Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Αναφορά για το 3ο Μέρος του Project

ACM Sigmod 2018

Συντελεστές:

Δημητρίου Γεώργιος - Σάββας: 1115 2018 00045

Τασιούλας Ραφαήλ - Χρήστος: 1115 2018 00191

Χατζηγιαννάκης Κυριαζής: 1115 2018 00211

Εισαγωγή

Υλοποιήθηκε, στα πλαίσια του μαθήματος το θέμα του διαγωνισμού ACM Sigmod τη χρονιά 2018. Χωρίστηκε σε 3 μέρη.

Στο 1ο μέρος, υλοποιήθηκε ο αλγόριθμος Partition Hash Join για πίνακες μιας στήλης και n-σειρών με στοιχεία ακέραιους. Στο πρόγραμμα δίνονται ως είσοδος δύο σχεσιακοί πίνακες με τα παραπάνω χαρακτηριστικά και εκτελεί την πράξη JOIN με τον παραπάνω αλγόριθμο μεταξύ των δύο πινάκων.

Στο 2ο μέρος, το make.sh μετατράπηκε σε Makefile για ευκολία χρήσης. Υλοποιήθηκαν οι συναρτήσεις του FileReader που διαβάζουν την είσοδο και τα queries. Επίσης

υλοποιήθηκαν οι συναρτήσεις που "κατακερματίζουν" τα queries, τα ομαδοποιούν σε batches και απαντάνε στα ερωτήματά τους. Ακόμη εισάγαμε τις ενδιάμεσες σχέσεις των JOIN. Τέλος, υλοποιήσαμε και τα tests (συμπεριλαμβανομένου και του harness) για τις περισσότερες συναρτήσεις του κώδικά μας.

Στο 3ο μέρος, το Hash table τροποποιήθηκε ώστε τα διπλότυπα να μπαίνουν σε αλυσίδες, πράγμα το οποίο βελτιώνει σημαντικά τον χρόνο εκτέλεσης. Επίσης, υλοποιήσαμε βελτιστοποιήσεις στη σειρά που εκτελούνται τα predicates σε ένα query χρησιμοποιώντας τα στατιστικά των στηλών όπως και την παραλληλοποίηση των υποεργασιών της PartitionedHashJoin σε threads με την βοήθεια Job Scheduler.

Αποτελέσματα

Εδώ παρουσιάζονται τα αποτελέσματα από την εκτέλεση του προγράμματος μαζί με όλους τους τρόπους βελτίωσης του χρόνου εκτέλεσης του προγράμματος. Εδώ παρουσιάζουμε τους τρόπους αυτούς ξεχωριστά και συνδυαστικά στο τέλος.

Separate chaining στα διπλότυπα

Όπως αναφέρθηκε στην εισαγωγή, ένας πολύ καλός τρόπος βελτίωσης του χρόνου, ήταν η τροποποίηση του Hash table ώστε να κάνει chaining στα διπλότυπα στοιχεία. Αυτή η ρύθμιση κάνει το πρόγραμμα να τρέχει περίπου 4 φορές πιο γρήγορα σύμφωνα με το harness test. Πιο συγκεκριμένα για το small τύπο input από 60 δευτερόλεπτα που έτρεχε στο 2ο μέρος, πλέον τρέχει στα 14, χωρίς παραλληλία.

