



deep Search

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 עיקריים בפתרון בעיה בטכנולוגיות הנדסה מתקדמות
15	גבולות הפרויקט
15	פיתוחים עתידיים
15	תיאור טכנולוגיות הנדסה
15	מסד נתונים
16	2. מבוא
16	הרקע לפרויקט
16	תהליך המחקר
18	סקירת ספרות
18	אתגרים מרכזיים
18	הבעיה איתה התמודדתי
19	הסיבות לבחירת הנושא
19	על איזה צורך הפרויקט עונה
19	הצגת 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-data
22	6. רקע תאורטי
30	7. תיאור מצב קיים
32	8. ניתוח חליפות מערכתי
32	סריקת תוכן הקבצים בצורה יעילה
34	שימוש במבנה נתונים לאינדקס

41	דירוג
44	שפות
46	9. תיאור החלופה הנבחרת
46	סריקת תוכן הקבצים בצורה יעילה
47	שימוש במבנה נתונים לאינדקס
49	דירוג
49	שפה - java
50	10. אפיון המערכת שהוגדרה / מוצעת
50	דרישות המערכת
50	מודול המערכת
51	אפיון פונקציונלי
51	ביצועים עיקריים
52	אילוצים
52	11. תיאור הארכיטקטורה
52	ארכיטקטורת רשת
52	תיאור פרוטוקולי תקשורת
52	שרת-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	עץ מודולים:
58	מבני נתונים בהם השתמשתי
60	תרשים מחלקות
60	Builder
60	Helper
61	Searcher
62	UI
64	14. רכיבי ממשק
64	16. תיאור התוכנה

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

1. הצעת פרויקט

תיאור פרויקט

פרויקט זה מתמקד בפיתוח מנוע חיפוש יעיל, המיועד לאיתור 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, תאריך יצירה, תאריך עדכון, סוג הקובץ, וגודל.

חיפוש מבוסס תוכן – מנוע החיפוש יאפשר חיפוש מתקדם בתוך התוכן של הקבצים עצמם, כולל איתור מילות מפתח, ביטויים או מחרוזות טקסט מסוימות הקיימות בתוך קבצי טקסט, ומסמכים דיגיטליים.

הגדרת בעיה אלגוריתמית

הבעיה: חיפוש יעיל של מילה/ביטוי בתכני הקבצים ובמאפייניהם ע"י בניית אינדקס יעיל ודירוג התוצאות המוחזרות.

n: מספר הקבצים במערכת.

m: מספר השאלות (queries) של משתמש ביום.

כאשר משתמש מחפש מילה ספציפית בכל הקבצים זמן החיפוש יהיה $O(n)$ לכל שאלתה, כלומר $O(nm)$ לסך השאלות ביום.

גישה כזו תדרוש המון זמן ותגרום למערכת להיות איטית.

בניית אינדקס יעיל (Indexing)

בניית אינדקס היא תהליך של יצירת מבנה נתונים שמאפשר חיפוש מהיר וממוקד במאגר המידע. האינדקס מכיל מיפוי בין המילים של המסמכים לבין מיקומם במאגר.

הבעיה האלגוריתמית היא כיצד לבנות אינדקס בצורה כזו שתספק גישה מהירה ובזמן סביר למילים במסמכים, במיוחד כאשר גודל המסמכים והנתונים עשוי להיות גדול מדי בשביל לשמור אותם בבת אחת בזיכרון המחשב (RAM), והגישה לדיסק הקשיח איטית יותר.

דירוג תוצאות המוחזרות (Ranking)

לאחר ביצוע חיפוש, השלב הבא הוא להציג את התוצאות למשתמש בצורה מסודרת ומדורגת, 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, ואפילו קורות חיים שהצטברו במשך שנים. איתור קובץ מסוים מתוך כמות גדולה של קבצים יכול להפוך למשימה מאתגרת, במיוחד כאשר לא זוכרים את שם הקובץ המדויק, את מיקומו או מאפיינים חיצוניים כמו תאריך השמירה. חיפוש בסיסי לפי מאפייני קובץ, כגון שם הקובץ, תאריך השמירה או גודלו, אינו תמיד מספק פתרון כאשר יש צורך לאתר מסמך על פי תוכנו הפנימי, כמו מילה או ביטוי ספציפי.

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

מנוע החיפוש בנוי בדרך כלל משלושה מרכיבים עיקריים:

1. **סורק (Spider)** – רכיב שתפקידו לאתר קבצים ותיקיות במחשב, לזהות את תוכנם ולחלץ מתוכם מילות מפתח. בדומה לפעולת הסריקה ברשת האינטרנט, בה הסורק עובר בין דפי אינטרנט על ידי לחיצה על קישורים, גם במחשב האישי הסורק עובר בין תיקיות וקבצים ומנתח את תוכנם.
2. **מנוע אינדקס** – רכיב זה אחראי על ארגון התוכן שנמצא על ידי הסורק ליצירת אינדקס, שהוא רשימה מסודרת של מילות מפתח, כך שניתן יהיה 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 אחזור** – לאחר הזנת שאילתה, מנוע האחזור מאתר את התוצאות מתוך האינדקס ומחזיר אותן למשתמש בצורה מדורגת וממוקדת. תוצאות החיפוש מוצגות לרוב בצורות שונות, כמו תצוגות מקדימות או רשימה של שמות קבצים ומסמכים, ומקלות על המשתמש לאתר במהירות את המידע הרלוונטי ביותר.

מנועי חיפוש נבדלים ומשתנים זה מזה בכל אחד מן הרכיבים: באלגוריתם החיפוש של איתור הקבצים, בניית האינדקס, ואיחזור הנתונים.

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

בשלב של האינדקס, משתרע תחום רחב של ביצוע מטלה זו, החל ממנועי חיפוש שאין להם מנוע אינדקס כלל, והם מבצעים חיפוש ישיר בקבצים, ועד למנועי חיפוש, שיוצרים אינדקס מפורט, עד לשמירת התכנים בשלמותם פעם נוספת אצלם.

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

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

מהו אינדקס למנועי חיפוש?

אינדקס הוא התהליך של עיון בקבצים, הודעות דואר אלקטרוני ותכנים אחרים במחשב, וקטלוג של המידע, למשל המילים והמטה-נתונים שכלולים בהם. כאשר אתה מבצע חיפוש במחשב לאחר יצירת אינדקס, החיפוש משתמש באינדקס של מונחים כדי למצוא תשובות ביתר מהירות.

בפעם הראשונה שאתה מתחיל ביצירת אינדקס, התהליך יכול להימשך כמה שעות. לאחר מכן, יצירת האינדקס תתבצע במחשב ברקע, תוך כדי שימוש רגיל במחשב, כשרק נתונים מעודכנים יתווספו לאינדקס.

איך יצירת אינדקס מאיצה חיפושים?

ממש כמו אינדקס בספר, גם אינדקס דיגיטלי מאפשר למחשב ולאפליקציות למצוא תכנים ביתר מהירות על-ידי עיון במונחים או במאפיינים נפוצים כמו תאריך היצירה של קובץ. אינדקס בנוי במלואו יכול להחזיר את קבצי המוסיקה הנכונים עבור מונח החיפוש "בטהובן" בתוך שבריר שנייה, לעומת הדקות שהפעולה עשויה להימשך ללא אינדקס.

חיפוש בדפים בודדים אחר מילות מפתח ונושאים יהיה תהליך איטי מאוד עבור מנועי החיפוש כדי לזהות מידע רלוונטי. במקום זאת, מנועי החיפוש (כולל גוגל) משתמשים באינדקס הפוך.

מהו אינדקס הפוך?

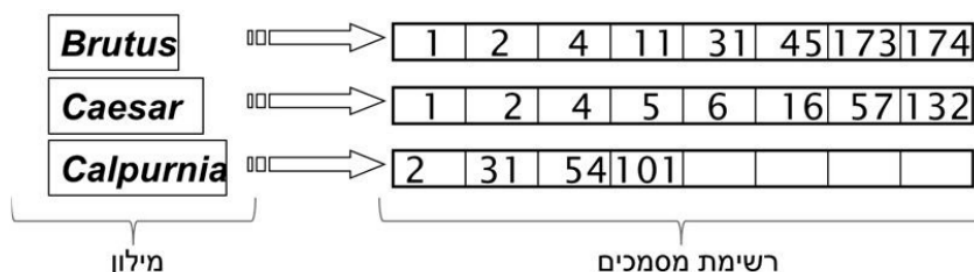
אינדקס הפוך (Inverted Index) הוא מבנה נתונים המשמש במערכות אחזור מידע, כמו מנועי חיפוש, כדי לייעל את תהליך החיפוש והשליפה של מסמכים המכילים מילות מפתח מסוימות. אינדקס הפוך הוא למעשה מיפוי של מילים או ביטויים מהמסמכים לרשימה של מסמכים שבהם כל מילה או ביטוי מופיעים, ובכך הוא "הפוך" בהשוואה למבנה רגיל, שבו כל מסמך מכיל רשימת מילים.

במקום לשמור רשימה של המילים בכל מסמך, האינדקס ההפוך 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 לעבור 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 פרסום (Posting Lists)** – עבור כל מונח במילון, נשמרת רשימה של מסמכים (לרוב כקוד זיהוי) שבהם מופיע המונח, ולעיתים קרובות נשמר גם המיקום המדויק של המילה במסמך כדי לאפשר שאילתות מתקדמות יותר.

דוגמא לאינדקס הפוך :



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

כדי לאפשר חיפוש של משפטים בתוך קבצים ולא רק מילים יש צורך בחיפוש מלא:

חיפוש בטקסט מלא כולל סקירת מספר רב של מסמכים וכמויות עצומות של טקסט. שירותי חיפוש באינטרנט משתמשים לעתים קרובות בחיפוש בטקסט מלא כדי לאחזר תוצאות רלוונטיות מהאינטרנט - בין אם זה תוכן דפי אינטרנט, קובצי PDF מקוונים ועוד. בהתחשב בנפח נתוני הטקסט המעורבים, נדרשת טכניקה לטיפול בנפח החיפוש - זה נקרא אינדקס טקסט מלא.

כדי ליצור אינדקס הפוך (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 עובר עיבוד כדי להפוך ל-term אחיד, שמייצג את צורתו הבסיסית.

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

4. מיון והסרת כפילויות – הרשימה ממוינת לפי הביטויים, ובשלב זה מוסרות כפילויות, כך שלכל term תיווצר רשימה מקושרת של המסמכים שבהם הופיע (posting list).

בסיום התהליך מתקבל אינדקס הפוך: מילון מונחים, כאשר כל מונח מקושר לרשימת המסמכים שבהם הופיע, מה שמאפשר חיפוש מהיר ויעיל במסמכים על פי תוכן.

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

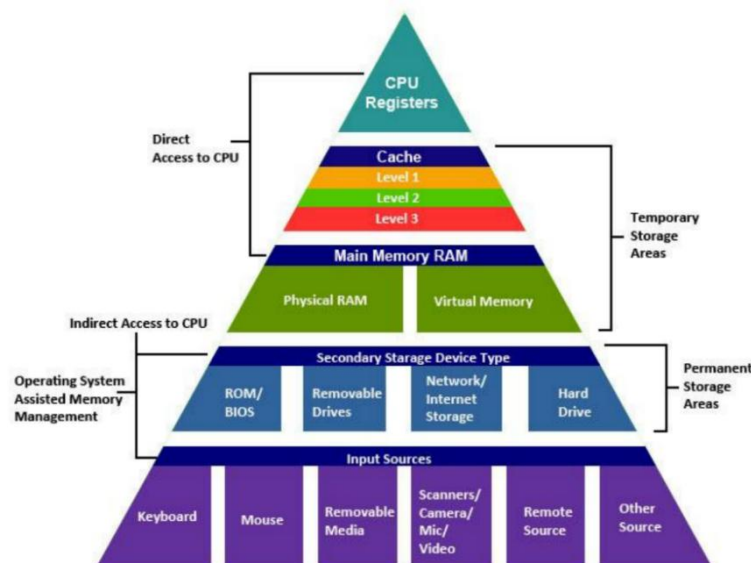
0	1	2	3	4	5	6	7	8	9	10
Apple	is	looking	at	buying	U.K.	startup	for	\$	1	billion

אלגוריתמים לבניית אינדקס יעיל

הנושא המרכזי בפרויקט זה עוסק בתהליך בניית האינדקסים (Indexing) ובשיטות השונות בהן מתבצע תהליך זה. כאשר כמות המסמכים והנתונים היא קטנה יחסית, ניתן לשמור את המילון ורשימות ה-posting במלואם בזיכרון המחשב. בהנחה זו, פעולת המיון של ה-terms יכולה להתבצע ישירות בזיכרון (RAM) בצורה יעילה ופשוטה.

כעת תיבחן מציאות שבה כמות הנתונים והמסמכים גדולה יותר, כך שלא ניתן לשמור את כל הנתונים בזיכרון המחשב בשלמותם. המטרה היא לזהות דרכים שבהן ניתן לבצע את פעולות האינדוקס בצורה יעילה גם כאשר הגישה לזיכרון המחשב מוגבלת.

לפני הצגת האלגוריתמים השונים, יינתן סקירה של מבנה החומרה, כיוון שבתחום אחזור הנתונים, רבים מההחלטות המערכתיות תלויות במבנה החומרה הזמינה. עיקרון מרכזי שנדרש להתייחס אליו הוא שהגישה לנתונים הנמצאים בזיכרון מהירה משמעותית בהשוואה לגישה לנתונים המאוחסנים בדיסק הקשיח, ולכן נדרשת אופטימיזציה של תהליכי הגישה והעיבוד בהתאם למגבלות החומרה.



אחד מהגורמים לזמני הגישה הגבוהים לנתונים המאוחסנים על גבי הדיסק הקשיח הוא מה שנקרא Time Seek, שמורכב משלושה גורמים: (1) הזמן שלוקח למקם את הראשים של הדיסק הקשיח במקומם, (2) הזמן שלוקח לסובב את הדיסק ולהביא את הנקודה הנכונה אל מתחת לראשים, ו-(3) זמן הקריאה עצמו של הראשים (אנו נתייחס לשתי הפעולות הראשונות ב-Time Seek). מכון שב-Time Seek, לא מתבצעת העברת נתונים, עדיף להעביר של גוש אחד גדול של נתונים, שהינה פעולה מהירה יותר, מאשר להעביר של מספר גושים קטנים של נתונים. נקודה נוספת שדורשת התייחסות היא שפעולות O/I של הדיסק נעשות בגושים, הנקראים בלוקים. כלומר כתיבה או קריאה של נתונים לא נעשית ברמת הבית, אלא בבלוקים (גוש

נתונים), המכילים מס' בתים כל אחד. מכון שברוב המקרים, לאחר שאנו קוראים בית אחד, אנו מעוניינים בקריאה של הבית שנמצא אחריו בקובץ, קריאה של בלוק שלם תהיה יעילה יותר מבחינת זמני התגובה. גודלו של בלוק ממוצע הוא בין 8KB ל-256KB. בזיכרון הוא של מספר GB, לפעמים עשרות. גודלו של הדיסק הקשיח הוא פי כמה (2-3) גדול מהזיכרון הקיים במחשב. נקודה נוספת אליה נתייחס היא מניעת תקלות בשרתי IR. מניעת תקלות מאד יקרה, ולכן במקום מחשב אחד עמיד ויקר עדיף כמה (עשרות/מאות/אלפים) מחשבים פשוטים וזולים. הטבלה הבאה מציגה מספר ערכים הקשורים לחומרה אשר ישמשו אותנו בהמשך בדוגמאות השונות שלנו. יש לקחת בחשבון שהערכים הללו הם לא ערכים מדויקים, ובהחלט יתכן שהם השתנו (והתקצרו) במהלך השנים. אולם, הם עדיין מהווים מדד לסדרי הגודל

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

השונים בעבודה בזיכרון המחשבים ובעזרת הדיסק הקשיח.

