ΤΟ ΣΥΣΤΗΜΑ ΔΙΑΧΕΙΡΙΣΗΣ ΒΑΣΗΣ ΔΕΔΟΜΕΝΩΝ Nidus DBMS

(εργασία για το μάθημα Υλοποίησης Βάσεων Δεδομένων)


```
/**
 * Αυγουστάκης Χρυσοβαλάντης (880)
 * Κρητικός Απόστολος (914)
 * Σκαλιστής Στέφανος (1024)
 * Φιλίππου Γεώργιος (1236)
 */
```

Περιεχόμενα

Περιγραφή του προβλήματος

Στην παρούσα εργασία καλούμαστε να υλοποιήσουμε ένα μικρό σύστημα διαχείρισης βάσης δεδομένων το οποίο ονομάσαμε Nidus DBMS.
Για να αντιμετωπίσουμε την πολυπλοκότητα του συστήματος αυτού, το χωρίσαμε σε 4 πακέτα τα οποία χειρίζονται διαφορετικές πτυχές του. Τα πακέτα αυτά είναι τα εξής: `bplus`, `util`, `parser` και `execution`. Προσπαθήσαμε να επιτύχουμε ένα ικανοποιητικό ποσοστό γενίκευσης ώστε το σύστημα να είναι επεκτάσιμο αλλά ακολουθώντας και τις απαιτήσεις που μας είχαν τεθεί. Η εφαρμογή ικανοποιεί όλες τις απαιτήσεις με κάποιες επιπρόσθετες λειτουργίες. Δόθηκε βάση στην ταχύτερη εκτέλεση των δοθέντων ερωτημάτων που επιτεύχθηκε με την απλούστευση αυτών πριν την εκτέλεση τους. Επίσης δώσαμε σημασία στην εμφάνιση κατανοητών από το χρήστη μηνυμάτων για κάθε περίπτωση λάθους ώστε να είναι πιο εύκολο για αυτόν να καταλάβει που βρίσκετε το πρόβλημα στο ερώτημα που έδωσε.

Ιδιαίτερα χρήσιμες φάνηκαν οι έτοιμες δομές της Java αν και υπήρχε πρόβλημα στο κατακερματισμό (hashing) κατά την εκτέλεση των πράξεων `UNION`, `INTERSECTION`, `MINUS`.
Στη συνέχει γίνεται μια ενδελεχής παρουσίαση των πακέτων και των κλάσεων που περιέχουν.

Το Πακέτο dbms.bplus

Το *Nidus DBMS* χειρίζεται πρωτεύοντες και δευτερεύοντες καταλόγους με τη χρήση των κλάσεων *PrimaryBPlusTree* και *SecondaryBPlusTree* αντιστοίχως. Κάθε μια απ' αυτές τις κλάσεις υλοποιεί ένα δένδρο B+.

Προτού αναλύσουμε θέματα υλοποίησης θα αναφέρουμε τις βασικότερες παραδοχές που χρησίμευσαν σαν οδηγοί κατά την ανάπτυξη του έργου.

Και το πρωτεύον και το δευτερεύον B+ βρίσκονται απ' την έναρξη μέχρι τον τερματισμό της εφαρμογής στην κύρια μνήμη και όχι σε κάποιο δίσκο. Το ίδιο δεν συμβαίνει όμως με τις εγγραφές του πίνακα. Αυτές βρίσκονται αποθηκευμένες στο δίσκο, οργανωμένες σε σελίδες. Έτσι λοιπόν πρέπει να υπάρξει μια λογική σύνδεση του δένδρου στη μνήμη με τις σελίδες στο δίσκο. Αυτή η σύνδεση πραγματοποιείται στα φύλλα του πρωτεύοντος B+, που δεν περιέχουν τίποτα άλλο παρά έναν αριθμό που δείχνει ποια σελίδα θα πρέπει να φορτώσει το σύστημα απ' το δίσκο για να βρει τη ζητούμενη εγγραφή.

Στο δευτερεύον B+ παρουσιάζεται το πρόβλημα της αντιστοίχησης κάθε κλειδιού με μια εγγραφή. Η λύση δόθηκε με τη δημιουργία μιας «οντότητας» που ονομάσαμε αναφορά. Μια αναφορά είναι μια τριάδα ακεραίων που προσδιορίζουν: το κλειδί της εγγραφής, τον αριθμό της σελίδας στην οποία είναι αποθηκευμένη και την μετατόπιση (με μέτρο το πλήθος των εγγραφών πριν απ' αυτήν) της εγγραφής απ' την αρχή της σελίδας. Τα φύλλα του δένδρου αυτού περιέχουν ένα πλήθος από αναφορές ταξινομημένες με βάση το κλειδί τους.

Σε αντίθεση με το πρωτεύον B+ όπου γνωρίζουμε ότι τα πρωτεύοντα κλειδιά είναι μοναδικά, στο δευτερεύον B+ ένα κλειδί μπορεί να υπάρξει περισσότερες από μια φορές. Το πρόβλημα δημιουργείται όταν το πλήθος των όμοιων κλειδιών γίνει τόσο μεγάλο που να μην μπορούν να εισαχθούν όλα στο ίδιο φύλλο. Η λύση ήταν να επιτρέψουμε να γίνεται εισαγωγή στο γειτονικό του φύλλο και να προσαρμόσουμε τα κλειδιά στον γονέα του φύλλου με τέτοιο τρόπο που να εξασφαλίζεται ότι τα κλειδιά με ίσες τιμές θα είναι συνεχόμενα.

Από άποψη υλοποίησης τώρα, οι κλάσεις *PrimaryBPlusTree* και *SecondaryBPlusTree* έχουν δομικές ομοιότητες αλλά και διαφορές. Ακολουθεί η ανάλυση των στοιχείων τους.

Ο απαριθμητός τύπος NodeType

Ο τύπος αυτός παίρνει τρεις τιμές: *INNER_NODE* για τους εσωτερικούς κόμβους, *PRIMARY_LEAF* για τα φύλλα του πρωτεύοντος B+ και *SECONDARY_LEAF* για τα φύλλα του δευτερεύοντος B+.

Η κλάση Node

Πρόκειται για μια αφηρημένη (abstract) κλάση που σκοπό της έχει να μας προσφέρει έναν κοινό τρόπο χειρισμού των αντικειμένων των κλάσεων που υλοποιούν τους εσωτερικούς κόμβους και τα φύλλα. Αυτό που πρέπει να θυμόμαστε είναι ότι κάθε υποκλάση της έχει ένα πεδίο που κρατά τον τύπο

του κόμβου (*NodeType*), ένα δείκτη προς τον γονέα του κόμβου, καθώς και μια μέθοδο που υποδεικνύει εάν ο κόμβος είναι πλήρης ή όχι.

Η κλάση InnerNode

Πρόκειται για μια υποκλάση της *Node* που υλοποιεί τους εσωτερικούς κόμβους τόσο του πρωτεύοντος όσο και του δευτερεύοντος B+. Κάθε εσωτερικός κόμβος μπορεί να έχει ένα πλήθος κλειδιών ίσο με *bucketFactor* και ένα πλήθος παιδιών ίσο με *bucketFactor* + *1*. Τα κλειδιά φυσικά είναι ταξινομημένα. Σημαντικότερη μέθοδος είναι η *insertKey(final int newKey, final Node child)* για την εισαγωγή του διδομένου ζεύγους κλειδιού – παιδιού στη σωστή θέση. Υπάρχουν και άλλες μέθοδοι εισαγωγής < *addLast(final int newKey, final Node child)* και *addFirst(final int key, final Node child)* > που χρησιμοποιούνται για ταχείες εισαγωγές στο τέλος και στην αρχή αντιστοίχως. Μια άλλη σημαντική μέθοδος είναι η *int findIndexOfChild(Node child)* που επιστρέφει τη θέση του διδόμενου παιδιού στον κόμβο.

Η κλάση PrimaryLeaf

Πρόκειται για μια υποκλάση της *Node* που υλοποιεί τα φύλλα του πρωτεύοντος δένδρου B+. Όπως προαναφέρθηκε η σημαντικότερη πληροφορία που κρατείται εδώ είναι ο αριθμός της σελίδας που κρατά τις εγγραφές που αντιστοιχούν στο φύλλο αυτό. Κάτι που πρέπει να παρατηρήσουμε είναι η ύπαρξη δεικτών προς τα γειτονικά φύλλα, γεγονός που μας επιτρέπει να σαρώσουμε σειριακά τα φύλλα του δένδρου.

