Ανάπτυξη Λογισμικού για Πληροφοριακά Συστήματα

Άσκηση 3

Γιαννακίδης Ιωάννης
1115201500025

Σταυρακάκης Νικόλαος
1115201500149

Περιεχόμενα

Εισαγωγή

Τα αρχεία που περιλαμβάνονται στο github είναι τα εξής:

- DataQueryStoring.c
- JobScheduler.c
- PredicatesExecution.c
- QueryManagement.c
- QueryManagement.h
- QueryOptimization.c
- RadixHashJoin.c
- RadixHashJoin.h
- cardinality.c

Επιπρόσθετα, ο φάκελος small που περιέχει τα data,το Makefile καθώς και το data.txt για να δοθεί ως input κατά την εκτέλεση του προγράμματος.

Εντολή εκτέλεσης: `./QueryManagement < data.txt`

Α. Περιγραφή Εργασίας

Α1. Αποθήκευση Δεδομένων

Αρχικά το πρόγραμμα μας στην main στο αρχείο QueryManagement.c διαβάζει όλα τα file names που περιέχουν τα δεδομένα του προγράμματος μας και γίνεται η αποθήκευση τους σε μία λίστα. Στην συνέχεια στην συνάρτηση data_storing() αποθηκεύονται τα δεδομένα. Τα δεδομένα μας αποθηκεύονται σε έναν πίνακα από struct τύπου data.

```
struct data{
    int num_columns;                    //αριθμός στηλών σχέσης
    int num_tuples;                     //αριθμός πλειάδων
    statistics *stats;                  //στατιστικά
    uint64_t **positions;               //δείκτες σε κάθε στήλη
} ;
```

Μετέπειτα, περνάμε στον υπολογισμό των στατιστικών για κάθε σχέση-στήλη. Αυτό γίνεται μέσω της συνάρτησης data_stats(), όπου υπολογίζονται τα στατιστικά και αποθηκεύονται στην δομή struct τύπου statistics.

```
struct statistics{
   uint64_t min;
   uint64_t max;
   int size;
   int distinct;
} ;
```

Μέσω της query_reading() γίνεται η ανάγνωση των queries και γίνεται η αποθήκευση τους σε μία λίστα μέχρις ότου να διαβαστεί ο χαρακτήρας "F". Μετά για κάθε query που αποθηκεύσαμε γίνεται η κλήση της συνάρτησης predicates_execution(), όπου γίνεται η εκτέλεση του query.

A2. RadixHashJoin

Μέσω της συνάρτησης RadixHashJoin() γίνεται η υλοποίηση του αλγορίθμου που ζητήθηκε στο πρώτο μέρος. Γίνεται η κλήση της itemization() για τα πρώτα 2 παράλληλα μέρη για κάθε relation και στην συνέχεια γίνεται το 3ο μέρος καθώς και η αποθήκευση των result. Αναλυτικότερα στην συνάρτηση itemization() τοποθετούνται αρχικά στον job scheduler οι θέσεις όπου θα πραγματοποιήσει τον κατακερματισμό το κάθε thread. Στην συνέχεια μέσω του pthread_cond_t barrier_cond ενημερώνουν τα thread πως υλοποίησαν το 1ο παράλληλο μέρος για να γίνει η εισαγωγή στον job scheduler των στοιχείων για το 2ο παράλληλο μέρος της κατασκευής των νέων relation. Μετά την ολοκλήρωση αυτής της διαδικασίας και για τα 2 relation περνάμε στο 3ο παράλληλο μέρος. Εκεί τοποθετούμε στον job scheduler τόσα jobs όσα και τα bucket έτσι ώστε κάθε thread να αναλαμβάνει ένα thread την φορά. Τα thread αποθηκεύουν τα αποτελέσματα το καθένα στην δική του δομή result (με μέγεθος buffer 128KB). Στην συνέχεια,αφού ολοκληρωθούν όλα τα jobs τα threads ολοκληρώνουν την λειτουργία τους και περνάμε στο merge των result σε ένα τελικό result (με μέγεθος buffer 1024MB).Στην δομή result έχει προστεθεί και ένας επιπλέον πίνακας από rowids για την ενημέρωση του intermediate,λειτουργία που περιγράφεται στην συνέχεια.

