Ατομική Διπλωματική Εργασία

ΑΛΓΟΡΙΘΜΟΙ ΕΞΩΤΕΡΙΚΗΣ ΤΑΞΙΝΟΜΗΣΗΣ ΓΙΑ ΜΝΗΜΕΣ FLASH

Παύλος Τούλουπος

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2010

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Αλγόριθμοι Εξωτερικής Ταξινόμηση για Μνήμες Flash

Παύλος Τούλουπος

Επιβλέπων Καθηγητής
Δημήτρης Ζεϊναλιπούρ

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2010

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον κ. Ζεϊναλιπουρ Δημήτρη επιβλέπον καθηγητή της παρούσας διπλωματικής εργασίας για την υποστήριξη, καθοδήγηση και τα μέσα που μου παρείχε κατά την διάρκεια της εκπόνησης της διπλωματικής μου εργασίας. Πρόκειται για ένα σπουδαίο καθηγητή και ερευνητή με πολλές γνώσεις σε όλους του τομείς της επιστήμης της Πληροφορικής, αλλά πάνω απ' όλα για ένα εξαίρετο άνθρωπο.

Θα ήθελα επίσης να ευχαριστήσω το Τμήμα της Πληροφορικής του Πανεπιστημίου Κύπρου για την αγορά και διάθεση όλων των συσκευών που ήταν απαραίτητα για την ολοκλήρωση της παρούσας διπλωματικής εργασίας.

Τέλος οφείλω να πω ένα μεγάλο ευχαριστώ στους φίλους μου εντός και εκτός Πανεπιστημίου οι οποίοι μου συμπαραστάθηκαν όλο αυτό τον καιρό αλλά και ένα τεράστιο ευχαριστώ στην οικογένειά μου και ιδιαίτερα στους γονείς μου Σάββα και Στέλλα για την ηθική και οικονομική υποστήριξη που μου πρόσφεραν όχι μόνο κατά τη διάρκεια εκπόνησης της παρούσας διπλωματικής εργασίας αλλά και καθ' όλη τη διάρκεια των σπουδών μου.

Περίληψη

Το πρόβλημα της Εξωτερικής Ταξινόμησης (External Sorting) αναφέρεται σε περιπτώσεις όπου τα προς ταξινόμηση δεδομένα δεν χωρούν όλα μαζί στην εσωτερική μνήμη [13]. Το πρόβλημα αυτό συναντάται με μεγάλη συχνότητα σε συστήματα βάσεων δεδομένων αλλά και μικρότερα ενσωματωμένα συστήματα όπως για παράδειγμα σε κινητές συσκευές.

Στόχος της παρούσας διπλωματικής εργασίας είναι η μελέτη και αξιολόγηση αλγορίθμων εξωτερικής ταξινόμησης σε μνήμες Flash και πιο συγκεκριμένα τύπου NAND-Flash, μνήμες που είναι διάχυτες σε Solid-State δίσκους, μνήμες κινητών συσκευών και USB κλειδιά. Επίσης στην παρούσα εργασία προτείνεται ένας νέος αποδοτικότερος αλγόριθμος Εξωτερικής Ταξινόμησης σε τέτοιου τύπου μνήμες, ο οποίος λαμβάνει υπόψη του την αρχιτεκτονική και τις ιδιαιτερότητες που παρουσιάζει η μνήμη NAND-Flash ως αποθηκευτικό μέσο. Στην ουσία, η απουσία κινητών μερών και η ασυμμετρία στις διαδικασίες ανάγνωσης και εγγραφής δεδομένων που παρατηρείται σε τέτοιου τύπου μνήμες διαφοροποιούν τον τρόπο υπολογισμού του κόστους των αλγορίθμων σε σχέση τον παραδοσιακό μαγνητικό δίσκο [29,30]. Ο προτεινόμενος αλγόριθμος επιλύει το πρόβλημα της εξωτερικής ταξινόμησης εκτελώντας σταθερό αριθμό χρονοβόρων εγγραφών στην μνήμη NAND-Flash, κάτι το οποίο επιτυγχάνεται με την αύξηση των λιγότερα χρονοβόρων αναγνώσεων από αυτή. Η νέα προσέγγιση απαλλάσσει τον αλγόριθμο από τις πολλές χρονοβόρες εγγραφές και εκτελεί την ταξινόμηση σε τρία βήματα κάνοντας χρήση ιστογραμμάτων και μιας τεχνικής που χωρίζει το προς ταξινόμηση αρχείο σε ενεργά και νεκρά σημεία.

