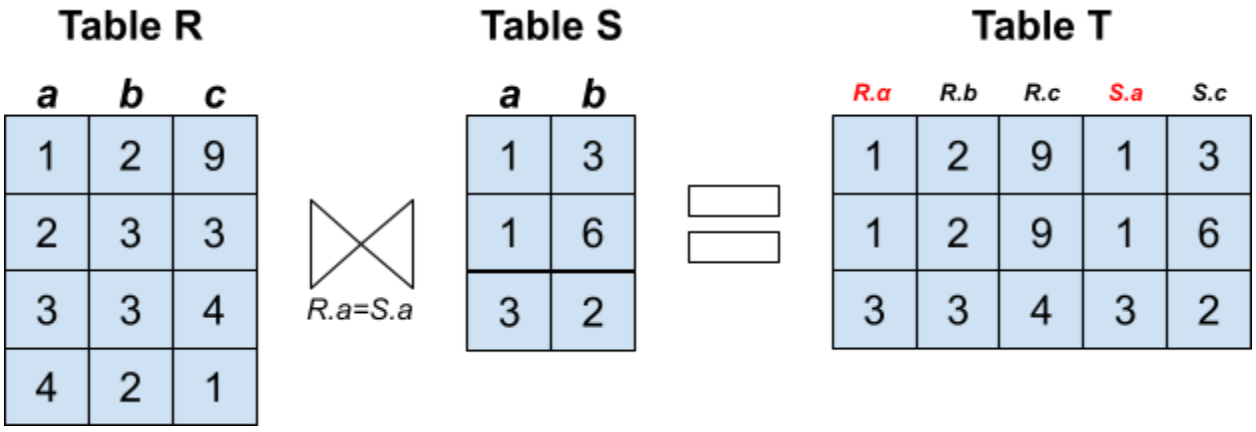
Κ23α - Ανάπτυξη Λογισμικού Για Πληροφοριακά Συστήματα

Χειμερινό Εξάμηνο 2025-2026

Άσκηση 2 – Παράδοση: Κυριακή 7 Δεκεμβρίου 2025

Row-store ή Column-store πίνακες

Παρακάτω περιγράφεται ο τρόπος που δουλεύει ένας τελεστής ζεύξης ισότητας. Στην πιο κάτω εικόνα, φαίνεται το αποτέλεσμα της πράξης $R.a = S.a$ μεταξύ δύο σχεσιακών πινάκων (R, S):



Η πιο πάνω απεικόνιση θεωρεί ότι οι πίνακες αποθηκεύονται στην μνήμη κατα σειρές (row-store). Αυτό πρακτικά σημαίνει, ότι το κάθε στοιχείο μιας στήλης θα απέχει από το επόμενο του στη μνήμη, τόσα στοιχεία μακριά όσα και ο αριθμός των στηλών της αντίστοιχης σχέσης. Αυτό σε αναλυτικά workloads μπορεί να αποβεί χρονοβόρο, καθώς συνήθως οι πράξεις γίνονται μεταξύ στηλών (όπως είδαμε και στην ζεύξη). Με τη row-store οργάνωση διατρέχοντας μια σχέση, κάνουμε προσβάσεις στη μνήμη μακριά μεταξύ τους για να διαβάσουμε το περιεχόμενο της ζητούμενης στήλης, και ως εκ τούτου μη χρησιμοποιώντας δεδομένα που βρίσκονται ήδη στην cache του επεξεργαστή. Προκειμένου να επιτευχθεί καλό cache locality, θα πρέπει τα δεδομένα που διαβάζονται διαδοχικά να είναι όσο το δυνατόν πιο κοντά στα προηγούμενα.

Μια άλλη τεχνική, αποθηκεύει τις σχέσεις ανά στήλες (column-store [2]). Αυτό σημαίνει, ότι στη μνήμη τα στοιχεία κάθε στήλης είναι σειριακά το ένα δίπλα στο άλλο. Με αυτόν τον τρόπο, αν θέλουμε να διαβάσουμε μια στήλη, “ακουμπάμε” συνεχόμενη μνήμη. Σημειώνουμε ότι τα δεδομένα δίνονται σε αυτό το format (διαβάστε αναλυτικά το readme του διαγωνισμού) κάτι που μπορούμε να εκμεταλλευτούμε στα analytical queries του JOB.

