ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Χειμερινό Εξάμηνο 2022 – 2023

Γρηγορίου Κωνσταντίνα 1115201700025

Πολίτης Δημήτριος 1115201700128

Φλύρη Παναγιώτα 11152009002711

Εισαγωγή

Στο μάθημα “Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα” κληθήκαμε μέσα σε τρία τμήματα να υλοποιήσουμε ένα υποσύνολο μιας βάσης δεδομένων που διαχειρίζεται δεδομένα που βρίσκονται εξ ολοκλήρου στη RAM. Στο πρώτο μέρος ασχοληθήκαμε με την υλοποίηση του βασικού σχεσιακού τελεστή που είναι ο τελεστής ζεύξης ισότητας. Στο δεύτερο μέρος της εργασίας υλοποιήθηκε η σύζευξη (join) και η εφαρμογή φίλτρου σε επερωτήσεις (queries) στις οποίες συμμετέχουν περισσότεροι πίνακες με αρκετές στήλες. Και στο τρίτο μέρος της εργασίας θα παρουσιάσουμε πως επηρεάζεται η απόδοση του κώδικα των δύο προηγούμενων παραδοτέων με τη χρήση πολυνηματισμού, την ενσωμάτωση δομής Job Scheduler, παραλληλοποίησης και βελτιστοποιητή επερωτήσεων.

Περιεχόμενα

1.Προδιαγραφές μηχανήματος

Τα πειράματα που ακολουθούν έχουν πραγματοποιηθεί σε μηχάνημα με:

- i7 8700k
- 16GB RAM
- 128GB SSD

2.Παραλληλοποίηση

2.1. Job Scheduler

Ο job scheduler αποτελείται στην ουσία από δύο πράγματα. Το πρώτο αφορά ένα πίνακα από threads το οποίο κρατά τα threads που έχουμε στη διάθεση μας για να κάνουν τις δουλειες. Γενικά, το αρχικό thread που δημιουργεί τα υπόλοιπα έχει master/slave σχέση με αυτά δηλαδή είναι υπεύθυνο να τους αναθέτει δουλειές και να τα συγχρονίζει και δεν κάνει τις ίδες δουλειές με αυτά. Το δεύτερο αφορά μία λίστα από jobs (struct JobList) η οποία κρατάει τις δουλειές που χρειάζεται να γίνουν. Καταλαβαίνουμε ότι η λίστα των jobs δεν έχει σταθερό μέγεθος σε αντίθεση με τον πίνακα από τα threads, το οποίο σημαίνει ότι κάθε thread θα πρέπει εν δυνάμει να τρέξει παραπάνω από ένα job.

2.2. Συναρτήσεις Job Scheduler

- InitializeMultiThread: παίρνει δύο κενούς δείκτες, έναν για το JobList και ένα για το ThreadPool και αναλαμβάνει να δεσμεύσει χώρο για αυτά, να τα αρχικοποιήσει και για το ThreadPool να ξεκινήσει να τρέχει τα threads με τη συνάρτηση ThreadStart.

- DestroyMultiThread: παίρνει το JobList, το threadPool και τον αριθμό των thread και αναλαμβάνει να στείλει ένα job στα threads που τρέχουν το οποίο τους δίνει το σήμα για να τερματίσουν, περιμένει όλα τα threads να τελειώσουν και αναλαμβάνει να ελευθερώσει το χώρο που έχει δεσμευτεί για τις δύο αυτές δομές.

- ThreadStart: είναι η συνάρτηση την οποία τρέχουν τα threads ώστε να πραγματοποιήσουν τα jobs που μπαίνουν στη λίστα και όταν λάβουν σήμα (job) από τη destroyMultiThread να τερματίσουν.

2.3. Jobs

Το job που έχει υλοποιηθεί αφορά το join job που αναφέρεται στην εκφώνηση. Δέχεται δύο buckets και κάνει το join μεταξύ τους βάζοντας τα τελικά αποτελέσματα στην κοινή λίστα final. Επιπλέον έχει υλοποιηθεί το job loadTable το οποίο διαβάζει δεδομένα από κάποιο αρχείο και δημιουργεί ένα relation στη μνήμη. Λόγω του ότι τα υπόλοιπα κομμάτια του προγράμματος χρειάζονται τα relations αυτά για να τρέξουν έχουμε υλοποιήσει και ένα job barrier το οποίο μας εξασφαλίζει ότι πρώτα θα έχουν τελειώσει όλα τα load table που χρειάζεται να γίνουν και μετά θα συνεχίσουν τα υπόλοιπα κομμάτια.