Συνολικά αξιολογήθηκαν τρεις υφιστάμενοι αλγόριθμοι εξωτερικής ταξινόμησης (External-Merge-Sort [13], Replacement-Selection-Sort [24] και FAST [20]), και δύο εκδόσεις του προτεινόμενου αλγόριθμου ο οποίος ονομάζεται XSORT. Η αξιολόγηση έγινε με σχεσιακά δεδομένα από το TPC-H benchmark [26] και από το Τμήμα Περιβαλλοντικών Σπουδών του Πανεπιστημίου της Ουάσιγκτον [28], πάνω σε πειραματική υποδομή που περιλάμβανε τρία διαφορετικά αποθηκευτικά μέσα (ένα SSD, μια κινητή συσκευή HTC Hero που κάνει χρήση μιας SDcard και ένα USB

κλειδί). Η γλώσσα υλοποίησης όλων των αλγόριθμων ήταν η C και η αξιολόγησή τους έγινε στη βάση του συνολικού χρόνου εκτέλεσης του κάθε αλγόριθμου και του I/O κόστους που προκαλούν.

Σύμφωνα με τα πειράματα που εκτελέσαμε οι δύο παραλλαγές του XSORT αλγόριθμου εμφανίζονται να έχουν καλύτερη απόδοση από όλους τους υφιστάμενους αλγόριθμους κατά 30-50% στα USB κλειδιά και τον Solid-State δίσκο και σε μικρότερο ποσοστό όσον αφορά το κινητό τηλέφωνο.

