

Υλοποίηση Προοδευτικού Ευρετηρίου

Παΐλα Αγνή

Διπλωματική Εργασία

Επιβλέπων: Ν. Μαμουλής

Ιωάννινα, Φεβρουάριος, 2025



ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Περίληψη

Στην παρούσα διπλωματική εργασία, εξετάζεται και υλοποιείται η τεχνική **Database Cracking**, η οποία αποτελεί μια δυναμική μέθοδο οργάνωσης δεδομένων καθώς εκτελούνται ερωτήματα, χωρίς να απαιτείται προ επεξεργασία ή πλήρης ταξινόμησή τους.

Αρχικά, παρουσιάζονται οι βασικές αρχές του database cracking. Αναλύονται οι κύριοι αλγόριθμοι two-piece cracking και three-piece cracking οι οποίοι συνιστούν τον βασικό αλγόριθμο cracking. Επιπρόσθετα, παρουσιάζονται οι στοχαστικοί αλγόριθμοι cracking DDC, DDR, DD1C, DD1R και MDD1R, οι οποίοι εισάγουν τυχαιότητα στην όλη διαδικασία για να αντιμετωπίσουν προβληματικούς φόρτους εργασίας.

Στη συνέχεια, περιγράφεται η υλοποίηση ενός συστήματος που χρησιμοποιεί database cracking, με ανάλυση των δομών δεδομένων και των μεθόδων που επιτρέπουν τη προοδευτική ευρετηρίαση.

Τέλος, παρουσιάζονται πειραματικά αποτελέσματα που συγκρίνουν την απόδοση του βασικού database cracking είτε με άλλες τεχνικές, όπως το sorting και το sequential scanning, είτε με το στοχαστικό cracking. Τα αποτελέσματα δείχνουν ότι το βασικό cracking είναι ιδιαίτερα αποδοτικό σε τυχαίους φόρτους εργασίας, ενώ το στοχαστικό cracking, και ιδίως ο αλγόριθμος MDD1R, παρουσιάζει καλύτερη απόδοση σε διαδοχικούς φόρτους. Γενικότερα όλα τα αποτελέσματα υπογραμμίζουν τη σημασία του cracking ως μια ευέλικτη, αποδοτική και με χαμηλό κόστος λύση για τη διαχείριση μεγάλων όγκων δεδομένων.

Λέξεις Κλειδιά: Βάση δεδομένων, «σπάσιμο», ταξινόμηση, ερωτήματα εύρους, κόστος.

Abstract

This thesis examines and implements the **Database Cracking** technique, a dynamic method for organizing data as queries are executed, without requiring preprocessing or full sorting.

First, the fundamental principles of database cracking are presented. The core algorithms, two-piece cracking and three-piece cracking, which form the basis of the cracking method, are analyzed. In addition, stochastic cracking algorithms DDC, DDR, DD1C, DD1R, and MDD1R are introduced. These algorithms incorporate randomness to handle challenging workloads more effectively.

Subsequently, the implementation of a system utilizing database cracking is described, including an analysis of data structures and methods that enable progressive indexing.

Finally, experimental results compare the performance of basic database cracking with other techniques such as sorting and sequential scanning, as well as with stochastic cracking methods. The findings indicate that standard cracking performs exceptionally well for random workloads, whereas stochastic cracking, particularly the MDD1R algorithm, achieves better performance on sequential workloads. Overall, the results highlight the importance of database cracking as a flexible, efficient, and low-cost solution for managing large-scale data.

Keywords: Database, cracking, sorting, range queries, cost.

Περιεχόμενα

Κεφάλαιο 1. Εισαγωγή.....	1
1.1 Αντικείμενο της Διπλωματικής.....	1
1.2 Οργάνωση του Τόμου.....	2
Κεφάλαιο 2. Θεωρητική Προσέγγιση του Cracking	3
2.1 Σχετική Βιβλιογραφία	3
2.2 Βασικό Cracking	4
2.2.1 Σύστημα Βάσης Δεδομένων Προσανατολισμένο σε Στήλες	4
2.2.2 Σύστημα MonetDB	4
2.2.3 Αρχιτεκτονική Cracking	5
2.2.4 Ευρετήρια: Cracking ή Sorting	7
2.2.5 Συμπεράσματα.....	9
2.3 Στοχαστικό Cracking.....	9
2.4 Σύνοψη	10
Κεφάλαιο 3. Αλγόριθμοι Cracking.....	11
3.1 Αλγόριθμοι Βασικού Cracking.....	11
3.1.1 Θεωρητική Διαφοροποίηση των Αλγορίθμων.....	11
3.1.2 Κόστος Αλγορίθμων.....	14
3.1.3 Παράδειγμα Επιλογής Αλγορίθμου.....	16
3.1.4 Παράδειγμα Εκτέλεσης Αλγορίθμου	17
3.1.5 Καλύτερη/Χειρότερη Περίπτωση Cracking	18
3.2 Αλγόριθμοι Στοχαστικού Cracking.....	19
3.2.1 Αλγόριθμος DDC (Data Driven Center).....	20
3.2.2 Αλγόριθμος DDR (Data Driven Random)	21
3.2.3 Αλγόριθμος DD1C (Data Driven 1 Center).....	22
3.2.4 Αλγόριθμος DD1R (Data Driven 1 Random).....	22
3.2.5 Αλγόριθμος MDD1R (Materialization Data Driven 1 Random)	22
3.2.6 Παράδειγμα Στοχαστικών Αλγορίθμων	24
3.3 Συμπεράσματα.....	26
Κεφάλαιο 4. Σχεδίαση και Ανάλυση του Συστήματος.....	29

4.1	Το Περιβάλλον.....	29
4.2	Εκτέλεση Προγράμματος.....	29
4.3	Τα Δεδομένα	30
4.3.1	Δεδομένα Πειραμάτων.....	30
4.3.2	Έξοδος Προγράμματος	31
4.4	Τα Ερωτήματα.....	31
4.5	Ανάλυση Διαδικασίας Cracking.....	32
4.6	Η Συνάρτηση Crack.....	34
4.6.1	Συνοπτική Επεξήγηση Συνάρτησης Crack.....	35
4.7	Σύνοψη	36
Κεφάλαιο 5. Πειραματική Αξιολόγηση		37
5.1	Αναλυτική Παρουσίαση Αποτελεσμάτων.....	37
5.1.1	Αναλυτικά Αποτελέσματα Πειράματος	37
5.2	Αποτελέσματα.....	40
5.2.1	Μελέτη Ταξινόμησης	40
5.2.2	Σωστή Εκτέλεση Ταξινόμησης.....	41
5.2.3	Μελέτη Χρόνου Εκτέλεσης Crack για Τυχαίο Φόρτο Εργασίας.....	42
5.2.4	Μελέτη Απόδοσης των Αλγορίθμων Crack, Sort και Scan.....	43
5.2.5	Μελέτη Απόδοσης των Στοχαστικών Αλγορίθμων για Τυχαίο Φόρτο Εργασίας	44
5.2.6	Μελέτη Χρόνου Εκτέλεσης Crack για Διαδοχικό Φόρτο Εργασίας.....	45
5.2.7	Μελέτη Απόδοσης των Στοχαστικών Αλγορίθμων για Διαδοχικό Φόρτο Εργασίας	46
Κεφάλαιο 6. Επίλογος		49
6.1	Σύνοψη και Συμπεράσματα.....	49
6.2	Μελλοντικές Επεκτάσεις.....	49
Κεφάλαιο 7. Βιβλιογραφία		51

Κεφάλαιο 1. Εισαγωγή

Το κεφάλαιο αυτό περιέχει μια σύντομη περιγραφή του αντικειμένου της διπλωματικής εργασίας, καθώς επίσης περιγράφει τις ενότητες που ακολουθούν.

1.1 Αντικείμενο της Διπλωματικής

Οι σύγχρονες βάσεις δεδομένων έχουν ανάγκη να εξελίσσονται, ώστε να μπορούν να ανταποκρίνονται όσο το δυνατόν καλύτερα στις ανάγκες των χρηστών. Τις περισσότερες φορές όταν τα αποθηκευμένα δεδομένα σε μια βάση είναι αριθμοί, υπάρχει η ανάγκη αυτοί να είναι οργανωμένοι έτσι ώστε να διευκολύνεται η επερώτηση τους. Ωστόσο, η πληροφορία που μπορούμε να συγκεντρώσουμε και να αποθηκεύσουμε σε μια βάση δεδομένων είναι τεράστια και για αυτό αξίζει να μελετάμε και να υλοποιούμε διαδικασίες οι οποίες μπορούν να μειώσουν το κόστος της οργάνωσης και κατ' επέκταση της εύρεσης πληροφορίας.

Στην παρούσα διπλωματική εργασία μελετάμε την οργάνωση των δεδομένων σε σχεσιακές βάσεις δεδομένων, μέσω μιας δυναμικής διαδικασίας ευρετηρίασης γνωστής ως «cracking». Μέχρι τώρα γνωρίζουμε πως στις βάσεις δεδομένων, τα ευρετήρια χρησιμοποιούνται για την ταχύτερη αναζήτηση των επιθυμητών εγγραφών. Ωστόσο, η συντήρησή τους έχει σημαντικό κόστος, καθώς απαιτούνται οι απαραίτητες ενημερώσεις κάθε φορά που τροποποιούνται τα δεδομένα της βάσης δεδομένων. Για τον λόγο αυτό, παρουσιάζουμε την εφαρμογή του cracking σε ένα πλήρες σχεσιακό σύστημα βάσεων δεδομένων, στο οποίο ενσωματώνεται αποτελεσματικά, χωρίς μεγάλη επιβάρυνση για την επεξεργασία των ερωτημάτων.

Η προσέγγιση του cracking στοχεύει στη σταδιακή και προοδευτική οργάνωση των δεδομένων κατά τη διάρκεια της εκτέλεσης των ερωτημάτων εύρους. Με αυτόν τον τρόπο, η βάση δεδομένων «σπάει» σε μικρότερα πιο διαχειρίσιμα κομμάτια, με βάση τα όρια του ερωτήματος, βελτιώνοντας σταδιακά την αποδοτικότητα των μελλοντικών αναζητήσεων. Το πλεονέκτημα αυτής της μεθόδου, είναι ότι προσαρμόζεται στις ανάγκες

των χρηστών, επιτρέποντας ταχύτερη πρόσβαση στα δεδομένα με ελάχιστο επιπλέον υπολογιστικό κόστος. Το cracking μπορεί να διερευνηθεί σε πολλούς τομείς έρευνας στις βάσεις δεδομένων, όπως στις κατακευκτικές και παράλληλες βάσεις δεδομένων και στις P2P βάσεις δεδομένων (Peer-to-Peer Databases), τεχνολογίες οι οποίες βοηθούν στην καλύτερη διαχείριση τεράστιων δεδομένων.

Τέλος, παρουσιάζουμε τα αποτελέσματα της υλοποίησής μας, τα οποία δείχνουν ότι το σύστημα μπορεί να αυτό-οργανώνεται προοδευτικά ανάλογα με τις ανάγκες των χρηστών (και τις αναζητήσεις που αυτοί επιλέγουν να κάνουν), προσφέροντας σαφή οφέλη στην απόδοση, ακόμα και όταν οι αναζητήσεις γίνονται σε διαφορετικά τμήματα των δεδομένων με τυχαίο ή μη τρόπο.

1.2 Οργάνωση του Τόμου

Η παρούσα διπλωματική εργασία αποτελείται από επτά κεφάλαια, καθένα από τα οποία αναλύεται στις επόμενες παραγράφους.

Στο δεύτερο κεφάλαιο, εξετάζεται η λειτουργία και η αρχιτεκτονική του database cracking, και δίνεται ένα παράδειγμα που να το αφορά. Παράλληλα, παρουσιάζεται η διαφοροποίησή του από την λειτουργία της ταξινόμησης. Επιπλέον, εισάγεται και η έννοια του στοχαστικού cracking ως μία καλύτερη προσέγγιση του database cracking.

Το τρίτο κεφάλαιο ξεκινά με την παράθεση των βασικών αλγορίθμων cracking που είναι χρήσιμοι για την κατανόηση του θέματος. Στη συνέχεια, περιγράφεται η προσέγγιση υλοποίησης του αλγορίθμου μέσω ενός βασικού παραδείγματος. Το κεφάλαιο τελειώνει με την παράθεση των στοχαστικών αλγορίθμων cracking.

Το τέταρτο κεφάλαιο επικεντρώνεται στην υλοποίηση του λογισμικού. Παρουσιάζονται τα αρχεία του κώδικα, καθώς και οι συναρτήσεις που διαδραματίζουν σημαντικό ρόλο για την λειτουργία του cracking. Επιπλέον, αναλύεται η μορφή των αρχείων εισόδου και η επιλογή ερωτημάτων.

Στο πέμπτο κεφάλαιο, παρουσιάζονται τα πειράματα που διεξήχθησαν, μαζί με τα αποτελέσματά τους. Επιπρόσθετα, γίνεται αναλυτική αξιολόγησή τους.

Το έκτο κεφάλαιο συνοψίζει τα βασικά συμπεράσματα της διπλωματικής εργασίας και αναφέρεται σε πιθανές μελλοντικές επεκτάσεις του λογισμικού, οι οποίες θα μπορούσαν να βελτιώσουν περαιτέρω την απόδοσή του.

Στο έβδομο και τελευταίο κεφάλαιο γίνεται αναφορά στην βιβλιογραφία, η οποία έχει χρησιμοποιηθεί για τη διεκπεραίωση της διπλωματικής αυτής εργασίας.

Κεφάλαιο 2. Θεωρητική Προσέγγιση του Cracking

2.1 Σχετική Βιβλιογραφία

Στην αναφορά [IdKM07] περιγράφεται η προσέγγιση του cracking η οποία βασίζεται στην υπόθεση ότι η συντήρηση των ευρετηρίων (indices) θα πρέπει να είναι προϊόν της επεξεργασίας ερωτημάτων. Κάθε ερώτημα ερμηνεύεται όχι μόνο ως αίτημα για ένα συγκεκριμένο σύνολο αποτελεσμάτων, αλλά και ως οδηγία για να «σπάσει» η φυσική αποθήκευση της βάσης δεδομένων σε μικρότερα κομμάτια.

Κάθε κομμάτι περιγράφεται από ένα ερώτημα, και όλα μαζί συναρμολογούνται σε ένα ευρετήριο cracker (cracker index) για να επιταχύνουν μελλοντικές αναζητήσεις. Μόνο τα τμήματα της βάσης που είχαν ενδιαφέρον στο παρελθόν είναι εύκολα προσβάσιμα, ενώ τα υπόλοιπα παραμένουν ανεξερεύνητα και μη ευρετηριασμένα μέχρι να τα χρειαστεί ένα ερώτημα. Ακόμη και σε ένα τεράστιο σύνολο δεδομένων, μόνο οι εγγραφές που απαντάνε στο ερώτημα αλλάζουν, οδηγώντας σε σημαντικά κέρδη απόδοσης. Αν το ενδιαφέρον μετατοπιστεί σε ένα διαφορετικό μέρος των δεδομένων, το ευρετήριο cracker προσαρμόζεται αυτόματα.

Η συνεχής αντίδραση στα αιτήματα των ερωτημάτων προσδίδει στην αρχιτεκτονική την ισχυρή ιδιότητα της αυτό-οργάνωσης. Το προοδευτικό ευρετήριο cracker index δημιουργείται δυναμικά κατά την εκτέλεση των ερωτημάτων και προσαρμόζεται κατάλληλα με βάση τα μεταβαλλόμενα ερωτήματα.

Στην αναφορά [HIKY12] περιγράφεται η προσέγγιση του stochastic cracking. Αφού το βασικό cracking βασίζεται στην τυχαιότητα του φόρτου εργασίας για να συγκλίνει σωστά, χρειαζόμαστε νέες μεθόδους cracking, οι οποίες να εισάγουν τυχαιότητα από μόνες τους, ώστε να ολοκληρώνεται επιτυχώς η διαδικασία ακόμη και με μη τυχαίο φόρτο εργασίας.

Οι νέες μέθοδοι εισάγουν στοχαστικά (randomized) στοιχεία στη διαδικασία του cracking. Πλέον, κάθε ερώτημα εξακολουθεί να χρησιμοποιείται ως ένδειξη για την αναδιοργάνωση των δεδομένων, αλλά με έναν πιο ευέλικτο τρόπο που επιτρέπει βήματα αναδιοργάνωσης που δεν υπαγορεύονται αυστηρά από το ίδιο το ερώτημα.

Η μελέτη των μεθόδων δείχνει ότι το στοχαστικό cracking διατηρεί τα πλεονεκτήματα του αρχικού cracking, ενώ επεκτείνει τα οφέλη του σε πιο ρεαλιστικό αλλά και μεγαλύτερο όγκο φόρτων εργασίας, στα οποία το αρχικό cracking αποτυγχάνει.

2.2 Βασικό Cracking

2.2.1 Σύστημα Βάσης Δεδομένων Προσανατολισμένο σε Στήλες

Ένα σύστημα βάσης δεδομένων προσανατολισμένο σε στήλες (column-oriented DBMS) είναι ένα είδος σχεσιακής βάσης δεδομένων που αποθηκεύει και διαχειρίζεται τα δεδομένα ανά στήλη, αντί ανά γραμμή, όπως γίνεται στις παραδοσιακές βάσεις δεδομένων γραμμών (row-oriented DBMS). Δηλαδή κάθε γνώρισμα μιας σχέσης αναπαρίσταται ως ξεχωριστή στήλη. Σε ένα τέτοιο σύστημα, αποθηκεύουμε στην μνήμη ή στον δίσκο, τα στοιχεία κάθε στήλης μαζί.

Το πλεονέκτημα μιας τέτοιας βάσης δεδομένων είναι η βελτιωμένη απόδοση όταν εκτελούμε ερωτήματα που φιλτράρουν ή υπολογίζουν σύνολα δεδομένων σε συγκεκριμένες στήλες, καθώς δεν χρειάζεται να φορτώσουμε όλες τις στήλες της βάσης παρά μόνο την απαιτούμενη. Οι βάσεις δεδομένων προσανατολισμένες σε στήλες είναι ιδανικές όταν δουλεύουμε με μεγάλους πίνακες πολλών εγγραφών (Big Data), όπως συμβαίνει στην παρούσα εργασία.

2.2.2 Σύστημα MonetDB

Το σύστημα MonetDB, μας βοηθάει να εισάγουμε το απαραίτητο υπόβαθρο για την επόμενη υπό-ενότητα. Εσκεμμένα περιγράφουμε την υλοποίηση του cracking πάνω στο συγκεκριμένο σύστημα, καθώς αποτελεί ένα σύστημα βάσης δεδομένων προσανατολισμένο σε στήλες, και μας βοηθάει να εκτελούμε ερωτήματα επιλογής (select) σε ένα γνώρισμα.

