Ανάπτυξη λογισμικού για πληροφοριακά συστήματα

Γιάννης Ρέππας – 1115201500137
Γιάννης Τασσόπουλος – 1115201500155
Αλέξανδρος Πολυχρονόπουλος - 1115201600138

Περιεχόμενα

Final Report

Εισαγωγή

Η εργασία η υλοποιήθηκε για το μάθημα Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα. Το περιεχόμενο έχει να κάνει με τη δημιουργία ενός in-memory data base system. Σκοπός είναι ο χρήστης να δίνει στο σύστημα ένα σύνολο από αρχεία, τα οποία περιέχουν ακέραιους αριθμούς σε στήλες. Τα δεδομένα του χρήστη αποθηκεύονται στη μνήμη κατά στήλες για κάθε αρχείο που δίνει. Με βάση αυτά τα δεδομένα, ο χρήστης έχει τη δυνατότητα να υποβάλει ερωτήματα φίλτρων και ένωσης. Για κάθε ερώτημα που υποβάλει παίρνει και το ανάλογο αποτέλεσμα, το οποίο είναι το άθροισμα των στοιχείων των συγκεκριμένων στηλών των παραγόμενων πινάκων. Η εργασία υλοποιήθηκε σε 3 μέρη.

Το πρώτο μέρος έχει να κάνει με την υλοποίηση μιας αποδοτικής ζεύξης 2 πινάκων. Το δεύτερο μέρος έχει να κάνει με την υλοποίηση των ερωτημάτων που δίνει ο χρήστης, κατά την οποία για κάθε ερώτημα που δίνει ο χρήστης, χρησιμοποιείται μία ενδιάμεση δομή για την εκτέλεσή του. Το τρίτο μέρος της εργασίας έχει να κάνει με την προσθήκη παραλληλίας, αλλά και με την χρήση στατιστικών πληροφοριών για την ταχύτερη εκτέλεση των ερωτημάτων.

Παρόλο που η τελική υλοποίηση αφορά κυρίως το δεύτερο και το τρίτο μέρος της εργασίας, η περιγραφή της σε αυτό το pdf αρχείο θα γίνει για κάθε κομμάτι ξεχωριστά, με στόχο να περιγραφεί με λεπτομερή και με σαφή τρόπο όλη η διαδικασία που ακολουθήσαμε για να φτάσουμε στο τελικό αποτέλεσμα.

Μέρος Α - Δημιουργία ενός αποδοτικού τελεστή ζεύξης 2 πινάκων

Για τον τελεστή ζεύξης 2 πινάκων ακολουθήθηκε η λογική ότι πρώτα θα πρέπει οι δύο πίνακες να ταξινομηθούν και στη συνέχεια με τη βοήθεια της ταξινόμησης που πραγματοποιήθηκε, να εκτελεστεί ο τελεστής ζεύξης πολύ πιο γρήγορα από το να εκτελούταν χωρίς οι πίνακες να είχαν ταξινομηθεί.

Αρχικά λοιπόν, χρειάστηκε να υλοποιήσουμε μία συνάρτηση ταξινόμησης. Η συνάρτηση που επιλέχθηκε ήταν η radixsort. Η λογική της radixsort είναι η εξής: Σε κάθε βήμα, χωρίζουμε τα δεδομένα του πίνακα σε ομάδες, με βάση το πιο σημαντικό byte του κάθε στοιχείου. Έτσι λοιπόν, μετά το πρώτο βήμα, θα έχουμε τα δεδομένα του πίνακα χωρισμένα σε το λιγότερο 1 και το πολύ 2^8 ομάδες, με βάση τα περιεχόμενα του πίνακα. Η ομαδοποίηση σε κάθε βήμα γίνεται με τη βοήθεια 2 πινάκων μεγέθους 2^8 ο καθένας, στη χειρότερη περίπτωση. Η διαδικασία ομαδοποίησης συνεχίζεται για κάθε ομάδα ξεχωριστά, με βάση το επόμενο πιο σημαντικό byte. Η χρησιμότητα μιας τέτοιας μεθόδου είναι ότι δε χρειαζόμαστε για κάθε πίνακα χώρο παραπάνω από τον χώρο που καταλαμβάνει ο ίδιος επί 2. Τα αποτελέσματα μετά από κάθε ομαδοποίηση καταγράφονται σε ένα αντίγραφο του αρχικού πίνακα και για την επόμενη ομαδοποίηση στον πίνακα που μόλις χρησιμοποιήσαμε, μιας και τα προηγουμένως ομαδοποιημένα αποτελέσματα είναι πλέον άχρηστα. Η διαδικασία για κάθε ομάδα από δεδομένα θα σταματήσει σε 2 σενάρια. Το πρώτο είναι όταν η ομαδοποίηση γίνει και για το τελευταίο byte, αφού τότε προφανώς τα δεδομένα δε θα μπορούν να ομαδοποιηθούν κι άλλο και θα έχουμε έναν πίνακα με τα ίδια στοιχεία. Το δεύτερο σενάριο τερματισμού της ομαδοποίησης θα είναι όταν το μέγεθος των δεδομένων που θα πρέπει να ομαδοποιηθούν είναι τόσο μικρό, που στην ουσία δεν υπάρχει λόγος να χρησιμοποιηθεί κι άλλος πίνακας και απλά καλούμε μία quicksort για να εκτελεστεί μία γρήγορη ταξινόμηση. Η radixsort καλείται αναδρομικά για κάθε νέο επίπεδο ομαδοποίησης. Τα τελικά αποτελέσματα γράφονται πάντα στον αρχικό πίνακα μετά από μία quicksort. Δεν υπάρχει περίπτωση τα αποτελέσματα αυτά να "χαθούν" λόγω κάποιας άλλης quicksort ή radixsort, αφού τα όρια για κάθε ομαδοποίηση είναι πάντα σταθερά και δεν επηρεάζουν κάποια άλλη ομαδοποίηση και επομένως τα στοιχεία θα παραμείνουν στο ίδιο "κομμάτι" του πίνακα καθ' όλη τη διάρκεια της ταξινόμησης.