Για αυτό το παραδοτέο και τα επόμενα μπορείτε να θεωρήσετε τα εξής:

- Οι ζεύξεις θα γίνονται πάνω σε στήλες INT32.
- Όλες οι στήλες είναι είτε INT32 είτε VARCHAR.
- Αν μια στήλη δεν έχει NULL τιμές τότε τα δεδομένα της είναι συμπαγή, δηλαδή μια σελίδα έχει ακριβώς τόσες τιμές όσες χωράει - εκτός της τελευταίας.

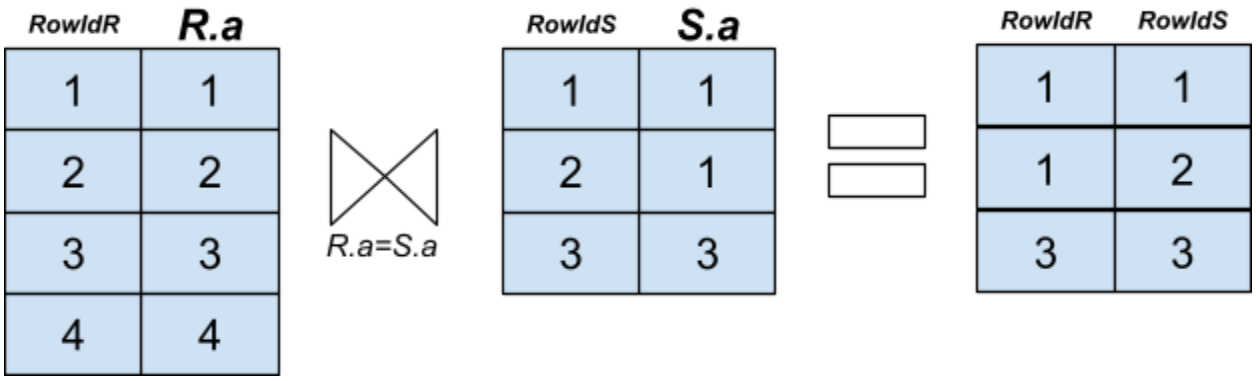
Με βάση αυτές τις παρατηρήσεις/υποθέσεις, τα ζητούμενα του παραδοτέου είναι:

(Σημείωση: για τα ζητούμενα 1 και 2 μπορείτε να χρησιμοποιήσετε όποιο hashtable θέλετε από το πρώτο παραδοτέο ή και το hashtable της STL) προκειμένου να δοκιμάσετε/ελέγξετε την υλοποίησή σας.

1. Καλύτερος χειρισμός strings

Late Materialization

Το late materialization είναι μια τεχνική βελτιστοποίησης query execution που καθυστερεί τη διαδικασία αντιγραφής των τιμών από τα δεδομένα της εισόδου μέχρι να καταστεί απολύτως απαραίτητη. Δηλαδή, στην αρχή του query αντί να ανακτά όλες τις τιμές των στηλών ταυτόχρονα, ξεκινά με τη στήλη που χρειάζεται η επόμενη ζεύξη και ανακτά τα υπόλοιπα δεδομένα σταδιακά, ακριβώς στο σημείο που χρειάζεται. Η ανάκτηση αυτή γίνεται με row ids (αναγνωριστικά γραμμών). Συγκεκριμένα, κάθε row μιας σχέσης έχει ένα μοναδικό τέτοιο αναγνωριστικό (απλώς ποια σειρά είναι). Όταν κάποιος operator χρειαστεί δεδομένα από κάποια στήλη, χρησιμοποιεί τα row ids για να φέρει τις τιμές της στήλης που του αντιστοιχούν. Αυτό μπορεί να μειώσει σημαντικά την άσκοπη μετακίνηση δεδομένων, εφόσον πλέον μεταφέρεται μόνο η απαραίτητη πληροφορία για την επόμενη ζεύξη, βελτιώνοντας την απόδοση για ερωτήματα που διαχειρίζονται μικρό υποσύνολο στηλών από έναν μεγάλο πίνακα. Στη παρακάτω εικόνα βλέπουμε ένα join με late-materialization, το οποίο χρησιμοποιεί τη στήλη R.a και παράγει τα αποτελέσματά του χρησιμοποιώντας row ids. Π.χ. αν η επόμενη ζεύξη χρειάζεται τα στοιχεία της R.b θα κάνει materialize μια στήλη με τις τιμές της στις γραμμές με row id 1, 1, 3.