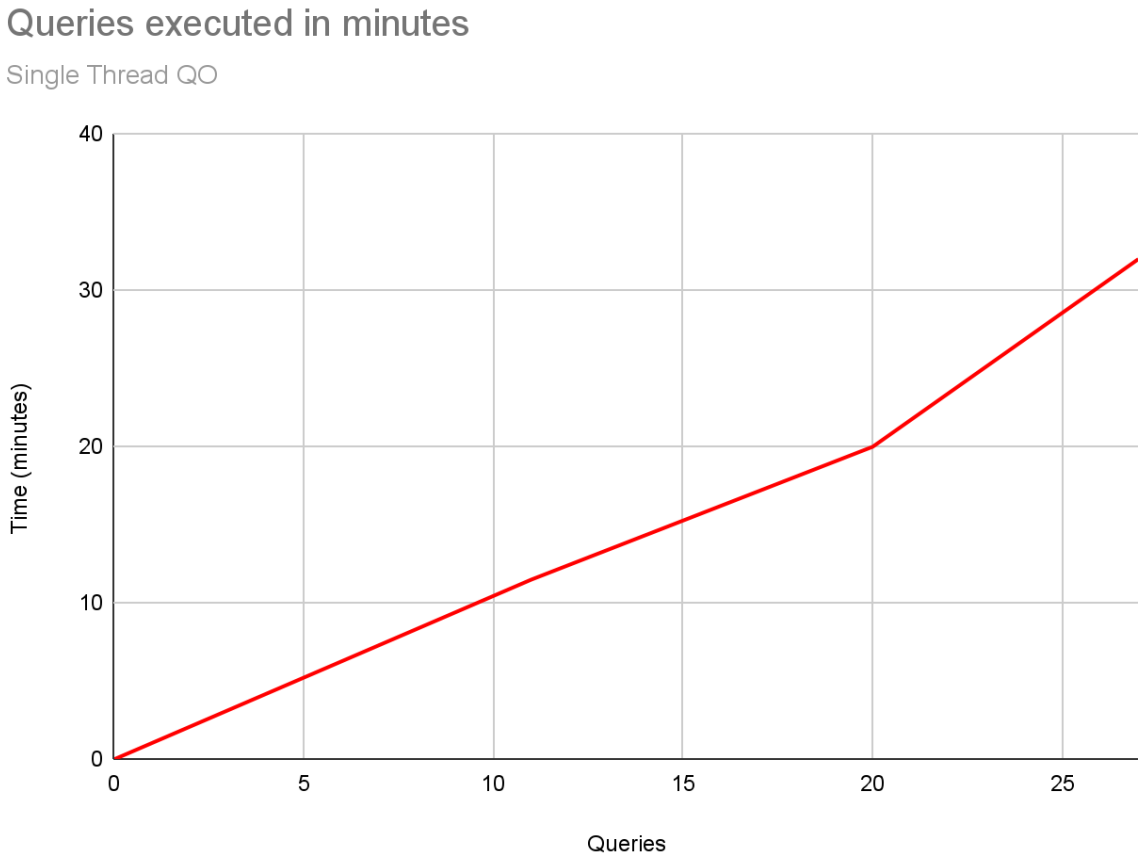
Οργάνωση των predicates σε ένα query

Αρχικά τα predicates, χωρίζονται σε 2 κατηγορίες. Η πρώτη κατηγορία είναι τα φίλτρα, τα οποία στο αριστερό μέλος έχουν την στήλη μιας σχέσης, ο τελεστής είναι, είτε “=”, είτε “<”, είτε “>” και το δεξί μέλος είναι ένας αριθμός. Η δεύτερη κατηγορία είναι οι ζεύξεις όπου και το αριστερό και το δεξί μέλος είναι στήλες σχέσεων και ο τελεστής είναι πάντα “=”.

Σε αυτό το τμήμα, οργανώνονται τα predicates για κάθε query ώστε να εκτελούνται με μία σειρά ή οποία θα διαρκέσει το λιγότερο δυνατό χρόνο. Για παράδειγμα, εάν έχουμε τα predicates σε ένα query, A⋈B, A⋈C, C⋈D και D>7000, όπου A,B,C και D στήλες σε σχέσεις που χρησιμοποιούνται στο query, θα πρέπει το πρόγραμμα να τα εκτελέσει με μία σειρά τέτοια ώστε ή συνολική τους εκτέλεση να διαρκέσει όσο λιγότερο γίνεται.

Για να επιτευχθεί αυτό, αρχικά τα φίλτρα έχουν προτεραιότητα, οπότε στο προηγούμενο παράδειγμα, το D>7000 θα εκτελεστεί πρώτο. Έπειτα, θα αναζητήσουμε τα στατιστικά κάθε στήλης ώστε να υπολογίσουμε τα ενδιάμεσα στατιστικά για κάθε join predicate και ενδιάμεσα αποτελέσματα μεταξύ πολλών join. Μέσω των στατιστικών και της χρήσης του δυναμικού αλγορίθμου αναζήτησης της γρηγορότερης σειράς από την εκφώνηση, θα βρεθεί έτσι το συνολικό κόστος σε πλειάδες που θα διασχίσει το πρόγραμμα αν εκτελέσει τα predicates με την κάθε τρέχουσα σειρά. Η σειρά που κοστίζει λιγότερο είναι η γρηγορότερη σε χρόνο.