Έστω ότι έχουμε το παρακάτω ερώτημα:

SELECT R.c FROM R WHERE $5 \leq R.a \leq 10$ AND $9 \leq R.b \leq 20$

Αυτό το ερώτημα μεταφράζεται ως ακολούθως:

Ra1 := algebra.select(Ra, 5, 10);

Rb1 := algebra.select(Rb, 9, 20);

Ra2 := algebra.OIDintersect(Ra1, Rb1);

Rc1 := algebra.fetch(Rc, Ra2);

Χρησιμοποιούνται τρεις βασικές δυαδικές πράξεις σχεσιακής άλγεβρας:

- **algebra.select(b, low, high)**: Αναζητά όλα τα (oid, attr) ζεύγη στο b που ικανοποιούν την συνθήκη $low \leq attr \leq high$.
- **algebra.OIDintersect(r, s)**: Επιστρέφει όλα τα (oid, attr) ζεύγη από το r, όπου το r.oid υπάρχει στο s.oid, υλοποιώντας τη σύζευξη (AND) των συνθηκών επιλογής.
- **algebra.fetch(r, s)**: Επιστρέφει όλα τα (oid, attr) ζεύγη που βρίσκονται στο r στις θέσεις που καθορίζονται από το s.oid, πραγματοποιώντας προβολή (projection).

Το παραπάνω απλό παράδειγμα ερωτήματος θα χρησιμοποιηθεί και στη συνέχεια, για να δείξουμε πως το cracking το επηρεάζει και ποιες τροποποιήσεις χρειάζονται για τη σωστή και αποδοτική χρήση του.

2.2.3 Αρχιτεκτονική Cracking

Θα περιγράψουμε πώς υλοποιείται το cracking σε μια βάση δεδομένων προσανατολισμένη σε στήλες.

- Την πρώτη φορά που εκτελείται ένα ερώτημα εύρους (range query) σε ένα γνώρισμα A, η βάση δεδομένων cracker δημιουργεί ένα αντίγραφο της στήλης A. Αυτό το αντίγραφο ονομάζεται **στήλη cracker του A** και συμβολίζεται ως **A_{CRK}**.
- Η στήλη **A_{CRK}** αναδιοργανώνεται συνεχώς με βάση τα ερωτήματα που αφορούν το γνώρισμα A.

Ορισμός

Το **cracking** ενός γνωρίσματος **A**, με βάση ένα **ερώτημα q**, είναι η **φυσική αναδιοργάνωση** της στήλης **A_{CRK}** έτσι ώστε οι **τιμές του A που ικανοποιούν το q** να αποθηκεύονται σε έναν **συνεχή χώρο μνήμης**.

- Φυσική αναδιοργάνωση (physical reorganization) είναι η διαδικασία τροποποίησης της φυσικής διάταξης των δεδομένων στον αποθηκευτικό χώρο. Τα δεδομένα αλλάζουν πραγματικά θέση στον αποθηκευτικό χώρο (π.χ. στον δίσκο) και δεν γίνεται χρήση δεικτών ή views για να εμφανίζονται αναδιοργανωμένα μετά το cracking.
- Οι βάσεις δεδομένων που αποθηκεύουν δεδομένα σε στήλες επωφελούνται ιδιαίτερα από τη συνεχή αποθήκευση, επειδή οι πράξεις πάνω σε μια στήλη γίνονται πολύ πιο γρήγορα. Όταν τα δεδομένα είναι αποθηκευμένα σε συνεχή χώρο μνήμης επιτρέπεται ταχύτερη πρόσβαση σε αυτά, πιο αποδοτικές αναζητήσεις τους και καλύτερη αξιοποίηση της μνήμης cache.

Η δημιουργία αντιγράφου της στήλης και η εφαρμογή cracking σε αυτό είναι χρήσιμη, καθώς διατηρεί ανέπαφη την αρχική στήλη, επιτρέποντας γρήγορη ανακατασκευή των εγγραφών αξιοποιώντας τη σειρά εισαγωγής.

Στις στήλες cracking, η αρχική θέση των εγγραφών αλλοιώνεται λόγω της φυσικής αναδιοργάνωσης. Συνεπώς:

- Οι στήλες cracking χρησιμοποιούνται για γρήγορη επιλογή τιμών.
- Οι αρχικές στήλες χρησιμοποιούνται για αποδοτική προβολή εγγραφών.

2.2.3.1 Στήλη Cracker

Η στήλη cracker χωρίζεται συνεχώς σε όλο και περισσότερα κομμάτια καθώς καταφθάνουν νέα ερωτήματα. Επομένως, χρειάζεται να είμαστε ικανοί να εντοπίζουμε γρήγορα τα επιμέρους κομμάτια της στήλης που μας ενδιαφέρουν.

Για τον σκοπό αυτό, εισάγουμε για κάθε στήλη cracker c ένα ευρετήριο cracker (cracker index), το οποίο διατηρεί πληροφορίες σχετικά με τον τρόπο κατανομής των τιμών στη στήλη c .

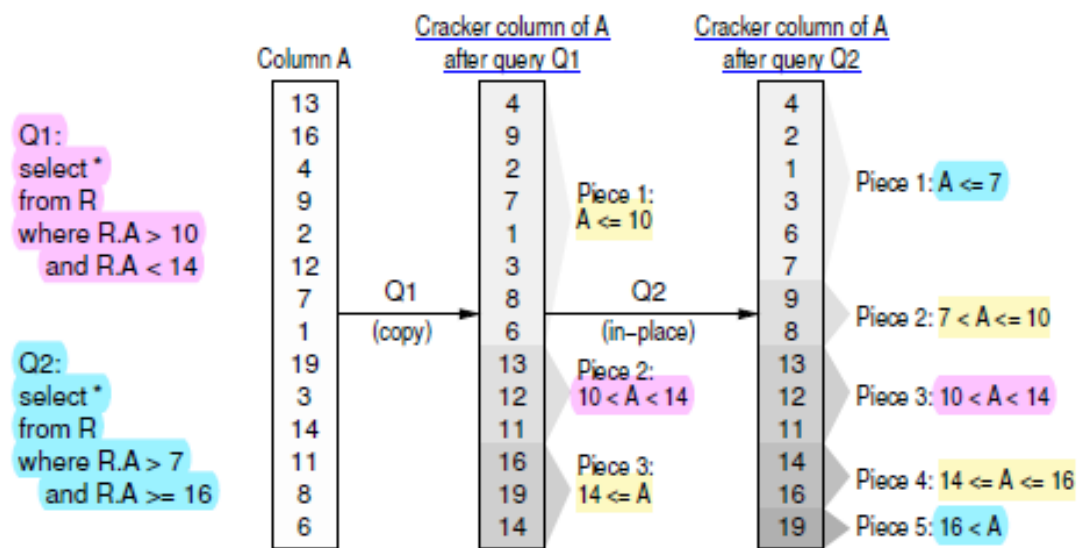
Στην παρούσα υλοποίηση, το ευρετήριο cracker είναι ένα AVL-δέντρο, (δυναμικό δέντρο αναζήτησης). Κάθε κόμβος στο δέντρο:

- Κρατά πληροφορίες για **μια τιμή v** .
- Αποθηκεύει **μια θέση p** , που αναφέρεται στη **στήλη cracker**, έτσι ώστε:
 - Όλες οι τιμές πριν από τη θέση p να είναι **μικρότερες από v** .
 - Όλες οι τιμές μετά από p να είναι **μεγαλύτερες από v** .
 - Επίσης, διατηρείται μια πρόσθετη πληροφορία που δείχνει αν η τιμή v συμπεριλαμβάνεται στα αριστερά ή δεξιά όρια.
- ▶ Τα AVL δέντρα είναι ένας τύπος δυαδικού δέντρου αναζήτησης (Binary Search Tree - BST) με κύριο χαρακτηριστικό ότι διατηρούν την ισορροπία τους μετά από κάθε εισαγωγή ή διαγραφή κόμβου. Ακριβώς επειδή διατηρούνται ισορροπημένα, η μέγιστη διαδρομή από τη ρίζα σε ένα φύλλο είναι λογαριθμική σε σχέση με το πλήθος των κόμβων, οπότε η αναζήτηση γίνεται σε χρόνο $O(\log n)$.

Η πληροφορία στο ευρετήριο cracker, μπορεί να χρησιμοποιηθεί για να επιταχύνει την απάντηση των μελλοντικών ερωτημάτων και να μειώσει το κόστος της αναζήτησης.

- **Αν ένα ερώτημα εύρους έχει ακριβή όρια γνωστές τιμές του ευρετηρίου, μπορεί να απαντηθεί μόνο με αναζήτηση στο ευρετήριο.**
- **Ακόμα και όταν δεν συμβαίνει αυτό, το ευρετήριο cracker περιορίζει τις τιμές της στήλης που πρέπει να αναλυθούν.**

2.2.3.2 Παράδειγμα Στήλης Cracker



Εικόνα 1: Παράδειγμα «σπασίματος» στήλης. Πηγή [ldKM07]

Έστω τα ακόλουθα ερωτήματα:

– Το ερώτημα **Q1** δημιουργεί τη στήλη **cracker** A_{CRK} , δηλαδή ένα αντίγραφο της αρχικής στήλης **A**. Οι εγγραφές ομαδοποιούνται σε **τρία κομμάτια**, τα οποία αποτελούν τα εύρη που προκύπτουν από το ερώτημα.

ο Το αποτέλεσμα του **Q1** λαμβάνεται **χωρίς επιπλέον κόστος**, απλώς ως μια **προβολή (view)** του κομματιού 2, χωρίς αντιγραφή δεδομένων.

– Αργότερα, το ερώτημα **Q2** εκμεταλλεύεται τις πληροφορίες στο ευρετήριο cracker. Μόνο τα **κομμάτια 1 και 3** χρειάζονται βελτίωση **για αυτό και «σπάνε» σε δύο νέα κομμάτια**. Το κομμάτι 2 παραμένει **ανέπαφο**.

ο Το αποτέλεσμα του **Q2** λαμβάνεται και πάλι με **μηδενικό κόστος**, καθώς αποτελείται από **τα συνεχόμενα κομμάτια 2, 3, και 4**.

– Ένα πιθανό **τρίτο ερώτημα** που έστω ότι ζητά την ανισότητα $A > 16$ **ταιριάζει ακριβώς** με το ήδη υπάρχων κομμάτι 5, και **δεν απαιτεί επιπλέον υπολογισμούς**.

2.2.4 Ευρετήρια: Cracking ή Sorting

Σε αυτή την υπό-ενότητα, θα προσπαθήσουμε να εξηγήσουμε γιατί το cracking αποτελεί εναλλακτική στρατηγική σε συγκεκριμένες συνθήκες περιβάλλοντος που δεν μπορούν να αντιμετωπιστούν αποτελεσματικά με την ταξινόμηση ή με μια παραδοσιακή στρατηγική βασισμένη σε ευρετήρια.

2.2.4.1 Ταξινόμηση

Η στρατηγική του cracking επιβάλλει προοδευτικά μια ταξινόμηση στη στήλη cracker. Δηλαδή τα κομμάτια μιας στήλης cracker είναι ταξινομημένα μεταξύ τους.

Κάθε **τιμή** σε ένα **κομμάτι** είναι:

- **μεγαλύτερη** από κάθε τιμή σε όλα τα **προηγούμενα κομμάτια**, και
- **μικρότερη** από κάθε τιμή σε όλα τα **επόμενα κομμάτια**.

♦ Θα μπορούσαμε να κάνουμε ταξινόμηση των δεδομένων εκ των προτέρων και στη συνέχεια να εκτελούσαμε δυαδική αναζήτηση με πολύ μικρό κόστος;

Αν γνωρίζουμε εξαρχής για ποια δεδομένα δείχνουν ενδιαφέρον οι χρήστες μέσω των ερωτημάτων, δηλαδή:

- Ποιο **γνώρισμα** (ή ποια γνωρίσματα) ζητείται πιο συχνά,
- Αν υπάρχει η πολυτέλεια **χρόνου και πόρων** για να δημιουργηθεί αυτή η φυσική ταξινόμηση των πλειάδων πριν έρθει το επόμενο ερώτημα, και
- Αν **δεν υπάρχουν ενημερώσεις** ή αν το χρονικό διάστημα μεταξύ μιας ενημέρωσης και ενός επόμενου ερωτήματος είναι **αρκετό** για τη διατήρηση της φυσικής ταξινόμησης (ή την διατήρηση των κατάλληλων ευρετηρίων που παρέχουν την ταξινόμηση, π.χ. B-trees),

τότε η **ταξινόμηση είναι ανώτερη στρατηγική**.

Το **cracking** **δεν προσπαθεί να ανταγωνιστεί** άλλες στρατηγικές υπό **τέτοιες ιδανικές συνθήκες**. Στοχεύει κυρίως σε περιβάλλοντα όπου:

- **Δεν υπάρχει γνώση** εκ των προτέρων για το **ποια δεδομένα** θα προκαλέσουν **ενδιαφέρον** (δηλαδή ποια γνωρίσματα και εύρη τιμών θα ζητηθούν και με ποια επιλεκτικότητα).
- **Δεν υπάρχει αρκετός χρόνος** για την αποκατάσταση ή τη συντήρηση της **φυσικής ταξινόμησης** μετά από μια **ενημέρωση**.
- Επιτρέπει τη διατήρηση ανεξάρτητων φυσικών ταξινομήσεων για κάθε γνώρισμα ξεχωριστά.

2.2.4.2 Στρατηγική Βασισμένη σε Ευρετήρια

Τα ίδια επιχειρήματα ισχύουν και ενάντια σε οποιαδήποτε στρατηγική βασισμένη σε ευρετήρια ή σε στρατηγικές που προσπαθούν να προετοιμάσουν τα δεδομένα με ομαδοποίηση (clustering) των ζητούμενων δεδομένων.

► Η ομαδοποίηση αποτελεί τεχνική αποθήκευσης δεδομένων μαζί ή σχετικά κοντά στον δίσκο ή στον χώρο της μνήμης, για να βελτιστοποιήσει την απόδοση των ερωτημάτων που αφορούν ίδιες περιοχές ενδιαφέροντος.

Για να λειτουργήσει μια τέτοια στρατηγική, απαιτείται:

- **Γνώση του φόρτου εργασίας** των ερωτημάτων (queries workload), και
- **Εκ των προτέρων έρευνα σχετική με την σπατάλη χρόνου και πόρων** που θα χρειαστούν για την κατάλληλη προετοιμασία των δεδομένων.

Επιπλέον, η στρατηγική ομαδοποίησης για να λειτουργήσει βασίζεται σε αυστηρά προκαθορισμένη στρατηγική ομαδοποίησης των δεδομένων, ενώ το cracking προσαρμόζεται δυναμικά με βάση τα εισερχόμενα ερωτήματα, χωρίς καμία προετοιμασία.

2.2.5 Συμπεράσματα

Μέσω του μηχανισμού cracking, κάθε ερώτημα μαθαίνει κάτι που μπορεί να χρησιμοποιηθεί για πιο γρήγορη απάντηση των μελλοντικών ερωτημάτων.

Το cracking είναι μια ελαφριά λειτουργία σε σύγκριση με την ταξινόμηση και δεν απαιτεί εκ των προτέρων γνώση για να επιτύχει γρήγορη πρόσβαση στα δεδομένα.

Όταν μια στρατηγική (πιθανόν ευρετηριοποίησης) **έχει γνώση των ερωτημάτων** -και συνεπώς και των δεδομένων που παρουσιάζουν τη μεγαλύτερη ζήτηση- και **οι πόροι της επαρκούν** για σωστή προετοιμασία και συντήρηση των δεδομένων, τότε αυτή είναι ανώτερη από το cracking και σωστά θα πρέπει να επιλέγεται για υλοποίηση.

Όταν όμως **αυτές οι συνθήκες δεν ισχύουν**, τότε το **cracking αποτελεί μια υποσχόμενη εναλλακτική λύση**.

Ωστόσο, όσο εξαιρετική και αν είναι η απόδοσή του σε περιπτώσεις με τυχαία ερωτήματα, με ένα φόρτο εργασίας πιο ρεαλιστικό (π.χ. διαδοχικά ερωτήματα), η απόδοσή του δεν ξεπερνά την απόδοση μια απλής εφαρμογής του αλγορίθμου Scan.

2.3 Στοχαστικό Cracking

Το βασικό cracking αντιμετωπίζει κάθε ερώτημα ως μια αυστηρή υπόδειξη για το πώς να αναδιοργανώσει τα δεδομένα, χωρίς να λαμβάνει υπόψη τη συνολική εικόνα. Ωστόσο, αυτός ο αυστηρός χαρακτήρας μπορεί επίσης να αποτελεί αδυναμία σε περιπτώσεις μη ιδανικού φόρτου εργασίας. Πιο συγκεκριμένα, σε αυτές τις περιπτώσεις, η αυστηρή τήρηση των ερωτημάτων και η αναδιοργάνωση του πίνακα ώστε να συγκεντρωθεί μόνο το αποτέλεσμα του εκάστοτε ερωτήματος, έχει ως συνέπεια η αναδιοργάνωση να

συμβαίνει ξανά και ξανά για νέα ερωτήματα, χωρίς συνολική βελτίωση στη δομή των δεδομένων.

Για να αντιμετωπίσουμε αυτό το πρόβλημα, εγκαταλείπουμε την αυστηρή απαίτηση του cracking και αντί αυτού, επιβάλλουμε ενέργειες αναδιοργάνωσης που δεν καθορίζονται αυστηρά από τις απαιτήσεις του κάθε ερωτήματος, αλλά που εξακολουθούν να είναι επωφελείς για το σύνολο του φόρτου εργασίας. Έτσι πλέον οι ενέργειες αναδιοργάνωσης καθορίζονται εν μέρει από τις ανάγκες των ερωτημάτων και εν μέρει είναι τυχαίες. Ονομάζουμε το cracking αυτό στοχαστικό (stochastic cracking), ώστε να υποδηλώνεται ο τυχαίος χαρακτήρας ορισμένων αναδιοργανώσεων.

2.4 Σύνοψη

Σε αυτό το κεφάλαιο έγινε μια σύντομη αναφορά στην μέθοδο και την αρχιτεκτονική του cracking, με χρήση ενός απλού παραδείγματος.

Σκοπός ήταν να δοθεί μια σφαιρική γνώση γύρω από την έννοια αυτή και την λειτουργία της, για την αποδοτική οργάνωση των δεδομένων κατά την εκτέλεση ερωτημάτων.

Επιπλέον, συγκρίθηκε το database cracking με την τεχνική του sorting, με σκοπό να αναδείξει τη δυνατότητά του να προσαρμόζεται δυναμικά στις ανάγκες του συστήματος, σε ορισμένες περιπτώσεις, ιδιαίτερα όταν η κατανομή των ερωτημάτων είναι απρόβλεπτη.

Ωστόσο σε περιπτώσεις όπου ο φόρτος εργασίας είναι πιο ρεαλιστικός άρα και πιο προβληματικός το cracking δεν παρουσιάζει αυξημένη απόδοση. Για τον λόγο αυτό παρουσιάσαμε την έννοια του στοχαστικού cracking το οποίο μπορεί να εφαρμοστεί αποδοτικά σε οποιονδήποτε φόρτο εργασίας, διατηρώντας το χαμηλό κόστος και την προσαρμοστικότητα που χαρακτηρίζει το απλό cracking.