A3. Εκτέλεση Queries

A3.1 Διαχωρισμός Query

Στην συνάρτηση predicates_execution() γίνεται η εκτέλεση των queries. Αρχικά "σπάμε" το κάθε query στα 3 μέρη του. Το πρώτο μέρος περιέχει τις σχέσεις,από όπου παίρνουμε κάθε μία και την αποθηκεύουμε σε έναν πίνακα προκειμένου να κάνουμε την αντιστοίχιση των κατηγορημάτων. Το 3ο μέρος που περιέχει τις προβολές το χρησιμοποιούμε μόνο κατά τον υπολογισμό του αποτελέσματος. Το 2ο μέρος που περιέχει τα predicates το αποθηκεύουμε σε έναν πίνακα struct τύπου pred.

```
 struct pred{
    int flag_0;
    int flag_1;
    char *c_0,*c_1;
    char *side_0,*side_1;
    char *rlt_id_0,*rlt_id_1;
    int compare;
} ;
```

Έχουμε έναν τέτοιο πίνακα για τα φίλτρα('>','<','=') στήλης με κάποια αριθμητική τιμή και έναν για όλα τα join(ανεξάρτητα αν αποτελούν κάποια ειδική περίπτωση).

A3.2 Intermediate

Η δομή intermediate είναι ένας πίνακας από struct του τύπου im_node μεγέθους MAX_RELATION (=4).

```
struct node{
    result *r;              //δείκτης σε result
    int position;
    int im_id;
};
```

Κάθε κελί του intermediate που περιέχει ένα im_node έχει και το im_id προκειμένου να γνωρίζουμε αν 2 προσωρινά αποτελέσματα στον intermediate είναι της ίδιας οντότητας intermediate. Δηλαδή,αν έχουμε πρώτα εκτελέσει ένα φίλτρο 0.1>152 το συγκεκριμένο κελί θα έχει im_id=0 και αν στην συνέχεια εκτελέσουμε το join 1.2=2.3 τα 2 κελιά για τις σχέσεις 1,2 θα έχουν im_id=1.Αν στην συνέχεια έχουμε 0.4=1.1 τότε και τα 3 κελιά θα έχουν im_id=2 κτλ.

Επίσης,για να γλυτώσουμε την αντιγραφή των αποτελεσμάτων στον intermediate μετά από την εκτέλεση κάθε query,κάθε κελί έχει έναν δείκτη στο result του predicate που εκτελέστηκε.

Πρώτα για τα κατηγορήματα φίλτρου παίρνουμε το μέρος που αποτελεί την σχέση μας και κατασκευάζουμε το relation από την αντίστοιχη στήλη ανάλογα με την περίπτωση όπου η σχέση μας βρίσκεται ήδη στη δομή μέσα στον intermediate ή όχι. Στην συνέχεια εκτελούμε το κατηγόρημα μέσω της predicate_filter() καθώς και ανανεώνουμε τα στατιστικά μέσω των συναρτήσεων assessment_() που θα περιγραφούν στην συνέχεια. Η predicate_filter() επιστρέφει έναν δείκτη σε result.

Στην συνέχεια για τα κατηγορήματα σύζευξης πρώτα γίνεται το query optimization μέσω της optimization() όπου θα περιγραφεί στην συνέχεια. Μετά για κάθε predicate κατασκευάζουμε τα relation ανάλογα με το αν το αριστερό και δεξί μέρος της σύζευξης βρίσκεται ή όχι στον intermediate. Στην συνέχεια έχουμε 3 περιπτώσεις:

1. Να αφορούν και τα 2 μέρη την ίδια σχέση

   Μέσω της same_array() γίνεται η διαδικασία. Ταυτόχρονα για όλες τις άλλες σχέσεις που βρίσκονται στον intermediate και έχουν ίδιο im_id με τις σχέσεις που συμμετέχουν στην σύζευξη γίνεται και σε αυτές η ενημέρωση τους.

2. Να βρίσκονται και τα 2 μέρη στον intermediate

   Μέσω της bothIM() γίνεται η διαδικασία. Όπως και στην περίπτωση a) γίνεται το update των άλλων σχέσεων του intermediate.