3.Query Optimization

Το query optimization έγινε κρατώντας τα εξής στατιστικά για κάθε στήλη {l,u,f,d} των πινάκων, όπου:

- l =minimum
- u=maximum
- f=count
- d= distinct count

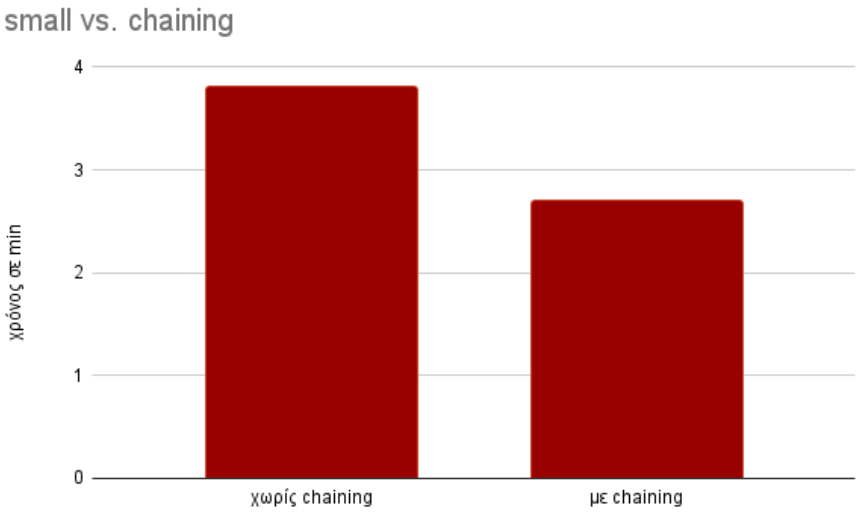
Για τον υπολογισμό του distinct count χωρίς μεγάλη δέσμευση μνήμης θεωρήθηκε maximum του hashtable το 5000000, όπως περιγράφεται στην εκφώνηση.

Με βάση αυτά τα στατιστικά η συνάρτηση cost υπολογίζει και επιστρέφει την αυξομείωση του ενδιάμεσου αποτελέσματος. Επιπλέον, ο αλγόριθμος που ακολουθήθηκε είναι ο DP-Linear-1({R1, . . . , Rn}) που περιγράφεται στο Building Query Compilers.

