Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Τελική Αναφορά

Μέλη ομάδας:

- **Ονοματεπώνυμο: ΙΑΤΡΟΠΟΥΛΟΥ ΣΤΑΥΡΟΥΛΑ**
  Α.Μ.: 1115201500048

- **Ονοματεπώνυμο: ΞΥΔΑΣ ΜΙΧΑΗΛ**
  Α.Μ.: 1115201500116

- **Ονοματεπώνυμο: ΣΤΑΗΣ ΒΑΛΕΡΙΟΣ**
  Α.Μ.: 1115201500148

Contents

Εισαγωγή

Περιγραφή του υλοποιημένου project

Σκοπός του project ήταν η υλοποίηση ενός προγράμματος που θα ικανοποιεί το πρόβλημα του προγραμματιστικού διαγωνισμού SIGMOD 2018. Το πρόβλημα είναι η ταχύτερη δυνατή επίλυση queries τα οποία περιέχουν ζεύξεις και φίλτρα.

Το πιο αργό κομμάτι σε αυτά τα queries είναι η αντιμετώπιση των ζεύξεων μεταξύ δύο διαφορετικών σχέσεων. Για αυτόν τον λόγο μας ζητήθηκε να υλοποιήσουμε την radix hash join.

-Πρώτο Μέρος Radix Hash Join (Reordering)

Στο πρώτο μέρος της Radix Hash Join, τμηματοποιούμε και τις δύο δοθείσες σχέσεις. Διαλέγουμε τη μικρότερη, στην οποία θα φτιάξουμε και ευρετήριο, και εκτιμάμε τον βέλτιστο αριθμό κουβάδων για την συνάρτηση κατακερματισμού της. Μετά εφαρμόζουμε την ίδια συνάρτηση κατακερματισμού και στην άλλη σχέση. Πιο λεπτομερώς, για την πρώτη σχέση:

- Διατρέχουμε τον πίνακα μία φορά για να βρούμε τη μέγιστη κι ελάχιστη τιμή του και κατασκευάζουμε έναν πίνακα από boolean με μέγεθος (max - min) αρχικοποιημένο σε 0. Στη συνέχεια τον ξαναδιατρέχουμε και, για κάθε τιμή του, ελέγχουμε τη θέση (value - min) στον boolean. Αν είναι αληθής, δεν κάνουμε τίποτα. Αν είναι ψευδής, την κάνουμε αληθή και αυξάνουμε τον μετρητή διακριτών τιμών, αρχικοποιημένο σε 0, κατά 1. Έτσι, αφού διατρέξουμε τον πίνακα έχουμε το πλήθος των διακριτών τιμών του. Σε περίπτωση που (max-min)>N, όπου N ένας αριθμός, περιορίζουμε τον boolean σε μέγεθος N και δουλεύουμε με mod. Σε αυτή την περίπτωση, ο αριθμός διακριτών τιμών dvalues που υπολογίζουμε είναι απλώς μια εκτίμηση. Αυτός ο αλγόριθμος είναι ο ίδιος που χρησιμοποιείται στη δειγματοληψία των σχέσεων. *

- Με βάση το dvalues, τη διαθέσιμη μνήμη και το μέγεθος κάθε πλειάδας του πίνακα, υπολογίζουμε το βέλτιστο αριθμό κουβάδων έτσι ώστε κάθε κουβάς, μαζί με τις απαραίτητες δομές του ευρετηρίου, να χωράει στη μνήμη.

- Υπολογίζουμε το ιστόγραμμα του πίνακα και την τιμή τιμή κατακερματισμού της κάθε πλειάδας του. Η συνάρτηση κατακερματισμού είναι ένα απλό hash_value = value % 2^n.

- Με βάση το ιστόγραμμα, κατασκευάζουμε τον πίνακα psum όπου, για κάθε κουβά, κρατάμε την θέση που αρχίζουν τα στοιχεία του αφού αναδιατάξουμε τον αρχικό πίνακα κατά hash_value.

- Μεταγράφουμε τον πίνακα σε έναν καινούριο, τον reorderedR, με τα στοιχεία του ταξινομημένα κατά hash_value.

Επαναλαμβάνουμε τη διαδικασία και για τη δεύτερη σχέση, παραλείποντας το πρωτό βήμα. Χρησιμοποιούμε τον ίδιο αριθμό κουβάδων με την πρώτη σχέση.

*Για οικονομία χρόνου, ελέγχουμε μόνο το πρώτο 1/K κομμάτι του πίνακα και πολλαπλασιάζουμε με K για να έχουμε την εκτίμησή μας.

-Δεύτερο Μέρος Radix Hash Join (Indexing)

Στο δεύτερο μέρος δημιουργούμε το ευρετήριο. Αρχικά, αποφασίζουμε ένα hash2 με βάση το μέγεθος της cache αλλά και μια μεταβλητή που λέγεται AVAILABLE_CACHE_SIZE που εξαρτάται από την τιμή της h1 που υπολογίστηκε στο 1ο μέρος (reordering). Στην συνέχεια δημιουργούμε το chain array που είναι μια αλυσίδα από τιμές που έχουν το ίδιο hash2 καθώς και ένα bucket array που θα περιέχει το πού τελειώνει η κάθε αλυσίδα για κάθε διακριτή τιμή της h2. Ο συνδυασμός της αλυσίδας με το bucket array αποτελούν το ευρετήριό μας.

Λαμβάνουμε υπόψιν την εκτιμώμενη διαθέσιμη cache. Σε περίπτωση που δεν χωράει ολόκληρο το ευρετήριο στην cache το χωρίζουμε σε μια αλυσίδα ευρετηρίων όπου το κάθε ευρετήριο έχει την δικιά του hash2.

-Τρίτο Μέρος Radix Hash Join (Searching)

Έχοντας πλέον το ευρετήριο για το ένα relation και τον reodered για το άλλο μπορούν να υπολογιστούν τα τελικά αποτελέσματα της ζεύξης. Για κάθε bucket του reordered οδηγούμαστε σε ένα συγκεκριμένο bucket του ευρετηρίου. Ύστερα για κάθε τιμή υπολογίζουμε την h2 η οποία μας οδηγεί στο τέλος της κάθε αλυσίδας. Ακολουθούμε αυτήν την αλυσίδα βρίσκοντας τα αποτελέσματα τα οποία και προσθέτουμε σε έναν buffer μεγέθους 128Kb. Όταν ο buffer γεμίσει τον αποθηκεύουμε σε μια λίστα.

Η τελική έξοδος είναι μια λίστα από buffers που περιέχουν ζευγάρια από rows που ικανοποιούν τις προϋποθέσεις του join.

Δεύτερο Μέρος: Εφαρμογή σε queries

Στο 2ο μέρος καλούμαστε να φτιάξουμε ένα πρόγραμμα το οποίο δέχεται batches από queries, τα εκτελεί και επιστρέφει τα αποτελέσματα.