Ειδικά για τις στήλες που συμμετέχουν μόνο στο τελικό αποτέλεσμα δεν χρειάζεται να αποθηκεύσουμε τις τιμές τους προτού τελειώσουν όλες οι ζεύξεις. Κατα κύριο λόγο οι στήλες αποτελέσματος στα queries της εργασίας είναι VARCHAR που δεν συμμετέχουν ποτέ σαν στήλες κλειδιά για joins. Μπορούμε λοιπόν να αλλάξουμε την αναπαράσταση των strings σε μια μορφή rowid που θα ανακτούμε την τιμή τους μόνο στο τέλος γλυτώνοντας πολλές αντιγραφές δεδομένων που δεν χρειαζόμαστε τελικά.

Επομένως, το πρώτο σκέλος του παραδοτέου είναι να σχεδιάσετε μια καινούργια δομή αναπαράστασης των δεδομένων VARCHAR που θα δεικτοδοτεί το αρχικό column store το οποίο σας δίνεται στην execute. Συγκεκριμένα θα πρέπει να φτιάξετε μια δομή μέχρι 64bit η οποία κρατάει τιμές για τον πίνακα, τη στήλη, τη σελίδα και τη θέση του string στο columnar format που περιγράφει το readme χωρίς όμως να κρατάει τα πραγματικά δεδομένα του string (Αν μπορείτε να σκεφτείτε κάποια καλύτερη αναπαράσταση με λιγότερα πεδία ή με λιγότερο χώρο χρησιμοποιήστε την αρκεί να είναι σωστή). Στη συνέχεια, έχοντας αυτή τον νέο τρόπο αναφοράς (indexing), μπορούμε να ορίσουμε έναν νέο τύπο δεδομένων για το row store μας που θα είναι το value_t, το οποίο θα πρέπει να μπορεί να αναπαραστήσει ταυτόχρονα int32 αλλά και τη νέα αναπαράσταση για τα strings μας, αποφεύγοντας τη χρήση variant που έχει αυτή τη στιγμή ο τύπος Data. Δώστε μεγάλη προσοχή στον σωστό χειρισμό των null και των long strings.

Στη συνέχεια θα πρέπει να αλλάξετε την υλοποίηση των ScanNodes ώστε να φτιάχνουν ένα row store της μορφής `vector<vector<value_t>>` όπου οι τιμές των στηλών int32 θα γίνονται όλες materialize και τα strings θα αποθηκεύονται με την αναπαράσταση που είπαμε παραπάνω. Ο τρόπος με τον οποίο θα γίνονται οι ζεύξεις δεν πρέπει να αλλάξει πολύ μιας και οι στήλες στις οποίες γίνονται, έχουν ακόμα τις πραγματικές τιμές τους.

Παραγωγή τελικού αποτελέσματος

1. Για να παράξετε το τελικό αποτέλεσμα μπορείτε να μετατρέψετε τα τελικά tuples σε variant και να χρησιμοποιήσετε την έτοιμη συνάρτηση που δίνεται για μετατροπή από Data row-store σε ColumnarTable όπως κάνει αυτή τη στιγμή η execute. Αυτή η μέθοδος είναι αργή μιας και κάνει άσκοπα copies των strings αλλά είναι εύκολη η υλοποίηση και προτείνεται για να κάνετε το

αρχικό testing ότι το παραπάνω δουλεύει.

2. Για να ολοκληρωθεί αυτό το σκέλος της εργασίας θα πρέπει το top level join να παράγει απευθείας το ColumnarTable format το οποίο είναι και το format που πρέπει να επιστρέψει η execute. Κάτι τέτοιο απαιτεί διαφορετικό χειρισμό στην ρίζα και ίσως αρκετές νέες ενδιάμεσες δομές για να επιτευχθεί, οπότε ξεκινήστε την υλοποίηση του αφού είστε βέβαιοι ότι η αλλαγή σε value_t είναι σωστή.