Κεφάλαιο 3. Αλγόριθμοι Cracking

Σε αυτήν την ενότητα, παρουσιάζουμε τους αλγόριθμους που πραγματοποιούν τη φυσική αναδιοργάνωση μιας στήλης. Η **φυσική αναδιοργάνωση** ή **cracking** είναι μια διαδικασία που εκτελείται είτε σε ολόκληρη τη στήλη είτε σε ένα κομμάτι της. Επιπλέον, παρουσιάζουμε τους στοχαστικούς αλγόριθμους cracking, που προτείνονται ως μια πιο αποτελεσματική αντικατάσταση των αρχικών αλγορίθμων.

3.1 Αλγόριθμοι Βασικού Cracking

3.1.1 Θεωρητική Διαφοροποίηση των Αλγορίθμων

Υπάρχουν δύο βασικές λειτουργίες cracking, οι οποίες ονομάζονται **two-piece cracking** και **three-piece cracking**, αντίστοιχα.

Και οι δύο έχουν ως αποτέλεσμα τη φυσική αναδιοργάνωση μιας στήλης ενός γνωρίσματος A , λαμβάνοντας υπόψη έναν περιορισμό εύρους, ώστε όλες οι τιμές του A που ικανοποιούν τον περιορισμό να τοποθετούνται σε ένα συνεχόμενο χώρο. Και οι δύο αλγόριθμοι διασπάνε τη δεδομένη στήλη σε δύο τμήματα, με τη διαφορά ότι:

- Το **two-piece cracking** χρησιμοποιεί περιορισμούς μονόπλευρης σύγκρισης, της μορφής $A \leq med$, ενώ
- Το **three-piece cracking** χρησιμοποιεί περιορισμούς διπλής σύγκρισης, της μορφής $low \leq A \leq high$.

Το three-piece cracking είναι ισοδύναμο με δύο διαδοχικές εκτελέσεις του two-piece cracking. Δηλαδή, η διπλή σύγκριση της μορφής: $low \leq A \leq high$, ισοδυναμεί με τις δύο μονόπλευρες συγκρίσεις: $low \leq A$ και $A \leq high$.

3.1.1.1 Αλγόριθμος two-piece cracking

Ο αλγόριθμος αυτός αναδιοργανώνει φυσικά ένα τμήμα της στήλης c μεταξύ των θέσεων **posL** και **posH**, έτσι ώστε όλες οι τιμές μικρότερες από **med** να βρίσκονται σε συνεχόμενο χώρο. Η μεταβλητή **inc** καθορίζει αν το **med** θα συμπεριλαμβάνεται ή όχι στη σύγκριση.

— Αν **inc = false**, τότε ο τελεστής **θ1** είναι το "<", και ο τελεστής **θ2** είναι το ">=".

Ο αλγόριθμος χρησιμοποιεί δύο δείκτες (**x1** και **x2**) για να σαρώσει τη στήλη και να ανταλλάξει τιμές όπου χρειάζεται.

CrackInTwo(*c*, *posL*, *posH*, *med*, *inc*)

```
1:  x1 = δείκτης στη θέση posL
2:  x2 = δείκτης στη θέση posH
3:  Όσο θέση(x1) < θέση(x2) επανάλαβε
4:      Αν τιμή(x1)  $\geq$  med τότε
5:          x1 = δείκτης στην επόμενη θέση
6:      Αλλιώς
7:          Όσο τιμή(x2)  $\leq$  med ΚΑΙ θέση(x2) > θέση(x1) επανάλαβε
8:              x2 = δείκτης στην προηγούμενη θέση
9:      Τέλος_Όσο
10:  Αντάλλαξε(x1, x2)
11:  x1 = δείκτης στην επόμενη θέση
12:  x2 = δείκτης στην προηγούμενη θέση
13:  Τέλος_Αν
14: Τέλος_Όσο
```

Αλγόριθμος 1: Αλγόριθμος CrackInTwo. Πηγή [\[IdKM07\]](#)

Εξήγηση της λειτουργίας του αλγόριθμου 1

- Ο *x1* αρχικοποιείται στην αριστερή θέση (*posL*) και ο *x2* στη δεξιά θέση (*posH*).
- Ο αλγόριθμος εκτελεί μια σάρωση από τα δύο άκρα του πίνακα προς το κέντρο.
- Αν η τιμή στη θέση *x1* είναι μικρότερη από *med*, ο δείκτης *x1* μετακινείται προς τα δεξιά.
- Αν η τιμή στη θέση *x1* είναι μεγαλύτερη από *med*, τότε:
 - Ο *x2* μετακινείται προς τα αριστερά μέχρι να βρεθεί μια τιμή που πρέπει να ανταλλαχθεί.
 - Ανταλλάσσονται οι τιμές στις θέσεις *x1* και *x2*.
 - Και οι δύο δείκτες (*x1* και *x2*) μετακινούνται αντίστοιχα για να συνεχιστεί η επεξεργασία.
- Ο αλγόριθμος τελειώνει όταν οι δείκτες *x1* και *x2* συναντηθούν ή διασταυρωθούν.

3.1.1.2 Αλγόριθμος three-piece cracking

Ο αλγόριθμος **CrackInThree** αναδιοργανώνει φυσικά ένα τμήμα της στήλης *c* μεταξύ των θέσεων *posL* και *posH*, έτσι ώστε όλες οι τιμές που ανήκουν στο διάστημα *[low*,

high] να βρίσκονται σε συνεχόμενο χώρο. Οι μεταβλητές **incl** και **incH** καθορίζουν αν τα όρια **low** και **high** συμπεριλαμβάνονται ή όχι στην σύγκριση.

— Αν **incl = false** και **incH = false**, τότε ο τελεστής **θ1** είναι το "**>=**", ο τελεστής **θ2** είναι το "**>**" και ο τελεστής **θ3** είναι το "**<**".

Ο αλγόριθμος χρησιμοποιεί **τρεις δείκτες (x1, x2, x3)** για να σαρώσει και να ταξινομήσει τα δεδομένα στη στήλη.

CrackInThree(c, posL, posH, low, high, incl, incH)

```
1:  x1 = δείκτης στη θέση posL
2:  x2 = δείκτης στη θέση posH
3:  Όσο τιμή(x2) θ1 high ΚΑΙ θέση(x2) > θέση(x1) επανάλαβε
4:      x2 = δείκτης στην προηγούμενη θέση
5:  Τέλος_Όσο
6:  x3 = x2
7:  Όσο τιμή(x3) θ2 low ΚΑΙ θέση(x3) > θέση(x1) επανάλαβε
8:      Αν τιμή(x3) θ1 high τότε
9:          Αντάλλαξε(x2, x3)
10:         x2 = δείκτης στην προηγούμενη θέση
11:     Τέλος_Αν
12:     x3 = δείκτης στην προηγούμενη θέση
13: Τέλος_Όσο
14: Όσο θέση(x1) ≤ θέση(x3) επανάλαβε
15:     Αν τιμή(x1) θ3 low τότε
16:         x1 = δείκτης στην επόμενη θέση
17:     Αλλιώς
18:         Αντάλλαξε(x1, x3)
19:     Όσο τιμή(x3) θ2 low ΚΑΙ θέση(x3) > θέση(x1) επανάλαβε
20:         Αν τιμή(x3) θ1 high τότε
21:             Αντάλλαξε(x2, x3)
22:             x2 = δείκτης στην προηγούμενη θέση
23:         Τέλος_Αν
24:         x3 = δείκτης στην προηγούμενη θέση
25:     Τέλος_Όσο
26: Τέλος_Αν
27: Τέλος_Όσο
```

Αλγόριθμος 2: Αλγόριθμος CrackInThree. Πηγή [IdKM07]

Εξήγηση της λειτουργίας του αλγόριθμου 2

- **Αρχικοποίηση των δεικτών:**
 - ▶ Ο $x1$ ξεκινά από το αριστερό όριο *posL*.
 - ▶ Ο $x2$ ξεκινά από το δεξιό όριο *posH* και μετακινείται προς τα αριστερά μέχρι να βρει μια τιμή που είναι μικρότερη από *high*.
 - ▶ Ο $x3$ αρχικοποιείται στην ίδια θέση με το $x2$.
- **Διαχωρισμός των δεδομένων:**
 - ▶ Ο $x3$ μετακινείται προς τα αριστερά και ανταλλάσσει τιμές με τον $x2$ όπου είναι απαραίτητο.
 - ▶ Ο $x1$ προχωρά και ανταλλάσσει τιμές με τον $x3$ εάν μια τιμή δεν είναι μικρότερη από *low*.
 - ▶ Επαναλαμβάνεται η ίδια διαδικασία για να μετακινηθούν οι κατάλληλες τιμές στις σωστές θέσεις.
- **Τελικό αποτέλεσμα:**
 - ▶ Οι τιμές $< low$ τοποθετούνται στο αριστερό τμήμα.
 - ▶ Οι τιμές στο διάστημα *[low, high]* τοποθετούνται στο κεντρικό τμήμα.
 - ▶ Οι τιμές $> high$ τοποθετούνται στο δεξιό τμήμα.

3.1.2 Κόστος Αλγορίθμων

Και οι δύο αλγόριθμοι επηρεάζουν όσο το δυνατόν λιγότερα δεδομένα γίνεται. Η βασική ιδέα είναι ότι, καθώς γίνεται σάρωση των εγγραφών μιας στήλης, ανιχνεύονται οι περιπτώσεις όπου δύο εγγραφές μπορούν να ανταλλάξουν θέσεις. Η αναδιοργάνωση γίνεται σταδιακά υλοποιώντας όσο το δυνατόν λιγότερες ανταλλαγές.

Ο αλγόριθμος *two-piece cracking* απαιτεί μόνο μία διέλευση (single pass) από τη στήλη για να βρει τις εγγραφές που ικανοποιούν το ερώτημα και να τις τοποθετήσει στο κατάλληλο τμήμα. Καθώς χρησιμοποιεί δύο δείκτες ($x1$ και $x2$, έναν προς τα αριστερά και έναν προς τα δεξιά), μπορεί να ανιχνεύει και να ανταλλάσσει δεδομένα σε $O(n)$ χρόνο, όπου n είναι οι εγγραφές της στήλης. Στην χειρότερη περίπτωση, θα χρειαστεί να διαβάσει κάθε στοιχείο της στήλης μία φορά και έτσι η πολυπλοκότητά του σε χρόνο να διαμορφωθεί σε $O(n)$. Ωστόσο εάν προϋπάρχουν οργανωμένα κομμάτια στη στήλη, αυτός ο χρόνος μειώνεται σημαντικά.

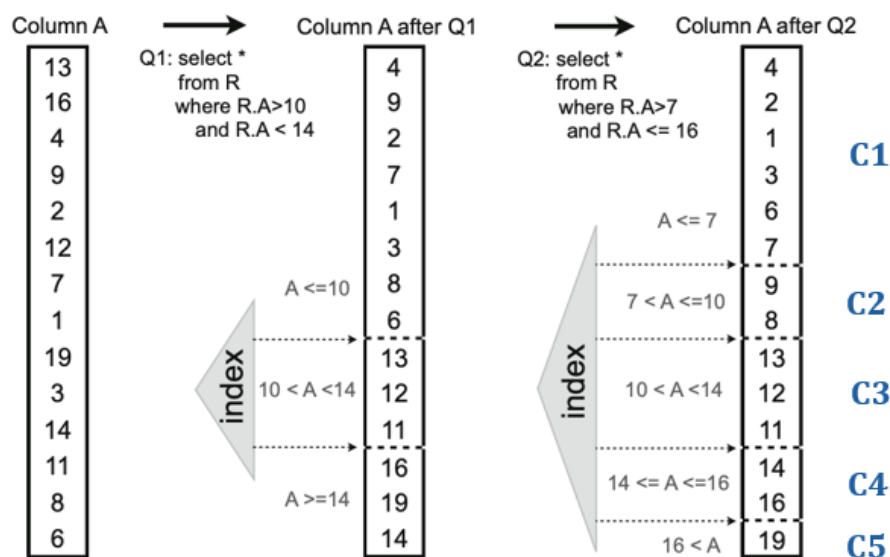
Ο αλγόριθμος *three-piece cracking* επεκτείνει τον αλγόριθμο *two-piece cracking* υλοποιώντας έναν επιπλέον διαχωρισμό των δεδομένων της στήλης. Καθώς χρησιμοποιεί τρεις δείκτες αντί για δύο, και επειδή περιλαμβάνει περισσότερα branches

(if και while), θεωρείται πιο δαπανηρός και η απόδοσή του συνήθως επηρεάζεται από την στήλη και το πόσα κομμάτια της είναι ανοργάνωτα. Η πολυπλοκότητά του είναι επίσης $O(n)$ στη χειρότερη περίπτωση, όμως στις περισσότερες περιπτώσεις είναι πιο αργός από τον αλγόριθμο *two-piece cracking*. Ο αλγόριθμος *three-piece cracking* αποτελεί πιο γρήγορη εναλλακτική του αλγορίθμου *two-piece cracking* για περιορισμούς διπλής σύγκρισης, διότι είναι μόνο ένας αλγόριθμος που απαντά και για τα δύο όρια της ερώτησης, αντί για δύο εκτελέσεις του αλγορίθμου *two-piece cracking*, μία για κάθε όριο. Καθώς η στήλη χωρίζεται σε περισσότερα κομμάτια, οι πιθανότητες να ζητηθεί ένα αποτέλεσμα που ανήκει σε μόνο ένα κομμάτι μειώνονται. Σε γενικές γραμμές, το ζητούμενο εύρος δεδομένων θα εκτείνεται σε πολλαπλά συνεχόμενα κομμάτια μιας στήλης cracker, οπότε μόνο το πρώτο και το τελευταίο κομμάτι πρέπει να αναδιοργανωθούν, χρησιμοποιώντας *two-piece cracking*. Για αυτόν τον λόγο παρατηρούμε πως **στην πράξη, ο *two-piece cracking* είναι ο πιο συχνά χρησιμοποιούμενος αλγόριθμος.** Το *three-piece cracking* χρησιμοποιείται **μόνο** όταν όλες οι εγγραφές που ικανοποιούν έναν περιορισμό επιλογής (select operator) ανήκουν **σε ένα μόνο** κομμάτι της στήλης cracker. Αυτό συμβαίνει **κυρίως στις πρώτες ερωτήσεις** που αναδιοργανώνουν μια στήλη.

Καθώς και στους δύο αλγορίθμους η αναδιοργάνωση γίνεται απευθείας πάνω στη στήλη χωρίς να δημιουργούνται νέα αντίγραφα, η διαφοροποίηση στην πολυπλοκότητα της μνήμης προέρχεται από τη χρήση του ενός επιπλέον δείκτη στον αλγόριθμο *three-piece cracking*, ο οποίος απαιτεί ελαφρώς περισσότερο χώρο για να τον αποθηκεύσει.

Γενικότερα, πριν καταλήξουμε σε αυτούς τους δύο απλούς αλγορίθμους, δοκιμάστηκαν διάφορες εναλλακτικές προσεγγίσεις. Ορισμένοι αλγόριθμοι που προσπάθησαν να μειώσουν τις ανταλλαγές δεδομένων ή να επιτρέψουν πρόωρο τερματισμό, αποδείχθηκαν πιο ακριβοί λόγω της πολυπλοκότητας του κώδικα.

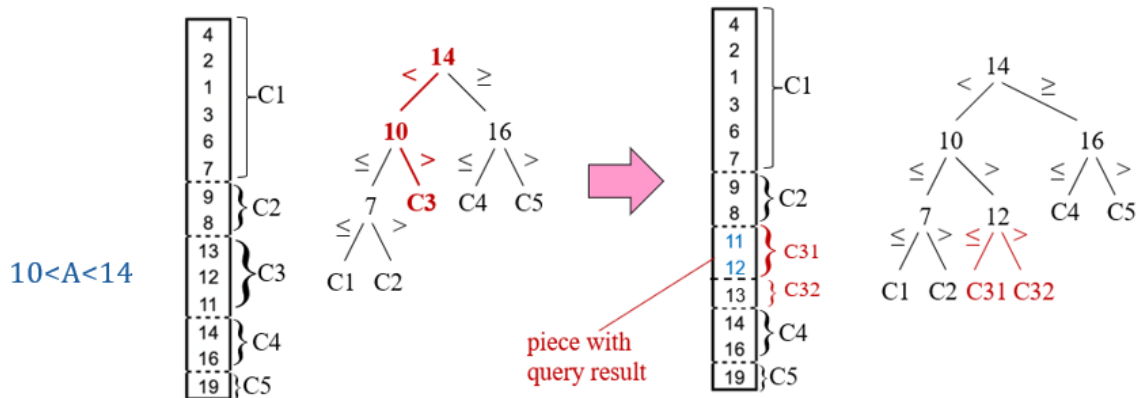
3.1.3 Παράδειγμα Επιλογής Αλγορίθμου



Εικόνα 2: Παράδειγμα «σπασίματος» στήλης. Πηγή [Mamo24]

Στο παράδειγμα της Εικόνας 2, το ερώτημα **Q1** χρησιμοποιεί έναν περιορισμό διπλής σύγκρισης. Επειδή είναι το **πρώτο ερώτημα** που εκτελεί **cracking** στη στήλη, δηλαδή το εύρος διαστήματος πέφτει εξ' ολοκλήρου σε ένα μη ταξινομημένο κομμάτι, χρησιμοποιείται το **three-piece cracking**. Ωστόσο, το δεύτερο ερώτημα **Q2**, που επίσης χρησιμοποιεί περιορισμό διπλής σύγκρισης, **δεν χρειάζεται να χρησιμοποιήσει three-piece cracking**, καθώς τα δεδομένα που απαντάνε στο **Q2** εκτείνονται στα **κομμάτια 1, 2 και 3**. Το κομμάτι 2 δεν χρειάζεται ανάλυση, καθώς είναι ήδη γνωστό ότι όλες οι εγγραφές εκεί πληρούν τις προϋποθέσεις του αποτελέσματος. Συνεπώς, **τα κομμάτια 1 και 3** αναδιοργανώνονται με **two-piece cracking**, ώστε να δημιουργηθούν δύο νέα υπό-κομμάτια από το κάθε ένα, έτσι ώστε το ένα να πληροί τις προϋποθέσεις του αποτελέσματος, και το άλλο να μην τις πληροί.

♦ Έστω ένα νέο ερώτημα το οποίο ζητάει όλες τις τιμές στο διάστημα [10,12]. Παρατηρούμε στην Εικόνα 3 ότι εκτελείται crack στο φύλλο **C3** του οποίου το εύρος επικαλύπτει το εύρος του νέου ερωτήματος ([10,12] ∈ C3). Ύστερα η συλλογή του αποτελέσματος γίνεται από το νέα κομμάτι που προέκυψε από το «σπάσιμο», εδώ δηλαδή το **C31**.