Αρχικά, φορτώνονται στην μνήμη οι σχέσεις που συμμετέχουν στο query. Ύστερα, γίνεται το parsing των queries τα οποία περιέχουν φίλτρα (=, >, <) και ζεύξεις μεταξύ των παραπάνω σχέσεων δημιουργώντας το κατάλληλο struct Query.

Εκτελούνται πρώτα τα φίλτρα και στη συνέχεια οι ζεύξεις. Σε κάθε πράξη (φίλτρο ή ζεύξη) δημιουργείται ή ενημερώνεται ένας ενδιάμεσος πίνακας ο οποίος διατηρεί τις γραμμές που περνάνε την πράξη. Υλοποιούνται κατάλληλα όλες οι δυνατές πράξεις μεταξύ των intermediates:

- Φίλτρο σε relation που δεν ανήκει σε intermediate
- Φίλτρο σε relation που ανήκει σε intermediate
- Join μεταξύ 2 relations που δεν ανήκουν σε intermediate
- Join μεταξύ ενός relation που ανήκει σε intermediate και ενός που δεν ανήκει
- Join μεταξύ 2 relations που ανήκουν στον ίδιο intermediate (λειτουργεί σαν φίλτρο)
- Join μεταξύ 2 relations που ανήκουν σε διαφορετικό intermediate

Τέλος, υπολογίζονται τα checksums που μας ζητούνται δηλαδή το άθροισμα των values των γραμμών ενός relation που υπάρχει στο τελικό intermediate (Έχουμε ως δεδομένο από την εκφώνση του διαγωνισμού πως δεν υπάρχουν cross-products).

Σκοπός της αναφοράς

Στο 3ο μέρος της εργασίας μας ανατέθηκε η βελτιστοποίηση του κώδικα, που αναπτύχθηκε στα πρώτα 2 μέρη, από άποψη χρόνου και μνήμης.

Τα tasks που δόθηκαν προς υλοποίηση ήταν:

- Threading σε επίπεδο reordering
- Threading σε επίπεδο αποτελεσμάτων
- Query Optimization μέσα από τον αλγόριθμο Job Enumeration

Τα extra tasks που υλοποιήθηκαν ήταν:

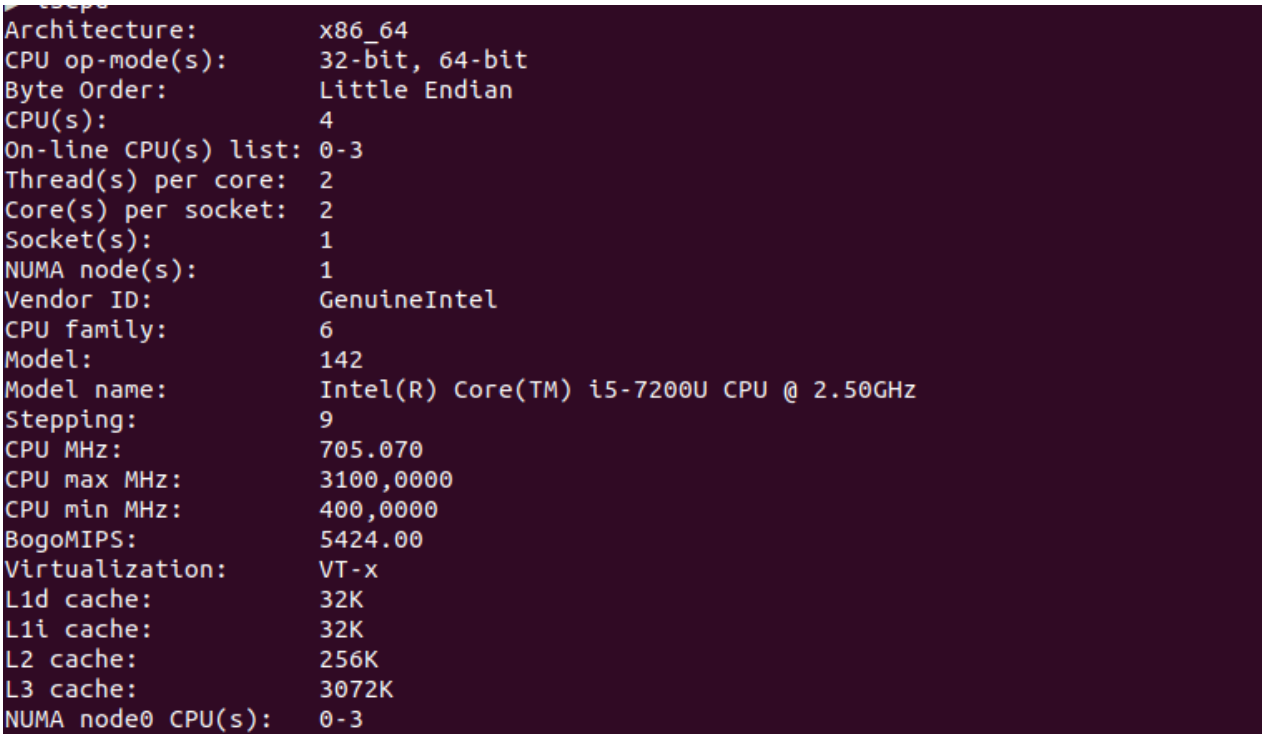
- Ομαδοποίηση σε πακέτα των rows των relations που δίνονται ως input στην radix hash join
- Υλοποίηση «Αποθήκης» για τα indexes που χρησιμοποιούμε

Ο σκοπός της αναφοράς είναι η μελέτη των παραπάνω tasks ως προς το χρονικό κυρίως όφελος παρατηρώντας παράλληλα και την χρήση μνήμης.

Χαρακτηριστικά υπολογιστή

Οι μετρήσεις έγιναν σε λάπτοπ με τα παρακάτω χαρακτηρηστικά:

- Επεξεργαστής: `Intel Core i5 7200U 2.5GHz 4 cores`
- RAM: `8GB DDR4`
- Λειτουργικό: `Ubuntu gnome 18.04`



Εκτέλεση της εντολής lscpu

Χρόνοι εκτέλεσης (πριν από optimization)

Το πρόγραμμα **χωρίς** threading και query optimization εκτελείται σε:

- 1750ms για το small dataset
- 155000ms για το public dataset

Χρήση μνήμης (πριν από optimization)

Για την μνήμη μετράμε τα συνολικά allocations (**όχι το peak**) του small dataset. Στο public λόγω δομών του valgrind η εκτέλεση του προγράμματος με valgrind είναι αδύνατη.

Επίσης στις μετρήσεις του προγράμματος με threading δεν παρατηρείται αξοισημείωτη διαφορά στην χρήση μνήμης με την εναλλαγή του αριθμού των threads.