Η κλάση PrimaryBPlusTree

Η κλάση αυτή υλοποιεί το πρωτεύον δένδρο B+. Το ποιο σημαντικό στοιχείο που πρέπει να γνωρίζουμε κατά την δημιουργία του είναι το *bucketFactor*, δηλαδή το μέγιστο πλήθος των στοιχείων που μπορεί να περιέχει κάθε κόμβος του. Πολύ σημαντικό είναι να πούμε ότι έχουμε κάνει την παραδοχή ότι στους εσωτερικούς κόμβους αριστερά του κάθε κλειδιού βρίσκεται το παιδί με τα μικρότερα κλειδιά και δεξιά του το παιδί με τα μεγαλύτερα ή το ίσο (σε περίπτωση που το παιδί είναι φύλλο). Αυτή η παραδοχή αντικατοπτρίζεται στη μέθοδο *PrimaryLeaf search(final int key)* που χρησιμοποιείται για να διασχίζουμε το δένδρο απ' τη ρίζα μέχρι τα φύλλα και να επιστρέφει το φύλλο που μας ενδιαφέρει. Οι σημαντικότερες μέθοδοι της κλάσης είναι οι ακόλουθες:

- ***Object[] insert(final Record newRecord)***: χρησιμοποιείται για την εισαγωγή μιας νέας εγγραφής στο δένδρο (στην πραγματικότητα στη σελίδα). Αρχικά βρίσκουμε σε ποιο φύλλο αντιστοιχεί η νέα

εγγραφή και στη συνέχεια φορτώνουμε τη σελίδα που αντιστοιχεί στο φύλλο αυτό. Εκτελούμε την εισαγωγή στη σελίδα και ελέγχουμε εάν έχει προκύψει υπερχείλιση. Στην τελευταία περίπτωση γίνεται διάσπαση, δηλαδή δημιουργείται μια νέα σελίδα που παίρνει τις μισές απ' τις εγγραφές και δημιουργείται και ένα νέο φύλλο στο δένδρο που αντιστοιχεί στη νέα σελίδα. Το δένδρο αναδομείται και ακολούθως οι σελίδες (ή η σελίδα, εάν δεν προέκυψε διάσπαση) ξαναγράφονται στο δίσκο. Πρέπει να πούμε ότι κατά τη λειτουργία της μεθόδου γίνεται συλλογή στοιχείων που αφορά την επιτυχία της εισαγωγής και την ενημέρωση των δευτερευόντων δένδρων. Η ανάγκη αυτή προκύπτει απ' το γεγονός ότι κατά την εισαγωγή μιας εγγραφής μπορεί να προκύψει μετατόπιση κάποιον άλλων, οπότε πρέπει να ενημερωθούν οι αναφορές των δευτερευόντων B+. Το ίδιο πρέπει να γίνει και κατά τη διάσπαση αφού μερικές εγγραφές μεταφέρονται από μια σελίδα σε μια άλλη.

- ***void reconstructTree(final PrimaryLeaf currentLeaf, final PrimaryLeaf newLeaf, final Page newPage)***: χρησιμοποιείται για την αναδόμηση του δένδρου. Λειτουργεί σε δυο φάσεις. Αρχικά εισάγει ένα ζεύγος κλειδιού – παιδιού στον κατάλληλο εσωτερικό κόμβο και στη συνέχεια (μέσω μιας άλλης μεθόδου της *`reconstructInnerNodes(int key, InnerNode currentNode, Node newNode)`*) αναδομεί τους εσωτερικούς κόμβους. Εξετάζει δηλαδή εάν ο εσωτερικός κόμβος υπερχείλισε, οπότε προκαλεί διάσπαση και δημιουργεί έναν νέο εσωτερικό κόμβο και μετά ενημερώνει τον γονέα του. Η διαδικασία αυτή επαναλαμβάνεται μέχρι τον κόμβο που δεν χρειάζεται να γίνουν διασπάσεις ή μέχρι τη ρίζα (κόμβος που δεν έχει γονέα).

- ***Object[] delete(final int primaryKey)***: χρησιμοποιείται για την διαγραφή ενός πρωτεύοντος κλειδιού. Εκμεταλλευόμενοι την μοναδικότητα του πρωτεύοντος κλειδιού αναζητούμε το φύλλο που αναφέρεται σε αυτό το κλειδί, φορτώνουμε την αντίστοιχη σελίδα και σαρώνουμε τις εγγραφές μέχρι να βρεθεί αυτή που θα διαγραφεί. Μόλις πραγματοποιηθεί η διαγραφή ξαναγράφουμε τη σελίδα στο δίσκο. Όπως και κατά την εισαγωγή κατά τη λειτουργία της μεθόδου γίνεται συλλογή στοιχείων που αφορά την επιτυχία της εισαγωγής και την ενημέρωση των δευτερευόντων δένδρων.

- ***Object[] deleteAllGreaterOrEqual(final int primaryKey))***:
  χρησιμοποιείται για την διαγραφή όλων των πρωτευόντων κλειδιών με τιμή μεγαλύτερη ή ίση με αυτή που δόθηκε. Και πάλι εκμεταλλευόμενοι την μοναδικότητα του πρωτεύοντος κλειδιού αναζητούμε το φύλλο που αναφέρεται σε αυτό το κλειδί, φορτώνουμε την αντίστοιχη σελίδα και σαρώνουμε τις εγγραφές μέχρι να βρεθεί η πρώτη που θα διαγραφεί. Στη συνέχεια διαγράφουμε αυτήν και όσες βρίσκονται δεξιά απ' αυτήν. Ξαναγράφουμε τη σελίδα στο δίσκο και μετά διαγράφουμε όλες τις εγγραφές σε όλες τις σελίδες που βρίσκονται δεξιά της προηγουμένης. Όπως και προηγουμένως, κατά τη λειτουργία της

μεθόδου γίνεται συλλογή στοιχείων που αφορά την επιτυχία της διαγραφής και την ενημέρωση των δευτερευόντων δένδρων.

- ***Object[] deleteAllLessOrEqual(final int primaryKey)***: χρησιμοποιείται για την διαγραφή όλων των πρωτευόντων κλειδιών με τιμή μικρότερη ή ίση με αυτή που δόθηκε. Και πάλι εκμεταλλευόμενοι την μοναδικότητα του πρωτεύοντος κλειδιού αναζητούμε το φύλλο που αναφέρεται σε αυτό το κλειδί, φορτώνουμε την αντίστοιχη σελίδα και σαρώνουμε τις εγγραφές (από πίσω προς τα εμπρός) μέχρι να βρεθεί η πρώτη που θα διαγραφεί. Στη συνέχεια διαγράφουμε αυτήν και όσες βρίσκονται αριστερά απ' αυτήν. Ξαναγράφουμε τη σελίδα στο δίσκο και μετά διαγράφουμε όλες τις εγγραφές σε όλες τις σελίδες που βρίσκονται αριστερά της προηγουμένης. Όπως και προηγουμένως, κατά τη λειτουργία της μεθόδου γίνεται συλλογή στοιχείων που αφορά την επιτυχία της διαγραφής και την ενημέρωση των δευτερευόντων δένδρων.

- ***Object[] delete(final LinkedList<Reference> referenceList)***: χρησιμοποιείται για την διαγραφή πολλών εγγραφών μετά από διαγραφή σε δευτερεύοντα δένδρο. Αρχικά ταξινομούνται οι αναφορές με βάση τους αριθμούς σελίδων τους και τη θέση μέσα στη σελίδα. Μετά παίρνουμε όλες τις αναφορές μια - μια και φορτώνουμε την αντίστοιχη σελίδα, βρίσκουμε την εγγραφή με την αντίστοιχη μετατόπιση και τη διαγράφουμε. Εάν υπάρχουν αναφορές για την ίδια σελίδα τις διαγράφουμε και αυτές. Διαφορετικά φορτώνουμε την επόμενη σελίδα που υποδεικνύει η αναφορά και διαγράφουμε με τον ίδιο τρόπο. Και πάλι κατά τη λειτουργία της μεθόδου γίνεται συλλογή στοιχείων που αφορά την επιτυχία της διαγραφής και την ενημέρωση των δευτερευόντων δένδρων (πλην αυτού που εκκίνησε τη διαγραφή).

- ***Object[] delete(final int secondaryKey, final int keyIndex)***: χρησιμοποιείται για τη σειριακή διαγραφή εγγραφών με βάση ένα δευτερεύον κλειδί. Εδώ εκμεταλλευόμαστε τη σύνδεση των φύλλων μεταξύ τους. Ξεκινώντας λοιπόν από το πρώτο φύλλο (η κλάση μας προσφέρει τέτοια μέθοδο), φορτώνουμε τη σελίδα που αντιστοιχεί στο φύλλο αυτό και σαρώνουμε τις εγγραφές διαγράφοντας όσες έχουν το αντίστοιχο δευτερεύον κλειδί. Όταν σαρώσουμε όλη τη σελίδα, την ξαναγράφουμε στο δίσκο και προχωρούμε στο επόμενο φύλλο. Επαναλαμβάνουμε τη διαδικασία μέχρι να φτάσουμε στο τελευταίο φύλλο (αυτό που δεν έχει δεξιό γείτονα). Όπως και προηγουμένως, κατά τη λειτουργία της μεθόδου γίνεται συλλογή στοιχείων που αφορά την επιτυχία της διαγραφής και την ενημέρωση των δευτερευόντων δένδρων.

