K23α - Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Άρης Γρηγορόπουλος AM: 1115201800036

Κωνσταντίνος Χωνάς Βαστάρδος AM: 1115201800215

Χειμερινό Εξάμηνο 2022-2023

Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Γενικά (ισχύει για κάθε κομμάτι της εργασίας):

Εντολή makefile για compilation και τρέξιμο με time: make run (δημιουργείται το executable main)

Εντολή για διαγραφή του executable main: make clean

ΜΕΡΟΣ 1

Για το πρόγραμμα έχουμε φτιάξει την βοηθητική συνάρτηση createRelation(int hop), η οποία κατασκευάζει ένα relation με τυχαίο αριθμό από tuples, και ως key αποθηκεύεται το rowID με τη σειρά (0,1,2,3 κλπ.), και ως payload το value το οποίο είναι ένας τυχαίος αριθμός μέσα σε ένα range. Αν το hop > 1, τότε τα rowID θα πηγαίνουν άνα hop, πχ. αν hop == 3, τότε τα rowID θα είναι με τη σειρά (0, 3, 6, 9).Η τιμή κάθε tuple είναι από 0 έως 49 (hardcoded). Όσο μεγαλύτερο είναι το διάστημα μεταξύ των 2 τιμών, τόσα μεγαλύτερο είναι το μέγεθος του result. Αυτό το κάναμε για διάφορους ελέγχους.

Όσο αφορά το partitioning έχει υλοποιηθεί το histogram και το Psum όπως γράφει η εκφώνηση και έχει δημιουργηθεί ο νέος πίνακας με την βοήθεια αυτών των δομών. Γίνεται έλεγχος για την χωρητικότητα του μεγαλύτερου bucket στην L2 2 επαναλήψεις (n=2 ή n=4).

Βρίσκουμε ποιο από τα 2 relations είναι το μικρότερο, και πάνω σε αυτό χτίζουμε τα hash tables (όπως έλεγε και στο paper). Χρησιμοποιώντας το Psum που έχει δημιουργηθεί, βρίσκουμε που ξεκινάει και που τελειώνει το κάθε partition, και δημιουργούμε ένα hash table για καθένα.

Έχει υλοποιηθεί hashtable που ακολουθεί την τεχνική hopscotch hashing, όπως περιγράφεται αλγοριθμικά στην εκφώνηση. Έχει μεταβλητό μέγεθος: Το αρχικό του μέγεθος σε περίπτωση που δεν έχει γίνει partitioning, είναι όσο το πλήθος των tuples του relation. Διαφορετικά, είναι το πηλίκο του πλήθους των tuples των relations με τον αριθμό των partitions. Η γειτονιά έχει μέγεθος logN, όπου N το αρχικό μέγεθος του hashtable. Στο rehash διπλασιάζεται και το μέγεθος του hashtable, αλλά και το μέγεθος της γειτονιάς.

Για να φτιαχτεί το result χρησιμοποιείται η joinRelation, η οποία ελέγχει κάθε στοιχείο του relation στα hash tables, και αν υπάρχει match κρατάει

τα rowIDs και το value, τα οποία και εκτυπώνονται στο τελος του προγράμματος.

Στις δοκιμές μας το πρόγραμμα δε φαίνεται να έχει leaks και βγάζει αποτέλεσμα γρήγορα (λιγότερο από μερικά δευτερόλεπτα), ακόμα και για πολύ μεγάλο πλήθος tuple (>500.000) .

Τα structs που χρησιμοποιήθηκαν για τα relations είναι ίδια με τα υποδειγματικά της εκφώνησης.

Τα **structs** που χρησιμοποιήθηκαν για τα **hash table** είναι:

hashMap: αποτελέιται από έναν πίνακα από hashNodes, και έναν πίνακα int για το bitmap (και μερικά άλλα βοηθητικά δεδομένα)

hashNode: περιέχει την πληροφορία για το payload που του έχει εισαχθεί.