Εικόνα 3: Παράδειγμα ερωτήματος $10 < A \leq 12$. Πηγή [Mamo24]

3.1.4 Παράδειγμα Εκτέλεσης Αλγορίθμου

Έστω ότι στο παράδειγμα που θα μελετήσουμε έχουμε το ακόλουθο ευρετήριο (απεικονίζονται μόνο οι τιμές των κλειδιών):

{52, 37, 63, 14, 17, 8, 6, 25}

Έστω ότι το πρώτο ερώτημα εύρους είναι το ακόλουθο:

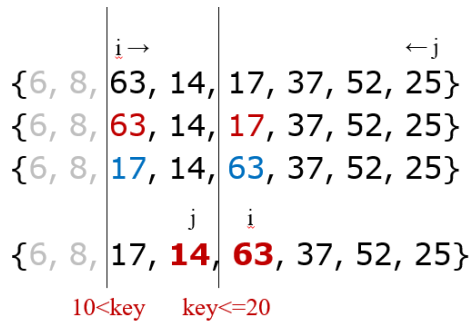
$10 < \text{key} \leq 20$

Αρχικά, χρησιμοποιούμε την πρώτη σύγκριση του παραπάνω ερωτήματος και «σπάμε», εφαρμόζουμε δηλαδή το crack, **με οδηγό το νούμερο 10**, όπως φαίνεται στην Εικόνα 4.

$i \rightarrow$ $\leftarrow j$
 {52, 37, 63, 14, 17, 8, 6, 25}
 {6, 37, 63, 14, 17, 8, 52, 25}
 {6, 37, 63, 14, 17, 8, 52, 25}
 {6, 8, 63, 14, 17, 37, 52, 25}
 {6, ^j8, ⁱ63, 14, 17, 37, 52, 25}
10 < key

Εικόνα 4: Διαδικασία για «σπάσιμο» στο νούμερο 10. Πηγή [Mamo24]

Έπειτα, χρησιμοποιούμε την δεύτερη σύγκριση του παραπάνω ερωτήματος και «σπάμε», εφαρμόζουμε δηλαδή το crack, **με οδηγό το νούμερο 20**, όπως φαίνεται στην Εικόνα 5. Να σημειωθεί ότι πλέον εφαρμόζουμε τον αλγόριθμο μόνο στο κομμάτι που μας ενδιαφέρει, χωρίς να επεξεργαστούμε καθόλου τα κλειδιά που εμφανίζονται να είναι πλέον γκριζαρισμένα στην Εικόνα 5, καθώς από την προηγούμενη αναδιοργάνωση γνωρίζουμε ότι σίγουρα σε αυτά δεν βρίσκεται πληροφορία η οποία να μας ενδιαφέρει.



Εικόνα 5: Διαδικασία για «σπάσιμο» στο νούμερο 20. Πηγή [\[Mamo24\]](#)

Μπορούμε να αναπαραστήσουμε πρόχειρα την προηγούμενη διαδικασία με μορφή δένδρου όπως φαίνεται για κάθε ξεχωριστό cracking στις Εικόνες 6 και 7.



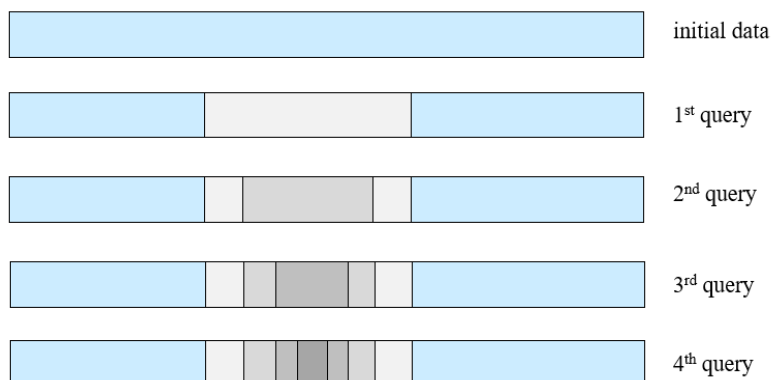
Εικόνα 6: Εφαρμογή 1^{ου} cracking. Πηγή [\[Mamo24\]](#)

Εικόνα 7: Εφαρμογή 2^{ου} cracking. Πηγή [\[Mamo24\]](#)

3.1.5 Καλύτερη/Χειρότερη Περίπτωση Cracking

3.1.5.1 Καλύτερη περίπτωση

Το καλύτερο σενάριο για τη λειτουργία του αλγορίθμου είναι όταν το αποτέλεσμα κάθε επερώτησης ανήκει στο αποτέλεσμα της προηγούμενης επερώτησης. Δηλαδή θέλουμε κάθε απάντηση να αποτελεί υπό-εύρος της προηγούμενης απάντησης, όπως φαίνεται και στην Εικόνα 8.

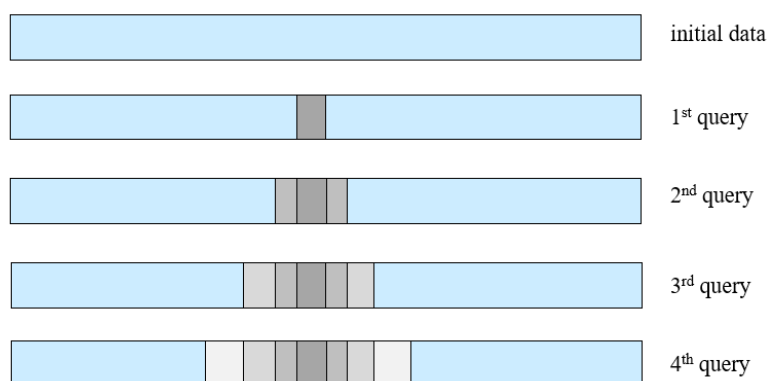


Εικόνα 8: Παράδειγμα καλύτερης περίπτωσης cracking. Πηγή [\[Mamo24\]](#)

Σε αυτό το σενάριο όλα τα κομμάτια που υπόκεινται σε «σπάσιμο» είναι μικρά, δηλαδή είναι μικρότερα ή ίσα σε μέγεθος με τα προηγούμενά τους.

3.1.5.2 Χειρότερη περίπτωση

Το χειρότερο σενάριο για τη λειτουργία του αλγορίθμου είναι όταν συμβαίνει το ακριβώς αντίθετο από αυτό που απεικονίζεται στην Εικόνα 8. Όταν δηλαδή το αποτέλεσμα κάθε επρώτησης αποτελείται από το αποτέλεσμα της προηγούμενης επρώτησης, με την προσθήκη επιπλέον δεδομένων. Σε αυτή την περίπτωση κάθε απάντηση αποτελεί υπέρ-εύρος της προηγούμενης απάντησης, όπως φαίνεται και στην Εικόνα 9.



Εικόνα 9: Παράδειγμα χειρότερης περίπτωσης cracking. Πηγή [\[Mamo24\]](#)

Σε αυτό το σενάριο τα μεγάλα κομμάτια υπόκεινται σε «σπάσιμο» πολλαπλές φορές.

3.1.5.3 Χειρότερη περίπτωση ||

Ένα επίσης κακό σενάριο για τη λειτουργία του αλγορίθμου είναι και αυτό που παρουσιάζεται στην Εικόνα 10. Όταν κάθε νιοστό ερώτημα ζητάει τη μικρότερη νιοστή αποθηκευμένη τιμή, έχουμε επίσης το ίδιο αρνητικό αποτέλεσμα με πριν, κατά το οποίο τα μεγάλα κομμάτια υπόκεινται σε «σπάσιμο» πολλαπλές φορές.



Εικόνα 10: Παράδειγμα χειρότερης περίπτωσης cracking. Πηγή [\[Mamo24\]](#)

3.2 Αλγόριθμοι Στοχαστικού Cracking

Το στοχαστικό cracking διατηρεί τις σχεδιαστικές αρχές του cracking για μια βάση δεδομένων στη μορφή στήλης. Όπως και στο βασικό cracking, αναδιοργανώνεται φυσικά

έναν πίνακα που αντιπροσωπεύει ένα μοναδικό γνώρισμα σε μια στήλη της βάσης δεδομένων, «σπάζοντας» το αντίγραφο της στήλης σε κομμάτια. Παράλληλα, μια δεντρική δομή διατηρεί πληροφορίες για το ποιο εύρος τιμών περιέχεται σε κάθε κομμάτι του πίνακα. Καθώς φτάνουν νέα ερωτήματα, εφαρμόζονται τα απαραίτητα cracking. Κάθε ερώτημα ζητάει ένα εύρος τιμών για ένα συγκεκριμένο γνώρισμα, και ο πίνακας αναδιοργανώνεται και συγκεντρώνει όλα τα στοιχεία που πληρούν τις προϋποθέσεις σε ένα συνεχόμενο κομμάτι του.

Η διαφοροποίηση του στοχαστικού cracking σε σχέση με το βασικό cracking είναι ότι αντί να βασιζόμαστε παθητικά στα όρια των ερωτημάτων για να εκτελέσουμε την αναδιοργάνωση, ασκούμε περισσότερο έλεγχο σε αυτήν. Δηλαδή, πριν εκτελέσουμε το cracking στα όρια του ερωτήματος, εφαρμόζουμε cracking στα δεδομένα με τυχαίο ή ημι-τυχαίο τρόπο, έτσι ώστε να πετύχουμε καλύτερη κατανομή των δεδομένων και γρηγορότερες επόμενες αναζητήσεις.

3.2.1 Αλγόριθμος DDC (Data Driven Center)

Ο αλγόριθμος Data Driven Center (DDC), λαμβάνει τις δικές του αποφάσεις λαμβάνοντας υπόψη του τα δεδομένα και τον χρησιμοποιούμε ως βάση για την ανάπτυξη των υπόλοιπων στοχαστικών παραλλαγών του. Στον DDC αλγόριθμο, κάθε αναδιοργάνωση θα πρέπει να χωρίζει το αντίστοιχο κομμάτι του πίνακα στη μέση. Ο DDC διχοτομεί αναδρομικά τα σχετικά κομμάτια του πίνακα καθώς κατευθύνεται προς τα ζητούμενα όρια του ερωτήματος, εισάγοντας αρκετά νέα κομμάτια με κάθε νέο ερώτημα.

Δεδομένου ενός ερωτήματος $[a, b]$, ο DDC διχοτομεί αναδρομικά το κομμάτι του πίνακα όπου βρίσκεται το $[a, b]$, έως ότου φτάσει σε ένα σημείο όπου το μέγεθος του κομματιού είναι επαρκώς μικρό. Στη συνέχεια, εκτελεί cracking σε αυτό το κομμάτι με βάση το $[a, b]$. Όπως και στο βασικό cracking, ένα αίτημα για $[a, b]$ σε μια ήδη κατακερματισμένη στήλη, γενικά θα οδηγήσει σε δύο crack: ένα για το $[a,)$ και ένα για το $[, b]$.

//Σπάει τον πίνακα C στα όρια a, b και δημιουργεί ένα υποσύνολο των δεδομένων.

Αλγόριθμος DDC(C, a, b)

```
1: positionLow = ddc_crack(C, a)
2: positionHigh = ddc_crack(C, b)
3: result = createView(C, positionLow, positionHigh)
```

//DDC cracking για μία τιμή v

Συνάρτηση ddc_crack(C, v)

```
4: Βρες το κομμάτι Piece που περιέχει την τιμή v
5: pLow = Piece.firstPosition()
```

```

6:  pHigh = Piece.lastPosition()
7:  Όσο (pHigh - pLow > CRACK_SIZE) επανάλαβε
8:      pMiddle = (pLow + pHigh) / 2
9:      Κάνε crack στη θέση pMiddle
10:     Αν (v < C[pMiddle]) τότε pHigh = pMiddle
11:     Αλλιώς pLow = pMiddle
12: position = crack στο διάστημα C[pLow, pHigh] με βάση το v
13: result = position

```

Αλγόριθμος 3: Αλγόριθμος DDC. Πηγή [HIKY12]

Εξήγηση της λειτουργίας της συνάρτησης ddc_crack(C, v)

- Εντοπίζει το κομμάτι που περιέχει το στοιχείο **v** και ορίζει τα όριά του.
- Όσο το κομμάτι είναι μεγαλύτερο από CRACK_SIZE, υπολογίζει το μέσο σημείο του κομματιού και «σπάει» σε αυτό το μέσο σημείο.
- Ελέγχει αν το **v** είναι μικρότερο ή μεγαλύτερο από το στοιχείο στη μέση του πίνακα και ανανεώνει κατάλληλα τα όρια.
- Όταν το μέγεθος του κομματιού είναι πλέον αρκετά μικρό, καλεί την μέθοδο crack για να βρεί την ακριβή θέση του **v** και να την επιστρέψει.

Εξήγηση της λειτουργίας του αλγόριθμου DDC(C, a, b)

- Εντοπίζει τη θέση positionLow όπου βρίσκεται η τιμή **a** στον πίνακα **C**.
- Εντοπίζει τη θέση positionHigh όπου βρίσκεται η τιμή **b** στον πίνακα **C**.
- Δημιουργεί μια όψη (view) του πίνακα **C** που περιλαμβάνει μόνο τα στοιχεία από positionLow έως positionHigh.

Ο αλγόριθμος DDC κάνει περισσότερα από το να ακολουθεί πιστά το αίτημα μιας ερώτησης «σπάζοντας» στα άκρα της, και ως αποτέλεσμα εισάγει επιπλέον κόστος. Οι υπόλοιποι στοχαστικοί αλγόριθμοί μας, προσπαθούν να βρουν την ισορροπία μεταξύ της πρόσθετης γνώσης που εισάγεται ανά ερώτηση και του κόστους που πληρώνουμε για αυτήν.

3.2.2 Αλγόριθμος DDR (Data Driven Random)

Ο αλγόριθμος Data Driven Random (DDR), εισάγει τυχαία στοιχεία στη λειτουργία του. Ο DDR διαφέρει από τον DDC, στο ότι δεν έχει την απαίτηση ένα κομμάτι να χωρίζεται ακριβώς στη μέση. Αντί για αυτό, χρησιμοποιεί τυχαία cracks, επιλέγοντας τυχαία σημεία διαχωρισμού μέχρι να φτάσει σε ένα σημείο όπου το μέγεθος του κομματιού είναι

επαρκώς μικρό. Όταν χωρίζει ένα κομμάτι σε δύο, προχωρά μόνο σε αυτό που περιέχει τα όρια του ερωτήματος [a, b].

Ο DDR δημιουργεί λιγότερο καλά τμήματα από τον DDC. Παρόλα αυτά, στην πράξη, καταβάλλει σημαντικά λιγότερη προσπάθεια για να απαντήσει σε μια ερώτηση, καθώς δεν χρειάζεται να βρει τις ακριβείς διαμέσους και συνεπώς έχει χαμηλότερο κόστος.

3.2.3 Αλγόριθμος DD1C (Data Driven 1 Center)

Ο αλγόριθμος Data Driven 1 Center (DD1C), αποτελεί μια παραλλαγή του αλγορίθμου DDC. Ενώ ο αλγόριθμος DDC εισάγει πληροφορίες ευρετηρίου που είναι χρήσιμες για τις επόμενες ερωτήσεις, η αναδρομική φυσική αναδιοργάνωση που χρησιμοποιεί, προκαλεί πολύ υψηλό κόστος στις πρώτες ερωτήσεις του φόρτου εργασίας. Για τον λόγο αυτό, ο αλγόριθμος DD1C αποφεύγει την αναδρομική φυσική αναδιοργάνωση. Ο αλγόριθμος εκτελεί το πολύ μία βοηθητική φυσική αναδιοργάνωση. Συγκεκριμένα, αφού κόψει ένα κομμάτι στη μέση, απλά σπάει το υπόλοιπο κομμάτι όπου βρίσκονται τα ζητούμενα όρια του ερωτήματος, ανεξάρτητα από το μέγεθός του.

3.2.4 Αλγόριθμος DD1R (Data Driven 1 Random)

Ο αλγόριθμος Data Driven 1 Random (DD1R), αποτελεί μια παραλλαγή του αλγορίθμου DDR. Ενώ ο αλγόριθμος DDR εισάγει πληροφορίες ευρετηρίου που είναι χρήσιμες για τις επόμενες ερωτήσεις, η αναδρομική φυσική αναδιοργάνωση που χρησιμοποιεί, προκαλεί πολύ υψηλό κόστος στις πρώτες ερωτήσεις του φόρτου εργασίας. Για τον λόγο αυτό, ο αλγόριθμος DD1R αποφεύγει την αναδρομική φυσική αναδιοργάνωση. Ο αλγόριθμος εκτελεί το πολύ μία βοηθητική φυσική αναδιοργάνωση. Συγκεκριμένα, εκτελεί μόνο μία τυχαία αναδιοργάνωση – crack και έπειτα σπάει το υπόλοιπο κομμάτι στα ζητούμενα όρια του ερωτήματος, ανεξάρτητα από το μέγεθός του.

3.2.5 Αλγόριθμος MDD1R (Materialization Data Driven 1 Random)

Ο αλγόριθμος Materialization Data Driven 1 Random (MDD1R), σχεδιάστηκε με σκοπό να μειώσει ακόμα περισσότερο το αρχικό κόστος, καθώς ακόμα και η μία μόνο επιπλέον αναδιοργάνωση των DD1C και DD1R μπορεί να αυξήσει το κόστος μιας μεμονωμένης ερώτησης, ειδικά όταν πρόκειται για την πρώτη ή τις πρώτες ερωτήσεις σε μια ακολουθία ερωτήσεων. Αυτό συμβαίνει διότι η διαδικασία του cracking στις αρχικές ερωτήσεις εκτελείται σε ολόκληρη τη στήλη, η οποία αρχικά είναι εντελώς ακατέργαστη.

Αυτός ο αλγόριθμος λειτουργεί παρόμοια με τον DD1R. Εκτελεί το ένα αρχικό τυχαίο crack και παραλείπει το crack που γίνεται με βάση τα όρια της ερώτησης. Αντί αυτού, υλοποιεί το αποτέλεσμα της ερώτησης σε έναν νέο πίνακα, όπως θα έκανε και ένας απλός τελεστής επιλογής (select). Για να διατηρήσουμε αποδοτικό τον αλγόριθμο σε ρεαλιστικό φόρτο εργασίας, εκτελούμε τυχαίο cracking (σε τυχαίο στοιχείο) συνδυαστικά με την υλοποίηση του νέου πίνακα. Για να επιστρέψουμε τα επιθυμητά στοιχεία, τα εντοπίζουμε και τα προσθέτουμε στο νέο πίνακα όσο εκτελούμε την διαδικασία του τυχαίου crack.

// Σπάει τον πίνακα C στα όρια a, b.

