

Κ23α - Ανάπτυξη Λογισμικού Για Πληροφοριακά Συστήματα

Χειμερινό Εξάμηνο 2025-2026

Άσκηση 3 – Παράδοση: Κυριακή 16 Ιανουαρίου 2026

Βελτιστοποίηση indexing

Στην τρέχουσα υλοποίηση πραγματοποιούνται αντιγραφές σε όλες τις σελίδες των στηλών των πινάκων. Αυτό έχει νόημα σε στήλες με VARCHAR και σε στήλες με INT32 που όμως περιέχουν κενά, προκειμένου να γίνουν πιο εύκολα διαχειρίσιμες. Σε στήλες με INT32 χωρίς null τιμές, δεν είναι απαραίτητο να γίνει κάποια αντιγραφή. Αντίθετα, μπορεί να δημιουργηθεί το index απευθείας στις σελίδες που παρέχονται από την είσοδο. Θα χρειαστεί να προσαρμόσετε τη δημιουργία του columnnt_t ώστε να μπορεί να δεικτοδοτεί την αρχική στήλη που δίνεται ως δεδομένο.

Υπόδειξη: Δείτε την επικεφαλίδα (header) των σελίδων των στηλών για να καταλάβετε ποιες στήλες έχουν null τιμές και ποιες όχι.

Παράλληλη Εκτέλεση

Η παράλληλη εκτέλεση μπορεί να πραγματοποιηθεί χωρίς τους παραδοσιακούς μηχανισμούς συγχρονισμού pthreads (mutexes, condition variables, semaphores, etc.). Αρκεί να χωριστεί σε φάσεις, όπου στο τέλος κάθε φάσης θα πρέπει να ολοκληρώνεται το έργο των παράλληλων thread (π.χ. με pthread_join() στο parent thread). Μπορείτε να επιλέξετε με ποιον τρόπο θα υλοποιήσετε την παράλληλη εκτέλεση (pthreads, STL <thread>, OpenMP, κλπ.)

Η παράλληλη εκτέλεση εργασιών μπορεί να υλοποιηθεί σε όλα τα στάδια της εκτέλεσης του προγράμματος: κατασκευή hashtable, probing hashtable, joins.

Παραλληλοποίηση κατασκευής hash table

Για να μπορέσουμε να επιτύχουμε αποδοτική παράλληλη κατασκευή του hashtable θα χρησιμοποιήσουμε partitions και 2 φάσεις κατασκευής. Στο πρώτο βήμα για την κατασκευή, συλλέγουμε όλα τα tuples και τις διαχωρίζουμε σε hash partitions. Κάθε thread κατανέμει τα tuples που κατακερματίζει σε προσωρινά κομμάτια μνήμης κατάλληλα να αποθηκεύουν tuples ανά partition. Στη συνέχεια, ένα thread ανά partition συλλέγει όλα τα κομμάτια μνήμης που αντιστοιχούν στο ίδιο partition, μετράει τον αριθμό των tuples που περιέχει και αποθηκεύει τα tuples σε ταξινομημένη σειρά κατακερματισμού στην τελική θέση μνήμης.

Συνοπτικά, για τον πίνακα γειτνίασης που χρησιμοποιούμε ως τελική αποθήκευση tuples, χρησιμοποιούμε ένα συνεχόμενο μπλοκ μνήμης. Στην τελική αποθήκευση, τα tuples ταξινομούνται με βάση τις τιμές κατακερματισμού τους. Όλα τα tuples που αντιστοιχούν σε μια καταχώρηση στον

κατάλογο αποθηκεύονται σε ένα συνεχόμενο μπλοκ μνήμης και οι γειτονικές καταχωρήσεις στον κατάλογο δείχνουν σε γειτονικά μπλοκ tuples, με σειρά κατακερματισμού. Για να προσδιορίσουμε την κατανομή των tuples στον τελικό χώρο αποθήκευσης, μετράμε τον αριθμό των tuples ανά καταχώρηση καταλόγου. Στη συνέχεια, τα tuples εγγράφονται στον τελικό χώρο αποθήκευσης, ενώ ταυτόχρονα ενημερώνεται ο κατάλογος.

