Υλοποίηση Λογισμικού για Πληροφοριακά Συστήματα 2018-2019

FINAL REPORT

Μιχαήλ Μπιζίμης 1115201500102
Κωνσταντίνος Τσιάρας 1115201500165
Μάρκος Βολίκας 1115201500018

Γενική Περιγραφή Υλοποίησης

- **Relations και Radix Hash Join**
  (Relation.cpp/.h, JoinRelation.cpp/.h, JoinResults.cpp/.h, RadixHashJoin.cpp/.h)

  Για τα relations χρησιμοποιούμε μια abstract pure virtual κλάση QueryRelation η οποία εξειδικεύεται σε απλή Relation και σε IntermediateRelation (υποκλάσεις). Οι πίνακες όπως φορτώνονται από το δίσκο αποθηκεύονται σε Relation και οι πίνακες για τα ενδιάμεσα αποτελέσματα σε IntermediateRelation, όπου κρατάμε και δείκτες στο αντίστοιχο αρχικό Relation για κάθε σχέση μαζί με τα row ids που έχουν φτάσει έως εκείνο το σημείο ενδιάμεσων αποτελεσμάτων). Η ιδέα πίσω από τη σχεδιαστική αυτή επιλογή είναι η πιο ευέλικτη και καθαρότερη υλοποίηση του query execution στον joiner, καλώντας μεθόδους της QueryRelation, χωρις να χρειάζεται διαφοροποίηση αν πρόκειται για αντικείμενο Relation ή IntermediateRelation (δείτε παρακάτω).

  Για το radix hash join χρησιμοποιούμε μια άλλη κλάση JoinRelation, η οποία περιέχει την στήλη που γίνεται join μαζί με τα row ids της, καθώς και τη λειτουργικότητα για το Partition Phase του RadixHashJoin (δημιουργεί το histogram, τα prefix sums και αναδιοργανώνει τα στοιχεία της στήλης σε buckets).

  Ο γενικός αλγόριθμος για το Radix Hash Join είναι ο εξής:
  1. Κάλεσε την PartitionRelation για κάθε μία από τις δύο σχέσεις
  2. Για κάθε bucket δημιούργησε το ευρετήριο για την μία σχέση και μετά χρησιμοποίησέ το καθώς κάνεις probe τα στοιχεία του αντίστοιχου bucket της άλλης σχέσης για να βρεις ίσες τιμές.

  Παραπάνω έχουμε υλοποιήσει ξεχωριστά τη δημιουργία του ευρετηρίου (δηλαδή την κατασκευή των πινάκων Chain και Bucket - 2η φάση) και την αναζήτηση για ζεύγη γραμμών που ταιριάζουν (εκεί παράγεται και το τελικό result - 3η φάση). Επίσης αποφασίσαμε να κάνουμε index πάντα για τη σχέση με την μικρότερη πληθικότητα (cardinality) για να ελαχιστοποιήσουμε το κόστος δημιουργίας ευρετηρίων.

  Το result έχει υλοποιηθεί ως μια linked list από buffers (abstracted με την κλάση Result) και γίνεται accessed χρησιμοποιώντας έναν Iterator (όπως και οι standard βιβλιοθήκες) που δίνει κάθε φορά το επόμενο tuple με row ids. Επειδή χρησιμοποιούμε πολλές τέτοιες λίστες που γεμίζουν παράλληλα και μετά τις συνενώνουμε (δείτε παραλληλοποίηση) ορίσαμε το μέγεθος του κάθε buffer να είναι 128KB (#defined στο *ConfigureParameters.h*).

Σχετικά με τις συναρτήσεις κατακερματισμού, για λόγους απόδοσης χρησιμοποιούμε bitwise operators για να παίρνουμε τα τελευταία bits ενός αριθμού. Πιο συγκεκριμένα, στη συνάρτηση για το partition της σχέσης, H1, χρησιμοποιούμε τα λιγότερο σημαντικά β = ⌈ max(|R|, |S|) / cache-size ⌉ bits της εκάστοτε τιμής, για να δημιουργήσουμε buckets μεγέθους cache-size on average. Θεωρούμε cache-size 512 (32KB), που είναι συνηθισμένο για L1-caches. Αντίστοιχα, στη συνάρτηση για το Bucket του ευρετηρίου, H2, κρατάμε τα β / 2 bits που ακολουθούν μετά τα πρώτα λιγότερο σημαντικά β bits.

- **Parsing και Query Execution (Joiner)**
  (SQLParser.cpp/.h, joiner-main.cpp)

  Στην αρχή στον joiner χρησιμοποιούμε την mmap() για την φόρτωση των σχέσεων (constructor της Relation) σε έναν πίνακα από Relation και στη συνέχεια υπολογίζουμε τα αρχικά στατιστικά που χρειάζονται για το query optimization για κάθε μία από αυτές (δείτε παρακάτω).

  Ο joiner εκτελεί ένα query την φορά. Υλοποιήσαμε έναν parser (*SQLParser.cpp/.h*) για τα queries που τα χωρίζει σε filters, projections και predicates. Αφού κάνουμε parse ένα query:

  1. Τρέχουμε τον Optimizer, ο οποίος μας επιστρέφει μια βελτιστοποιημένη σειρά για τα join predicates.

  2. Ταυτόχρονα (δείτε παραλληλοποίηση), εκτελούμε το FROM δημιουργώντας ένα QueryRelation για κάθε σχέση στο FROM, το οποίο αρχικά θα είναι δείκτης στο κατάλληλο Relation (ή IntermediateRelation με όλα τα row ids αν μια σχέση εμφανίζεται πάνω από μία φορά).

