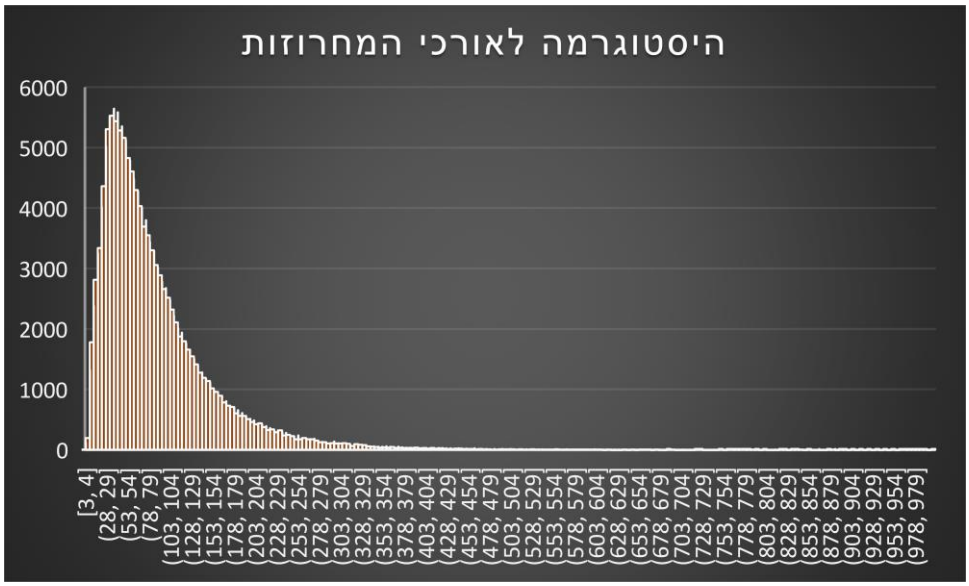
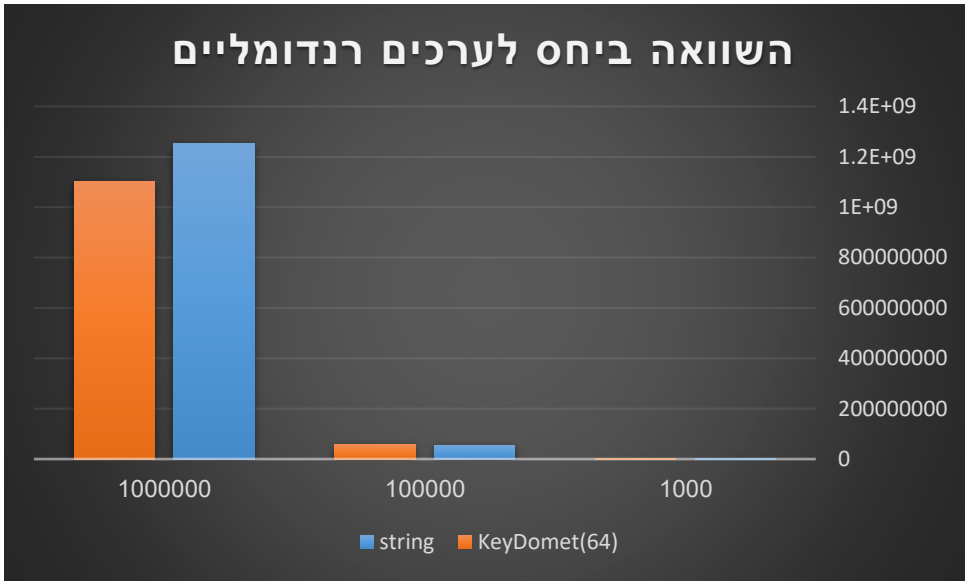
ניתוח ביצועים - KeyDomet

מסמך זה מרכז תוצאות ניתוח ביצועים של הטיפוס KeyDomet מול טיפוסים קיימים אחרים (string, wstring...), השוואה בין גדלים שונים עבור הקידומת (16,32,64,128), ומקרים שונים (בדיקה על ערכים רנדומליים, בדיקה על ערכים מתוך מאגר מילים, בדיקות המכילות רק פעולות חיפוש, בדיקות המכילות פעולות מגוונות ומקרים נוספים).
נחלק את התוצאות לפי סוגי המקרים שנבדקו.
אנחנו צופים שהשימוש ב KeyDomet יספק ביצועים טובים יותר ביחס לאלטרנטיבות הקיימות במקרי הבדיקה שנציג כאן.
נתחיל מהתייחסות להשפעת השימוש ב KeyDomet מבחינת זמני ההרצה.
כל נקודה מתייחסת לבדיקה אותה ביצענו, לכל בדיקה מצורף הסבר על הבדיקה עצמה, גרף המציג את ההבדלים בצורה ויזואלית וטבלה המכילה את הערכים אשר מהם יצרנו את הגרף.
עבור הבדיקות אשר משתמשות בערכים ממאגר, מדובר במחרוזות שלקחנו מ urban dictionary, מספר רב של מחרוזות באורכים שונים, ובהן השתמשנו בצורות שונות עליהן נפרט בהמשך.