- Συνολικά allocations: 1389MB

Βέβαια αυτή η μέτρηση δεν είναι απόλυτα αντιπροσωπευτική της χρήσης μνήμης. Ιδανικά θα γινόταν μέτρηση του peak της μνήμης αλλά αντιμετωπίσαμε προβλήματα στην χρήση των διαθέσιμων tools και heap profilers.

Threading

Job Scheduler

Ο job scheduler που υλοποιήθηκε είναι μια απλή ουρά με έναν writer (main process) και πολλούς readers (threads). Ο συγχρονισμός επιτεύχθηκε μέσα από mutexes και conditional variables.

Ο writer γράφει το job στην παρακάτω μορφή

```
struct Job {
    void (*function)(void *);
    void *argument;
};
```

Το οποίο και εκτελείται από το πρώτο ελεύθερο thread που θα το διαβάσει.

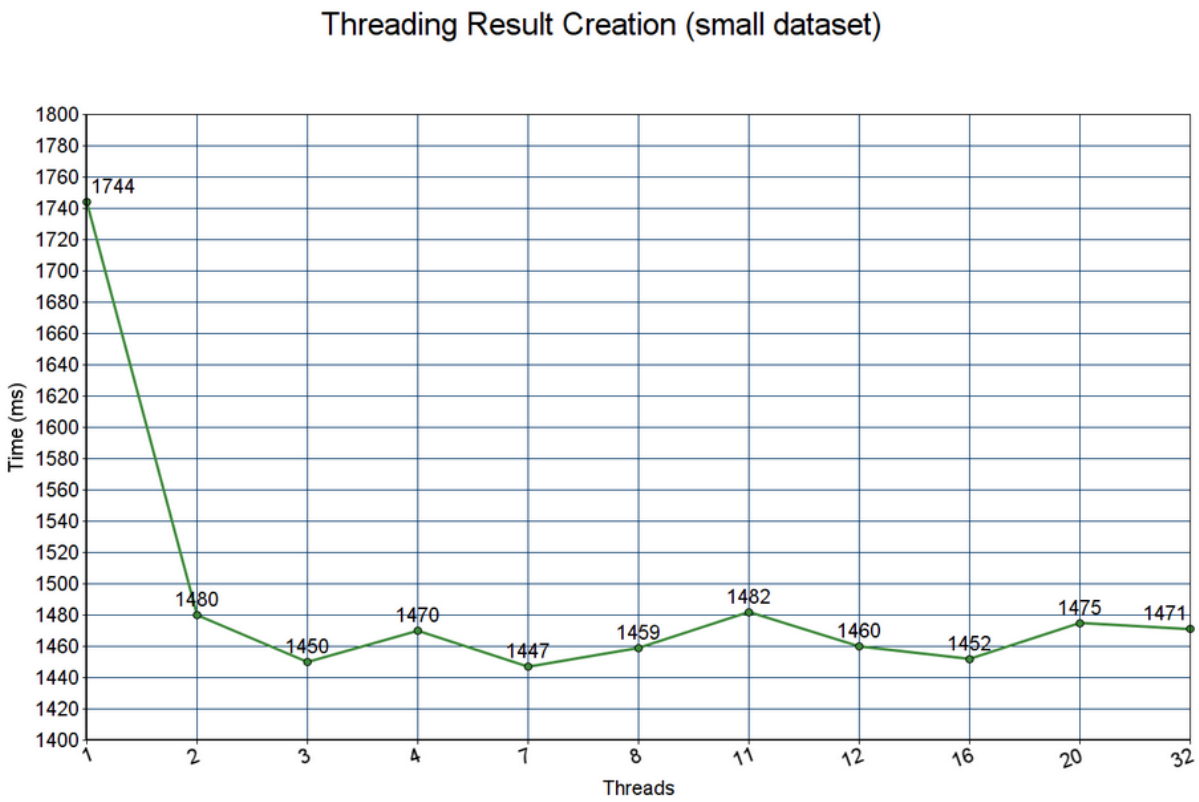
Πιθανή βελτιώση: Αυτή την στιγμή η ουρά υλοποιείται ως λίστα με κάθε node να είναι και ένα job που περιμένει κάποιο thread (από το thread pool) να το εκτελέσει. Μία πιθανή βελτίωση είναι η μετατροπή αυτής της λίστας σε έναν κυκλικό buffer.

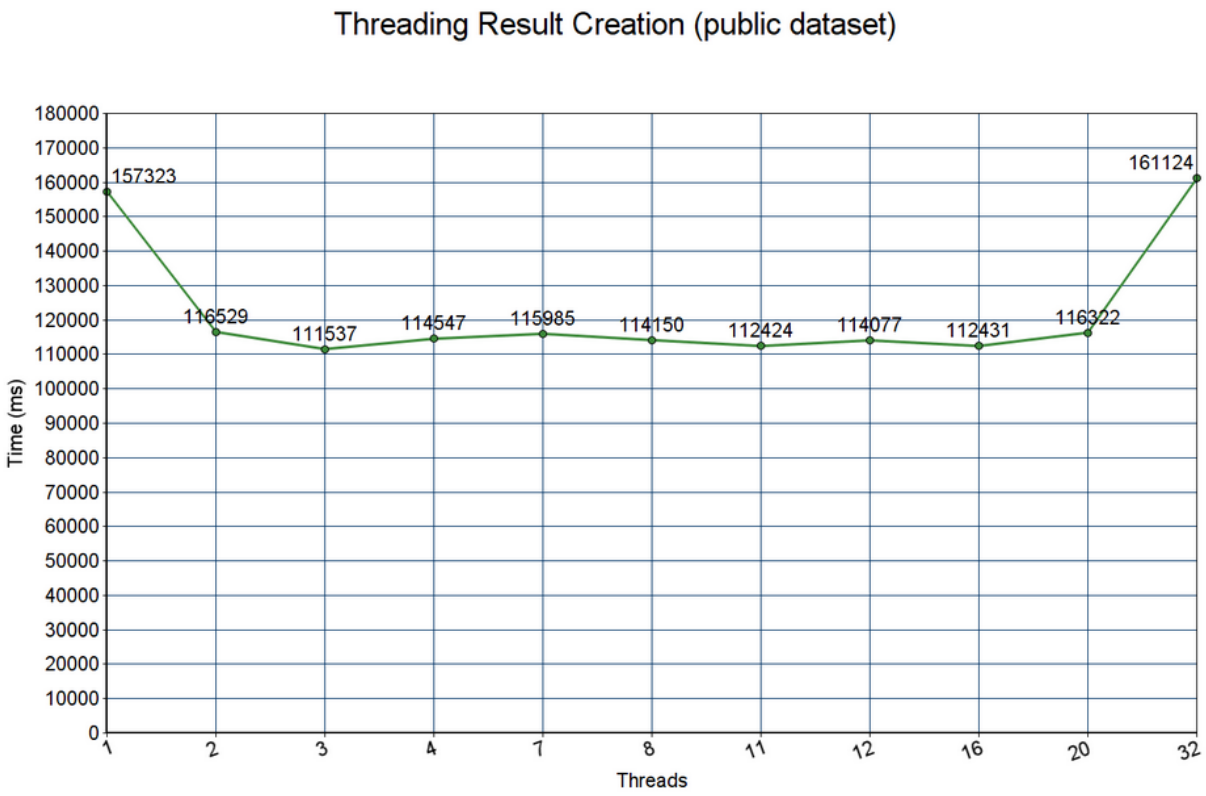
Threading result list creation

Έστω S το relation χωρίς ευρετηρίο. Δημιουργούμε τόσα jobs όσα και τα buckets του reordered S. Το κάθε job είναι υπεύθυνο να κάνει το join μεταξύ του bucket που του ανατέθηκε και του ευρετηρίου. Ως αποτέλεσμα δημιουργούνται πλήθος(buckets) result lists τα οποία και ενώνουμε στο τέλος για τον σχηματισμό της τελικής λίστας.