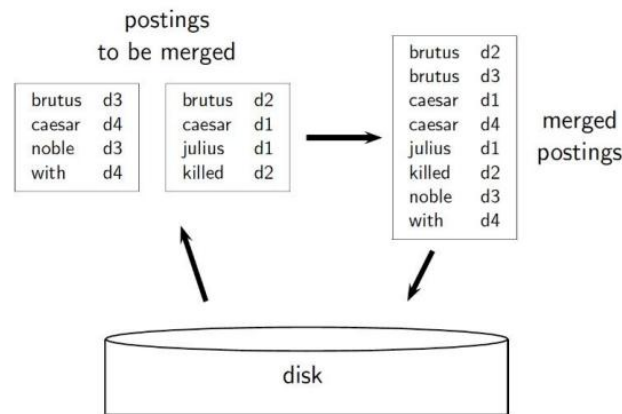
Blocked Sort-Based Indexing :BSBI

נניח ויש למיין כ-100 מיליון רשומות בגודל 12 בתים (המכילות את השדות term, docID, freq-ו שהופקו מעיבוד המסמכים. על הרשומות להימין תחילה לפי term ולאחר מכן, במיין משני, לפי docID. בשל מגבלות זיכרון, לא ניתן להחזיק את כל הרשומות בזיכרון המחשב לצורך המיין, ולכן נדרשת גישה חלופית.

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

לאחר השלמת הבנייה, כל בלוק ממויין בנפרד בזיכרון, שכן גודל הבלוקים מאפשר מיון מהיר ויעיל. בשלב הבא, מאחדים את כל הבלוקים שנשמרו לדיסק לרשימה ממוינת אחת באמצעות תהליך איחוד דמוי merge

sort: נטענים תחילת הבלוקים לזיכרון, ומשווים את הרשומות כדי ליצור רצף ממויין. כאשר נגמר הזיכרון עובר אחד הבלוקים, נטען קטע נוסף מאותו בלוק לזיכרון עד לסיום האיחוד.



דירוג את תוצאות החיפוש

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

תדירות מונח (term frequency - tf)

נניח שיש לנו קבוצה של מסמכי טקסט בעברית, וברצוננו לדרג אותם לפי רלוונטיות לשאילתה "אבטיח ללא גרעינים". דרך פשוטה להתחיל היא על ידי ביטול מסמכים שאינם מכילים את כל שלושת המילים "אבטיח", "ללא" ו-"גרעינים", אך עדיין ייוותרו בידנו מסמכים רבים. כדי להבדיל ביניהם, נרצה לספור את מספר הפעמים שכל מונח מופיע בכל מסמך, ונתון זה נקרא תדירות במונח (*term frequency*). עם זאת, כאשר אנו ניצבים בפני מסמכים בעלי אורך שונה, נרצה למדוד תדירות זו באחוזים מכלל המילים במסמך, ולא בנתון מספרי יבש.

תדירות מסמכים הופכית (Inverse document frequency - idf)

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

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

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

הן TF והן IDF למעשה מהווים גורמי משקל לחשיבות מונח מסוים. TF גבוה מעלה את משקל החשיבות של המונח, ו-IDF גבוה מקטין את משקל החשיבות.

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 שמונח t מופיע במסמך d . המכנה הוא המספר הכולל של מונחים במסמך d (ספירת כל מופע של אותו מונח בנפרד). ישנן דרכים נוספות אחרות המגדירות נתון זה, אך האמורה לעיל היא העיקרית והבסיסית ביותר.

תדירות מסמכים הופכית

תדירות מסמכים הופכית היא מדד לכמות המידע שהמילה מספקת, כלומר אם היא נפוצה או נדירה בכל המסמכים. המדד הוא השבר ההפוך בקנה מידה לוגריתמי של המסמכים שמכיל את המילה, ומושג על ידי חלוקת המספר הכולל של המסמכים (למשל, המספר 3 בקורפוס שמונה שלושה מסמכים בלבד) במספר המסמכים המכילים את המונח, ולאחר מכן לקיחת הלוגריתם של אותה מנה:

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

כאשר:

N : המספר הכולל של מסמכים בקורפוס

df_t : מספר המסמכים שבהם המונח t מופיע.

תדירות מונח – תדירות מסמכים הופכית (TF-IDF)

לאחר שחישבנו כל גורם בנפרד, השילוב של השניים, idf-tf מחושב כ:

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

משקל גבוה ב-TF-IDF מגיע על ידי תדירות מונח גבוהה (במסמך הנתון) ותדירות מסמכים נמוכה של המונח בכל אוסף המסמכים; לכן המשקולות נוטות לסנן מונחים נפוצים. מכיוון שהיחס בתוך הפונקציה הלוגריתמית של תדירות המסמכים ההופכית תמיד גדול או שווה ל-1, הערך של תדירות המסמכים ההופכית ((ובהתאמה של TF-IDF) גדול או שווה ל-0. ככל שמונח מופיע במסמכים נוספים, היחס בתוך הלוגריתם מתקרב ל-1, ומקרב את ה-IDF ואת TF-IDF ל-0.

הליכים עיקריים בפתרון בעיה בטכנולוגיות הנדסה מתקדמות

1. סריקת כלל הקבצים במערכת (10-12) – ביצוע סריקה מקיפה של כל קבצי המערכת במטרה לזהות ולאתר את כלל הקבצים הקיימים, כולל נתיבי גישה ומיקום מדויק.
2. חילוץ מטה-דטה מהקבצים (12-20) – איסוף מאפייני הקובץ המרכזיים, כגון שם הקובץ, תאריכי יצירה ושינוי, סוג הקובץ ומידע נוסף המאפשר זיהוי ותיעוד מסודר של הקבצים.
3. חילוץ תוכן מקבצים מסוגי טקסט, Word ו-PDF (01-01) – שליפת התוכן הטקסטואלי מתוך קבצים בפורמטים נפוצים במטרה להנגיש את המידע לניתוח ועיבוד בהמשך.
4. תוכני הקבצים (full-text-search) (01-02) – עיבוד התכנים שנשלפו כדי לזהות מידע מהותי, כגון מילות מפתח, ביטויים עיקריים ונתונים רלוונטיים נוספים להמשך ניתוח הפרויקט.
5. שמירת המידע במבנה נתונים גמיש לשליפה (01-05) – ארגון כל הנתונים שהופקו במבנה נתונים יעיל, 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 המסמכים לפי תדירות המילים במסמך ונדירותן במאגר. המטרה היא להציג את התוצאות הרלוונטיות ביותר ראשונות.

גבולות הפרויקט

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

פיתוחים עתידיים

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

תיאור טכנולוגיות הנדסה

- java צד שרת מקומי -
- C# צד לקוח -

מסד נתונים

NoSql

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

מסד נתונים NoSQL מאפשר חיפוש בטקסט חופשי ובמבנים גמישים כמו מסמכים, כך שתוכל לשמור ולחפש תוכן משתנה או לא-מובנה. לדוגמה, אם רוצים לבצע חיפושים בתוך טקסטים, מסדי נתונים (שהוא NoSQL מבוסס על אינדקסים) מתמחים בחיפושים בטקסט חופשי ויודעים לספק תוצאות מהירות גם בכמויות נתונים גדולות.

2. מבוא

הרקע לפרויקט

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

מנועי החיפוש המובנים במערכות ההפעלה לרוב אינם מספקים חיפוש יעיל ומדויק בתוכן הקבצים, ומתמקדים בעיקר בשמות קבצים ומטא-דאטה בסיסי. בעיה נוספת היא שחיפוש ישיר בתוכן הקבצים (ללא אינדקס) הוא תהליך איטי מאוד, במיוחד כאשר יש אלפי קבצים במערכת.

תהליך המחקר

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

ע"מ לבנות טבלה הממפה את הקבצים ומאפשרת חיפוש בצורה מהירה, העמקתי איך המסדי הנתונים עובדים:

- חקרתי איך הוא מאחסן את הנתונים. (באיזה סוג קבצים איך הם מחולקים....)
 - וכן איך הוא יוצר את ה index - באיזה מבנה נתונים הם משתמשים ומה האפשרות הכי יעילה.
 - חקרתי איפה כדאי לאחסן את הטבלת index
- מה היתרונות באחסון בדיסק מה היתרונות באחסון בזיכרון RAM ומה היתרונות בשילוב של האחסון בדיסק ובזיכרון ה 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 שולחני שבו אין יתרון להקשר סטטיסטי של דפי אינטרנט אלא מדובר במסמכים עצמאיים.

נושאים כללים שחקרתי:

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

כמו כן, הקדשתי זמן להבנת היבטי יעילות של זמן ומקום – כיצד לתכנן את המערכת כך שתספק תוצאות במהירות תוך שמירה על צריכת זיכרון ואחסון נמוכה. השווייתי בין שימוש במבני נתונים בזיכרון (כמו HashMap ו-Trie) לבין פתרונות המבוססים על אחסון קבוע בדיסק, כגון מסדי נתונים או קבצי אינדוקס.

בנוסף, חקרתי כלים טכניים וספריות רלוונטיות שיכולות לתמוך בפרויקט, כמו ספריות לקריאת קבצים, ניתוח טקסט, ו-tokenization. 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.

סקירת ספרות

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

- האתר [Geeks for Geeks](#) שימש להבנה של מושגים בסיסיים ומתקדמים בתחום מבני הנתונים ומנועי החיפוש.
- הספר [Introduction to Information Retrieval](#) מאת Manning, Raghavan ו-Schütze, ובפרט הפרקים [Blocked Sort-Based Indexing](#) ו-[Single-Pass In-Memory Indexing](#), סייעו להבנת תהליך בניית אינדקסים יעיל בזיכרון, כולל המושגים [Inverted Index](#) ו-[Posting List](#).
- באתר [kmwllc.com](#) נלמדו ההבדלים בין גישות שונות בתחום מערכות אחזור מידע.
- באתר [pinecone.io](#) נבחנו ספרות מקצועית בנושא שיטות דירוג לתוצאות חיפוש, כולל שימוש במדדים סטטיסטיים כגון [IDF \(Inverse Document Frequency\)](#), [TF \(Term Frequency\)](#), מדד דמיון קוסינוס ([Cosine Similarity](#)), וכן האלגוריתם [BM25](#), הנחשב לאחת מהשיטות המתקדמות ביותר בתחום.
- להבנת מבני נתונים תומכים לאינדוקס ואחזור יעיל, כגון [B+ Tree](#), נעשה שימוש במאמרים ובתיעוד מאתר [PlanetScale](#), העוסק במסדי נתונים מבוזרים.

אתגרים מרכזיים

הבעיה איתה התמודדתי

1. **בניית אינדקס יעיל** - אחד האתגרים המרכזיים היה פיתוח אלגוריתם יעיל לבניית האינדקס ההפוך, במיוחד כאשר מדובר בכמויות גדולות של טקסט. היה צורך למצוא פתרון שיאזן בין מהירות הבנייה לבין יעילות החיפוש העתידי.
2. **ניהול זיכרון אופטימלי** - בזמן סריקת קבצים רבים ובניית האינדקס, היה אתגר משמעותי בניהול הזיכרון כדי למנוע דליפות 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 לנווט בכמויות המידע הגדולות, ובמקביל ללמוד לעומק על טכנולוגיות מתקדמות בתחום אחזור המידע ומבני נתונים יעילים.

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 ובמערכות ארגוניות הולכת וגדלה, קיים צורך ממשי בכלי חיפוש מהיר, מדויק ונוח לשימוש, שאינו תלוי בחיבורים לרשת, ואינו דורש התקנת תוכנות צד שלישי. משתמשים רבים נתקלים בקושי לאתר קבצים או תוכן טקסטואלי בתוך קבצים באופן מיידי, במיוחד כאשר מדובר במבני תיקיות מורכבים או בכמויות 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 חוויית שימוש אינטואיטיבית ויעילה, המותאמת לצרכים אמיתיים של חיפוש קבצים בסביבות עבודה אישיות וארגוניות, ללא תלות בטכנולוגיות חיצוניות או משאבים מורכבים.

הצגת פתרונות לבעיה

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 יעיל של מסמכים:** חלוקת התוכן ליחידות מידע קטנות (Tokenization), והמרתם למבנה חיפוש מותאם, כולל טיפול במילון מונחים ומזהי מסמכים.

- **ניהול משאבים בזיכרון ודיסק:** שימוש בגישות שמאפשרות טיפול באוספים גדולים, באמצעות חלוקה לבלוקים, מיון חוץ-זכרונות או בנייה ישירה 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 איכות ההתאמה, באמצעות מודלים כגון TF-IDF, BM25 ודמיון קוסיני (Cosine Similarity).
- **התמקדות בחוויית המשתמש:** שמירה על זמני תגובה מהירים, ממשק חיפוש אינטואיטיבי, ויכולת ניתוח שאילתות 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 בתוך 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.
- הפחתת הזמן הנדרש לאיתור מידע ספציפי בתוך קבצים.
- הנגשת יכולות חיפוש מתקדמות שבדרך כלל קיימות רק במנועי חיפוש אינטרנטיים.
- יצירת תשתית שניתנת להרחבה לסוגי קבצים נוספים בעתיד.

יעדים טכניים

- **חילוץ תוכן מקבצים** - פיתוח יכולת לחלץ טקסט מקבצים בפורמטים שונים (txt, pdf, doc, docx, json, csv, log, md, java, html, xml) לצורך אינדוקס.
- **בניית אינדקס הפוך יעיל** - פיתוח מנגנון לבניית אינדקס הפוך (Index Inverted) 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 לשמירת האינדקס ישירות בדיסק ללא תלות במסד נתונים חיצוני, תוך מקסום ביצועי גישה.
- **ממשק משתמש אינטואיטיבי ב JavaFX** - פיתוח ממשק משתמש מודרני ויזואלי באמצעות JavaFX שיאפשר למשתמשים לחפש ולסנן תוצאות בקלות.
- **ביצועים מהירים** - השגת זמן תגובה מהיר בחיפושים (פחות משנייה לרוב החיפושים).
- **יעילות בשימוש במשאבים** - אופטימיזציה של צריכת המשאבים (זיכרון ומעבד) במהלך הסריקה, האינדוקס והחיפוש.

5. מדדי הצלחה למערכת

1. **זמן תגובה** - חיפוש מילה או ביטוי בתוך מאגר של לפחות 10,000 קבצים בזמן קצר מ-1 שנייה.
2. **שימוש משאבים אופטימלי** - צריכת זיכרון RAM לא תעלה על 500MB בזמן פעולה רגילה, ולא יותר מ-3GB בזמן בניית האינדקס.
3. **חיסכון בזמן למשתמש** - קיצור זמן האיתור של מידע ספציפי.
4. **תמיכה בסוגי קבצים** - תמיכה 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 הפוך באופן שיאפשר גישה מהירה, תוך שמירה על יעילות בשימוש במשאבים.
3. **איזון בין זיכרון לביצועי דיסק** - כשהאינדקס גדול מדי להיטען כולו לזיכרון RAM, נוצרת דילמה בין העלאת האינדקס כולו (שפוגעת בביצועי המערכת ולא יעילה עם כמויות 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 בינארי לחיסכון במקום דיסק הקשה משמעותית על תהליכי הדיבוג והפיתוח, מכיוון שהמידע לא היה קריא 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 תאורטי

מהו זיכרון RAM?