Περιεχόμενα

Κεφάλαιο 1

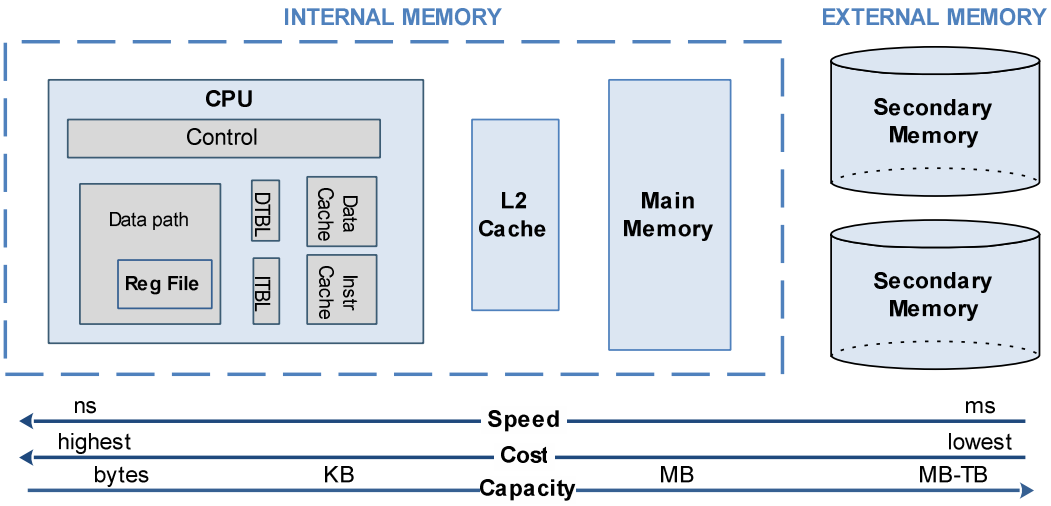
Εισαγωγή

1.1 Το Βασικό Υπόβαθρο

Το πρόβλημα της Ταξινόμηση αφορά την τοποθέτηση μιας ακολουθίας αντικειμένων σε μια συγκεκριμένη λογική σειρά. Τα προς ταξινόμηση αντικείμενα θα πρέπει να είναι του ιδίου τύπου και να μπορούν να συγκριθούν μεταξύ τους στη βάση ενός κοινού τους χαρακτηριστικού. Το πρόβλημα της ταξινόμησης και παρομοίου τύπου προβλήματα καταλαμβάνουν ένα μεγάλο ποσοστό της βιβλιογραφίας στην Επιστήμη της Πληροφορική. Γενικά στο χώρο αυτό υπάρχει μια πληθώρα κατηγοριοποιήσεων του προβλήματος. Εμείς θα αναφερθούμε στο πρόβλημα της *Εξωτερικής Ταξινόμησης (External Sorting)*. Προτού όμως προχωρήσουμε στην επεξήγηση οποιοδήποτε όρου θα ήταν καλό να δώσουμε ένα υπόβαθρο γύρω από το τι ονομάζουμε εσωτερική και τι εξωτερική μνήμη σε ένα τυπικό υπολογιστικό σύστημα.

Ένα τυπικό υπολογιστικό σύστημα διαθέτει διάφορα επίπεδα ιεραρχίας μνήμης, όπου η ιεράρχηση γίνεται βάσει της απόστασή τους από τον επεξεργαστή, όπως φαίνεται και στο Σχήμα 1.1. Συνήθως σε ένα υπολογιστικό σύστημα τα επίπεδα ιεραρχίας μνήμης

είναι οι καταχώρησες, 2-3 επίπεδα κρυφής μνήμης, η κύρια και η δευτερεύουσα μνήμη [29,30]. Επίσης κάθε επίπεδο παρουσιάζει διαφορετική ταχύτητα, κόστος και χωρητικότητα ανάλογα με την απόσταση που έχει από τον επεξεργαστή.



Σχήμα 1.1 : Η ιεραρχία μνήμης σε ένα τυπικό υπολογιστικό σύστημα

Ο όρος *Εσωτερική Μνήμη (Iinternal Memory)* αναφέρεται στο σύνολο των επιπέδων της μνήμης που μπορούν να είναι άμεσα προσβάσιμα από τον επεξεργαστή χωρίς την χρήση των καναλιών εισόδου και εξόδου του συστήματος. Τα επίπεδα αυτά είναι οι καταχωρητές, τα διάφορα επίπεδα κρυφής μνήμης και η κύρια μνήμη. Αντίθετα το σύνολο των επιπέδων της μνήμης όπου η άμεση πρόσβαση από τον επεξεργαστή δεν είναι εφικτή ονομάζεται *Εξωτερική Μνήμη (External Memory)*. Σ' αυτή την περίπτωση η πρόσβαση των δεδομένων γίνεται με τη χρήση των καναλιών εισόδου και εξόδου του συστήματος και είναι ιδιαίτερα χρονοβόρα σε σχέση με τις προσβάσεις σε άλλα επίπεδα μνήμης. Στην ουσία η εξωτερική μνήμη είναι η δευτερεύουσα μνήμη (π.χ., μνήμες flash, δίσκοι, ταινίες κ.λ.π).

Το πρόβλημα της *Εσωτερική Ταξινόμηση* (*Internal Sorting)* αναφέρεται σε περιπτώσεις όπου τα προς ταξινόμηση δεδομένα χωρούν όλα μαζί στην εσωτερική μνήμη και η ταξινόμηση είναι εφικτή μέσα σε αυτήν [13]. Έτσι οι αλγόριθμοι εσωτερικής ταξινόμησης κάνουν άμεσα ή έμμεσα την παραδοχή ότι τα προς ταξινόμηση δεδομένα βρίσκονται ολόκληρα στην εσωτερική μνήμη κατά τη διαδικασία της ταξινόμησης. Η παραδοχή αυτή, προϋποθέτει ότι στο σύστημα υπάρχει ένα και μόνο επίπεδο μνήμης.