Το κάθε node της λίστας περιέχει 128Kb αποτελεσμάτων.

Στην συνέχεια με μεταβλητό αριθμό threads παρατηρήθηκε μια μείωση περίπου 300ms στο small dataset και 45sec στο public dataset. Η μείωση αυτή φαίνεται στα παρακάτω γραφήματα.





Στα παραπάνω 2 γραφήματα παρατηρούμε τον χρόνο εκτέλεσης να σταθεροποιείται μεταξύ των 3 – 16 threads. Για μεγαλύτερο αριθμό το overhead που προσθέτει ο συγχρονισμός και το thread switching καταλλήγουν να αυξάνουν τον χρόνο εκτέλεσης.

Τελικά:

Καλύτερος Χρόνος:

- 1447ms στο small dataset (7 threads)
- 111537ms στο public dataset (3 threads)

Επιτάχυνση (σε σχέση με το αρχικό version):

- 27.5% στο small dataset
- 28% στο public dataset

Συνολικά allocations (σε σχέση με το αρχικό version):

- 1513MB (7% αύξηση)

Threading στο reordering

-Στο ιστόγραμμα

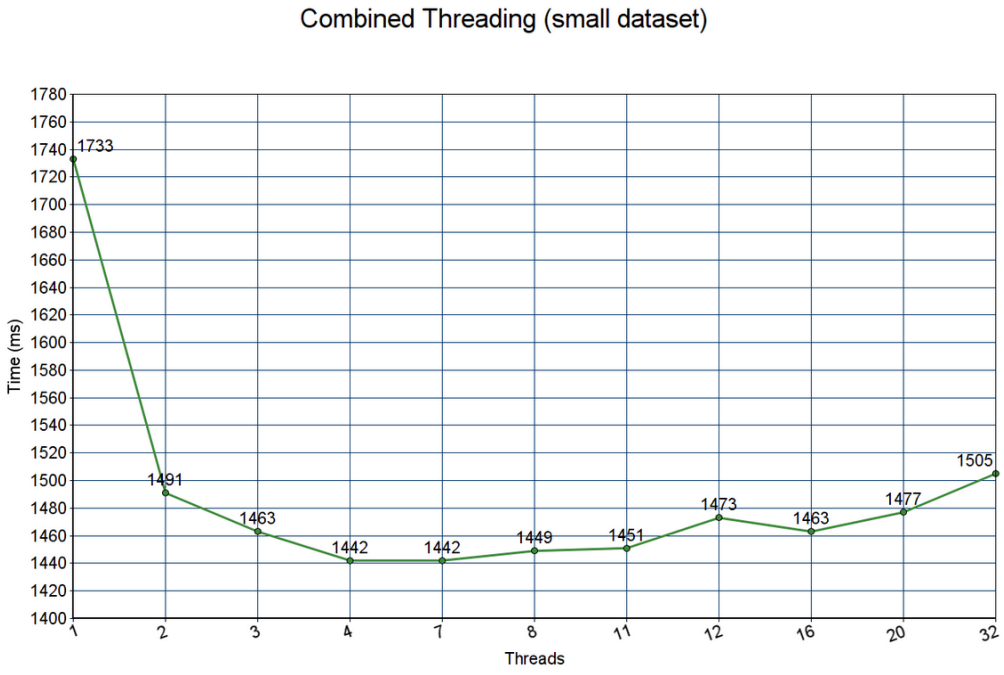
Για M νήματα, χωρίζουμε τον πίνακα σε M μέρη και δημιουργούμε τα αντίστοιχα M jobs ώστε να υπολογίσουμε για κάθε μέρος του πίνακα το αντίστοιχο ιστόγραμμα και τις τιμές της συνάρτησης κατακερματισμού για κάθε πλειάδα. Ενώνουμε τα επιμέρους ιστογράμματα και πίνακες με τιμές κατακερματισμού hash_values.

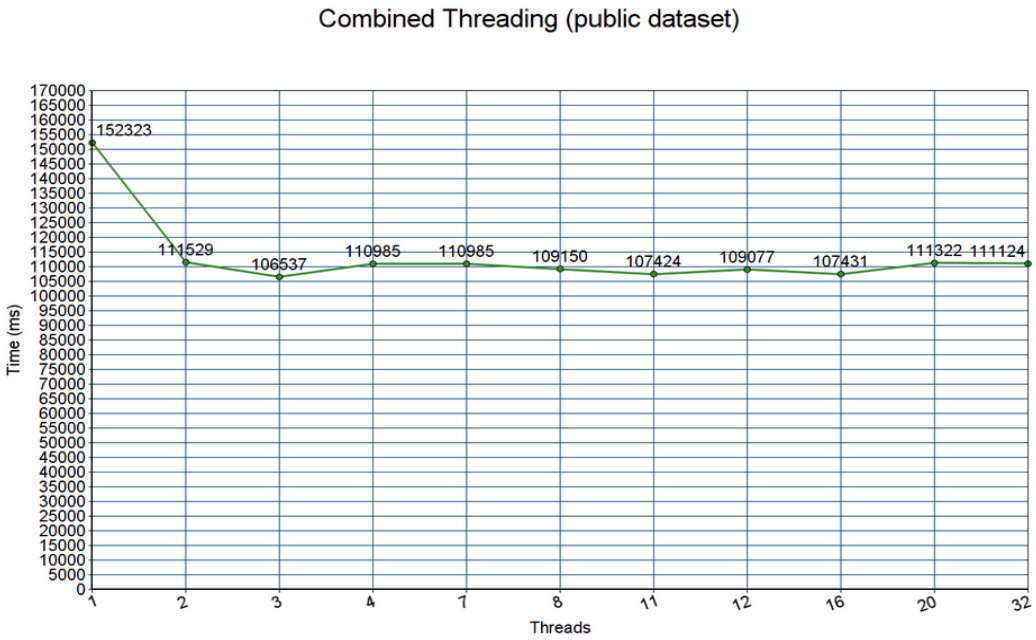
-Αντιγραφή των R και S στους reorederdR και reorderedS

Γνωρίζοντας αυτά χωρίζουμε πάλι τον πίνακα σε M τμήματα και ορίζουμε M jobs, με κάθε job να μεταγράφει τα στοιχεία του αντίστοιχου μέρους του αρχικού πίνακα σε έναν καινούριο πίνακα, τον reorderedR, ο οποίος είναι ταξινομημένος ως προς hash_value.

