

deep Sear**Ch**

Search Deeper Find Faster

שם : רחל עוברני

תוכן

5	1. הצעת פרויקט
5	תיאור פרויקט
5	הגדרת בעיה אלגוריתמית
6	רקע תאורטי בתחום פרויקט
14	הליכים העיקריים בפתרון בעיה בטכנולוגיות הנדסה מתקדמות
15	גבולות הפרויקט
15	פיתוחים עתידיים
15	תיאור טכנולוגיות הנדסה
15	מוד נתונים
16	2. מבוא
16	הרקע לפרוייקט
16	תהליכי מחקר
18	סקירה ספרות
18	אתגרים מרכזיים
18	הבעיה אליה התמודדתי
19	הסיבות לבחירת הנושא
19	על איזה צורך הפרויקט עונה
19	הציג פתרונות לבעיה
20	3. מטרות/יעדים
20	מטרות הפרויקט
21	יעדים טכניים
21	5. מדדי הצלחה למערכת
22	4. אתגרים
22	אתגרים טכניים
22	אתגרים במימוש ה- <i>data</i>
22	6. רקע תאורטי
30	7. תיאור מצב קיימ
32	8. ניתוח חליפות מערכת
32	סריקת תוכן הקבצים בצורה יעילה
34	שימוש במבנה נתונים לאינדקס

41	דירוג
44	שפות
46	9. תיאור הבחירה הנבחרת
46	סריית תוכן הקבצים בקרה עיליה
47	שימוש במבנה נתונים לאינדקס
49	דירוג
49	שפה - java
50	10. אפיון המערכת שהוגדרה / מוצעת
50	דרישות המערכת
50	מודול המערכת
51	אפיון פונקציוני
51	ביצועים עיקריים
52	אלוצים
52	11. תיאור הארכיטקטורה
52	ארכיטקטורת רשת
52	תיאור פרוטוקולי תקשורת
52	שרות -לקוח
53	תרשים ארכיטקטורה
54	12. תיאור תהליכי אבטחת מידע במערכת
54	13. ניתוח ותרשים UML / Use cases של המערכת המוצעת
54	Diagram class Design
58	ע"ז מודולים:
58	מבנה נתונים בהם השתמשתי
60	תרשים מחלקות
60	Builder
60	Helper
61	Searcher
62	UI
64	14. רכיבי ממשק
64	16. תיאור התוכנה

17.	תיאור מסכימים	65
18.	תרשים מסכימים	66
19.	פירוט מסכימים	67
20.	קוד התוכנית	70
21.	תיאור מסד הנתונים	73
	מבנה קבצי הנתונים שיצרת'	73
22.	מדריך למשתמש	73
	הפעלה ראשונית	74
	שימוש במערכת	74
	ביצוע חיפוש	74
	הגדירות מתקדמות	74
23.	בדיקות והערכתה	75
24.	ניתוח ייעילות	79
	סיבוכיות מקום	79
	סיבוכיות זמן	80
25.	מסקנות	81
26.	פיתוחים עתידיים	82
27.	ביבליוגרפיה	82

1. הצעת פרויקט

תיאור פרויקט

פרויקט זה מתמקד בפיתוח מנוע חיפושiesel, המועד לאייתור קבצים במחשב באופן מהיר וממוקד. מנוע החיפוש יתמוך בשני סוגי חיפוש עיקריים:

חיפוש מבוסס מאפייני קובץ – חיפוש לפי תכונות סטנדרטיות של קבצים, כגון שם הקובץ, תאሪיך יצירתו, תאሪיך עדכון, סוג הקובץ, וגודלו.

חיפוש מבוסס תוכן – מנוע החיפוש יכול לבצע מתקדם בתוך התוכן של הקבצים עצמם, כולל אייתור מילויות מפתח, ביטויים או מחרוזות טקסט מסוימות הקשורות קבצי טקסט, ומסמכים דיגיטליים.

הגדרת בעיה אלגוריתמית

הבעיה: חיפושiesel של מילה/ביטוי בתוכני הקבצים ובמאפייניהם ע"י בניית אינדקסiesel ודרוג התוצאות המוחזרות.

ח: מספר הקבצים במערכת.

צ: מספר השאלות (queries) של משתמש ביום.

כאשר משתמש מחפש מילה ספציפית בכל הקבצים בזמן החיפוש *יהיה(O)* לכל שאלתה, כלומר(O) לסקר השאלות ביום.

גישה כזו תדרושים המונן ותגרום למערכת להיות איטית.

בנייה אינדקסiesel ודרוג (Indexing)

בבנייה אינדקסisel היא תהליך של ייצירת מבנה נתונים שמאפשר חיפושiesel מהיר וממוקד במאגר המידע. האינדקסisel מכיל מיפוי בין המילים של המסמכים לבין מיקומם במאגר.

הבעיה האלגוריתמית היא כיצד לבנות אינדקסisel בצורה כזו שתספק גישה מהירה ובזמן סביר למילים במסמכים, במיוחד כאשר גודל המסמכים והנתונים עשוי להיות גדול מדי בשבייל לשומר אותו בביטחון בזיכרון המחשב (RAM), והגישה לדיסק הקשיה איטית יותר.

דרוג תוצאות המוחזרות (Ranking)

לאחר ביצוע חיפוש, השלב הבא הוא להציג את התוצאות למשתמש בצורה מסודרת ומודרגת, כך שהתוצאות הרלוונטיות ביותר יהיו בראש הרשימה.

רקע תאורטי בתחום פרויקט

במציאות הדיגיטלית של היום, מחשבים אישיים מכילים כמויות עצומות של מסמכים וקבצים מסוגים שונים – דוחות, מסמכים אישיים, מיילים, רשימות קניות, ואףלו קורות חיים שהצטברו במשך שנים. איתור קובץ מסוים מתוך גדרה של קבצים יכול להפוך למצימה מתוגרת, במיוחד כאשר לא זכרים את שם הקובץ המדויק, את מיקומו או מאפייניהם חיצוניים כמו תאריך השמירה. חיפוש בסיסי לפי מאפייני קובץ, כגון שם הקובץ, תאריך השמירה או גודלו, אינו תמיד מספק פתרון כאשר יש צורך לתר מסמך על פי תוכנו הפנימי, כמו מילה או ביטוי ספציפי.

כאן עולה הצורך למנוע חיפוש שולחני מתקדם, שהוא למעשה מערכת אחיזור מיידית שתפקידה לאתר מידע דיגיטלי המאוחסן במחשב או במערכת מידע. למנוע חיפוש מסווג זה נדרש למשתמשים לחפש מידע הן במאפייני הקבצים והן בתוכן הפנימי שלהם, ובכך להתגבר על הצפת המידע ולהפחית את הזמן הנדרש לאיתו. לדוגמה, אם המשתמש מhapus מסמך המכיל את הביטוי "ישיבת צוות", מנוע החיפוש יוכל למצוא את כל הקבצים שבהם הביטוי הזה מופיע בתוכן, ולא רק בשם הקובץ.

מנוע החיפוש בניו בדרך כלל משלווה מרכיבים עיקריים:

1. **סורק (Spider)** – רכיב שתפקידו לאתר קבצים ותיקיות במחשב, לזהות את תוכנם ולחלץ מתוכם מילוט מפתח. בדומה לפועל הסקייה בראש האינטרנט, בה הסורק עובר בין דפי אינטרנט על ידי לחיצה על קישורים, גם במחשב האישי הסורק עובר בין תיקיות וקבצים ומנתה את תוכנם.
2. **מנוע אינדקס** – רכיב זה אחראי על ארגון התוכן שנמצא על ידי הסורק ליצירת אינדקס, שהוא רשימה מסודרת של מילוט מפתח, כך שניתן היה למצוא את המידע במהירות וביעילות. מנוע אינדקס מאפשר חיפוש מהיר על בסיס תוכן הקבצים, בדומה לאינדקסים שמנועי חיפוש אינטרנטיים מייצרים כדי לאפשר שליפה מהירה של נתונים.
3. **מנוע אחיזור** – לאחר הזנת שאלתה, מנוע האחיזור מאתר את התוצאות מתוך האינדקס ומחזיר אותן למשתמש בצורה מדורגת ומדויקת. תוצאות החיפוש מוצגות לרוב במצבות שונות, כמו תצוגות מקדים או רשימה של שמות קבצים ומסמכים, ומקלות על המשתמש לאתר במהירות את המידע הרלוונטי ביותר.

מנועי החיפוש נבדלים ומשתנים זה מזה בכל אחד מן הרכיבים: באלגוריתם החיפוש של איתור הקבצים, בבנייה האינדקס, ואיזור הנתונים.

בשלב איתור הקבצים ישנו מנوعי חיפוש שמאגרי המידע שלהם נבנים באופן אוטומטי, באמצעות סריקה של האינטרנט על ידי רובוט, וישנו כאלה שהאינדקס שבו מקוטלג המידע אצלם נעשה בידי בני אדם. לעיתים ישנו מקרים שבהם האיתור הוא חצי אוטומטי, וישנה מעורבות אנושית בחלק מהמקרים כמו למשל במנוע החיפוש גוגל.

בשלב של האינדקס, משתרע תחום רחוב של ביצוע מטלה זו, החל מנوعי חיפוש שאין להם מנוע אינדקס כלל, והם מבצעים חיפוש ישיר בקבצים, ועד למנועי חיפוש, שיוצרים אינדקס מפורט, עד לשמרות התכנים בslashes פעם נוספת אצלם.

בשלב של איזור המידע, יש מנועי חיפוש שמאחזרים תצוגה מקדימה של שורה או מספר שורות לכל תוצאה שהם מוצאים, ויש כאלה שמאחזרים שמות של קבצים או אתרים, שבהם נמצא התוכן, ללא כל תצוגה מקדימה.

בדרך כלל בשלב הראשון של איתור הקבצים ויצירת האינדקס נעלמים מעיני המשתמש, שמקבל רק את החלק של איזור המידע.

מהו אינדקס למנועי חיפוש?

איןדקס הוא התהילך של עיון בקבצים, הודעות דואר אלקטרוני וטכנולוגיים אחרים במחשב, וקטלוג של המידע, למשל המילים והמטה-נתונים שכליולים בהם. כאשר אתה מבצע חיפוש במחשב לאחר ייצרת אינדקס, החיפוש משתמש באינדקס של מונחים כדי למצוא תשיבות ביתר מהירות.

בפעם הראשונה שאתה מתחילה ביצירת אינדקס, התהילך יכול להימשך כמה שעות. לאחר מכן, ייצרת האינדקס תבצע במחשב ברקע, תוך כדי שימוש רגיל במחשב, כשרק נתונים מעודכנים יתווסף לאינדקס.

איך ייצרת אינדקס מאיצה חיפושים?

משמעותו של אינדקס בספר, גם אינדקס דיגיטלי אפשר למחשב ולאפליקציות למצוא תכנים ביותר מהירות על-ידי עיון במונחים או במאפיינים נפוצים כמו תאריך היצירה של קובץ. איןדקס בניו במלואו יכול להציג את קבצי המוסיקה הנכונים עבור מונח החיפוש "בטהובן" בטור שבריר שנייה, לעומת זאת השפעולה עשויה להימשך ללא אינדקס.

חיפוש בדפים בודדים אחר מילوت מפתח ונתאים יהיה תהליך איטי מאוד עבור מנوعי החיפוש כדי לזהות מידע רלוונטי. במקום זאת, מנועי החיפוש (כול גוגל) משתמשים באינדקס הפוך.

מהו אינדקס הפוך?

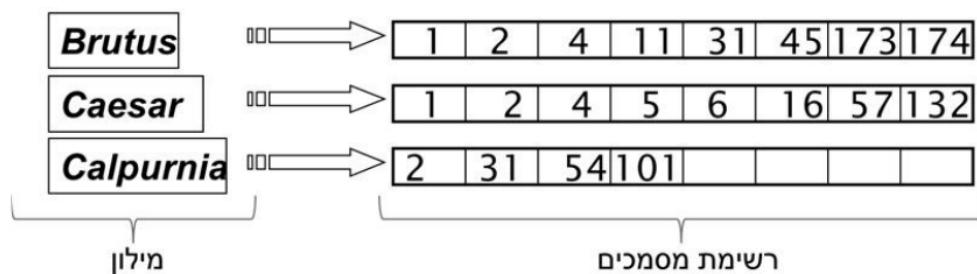
אינדקס הפוך (Inverted Index) הוא מבנה נתונים המשמש במערכות אחזור מידע, כמו מנועי חיפוש, כדי ליעל את תהליך החיפוש והשליפה של מסמכים המכילים מילوت מפתח מסוימת. אינדקס הפוך הוא למעשה מיפוי של מילים או ביטויים מהמסמכים לרשימה של מסמכים שבהם כל מילה או ביטוי מופיעים, ובכך הוא "הפוך" בהשוואה לבניה רגילה, שבו כל מסמך מכיל רשימה של מילים.

במוקם לשומר רשימה של המילים בכל מסמך, האינדקס הפוך יוצר רשימה מסודרת של כל המילים (או המונחים) המופיעות בכל מסמכי המאגר ומקשר כל מילה לרשימת המסמכים שבהם היא מופיעה. מבנה זה מאפשר שליפה מהירה של כל המסמכים שמכילים מילה מסוימת ומפחית את הצורך בעבר על כל המסמכים באופן ישיר.

אינדקס הפוך מורכב משני חלקים עיקריים:

1. **מילון מונחים (Dictionary)** – רשימה של כל המילים או המונחים המופיעים במאגר.
2. **רשימות פרטום (Posting Lists)** – עבור כל מונח במילון, נשמרת רשימה של מסמכים (לרבות Çokודז'יהו) שבהם מופיע המונח, ולעתים קרובות נשמר גם המיקום המדויק של המילה במסמך כדי לאפשר שאלות מתקדמות יותר.

דוגמא לאינדקס הפוך :



הטכנולוגיה מאפשרת לנו לחפש מילה בתוך ים של מידע ב מהירות ובקלות. אך כאשר אנו מעוניינים למצוא משפט או ביטוי, המערכת נתקלת בקשיי ממשמעות. בניית אינדקס לכל המשפטים האפשריים דורשת משאבי מחשב עצומים ואיינה מעשית. כיצד, אם כן, אפשר למצוא את המידע המדויק שהמשתמשים מחפשים, מבלי להטיל עליהם משימה מורכבת זו?

כדי לאפשר חיפוש של משפטיים בתוך קבצים ולא רק מילים יש צורך בחiprosh מלאן:

חיפוש בטקסט מלא כולל סקירת מספר רב של מסמכים וcomaיות עצומות של טקסט. שירות חיפוש באינטרנט משתמש לעיתים קרובות בחיפוש בטקסט מלא כדי לאחזר תוצאות רלוונטיות מהאינטרנט - בין אם זה תוכן דפי אינטרנט,קובצי PDF מקוונים ועוד. בהתחשב בתוכן הטקסט המעורבם, נדרשת טכניקה לטיפול בנפקח החיפוש - זה נקרא אינדקס טקסט מלא.

כדי ליצור אינדקס הפור (inverted index) לחיפוש יעיל במידע טקסטואלי, יש לבצע את השלבים הבאים:

1. פירוק לתווים (Tokenization) – כל מסמך מפורק למילים וסימני פיסוק, כאשר כל מילה או ביטוי הופכים לאסימון (token).

2. עיבוד לשוני (Linguistic Processing) – כל אסימון עבר עיבוד כדי להפוך ל-term אחד, שמייצג את צורתו הבסיסית.

3. שייכות למסמכים (DocID Assignment) – כל term משוייך למזהה המסמן שבו הופיע, כך ש לכל term שמתקבלת רשימה של זוגות <term, מזהה מסמך>.

4. מיזן והסרת כפליות – הרשימה ממוננת לפי הביטויים, ובשלב זה מוסרת כפליות, כך שלכל term תיווצר רשימה מוקורת של המסמכים שבהם הופיע (posting list).

בסיום התהליך מתתקבל אינדקס הפור: מיליון מונחים, אשר כל מונח מקשר לרשימה המסמכים בהם הופיע, מה שמאפשר חיפוש מהיר ויעיל במסמכים על פי תוכן.

"contentDoc ="Apple is looking at buying U.K. startup for \$1 billion

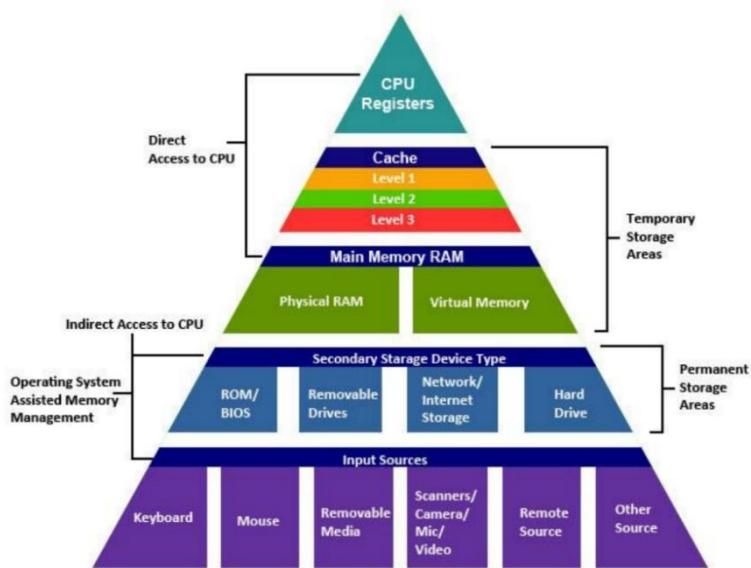
0	1	2	3	4	5	6	7	8	9	10
Apple	is	looking	at	buying	U.K.	startup	for	\$	1	billion

אלגוריתמים לבניית אינדקס יעיל

הנושא המרכזי בפרויקט זה עוסק בתחום בניית האינדקסים (Indexing) ובשיטות השונות בהן מבצעת הוראה. כאשר כמות המסמכים והנתונים היא קטנה, ניתן לשמור את המילון ורשימת ה-posting תהיליך זה. במלואם בזיכרון המחשב. בהנחה זו, פועלות המיון של ה-terms יכולה להבצע ישירות בזיכרון (RAM) בדומה לעיליה פשוטה.

כעת תבחן מזיאות שבהן כמהות הנתונים והמסמכים גדוליה יותר, כך שלא ניתן לשמר את כל הנתונים בזיכרון המחשב בשלמותם. המטרה היא לזהות דרכיים שבהן ניתן לבצע את פעולות האינדוקס בצורה יעילה גם כאשר הגיעו לזמן קיום המוחשב מוגבלת.

לפני הצגת האלגוריתמים השונים, ינתן סקירה של מבנה החומרה, כיוון שבתחום אחזור הנתונים, רבים מההחלטות המערכתתיות תלויות במבנה החומרה הזמין. עיקרונות מרכזיים שנדרש להתייחס אליו הוא שהגישה לנוטונים הנמצאים בזיכרון מהירה משמעותית בהשוואה לגישה לנוטונים המאוחסנים בדיסק הקשיח, וכן גדרשת אופטימיזציה של תהליכי הגישה והיצbold בהתאם למגבליות החומרה.



אחד מהגורםים לזמן הגישה הגבוהים לנוטונים המאוכסנים על גבי הדייסק הקשיח הוא מה שנקרא Time Seek, שמורכב משלושה גורמים: (1) הזמן שלוקח למקום את הראשים של הדייסק הקשיח במקום, (2) הזמן שלוקח לסובב את הדייסק ולhabיא את הנקודה הנכונה אל מתחת לראשים, ו-(3) זמן הקריאה עצמה של הראשים (אנו נתיחס לשתי הפעולות הראשונות ב-*Time Seek*, מכיוון שב-*Time*, לא מתבצעת העברת נתונים, עדיף להعبرיר של גוש אחד גדול של נתונים, שהינה פעולה מהירה יותר, מאשר להعبرיר של מספר גושים קטנים של נתונים. נקודה נוספת שדורשת התייחסות היא שפעולות I/O של הדייסק נעשות בזוגיים, הנקראים בלוקים. כלומר כתיבה או קריאה של נתונים לא נעשית ברמת הבית, אלה בבלוקים (גושים,

נתונים), המכילים מס' בתים כל אחד. מכיוון שברוב המקרים, לאחר שאנו קוראים בית אחד, אנו מעוניינים בקריאה של הבית שנמצא אחריו בקובץ, קריאה של בלוק שלם תהיה עילית יותר מבחינת זמני התגובה. גודלו של בלוק ממוצע הוא בין KB.256 ל-KB8 בשרתים RI מ모וצעים, הדינמי הוא של מספר GB, לעיתים עשרות. גודלו של הdisk הקשיח הוא פי כמה (2-3) גדול מהזיכרון הקיים במחשב. נקודה נוספת אליה נתיחס היא מניעת תקלות בשרתים RI. מניעת תקלות מאד יקרה, ולכן במקום מחשב אחד עמיד ויקר עדיף כמה (עשרות/מאות/אלפים) מחשבים פשוטים וזולים. הטבלה הבאה מציגה מספר ערכים הקשורים לחומרה אשר ישמשו אותנו בהמשך בדוגמאות השונות שלנו. יש לזכור בחשבון שהערכים הללו הם לא ערכים מדויקים, ובහלט יתכן שהם השתו (וחתקצרו) במהלך השנים. אולם, הם עדין מהווים מدد לסדרי הגודל

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

השונים בעבודה בזכרון המחשבים ובעזרת הdisk הקשיח.