לכל מחשב בימינו ישנו כרטיס זיכרון ram אחד לפחות, כרטיס זה אינו מיועד לשמירת מידע לצמיתות אלא בנוי לשמירת מידע זמני, כרטיס זיכרון זה מהיר מאוד ביחס לדיסק הקשיח, ומיועד בעיקר לאחסון התוכנות שאנו עובדים עליהם ברגע זה באופן זמני כדי שיפעלו במהירות מירבית, זיכרון זה מוגבל בכמות המידע אותו הוא יכול לאחסן לכן מומלץ להרחיב אותו במידה ואנו רוצים לעבוד על מספר רב של תוכנות במקביל, בשפה המקצועית זיכרון זה נקרא "זיכרון נדיף" או באנגלית ראשי תיבות של random access memory למה הוא

נקרא זיכרון נדיף? כי הוא אינו שומר את המידע עד שמוחקים אותו (זה מה שעושה הדיסק הקשיח) המידע מאוחסן בו מתנדף ממנו ברגע שאנו סוגרים תוכנה מסויימת שעבדנו עליה. שם נוסף שהוא זכה לו הוא "זיכרון עבודה" הוא אחראי על אחסון התוכנות איתם אנו עובדים בלבד ובאופן זמני כמובן.

כפי שצינו בהתחלה, ה RAM -משמש כזיכרון לטווח קצר של המחשב כדי לייעל את העבודה שלו, אבל מה באמת הוא עושה ? ובכן, מפני שהמעבד לא יכול לשמור אצלו הרבה מידע בזמן עיבוד נתונים, כמו למשל עבודה על קובץ טקסט, הוא חייב "לאחסן" את המידע במקום מסוים שהוא יוכל לגשת אליו בצורה הכי מהירה שאפשר.

אבל כאן יש איזושהי בעיה קטנה, אם המשתמש יבקש מהמחשב לשלוף מידע מהכונן הקשיח, ייקח לו הרבה זמן יחסית לשלוף 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 "זמניים" על מנת שיהיו מוכנים לשליפה "מהירה" לעומת הכוננים הקשיחים. לדוגמה: כל המידע שנאגר מעבודה עם תוכנות או קבצים מסוימים במחשב, גלישה באינטרנט ואפילו צפייה בסרטונים מאוחסן בזיכרון ה RAM - לשימוש בעת הצורך.



איך RAM עובד?

זיכרון ה- RAM מורכב מהרבה קבלים וטרנזיסטורים קטנים מאוד שיכולים לאחסן מטען חשמלי המייצג בעצם פיסות מידע קטנות, בדיוק כמו במעבדים ושאר רכיבי חומרה אחרים שקיימים במחשב. מפני שה- RAM מסוגל לאחסן מטען חשמלי לזמן 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 (HDD) Hard disk drive), הוא רכיב במחשב המשמש לשמירת נתונים. דיסק קשיח יכול להכיל בדרך כלל כמות גדולה של נתונים לעומת זיכרונות אחרים, אך פעולתו איטית לעומת הזיכרון הפנימי של המחשב (RAM). הדיסק הקשיח הוא התקן זיכרון בלתי נדיף המאפשר אחסנה אמינה של נתונים דיגיטליים בנפח גדול ובזמן גישה קצר יחסית להתקני זיכרון מכניים אחרים. בהשוואה להתקני זיכרון אחרים באותה הקיבולת, הדיסק הקשיח זול משמעותית, אולם זמן הגישה לנתונים בדיסק ארוך בהשוואה לזיכרון הפנימי (RAM), דיסק קשיח מכני איטי פי 100,000 מהזיכרון הפנימי.



אופן פעולת 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 לעומק את אופן פעולתו של אמצעי האחסון – הדיסק. פעולות אלו, למרות שהן נראות פשוטות, עשויות להיות נקודת תורפה משמעותית מבחינת 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 הכולל זרוע מכנית הנעה מעל פלטות מסתובבות. כל פעולת קריאה או כתיבה מחייבת חיפוש (seek) של הזרוע אל המיקום המתאים, פעולה המוסיפה השהיה. זמני גישה ממוצעים נעים סביב 5–15 מילישניות, כאשר מהירות הקריאה הרציפה מגיעה לרוב עד 150 MB/s (תלוי בדגם).

2. (SSD) Solid State Drive:

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

מעל GB/s3). עם זאת, גם ב-SSD, ביצוע קריאות רבות של קבצים קטנים בקפיצות לא רציפות (Random Access) עלול להיות איטי יותר מהצפוי עקב עיכובים פנימיים ובקרה של בקר הזיכרון.

ההבדל בין RAM לדיסק קשיח

פונקציונליות:

- זיכרון RAM הוא זיכרון נדיף הפועל כאזור אחסון זמני עבור נתונים שהמחשב משתמש בהם באופן פעיל. הוא מספק גישה מהירה למידע שהמעבד צריך כדי לבצע משימות, כגון הפעלת תוכניות ואחסון נתונים שנמצאים בעיבוד פעיל.
- HDD הוא זיכרון לא נדיף המספק אחסון לטווח ארוך עבור קבצים, יישומים ומערכת ההפעלה. הוא מורכב מדיסקים מסתובבים וחלקים מכניים שקוראים וכותבים נתונים בצורה מגנטית. כונני HDD איטיים יותר מכונני SSD אך לרוב מציעים נפחי אחסון גדולים יותר בעלות נמוכה יותר.

מהירות:

- זיכרון RAM הוא סוג הזיכרון המהיר ביותר הזמין במערכת מחשב. הוא מספק גישה כמעט מיידי לנתונים, ומאפשר למעבד לאחר מידע במהירות.
- דיסקים קשיחים מסוג HDD איטיים יותר בהשוואה ל-RAM וגם ל-SSD. הדיסקים המסתובבים והחלקים המכניים יוצרים עיכוב בגישה לנתונים, וכתוצאה מכך מהירויות קריאה וכתובה איטיות יותר.
- כונני SSD מהירים יותר מכונני HDD אך איטיים יותר מ-RAM. הם מציעים מהירויות קריאה וכתובה מהירות יותר משמעותית בהשוואה לכונני HDD מסורתיים, וכתוצאה מכך זמני אתחול וקצבי העברת נתונים מהירים יותר.

קיבולת:

- קיבולת זיכרון RAM קטנה בדרך כלל מקיבולת האחסון של SSD או HDD. זיכרון RAM נמדד בדרך כלל בג'יגה-בייט (GB) ובדרך כלל נע בין 4 GB ל-32 GB או יותר.
- דיסקים קשיחים מסוג HDD מציעים בדרך כלל קיבולות אחסון גדולות יותר בהשוואה לכונני SSD. הם זמינים בגדלים 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 שהקיבולת והמהירות גדלים.
- HDD: דיסקים קשיחים הם בדרך כלל האפשרות החסכונית ביותר לאחסון. הם מציעים יכולות אחסון גדולות יותר בעלות נמוכה יותר בהשוואה לזיכרון 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 שאילתות. מנוע החיפוש סורק את האינטרנט באופן מתמיד, יוצר אינדקס של התוכן ומדרג את התוצאות על פי אלגוריתמים מורכבים. כאשר משתמש מקיש שאילתה, המנוע מציג את התוצאות הרלוונטיות ביותר.

מנועי חיפוש מובילים:

- 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 Office ו-OneDrive.
- דאק דאק גו (DuckDuckGo): מנוע חיפוש המדגיש את פרטיות המשתמשים ואינו עוקב אחרי פעילותם. מספק תוצאות ללא התאמה אישית, מה שמבטיח חוויית חיפוש ניטרלית ומאובטחת.

מנוע חיפוש קבצים

מנוע חיפוש קבצים הוא כלי 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 במסדי נתונים

אינדקס במסד נתונים הוא עותק מאורגן של חלק מהטבלה, בדרך כלל שדה קצר, השמור במקום פיזי נפרד בדיסק. כאשר מתבצעת שאילתה, 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

מהי טבלת אינדקס הפוכה? (Inverted Index)

במדעי המחשב, אינדקס הפוך (המכונה גם רשימת פרסומים, קובץ פרסומים או קובץ הפוך) הוא אינדקס בסיס נתונים המאחסן מיפוי מתוכן, כגון מילים או מספרים, למיקומיו בטבלה, או במסמך או בקבוצת מסמכים (בניגוד לאינדקס קדמי, הממפה ממסמכים לתוכן). מטרתו של אינדקס הפוך היא לאפשר חיפוש טקסט מלא מהירים, במחיר של עיבוד מוגבר כאשר מסמך מתווסף לבסיס הנתונים.

אינדקס הפוך הוא מבנה נתונים המשמש במערכות אחזור מידע לאחזור יעיל של מסמכים או דפי אינטרנט המכילים מונח ספציפי או קבוצת מונחים. באינדקס הפוך, 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 קבוצת מונחים, מנוע החיפוש שואל את האינדקס ההפוך עבור אותם מונחים ומאחזר את רשימת המסמכים הקשורים לכל מונח.

אחסון נתונים ואינדקסים

אחסון נתונים בטבלאות:

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

כל בלוק מכיל שורות מהטבלה או חלקי שורות

השורות נשמרות ברצף או מפוזרות לפי סדר הכנסתן

אחסון אינדקסים:

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

מהו דירוג תוצאות?

TF-IDF מייצג תדירות מונח-תדירות מסמך הפוכה. משקל TF-IDF הוא משקל המשמש לעתים קרובות באחזור מידע וכריית טקסט. וריאציות של סכימת השקלול TF-IDF משמשות לעתים קרובות על ידי מנועי חיפוש בניקוד ודירוג הרלוונטיות של מסמך הנתון שאילתה. משקל זה הוא מדד סטטיסטי המשמש להערכת חשיבותה של מילה למסמך באוסף או קורפוס. החשיבות גדלה באופן פרופורציונלי למספר הפעמים שמילה מופיעה במסמך אך מקוזזת על ידי תדירות המילה בקורפוס (מערך הנתונים).

בהקשר של תוצאות חיפוש TF-IDF, עוזר לדרג דפי אינטרנט על בסיס הרלוונטיות שלהם לשאילתה ספציפית. זה מבטיח שמנועי חיפוש מספקים את התוצאות הרלוונטיות ביותר לשאילתות המשתמשים. בכל פעם שמשתמש מקליד שאילתת חיפוש למנוע חיפוש כמו Google, רשת מורכבת של אלגוריתמים קופצת לפעולה, עובדת קשה כדי לספק את תוצאות החיפוש הרלוונטיות ביותר. אחד המדדים היסודיים שעליהם האלגוריתמים האלה מסתמכים כדי לקבוע רלוונטיות ותוכן הוא TF-IDF

מכיוון ש TF-IDF-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 למשתמש מקבלות ציוני TF-IDF גבוהים יותר TF-IDF. בוחן מילים בהתאם לרלוונטיות שלהן.

קריטריונים אפשריים לדירוג

ה TF-IDF הוא המכפלה של שתי סטטיסטיקות, תדירות מונח ותדירות מסמך הפוכה. ישנן דרכים שונות לקביעת הערכים המדויקים של שתי הסטטיסטיקות. תדירות מונח, $TF(t,d)$, היא התדירות היחסית של מונח t בתוך מסמך d , כאשר d הוא הספירה הגולמית של מונח במסמך, כלומר, מספר הפעמים שמונח t מופיע במסמך d

תדירות המסמך ההפוכה היא מדד לכמה מידע המילה מספקת, כלומר, כמה נפוצה או נדירה היא בכל המסמכים. משקל גבוה ב TF-IDF מושג על ידי תדירות מונח גבוהה (במסמך הנתון) ותדירות מסמך נמוכה של המונח בכל האוסף של המסמכים; המשקלים נוטים לסנן מונחים נפוצים

מילים נפוצות כמו "the" או "and" מופיעות בתדירות גבוהה בכמעט כל הטקסטים באנגלית. אם היינו מסתמכים רק על תדירות מונח, אלגוריתם מנוע חיפוש עלול בטעות להקצות חשיבות יתר לדפים שמשמשים בתדירות גבוהה במילים נפוצות אלה תוך התעלמות מדפים המכילים מילים משמעותיות יותר ופחות נפוצות כמו "אורגני", "קפה" ו"שעועית". כדי לתקן זאת, נעשה שימוש בגורם הידוע כתדירות מסמך הפוכה (IDF). IDF מקטין את המשקל של מונחים המופיעים בתדירות גבוהה מאוד במסמכים מרובים) כמו "the" או "and" ומגביר את המשקל של מונחים המופיעים לעתים רחוקות (כמו "שעועית קפה אורגני")

דוגמה לשיטות דירוג כמו TF-IDF

TF-IDF (תדירות מונח-תדירות מסמך הפוכה) הוא סטטיסטיקה מספרית המשמשת למדידת חשיבותה של מילה בתוך מסמך ביחס לקורפוס של מסמכים. הוא משמש בדרך כלל במשימות אחזור מידע ועיבוד שפה טבעית, 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 גבוהה יותר מציעה חשיבות גדולה יותר. אם מונח מופיע בתדירות גבוהה במסמך, הוא כנראה רלוונטי לתוכן המסמך TF. לא לוקח בחשבון את החשיבות הגלובלית של מונח בכל הקורפוס. מילים נפוצות כמו "the" או "and" עשויות להיות להן ציוני TF גבוהים אך אינן משמעותיות בהבחנה בין מסמכים.

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

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

הוא שימש לעתים קרובות כגורם שקלול בחיפושים של אחזור מידע, כריית טקסט ומידול משתמשים. סקר שנערך בשנת 2015 הראה ש-83% ממערכות ההמלצה מבוססות הטקסט בספריות דיגיטליות השתמשו

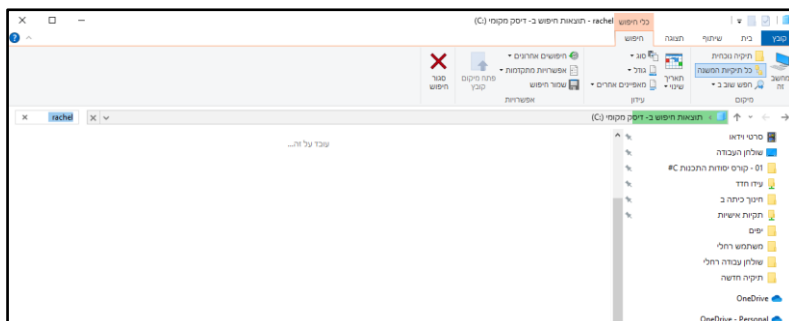
בTF-IDF -וריאציות של סכימת השקלול TF-IDF שימשו לעתים קרובות על ידי מנועי חיפוש ככלי מרכזי בניקוד ודירוג הרלוונטיות של מסמך הנתון שאילתת משתמש .

אחת מפונקציות הדירוג הפשוטות ביותר מחושבת על ידי סיכום TF-IDF עבור כל מונח שאילתה; פונקציות דירוג מתוחכמות רבות יותר הן וריאנטים של המודל הפשוט הזה. מנועי חיפוש בעוצמה תעשייתית עובדים על ידי שילוב של מאות אלגוריתמים שונים לחישוב רלוונטיות, אבל נממש רק שניים: תדירות מונח ותדירות מסמך הפוכה (TF-IDF) עם דירוג דמיון קוסינוס .

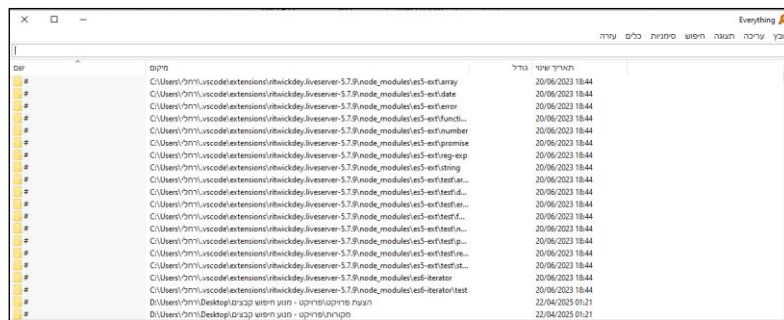
7. תיאור מצב קיים

כיום קיימים מספר פתרונות לחיפוש מידע במחשב אישי, אך רובם מוגבלים בהיקף או ביכולות שלהם:

1. **חיפוש מובנה של Windows** - מערכת ההפעלה Windows מספקת יכולת חיפוש בסיסית המתמקדת בעיקר בשמות קבצים ותיקיות. החיפוש בתוכן קבצים נהיה איטי משמעותית ככל שמעמיקים בהיררכיית התיקיות או כאשר הנפח הכולל של הנתונים גדל.