Έτσι λοιπόν, αφού οι δύο πίνακες είναι ταξινομημένοι, προχωράμε στην ένωση των 2 πινάκων. Αφού οι πίνακες είναι πλήρως ταξινομημένοι μπορούμε να κάνουμε merge join αρκετά γρήγορα. Γίνεται χρήση 2 pointer ώστε να εκμεταλλευθούμε την ταξινόμηση των πινάκων και κάθε φορά που βρίσκουμε ίδια τιμή περνάμε σε μία τελική δομή τα στοιχεία (rowId και δεδομένα). Στη συνέχεια, πηγαίνουμε στο επόμενο σημείο που μπορεί να βρούμε το επόμενο ταίρι. Τα αποτελέσματα της Merge Join γράφονται σε μία λίστα αποτελεσμάτων, όπου ο κάθε κόμβος της λίστας έχει συγκεκριμένο, σταθερό μέγεθος.Η όλη διαδικασία ονομάζεται SortMergeJoin και χρησιμοποιείται για το υπόλοιπο κομμάτι της εργασίας.

Μέρος Β - Δημιουργία περιβάλλοντος διατύπωσης ερωτημάτων από το χρήστη

Για το δεύτερο μέρος της εργασίας, ο χρήστης δίνει τα ονόματα των αρχείων που θέλει να συμμετέχουν στα ερωτήματα που θα υποβάλει. Τα αρχεία αποθηκεύονται στη μνήμη κατά στήλες. Έχουμε λοιπόν μία δομή στην οποία είναι αποθηκευμένοι όλοι οι πίνακες που έδωσε ο χρήστης.

Μετά τη δημιουργία της δομής, ο χρήστης ξεκινάει να υποβάλει ερωτήματα. Όταν δώσει τον χαρακτήρα F για είσοδο, του παρουσιάζονται τα αποτελέσματα για τα έως τώρα ερωτήματα που έδωσε. Το κάθε ερώτημα που δίνει ο χρήστης δεν είναι απλά ένα φίλτρο ή ένα join, αλλά ένα σύνολο από "υποερωτήματα", δηλαδή πολλά join και πολλά φίλτρα, τα οποία για τους πίνακες που έδωσε, ισχύουν ταυτόχρονα. Επομένως, για να κρατάμε τα δεδομένα για κάθε "υποερώτημα" που υποβάλει ο χρήστης και που υπολογίζει το πρόγραμμα, χρησιμοποιούμε μία ενδιάμεση δομή. Η ενδιάμεση δομή είναι μία απλά συνδεδεμένη λίστα. Για κάθε σκέλος του ερωτήματος που υποβάλει ο χρήστης, η ενδιάμεση δομή ενημερώνεται. Κάθε κόμβος της ενδιάμεσης δομής περιλαμβάνει έναν δυσδιάστατο πίνακα με τα rowIds και έναν μονοδιάστατο πίνακα, ο οποίος περιέχει τους πίνακες στους οποίους ανήκει η κάθε στήλη. Εφόσον το υποερώτημα το οποίο το πρόγραμμα καλείται να εκτελέσει περιέχει δεδομένα τα οποία δεν ανήκουν στην ενδιάμεση δομή, τότε σημαίνει πως θα έχουμε καινούριο κόμβο, διαφορετικά, αν η ενδιάμεση δομή έχει πληροφορίες έστω και για έναν πίνακα του υποερωτήματος, θα έχουμε update στους ήδη υπάρχοντες κόμβους. Σχεδόν για κάθε ερώτημα λοιπόν, παίρνουμε δεδομένα από την ενδιάμεση δομή.