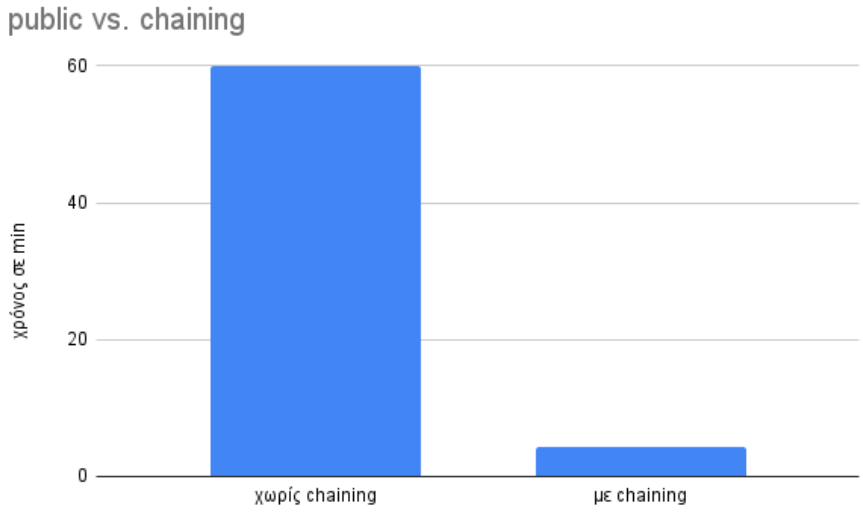
4.Πειράματα - Βελτιστοποιήσεις

4.1 Chaining

Σε αυτό το παραδοτέο εφαρμόστηκε chaining των διπλότυπών τιμών στο hopscotch. Πριν το chaining, το H του hopscotch θα έπρεπε να μεγαλώνει ταυτόχρονα με το μέγεθος του N για να φιλοξενήσει όλα τα διπλότυπα. Στα παρακάτω διαγράμματα φαίνεται η επίδραση που είχε αυτή η βελτίωση στο πρόγραμμα.



Πιο συγκεκριμένα, στο small input παρατηρούμε μια μικρή βελτίωση, αλλά όχι αξιοσημείωτη.

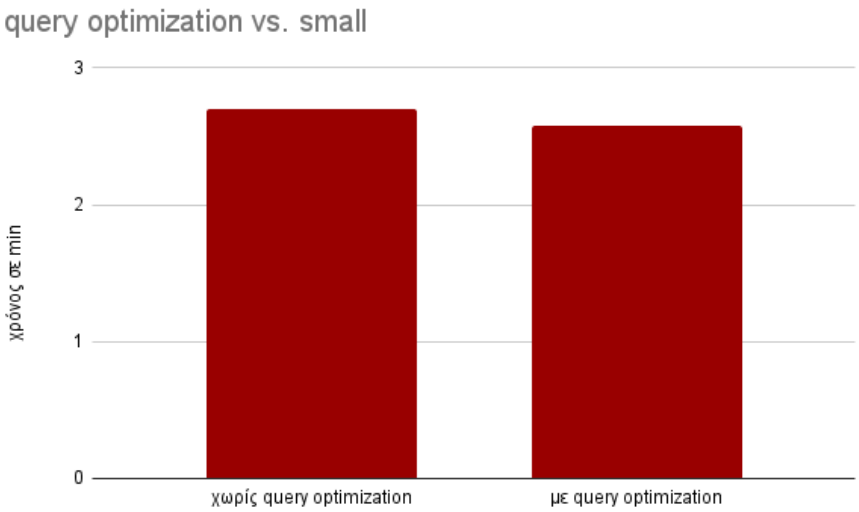


Στο public input, ωστόσο, βλέπουμε ραγδαία επιτάχυνση στον χρόνο (πάνω από 93%), το οποίο είναι λογικό αφού χωρίς chaining θα γίνουν πολλά resize στον hopscotch και η αύξηση του H θα καθυστερήσει το insert και αργότερα το search.

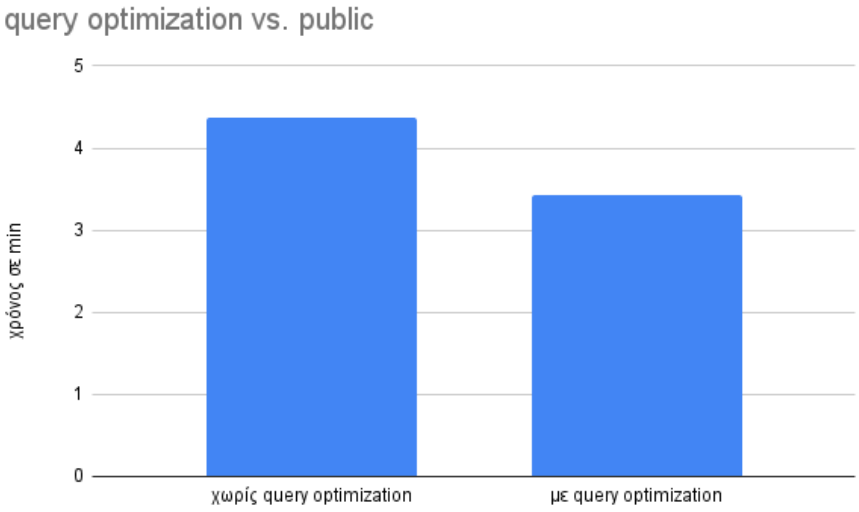
Εκτός του χρόνου, η υλοποίηση χωρίς chaining θα σπαταλήσει πολύ περισσότερη μνήμη, όπως είναι αναμενόμενο, καθώς το μέγεθος του πίνακα θα πρέπει να αυξηθεί αρκετά ώστε όλα τα διπλότυπα να χωρέσουν σε μια γειτονιά. Πιο συγκεκριμένα, όσο αφήσαμε να τρέξει η υλοποίηση χωρίς chaining είχε ήδη δαπανήσει 2GB περισσότερα από ότι το maximum της υλοποίησης με chaining.