Χρονομέτρηση συνδυασμένου threading

Παρακάτω ακολουθούν οι μετρήσεις με μεταβλητό αριθμό από threads συνδυάζοντας και τις 3 παραλληλοποιήσεις (ιστογράμματα, αντιγραφή, δημιουργία λίστας αποτελεσμάτων).





Στο small dataset δεν παρατηρούμε μείωση στον χρόνο εκτέλεσης του προγράμματος σε σχέση με το να έχουμε threading μόνο στην λίστα αποτελεσμάτων. Αυτό οφείλεται στο αυξημένο overhead του job scheduler καθώς και στην αδυναμία παραλληλοποίησης του small dataset.

Στο public dataset παρατηρούμε μια μικρή σχετικά επιτάχυνση περίπου των 8sec σε σχέση με το να έχουμε threading μόνο στην λίστα αποτελεσμάτων.

Τελικά:

Καλύτερος Χρόνος:

- 1447ms στο small dataset (7 threads)
- 106537 στο public dataset (3 threads)

Επιτάχυνση (σε σχέση με το αρχικό version):

- 27.5% στο small dataset
- 32% στο public dataset

Συνολικά allocations (σε σχέση με το αρχικό version):

- 1625MB (15% αύξηση)

Query Optimization

Εξήγηση αλγορίθμου Join Enumeration

Ο αλγόριθμος χωρίζεται σε τρία μέρη, την ενημέρωση των αρχικών δομών με τα στατιστικά μετά την εκτέλεση των φίλτρων και γενικά την αρχικοποίηση των δομών που θα χρειαστούν στον αλγόριθμο, την ενημέρωση των στατιστικών μετά την εκτέλεση των ζεύξεων και τον καθορισμό κατάλληλης σειράς ζεύξεων(κορμός αλγορίθμου) και τον καθορισμό των προτεραιοτήτων των ζεύξεων, των φίλτρων-ζεύξεων καθώς και "διπλότυπων ζεύξεων".

Το πρώτο μέρος υλοποιείται από τη συνάρτηση executeFilterPredicates (getStats.c), στην οποία, ανάλογα με την περίπτωση φίλτρου που χρησιμοποιείται, ενημερώνονται αντίστοιχα τα στατιστικά της στήλης/ών που φιλτράρεται/όνται καθώς και όλων των υπόλοιπων στηλών του πίνακα/ων. (σε όλες τις πράξεις των στατιστικών που υπάρχουν διαιρέσεις, σε όλα τα μέρη του αλγορίθμου, χρησιμοποιείται η μαθηματική συνάρτηση ceil για την αποφυγή της αποκοπής και της τιμής 0 σε δεκαδικά αποτελέσματα >0 και <1). Έπειτα, γίνεται η αρχικοποίηση του γράφου ο οποίος χρησιμοποιείται για να αποθηκεύει τις συνδέσεις ανάμεσα στα relations και η αρχικοποίηση του hash tree με τα μονοσύνολα, τις σχέσεις που συμμετέχουν στο query καθώς και τις απαραίτητες πληροφορίες για αυτές.

Για το hash tree χρησιμοποιείται ένας πίνακας 2^n θέσεων όπου n ο αριθμός των relations που συμμετέχουν στο query. Κάθε κελί του πίνακα κωδικοποιείται σε μία ακολουθία από n bits, στην οποία ο αριθμός των set bits δείχνει πόσα και ποια relations συμμετέχουν στον εκάστοτε συνδυασμό. Π.χ. σε ένα query 4 relations στο κελί με index 11 (1011), συμμετέχουν στον συνδυασμό 3 σχέσεις, η $4^η$, η $2^η$ και η $3^η$ και η σειρά είναι καθορισμένη στο string comb.

Ο κάθε κόμβος του hash tree (HTNode)κρατάει αρχικά ένα char * comb με την καλύτερη ακολουθία/συνδυασμό των σχέσεων μέχρι αυτόν τον κόμβο το οποίο αποτελεί και κλειδί μέσω του οποίου προκύπτει το index του πίνακα (hashTree) μέσω της "hash function" combToIndex(hashTreeManip.c), πόσα relations συμμετέχουν στον συνδυασμό καθώς και έναν πίνακα με τα στατιστικά τους καθώς και το κόστος του συνδυασμού. Ενημερώνεται επίσης κατά τη διάρκεια του αλγορίθμου το char * joinSeq, η ακολουθία δηλαδή των joins που πρέπει να γίνουν χάριν διευκόλυνσης για τον καθορισμό των προτεραιοτήτων μετέπειτα (στους κόμβους με τα μονοσύνολα είναι κενό).

Στο δεύτερο μέρος υλοποιείται ο κορμός του αλγορίθμου με τη συνάρτηση joinEnumeration (joinEnumeration.c) όπου εξετάζονται όλοι οι πιθανοί συνδυασμοί S μεγέθους 1, 2... #relations_in_query – 1 με κάθε ένα από τα relations(μονοσύνολα) R_j που δεν ανήκουν στον εκάστοτε συνδυασμό S_i και είναι connected και καλείται η συνάρτηση CreateJoinTree έτσι ώστε να δημιουργήσει τον κόμβο HTNode για τον καινούριο συνδυασμό που δημιουργείται. Εκτελώντας τις πράξεις των στατιστικών, ενημερώνεται το κόστος της δημιουργίας του συνδυασμού , το οποίο υπολογίζεται από το κόστος του συνδυασμού S_i συν τον αριθμό των tuples που προκύπτουν μετά από τη σύζευξή του με το relation R_j. Εάν ο κόμβος που δημιουργήθηκε δεν έχει μικρότερο κόστος από τον κόμβο στο αντίστοιχο index του πίνακα hashTree ελευθερώνεται διαφορετικά παίρνει τη θέση του προηγούμενου συνδυασμού(αν υπάρχει).

Τέλος, στο τρίτο μέρος καθορίζονται οι προτεραιότητες των ζεύξεων έτσι όπως προέκυψαν από το δεύτερο μέρος μέσω του string joinSeq. Σχέσεις των οποίων οι δύο participant relations έχουν ήδη κάνει join σε διαφορετικές στήλες λειτουργούν σαν filter join και παίρνουν προτεραιότητα αμέσως μετά από το αντίστοιχο join.”Διπλότυπα” join παραλείπονται και δεν παίρνουν προτεραιότητα για εξοικονόμηση χρόνου.

