Transactional Hash table

-ספר פרויקט-

GitHub: <https://github.com/itssapir/TXHashMap>

מנחה:

גל אסא

סטודנטים:

**ספיר מלכה**

**יעל שביט**

מספר הפרויקט:

**5777**

סמסטר חורף 2020/2021

תוכן עניינים

[חלק 1 - סקירה](#_heading=h.gjdgxs) **3**

[חלק 2 – Transactional Hash table](#_heading=h.3znysh7) **6**

[*הקדמה - Hash Table:*](#_heading=h.tyjcwt) *6*

[המטרה:](#_heading=h.m94wubwiu4) 6

[מימוש Hash Table בסיסי(מנוון):](#_heading=h.3dy6vkm) 6

[תמיכה בכמה HashTable:](#_heading=h.4d34og8) 9

[תמיכה ב Resize:](#_heading=h.17dp8vu) 9

[תמיכה ב Singletons :](#_heading=h.26in1rg) 11

[חלק 3 - בדיקת ביצועים וניתוח תוצאות](#_heading=h.35nkun2) **12**

[**חלק 4 - סיכום ומסקנות**](#_heading=h.hitemyke3q27) **18**

[נספח א' ביבליוגרפיה](#_heading=h.44sinio) **19**

# חלק 1 - סקירה

מוטיבציה:

בעשור האחרון קצב שיפור ביצועי המעבדים כבר לא הולך לפי חוק מור (Moore law), ובשל כך המגמה הרווחת היום בעולם היא להגדיל את מספר המעבדים ולא את ביצועי המעבד. הגדלת מספר המעבדים מביאה לשיפור ביצועים רק בתנאי שאכן משתמשים בכל המעבדים ומשתמשים בהם בצורה טובה ויעילה. השימוש במספר מעבדים במקביל לשם ביצוע משימות הכניס אתגרים חדשים לעולם התכנות וביניהם:

1. חשיבה על ביצוע התכנית בצורה מקבילית ולא טורית כמו בתכנות הקלאסי.
2. עבודה בסביבה בה ריצת התוכנית היא לא דטרמיניסטית. ואין התחייבות לסדר ביצוע התהליכים – ומתוך כך הצורך בסנכרון בין התהליכים.
3. שימוש של מספר חוטים במשאבים משותפים – ומתוך כך הצורך בהגנות ומנעולים.
4. חלוקה נכונה של המשאבים בין החוטים לטובת ביצועים טובים ומניעת נעילה הדדית (deadlock).
5. ביצוע debug לתוכנית מקבילית – הן מצד הכלים והן מכיוון שמהלך הריצה הוא לא דטרמיניסטי ולכן בעיות לא חוזרות בוודאות בכל ריצה וקשה 'לתפוס' תקלות.

אחד האתגרים המרכזיים הוא אתגר מספר 3- הגנה על משאבים משותפים. אתגר זה מוביל לזמן תכנון ופיתוח ארוך במיוחד בשל הקונפליקט המובנה בין הגנות – כאשר הגנה מלאה יכולה לגרום לקוד לחזור להיות למעשה טורי, לבין הצורך בביצועים – כאשר ללא הגנות הנכונות יכולה להיפגע. ובנוסף, זמן הdebug עולה משמעותית הן בגלל הגנות לא מספיקות או לא נכונות שמביאות לפגיעה בנכונות והן בגלל הגנות שמביאות לנעילה הדדית וגורמות לקריסת התוכנית. כאשר מביאים בחשבון שמכיוון שהריצה לא דטרמיניסטית וקשה עד בלתי אפשרי לשחזר ריצות קודמות של התוכנית בצורה מדויקת הקושי עולה אף יותר.

בעוד שהדרך הנפוצה להתמודדות עם אתגר זה היא בעזרת מנגנוני מנעולים, אנו נעסוק בדרך שונה להתמודדות עם הבעיה שנקראת Software Transactional Memory ובקיצור STM.

Software Transactional Memory:

STM היא שיטה שנלקחה מעולם הDatabases אשר ממדלת כל קטע קריטי בו נגשים למשאבים משותפים כטרנזקציה, כאשר בסוף הטרנזקציה יקרו אחד משני המצבים הבאים:

1. הטרנזקציה הצליחה וכל משתמשי המערכת יראו את השינויים שבוצעו במשאבים המשותפים ע"י הטרנזקציה כאילו בוצעו בצורה אטומית (commit).
2. הטרנזקציה נכשלה ולא התבצע אף שינוי במשאבים המשותפים ע"י הטרנזקציה (abort).

לכן, כאשר מתכנת שמשתמש בSTM ניגש לקטע קריטי, כל שעליו לעשות הוא: ליצור טרנזקציה שבתוכה מתבצעות הגישות למשאבים המשותפים, לבדוק האם הצליחה או לא ובמידה ונכשלה לפעול בהתאם (לנסות שוב, לבצע לוגיקה כלשהי וכד'). ע"מ לפשט את התהליך ולשמור על נכונות התוכנית, STM דורשת לכל מימוש עמידה בכללים הבאים:

1. אטומיות (atomicity): מבחינת כל תהליך שמשתמש במשאבים המשותפים ששונו במהלך טרנזקציה, כל השינויים יראו בצורה אטומית. כלומר, כאילו קרו יחד באותו הזמן.
2. בידוד והפרדה (isolation): טרנזקציה לא תהיה מודעת לטרנזקציות אחרות שרצות במקביל כל עוד הן לא סיימו את ריצתן.
3. עקביות (consistency): סדר ביצוע השינויים במשאב משותף יראה אותו דבר לכל תהליך וטרנזקציה במערכת.
4. אטימות (opacity): כל טרנזקציה בכל זמן נתון תראה מצב תקין של המערכת.

STM אופן פעולה:

ע"מ לעמוד בקריטריונים של STM במהלך טרנזקציה יבוצעו הפעולות הבאות:

* נשמור את זמן תחילת הטרנזקציה (בעזרת שעון גלובאלי או בכל דרך אחרת בה ניתן למדל שעון).
* בעת ביצוע קריאה של משאב מהזיכרון המשותף נבדוק את זמן העדכון שלו, אם זמן העדכון מאוחר יותר מזמן תחילת הטרנזקציה עקרון הopacity לא נשמר ולכן נבצע abort לטרנזקציה, אחרת נשמור את ערכו של המשתנה בעותק מקומי.
* בעת כתיבה למשאב מהזיכרון נעדכן ברשימה מקומית את ערכו של המשאב.
* במידה ולא קבלנו abort במהלך הטרנזקציה, ננסה לשמור את השינויים (commit) בזיכרון המשותף, ע"מ לשמור על הקריטריונים, נבצע את השלבים הבאים:
  1. נעילת (lock) כל המשאבים המשותפים ברשימת הכתיבות המקומית (במקרה של כשלון - abort).
  2. וידוא תקינות זמני העדכון (validate) של כל המשאבים שנקראו ו/או נכתבו, בדומה לבדיקה שנעשית בעת קריאת משתנה (במקרה של כשלון - abort).
  3. עדכון ערכי המשאבים (commit) מסט הכתיבות המקומי לזיכרון המשותף.
  4. שחרור נעילת המשאבים המשותפים הנעולים.
* בעת ביצוע abort סט הקריאות וסט הכתיבות נמחקים, הטרנזקציה יוצאת ולא מתבצעים כל שינויים בזיכרון המשותף.

הבעיה בSTM:

התברר שבשל הקריטריונים שSTM הגדירה, הביצועים של תוכנית המשתמשת בSTM ירדו בצורה משמעותית ביותר לעומת תוכנית המבצעת את אותה הלוגיקה בעזרת תכנות מקבילי רגיל. בשל הירידה החדה בביצועים STM בצורתה הכללית איננה כלי פרקטי ולא אומצה בצורה רחבה אלא נשארה בגדר מחקר אקדמי. לאחר ניתוח מעמיק של STM התברר שהבעיה העיקרית שלה היא שSTM מנסה להיות גנרית ככל האפשר ולאפשר שימוש בכל משאב משותף אפשרי. שימוש בכל משאב אפשרי דורשת עמידה בתנאים של כל המשאבים, יוצרת סרבול ותקורה רבה ומונעת אופטימיזציות רבות. ספרייתTDSL (transactional data structure library) באה לפתור את הבעיה הזאת.

Transactional Data Structure Library:

TDSL הוצגה על מנת לשפר את הביצועים של STM ע"י צמצום השימוש בSTM למבני נתונים מסוימים בלבד. עבור כל מבנה נתונים שמומש וקיים בספרייה בוצעו אופטימיזציות פרטניות להתאמה לSTM לאחר ניתוח מעמיק של תכונותיו.