4.2 Query optimization

Με την εφαρμογή του query optimization παρατηρούμε αξιοσημείωτη αύξηση στο πρόγραμμα.



Στο small input ειδικά, βλέπουμε μια μικρή βελτίωση (περίπου 7%).



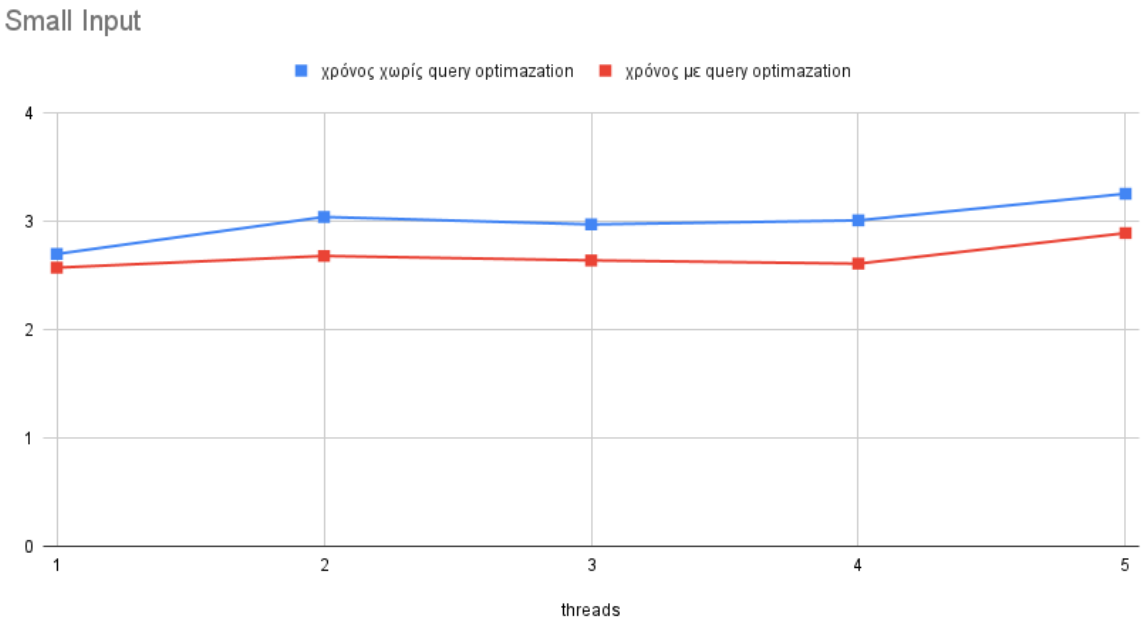
Στο public η επιτάχυνση είναι αρκετά μεγαλύτερη (περίπου 21%).

Κάτι ενδιαφέρον που παρατηρήθηκε σχετικά με το query optimization είναι η διαφορά που κάνει 1 ερώτημα. Πιο συγκεκριμένα, η βελτίωση του χρόνου (επι τις 100) είναι πολύ μικρότερη (σχεδόν 0), αν τρέξουμε το public χωρίς το 12 ερώτημα. Το 12ο ερώτημα του public είναι το πιο βαρύ ερώτημα καθώς είναι το μοναδικό self-join.