Το ενδιαφέρον κομμάτι σε αυτό το σημείο είναι η υποχρεωτική αλλαγή που κάναμε στη δομή που περνάμε στη radixsort ως στοιχείο του πίνακα. Για να μπορέσουμε να ενημερώσουμε την ενδιάμεση δομή με τα νέα αποτελέσματα, θα χρειαστεί να γνωρίζουμε το σημείο που βρισκόταν το rowId στην ενδιάμεση δομή, σε περίπτωση που το στοιχείο υπάρχει 2 φορές στην ίδια στήλη. Επίσης, με αυτόν τον τρόπο χάνουμε ναι μεν χώρο, αφού αυξάνεται το μέγεθος της δομής που περνάμε στον πίνακα ως στοιχείο, αλλά ο χρόνος εκτέλεσης του προγράμματος μειώνεται αισθητά, αφού στην ουσία με το rowId της ενδιάμεσης δομής δεν γίνεται αναζήτηση κατά την ενημέρωσή της.

Έτσι λοιπόν, αν θεωρήσουμε ότι όλοι οι πίνακες που έδωσε ο χρήστης συμμετέχουν στην ερώτηση, όλα τα αποτελέσματα θα είναι αποθηκευμένα στο τέλος στην ενδιάμεση δομή. Έτσι λοιπόν, για τα rowIds που υπάρχουν στην ενδιάμεση δομή στο τέλος, ανατρέχουμε την αρχική δομή με τους πίνακες και παράγουμε τα τελικά αποτελέσματα, δηλαδή τα αθροίσματα των στηλών.

Τέλος, για να μειώσουμε το χρόνο εκτέλεσης των ερωτημάτων που υποβάλει ο χρήστης, αλλά και το χώρο που απαιτείται στη μνήμη για την εκτέλεσή τους, υλοποιήθηκε και η συνάρτηση swap_predicates(), η οποία τοποθετεί όλα τα φίλτρα (τόσο αριθμητικά φίλτρα όσο και φίλτρα ένωσης(λόγω της ενδιάμεσης δομής)) στην αρχή του ερωτήματος, έτσι ώστε να εκτελεστούν πρώτα. Οι λόγοι που κάτι τέτοιο βοηθάει στο χρόνο εκτέλεσης είναι προφανείς (τα φίλτρα παράγουν σχεδόν πάντα μικρότερα αποτελέσματα από τον αρχικό πίνακα, ενώ τα joins συνήθως τον αυξάνουν και έχουν μεγαλύτερο χρόνο εκτέλεσης).

Μέρος Γ - Παραλληλοποίηση και Στατιστικά

Η παραλληλοποίηση που προστέθηκε αφορά 2 κομμάτια του κώδικα.

Πρώτο μέρος:

Το πρώτο μέρος αφορά την παραλληλοποίηση των ερωτήσεων που δίνει ο χρήστης σαν είσοδο. Για κάθε ομάδα από queries, δημιουργείται ένα καινούριο thread, το οποίο εκτελεί το query. Μόλις το πρόγραμμα δεχθεί σαν είσοδο το χαρακτήρα F, τότε περιμένει μέχρι να τελειώσουν όλα τα queries που δόθηκαν για να παρουσιαστούν στο τέλος τα αποτελέσματα. Πέραν του thread, προστέθηκαν και δύο mutexes, τα οποία χρησιμοποιούνται για να βεβαιωθούμε πως στη λίστα των αποτελεσμάτων κάθε query θα γράφεται μόνο ένα στοιχείο και όχι δύο ή παραπάνω ταυτόχρονα. Έτσι, αποφεύγουμε τυχόν λάθη που μπορούν να προκύψουν κατά την εκτέλεση.

Δεύτερο μέρος:

Το δεύτερο μέρος αφορά την παραλληλοποίηση των αναδρομικών εκτελέσεων της radixsort. Ακολουθώντας παρόμοια λογική με αυτήν στην παραλληλοποίηση του πρώτου μέρους, κάθε αναδρομική κλήση μιας radixsort εκτελείται σε ένα thread. Όπως αναφέρθηκε και στο πρώτο μέρος, τα όρια κάθε ομαδοποίησης είναι σταθερά και δεν επηρεάζουν κάποια άλλη ομαδοποίηση. Για αυτόν το λόγο, οι ταξινομήσεις μπορούν και εκτελούνται παράλληλα. Τα join των thread γίνονται ανά επίπεδο, δηλαδή όταν κληθούν όλες οι αναδρομικές κλήσεις για κάθε ομάδα σε ένα πίνακα R, περιμένουμε μέχρι να ολοκληρωθούν, έτσι ώστε να προχωρήσουμε στο επόμενο επίπεδο.