TDSL מומשה בשפת Java וסכמת השימוש בה:

**try**

**try**

TX**.**begin**()**

// operations with TDSL data structures e.g.:

Queue1**.**enqueue**(**VAL1**)**

LinkedList1**.**add**(**VAL2**)**

var **<-** Queue2**.**dequeue**()**

// etc.

**finally**

TX**.**end**()**

**catch** **(**abortException**)**

//do something

כאשר:

* begin() מאתחלת את מבני הנתונים של הטרנזקציה (לא הספרייה).
* במידה ופעולה על מבנה נתונים של הספרייה נכשלת בשל חריגה מהקריטריונים, תיזרק חריגה והטרנזקציה תעבור לשלב הסיום (finally).
* end() מבצעת את הפעולות commit או abort לפי המצב.

ע"מ לוודא את ביצוע הטרנזקציה (commit), ניתן לעטוף את סכמת השימוש בצורה הבאה:

**while** **(true)**

**try**

**try**

TX**.**begin**()**

// ...

**finally**

TX**.**end**()**

**catch** **(**abortException**)**

**continue**

**break**

וזאת מכיוון שבסכמה הנ"ל הטרנזקציה תתבצע בלולאה עד להצלחה.

פעולה בודדת, שנקראת singleton transactions (או בקיצור singleton) נתמכת גם היא ב TDSL.

TDSL הוא בפרט CDSL, ולכן צריך לדעת לתמוך בקוד של CDSL שמבצע פעולות מחוץ לטרנזקציה. לפעולות מסוג זה נתייחס כ singleton. סינגלטונים מקבלים יחס מיוחד: אין להם את התקורה שנובעת מניהול הטרנזקציה, והמתכנת לא צריך להתמודד עם abort.

TDSL כיום:

בדיקות הראו שהשימוש בTDSL משפר משמעותית את הביצועים ביחס לSTM סטנדרטי, ומקרב אותה לרמת הביצועים שמתקבלת משימוש במנעולים, כך שניתן להתייחס לTDSL כחלופה אפשרית לתכנות מקבילי בעזרת מנעולים.

כיום TDSL מספקת תמיכה למבני הנתונים : Queue, SkipList, Red-Black Tree, Producer-Consumer Pool.

# חלק 2 – Transactional Hash table

## הקדמה - Hash Table:

טבלת ערבול (Hash table) היא מבנה נתונים מילוני.

מילון הוא מבנה נתונים מופשט המגדיר אוסף של מפתחות וערכים, כאשר המילון מורכב ממיפוי חד-ערכי בין מפתח(key) לערך (value). הממשק הבסיסי של המילון תומך בפעולות: הוספת איבר, חיפוש איבר ומחיקת איבר. ישנם מימושים התומכים במצב שבו המפתחות מהווים טווח צר של מספרים. כאשר זה אינו המצב, טבלת ערבול מתעלה על מבני הנתונים המילוניים הנפוצים בכך שהיא מספקת מימוש לפעולות הבסיסיות בסיבוכיות זמן הנמוכה ביותר.

מבנה טבלת הערבול עובד באמצעות הפיכת המפתח ע"י פונקציית ערבול (hash function) למספר המייצג אינדקס במערך. התא המתאים במערך מפנה לערך הרצוי. במיפוי חד-חד-ערכי מתעוררת בעיה: במידה וטווח המפתחות האפשריים גדול בהרבה מטווח המפתחות שבפועל נרצה לשמור במבנה הנתונים, נאלץ להקצות מקום למערך הרבה יותר גדול ממה שאנחנו צריכים. הפתרון לבעיה זו הוא לאפשר פונקציית ערבול שאיננה חד-חד ערכית. באופן זה נקצה מערך קטן יותר, ויתכנו שני מפתחות שהפעלת פונקציית הערבול עליהם תחזיר את אותו אינדקס. זוהי בעיה שנקראת "התנגשות". אחד מהפתרונות של בעיית ההתנגשויות הוא שימוש בטבלה שבה כל תא במערך מפנה למבנה נתונים נוסף, ששם נשמור את הערכים שמופו לאינדקס זה ע"י טבלת הערבול. מבנה זה יספק את אותן הפעולות כמו טבלת הגיבוב. כדי לשמור על סיבוכיות זמן נמוכה, ולא להגיע למצב שבו עבור חלק מהאינדקסים מצביעים לערכים רבים וחלק בעלי איברים בודדים, פונקציית הערבול צריכה להחזיר ערכים באופן קרוב עד כמה שאפשר להתפלגות אחידה בדידה. כדי לאפשר ביצוע פעולות על המערך להיות גדול, כדי להקטין את ההתנגשויות. מנגד אנחנו לא רוצים לשמור מערך גדול מדי. לכן המערך בטבלת הערבול יהיה מערך בעל גודל משתנה, אשר יגדל רק במידה במידה ויש צורך אמיתי בהגדלה, כלומר כאשר יש יותר מדי התנגשויות. ב-JAVA טבלת הערבול ממומשת באופן שפותר את הבעיה הזו. כאשר כמות האיברים שהטבלה מכילה מגיעה לסף מסוים, הטבלה תכפיל את גודל המערך הפנימי, ותפזר את האיברים מחדש וכך תצמצם את מספר האיברים ששייכים לתא לאינדקס מסוים במערך.

## המטרה:

להוסיף Hash Table לספריית TDSL, ולממש אותו תוך שמירה על נכונות מאפייני הטרנזקציה ומתן דגש על מקביליות מירבית.

נתחיל במימוש "מנוון" (טבלה יחידה בגודל קבוע וללא סינגלטונים ברקע) במטרה להכיר את TDSL כפי שהיא ממומשת כיום. נרצה להבין לעומק את אופן פעולת הטרנזקציה והנתונים שעלינו לשמור בזיכרון המקומי באופן המותאם לתכונות טבלת הערבול. לאחר הכנת בסיס יציב (מבחינת מימוש ומבחינת ההבנה שלנו) נעבור למימוש התומך במספר טבלאות ערבול, לאחר מכן נוסיף תמיכה בשינוי דינאמי של גודל הטבלה. לבסוף נתמוך גם במקרה של פעולות סינגלטון אשר נעשות על טבלת הערבול.

## מימוש Hash Table בסיסי(מנוון):

התחלנו במימוש טבלת ערבול טרנזקציונית "מנוונת". כלומר, מימוש תחת ההנחה שהטרנזקציה תפעל על טבלת ערבול יחידה, גודל טבלה קבוע, ואין פעולות סינגלטון שפועלות ברקע. חקרנו את המימוש שקיים ב-Oracle להגדרה של טבלת ערבול, והתמקדנו בשני מימושים: HashMap, HashTable. גילינו שהמימוש של HashMap מתאים לדרישות שלנו יותר מאשר המימוש של HashTable (אסינכרוני, ומסוגל לקבל מפתחות וערכים שהם null). לכן בחרנו להוסיף ל-TDSL מימוש של HashMap, הנקרא TXHashMap. תחילה ביססנו את המימוש של TXHashMap על המימוש הקיים ב- Oracle, כלומר נעזרנו בשלד הבסיסי, אך ביצענו שינויים ותוספות בהתאם לעבודה עם טבלת ערבול מתוך טרנזקציה, והדרישות העולות מכך.

הערה: מכאן נתייחס אל HashMap ואל HashTable כאל אותו מבנה.

HashMap של Oracle:

מבנה הטבלה התבסס על המבנה של HashMap , ולכן נסביר בקצרה על המימוש של מבנה זה.  
 HashMapשומר זוגות של <Key, Val>. הוא בנוי ממערך של Node, כאשר כל Node יהיה איבר של רשימה מקושרת, המכיל את השדות:

* K key- המפתח של הזוג <Key, Val> שה-node שומר.
* V value- הערך של הזוג <Key, Val> שה-node שומר.
* int hash- לאובייקט K קיימת מתודה שמחזירה ערך HashCode המתאים לאותו key.

בהוספת זוג <Key, Val> מחושב ערך ה-HashCode של המפתח. כדי להימנע מחישובים חוזרים של פונקציית הhash בעתיד, נשמור את ערך זה.

* Node next- הפתרון להתנגשויות. במידה ונכניס זוג נוסף שעבורו יתקבל אותו hash, נשמור זוג זה בnode חדש והוא ישורשר לnode הקיים באותו האינדקס ע"י שרשורו בשדה זה.

מבנה של TXHashMap והשיקולים בבחירתו:

מבנה ראשוני

התחלנו במימוש מבנה הדומה מאוד למבנה של HashMap. כדי לקיים את הדרישות של הטרנזקציה, בחנו מה הם הנתונים שנרצה לשמור בזיכרון לוקאלי של הטרנזקציה.