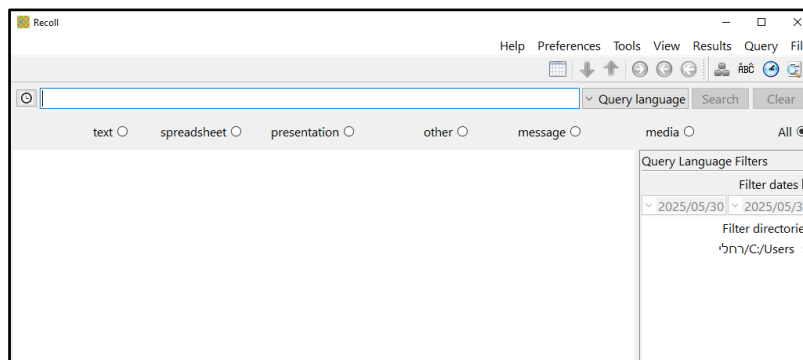


2. **Everything** - תוכנה המתמחה בחיפוש מהיר של שמות קבצים ותיקיות, אך אינה מבצעת חיפוש בתוכן הקבצים.

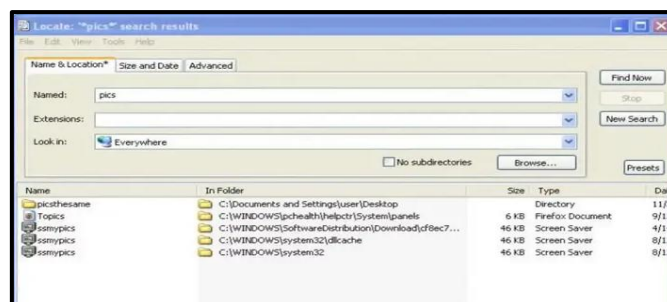


3.

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 ללא יכולת חיפוש בתוכן.



המגבלות העיקריות של הפתרונות הקיימים כוללות:

- חוסר יכולת לחפש בתוכן קבצים בפורמטים מגוונים
- חוסר יעילות בדירוג תוצאות לפי רלוונטיות

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

8. ניתוח חליפות מערכתי

סריקת תוכן הקבצים בצורה יעילה

בעת בניית אינדקס הפוך (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 מיון כללי: תחילה לפי term בסדר לקסיקוגרפי, ואז מיון משני לפי docID. אם כל רשומה תופסת 12 בתים, והכמות הכוללת היא כ-100 מיליון רשומות (כמו במקרה של אוסף 1RCV), אז מדובר בכ-1.12GB בזיכרון – גודל שעדיין מאפשר עבודה בזיכרון (RAM) בלבד.

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

מדוע לא ניתן למיין ישירות על הדיסק?

אם נשתמש באלגוריתם מיון 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-92.6 ימים. לכן נדרש פתרון חלופי.

BSBI – Blocked Sort-Based Indexing

תיאור האלגוריתם:

- מפצל את אוסף הזוגות (term, docID) לבלוקים בגודל שמתאים לזיכרון.
- ממיין כל בלוק בזיכרון ושומר אותו לדיסק.

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

יתרונות:

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

חסרונות:

- **דורש שמירת כל הטרמינולוגיה המקורית:** 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 לבזבז זמן ומשאבים.
- **יצירת posting lists – רק בסוף:** כל רשימת הפניות נוצרת רק לאחר סיום כל תהליך המיזוג.

SPIMI – Single-Pass In-Memory Indexing

תיאור האלגוריתם:

- בונה את ה-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 מיפוי ID → term בזמן הריצה; כל term נשמר רק בבלוק שלו.
- **יעילות גבוהה:** לא נדרש מיון ביניים של ה-terms – פחות עומס חישובי.

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

חסרונות:

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

שימוש במבנה נתונים לאינדקס

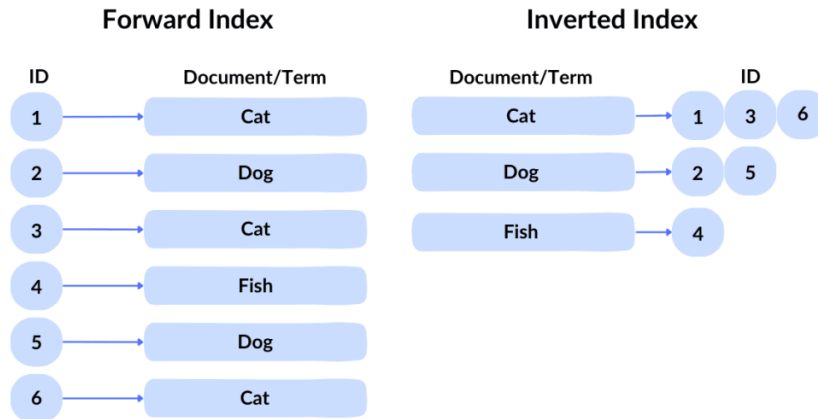
אינדקס הפוך (Inverted Index):

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. שמו של האינדקס נובע מהעובדה שהמיפוי הוא "הפוך" ביחס למבנה נתונים רגיל, שבו נשמרים המונחים לפי מסמך; 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 הפרסום היא הרשימה המצורפת לכל מונח באינדקס ההפוך, הכוללת את מזהי המסמכים שבהם מופיע המונח. רשימה זו יכולה לכלול גם מידע משלים, כגון תדירות הופעת המונח בכל מסמך, מיקומי המונח בתוך המסמך או נתונים נוספים, אשר משפרים את איכות דירוג התוצאות וחווית החיפוש הכוללת.

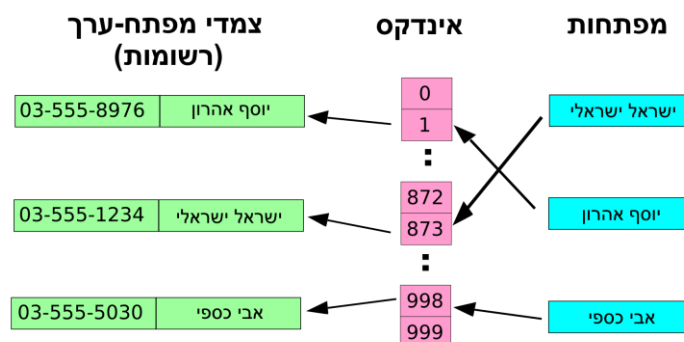


כדי לשפר את ביצועי מנוע החיפוש, יש לשלב את מבנה ה-inverted index עם מבני נתונים בעלי גישה מהירה ויעילה. בשלב זה, אערוך סקירה מקצועית של מספר מבני נתונים רלוונטיים, תוך הדגשת יתרונותיהם וחסרונותיהם והשפעתם על מהירות המערכת.

hash table

תיאור המבנה

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



איך עובד

- כל מפתח מעובד לפוזיציה בטבלה באמצעות פונקציית גיבוב.
- פתרון התנגשויות מתבצע באמצעות שרשור (chaining) או טיפול פתוח (open addressing).
- מאפשר 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

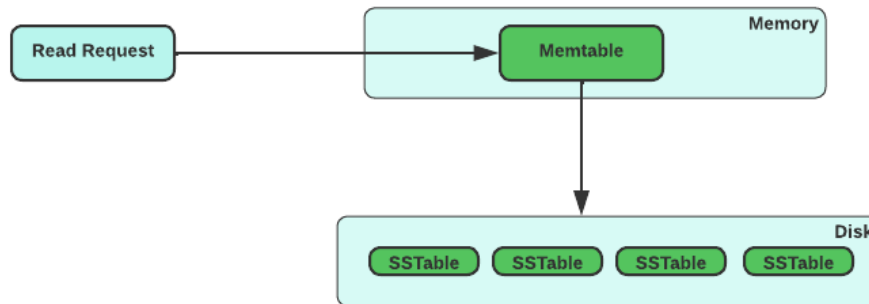
חסרונות:

- לא מאפשר חיפוש טווח
- ככל שמספר המונחים גדל עלול להתרחש התנגשויות (כשמספר רב של מילים ממופות לאותו מקום בטבלה), ואז זמן החיפוש הוא $O(n)$
- מחייב שכל המונחים יהיו בזיכרון (יקר)

LSM Tree (Log-Structured Merge-Tree)

תיאור המבנה:

מבנה נתונים המתאים לכתיבה אינטנסיבית, המשלב מספר רמות של קבצים מסודרים.



איך המבנה עובד:

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

סיבוכיות זמן:

- כתיבה: $O(\log n)$ - מהיר מאוד!
- חיפוש: $O(\log n \times \text{מספר SSTables})$ - יכול להיות איטי
- איחוד: $O(n \log n)$ - רק ברקע

יתרונות:

- כתיבה מהירה מאוד – הנתונים נרשמים תחילה בזיכרון ובלוג בלבד.
- יעיל לעומסי כתיבה גבוהים – מאפשר עבודה עם כמויות גדולות של מידע משתנה.
- דחיסת נתונים טבעית – תהליך האיחוד (compaction) מוחק נתונים ישנים ומפנה מקום.

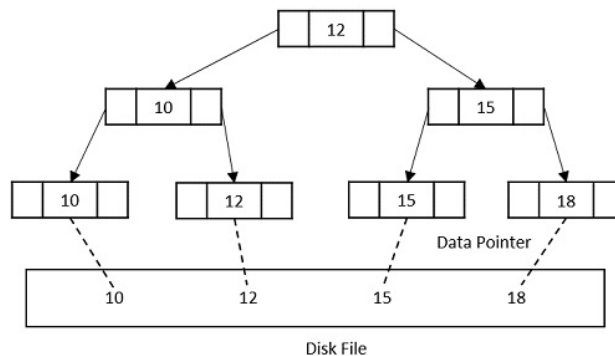
חיסרון עיקרי:

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

B+ Tree

תיאור המבנה:

עץ איזון מסוג B שבו כל המפתחות נמצאים בעלים בלבד, והצמתים הפנימיים מכילים K מפתחות המשמשים ניווט בלבד. עלים מחוברים ברשימה מקושרת.



איך המבנה עובד:

- מבנה עץ מאוזן בו כל הצמתים יכולים להכיל מספר מפתחות ומצביעים.
- חיפוש נעשה באמצעות ניווט מהשורש אל העלים לפי מפתחות.
- הוספה ומחיקה מלוות באיזון מחדש ושמירה על תנאי העץ.

סיבוכיות זמן:

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

יתרונות:

- מאוזן תמיד לכן כל החיפושים הם בזמן קבוע
- יעיל לחיפושי טווח בזכות הקישור בין העלים.
- מותאם לעבודה עם דיסק, כל צומת יכול להכיל הרבה מפתחות, מה שמקטין את מספר הגישות לדיסק.
- כל הערכים מרוכזים רק בעלים מה שמסייע בשמירה על מבנה פשוט לחיפוש וסריקה.

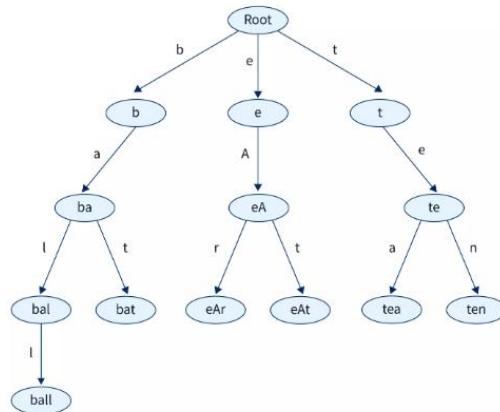
חסרונות

- מורכבות יישום גבוהה - פיצול וחיבור צמתים בבניה בהוספה ובעדכון
- לא אופטימלי לנתונים קטנים - יש overhead

Trie

תיאור המבנה:

עץ מילולי שמייצג אוסף מחרוזות, כאשר כל צומת מייצג תו 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 מילים עם קידומת משותפת (autocomplete).
- חוסך מקום באחסון מחרוזות עם קידומות משותפות.
- מתאים לחיפוש לפי תחילית או חיפוש טווח

חסרונות:

- צריכת זיכרון גבוהה - הרבה צמתים עבור אלפבית גדול
- עץ לא מאוזן - יכול להיות עמוק עבור מילים ארוכות

דירוג

TF-IDF (Term Frequency–Inverse Document Frequency)

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

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

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

$$TF\ IDF = TF * IDF$$

כאשר $TF(t,d)$ היא תדירות המונח במסמך, N מספר המסמכים בקורפוס ו- $DF(t)$ מספר המסמכים שבהם מופיע המונח.

יתרונות:

פשטות ויעילות חישובית, מתאים לטקסטים בעלי מבנה אחיד.

חסרונות:

אינו 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 בכלל המסמכים (IDF), ואורך המסמך.

השיטה נותנת משקל גבוה יותר למונחים שחוזרים בשאלתה, תוך איזון שמונע הטיה לטובת מסמכים

ארוכים מדי. יתרונות: מדויק יותר ביישומי 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 לטובת מסמכים ארוכים
- מדויק יחסית במנועי חיפוש טקסטואליים
- אינו דורש אימון או מודלים מורכבים – פשוט וחזק
- מתאים מאוד לשאלות טקסטואליות רגילות עם מונחים בודדים

חסרונות:

- דורש כוונן פרמטרים
- K (רמת רוויה של תדירות המונח): קובע עד כמה תדירות המונח במסמך משפיעה על ציון הרלוונטיות – ערך גבוה מאפשר לתדירות להשפיע יותר.
- b (השפעת אורך המסמך): מאזן בין אורך המסמך לתדירות המונח – ערך גבוה יותר מקטין את יתרון המסמכים הארוכים.

- עדיין מבוסס על חיפוש מונחים ולא הבנה שפתית.

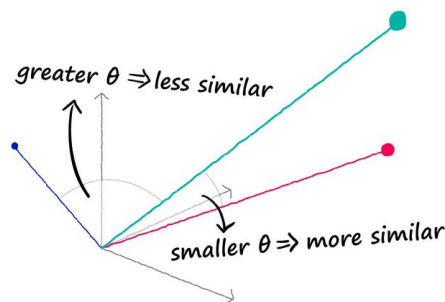
Cosine Similarity

Cosine Similarity (דמיון קוסיני) היא שיטה מתמטית למדידת הדמיון בין שני וקטורים במרחב רב־ממדי, והיא נפוצה במיוחד בתחום של אחזור מידע, עיבוד שפה טבעית (NLP) ו־למידת מכונה.

השיטה מודדת את הזווית בין שני וקטורים (מסמכים) ולא את המרחק ביניהם. ככל שהזווית קטנה יותר – כלומר, הווקטורים "מצביעים" לאותו כיוון – כך הדמיון גבוה יותר.

- ממירים כל **מסמך** ו־**שאלתה** לוקטור (כל תא בווקטור מייצג מילה אחת מהאוצר מילים).
- מחשבים את הזווית בין הווקטורים באמצעות המכפלה הפנימית.
- מנרמלים לפי גודל הווקטור.
- מקבלים ציון בין 0 ל־1 (או 1 ל־1, אם יש ערכים שליליים – נדיר בטקסטים רגילים).

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



- $A \cdot B$ = מכפלה פנימית של הווקטורים
- $|A|$ ו- $|B|$ אורך של כל וקטור (נורמה)

יתרונות:

- אינו רגיש לאורך המסמך – מתבצע נירמול.
- פשוט ומהיר למימוש.
- טוב בהשוואת דמיון בין מסמכים עם מבנה מילים דומה.