Αφού υλοποιήσετε το ColumnarTable για το root join δεν θα πρέπει να απαιτείται πλέον κάποιος variant τύπος δεδομένων.

2. Ζεύξεις σε column store

Τα οφέλη της column store έναντι της row store οργάνωσης έχουν ήδη αναλυθεί παραπάνω. Ωστόσο, αυτή τη στιγμή η υπάρχουσα υλοποίηση χρησιμοποιεί row store για τα ενδιάμεσα αποτελέσματα. Μπορούμε αντί αυτού να κρατάμε τα ενδιάμεσα αποτελέσματα σε μια column store δομή, ώστε να εκμεταλλευτούμε το cache locality και να επιταχύνουμε την εκτέλεση των joins. Επομένως, για αυτό το σκέλος θα πρέπει να φτιάξετε μια σελιδοποιημένη δομή column store για να κρατάτε τα ενδιάμεσα αποτελέσματα των joins.

Το ζητούμενο για αυτό το ερώτημα είναι να αλλάξετε πάλι τα ScanNodes ώστε να παράγουν μια νέα αναπαράσταση για τα ενδιάμεσα αποτελέσματα, της μορφής `vector<column_t>`. Η νέα δομή column_t θα είναι μια σελιδοποιημένη δομή (βλ. παρακάτω) η οποία θα κρατάει σε value_t τις τιμές της αντίστοιχης στήλης τη μια δίπλα στην άλλη. Θα πρέπει, επίσης, να αλλάξετε και την υπάρχουσα υλοποίηση των hash joins ώστε να κάνει iterate αυτή την νέα δομή και να παράγει τα αποτελέσματα σε αυτή τη νέα δομή.

Αφού ολοκληρώσετε αυτό το ζητούμενο δεν θα πρέπει πλέον να υπάρχει κάποια row store δομή στην execute().

Σελιδοποιημένος χώρος αποθήκευσης ενδιάμεσων αποτελεσμάτων

Ο χώρος αποθήκευσης των δεδομένων στα ενδιάμεσα αποτελέσματα του κάθε join θα πρέπει να είναι μια συλλογή από φυσικά συνεχόμενους buffers (σελίδες), κάθε ένας από τους οποίους περιέχει πολλά στοιχεία. Με τον τρόπο αυτό ελαχιστοποιείται το κόστος της δέσμευσης μνήμης για την τοποθέτηση του κάθε αποτελέσματος μιας ζεύξης μέσα στο hot path του αλγορίθμου. Ακόμη, μια τέτοια δομή διευκολύνει την παραλληλοποίηση του probing, η οποία θα υλοποιηθεί σε επόμενο παραδοτέο, καθώς το κάθε thread θα μπορεί να τοποθετεί τα αποτελέσματα που παράγει σε δικές του σελίδες. Τα επιμέρους αποτελέσματα μπορούν στην συνέχεια εύκολα να συνδυαστούν, συλλέγοντας τις σελίδες που παρήγαγε το κάθε thread.

3. Επιλογή hashtable για hash joins

Τα hash joins είναι εξαιρετικά σημαντικά στην επεξεργασία των σχεσιακών δεδομένων και η απόδοση τους είναι ουσιώδης για τη γενική απόδοση ενός συστήματος βάσης δεδομένων. Εξαιτίας της δυσκολίας στην πρόβλεψη της φύσης των ενδιάμεσων αποτελεσμάτων, μία ιδανική υλοποίηση hash join πρέπει να είναι και γρήγορη για τυπικές επερωτήσεις (queries) όσο και ανθεκτική σε ασύνηθη κατανομή δεδομένων. Θα δούμε ένα απλό, αλλά αποτελεσματικό σχεδιασμό πίνακα κατακερματισμού χωρίς αλυσίδες (unchained) που λειτουργεί εξ ολοκλήρου στη μνήμη (in memory). Οι πίνακες κατακερματισμού χωρίς αλυσίδες συνδυάζουν τις τεχνικές της διαμέρισης της πλευράς που παράγει το hash table (build side partitioning), της διάταξης πίνακα γειτνίασης (adjacency array layout), σωληνωτές δοκιμές (pipelined probes), Bloom filters, και buffers write-combines στο λογισμικό για να επιτευχθεί σημαντική βελτίωση σε n:m joins, διατηρώντας υψηλή απόδοση σε 1:n joins.