Αν και αυτό δεν είναι ρεαλιστικό από τη στιγμή που τα δεδομένα χωρούν όλα στην εσωτερική μνήμη τότε η συγκεκριμένη παραδοχή μπορεί να θεωρηθεί ορθή. Γενικά, υπάρχει ένας μεγάλος αριθμός αλγόριθμων εσωτερικής ταξινόμησης όπως για παράδειγμα ο merge sort, ο quick sort, ο bucket sort κτλ. Η ανάλυση της απόδοσης τέτοιου είδους αλγόριθμων γίνεται στη βάση των ακόλουθων χαρακτηριστικών:

- *Υπολογιστική Πολυπλοκότητα (Time Complexity),* δηλαδή ανάλυση στη βάση βέλτιστης και χείριστης περίπτωσης,
- Π*ολυπλοκότητα Χώρου (Space Complexity),* δηλαδή ο επιπλέον απαιτούμενος χώρος για την εκτέλεση του αλγόριθμου
- *Σταθερότητα Αλγόριθμου (stability),* που καθορίζει αν ο αλγόριθμος διατηρεί τη σχετική σειρά των στοιχείων ενός αρχείου με διπλά κλειδιά
- *Επι-τόπου Ταξινόμηση (in-place),* που καθορίζει αν ο αλγόριθμος χρειάζεται βοηθητικό χώρο για την ταξινόμησης.

Επίσης ο κάθε αλγόριθμος ακολουθεί μια από τις τέσσερεις γενικότερες μεθοδολογίες (Insertion, Selection, Exchange ή Distribution) που στην ουσία είναι αυτή που καθορίζει κατά κύριο λόγο και τα επιμέρους χαρακτηριστικά του αλγόριθμου. Ο Πίνακας 1.1 που ακολουθεί συγκρίνει 7 βασικούς αλγόριθμους εσωτερικής ταξινόμησης βάση των χαρακτηριστικών που αναφέρθηκαν πιο πάνω. Να ανφέρουμε επίσης ότι η μεταβλητή N που εμφανίζεται στον πίνακα καθορίζει το μέγεθος της υπό ταξινόμηση ακολουθίας και η μεταβλητή m το μέγεθος μιας βοηθητικής δομής κάδων για τους αλγόριθμους Bucket sort και Radix sort.

Αλγόριθμος	Σταθερότητα	Επι-τόπου Ταξινόμηση	Τύπος Αλγόριθμου	Χείριστη Περίπτωση	Βέλτιστη Περίπτωση	Επιπλέον Χώρος	Σημειώσεις
Insertion sort	Ναι	Ναι	Insertion	N^2	N	1	Εξαρτάται από την σειρά των δεδομένων στην υπό ταξινόμηση ακολουθία
Selection sort	Όχι	Ναι	Selection	N^2	N^2	1	Ακόμα και για μια ταξινομημένη ακολουθία εισόδου, απαιτείται να διαβαστεί ολόκληρη υπό ταξινόμηση ακολουθία
Heap sort	Όχι	Ναι	Selection	N lg N	N lg N	1	Παραλλαγή του Selection Sort όπου γίνεται χρήση μιας δομής σωρού
Quick sort	Ναι	Ναι	Exchange	N^2	N lg N	1	Εξαρτάται από την κατανομή των δεδομένων στην υπό ταξινόμηση ακολουθία
Merge sort	Όχι	Όχι	Merge	N lg N	N lg N	N	Χρειάζεται επιπλέον χώρος για την ταξινόμηση
Bucket sort	Όχι	Όχι	Distribution	m+N	m+N	1	Δεν γίνονται συγκρίσεις
Radix Sort	Ναι	Όχι	Distribution	m+N	m+N	1	Παραλλαγή του Bucket sort που λειτουργεί μονό σε δεδομένα σταθερού μεγέθους

Πίνακας 1.1 : Τα χαρακτηριστικά των αλγόριθμων εσωτερικής ταξινόμησης, όπου N είναι το μέγεθος της υπό ταξινόμηση ακολουθίας και m το μέγεθος μιας βοηθητικής δομής κάδων για τους αλγόριθμους Bucket sort και Radix sort.