Ενδεικτικοί χρόνοι:

Πλήθος tuple των Relation	**Χρόνος (σε second)**
1000	0.00
10000	0.00
100000	0.022
250000	0.52
500000	1.25
1000000	2.67

ΜΕΡΟΣ 2

Εντολές: make run για compiling και executing του main, μαζί με time. Με το make run δίνονται τα default directories workloads/small/small.init και workloads/small/small.work Τα αρχεία r0,r1,r2... πρέπει να βρίσκονται στο ίδιο directory με τη main.

Το πρόγραμμα επιστρέφει σωστά αποτελέσματα για 34/50 queries του small.work και ελευθερώνει όλο το χώρο που δεσμεύει.

Στην αρχή του προγράμματος γίνεται parsing των bitwise αρχείων r0,r1,r2..., και τα αποτελέσματα τοποθετούνται σε δομή relationInfo, όπου μέσω πολυδιάστατου array μπορούμε να λαμβάνουμε τα payloads από κάθε column κάθε relation. Έπειτα αρχίζει το parsing των queries, οπού όλα τα στοιχεία από τα 3 πεδία τοποθετούνται σε κατάλληλες δομές. Χρησιμοποιούμε το βοηθητικό struct predicate για να διαχειριζόμαστε τα predicates και τις διαφορετικές μορφές που μπορεί να έχουν.

Τα predicates που είναι φίλτρα, εκτελούνται πάντα πρώτα, που κάνει το πρόγραμμα πιο γρήγορο, καθώς πολλά rows κόβονται πριν το κομμάτι των join, απλουστεύοντάς το.

Στην ενδιάμεση δομή, αποθηκεύονται μόνο rowIDs από κάθε δομή, μέσω των οποίων λαμβάνουμε τις τιμές από τα relations όποτε αυτές χρειάζονται. Στην περίπτωση που υπάρχει self-join (πχ. 0.1=0.2), το διαχειριζόμαστε ως φίλτρο (δηλαδή δεν κάνουμε join).

Τα αποτελέσματα αποθηκεύονται σε ένα buffer, το οποίο εκτυπώνεται και αδειάζει κάθε φορά που το πρόγραμμα διαβάζει “F”.

Δεν αλλάξαμε κάτι σχετικά με το πως γίνεται το join από το 1ο μέρος.

Τα **structs** που χρησιμοποιήθηκαν για τα **relation** (επιπλέον σε αυτά από το 1ο μέρος) είναι:

relationInfo: κάθε relInfo περιέχει το από πόσα rows και columns αποτελείται το συγκεκριμένο relation, καθώς και έναν δισδιάστατο πίνακα που περιέχει κάθε value που ανήκει στο relation.

Για τα **queries** δημιουργήθηκε το struct predicate, το οποίο περιέχει όλες τις απαραίτητες πληροφορίες που χρειαζόμαστε για το handling του κάθε query (τι operation είναι, αν πρόκειται για filter ή για join, τα relations και τα columns που συμμετέχουν κλπ.)

Ενδεικτικός χρόνος για το small workload: **13.30 seconds**

ΜΕΡΟΣ 3

Σχετικά με τον job scheduler και τα threads:

Στην αρχή της main δημιουργείται ένας job scheduler όπου αποτελείται από:

- τον αριθμό των threads
- μια ουρά από jobs
- έναν πίνακα από thread_ids
- και μια μεταβλητή isFinished που αν είναι 1 σημαίνει ότι έχουμε παραδώσει όλες τις δουλειές που έπρεπε και μόλις αδειάσει η δουλειά πρέπει να τερματίσουν τα threads

Για την ουρά χρησιμοποιούνται 2 σημαφόροι queue_lock και queue_full

Ο ένας για να ελέγχουμε ότι μόνο ένα thread ανά πάσα στιγμή μπορεί να γράψει στην ουρά μας και ο άλλος αυξάνεται κάθε φορά που μία νέα δουλειά μπαίνει στην ουρά.

Κάθε δουλειά είναι ένα **struct Job** που αποτελείται από έναν δείκτη σε συνάρτηση ένα δείκτη στα ορίσματα της συνάρτησης και έναν δείκτη στην επόμενη δουλειά της ουράς.

Η παραλληλοποίηση έγινε μόνο στα queries. Κάθε query έμπαινε ως job στην ουρά.