סטטיסטיקות עבור מאגר המחרוזות:
אורך ממוצע של מחרוזת - 88.99909 תווים.
אורך מקסימלי למחרוזת - 29,931 תווים.
אורך מינימלי למחרוזת - 3 תווים.
חציון אורכי המחרוזות - 70.



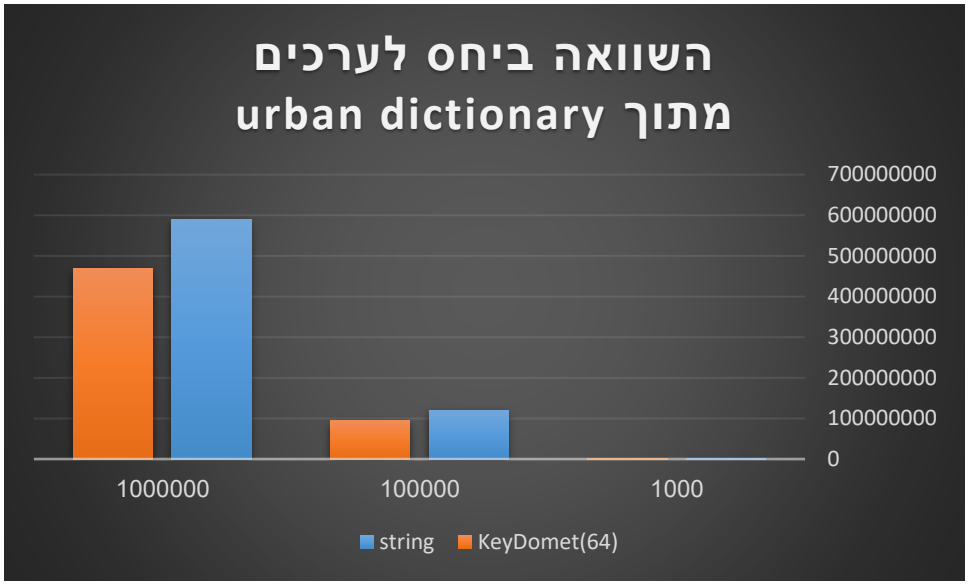
- **<u>BM generated</u>** - השוואה בין ביצועי string ל- keydomet על ערכים רנדומליים,
ציפינו לא לקבל שיפור משמעותי מכיוון שהיכולות של ה keydomet באות לידי ביטוי כאשר הסיכויים לקבלת קידומת זהה עבור שתי מחרוזות שונות הוא נמוך, ישנו סיכוי גדול יותר לקבל קידומות זהות עבור ערכים רנדומליים.
ואכן ניתן לראות שישנו שיפור קל מאוד רק במספר גדול של חיפושים (1,000,000).



\

	1000	100,000	1,000,000
KeyDomet(64)	151026 ns	59445545 ns	**1102429048 ns**
String	**131630 ns**	**55326472 ns**	1252600012 ns
Improvement	14.73%	7.44%	11.98%

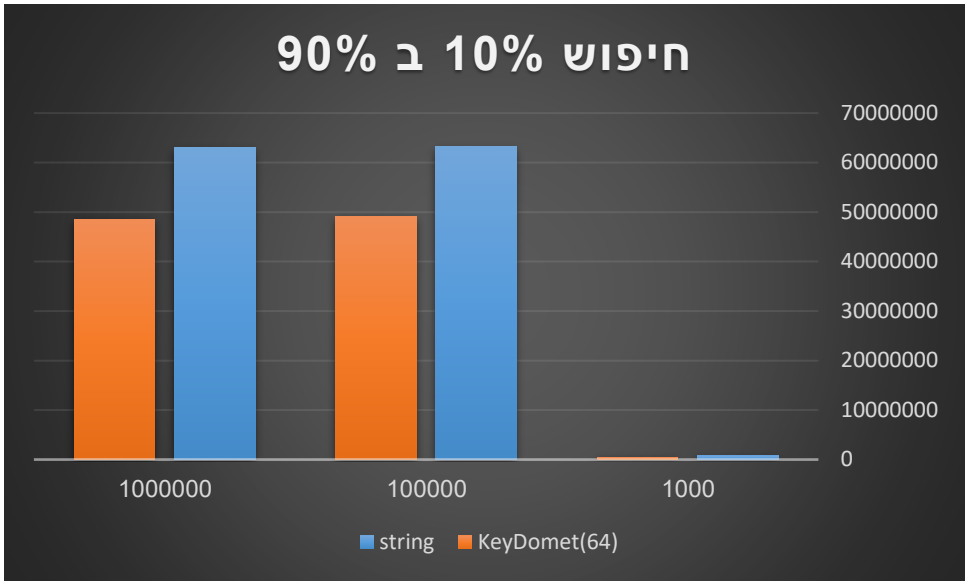
- **<u>BM parsed data</u>** - השוואה בין ביצועי string ל- keydomet על ערכים מתוך urban dictionary. כלומר, השוואה על מחרוזות בעלות משמעות.
כאן הסיכוי לקבלת קידומת זהה עבור מחרוזות שונות הוא נמוך, שכן כל מחרוזת הינה בעלת משמעות אמתית ואינה אוסף של תווים.
כאן ציפינו לקבל שיפור, ואכן ניתן לראות זאת בתוצאות.