Μετρήσεις με Query Optimization

Οι παρακάτω μετρήσεις έγιναν εφαρμόζοντας το query optimization στην αρχική version (δηλαδή χωρίς το threading).

Τελικά:

Καλύτερος Χρόνος:

- 1555ms στο small dataset
- 110840ms στο public dataset

Επιτάχυνση (σε σχέση με το αρχικό version):

- 22% στο small dataset
- 30% στο public dataset

Συνολικά allocations (σε σχέση με το αρχικό version):

- 1435MB (3% αύξηση)

Ομαδοποίηση των γραμμών σε πακέτα

Εξήγηση ομαδοποίησης σε πακέτα

Θυμηθείτε ότι οι intermediates κρατάνε τα rowIds της αντίστοιχης σχέσης. Επίσης, ότι όταν η σχέση που θα μπει στον radixHashJoin προέρχεται από intermediate το rowId που περνάμε είναι το rowId του intermediate και όχι της σχέσης στην οποία αυτός δείχνει.

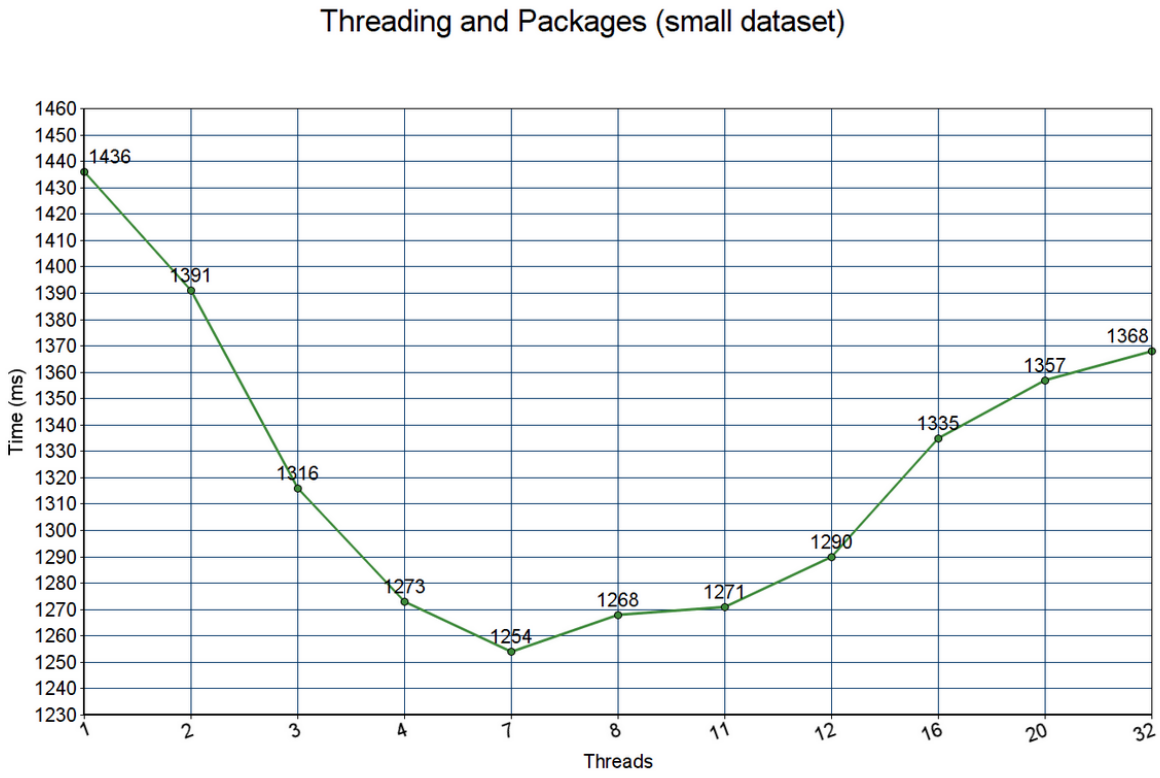
Μετά από διαδοχικές ζεύξεις, είναι πιθανό να υπάρχει intermediate με πολλές ίδιες τιμές σε κάποια σχέση. Σκεφτήκαμε λοιπόν να ομαδοποιούμε αυτές τις τιμές (όσες είναι συνεχόμενες) σε μία, για να γλυτώνουμε χρόνο στη radixHashJoin.

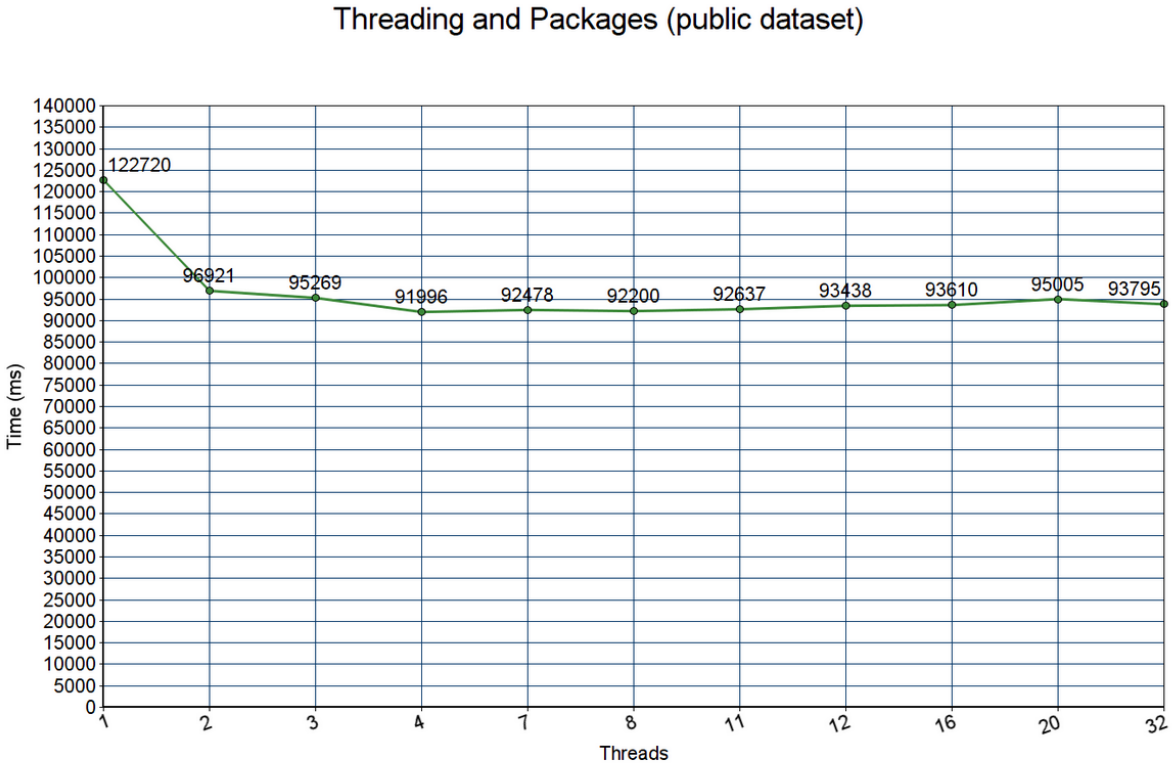
Πιο λεπτομερώς, όπως διαβάζουμε τον πίνακα για να σχηματίσουμε το relation που θα περαστεί στον radix, αν κάποια τιμή είναι ίδια με την προηγούμενή της, την παραλείπουμε. Να σημειωθεί ότι αυτό συμβαίνει μόνο στους intermediate, που περιέχουν rowIds στα οποία γίνεται ο έλεγχος παράλειψης, και όχι στις σχέσεις που είναι στην αρχική τους μορφή.