חסרונות:

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

שפות

Java

יתרונות:

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

חסרונות:

- דורשת התקנת JVM – לא תמיד נוח בפריסה ללקוחות.
- פחות שליטה נמוכה על חומרה – לא מתאימה למערכות שדורשות ביצועים קריטיים או גישה ישירה לזיכרון.
- מהירות מעט נמוכה לעומת ++C – בעיקר בתהליכים קריטיים לזמן אמת.
- צריכת זיכרון גבוהה יחסית – בגלל הרצת המכונה הווירטואלית.

++C

יתרונות:

- ביצועים גבוהים מאוד – שפת מכונה שמתקמפלת ישירות, מתאימה למערכות זמן אמת.
- שליטה מלאה בזיכרון ובחומרה – ניתן לנהל זיכרון, כתיבה ישירה, עבודה עם pointers.
- מתאימה למשחקים, מערכות הפעלה, רכיבי חומרה, מערכות משובצות.
- תומכת בפרדיגמת OOP וגם בתכנות פרוצדורלי – גמישות מרבית.

חסרונות:

- ניהול זיכרון ידני – דורש אחריות, ויכול לגרום לדליפות ובעיות קשות.
- תחביר מסובך – קשה יותר ללמוד ולתחזק, דורשת ניסיון.
- קידוד ארוך – הרבה קוד נדרש לפעולות פשוטות.
- לא חוצה פלטפורמות בקלות – כל מערכת דורשת הידור שונה.

#C

יתרונות:

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

חסרונות:

- בעבר היה מוגבל ל-Windows – היום NET Core פותר חלקית, אבל עדיין פחות פרוס מ-Java.
- פחות ספריות קוד פתוח ביחס ל-Java – חלק מהכלים הם פורטים מספריות Java

9. תיאור החלופה הנבחרת

סריקת תוכן הקבצים בצורה יעילה

SPIMI (Single-Pass In-Memory Indexing) עם מיקבול בעיבוד

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

נימוק לבחירה:

SPIMI מאפשר בניית אינדקס בצורה ישירה וללא צורך במיון כללי, דבר שמפחית 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 ולבנות בלוקים זמניים במקביל – מה שתורם לקיצור זמן הבנייה של האינדקס ולשיפור הסקלביליות של המערכת. כך אין צורך להחזיק את כל הטרמינולוגיה או מיון ID→term - מה שמיעל את הזיכרון ראשי

שימוש במבנה נתונים לאינדקס

Inverted index לאחר בחינה מעמיקה של מספר חלופות למימוש אינדקס נתונים יעיל בפרויקט, בחרתי בחלופה המבוססת על שילוב בין **Inverted Index** לבין **B+ Tree**, במימוש היברידי מותאם.

במהלך תהליך הבחינה, שקלתי 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) עלול לפגוע בביצועים ככל שכמות המונחים גדלה, ובמקרים מסוימים אף להוביל לזמן חיפוש של $O(n)$. שנית, השיטה מחייבת טעינה של כלל המונחים לזיכרון הראשי, דבר שאינו יעיל כאשר מדובר בכמות גדולה של מונחים ובתמיכה בשפות מרובות.

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

- יכולת לשמור את מרבית העץ בדיסק, ולהעלות רק צמתים רלוונטיים לזיכרון 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 המקנה יציבות ויכולת ביצוע של חיפושים טווחיים (Range Queries).

למרות היתרונות, בשלב ניסוי ראשוני שבו בצעתי חיפוש ישירות מדיסק (העלאת הצמתים הרלוונטיים בלבד), התקבלו זמני תגובה איטיים יחסית במיוחד עבור מונחים שכיחים, עקב מספר רב של פעולות I/O. לכן הוחלט לאמץ **שיטה היברידית**: מבנה העץ (B+ Tree) כולו נטען לזיכרון הראשי, אך במקום שהערכים עצמם (המונחים ורשימות המסמכים) יאוחסנו ישירות בעץ – כל ערך מכיל **offset** המצביע על מיקום המונח בתוך קובץ ה-Inverted Index בדיסק.

גישה זו מאפשרת:

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

- ביצוע קריאה ממוקדת לקובץ האינדקס לפי מיקום מדויק, ובכך נחסכות גישות רבות לקריאה של מידע לא רלוונטי.

- צמצום משמעותי בנפח הנתונים בזיכרון.

בנוסף, העץ נבנה מראש בשיטה של **Bottom-Up**, אשר מאפשרת זמן בנייה של $O(n)$, תוך שימוש מקסימלי בצמתים ומניעת בזבז בזיכרון, בניגוד לשיטת Top-Down אשר דורשת זמן $O(n \log n)$ ועשויה לייצר צמתים חלקיים.

שילוב זה בין **Inverted Index** לדגם יעיל של **B+ Tree** – ובאופן ממוטב תוך הפרדה בין מיקומי מונחים לזיכרון ודיסק – מהווה את הפתרון הנבחר, תוך איזון בין מהירות, חיסכון במשאבים, ויכולת הרחבה עתידית לרבות תמיכה בעדכון, הוספה ומחיקה של נתונים.

דירוג

כחלק 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 **BM25**, **TF-IDF**, ו-**Cosine Similarity**.

TF-IDF הייתה האלטרנטיבה הראשונה שבחנתי. היתרון המרכזי שלה הוא הפשטות והיעילות: היא מחשבת משקל לכל מונח על בסיס שכיחות בתוך המסמך מול שכיחותו בקורפוס כולו. עם זאת, היא לא מתחשבת באורך המסמך, מה שעלול לגרום להעדפה של מסמכים ארוכים שבהם מונח חוזר באופן מקרי. לכן פסלתי אותה כבר בשלב מוקדם.

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

לבסוף הגעתי ל-**BM25**, שהוא שיפור של TF-IDF. התרשמתי מהאופן שבו BM25 מאזנת בין שכיחות מונח, חשיבותו, ואורך המסמך. לדוגמה, אם מילה מסוימת מופיעה פעמיים במסמך קצר מאוד, אבל רק שלוש פעמים במסמך ארוך מאוד – BM25 תעדיף את המסמך הקצר. בעיניי זו תוצאה הגיונית ונכונה יותר, כי ככל הנראה המסמך הקצר מדבר ישירות על המונח, ואילו במסמך הארוך ייתכן שהוא רק הוזכר כבדרך אגב.

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

שפה - java

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

- **ניהול זיכרון:** Java כוללת Garbage Collector אוטומטי, מה שמפחית תקלות ודליפות זיכרון – בניגוד ל-C++ שדורשת ניהול ידני.
- **חוצה פלטפורמות:** Java רצה בכל מערכת עם JVM בלי צורך בהידור מחדש, מה שנותן ניידות גבוהה. ב-C++ חייבים הידור מחדש, וב-C# התמיכה חוצה פלטפורמות עדיין מתפתחת.
- **תחזוקה ונוחות:** Java מאפשרת פיתוח מהיר, מודולרי ונקי, עם קהילת תמיכה ענקית וספריות קוד פתוח רבות.
- **ביצועים:** אולי C++ מהירה יותר בתיאוריה, אבל ביישומים של סריקה ואינדוקס – הביצועים של Java מצוינים בפועל.

10. אפיון המערכת שהוגדרה / מוצעת

דרישות המערכת

סביבת פיתוח:

- **חומרה:** מחשב עם מעבד Core i5, i7, זיכרון RAM של 16GB, ו-250GB שטח אחסון
- **עמדת פיתוח:** Windows 10, 11
- **מערכת ההפעלה:** Windows 10 ומעלה
- **דיסק קשיח:** SSD
- **חיבור לרשת:** לא נדרש לפעילות בסיסית, נדרש רק להתקנה ראשונית
- **תוכנות:** JDK 11 ומעלה, כלי פיתוח JavaFX
- **תוספים טכנולוגיים:** לא נדרשים תוספים מיוחדים

מודול המערכת

תחומים בהם המערכת עוסקת:

- חיפוש תיקיות וקבצים לפי שמות
- חיפוש בתוכן טקסטואלי בקבצים מסוגים שונים (txt, pdf, word, html, md, js 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. **בחירת כוננים לאינדוקס** – המשתמש 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. **פתיחת הקובץ ישירות** מהמערכת.
6. **פתיחת מיקום הקובץ** בסייר הקבצים.

ביצועים עיקריים

המערכת מסוגלת להתמודד עם כמויות גדולות של קבצים ללא ירידה ניכרת בביצועים. זמן התגובה לחיפוש הוא לרוב מידי, כאשר תוצאות מתקבלות תוך שניות ספורות, גם כאשר בסיס הנתונים כולל אלפי קבצים. תהליך האינדוקס הראשוני בנוי כך שיוכל לעבור ביעילות על ספריות וכוננים שלמים, לסרוק את שמות הקבצים ואת תוכנם (כאשר מדובר בקבצי טקסט), ולהכין את המערכת לפעולה שוטפת במהירות יחסית.

המערכת תומכת בפורמטים טקסטואליים נפוצים כגון .txt, .xml, ומתמקדת בקריאה מהירה ואפקטיבית של תכנים מתוך קבצים אלו. כל זאת נעשה תוך שמירה על צריכת משאבים מבוקרת – הן בזיכרון (RAM) והן במעבד (CPU) – כך שהשימוש במערכת אינו מכביד על המחשב גם כאשר מדובר בסביבת עבודה עמוסה. השילוב בין יעילות, מהירות, ועמידות הופך את המערכת לכלי אמין לשימוש יומיומי.

אילוצים

המערכת דורשת סביבת הרצה הכוללת Java ו-JavaFX מותקנים.

11. תיאור הארכיטקטורה

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

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

ארכיטקטורת רשת

המערכת הינה יישום Desktop עצמאי שאינו משתמש ברשת או בתקשורת מרוחקת.

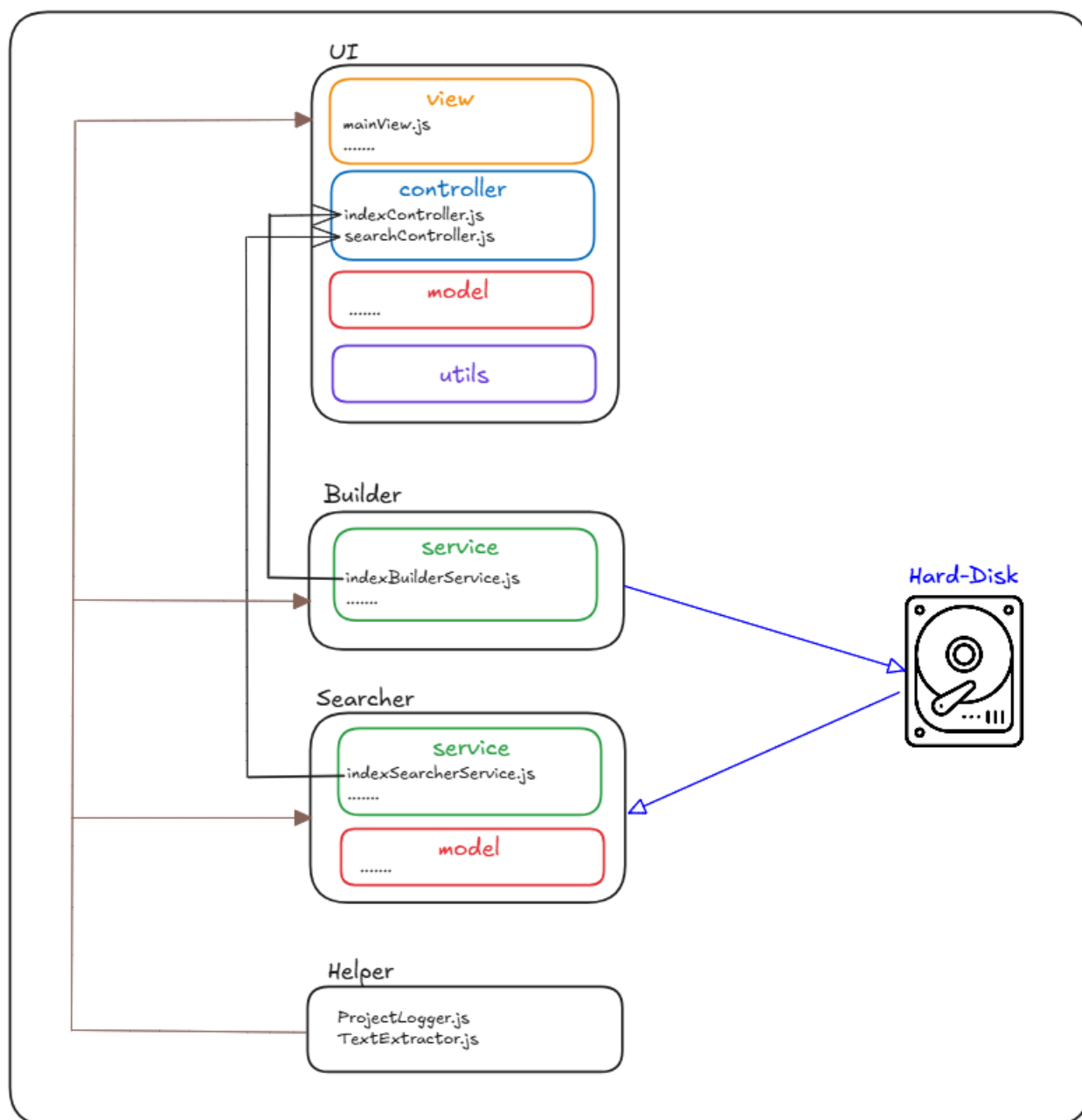
תיאור פרוטוקולי תקשורת

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

שרת-לקוח

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

תרשים ארכיטקטורה



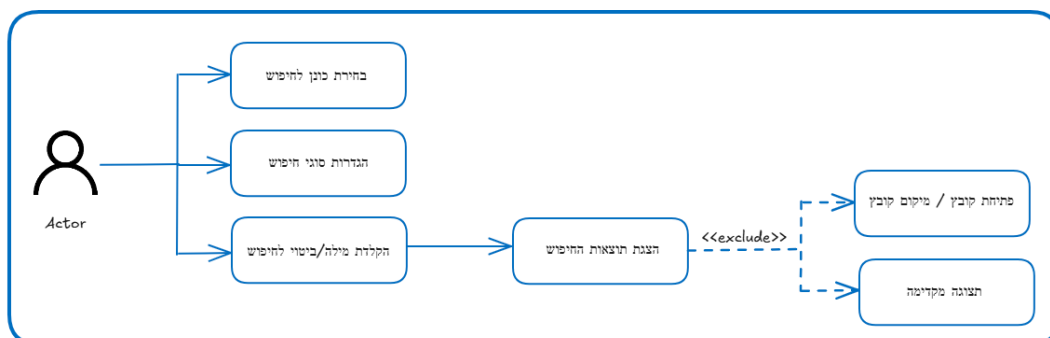
12. תיאור תהליכי אבטחת מידע במערכת

לאחר השלמת תהליך בניית האינדקס, המערכת מיישמת הגנה על קבצי האינדקס. קבצי האינדקס מוגדרים כקבצים מוסתרים (hidden files) במערכת ההפעלה ומוגדרים כקריאה בלבד (read-only) לכל המשתמשים.