Συγχωνεύσεις:

Αφού υλοποιήθηκαν οι δύο παραλληλίες ξεχωριστά, έγινε μια προσπάθεια να τρέξουν και οι δύο ταυτόχρονα. Παρόλα αυτά, για να μπορέσουν να παραχθούν σωστά αποτελέσματα, απαιτείται μια μεγάλη διαδικασία συγχρονισμού. Προσπαθώντας λοιπόν να υλοποιήσουμε αυτήν την συγχώνευση των παραλληλοποιήσεων και να αποφύγουμε τα λάθη, αναγκαστήκαμε να κάνουμε αρκετές «υποχωρήσεις», έτσι ώστε τα αποτελέσματα να είναι σωστά. Έγιναν δύο προσπάθειες, μία με την απλή χρήση thread και μία χρησιμοποιώντας έναν job scheduler

Threads:

Απλή συγχώνευση των 2 παραπάνω παραλληλοποιήσεων. Ο χρόνος εκτέλεσης μειώνεται, παρόλα αυτά υπάρχει κίνδυνος λαθών λόγω του συγχρονισμού.

Job scheduler:

Για την παραλληλοποίηση της RadixSort έχουμε κάνει έναν JobScheduler ο οποίος αποτελείται από μια λίστα FIFO. Η λίστα FIFO(First in First out) περιέχει κόμβους όπου μέσα τους έχουν έναν pointer n που είναι για να δείχνει στον επόμενο κόμβο και έχουν και ένα struct JobInfo. Το struct JobInfo έχει μέσα τα ορίσματα της RadixSort. Κάθε φόρα αντί να καλείται η RadixSort, καλούμε τον JobScheduler όποιος κάθε φόρα βάζει έναν νέο κόμβο στην λίστα(προσεσκτικά με τα κατάλληλα Mutex) με τα ορίσματα της RadixSort. Την πρώτη φορα που τον καλούμε κάνει initialize την λίστα μας, τα Mutex και τα Threads. Η συνάρτηση

των Threads έχει ως εξής: Μέσα σε ένα While(1) κάθε Thread παίρνει ένα Job(κόμβο της λίστας)(πάλι προσεκτικά με τα κατάλληλα Mutex), εκτός αν η λίστα είναι άδεια που τότε περιμένει να λάβει σήμα από τον JobScheduler ότι έβαλε Job στην λίστα, μόλις πάρει τον κόμβο τότε εκτελεί την συνάρτηση που πρέπει (την RadixSort ή την QuickSort). Τέλος όταν τελειώσει το πρόγραμμα βάζουμε όσους κόμβους στην λίστα όσα είναι και τα Threads ώστε μολις τους πάρουν τα Threads να κάνουν exit και να κάνουμε το join. Τέλος πριν το Merge φροντίζουμε να βεβαιωθούμε ότι κάθε Thread έχει τελειώσει την εργασία του για να μπορέσει να γίνει το Merge.