- ***Object[] deleteAllGreater(final int secondaryKey, final int keyIndex)***: χρησιμοποιείται για τη σειριακή διαγραφή εγγραφών που έχουν τιμή μεγαλύτερη από το διδόμενο δευτερεύον κλειδί. Και εδώ εκμεταλλευόμαστε τη σύνδεση των φύλλων μεταξύ τους.

Ξεκινώντας λοιπόν από το πρώτο φύλλο, φορτώνουμε τη σελίδα που αντιστοιχεί στο φύλλο αυτό και σαρώνουμε τις εγγραφές διαγράφοντας όσες έχουν τιμή μεγαλύτερη απ' το αντίστοιχο δευτερεύον κλειδί. Όταν σαρώσουμε όλη τη σελίδα, την ξαναγράφουμε στο δίσκο και προχωρούμε στο επόμενο φύλλο. Επαναλαμβάνουμε τη διαδικασία μέχρι να φτάσουμε στο τελευταίο φύλλο (αυτό που δεν έχει δεξιό γείτονα). Όπως και προηγουμένως, κατά τη λειτουργία της μεθόδου γίνεται συλλογή στοιχείων που αφορά την επιτυχία της διαγραφής και την ενημέρωση των δευτερευόντων δένδρων.

- ***Object[] deleteAllLess(final int secondaryKey, final int keyIndex)***: χρησιμοποιείται για τη σειριακή διαγραφή εγγραφών που έχουν τιμή μικρότερη από το διδόμενο δευτερεύον κλειδί. Και εδώ εκμεταλλευόμαστε τη σύνδεση των φύλλων μεταξύ τους. Ξεκινώντας λοιπόν από το πρώτο φύλλο, φορτώνουμε τη σελίδα που αντιστοιχεί στο φύλλο αυτό και σαρώνουμε τις εγγραφές διαγράφοντας όσες έχουν τιμή μικρότερη απ' το αντίστοιχο δευτερεύον κλειδί. Όταν σαρώσουμε όλη τη σελίδα, την ξαναγράφουμε στο δίσκο και προχωρούμε στο επόμενο φύλλο. Επαναλαμβάνουμε τη διαδικασία μέχρι να φτάσουμε στο τελευταίο φύλλο (αυτό που δεν έχει δεξιό γείτονα). Όπως και προηγουμένως, κατά τη λειτουργία της μεθόδου γίνεται συλλογή στοιχείων που αφορά την επιτυχία της διαγραφής και την ενημέρωση των δευτερευόντων δένδρων.

- ***Object[] delete(final String key, final int keyIndex)***: χρησιμοποιείται για τη σειριακή διαγραφή εγγραφών που έχουν τιμή μιας συμβολοσειράς ίση με τη διδόμενη συμβολοσειρά. Και εδώ εκμεταλλευόμαστε τη σύνδεση των φύλλων μεταξύ τους. Ξεκινώντας λοιπόν από το πρώτο φύλλο, φορτώνουμε τη σελίδα που αντιστοιχεί στο φύλλο αυτό και σαρώνουμε τις εγγραφές διαγράφοντας όσες έχουν τιμή συμβολοσειράς ίση με τη διδόμενη συμβολοσειρά. Όταν σαρώσουμε όλη τη σελίδα, την ξαναγράφουμε στο δίσκο και προχωρούμε στο επόμενο φύλλο. Επαναλαμβάνουμε τη διαδικασία μέχρι να φτάσουμε στο τελευταίο φύλλο (αυτό που δεν έχει δεξιό γείτονα). Όπως και προηγουμένως, κατά τη λειτουργία της μεθόδου γίνεται συλλογή στοιχείων που αφορά την επιτυχία της διαγραφής και την ενημέρωση των δευτερευόντων δένδρων.

Η κλάση προσφέρει μια σειρά από μεθόδους *`find` που ακολουθούν ακριβώς την ίδια λογική με τις διάφορες μορφές της `delete`*. Η θεμελιώδης διαφορά τους είναι ότι δεν αλλάζουν τις σελίδες αλλά μόνο τις διαβάζουν για να βρουν τις εγγραφές που τις ικανοποιούν. Συνεπώς καμιά σελίδα στο δίσκο δεν ενημερώνεται, ούτε συλλέγονται στοιχεία για ενημέρωση των δευτερευόντων δένδρων. Επιστρέφουν όμως τιμές ενδεικτικές της επιτυχίας της εύρεσης και μια λίστα με όλες τις εγγραφές που τις ικανοποιούν. Επιπλέον προσφέρουν τις μεθόδους `Object[] findAllInRange(final int firstPrimaryKey, final int lastPrimaryKey)` και `Object[]`

findAllInRange(final int firstSecondaryKey, final int lastSecondaryKey, final int keyIndex) που λειτουργούν όπως οι *Object[] findAllGreaterOrEqual(final int primaryKey)* και *Object[] findAllGreater(final int secondaryKey, final int keyIndex)* αντιστοίχως, με τη διαφορά ότι πλέον ο έλεγχος της τιμής γίνεται σε ένα διάστημα.

Η κλάση Reference

Η κλάση αυτή υλοποιεί την οντότητα που ονομάσαμε αναφορά. Επαναλαμβάνουμε ότι μια αναφορά είναι ουσιαστικά μια τριάδα (θετικών) ακεραίων. Ο πρώτος ακέραιος αναπαριστά το κλειδί της εγγραφής που μας ενδιαφέρει, ο δεύτερος τον αριθμό της σελίδας στην οποία είναι αποθηκευμένη η εγγραφή και ο τρίτος την μετατόπιση (με μέτρο το πλήθος των εγγραφών πριν απ' αυτήν) της εγγραφής απ' την αρχή της σελίδας. Στην τριάδα αυτή προσθέσαμε και μια δίτιμη (boolean) μεταβλητή που υποδεικνύει εάν η αναφορά είναι «ζωντανή» ή «νεκρή». Εάν είναι «ζωντανή», τότε η αναφορά είναι έγκυρη, διαφορετικά είναι άκυρη οπότε μπορεί να αντικατασταθεί από μια άλλη.

Η κλάση SecondaryLeaf

Πρόκειται για την τρίτη και τελευταία υποκλάση της *Node* που υλοποιεί τα φύλλα του δευτερεύοντος δένδρου B+. Κάθε (δευτερεύον) φύλλο μπορεί να έχει ένα πλήθος αναφορών ίσο με *bucketFactor*. Είναι σημαντικό να γνωρίζουμε ότι όπως και στα φύλλα του πρωτεύοντος δένδρου, υπάρχουν δείκτες προς τα γειτονικά φύλλα, γεγονός που μας επιτρέπει να σαρώσουμε σειριακά τα φύλλα του δένδρου.

Η σημαντικότερη μέθοδος της κλάσης αυτής είναι η *SecondaryLeaf insertReference(final Reference newReference)*. Δεν ακολουθείται η συνήθης τακτική εισαγωγής, όπου το στοιχείο με το μεγαλύτερο κλειδί εισάγεται τελευταίο. Εδώ εάν ένα στοιχείο (πιο συγκεκριμένα μια αναφορά) έχει κλειδί μεγαλύτερο (ή ίσο) από όλα τα άλλα και το φύλλο είναι πλήρως συμπληρωμένο (δεν περιέχει δηλαδή νεκρές εγγραφές), τότε η εισαγωγή θα πραγματοποιηθεί στο δεξιό γείτονα του φύλλου αυτού (εάν υπάρχει αλλιώς θα τοποθετηθεί στο τέλος προκαλώντας έτσι υπερχείλιση). Η σκοπιμότητα είναι να μπορέσουμε έτσι να υποστηρίξουμε ένα πλήθος ίσων κλειδιών που ξεπερνά το *bucketFactor*. Έτσι λοιπόν ίσα κλειδιά μπορεί να μοιράζονται σε γειτονικά φύλλα. Η παρενέργεια που υπάρχει είναι ότι κάτω από κάποιες προϋποθέσεις μπορεί να δημιουργηθούν κάποια φύλλα που θα δέχονται αναφορές με ένα συγκεκριμένο μόνο κλειδί. Αυτό φυσικά μπορεί να σημαίνει σπατάλη χώρου, αλλά είναι το αντάλλαγμα που δεχθήκαμε για την επίλυση του προβλήματος μας. Για το λόγο αυτό οι μέθοδοι που πραγματοποιούν τις διαγραφές στα δευτερεύοντα φύλλα εξετάζουν την περίπτωση να χρειαστεί να ερευνήσουν και τα γειτονικά φύλλα.

Σημαντική επίσης είναι και η μέθοδος *deleteReference(final int index)* που διαγράφει μια αναφορά και εν συνεχεία την μετακινεί στο τέλος του φύλλου για να διατηρηθούν ταξινομημένες οι «ζωντανές» αναφορές.