המחשבה הנאיבית הייתה לאפשר נעילה ברמת הnode. כלומר, להקצות לכל node מנעול, ולשמור לו גרסה.  
כפי שמתבצע במימוש הבסיסי של הרשימה המקושרת, בסט הקריאה אנחנו נרצה לשמור כל node שקראנו ממנו, ובסט הכתיבה את ה-node שכתבנו אליהם. בסוף הטרנזקציה נעבור על סטים אלו כפי שמוסבר בהקדמה, ננעל במידת האפשר את האיברים הרלוונטים, נבצע ולידציה ובמידה ולא נדרשנו לעשות abort נעדכן את ה-node'ים בזיכרון המשותף לפי הצורך.

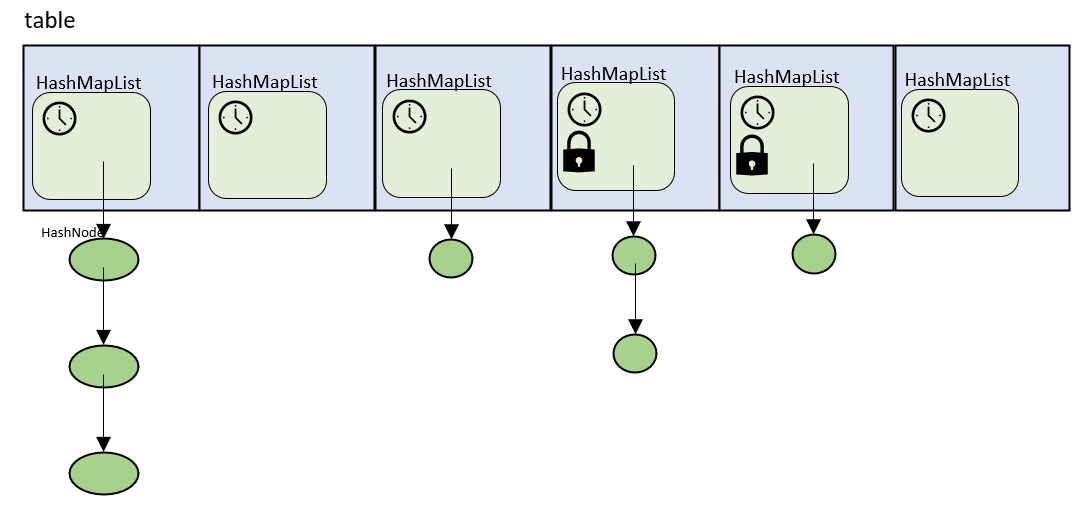
אנחנו מניחים שהרשימות המקושרות יכילו מספר קטן של איברים. הנחה זו נובעת מתכונות הטבלה:

* הנחה כי פונקציית הhash מספקת פיזור אחיד
* מימוש פונקציית resize אשר תבטיח שכאשר האיברים בטבלה יעברו סף מסוים, טווח האינדקסים יוכפל והאיברים יפוזרו מחדש בצורה אחידה כך שאורך הרשימות בממוצע יקטן פי 2.

מבנה TXHashMap:

תחת ההנחה שהוזכרה, פעולות שנבצע על איבר ברשימה מסויימת ישפיעו על כמות לא זניחה של איברים באותה הרשימה. לכן, הוחלט על נעילה בגרנולריות של רשימה, ולא ברמת הnode. גישה זו תפחית את כמות המנעולים שיש לתחזק במערכת, תפשט את הלוגיקה, ותאפשר זיהוי מוקדם יותר של התנגשויות.

מבנה הTXHashMap שנבחר:

****

TXHashMap בנוי ממערך של HashNodeList, המכיל את השדות:

* HashNode head- הראש של רשימה מקושרת של HashNode. HashNode מכיל את אותם השדות כמו- Node, ובנוסף מכיל דגל isDeleted (הסבר על תפקידו בהמשך), ומצביע ל HashNodeהבא.
* מנעול לרשימה כולה
* Version – הזמן שבו הרשימה שונתה בפעם האחרונה.

כלומר עטפנו כל רשימה במבנה HashNodeList .

בחירת הנתונים של הטרנזקציה שנשמור בצד:

כאמור, TDSL מתחזקת זיכרון מקומי שמכיל את נתוני הטרנזקציה:

* readVersion, שהוא ה-GVC בתחילת הטרנזקציה
* דגל שאומר האם אנחנו בטרנזקציה
* דגל שמסמל האם פעולות הטרנזקציה הן פעולות read בלבד או לא
* סט הכתיבה וסט הקריאה, כאשר יש סט מתאים לכל אחד מהמבנים ש-TDSL תומכת בהם.

אנחנו נוסיף את סט הכתיבה וסט הקריאה שמתאימים לטבלת ערבול טרנזקציונית. במהלך הטרנזקציה, בעת קריאה של איבר מרשימה מקושרת כלשהי, נוסיף את הרשימה לסט הקריאה. כאשר נרצה לבצע פעולת כתיבה של איבר לרשימה מסוימת, אנחנו קודם כל קוראים מהרשימה. לכן נוסיף רשימה זו לסט הקריאות. כמו כן, נרצה להוסיף לסט הכתיבות את הרשימה, כדי לתת אינדיקציה שזוהי רשימה שאנחנו רוצים לשנות, ובנוסף את ה-HashNode אותו אנחנו רוצים להוסיף/לשנות.

לכן, בחרנו לממש את סט הקריאות בעזרת HashSet של HashNodeList. את סט הכתיבות בחרנו לממש בעזרת HashMap , כאשר ה-key הינו HashNodeList וה-value הוא מיפוי בין המפתחות שיש להכניס/לשנות ב-NodeList לבין ה-HashNodes המתאימים להם. מיפוי זה מקל על ניהול סט הכתיבה ומאפשר חיפוש ושינוי מהיר של איברים בו. בדרך זו, כאשר נבצע את הכתיבה לזיכרון המשותף בשלב ה-commit, נוכל לעבור על כל אחת מהרשימות שמופיעות בסט הכתיבה, ולעדכן את האיברים הרלוונטים בזיכרון המשותף.

אתחול הסטים של טבלאת הערבול יתבצעו במהלך ביצוע של פעולה של טבלת הערבול הטרנזקציונית (בכל אחת מהמתודות נבדוק תחילה האם הסטים של הטבלה אותחלו, ואם לא נאתחלם).

מימוש פעולות בסיסיות.

נתמוך בפעולות הבאות:

put(K,V), get(K), remove(K), containsKey(K)

Get:

* נמצא את הרשימה המתאימה ל-K הנתון.
* נבדוק האם יש לבצע abort מוקדם: אם הרשימה נעולה או שהשעון של הרשימה גדול מהשעון של תחילת הטרנזקציה.
* נוסיף את הרשימה לסט הקריאות.
* נבדוק האם האיבר המתאים נמצא בסט הכתיבות. אם כן, סימן שערך זה מבחינת הטרנזקציה הוא החדש ביותר, ונחזיר אותו.
* אחרת, נרצה לבדוק האם האיבר המתאים קיים בזיכרון המשותף:
  + ניגש לרשימה ההמקושרת המתאימה בטבלת. אם הרשימה נעולה, abort.
  + נעבור על איברי הרשימה. נגן על המעבר בעזרת בדיקת נעילות, בדיקת גרסאות, וניעזר בloadFence בשביל להבטיח את סדר הקריאות והבדיקות לצורך שמירת הנכונות.
  + אם מצאנו את האיבר המתאים, ואין לו אינדיקציה שנמחק נחזיר אותו.

Put:

* נמצא את הרשימה המתאימה ל-K הנתון.
* נבדוק האם יש לבצע abort מוקדם: אם הרשימה נעולה או שהשעון של הרשימה גדול מהשעון של תחילת הטרנזקציה.
* נוסיף את הרשימה לסט הקריאות.
* נבדוק האם האיבר המתאים כבר נמצא בסט הכתיבות. אם כן, נחליף את הערך הישן בערך החדש, ונחזיר את הערך הישן.
* אחרת, נרצה לבדוק האם האיבר המתאים קיים בזיכרון המשותף (באותה צורה שמבוצעת בget)
* נוסיף את האיבר החדש לסט הכתיבות.

containsKey מבוצע באופן דומה לget (מחזיר true/false במקום ערך).

remove מבוצע באופן דומה לput (מסמנים דגל isDeleted במידה והאיבר קיים).