Algorithm MDD1R(C, a, b)

```

1:  Βρες το κομμάτι P1 που περιέχει τη τιμή a
2:  Βρες το κομμάτι P2 που περιέχει τη τιμή b
3:  Av (P1 == P2)
4:    result = split_and_materialize(P1,a,b)
5:  Αλλιώς
6:    res1 = split_and_materialize(P1,a,b)
7:    res2 = split_and_materialize(P2,a,b)
8:    view = createView(C, P1.lastPos+1, P2.firstPos-1)
9:    result = concat(res1, view, res2)

```

// Διαχωρίζει το κομμάτι Piece και αποθηκεύει τα δεδομένα που ικανοποιούν το ερώτημα.

function split_and_materialize(Piece, a, b)

```

10:  L = Piece.firstPosition
11:  R = Piece.lastPosition
12:  result = newArray()
13:  X = C[L + rand()%(R-L+1)]
14:  Όσο (L <= R)
15:    Όσο (L <= R and C[L] < X)
16:      Av (a <= C[L] && C[L] < b) result.Add(C[L])
17:      L = L + 1
18:    Όσο (L <= R and C[R] >= X)
19:      Av (a <= C[R] && C[R] < b) result.Add(C[R])
20:      R = R - 1
21:  Av (L < R) swap(C[L],C[R])

```

22: Πρόσθεσε crack στο X στην θέση L

Αλγόριθμος 4: Αλγόριθμος MDD1R. Πηγή [HIKY12]

Εξήγηση της λειτουργίας της συνάρτησης `split_and_materialize(Piece, a, b)`

- Ορίζει τα όρια **L** και **R** του κομματιού.
- Δημιουργεί έναν **νέο πίνακα** για την αποθήκευση των **επιθυμητών στοιχείων**.
- Επιλέγει μία τυχαία τιμή **X** από τις τιμές του κομματιού.
- Όσο $L \leq R$
 - ▶ Προχωράει προς τα **δεξιά**, ελέγχοντας αν οι **τιμές** είναι **μικρότερες** από **X**. Αν βρίσκονται στο διάστημα **[a, b)**, τις **αποθηκεύει**.
 - ▶ Προχωράει προς τα **αριστερά**, ελέγχοντας αν οι **τιμές** είναι **μεγαλύτερες** ή ίσες με **X**. Αν βρίσκονται στο διάστημα **[a, b)**, τις **αποθηκεύει**.
 - ▶ Αν $L < R$, ανταλλάσσει τις τιμές στις θέσεις **L** και **R**.
- Δημιουργεί **crack** στη θέση **L** χρησιμοποιώντας την τιμή **X**.

Εξήγηση της λειτουργίας του αλγόριθμου MDD1R(C, a, b)

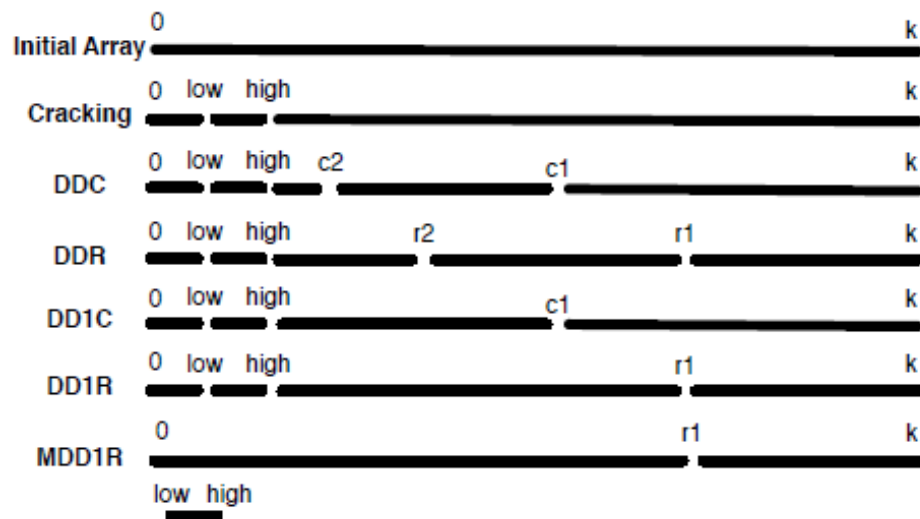
- Εντοπίζει τα κομμάτια **P1, P2** που **περιέχουν** τις τιμές **a, b** αντίστοιχα.
- Αν τα όρια **a** και **b** ανήκουν στο **ίδιο** κομμάτι (**P1 == P2**), εκτελεί την συνάρτηση `split_and_materialize` στο κομμάτι **P1**.
- Αν τα **P1** και **P2** είναι **διαφορετικά** κομμάτια
 - ▶ Εκτελεί την συνάρτηση `split_and_materialize` στο κομμάτι **P1**.
 - ▶ Εκτελεί την συνάρτηση `split_and_materialize` στο κομμάτι **P2**.
 - ▶ Δημιουργεί μια όψη (**view**) που περιλαμβάνει τα στοιχεία του πίνακα μεταξύ των **P1** και **P2**.
 - ▶ **Συγχωνεύει** τα αποτελέσματα των δύο διαδικασιών με την όψη, για να επιστραφεί το **τελικό αποτέλεσμα**.

Τέλος, αξίζει να αναφέρουμε ότι με τη χρήση του αλγόριθμου αυτού αποφεύγουμε ολοκληρωτικά τη δημιουργία νέου πίνακα όταν ένα ερώτημα ταιριάζει ακριβώς με ένα κομμάτι. Έτσι πετυχαίνουμε περαιτέρω μείωση του συνολικού κόστους του αλγορίθμου.

3.2.6 Παράδειγμα Στοχαστικών Αλγορίθμων

Στην Εικόνα 11 παρουσιάζεται σχηματικά η λειτουργία των αλγορίθμων cracking. Φαίνεται το τελικό αποτέλεσμα μιας απλοποιημένης αναδιοργάνωσης δεδομένων, τόσο με τους διάφορους στοχαστικούς αλγορίθμους cracking που παρουσιάσαμε, όσο και με

το αρχικό cracking. Έστω ότι ένας πίνακας, αρχικά άθικτος, λαμβάνει μία ερώτηση για το εύρος [low, high].



Εικόνα 11: Λειτουργία αλγορίθμων cracking. Πηγή [\[HIKY12\]](#)

► Το απλό cracking αναδιοργανώνει τον πίνακα αποκλειστικά με βάση το [low, high], δηλαδή ακριβώς όπως το ζητά το ερώτημα.

► Ο DDC πρώτα διχοτομεί τον πίνακα στη θέση c1, μετά στη c2 και τότε στο [low, high].

— c1: διάμεσος που χωρίζει ολόκληρο τον πίνακα σε δύο ίσα κομμάτια.

— c2: διάμεσος που χωρίζει το αριστερό κομμάτι σε δύο ίσα κομμάτια.

Στη συνέχεια, το κομμάτι [0, c2] διαπιστώνεται ότι είναι αρκετά μικρό και έτσι ο DDC σταματά να αναζητά διαμέσους και εκτελεί cracking στο κομμάτι με βάση τα όρια [low, high] του ερωτήματος.

► Ο DDR χωρίζει έναν πίνακα, επιλέγοντας αρχικά μία τυχαία θέση r1, στη συνέχεια χωρίζει το νέο αριστερό κομμάτι σε μία νέα θέση r2 και τέλος εφαρμόζει crack με βάση το εύρος τιμών [low, high] που ζητήθηκε.

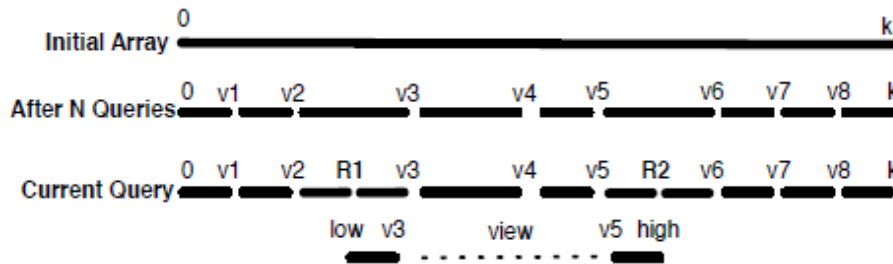
► Ο DD1C χωρίζει το πρώτο κομμάτι στη μέση δηλαδή στη θέση c1 και στη συνέχεια κάνει cracking στο [low, high].

► Ο DD1R χωρίζει το πρώτο κομμάτι με βάση μία τυχαία θέση r1 και στη συνέχεια κάνει cracking στο [low, high].

► Ο MDD1R εκτελεί το ίδιο τυχαίο cracking όπως ο DD1R, αλλά δεν εκτελεί το cracking που βασίζεται στα όρια [low, high] της ερώτησης. Αντί αυτού, υλοποιεί τα αποτελέσματα απευθείας.

♦ Για λόγους απλότητας, στο παράδειγμα τόσο το low όσο και το high πέφτουν στο ίδιο κομμάτι, και μόνο με δύο επαναλήψεις φτάνουμε σε αρκετά μικρό μέγεθος κομματιού.

♦ Σε γενικές γραμμές, ο DDC συνεχίζει να κόβει τα κομμάτια στη μέση, μέχρι να φτάσει στο ελάχιστο επιτρεπτό μέγεθος, και το αίτημα για [low, high] αξιολογείται ως δύο ξεχωριστά αιτήματα, ένα για κάθε όριο, καθώς κάθε ένα από αυτά μπορεί να βρίσκεται σε διαφορετικό κομμάτι.



Εικόνα 12: Λειτουργία αλγόριθμου MDD1R. Πηγή [HIKY12]

Η Εικόνα 12 δείχνει ένα πιο αναλυτικό παράδειγμα του αλγόριθμου MDD1R σε μια στήλη που έχει ήδη υποστεί cracking από προηγούμενες ερωτήσεις. Τα δύο όρια low, high του ερωτήματος πέφτουν σε δύο διαφορετικά κομμάτια μιας ήδη κατακερματισμένης στήλης. Ο MDD1R χειρίζεται αυτά τα δύο κομμάτια ξεχωριστά. Πρώτα επεξεργάζεται το αριστερότερο κομμάτι (v2, v3) και στην συνέχεια επεξεργάζεται το δεξιότερο κομμάτι (v5, v6), εκτελώντας ένα τυχαίο crack σε κάθε κομμάτι.

Επιπλέον, η υλοποίηση του νέου πίνακα δεν περιέχει εξ' ολοκλήρου όλα τα δεδομένα που επαληθεύουν το ερώτημα. Τα ενδιαμέσα κομμάτια που δεν έχουν υποστεί cracking επιστρέφονται ως όψη (view), ενώ μόνο τα επιλεγμένα στοιχεία από τα ακραία κομμάτια υλοποιούνται στον νέο πίνακα.

3.3 Συμπεράσματα

Σε αυτό το κεφάλαιο περιγράψαμε όλη τη θεωρία που είναι απαραίτητη για την υλοποίηση του αλγόριθμου cracking ώστε να πετύχουμε την προοδευτική ταξινόμηση. Επιπλέον, αναφερθήκαμε σε όλους τους βασικούς όρους της διαδικασίας της υλοποίησης.

Από την ανάλυση των αλγορίθμων two-piece cracking και three-piece cracking, προκύπτουν σημαντικά συμπεράσματα σχετικά με την αποδοτικότητα και τη συμπεριφορά του database cracking. Τα συμπεράσματα της θεωρητικής αυτής μελέτης, υπόσχονται ότι μετά από μεγάλο αριθμό τυχαίων ερωτημάτων, η στήλη που έχει αποθηκευμένα τα αρχικά μας δεδομένα, ταξινομείται πλήρως.

Ωστόσο, η απόδοση του cracking εξαρτάται από τη φύση του φόρτου εργασίας (workload). Όπως φάνηκε, στην καλύτερη περίπτωση, όπου κάθε νέο ερώτημα εξειδικεύει το αποτέλεσμα του προηγούμενου, το cracking είναι εξαιρετικά αποδοτικό.

Αντίθετα, στη χειρότερη περίπτωση, όπου τα ερωτήματα διευρύνουν το εύρος αναζήτησης ή είναι τυχαία κατανεμημένα, η απόδοσή του επιβαρύνεται λόγω πολλαπλών διασπάσεων των ίδιων περιοχών δεδομένων.

Ως γενική λύση εισάγαμε την έννοια του στοχαστικού cracking το οποίο είναι αποδοτικό ακόμη και σε προβληματικό φόρτο εργασίας, λόγω της τυχειότητας που το χαρακτηρίζει. Κάθε ένας αλγόριθμος στοχαστικού cracking, παρουσιάζει όλο και καλύτερη απόδοση από τον προηγούμενό του, καθώς χρησιμοποιεί όλο και περισσότερη τυχειότητα για την εφαρμογή των crack.

Πιο συγκεκριμένα, αναλύσαμε τον αλγόριθμο DDC ο οποίος ακολουθεί τη φιλοσοφία του cracking, αλλά εμπλουτίζεται με στοιχεία που βασίζονται στα δεδομένα, για να εξασφαλιστεί η ανθεκτικότητα στον φόρτο εργασίας. Επιπλέον αναλύσαμε τους αλγορίθμους DDR, DD1C, DD1R και τέλος MDD1R με τον τελευταίο να υπόσχεται την καλύτερη απόδοση από όλους.

Κεφάλαιο 4. Σχεδίαση και Ανάλυση του Συστήματος

Σε αυτό το κεφάλαιο θα πραγματευτούμε την ανάλυση και την σχεδίαση του συστήματος, καθώς και την υλοποίηση του βασικού αλγορίθμου cracking, ώστε να συμβαίνει η προοδευτική ταξινόμηση της βάσης δεδομένων μας, κάθε φορά που απαντάμε σε κάποια ερώτηση.

4.1 Το Περιβάλλον

Στην παρούσα διπλωματική εργασία εργαζόμαστε σε περιβάλλον με λειτουργικό σύστημα Ubuntu. Ο αλγόριθμος του cracking υλοποιείται σε γλώσσα προγραμματισμού C / C++.

4.2 Εκτέλεση Προγράμματος

Για να εκτελέσουμε έναν αλγόριθμο σε ένα σύνολο δεδομένων, πρέπει να εκτελέσουμε την εντολή της μορφής:

```
./run.sh [data] [algo] [nqueries] [workload] [selectivity] [update] [timelimit]
```

[data]

Πρέπει να είναι το όνομα του αρχείου δεδομένων που έχουμε δημιουργήσει και θέλουμε να χρησιμοποιήσουμε σε κάθε εκτέλεση.

[algo]

Πρέπει να είναι το όνομα του αλγορίθμου που θέλουμε να χρησιμοποιήσουμε σε κάθε εκτέλεση. Αυτό μπορεί να είναι crack, sort, scan, ddc, ddr, dd1c, dd1r ή mdd1r.

[nqueries]

Πρέπει να είναι ένας ακέραιος αριθμός που θα υποδηλώνει τον αριθμό των ερωτημάτων που θέλουμε να εκτελεστούν κάθε φορά.

[workload]

Πρέπει να υποδηλώνει το είδος του φόρτου εργασίας. Στην υλοποίησή μας μπορεί να είναι είτε Random δηλαδή τυχαίος, είτε SeqOver δηλαδή διαδοχικός.

[selectivity]

Πρέπει να είναι ένας πραγματικός αριθμός από τους τρεις ακόλουθους:

- 0.5 (50% επιλεκτικότητα)
- 1e-2 (1% επιλεκτικότητα)
- 1e-7 (0.00001% επιλεκτικότητα)

Συνήθως θέλουμε 1% επιλεκτικότητα, άρα χρησιμοποιούμε το 1e-2 .

[update]

Στη συγκεκριμένη υλοποίηση δεν κάνουμε χρήση καμίας ενημέρωσης, για αυτό χρησιμοποιούμε πάντα το όρισμα NOUP που σημαίνει ότι μπορούμε μόνο να διαβάζουμε τα ερωτήματα, χωρίς αυτά να αλλάζουν.

[timelimit]

Πρέπει να είναι ένας ακέραιος αριθμός που να υποδηλώνει τον μέγιστο χρόνο εκτέλεσης σε δευτερόλεπτα πριν τερματιστεί πρόωρα η εκτέλεση εάν ξεπεραστεί η τιμή του.

► Παράδειγμα εντολής εκτέλεσης αποτελεί το ακόλουθο:

```
./run.sh 100000000.data crack 100000 Random 1e-2 NOUP 30
```

4.3 Τα Δεδομένα

Στην συγκεκριμένη υλοποίηση ως βάση δεδομένων για την αποθήκευση των δεδομένων, χρησιμοποιούμε μια δομή χάρτη (map). Κλειδί (key) είναι η τιμή του στοιχείου στη βάση δεδομένων, δηλαδή ένα νούμερο. Τιμή (value) είναι μια δομή που περιέχει πληροφορίες για τη θέση και την ταξινόμηση του στοιχείου.

Η δομή αυτή επιλέχθηκε καθώς έχει σχεδόν ίδιες λειτουργίες με την δομή ενός δέντρου δυαδικής αναζήτησης (*BST - Binary Search Tree*), και πιο συγκεκριμένα αυτής ενός ασπρόμαυρου δέντρου (*red - black tree*). Επιπλέον, με χρήση χάρτη μπορούμε να εκμεταλλευτούμε τις έτοιμες συναρτήσεις που υλοποιούν τις λειτουργίες εισαγωγής (*insert()*), διαγραφής (*erase()*) και αναζήτησης (*find()*), διευκολύνοντας έτσι την χειροκίνητη υλοποίηση ενός δυαδικού δέντρου.

4.3.1 Δεδομένα Πειραμάτων

Για τα πειράματα που υλοποιούμε, χρησιμοποιούμε συνθετικά δεδομένα. Αρχικά δημιουργούμε έναν πίνακα και τον γεμίζουμε με όσα και όποια δεδομένα (αριθμούς) χρειαζόμαστε κάθε φορά. Έπειτα, δημιουργούμε ένα αρχείο και το ανοίγουμε για δυαδική εγγραφή. Τέλος, γράφουμε τα δεδομένα του πίνακα στο αρχείο, ελέγχουμε ότι

γράφτηκαν όλα και το κλείνουμε. Με την κατάλληλη τροποποίηση της διαδικασίας, μπορούμε να φτιάχνουμε διαφορετικά δυαδικά αρχεία κάθε φορά.

Το αρχείο που θα χρησιμοποιηθεί σε κάθε εκτέλεση το καθορίζει ο χρήστης. Τα datasets που χρησιμοποιήσαμε για την εκτέλεση των πειραμάτων μας είναι τα ακόλουθα:

- 100000000.data, που περιέχει 100000000 τυχαίους αριθμούς σε τυχαία σειρά.
- test.data, περιέχει τους τυχαίους αριθμούς που έχουν χρησιμοποιηθεί στο παράδειγμα της Εικόνας 2.
- 100.data, που περιέχει 100 τυχαίους αριθμούς στο διάστημα $[0, 99]$.
- *tychaia20.data*, που περιέχει 20 τυχαίους αριθμούς στο διάστημα $[0, 99]$.
- 100000.data που περιέχει 100000 τυχαίους αριθμούς στο διάστημα $[0, 99.999]$.