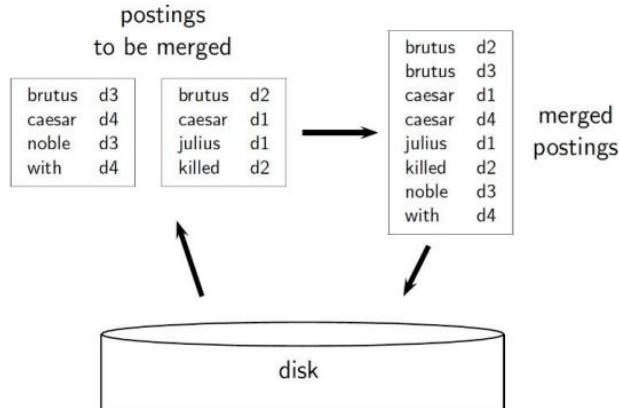
Blocked Sort-Based Indexing :BSBI

נניח ויש לנו כ-100 מיליון רשומות בגודל 12 בתים (המכילות את השדות term, docID, freq, ... שהופקו מייבוד המסמכים. על הרשומות להימין תחילת לפי term ולאחר מכן, במילון שני, לפי docID. בשל מגבלות זיכרון, לא ניתן להציג את כל הרשומות בזכרון המחשב לצורך המילון, ולכן נדרש גישה חלופית.

לצורך כך, הרשומות יחולקו לבלוקים קטנים יותר, בגודל של כ-10 מיליון רשומות לכל בלוק, וכך שניתן להציג בזכרון שני בלוקים בו-זמנית. תהליך בניית הבלוקים מתבצע באופן הבא: במהלך הסיריקה של הקבצים, כל term שנמצא נרשם יחד עם docID שלו לבלוק הנוכחי. כאשר בלוק מתמלא, הוא נשמר לדיסק, ומתחילה למלא בלוק חדש. תהליך זה נמשך עד לסיום הסיריקה של כל המסמכים.

לאחר השלמת הבנייה, כל בלוק מופיע בנפרד בזכרון, שכן גודל הבלוקים מאפשר מילון מהיר ויעיל. בשלב הבא, מאחדים את כל הבלוקים שנשמרו לדיסק לרשימה ממויינת אחת באמצעות תהליך איחוד דמוי merge.

sort: נתונים תחילת הבלוקים לזכרון, ומשווים את הרשומות כדי ליזור רצף ממוין. כאשר נגמר הזיכרון עברו אחד הבלוקים, נתונים קטע נוסף מאותו בלוק לזכרון עד לשיום האיחוד.



דירוג את תוצאות החיפוש

כדי להבטיח שתוצאות חיפוש יהיו רלוונטיות וממוקדות לצרכי המשתמש, יש צורך במנגנון דירוג מתקדם שיקבע אילו מילים מתאימים ביותר לשאלתה שניתנה. אחת הגישות הנפוצות לכך היא דירוג המבוסס על משקלות TF-IDF, שיטה המודדת את הרלוונטיות של כל מילה במסמך ביחס לכל המאגר.

תדירות מונח (term frequency - tf)

נניח שיש לנו קבוצה של מילים טקסט בעברית, וברצוננו לדרג אותם לפי רלוונטיות לשאלתה "אבטיח לא גרעינים". דרך פשוטה להתחיל היא על ידי ביטול מילים שאינן מכילים את כל שלושת המילים "אבטיח", "לא" ו-"גרעינים", אך עדין יוותרו בידינו מילים רבים. כדי להבדיל ביניהם, נרצה לספר את מספר הפעמים שכל מונח מופיע בכל מסמך, ונתון זה נקרא תדירות מונח (term frequency). עם זאת, כאשר אנו ניצבים בפני מילים בעלי אורך שונה, נרצה למדוד תדריות זו באחוזים מכל המילים במסמך, ולא בנתון מספרי יבש.

תדירות מילים הופכית (inverse document frequency - idf)

מכיוון שהמונח "לא" נפוץ כל כך, תדריות המונחים כפי שתוארה לעיל, עלולה להציג באופן שגוי מילים שבסמקרה משתמשים במילה "לא" לעיתים קרובות יותר, מבלי לתת מספיק משקל למונחים המשמעותיים יותר "אבטיח" ו"גרעינים". המונח "לא" אינו מילת מפתח טובה להבחין בין מילים ומונחים רלוונטיים ולא רלוונטיים, בניגוד למילים הפחות נפוצות "אבטיח" ו"גרעינים". לפיכך, יש לשלב את גורם תדריות מילים הופכית

(*Inverse document frequency*) אשר מקטין את משקלם של מונחים המופיעים בתדירות גבוהה מאוד במשמעותנו, ומגדיל את משקלם של מונחים המופיעים לעיתים רחוקות.

השימוש במילה "הופכית" מתייחס לכך שחשיבות המילה היא הופכית למספר המטמכים בהם המילה מופיעה.^[3]

הן TF והן IDF למעשה מהווים גורמי משקל לחישיבות מונח מסוים. TF גבוהה מעלה את משקל החישיבות של המונח, ו-IDF גבוהה מקטין את משקל החישיבות.

תדירות מונח

תדירות מונח, היא התדירות היחסית של מונח t בתוך מסמך

$$tf(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

כאשר $f_{t,d}$ היא מספר הפעמים שמנוח t מופיע במסמך d . המכנה הוא המספר הכולל של מונחים במסמך d (ספרת כל מופיע של מונח בפרט). ישנן דרכים נוספות המגדירות נתון זה, אך האמורה לעיל היא העיקרית והבסיסית ביותר.

תדירות מסמכים הופכית

תדירות מסמכים הופכית היא מدد לכמות המידע שהמילה מספקת, כלומר אם היא נפוצה או נדירה בכל המסמכים. המדריך הוא השבר ההיפוך [בקניה מידה לוגריתמי](#) של המסמכים שמקל את המילה, ומוסג על ידי חלוקת המספר הכולל של המסמכים (למשל, המספר 3 בקורסוס שמונה שלושה מסמכים בלבד) במספר המסמכים המכילים את המונח, ולאחר מכן לkiehot הלוגריתם של אותה מנתה:

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

כאשר:

N : המספר הכולל של מסמכים בקורסוס

df_t : מספר המסמכים שבהם המונח t מופיע.

תדירות מונח – תדירות מסמכים הופכית (TF-IDF)

לאחר שהישבנו כל גורם בנפרד, השילוב של השניים, tf-idf מוחשב כ:

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D)$$

משקל גובה ב-IDF-TF מגיע על ידי תדירות מונח גבוהה (במסגר הנטוון) ותדירות מסמכים נמוכה של המונח בכל אוסף המסמכים; لكن המשקלות נוטות לסן מונחים נפוצים. מכיוון שהיחס בתוך הפונקציה הלוגריתמית של תדירות המסמכים ההופכית תמיד גדול או שווה ל-1, הערך של תדירות המסמכים ההופכית (ובהתאםה של IDF-IDF-TF) גדול או שווה ל-0. ככל שמונה מופיע במסמכים נוספים, היחס בתוך הлогריתם מתקרב ל-1, ומקרב את IDF-IDF-TF ל-0.

הלים עיקריים בפתרון בעיה בטכנולוגיות הנדסה מתקדמות

1. סריקת כל הקבצים במערכת (10-12) – ביצוע סריקה מקיפה של כל קבצי המערכת במטרה לזהות ולאתר את כל הקבצים הקיימים, כולל נתיבי גישה ומיקום מדויק.

2. חילוץ מטה-דטה מהקבצים (20-12) – איסוף מאפייני הקובץ המרכזיים, כגון שם הקובץ, תארכិ, צירה ושינוי, סוג הקובץ ומידע נוסף המאפשר זיהוי ותיעוד מסודר של הקבצים.

3. חילוץ תוכן מקבצים מסווג טקסט, Word ו-PDF (01-01) – שליפת תוכן הטקסטואלי מתוך קבצים בפורמטים נפוצים במטרה להציג את המידע לניטוח ועיבוד בהמשך.

4. תוכני הקבצים ((full-text-search) 01-02) – עיבוד התכנים שנשלפו כדי לזהות מידע מהותי, כגון מילوت מפתח, ביטויים עיקריים ונתונים רלוונטיים נוספים להמשך ניתוח הפרויקט.

5. שמירת המידע מבנה נתונים גמיש לשליפה (01-05) – ארגון כל הנתונים שהופקו במבנה נתונים עיל, המאפשר גישה מהירה ונוחה לשליפת מידע בהתאם לצורכי המערכת.

6. דירוג התוצאות (01-06) - התוצאות שנשלפו ממוקמות לפי רלוונטיות לשאלתה, תוך שימוש באלגוריתמים כמו TF-IDF, שמדרגים את המילים לפי תדירות המילים במסמך ונדרותן במאגר. המטרה היא להציג את התוצאות הרלוונטיות ביותר ראשונות.

גבולות הפרויקט

1. הפרויקט יתמוך במערכת ההפעלה Windows בלבד, ובמערכת קבצים מסווג NTFS.
2. הפרויקט יטפל רק בקבצים קיימים וסטטיים, מבלי לכלול קבצים חדשים, מעודכנים או שנמחקנו.
3. התוכן הטקסטואלי יהיה מהפורמטים הבאים (TXT, Word, PDF).

פיתוחים עתידיים

1. חיפוש בעזרת NLP שיאפשר חיפוש לפי שורש, ומboseו הקשר.
2. דחיסה של האינדקסים
3. רשת מקומית סריקה של מרחב מקומי ולא מחשב פרטי

תיאור טכנולוגיות הנדסה

java – שרת מקומי

C# – קוד לקוחות

מסד נתונים

NoSql

מסדי נתונים NoSQL, ובמיוחד אלה מבוססי מסמכים (כמו MongoDB), מאפשרים לשמר רשומות ללא מבנה קבוע מראש. כך, כל מסמך יכול להכיל שדות ייחודיים לקובץ הספציפי, מה שמקל על שימירת נתונים לא אחידים.

מסדי נתונים NoSQL מאפשר חיפוש בטקסט חופשי ובמבנה גמישים כמו מסמכים, כך שתוכל לשמר ולחשוף תוכן משתנה או לא-מבנה. לדוגמה, אם רוצים לבצע חיפושים בתחום טקסטים, מסדי נתונים (שהוא NoSQL מבוסס על אינדקסים) מתחמכים בחיפושים בטקסט חופשי וודעים לספק תוצאות מהירות גם בכמותות נתונים גדולות.

2. מבוא

הרקע לפרויקט

פרויקט DeepSearch מתמקד בתחום אחזור מידע (Retrieval Information) וניהול נתונים מקומיים. בעידן הדיגיטלי הנוכחי, משתמשים מתמודדים עם כמות עצומה של קבצים ומסמכים במחשבים האישיים שלהם. מציאת מידע ספציפי הופכת למשימה מורכבת, במיוחד כאשר המשתמש אינו זכר את שם הקובץ המדויק או את מקומו, אלא רק חלק מהתוכן שלו.

מנועי החיפוש המובניםמערכות הפעלה לרוב אינם מספקים חיפוש יעיל ומדויק בתוכן הקבצים, ומתקדים בעיקר בשמות קבצים ומטא-דטה בסיסי. לעומת נספהת היא שchiposh ישיר בתוכן הקבצים (לא אינדקס) הוא תהליך איטי מאוד, במיוחד כאשר יש אלפי קבצים במערכת.

תהליכי המחקר

בשלב הראשון של המחקר, חקרותי את עולם מנועי החיפוש, הן ברמה האינטרנטית והן ברמה השולחנית (Desktop Search Engines), במטרה להבין כיצד פועלים מנגנוני חיפוש על גבי מערכות קבצים מקומיות. השוויתי בין כלים קיימים כמו DocFetcher, Windows Search, Everything, ו-Everything כדי להבין אילו שיטות אינדקס וחיפוש הם מיישמים, מה היתרונות והחסרונות של כל אחד, ואילו פערים ניתן לשפר בפרויקט שלי.

עמ' לבנות טבלה הממחישה את הקבצים ומאפשרת חיפוש בצורה מהירה, העמekahtי אין המסדי הנתונים
עובדים:

- חקרתי איך הוא מאחסן את הנתונים. (באיזה סוג קבצים איך הם מוחולקים....)
- וכן איך הוא יוצר את ה `index` - באיזה מבנה נתונים הם משתמשים ומה האפשרות הći עיליה.
- חקרתי איפה כדאי לאחסן את הטבלת `index` מה היתרונות באחסון בדיסק מה היתרונות באחסון בזיכרון RAM ומה היתרונות בשילוב של האחסון בדיסק ובזיכרון RAM.
ע"י הרצת החיפוש ובדיקה של מהירות התגובה.

ע"מ לדרג את תוצאות האיחזור:

עסקתי בחקר אלגוריתמים לדירוג תוצאות חיפוש.

סקרתי שיטות כמו PageRank, TF-IDF, BM25, Cosine Similarity

לכל אחת מהשיטות בדקתי יתרונות וחסרונות, בפרט בהקשר של מנוע חיפוש שלוחני שבו אין יתרון להקשר סטטיסטי של דפי אינטרנט אלא מדובר במסמכים עצמאיים.

נושאים כלליים שחקרתי:

כדי לשפר את ביצועי המערכת, למדתי לעומק על אופן פועלות רכיבי המחשב המשפיעים על פרויקט מסווג זה – בראשם כונני דיסק וזיכרון. בדקתי את הבדלי הביצועים בין דיסקים מסווג HDD ל-SSD, ניהול זיכרון RAM, ושיטות לצמצום צרכית משאבים. ניתחתי את זמן התגובה של פעולות IO, וקבעתי החלטות בהתאם לתעדוף של מהירות תגובה על פני עומק אינדקס.

כמו כן, הקדשתי זמן להבנת היבטי עיליות של זמן ומקום – כיצד לתוכנן את המערכת כך שתספק תוצאות במהירות תוך שמירה על צרכית זיכרון ואחסון נמוכה. השוויתי בין שימוש במبني נתונים בזיכרון (כמו HashMap ו-Trie) לבין פתרונות המבוססים על אחסון קבוע בדיסק, כגון מסדי נתונים או קבצי אינדקס.

בנוסף, חקרתי כלים טכניים וספריות רלוונטיות שיכולות לתרום בפרויקט, כמו ספריות לקרוא קבצים, ניתוח tokenization, ו-Hashmap. חיפשתי איזון בין שימוש בכלים קיימים לבין כתיבה עצמית של חלק מהפונקציונליות, בהתאם לביצועים הנדרשים וליכולת שליטה על הקוד.

לצד כל אלו, עקבתי אחר תיעוד, מאמרים מקצועיים, סרטונים טכנולוגיים ומקורות קוד פתוח כדי להעמיק בכל אחד מהתחומים שנדרשו למימוש הפרויקט.

סיכום ספרות

במהלך ביצוע הפרויקט נעזרתי במקורות מגוונים על מנת להבין לעומק את הנושאים המרכזיים הקשורים לבניית מנוע חיפוש, אינדקס מידע, ושיטות דירוג. בין המקורות המרכזיים:

- האתר Geeks for Geeks שימש להבנה של מושגים בסיסיים ומתקדמים בתחום מבני הנתונים ומגוון החיפוש.
- הספר Schütze- Manning, Raghavan *Introduction to Information Retrieval* מאת הפסקה Index-Based Indexing ו- Single-Pass In-Memory Indexing.
- באתר com.kmwallc.com נלמדו ההבדלים בין גישות שונות בתחום מערכות אחיזור מידע.
- באתר so.ee/pinecone נבחנה ספרות מקצועית בנושא שיטות דירוג לוצאות חיפוש, כולל שימוש במידדים סטטיסטיים כגון IDF (Inverse Document Frequency), TF (Term Frequency), מדדי קוסינוס (Cosine Similarity), וכן האלגוריתם BM25, הנחשב לאחת מהשיטות המתקדמות ביותר בתחום.
- להבנת מבני נתונים תומכים לאינדקס ואחיזור יעיל, כגון Tree B+, נעשה שימוש במאמרים ובתיעוד באתר PlanetScale, העוסק במסדי נתונים מבזרים.

אתגרים מרכזיים

הבעיה אליה התמודדתי

1. **בנייה אינדקס יעיל** - אחד האתגרים המרכזיים היה פיתוח אלגוריתם יעיל לבניית האינדקס ההפוך, במיוחד כאשר מדובר בכמותות גדולות של טקסט. היה צורך למצוא פתרון שייאזן בין מהירות הבניה לבין יעילות החיפוש העתידי.
2. **ניהול זיכרון אופטימלי** - בזמן סריקת קבצים רבים ובניית האינדקס, היה אתגר משמעותי בניהול הזיכרון כדי למנוע דיליפות זיכרון או שימוש יתר במשאבים.

3. **דירוג תוצאות אפקטיבי** - פיתוח אלגוריתם דירוג שיציג את התוצאות הרלוונטיות ביותר למשתמש היה אתגר טכני מורכב, במיוחד כאשר מדובר בחיפוש ביטויים או מילים נפוצות.

הסיבות לבחירת הנושא

הרצון ליצור כלי שישפר את יכולת המשתמשים לנוט בכמות המידע הגדלות, ובמקביל ללמידה לעומק על טכנולוגיות מתקדמות בתחום אחזור המידע ומבנה נתונים ייעילם.

הפרויקט עונה על הצורך בחיפוש מהיר ויעיל בתוכן הקבצים, המאפשר למשתמשים למצוא מידע רלוונטי גם כאשר הם זוכרים רק חלקים מהמידע ולא את שם הקובץ או מיקומו.

על איזה צורך הפרויקט עונה

בעידן שבו כמות הקבצים במחשב האישי ומערכות ארגוניות הולכת וגדלה, קיים צורך ממשי בכל חיפוש מהיר, מדויק ונוח לשימוש, שאינו תלוי בחיבורים לרשת, ואין דרוש התקנת תוכנות צד שלישי. משתמשים רבים נתקלים בקשיים לאתר קבצים או תוכן טקסטואלי בתוך קבצים באופן מיידי, במיוחד כאשר מדובר במבנה תקין מורכבים או בכמותות גדולות של מידע.

מנוע החיפוש השולחני שפותח בפרויקט זה נותן מענה לצורך זה באמצעות מערכת מקומית, קלה להתקנה, המאפשרת לבצע חיפושים מדויקים הן לפי שם הקובץ או התיקייה והן לפי תוכן פנימי של קבצי טקסט, תוך תמיכה בדירוג תוצאות לפי רלוונטיות. המערכת מספקת למשתמש חוותית שימוש אינטואיטיבית ויעילה, המותאמת לצרכים אמיתיים של חיפוש קבצים בסביבות עבודה אישיות וארגוניות, ללא תלות בטכנולוגיות חיצונית או משאבים מורכבים.

הצגת פתרונות לבעה

לאור האתגרים בזיהוי, חיפוש ואחזור מידע טקסטואלי מקבצים מקומיים בהיקפים גדולים, הוצעו מספר פתרונות עקרוניים לפתרון הבעיה:

- **יצירת אינדקס חיפוש מקומי:** בניית אינדקס הפוך (Inverted Index) לצורך חיפוש מהיר ואפקטיבי של מונחים בטקסטים ממספר רב של מסמכים.
- **עיבוד עיל של מסמכים:** חלוקת התוכן לחידות מידע קטנות (Tokenization), והמרתם לבניה חיפוש מותאם, כולל טיפול במילון מונחים ומזהי מסמכים.

- **ניהול משאבים בזיכרון ו디יסק:** שימוש בגישות שמאפשרות טיפול באוספים גדולים, באמצעות חלוקה לבלוקים, מיזן חוץ-זיכרון או בניית ישרה של אינדקסים מהזיכרון.
- **שימוש במבני נתונים מתקדמים:** שילוב מבני נתונים שונים (כגון Hash Table, LSM Tree, B+, Tree, Trie) על פי מגבלות ביצוע, קצב גישה ומשאבים.
- **מודלים לחישוב רלוונטיות תוצאות:** דירוג תוצאות החיפוש לפי איקות התאמת, באמצעות מודלים כגון BM25, TF-IDF ודמיון קוסיני (Cosine Similarity).
- **התמקדות בחוויה המשתמש:** שמירה על זמני תגובה מהירים, ממשק חיפוש אינטואיטיבי, ויכולת ניתוח שאילותות באופן פשוט.

3. מטרות/יעדים

מטרות הפרויקט

המטרה העיקרית של פרויקט DeepSearch היא פיתוח מנוע חיפוש יעיל ומהיר למחשב אישי, שיאפשר למשתמשים לאתר מידע בקלות תוך קבצים המאוחסנים במחשבם. המערכת נועדה להתגבר על הקושי באיתור מידע ספציפי בתוך כמויות גדולות של קבצים.