Το πρόβλημα της εξωτερικής ταξινόμησης από την άλλη, αναφέρεται σε περιπτώσεις όπου τα προς ταξινόμηση δεδομένα δεν χωρούν όλα μαζί στην εσωτερική μνήμη και θα πρέπει να γίνει χρήση και της εξωτερικής μνήμης για να είναι εφικτή η ταξινόμηση όλων των δεδομένων [13]. Θα αναφερθούμε εκτενέστερα στο πρόβλημα αυτό στο κεφάλαιο 3. Γενικά στις περιπτώσεις όπου τα δεδομένα δεν χωρούν στην εσωτερική μνήμη, η μεταφορά δεδομένων ανάμεσα στην γρήγορη εσωτερική και την αργή εξωτερική μνήμη αποτελεί και το *Κώλυμα (bottleneck)* του συστήματος. Έτσι η οποιαδήποτε ανάλυση έχει γίνει κάτω από παραδοχή της ύπαρξης μόνο ενός επίπεδου μνήμης είναι εντελώς άκυρη αφού αγνοεί τις χρονοβόρες μεταφορές δεδομένων ανάμεσα σε εσωτερική και εξωτερική μνήμη. Γενικότερα, η ανάγκη χρήσης της εξωτερικής μνήμης ως μέρος της διαδικασίας επεξεργασία των δεδομένων, οδήγησε στην σχεδίαση ενός νέου τύπου αλγορίθμων που ονομάστηκαν *Αλγόριθμοι Εξωτερικής Μνήμης (AEM)* [29]. Οι εν λόγω αλγόριθμοι κάνουν χρήση της εξωτερικής μνήμης σαν χώρο αποθήκευσης ενδιάμεσων αποτελεσμάτων κατά την επεξεργασία των δεδομένων και προσπαθούν να απαλείψουν το κόστος που συνεπάγεται η χρήση των καναλιών εισόδου και εξόδου για μεταφορά δεδομένων. Στις πλείστες των περιπτώσεων η ανάλυση τέτοιων αλγόριθμων γίνεται στην βάση του *Μοντέλου Εξωτερικής Μνήμης (External Memory Model)* [1] και η απόδοσή τους κρίνεται σε σχέση με το I/O κόστους που προκαλούν (δηλ. τον αριθμό των πράξεων εγγραφής και ανάγνωσης στην εξωτερική μνήμη). Θα αναφερθούμε στο Μοντέλο Εξωτερικής Μνήμης στο κεφάλαιο 2. Οι αλγόριθμοι που λύνουν το πρόβλημα της εξωτερικής ταξινόμησης ονομάζονται *Αλγόριθμοι Εξωτερικής Ταξινόμησης (AET)* και προφανώς ανήκουν στην γενικότερη κατηγορία των AEM.

Λόγω της κυριαρχίας του μαγνητικού δίσκου και των ταινιών για δεκαετίες στο χώρο της μόνιμης αποθήκευσης των δεδομένων, η συντριπτική πλειοψηφία της βιβλιογραφίας αφορά την ανάλυση AEM σε τέτοιου είδους μνήμες. Οι κυριότεροι AET σε δίσκους και ταινίες είναι ο *External-Merge-Sort* [13] και ο *Replacement-Selection-Sort* [20]. Η εμφάνιση νέων μνημών μόνιμης αποθήκευσης όπως η μνήμη flash επιβάλει την εκ νέου σχεδίαση αλγορίθμων λόγω των διαφορετικών χαρακτηριστικών που παρουσιάζουν σε σχέση με τον δίσκο και τις ταινίες. Για παράδειγμα η απουσία κινητών μερών και η ασυμμετρία στις διαδικασίες ανάγνωσης και εγγραφής δεδομένων που παρατηρείται στις μνήμες flash διαφοροποιούν τον τρόπο υπολογισμού του I/O