  3. Ύστερα, για να μειώσουμε όσο το δυνατόν περισσότερο το μέγεθος των ενδιάμεσων αποτελεσμάτων, εκτελούμε πρωτα τα φίλτρα και τα equal columns predicates που υπάρχουν στο WHERE του query (σε arbitrary σειρά).

  4. Έπειτα, αφού έχει τελειώσει το Query Optimization, εκτελέσουμε τα join predicates στην σειρά που εκτιμήθηκε καλύτερη, μαζεύοντας τα αποτελέσματα κάθε φορά στην QueryRelation με το μικρότερο index στον πίνακα από QueryRelations. Μερικά από αυτά τα join predicates μπορεί να καταλήξουν να είναι equal columns predicates, αν ένα προηγουμένως join predicate αφορά δύο προηγουμένως ξεχωριστά QueryRelations

που πλέον έχουν συγχωνευτεί στο ίδιο QueryRelation (ενδιάμεσο αποτέλεσμα) με κάποιο άλλο join.

5. Αφού τελειώσουν τα joins, ελέγχουμε αν έχουν συγχωνευτεί όλες οι σχέσεις του FROM (στο QueryRelations[0]) ή όχι. Αν όχι, τότε εκτελούμε τα κατάλληλα cross-products μέχρι να υπάρχουν όλες σε ένα IntermediateRelation, στο QueryRelations[0].

6. Τέλος, υπολογίζεται το sum των στηλών του QueryRelations[0] στα projections και εκτυπώνεται στο std::out (σε ξεχωριστό thread για εξοικονόμηση χρόνου - δείτε παραλληλοποίηση).

Λογική Παραλληλοποίησης

Για τους λόγους την παραλληλοποίησης φτιάξαμε έναν ευέλικτο δικό μας job scheduler (*JobScheduler.cpp/.h*), βασισμένο στην κληρονομικότητα της C++.

Όσον αφορά την παραλληλοποίηση, εστιάσαμε στην παραλληλοποίηση σε επίπεδο query με την λογική ότι αν ένα query τρέχει πιο γρήγορα εκμεταλλευόμενο τους σημερινούς multithreaded επεξεργαστές, τότε και το συνολικό input με πολλά batches από queries θα ολοκληρωθεί επίσης γρηγορότερα.

Έτσι, ο joiner (*joiner-main.cpp*) εξακολουθεί (όπως και στο 2ο παραδοτέο) να εκτελεί ένα query την φορά με τις εξής βελτιστοποιήσεις παραλληλίας σε αυτό:

1. **Παραλληλοποίηση join (*JoinJob*):** Κατά την εκτέλεση του αλγορίθμου Radix Hash Join μεταξύ δύο σχέσεων (ενδιάμεσων ή μη), χωρίζουμε τις δύο αυτές σχέσεις (*JoinRelations*) σε buckets (1η φάση). Από εκεί και έπειτα, οι επόμενες φάσεις του Radix Hash Join (2η : indexing και 3η : probing results) είναι τελείως ανεξάρτητες μεταξύ διαφορετικών buckets. Αυτό σημαίνει ότι μπορούν να εκτελεστούν παράλληλα! Αυτό που κάνουμε είναι να κάνουμε schedule ένα τέτοιο JoinJob για κάθε bucket που προκύπτει από την 1η φάση, το οποίο θα εκτελέσει την 2η και 3η φάση του αλγορίθμου σε αυτό το bucket. Κάθε JoinJob δημιουργεί την δική του λίστα με αποτελέσματα από row ids (*JoinResults*). Όταν όλα τα scheduled JoinJobs τερματίσουν, συνενώνουμε σε *O(αριθμός bucket)* χρόνο τα αποτελέσματα σε μία τέτοια λίστα που θα είναι το αποτέλεσμα του Radix Hash Join.

2. **Παραλληλοποίηση partition (HistJob, PartitionJob):** Πέρα από τις φάσεις 2 και 3 του Radix Hash Join που ήταν η πιο προφανής και καρποφόρα (εφόσον δεν υπήρχε ιδιαίτερο κόστος συνάθροισης αποτελεσμάτων) παραλληλοποίηση, μπορεί επίσης να παραλληλοποιηθεί και η 1η φάση: το partition. Σε αυτό το σημείο εξετάσαμε δύο μεθόδους (μεταξύ των οποίων μπορούμε να επιλέξουμε με κατάλληλο #define στο *ConfigureParameters.h*):

    a) Αφού εκτελούμε δύο ανεξάρτητα μεταξύ τους partition σε δύο σχέσεις, μία απλή και αποτελεσματική ιδέα θα ήταν να εκτελέσουμε αυτά τα partition σε 2 διαφορετικά threads, παράλληλα.
    b) Μια άλλη ιδέα είναι να προσπαθήσουμε να κάνουμε το κάθε partition πιο γρήγορο μοιράζοντας τον πίνακα σε ίσου (περίπου) μεγέθους υποπίνακες:
        i) Κατά την φάση της δημιουργίας του ιστογράμματος (HistJob). Σε κάθε HistJob φτιάχνουμε ένα μερικό ιστόγραμμα και αφού ολοκληρωθούν όλα τα συναθροίζουμε στο συνολικό ιστόγραμμα (O(αριθμός bucket)).

ii) Κατά την φάση της δημιουργίας του νέου partitioned πίνακα που σε συνδυασμό με το Psum θα χρησιμοποιηθεί σαν hash table (PartitionJob). Σε κάθε PartitionJob διατρέχουμε έναν υποπίνακα του αρχικού πίνακα και βάσει της τιμής της H1() hash function και του Psum, προσθέτουμε την κάθε τιμή αυτού του υποπίνακα στην σωστή θέση του partitioned πίνακα. Αυτό βέβαια απαιτεί κατά ελάχιστον ένα lock για κάθε bucket, για να αποφύγουμε race conditions, κάτι που στην χειρότερη περίπτωση κάνει τον χρόνο να τελειώσουν όλα τα PartitionJobs ίδιο με την σειριακή εκδοχή.

