציה.

מקורות

1. *The Bloom Paradox: When not to Use a Bloom Filter?*
http://webee.technion.ac.il/~isaac/p/tr11-06_paradox.pdf

2. *Approximate Heavy Hitters and the Count-Min Sketch*
http://theory.stanford.edu/~tim/s15/l/l2.pdf