Συλλογή tuples

Στο σχέδιο εκτέλεσης ενός ερωτήματος βάσης δεδομένων, μια ζεύξη βρίσκεται πάνω από αυθαίρετους τελεστές που παράγουν δεδομένα που εισάγονται στη ζεύξη. Αυτά τα δεδομένα παράγονται ως ροή tuples. Ο συνολικός αριθμός tuples δεν είναι γνωστός εκ των προτέρων και οι εκτιμήσεις αυτού ενδέχεται να μην είναι ακριβείς. Η συλλογή tuples πρέπει να αντιμετωπίσει αυτήν την αβεβαιότητα και να είναι σε θέση να χειριστεί έναν μεταβλητό αριθμό ταυτόχρονα παραγόμενων tuples, να τις υλοποιήσει και να τις διαμερίσει με κατακερματισμό πριν από την κατασκευή του καταλόγου.

Η συλλογή tuples συχνά παρουσιάζει συμφόρηση από τον memory allocator, καθώς πολλά μεμονωμένα tuples πρέπει να κατανεμηθούν και να υλοποιηθούν ταυτόχρονα. Για να αποφύγουμε αυτή τη συμφόρηση, χρησιμοποιούμε έναν slab allocator που κατανέμει μνήμη σε μεγάλα κομμάτια και στη συνέχεια κατανέμει μνήμη από αυτά τα κομμάτια σε μεμονωμένα tuples. Αυτό μειώνει τον αριθμό των κλήσεων συστήματος και τον ανταγωνισμό εντός του καθολικού κατανεμητή μνήμης. Η μνήμη απελευθερώνεται στη συνέχεια μεμιας μετά την ολοκλήρωση της κατασκευής. Η στρατηγική κατανομής bump περιπλέκεται από την διαμέριση tuples, καθώς θέλουμε τα tuples εντός της ίδιας διαμέρισης να είναι ως επί το πλείστον συνεχόμενα στη μνήμη, για να κάνουμε τις μεταγενέστερες επαναλήψεις σε tuples μέσα σε μεμονωμένα διαμερίσματα αποδοτικές. Για να το πετύχουμε αυτό, χρησιμοποιούμε έναν slab allocator τριών επιπέδων:

- Το πρώτο επίπεδο κατανέμει μνήμη σε κομμάτια για κάθε thread
- Το δεύτερο επίπεδο κατανέμει μικρότερα κομμάτια ανά διαμερίσματα και
- Το τρίτο επίπεδο κατανέμει μεμονωμένα tuples από τα μικρά κομμάτια.

Ο ψευδοκώδικας για αυτό φαίνεται στο Σχήμα 1.

```
1 GlobalAllocator & level1 ;
2 BumpAlloc level2 , level3 [ numPartitions ];
3 size_t counts [ numPartitions ];
4 void consume (T tuple ) {
5     u64 part = tuple . hash >> (64 - log2 ( numPartitions ));
6     if level3 [ part ]. freeSpace () < sizeof ( tuple ):
7         if level2 . freeSpace () < sizeof ( BumpAlloc ):
8             level2 . addSpace ( level1 . allocate < LargeChunk >());
9     level3 [ part ]. addSpace ( level2 . allocate < SmallChunk >());
10    * level3 [ part ]-> allocate <T >() = tuple ;
11    counts [ part ] += 1;
12 }
```

Σχήμα 1. Λογική κατανομής τριών επιπέδων σε τοπικό thread για τη συλλογή tuples. Τα υψηλότερα bit του hash χρησιμοποιούνται για τον προσδιορισμό της διαμέρισης (partition). Ο πίνακας counts χρησιμοποιείται για την παρακολούθηση του αριθμού των tuples ανά διαμέριση.

Αφού συλλεχθούν όλα τα tuples, τα partitions πρέπει να ανταλλαχθούν μεταξύ των threads. Αυτό γίνεται χρησιμοποιώντας τις εσωτερικές δομές των slab allocators. Οι slab allocators αποθηκεύουν τα κομμάτια μνήμης τους σε μια συνδεδεμένη λίστα. Πριν ένα thread επεξεργαστεί ένα partition, συγχωνεύει όλες τις συνδεδεμένες λίστες των κομματιών που αντιστοιχούν στο partition από όλα τα threads. Στη συνέχεια, το thread μπορεί να διασχίσει τα tuples μπορούν αποτελεσματικά ως μία ενιαία λίστα σε κομμάτια.

Μέτρηση tuples

Μετά την ολοκλήρωση της συλλογής των tuples, χρειάζεται να κατασκευαστεί το directory και να αντιγραφούν όλα τα tuples στον τελικό χώρο συμπαγούς αποθήκευσης. Πριν την αντιγραφή στην τελική θέση αποθήκευσης, πρέπει να καθοριστούν οι ζώνες μνήμης στις οποίες θα αποθηκευτούν τα tuples που έχουν το ίδιο πρόθεμα hash. Αυτό μπορεί να γίνει μετρώντας τον αριθμό των tuples ανά εγγραφή καταλόγου. Στη συνέχεια παράγεται ένα άθροισμα προθεμάτων (prefix sum) για να καθοριστούν οι ζώνες στις οποίες θα αποθηκευτούν τα tuples. Οι ζώνες χρησιμοποιούνται στη συνέχεια για να αντιγραφούν τα tuples στη σωστή τοποθεσία στην τελική θέση αποθήκευσης.