4.3.2 Έξοδος Προγράμματος

Στη συγκεκριμένη υλοποίηση αυτό που μας ενδιαφέρει κυρίως είναι να δείξουμε τους χρόνους τους οποίους χρειάζεται για κάθε πείραμα ο κάθε αλγόριθμος cracking και να τους συγκρίνουμε, ώστε να καταλήξουμε σε συμπεράσματα. Για τον λόγο αυτό, στο τέλος κάθε εκτέλεσης τυπώνεται στο τερματικό ο συνολικός χρόνος που χρειάστηκε ο αλγόριθμος που εκτελούμε, για την ολοκλήρωση όλων των ερωτημάτων που ζητήσαμε να συμβούν.

Μπορούμε επίσης να διακρίνουμε όλες τις διαδικασίες που συμβαίνουν κατά την εκτέλεση ενός αλγορίθμου, τυπώνοντας μετά από κάθε ερώτηση τα σημεία που έγινε το crack, αλλά και ολόκληρο τον πίνακα δεδομένων μας ώστε να φανεί η προοδευτική ταξινόμηση.

4.4 Τα Ερωτήματα

Στην υλοποίηση μας το πλήθος των ερωτημάτων που θα εκτελεστούν σε κάθε εκτέλεση, καθορίζεται από τον χρήστη. Για κάθε νέο ερώτημα, επιλέγονται νέοι αριθμοί για να αποτελέσουν τα άκρα του ερωτήματος.

Επιπλέον από τον χρήστη καθορίζεται και το ποιος θα είναι ο φόρτος εργασίας για κάθε εκτέλεση του προγράμματος.

Όταν ο φόρτος εργασίας επιλέγουμε να είναι τυχαίος (Random), τότε επιλέγονται τυχαία δύο αριθμοί για να αποτελέσουν τα άκρα του ερωτήματος εύρους κάθε φορά.

Όταν ο φόρτος εργασίας επιλέγουμε να είναι διαδοχικός (SeqOver), τότε επιλέγεται το αριστερό άκρο να ξεκινάει από το νούμερο 10 και σε κάθε επανάληψη το ζητούμενο εύρος να αυξάνεται κατά 20 μονάδες. Έτσι πετυχαίνουμε τη διαδοχικότητα του φόρτου.

Δίνεται η επιλογή τροποποιώντας το πρόγραμμα πριν την εκτέλεση, να μπορούμε να δημιουργούμε δικά μας μη τυχαία ερωτήματα όταν χρειάζεται, αλλάζοντας χειροκίνητα τους αριθμούς των δύο ορίων του ερωτήματος.

4.5 Ανάλυση Διαδικασίας Cracking

Στον Πίνακα 1 που ακολουθεί παρακάτω, αναλύονται οι συναρτήσεις, που διαδραματίζουν καθοριστικό ρόλο στην γενικότερη υλοποίηση του αλγορίθμου cracking, ώστε να εκτελεί αποτελεσματικά το σπάσιμο.

<u>Συνάρτηση</u>	<u>Λειτουργία</u>	<u>Τι Επιστρέφει</u>
init	Αρχικοποιεί το σύστημα εκκαθαρίζοντας τα δεδομένα των προηγούμενων εκτελέσεων. Κάνει την αντιγραφή των αρχικών δεδομένων σε νέο πίνακα.	-
view_query	Εκτελεί ένα ερώτημα εύρους [a, b] χρησιμοποιώντας την συνάρτηση crack.	Το αποτέλεσμα της κλήσης της συνάρτησης crack, δηλαδή, τον αριθμό των πλειάδων που βρίσκονται ανάμεσα στις τιμές a και b.
crack	Πραγματοποιεί cracking σε μια στήλη δεδομένων, διασπώντας την σε κομμάτια σύμφωνα με τις τιμές a και b.	Τον αριθμό των πλειάδων που βρίσκονται ανάμεσα στις τιμές a και b.
find_piece	Βρίσκει σε ποιο κομμάτι του cracker index ανήκει μια συγκεκριμένη τιμή και επιστρέφει τα όριά του. Χρησιμοποιείται για να βρει γρήγορα το σωστό υποσύνολο δεδομένων που μας ενδιαφέρει, χωρίς πλήρη σάρωση της βάσης.	Τον επαναλήπτη που δείχνει στο κομμάτι του χάρτη που σχετίζεται με την συγκεκριμένη τιμή.
split_ab	Χωρίζει έναν πίνακα δεδομένων σε τρία μέρη με βάση δύο τιμές (a και	split_ab

	<p>b).</p> <p>Η διάσπαση γίνεται κατά τέτοιο τρόπο ώστε:</p> <ul style="list-style-type: none"> ► Οι τιμές μικρότερες από a να μετακινηθούν στο αριστερό κομμάτι. ► Οι τιμές μεταξύ a και b να βρίσκονται στο μεσαίο κομμάτι. ► Οι τιμές μεγαλύτερες ή ίσες με b να μετακινηθούν στο δεξιό κομμάτι. 	
partition	<p>Διαχωρίζει τα στοιχεία του πίνακα βάσει μίας συγκεκριμένης τιμής.</p> <ul style="list-style-type: none"> ► Τα στοιχεία που είναι μικρότερα από την τιμή, βρίσκονται στην αρχή. ► Τα στοιχεία που είναι μεγαλύτερα ή ίσα με την τιμή, βρίσκονται στο τέλος. 	Έναν δείκτη στο πρώτο στοιχείο που δεν ικανοποιεί την συνθήκη.
add_crack	<p>Δημιουργεί δύο νέα κομμάτια από ένα προηγούμενο κομμάτι προσθέτοντας ένα νέο crack στο cracked index. Αν υπάρχει ήδη το crack, δεν το επαναλαμβάνει. Η κλήση της βοηθά στη δυναμική κατασκευή του cracked index.</p>	Την θέση στην οποία δημιουργεί το crack.

Πίνακας 1: Συναρτήσεις που υλοποιούν την διαδικασία του crack

Γενικότερα, η ροή του προγράμματός μας είναι η εξής:

- Πραγματοποιούμε έλεγχο ώστε να ελέγχουμε ότι τα ορίσματα που δίνουμε ως είσοδο στο τερματικό για την εκτέλεση του προγράμματος είναι σωστά.
- Ανοίγουμε για διάβασμα το δυαδικό αρχείο που δίνουμε ως είσοδο.
- Ξεκινάμε τη διαδικασία χρονομέτρησης.
- Με κλήση της συνάρτησης *init* αρχικοποιούμε το σύστημα μας.
- Εκτελούμε την διαδικασία για όσα ερωτήματα επιλέξαμε.
- Στο τέλος, τυπώνουμε το συνολικό χρόνο που διήρκησε η εκτέλεση.

4.6 Η Συνάρτηση Crack

Η συνάρτηση crack είναι αυτήν που υλοποιεί τον βασικό μηχανισμό του αλγόριθμου cracking, έτσι ώστε όλες οι τιμές που ανήκουν στο εύρος $[a, b]$ να τοποθετούνται σε ένα συνεχόμενο τμήμα του πίνακα.

- Όπως φαίνεται παρακάτω, λαμβάνει ως ορίσματα δύο τιμές a και b .

int crack(value_type a, value_type b)

- Καλεί τη συνάρτηση **find_piece**(*ci, N, a, L1, R1*); για να βρει το κομμάτι του πίνακα που περιέχει την τιμή a και έτσι να ενημερωθούν και οι μεταβλητές $L1$ (*Left*), $R1$ (*Right*) οι οποίες κρατάνε την αρχή και το τέλος του κομματιού.

- Αντίστοιχα υπάρχει και η κλήση για το b , όπως φαίνεται παρακάτω:

int L1, R1, i1; find_piece(ci, N, a, L1, R1);

int L2, R2, i2; find_piece(ci, N, b, L2, R2);

- Μετρά πόσα στοιχεία υπέστησαν επεξεργασία προσθέτοντας στην μεταβλητή $n_touched$ το μέγεθος του αριστερού κομματιού ($R1 - L1$).

n_touched += R1 - L1;

- Εάν τα a και b βρίσκονται στο ίδιο κομμάτι του πίνακα, δηλαδή αν και τα αριστερά τους άκρα, αλλά και τα δεξιά τους είναι ίδια, καλείται η συνάρτηση *split_ab* η οποία σπάει το κομμάτι αυτό σε τρία μέρη ως εξής:

1. Στοιχεία $< a$
2. Στοιχεία μεταξύ των a και b
3. Στοιχεία $\geq b$

if (L1==L2){

assert(R1 == R2);

split_ab(arr, L1, R1, a, b, i1, i2);

- ΣΗΜΕΙΩΣΗ:

$i1$: Δείκτης εκεί που τελειώνουν τα στοιχεία $< a$.

$i2$: Δείκτης εκεί που τελειώνουν τα στοιχεία $[a, b]$.

- Εάν τα a και b βρίσκονται σε διαφορετικά κομμάτια του πίνακα, μετρά πόσα επιπλέον στοιχεία υπέστησαν επεξεργασία προσθέτοντας στην μεταβλητή $n_touched$ το μέγεθος του δεξιού κομματιού ($R2 - L2$). Έπειτα, διαχωρίζει τα στοιχεία του 1^{ου} κομματιού με βάση την τιμή a και τα στοιχεία του 2^{ου} κομματιού με βάση την τιμή b . Έτσι στην αρχή του 1^{ου} κομματιού βρίσκονται οι τιμές που είναι μικρότερες του a και στο τέλος οι τιμές που είναι μεγαλύτερες από αυτό. Αντίστοιχα και για το b . Ενημερώνει τους δείκτες $i1$ και

i2 ώστε να δείχνουν στο 1^ο στοιχείο που δεν ικανοποιεί την ανισότητα και στις δύο περιπτώσεις αντίστοιχα.

```
} else {  
    n_touched += R2 - L2;  
    i1 = partition(arr, a, L1, R1);  
    i2 = partition(arr, b, L2, R2);  
}
```

► Προσθέτει νέα cracks στα σημεία *i1* και *i2*, όπως φαίνεται παρακάτω:

```
add_crack(ci, N, a, i1);  
add_crack(ci, N, b, i2);
```

► Εκτυπώνει τα σημεία διαχωρισμού, δηλαδή τα σημεία στα οποία έχει εκτελεστεί crack.

```
fprintf(stderr, "--print_crackers--\n");  
  
print_crackers(ci);
```

► Η συνάρτηση επιστρέφει έναν ακέραιο αριθμό που αντιπροσωπεύει το πλήθος των στοιχείων / εγγραφών που ανήκουν στο εύρος [*a*, *b*].

```
return i2 - i1;
```

♦ Καθ' όλη τη διάρκεια των κλήσεων των επιμέρους συναρτήσεων κατά τη χρήση της συνάρτησης *crack*, χρονομετρούμε με τη βοήθεια των συναρτήσεων *start()* και *stop()*, του *struct Timer* που έχει οριστεί στο αρχείο *tester.h*, τους χρόνους που χρειάστηκαν για την ολοκλήρωση των διαδικασιών του cracking.

4.6.1 Συνοπτική Επεξήγηση Συνάρτησης Crack

Η συνάρτηση *crack* εκτελεί δυναμικό cracking των δεδομένων σε έναν πίνακα:

1. Βρίσκει σε ποια κομμάτια ανήκουν τα άκρα *a* και *b*.

2. Διαχωρίζει τα δεδομένα σε κομμάτια ως εξής:

2.1 Μέσω διαχωρισμού σε τρία κομμάτια, όταν το εύρος του ερωτήματος εμπίπτει εξ ολοκλήρου σε ένα προηγούμενως μη ταξινομημένο κομμάτι.

2.2 Μέσω διαχωρισμού σε δύο κομμάτια, όταν δεν ισχύει το παραπάνω.

3. Ενημερώνει την δομή δεδομένων (δηλαδή το cracking index, *ci*) δημιουργώντας τα νέα κομμάτια, για μελλοντική επιτάχυνση της διαδικασίας αναζήτησης.

4. Μετρά τον χρόνο εκτέλεσης με τη βοήθεια συναρτήσεων χρόνου.

Αυτή η τεχνική βελτιώνει την απόδοση των αναζητήσεων εύρους καθώς ο πίνακας γίνεται σταδιακά πιο ταξινομημένος κατά τις αναζητήσεις.

4.7 Σύνοψη

Παρουσιάσαμε αναλυτικά την υλοποίηση του λογισμικού database cracking, με έμφαση στις κύριες μεθόδους που χρησιμοποιούνται.

Στον Πίνακα 1 απεικονίζονται οι βασικές λειτουργίες του αλγορίθμου μέσα από τις μεθόδους του κώδικα που τις υλοποιούν.

Επιπλέον, αναλύονται όλες οι χρήσιμες πληροφορίες για την εκτέλεση του προγράμματος. Αναλύθηκε το αρχείο εισόδου, η διαμόρφωση των παραμέτρων (όπως η επιλογή φόρτου εργασίας και ο αριθμός ερωτημάτων) και η μορφή της εξόδου, η οποία αποτυπώνει τα αποτελέσματα της cracking διαδικασίας.

Η ανάλυση αυτή επιτρέπει σε οποιονδήποτε χρήστη να κατανοήσει και να τρέξει το πρόγραμμα, αξιοποιώντας τις δυνατότητες του database cracking για αποδοτική διαχείριση δεδομένων.

Κεφάλαιο 5. Πειραματική Αξιολόγηση

Σε αυτό το κεφάλαιο, αρχικά παρουσιάζουμε ένα ολοκληρωμένο παράδειγμα για να κατανοήσουμε την λειτουργία του βασικού αλγόριθμου cracking. Έπειτα, μέσα από μετρήσεις παρουσιάζουμε τα αποτελέσματα για διαφορετικά αρχεία και διαφορετικό αριθμό και είδος ερωτημάτων. Συγκρίνουμε την απόδοση των αλγορίθμων που υλοποιήσαμε και εξάγουμε συμπεράσματα.

Σκοπός των πειραμάτων είναι να επαληθεύσουμε τη λειτουργία του προγράμματός μας, καθώς και να κάνουμε παρατηρήσεις πάνω στον χρόνο εκτέλεσης διαφορετικών πειραμάτων.

Όλα τα πειράματα που ακολουθούν στην συνέχεια, εκτελέστηκαν στους υπολογιστές του εργαστηρίου του Τμήματος Μηχανικών Η/Υ και Πληροφορικής. Οι υπολογιστές φέρουν τετραπύρηνους επεξεργαστές Intel(R) Xeon(R) CPU L5520 @2.27GHz, η διαθέσιμη μνήμη RAM είναι 1975 MB και το λειτουργικό σύστημα που τρέχουν είναι Ubuntu 22.04.1 LTS.

5.1 Αναλυτική Παρουσίαση Αποτελεσμάτων

Εφόσον στην συγκεκριμένη διπλωματική εργασία έχουμε αναλύσει εκτενώς το παράδειγμα της Εικόνας 2, επιλέγουμε να παρουσιάσουμε την εκτέλεσή του και αναλυτικά στην συγκεκριμένη υπό-ενότητα.

5.1.1 Αναλυτικά Αποτελέσματα Πειράματος

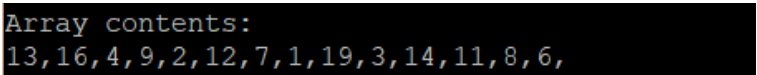
♦ Αρχικά, δημιουργούμε το δυαδικό αρχείο *test.data* και του προσθέτουμε τα δεδομένα του παραδείγματος της Εικόνας 2.

♦ Έπειτα, τροποποιούμε κατάλληλα την μέθοδο *query* στο αρχείο *workload.h*, έτσι ώστε αντί για τυχαία ερωτήματα να εκτελεί τα ερωτήματα του παραδείγματός μας, δηλαδή τα $Q1 = (10,14)$, $Q2 = (7,16)$ και $Q3 = (10,12)$.

♦ Τέλος εκτελούμε την εντολή:

```
./run.sh test.data crack 3 Random 1e-2 NOUP 30
```

Πρώτα εμφανίζονται τα δεδομένα μας στο τερματικό όπως φαίνεται στην Εικόνα 13.



```
Array contents:  
13, 16, 4, 9, 2, 12, 7, 1, 19, 3, 14, 11, 8, 6,
```

Εικόνα 13: Αρχικά δεδομένα στήλης

Το αποτέλεσμα μετά την εκτέλεση του ερωτήματος **Q1 = (10,14)** παρουσιάζεται στην Εικόνα 14.

Παρατηρούμε ότι συμβαίνουν **δύο crack**.

- Το πρώτο προκαλείται από την τιμή **10** και γίνεται στη θέση 8, και
- Το δεύτερο προκαλείται από την τιμή **14** και γίνεται στη θέση 11. Επιπρόσθετα, τυπώνεται ο cracker index που κρατάει τις τιμές (Value) και τις θέσεις (Pos) αντίστοιχα, στις οποίες έχει συμβεί «σπάσιμο».

```
Query generated: [10, 14]

---> add crack v = 10, p = 8

---> add crack v = 14, p = 11
--print cracker index--
Value: 10, Pos: 8
Value: 14, Pos: 11
```

Εικόνα 14: Εκτέλεση ερωτήματος (10, 14)

Τέλος, παρουσιάζονται τα δεδομένα του πίνακά μας μετά την εφαρμογή του αλγορίθμου και έτσι βλέπουμε στην Εικόνα 15 ότι αυτά έχουν ταξινομηθεί σε **τρία κομμάτια**, σε σύγκριση με τα αρχικά δεδομένα της Εικόνας 13.

```
--RESULT AFTER CRACKING--
Array contents: [
6, 4, 9, 2, 7, 1, 8, 3, 13, 12, 11, 14, 19, 16,
```

Εικόνα 15: Αποτέλεσμα ερωτήματος (10, 14)

Το αποτέλεσμα μετά την εκτέλεση του ερωτήματος **Q2 = (7,16)** παρουσιάζεται στην Εικόνα 16.

Παρατηρούμε ότι συμβαίνουν πάλι **δύο crack**.

- Το πρώτο προκαλείται από την τιμή **7** και γίνεται στη θέση 5, και
- Το δεύτερο προκαλείται από την τιμή **16** και γίνεται στη θέση 12.

Επιπρόσθετα, τυπώνεται ο cracker index όπως συνέβη και πριν. Παρατηρούμε ότι πλέον έχει ενημερωθεί και αυτός, καθώς προστέθηκαν οι δύο εγγραφές που αφορούν το Q2.

Τέλος, παρουσιάζονται τα δεδομένα του πίνακά μας μετά την εφαρμογή του αλγορίθμου και έτσι βλέπουμε στην Εικόνα 16 ότι αυτά έχουν ταξινομηθεί πλέον σε **πέντε κομμάτια**, αντί για τρία όπως συμβαίνει στην Εικόνα 15.

```

Query generated: [7, 16]

---> add crack v = 7, p = 5

---> add crack v = 16, p = 12
--print cracker index--
Value: 7, Pos: 5
Value: 10, Pos: 8
Value: 14, Pos: 11
Value: 16, Pos: 12
--RESULT AFTER CRACKING--
Array contents:
6, 4, 3, 2, 1, 7, 8, 9, 13, 12, 11, 14, 19, 16,

```

Εικόνα 16: Αποτέλεσμα εκτέλεσης ερωτήματος (7, 16)

Το αποτέλεσμα μετά την εκτέλεση του ερωτήματος **Q3 = (10,12)** παρουσιάζεται στην Εικόνα 17.