המטרות המשניות:

- **יצירת ממשק משתמש נוח וידידותי שיאפשר חיפוש מהיר וקל.**
- **שיפור חוויית המשתמש בניהול ואיתור מידע במחשב האישי.**
- **הפחיתה הזמן הנדרשiaeitor מידע ספציפי בתוך קבצים.**
- **הנגשת יכולות חיפוש מתקדמות שבדרך כלל קיימות רק במנועי חיפוש אינטרנטיים.**
- **יצירת תשתיות שניتنת להרחבה לסוגי קבצים נוספים בעתיד.**

יעדים טכניים

- **חילוץ תוכן מקבצים** - פיתוח יכולת להלץ טקסט מקבצים בפורמטים שונים (txt, pdf, doc, docx,) xml, json, csv, log, md, java, htm, html) לצורך אינדוקט.
- **בנייה אינדקס הפקה יעיל** - פיתוח מנגנון לבניית אינדקס הפקה (Inverted Index) המבוסס על מבנה נתונים, שיאפשר חיפוש מהיר של מילים וביטויים בתוך תוכן הקבצים.
- **אלגוריתם דירוג תוצאות** - מימוש אלגוריתם לדירוג תוצאות החיפוש לפי רלוונטיות.
- **שמירת אינדקס בדיסק** - פיתוח מנגנון יעיל לשמר את האינדקס ישירות בדיסק ללא תלות במסד נתונים חיצוני, תוך מקסום ביצועי גישה.
- **ממשק משתמש אינטואיטיבי ב JavaFX** - פיתוח ממשק משתמש מודרני וידידותי באמצעות JavaFX שיאפשר למשתמשים לחפש ולסנן תוצאות בקלות.
- **ביצועים מהירים** - השגת זמן תגובה מהיר בחיפושים (פחות משנה לרוב החיפושים).
- **יעילות בשימוש במשאבים** - אופטימיזציה של צריכת המשאבים (זיכרון ומעבד) במהלך הסריקה, האינדוקט והחיפוש.

5. מדריך הצלחה למערכת

1. **זמן תגובה** - חיפוש מילה או ביטוי בתוך מאגר של לפחות 10,000 קבצים בזמן קצר מ-1 שניה.
2. **שימוש משאבים אופטימלי** - צריכת זיכרון RAM לא תעלה על 500 MB בזמן פעולה רגילה, ולא יותר מ-3 GB בזמן בניית האינדקס.
3. **חיסכון בזמן למשתמש** - קיצור זמן האיתור של מידע ספציפי.
4. **תמייה בסוגי קבצים** - תמייה מלאה בקבצי טקסט (TXT), Word ו-PDF ללא שגיאות בחילוץ התוכן.

4. אתגרים

אתגרים טכניים

1. **טיפול בשגיאות ויציבות** - התמודדות עם קבצים פגומים, הרשאות חסרות, או קבצים הנמצאים בשימוש על ידי תוכניות אחרות.
2. **אחסון יעיל של האינדקס** - בחירת מבנה נתונים אופטימלי עבור אחסון האינדקס ההפוך באופן שיאפשר גישה מהירה, תוך שימוש יעילות בשימוש במשאבים.
3. **איזון בין דיסק לביוצוי דיסק** - כשהאינדקס גדול מדי להיטען כולו לזכרון RAM, נוצרת דילמה בין הعلاאת האינדקס כולו (שפוגעת בביוצוי המערכת ולא עילה עם כמות נתונים גדולות) לבין קריאות מרובות מהדיסק במהלך החיפוש (שגורמות לעיכוב משמעותי ומאטות את התוצאות באופן דרמטי).
4. **פורמט בינארי מול קריאות** - יישום האלגוריתם בכתיבת האינדקסים בפורמט ביןארי לחיסכון במקום דיסק הקשה משמעותית על תהליכי הדיבוג והפיתוח, מכיוון שהמידע לא היה קרא ישירות.

אתגרים במימוש ה-data

- טיפול בקידודי טקסט שונים** - התמודדות עם קבצים בקידודים שונים (UTF-8, ASCII וכו') דרשה פיתוח מנגנון גמיש לזיהוי ועיבוד של הטקסט.

6. רקע תאורי

מהו זכרון RAM?

לכל מחשב בימינו ישנו כרטיס זכרון ram אחד לפחות, כרטיס זה אינו מיועד לשימרת מידע לצמויות אלא לבני לשימרת מידע זמני, כרטיס זיכרון זה מהיר מאוד ביחס לדיסק הקשיח, ומיעוד בעיקר לאחסון התוכנות שאנו עובדים עליהם ברגע זה באופן זמני כדי שייפעלו במתינות מירבית, זיכרון זה מוגבל בכמות המידע אותו הוא יכול לאחסן אך מומלץ להרחיב אותו במידה ונתנו רצאים לעבוד על מספר רב של תוכנות במקביל, בשפה המקצועית זיכרון זה נקרא "זיכרון נדי" או באנגלית Random Access Memory简称 random access memory למה הוא

נראה זיכרון נדייף? כי הוא אינו שומר את המידע עד שימושיים אותו (זה מה שעושה הדיסק הקשיח) המידע מאוחסן בו מתנדף ממנו ברגע שאנו סוגרים תוכנה מסוימת שעבדנו עליה. שם נוסף שהוא לו הוא "זיכרון העבודה" הוא אחראי על אחסון התוכנות אותן עוסדים בלבד ובאופן זמני כMOVED.

כפי שציינו בהתחלה, ה RAM - משמש זיכרון לטוווח קצר של המחשב כדי ליעיל את העבודה שלו, אבל מה באמת הוא עושה ? ובכן, מפני שהמעבד לא יכול לשמר אצלו הרבה מידע בזמן עיבוד נתונים, כמו למשל עבודה על קובץ טקסט, הוא חייב "לאחסן" את המידע במקום מסוים שהוא יכול לגשת אליו בצורה הcy מהירה שאפשר.

אבל כאן יש איזושהי בעיה קטנה, אם משתמש יבקש מהמחשב לשלווף מידע מהכונן הקשיח, ייקח לו הרבה זמן יחסית לשלווף אותו ובסופו של דבר הוא יצטרך להמתין עד שקובץ או תוכנה מסוימת יפעלו.

ה RAM - הוא זיכרון אחסון מהיר שבו המעבד משתמש כדי לאחסן קבצים "זמןניים" על מנת שייהיו מוכנים לשיליפה "מהירה" לעומת הכנים הקשיחים. לדוגמה: כל המידע שנאגר מעבודה עם תוכנות או קבצים מסוימים במחשב, גלישה באינטרנט ואפילו צפיה בסרטונים מאוחסן בזיכרון RAM - לשימוש בעת הצורך.



אי RAM עובד?

זיכרון ה- RAM מורכב מהרבה קבלים וטרנזיסטורים קטנים מאוד שיכולים לאחסן מטען חשמלי המיצג בעצם פיסות מידע קטנות, בדיקן כמו במעבדים ושאר רכיבי חומרה אחרים שקיימים במחשב. מפני שה- RAM מסוגל לאחסן מטען חשמלי במשך זמן קצר מאוד, יש לרענן את המטען החשמלי הזה לעיתים קרובות יותר, אחרת המידע בזיכרון ה- RAM עלול להיעלם.

מהו דיסק קשיח?

דיסק קשיח (בשמו המלא Hard disk drive (HDD)), הוא רכיב במחשב המשמש לשימרת נתונים. דיסק קשיח יכול להכיל בדרך כלל כמות גדולה של נתונים לעומת זיכרונות אחרים, אך פועלתו איטית לעומת הזיכרון הפנימי של המחשב (RAM). הדיסק הקשיח הוא התקן זיכרון בלתי נדיף המאפשר אחסנה אמינה של נתונים דיגיטליים בנפח גדול ובזמן קצר יחסית לתקני זיכרון מכניים אחרים. בהשוואה לתקני זיכרון אחרים באותה הקיבולת, הדיסק הקשיח זול משמעותית, אולם זמן הגישה לנ נתונים בדיסק ארוך בהשוואה לזכרון הפנימי (RAM), דיסק קשיח מכני איטי פי 100,000 מהזיכרון הפנימי.



אופן פועלות הדיסק והשפעתו על מהירות הקריאה והכתיבה

בעת פיתוח מנוע חיפוש מקומי, שבו עיקר הפעולות מבוססות על גישה לקבצים, קריאה וכתיבה תכופות של נתונים, נדרש להבין את אופן פועלתו של אמצעי האחסון – הדיסק. פועלות אלו, למרות שהן נראות פשוטות, עשויות להיות נקודות תורפה משמעותית ביצועים, בעיקר כאשר לא נלקחות בחשבון מגבלות טכנולוגיות בסיסיות של חומרת האחסון.

סוגי דיסקים: HDD לעומת SSD

1. (HDD) Hard Disk Drive:

דיסק מגנטי מסורתי הכלול זרוע מכנית הנעה מעלה פלטוט מסתובבות. כל פעולה קריאה או כתיבה מחייבת חיפוש (seek) של הזרוע אל המיקום המתאים, פעולה המօסיפה השהיה. زمن גישה ממוצעים נעים סביב 5–15 מילישניות, כאשר מהירות הקריאה הרציפה מגיעה לרוב עד 150 MB/s (תליי בדגם).

2. (SSD) Solid State Drive:

דיסק המבוסס על זיכרון פלאש ללא רכיבים מכניים. זמן הגישה נמוכים מאוד (בממוצע פחות מ-0.1 מילישניות), ומהירות הקריאה והכתיבה עשויה להגיע ל-500 MB/s ואף יותר (ב-NVMe – אף

מעל 3/s/GB). עם זאת, גם ב-SSD, ביצוע קריאות רבות של קבצים קטנים בקצבים לא רציפים (Random Access) עלול להיות איטי יותר מהצפי עקב עיכובים פנימיים ובקה של בקר הזיכרון.

הבדל בין RAM לדיסק קשיח

פונקציונליות:

- זיכרון RAM הוא זיכרון נדייף הפועל כאזרור אחסון זמני עבור נתונים שהמחשב משתמש בהם באופן פעיל. הוא מספק גישה מהירה למידע שהמעבד צריך כדי לבצע משימות, כגון הפעלת תוכניות ו אחסון נתונים שנמצאים בעיבוד פעיל.
- HDD הוא זיכרון לא נדייף המספק אחסון לטוח אורך עבור קבצים, יישומים ומערכות הפעלה. הוא מורכב מדיסקים מסתובבים וחילקים מכניים שקוראים וכותבים נתונים בצורה מגנטית. לעומת HDD איטיים יותר מכונני SSD אך לרוב מציעים נפח אחסון גדולים יותר בעלות נמוכה יותר.

מהירות:

- זיכרון RAM הוא סוג הזיכרון המהיר ביותר הזמן במערכת מחשב. הוא מספק גישה כמעט מיידית לנתונים, ומאפשר למעבד לאחסן מידע במהירות.
- דיסקים קשיחים מסוג HDD איטיים יותר בהשוואה ל-RAM וגם ל-SSD. הדיסקים המסתובבים והחילקים המכניים יוצרים עיכוב בגיןה לנתונים, וכתוצאה מכך מהירות קריאה וכתיבה איטיות יותר.
- כונני SSD מהירים יותר מכונני HDD אך איטיים יותר מ-RAM. הם מציעים מהירותים קריאה וכתיבה מהירותים יותר משמעותית בהשוואה לכונני HDD מסורתיים, וכתוצאה מכך זמני אתחול וקצב הטעינה נתונים מהירים יותר.

קיבולת:

- קיבולת זיכרון RAM קטנה בדרך כלל מקיבולת האחסון של SSD או HDD. זיכרון RAM נמדד בדרך כלל בג'גה-בייט (GB) ובדרך כלל נע בין 4 GB ל-32 GB או יותר.
- דיסקים קשיחים מסוג HDD מציעים בדרך כלל קיבולת אחסון גדולות יותר בהשוואה לכונני SSD. הם זמינים בגודלים שונים, הנעים בדרך כלל בין כמה מאות גיגה-בייט (GB) למספר טרה-בייט (TB).
- כונני SSD מציעים מגוון של יכולות אחסון, מכמה מאות גיגה-בייט (GB) ועד למספר טרה-בייט (TB). הם מספקים איזון בין קיבולת אחסון וביצועים.

עלות:

- זיכרון RAM בדרך כלל יקר יותר ליחידת אחסון בהשוואה ל-SSD ו-HDD. העלות של זיכרון RAM עולה ככל שהקיibilitה והמהירות גדלים.
- HDD: דיסקים קשיים הם בדרך כלל האפשרות החסכונית ביותר לאחסון. הם מציעים יכולות אחסון גדולות יותר בעלות נמוכה יותר בהשוואה לכונני SSD.
- SSD: בדרך כלל יקרים יותר מכונני HDD אך זולים יותר מ-RAM ליחידת אחסון. העלות של כונני SSD ירדה עם הזמן, מה שהופך אותם לזולים יותר.

מנועי חיפוש באינטרנט

מנוע חיפוש הוא מערכת מידע המאפשרת למצוא תוכן באינטרנט באמצעות שאלות. מנוע החיפוש סורק את האינטרנט באופן מתמיד, יוצר אינדקס של התוכן ומדרג את התוצאות על פי אלגוריתמים מורכבים. כאשר משתמש מבקש שאלה, המנוע מציג את התוצאות הרלוונטיות ביותר.

מנועי חיפוש מוביילים:

- גוגל (Google): מנוע החיפוש הפופולרי ביותר בעולם. מציע תוצאות רלוונטיות במהירות רבה ומספק שירותים נוספים כמו חיפוש תמונות, וידאו, חדשות ומפות. האלגוריתם מתעדכן באופן קבוע לשיפור איכות התוצאות.
- בינג (Bing): מנוע החיפוש של מיקרוסופט. מציע תוצאות מדויקות עם תכונות ייחודיות כמו חיפוש וידאו משופר ואינטגרציה עם שירותי מיקרוסופט אחרים כמו OneDrive ו-Office.
- דאק דאק גו (DuckDuckGo): מנוע חיפוש המדגש את פרטיות המשתמשים ואין עוקב אחריהם. מספק תוצאות ללא התאמת אישית, מה שמבטיח חווית חיפוש ניטרלית ומאובטחת.

מנוע חיפוש קבצים

מנוע חיפוש קבצים הוא כל תוכנה המאפשר לאתר במהירות קבצים במחשב או ברשת ארגונית. הוא פועל באמצעות סריקת תיקיות ויצירת אינדקס של הקבצים, מה שמאפשר תוצאות מהירות גם בסביבות עם אלפי קבצים. מנועי חיפוש מתקדמים יכולים לחפש לפי שם הקובץ, תוכן, תאריך יצירה, גודל וסוג קובץ.

אינדקסים - הבסיס למנועי חיפוש

מהו אינדקס?

אינדקס הוא מבנה נתונים שמייעל חיפוש מידע, בדומה לאינדקס באנציקלופדיה. במקום לסרוק את כל המידע, האינדקס מאפשר גישה מהירה למיקום המדויק של המידע הרלוונטי.

אינדקסים בסיסי נתונים

אינדקס בסיסי נתונים הוא עותק מאורגן של חלק מהטבלה, בדרך כלל שדה קצר, השמור במקום פיזי נפרד בדיסק. כאשר מטבחעת השאילתה, תהליך בשם Query Optimizer מחליט אם לגשת לטבלה המקורית או להשתמש באינדקס.

עקרונות לאינדקס יעיל

- אין להפוך יותר מדי טורים לאינדקס
- אין לכלול שדות ארוכים (כמו LONGTEXT) באינדקס

מהו טבלת אינדקס הפוכה? (Inverted Index)

במדי' המחשב, אינדקס הפור (המכונה גם רשימת פרסומים, קובץ פרסומים או קובץ הפור) הוא אינדקס בסיס נתונים המאחסן מיפוי מתווכן, כגון מילים או מספרים, למיקומי בטבלה, או במסמך או בקובצת מסמכים (בניגוד לאינדקס קדמי, הממחה מסמכים לתווכן). מטרתו של אינדקס הפור היא לאפשר חיפושי טקסט מלא מהירים, במחair של עיבוד מוגבר כאשר מסמך מתווסף לבסיס הנתונים.

אינדקס הפור הוא מבנה נתונים המשמש במערכות אחיזור מידע לאחיזור עיל של מסמכים או דפי אינטרנט המכילים מונח ספציפי או קובצת מונחים. באינדקס הפור, האינדקס מאורגן לפי מונחים (מילים), וכל מונח מצביע על רשימת מסמכים או דפי אינטרנט המכילים את אותו מונח. כדי לחפש מסמכים המכילים מונח מסוים או קובצת מונחים, מנוע החיפוש שואל את האינדקס ההפור עבור אותם מונחים ומORIZר את רשימת המסמכים הקשורים לכל מונח.

אחסון נתונים ואינדקסים

אחסון נתונים בטבלאות:

הדים מחולק לבלוקים (Pages) בגודל קבוע (לדוגמה 8KB)

כל בלוק מכיל שורות מהטבלה או חלקו של שורות

השורות נשמרות ברכף או מפוזרות לפי סדר הכנסות

אחסון אינדקסים:

- האינדקס נשמר בקבצים נפרדים על הדיסק
- כל אינדקס מיוצג עצם נפרד

מהו דירוג תוצאות?

TF-IDF מייצג תדריות מונח-תדריות מסמך הפוכה. משקל IDF-TF הוא משקל המשמש לעיתים קרובות באחזור מידע וכריית טקסט. וריאצית של סכימת השקלול IDF-TF משמשות לעיתים קרובות על ידי מנועי חיפוש בኒקוד ודירוג הרלוונטיות של מסמך הנთון שאילתה. משקל זה הוא מدد סטטיסטי המשמש להערכת חשיבותה של מילה למסמך באוסף או קורפוס. החשיבות גדלה באופן פרופורציוני במספר הפעמים שמילה מופיעה במסמך אך מקוזחת על ידי תדריות המילה בקורסוס (מערך הנתונים).

בקשר של תוצאות חיפוש TF-IDF, עוזר לדרג דפי אינטרנט על בסיס הרלוונטיות שלהם לשאלתה ספציפית. זה מבטיח שמנועי חיפוש מספקים את התוצאות הרלוונטיות ביותר לשאלות המשתמשים. בכל פעם שמשתמש מקליך שאילתת חיפוש למנוע חיפוש כמו Google רשות מורכבת של אלגוריתמים קופצת לפעולה, עובדת קשה כדי לספק את תוצאות החיפוש הרלוונטיות ביותר. אחד המדריכיםasisים שעלייהם האלגוריתמים האלה מסתמכים כדי לקבוע רלוונטיות ותוכן הוא TF-IDF.

麥肯zie ש TF-IDF-הו מאפשר להעריך את החשיבות היחסית של מונח בתוך מסמך, מנוע חיפוש יכול להשתמש בשיטה זו לצורך דירוג תוצאות החיפוש לפי מידת הרלוונטיות, כאשר תוצאות הרלוונטיות יותר משתמשות ציוני TF-IDF גבוהים יותר TF-IDF. בוחן מילים בהתאם לרלוונטיות שלהן.

קריטריונים אפשריים לדירוג

ה TF-IDF-הו המכפלה של שתי סטטיסטיות, תדריות מונח ותדריות מסמך הפוכה. ישנן דרכים שונות לקביעת הערכים המדוייקים של שתי הסטטיסטיות. תדריות מונח, $TF(t,d)$, היא התדריות היחסית של מונח t בתוך מסמך d , כאשר p , ft הוא הספירה הגלומית של מונח במסמך, כלומר, מספר הפעמים שמונח t מופיע במסמך d .

תדירות המסמך ההפוכה היא ממד לכמה מידע המילה מספקת, כלומר, כמה נפוצה או נדירה היא בכל המסמכים. משקל גבוה בTF-IDF מושג על ידי תדירות מונח גבואה (במסמך הנוכחי) ותדירות מסמך נמוכה של המונח בכלל המסמכים; המשקלים נוטים לסנן מונחים נפוצים

מילים נפוצות כמו "the" או "and" מופיעות בתדירות גבואה כמעט כל הטקסטים באנגלית. אם היינו משתמשים רק על תדירות מונח, אלגוריתם מנוע חיפוש עלול בטעות להקצות חשיבות יתר לדפים ופחות נפוצות כמו "אורגני", "קופה" ו"شعועית". כדי לתקן זאת, נעשה שימוש בגין הדיעת תדירות מסמך הפוכה IDF. IDF מקטין את המשקל של מונחים המופיעים בתדירות גבואה מאוד במסמכים רבים) כמו "the" או ("and" ו מגביר את המשקל של מונחים המופיעים לעיתים רחוקות (כמו "شعועית קפה אורגני")

דוגמה לשיטות דירוג כמו TF-IDF

TF-IDF (תדירות מונח-תדירות מסמך הפוכה) הוא סטטיסטיקה מספרית המשמשת למדידת חשיבותה של מילה בתחום ביחס לקורפוס של מסמכים. הוא משמש בדרך כלל במשימות אחזור מידע ועיבוד שפה טبيعית, כגון אופטימיזציה למנועי חיפוש.