## תמיכה בכמה HashTable:

כדי לתמוך בתמוך בכמה טבלאות ערבול שונות מתוך אותה הטרנזקציה, נרצה שהזיכרון המקומי של הטרנזקציה ידע לקשר בין סט הכתיבות של הטבלאות השונות. לשם כך נשדרג את הזיכרון המקומי. נשמור HashMap שממפה בין מפתחות שהם טבלה טרנזקציונית ספציפית TXHashMap, לבין ערך שהוא LocalHashMap: סט הקריאות וסט הכתיבות המקומי של הטרנזקציה עבור אותו TXHashMap.

בזמן קריאה ראשונה למתודה של HashMap מסוים, ניצור את הLocalHashMap שלו ונוסיף אותו למפת הHashMap של הטרנזקציה. בעת ביצוע מתודות של HashMap, כל הגישות למבנים הטרנזקציוניים יופנו לגרסה המתאימה לHashMap הנוכחי.

לבסוף, בשלב הCommit, יש לבצע את הפעולות על כל אחד מהLocalHashMap במערכת.

## תמיכה ב Resize:

אלמנט חשוב של טבלאות ערבול הוא הגודל הדינאמי שלהן. מכיוון שהטבלה מכילה בתוכה מערך, נרצה שגודל המערך לא יהיה גדול מדי כדי לא לצרוך זיכרון שלא לצורך. מצד שני, עבור מערך פנימי קטן מדי יגדלו כמות הפעולות שיש לבצע על מנת למצוא איבר ברשימה עקב גדילת הרשימות המקושרות שבמערך.

בעיה זו נפתרת על ידי ביצוע פעולת Resize אשר מגדילה את גודל הטבלה הפנימית פי 2.  
עבור ביצוע Resize במקרה הרגיל, יש לבצע את הפעולות הבאות:

* להקצות טבלה בגודל החדש
* עבור כל תא בטבלה הישנה:
  + לעבור על כל אחד מהאיברים ברשימה המקושרת שבתא, ולחשב את האינדקס שלו מחדש עבור גודל הטבלה הכפול.
  + למקם אותו ברשימה המקושרת הנכונה עבור הטבלה החדשה.

במקרה הטרנזקציוני, פעולת הResize יוצרת בעייתיות רבה מכיוון שלמרות שהמשתמש סך הכל ביצע put אשר לכל היותר יכל לגרום להתנגשות עבור אלמנט בודד ברשימה, בפועל בעת ביצוע Resize יש נגיעה ושינוי של כל המבנה הפנימי של טבלאת הערבול, וכתוצאה מכך יכולות להיווצר התנגשויות נוספות בלתי רצויות.

*פתרונות אפשריים לResize:*

1. ברגע שחוט יגיע לThreshold שבו יש לבצע Resize, החוט יתחיל על ידי סימון דגל resize ונעילת כל הרשימות המקושרות בטבלה הישנה. לאחר מכן החוט ישלים את פעולת הResize כפי שנעשית במקרה הסטנדרטי ללא טרנזקציות. לבסוף יוריד את דגל הresize ושאר החוטים במערכת יוכלו לחזור לביצוע הטרנזקציות שלהם.
2. החוט אשר מבצע את Resize לא ינעל את כל הטבלה הישנה, אלא ינעל אינדקסים בהדרגה בסדר עולה.

כך נוכל לאפשר לחלק מסויים של החוטים במערכת לסיים את הטרנזקציות הפעילות שלהם ולבצע commit.  
בנוסף, חוטים אשר היו אמורים לבצע abort עקב התנגשות עם החוט שמבצע Resize, לא יבצעו abort, אלא יחכו לסיום פעולת הResize, יעדכנו את המצב הפנימי שלהם למקרה שבו מרחב האינדקסים גדל פי 2, וימשיכו את הטרנזקציה כרגיל.

פתרון זה מאפשר לנו למנוע לחלוטין aborts אשר נובעים מResize. דבר אשר רצוי מאוד מכיוון שResize הוא מנגנון פנימי של המבנה ואינו באמת מהווה התנגשות כלל.

1. שיפור אפשרי נוסף למנגנון הוא מקבול עבודת הResize. כבר דרשנו מכל חוט אשר זיהה התנגשות שמקורה בResize לחכות. לכן, נוכל לדרוש מאותם חוטים, במקום לחכות, להצטרף לביצוע פעולת הResize באופן מקבילי. שיפור זה יזרז את הזמן שלוקח לבצע Resize ויאפשר למערכת לחזור למצב שגרתי מהר יותר.

בחרנו בפתרון השני מכיוון שהוא מכיל שיפור משמעותי על פני הפתרון הנאיבי, בעוד שהמימוש השלישי מכיל שיפור פחות משמעותי ומגדיל את סיבוכיות המימוש באופן ניכר.

שינויים נדרשים לתמיכה בResize:

* יש לאפשר לכל חוט לבדוק את גודל הטבלה במהלך הטרנזקציה.

★פעולה זו אינה טריויאלית, מכיוון ששמירת גודל מוחלט דורשת הוספת משתנה אטומי של size.

משתנה כזה יוסיף נקודת contention קריטית, מכיוון שיש לגשת אליו בכל קריאה לput כדי להחליט האם לבצע resize.  
תחזוק משתנה size מוחלט גורר בעיות ביצועים ונכונות מורכבות ורבות, ולכן הוחלט לא לתמוך בו במסגרת הפרויקט.

לכן, הוחלט להוסיף משתנה size ללא דיוק מוחלט. במהלך כל טרנזקציה נשמור את השינוי בגודל הטבלה כתוצאה מפעולות put/remove. בשלב הcommit נעדכן את size לפי כמות האיברים שנוספו/נמחקו בטרנזקציה בעזרת פעולת fetch and add.

בעת תחילת טרנזקציה, נקרא את הערך הנוכחי של size העדכני, ונשמור אותו בזכרון המקומי של הטרנזקציה.  
כעת, נוכל להשתמש בערך size המקומי, ובערך sizeDiff כדי להעריך את גודל הטבלה במהלך הטרנזקציה.  
בדיקה באופן זה אינה מדויקת, אבל היא מבוצעת רק לצורך resize, אשר אינו משפיע על הטבלה באופן שנראה למשתמש, אלא רק על המבנה הפנימי המוסתר מהמשתמש. כלומר, לא קיימת בעיית נכונות עקב השימוש בsize משוער, אלא רק שיקולי אופטימיזציה/ביצועים.

* נחזיק בטבלה מנעול inResize אשר ינעל על ידי חוט שמתחיל ביצוע resize. מטרת המנעול היא מניעה של ביצוע מספר פעולות resize על ידי חוטים שונים במקביל.
* בכל HashNodeList נחזיק משתנה isDeprecated. משתנה זה יעודכן על ידי החוט שמבצע resize ברגע שהחוט מתחיל להעביר את איברי הרשימה לטבלה החדשה.
* על כל חוט להחזיק רפרנס לטבלה הפנימית שאיתה הוא עובד.

בעת סיום פעולת resize, הטבלה הגלובלית מוחלפת. לכן, כדי למנוע מצב שבו הטבלה מוחלפת ללא ידיעת החוט, נרצה לשמור גרסה לוקאלית שלה.

* נחזיק בכל טבלה latch כדי לאפשר לחוטים אחרים במערכת לחכות לסיום הresize.

ברגע שהחוט שמבצע resize יסיים את פעולתו, הוא יוריד את הlatch ושאר החוטים יחזרו לפעולה.

*אופן פעולת Resize:*

כאשר חוט מבצע Put ומזהה שגודל טבלת ההאש עבר את Threshold (ברירת מחדל 75% מגודל הטבלה הפנימית), הוא יתחיל בפעולת Resize על ידי נעילת מנעול resize בטבלה הישנה אשר מטרתו היא מניעה של ביצוע Resize במקביל על ידי מספר חוטים.  
כעת החוט יכול ליצור טבלה חדשה, ולהתחיל לעבור על אינדקסי הטבלה המקורית בסדר עולה.  
עבור כל אינדקס, ראשית החוט יסמן את הרשימה המקושרת של האינדקס כDeprecated, כך שבמקרה שבו חוט אחר יגש אל הרשימה, הוא ידע שזוהי התנגשות עקב Resize, ולא התנגשות רגילה.  
לאחר מכן, החוט יעבור על איברי הרשימה המקושרת, יחשב עבורם אינדקס חדש, ויכניס לרשימה החדשה בהתאם.

מכיוון שחישוב האינדקסים מבוצע בעזרת פעולת מודולו של אורך הטבלה הפנימית, כל אינדקס בטבלה הקודמת יפוצל לשני אינדקסים בטבלה החדשה. לכן, נסמן את הגרסה של שני האינדקסים הללו בתור הגרסה של האינדקס בטבלה המקורית.