3. RadixHashJoin

   Εκτέλεση της RadixHashJoin().Στην δομή result έχουμε και έναν πίνακα από position και για τα 2 relation που συμμετείχαν στην σύζευξη. Αυτός ο πίνακας κρατάει τις θέσεις των rowid_S,rowid_R στον intermediate πριν το join που αποθηκεύτηκαν στο result. Με αυτόν

τον τρόπο κάνουμε το update μέσω της im_update().Αναλυτικότερα,θα αναλυθεί καλύτερα μέσω ενός παραδείγματος. Έστω intermediate:

θέση	**relation**	**im_id**
0	1	3
1	2	3
2	3	4
3	4	4

Και έχουμε την σύζευξη 2.1=3.1 τότε αφού γίνει η RadixHashJoin() για τις σχέσεις 2,3 θα πρέπει να κάνουμε ενημέρωση της relation 1 με βάση το result για την 2 και της relation 4 με βάση το result της 3.Μέσω της δομής που κρατάμε τις προηγούμενες θέσεις των σχέσεων 2,3 στον intermediate για κάθε rowid που γράφτηκε στο result μπορούμε να κάνουμε το update με ένα πέρασμα του result και για κάθε rowid πχ της σχέσης 2 να πάρουμε την θέση στον intermediate και να αποθηκεύσουμε το αντίστοιχο rowid της σχέσης 1 στον intermediate κτλ.

Τέλος το im_id όλων θα γίνει ίσο με 5.

Με την εκτέλεση και των 3 περιπτώσεων γίνεται και η ενημέρωση των im_id.

Τέλος, παίρνουμε το 3ο μέρος από το query και υπολογίζουμε με βάση τον intermediate το τελικό ζητούμενο αποτέλεσμα.

A4. Query Optimization

A4.1 Εκτίμηση Πληθικότητας

Η εκτίμηση της πληθικότητας επιτυγχάνεται μέσω των συναρτήσεων:

- assessment_equal() , R.A=k
- assessment_bigger() , R.A>k
- assessment_smaller() , R.A<k
- assessment_same_rel() , R.A=R.B
- assessment_same_rel_col() , R.A=R.A
- assessment_join() , R.A=S.B

A4.2 Join Enumeration

Μέσω της συνάρτησης optimization() επιτυγχάνεται το query optimization. Αρχικά,για όλα τα relation που έχουμε για το query υπολογίζουμε το κόστος για όλα τα μονοσύνολα. Ως κόστος του κάθε μονοσύνολου είναι ο αριθμός των πλειάδων του. Στην συνέχεια για join του query βρίσκουμε όσα είναι connected με το καλύτερο μονοσύνολο και κρατάμε το join με το μικρότερο κόστος. Ως κόστος για τα σύνολα 2 σχέσεων είναι το πλήθος των ενδιάμεσων αποτελεσμάτων,δηλαδή ο αριθμός των πλειάδων και των 2 σχέσεων μετά τον υπολογισμό τους. Μετέπειτα,αν έχουμε 3 σχέσεις,βρίσκουμε όσες είναι connected και υπολογίζουμε το κόστος. Τέλος,αναδιατάσσουμε την σειρά των κατηγορημάτων με βάση τις σχέσεις που έχουμε στο δέντρο μας. Το δέντρο μας είναι struct τύπου lefttree.

```
struct lefttree{
    int pos[MAX_RELATIONS];
    int cost;
}
```

Β. Πειράματα – Μετρήσεις

Οι μετρήσεις πραγματοποιήθηκαν στα Linux της σχολής,συγκεκριμένα στο linux23. Πραγματοποιήθηκαν 10 φορές για το καθένα και κρατήσαμε τον μέσο όρο με ακρίβεια 2 δεκαδικών ψηφίων.

Β1. Χωρίς Query Optimization

Αριθμός thread	**Χρόνος (sec)**
2	2.78
4	2.69
10	2.78
25	2.74
50	2.91
100	3.19

Β2. Με Query Optimization

Αριθμός thread	**Χρόνος(sec)**
2	2.85
4	2.59
10	2.68
25	2.61
50	2.71
100	2.91

Β3. Ιδανικά Αποτελέσματα

Τα αποδοτικότερα χρονικά αποτελέσματα ήταν με query optimization και αριθμό από thread ίσο με τον αριθμό των πυρήνων του μηχανήματος,μέσω της συνάρτησης get_nprocs().