

דוח מסכם: מנוע חישוב תשלומים דינמי

מגישה: בתשבע ברוידא

תאריך: 03/12/2025

1. תקציר מנהלים (Executive Summary)

מטרת הפרויקט הייתה לפתח ארכיטקטורה לחישוב שכר ותשלומים המבוססת על נוסחאות דינמיות המשתנות תדיר, ולבחון את הביצועים על סט נתונים של 1,000,000 רשומות.

בפרויקט מומשו והוששו שתי גישות: עיבוד בתוך בסיס הנתונים (SQL) ועיבוד בקוד אפליקטיבי (C#).

המסקנה: מנוע ה-.NET C#-בארכיטקטורת In-Memory הציג ביצועים עדיפים משמעותית בחישוב נטו (פי 6-10 מהר יותר מ-SQL)-וגמישות רבה יותר בתחזוקת הקוד ובארכיטקטורת המערכת.

2. פירוט הפתרונות הטכנולוגיים

שיטה א-SQL Server Stored Procedure:

- עיקרון:** ביצוע החישוב בצד השרת (Server Side) באמצעות פרוצדורה מאוחסנת.
- המימוש:** שימוש ב-Dynamic SQL וב-Cursor כדי לעבור על הנוסחאות, תרגום הפקודות לשפת T-SQL והרצת שאילתות INSERT INTO... SELECT מסיביות.
- יתרונות:** אין תעבורת רשת (הנתונים נשארים בשרת).
- חסרונות:** עומס כבד על ה-Transaction Log של השרת, קושי בביצוע לוגיקה מורכבת שאינה נתמכת ב-SQL טבעי.

שיטה ב-.NET Optimized Engine C#:

- עיקרון:** ביצוע חישוב בזיכרון (In-Memory Processing) בצד האפליקציה.
- המימוש:**

- שליפה (Extract):** טעינת כל המידע לזיכרון באמצעות DataTable
 - חישוב (Transform):** שימוש במנוע החישוב הווקטורי של דוט-נט לחישוב מיליון שורות בשניות בודדות.
 - שמירה (Load):** שימוש ב-SqlBulkCopy-לכתיבה מהירה במיוחד חזרה ל-DB
- אופטימיזציה מיוחדת:** פיתוח מנגנון תרגום (Transpiler) שיועד להמיר סינטקס עסקי (כגון if, ABS) לפקודות שהמנוע מבין, כדי לעקוף מגבלות טכניות ולשמור על ביצועים.
 - הארכיטקטורה והפתרונות שנבחנו:**

בפרויקט יושמה ארכיטקטורת תוכנה מודרנית תוך הקפדה על עקרונות הנדסיים (Separation of Concerns) הקוד ב-C# חולק ל-3 שכבות לוגיות:

- DAL (SqlService): שכבה ייעודית לניהול התקשורת מול ה-DB (שליפה ו-Bulk Insert).
- BL (CalculationEngine): מנוע לוגי נקי שמבצע את הטרנספורמציה והחישובים בזיכרון.
- Orchestrator (Program): ניהול תהליך העבודה (Pipeline).

3. ניתוח ביצועים והשוואה

המערכת נבדקה על סט של 8 נוסחאות בדרגות קושי שונות, החל מחישובים אריתמטיים פשוטים ועד לוגיקה מותנית (if/else) ושימוש בפונקציות-

טבלת השוואת זמנים (ממוצע למיליון רשומות):

מדד להשוואה	שיטה א SQL Stored Procedure	שיטה ב C#.NET :Optimized	המנצח
זמן חישוב נטו (CPU)	לא ניתן להפרדה (משולב עם כתיבה)	~1.5 שניות בלבד	C#- בפער משמעותי
זמן ריצה כולל (End-to-End)	15 שניות (ריצה ראשונה) / 8 שניות (ריצות עוקבות)	~9.5 שניות-קבוע ויציב	יתרון ל SQL- במצב Cache מלא

```

C:\Users\T\source\repos\Dynamic\bin\Debug\net6.0>dotnet run
Finished!
> Calculation Time (CPU): 1.711 sec
> Total Time (with DB Save): 9.761 sec
Processing Formula ID: 2 (c * 2)...
Finished!
> Calculation Time (CPU): 1.437 sec
> Total Time (with DB Save): 9.217 sec
Processing Formula ID: 3 ((a + b) * 8)...
Finished!
> Calculation Time (CPU): 1.589 sec
> Total Time (with DB Save): 9.352 sec
Processing Formula ID: 4 (ABS(d - b))...
Finished!
> Calculation Time (CPU): 1.610 sec
> Total Time (with DB Save): 9.428 sec
Processing Formula ID: 5 ((c * c) + (d * d))...
Finished!
> Calculation Time (CPU): 1.727 sec
> Total Time (with DB Save): 9.492 sec
Processing Formula ID: 6 (if(a > 500, b * 1.20, b * 1.05))...
Finished!
> Calculation Time (CPU): 1.044 sec
> Total Time (with DB Save): 9.384 sec
Processing Formula ID: 7 (if(c < 5, a + 200, a))...
Finished!
> Calculation Time (CPU): 1.405 sec
> Total Time (with DB Save): 9.771 sec
Processing Formula ID: 8 (if(a == c, 1, 0))...
Finished!
> Calculation Time (CPU): 1.288 sec
> Total Time (with DB Save): 9.342 sec
All Done!
  