TF-IDF משלב שני רכיבים: תדירות מונח (TF) ותדירות מסמך הפוכה (IDF). תדירות מונח (TF) מודד כמה פעמים מילה מופיעה במסמך. תדירות גבואה יותר מצעיה חשיבות גדולה יותר. אם מונח מופיע בתדירות גבואה במסמך, הוא נראה רלוונטי לתוך המסמך TF. לא לוקח בחשבון את החשיבות האנגלילת של מונח בכל הקורפוס. מילים נפוצות כמו "the" או "and" עשויות להיות להן ציוני TF גבוהים אך אין ממשמעות בהבחנה בין מסמכים.

תדירות מסמך הפוכה (IDF) מפחיתה את המשקל של מילים נפוצות במסמכים רבים תוך הגדלת המשקל של מילים נדירות. לדוגמה, אם המילה "blockchain" מופיעה 10 פעמים במסמך אך לעיתים רחוקות בآخرם, יהיה לה ציון TF-IDF גבוה, מה שמסמן את הרלוונטיות שלה לאותו מסמך.

TF-IDF עבור מסמך 1 הוא $(0.17 * 0.18) + (0.18 * 0.17) \approx 0.06$. מסמכים שבהם לשני המונחים יש ציוני TF-Gבואהים יותר (כמו מסמך 1) מדורגים גבואה יותר. שיטה זו מבטיחה שMONCHIM נפוצים במסמך אך נדירים בסך הכל מניעים רלוונטיות, משפרים את דיקח החיפוש.

הוא שימש לעיתים קרובות כגורם שקלול בחיפושים של אחזור מידע, קרית טקסט ומודול משתמשים. סקר שנערך בשנת 2015 הראה ש- 83% ממערכות המלצתה מבוססות הטקסט בספריות דיגיטליות השתמשו

ב-IDF-TF – וריאציות של סכימת השקלול IDF-TF שימושו לעיתים קרובות על ידי מנועי חיפוש ככלי מרכזי ביצירוף ודרוג הרגלונטיות של מסמר הנטו שאילתת משתמש.

את מונחים מסוימים בביטוי מחושבת על ידי סיכון TF-IDF עבור כל מונח שאליה; פונקציות דירוג מתוחכמתות רבות יותר הן וריאנטים של המודל הפשוט זהה. מנגנון חיפוש בעוצמה תעשייתית עובדים על ידי שימושם של מאות אלגוריתמים שונים לחישוב רלוונטיות, אבל נממש רק שניים: תדירות מונח ותדיות מסמר הנקה (TF-IDF) עם דירוג דמיון קוונטינום.

7. תיאור מצב קיימ

כיום קיימים מספר פתרונות לחיפוש מידע במחשב איש', אך רובם מוגבלים בהיקף או ביכולת שלהם:

1. **חיפוש מובנה של Windows** - מערכת הפעלה Windows מספקת יכולת חיפוש בסיסית המתמקדת בעיקר בשמות קבצים ותיקיות. החיפוש בתוכן קבצים יהיה איטי ומשמעותי ככל שמעמיקים בהיררכיית התיקיות או כאשר הנפח הכלול של הנתונים גדול.



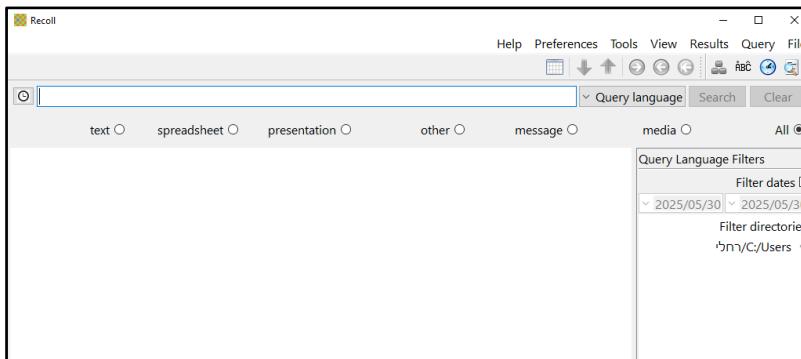
2. **Everything** - תוכנה המתמחה בחיפוש מהיר של שמות קבצים ותיקיות, אך אינה מבצעת חיפוש בתוכן הקבצים.

שם	תאריך סיום
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\array	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\date	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\error	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\function	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\number	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\promise	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\regexp	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\string	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\symbol	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\test\array	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\test\date	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\test\error	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\test\function	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\test\integer	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\test\iterator	20/06/2023 18:44
C:\Users\omni\.vscode\extensions\rnwebdev.vscode-live-server-5.7.0\node_modules\es5-ext\test\symbol	20/06/2023 18:44
היעד חישוקה ורתקן - מנגנון קביעת דרישות - מנגנון קביעת דרישות	22/04/2025 01:21
D:\Users\omni\Desktop\כברטן כברטן	22/04/2025 01:21

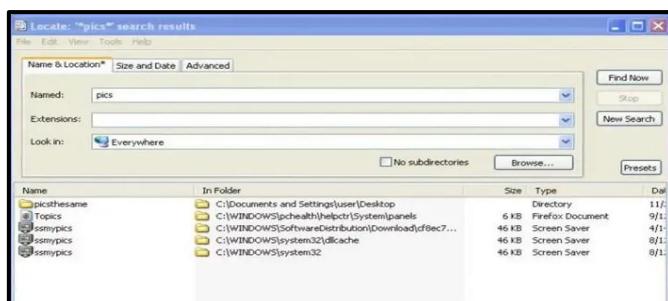
.3

3. Recoll - מנוע חיפוש קוד פתוח (Open Source) המאפשר חיפוש בתוכן מסמכים, אך הממשק

שלו פחות ידידותי למשתמש ודורש ידע טכני מסוים.



4. Locate32 - תוכנת חיפוש המתמקדת בשמות קבצים ומספקת חיפוש מהיר, אך לא יכולה חיפוש בתוכן.



המגבילות העיקריות של הפתרונות הקיימים כוללות:

- חוסר יכולת לחפש בתוכן קבצים בפורמטים מגוונים
- חוסר עילوت בדיאוג תוצאות לפי רלוונטיות

- ממשקי משתמש מורכבים או לא אינטואיטיביים
- שימוש לא יעיל במשאבי מערכת

8. ניתוח חליפות מערכתי

סיקת תוכן הקבצים בצורה יעילה

בעת בניית אינדקס ההפוך (inverted index), אנו עוברים על כל המסמכים ומבצעים תהליך שנקרא Tokenization – חלוקה של הטקסט למילים (terms). מכל מסמך אנו מקבלים רשימה של זוגות (term, docID), ואוספים את כל הזוגות מכלל המסמכים.

בשלב זה, נדרשת פעולה של מין כללי: תחילת לפי term בסדר לקסיקוגרפי, ואז מין משנה לפי docID. אם כל רשומה תופסת 12 בתים, והכמות הכוללת היא כ-100 מיליון רשומות (כמו במקרה של אוסף RCV1), אז מדובר בכ-1.12GB בזיכרון – גודל שעדיין אפשר עבודה בזיכרון (RAM) בלבד.

אבל כשעובדים עם אוספים גדולים בהרבה, כמו הארכיוון של ה-New York Times עם מסמכים מלמעלה מ-150 שנה, לא ניתן למין את כל הרשומות בזיכרון. נדרש פתרון אחר: מין חוץ-זיכרון, שבו חלקים מהנתונים נשמרים ומטופלים בדיסק.

מדוע לא ניתן למין שירות על הדיסק?

אם נשימוש באלגוריתם מין רגיל שדורש השוואות רבות, ובכל השוואה שתי פעולות seek בדיסק (שנמשכות כ-5 מילישניות כל אחת), נגיע לזמן ריצה לא מעשי. לדוגמה: 100 מיליון רשומות דורשות כ-8 מיליארד השוואות – מה שייקח כ-6.92 ימים. لكن נדרש פתרון חלופי.

BSBI – Blocked Sort-Based Indexing

תיאור האלגוריתם:

- מפצל את אוסף הזוגות (term, docID) לבlokים בגודל שמתאים לזיכרון.
- ממין כל בלוק בזיכרון ושומר אותו בדיסק.

- לאחר מכן מבצע מיזוג של כל הבלוקים הממויינים ל-index אחד מסודר.
- רק לאחר המיזוג נבניהם המילון וה-lists posting.

יתרונות:

- מיון מדויק ומלא:** המיון מבטיח שכלי term יופיע בראשימת הפניות אחת בלבד.
- שימוש חסכוני בזיכרון:** רק שני בלוקים נדרשים בזיכרון בכלל רגע (לצורך מיזוג).
- פשוט לשימוש:** מבוסס על עקרונות מוכרים של מיון חיצוני (external merge sort).

חסרונות:

- דורש שמרת כל הטרמינולוגיה המקורי:** יש לשמור את כל terms ואת המיפוי ל-ID עד לסיום תהליך המיזוג.
- שלב מיון מיותר לפעמים:** גם רשומות נדירות או ייחודיות נדרשות למיון, מה שעלול להזבז זמן ומשאבים.
- יצירת posting lists – רק בסוף – רק בסוף:** כל רשימת הפניות נוצרת רק לאחר סיום כל תהליך המיזוג.

SPIMI – Single-Pass In-Memory Indexing

תיאור האלגוריתם:

- בונה את index תוך כדי עיבוד המסמכים.
- כל בלוק מקבל מילון עצמאי, ובזמן הקריאה מתבצע עדכון של רשימות הפניות בהתאם.
- אין צורך במיון של הטרמינולוגיה – כל term נכנס לפי הסדר בו הגיע.

יתרונות:

- חסכון משמעותי בזיכרון:** לא נדרש מייפוי ID → term בזמן הריצה; כל term נשמר רק בבלוק שלו.
- יעילות גבוהה:** לא נדרש מיון ביןים של terms – פחות עומס חישובי.

- **בנייה ישירה של index:** מאפשר שימוש מהיר באינדקסים גם בשלבים מוקדמים.

חרוגונות:

- כתיבת **בלוקים גדולים יותר** - תהליך ארוך.
- **צריך למיין כל בלוק לפני כתיבת**ה - עוד זמן עיבוד.

שימוש במבנה נתונים לאינדקס

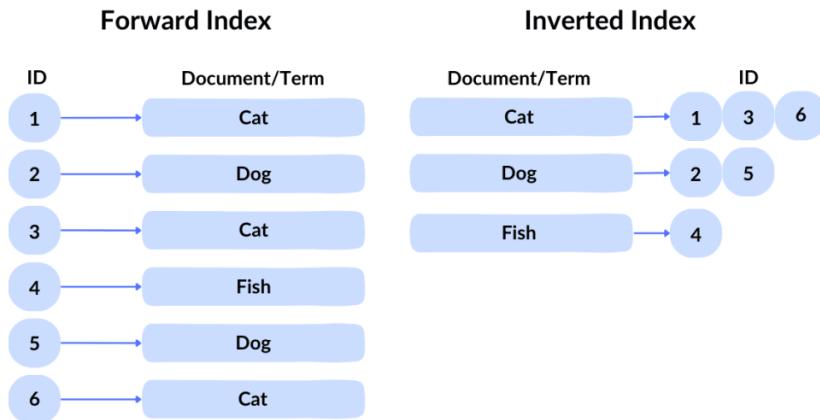
אינדקס הפוך (Inverted Index):

אינדקס הפוך הוא מבנה נתונים מרכזי במערכות חיפוש טקסט, המאפשר איתור מהיר של מסמכים המכילים מונחים ספציפיים. במבנה זה נשמר מיפוי בין כל מונח במאגר לבין רשימת המסמכים שבהם מופיע המונח. שם של האינדקס נובע מהעובדת שהמיפוי הוא "הפוך" ביחס למטרת נתונים רגיל, שבו נשמרים המונחים לפי מסמר; כאן, לעומת זאת, נשמרים המסמכים לפי מונח.

מבנה זה מאפשר ביצוע חיפושים יעיל ומהיר, שכן במקום לסרוק את כל המסמכים, ניתן לגשת ישירות אל המסמכים הרלוונטיים על פי המונח המבוקש.

רשימת פרסום (Posting List):

רשימת פרסום היא הרשימה המצורפת לכל מונח באינדקס הפוך, הכוללת את מזהה המסמכים שבהם מופיע המונח. רשימה זו יכולה לכלול גם מידע משלים, כגון תדירות הופעת המונח בכל מסמר, מיקומי המונח בתחום המסמר או נתונים נוספים, אשר משפרים את יכולות דירוג התוצאות וחווית החיפוש הכלולות.

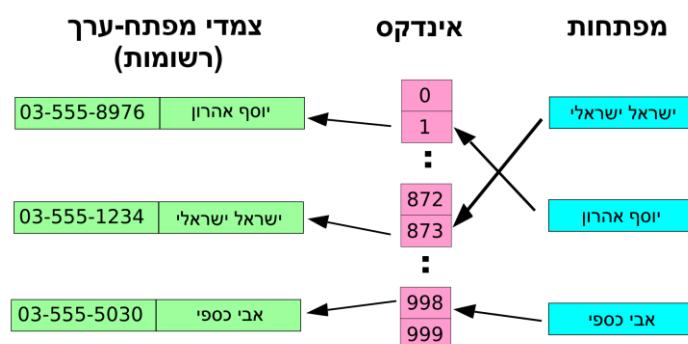


כדי לשפר את ביצועי מנוע החיפוש, יש לשלב את מבנה **the inverted index** עם מבני נתונים בעלי גישה מהירה ויעילה. בשלב זה, אurons סקירה מקצועית של מספר מבני נתונים רלוונטיים, תוך הדגשת יתרונותיהם וחסרונותיהם והשפעתם על מהירותה המערכת.

hash table

תיאור המבנה

טבלה הממפה מפתחות לערכים באמצעות פונקציית גיבוב (hash function).



AIR עובד

- כל מפתח מעובד לפוזיציה בטבלה באמצעות פונקציית גיבוב.
- פתרון התנשויות מתבצע באמצעות שרשור (chaining) או טיפול פתוח (open addressing).
- אפשר גישה ישירה למפתחות.

סיבוכיות זמן

- חיפוש ממוצע: $O(1)$
- הוספה ממוצעת: $O(1)$
- מחיקה ממוצעת: $O(1)$
- במקרה הגרוע: $O(n)$

יתרונות:

זמן החיפוש, ההוספה, והמחיקה, מהיר מאוד, $O(1)$ בממוצע

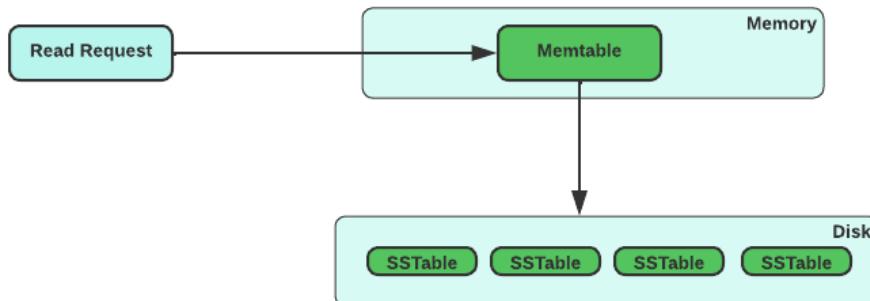
חסרונות:

- לא אפשר חיפושי טווח
- ככל שמספר המונחים גדול עלול להתרחש התנשויות (כמספר רב של מילים ממופות לאותו מקום בטבלה), אז זמן החיפוש הוא $O(n)$
- מחיבר שכל המונחים יהיו בזיכרון (יקר)

LSM Tree (Log-Structured Merge-Tree)

תיאור המבנה:

מבנה נתונים המתאים לכטיבה אינטנסיבית, המשלב מספר רמות של קבצים מסוודרים.



איך המבנהעובד:

- נתונים נכתבים תחילה בזיכרון (MemTable).
- כשהזיכרון מלא, הנתונים נכתבים לדיסק כקובץ מסוודר (SSTable).
- מבצעות פעולות מיזוג (merge) של קבצים כדי לשמר על סדר.
- החיפוש צריך התבצע

סיבוכיות זמן:

- כתביה: $O(\log n)$ - מהיר מאוד!
- חיפוש: $O(n \log n)$ × מספר SSTables - יכול להיות איטי
- אחד: $O(n \log n)$ - רק ברקע

יתרונות:

- כתיבה מהירה מאוד – הנתונים נשמרים תחילה בזכרון בלבד בלבד.
- iesel לעומס כתיבה גבוהים – אפשר עבודה עם כמותות גדולות של מידע משתנה.
- דחיסת נתונים טבעי – תהליך האיחוד (compaction) מוחק נתונים ישנים ומפנה מקום.

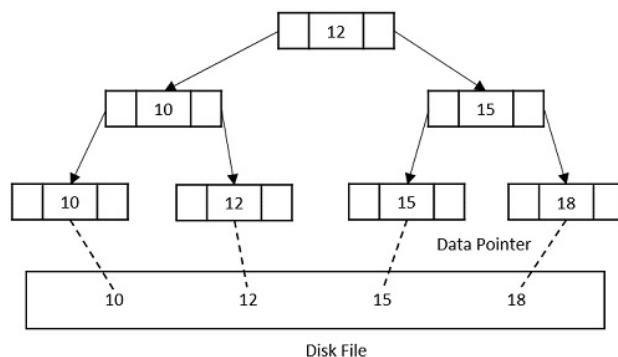
חיסון עיקרי:

- חיפוש איטי יחסית – אם המפתח לא בזכרון, יש לעבור בין קבצי SST בדיסק. ככל שיש יותר רמות, החיפוש נעשה מורכב ויטוי יותר.

B+ Tree

תיאור המבנה:

עץ איזון מסוג B שבו כל המפתחות נמצאים בעליים בלבד, והצמתים הפנימיים מכילים K מפתחות המשמשים ניווט בלבד. עליים מחוברים בראשימה מקוורת.



איך המבנה עובד:

- מבנה עץamazon בו כל הצמתים יכולים להכיל מספר מפתחות ומצביעים.
- חיפוש געשה באמצעות ניוט מהשורש אל העלים לפי מפתחות.
- הוספה ומחיקה מלאות באיזון חדש ושמירה על תנאי העץ.

סיבוכיות זמן:

- חיפוש: $O(\log n)$
- הוספה: $O(\log n)$
- מחיקה: $O(\log n)$

יתרונות:

- amazon תמיד נכון כל החיפושים הם בזמן קבוע
- יעיל לחיפושי טווח בזכות הקישור בין העלים.
- מוצאים לעבודה עם דיסק, כל צומת יכול להכיל הרבה מפתחות, מה שמקטין את מספר הגישות לדיסק.
- כל הערכים מרכזים רק בעלים מה שמשריע בשמירה על מבנה פשוט לחיפוש וסיריקה.

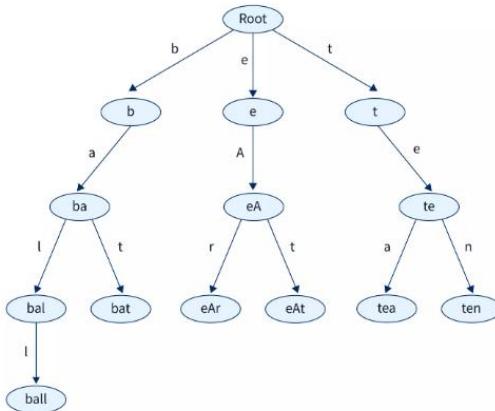
חסרונות

- מרכיבות יישום גבוהה - פיצול וחיבור צמתים במבנה בהוספה ובעדכון overhead
- לא אופטימלי לננתונים קטנים - יש overhead

Trie

תיאור המבנה:

עץ מילולי שמייצג אוסף מחרוזות, כאשר כל צומת מייצג תוו יחיד.



איך המבנה עובד:

- כל מחרוזת מיוצגת על ידי מסלול מהשורש לעלה.
- חיפוש, הוספה ומחיקה נעשים על ידי מעבר צומת-צומת לפי תווים.
- מנצל קידומות משותפות לחיסכון בזיכרון.

סיבוכיות זמן:

- חיפוש: $O(m)$ (m = אורך המחרוזת)
- הוספה: $O(m)$
- מחיקה: $O(m)$

יתרונות:

- חיפוש, הוספה ומחיקה מהירים לפי אורך המפתח בלבד.
- מתאים במיוחד לחיפוש מיללים עם קידומת משותפת (autocomplete).
- חוסך מקום באחסון מחרוזות עם קידומות משותפות.
- מתאים לחיפוש לפי תחילית או חיפוש טווח

חרוגונות:

- צרכי ציכרון גבוהה - הרבה צמותים עבור אלפבית גדול
- עץ לא AMAZON - יכול להיות עמוק עבור מילימטרים ארוכים

דירוג

TF-IDF (Term Frequency–Inverse Document Frequency)

TF-IDF הוא מודל דירוג סטטיסטי המשלב בין תדירות הופעת מונח במסמך (TF) לבין נדירותו בקורסוס המסמכים (IDF). מטרתו להעניק משקל גבוה למונחים שופיעים הרבה במסמך אחד אך נדירים במסמכים אחרים, ובכך להציג את הרלוונטיות היחסית של המסמך לשאלתה.

$$TF = \frac{\text{Number of times a word "X" appears in a Document}}{\text{Number of words present in a Document}}$$

$$IDF = \log \left(\frac{\text{Number of Documents present in a Corpus}}{\text{Number of Documents where word "X" has appeared}} \right)$$

$$TF \cdot IDF = TF * IDF$$

כאשר (p,t) היא תדירות המונח במסמך, N מספר המסמכים בקורסוס ו- (t) מספר המסמכים שבهم מופיע המונח.