Η κλάση SecondaryBPlusTree

Η κλάση αυτή υλοποιεί το δευτερεύον δένδρο B+. Όπως και το πρωτεύον B+, πρέπει να γνωρίζουμε το *bucketFactor*, δηλαδή το μέγιστο πλήθος των στοιχείων που μπορεί να περιέχει κάθε κόμβος του. Σημαντικό είναι να πούμε ότι δομείται λαμβάνοντας υπόψη τις υπάρχουσες εγγραφές στο πρωτεύον δένδρο. Έτσι λοιπόν το δένδρο αυτό δομείται αντίστροφα. Σαρώνουμε μια – μια τις εγγραφές και για κάθε μια απ' αυτές δημιουργούμε μια αναφορά (βλ. κλάση *Reference* και τη μέθοδο *LinkedList<Reference> createReferences(final int keyIndex, final PrimaryBPlusTree primaryTree)*). Στη συνέχεια οι αναφορές ταξινομούνται με βάση το κλειδί τους και εισάγονται σε φύλλα. Ακολούθως δημιουργούνται εσωτερικοί κόμβοι που γεμίζουν με κλειδιά απ' τα φύλλα, προκαλώντας την αναδόμηση του δένδρου όπου χρειάζεται. Όταν το τελευταίο φύλλο μπει στο δένδρο η δόμηση έχει τελειώσει. Επιπλέον θα πραγματοποιηθεί μόνο ένας έλεγχος για το πλήθος των κλειδιών που έχει ο εσωτερικός κόμβος που συμπληρώθηκε τελευταίος. Εάν το πλήθος αυτό είναι μικρότερο από *bucketFactorDiv2*, τότε ο κόμβος αυτός θα πάρει όσα κλειδιά χρειάζεται από τον αριστερό γείτονα του (που είναι συμπληρωμένος με *bucketFactor* πλήθος κλειδιών).

Σε αντίθεση με ότι ίσχυε στο πρωτεύον δένδρο, στο δευτερεύον δένδρο έχουμε κάνει την παραδοχή ότι αριστερά του κάθε κλειδιού βρίσκεται το παιδί με τα μικρότερα ή ίσα κλειδιά και δεξιά του το παιδί με τα μεγαλύτερα. Η παραδοχή αυτή σχετίζεται με τη λύση που δόθηκε στο πρόβλημα της ύπαρξης πολλαπλών ίσων κλειδιών που μπορεί να υπερχειλίσουν τον κόμβο. Αποφασίσαμε λοιπόν να προσπαθούμε να εισάγουμε το κλειδί - αναφορά στο φύλλο που υπάρχουν τα μικρότερα ή ίσα κλειδιά (βλ. τη μέθοδο *SecondaryLeaf insertReference(final Reference newReference)* της κλάσης *SecondaryLeaf*).

Αυτή η παραδοχή αντικατοπτρίζεται στη μέθοδο *SecondaryLeaf search(final int key)* που χρησιμοποιείται για να διασχίζουμε το δένδρο απ' τη ρίζα μέχρι τα φύλλα και να επιστρέφει το φύλλο που μας ενδιαφέρει. Οι σημαντικότερες μέθοδοι της κλάσης είναι οι ακόλουθες:

- ***LinkedList<Reference> createReferences(final int keyIndex, final PrimaryBPlusTree primaryTree)***: χρησιμοποιείται για τη δημιουργία των αναφορών που βρίσκονται στα φύλλα. Η διαδικασία είναι πολύ απλή. Σαρώνουμε τα ένα – ένα τα φύλλα του πρωτεύοντος δένδρου και φορτώνουμε στη μνήμη την αντίστοιχη σελίδα. Για κάθε εγγραφή δημιουργούμε και μια αναφορά. Οι αναφορές συγκεντρώνονται σε μια λίστα που θα χρησιμοποιηθεί αργότερα.

- ***void insert(final Reference reference)***: χρησιμοποιείται για την εισαγωγή μιας νέας αναφοράς στο δευτερεύον B+. Αρχικά

βρίσκουμε σε ποιο φύλλο αντιστοιχεί η νέα αναφορά, στη συνέχεια εκτελούμε την εισαγωγή στο φύλλο και ελέγχουμε εάν έχει προκύψει υπερχείλιση. Στην τελευταία περίπτωση γίνεται διάσπαση, δηλαδή δημιουργείται ένα νέο φύλλο που παίρνει τις μισές απ' τις αναφορές και το δένδρο αναδομείται.

- ***Object[] delete(final int secondaryKey)***: χρησιμοποιείται για τη διαγραφή ενός δευτερεύοντος κλειδιού. Επειδή τα κλειδιά δεν είναι μοναδικά μπορεί να διαγραφούν περισσότερες της μιας αναφορές. Αρχικά βρίσκουμε σε ποιο φύλλο βρίσκεται η αναφορά και στη συνέχεια σαρώνουμε τις αναφορές διαγράφοντας όσες έχουν το ίδιο κλειδί μέχρι να βρούμε μια που έχει κλειδί μεγαλύτερο. Εάν δεν βρεθεί ένα μεγαλύτερο κλειδί πριν φτάσουμε στο τέλος του φύλλου, τότε συνεχίζουμε να σαρώνουμε τις αναφορές και του δεξιού του γείτονα.

- ***Object[] deleteAllGreaterOrEqual(final int secondaryKey)***: χρησιμοποιείται για τη διαγραφή όλων των δευτερευόντων κλειδιών με τιμή μεγαλύτερη ή ίση απ' αυτή που δόθηκε. Βρίσκουμε σε ποιο φύλλο βρίσκεται η αναφορά και σαρώνουμε τις αναφορές μέχρι να βρούμε αυτήν που έχει το ζητούμενο κλειδί. Στη συνέχεια διαγράφουμε αυτήν και όσες βρίσκονται δεξιά απ' αυτήν. Ακολούθως διαγράφουμε όλα τα φύλλα που βρίσκονται δεξιά του φύλλου. Κατά τη λειτουργία της μεθόδου γίνεται συλλογή στοιχείων που αφορά την επιτυχία της διαγραφής και τις διαγραφές που θα ακολουθήσουν στο πρωτεύον δένδρο.

- ***Object[] deleteAllLessOrEqual(final int secondaryKey)***: χρησιμοποιείται για τη διαγραφή όλων των δευτερευόντων κλειδιών με τιμή μικρότερη ή ίση απ' αυτή που δόθηκε. Βρίσκουμε σε ποιο φύλλο βρίσκεται η αναφορά και σαρώνουμε τις αναφορές μέχρι να βρούμε αυτήν που έχει το ζητούμενο κλειδί. Στη συνέχεια διαγράφουμε αυτήν και όσες βρίσκονται αριστερά απ' αυτήν. Ακολούθως διαγράφουμε όλα τα φύλλα που βρίσκονται αριστερά του φύλλου. Κατά τη λειτουργία της μεθόδου γίνεται συλλογή στοιχείων που αφορά την επιτυχία της διαγραφής και τις διαγραφές που θα ακολουθήσουν στο πρωτεύον δένδρο.

- ***void delete(final LinkedList<Object[]> deleted)***: χρησιμοποιείται για την διαγραφή πολλών αναφορών μετά από διαγραφή στο πρωτεύον δένδρο. Με βάση τα κλειδιά των εγγραφών (που αντιστοιχούν σ' αυτό το δένδρο) που διαγράφηκαν αναζητούμε τις αντίστοιχες αναφορές και τις διαγράφουμε. Πιο συγκεκριμένα βρίσκουμε σε ποιο φύλλο βρίσκεται η αναφορά και σαρώνουμε τις αναφορές διαγράφοντας αυτήν που δείχνει στη σωστή θέση (η θέση προσδιορίζεται μοναδικά από τον αριθμό σελίδας και τη μετατόπιση μέσα στη σελίδα). Εάν χρειαστεί επεκτείνουμε την αναζήτηση στο γειτονικό φύλλο.

- ***updateReferences (final LinkedList <Record> recordsMoved, final LinkedList<Reference> oldReferenceList, final LinkedList<Reference> newReferenceList)***: χρησιμοποιείται για την ενημέρωση των αναφορών των δευτερευόντων δένδρων. Μετά από εισαγωγή ή διαγραφή, οι εγγραφές μέσα στις σελίδες μετακινούνται, οπότε προκύπτει η ανάγκη να ενημερωθούν οι αντίστοιχες αναφορές. Πιο συγκεκριμένα βρίσκουμε σε ποιο φύλλο βρίσκεται η αναφορά και σαρώνουμε τις αναφορές μέχρι να βρούμε αυτήν που δείχνει στην παλαιά θέση. Ακολούθως την τροποποιούμε ώστε να δείχνει στη νέα θέση της εγγραφής.

Όπως και η κλάση `PrimaryBPlusTree`, η κλάση αυτή προσφέρει μια σειρά από μεθόδους `find` *που ακολουθούν ακριβώς την ίδια λογική με τις διάφορες μορφές της* `delete`. Επιπλέον προσφέρει τη μέθοδο `Object[] findAllInRange(final int firstSecondaryKey, final int lastSecondaryKey)`, με τη διαφορά ότι πλέον ο έλεγχος της τιμής γίνεται σε ένα διάστημα. Η μέθοδος αυτή λειτουργεί συμπληρωματικά με τη μέθοδο `Object[] find(final LinkedList<Reference> referenceList)`.