	1000	100,000	1,000,000
KeyDomet(64)	**557267 ns**	**94792501 ns**	**469062123 ns**
String	710273 ns	119738226 ns	589663424 ns
Improvement	21.54%	20.83%	20.45%

ניתן לראות עקביות בשיפור, בערך 20% שיפור ל KeyDomet(64) ביחס ל String במקרה זה.

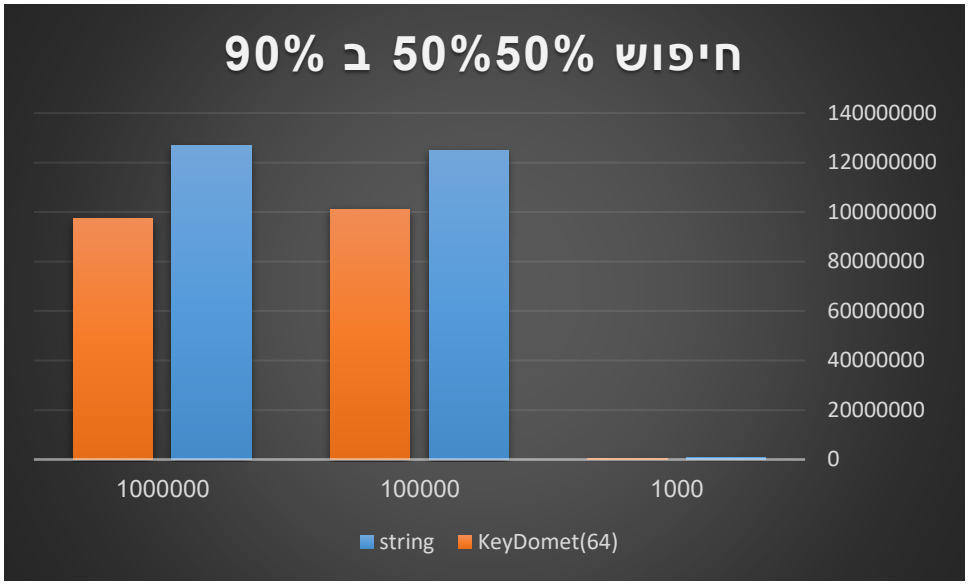
- **<u>90% parsed data 10% unparsed</u>** - השוואה בין ביצועי string ל- keydomet על ערכים מתוך urban dictionary, ההבדל הוא שכעת אנחנו יוצרים set שמכיל 90% מהמחרוזות ומחפשים בתוכו את ה 10% הנותרות, כלומר כל החיפושים אמורים לא להצליח.
אנחנו מצפים לשיפור בביצועים של keydomet, השימוש ב keydomet אמור למנוע את הבאת כל המידע מהזיכרון, החיפוש צריך להידחות בגלל קידומות שונות.
בשימוש ב string בכל חיפוש נביא את המחרוזות מהזיכרון בשביל לראות בסופו של דבר שהיא אינה מתאימה, וזה פוגע מאוד בביצועים.



	1000	100,000	1,000,000
KeyDomet(64)	**508582 ns**	**49142092 ns**	**48559609 ns**
String	768020 ns	63183668 ns	62995183 ns
Improvement	33.78%	22.22%	22.91%

ניתן לראות כי קיבלנו שיפור גדול יותר מהמקרה הקודם, זה צפוי בגלל שבמקרה זה כל החיפושים צריכים להיכשל, והשימוש ב- keydomet נותן את השיפור הכי משמעותי כאשר הוא מונע חיפוש שהיה עתיד להיכשל.

- **<u>90% parsed data 50%50% parsed unparsed</u>** - בדומה לשתי הבדיקות הקודמות, כעת נבנה set המכיל 90% מהמחרוזות ונחפש בו מחרוזות כאשר מחצית מהן נמצאות בו ומחצית מהן לא.
נצפה לקבל שיפור בביצועים של KeyDomet ביחס ל String, שיפור בין מה שקיבלנו בשתי הבדיקות האחרונות.