Αντιγραφή

Το τελικό βήμα της διαδικασίας κατασκευής είναι η αντιγραφή των tuples στην τελική τους θέση. Μετά την καταμέτρηση, κάθε καταχώρηση καταλόγου περιέχει την αρχή του εύρους των tuples που μοιράζονται το ίδιο πρόθεμα κατακερματισμού. Καθώς διασχίζουμε τα tuples, αντιγράφουμε τα tuples στην αρχή και στη συνέχεια αυξάνουμε τον δείκτη. Στο τέλος, κάθε καταχώρηση καταλόγου θα δείχνει στο τέλος του αντίστοιχου εύρους και, επομένως, ο δείκτης της προηγούμενης καταχώρησης μπορεί να χρησιμοποιηθεί για να καθορίσει την αρχή. Επιπλέον, ένας ειδικός κατάλογος καταχωρήσεων[-1] χρησιμοποιείται για να δείχνει στην αρχή της αποθήκευσης tuples. Αυτό επιτυγχάνεται αρχικά εκχωρώντας έναν επιπλέον χώρο για τον πίνακα, ορίζοντας αυτήν την πρώτη καταχώρηση στην αρχή της αποθήκευσης tuples και στη συνέχεια ολισθαίνοντας τον δείκτη του καταλόγου κατά μία θέση. Αυτό γίνεται για να αποφευχθεί πιθανή διακλάδωση κατά την αναζήτηση.

Ο ψευδοκώδικας για τη μέτρηση και την αντιγραφή παρουσιάζεται στο Σχήμα 2:

```
1 T partitionTuples [][];
2 T* tupleStorage ;
3 void postProcessBuild ( u64 partition , u64 prevCount ) {
4     for (T tuple : partitionTuples [ partition ]) {
5         u64 slot = tuple . hash >> shift ;
6         directory [ slot ] += sizeof (T) << 16;
7         directory [ slot ] |= computeTag ( tuple . hash );
8     }
9     // prevCount is the total tuple count of previous partitions
10    u64 cur = tupleStorage + prevCount ;
11    u64 k = 64 - shift ;
```

```

12  u64 start = ( partition << k) / numPartitions ;
13  u64 end = (( partition + 1) << k) / numPartitions ;
14  for ( u64 i = start ; i < end ; ++i) {
15      u64 val = directory [i] >> 16;
16      directory [i] = ( cur << 16) | (( u16 ) directory [i]);
17      cur += val ;
18  }
19  for (T tuple : partitionTuples [ partition ]) {
20      u64 slot = tuple . hash >> shift ;
21      T* target = directory [ slot ] >> 16;
22      * target = tuple ;
23      directory [ slot ] += sizeof (T) << 16;
24  }
25 }

```

Σχήμα 2. Πλήθος, άθροισμα αποκλειστικού προθέματος και αντιγραφή. Ο κατάλογος χρησιμοποιείται για την καταμέτρηση του αριθμού των tuples ανά πρόθεμα κατακερματισμού, και στη συνέχεια οι μετρήσεις χρησιμοποιούνται για τον προσδιορισμό των ζωνών στις οποίες θα αποθηκευτούν τα tuples. Τέλος, τα tuples αντιγράφονται στην τελική τους θέση. Πρέπει να δοθεί προσοχή στον χειρισμό των φίλτρων Bloom.

Σύνοψη παραλληλοποίησης join

1. Threaded partitioning (threads όσα και τα partitions)
2. Ένα μόνο thread θα πρέπει να συλλέξει όλα τα partitions και να τα συγχωνεύσει ανά πρόθεμα (directory entry)..
3. Για κάθε directory entry, θα πρέπει κάποιο thread να εκτελέσει την κατασκευή για αυτό το τμήμα του hashtable
4. Threaded probing
5. Single-Threaded aggregation των αποτελεσμάτων

Work stealing

Στην περίπτωση που χρησιμοποιείται ένας συγκεκριμένος αριθμός threads και ένα ή περισσότερα από αυτά έχει ολοκληρώσει την εργασία του, μπορεί να "κλέψει" δουλειά που υπάρχει ακόμα σε κάποιο από τα άλλα threads.