לבסוף, לאחר ביצוע הפעולות הנ"ל לכל אחד מאינדקסי הטבלה המקורית, החוט ישנה את הreference לטבלה הגלובלית אל הטבלה החדשה, ויסמן בטבלה הישנה שביצוע הResize הושלם כדי לאפשר לחוטים שמחכה לשוב לפעולה.

על כל חוט שמגלה שבוצע Resize, כולל זה שביצע את אותו, יש לבצע עדכון של המצב הטרנזקציוני שלו כדי להתיישר אל מצב המבנה החדש.  
ברגע שהResize הושלם, החוט יקח את הreference לטבלה החדשה, ויעדכן את הread/write sets שלו כך שכל איבר בהם יותאם לאינדקסים הנכונים עבור תצורת הטבלה החדשה.

## תמיכה ב Singletons :

עד כה אפשרנו תמיכה בפעולות טרנזקציוניות עבור טבלת הערבול. נרצה לאפשר בנוסף שימוש רגיל במבנה הנתונים, כך שקוד שנכתב לטבלת ערבול סטנדרטית יתמך גם על ידי הטבלה שלנו.

פעולות אלה נקראות סינגלטון – פעולות בודדות אשר לא שייכות לטרנזקציה, ומבוצעות באופן אטומי וThread safe.  
עלינו לתמוך בביצוע פעולות סינגלטון גם במקביל לקיום טרנזקציות. כלומר, אין להסיק משימוש בסינגלטונים שאין טרנזקציות במערכת.

הבעיה:  
הטרנזקציות משתמשות ב"שעון" של TX, הGVC, כדי לבצע הגנה וסינכרון של גישות.

שעון זה יוצר נקודת contention אשר משותפת לכלל הטרנזקציות מכיוון שיש לעדכן אותו בסוף שלב הcommit. תקורה זו ממוסכת ככל שגודל הטרנזקציות עולה מכיוון שהיא אינה תלויה בגודל הטרנזקציה אלא רק בכמות הטרנזקציות במערכת.

במקרה של סינגלטונים, במידה וניאלץ לעדכן את הGVC, נקבל נקודת contention עבור כל גישה בודדת למבנה הנתונים. כלומר, ניצור תחרות בגישה בין כל זוג איברים של מבנה הנתונים, גם אם אלו איברים שונים אשר יכלנו לאפשר גישה מקבילית לחלוטין אליהם.

מסיבה זו נרצה שגישות סינגלטון לא יעדכנו את הGVC. דבר זה אינו טריויאלי מכיוון שאנו רוצים לאפשר שימוש בסינגלטונים לצד טרנזקציות, ולא באופן בלעדי אחד לשני. **במידה ולא נעדכן GVC עבור גישות סינגלטון ניתקל בבעיית נכונות.**

*נמחיש את הבעיה בעזרת דוגמא:*

נניח שקיימים שני חוטים במערכת, כאשר אחד מבצע טרנזקציה והשני מבצע פעולות סינגלטון

|  |  |
| --- | --- |
| Thread 2:  hashMap.put(2,2)  hashMap.put(2,3)  hashMap.get(1)  hashMap.get(2) | Thread 1:  **TX {**  hashMap.put(1, hashMap.get(2))  if (hashMap.get(2) == 3) {  hashMap.put(2,4)  }  **}** |

עבור המקרה שבו הקריאות יקרו בסדר הבא:

Thread 1

Thread 2

t

t

put(2,2)

put(2,3)

TX begin

get(2)

put(1,2)

put(2,4)

get(2)

3

get(1)

2

get(2)

4

TX end

2

בגלל שהסינגלטון לא השתמש בGVC, החוט שביצע את הטרנזקציה לא זיהה את ההתנגשות, וביצע commit לטרנזקציה, למרות שהערך של מפתח 2 השתנה במהלכה. כעת קיבלנו טרנזקציה שלא עמדה בתנאי הatomicity ובתנאי הisolation.

הפתרון:

הוספת דגל singleton לכל אחת מהרשימות המקושרות בטבלת הערבול.  
כאשר תתבצע פעולת singleton, החוט המבצע יעדכן את זמן השינוי של הרשימה לזמן הנוכחי במערכת (GVC), דבר זה צורך רק קריאה של הGVC ואינו צורך עדכון שלו. בנוסף, החוט ירים את דגל ה singleton של הרשימה.

כעת, כאשר חוטים אשר מבצעים טרנזקציות ניגשים לרשימה כלשהי, בנוסף לבדיקה שזמן השינוי של הרשימה אינו גדול מזמן ההתחלה של הטרנזקציה, יבדקו את המקרה שבו הזמן שווה לזמן תחילת הטרנזקציה וגם הורם דגל הsingleton. במקרה זה, החוט לא יכול להכריע האם הרשימה עודכנה לפני תחילת הטרנזקציה או אחרי, ולכן יש לבצע abort.

לפני ביצוע הabort, כדי להימנע מabort בנסיון הבא להריץ את הטרנזקציה (אם פשוט ננסה שוב כנראה החוט יתקל באותה סיטואציה בדיוק וייפול שוב), נדאג להגדיל את הGVC כך שיובטח לנו שנראה זמן גדול יותר בניסיון הבא ונוכל להתקדם.

מימוש put/get לסינגלטון:

get: נרצה לממש את get בלי שימוש בשום מנעולים. על המתודה להחזיר ערך אשר היה חוקי ברגע כלשהו בין הנקודה שבה נקראה המתודה לבין הנקודה שבה חזרה התשובה.

לצורך כך, נבצע ריצה על הרשימה המקושרת הרלוונטית עבור האיבר המבוקש באופן הבא:

1. נוודא שלא בוצע resize, אם בוצע resize ניקח את הרשימה המקושרת החדשה המתאימה ונתחיל מחדש.
2. נשמור את הGVC
3. נקרא את האיבר הנוכחי ברשימה המקושרת ונשמור את ערכיו לוקאלית.
4. נבדוק שהרשימה לא נעולה, אם הרשימה נעולה נתחיל שוב מראש הרשימה (ונחזור לשלב 1)
5. נבדוק שהGVC ששמרנו זהה לGVC הנוכחי. במידה והGVC השתנה נתחיל מחדש.
6. נשווא את המפתח של האיבר הנוכחי לזה שאנו מחפשים, אם יש התאמה נחזיר תשובה, אחרת נעדכן  
   current = current.next ונחזור לשלב 1.

ביצוע השלבים הנ"ל מבטיח איטרציה חוקית על הרשימה המקושרת, וקריאה של ערכים חוקיים ולא ערכי ביניים.

put: מימוש put פשוט יותר מכיוון שניאלץ לבצע נעילה על הרשימה בכל מקרה. לכן, ראשית נמצא וננעל את הרשימה המקושרת המתאימה למפתח. (תוך כדי התחשבות במקרים כמו resize וכדומה)  
אחרי לקיחת המנעול נבצע put באופן רגיל כפי שמבוצע בטבלת ערבול רגילה, תוך כדי סימון גרסת הרשימה המקושרת ודגל הסינגלטון כפי שהוזכר לעיל.

לשאר המתודות הותאמו גם גרסאות singleton כאשר אופן הפעולה בגרסאות הללו דומה במהותו למה שהוזכר עבור put/get.

# חלק 3 - בדיקת ביצועים וניתוח תוצאות

לצורך בדיקת הביצועים יצרנו benchmark שמבצע פעולות put/get על HashMap יחיד, ומודד את זמן הביצוע.

מבנה ה-benchmark:

מטרת הbenchmark היא השוואה בסיסית של ביצועי TXHashMap מול Concurrent HashMap סטנדרטי (Oracle)

בכל הרצה, נרצה לבצע כמות מסוימת של פעולות put או get, כאשר אנו דורשים שהפעולות יתבצעו בקבוצות.

למשל, עבור 10000 פעולות וקבוצות של 20, נדרש לבצע את הפעולות כך שבכל פעם יבוצעו 20 פעולות באופן אטומי.

עבור TXHashMap הדבר יתבצע על ידי הגדרת טרנזקציות כאשר כל טרנזקציה מבצעת 20 פעולות (put/get), בעוד שעבור Concurrent HashMap זה יתבצע בעזרת בלוק אשר מוגן בעזרת מנעול ומבצע את 20 הפעולות.