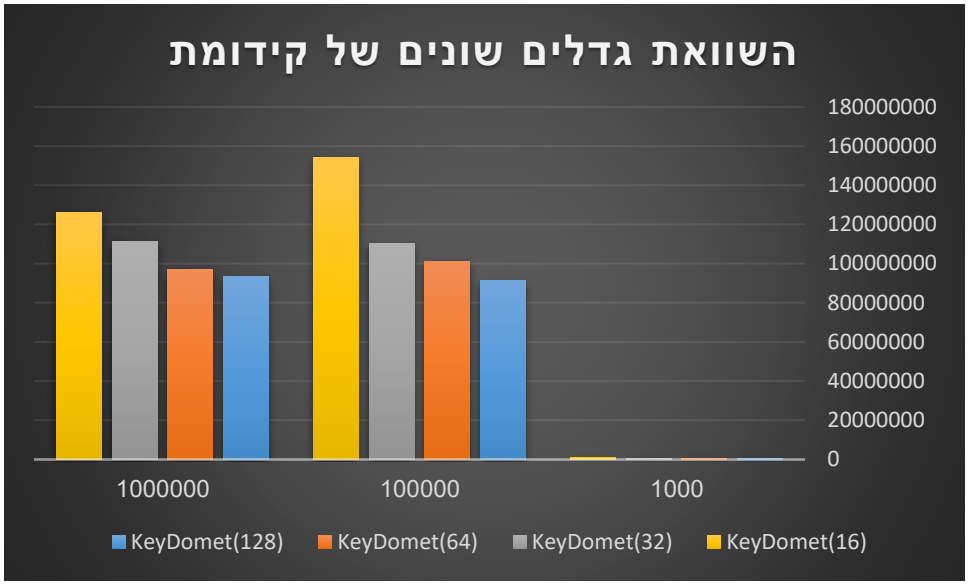


	1000	100,000	1,000,000
KeyDomet(64)	**578380 ns**	**101185316 ns**	**97198322 ns**
String	762251 ns	125020000 ns	126744573 ns
Improvement	24.12%	19.06%	23.31%

- **<u>90% parsed data 50%50% parsed unparsed different size of KeyDomet</u> -**

המקרה הקודם אותו בדקנו נותן ייצוג טוב ומציאותי לשימוש בחיפושים, נשתמש במקרה זה על מנת לבצע השוואה לגדלים שונים של קידומות במימוש של KeyDomet.

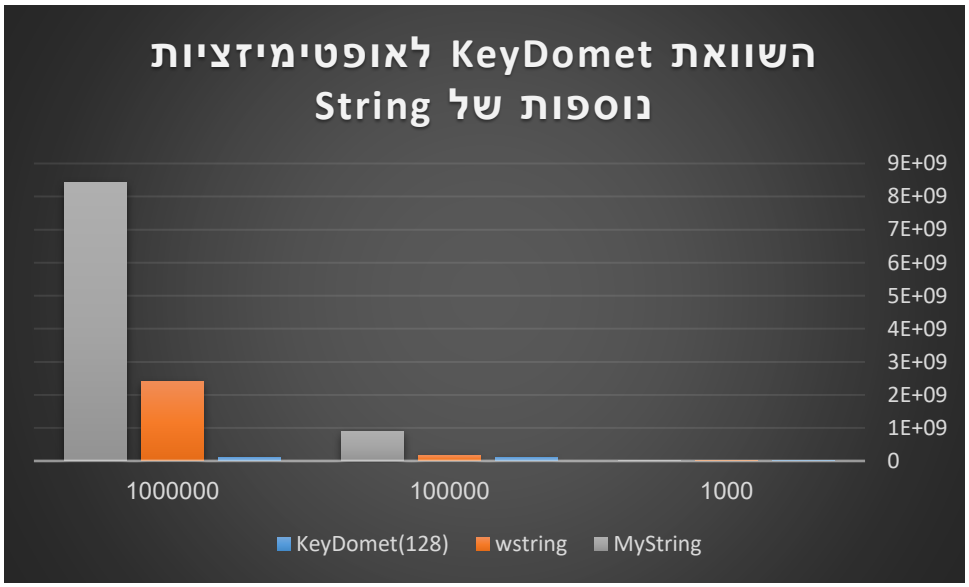
לגדלים שונים יתרונות וחסרונות, ככל שהקידומת גדולה יותר ניתן לשמור יותר מידע על המחרוזות וליצור פונקציה בין מחרוזת לקידומות שיש לה פחות התנגשויות, אך עם זאת, קידומת גדולה יכולה גם לפגוע בביצועים.