יתרונות:

פשטות ויעילות חישובית, מתאים לטקסטים בעלי מבנה אחיד.

חסרונות:

אין מתחשב באורך המסמך, ועלול להעניק יתרון למסמכים ארוכים; לא מונע השפעה מוגזמת של מונחים שאינם מתחשב באורך המסמך, ועלול להעניק יתרון למסמכים ארוכים; לא מונע השפעה מוגזמת של מונחים חוזרים.

BM25 (Best Matching 25)

BM25 מחשב ציון רלוונטיות לכל מסמך ע"י שקלול תדירות המונח במסמך (TF), תדירות ההופעה של המונח בכל המסמכים (IDF), ואורך המסמך. השיטה נותנת משקל גבוה יותר למונחים שחוזרים בשאלתה, תוך איזון שמנע הטיה לטובת מסמכים

ארוכים מדי. יתרונות: מדויק יותר בישומי חיפוש, מתמודד טוב עם מסמכים באורך משתנה, נפוץ במנועי חיפוש.

$$BM25 = \sum_{t \in q} \log \left[\frac{N}{df(t)} \right] \cdot \frac{(k_1 + 1) \cdot tf(t, d)}{k_1 \cdot \left[(1 - b) + b \cdot \frac{dl(d)}{dl_{avg}} \right] + tf(t, d)}$$

- k_1, b – parameters
- $dl(d)$ – length of document d
- dl_{avg} – average document length

יתרונות:

- לוקח בחשבון את אורך המסמך, ומונע הטיה לטובות מסמכים ארוכים
- מדויק יחסית במנועי חיפוש טקסטואליים
- אינו דורש אימון או מודלים מורכבים – פשוט וחזק
- מתאים מאד לשאלות טקסטואליות רגילים עם מונחים בודדים

חרונות:

- דורש כוונון פרמטרים
- K (רמת רוחה של תדיות המונח): קבוע עד כמה תדיות המונח במסמך משפיעה על ציון הרלוונטיות – ערך גבוה מאפשר לתדיות להשפיע יותר.
- a (השפעת אורך המסמך): מażן בין אורך המסמך לתדיות המונח – ערך גבוה יותר מקטין את יתרון המסמכים הארוכים.

- עדין מבוסס על חיפוש מונחים ולא הבנה שפתית.

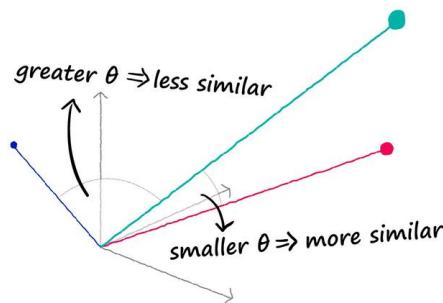
Cosine Similarity

Cosine Similarity (דמיון קוסיני) היא שיטה מתמטית למדידת הדמיון בין שני וקטורים במרחב רב-ממדי, והיא נפוצה במיוחד בתחום של אחור מידע, עיבוד שפה טבעיות (NLP) ולמדידת מכונה.

השיטה מודדת את הזווית בין שני וקטורים (מסמכים) ולא את המרחק ביניהם. ככל שהזווית קטנה יותר – ככלו, הווקטורים "מצבעים" אותו ציון – קר הדמיון גבוה יותר.

- ממירם כל מסחר ושאליתה לוקטור (כל תא בווקטור מייצג מילה אחת מהאוצר מילים).
- מחשבים את הזווית בין הווקטורים באמצעות המכפלת הפנימית.
- מנרמלים לפי גודל הווקטור.
- מקבלים ציון בין 0 ל-1 (או -1 ל-1, אם יש ערכים שליליים – נDIR בטקסטים רגילים).

$$\text{similarity}(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$$



- $B \cdot A =$ מכפלה פנימית של הווקטורים

- $|A| \cdot |B|$ אורך של כל וקטור (נורמה)

יתרונות:

- אין רגיש לאורך המסמרק – מtbody נירמול.
- פשוט ומהיר למימוש.
- טוב בהשוואה דמיון בין מסמכים עם מבנה מיילים דומה.

חסרונות:

- מתעלם לחילוטין מסדר המילים.
- לא מזהה ביטויים שלמים – רק חפיפות בין מילים.

שפות

Java

יתרונות:

- ניהול זיכרון אוטומטי – יש Garbage Collector שמתפלב בזיכרון עצמו, מה שמחיתת תקלות.
- חוצה פלטפורמות – רצה על כל מערכת הפעלה עם JVM.
- ספריות רבות – יש המונן כלים מוכנים (כמו Apache Tika, Lucene וכו').
- קלה ללמידה – התchapbir פשוט וברור יחסית, ומתחילה גם לפרויקטים גדולים.
- תחזקה קלה – קוד מודולרי, נוח לבדיקה, ותומך ב-OOP מלא.
- קהילת מפתחים ענקית – תיעוד מצוין, פורומים, מדריכים, תמיכה.

חרוגנות:

- דורשת התקנת MySQL – לא תמיד נוח בפריסה ללקוחות.
- פחות שליטה נמוכה על חומרה – לא מתאימה למערכות שדורשות ביצועים קרייטיים או גישה ישירה לזכרון.
- מהירות מעט נמוכה לעומת C++ – בעיקר בתהליכיים קרייטייםلزمן אמת.
- צריכת זיכרון גבוהה יחסית – בגלל הרצת המכונה הוירטואלית.

++C

יתרונות:

- ביצועים גבוהים מאוד – שפת מכונה שמתකמפלת לשירות, מתאימה למערכות זמן אמת.
- שליטה מלאה בזיכרון ובחומרה – ניתן לנוהל זיכרון, כתיבה ישירה, עבודה עם pointers.
- מתאימה למשחקים, מערכות הפעלה, רכיבי חומרה, מערכות מושבצות.
- תומכת בפראגמנט OOP וגם בתכנות פרוצדורי – גמישות מרבית.

חרוגנות:

- ניהול זיכרון ידני – דרוש אחיזות, יכול לגרום לדיליפות ובעיות קשות.
- תחברת מסובך – קשה יותר ללמידה ולתחזק, דרושת ניסיון.
- קידוד ארוך – הרבה קוד נדרש לפעולות פשוטות.
- לא חוצה פלטפורמות בקלות – כל מערכת דרושת הידור שונה.

#C

יתרונות:

- פיתוח מהיר ונוח – סביבת סבב Visual Studio מספקת כלים מתקדמים, כולל GUI מעולה.
- ניהול זיכרון אוטומטי – כמו Java.
- תמיכה חזקה ב-Windows – משלב היטב עם שירותים מיקרוסופט.
- תכונות מונחה עצמים חזק, תמיכה ב-LINQ, Entity Framework ועוד.
- עם .NET Core אפשר לעבוד גם על לינוקס ו-Mac.

חסרונות:

- בעבר היה מוגבל ל-Windows – היום NET Core פותר חלקית, אבל עדין פחות פרווס מ-Java.
- פחות ספירות קוד פתוח ביחס ל-Java – חלק מהכלים הם פורטים מספירות Java

9. תיאור החלופה הנבחרת

סרייקת תוכן הקבצים بصورة יעילה

SPIMI (Single-Pass In-Memory Indexing)

לאחר בחינת שתי שיטות לבניית אינדקס הפור – BSBI ו-SPIMI – בחרתי בשיטת SPIMI, בשילוב עם מיקבול (parallelism) בתהליך עיבוד התוכן ובנית הבלוקים. הבחירה ב-SPIMI נובעת מהתאמתה הטבעית למערכות גדולות ו互動ראקטיביות, הדורשות ביצועים מהירים וזיכרון יעיל.

נימוק לבחירה:

SPIMI מאפשר בניית אינדקס بصورة ישירה ולא צריך במילון כללי, דבר שמחית משמעותית את זמן העיבוד ומוריד את העומס על הזיכרון. כל term נכנס מיד לבלוק המתאים, וה-Posting Lists נבנות באופן מיידי. בשילוב עם מיקבול, ניתן לעבד מסמכים ולبنות בלוקים בזמן real-time – מה שתורם לקיצור זמן הבניה של האינדקס ולשיפור הסקלබיליות של המערכת. כך אין צורך להוכיח את כל הטרמינולוגיה או מיפוי ID→term – מה שמייעל את הזיכרון הראשי.

שימוש במבנה נתונים לאינדקס

אחר בחינה עמוקה של מספר חלופות למימוש אינדקס נתונים ייעיל בפרויקט, בחרתי בחלופה המבוססת על שילוב בין **Inverted Index** לבין **B+ Tree**, במימוש היברידי מותאם.

במהלך תהליך הבדיקה, שקلت שימוש ב-**Hash Table** כאופציה ראשונית, בשל יתרון המהירות התאורטית של חיפוש בזמן קבוע ($O(1)$). עם זאת, החלטתי לפסול אפשרות זו בשל מגבלות מהותיות: ראשית, קיומן של התנגשויות (collisions) עשוי לפגוע בביטחוניות ככל שכמות המונחים גדלה, ובמקרים מסוימים אף להוביל לפחות חיפוש של ($O(n)$). שנית, השיטה מחיבת טענה של כלל המונחים לזיכרון הראשי, דבר שאינו עיל כאשר מדובר בכמות גדולה של מונחים ובתמייה בשפות רבות.

לעומת זאת, מבנה נתונים מסווג **B+ Tree** נמצא מתאים יותר לפרויקט שלי, בשל מאפייניו הייחודיים:

- יכול לשמר את מרבית העץ בדיסק, ולהעלות רק צמתים רלוונטיים לזכרון בעת הצורך.
- גובה עץ נמוך יחסית (למשל, עץ בגובה 3 מאפשר אחסון של מאות אלפי מונחים), תורם לצמצום מספר הגישות לדיסק.
- סדר לוגי מלא של המונחים, דבר המKENה יכולת ביצוע של חיפושים טווחיים (Range queries).

למרות היתרון, בשלב ניסוי ראשוני שבו בצעתי חיפוש ישירות מדיסק (העלאת הצמתים הרלוונטיים בלבד), התקבלו זמני תגובה איטיים יחסית במיחוד עבור מונחים שכיחים, עקב מספר רב של פעולות O/I . לכן הוחלט לאמץ שיטה היברידית: מבנה העץ ($B+$) כולל נתען לזכרון הראשי, אך במקום שהערכים עצם (המונחים ורשימות המסמכים) יאוחסנו ישירות בעץ – כל ערך מכיל **offset** המצביע על מקום המונה בתוך קובץ ה-**Inverted Index** בדיסק.

גישה זו אפשרה:

- ביצוע חיפוש מהיר בעץ בזיכרון, עם זמן חיפוש של $O(\log k)$ בכל רמה, כאשר k הוא מספר הערכים בצומת (קבוע).

- ביצוע קרייה ממוקדת לקובץ האינדקס לפי מיקום מדויק, ובכך נחסכות גישות רבות לקריאה של מידע לא רלוונטי.
 - צמצום משמעותי בנפח הנתונים בזיכרון.
- בנוסף, העז גבנה מראש בשיטה של **Bottom-Up**, אשר מאפשרת זמן בנייה של $\Theta(n)$, תוך שימוש מקסימלי בצמתים ומונעת בזבוז בזכרון, בניגוד לשיטת **Top-Down** אשר דורשת זמן $\Theta(n \log n)$ ועשיה לייצר צמתים חלקיים.
- שילוב זה בין **Inverted Index** לדגם יעיל של **B+ Tree** – ובאופן מモטב תוך הפרדה בין מיקומי מונחים לזכרון ודיסק – מהוות את הפתרון הנבחר, תוך איזון בין מהירות, חיסכון במשאבים, ויכולת הרחבה עתידית לרבות תמיכה בעדכון, הוספה ומחיקה של נתונים.

דירוג

חלק מהפרויקט, היה לי חשוב לא רק לאחזר את המסמכים המכילים את המונחים שבשאלתה, אלא גם לדרג אותם בצורה חכמה – כך שהמסמכים הרלוונטיים ביותר יופיעו ראשונים. לשם כך חקרתי מספר שיטות דירוג נפוצות, ביניהן **Cosine Similarity**, **TF-IDF**, **BM25**, ו-**NLP**.

TF-IDF הייתה האלטרנטיבה הראשונה שבחנתי. היתרון המרכזי שלו הוא הפשטות והיעילות: היא מחשבת משקל לכל מונח על בסיס שכיחותו בתוך המסמך מול שכיחותו בקורסוי כולו. עם זאת, היא לא מתחשבת באורך המסמך, מה ש?url{allowUserScripting} יכול לגרום להעדפה של מסמכים ארוכים שבהם מונח חוזר באופן מקרי. לכן פסלתה אותה כבר בשלב מוקדם.

לאחר מכן בוחנתי את **Cosine Similarity** – שיטה מתמטית למדידת זווית בין וקטורים – שהיא נפוצה מאוד ב-NLP. אמנם היא פשוטה יחסית לשימוש, אך היא מתעלמת מסדר המילים ולא מתחשבת בהקשר – וזה לא התאים לאופי החיפוש שבניתי, שהתבסס על חיפוש מונחים מדויקים.

לבסוף הגיעתי ל-**BM25**, שהוא שיפור של TF-IDF. התרשםתי מהאופן שבו BM25 מאיימת בין שכיחות מונח, חשיבותו, ואורך המסמך. לדוגמה, אם מילה מסוימת מופיעה פעמיים במסמך קצר מאוד, אבל רק שלוש פעמיים במסמך ארוך מאוד – BM25 תעדיף את המסמך הקצר. בעניין זו תוצאה הגיונית ונכונה יותר, כי ככל הנראה המסמך הקצר מדובר בשירות על המונח, ואילו במסמך הארוך יתכן שהוא רק הזכר כבדר אגב.

לכן בחרתי למש את BM25 בתור מודל הדירוג של תוצאות החיפוש בפרויקט שלי. הוא לא רק מדויק יותר לשאלות מבוססות טקסט, אלא גם מעניק דירוג אינטואיטיבי שמאפשר לי לסדר את התוצאות לפי רלוונטיות – וזה בדיק מה שהייתי צריך.

שפה - java

- **חילוץ טקסט:** ל-Java יש ספריות מוכנות ועוצמתיות כמו Apache Tika ו-Lucene – פתרון מוכן וומוכן לעובדה. ב-C++ חסר, וב-C# הספריות חלשות יותר.

- **ניהול זיכרון:** Java כולל Garbage Collector אוטומטי, מה שמחיתת תקלות ודיליפות זיכרון – בנויגוד ל-C++ שדורשת ניהול ידני.
- **חויצה פלטפורמות:** Java רצה בכל מערכת עם JVM בלי צורך בהידור חדש, מה שנoston נידות גבואה. ב-C++ חיבים הידור חדש, וב-C# התמיכה חוות פלטפורמות עדין מתפתחת.
- **תחזוקה ונוחות:** Java מאפשרת פיתוח מהיר, מודולרי ונקי, עם קהילת תומכה ענקית וספריות קוד פתוח רבות.
- **ביצועים:** אולו C++ מהירה יותר בתיאוריה, אבל ביישומים של סריקה או אינדוקס – הביצועים של Java מצוינים בפועל.

10. אפיון המערכת שהוגדרה / מוצעת

דרישות המערכת

סביבת פיתוח:

- **חומרה:** מחשב עם מעבד i5 Core, 16GB RAM של 2.7GHz זיכרון SSD של 250GB וỔ אחסון
- **עמדת פיתוח:** Windows 10, 11
- **מערכת הפעלה:** Windows 10 ומעלה
- **דיסק קשיח:** SSD
- **חיבור לרשת:** לא נדרש לפחות בסיסית, נדרש רק להתקנה ראשונית
- **תוכנות:** JDK 11 ומעלה, כל פיתוח JavaFX
- **תוספים טכנולוגיים:** לא נדרשים תוספים מיוחדים

מודול המערכת

תchromים בהם המערכת עוסקת:

- חיפוש תיקיות וקבצים לפי שם
- חיפוש בתוכן טקסטואלי בקבצים מסוגים שונים (pdf, word, html, md, js) (txt וכו')
- דירוג תוצאות לפי רלוונטיות

תחומים בהם המערכת אינה עוסקת:

- חיפוש בקבצי מדיה (תמונות, וידאו, אודיו)
- ניתוח סמנטי מתקדם של תוכן
- הוספה עדכון או מחירה של הקבצים

אפיון פונקציונלי

1. **בחירה כוננים לאינדוקס** – המשמש בוחר אילו כוננים לסרוק.
2. **אפשרויות חיפוש**: לפי שם בלבד, לפי תוכן בלבד, הגדרת סוג קובץ.
3. **שלב אחזור ודירוג תוצאות** – החזרת תוצאות מסודרות לפי רלוונטיות.
4. **הצגת תוצאות חיפוש** – כולל תצוגה מקדימה של הקובץ.
5. **פתיחה הקובץ שירות** מהמערכת.
6. **פתיחה מקום הקובץ בסיר הקבצים**.

ביצועים עיקריים

המערכת מסוגלת להתמודד עם כמות גודלות של קבצים ללא ירידת נিיר בביטויים. זמן התגובה לחיפוש הוא לרוב מיידי, כאשר תוצאות מתקבלות תוך שניות ספורות, גם כאשר בסיס הנתונים כולל אלפי קבצים. תהליך האינדוקס הראשוני בניי כרך שיאכל לעבור ביעילות על ספריות וכוננים שלמים, לסרוק את שמות הקבצים ואת תוכנם (כאשר מדובר בקבצי טקסט), ולהכין את המערכת לפועלה שוטפת בהירות יחסית.

המערכת תומכת בפורמטים טקסטואליים נפוצים כגון .java, .xml, .txt, ועוד, ומתקדמת בקריאה מהירה ואפקטיבית של תכנים מתוך קבצים אלו. כל זאת נעשו תוך שימוש על צירכית משאים מבוקרת – הן בזיכרון (RAM) והן במעבד (CPU) – כרך שהשימוש במערכת אינו מכבי על המחשב גם כאשר מדובר בסביבת עבודה עמוסה. השילוב בין יעילות, מהירות, ועמידות הופך את המערכת לכל' אמין לשימוש יומיומי.

אלוצים

המערכת דורשת סביבת הריצה הכוללת Java ו-JavaFX מותקנים.

11. תיאור הארכיטקטורה

המערכת בנויה בארכיטקטורת שכבות בסביבת Java/JavaFX. הארכיטקטורה מורכבת משלוש שכבות עיקריות:

- **שכבה הממשק (UI Layer)** - מיושמת ב-JavaFX, אחראית על האינטראקציה עם המשתמש
- **שכבה השירות (Service Layer)** - מיושמת ב-Java, מבצעת את הלוגיקה האלגוריתמית.
- **שכבה הנתונים (Data Layer)** - מיושמת באמצעות מבני נתונים מותאמים ושמירה ישירה לדיסק, אחראית על אחסון האינדקס והמטא-דאטा

ארQUITקטורת רשות

המערכת הינה ישות Desktop עצמאית שאינו משתמש ברשות או בתקשורת מרוחקת.