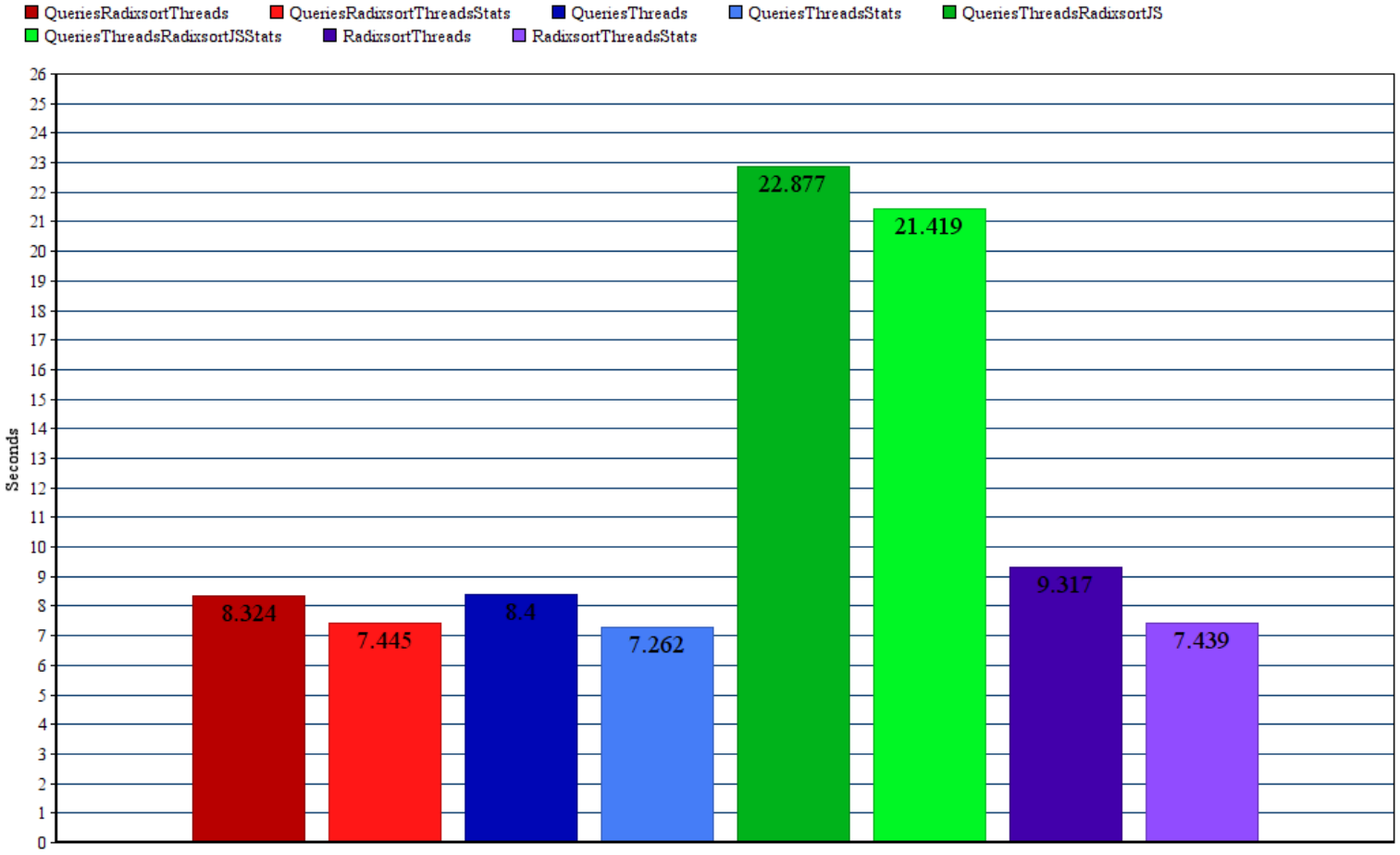
Προβλήματα και δυσκολίες: Όταν τρέχουμε το προγραμμα χωρίς ο JobScheduler δηλάδη τα Threads να εκτελούν την QuickSort και με ενα με ένα sleep(1) πριν το merge τρεχεί "ολόσωστα" το πρόγραμμα. Όταν το τρέχουμε χωρίς το sleep(1) και χωρις την QuickSort υπάρχει Segmentation Fault. Όταν το τρέχουμε με την QuickSort και χωρίς το sleep τρέχει κανονικά αλλά με λάθος αποτελέσματα. Οι σκέψεις μας είναι ότι αφού τρέχει κανονικά με την sleep(1) θα υπάρχει κάποιο πρόβλημα ότι κάποιο Thread δεν έχει τελειώσει την δουλεία του πριν το merge και έτσι μας πάει στο Segmetation Fault. Οι τρόποι που επιχειρήσαμε για να μην γίνει αυτό είναι στο να μπουν πρώτα όλες οι δουλειές στον JobScheduler και μετά να δημιουργηθούν τα threads και μέτα να βάλουμε στην λιστά κόμβους που θα κάνουν τα Threads να κλείσουν, οπότε να ανοίγουν και κλείνουν Threads σε κάθε sortMergeJoin. Ένας άλλος τρόπος είναι μόλις δούμε ότι η λίστα είναι άδεια πριν το Merge να βάζουμε στην λίστα κόμβους όσους και τα Threads που θα κάνουμε τα Threads να περιμένουν μέχρι να μπουν νέα Job στην λίστα για το επόμενο sortMergeJoin. Οι Υλοποιήσεις σε αυτά δεν προλαβαν να γινούν ορθά ώστε να τρέξει το πρόγραμα άλλα εγίναν πολλές προσπάθειες να γίνουν και έτσι έχουμε αφήσει τις "συναρτησεις/προσπάθειες" στο αρχείο.

<u>Μερικές πληροφορίες όσον αφορά τα στατιστικά για τη βελτίωση του χρόνου εκτέλεσης:</u>

Στην εργασία ενσωματώθηκαν και στατιστικά σχετικά με τις στήλες των πινάκων ώστε να αλλάζει η σειρά εκτέλεσης των predicates και να μειωθεί ο χρόνος εκτέλεσης. Αρχικά, ασχέτως με ποια σειρά διαβάζονται τα predicate προτεραιότητα έχουν τα φίλτρα είτε με σταθερή τιμή είτε φίλτρα στον ίδιο πίνακα. Αφού εκτελεστούν αυτά και παραχθούν τα στατιστικά τους (σύμφωνα με τις συναρτήσεις που δόθηκαν στην εκφώνηση) και ενημερωθούν οι αντίστοιχες δομές των πινάκων σειρά έχουν τα τζόινς. Αρχικά, δημιουργείται ένας γράφος με κέντρο τα predicates και όχι τους πίνακες για λόγους επιτάχυνσης, και ύστερα με βάση τον γράφο και τα αποτελέσματα των στατιστικών (ως κόστος ορίζουμε το πλήθος των στηλών που επιστρέφονται). Επιπλέον, ορίστηκε ένας περιορισμός στα στατιστικά που αφορούν τις μοναδικές τιμές των πινάκων (50.000.000) ώστε να μην σπαταληθεί πάρα πολύ μνήμη. Η εισαγωγή των περισσότερων στατιστικών φροντίσαμε να γίνει την στιγμή που διαβάζονται τα δεδομένα καθώς ο συνολικός χρόνος των στατιστικών δεν θα πρέπει να είναι μεγάλος
καθότι αποτελεί pre processing διαδικασία.