Μεταξύ των a) και b) παρατηρήσαμε πρακτικά ότι το a) είναι συνεπώς πιο γρήγορο από το b) (για το small και το public workload), πιθανότατα λόγω των locks στο b)ii).

3. **Παραλληλοποίηση Φίλτρων και Equal Columns (FilterJob, EqColumnsJob):** Μία παρόμοια λογική με το 2. b) ii) εφαρμόσαμε για να παραλληλοποιήσουμε τα φίλτρα και τις ισότητες μεταξύ στηλών ίδιας σχέσης (υπάρχουν σχετικές επιλογές μεταξύ παράλληλης και σειριακής έκδοσης στο *ConfigureParameters.h*). Χωρίζουμε τον πίνακα σε ίσου (περίπου) μεγέθους υποπίνακες και σε κάθε FilterJob και EqColumnsJob υπολογίζουμε ποιες στήλες αυτού του υποπίνακα «περνάνε» από το φίλτρο. Στο τέλος χρειάζεται μόνο να αθροίσουμε το πόσα rows συνολικά «περάσανε» από το φίλτρο. (Ύστερα η δημιουργία του νέου πίνακα από rowids και του IntermediateRelation γίνεται αναγκαστικά ακολουθιακά).

4. **Παραλληλοποίηση sum στηλών και IO (IOJob):** Εκτός από την παραλληλοποίηση διαδικασιών στο WHERE κομμάτι του κάθε query, θεωρήσαμε κερδοφόρο το να απαλλάξουμε το main thread από το SELECT κομμάτι, δηλαδή να πρέπει να υπολογίσει το sum των στηλών των projections του query και να εκτυπώσει τα αποτελέσματα. Από την στιγμή που έχει εκτελέσει όλο το WHERE, το main thread του joiner κάνει schedule ένα IOJob το οποίο θα υπολογίσει το παραπάνω sum και θα εκτυπώσει τα αποτελέσματα στο std::out σε ξεχωριστό thread, έτσι ώστε να προχωρήσει αμέσως στο επόμενο query.

5. **Παραλληλοποίηση Optimizer (OptimizeJob):** Εφόσον ο Optimizer μας επηρεάζει μόνο την σειρά με την οποία θα εκτελεστούν τα joins, μπορούμε να κερδίσουμε χρόνο τρέχοντας τον από την στιγμή που γίνεται parse ένα query, παράλληλα με το FROM και τα φίλτρα και equal columns του WHERE (τα οποία γίνονται πάντα πρώτα). Όταν φτάσει η ώρα να εκτελέσουμε τα joins, ο joiner περιμένει το OptimizeJob να τελειώσει αν δεν έχει ήδη (που λογικά θα έχει) και τρέχει τα joins με την σειρά που βρήκε ο Optimizer καλύτερη.

Αξίζει να σημειωθεί ότι το μεγαλύτερο κέρδος σε χρόνο πρακτικά είδαμε με το 1. και 4.

Λογική Query Optimizer

Σε ότι αφορά τον εντοπισμό της βέλτιστης σειράς πραγματοποίησης των joins, γίνεται χρήση των στατιστικών και του αλγορίθμου που περιγράφονται στην εκφώνηση. Πιο συγκεκριμένα:

- **Στατιστικά**
  Αυτά υπολογίζονται για κάθε στήλη κάθε πίνακα μία φορά στην αρχή του προγράμματος όπως περιγράφονται στην εκφώνηση. Η μόνη διαφοροποίηση που επιλέξαμε για εξοικονόμηση μνήμης είναι να κρατάμε το f μία φορά για κάθε πίνακα και όχι, όπως τα u, l και d για κάθε στήλη κάθε πίνακα, καθώς προφανώς στον ίδιο πίνακα όλες οι στήλες θα έχουν το ίδιο πλήθος γραμμών. Σημειώνεται ότι, συγκεκριμένα για τον υπολογισμό του d, κάνουμε χρήση bitmap έναντι απλού bool πίνακα προκειμένου να εξοικονομήσουμε μνήμη.

  Έπειτα, για κάθε query γίνεται εκτίμηση των στατιστικών που θα προκύψουν μετά την εφαρμογή των φίλτρων και των equal columns (αντιμετωπίζονται σαν φίλτρα) πάλι με τους τύπους που δίνονται στην εκφώνηση. Τέλος, κάνοντας χρήση των στατιστικών αυτών εκτελείται ο αλγόριθμος του BestTree για την εύρεση της βέλτιστης σειράς εκτέλεσης των ζεύξεων του εκάστοτε query.

- **Join Enumeration**
  Έχοντας θεωρήσει ότι τα φίλτρα έχουν ήδη γίνει (και τα στατιστικά τους υπολογιστεί παραπάνω) το τελευταίο βήμα είναι να βρούμε τη σειρά με την οποία συμφέρει να γίνουν οι ζεύξεις με σκοπό την ελαχιστοποίηση του μεγέθους των ενδιάμεσων αποτελεσμάτων (f). Προκειμένου να μη χρειαστεί να εξετάσουμε N! συνδυασμούς, πράγμα απαγορευτικό όταν μας ενδιαφέρει η απόδοση, υλοποιήσαμε τον αλγόριθμο του οποίου ο ψευδοκώδικας δίνεται στην εκφώνηση. Μέσω αυτού βρίσκεται πρώτα η βέλτιστη σειρά για 2 σχέσεις, έπειτα και χρησιμοποιώντας την προηγούμενη βρίσκεται αυτή για 3 σχέσεις και συνεχίζοντας με τον τρόπο αυτό βρίσκουμε τελικά τη βέλτιστη σειρά για όλες τις N σχέσεις που χρησιμοποιούνται στο τρέχον query.