	1000	100,000	1,000,000
KeyDomet(16)	809328 ns	154006748 ns	126170602 ns
KeyDomet(32)	611447 ns	110028520 ns	111499157 ns
KeyDomet(64)	578380 ns	101185316 ns	97198322 ns
KeyDomet(128)	**537746 ns**	**91157755 ns**	**93365044 ns**
Improvement KeyDomet(64) - KeyDomet(128)	7.02%	9.91%	3.94%

ניתן לראות שהשיפור המשמעותי ביותר מתקבל עבור **KeyDomet(128)**, כלומר המימוש של KeyDomet שבו גודל הקידומת הנשמרת עבור כל מחרוזת היא 128bit. המימוש של KeyDomet(64) בעל תוצאות טובות גם הוא, השיפור המתקבל בזמנים עבור KeyDomet(128) בא על חשבון הרעה מבחינת זיכרון (נראה בהמשך המסמך), ולכן נמשיך לעבוד עם KeyDomet(64) בתור המודל האופטימלי.

- **<u>90% parsed data 50%50% parsed unparsed different types</u>** - אחרי שהבנו שהמימוש של KeyDomet אכן נותן שיפור ביצועים ביחס ל String, ובפרט כי השיפור של KeyDomet(128) נותן את השיפור הטוב ביותר, כעת נציג את המימוש מול אופטימיזציות של String, נרצה לבדוק האם KeyDomet נותן לנו ביצועים טובים יותר משיפורים הקיימים ל String.
נשווה את הביצועים של KeyDomet לטיפוס wstring (אופטימיזציה קיימת ל String) ולטיפוס MyString (מחלקה שכתבנו, מטרתה לאפשר לנו למדוד ביצועים של String ללא האופטימיזציה SSO שקיימת במימוש הסטנדרטי של String, על מנת להסיר את האופטימיזציה כתבנו מחלקה הפועלת כמו String ללא SSO).
נשתמש בבדיקה 90% parsed data 50%50% parsed unparsed different כפי שהוסבר קודם.



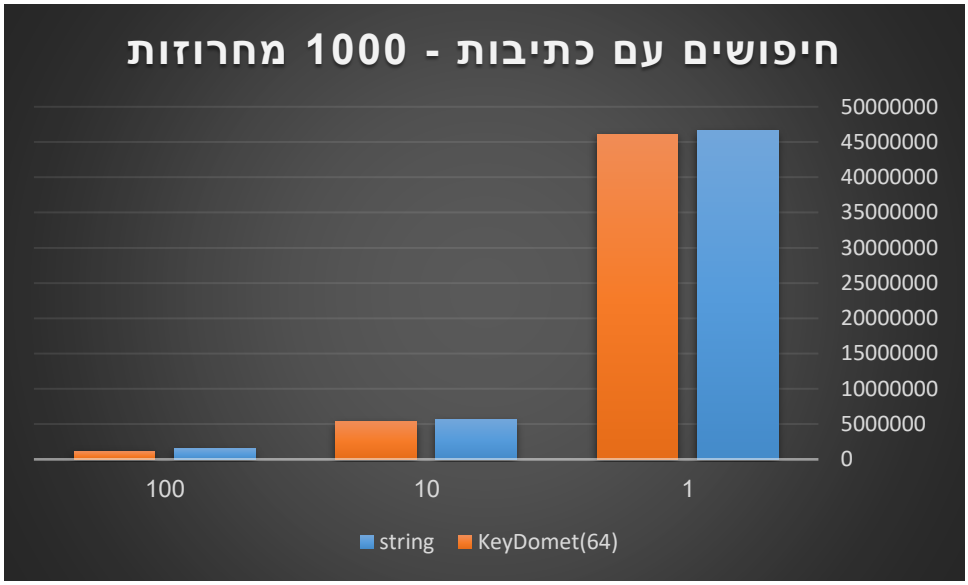
	1000	100,000	1,000,000
KeyDomet(128)	**537746 ns**	**91157755 ns**	**93365044 ns**
wstring	897545 ns	169754793 ns	2399796645 ns
MyString	6873196 ns	893599237 ns	8424892979 ns

ניתן לראות כי ביחס לאופטימיזציות הנוספות KeyDomet עדיין מקבל את הביצועים הטובים ביותר.