Συμπληρωματικά με τις κλάσεις των δένδρων δρουν οι κλάσεις `Record` και `Page`. Αυτές θα ήταν «ενσωματωμένες» στο πρωτεύον δένδρο εάν δεν είχαμε αποφασίσει να αποθηκεύουμε τα δεδομένα στο δίσκο.

Το Πακέτο dbms.util

Το πακέτο αυτό περιέχει κλάσεις ευρέως χρησιμοποιούμενες από τα υπόλοιπα υποσυστήματα της εφαρμογής. Η λειτουργία του είναι να δρα συμπληρωματικά με τα συστήματα αυτά και να προσφέρει βοηθητικές λειτουργίες καθώς και δομές δεδομένων.
Οι κλάσεις του πακέτου αυτού περιγράφονται παρακάτω.

Η κλάση Record

Η κλάση αυτή αποθηκεύει τα δεδομένα μιας γραμμής του πίνακα. Εκτός από τα δεδομένα των πεδίων – στηλών του πίνακα κρατά, μια δίτιμη (boolean) μεταβλητή που υποδεικνύει εάν η εγγραφή είναι «ζωντανή» ή «νεκρή» και κάποια άλλα στοιχεία που χρησιμεύουν στον ευκολότερο χειρισμό των εγγραφών. Όπως προείπαμε οι εγγραφές βρίσκονται γραμμένες στο δίσκο. Έτσι λοιπόν η σημαντικότερες μέθοδοι είναι οι εξής δυο:

- ***Record(final ByteBuffer recordBuffer)***: πρόκειται για έναν δομητή που δέχεται μια σειρά από byte όπως προέρχονται από το δίσκο και από αυτά δημιουργεί μια εγγραφή. Για να γίνει βέβαια αυτό απαραίτητη είναι η γνώση του τύπου του κάθε πεδίου – στήλης της

εγγραφής, αφού διαφορετικό πλήθος byte αποτελούν έναν INTEGER και διαφορετικό πλήθος έναν CHAR.

- ***byte[] toByteArray()***: πρόκειται ουσιαστικά για την αντίστροφη της παραπάνω μεθόδου. Εδώ παίρνουμε μια εγγραφή και την μετατρέπουμε σε μια σειρά από byte που θα εγγραφούν στο δίσκο. Και σ' αυτήν την περίπτωση είναι απαραίτητη η γνώση του τύπου του κάθε πεδίου – στήλης της εγγραφής.

Η κλάση Page

Η κλάση αυτή χρησιμεύει ως ένα σύνολο οργάνωσης των εγγραφών. Κάθε σελίδα αποτελείται από `bucketFactor` πλήθος εγγραφών (είτε «νεκρών» είτε «ζωντανών»). Αυτό που φορτώνεται από το δίσκο στη μνήμη και αυτό που γράφεται στο δίσκο από τη μνήμη είναι μια σελίδα. Και εδώ υπάρχουν επιπλέον δεδομένα που αφορούν τον ευκολότερο χειρισμό της σελίδας. Οι κυριότερες μέθοδοι της κλάσης είναι:

- ***Page(final int bucketFactor, final ByteBuffer buffer, final int recordSize)***: χρησιμοποιείται για τη δημιουργία μιας σελίδας από μια σειρά από byte. Η δημιουργία της σελίδας συνίσταται στη δημιουργία των εγγραφών που την αποτελούν (με τη κλήση της αντίστοιχης μεθόδου των εγγραφών) και την απόδοση των σωστών τιμών στα επιπλέον πεδία της κλάσης.

- ***byte[] toByteArray()***: πρόκειται ουσιαστικά για την αντίστροφη της παραπάνω μεθόδου. Εδώ παίρνουμε μια σελίδα και την μετατρέπουμε σε μια σειρά από byte που θα εγγραφούν στο δίσκο. Πρώτα μετατρέπονται οι εγγραφές σε μια σειρά από byte (με τη κλήση της αντίστοιχης μεθόδου των εγγραφών) και στη συνέχεια τα επιπλέον πεδία.

- ***Object[] insertRecord(final Record newRecord)***: χρησιμοποιείται για την εισαγωγή μιας νέας εγγραφής στη σελίδα. Η νέα εγγραφή είτε θα πάρει τη θέση της πρώτης νεκρής που θα βρεθεί είτε θα εισαχθεί πριν από αυτήν που έχει το αμέσως μεγαλύτερο πρωτεύον κλειδί, είτε θα εισαχθεί στο τέλος της σελίδας. Στη δεύτερη περίπτωση για να ελέγχουμε εάν η τελευταία εγγραφή είναι «νεκρή». Εάν είναι την αφαιρούμε και έτσι δεν προκύπτει υπερχείλιση. Στην τρίτη περίπτωση είναι σίγουρο ότι θα προκύψει υπερχείλιση. Η μέθοδος επιστρέφει μια τιμή επιτυχίας ή αποτυχίας της εισαγωγής (το πρωτεύον κλειδί είναι μοναδικό και δεν επιτρέπεται η νέα εγγραφή να έχει το ίδιο κλειδί με μια παλαιά) και τη θέση εισαγωγής ως μετατόπιση από την αρχή της σελίδας.

- ***void deleteRecord(final int index)***: χρησιμοποιείται για την διαγραφή μιας εγγραφής και εν συνεχεία την μετακινεί στο τέλος του φύλλου για να διατηρηθούν ταξινομημένες οι «ζωντανές» εγγραφές.

Η κλάση Char

Η κλάση αυτή χρησιμοποιείται για την υλοποίηση του τύπου CHAR. Ο τύπος αυτός αντιστοιχεί σε μια σειρά από 50 χαρακτήρες. Εάν προσπαθήσουμε να δώσουμε περισσότερους από 49 χαρακτήρες (ο τελευταίος υποχρεωτικά είναι ο χαρακτήρας τερματισμού) θα προκύψει εξαίρεση του τύπου *EnormousCharacterException*. Επειδή είναι στοιχείο των εγγραφών υποστηρίζει μια μέθοδο μετατροπής σε μια σειρά από byte, αλλά και την αντίστροφη μέθοδο δημιουργίας από μια σειρά από byte. Επίσης υποστηρίζει τη μέθοδο *String toString()* που επιστρέφει ένα αντικείμενο της κλάσης *String* που αποτελείται μόνο από τους έγκυρους χαρακτήρες.

Η κλάση DiskHandler

Η κλάση αυτή υλοποιεί τις μεθόδους φορτώματος και εγγραφής των σελίδων στο δίσκο, αλλά και τις μεθόδους φορτώματος και εγγραφής ενός ολόκληρου πίνακα απ' το δίσκο. Πρέπει αν πούμε ότι για κάθε πίνακα δημιουργείται μια μοναδική δυάδα αρχείων.

Το πρώτο αρχείο (<όνομα πίνακα>.pgs)είναι αυτό που περιέχει όλες τις σελίδες του πίνακα. Εκεί σε μια συγκεκριμένη μετατόπιση από την αρχή του αρχείου εγγράφονται οι αντίστοιχες σελίδες (η μετατόπιση εξαρτάται από το μέγεθος της σελίδας). Έτσι λοιπόν η σελίδα 5, όταν το μέγεθος της σελίδας είναι 128 byte, θα έχει μετατόπιση 740 byte απ' την αρχή του αρχείου. Σε μορφή byte η ίδια σελίδα εκτείνεται από το 740^{ο} byte μέχρι και το 867^{ο} byte.

Το δεύτερο (<όνομα πίνακα>.tbl) είναι ένα αρχείο από το οποίο διαβάζουμε τον πίνακα κατά την έναρξη της εφαρμογής και στο οποίο τον ξαναγράφουμε όταν τερματίζεται η εφαρμογή για να αποθηκευτούν οι πιθανές αλλαγές επ' αυτού.

Η κλάση Table

Η κλάση αυτή χρησιμοποιείται για την εσωτερική αναπαράσταση των πινάκων του `Nidus DBMS`.Τα βασικότερα πεδία της είναι:

- *name* – Το όνομα του πίνακα.
- *columnNames* – Τα ονόματα των στηλών,
- *columnTypes* – Οι τύποι των στηλών
- *primaryIndex* – Ο πρωτεύων κατάλογος.
- *secondaryIndexes* - Οι δευτερεύοντες κατάλογοι.

Οι βασικότερες μέθοδοι τις κλάσης αυτή είναι οι εξής:

- ***void deleteRecords(Char value, int columnIndex)***: διαγράφει όλες τις εγγραφές που έχουν στην αντίστοιχη στήλη την τιμή *value*.

- ***void deleteRecords(int columnIndex, SQLTokens operator, Integer value)***: διαγράφει όλες τις εγγραφές τις αντίστοιχης στήλης που έχουν τιμή μεγαλύτερη, μικρότερη ή ίση ανάλογα με τον *operator* από την *value*.