Στην πραγματικότητα όμως, δεν παρατηρήθηκε αληθινή βελτίωση στην επίδοση του προγράμματος με την χρήση αυτού του αλγορίθμου. Όταν εκτελείται με το small workload, τρέχει με 15 δευτερόλεπτα, 1 δευτερόλεπτο παραπάνω από ότι χωρίς αυτό. Όταν εκτελείται με το public workload, παρατηρούμε ότι τρέχει αρκετά αργά αλλα φτάνει πιο μακριά από ότι με 4 threads(βλέπε παρακάτω) από 20 queries σε 27 queries. Στην γραφική παράσταση στην επόμενη σελίδα φαίνονται τα αποτελέσματα της public με query optimization.


Queries executed in minutes
Single Thread QO
40
30
20
10
0
Time (minutes)
0
5
10
15
20
25
Queries

Multithreading για κάθε Join Operation

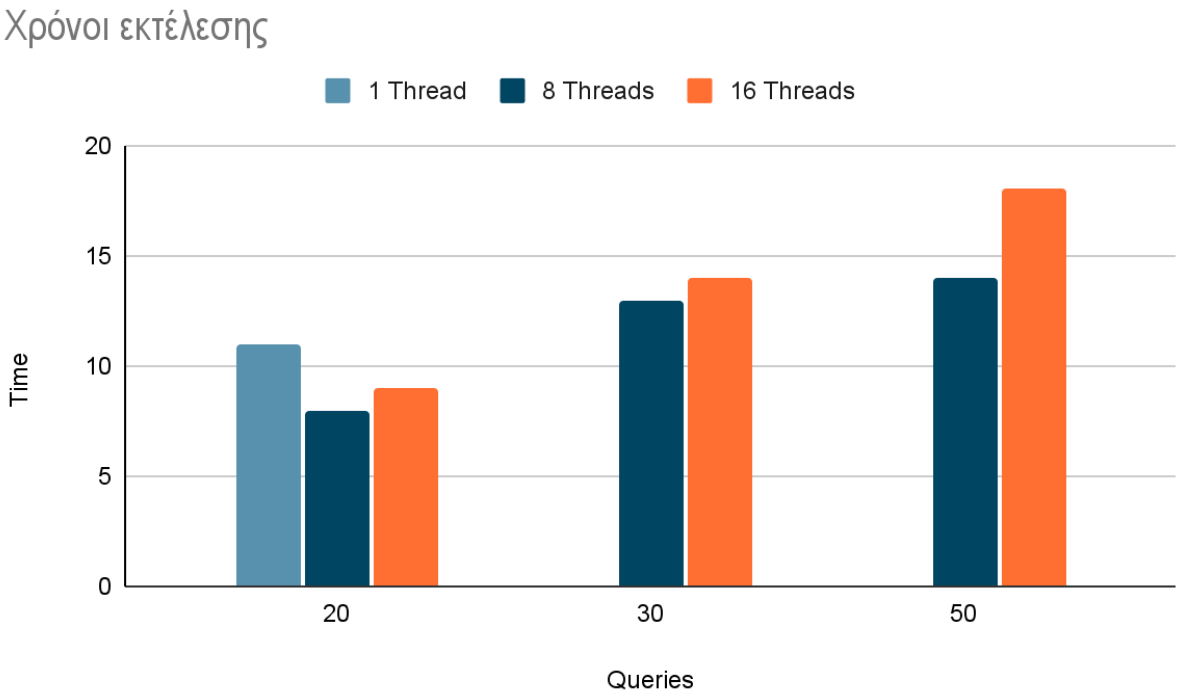
Σε αυτήν την βελτιστοποίηση, σκοπός μας είναι η παραλληλοποίηση όλων των υποεργασιών που γίνονται από την PartitionedHashJoin. Αυτό επιτυγχάνεται με την χρήση Job Scheduler ο οποίος προγραμματίζει και αναθέτει σε threads διαφορετικούς τύπους υποεργασιών για μία κλήση της PartitionedHashJoin.