- **-BM 90 percent parsed data 5050 parsed unparsed lookups with writes**

מטרתה של בדיקה זו היא לבדוק את הביצועים של KeyDomet גם במצבים בהם לא מתבצע רק חיפוש, שכן יותר מציאותי לבחון מצב שבו מתבצעים גם חיפושים וגם כתיבות למאגר המחרוזות.
נבדקת כאן ההשפעה של מספר המחרוזות כמו בבדיקות הקודמות (1000,100000,1000000), וגם מספר החיפושים על כל פעולת כתיבה (1,10,100).
ציפינו לקבל שוב שיפור בזמנים עבור השימוש ב KeyDomet, זאת כיוון שגם כאן, כאשר אנחנו "מזהמים" את הזיכרון עם פעולות כתיבה, בא לידי ביטוי היתרון של KeyDomet שמטרתה לחסוך הבאת מידע עבור מחרוזות שלמות מהזיכרון.
נחלק את הניתוח של התוצאות לפי מספר המחרוזות (1000,100000,1000000).

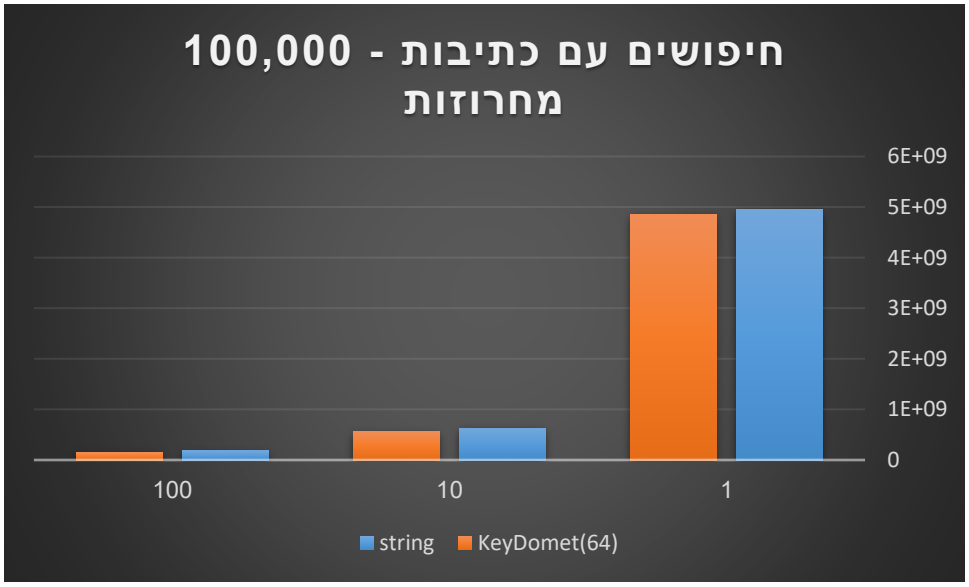
<u>עבור set של 1000 מחרוזות:</u>



	1	10	100
KeyDomet(64)	**46119860 ns**	**5315976 ns**	**1159563 ns**
String	46625604 ns	5646686 ns	1540947 ns
Improvement	1.08%	5.85%	24.74%

ניתן לראות שאכן קיים שיפור, השיפור יותר דומיננטי כאשר ישנן פחות פעולות כתיבה (שיפור של 24.74% עבור פעולת כתיבה לכל 100 חיפושים לעומת 1.08% כאשר מדובר בפעולת כתיבה על כל חיפוש).

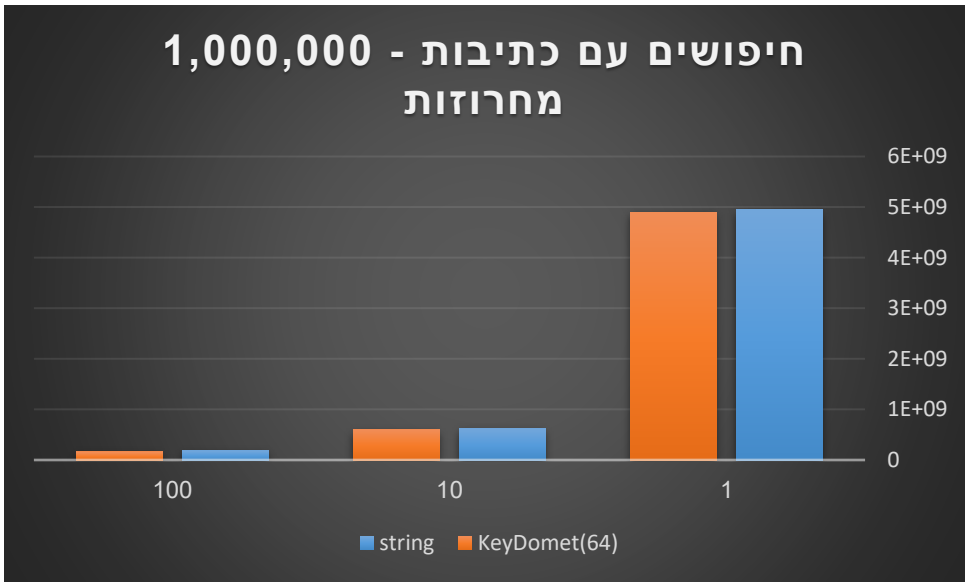
עבור set של 100,000 מחרוזות:



	1	10	100
KeyDomet(64)	**4858047541 ns**	**567345099 ns**	**154111435 ns**
String	4952621791 ns	625256245 ns	186693423 ns
Improvement	1.9%	9.26%	17.45%

גם כאן, עבור 100,000 מחרוזות, ניתן לראות שיפור, ושוב קיים שיפור גדול יותר ככל שמספר פעולות הכתיבה קטן.