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

13. ניתוח ותרשים UML / Use cases של המערכת המוצעת

Diagram class Design



שם מקרה שימוש	בחירת כוון לחיפוש
תיאור	המערכת בונה את האינדקס לחיפוש מהיר בהתאם לבחירת הכוון/נים.
שחקנים	משתמש
תנאי מקדים	-
הזנק	הרצון לחיפוש קובץ / תיקייה
מסלול עיקרי	1. המשתמש בוחר כוון/ים 2. המערכת בונה את האינדקס
תנאי סיום	המערכת סיימה לבנות את האינדקס
תדירות	פעם ביום

שם מקרה שימוש	הגדרות סוגי חיפוש
תיאור	המערכת מגדירה את סוגי החיפושים בהתאם להגדרת המשתמש
שחקנים	משתמש
תנאי מקדים	-
הזנק	הרצון לחיפוש קובץ / תיקייה
מסלול עיקרי	המשתמש בוחר את סוגי החיפוש
תנאי סיום	-
תדירות	בכל חיפוש

שם מקרה שימוש	הקלדת מילה או ביטוי לחיפוש
תיאור	המשתמש מקליד את המילה או הביטוי לחיפוש המערכת מחפשת את הקובץ/ים המכילים את המילה או הביטוי
שחקנים	משתמש
תנאי מקדים	הגדרת כוון לחיפוש
הזנק	הרצון לחיפוש קובץ / תיקייה
מסלול עיקרי	1. המשתמש מקליד מילה או ביטוי לחיפוש 2. המערכת מחפשת את הקובץ/ים בכוון/ים המתאימים בהתאם להגדרות החיפוש
תנאי סיום	המערכת מצאה את הקבצים המתאימים
תדירות	עשרות פעמים ביום

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

שם מקרה שימוש	פתיחת קובץ / מיקום קובץ
תיאור	המערכת פותחת את הקובץ / מיקומו
שחקנים	משתמש
תנאי מקדים	הצגת תוצאות החיפוש
הזנק	המשתמש רוצה לפתוח 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 לפתוח את הקובץ / מיקומו

שם מקרה שימוש	תצוגה מקדימה
תיאור	המערכת מציגה את הקובץ בתצוגה מקדימה
שחקנים	המשתמש
תנאי מקדים	הצגת 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 הציגה את הקובץ בתצוגה מקדימה
תדירות	כשהמשתמש רוצה לראות את הקובץ בתצוגה מקדימה

הקשרים בין היחידות השונות:

הקשרים ברצף בניית האינדקס (צד שמאל):

1. **Scan Files & Folders → Tokenizer**

○ העברת תוכן הקבצים הגולמי לעיבוד

2. **Tokenizer → Normalizer**

○ העברת מילים/טוקנים מפוצלים לנרמול

3. **Normalizer → Indexer**

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

4. **Indexer → Disk**

○ שמירת האינדקסים הבינאריים לדיסק

קשרים ברצף החיפוש (צד ימין):

5. **User Query → Query Processor**

○ קבלת שאילתת חיפוש מהמשתמש

6. **Query Processor → Retriever**

○ העברת שאילתה מעובדת לאחזור

7. **Retriever ← Disk**

○ קריאת נתונים מהאינדקסים שנשמרו בדיסק

8. **Retriever → Scorer + Ranker**

○ העברת תוצאות גולמיות לדירוג וניקוד

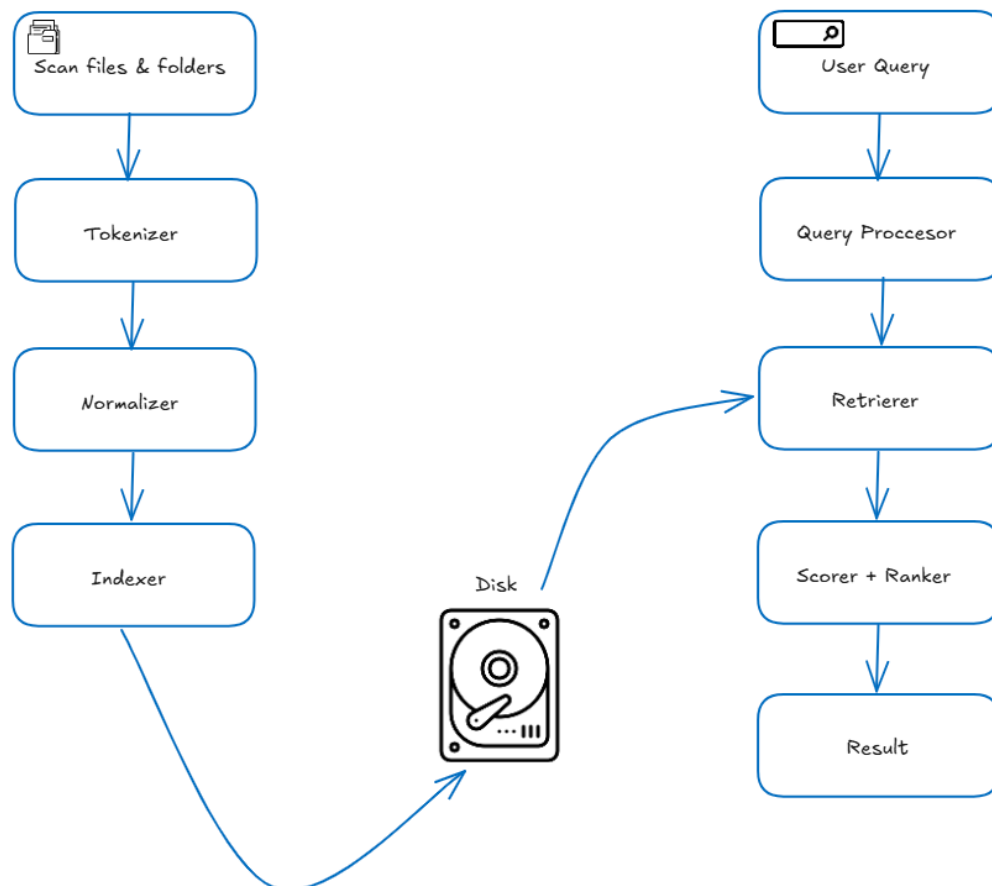
9. **Scorer + Ranker → Result**

○ העברת תוצאות מדורגות למשתמש

הקשר המרכזי:

Disk - מהווה את נקודת החיבור בין שני התהליכים - כאן נשמרים האינדקסים משלב הבנייה ונקראים בשלב החיפוש.

עץ מודולים:



מבני נתונים בהם השתמשתי

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

1. בניית האינדקס (SPIMI)

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

2. העלאת האינדקס לשלב החיפוש

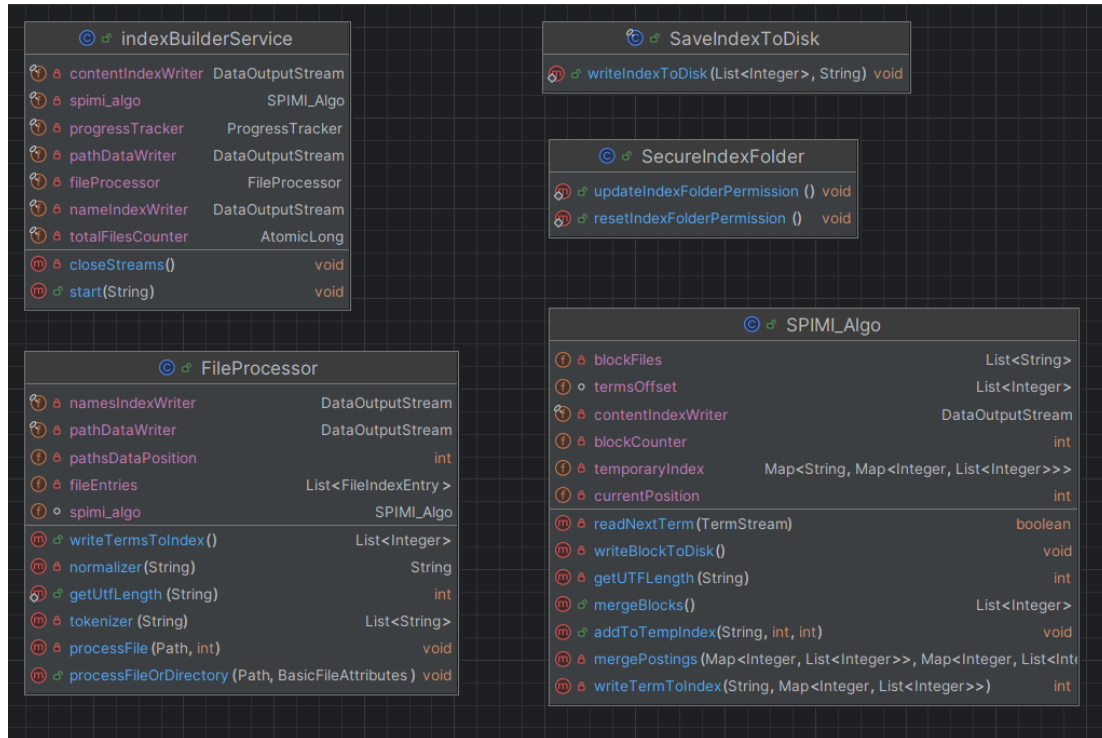
- – **B+ Tree** בעץ נשמרים המיקומים המדויקים (offset) של כל מילה בקובץ האינדקס. כך, במהלך החיפוש ניתן לגשת ישירות למיקום הספציפי של מילה בדיסק, מבלי לקרוא מידע מיותר. בזכות מבנה העץ, שבו כל הצמתים הפנימיים מכילים טווחים רחבים של ערכים וגובהו נמוך (בממוצע 2–3 רמות), מספר הגישות לדיסק מצטמצם למינימום.

3. חישוב יעילות האלגוריתם

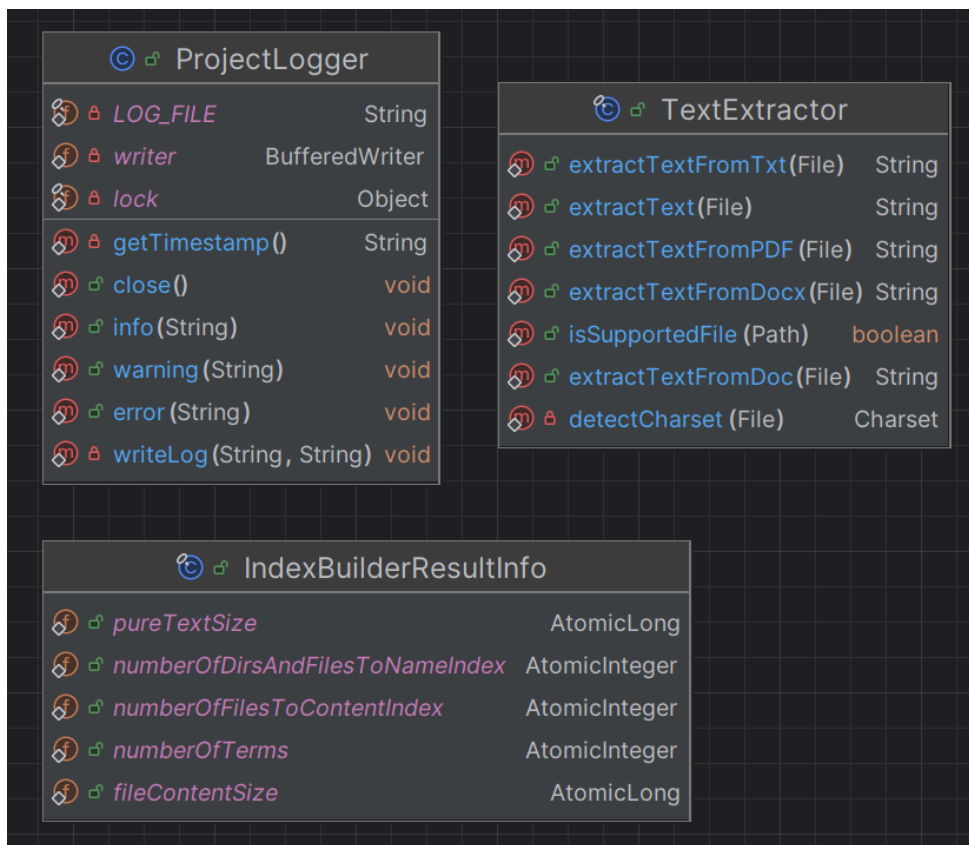
- זמן בניית האינדקס הוא – $O(n)$ נובע מסריקה ליניארית של הקבצים
- זמן בניית ה- $B+ Tree$ הוא – $O(n)$ נובע מגישה הבנייה bottom-up.
- זמן חיפוש של מילה הוא – $O(\log n)$ בזכות מבנה העץ המאורגן, שבו כל חיפוש עובר רק מספר מצומצם של צמתים.

תרשים מחלקות

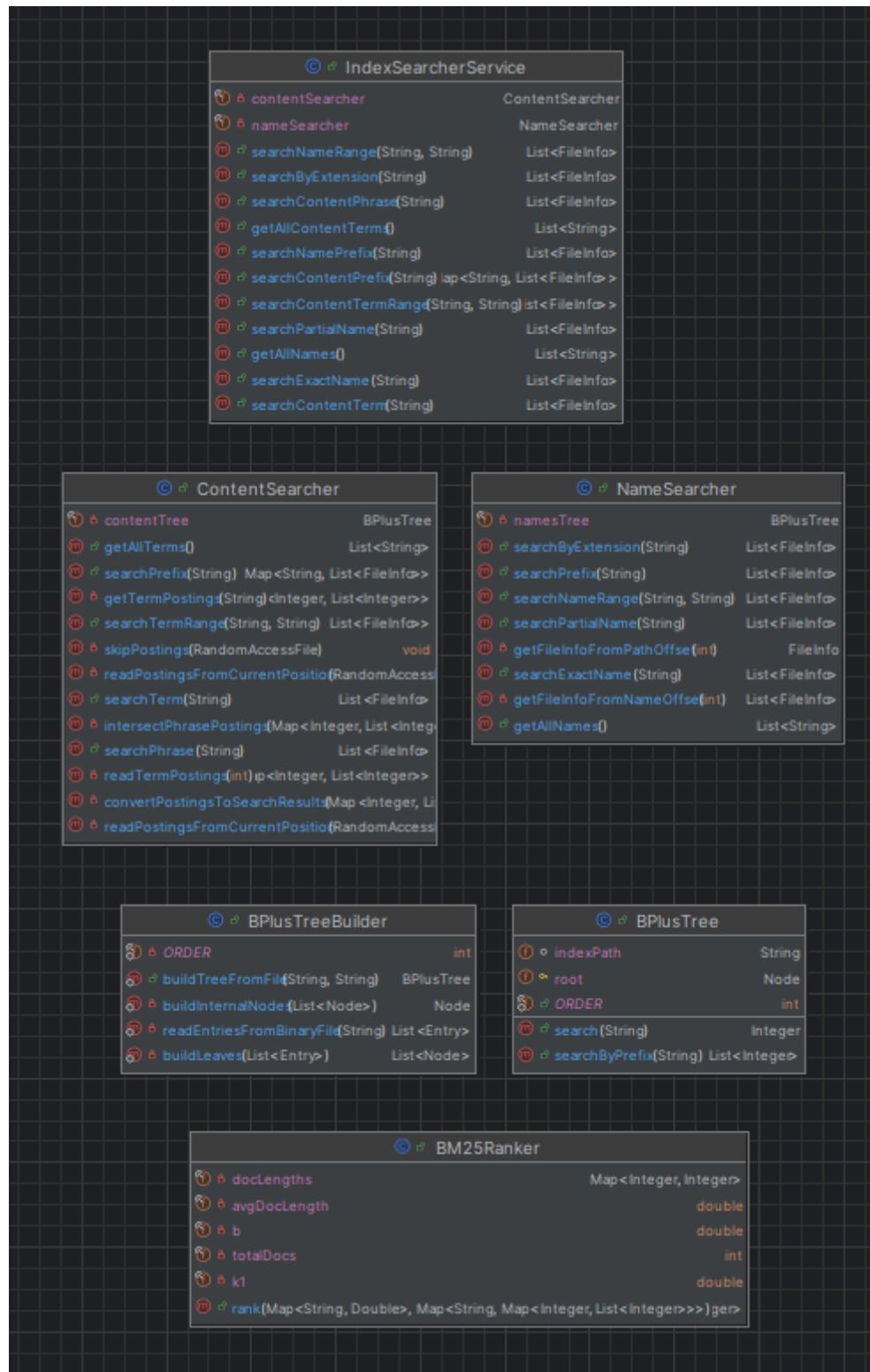
Builder



Helper



Searcher



UI

Utils ○

FileIconUtil
toBufferedImage (Image) BufferedImage
getSystemIcon (File) ImageView