κόστους αφού δεν υπάρχουν οι καθυστερήσεις που οφείλονται στα κινητά μέρη και επιπλέον μια ανάγνωση δεν έχει το ίδιο κόστος με μια εγγραφή όπως συμβαίνει στον δίσκο [34] . Το 2009 οι P. Andreou, O. Spanos, D. Zeinalipour, G. Samaras και P. K. Chrysanthis πρότειναν στο [2] μια παραλλαγή του Replacement-Selection-Sort τον *F-Sort* για την εκτέλεση εξωτερικής ταξινόμησης σε συστήματα αισθητήρων όπου γίνεται χρήση της NAND-Flash σαν αποθηκευτικό μέσο. Κατά την ίδια χρονιά οι H. Park και K. Shim πρότειναν στο [20] ένα νέο AET για συσκευές που κάνουν χρήση της μνήμης flash σαν μνήμη αποθήκευση και τον οποίο ονόμασαν *FAST: Flash-Aware External Sorting Algorithm*. Θα αναφερθούμε στα χαρακτηριστικά της μνήμης flash στο κεφάλαιο 2 και θα παρουσιάσουμε λεπτομερώς όλους τους AET που αναφέρθηκαν πιο πάνω στο κεφάλαιο 3.

Σχήμα 1.2 : Ενδεικτικά μέσα αποθήκευσης Flash

Οι μνήμες Flash αποτελούν την νεότερη τεχνολογία στον χώρο των EEPROM (δηλαδή μνημών μόνιμης αποθήκευσης οι οποίες μπορούν να διαγραφούν και να επαναπρογραμματιστούν με ηλεκτρικό τρόπο). Προσφέρουν επίσης μια πλειάδα πλεονεκτημάτων, όπως για παράδειγμα γρήγορη ανάγνωση, Α*ντοχή σε Κρούσεις (Shock Resistant)*, καλή διαχείριση της ενέργειας που καταναλώνεται, αθόρυβη πρόσβαση και ελαφρότητα. Οι πιο πάνω λόγοι κάνουν τη μνήμη flash ιδανική για χρήση σε κινητές συσκευές. Έτσι, στις μέρες μας αποτελεί το κατεξοχήν αποθηκευτικό μέσο για τις σύγχρονες κινητές συσκευές όπως για παράδειγμα σε ψηφιακές συσκευές ήχου, ψηφιακές φωτογραφικές μηχανές, κινητά τηλέφωνα, PDAs και USB κλειδιά. Πολλοί είναι αυτοί που υποστηρίζουν ότι μελλοντικά η μνήμη flash θα κυριαρχήσει και στον χώρο των φορητών υπολογιστών. Ήδη κάποιοι κατασκευαστές στον χώρο αυτό έχουν προμηθεύσει την αγορά με νέα μοντέλα φορητών υπολογιστών όπως για παράδειγμα το

Apple MacBook Air, το Sony Vaio UX90 και το Samsung Q1-SSD και Q30-SSD, τα οποία έχουν αντικαταστάσει τους παραδοσιακούς μαγνητικούς δίσκους με μνήμη flash (γνωστότερη και ως *Solid-State-Disk* ή απλά *SSD*). Η εταιρεία ερευνών αγοράς In-Stat [10] προέβλεψε ότι μέχρι το 2013 το 50% των φορητών υπολογιστών θα κάνει χρήση της μνήμης Flash (έναντι του σκληρού δίσκου). Επίσης Solid-State δίσκοι χρησιμοποιούνται αντί του μαγνητικού δίσκου και σε συστήματα βάσεων δεδομένων εταιρειών όπου ο φόρτος εργασίας είναι κατά κύριο λόγο αναγνώσεις της βάσης [16], ενώ σύμφωνα με πρόσφατη μελέτη της Intel στο [33] η χρήση Solid-State δίσκων βελτιώνει την απόδοση του συστήματος μέχρι και οκτώ φορές όταν ο φόρτος εργασίας αφορά μόνο εγγραφές (read-only workloads).