Αντίστοιχα μετά την radix, όταν διαβάζουμε το αποτέλεσμα για να σχηματίσουμε τον καινούριο intermediate, για κάθε rowId του προηγούμενου intermediate κοιτάμε και όλες τις επόμενες τιμές του (δηλαδή τα rowIds της αρχικής σχέσης) μέχρι να βρούμε κάποια διαφορετική.

Έτσι, πετυχαίνουμε σημαντική μείωση του χρόνου της radix με αντάλλαγμα ένα overhead στην αποσυμπίεση των αποτελεσμάτων.

Μετρήσεις ομαδοποίησης σε πακέτα





Τελικά:

Καλύτερος Χρόνος:

- 1254ms στο small dataset (7 threads)
- 91966ms στο public dataset (3 threads)

Επιτάχυνση (σε σχέση με το αρχικό version):

- 29% στο small dataset
- 41% στο public dataset

Συνολικά allocations (σε σχέση με το αρχικό version):

- 3270MB (130% αύξηση)

Χρήση αποθήκης για τα indexes

Εξήγηση αποθήκης indexes

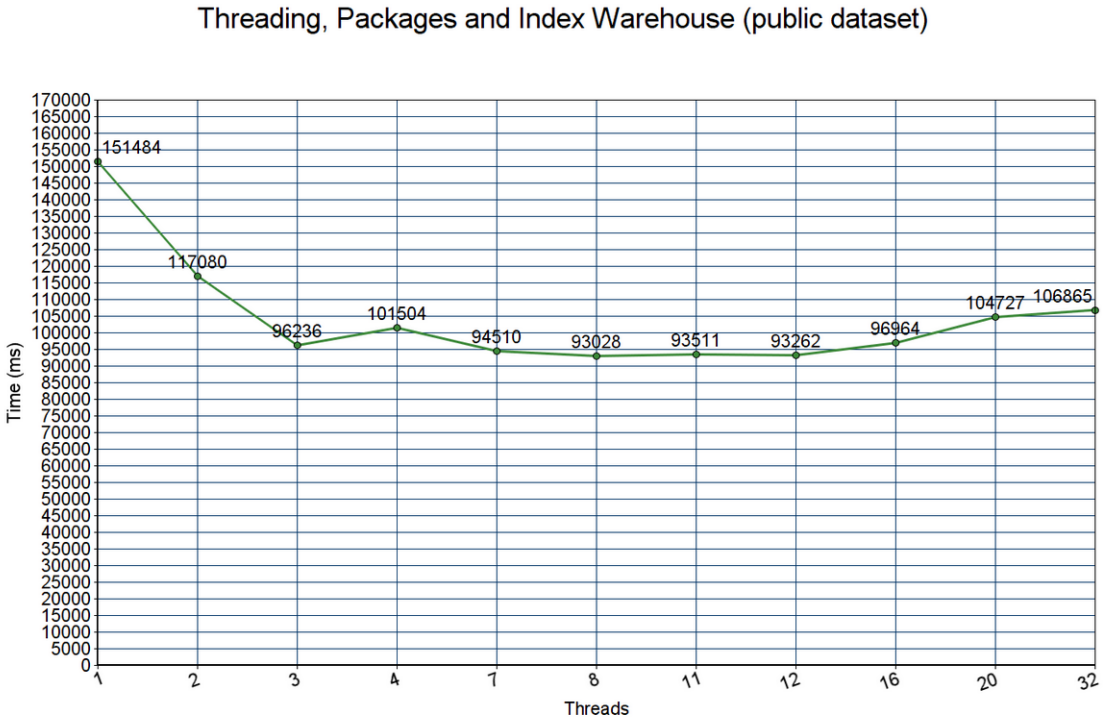
Παρατηρούμε μεγάλο αριθμό από ευρετήρια που δημιουργούνται να χρησιμοποιούνται πολλές φορές σε διαφορετικά queries. Η radix hash join σε κάθε κλήση θα δημιουργούσε αυτό το ευρετήριο ακόμα και αν στο παρελθόν το είχε ξαναδημιουργήσει.

Για αποφυγή επανάληψης ίδιων διαδικασιών, τα ευρετήρια και οι reordered πίνακες που δημιουργούνται για κάθε σχέση, στήλη και αριθμό κουβάδων κατακερματισμού διατηρούνται σε μια δομή (αποθήκη) και επαναχρησιμοποιούνται όποτε χρειάζονται. Αυτό συμβαίνει ΜΟΝΟ για σχέσεις που δεν έχουν υποστεί προηγούμενη επεξεργασία. Δηλαδή, μια σχέση στην οποία έχει προηγουμένως εφαρμοστεί ένα φίλτρο ή μια σύζευξη δεν θεωρείται ότι είναι στην αρχική της μορφή και επομένως δεν ανασύρονται ούτε αποθηκεύονται δομές για αυτήν στην αποθήκη.

Για αυτόν τον λόγο, σε κάθε ζεύξη που περιέχει τουλάχιστον 1 σχέση στην αρχική της μορφή δίνεται προτεραιότητα για τη δημιουργία ευρετηρίου στη σχέση που έχει ήδη έτοιμο ευρετήριο ή ταξινομημένο πίνακα. Μόνο αν δεν υπάρχει κάποια αντίστοιχη δομή στην αποθήκη συγκρίνονται οι σχέσεις και αποδίδεται προτεραιότητα στην μικρότερη σε αριθμό γραμμών.

Μετρήσεις αποθήκης indexes

Οι παρακάτω μετρήσεις έγιναν με το πρόγραμμα να περιέχει τα παραπάνω threading και ομαδοποίηση σε πακέτα που αναφέρθηκαν.