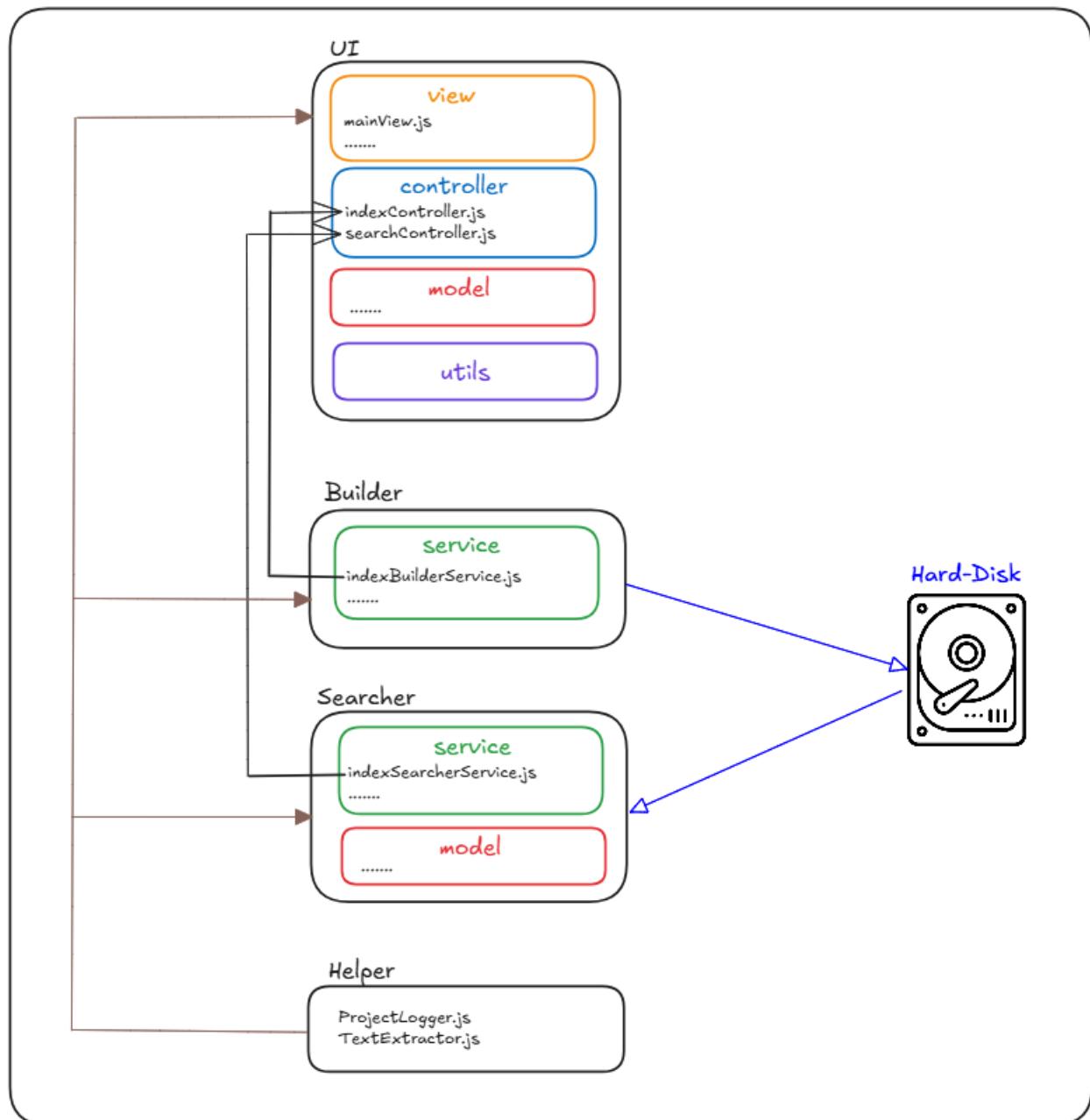
תיאור פרוטוקול תקשורת

לא רלוונטי, מכיוון שהמערכת אינה כוללת תקשורת רשות או אינטראקציות שרת-לקוח

שרות-לקוח

המערכת אינה מבוססת על ארQUITקטורת שרת-לקוח; היא פועלת באופן מקומי על מחשב המשתמש.

תרשים ארכיטקטורה



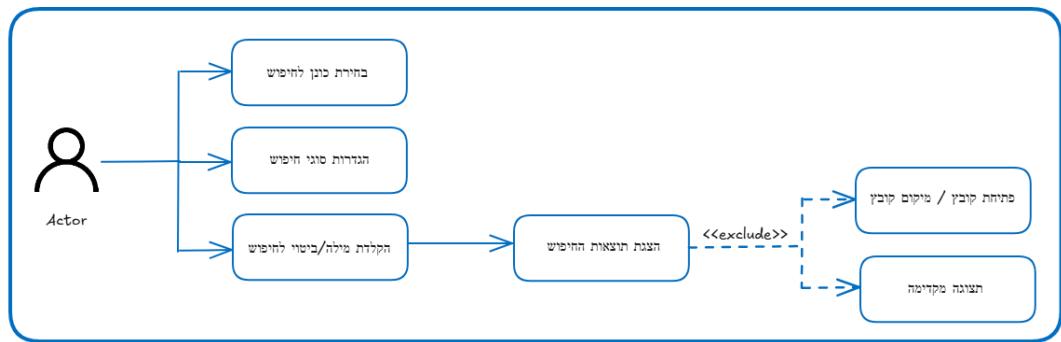
12. תיאור תהליכי אבטחת מידע במערכת

לאחר השלמת תהליכי בנית האינדקס, המערכת מיישמת הגנה על קבצי האינדקס. קבצי האינדקס מוגדרים כקבצים מסווגרים (hidden files) במערכת הפעלה ומוגדרים כקריאה בלבד (read-only) לכל המשתמשים.

הגדרות אלו מבטיחות שלמות נתונים ומנועות שינוי או השחתה של האינדקסים, תוך הבטחת יציבות פעולת מנוע החיפוש. מדובר באבטחה ברמת מערכת הקבצים המספקת הגנה עיליה מפני פגיעה לא מכוונת או גישה לא מורשית.

13. ניתוח ותרשימים UML / Use cases של המערכת המוצעת

Diagram class Design



שם מקורה שימוש	בחירה כוון לחיפוש
תיאור	המערכת בונה את האינדקס לחיפוש מהיר בהתאם לבחירת הכוון/נים.
שחknים	משתמש
תנאי מקדים	-
הזנק	הרצן לחיפוש קובץ / תיקייה
מסלול עיקרי	1. המשמש בוחר כוון/ים 2. המערכת בונה את האינדקס
תנאי סיום	המערכת סיימה לבנות את האינדקס
תדרות	פעם ביום

שם מקורה שימוש	הגדרות סוגי חיפוש
תיאור	המערכת מגדרה את סוגי החיפושים בהתאם להגדרת המשתמש
שחknים	משתמש
תנאי מקדים	-
הזנק	הרצן לחיפוש קובץ / תיקייה
מסלול עיקרי	המשמש בוחר את סוגי החיפוש
תנאי סיום	-
תדרות	בכל חיפוש

שם מקורה שימוש	הקלחת מילה או ביטוי לחיפוש
תיאור	המשמש מקליד את המילה או הביטוי לחיפוש המערכת מחפש את הקובץ/ים המכילים את המילה או הביטוי
שחknים	משתמש
תנאי מקדים	הגדרת כוון לחיפוש
הזנק	הרצן לחיפוש קובץ / תיקייה
מסלול עיקרי	1. המשמש מקליד מילה או ביטוי לחיפוש 2. המערכת מחפש את הקובץ/ים בכוון/ים המתאים בהתאם להגדרות החיפוש
תנאי סיום	המערכת מצאה את הקבצים המתאימים
תדרות	עשרות פעמים ביום

שם מקרה שימוש	הציג תוצאות החיפוש
תיאור	המערכת מציגה את כל הקבצים שמצאה בהתאם להגדרות החיפוש.
שחקנים	המערכת
תנאי מיקדים	הקלדת מילה או ביטוי לחיפוש
הזנק	המערכת מצאה את הקבצים המתאים
מסלול עיקרי	המערכת מציגה את הקבצים שמצאה
תנאי סיום	המערכת מציגה את הקבצים שמצאה
תדירות	כשיש תוצאות מהחיפוש

שם מקרה שימוש	פתחת קובץ / מיקום קובץ
תיאור	המערכת פותחת את הקובץ / מיקומו
שחקנים	משתמש
תנאי מיקדים	הציג תוצאות החיפוש
הזנק	המשתמש רוצה לפתח את הקובץ / מיקומו
מסלול עיקרי	1. המשתמש לוחץ לחצן ימני על שורת הקובץ המוצגת בתוצאות החיפוש ובוחר לפתיחת הקובץ / מיקומו 2. המערכת פותחת את הקובץ / מיקומו שבחר המשתמש
תנאי סיום	המערכת פתחה את הקובץ / מיקומו
תדירות	כשהמשתמש רוצה לפתח את הקובץ / מיקומו

שם מקרה שימוש	תצוגה מקדימה
תיאור	המערכת מציגה את הקובץ בתצוגה מקדימה
שחקנים	המשתמש
תנאי מיקדים	הציג תוצאות החיפוש
הזנק	המשתמש רוצה לראות את הקובץ בתצוגה מקדימה
מסלול עיקרי	1. המשתמש לוחץ לחיצה שמאלית על שורת הקובץ המוצגת בתוצאות החיפוש 2. המערכת מציגה את הקובץ בתצוגה מקדימה
תנאי סיום	המערכת מציגה את הקובץ בתצוגה מקדימה
תדירות	כשהמשתמש רוצה לראות את הקובץ בתצוגה מקדימה

הקשרים בין היחידות השונות:

הקשרים ברצף בניית האינדקס (צד שמאל):

Scan Files & Folders → Tokenizer .1

- העברת תוכן הקבצים הגלמי לעיבוד

Tokenizer → Normalizer .2

- העברת מילים/טוקנים מפוצלים לנורמל

Normalizer → Indexer .3

- העברת מילים מנורמלות (אותיות קטנות, ללא סימני פיסוק)

Indexer → Disk .4

- שמירת האינדקסים הבינהריים לדיסק

קשרים ברצף החיפוש (צד ימין):

User Query → Query Processor .5

- קבלת שאלת חיפוש מהמשתמש

Query Processor → Retriever .6

- העברת שאלתה מעובדת לאחזר

Retriever ← Disk .7

- קריאת נתונים מהאינדקסים שנשמרו בדיסק

Retriever → Scorer + Ranker .8

- העברת תוצאות גולמיות לדירוג וניקוד

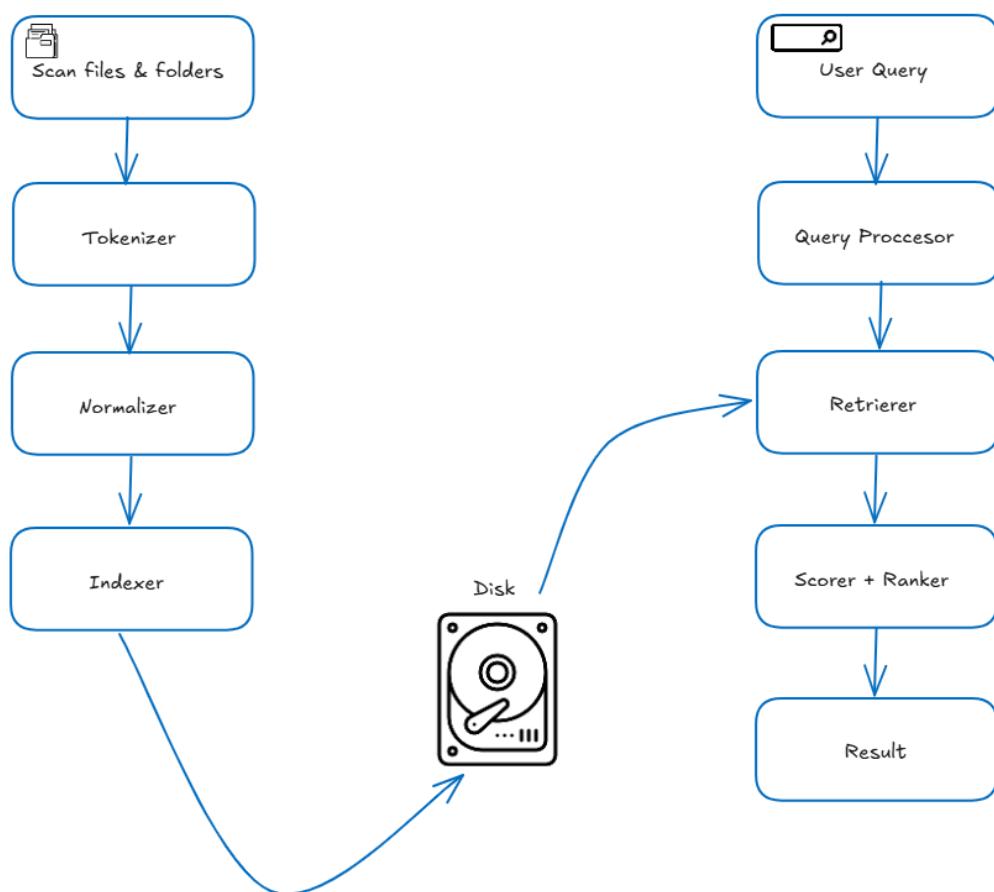
Scorer + Ranker → Result .9

- העברת תוצאות מדורגות למשתמש

הקשר המרכזי:

- מהוות את נקודת החיבור בין שני התהליכיים - CAN נשמרם האינדקסים משלב הבניה ונקראים בשלב החיפוש.

עכ. מודולים:



מבנה נתונים בהם השתמשתי

במימוש המערכת נעשה שימוש בכמה מבני נתונים עיקריים, בהתאם לשלי העבודה השונים:

1. בניית האינדקס(SIMI).

- **Hash Table** לצורך יצירת בלוקים של אינדקס מקומי במהלך קבצי הטקסט, תוך שימוש בזיכרון בלבד.
- **Priority Queue** שולבה בשלב מיזוג הבלוקים השונים לאינדקס מאוחד.

2. העלאת האינדקס לשלב החיפוש

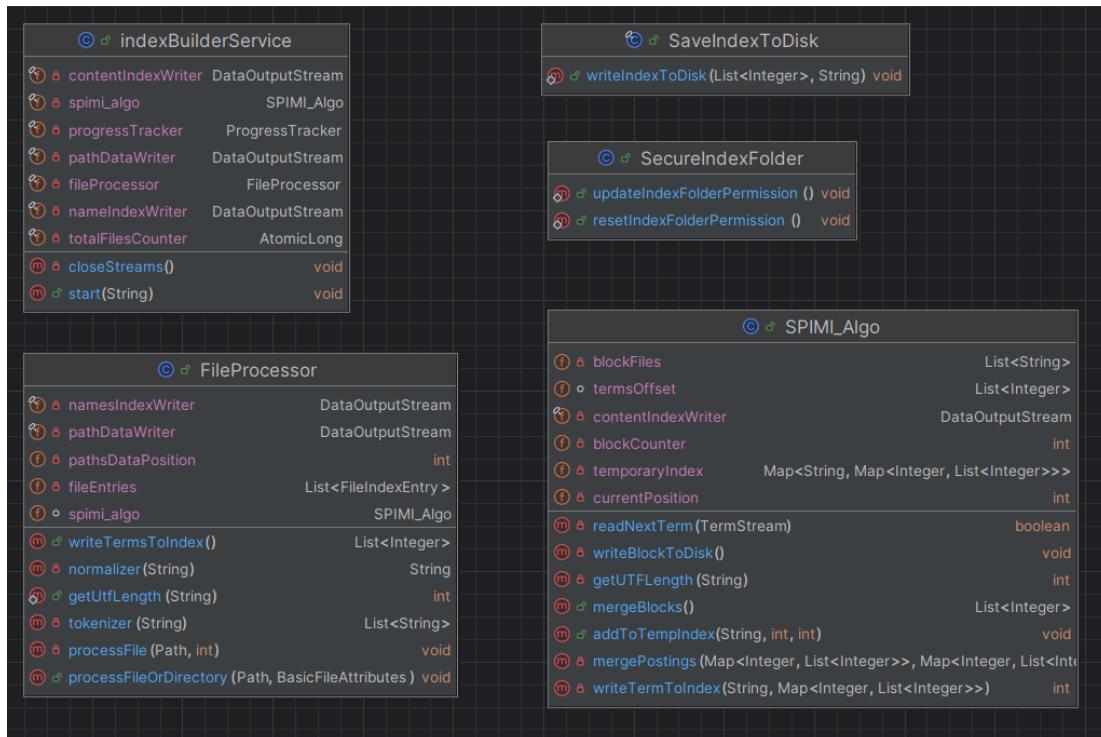
- בB+Tree נשמרים המיקומים המדויקים (offset) של כל מילה בקובץ האינדקס. כך, במהלך החיפוש ניתן לגשת ישירות למיקום הספציפי של מילה בדיסק, מבלי לקרוא מידע נוסף. בזכות מבנה העץ, שבו כל הצמתים הפנימיים מכילים טווחים רחבים של ערכים וגובהו נמוך (בממוצע 2–3 רמות), מספר הגישות לדיסק מצומצם למינימום.

3. חישוב יעילות האלגוריתם

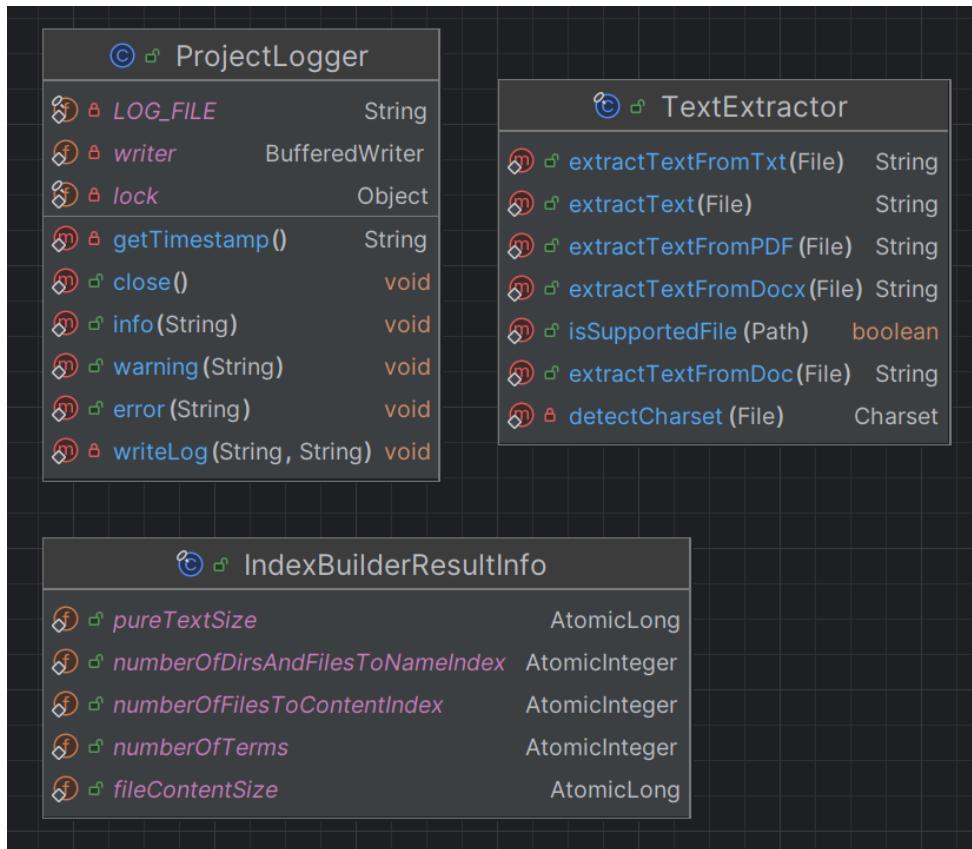
- זמן בניית האינדקס הוא – $O(n)$ נובע מסריקה ליניארית של הקבצים
- זמן בניית ה- O Tree – $O(n)$ נובע מגישת הבניה.kup-bottom
- זמן חיפוש של מילה הוא – $O(\log n)$ בזכות מבנה העץ המאורגן, שבו כל חיפוש עובר רק במספר מצומצם של צמתים.