FileTypeUtil
getFileExtension (String) String
getFileTypeDescription (String) String
isPreviewableTextFile (String) boolean
getFileType (String) FileType
isPreviewableImageFile (String) boolean

Controller ○

indexController
model SearchEngine
setSearchPath (String) void
getDrivers () List<String>
getIndexedPaths () List<String>
startIndexing (List<String>) void

SearchController
model SearchEngine
performSearch (String) void
openFileLocation (String) void
getResults () ObservableList<SearchResult>
statusMessageProperty () ReadOnlyStringProperty
openFile (String) void
getStatusMessage () String
getSearchOptions () SearchOptions

model ○

SearchResult	SearchEngine	SearchOptions
name String	searchResults ObservableList<SearchResult>	searchPath StringProperty
path String	instance SearchEngine	maxResults IntegerProperty
size String	isSearching boolean	searchTerm StringProperty
modified String	resultsCount ReadOnlyIntegerWrapper	searchInContent BooleanProperty
created String	indexSearcher IndexSearcherService	fileType ObjectProperty<FileType>
getModified() String	statusMessage ReadOnlyStringWrapper	exactMatch BooleanProperty
getType() String	searchOptions SearchOptions	caseSensitive BooleanProperty
getCreated() String	searchContentTerm(String) st<SearchResult>	setMaxResults(int) void
getSize() String	extractPath(String) String	setFileType(FileType) void
getName() String	getInstance() SearchEngine	isSearchInContent() boolean
getPath() String	getResultsCount() int	searchInContentProperty() BooleanProperty
getFullPath() String	searchNamePrefix(String) List<SearchResult>	resetToDefaults() void
	getSearchResults() ObservableList<SearchResult>	fileTypeProperty() ObjectProperty<FileType>
	isSearching() boolean	searchPathProperty() StringProperty
	convertFileInfoToSearchResult(List<FileInfo>) void	getSearchPath() String
	performSearch(String) Task<Void>	setCaseSensitive(boolean) void
	updateResults(ObservableList<SearchResult>) void	getSearchTerm() String
	formatFileSize(long) String	searchTermProperty() StringProperty
	initializeSearchEngine() void	getMaxResults() int
	resultsCountProperty() ReadOnlyIntegerProperty	maxResultsProperty() IntegerProperty
	searchContentPhrase(String) List<SearchResult>	exactMatchProperty() BooleanProperty
	getStatusMessage() String	setSearchPath(String) void
	reindexDirectory(String) void	setSearchTerm(String) void
	extractFileName(String) String	caseSensitiveProperty() BooleanProperty
	getSearchOptions() SearchOptions	getFileType() FileType
	statusMessageProperty() ReadOnlyStringProperty	isCaseSensitive() boolean
		setSearchInContent(boolean) void
		isExactMatch() boolean
		setExactMatch(boolean) void

View

MainView	SearchPane
mainLayout BorderPane	searchController SearchController
searchPane SearchPane	searchField TextField
contentSplitPane SplitPane	searchPane VBox
statusLabel Label	statusLabel Label
resultsPane ResultsPane	performSearch() void
mainSplitPane SplitPane	bindStatusLabel() void
searchController SearchController	openSettingsWindow() void
previewPane PreviewPane	getPane() VBox
filterPane FilterPane	createSettingsButton() Button
setupEventHandlers() void	
getMainLayout() BorderPane	
ResultsPane	FilterPane
resultsPane BorderPane	filterPane VBox
searchController SearchController	searchController SearchController
resultsTableView TableView<SearchResult>	fileTypeFilter ComboBox<String>
countLabel Label	fileNameRadio RadioButton
createModifiedColumn() Column<String>	contentRadio RadioButton
getResultsTable() TableView<SearchResult>	exactMatchCheck CheckBox
createCreatedColumn() Column<String>	bindToModel(SearchOptions) void
getPane() BorderPane	resetOptions() void
setupStatusBar() void	getPane() VBox
setupResultsTable() void	
createNameIconColumn() Column<SearchResult>	
setupRowClickHandler() void	
createSizeColumn() Column<SearchResult, String>	
bindToModel() void	
createPathColumn() Column<SearchResult, String>	
PreviewPane	
DEFAULT_HEIGHT double	
fileContentPreview TextArea	
spinner ProgressIndicator	
MAX_PREVIEW_SIZE int	
previewPane VBox	
readTextFileContent(String) String	
generatePreviewMessage(SearchResult) String	
getPane() VBox	
showFilePreview(SearchResult) void	

14. רכיבי ממשק

- **שדה חיפוש** - קלט טקסטואלי למילות החיפוש.
- **כפתורי פעולה** -
- **חפש** – מפעיל את מנגנון החיפוש.
- **הגדרות ניתוב** – פותח חלון קופץ לבחירת כוננים שיש לאנדקס.
- **טבלת תוצאות** - תצוגת רשימת הקבצים שנמצאו, כל תוצאה מכילה: שם הקובץ, נתיב, גודל, תאריך שינוי, תאריך יצירה.
- **הגדרות חיפוש** - בחירת אפשרויות לחיפוש (סוג חיפוש, סוג קבצים, חיפוש מדויק).
- **תצוגה מקדימה** - "תצוגה מקדימה של תוכן הקובץ".
- **שורת סטטוס** - מציגה את מספר התוצאות שחזרו.

16. תיאור התוכנה

שפות תכנות:

- Java

סביבת פיתוח:

- IntelliJ IDEA

טכנולוגיות ותשתיות נוספות:

- Maven (לניהול תלויות)
- JavaFX (UI)

17. תיאור מסכים

מסך החיפוש הראשי

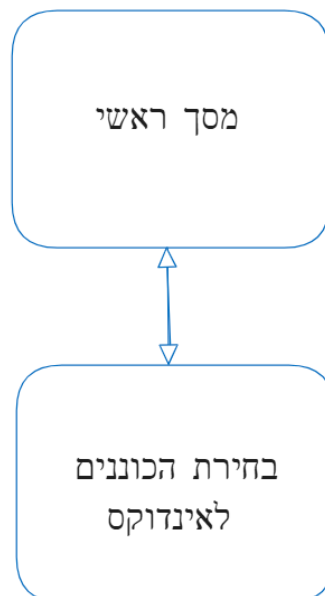
המסך הראשי של המערכת מחולק ל-4 פאנלים עיקריים:

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

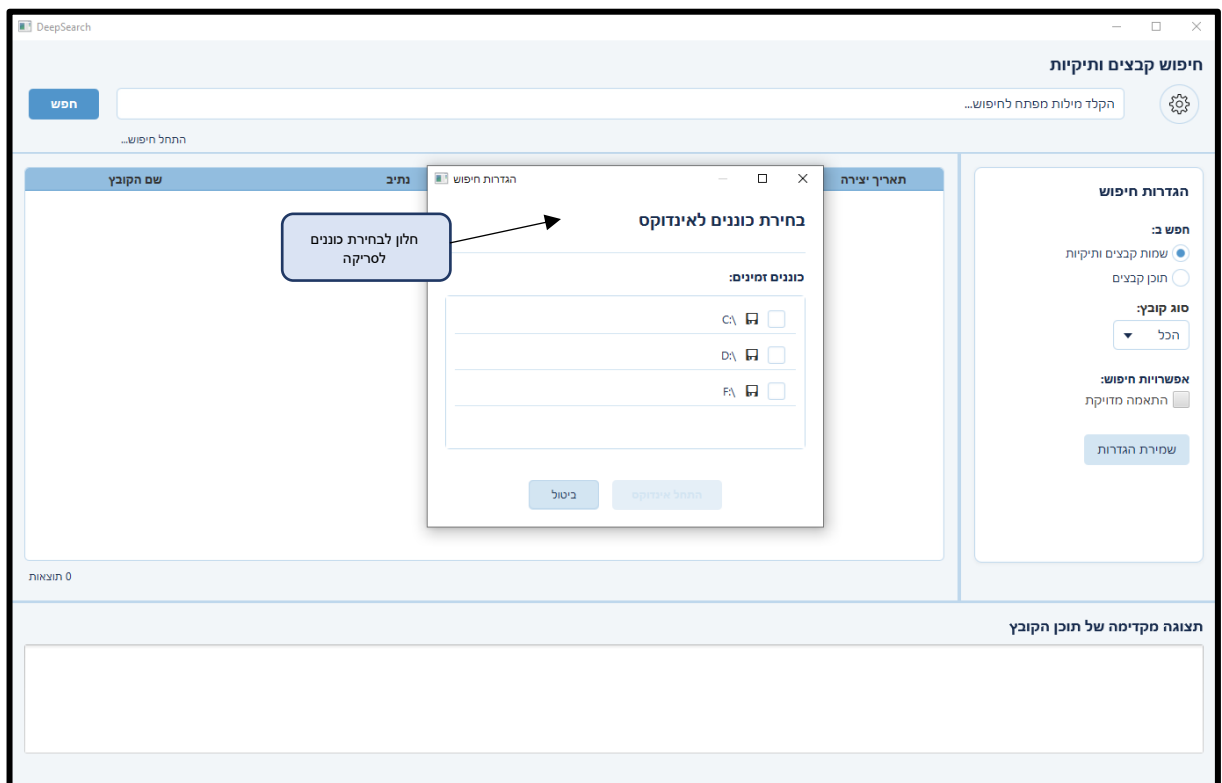
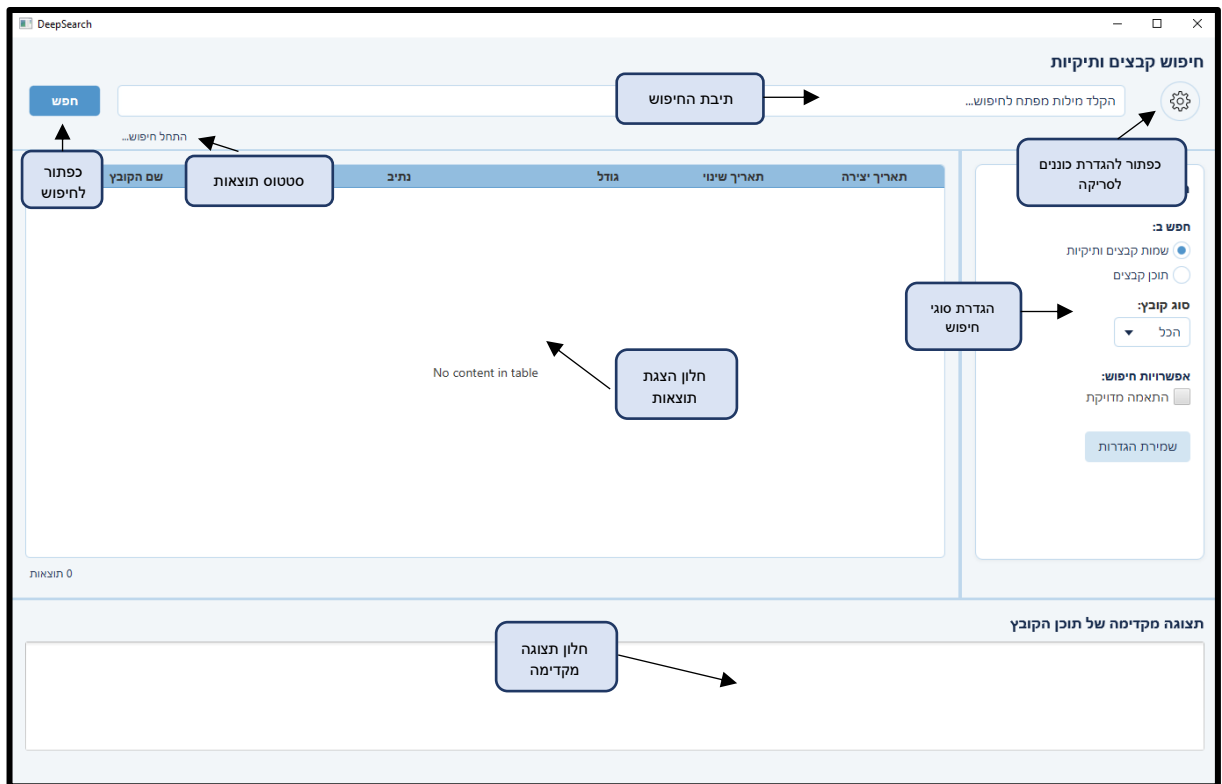
חלון הגדרת נתיב האינדקס ובנייתו

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

18. תרשים מסכים



19. פירוט מסכים



DeepSearch
חיפוש קבצים ותיקיות

dsc

⚙️

לאחר הכנסת מילה/ביטוי לחיפוש ולחיצה על כפתור ה"חפש" מוצגות התוצאות

שם הקובץ	נתיב	גודל	תאריך שינוי	תאריך יצירה
dsc	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	0 KB	22-04-2025 01:15:28	22-04-2025 01:15:27
dsc	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	0 KB	22-04-2025 01:15:27	22-04-2025 01:15:26
DSC_0060-2-364x500.jpg	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	25 KB	12-10-2023 09:55:30	09-11-2023 03:48:53
DSC_0060-2-364x500.jpg	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	25 KB	12-10-2023 09:55:30	09-11-2023 03:50:10
DSC_0060-2-364x500.jpg	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	25 KB	12-10-2023 10:55:30	19-10-2023 02:18:18
DSC_0060-2-364x500.jpg	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	25 KB	12-10-2023 10:55:30	19-10-2023 02:19:34
DSC_0060-2-364x500.jpg	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	25 KB	12-10-2023 09:55:30	16-11-2023 06:32:16
DSC_0060-2-364x500.jpg	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	25 KB	12-10-2023 09:55:30	16-11-2023 06:33:35
DSC_0060-2-364x500.jpg	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	25 KB	12-10-2023 09:55:30	23-11-2023 09:35:32
DSC_0060-2-364x500.jpg	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	25 KB	12-10-2023 09:55:30	23-11-2023 09:37:04
DSC_0060-2-364x500.jpg	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	25 KB	12-10-2023 10:55:30	26-10-2023 01:08:25
DSC_0060-2-364x500.jpg	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	25 KB	12-10-2023 10:55:30	26-10-2023 01:10:02
DSC_0060-2-364x500.jpg	\Users\... \Desktop\קבצים\מנוע חיפוש קבצים\...	25 KB	12-10-2023 09:55:30	09-11-2023 14:02:49

חפשי ב: שמות קבצים ותיקיות / תוכן קבצים / סוב קובצי: הכל

אפשרויות חיפוש:
 התאמה מדויקת ☐

 שמירת הגדרות

תצוגה מקדימה של תוכן הקובץ

חיפוש קבצים ותיקיות

דפוס

נמצאו 6280 תוצאות ב-C:\

שם הקובץ	נתיב	גודל	תאריך שינוי	תאריך יצירה
DSC_0060-2-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	25 KB	12-10-2023 09:55:30	09-11-2023 03:50:10
DSC_0060-2-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	25 KB	12-10-2023 10:55:30	19-10-2023 02:18:18
DSC_0060-2-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	25 KB	12-10-2023 10:55:30	19-10-2023 02:19:34
DSC_0060-2-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	25 KB	12-10-2023 09:55:30	16-11-2023 06:32:16
DSC_0060-2-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	25 KB	12-10-2023 09:55:30	16-11-2023 06:32:16
DSC_0060-2-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	25 KB	12-10-2023 09:55:30	23-11-2023 06:32:16
DSC_0060-2-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	25 KB	12-10-2023 09:55:30	23-11-2023 09:37:04
DSC_0060-2-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	25 KB	12-10-2023 10:55:30	26-10-2023 01:08:25
DSC_0060-2-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	25 KB	12-10-2023 10:55:30	26-10-2023 01:10:02
DSC_0060-2-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	25 KB	12-10-2023 09:55:30	09-11-2023 14:02:49
DSC_0060-2-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	25 KB	12-10-2023 09:55:30	09-11-2023 14:03:50
DSC_0092b-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	17 KB	12-10-2023 09:55:20	09-11-2023 03:48:53
DSC_0092b-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	17 KB	12-10-2023 09:55:20	09-11-2023 03:50:10
DSC_0092b-364x500.jpg	D:\Users\רועי\Desktop\חדשה\חדשים\עיוני... D:\Users\רועי\Desktop\חדשה\חדשים\עיוני...	17 KB	12-10-2023 10:55:30	10-11-2023 03:19:18

סינון התוצאות

הדרות חיפוש

חפש ב:

- ☒ שמות קבצים ותיקיות
- ☐ תוכן קבצים

סוג קובץ:

תמונה ▼

אפשרויות חיפוש:

☐ התאמה מדויקת

שמירת הדרות

תצוגה מקדימה של תוכן הקובץ

תצוגה מקדימה של
הקובץ המכיל את
ביטויי החיפוש

לחיצה ימנית על תוצאת החיפוש
לבחירת האפשרויות:
פתיחה של הקובץ
ופתיחת מיקום הקובץ

20. קוד התוכנית

הקוד מבצע סריקה רקורסיבית של תיקיות וקבצים מהספרייה הראשית, תוך עיבוד בטוח של כל רכיב. בנפרד, מבלי לטעון את כל התוכן לזיכרון בבת אחת, 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<>() { new *
    @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 *
    // חלוקה לפי רווחים
    String[] rawTokens = text.split(regex: "\\s+");
    return Arrays.asList(rawTokens);
}

private String normalize(String token) { 1 usage new *
    // הסרה של תווים לא רצויים (פיסוק, תווים מיוחדים)
    return token.toLowerCase()
        .replaceAll(regex: "[\\p{Punct}]+", replacement: "") // הסרה מהתחלה
        .replaceAll(regex: "[\\p{Punct}]+$", replacement: ""); // הסרה מהסוף
}
```

```
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);
    }
}
```

בניית האינדקס ושמירתו בדיסק:

```
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, 0);

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

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

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


21. תיאור מסד נתונים

במקום להסתמך על מסד נתונים מסורתי, בחרתי ליישם ארכיטקטורה של מנוע חיפוש עצמאי שמנהל את הנתונים ברמה נמוכה.