Διαγράμματα

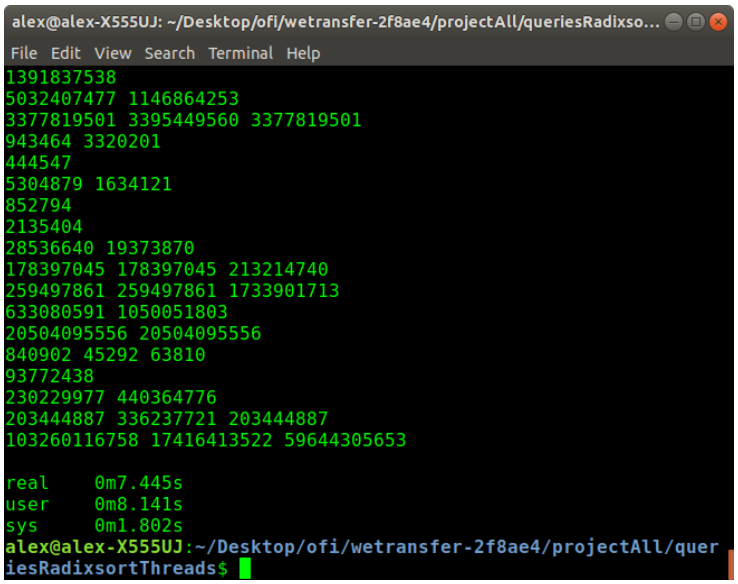
Σχετικά με τις τελικές σχεδιαστικές μας επιλογές (πέραν αυτών που αναλύθηκαν στα παραπάνω) δοκιμάσαμε ουσιαστικά 4 προσεγγίσεις ώστε να δοκιμάσουμε τον αλγόριθμο με αρκετές διαφοροποιήσεις. Η πρώτη έχει παραλληλοποίηση σε επίπεδο querries και radixsort η δεύτερη εφαρμόζει παραλληλοποίηση και μόνο σε επίπεδο querries η τρίτη εφαρμόζει την λογική του Job scheduler και η τελευταία εφαρμόζει παραλληλοποίηση μόνο σε επίπεδο radixsort.

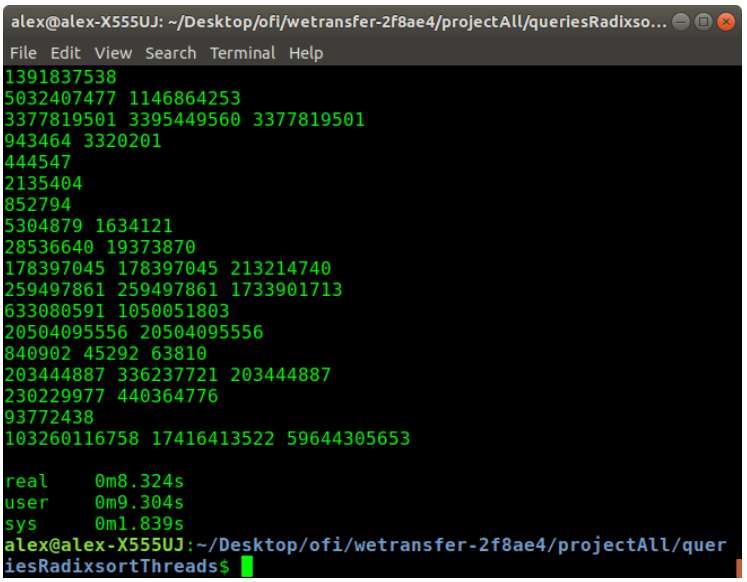


Στο παραπάνω διάγραμμα παρατηρείται και η βελτίωση του χρόνου εκτέλεσης εξαιτίας των στατιστικών (οι ράβδοι δίχως το Stats δεν υλοποιούν στατιστικά).