- ***LinkedList findRecords(Char value, int columnIndex)***: βρίσκει όλες τις εγγραφές που έχουν στην αντίστοιχη στήλη την τιμή *value* και τις επιστρέφει.

- ***LinkedList findRecords(int columnIndex, SQLTokens operator, Integer value)***: βρίσκει όλες τις εγγραφές τις αντίστοιχης στήλης που έχουν τιμή μεγαλύτερη, μικρότερη ή ίση ανάλογα με τον *operator* από την *value* και τις επιστρέφει.

- ***LinkedList findRange(int columnIndex, Integer lowValue, Integer highValue)***: βρίσκει όλες τις εγγραφές τις αντίστοιχης στήλης που έχουν τιμή μεγαλύτερη από την *lowValue* και μικρότερη από την *highValue* και τις επιστρέφει.

- ***LinkedList getAll()***: βρίσκει όλες τις εγγραφές τις αντίστοιχης στήλης και τις επιστρέφει.

- ***boolean insertRecord(Record newRecord)***: εισάγει την νέα εγγραφή τον πίνακα ελέγχοντας αν υπάρχει ήδη εγγραφή με το ίδιο πρωτεύων κλειδί.

- ***void addSecondaryIndex(int columnIndex, SecondaryBPlus newIndex)***: εισάγει τον δευτερεύον κατάλογο στην κατάλληλη στήλη.

- ***boolean removeSecondaryIndex(String indexName)***: αφαιρεί το δευτερεύον κατάλογο με όνομα *indexName* από τον πίνακα.

- ***boolean hasIndex(int columnIindex)***: ελέγχει αν η στήλη έχει κάποιον κατάλογο από πάνω της.

Το Πακέτο dbms.parser

Το *Nidus DBMS* χρησιμοποιεί έναν Parser για να αναγνωρίζει ένα υποσύνολο της SQL–92. Ο Parser αφού εντοπίσει την πρώτη εντολή(στην περίπτωση του αρχείου), ελέγχει για την συντακτική ορθότητα της εντολής αυτής και μετά καλεί την αντίστοιχη μέθοδο της μηχανής εκτέλεσης ώστε να εκτελεσθεί η αναγνωρισθείσα εντολή. Πρέπει να επισημάνουμε πως οι εντολές μπορούν να εισαχθούν μία-μία από τον χρήστη μέσω της κονσόλας ή να γίνει ταυτόχρονη εισαγωγή πολλών εντολών, οι οποίες θα εκτελεστούν σειριακά, με εισαγωγή τους από ένα αρχείο (*τυπικά input.txt*).

Οι εντολές που υποστηρίζονται είναι οι εξής:

Οι εντολές που ακολουθούν είναι στην τυπική τους μορφή. Πρέπει να σημειωθεί ότι όλοι οι απαραίτητοι σημασιολογικοί έλεγχοι γίνονται από την μηχανή εκτέλεσης.

- CREATE TABLE *Name* (*id_1 $type_1$, id_2 $type_2$, …,* PRIMARY KEY (*id_x*));

Δημιουργείται ο πίνακας με όνομα *Name*, στήλες *id_1, id_2,…* και σαν πρωτεύων κλειδί η στήλη *id_x* (όπου το *id_x* πρέπει να είναι μια από της *id_1, id_2,…* και το *type* πρέπει να είναι INTEGER).

σημ. Το *type* των στηλών μπορεί να είναι είτε INTEGER είτε CHAR.

- INSERT INTO *Name* VALUES (*$value_1$, $value_2$, …*);

Εισάγεται μια νέα εγγραφή με τιμές *$value_1$, $value_2$, …* στον πίνακα με όνομα *Name.* Οι τιμές πρέπει να είναι ίδιου τύπου συμφώνα με τον ορισμό του πίνακα όταν δημιουργήθηκε.

- DELETE FROM *Name* WHERE *condition;*

Διαγραφή όλων των εγγραφών του πίνακα με όνομα *Name* που ικανοποιούν την συνθήκη *condition.* Η συνθήκη αυτή τυπικά αποτελείται από ένα όνομα στήλης (*id*) ένα τελεστή(*operator*) και μία τιμή(*value*). Αν η στήλη είναι τύπου CHAR υποστηρίζεται μόνο ο τελεστής ισότητας (=) ενώ αν είναι τύπου INTEGER υποστηρίζονται όλοι οι γνωστοί τελεστές εκτός από τον τελεστή διάφορο(!=).

- SELECT *$column\_id_1$,$column\_id_2$, …*
  FROM *$table\_name_1$, $table\_name_2$,…*
  [WHERE *condition* | *$subcondition_1$* (AND | OR) *$subcondition_2$*]
  [ORDER BY *$column\_id_x$*]

Επιλογή και εμφάνιση αποτελεσμάτων από συγκεκριμένες στήλες και πίνακες. Το τμήμα του WHERE όπως και του ORDER BY είναι προαιρετικά. Σε περίπτωση που υπάρχει WHERE περιορίζει τα αποτελέσματα συμφωνά με την συνθήκη ή τις συνθήκες ενώ το ORDER BY ταξινομεί τα αποτελέσματα, με βάσει την στήλη που επιλέχθηκε, σε αύξουσα σειρά.
Περιορισμοί:

1. Στο τμήμα του SELECT δεν υπάρχει κανένας περιορισμός όσον αφορά το πλήθος των στηλών αρκεί η κάθε στήλη να υπάρχει σε έναν και μόνο έναν πίνακα.
2. Στο τμήμα του FROM μπορούν να εισαχθούν μέχρι τρεις πίνακες. (Αν και ο Parser αναγνωρίζει "άπειρο" πλήθος πινάκων ο μηχανή εκτέλεσης δέχεται μέχρι τρεις.)

3. Στο τμήμα του `WHERE` μπορούν να εισαχθούν μία ή δύο υποσυνθήκες οι οποίες πρέπει να έχουν την ίδια μορφή με αυτή που περιγράφηκε στην εντολή `DELETE`.
4. Στο τμήμα του `ORDER BY` η στήλη πρέπει να είναι μια από αυτές που βρίσκονται στο τμήμα του `SELECT`.

- `CREATE INDEX` *`IndexName`* `ON` *`TableName`* `(`*`ColumnName`*`) ;`

Με την εντολή αυτή κατασκευάζεται ένας δευτερεύοντας κατάλογος (B+ δένδρο) με όνομα *`IndexName`* ,πάνω στη στήλη *`ColumnName`* στον πίνακα *`TableName`*. Η στήλη αυτή μπορεί να είναι οποιαδήποτε στήλη του πίνακα εκτός αυτής που έχει οριστεί ως στήλη του πρωτεύοντος κλειδιού. Ο λόγος για αυτό είναι ότι στο πρωτεύων κλειδί δημιουργείται πάντα ένα B+ δένδρο (πρωτεύων κατάλογος) με το που δημιουργείται. Επομένως οι επόμενοι κατάλογοι που μπορούν να δημιουργηθούν πάνω σε έναν πίνακα θα είναι πάντοτε δευτερεύοντες. Κατάλογοι μπορούν να δημιουργηθούν μόνο σε στήλες που έχουν τύπο `INTEGER`. Δεν υποστηρίζεται δημιουργία καταλόγων σε στήλες τύπου `CHAR`.

- DROP INDEX *IndexName;*

Γίνεται διαγραφή του δευτερεύοντος καταλόγου με όνομα *`IndexName`*. Πρέπει να επισημάνουμε πως δεν γίνεται να διαγραφεί πρωτεύων κατάλογος.

Οι εντολές και οι τύποι που υποστηρίζονται είναι οι εξής:

Για τις ίδιες τις εγγραφές υποστηρίζονται τιμές τύπου `INTEGER` και `CHAR` μέχρι 50 χαρακτήρων .

Επιπρόσθετα

Υλοποιήθηκαν τα επιπλέον:

1. Στο `SELECT`, στο τμήμα του `WHERE`, μπορούν να υπάρχουν και στήλες με αλφαριθμητικά δεδομένα (π.χ. `SELECT… FROM… WHERE name = 'John'`).

Ο απαριθμητός τύπος SQLReserved

Ο τύπος αυτός κρατάει όλες τις δεσμευμένες λέξει SQL που ένα ερώτημα του χρήστη μπορεί περιέχει. Πιθανές τιμές του είναι το `SELECT`, `FROM`, `WHERE`, `ORDER BY`, `AND`, `DROP` κτλ.

Ο απαριθμητός τύπος SQLTokens

Ο τύπος αυτός περιέχει όλες τις λεξικές μονάδες που παράγει ο Parser του ΣΔΒΔ όταν διαβάζει ένα αρχείο εντολών ή την εντολή του χρήστη και όταν αναγνωρίζει τις κατάλληλες λεξικές μονάδες. Πιθανές τιμές του είναι τα: `AND`,

COMMA, DOT, END_OF_COMMAND, EQUAL, GREATER_OR_EQUAL, LEFT_PARENTHESIS, κτλ.