הפתרון שפיתחתי מממש ניהול נתונים בעזרת מבני נתונים מותאמים אישית ואלגוריתמי אינדוקס מתקדמים. גישה זו מעניקה לי שליטה מלאה על אופן ארגון הנתונים, מאפשרת אופטימיזציה לביצועים ספציפיים של חיפוש טקסט, ומבטלת תלות בתוכנות חיצוניות.

מבנה קבצי הנתונים שיצרתי

- **paths.dat** - קובץ נתיבים רציף המכיל את כל נתיבי הקבצים והתיקיות במערכת. כל נתיב נשמר בפורמט UTF-8 ומזהה המסמך הוא ה-offset של הנתיב בקובץ זה.
 - **name_index.idx** ו- **content_index.idx** - אינדוקס היפוך (inverted index) המאורגן בסגמנטים משתנים. כל סגמנט מכיל מילה אחת עם רשימת כל המסמכים המכילים אותה, כולל תדירות המילה במסמך ורשימת מיקומיה המדויקים בטקסט.
 - **tree_name_index.idx** ו- **tree_content_index.idx** - קבצי עצי B+ המכילים רשימות offset לתחילת כל סגמנט באינדוקסים הראשיים, המאפשרים חיפוש binary search בסיבוכיות $O(\log n)$ וטעינת עץ לזיכרון בסיבוכיות $O(n)$.
- העיצוב בסגמנטים משתנים מאפשר ניצול אופטימלי של דיסק, קריאות רציפות יעילות, וגמישות בגדלי הנתונים לפי התוכן בפועל.

22. מדריך למשתמש

שלום לך משתמש יקר!

הפעלה ראשונית

- התקן את מנוע החיפוש במחשב שלך והפעל אותו
- ודא שיש לך הרשאות קריאה לתיקיות שברצונך לחפש בהן
- בהפעלה הראשונה, תצטרך להגדיר את נתיב תיקיית האינדקסים

שימוש במערכת

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

- ✓ **פאנל חיפוש** - הזן 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 ההגדרות:
 - a. שמות קבצים ותיקיות
 - b. תוכן קבצים (טקסט)
2. **הקלד** את מילת החיפוש בשדה החיפוש
3. **לחץ** על כפתור "חפש"
4. **בחר** קובץ מרשימת התוצאות לצפייה בפרטים נוספים

הגדרות מתקדמות

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

23. בדיקות והערכה

זמן בניית האינדקס

זמן סריקת הקבצים ועיבודם – חסכון של חצי מהזמן.

לפני ניצול המקביליות:

```
Run index.Main x
"C:\Program Files\Java\jdk-23\bin\java.exe" ...
Permissions reset successfully.
מערכת אינדוקס וחפוש קבצים
=====
C:\ : הכנס נתיב לסריקה
C:\ : מתחיל סריקה
זמן סריקת הקבצים ועיבודם: 30.44 דקות
```

אחרי ניצול המקביליות:

```
Run index.parallel.Main x
"C:\Program Files\Java\jdk-23\bin\java.exe" ...
Permissions reset successfully.
מערכת אינדוקס וחפוש קבצים
=====
C:\ : הכנס נתיב לסריקה
C:\ : מתחיל סריקה
זמן סריקת הקבצים ועיבודם: 15.36 דקות
```

בזמן בניית האינדקס - הזיכרון RAM לא עולה על GB3

שם	מצב	CPU	זיכרון	דיסק	רשת	צריכת חשמל	מגמת שימוש ב...
IntelliJ IDEA Ultimate Edition		0.3%	2,487.1 MB	0 MB לשנ...	0 Mbps	נמוכה מאוד	נמוכה מאוד
OpenJDK Platform binary		0%	4.7 MB	0 MB לשנ...	0 Mbps	נמוכה מאוד	נמוכה מאוד
OpenJDK Platform binary		0%	1.9 MB	0 MB לשנ...	0 Mbps	נמוכה מאוד	נמוכה מאוד

חיפוש בתוכן קבצי טקסט שונים:

חיפוש בתוכן קובץ json :

The screenshot shows the DeepSearch application interface. At the top, there's a search bar and a 'חיפוש' (Search) button. Below it, a table displays search results. The table has columns for 'שם הקובץ' (File Name), 'נתיב' (Path), 'גודל' (Size), 'תאריך שינוי' (Last Modified), and 'תאריך יצירה' (Created). The results show two files: 'Driver.xml' and 'Application.json'. The 'Application.json' file is selected, and its details are shown in a sidebar on the right. The sidebar includes a 'חיפוש קבצים ותיקיות' (Search files and folders) section with a search bar and a 'הגדרות חיפוש' (Search settings) section with various options. Below the sidebar, there's a 'תצוגה מקדימה של תוכן הקובץ' (Preview of file content) section showing the JSON content of the selected file. The JSON content is displayed in a code editor, and the 'Name' field is highlighted with a red box, showing 'After Effects CC'.

חיפוש בתוכן קובץ PDF עם תמונות:

חיפוש קבצים ותקיות

נציג יקר

הגדרות חיפוש

חשב ב:

שמות קבצים ותקיות

תוכן קבצים

סוג קובץ

הכל

אפשרויות חיפוש:

התאמה מדויקת

שמירת הגדרות

תוצאות מקדימה של תוכן הקובץ

תוצאות

הקובץ נמצא בתיקייה: C:\Users\user\Documents\...
שם הקובץ: תוכן אאאאאאאא
גודל: 1.4 MB
תאריך שינוי: 18-03-2020 04:17:20
תאריך יצירה: 13-05-2020 04:40:49

הקובץ pdf(z) הוא תוכן אאאאאאאא

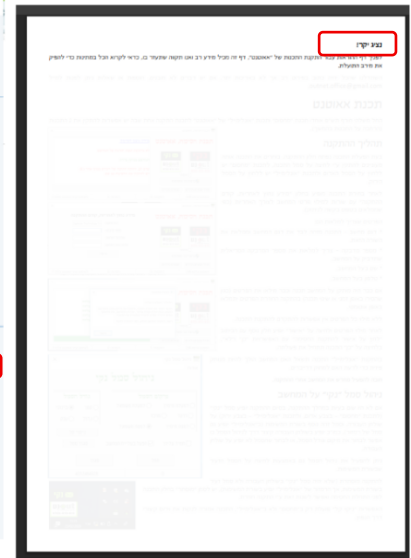
0 תוצאות

תוצאות מקדימה של תוכן הקובץ

תוצאות

הקובץ נמצא בתיקייה: C:\Users\user\Documents\...
שם הקובץ: תוכן אאאאאאאא
גודל: 1.4 MB
תאריך שינוי: 18-03-2020 04:17:20
תאריך יצירה: 13-05-2020 04:40:49

הקובץ pdf(z) הוא תוכן אאאאאאאא



תוצאות מהירות תוך פחות משניה

```

זמן ביצוע החיפוש (searchContentPhrase): 7 מילישניות
מס תוצאות: 2

מחפש באנדקס: 'the'
זמן ביצוע החיפוש (searchContentPhrase): 3 מילישניות
מס תוצאות: 6

מחפש באנדקס: 'נציג יקר'
זמן ביצוע החיפוש (searchContentPhrase): 4 מילישניות
מס תוצאות: 1

מחפש באנדקס: 'נציג יקר'
זמן ביצוע החיפוש (searchContentPhrase): 5 מילישניות
מס תוצאות: 1

מחפש באנדקס: 'נציג יקר'
זמן ביצוע החיפוש (searchContentPhrase): 5 מילישניות
מס תוצאות: 1

מחפש באנדקס: 'נציג יקר'
זמן ביצוע החיפוש (searchContentPhrase): 5 מילישניות
מס תוצאות: 1

```

זמן חיפוש למילה נפוצה (3 שניות):

'ds' : מחפש באנדקס
מילישניות 3180 זמן ביצוע החיפוש
מס תוצאות: 6283

ביצועי קריאה מהדיסק:

קריאה ישירה - DataInputStream

מול

קריאה עם באפר BufferedInputStream

```

20 public static BPlusTree buildTreeFromFile(String filePath, String indexPath) throws IOException {
21     return new BPlusTree();
22 }
23
24 // מחקוקץ חבילת offsets קריאת
25 @
26 private static List<BPlusTree.Entry> readEntriesFromBinaryFile(String filePath) throws IOException {
27     List<BPlusTree.Entry> entries = new ArrayList<>();
28
29     try (DataInputStream dis = new DataInputStream(new FileInputStream(filePath))) {
30         while (dis.available() >= 4) {
31             int offset = dis.readInt();
32             entries.add(new BPlusTree.Entry(offset));
33         }
34     }
35     return entries;
36 }

```

Run search.UI.Main x

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

Time taken to read entries: 25.793885 seconds

```

23
24 // מחקוקץ חבילת offsets קריאת
25 @
26 private static List<BPlusTree.Entry> readEntriesFromBinaryFile(String filePath) throws IOException {
27     List<BPlusTree.Entry> entries = new ArrayList<>();
28
29     try (DataInputStream dis = new DataInputStream(
30         new BufferedInputStream(new FileInputStream(filePath)))) {
31         while (true) {
32             try {
33                 int offset = dis.readInt();
34                 entries.add(new BPlusTree.Entry(offset));
35             } catch (EOFException e) {
36                 break;
37             }
38         }
39     }
40     return entries;
41 }

```

Run search.UI.Main x

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

Time taken to read entries: 0.2881306 seconds

24. ניתוח יעילות

המערכת תוכננה לאופטימיזציה של שני פרמטרי היעילות העיקריים: סיבוכיות זמן מינימלית וניצול זיכרון יעיל.

סיבוכיות מקום

בזיכרון הראשי:

במקום לטעון את כל המונחים עצמם לזיכרון, המערכת טוענת רק **מצביעים** (offsets) למיקום המונחים בקובץ. כלומר, במקום לשמור את התוכן האמיתי של המילה "מחשב" בזיכרון, נשמר רק מספר של **4 בתים** שמצביע 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** - "מיקרופרוצסור": 26 בתים ← 4 בתים (offset) - חיסכון 85%

חיסכון כולל בזיכרון: בהתבסס על אורך ממוצע של 5.5 תווים למילה בעברית (11 בתים ב-UTF-8), במקום לטעון מיליוני מונחים של 11 בתים כל אחד לזיכרון, נטענים רק מיקומיהם של 4 בתים כל אחד - חיסכון ממוצע של **64%** בתפיסת זיכרון.

בדיסק קשיח:

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

דוגמה: המונח "a" 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
דוגמה: 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- $offset$. הניווט בעץ כולל: קריאה של המילה מהדיסק, השוואה, ואז ניווט לצומת הבא.
זמן קריאה מהדיסק (SSD): 50–100 מיקרושניות

קיבולת 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 המקורית (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 שביצעתי - חיפוש בינארי בתוך כל צומת:

במקום לסרוק ליניארית את הערכים בכל צומת, יישמתי חיפוש בינארי:

הסיבוכיות המשופרת: $O(h \times \log k) = O(\log_k(n) \times \log k) = O(\log n)$

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

- **רמה 1:** $\log_2(200) \approx 8$ השוואות
- **רמה 2:** $\log_2(200) \approx 8$ השוואות
- **רמה 3:** $\log_2(200) \approx 8$ השוואות
- **סה"כ:** כ-24 השוואות (במקום 1600!)

מערכת עם 8 מיליון מונחים נדרשות רק **כ-24 השוואות** וזמן חיפוש של **פחות ממילישנייה** (בהתחשב בזמן קריאה מ-SSD של 50–100 מיקרושניות) לחיפוש מילה בודדת!

מכיוון ש- k הוא **קבוע** (מספר קבוע של ערכים בצומת), הוא לא משמעותי לחישוב סיבוכיות הזמן הכללית. הסיבוכיות הסופית היא $O(\log n)$ - לוגריתמית במספר המונחים במערכת.

25. מסקנות

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

התכנון הקפדני של מבנה הנתונים והחלטת השמירה בפורמט בינארי הוכיחו את עצמן כקריטיות להצלחת הפרויקט. הבחירה בעצי $B+B$ עם חיפוש בינארי בתוך כל צומת הביאה לשיפור דרמטי בביצועים - מ-600 השוואות ל-24 השוואות בלבד למערכת של 8 מיליון מונחים.

במהלך הפיתוח התמודדתי עם אתגרים טכניים מורכבים, במיוחד בנושא ניהול הזיכרון ואופטימיזציות גישות הדיסק. אתגר נוסף היה הקושי בדיבוג קבצי האינדקס עקב הפורמט הבינארי, שדרש פיתוח כלים נוספים לבדיקה ואימות של תקינות הנתונים. הפתרון בשמירת offsets במקום המונחים עצמם הביא לחיסכון של 64% בזיכרון, תוך שמירה על זמני חיפוש מהירים.

הפרויקט לימד אותי על חשיבות האיזון בין סיבוכיות זמן לסיבוכיות מקום (בזיכרון RAM ובדיסק קשיח), ועל כך שעיצוב נכון של מבני נתונים יכול להביא לשיפור משמעותי בביצועים מבלי לפגוע בפונקציונליות.

26. פיתוחים עתידיים

המערכת הנוכחית 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 חלקי מילים, סיומות 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
- **דחיסת האינדקסים** - יישום אלגוריתמי דחיסה מתקדמים 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/>

מבני 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

[computing/?utm_source=chatgpt.com](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/scalabil](https://handbook.eng.kempnerinstitute.harvard.edu/s5_ai_scaling_and_engineering/scalability/introduction_to_parallel_computing.html?utm_source=chatgpt.com)

[ity/introduction_to_parallel_computing.html?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)