תרשים מחלקות

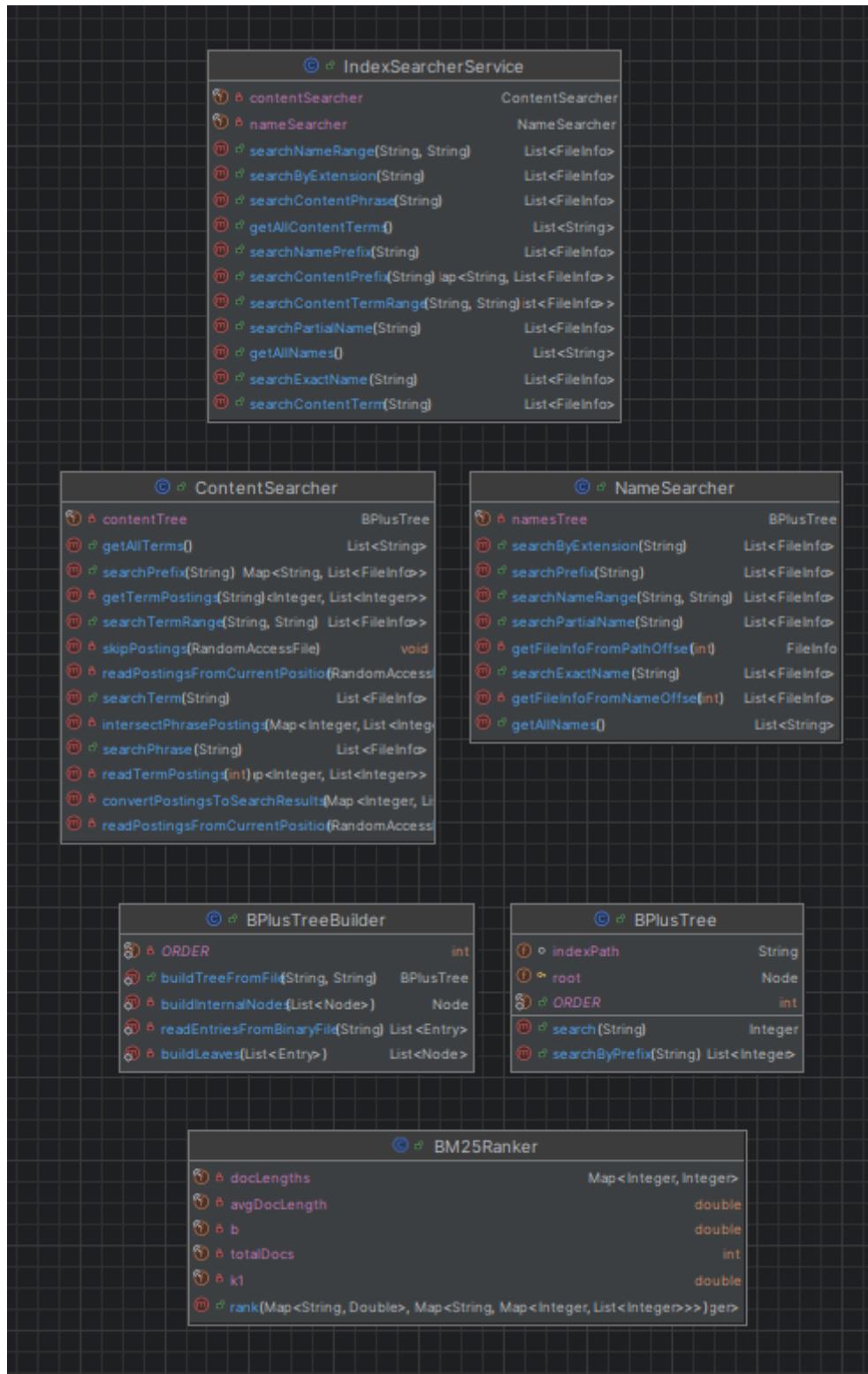
Builder



Helper

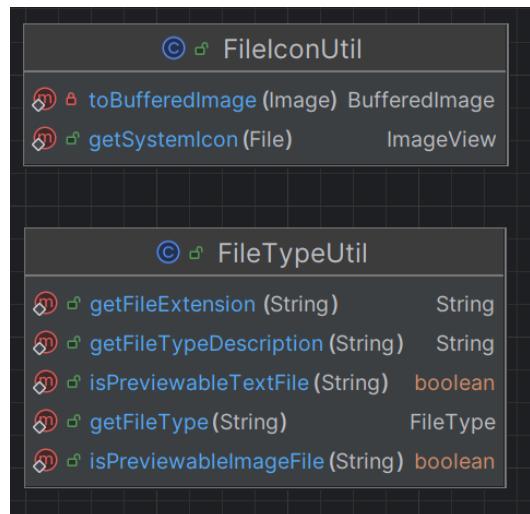


Searcher

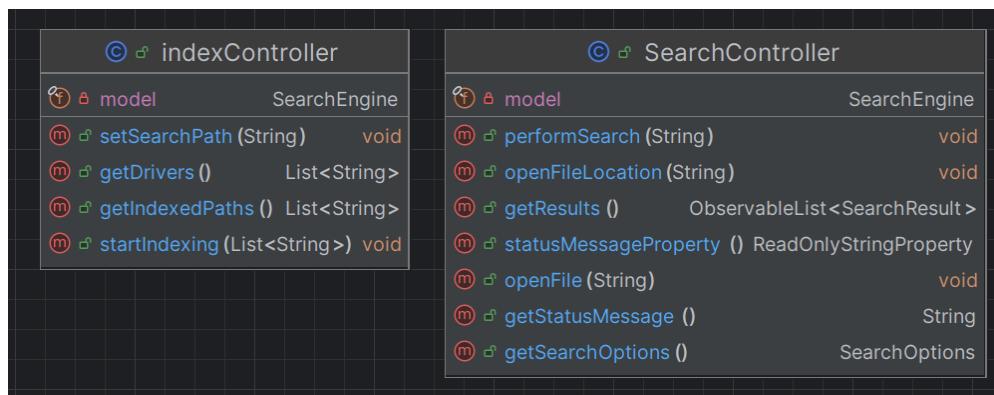


UI

Utils ○



Controller ○



model ○

SearchResult	SearchEngine	SearchOptions
<ul style="list-style-type: none"> ⓘ name String ⓘ path String ⓘ size String ⓘ modified String ⓘ created String ⓘ getModified() String ⓘ getType() String ⓘ getCreated() String ⓘ getSize() String ⓘ getName() String ⓘ getPath() String ⓘ getFullPath() String 	<ul style="list-style-type: none"> ⓘ instance SearchEngine ⓘ isSearching boolean ⓘ resultsCount ReadOnlyIntegerWrapper ⓘ indexSearcher IndexSearcherService ⓘ statusMessage ReadOnlyStringWrapper ⓘ searchOptions SearchOptions ⓘ searchContentTerm(String) st<SearchResult> ⓘ extractPath(String) String ⓘ getInstance() SearchEngine ⓘ getResultsCount() int ⓘ searchNamePrefix(String) List<SearchResult> ⓘ getSearchResults() ObservableList<SearchResult> ⓘ isSearching() boolean ⓘ convertFileInfoToSearchResult(List<FileInfo>) ⓘ performSearch(String) Task<Void> ⓘ updateResults(ObservableList<SearchResult>) ⓘ formatFileSize(long) String ⓘ initializeSearchEngine() void ⓘ resultsCountProperty() idOnlyIntegerProperty ⓘ searchContentPhrase(String) <SearchResult> ⓘ getStatusMessage() String ⓘ reindexDirectory(String) void ⓘ extractFileName(String) String ⓘ getSearchOptions() SearchOptions ⓘ statusMessageProperty() idOnlyStringProperty 	<ul style="list-style-type: none"> ⓘ searchPath StringProperty ⓘ maxResults IntegerProperty ⓘ searchTerm StringProperty ⓘ searchInContent BooleanProperty ⓘ fileType ObjectProperty <FileType> ⓘ exactMatch BooleanProperty ⓘ caseSensitive BooleanProperty ⓘ setMaxResults(int) void ⓘ setFileType(FileType) void ⓘ isSearchInContent() boolean ⓘ searchInContentProperty() BooleanProperty ⓘ resetToDefaults() void ⓘ fileTypeProperty() ObjectProperty <FileType> ⓘ searchPathProperty() StringProperty ⓘ getSearchPath() String ⓘ setCaseSensitive(boolean) void ⓘ getSearchTerm() String ⓘ searchTermProperty() StringProperty ⓘ getMaxResults() int ⓘ maxResultsProperty() IntegerProperty ⓘ exactMatchProperty() BooleanProperty ⓘ setSearchPath(String) void ⓘ setSearchTerm(String) void ⓘ caseSensitiveProperty() BooleanProperty ⓘ getFileType() FileType ⓘ isCaseSensitive() boolean ⓘ setSearchInContent(boolean) void ⓘ isExactMatch() boolean ⓘ setExactMatch(boolean) void

View ○

<th> MainView </th> <th> SearchPane </th>	MainView	SearchPane
<ul style="list-style-type: none"> ⓘ mainLayout BorderPane ⓘ searchPane SearchPane ⓘ contentSplitPane SplitPane ⓘ statusLabel Label ⓘ resultsPane ResultsPane ⓘ mainSplitPane SplitPane ⓘ searchController SearchController ⓘ previewPane PreviewPane ⓘ filterPane FilterPane ⓘ setupEventHandlers() void ⓘ getMainLayout() BorderPane 	<ul style="list-style-type: none"> ⓘ searchController SearchController ⓘ searchField TextField ⓘ searchPane VBox ⓘ statusLabel Label ⓘ performSearch() void ⓘ bindStatusLabel() void ⓘ openSettingsWindow() void ⓘ getPane() VBox ⓘ createSettingsButton() Button 	
<th> ResultsPane </th> <th> FilterPane </th>	ResultsPane	FilterPane
<ul style="list-style-type: none"> ⓘ resultsPane BorderPane ⓘ searchController SearchController ⓘ resultsTable TableView<SearchResult> ⓘ countLabel Label ⓘ createModifiedColumn() .ult, String> ⓘ getResultsTable() rw<SearchResult> ⓘ createCreatedColumn() .ult, String> ⓘ getPane() BorderPane ⓘ setupStatusBar() void ⓘ setupResultsTable() void ⓘ createNameIconColumn() .chResult> ⓘ setupRowClickHandler() void ⓘ createSizeColumn() hResult, String> ⓘ bindToModel() void ⓘ createPathColumn() hResult, String> 	<ul style="list-style-type: none"> ⓘ filterPane VBox ⓘ searchController SearchController ⓘ fileTypeFilter ComboBox<String> ⓘ fileNameRadio RadioButton ⓘ contentRadio RadioButton ⓘ exactMatchCheck CheckBox ⓘ bindToModel(SearchOptions) void ⓘ resetOptions() void ⓘ getPane() VBox 	
	<th> PreviewPane </th>	PreviewPane
	<ul style="list-style-type: none"> ⓘ DEFAULT_HEIGHT double ⓘ fileContentPreview TextArea ⓘ spinner ProgressIndicator ⓘ MAX_PREVIEW_SIZE int ⓘ previewPane VBox ⓘ readTextFileContent(String) String ⓘ generatePreviewMessage(SearchResult) ⓘ getPane() VBox ⓘ showFilePreview(SearchResult) void 	

14. רכיבי ממשך

- **שדה חיפוש** - קלט טקסטואלי למילות החיפוש.
- **כפטוריו פעולה** -
 חפש – מפעיל את מנגנון החיפוש.

הגדרות ניתוב – פותח חלון קופץ לבחירת כוננים שיש לандקו.

- **טבלת תוצאות** - תצוגת רשימת הקבצים שנמצאו, כל תוצאה מכילה: שם הקובץ, נתיב, גודל, תאריך שינוי, תאריך ייצירה.
- **הגדרות חיפוש** - בחירת אפשרויות לחיפוש (סוג חיפוש, סוג קבצים, חיפוש מדויק).
- **תצוגה מקדימה** - "תצוגה מקדימה של תוכן הקובץ".
- **שורת סטטוס** - מציגה את מספר התוצאות שחרזו.

16. תיאור התוכנה

שפנות תוכנות:

- Java

סביבת פיתוח:

- IntelliJ IDEA

טכנולוגיות ותשתיות נוספות:

- Maven (לניהול תלויות)

- JavaFX (UI)

17. תיאור מסכים

מסמך החיפוש הראשי

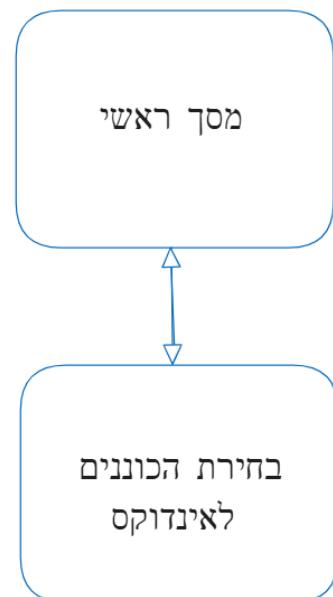
המסמך הראשי של המערכת מחולק ל-4 פאנלים עיקריים:

- פאנל הגדרות חיפוש
- פאנל חיפוש
- פאנל התוצאות
- פאנל תצוגה מקדימה

חלון הגדרת נתיב האינדקס ובנייהתו

חלון קופץ הנפתח בעת לחיצה על כפתור הגדרות, המאפשר למשתמש להגדיר את נתיב תיקיית האינדקסים במערכת הקבצים.

18. תרשימים מסכימים



19. פירוט מסכימים

The image consists of two vertically stacked screenshots of the DeepSearch application interface.

Top Screenshot:

- Search Bar:** Contains fields for "חיפוש" (Search), "שם הקובץ" (File Name), and "טיטו טוצאות" (Title Results). A tooltip "התחל חיפוש..." (Start search...) is shown near the search button.
- Table Headers:** "נתיב" (Path), "גודל" (Size), "תאריך שינוי" (Change Date), and "תאריך יצירה" (Create Date).
- Table Content:** A message "No content in table" (No content in table) is displayed.
- Right Panel:** Titled "חיפוש קבצים ותיקיות" (Search for files and history).
 - Buttons: "הקלד מילוט מפתח לחיפוש..." (Type a keyphrase to search...), "הקלד קבץ ותיקיה...", and "הקלד קבץ ותיקיה..." (Search for file and history).
 - Radio Buttons: "שווים קבצים ותיקיות" (Match files and history) (selected), "תוכן קבצים" (Content of files), and "תוכן ותיקיות" (Content of history).
 - Dropdown: "סוג קובץ:" (File type:).
 - "כל" (All) (selected)
 - "המפתח" (The keyphrase)
 - Checkboxes: "אפשרות חיפוש:" (Search option:).
 - "התאמאה מדויקת" (Exact match) (unchecked)
 - Buttons: "שמירת הגדירות" (Save settings).
- Bottom Panel:** Displays the search results: "מציג פקידה של תוכן הקובץ" (Showing a command for file content) and a large empty text area.

Bottom Screenshot:

- Search Bar:** Similar to the top one, with "שם הקובץ" (File Name) and "נתיב" (Path) fields.
- Modal Window:** Titled "בחירה בין נספחים לאינדקס" (Select between attachments for indexing).
 - Text: "חולון לבחירת נספחים לשrique" (Please choose attachments for indexing).
 - List: "נספחים זמינים:" (Available attachments:).
 - checkboxes: CA, CB, CC, CD, CE, CF.
 - Buttons: "ביטול" (Cancel) and "הנתק ואנדים" (Disconnect and index).
- Right Panel:** Same as the top one, titled "חיפוש קבצים ותיקיות" (Search for files and history).
- Bottom Panel:** Displays the search results: "מציג פקידה של תוכן הקובץ" (Showing a command for file content) and a large empty text area.

לחיצה על תוצאה
החפשית להציג
תוצאת מקבץ של
הקובץ

הגדלת חיפוש
חיפוש: אבטחת סוכר

חיפוש לפי תוקן

תוצאות חיפוש קבצים ותיקיות

תוצוגה סקירה של תוכן הקובץ

שם הקובץ	נתיב	גודל	תאריך שינוי	תאריך יצירה
products.csv	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	337 KB	19-05-2025 13:26:49	19-05-2025 18:49:16
productDetails.csv	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\DB\...	726 KB	19-05-2025 13:26:51	19-05-2025 13:26:51
productDetails.csv	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	726 KB	12-05-2025 14:20:49	12-05-2025 12:43:29
Conversions.html	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	12 KB	08-01-2024 15:59:38	22-04-2025 01:37:10
products.csv	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	337 KB	12-05-2025 17:29:29	12-05-2025 12:43:29
Conversions.html	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	12 KB	20-01-2025 15:04:31	22-04-2025 01:37:02
pdf.(2)	D:\Users\racheli\Downloads	7.7 MB	09-04-2025 00:26:33	22-04-2025 01:17:49
מחכום לפסח תשפ"א docx	D:\Users\racheli\Documents	81 KB	09-10-2023 23:50:35	22-04-2025 02:39:12
פסח- תענית (2)	D:\Users\racheli\Downloads	1002 KB	09-04-2025 00:26:35	22-04-2025 01:17:52
מחכום לפסח תשפ"א (2) pdf	D:\Users\racheli\Downloads	7.7 MB	09-04-2025 00:36:58	22-04-2025 01:17:49
productDetails.csv	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	726 KB	19-05-2025 13:26:51	19-05-2025 18:49:16
products.csv	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\DB\...	337 KB	19-05-2025 13:26:49	19-05-2025 13:26:49
pdf.(1) pdf	D:\Users\racheli\Documents	127 KB	09-01-2024 11:30:19	22-04-2025 02:39:12
יעורות שוקול מושלחות	D:\Users\racheli\Downloads	5 KB	00-04-2025 00:27:00	22-04-2025 01:17:42

תוצאות חיפוש: 14 תוצאות ב-C:\

לחיצה ימינה על תוצאה
החפשית לבחירת אפשרות:
פתרונות לקובץ
ופתיחת מיקום הקובץ

הגדלת חיפוש
חיפוש: אבטחת סוכר

תוצאות חיפוש קבצים ותיקיות

תוצוגה סקירה של תוכן הקובץ

שם הקובץ	נתיב	גודל	תאריך שינוי	תאריך יצירה
ContentIndexReader.java	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	10 KB	14-05-2025 00:57:31	15-05-2025 15:11
files_metadata.txt	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	49.9 MB	25-04-2025 06:24:18	25-04-2025 05:00
random_access.txt	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	124.2 MB	03-04-2025 23:29:38	22-04-2025 01:31
buffered_output.txt	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	10.1 MB	03-04-2025 22:59:33	22-04-2025 01:22
files_metadata.txt	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	49.9 MB	25-04-2025 06:24:18	29-04-2025 02:41
buffered_writer.txt	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	125.2 MB	03-04-2025 23:30:36	22-04-2025 01:31
block_0.txt	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	132 KB	11-02-2025 00:44:28	22-04-2025 01:31
inverted_index.txt	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	121.2 MB	25-04-2025 06:24:18	25-04-2025 04:00
docx.העדר	D:\Users\racheli\Documents\test	12 KB	04-02-2025 02:31:01	22-04-2025 02:31
log.txt	D:\Users\racheli\Documents\DeepSearch	29.7 MB	29-05-2025 15:43:17	23-05-2025 03:31
index.txt	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	9 KB	09-02-2025 04:44:50	22-04-2025 01:31
metadata.txt	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	7.1 MB	30-03-2025 20:04:23	22-04-2025 01:40
inverted_index.txt	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	121.2 MB	25-04-2025 06:24:18	27-04-2025 13:31
inverted_index.txt	D:\Users\racheli\Desktop\SmartCart - Arduino\Server\#ba...	121.2 MB	25-04-2025 06:24:19	20-04-2025 00:27:00

תוצאות חיפוש: 28 תוצאות ב-C:\

תוצוגה סקירה של תוכן הקובץ

```
:package testFilesContent
:import helper.IndexConstant
:import java.util.ArrayList
:import java.util.HashMap
```

20. קוד התוכנית

הקוד מבצע סריקה רקורסיבית של תיקיות וקבצים מהספרייה הראשית, תוך שימוש בটווח של כל רכיב בנפרד, מבלי לטען את כל התוכן לזכור בביטחון, ומאפשר מעבר חלק גם במקרה של שגיאות גישה.

```

Path rootPath = Path.of(root);
EnumSet<FileVisitOption> options = EnumSet.noneOf(FileVisitOption.class);
Files.walkFileTree(rootPath, options, Integer.MAX_VALUE, new SimpleFileVisitor<>() {
    @Override new *
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
        safeProcess(file, attrs);
        return FileVisitResult.CONTINUE;
    }

    @Override new *
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws IOException {
        safeProcess(dir, attrs);
        return FileVisitResult.CONTINUE;
    }

    @Override new *
    public FileVisitResult visitFileFailed(Path file, IOException exc) {
        ProjectLogger.warning("Failed to access: " + file);
        return FileVisitResult.CONTINUE;
    }

    private void safeProcess(Path path, BasicFileAttributes attrs) throws IOException { 2 usages new *
        try {
            fileProcessor.processFileOrDirectory(path, attrs);
        } catch (NullPointerException e) {
            ProjectLogger.error("Null path component at: " + path);
        }
    }
});
}

```

עיבוד תוכן קובץ:

```

private List<String> tokenizer(String text) { 1 usage new *
    // פולינקם לפי רווחים
    String[] rawTokens = text.split( regex: "\s+");
    return Arrays.asList(rawTokens);
}

private String normalize(String token) { 1 usage new *
    // הסרת כל תווים לא רצויים (פיסוק, תווים מיוחדים)
    return token.toLowerCase()
        .replaceAll( regex: "^\p{Punct}+", replacement: "") // הסרת מתחילה
        .replaceAll( regex: "\p{Punct}+$", replacement: ""); // הסרת מסיימת
}

```

```

private void processFile(Path path, int docId) throws IOException { 1 usage new *
    File file = path.toFile();
    String text = TextExtractor.extractText(file);

    // שלב 1: טוקניזציה
    List<String> tokens = tokenizer(text);

    // שלב 2: נירמול
    List<String> normalizedTokens = new ArrayList<>();
    for (String token : tokens) {
        String normalized = normalize(token);
        if (!normalized.isEmpty()) {
            normalizedTokens.add(normalized);
        }
    }

    // שלב 3: הוספה לאינדקס
    for (int i = 0; i < normalizedTokens.size(); i++) {
        String term = normalizedTokens.get(i);
        spimi_algo.addToTempIndex(term, docId, i);
    }
}

```

בנייה האינדקס ושמירתו ב-Disk:

```
List<Integer> nameTermsOffset = name_spimi_algo.mergeBlocks();
SaveIndexToDisk.writeIndexToDisk(nameTermsOffset, IndexConstants.B_PLUS_TREE_NAMES_INDEX);

List<Integer> contentTermsOffset = content_spimi_algo.mergeBlocks();
SaveIndexToDisk.writeIndexToDisk(contentTermsOffset, IndexConstants.B_PLUS_TREE_CONTENT_INDEX);
```

דירוג התוצאות:

```
public List<Integer> rank(Map<String, Double> queryTerms, Map<String, Map<Integer, List<Integer>>> termPostings) {
    Map<Integer, Double> docScores = new HashMap<>();

    for (String term : queryTerms.keySet()) {
        double idf = queryTerms.get(term);
        Map<Integer, List<Integer>> postings = termPostings.get(term);
        if (postings == null) continue;

        for (Map.Entry<Integer, List<Integer>> entry : postings.entrySet()) {
            int docId = entry.getKey();
            int freq = entry.getValue().size();
            int docLen = docLengths.getOrDefault(docId, defaultValue: 0);

            double score = idf * (freq * (k1 + 1)) /
                (freq + k1 * (1 - b + b * docLen / avgDocLength));

            docScores.merge(docId, score, Double::sum);
        }
    }

    // מיזוג תוצאות לפי ציון בסדר WORD
    return docScores.entrySet().stream() Stream<Entry<...>>
        .sorted(Map.Entry.<Integer, Double>comparingByValue().reversed())
        .map(Map.Entry::getKey) Stream<Integer>
        .collect(Collectors.toList());
}
```

21. תיאור מסד הנתונים

במוקם להסתמך על מסד נתונים מסורתי, בחרתי לישם ארכיטקטורה של מנוע חיפוש עצמאי שמנהله את הנתונים ברמה נמוכה.