Σχήμα 1.3 : Το Apple MacBook Air (Αριστερά) και το Sony Vaio UX90 (δεξιά)

1.2 Υποκίνηση Εργασίας

Πρόσφατα έχει δημιουργηθεί μια τάση προς την κατεύθυνση της χρήσης της μνήμης flash για καλύτερη απόδοση υπολογιστικών συστημάτων εκμεταλλευόμενοι τα ιδιαίτερα χαρακτηριστικά που αυτή παρουσιάζει σε σχέση με τους παραδοσιακούς δίσκους. Τα συστήματα αυτά περιλαμβάνουν τόσο συστήματα βάσεων δεδομένων που χρησιμοποιούν SSD δίσκους για αποθήκευση των δεδομένων [3,15,18,27,14] αλλά και ενσωματωμένα συστήματα που κάνουν χρήση της flash σαν μνήμη αποθήκευσης [5,19]. Ερεύνα υπάρχει και στον τομέα της δημιουργίας αποδοτικότερων δομών και αλγόριθμων που θα λειτουργούν καλύτερα σε μνήμες flash. Τέτοια παραδείγματα αποτελούν οι υλοποιήσεις B-trees στο [32] , R-trees στο [31], ουρών και στοιβών στο [17] καθώς επίσης και hash tables στο [34]. Επίσης στο [6] παρουσιάζεται η υλοποίηση

έξυπνων αλγόριθμων για τον υπολογισμό του κοντινότερου μονοπατιού σε Pocket PCs όπου οι παραγόμενοι γράφοι ήταν αποθηκευμένοι σε μνήμη flash.

Αν και η ολοένα αυξανόμενη πορεία των ποσοστών της βιομηχανίας παραγωγής μνημών flash στον τομέα των μνημών μόνιμης αποθήκευσης και η κυριαρχία της μνήμης flash ως μνήμη αποθήκευσης σε κινητές συσκευές αποτελεί από μόνη της μια υποκίνηση για δημιουργία αποδοτικότερων αλγόριθμων ταξινόμησης σε αυτού του είδους την μνήμη, ένας τέτοιος αλγόριθμος θα μπορούσε να βρει και πολλές άλλες εφαρμογές. Για παράδειγμα θα μπορούσε να χρησιμοποιηθεί σε μεγάλα συστήματα που διαχειρίζονται μεγάλο όγκο δεδομένων, όπως σε μια βάση δεδομένων που κάνει χρήση ενός SSD. Επίσης θα μπορούσε να αποτελέσει μια νέα βιβλιοθήκη αποδοτικής ταξινόμησης σε ενσωματωμένες βάσεις δεδομένων όπως η SQLite [25] που βρίσκει εφαρμογές στο χώρο των ενσωματωμένων συστημάτων όπου κυριαρχεί η χρήση της NAND-Flash μνήμης όπως σε κινητά τηλέφωνα κτλ.