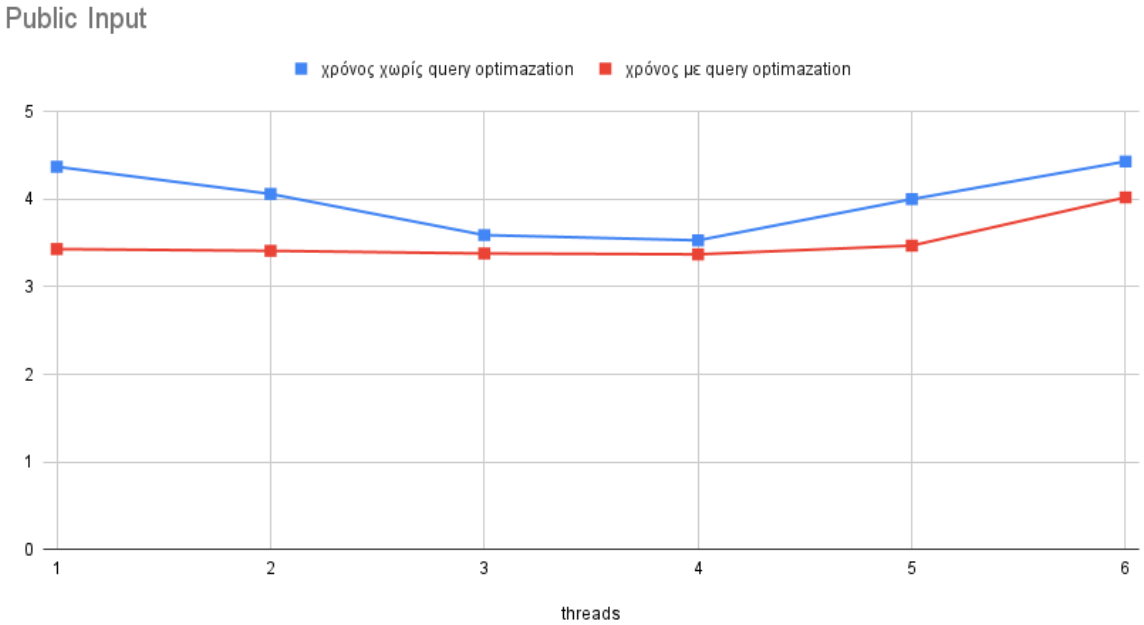
Όπως βλέπουμε από το διάγραμμα η ύπαρξη του 12 ερωτήματος καθορίζει αν το query optimization αξίζει το overhead του ή οχι. Δηλαδή, μέσα από το public και το small φαίνεται ότι το query optimization δικαιολογεί το overhead του με το πρώτο self-join που θα μας γλυτώσει.

4.3 Παραλληλοποίηση

Εφαρμόζοντας την παραλληλοποίηση στο join job παρατηρούμε στο small πως υπάρχει μια πολύ μικρή αύξηση στον χρόνο εκτέλεσης του προγράμματός μας. Αυτό συμβαίνει διότι τα δεδομένα δεν είναι αρκετά μεγάλα ώστε η παραλληλοποίηση να είναι αποδοτική και να μην επηρεάζεται αρνητικά από το overhead των threads.



Στο public αρχείο από την άλλη, παρατηρείται σημαντική μείωση του χρόνου ειδικά χωρίς τη χρήση του query optimization. Είναι σημαντικό επίσης να σημειωθεί πως στο συγκεκριμένο πείραμα το N = 2 οπότε αφού υπάρχουν 2^v buckets όσο τα threads αυξάνονται βλέπουμε μείωση στον χρόνο με βέλτιστη την περίπτωση των 2^v threads. Μόλις τα threads ξεπεράσουν αυτό το όριο ο χρόνος εκτέλεσης του προγράμματος αυξάνεται όλο και πιο πολύ.



5. Σύνοψη

Μετά τις αλλαγές και τα πειράματα που πραγματοποιήσαμε παρατηρήσαμε την μεγαλύτερη βελτίωση με την εφαρμογή του chaining στο hopscotch hash table. Το query optimization φαίνεται να βελτιώνει το πρόγραμμά μας, ειδικά στις περιπτώσεις που υπάρχει self join. Τέλος, στο κομμάτι των threads παρατηρήθηκε μικρή μείωση του χρόνου όσο ο αριθμός των threads δεν ξεπερνάει τον αριθμό των buckets διότι σε αυτή την περίπτωση υπάρχει μεγάλο overhead.