Σαν 3ο και τελευταίο ζητούμενο του παραδοτέου θα πρέπει να υλοποιήσετε το unchained hashtable που περιγράφεται στο paper [1]. Θα πρέπει να φτιάχνεται ένα μόνο partition στο στάδιο κατασκευής του hashtable και σε επόμενα παραδοτέα θα εστιάσουμε παραπάνω στην παραλληλοποίηση του. Για hash function δοκιμάστε crc32 και fibonacci hashing ή βρείτε εσείς κάτι πιο γρήγορο και κρατήστε το καλύτερο. Για τα bloom filters μπορείτε να κάνετε precompute popcount για όλα τα δυνατά 16 bit hash prefixes. Το paper εξηγεί πολύ καλά τη δομή του παρ' όλα αυτά ακολουθεί μια δική μας περιγραφή για το τι πρέπει να υλοποιήσετε. Σε κάθε περίπτωση αμφιβολίας ακολουθήστε τι λέει το paper.

Unchained Hashtable

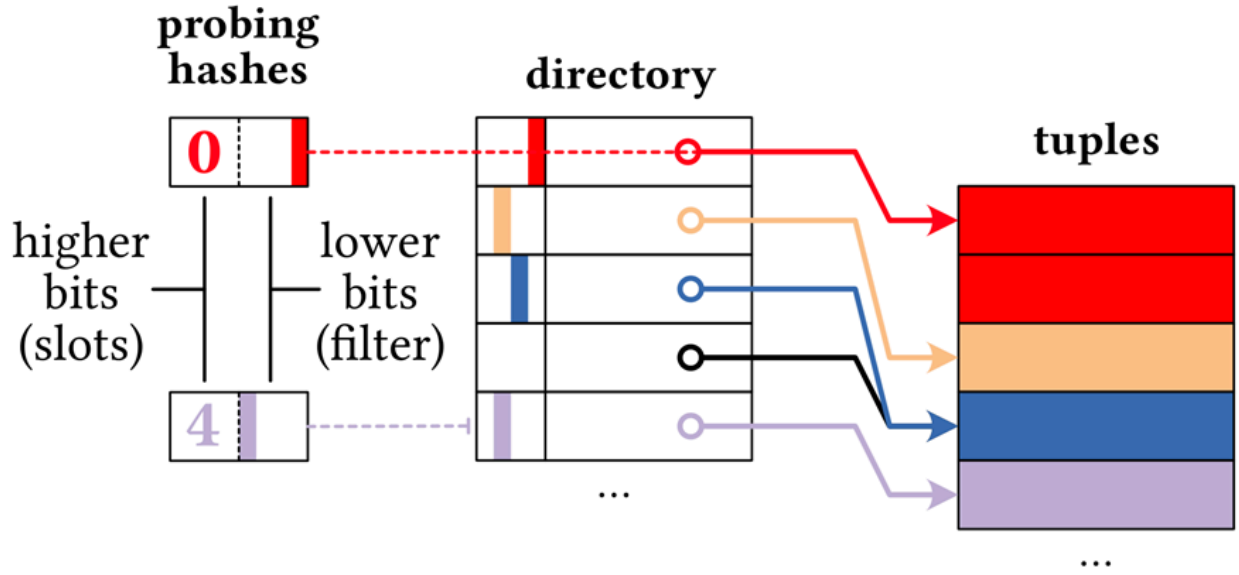
Διάταξη

Τα σχήματα ανοιχτής διευθυνσιοδότησης (open addressing) αποθηκεύουν όλες τις πλειάδες (tuples) σε έναν μόνο συνεχόμενο πίνακα, ο οποίος είναι αποτελεσματικός για την ανίχνευση με μη επιλεκτικά (non-selective) κατηγορήματα (predicates) και χωρίς διπλότυπα κλειδιά. Όταν η ζεύξη είναι επιλεκτική, τα σχήματα ανοιχτής διευθυνσιοδότησης δεν μπορούν να φιλτράρουν αποτελεσματικά τις μη ταιριαστές πλειάδες. Όταν υπάρχουν πλειάδες με διπλότυπα κλειδιά, αυτές οι πλειάδες συγκρούονται με άλλα κλειδιά, οδηγώντας σε υψηλό κόστος κατασκευής (building) και ανίχνευσης (probing). Αντίθετα, οι πίνακες κατακερματισμού με αλυσίδα (chained hashtables) δημιουργούν έναν κατάλογο (directory) που είναι ξεχωριστός από τις πλειάδες. Οι εγγραφές καταλόγου δείχνουν στις πλειάδες που μοιράζονται το ίδιο πρόθεμα κατακερματισμού, οι οποίες στη συνέχεια αλυσιδώνονται σε συνδεδεμένες λίστες, μειώνοντας τις συγκρούσεις λόγω διπλότυπων. Ωστόσο, η διέλευση από συνδεδεμένες λίστες είναι δαπανηρή, ειδικά για μεγάλες αλυσίδες. Για την αντιμετώπιση αυτών των ζητημάτων, θα υλοποιηθεί η διάταξη μη αλυσιδωτού πίνακα κατακερματισμού (unchained hashtable) που συνδυάζει τα οφέλη τόσο της ανοιχτής διευθυνσιοδότησης όσο και της αλυσιδωτής σύνδεσης.