  Γίνεται ειδική πρόβλεψη για την περίπτωση όπου η ζεύξη μεταξύ δύο σχέσεων είναι εφικτή με περισσότερους από έναν τρόπους (σε διαφορετικές στήλες δηλαδή). Για το λόγο αυτό, όταν λέμε όταν λέμε ότι υπολογίζουμε το f που προκύπτει μετά τη ζεύξη δύο σχέσεων σημαίνει ότι υπολογίζουμε το f που προκύπτει μετά τη ζεύξη αυτών σε κάθε στήλη που μπορεί να γίνει στο query που έχει δοθεί και θεωρούμε ότι η ζεύξη θα πραγματοποιηθεί σε αυτή με το ελάχιστο f. Στην περίπτωση αυτή, οι ζεύξεις στις υπόλοιπες στήλες των δύο αυτών σχέσεων αφήνονται να πραγματοποιηθούν ως φίλτρα (equal columns) στο τελικό αποτέλεσμα και μετά το τέλος όλων των joins.

Παρατηρήσαμε επίσης ότι οι τύποι που δίνονται στην εκφώνηση για τη ζεύξη δύο διαφορετικών πινάκων αν και υπολογίζουν τα d που προκύπτουν δεν τα χρησιμοποιούν ποτέ για τον υπολογισμό του f, το οποίο προφανώς είναι κι αυτό που μας ενδιαφέρει ως συνάρτηση κόστους μας! Ψάχνοντας διάφορες πηγές συναντήσαμε την [1] από την οποία καταλήξαμε στο να χρησιμοποιούμε, αντί για το u-l+1 που προτείνεται, το μέγιστο των d των 2 σχέσεων της ζεύξης. Παρ'όλ'αυτά, στο *ConfigureParameters.h* υπάρχει η επιλογή να χρησιμοποιηθεί ο τύπος της εκφώνησης ή και ο ίδιος τύπος χωρίς παρανομαστή (υποθέτοντας έτσι το κόστος των joins ως κόστος cross products).

Τέλος να σημειωθεί ότι, ενώ η υλοποίησή μας υποστηρίζει τη δυνατότητα πραγματοποίησης queries που περιλαμβάνουν Cross Products, σε αυτά δεν πραγματοποιείται optimization καθώς δεν είναι συμβατά με τον παραπάνω αλγόριθμο.

Μετρήσεις και Συμπεράσματα

Για μετρήσεις είχαμε στην διάθεσή μας δύο μηχανήματα:

1. Μηχάνημα A (64bit):
   Intel i5-6200U CPU @ 2.30GHz (4 cores, 2 threads per core), 8 GB RAM, 32K L1 cache
2. Μηχάνημα B (64bit):
   Intel i3-6006U CPU @ 2.00GHz (4 cores, 2 threads per core), 4GB RAM, 32K L1 cache

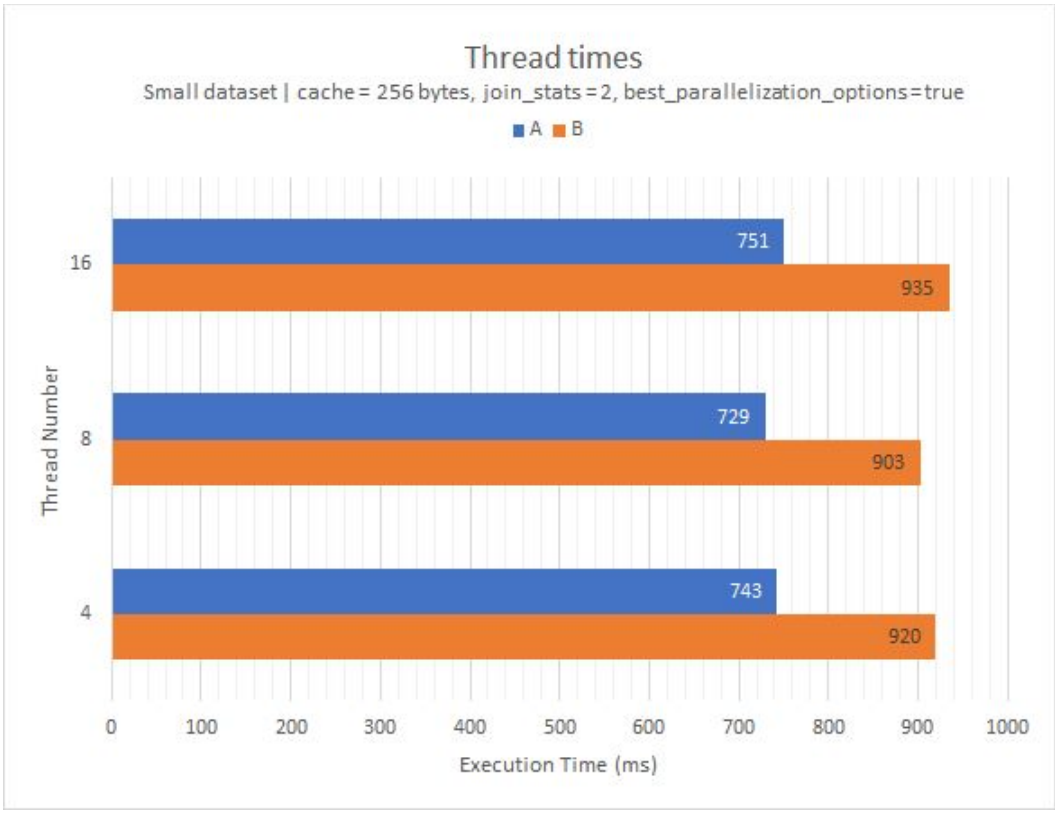
Σαν μέτρο σύγκρισης, το submission που δίνεται στην εκφώνηση του SIGMOD 2018 τρέχει το small workload του SIGMOD 2018 on average σε 951 ms και 1302 ms στα μηχανήματα A και B αντίστοιχα και το public workload σε 184873 ms στο μηχάνημα A.