הפתרון שפיתחתי מימוש ניהול נתונים בניו נתונים מותאמים אישית ואלגוריתמי אינדקס מתקדמים. גישה זו מעניקה לי שליטה מלאה על אופן ארגון הנתונים, מאפשרת אופטימיזציה לביצועים ספציפיים של חיפוש טקסט, ומבללת תלות בתוכנות חיצונית.

מבנה קבצי הנתונים שיוצרים

- **paths.dat** - קובץ נתיבים רציף המכיל את כל נתיבי הקבצים והתיקיות במערכת. כל נתיב נשמר בפורמט 8-UTF ומהזהה המסמך הוא ה-*offset* של הנתיב בקובץ זה.
- **content_index.idx** ו- **name_index.idx** - אינדקס היפוך (inverted index) המאורגן בסגמנטים משתנים. כל סגמנט מכיל מילה אחת עם רשימה כל הממכילים אותה, כולל תדיורות המילה במסמך ורשימת מקומות המדויקים בטקסט.
- **tree_content_index.idx** ו- **tree_name_index.idx** - קבצי עצי B+ המכילים רשימות offset לתחילת סגמנט באינדקסים הראשיים, המאפשרים חיפוש *binary search* בסיבוכיות ($\log n$) וטעינה עצ לזריכן בסיבוכיות ($\log n$).

העיצוב בסגמנטים משתנים מאפשר ניצול אופטימלי של דיסק, קריאות רציפות ייעילות, וგמישות בגודל הנתונים לפני התוכן בפועל.

22. מדריך למשתמש

שלום לך משתמש יקר!

הפעלה ראשונית

- התקן את מנוע החיפוש במחשב שלך והפעיל אותו
- ודא שיש לך הרשות לקרוא לתיקיות שברצונך לחפש בהן
- בהפעלה הראשונה, תצטרך להגיד את נתיב תיקיית האינדקסים

שימוש במערכת

בכניסתך לתוכנה זו יפתח מולך המסך הראשי, שבו מופיעות האפשרויות הבאות:

- ✓ **פאנל חיפוש** - הציג את המילה או הביטוי שברצונך לחפש
- ✓ **פאנל הגדרות חיפוש** - בחר האם לחפש בשמות קבצים ותיקיות או בתוכן
- ✓ **פאנל התוצאות** - תציג את רשימת הקבצים שנמצאו
- ✓ **פאנל תצוגה מקדימה** - יציג פרטים נוספים על הקובץ הנבחר

ביצוע חיפוש

1. בחר את סוג החיפוש מפאנל הגדרות:
 - שמות קבצים ותיקיות
 - תוכן קבצים (טקסט)
2. **הקלד** את מילת החיפוש בשדה החיפוש
3. **לחץ על כפתור "חפש"**
4. בחר קובץ מרשימה התוצאות לצפייה בפרטים נוספים

הגדרות מתקדמות

לחיצה על כפתור ההגדרות תפתח חלון להגדרת נתיב תיקיות לאינדקס. המערכת מספקת חיפוש מהיר ויעיל המאפשר מציאת קבצים תוך שניות ספורות!

23. בדיקות והערכתה

זמן בנית האינדקס

זמן סריקת הקבצים ועיבודם – חסכון של חצי מהזמן.
לפני ניצול המקבילות:

```
"C:\Program Files\Java\jdk-23\bin\java.exe" ...
Permissions reset successfully.
מערכת אינדקס וZIPosh קבצים
=====
\:הכנס נתיב לסריקה
\:מתחילה סריקת
זמן סריקת הקבצים ועיבודם: 30.44 דקות
```

אחרי ניצול המקבילות:

```
"C:\Program Files\Java\jdk-23\bin\java.exe" ...
Permissions reset successfully.
מערכת אינדקס וZIPosh קבצים
=====
\:הכנס נתיב לסריקה
\:מתחילה סריקת
זמן סריקת הקבצים ועיבודם: 15.36 דקות
```

זמן בנית האינדקס - הזיכרון RAM לא עולה על 3 GB

	מגמות שימוש ב...	צריכת חשמל	0%	רשות	2%	94% זיכרון	51% CPU	מצב	שם
	נמוכה מודע	0 Mbps	0 MB לשנה...	2,487.1 MB	0.3%				IntelliJ IDEA Ultimate Edition
	נמוכה מודע	0 Mbps	0 MB לשנה...	4.7 MB	0%				OpenJDK Platform binary
	נמוכה מודע	0 Mbps	0 MB לשנה...	1.9 MB	0%				OpenJDK Platform binary

חיפוש בתוכן קבצי טקסט שונים:

חיפוש בתוכן קובץ json :

The screenshot shows the DeepSearch application interface. On the left, there is a file tree with several files listed. In the center, a search results table displays two entries:

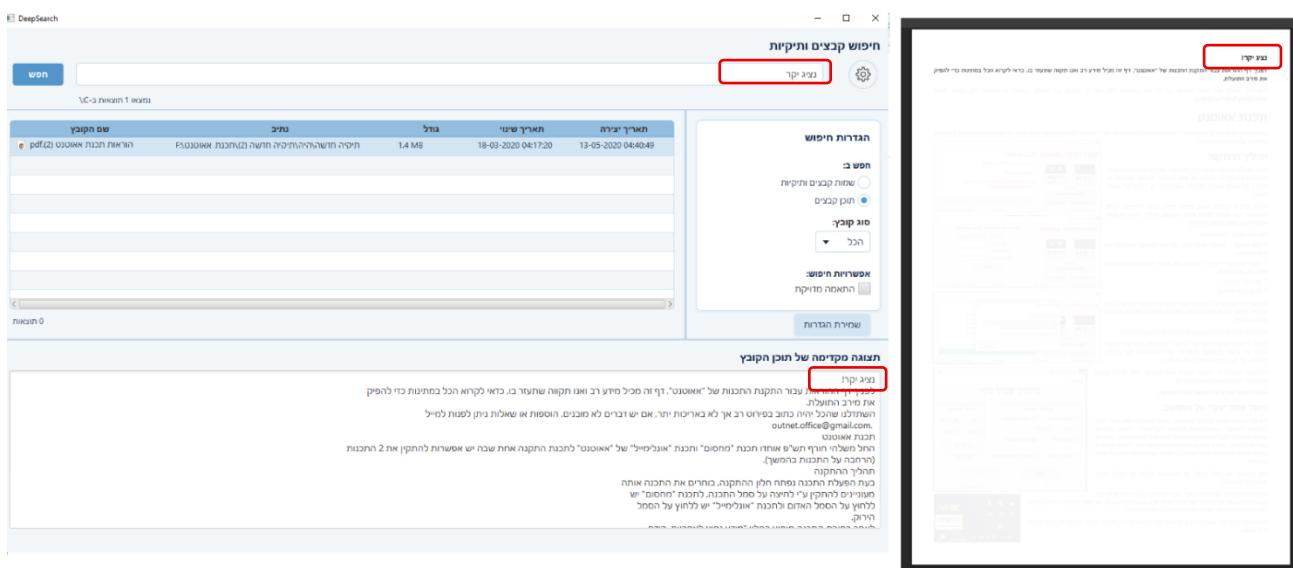
שם הקובץ	נתיב	גודל	תאריך שני	תאריך יירוח
Driver.xml	F:\After Effect\products	1 KB	02-10-2018 01:17:38	18-07-2022 20:47:30
Application.json	F:\After Effect\products\AEFT	46 KB	02-10-2018 01:17:42	18-07-2022 20:47:30

To the right of the table, there is a search configuration panel titled "חיפוש קבצים ותיקיות". It includes fields for "חיפוש:" (Search for:), "סוג קובץ:" (File type:), and checkboxes for "אפשרויות חיפוש" (Search options) and "הדרות חזרה" (Return history). Below the table, a code editor window shows the contents of the Application.json file:

```
{
  "Name": "After Effects CC",
  "SAPCode": "AEFT",
  "ProductVersion": "16.0.0.235",
  "CodexVersion": "16.0",
  "BaseVersion": "16.0",
  "Platform": "win64",
  "SupportedLanguages": {
    "Language": [
      {
        "value": "",
        "locale": "en_US"
      },
      {
        "value": "",
        "locale": "en_GB"
      }
    ]
  }
}
```

The code editor also displays the file's encoding (UTF-8), line count (100%), and cursor position (Ln 1, Col 1).

חיפוש בתוכן קובץ PDF עם תמונות:



תוצאות מהירות תוך פחת מושנית

```
zman biizut hahefot : 7 milishnitiot
mas tozotot: 2

mazefet b'andek : 'the'
zman biizut hahefot : 3 milishnitiot
mas tozotot: 6

mazefet b'andek : 'בציב יקר'
zman biizut hahefot : 4 milishnitiot
mas tozotot: 1

mazefet b'andek : 'בציב יקר'
zman biizut hahefot : 5 milishnitiot
mas tozotot: 1

mazefet b'andek : 'בציב יקר'
zman biizut hahefot : 5 milishnitiot
mas tozotot: 1

mazefet b'andek : 'בציב יקר'
zman biizut hahefot : 5 milishnitiot
mas tozotot: 1
```

זמן חיפוש למילה נפוצה (3 שניות):

```
'ds' : מוחפש באנדקס
מילים נפוצות 3180 דהן ביצוע חיפוש
טס תוצאות: 6283
```

ביצועי קריאה מהדיםק:

קריאה ישירה - DataInputStream

מול

קריאה עם באפר BufferedInputStream

```

public static BPlusTree.BUILTREEFROMFILE(String filePath, String indexPath) throws IOException {
    ...
}

// מוחבץ חביבנאי offsets דרישת
private static List<BPlusTree.Entry> readEntriesFromBinaryFile(String filePath) throws IOException {
    List<BPlusTree.Entry> entries = new ArrayList<>();

    try (DataInputStream dis = new DataInputStream(new FileInputStream(filePath))) {
        while (dis.available() >= 4) {
            int offset = dis.readInt();
            entries.add(new BPlusTree.Entry(offset));
        }
    }
    return entries;
}

```

Run search.UI.Main

"C:\Program Files\Java\jdk-23\bin\java.exe" ...

Time taken to read entries: 25.7938885 seconds

```

// מוחבץ חביבנאי offsets דרישת
private static List<BPlusTree.Entry> readEntriesFromBinaryFile(String filePath) throws IOException {
    List<BPlusTree.Entry> entries = new ArrayList<>();

    try (DataInputStream dis = new DataInputStream(
        new BufferedInputStream(new FileInputStream(filePath)))) {
        while (true) {
            try {
                int offset = dis.readInt();
                entries.add(new BPlusTree.Entry(offset));
            } catch (EOFException e) {
                break;
            }
        }
    }
    return entries;
}

```

Run search.UI.Main

"C:\Program Files\Java\jdk-23\bin\java.exe" ...

Time taken to read entries: 0.2881306 seconds

24. ניתוחיעילות

המערכת תוכננה לאופטימיזציה של שני פרמטרי היעילות העיקריים: סיבוכיות זמן מינימלית ונצול זיכרוןיעיל.

סיבוכיות מקום

בזיכרון הראשי:

במקום לטעון את כל המונחים עצם לזיכרון, המערכת טוענת רק **מצביעים** (offsets) למקום המונחים בקובץ. למשל, במקום לשמר את התוכן האמתי של המילה "מחשב" בזיכרון, נשמר רק מספר של **4** בתים שמצוין על המיקום בו המילה "מחשב" נמצאת בקובץ על הדיסק.

דוגמאות חיסכון בזיכרון:

1. **מילה קצרה** - "כן": 4 בתים → 4 בתים (offset) - חיסכון 0%
2. **מילה בינונית** - "מחשב": 10 בתים → 4 בתים (offset) - חיסכון 60%
3. **מילה ארוכה** - "מיקרופrocessor": 26 בתים → 4 בתים (offset) - חיסכון 85%

חיסכון כולל בזיכרון: בהתבסס על אורך ממוצע של 5.5 תווים למילה בעברית (11 בתים ב-UTF), במקום לטעון מיליון מונחים של 11 בתים כל אחד לזיכרון, נטענים רק מיקומיהם של 4 בתים כל אחד - חיסכון ממוצע של **64%** בתפישת זיכרון.

בדיסק קשייה:

אורך המונחים משתנים - במקום לתפוס מספר בתים קבוע לכל מונח, קבצי האינדקס מכילים סגמנטים משתנים לכל מונח, זהה מתאפשר בגלל שמירת offset (מיקום של המונח בקובץ). כך שבמקום לתפוס לכל מונח 100 בתים (הגבלת אורך המילה לאינדקס), אני תופסת רק את מספר הבתים הנדרשים לפי אורך של כל מונח.

דוגמה: המונח "א" יתפוס 1 בית, המונח "אלגוריתם" יתפוס 16 בתים, במקום שכל מונח יתפוס 100 בתים קבועים.

האינדקסים נכתבים בפורמט בינארי במקום טקסטואלי, כך גודל קבצי האינדקס קטן משמעותית **דוגמיה:** בפורמט טקסטואלי המונח "123456789" יתפוס 9 בתים, לעומת פורמט בינארי שיתפוס 4 בתים (int).

סיבוכיות זמן

תהליך החיפוש מתבצע על ידי סריקה בעץ B^+ של ה- offset . הנווט בעץ כולל: קריאה של המילה מהדיסק, השוואה, ואז ניוט לצומת הבא. זמן קריאה מהדיסק 50–100 (SSD) מיקרו-שניות

קבולות הערכים בעץ:

עץ B^+ בגובה 3 עם $k=200$ יכול להכיל:

- גובה 1: 200 מונחים
- גובה 2: 40,000 מונחים
- גובה 3: **8,000,000 מונחים**

שיטת החיפוש המקורי (ללא אופטימיזציה):

סיבוכיות: $O(h \times k)$ כאשר:

- h = גובה העץ $= \log_k(n)$
- k = מספר הערכים בכל צומת
- n = מספר המונחים במערכת

דוגמה: עץ בגובה 3 עם $k=200$ ערכים לצומת:

- **רמה 1:** עד 200 השוואות בצומת השורש
- **רמה 2:** עד 200 השוואות בצומת הביניים
- **רמה 3:** עד 200 השוואות בעלה
- **סה"כ:** עד 600 השוואות

האופטימיזציה שבייצעה - חיפוש ביןארי בתוך כל צומת:

במקום לסרוק ליניארית את הערכים בכל צומת, ישמש חיפוש ביןארי:

הסיבוכיות המשופרת: $(n \log k) \times \log k = O(n \log k \log k) = O(n \log^2 k)$

דוגמה משופרת: אותו עץ בגובה 3 עם $k=200$:

- **רמה 1:** $8 \approx \log_2(200)$ השוואות
- **רמה 2:** $8 \approx \log_2(200)$ השוואות
- **רמה 3:** $8 \approx \log_2(200)$ השוואות
- **סה"כ:** כ-24 השוואות (במקום 600!)

מערכת עם 8 מיליון מונחים נדרשת רק **C-24** השוואות וזמן חיפוש של **פחות ממילישניה** (בהתחשב בזמן קריאה מ-SSD של 50–100 מיקרו-שניות) לחיפוש מילה בודדת!

מכיוון ש- a הוא **קבוע** (מספר קבוע של ערכים בצומת), הוא לא שימושי לחישוב סיבוכיות הזמן הכללי. הסיבוכיות הסופית היא **(n log n)** - לוגריתמית במספר המונחים במערכת.

25. מסקנות

כשניגשתי לתכנן את מנוע החיפוש, הבנתי שהמשימה מורכבת ומאתגרת, הדורשת הבנה عمיקה של מבני נתונים ואלגוריתמים מתקדמים. ראשית השקעת זמן רב בחקירת השיטות הקיימות ובחירת הארכיטקטורה הנכונה - שילוב של `index inverted` ועצי B+ שיספק את הייעילות המקסימלית.

התכוון הקפדי של בניית הנתונים והחלטת השמירה בפורמט ביןארי הוכיחו את עצמן כקריטיות להצלחת הפROYיקט. הבחירה בעצי B+ עם חיפוש ביןארי בתוך כל צומת הבਆ לשיפור דרמטי בביטויים - מ-600 השוואות ל-24 השוואות בלבד למערכת של 8 מיליון מונחים.

במהלך הפיתוח התמודדתי עם אתגרים טכניים מורכבים, במיוחד בתחום ניהול הזיכרון ואופטימיזציהGIS הדיסק. אתגר נוסף היה הקושי בדיבוג קבצי האינדקס עקב הפורמט הבינארי, שדרש פיתוח כלים נוספים לבדיקה ואיומות של תקינות הנתונים. הפתרון בשיטת `offsets` במקום המונחים עצם הביא לחיסכון של 64% בזכרון, תוך שמירה על זמן חיפוש מהירים.

הפרויקט לימד אותי על חשיבות האיזון בין סיבוכיות זמן לסייעות מקום (בזיכרון RAM ובdisk קשיich), ועל כך שעיצוב נכון של מבני נתונים יכול להביא לשיפור שימושי בביטויים מלבני לפחות בפונקציונליות.

26. פיתוחים עתידיים

המערכת הקיימת מספקת בסיס חזק לפיתוחים נוספים שיוכלו להרחיב את יכולות החיפוש:

- **הוספת NLP לחיפוש סמנטי** - אפשרות לחיפוש לפי משמעות ולא רק התאמה מדויקת של מילים
- **חיפוש בתוך מילה** - יכולת לחפש חלק מילים, סימנות וקידומות (substring search)
- **חיפוש מטא-נתונים** - אפשרות לחפש לפי תאריך יצירה, גודל קובץ
- **חיפוש מטושטש (Fuzzy Search)** - יכולת למצוא מילים דומות או עם שגיאות הקלדה
- **דחיפות האינדקסים** - יישום אלגוריתמי דחיפה מתקדים לקטינה נוספת בגודל קבצי האינדקס

27. ביבליוגרפיה

IR

<https://studyglance.in/dbms/display.php?tno=51&topic=File-Organization-and-Indexing-in-DBMS>

<https://nlp.stanford.edu/IR-book/html/htmledition/single-pass-in-memory-indexing-1.html>

<https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf>

<https://www.dc.fi.udc.es/~roi/publications/rblanco-phd.pdf>

<https://nlp.stanford.edu/IR-book/>

<https://www.geeksforgeeks.org/inverted-index/>

מבנה נתונים ואינדקסים

<https://8thlight.com/insights/an-introduction-to-database-indexing>

<https://www.atlassian.com/data/databases/how-does-indexing-work>

<https://dev.to/aws-builders/understanding-database-indexes-and-their-data-structures-hashes-ss-tables-lsm-trees-and-b-trees-2dk5>

<https://www.sqlpipe.com/blog/b-tree-vs-hash-index-and-when-to-use-them>
<https://planetscale.com/blog/btrees-and-database-indexes>
<https://builtin.com/data-science/b-tree-index>

דירוג

<https://www.johnbryce.co.il/lobby-magazine/articles/bert-update-what-does-he-say/>
<https://www.pinecone.io/learn/semantic-search/>
<https://kmwllc.com/index.php/2020/03/20/understanding-tf-idf-and-bm-25>
https://en.wikipedia.org/wiki/Okapi_BM25

רכיבי המחשב

https://www.geeksforgeeks.org/introduction-to-parallel-computing/?utm_source=chatgpt.com
https://www.geeksforgeeks.org/central-processing-unit-cpu/?utm_source=chatgpt.com
<https://www.geeksforgeeks.org/memory-management-in-operating-system/>
https://www.ibm.com/think/topics/parallel-computing?utm_source=chatgpt.com
https://handbook.eng.kempnerinstitute.harvard.edu/s5_ai_scaling_and_engineering/scalability/introduction_to_parallel_computing.html?utm_source=chatgpt.com