Το Σχήμα 1 απεικονίζει τη βασική διάταξη του πίνακα κατακερματισμού μας με έναν κατάλογο και αποθήκευση πλειάδων. Ως εναλλακτική λύση στην αλυσιδωτή σύνδεση, χρησιμοποιούμε έναν πυκνό πίνακα γειτνίασης για την επίλυση συγκρούσεων: Οι πλειάδες αποθηκεύονται σε ένα συνεχόμενο (contiguous) buffer ταξινομημένο με βάση τα προθέματα κατακερματισμού τους. Με αυτόν τον τρόπο, οι πλειάδες αποθηκεύονται με την ίδια σειρά με τους δείκτες στον κατάλογο. Οι γειτονικές καταχωρήσεις

καταλόγου δίνουν το εύρος των πλειάδων με το ίδιο πρόθεμα κατακερματισμού και τώρα μπορούμε να εκτελέσουμε μια διαδοχική σάρωση για να προσπελάσουμε όλες αυτές τις πλειάδες.

- Σε σύγκριση με τους αλυσιδωτούς πίνακες κατακερματισμού, η αποσύνδεση εξαλείφει την δαπανηρή αναζήτηση δεικτών.
- Σε αντίθεση με τα ανοιχτά σχήματα διευθυνσιοδότησης, οι καταχωρήσεις στον κατάλογο αναφέρονται μόνο στους κάδους με τις τιμές και τα διπλότυπα δεν διαδίδονται σε γειτονικές καταχωρήσεις.

.



Σχήμα 1: Αναζήτηση στον πίνακα κατακερματισμού. Οι πλειάδες αποθηκεύονται σε ένα συνεχόμενο buffer, ταξινομημένο με βάση τα προθέματα κατακερματισμού τους. Ο κατάλογος (directory) αποτελείται από καταχωρήσεις που αντιπροσωπεύουν πλειάδες που μοιράζονται ένα πρόθεμα κατακερματισμού. Κάθε καταχώρηση στον κατάλογο περιέχει ένα μικροσκοπικό φίλτρο Bloom και έναν δείκτη προς το buffer που αποθηκεύεται το εύρος πλειάδων.. Κατά την διερεύνηση, το πρόθεμα κατακερματισμού χρησιμοποιείται για την εισαγωγή στον κατάλογο, το επίθεμα χρησιμοποιείται για τον έλεγχο του φίλτρου Bloom, και ο δείκτης ακολουθείται προς το εύρος των αντίστοιχων πλειάδων.