Παρατηρούμε ότι αυτή τη φορά συμβαίνει μόνο **ένα crack**. Αυτό γιατί ο αριθμός 10 αποτέλεσε άκρο του ερωτήματος Q1 και έτσι υπάρχει ήδη crack στην θέση 8 του πίνακα για αυτόν.

► Το crack προκαλείται από την τιμή **12** και γίνεται στη θέση 9. Επιπρόσθετα, τυπώνεται ο cracker index όπως συνέβη και πριν. Παρατηρούμε ότι πλέον έχει ενημερωθεί και αυτός, καθώς προστέθηκε η εγγραφή που αφορά το Q3.

Τέλος, παρουσιάζονται τα δεδομένα του πίνακά μας μετά την εφαρμογή του αλγορίθμου και έτσι βλέπουμε στην Εικόνα 17 ότι αυτά έχουν ταξινομηθεί πλέον σε έξι κομμάτια, αντί για πέντε όπως συμβαίνει στην Εικόνα 16.

```

Query generated: [10, 12]

---> add crack v = 12, p = 9
--print cracker index--
Value: 7, Pos: 5
Value: 10, Pos: 8
Value: 12, Pos: 9
Value: 14, Pos: 11
Value: 16, Pos: 12
--RESULT AFTER CRACKING--
Array contents:
6, 4, 3, 2, 1, 7, 8, 9, 11, 12, 13, 14, 19, 16,

```

Εικόνα 17: Αποτέλεσμα εκτέλεσης ερωτήματος (10, 12)

Όπως φαίνεται στην Εικόνα 18, η εκτέλεση του προγράμματος στο τέλος τυπώνει με *T* τον χρόνο που χρειάστηκε σε δευτερόλεπτα και με *Q* τα ερωτήματα που απαντήθηκαν.

```
T= 0.000442 Q=3
```

Εικόνα 18: Χρόνος εκτέλεσης αλγορίθμου για τρία ερωτήματα

5.2 Αποτελέσματα

Αφού αποδείξαμε στην προηγούμενη υπό-ενότητα ότι ο αλγόριθμος του βασικού cracking λειτουργεί σωστά, σε αυτήν την υπό-ενότητα παρουσιάζουμε με σχεδιαγράμματα τα αποτελέσματα από την εφαρμογή των διάφορων αλγορίθμων cracking σε διαφορετικές περιπτώσεις και φόρτους εργασίας.

5.2.1 Μελέτη Ταξινόμησης

Σε αυτό το πείραμα παρουσιάζουμε το αποτέλεσμα της εκτέλεσης του προγράμματος αναφορικά με το ποσοστό επιτυχημένης ταξινόμησης. Χρησιμοποιούμε το αρχείο *100.data* που φτιάξαμε και περιέχει 100 τυχαίους αριθμούς στο διάστημα [0, 99].

Αρχικά το ποσοστό ταξινόμησης στον πίνακα είναι μόλις **1%**. Δηλαδή μόνο ένα στοιχείο από τα 100 του αρχείου βρίσκεται στη σωστή θέση.

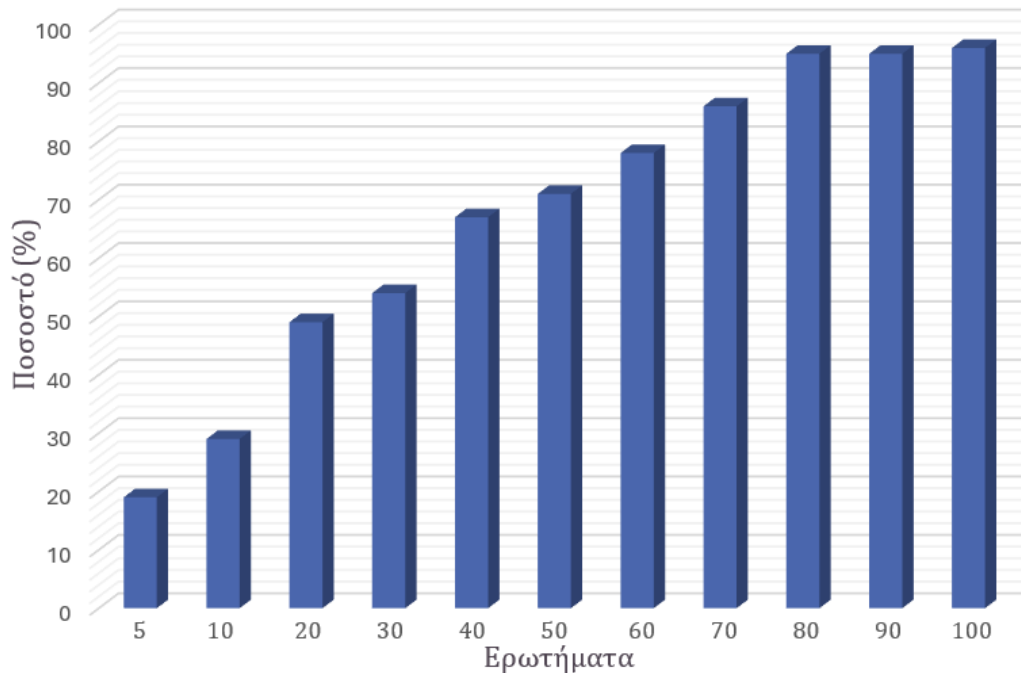
Όσο αυξάνονται τα ερωτήματα, τόσο περισσότερο ταξινομείται ο πίνακας μας. Στον Πίνακα 2 φαίνεται για διαφορετικό αριθμό ερωτημάτων το ποσοστό ταξινόμησης που παρουσιάζει ο πίνακας. Παρατηρούμε ότι το ποσοστό ταξινόμησης για 80 μόλις ερωτήματα, μπορεί να αγγίξει μέχρι και την τιμή **95%**.

Ερωτήματα	5	10	20	30	40	50	60	70	80	90	100
Ποσοστό (%)	19	29	49	54	67	71	78	86	95	95	96

Πίνακας 2: Ποσοστό ταξινόμησης πίνακα για διαφορετικό πλήθος ερωτημάτων

Στην Εικόνα 19 φαίνεται διαγραμματικά αυτή η ταξινόμηση. Παρατηρούμε ότι ακόμα και με λίγα ερωτήματα ο πίνακας παρουσιάζει αισθητή ταξινόμηση. Μετά τα 60 ερωτήματα ο πίνακας σχεδόν έχει ταξινομηθεί ολόκληρος, αφού πλέον περισσότερα από 75 στοιχεία βρίσκονται στη σωστή θέση.

Ποσοστά Ταξινόμησης



Εικόνα 19: Ποσοστό επιτυχούς ταξινόμησης του αλγορίθμου όσο αυξάνονται τα ερωτήματα που εκτελούνται.

5.2.2 Σωστή Εκτέλεση Ταξινόμησης

Στο τρίτο αυτό πείραμα, παρουσιάζουμε το αποτέλεσμα της εκτέλεσης του προγράμματος, για να δείξουμε την ταξινόμηση των στοιχείων. Χρησιμοποιούμε το αρχείο *tyxaia20.data* που φτιάξαμε και περιέχει 20 τυχαίους αριθμούς στο διάστημα [0, 99].

Αρχικά το αρχείο μας έχει τη μορφή που φαίνεται στην Εικόνα 20.

```
Array contents:  
52,20,27,37,11,63,14,92,17,81,76,8,78,6,25,28,19,3,71,64,
```

Εικόνα 20: Αρχική μορφή πίνακα.

Αντίθετα με την Εικόνα 20, στην οποία τα στοιχεία δεν παρουσιάζουν καμία ταξινόμηση, στην Εικόνα 21 παρουσιάζεται ο πίνακας εντελώς ταξινομημένος.

Αυτό έχει συμβεί έπειτα από εκτέλεση της εντολής:

```
./run.sh tyxaia20.data crack 20 Random 1e-2 NOUP 30
```

Η εντολή αυτή εκτελεί τον αλγόριθμο cracking για μόλις 20 ερωτήματα και έχει καταφέρει να δώσει την εικόνα μιας πλήρους ταξινόμησης.

```
--RESULT AFTER CRACKING--  
Array contents:  
3, 6, 8, 11, 14, 17, 19, 20, 25, 27, 28, 37, 52, 63, 64, 71, 76, 78, 81, 92,
```

Εικόνα 21: Τελική μορφή πίνακα.

♦ Αφού μέσω των προηγούμενων πειραμάτων έχουμε αποδείξει με αποτελέσματα την ορθή λειτουργία του αλγορίθμου, αλλά και την ικανότητα ταξινόμησης που παρουσιάζει, μπορούμε να εκτελέσουμε μερικά ακόμη πειράματα ώστε να τον συγκρίνουμε με τους αλγόριθμους sort και scan.

- Ο αλγόριθμος **Sort, ταξινομεί τα δεδομένα** πριν απαντήσει στα ερωτήματα. Αυτό βοηθά στη γρήγορη εύρεση των απαντήσεων, αλλά απαιτεί πολύ χρόνο.
- Ο αλγόριθμος **Scan, σαρώνει ολόκληρο το σύνολο δεδομένων** για κάθε ερώτημα. Αυτό είναι πολύ χρονοβόρο, ειδικά για μεγάλα σύνολα δεδομένων και μεγάλο αριθμό ερωτημάτων.

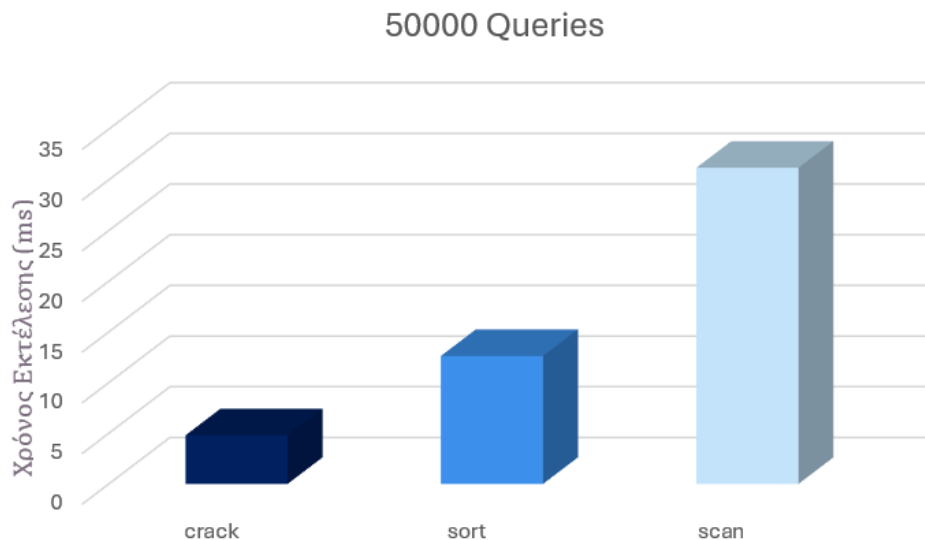
5.2.3 Μελέτη Χρόνου Εκτέλεσης Crack για Τυχαίο Φόρτο Εργασίας

Σε αυτό το πείραμα χρησιμοποιούμε το αρχείο *100000000.data*, το οποίο περιέχει 100 εκατομμύρια τυχαίες τιμές στο εύρος $[0, 99.999.999]$. Τα τόσα πολλά δεδομένα αξίζει να εξετάζονται με πειράματα, καθώς οι περισσότερες βάσεις δεδομένων σήμερα, αποθηκεύουν και επεξεργάζονται τεράστιες ποσότητες δεδομένων.

Όπως φαίνεται στην Εικόνα 22, για σταθερό αριθμό τυχαίων επερωτήσεων ίσο πάντα με 50.000, ο αλγόριθμος crack πετυχαίνει καλύτερη επίδοση σε σχέση με τον sort που είναι ο επόμενος καλύτερος αλγόριθμος.

Για την εκτέλεση των ερωτήσεων που υλοποιήσαμε ο αλγόριθμος crack παρουσιάζεται να είναι **3.5 φορές γρηγορότερος** από τους υπόλοιπους. Μία διαφορά χρόνου τέτοιου βεληνεκούς, αποτελεί τεράστιο προσόν για τον αλγόριθμο αυτό.

Χειρότερος αλγόριθμος με μεγάλη διαφορά αποτελεί ο αλγόριθμος scan.



Εικόνα 22: Χρόνος εκτέλεσης των αλγορίθμων *crack*, *sort* και *scan* για τυχαίο φόρτο εργασίας.

5.2.4 Μελέτη Απόδοσης των Αλγορίθμων Crack, Sort και Scan

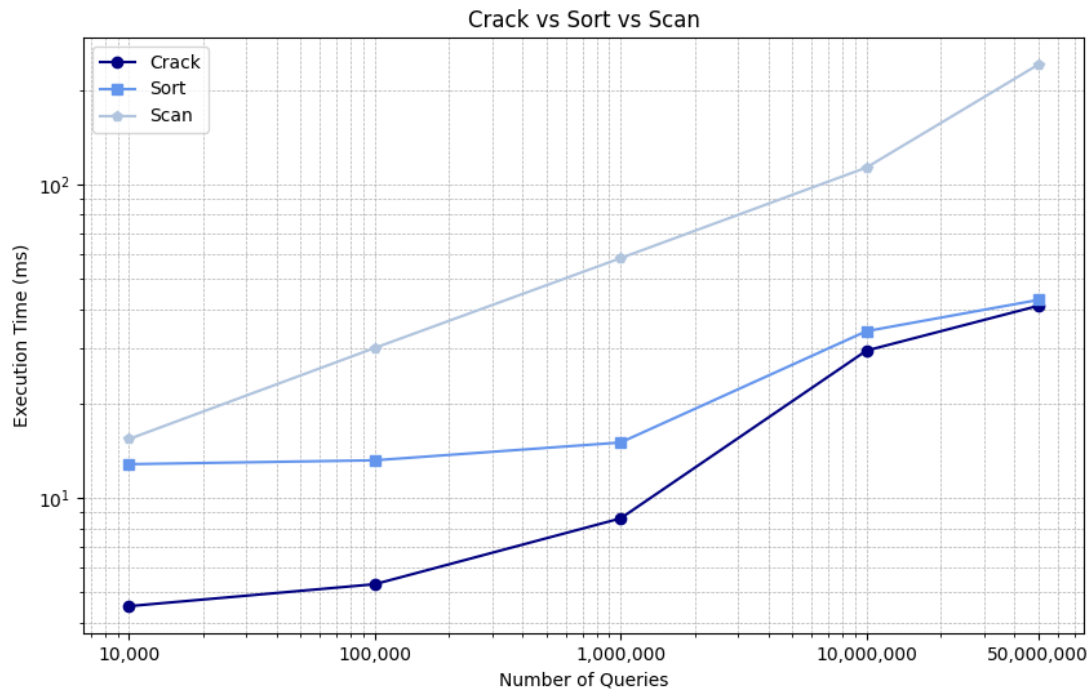
Σε αυτό το πείραμα χρησιμοποιούμε το αρχείο *100000000.data*, το οποίο περιέχει 100 εκατομμύρια τυχαίες τιμές στο εύρος $[0, 999.999.999]$.

Με πολλαπλά πειράματα, συγκρίνουμε την απόδοση του αλγορίθμου *crack* σε σχέση με την απόδοση των αλγορίθμων *sort* και *scan*, ανάλογα με τον αριθμό των ερωτημάτων που εκτελούνται. Τα ερωτήματα είναι πάντα τυχαία.

Ανάλυση των μεθόδων:

- Ο αλγόριθμος *Crack* φαίνεται να έχει την καλύτερη απόδοση από τους τρεις, ειδικά για μεγάλο αριθμό ερωτημάτων. Επίσης, ο χρόνος εκτέλεσης αυξάνεται σχετικά αργά καθώς αυξάνεται ο αριθμός των ερωτημάτων.
- Ο αλγόριθμος *Sort* (ταξινόμηση) έχει μια ενδιάμεση απόδοση. Ο χρόνος εκτέλεσης ωστόσο παρουσιάζεται αυξημένος από την πρώτη κιόλας μέτρηση.
- Ο αλγόριθμος *Scan* (σάρωση) έχει τη χειρότερη απόδοση από τους τρεις. Ο χρόνος εκτέλεσης αυξάνεται πολύ γρήγορα και σχεδόν γραμμικά με τον αριθμό των ερωτημάτων.

Όπως φαίνεται στο σχεδιάγραμμα της Εικόνας 23, στον αλγόριθμο *crack* όσο τα ερωτήματα είναι λιγότερα, ο χρόνος είναι φανερά πιο μειωμένος. Αξίζει να σημειώσουμε αυτή την ανωτερότητα του αλγορίθμου η οποία παρατηρείται διότι ο αλγόριθμος ψάχνει σε μικρότερα κομμάτια κάθε φορά τις απαντήσεις για τα ερωτήματα, αντί για ολόκληρο τον πίνακα δεδομένων. Έτσι μετριάζεται ο μεγάλος όγκος δεδομένων.



Εικόνα 23: Απόδοση των αλγορίθμων crack, sort και scan για τυχαίο φόρτο εργασίας.

Το διάγραμμα αυτό έχει μεγάλη σημασία για την επιλογή της κατάλληλης μεθόδου ανάλογα με τις ανάγκες του χρήστη. Αν έχουμε πολλά ερωτήματα να εκτελέσουμε σε ένα μεγάλο σύνολο δεδομένων, ο αλγόριθμος crack αποτελεί την καλύτερη επιλογή.

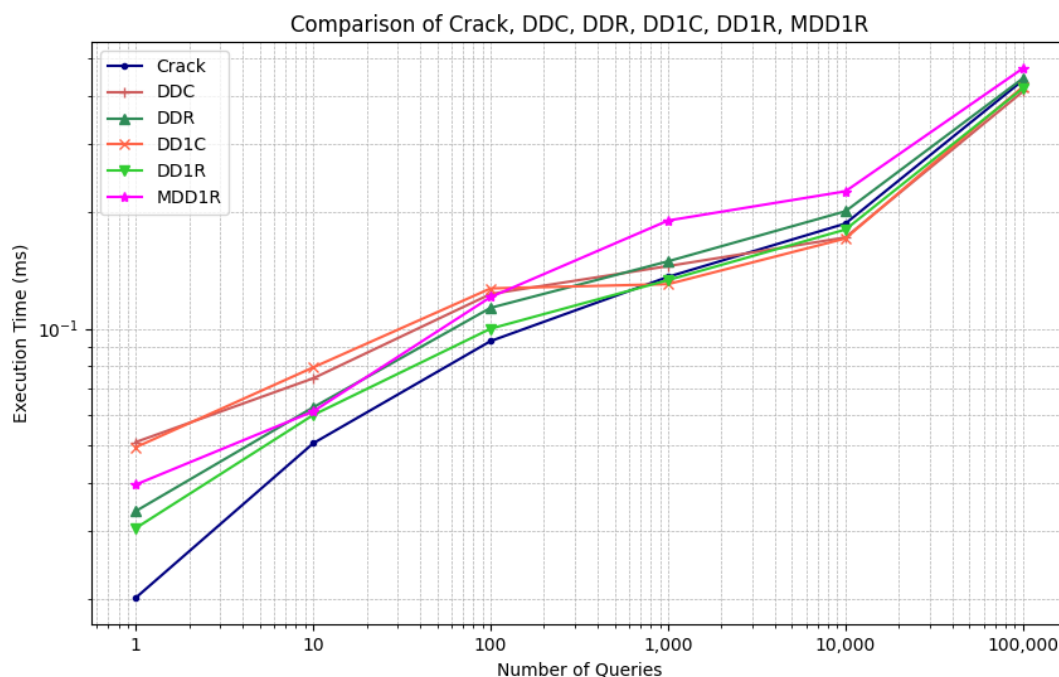
5.2.5 Μελέτη Απόδοσης των Στοχαστικών Αλγορίθμων για Τυχαίο Φόρτο Εργασίας

Σε αυτό το πείραμα χρησιμοποιούμε το αρχείο *100000.data* που φτιάξαμε και περιέχει 100.000 τυχαίους αριθμούς. Στην Εικόνα 24 παρουσιάζουμε αναλυτικά την απόδοση των στοχαστικών αλγορίθμων DDC, DDR, DD1C, DD1R και MDD1R σε σχέση με αυτή του crack, όταν ο φόρτος εργασίας είναι τυχαίος.