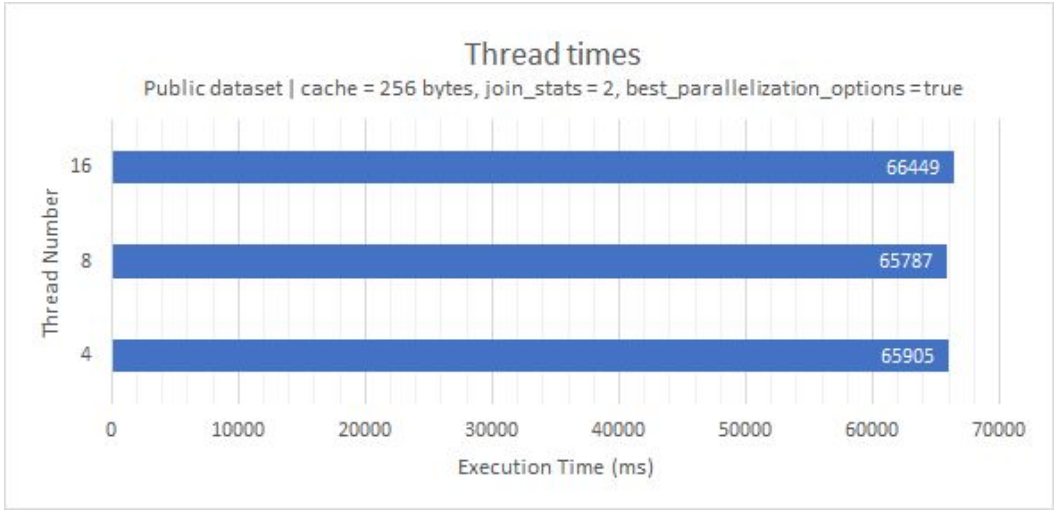
Δεν καταφέραμε να τρέξουμε το public workload στο μηχάνημα B καθώς τα 4GB RAM και μολις 2GB swap χώρου δεν ήταν αρκετά.

Αν κοιτάξετε στο *ConfigureParameters.h*, έχουμε παραμετροποιήσει διάφορα σημεία υλοποίησης του προγράμματος μας. Δοκιμάζοντας διαφορετικές επιλογές για κάποια παράμετρο και κρατώντας τις άλλες σταθερές καταλήξαμε στα παρακάτω αποτελέσματα on average χρόνου εκτέλεσης (τρέξαμε κάθε configuration - του small - 10 φορές και πήραμε τον μέσο όρο).

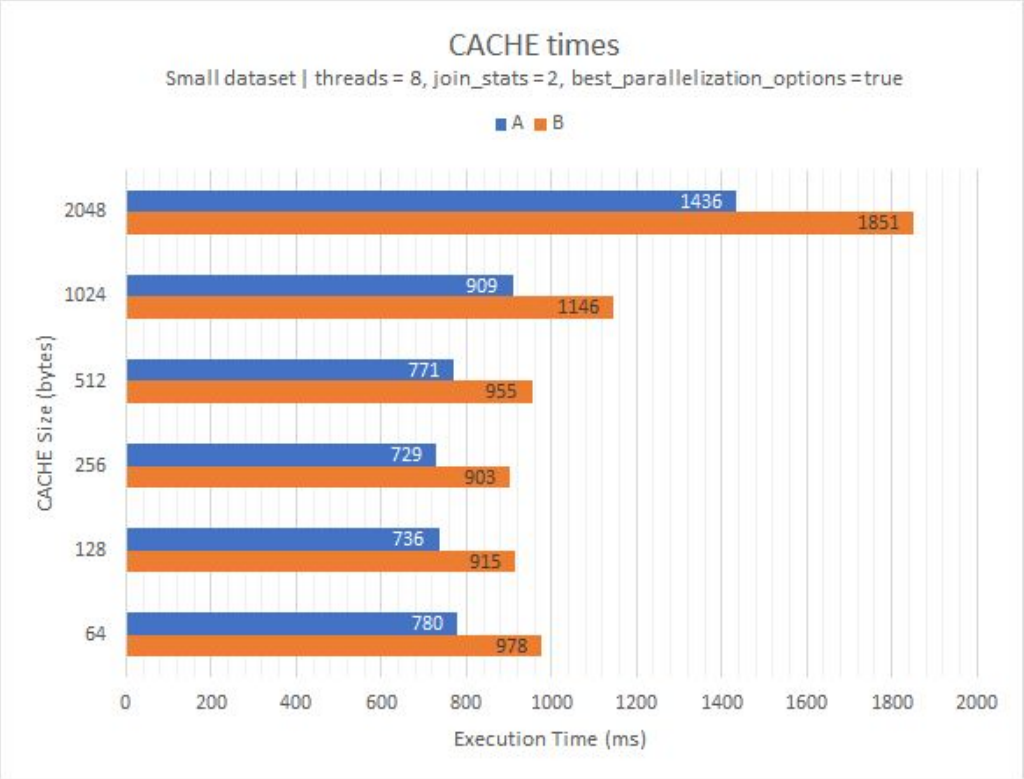
- Παράμετρος NUMBER_OF_THREADS: Η παράμετρος αυτή καθορίζει το μέγεθος του thread pool του JobScheduler. Αυτο επηρεάζει το πόσα threads υπάρχουν για να εκτελέσουν JoinJobs, HistJobs, PartitionJobs, FilterJobs και EqColumnJobs (τα IOJobs και OptimizeJobs, έχουν το δικό τους scheduler από μόλις ένα thread καθώς δεν θέλουμε να μπλέξουμε τις αναμονές για ολοκλήρωση Job μεταξύ τους).