Δεδομένου ότι οι ζεύξεις είναι εξαιρετικά επιλεκτικές, μόνο ένα μικρό κλάσμα των πλειάδων στην πλευρά της διερεύνησης (probing side) επιλέγεται. Επομένως, είναι κρίσιμο να εξαλειφθούν οι πλειάδες που δεν μετέχουν στη ζεύξη νωρίς και αποτελεσματικά. Ενσωματώνουμε ένα φίλτρο Bloom μεγέθους καταχωρητή σε κάθε υποδοχή στον κατάλογο για να απορρίψουμε πιθανοτικά τις πλειάδες που σίγουρα δεν μετέχουν στη ζεύξη. Δεδομένου ότι τα τρέχοντα συστήματα χρησιμοποιούν μόνο τα χαμηλότερα 2^{48} bytes του χώρου διευθύνσεων, τα ανώτερα 16 bit των δεικτών δεν χρησιμοποιούνται και μπορούμε να χρησιμοποιήσουμε αυτά για να αποθηκεύσουμε το φίλτρο. Για αποτελεσματική πρόσβαση, αποθηκεύουμε τον δείκτη στα ανώτερα bit και το φίλτρο στα χαμηλότερα bit. Τα φίλτρα Bloom ελέγχονται πριν από την πρόσβαση στην αποθήκευση πλειάδων και μπορούν επίσης να ωθηθούν σε άλλους τελεστές ως μειωτές ημι-ζεύξεων (semi-joins).

Αποτελεσματικές διερευνήσεις (Efficient Probes)

Η διερεύνηση του πίνακα κατακερματισμού είναι συνήθως το πιο ακριβό μέρος της εκτέλεσης της ζεύξης. Η πλευρά της διερεύνησης μπορεί να είναι τάξεις μεγέθους μεγαλύτερη από την πλευρά της κατασκευής. Επομένως, η βελτιστοποίηση των αναζητήσεων στον πίνακα κατακερματισμού και η ελαχιστοποίηση της εργασίας ανά πλειάδα είναι ύψιστης σημασίας. Στη συνέχεια, περιγράφουμε τις αποτελεσματικές διερευνήσεις. Το Σχήμα 2 συνοψίζει τη λογική για την ανίχνευση πλειάδων στον πίνακα κατακερματισμού. Μπορούν να βελτιστοποιηθούν αρκετές πτυχές όπως συναρτήσεις κατακερματισμού, φίλτρα Bloom και κατάλληλη διάταξη του πίνακα κατακερματισμού για να επιτευχθεί αυτό. Ως αποτέλεσμα, το φιλτράρισμα πλειάδων που δεν επιλέγονται απαιτεί μόνο λίγες εντολές.

Λόγω της επιλεκτικότητας ζεύξης, η πιο γρήγορη διαδρομή κατά την επεξεργασία της ζεύξης είναι το φιλτράρισμα των πλειάδων που δεν μετέχουν στη ζεύξη. Για το φιλτράρισμα, χρησιμοποιούνται φίλτρα Bloom μεγέθους καταχωρητή ανά υποδοχή με 16 bit. Για να επιτευχθεί χαμηλό ποσοστό ψευδώς θετικών, ορίζονται 4 bit για κάθε πλειάδα στη θέση του καταλόγου, κάτι που επιτρέπει να απορρίπτονται πλειάδες διερεύνησης όπου οποιαδήποτε από τα αντίστοιχα bits στο φίλτρο δεν έχει οριστεί.

```
1  u64 shift; // Used to reduce a hash to a directory slot
2  u64 directory[1 << (64 - shift)];
3  void lookup(K key, u64 hash) {
4    u64 slot = hash >> shift;                    // shr
5    u64 entry = directory[slot];                 // mov
6    if (!couldContain((u16)entry, hash)) return; // Fig. 6
7    produceMatches(key, slot, entry);
8  }
9  void produceMatches(K key, u64 slot, u64 entry) {
10   T* start = directory[slot - 1] >> 16;
11   T* end = entry >> 16;
12   for (T* cur = start; cur != end; ++cur)
13     if (cur->key == key)
14       produce(cur);
15 }
```

Σχήμα 2: Λογική αναζήτησης για πλειάδες στον πίνακα κατακερματισμού: Η αναζήτηση στον κατάλογο μεταγλωττίζεται σε λίγες εντολές.

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 τελευταιο commit πριν την προθεσμία υποβολής στο git repository σας. **Η χρήση git είναι υποχρεωτική**.
Στο αρχείο README.md θα αναφέρονται τα εξής:

- Ονοματεπώνυμα 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
2. Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 54–65. https://www.vldb.org/conf/1999/P5.pdf