Αυτό σημαίνει ότι οι επιδόσεις θα διαφέρουν σημαντικά από υπολογιστή σε υπολογιστή. Για παράδειγμα, στο υπολογιστικό σύστημα του Τμήματος με linux, το οποίο διαθέτει επεξεργαστή με 4 πυρήνες, θα έχει διαφορετικά αποτελέσματα από έναν άλλο υπολογιστή.

Στο small workload δεν παρατηρήθηκαν μεγάλες διαφορές επίδοσης αλλά στο public workload οι διαφορές είναι έντονες. Αναλυτικότερα, στον πίνακα και στο γράφημα της επόμενης σελίδας, βλέπουμε πόσο χρόνο και πόσα ερωτήματα κατάφεραν να υλοποιηθούν από την εφαρμογή από το public workload το οποίο δόθηκε από τον διαγωνισμό.

Είναι προφανές ότι το public workload δεν θα εκτελεστεί με καλούς χρόνους με 1 και 2 threads οπότε αυτές οι παράμετροι θα παραλειφθούν στα επόμενα στατιστικά.

Αποτελέσματα Multithreading για κάθε Join Operation			
	20 Queries	30 Queries	50 Queries
Threads = 4	11 λεπτα	kill	kill
Threads = 8	8 λεπτα	13 λεπτα	15 λεπτα
Threads = 16	9 λεπτα	14 λεπτά	18 λεπτα



Σημείωση: Παραλείφθηκαν τα ερωτήματα 12 και 30, διότι δεσμεύουν πάρα πολύ χώρο στη RAM, με αποτέλεσμα το τελικό kill των διαδικασιών του προγράμματος.

Παρατηρούμε ότι με 4 threads, η εκτέλεση του public μετά το query 20 είναι αδύνατη. Για αυτό η εκτέλεσή του προγράμματος με αυτό το input είναι αδύνατη σε υπολογιστές με

επεξεργαστές με 4 ή λιγότερους πυρήνες. Για αυτό, δοκιμάσαμε το πρόγραμμα σε έναν υπολογιστή με τα εξής χαρακτηριστικά:

Operating System

Windows 10 Education 64-bit (Το πρόγραμμα έτρεχε σε Windows Subsystem for Linux)

CPU

Intel Core i7 7820X @ 3.60GHz (8 Cores - 16 Threads)

Skylake-X 14nm Technology

RAM

16.0GB Dual-Channel @ 2133MHz (10-10-10-28)

Δοκιμάσαμε να τρέξουμε το πρόγραμμα με 8 και 16 threads. Υπάρχει σημαντική διαφορά χρόνων μεταξύ των 2 ρυθμίσεων λόγω του overhead που προκαλείται με 16 threads επειδή ο επεξεργαστής είναι 8πύρηνος. Για αυτόν τον λόγο συμπεραίνουμε ότι τα 8 threads για κάθε υποεργασία της PartitionedHashJoin είναι ο βέλτιστος αριθμός των threads.

Συνδυαστικά Αποτελέσματα

Εδώ θα παρουσιαστούν τα συνολικά αποτελέσματα με όλες τις βελτιστοποιήσεις και θα καθοριστεί ποιες είναι οι βελτιστοποιήσεις με τις οποίες το πρόγραμμα εκτελείται γρηγορότερα.

Το small workload δεν επηρεάζεται συνολικά από καμία βελτιστοποίηση πέρα από το separate chaining στα διπλότυπα.