Τελικά:

Καλύτερος Χρόνος:

- 1289 στο small dataset (4 threads)
- 93028 στο public dataset (8 threads)

Επιτάχυνση (σε σχέση με το αρχικό version):

- 27% στο small dataset
- 40% στο public dataset

Συνολικά allocations (σε σχέση με το αρχικό version):

- 3280MB (130% αύξηση)

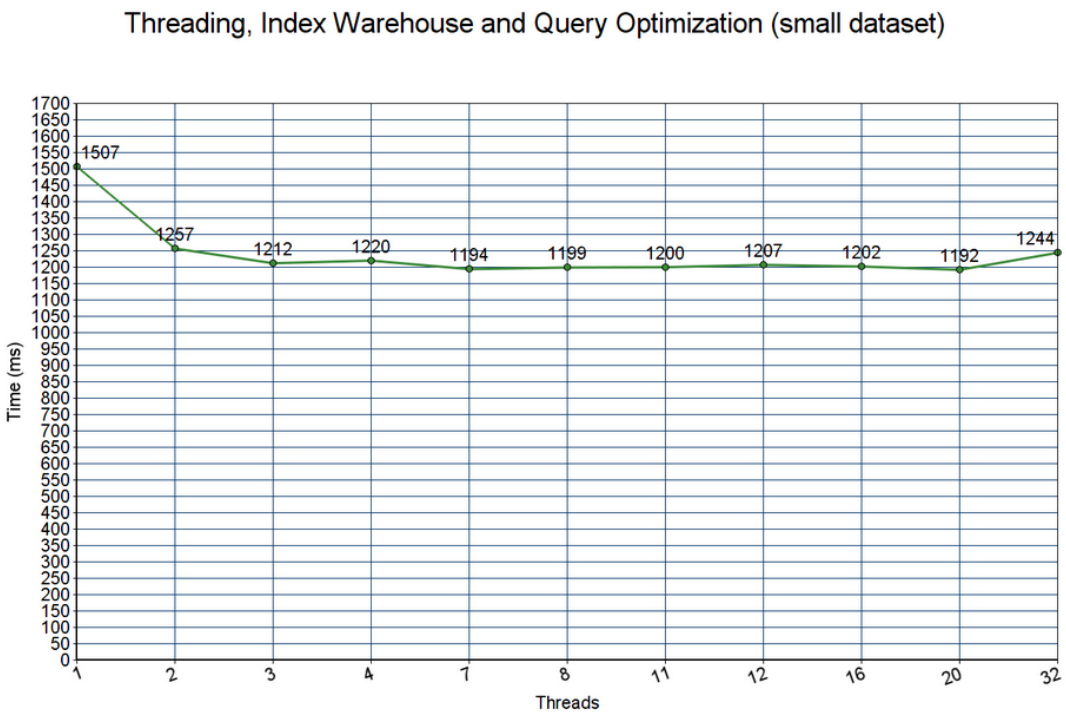
Οι μετρήσεις δεν δείχνουν κάποια βελτίωση στον χρόνο. Όμως στην συνέχεια θα φανεί η χρησιμότητα τους καθώς δεν θα μπορούν να συνδυαστούν τα πακέτα με το query optimization που αναφέρθηκαν πιο πάνω.

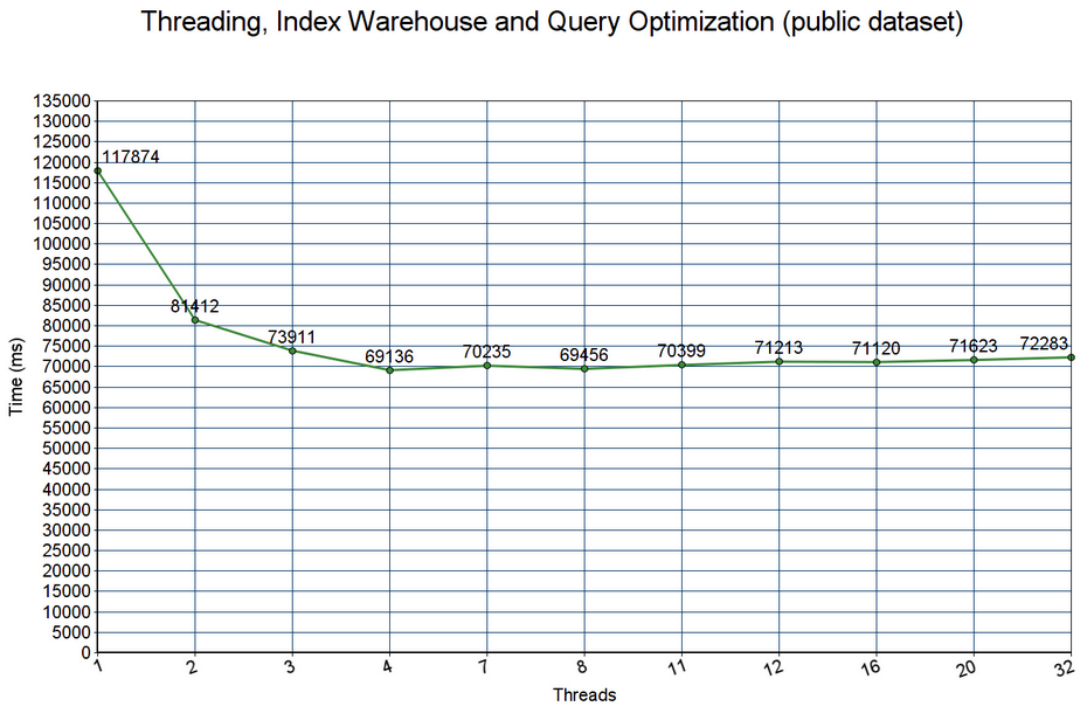
Συνδυασμός των παραπάνω - Τελικά συμπεράσματα

Στην τελική έκδοση του κώδικα καταληξαμε να διατηρούμε τα παρακάτω optimizations:

- Threading σε επίπεδο reordering
- Threading στην δημιοργία των αποτελεσμάτων
- Query optimization
- Αποθήκη Indexes

Δεν διατηρήσαμε τα πακέτα καθώς όταν δεν είχαμε το query optimization παρά το μεγάλο overhead παρατηρούσαμε μεγάλη βελτίωση στον χρόνο. Όμως, με το query optimization καταλλήγουμε να αποφεύγουμε μεγάλο αριθμό από συνεχόμενες ίδιες τιμές με αποτέλεσμα το overhead για τα πακέτα να κοστίζει περισσότερο σε χρόνο από τον χρόνο που μας κερδίζει. Βάλαμε έναν έλεγχο ώστε η συμπίεση/αποσυμπίεση να εκτελούνται μόνο το εκτιμώμενο κέρδος είναι μεγαλύτερο από το overhead αλλά μετά ανακαλύψαμε πως αυτό δεν συμβαίνει σχεδόν ποτέ στα datasets που έχουμε, οπότε το κάναμε comment-out για να μην έχουμε το overhead του ελέγχου.





Τελικά:

Καλύτερος Χρόνος:

- 1194 στο small dataset (7 threads)
- 69136 στο public dataset (4 threads)

Επιτάχυνση (σε σχέση με το αρχικό version):

- 32% στο small dataset
- 55% στο public dataset

Συνολικά allocations (σε σχέση με το αρχικό version):

- 1407MB (2% αύξηση)