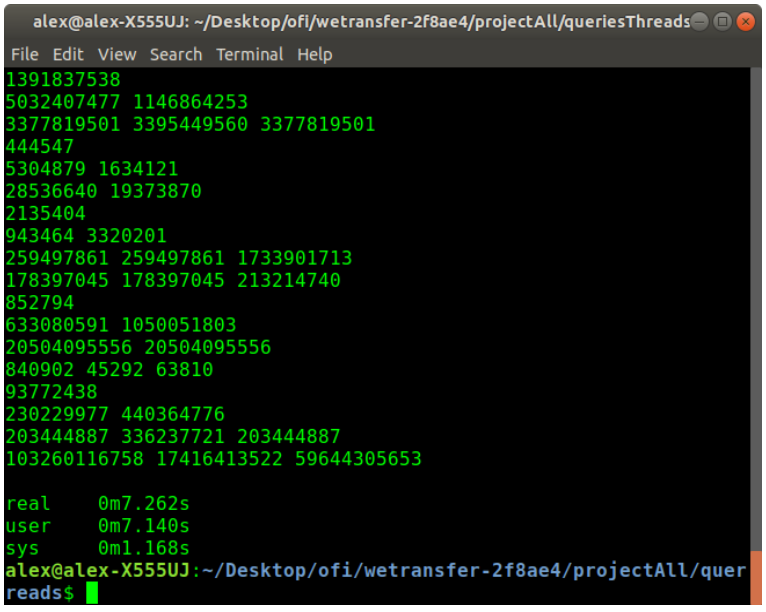
Εκτελέσεις σε bash

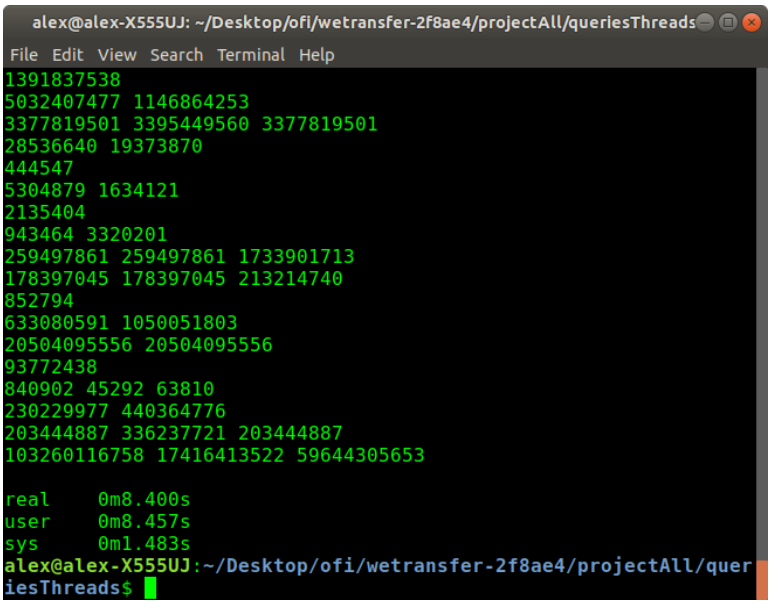
QueriesRadixsortThreads:



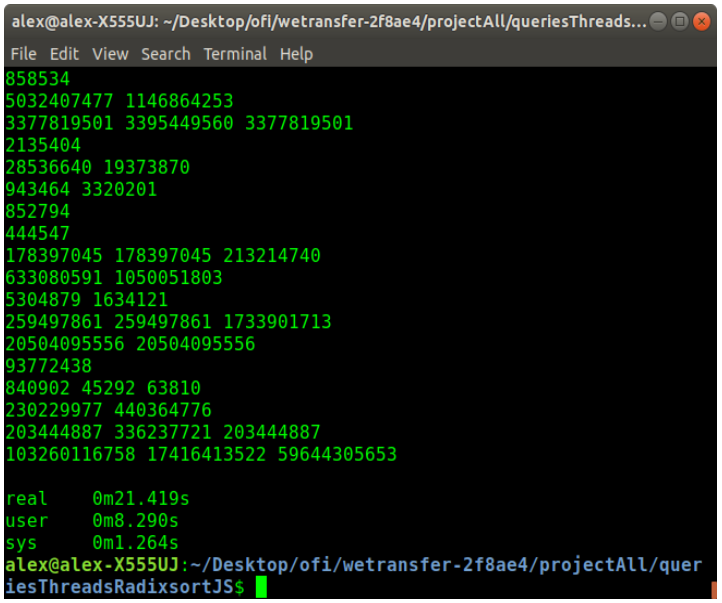


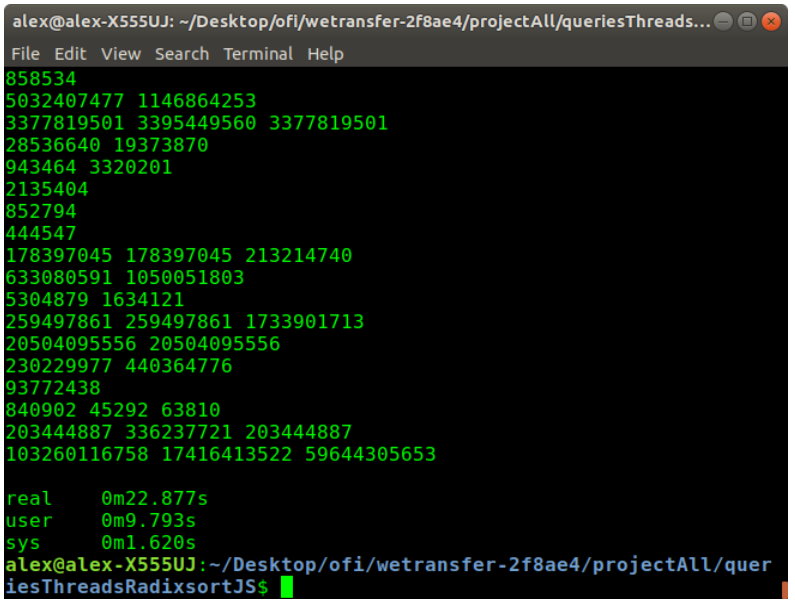
QueriesThreads:



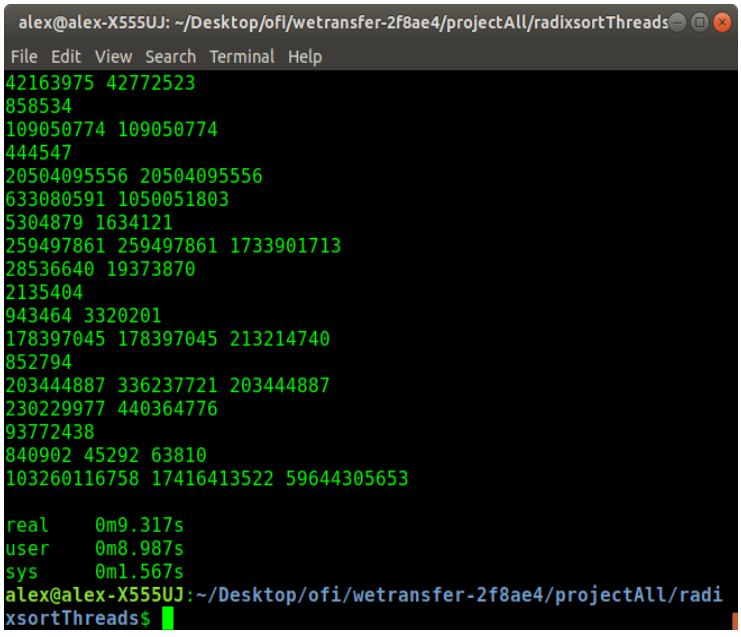


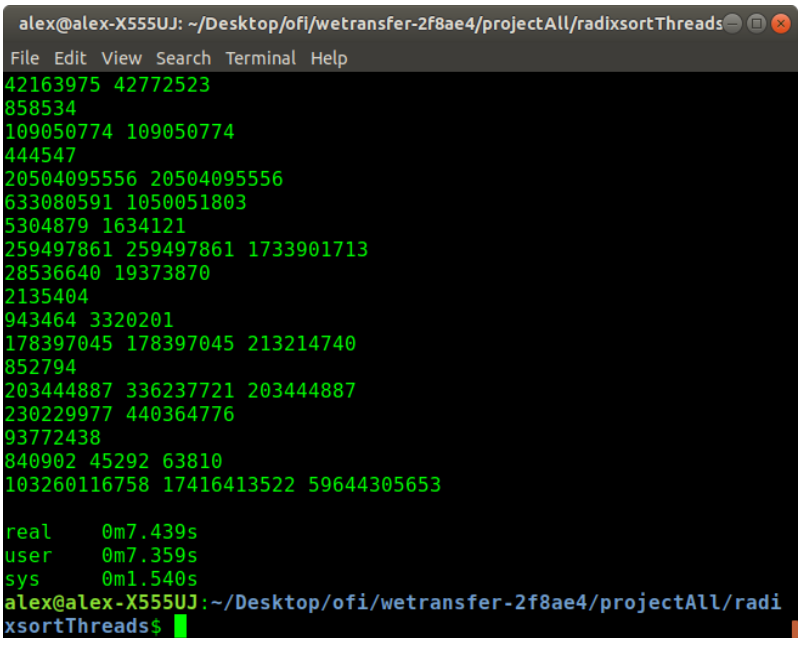
QueriesThreadsRadixsortJS:





RadixsortThreads:





Πληροφορίες μηχανήματος εκτέλεσης :

- CPU : Intel i7-6500U 2.50GHz -2.59GHz.
- RAM : 8.00 GB.
- OS : Linux Ubuntu L.T.S 18.04.
- Type : 64 bit.