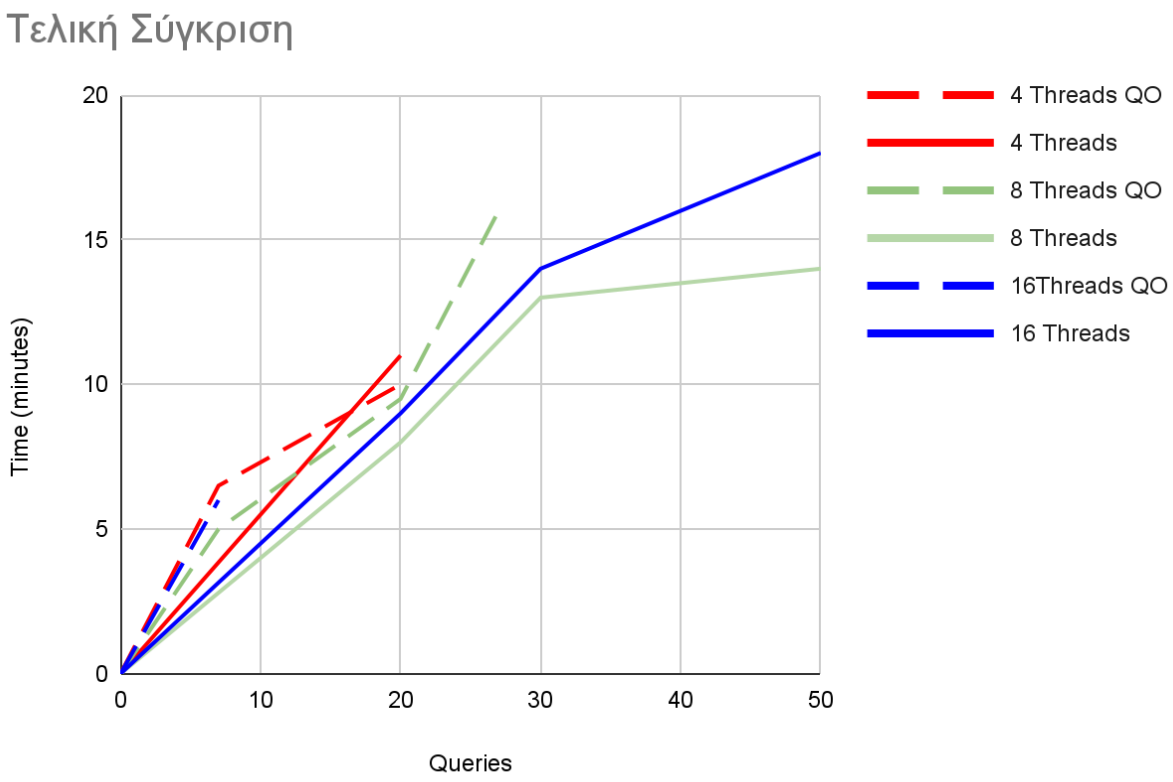
Σε όλες τις επόμενες υλοποιήσεις χρησιμοποιείται η βελτιστοποίηση separate chaining στα διπλότυπα μιας και είναι με διαφορά η πιο ισχυρή βελτιστοποίηση από αυτές που έχουν προταθεί.

Ο παρακάτω πίνακας έχει τα συνδυαστικά αποτελέσματα από το Optimization των Queries και του Multithreading της Join Operation για το public workload.

Αποτελέσματα Multithreading για κάθε Join Operation με Query Optimization			
	7 Queries	20 Queries	27 Queries
Threads = 4	6.5 λεπτά	10 λεπτά	kill
Threads = 8	5 λεπτά	9.5 λεπτά	16 λεπτά
Threads = 16	6 λεπτά	kill	kill

Παρατηρούμε ότι το Query Optimization κάνει το πρόγραμμα υπερβολικά αργό ώστε να σκοτώνεται μετά από 27 queries (χωρίς το ερώτημα 12). Παρατηρούμε επίσης ότι με 4 threads το πρόγραμμα σταματάει στα 20 queries και στα 16 threads σταματάει στα 7 queries. Δεν είμαστε σίγουροι εάν, αυτό οφείλεται στον όγκο του public input ή εάν οι πολλαπλές δοκιμές του προγράμματος “γονάτισαν” τον υπολογιστή στον οποίο εκτελούνταν.

Παρακάτω υπάρχει το γράφημα σύγκρισης των δύο βελτιστοποιήσεων με βάση threads queries και χρόνο. Οι διακεκομμένες γραμμές αναπαριστούν τις δοκιμές με Query Optimization.



Άρα από αυτό το γράφημα συμπεραίνουμε ότι ο καλύτερος συνδυασμός βελτιστοποιήσεων ο οποίος εκτελεί τα περισσότερα queries από το public workload σε λιγότερο χρόνο είναι οι εξής βελτιστοποιήσεις:

- Separate Chaining στα διπλότυπα
- Multithreading για κάθε Join Operation με 8 threads
- Χωρίς Query Optimization