נרצה להשוות את שני מבני הנתונים כתלות בפרמטרים הבאים:

* גודל הקטע הקריטי (גודל טרנזקציה)
  + 1 / 10 / 20 / 30 / 40 / 50 / 60 / 70 / 80 / 90 / 100
* כמות הפעולות הכוללת בריצה
  + 200,000 / 1,000,000 / 2,000,000
* סוג הפעולה
  + Put / Get
* טווח המפתחות שבשימוש
  + (0-10,000) / (0-100,000)
* מספר חוטים
  + 2 / 4 / 8 / 16 / 32

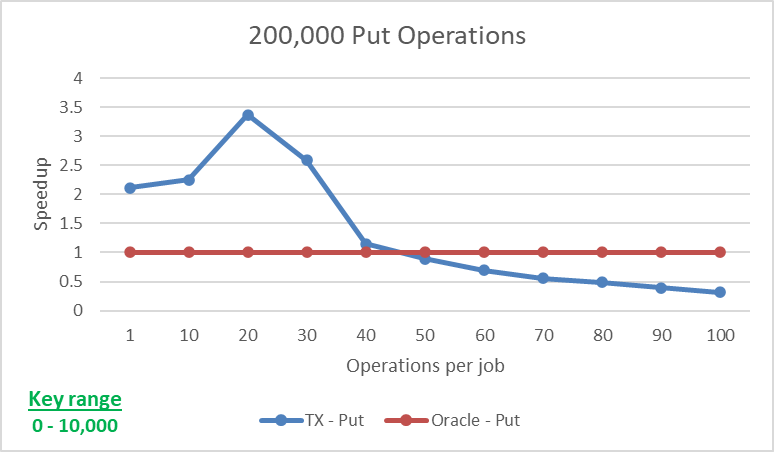
עבור כל תצורת בדיקה, ביצענו 100 ריצות ולבסוף נלקח ממוצע הריצות.  
בתחילת כל ריצה, בוצעו 10,000 פעולות Warmup על מבנה הנתונים, כדי להימנע מהשפעת תקורות אפשריות של מבני הנתונים החדשים.

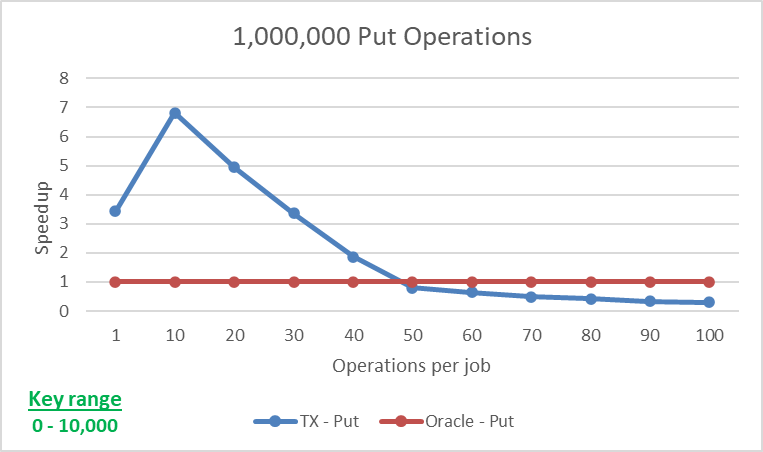
הרצת הטסטים התבצעה על הOracle במעבדה (בעל 16 ליבות כפולות) עבור 16 threads

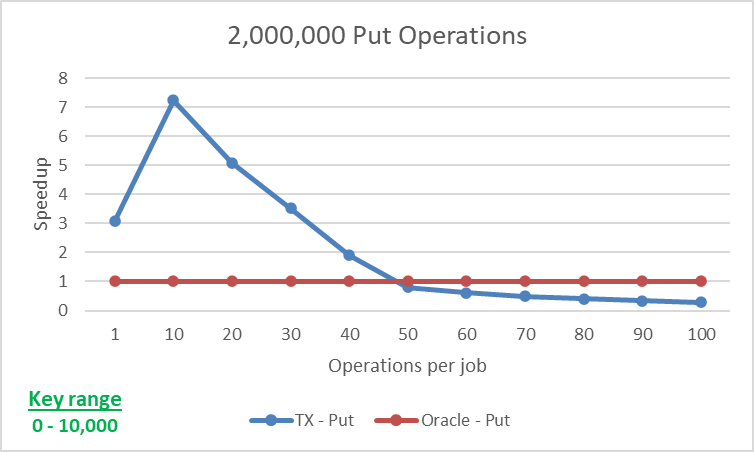
תוצאות:

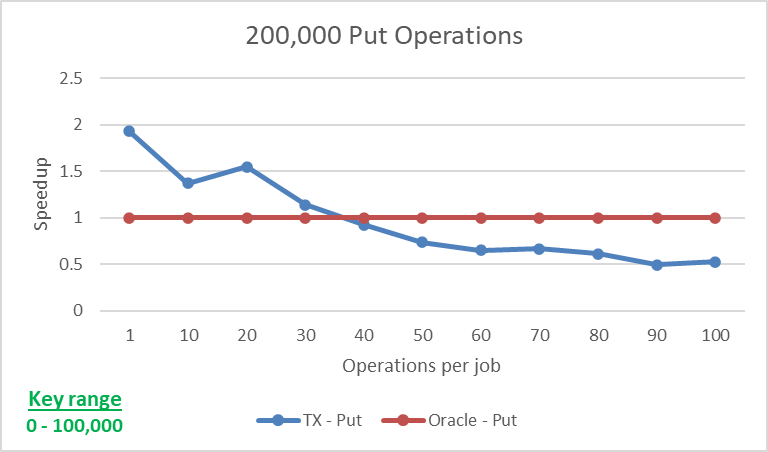
*כתלות בגודל הטרנזקציה:* (הגרפים מנורמלים ביחס לביצועי Concurrent HashMap)

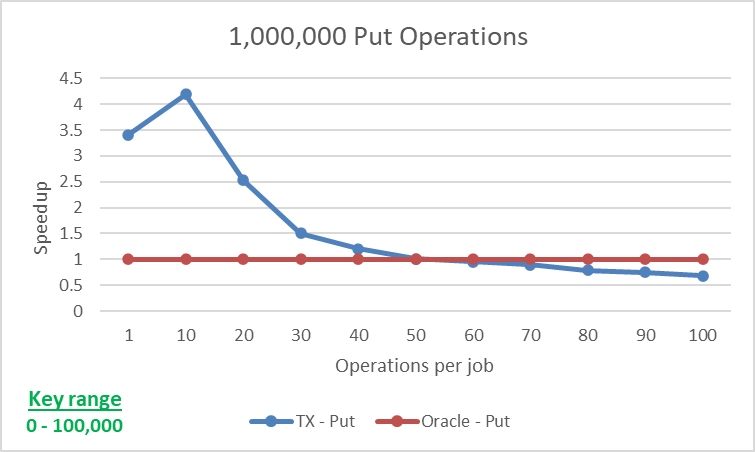
Put:

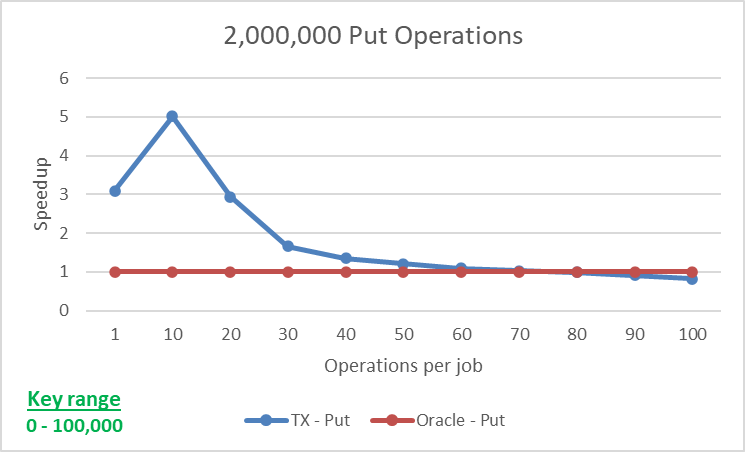








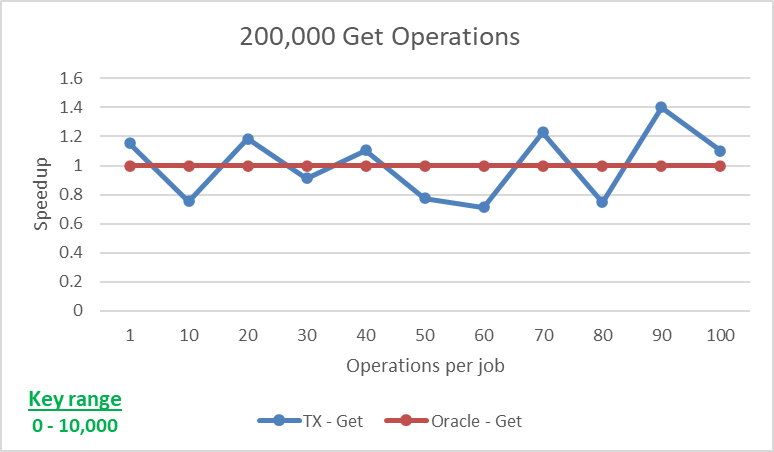


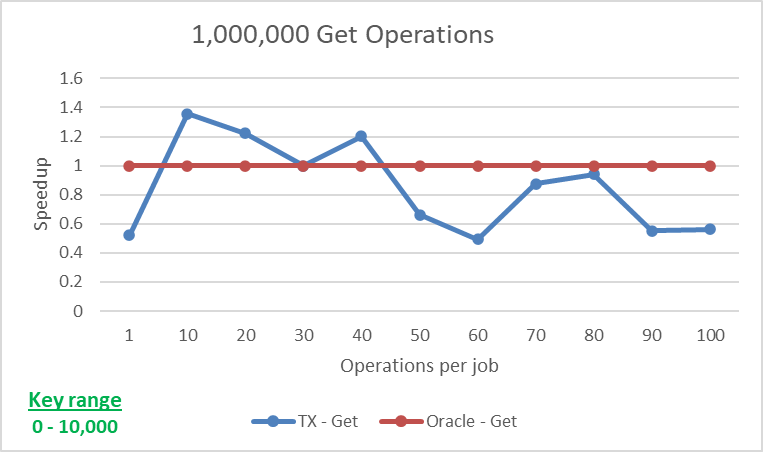


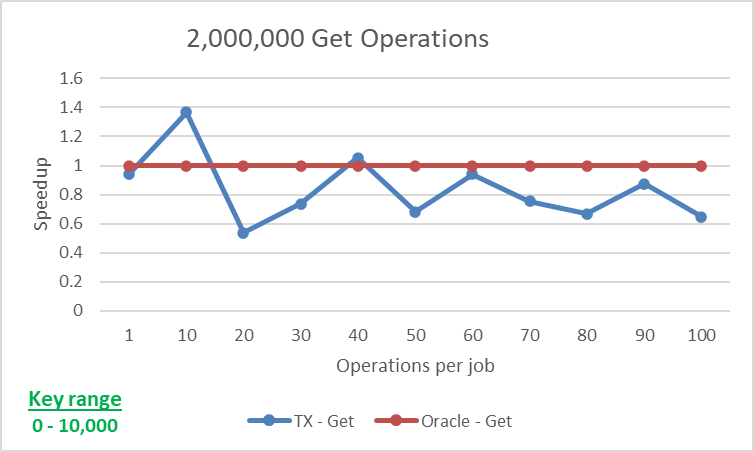
ניתן לראות בגרפים כי עבור הקטעים הקריטיים בגדלים הקטנים יותר (1-50), קיימת האצה משמעותית בריצה בהשוואה לריצה עם Concurrent HashMap. האצה זו מתקבלת כתוצאה מהמקביליות הרבה ששימוש בטרנזקציות יכול לספק, ומabort rate נמוך יחסית אשר גורם לניצול טוב של מקביליות זו.

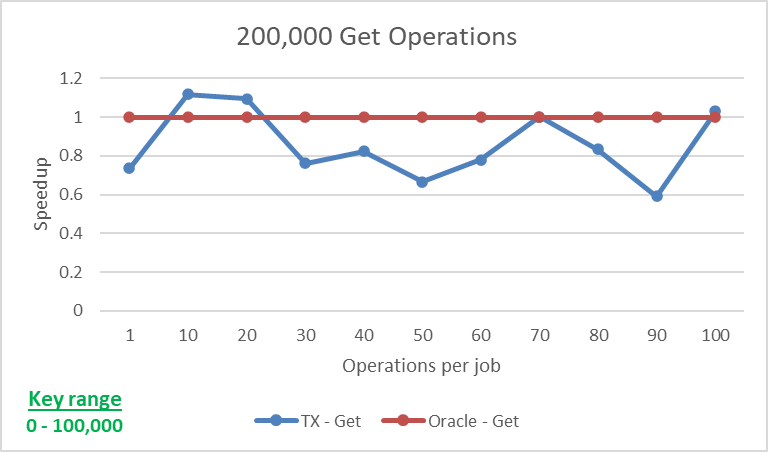
לעומת זאת, עבור גדלי קטע קריטי גדולים (60-100) ביצועי TXHashMap יורדים באופן ניכר בהשוואה   
ל-Concurrent HashMap. תופעה זו צפויה עקב היווצרות התנגשויות רבות בתצורת הבדיקה הזו. הגדלת ההתנגשויות יוצרת מספר רב של aborts וכתוצאה מכך הורדה ניכרת בביצועים.

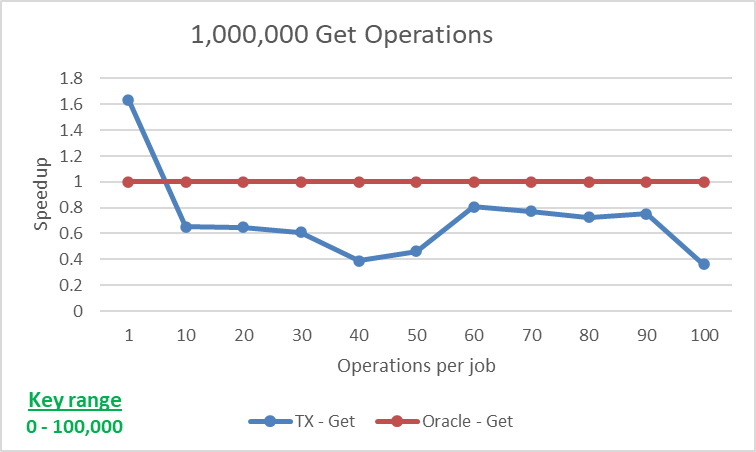
Get:

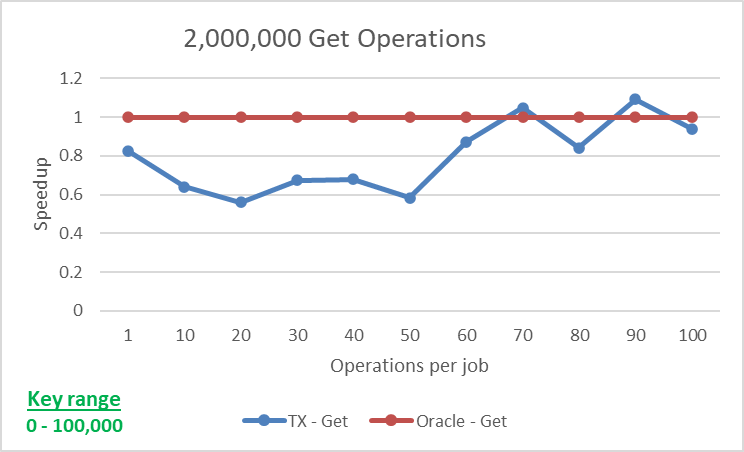








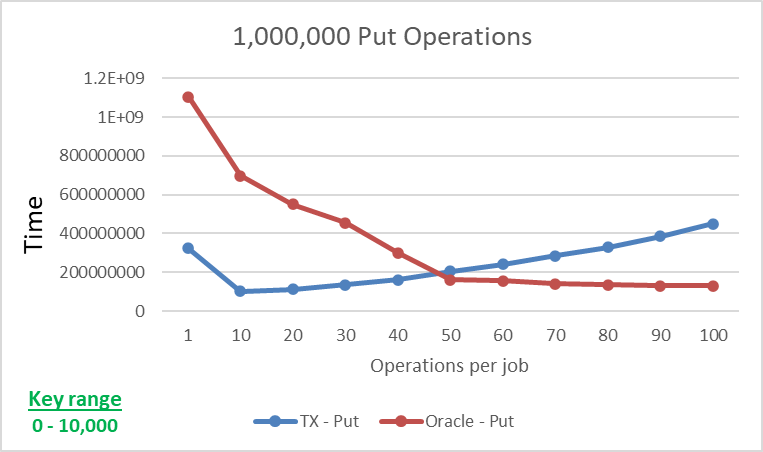




ניתן לראות בגרפים שקיים יתרון גורף של Concurrent HashMap על פני TXHashMap ברוב תצורות הבדיקה.  
אנו סבורים כי הבדלים אלו נובעים משני גורמים מרכזיים:

* תקורות שונות של TXHashMap כגון שימוש בload fence וקריאה של משתנים אטומיים במהלך חיפוש איברים במבנה הפנימי של הHashMap.
* קיום אופטימיזציות נוספות ב-Concurrent HashMap למקרים של התנגשויות מרובות ולמקרים נוספים, אשר משפרות את ביצועי פעולת ה get באופן ניכר במקרים אלו אך לא מומשו במבנה שלנו.