עבור set של 1,000,000 מחרוזות:

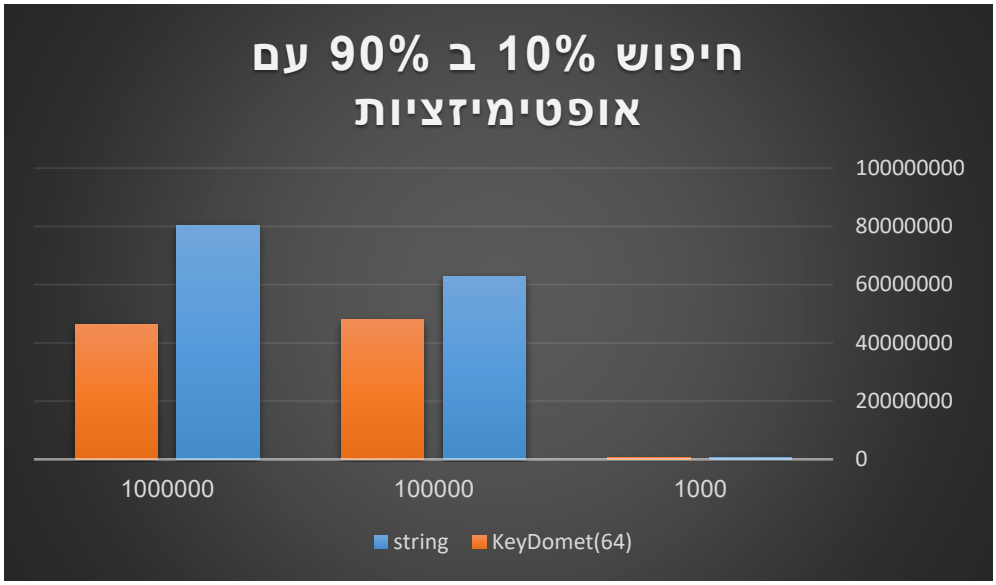


	1	10	100
KeyDomet(64)	**4902789916 ns**	**597104769 ns**	**158698867 ns**
String	4946336482 ns	614839553 ns	187459671 ns
Improvement	0.88%	2.88%	15.34%

שוב גם כאן, עבור 1,000,000 מחרוזות, ניתן לראות שיפור בביצועים עבור KeyDomet, וכן מתקיים שוב שהשיפור המשמעותי הוא עבור מספר קטן יותר של פעולות כתיבה על מספר חיפושים.

בכל הנתונים שהצגנו בגרפים הקודמים החישובים התבצעו ללא אופטימיזציות (-O3), נציג תוצאות גם עבור הרצה תחת האופטימיזציות.
ראינו שיפור משמעותי עבור KeyDomet בבדיקות הקודמות, הצפי בבדיקות אלה הוא לקבל עדיין שיפור עבור KeyDomet.
ניקח בתור מקרי המבחן את הבדיקה שבה אנו מבצעים חיפושים אשר נכשלים (90% ו- 10% כפי שהוצג קודם, בניית set מחרוזות המכיל מחרוזות מתוך 90% מהמאגר - urban dictionary, וביצוע חיפוש על 10% המחרוזות הנותרות, כלומר כל אלה שאינן ב set), ואת הבדיקה של 50%50% אותה גם הצגנו קודם.