Αυτό μπορεί να συμβεί π.χ. κατά τη διάρκεια του probing. Όταν ένα thread τελειώσει με τη δική του δουλειά, μπορεί να "κλέψει" δουλειά από ένα άλλο thread που είναι έχει ακόμη υπόλοιπο δουλειάς, δηλαδή να επεξεργαστεί σελίδες που είχαν ανατεθεί αρχικά σε άλλο thread.

Η γενική ιδέα υλοποίησης είναι η εξής: Χρησιμοποιώντας έναν ατομικό μετρητή, μπορεί το thread να (α) διαβάσει και να αυξάνει κατά 1 την τιμή του μετρητή σε μία ατομική πράξη και (β) να προχωράει με την ολοκλήρωση της ενέργειας. Ένα thread που θέλει να "κλέψει" δουλειά μπορεί να κάνει ακριβώς το ίδιο

πράγμα αυξάνοντας τον ατομικό μετρητή και εκτελώντας το επόμενο κομμάτι δουλειάς που είχε να κάνει το αρχικό thread.

Στο τέλος θα χρειαστεί κάποια διαχείριση στη συγχώνευση των αποτελεσμάτων.

Τελική Αναφορά

Στη τελική αναφορά θα παρουσιάσετε μια σύνοψη ολόκληρης της εφαρμογής που υλοποιήθηκε. Μπορείτε να αναφέρετε πράγματα που παρατηρήσατε κατά την μοντελοποίηση/υλοποίηση της εφαρμογής σας, με αποτέλεσμα να σας οδηγήσουν σε συγκεκριμένες σχεδιαστικές επιλογές που βελτίωσαν την εφαρμογή σας σε επίπεδο χρόνου, μνήμης, κτλ.

Στην αναφορά πρέπει ακόμη να παρουσιάσετε ένα σύνολο από πειράματα, τα οποία θα δείχνουν το χρόνο εκτέλεσης για τις επιλογές των δομών που θα επιλέξετε. Επειδή δεν μπορείτε να κάνετε πειράματα που να δείχνουν την επίδραση κάθε παραμέτρου, θα πρέπει να επιλέξετε εσείς ποια παράμετρο θα παρουσιάζετε κάθε φορά.. Για παράδειγμα μπορείτε να αναφέρετε ή/και να παρουσιάσετε με διαγράμματα τον χρόνο εκτέλεσης, κατανάλωση μνήμης, παράλληλης εκτέλεσης κ.α.

Τέλος, είναι απαραίτητο να παρουσιάσετε τις δομές που σας έδωσαν τους καλύτερους χρόνους εκτέλεσης για τα datasets και τον τελικό χρόνο/μνήμη, μαζί με τις προδιαγραφές του μηχανήματος που εκτελέσατε τα πειράματα. Η αναφορά δεν πρέπει να ξεπερνά τις 10 σελίδες.

Παράδοση εργασίας

Η εργασία είναι ομαδική, **2 ή 3 ατόμων**.

Γλώσσα υλοποίησης: C++

Περιβάλλον υλοποίησης: Linux (gcc >= 9.4+).

Παραδοτέα: Η παράδοση της εργασίας θα γίνει με βάση το τελευταίο commit πριν την προθεσμία υποβολής στο git repository σας. **Η χρήση git είναι υποχρεωτική.**

Στο αρχείο README.md θα αναφέρονται τα εξής:

- Ονοματεπώνυμο και ΑΜ των μελών της ομάδας
- Αναφορά στο ποιο μέλος της ομάδας ασχολήθηκε με ποιο αντικείμενο

Επιπλέον, εκτός από τον πηγαίο κώδικα, θα παραδώσετε μια σύντομη αναφορά με την ανάλυση της απόδοσης των αλγορίθμων. Θα πρέπει ακόμα να εφαρμόσετε ελέγχους ως προς την ορθότητα του λογισμικού με τη χρήση ανάλογων βιβλιοθηκών ([Software testing](#)). Η ορθότητα τυχόν μεταβολών θα ελέγχεται με αυτοματοποιημένο τρόπο σε κάθε commit/push μέσω github actions.

Αναφορές

1. Altan Birler, Tobias Schmidt, Philipp Fent, and Thomas Neumann. 2024. Simple, Efficient, and Robust Hash Tables for Join Processing. In Proceedings of the 20th International Workshop on Data Management on New Hardware (DaMoN '24). Association for Computing Machinery, New York, NY, USA, Article 4, 1–9. <https://doi.org/10.1145/3662010.3663442>