Η κλάση SQLSymbols

Η κλάση αυτή αντιστοιχεί όλα τα σύμβολα που χρειάζεται να αναγνωρίζει και να χειρίζεται το σύστημα μας με μία λέξη κλειδί. Πιθανές τιμές της είναι οι
END_OF_COMMAND = ";"
DOT = "."
COMMA = ","
RIGHT_PARENTHESIS = ")", κτλ.

Η κλάση SQLPatterns

Η κλάση αυτή περιγράφει κάθε λεξική μονάδα που δεν ανήκει στις δεσμευμένες και το σύστημα μας μπορεί να αναγνωρίσει, με τη βοήθεια μιας κανονικής έκφρασης. Π.χ. INTEGER_NUMBER_PATTERN = "0|[1-9][\\p{Digit}]*";

Η κλάση SQLParser

Η κλάση αυτή υλοποιεί έναν Parser για την SQL-92. Προς στιγμήν δεν υποστηρίζει τον τελεστή '!=' όπως και τους '(' ')' οι οποίοι αλλάζουν την προτεραιότητα των πράξεων. Έχει μόνο μία *public* μέθοδο:

- ***void parseCommands (String batchOfCommands)***: καλείται για να εκτελέσει ένα πλήθος από SQL queries. Αναγνωρίζει το τέλος της κάθε εντολής και καλεί την *parse()* για να την ελέγξει συντακτικά και αυτή με την σειρά της να την παραδώσει στην μηχανή εκτέλεσης για να την εκτελέσει.
- ***String getNextLexicalUnit()***: αναζητά την επόμενη λεξική μονάδα μετά το *indexEndOfLexicalUnit* (που δηλώνει που τελείωνε η προηγούμενη λεξική μονάδα). Επιστρέφει την μονάδα που βρήκε αφού πρώτα ενημερώσει τα πεδία *indexEndOfLexicalUnit, indexStartOfLexicalUnit, currentLexicalString.*

- ***void recognizeLexicalUnit(String lexicalUnit)***: αναγνωρίζει τι είδους λεξική μονάδα είναι αυτή που δέχεται ως όρισμα.

- ***void parse ()***: ελέγχει (με την βοήθεια των *getNextLexicalUnit() & recognizeLexicalUnit()*) σε ποια από τις βασικές εντολές ανήκει η εντολή που είναι αποθηκευμένη στο πεδίο *currentCommand* του SQLParser και έπειτα καλεί την ανάλογη μέθοδο.

- ***void parseCreateIndex():*** καλείται αφού έχει αναγνωρισθεί ότι η εντολή currentCommand είναι η CREATE INDEX. Αναγνωρίζει (με την βοήθεια των *getNextLexicalUnit() & recognizeLexicalUnit()*) την

εντολή ως ορθή αν και μόνο αν ακολουθεί τον τυπικό ορισμό που δώσαμε παραπάνω. Στην περίπτωση αυτή καλεί την μηχανή εκτέλεσης να εκτελέσει την εντολή αλλιώς τυπώνει με την χρήση της *printError* ένα κατάλληλο μήνυμα προς το χρήστη.

Παρόμοια λειτουργούν οι *parseCreateTable()*, *parseDeleteFrom()*, *parseDropIndex()*, *parseInsertInto()*, *parseSelect()* και για αυτό δεν θα προβούμε σε περαιτέρω σχολιασμό.

Το Πακέτο dbms.execution

Το Nidus DBMS χρησιμοποιεί μία μηχανή εκτέλεσης για τον σημασιολογικό έλεγχο και την εκτέλεση των ερωτημάτων που έχουν ελεγχθεί από τον Parser για την συντακτική ορθότητα τους.

Η κλάση ExecutionMachine

Η κλάση αυτή έχει μεθόδους για την εκτέλεση κάθε εντολής που αναγνωρίζει ο Parser. Σε κάθε εντολή πρώτα γίνεται ο σημασιολογικός έλεγχος και έπειτα εκτελείται η εντολή με το βέλτιστο δυνατό τρόπο. Οι σημαντικότερες μέθοδοι της κλάσης είναι οι ακόλουθες:

- ***void createIndex String indexName,String tableName,String columnName)***: δημιουργεί ένα δευτερεύοντα κατάλογο πάνω από την στήλη με όνομα *columnName* του πίνακα με όνομα *tableName.*
  Ελέγχει:
  1. Αν το όνομα του δευτερεύοντα καταλόγου προϋπάρχει.
  2. Αν υπάρχει πίνακα με όνομα *tableName.*
  3. Αν υπάρχει στήλη με όνομα *columnName* στον πίνακα αυτό.
  4. Αν υπάρχει κάποιος άλλος δευτερεύοντας κατάλογος πάνω από την στήλη αυτή.
- ***void createTable(String name, Object[][] columns, String primaryColumn):*** δημιουργεί ένα πίνακα με όνομα *name* , στήλες με όνομα *columns[][0]* και τύπο *columns[][1],* και σαν πρωτεύων κλειδί την στήλη με όνομα *primaryColumn.*
  Ελέγχει:
  1. Αν ο πίνακας προϋπάρχει στην βάση δεδομένων.
  2. Αν το όνομα στήλης υπάρχει στις στήλες *columns.*
  3. *Δεν γίνεται έλεγχος για διπλά ονόματα στηλών πράγμα που ενδέχεται να προκαλέσει προβλήματα.*

- ***void deleteFrom(String tableName, Object[] condition)***: διαγράφει από τον πίνακα με όνομα *tableName* όλες τις εγγραφές που ικανοποιούν την συνθηκη *condition.* Η συνθήκη είναι της μορφής στήλη(*column*), τελεστής(*operator*), τιμή(*value*). Για την συνθήκη ισχύουν όσα αναφέρθηκαν πιο πάνω(SQLParser).
  Ελέγχει:

1. Αν ο πίνακας υπάρχει.
2. Αν το όνομα στήλης υπάρχει στις του πίνακα.
3. Αν είναι τύπου `Char` η στήλη ο τελεστής πρέπει να είναι ο τελεστής ισότητας.

- ***void dropIndex(String indexName)***: διαγράφει τον δευτερεύοντα κατάλογο με όνομα `indexName`.
  Ελέγχει:
  1. Αν υπάρχει δευτερεύον κατάλογος με αυτό το όνομα.

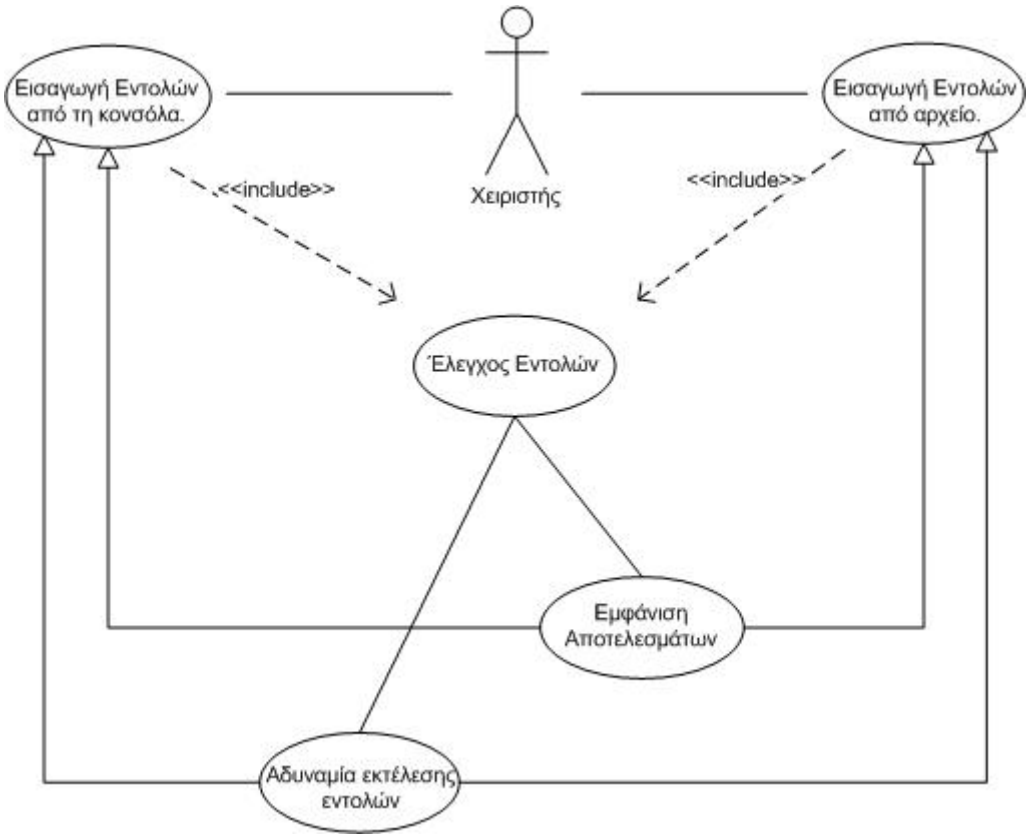
- ***void insertInto(String tableName, Object[] values)***: εισάγει στον πίνακα με όνομα `tableName` μια νέα έγγραφη με τιμές `values`.
  Ελέγχει:
  1. Αν υπάρχει ο πίνακας με όνομα `tableName`.
  2. Αν το μήκος των τιμών είναι ίσο με το πλήθος των στηλών.
  3. Για κάθε τιμή αν ο τύπος της ταιριάζει με τον τύπο της αντίστοιχης στήλης.