Έπειτα από πειράματα συμπεραίνουμε ότι σε τυχαίο φόρτο εργασίας σχεδόν όλοι οι αλγόριθμοι στοχαστικού cracking παρουσιάζουν παρόμοια απόδοση με το βασικό cracking, διατηρώντας την προσαρμοστικότητά του και τις καλές ιδιότητες, σχετικά με το κόστος αρχικοποίησης και τη σύγκλιση.

Το βασικό cracking είναι ελάχιστα ταχύτερο κατά τη διάρκεια της περιόδου αρχικοποίησης, δηλαδή κατά τη διάρκεια των πρώτων λίγων ερωτημάτων. Όσο όμως τα ερωτήματα αυξάνονται, τα βοηθητικά cracking που εκτελούν οι στοχαστικοί αλγόριθμοι cracking, αρχίζουν να επηρεάζουν θετικά την απόδοσή τους. Το κέρδος αυτό ωστόσο, είναι οριακό, με αποτέλεσμα ο συνδυασμός του προοδευτικού στοχαστικού cracking με

το cracking βασισμένο στα ερωτήματα, να πετυχαίνουν την ίδια συμπεριφορά με το βασικό cracking.



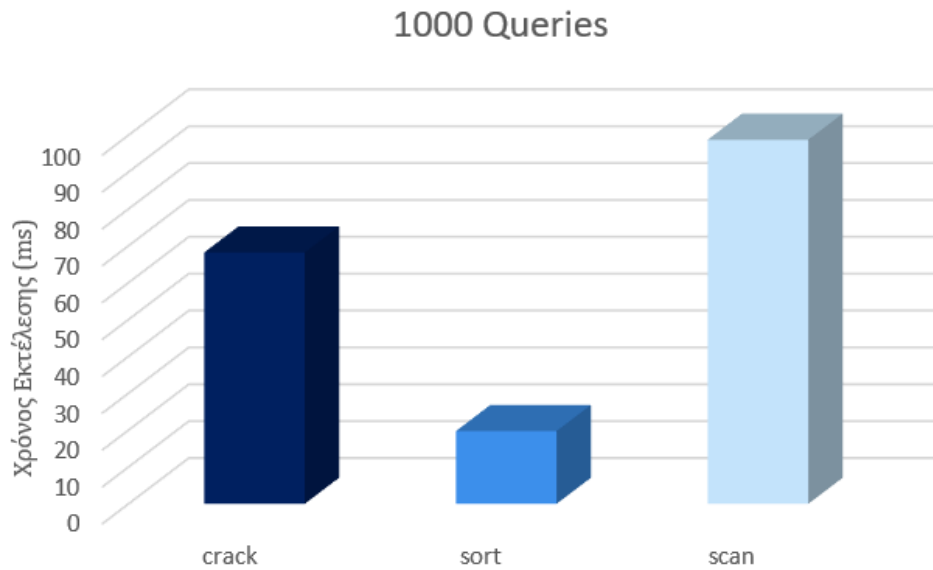
Εικόνα 24: Απόδοση των αλγορίθμων crack, ddc, ddr, dd1c, dd1r και mdd1r για τυχαίο φόρτο εργασίας.

5.2.6 Μελέτη Χρόνου Εκτέλεσης Crack για Διαδοχικό Φόρτο Εργασίας

Σε αυτό το πείραμα χρησιμοποιούμε το αρχείο *1000000000.data*, το οποίο περιέχει 100 εκατομμύρια τυχαίες τιμές στο εύρος $[0, 999.999.999]$. Εξετάζουμε τον συνολικό χρόνο απόκρισης των μεθόδων crack, sort και scan όταν έχουμε διαδοχικό φόρτο εργασίας.

Εκτελέσαμε την εντολή: `./run.sh 1000000000.data crack 1000 SeqOver 1e-2 NOUP 30` και αντίστοιχα για τους αλγορίθμους sort και scan.

Πιο συγκεκριμένα, με χρήση του SeqOver workload, ο φόρτος εργασίας αποτελείται από ερωτήματα τα οποία ζητάνε εύρος τιμών το οποίο αυξάνεται κατά 20 μονάδες κάθε φορά, ξεκινώντας με το νούμερο 10 για αριστερό όριο. Όπως παρατηρούμε στην Εικόνα 25, το crack χάνει το βασικό του πλεονέκτημα, καθώς η απόδοσή του είναι χειρότερη από ότι του αλγορίθμου sort και θα μπορούσαμε να θεωρήσουμε ότι πλησιάζει την απόδοση του αλγορίθμου scan.



Εικόνα 25: Χρόνος εκτέλεσης των αλγορίθμων *crack*, *sort* και *scan* για διαδοχικό φόρτο εργασίας.

♦ Σκοπός του προηγούμενου πειράματος ήταν ο δείξουμε την μη καταλληλότητα του αλγορίθμου *crack*, σε περιπτώσεις όπου ο φόρτος εργασίας είναι διαδοχικός. Οποιοσδήποτε φόρτος εργασίας πιο ρεαλιστικός, όπως είναι ο διαδοχικός φόρτος, μπορεί να διαδραματίσει σημαντικό ρόλο και μάλιστα αρνητικό στη λειτουργία του βασικού αλγορίθμου *crack*.

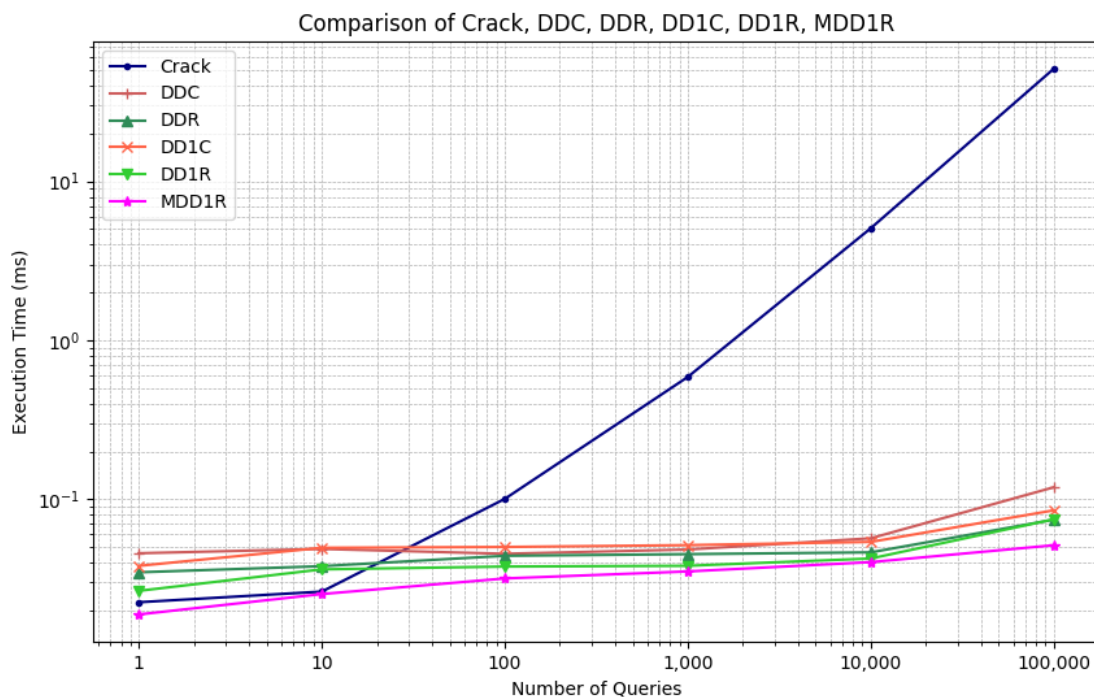
Στα πειράματα που ακολουθούν, χρησιμοποιούμε τους αλγόριθμους στοχαστικού *cracking* και συγκρίνουμε τις αποδόσεις τους με αυτές του βασικού *cracking*. Χρησιμοποιούμε πάντα διαδοχικό φόρτο εργασίας (*sequential workload*), καθώς για τέτοιους «προβληματικούς» φόρτους εργασίας χρειάζεται να δείξουμε ότι το στοχαστικό *cracking* παρουσιάζει φανερά καλύτερη απόδοση και πιο συγκεκριμένα ο αλγόριθμος MDD1R πετυχαίνει την καλύτερη γενική απόδοση.

5.2.7 Μελέτη Απόδοσης των Στοχαστικών Αλγορίθμων για Διαδοχικό Φόρτο Εργασίας

Σε αυτό το πείραμα χρησιμοποιούμε το αρχείο *100000.data* που φτιάξαμε και περιέχει 100.000 τυχαίους αριθμούς. Έπειτα από πειράματα για διαδοχικό φόρτο εργασίας, παρουσιάζουμε αναλυτικά στην Εικόνα 26 την απόδοση των στοχαστικών αλγορίθμων DDC, DDR, DD1C, DD1R και MDD1R σε σχέση με αυτή του *crack*.

Ανάλυση των μεθόδων:

- ▶ Ο αλγόριθμος crack εμφανίζει τη χειρότερη απόδοση από όλους τους αλγορίθμους. Η εκτελεστική του ικανότητα αυξάνεται σχεδόν εκθετικά με τον αριθμό των ερωτημάτων που εκτελούνται. Σίγουρα για τέτοιους φόρτους εργασίας δεν είναι αποτελεσματικός.
- ▶ Οι αλγόριθμοι DDC και DD1C είναι σημαντικά καλύτεροι από τον αλγόριθμο crack, καθώς σπάνε τη βάση στη μέση και έπειτα απαντάνε στο ερώτημα, αλλά δεν είναι οι καλύτεροι σε αυτό το σύνολο. Μεταξύ τους, ο αλγόριθμος DD1C παρουσιάζεται ελαφρώς καλύτερος από τον αλγόριθμο DDC, λόγω της εκτέλεσης ενός αντί για δύο crack κάθε φορά.
- ▶ Οι αλγόριθμοι DDR και DD1R επίσης είναι καλύτεροι από τον αλγόριθμο crack, αλλά παρουσιάζονται να είναι καλύτεροι και από τους αλγορίθμους DDC και DD1C. Η μεγαλύτερη αποτελεσματικότητά τους οφείλεται στο ότι εκτελούν το σπάσιμο σε τυχαία σημεία αντί να σπαταλάνε χρόνο για να εντοπίσουν τα κέντρα κάθε φορά. Ωστόσο ούτε αυτοί φαίνεται να είναι οι καλύτεροι σε αυτό το σύνολο. Μεταξύ τους, ο αλγόριθμος DD1R παρουσιάζεται ελαφρώς καλύτερος από τον αλγόριθμο DDR, λόγω της εκτέλεσης ενός αντί για δύο crack κάθε φορά.
- ▶ Ο αλγόριθμος MDD1R είναι αυτός που εμφανίζει την καλύτερη απόδοση από όλους τους υπόλοιπους αλγορίθμους, με την εκτελεστική του ικανότητα να παραμένει σχεδόν σταθερή και πάντα γρήγορη, σε κάθε πείραμα και ας αυξάνουμε τον αριθμό των ερωτημάτων.



Εικόνα 26: Απόδοση των αλγορίθμων crack, ddc, ddr, dd1c, dd1r και mdd1r για διαδοχικό φόρτο εργασίας.

Το διάγραμμα αυτό έχει μεγάλη σημασία για την επιλογή του κατάλληλου αλγορίθμου ανάλογα με τις ανάγκες του χρήστη. Αν χρειαζόμαστε υψηλή απόδοση για μεγάλο όγκο διαδοχικών ερωτημάτων τότε ο αλγόριθμος MDD1R αποτελεί την καλύτερη επιλογή. Ωστόσο, καλές επιλογές αποτελούν με τη σειρά τους οι αλγόριθμοι DD1R και DDR. Τέλος οι αλγόριθμοι DDC και DD1C δεν αποτελούν καλές επιλογές, καθώς στο συνολικό τους κόστος προστίθεται το κόστος εύρεσης του κέντρου, στο οποίο οι συγκεκριμένοι αλγόριθμοι κάνουν το crack.

Κεφάλαιο 6. Επίλογος

6.1 Σύνοψη και Συμπεράσματα

Στην παρούσα διπλωματική εργασία, παρουσιάσαμε τη μεθοδολογία του database cracking, μια τεχνική ευρετηρίασης και προοδευτικής οργάνωσης δεδομένων, που προσφέρει σημαντικά πλεονεκτήματα στην απόδοση των ερωτημάτων σε συστήματα βάσεων δεδομένων.

Αρχικά, παρουσιάσαμε βασικές γνώσεις σχετικά με το database cracking. Εξετάσαμε τους βασικούς και στοχαστικούς αλγορίθμους που εκτελούν cracking σε μια στήλη, εξηγώντας τη λειτουργία τους και τις περιπτώσεις στις οποίες είναι πιο αποτελεσματικοί για χρήση. Δώσαμε έμφαση στην υλοποίηση που χρησιμοποιεί το database cracking για την εκτέλεση ερωτημάτων εύρους σε πίνακες δεδομένων, παρουσιάζοντας αναλυτικά την αρχιτεκτονική και τη λειτουργία του συστήματος.

Όπως φάνηκε στο κεφάλαιο αξιολόγησης, ο αλγόριθμος cracking παρουσιάζει υψηλή απόδοση για τυχαίο φόρτο εργασίας και ο αλγόριθμος MDD1R για διαδοχικό φόρτο εργασίας. Επιβεβαιώσαμε την δυνατότητά όλων των αλγορίθμων cracking για γρήγορη επεξεργασία ερωτημάτων και ταξινόμηση. Γενικότερα τα αποτελέσματα είναι ενθαρρυντικά για την ευρύτερη χρήση τέτοιων αλγορίθμων σε συστήματα βάσεων δεδομένων.

Συνοψίζοντας, θεωρούμε ότι το cracking αποτελεί μια καινοτόμο και δυναμική προσέγγιση για την οργάνωση και την ευρετηρίαση δεδομένων. Η ικανότητά του να προσαρμόζεται στις ανάγκες των ερωτημάτων των διαφόρων φόρτων εργασίας, αλλά και η ικανότητα να προσφέρει υψηλή απόδοση, το καθιστά μια ελκυστική λύση για την αντιμετώπιση των προκλήσεων που θέτουν οι σύγχρονες εφαρμογές διαχείρισης δεδομένων.

6.2 Μελλοντικές Επεκτάσεις

Μία από τις πιθανές μελλοντικές επεκτάσεις της παρούσας διπλωματικής εργασίας, θα μπορούσε να είναι η τροποποίηση του κώδικά μας, σχετικά με τη διαδικασία του cracking, ώστε να υπάρξει περαιτέρω βελτίωση στο συνολικό κόστος του.

Πιο συγκεκριμένα, θα αποτελούσε μια ενδιαφέρουσα προσέγγιση η αναδιοργάνωση του πίνακά μας σε κομμάτια, τα οποία θα δημιουργούνται και πάλι με βάση τα άκρα του κάθε

ερωτήματος. Στην συνέχεια, εφόσον υπάρχει η γνώση των στοιχείων που βρίσκονται σε κάθε κομμάτι, θα μπορούσαμε να βρούμε το διάμεσο (median) στοιχείο του κάθε κομματιού και να εφαρμόσουμε το crack στο σημείο που βρίσκεται το στοιχείο αυτό, αντί για το άκρο του ερωτήματος που χρησιμοποιούμε στην παρούσα εργασία.

Η προσέγγιση αυτή θα μπορούσε να κατανέμει καλύτερα τα δεδομένα, καθώς το διάμεσο στοιχείο θα μπορούσε να αποφύγει ακραίες περιπτώσεις αριθμών που ίσως υπάρχουν ως άκρα ερωτήματος. Παρά το γεγονός ότι η προσέγγιση αυτή απαιτεί γνώση του διαμέσου στοιχείου για κάθε κομμάτι δεδομένων, μετά την εκτέλεση αρκετών ερωτημάτων θα μπορούσε να γίνει απόσβεση του χρόνου εύρεσής του. Συνεπώς, ίσως η προσέγγιση αυτή να μπορέσει να πετύχει ακόμη καλύτερες επιδόσεις σε σχέση με το βασικό και το στοχαστικό cracking.

Κεφάλαιο 7. Βιβλιογραφία

- [IdKM07] S. Idreos, M. L. Kersten, and S. Manegold. “Database Cracking”. In Proceedings of the 3rd International Conference on Innovative Data Systems Research (CIDR), pp. 68-78, 2007.
- [Mamo24] Nikos Mamoulis. “Opportunistic, Progressive, Adaptive Multidimensional Indexing”. *University of Ioannina*, 2024.
- [HIKY12] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap, “Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores”. *Proceedings of the Very Large Databases Endowment (PVLDB)*, vol. 5, no. 6, pp. 502-513, 2012
- [ZaAK20] F. Zardbani, P. Afshani, and P. Karras. “Revisiting the Theory and Practice of Database Cracking”. In *Proceedings of the 23rd International Conference on Extending Database Technology (EDBT)*, pp. 415–418, 2020.
- [Hali23] Felix Halim. 2023. scrack: Stochastic Database Cracking. GitHub repository. Retrieved from <https://github.com/felix-halim/scrack>
- [Wern18] D. Werner. “Database Cracking”. 2018 Available at <https://db.in.tum.de/teaching/ws1718/seminarHauptspeicherdbspaper/werner.pdf>
- [ScJD16] Schuhknecht, Felix Martin, Alekh Jindal, and Jens Dittrich. “An Experimental Evaluation and Analysis of Database Cracking.” *The VLDB Journal* 25.1, pp.27-52, 2016. Available at https://dspace.mit.edu/bitstream/handle/1721.1/106180/778_2015_Article_397.pdf?sequence=1&isAllowed=y