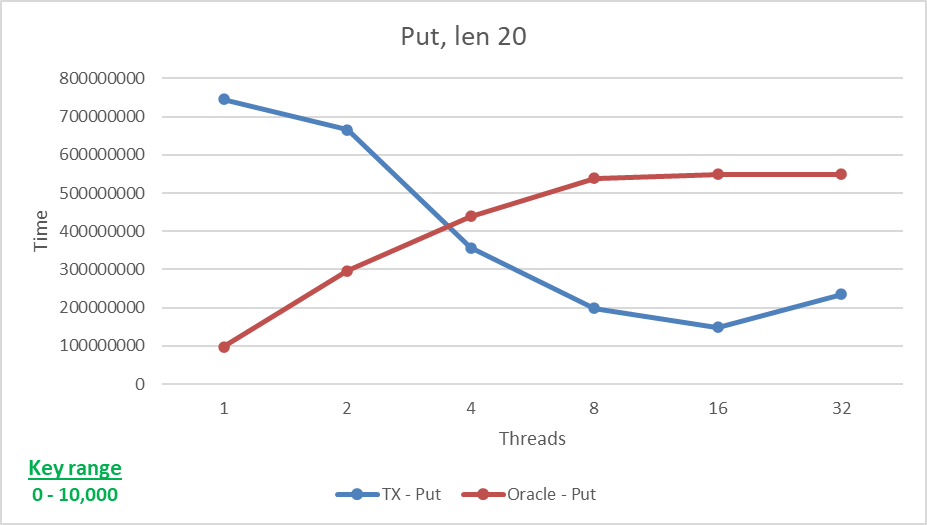
גרף לא מנורמל לדוגמא:



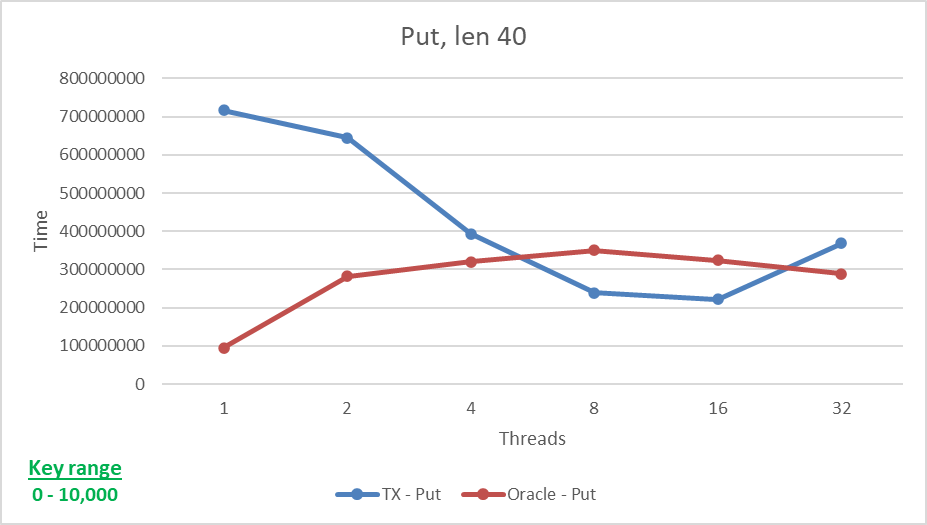
מהגרף ניתן לראות כיצד הספריות מושפעות מהשינוי בדרישת המקביליות. ככל שנדרשו יותר פעולות במקביל, כך עלה זמן הביצוע במקרה של TXHashMap, עקב הגדלת הטרנזקציות וקבלת התנגשויות רבות יותר. לעומת זאת, במקרה של Oracle, הגדלת הקטע הקריטי הקטינה את כמות הנעילות שיש לבצע במערכת ובכך זמן הביצוע השתפר.

*כתלות במספר החוטים:*

* עבור 1,000,000 פעולות עם קטע קריטי בגודל 20:



* עבור 1,000,000 פעולות עם קטע קריטי בגודל 40:



כמצופה, ניתן לראות מהגרפים שיפור משמעותי בביצועי TXHashMap Put עם הגדלת מספר החוטים, עקב הגדלת המקביליות.

סיכום התוצאות

באופן כללי, במקרה של טרזקציות ארוכות צפויה האטה משמעותית בביצועים עקב עלייה רבה במספר ההתנגשויות.  
לכן, הציפייה היא שבמקרים הללו ביצועי הספרייה הטרנזקציונית יהיו פחות טובים מאלו של הספרייה הסטנדרטית.

ראינו במקרה של Put כי הספרייה הטרנזקציונית נתנה ביצועים טובים יותר עבור טרנזקציות שאינן גדולות מאוד ובמקרה של טרנזקציות גדולות התקבלו ביצועים פחות טובים כמצופה.

במקרה של Get, ראינו ירידה בביצועים עבור רוב תצורות הבדיקה. ככל הנראה, מכיוון שפעולות get מהירות יותר, הגורמים שהוזכרו לעיל מורגשים כעת באופן משמעותי, בניגוד למקרה של put.

מימוש האופטימיזציות הקיימות ב-Concurrent HashMap, כגון הפיכת אינדקסים בעלי רשימות מקושרות ארוכות לעצים, יכול לשפר את התוצאות של TXHashMap.

# חלק 4 - סיכום ומסקנות

הצענו מספר אפשרויות למימוש טבלת ערבול כחלק מספריית TDSL. בחרנו במימוש בעזרת מערך פנימי של רשימות מקושרות, כאשר עבור כל רשימה מקושרת מוצמד מנעול ונשמר מספר הגרסה של השינוי האחרון ברשימה.  
מימוש זה נבחר מתוך ההנחה שהרשימות המקושרות קטנות מאוד במהלך זמן הריצה, כפי שנובע מתכונות טבלאות ערבול.

הוספנו תמיכה בעבודה עם מספר רב של טבלאות ערבול ע"י שינוי אופן שמירת נתוני הטרנזקציות.

כדי לשמר את רמת הביצועים כאשר מספר האיברים בטבלה גדל, היה צורך במימוש מנגנון הגדלת טבלה דינאמי. ראינו את הקשיים שנובעים במימוש מנגנון זה עקב הצורך בהחלפת הטבלה הפנימית באחת חדשה. בחנו מספר פתרונות לקשיים הללו, ולבסוף מימשנו את המנגנון עם הפתרונות שנבחרו.

כדי לאפשר לקוד Legacy לרוץ תוך שימוש בטבלת הערבול שלנו, הוספנו תמיכה במנגנון Singleton באופן דומה לשאר מבני הנתונים הקיימים בTDSL.

לבסוף יצרנו Benchmark במטרה להשוות את רמת הביצועים המתקבלת מטבלת הערבול שלנו אל מול מימוש סטנדרטי של טבלת ערבול מקבילית בג'אווה.  
ראינו כי בחלק מהמקרים מתקבלת האצה משמעותית כתוצאה משימוש בטבלת הערבול החדשה, ובמקרים אחרים האטה. הצגנו הסברים אפשריים לתוצאות והצעות לשיפור מקרי ההאטה.

הצעות להמשך

אחת הבעיות המרכזיות בשימוש בטרנזקציות היא הקנס הגדול בביצועים עבור ביצוע טרנזקציות ארוכות. ניתן למזער את השפעת בעיה זו באופן משמעותי על ידי הוספת תמיכה במנגנון הכינון (Nesting).

ניתן לייעל את פונקציית הResize ע"י ביצועה באופן מקבילי. ברגע שחוט יזהה כי מתבצע Resize על טבלת הערבול, במקום להמתין לסוף הפעולה, החוט יצטרף לביצועה באופן אקטיבי.

במימוש שלנו, השתמשנו ברשימות מקושרות לצורך פתירת בעיית ההתנגשויות הפנימיות בטבלת הערבול. ראינו כי פתרון זה יכול ליצור קשיים עקב הצורך במעבר על רשימות מקושרות משותפות ללא שימוש במנעולים. קיימים פתרונות נוספים לבעיית ההתנגשויות, למשל מספר דרגות ערבול. ניתן לבחון האם שימוש בפתרון התנגשויות חלופי מתאים יותר עבור המקרה של ספרייה טרנזקציונית לצורך שיפור הביצועים.

ראינו כי קיימות אופטימיזציות רבות נוספות בטבלאות הערבול הסטנדרטיות, כגון החלפת רשימות מקושרות גדולות בעצים לצורך חיפוש מהיר יותר. מימוש האופטימיזציות הללו יכול לשפר את ביצועי טבלת הערבול באופן ניכר.

# נספח א' ביבליוגרפיה

[1] Alexander Spiegelman, Guy Golan-Gueta, Idit Keidar – Transactional Data Structure Libraries.