- ***private void simpleSelect(Object[] tableNames , Object[] columnNames, Object[] condition , String orderingColumn)***: εκτελείται ένα απλό `select`, δηλ. ένα `select` που δεν συνδυάζεται με ένα άλλο μέσω των `UNION`, `INTERSECTION`, `MINUS`. Επίσης βελτιστοποιεί το ερώτημα ώστε να μην έχουμε άσκοπες αναγνώσεις από τον δίσκο. Αυτό γίνεται ελέγχοντας αν μπορεί να απλοποιηθεί το ερώτημα (π.χ. id < 5 and id <10 απλοποιείται στο id < 5) και αν υπάρχει δευτερεύον κατάλογος πάνω από κάποια στήλη των συνθηκών.
  Ελέγχει:
  1. Οι πίνακες να μην είναι περισσότεροι από τρεις.
  2. Να μην υπάρχουν περισσότερες από δύο υποσυνθήκες.
  3. Αν υπάρχει στήλη ταξινόμησης αυτή πρέπει να υπάρχει και στα ονόματα των στηλών προς εμφάνιση.
  4. Αν υπάρχουν οι πίνακες με τα καθορισμένα ονόματα.
  5. Αν υπάρχουν οι στήλες με τα καθορισμένα ονόματα και αν ναι να υπάρχουν μόνο σε έναν πίνακα.
  6. Αν οι δύο υποσυνθήκες συνδέονται με `OR` οι στήλες που παίρνουν μέρος στις συνθήκες αυτές πρέπει να ανήκουν στον ίδιο πίνακα(αλλιώς το ερώτημα δεν βγάζει νόημα).
  7. Αν κάποια από τις υποσυνθήκες έχουν στήλη με τύπο `Char` τότε η συνθήκη αυτή να έχει τον τελεστή της ισότητας.

- ***void select(ArrayList<SQLTokens> operationBetweenEachSelect, ArrayList< ArrayList<String>> tablesOfEachSelect , ArrayList< ArrayList<String>> columnsOfEachSelect , ArrayList< ArrayList<Object[]>> conditionOfEachSelect , ArrayList<String> orderingColumnOfEachSelect)***: εκτελεί πολλαπλά select τα οποία συνδέονται με μία από τις τρεις πράξείς `UNION`, `INTERSECTION` ή `MINUS`. Εκτελεί πρώτα ξεχωριστά τα δύο πρώτα select καλώντας την `simpleSelect`, έπειτα τα ενώνει κατάλληλα και η ίδια διαδικασία συνεχίζεται μέχρι να ολοκληρωθούν όλες οι πράξεις.
  Ελέγχει:

1. Αν υπάρχουν πράξεις να γίνουν ή έχουμε ένα απλό `select`.
2. Αν τα αποτελέσματα έχουν ίδιο αριθμό στηλών.
3. Αν για κάθε στήλη ταιριάζει ο τύπος της με τον τύπο της αντίστοιχης στήλης.

Βασική περίπτωση χρήσης του συστήματος



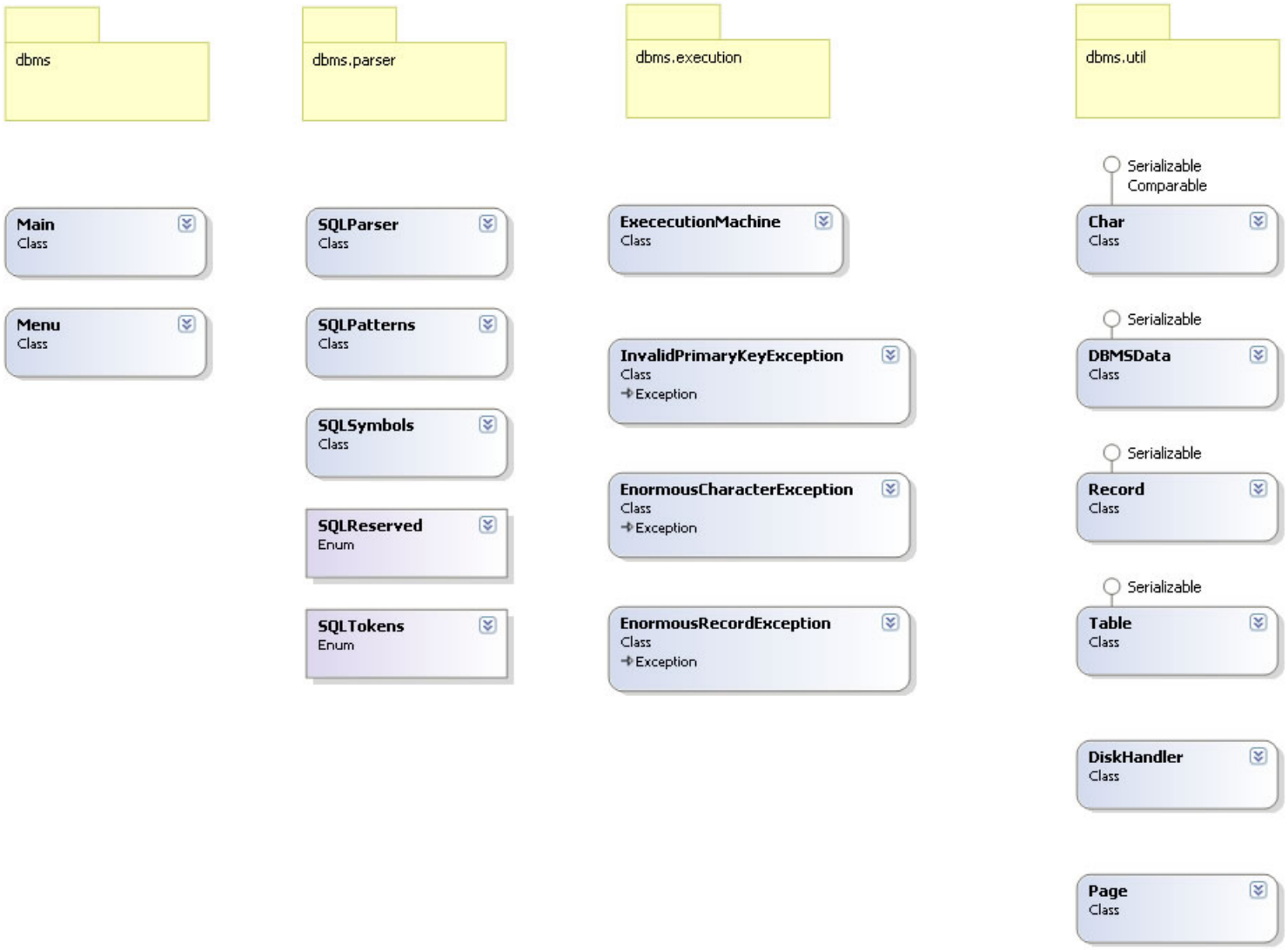
Πηγές
Βιβλιογραφία

• Συστήματα Βάσεων Δεδομένων, Θεωρία & Πρακτική Εφαρμογή, Ιωάννης Μανωλόπουλος, Απόστολος Παπαδόπουλος.
• SQL In A Nutshell, First Edition (Jan 2001), Kevin Kline & Daniel Kline - O'Reilly.
• Αντικειμενοστραφής ανάπτυξη λογισμικού με τη UML, Β. Γερογιάννης, Γ. Κακαρότζας, Α. Καμέας, Γ. Σταμέλος, Π. Φιτσιλής, - Κλειδάριθμος.
• Core java 2 vol.1, fundamentals, 5th edition, Cay S. Horstmann & Gary Cornell - Prentice Hall.
• Java 2 how to program, Deitel & Deitel, 4η εκδοση (2002), Prentice Hall.

Ιστοσελίδες

• Performance of B + Tree Concurrency Control Algorithms
http://www.vldb.org/journal/VLDBJ2/P361.pdf (paper)
• http://babbage.clarku.edu/~achou/cs160/B+Trees/B+Trees.htm (overview)
• (Hashing) http://www.dcc.uchile.cl/~rbaeza/handbook/hbook.html
• (Perfect Hashing) http://burtleburtle.net/bob/hash/perfect.html
• (Double Hashing)
http://www.nist.gov/dads/HTML/doublehashng.html
http://www.brpreiss.com/books/opus5/html/page205.html
http://planetmath.org/encyclopedia/Hashing.html

ΔΙΑΓΡΑΜΜΑ ΚΛΑΣΕΩΝ



ΔΙΑΓΡΑΜΜΑ ΚΛΑΣΕΩΝ