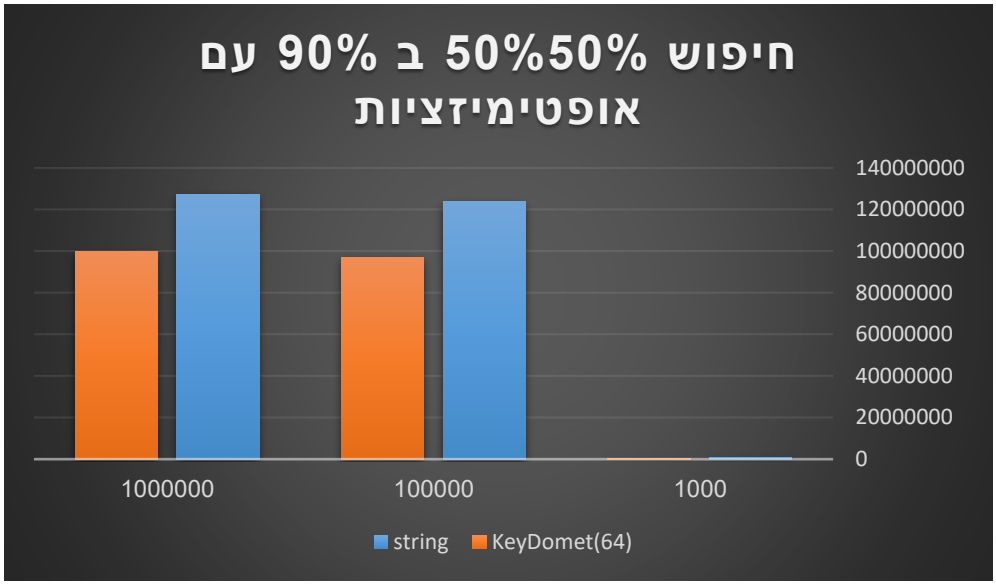
<u>ראשית, בדיקת 90% 10%:</u>



	1000	100,000	1,000,000
KeyDomet(64)	**550206 ns**	**47924501 ns**	**46373327 ns**
String	759987 ns	62737794 ns	80440933 ns
Improvement	27.6%	23.61%	42.35%
Improvement without optimization	33.78%	22.22%	22.91%

ניתן לראות שקיים שיפור גדול יותר במקרה זה כאשר ישנו שימוש באופטימיזציה.

כעת נבצע את הבדיקה עבור המקרה של 50% 50%:



	1000	100,000	1,000,000
KeyDomet(64)	**548263 ns**	**96774519 ns**	**99796573 ns**
String	766616 ns	123681382 ns	127156744 ns
Improvement	28.48%	21.75%	21.51%
Improvement without optimization	24.12%	19.06%	23.31%

גם כאן ניתן לראות שבאופן כללי קיבלנו שיפור גדול יותר.

בסופו של דבר קיבלנו כי גם כאשר ישנו שימוש באופטימיזציות ל- KeyDomet עדיין יש ביצועים טובים יותר בהשוואה ל- String.

ראינו שיפור בזמנים עבור KeyDomet על String ועל אופטימיזציות שונות של String (sso, wstring), ראינו שיפור במקרים שונים הבוחנים מצבים שונים (מספר מחרוזות שונה, כתיבות לזיכרון, אופי החיפוש- 10% 90% 50% 50%).
כעת, נותר להתייחס לשימוש בזיכרון.
נבחן את השימוש בזיכרון כאשר ישנו שימוש ב KeyDomet מול השימוש בזיכרון כאשר אנו משתמשים ב String.
עבור בדיקת הזיכרון קיבלנו את התוצאות:

memory consumption (using size) of set<KeyDometStr64> with 463393 strings is: 95024162 bytes

memory consumption (using capacity) of set<KeyDometStr64> with 463393 strings is: 95026827 bytes

memory consumption (using size) of set<string> with 463393 strings is: 87655224 bytes

memory consumption (using capacity) of set<string> with 463393 strings is: 87657961 bytes

ניתן לראות כי השימוש ב KeyDomet דורש יותר זיכרון, זה הגיוני שכן בנוסף למחרוזות עלינו לשמור גם 64bits עבור הקידומת עצמה לכל מחרוזת.
קיבלנו הרעה של 8.4%, ראינו שהשיפור בזמנים הינו משמעותי יותר, ולכן ניתן להגיד ש**<u>עבור המקרים הנבדקים, אשר מייצגים מקרים מגוונים, השימוש ב KeyDomet נתן ביצועים טובים יותר.</u>**
ציפינו לקבל תוצאה זו, השימוש ב KeyDomet נותן את האופציה לחסוך בהבאת מידע מיותר מהזיכרון והבאת מידע זה לוקחת זמן משמעותי.
KeyDomet מספק אלטרנטיבה טובה ל- String ולמימושים הקיימים (במקרים שנבדקו).

<u>הערה נוספת</u>: קודם ראינו את השוואה בין keydomet<64> ל- keydomet<128>. ראינו כי עבור השימוש ב keydomet<128> קיבלנו שיפור גדול יותר בזמנים, כעת נציג את ההתייחסות לזיכרון אותה ציינו קודם.
קיבלנו כי:

memory consumption (using capacity) of set<KeyDometStr128> with 463393 strings is: 102428850 bytes

memory consumption (using size) of set<KeyDometStr128> with 463393 strings is: 102423956 bytes

ביחס ל- keydomet<64> מדובר בהרעה של 7.8% מבחינת השימוש בזיכרון, ולכן כפי שכבר ציינו השיפור בזמנים מאבד מהרלוונטיות שלו.