Το αποτέλεσμα κάθε query μπαίνει σε μία ξεχωριστή ουρά προτεραιότητας για να είμαστε σίγουροι ότι τα αποτελέσματα εμφανίζονται με την σωστή σειρά όπως μπήκανε.

Η ουρά προτεραιότητας έχει την μορφή:

```
typedef struct {
    Element *heap;
    int size;
    int capacity;
    sem_t lock;
    sem_t full;
} resultQ;
```

όπου heap ένας πίνακας από elements όπου κάθε element περιέχει ένα priority και ένα string (το αποτέλεσμα του εκάστοτε query).

size ο αριθμός των elements μέσα στο heap.
capacity είναι το μέγιστο μέγεθος που έχει το heap όπου αν το φτάσουμε κάνει resize.

2 σημαφόροι ο ένας για να σιγουρευτούμε ότι μόνο ένα thread γράφει στην ουρά ανά πάσα στιγμή και το άλλο αυξάνεται κάθε φορά που μπαίνει ένα element στο heap.

Σχετικά με τα στατιστικά:
Τα αρχικά στατιστικά (min, max, count, discrete), υπολογίζονται κατά το αρχικό parsing των αρχείων και αποθηκεύονται μέσα στο relationInfo, για το οποίο κατασκευάσαμε το νεο **struct** columnInfo, μέσω του οποίου αναθέτουμε στατιστικά για κάθε column του κάθε relation.
Τα στατιστικά αυτά δεν αλλάζουν ποτέ, καθώς ενώ τρέχουμε το πρόγραμμα, η JoinEnumeration δημιουργεί αντίγραφα των αρχικών στατιστικών, τα οποία ανανεώνονται κατάλληλα για το κάθε query.

Σχετικά με το Join Enumeration:
Ο βασικός αλγόριθμος που ακολουθήθηκε, είναι ο Exhaustive Greedy Algorithm (σελίδα 29). Η ιδέα του αλγορίθμου είναι ως εξής:

1. Αρχικά ελέγχουμε το κόστος κάθε predicate, με τη σειρά που βρίσκονται by default. (κόστος θεωρούμε το πλήθος ξεχωριστών tuple που προκύπτουν από το join, το οποίο προκύπτει από τα στατιστικά από την εκφώνηση).
2. Από τα predicates, επιλέγεται να γίνει πρώτο αυτό με το μικρότερο κόστος. Ανανεώνουμε τα στατιστικά (προσωρινά), με βάση αυτό.

3. Ελέγχουμε τα υπόλοιπα predicates, και επιλέγουμε να γίνει 2ο αυτό με το μικρότερο κόστος. Επαναλαμβάνουμε τη διαδικασία μέχρι να ελεγχθούν όλα τα predicates, και επιστρέφουμε τα predicates στην σειρά που βρήκαμε να είναι πιο αποδοτική.

Επειδή ο αλγόριθμος είναι greedy, πιθανώς να μη δίνει το βέλτιστο αποτέλεσμα για το κάθε query. Επιλέξαμε αυτόν όμως, επειδή είναι πολύ απλός στην κατανόηση, και μέσω των δοκιμών μας, είδαμε πως βελτιώνει τον χρόνο της εκτέλεσης κατά 10%-30%.

Όπως και στο Μέρος 2, το πρόγραμμα πάντα εκτελεί πρώτα τα predicates που είναι φίλτρα.

Ελέγξαμε τη μνήμη του προγράμματος με valgrind (στα linux της σχολής), και τα αποτελέσματα φάνηκαν σωστά, αλλά το πρόγραμμα έκανε πολύ ώρα για να τελείωση (>20 λεπτά).

η cpu του υπολογιστη στον οποιο το τρεξαμε ειναι:

```
Model name:            Intel(R) Core(TM) i3-4170 CPU @ 3.70GHz
  CPU family:          6
  Model:               60
  Thread(s) per core:  2
```

χρονοι για το αρχειο small:

Αριθμος threads	**Χρόνος με join enumeration**	**Χρόνος <u>χωρίς</u> join enumeration**
1	12.19 sec	13.89 sec
2	9.06 sec	11.61 sec
4	9.44 sec	10.92 sec
8	9.69 sec	12.35 sec