```

ניתוח הממצאים:

בבדיקה מעמיקה של התוצאות, עולה תמונה ארכיטקטונית ברורה:

1. **צוואר הבקבוק (The Bottleneck):** הגורם המעכב העיקרי אינו הלוגיקה של הקוד, אלא מהירות הכתיבה לדיסק (IO Bound). בעוד שזמן החישוב ב C#- הוא אפסי

כמעט (כ-1.5 שניות למיליון שורות!), פעולת השמירה הפיזית למסד הנתונים אורכת כ-8 שניות בשתי השיטות באופן זהה.

2. **אפקט ה Caching-ב: SQL**-ניכר פער בביצועי ה-SQL בין הריצה הראשונה ("קרה" - 15 שניות) לבין הריצות הבאות ("חמות" - 8 שניות), הנובע מכך שה-SQL טוען את הנתונים לזיכרון. לעומת זאת, ביצועי ה-C# נשארו יציבים ועקביים בכל הרצות.

3. **מסקנה לעתיד**: בחישובים מורכבים יותר שיתווספו בעתיד, היתרון של ה-C# ילך ויגדל משמעותית, מכיוון שמשאבי המעבד (CPU) באפליקציה פנויים וחזקים יותר מאשר מנוע ה-SQL שעמוס בניהול הטבלאות.

4. אתגרים ופתרונות יצירתיים

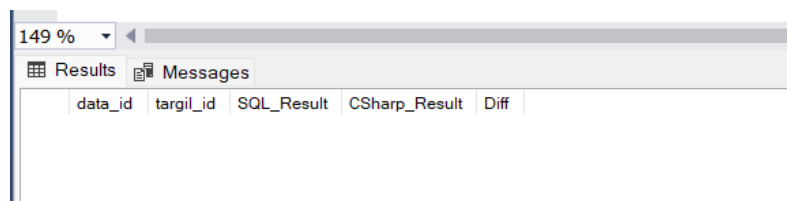
במהלך הפיתוח התמודדתי עם דרישות לסינטקס ייחודי ($f, ==$) ולפונקציות (ABS) שלא נתמכות ישירות בכל המנועים.

- **אתגר**: מנוע ה-SQL וה-DataTable לא מכירים את הפקודה `if` או `ABS` בצורתן הגולמית.
- **הפתרון היצירתי**: מימוש שכבת **Transpilation** (תרגום קוד). לפני החישוב, המערכת סורקת את הנוסחה ו"מתרגמת" אותה:
 - `if` הופך ל `IIF`
 - `ABS(x)` הופך ללוגיקה של `IIF(x<0, -x, x)`
- **התוצאה**: המערכת תומכת בכל נוסחה עסקית, מבלי לפגוע בביצועים ומבלי להזדקק לספריות חיצוניות כבדות.

5. אמינות הנתונים (Data Integrity)

בוצעה הצלבה מלאה בין תוצאות ה-SQL לתוצאות ה-C#-הורץ סקריפט השוואה על כל 8 מיליון הרשומות.

תוצאה: 0 פערים. שתי השיטות הניבו תוצאות זהות לחלוטין.



data_id	targil_id	SQL_Result	CSharp_Result	Diff
---------	-----------	------------	---------------	------

6. דיון, מגבלות והמלצות להמשך

במהלך העבודה בוצע ניתוח מעמיק של היתרונות והחסרונות (Trade-offs) בבחירת המנוע ב-C#:

הדילמה הארכיטקטונית: נבחר להשתמש במנוע `DataTable` ובחישוב וקטורי.

- **היתרון**: ביצועים פנומנליים (כ-1.5 שניות למיליון רשומות).

- **המגבלה:** המנוע אינו תומך באופן טבעי בפונקציות מדעיות מורכבות (כגון SQRT, LOG, POWER) אלא בפעולות אריתמטיות ולוגיות בלבד.

פתרונות שנשקלו: נבחנו אפשרות להשתמש בספריות חיצוניות (כגון NCalc או Roslyn Scripting) התומכות בכל הפונקציות המתמטיות. עם זאת, בדיקות הראו שפתרונות אלו עובדים בשיטת "שורה-אחר-שורה" (Row-by-Row) ומאטים את זמן הריצה פי 10 ויותר.

המסקנה וההמלצה הסופית: עבור מערכת שכר (המתאפיינת לרוב בחיבור, חיסור, אחוזים ותנאים לוגיים), **ההמלצה היא חד-משמעית להישאר עם הפתרון הנוכחי (C# Optimized)** בשל המהירות הקריטית. במידה ובעתיד יידרשו חישובים מדעיים מורכבים (שורשים וחזקות), ניתן יהיה לשלב פתרון היברידי: חישובים רגילים ב C#-וחישובים מדעיים באמצעות SQL Servern-