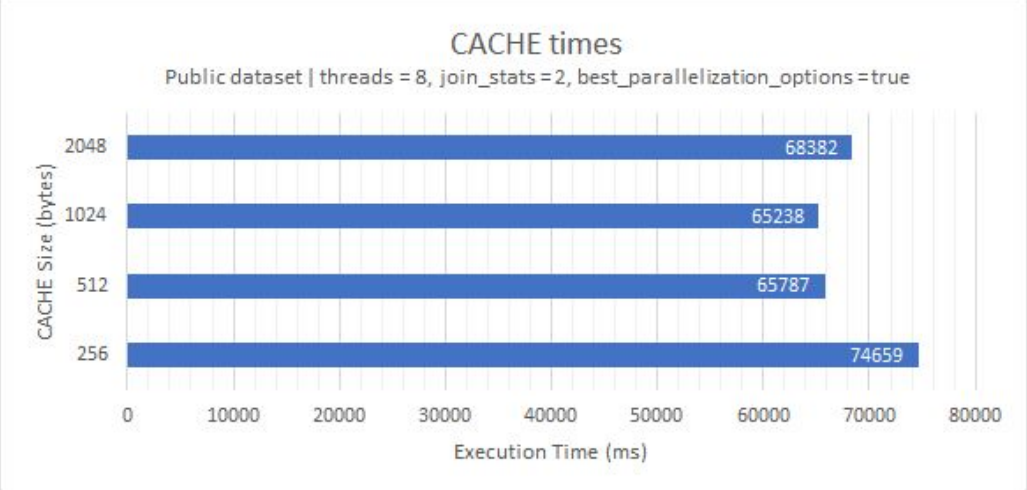
  Εφόσον και τα δύο μηχανήματα μας έχουν 4 πυρήνες με 2 threads per core, είναι λογικό να πάρουμε το βέλτιστο speed up για NUMBER_OF_THREADS = 8. Λιγότερα threads σημαίνει όχι πλήρης εκμετάλλευση του multi-threaded CPU ενώ περισσότερα θα ήταν απλά σπατάλη χρόνου σε context-switching μεταξύ αυτών των threads.


Thread times
Small dataset | cache = 256 bytes, join_stats = 2, best_parallelization_options = true
A
B
Thread Number
16
8
4
751
935
729
903
743
920
0
100
200
300
400
500
600
700
800
900
1000
Execution Time (ms)

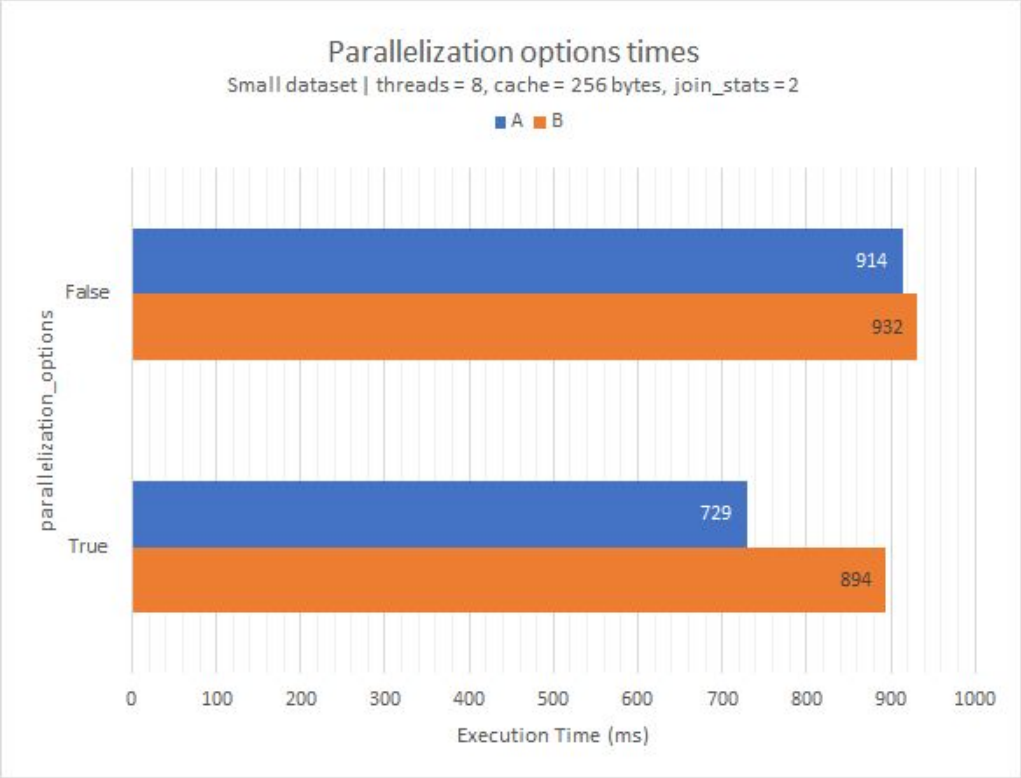

Thread times
Public dataset | cache = 256 bytes, join_stats = 2, best_parallelization_options = true
Thread Number
16
8
4
66449
65787
65905
0
10000
20000
30000
40000
50000
60000
70000
Execution Time (ms)

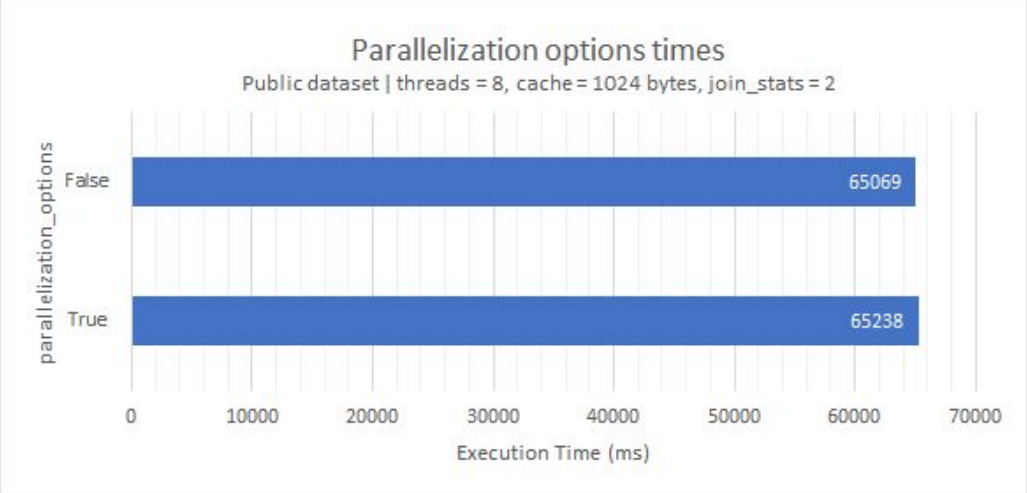
➢ <u>Παράμετρος CACHE:</u> Αυτή η παράμετρος καθορίζει το μέγεθος των λιγότερο σημαντικών bit που θα χρησιμοποιηθούν από την H1() hash function στην 1η φάση του Radix Hash Join. Μεγαλύτερη τιμή σημαίνει μικρότερα σε μέγεθος buckets και άρα περισσότερα σε πληθος buckets. Αυτό σημαίνει περισσότερα JoinJobs (το οποίο είναι καλό), αλλά ταυτόχρονα μεγαλύτερο κόστος της partition (η οποία βέβαια μας έβγαινε πιο αργή σε κάθε configuration) και λιγότερη εκμετάλλευση της cache (αν αυτή ήταν μεγαλύτερη).





➢ <u>Παράμετροι Παραλληλοποίησης:</u> Όσον αφορά τις παραμέτρους παραλληλοποίησης, βρήκαμε πειραματικά ότι το να κάνουμε το κάθε partition παραλληλοποιημένο έναντι να τρέξουμε παράλληλα το ακολουθιακό partition για τις δύο σχέσεις του Radix Hash Join είναι πάντα χειρότερο (πιθανότατα λόγω των locks στο PartitionJob). Από εκεί και πέρα, η παραλληλοποίηση των φίλτρων και equal columns φάνηκε να κάνει μικρή διαφορά (πότε θετικά, πότε αρνητικά). Θεωρώντας ως parallelization_options το να κάνουμε παράλληλα μόνο τα φίλτρα και equal columns (τα υπόλοιπα για τα οποία δεν υπάρχει επιλογή στο *ConfigureParameters.h* γίνονται πάντα πχ JoinJobs, IOJobs, etc) πήραμε τα εξής:

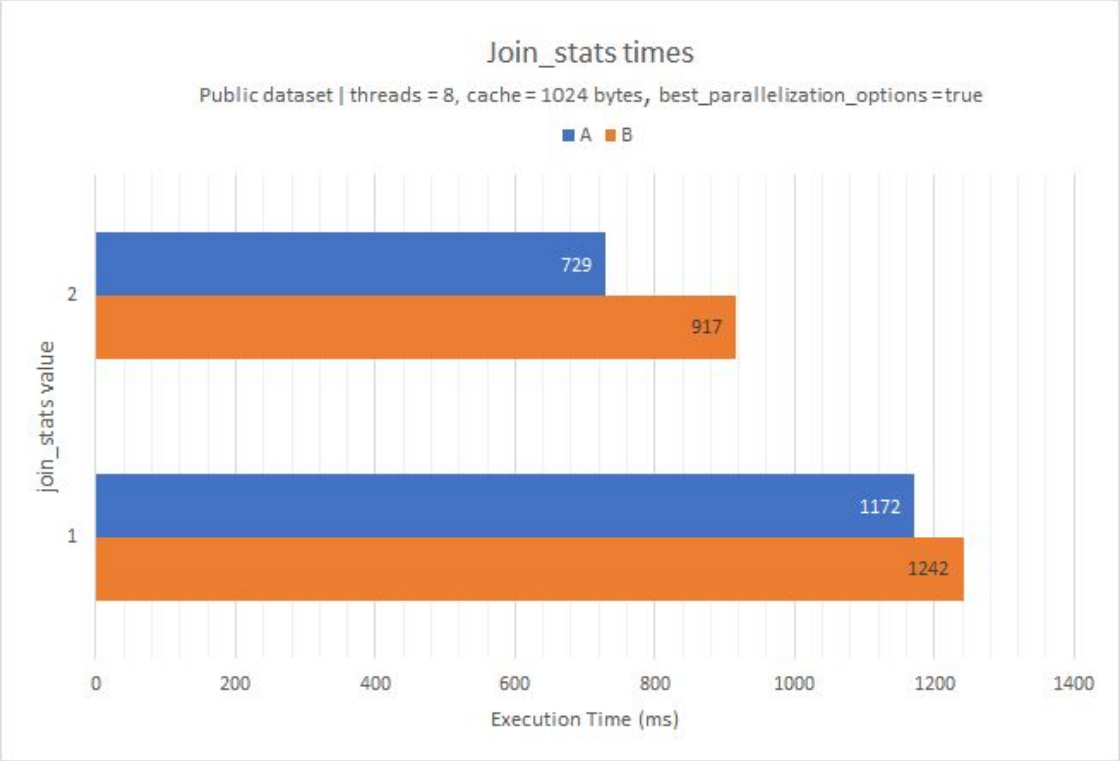


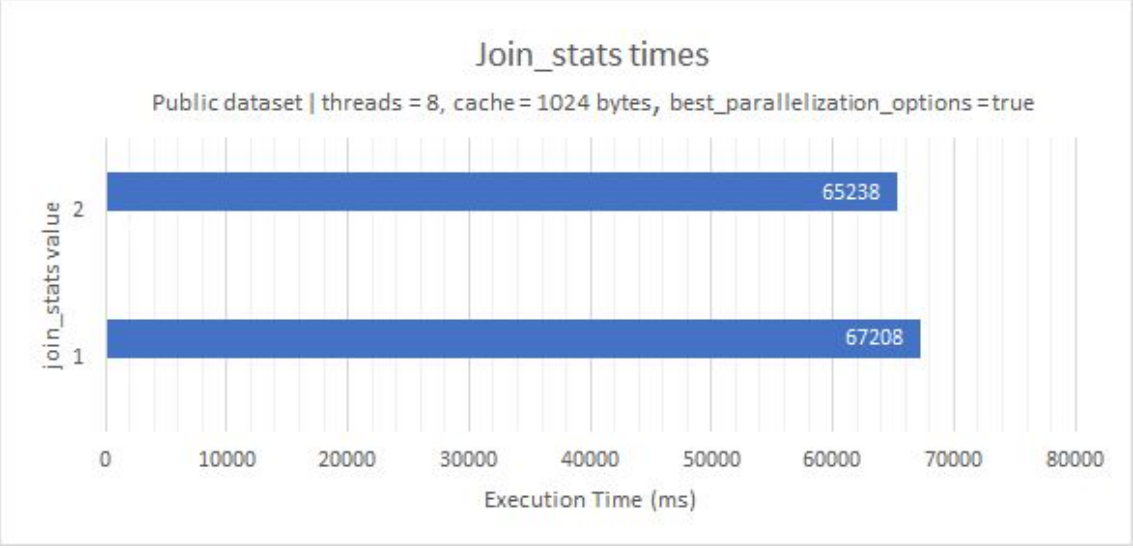


➢ <u>Παράμετρος JOIN_STATS_FOR_F:</u> Αυτή η παράμετρος καθορίζει ποιός τύπος θα χρησιμοποιηθεί για την εκτίμηση του f στα joins. Για:

- join_stats = 0 : εκτιμούμε το κόστος καρτεσιανού γινομένου (fA * fB)
- join_stats = 1 : χρησιμοποιούμε τον τύπο της εκφώνησης (fA * fB / (u-l+1))
- join_stats = 2 : εκτιμούμε το κόστος ως fA * fB / MAX(dA, dB) σύμφωνα με [1]

Ενδεικτικά συγκρίναμε τις τιμές 1 και 2 και πήραμε τα εξής:





Παρατηρήσεις

- Εκτός από τις βελτιστοποιήσεις που αφορούσαν την Παραλληλοποίηση και το Query Optimization για το 3ο παραδοτέο, χρησιμοποιήσαμε επίσης τον profiler του valgrind, callgrind, για να βελτιστοποιήσουμε και τον ακολουθιακό κώδικα του 2ου παραδοτέου. Συγκεκριμένα, ανακαλύψαμε, τρέχοντας το small workload, ότι το 55% των instruction που έτρεχαν ήταν για τον std::unordered_map<unsigned int,...>::operator[] ! Κι αυτό γιατί δεν είχαμε προσέξει ότι κάναμε συνεχώς lookups μέσα σε μεγάλα loops. Αφαιρώντας τελείως το unordered_map στο IntermediateRelation και χρησιμοποιώντας απλούς πίνακες ως hash tables μειώθηκαν τα instruction που τρέχουν στο small από ~10.200.000 σε ~4.500.000!

- Ως άλλη μία βελτίωση του ακολουθιακού κώδικα κάναμε τον joiner (*joiner-main.cpp*) να σταματάει την εκτέλεση ενός query αν ένα QueryRelation περιέχει πίνακα (ενδιάμεσο αποτέλεσμα) με μηδενικό πλήθος από rows, μετά από κάποιο φίλτρο, equal columns ή join του WHERE, αφού τότε σίγουρα το αποτέλεσμα θα είναι κενό.

Βιβλιογραφία:

1. A. Swami, K. B. Schiefer. On the Estimation of Join Result Sizes. Proceedings of the 4th International Conference on Extending Database Technology (EDBT), pp. 287-300, 1994