1.3 Περίγραμμα Εργασίας

Στο πρώτο κεφάλαιο παρουσιάσαμε το βασικό υπόβαθρο και την υποκίνηση που υπάρχει για δημιουργία πιο αποδοτικών αλγόριθμων εξωτερικής ταξινόμησης σε μνήμες flash. Στο δεύτερο κεφάλαιο της εργασίας αυτής θα παρουσιάσουμε το μοντέλο συστήματος, τα χαρακτηριστικά που έχει η μνήμη flash ως αποθηκευτικό μέσω. Θα αναφερθούμε επίσης σε όλους τους τεχνικούς όρους, τους αντίστοιχους συμβολισμούς που θα χρησιμοποιηθούν καθώς επίσης και σε όλες τις παραδοχές που πρέπει να γίνουν. Στο τρίτο κεφάλαιο θα γίνει μια παρουσίαση του προβλήματος της εξωτερικής ταξινόμησης και μια λεπτομερέστατη περιγραφή τόσο του τρόπου λειτουργίας αλλά και της απόδοσης του κάθε αλγόριθμου βάση του μοντέλου συστήματος που ορίσαμε. Στο τέταρτο κεφάλαιο θα παρουσιάσουμε ένα νέο αλγόριθμο εξωτερικής ταξινόμησης για μνήμες flash ο οποίος ακολουθεί μια νέα στρατηγική και εκτελεί την ταξινόμηση με τη χρήση ιστογραμμάτων. Στο πέμπτο κεφάλαιο θα γίνει μια παρουσίαση της πειραματικής διαδικασίας που ακολουθήσαμε για την σύγκριση των αλγορίθμων ενώ στο έκτο θα παρουσιάζουμε και θα αναλύσουμε τα αποτελέσματα αυτής της διαδικασίας. Στο έβδομο και τελευταίο κεφάλαιο θα παραθέσουμε τα τελικά συμπεράσματα και τις επιπλέον βελτιστοποιήσεις που μπορούν να γίνουν. Στο

Παράρτημα Α παρουσιάζουμε ένα οδηγό εγκατάστασης και χρήσης εφαρμογών γραμμένων σε γλώσσα προγραμματισμού C πάνω σε περιβάλλον Android, στο Β τους αναλυτικούς πίνακες αποτελεσμάτων, στο Γ τον πηγαίο κώδικα των εφαρμογών που έχουμε γράψει και τέλος στο Δ τον συνοδευτικό ψηφιακό δίσκο.

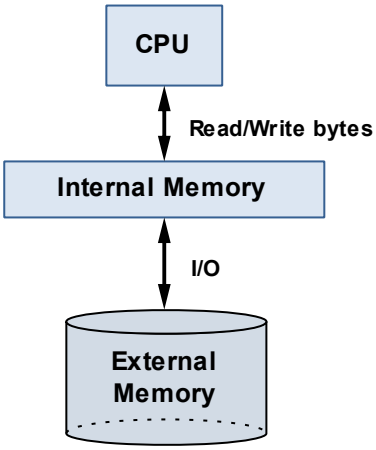
Κεφάλαιο 2

Μοντέλο Συστήματος

Στο κεφάλαιο αυτό θα αναφερθούμε σε όλους τους τεχνικούς όρους που αφορούν το αντικείμενο που μελετάμε και θα δώσουμε ένα γενικό μοντέλο συστήματος που θα μας βοηθήσει στην αξιολόγηση των αλγόριθμων που μελετάμε. Απλά να αναφέρουμε ότι θα χρησιμοποιήσουμε μια παραλλαγή του μοντέλου εξωτερικής μνήμης (External Memory model) που προτάθηκε από τους Aggarwal and Vitter [1]. Στην συνέχεια θα παρουσιάσουμε τα ιδιαίτερα χαρακτηριστικά της μνήμη flash ως αποθηκευτικού μέσου και ακολούθως θα αναφερθούμε σε όλες τις επιπλέον παραδοχές που είναι αναγκαίο να γίνουν. Στο τέλος του κεφαλαίου θα συνοψίσουμε σε ένα πίνακα συμβόλισμών, εννοιών και παραδοχών όλες τις έννοιες και τους συμβολισμούς που θα χρησιμοποιηθούν καθώς επίσης και τις επιπλέον παραδοχές που έγιναν.

2.1 Το Μοντέλο Εξωτερικής Μνήμης

Το εν λόγω μοντέλο, γνωστό και ως *Μοντέλο Εισόδου/Εξόδου (I/O Model)* χρησιμοποιείται κυρίως για την αξιολόγηση αλγορίθμων εξωτερικής μνήμης σε μαγνητικούς δίσκους. Σύμφωνα λοιπόν με το μοντέλο αυτό υποθέτουμε την ύπαρξη μιας κεντρικής μονάδας επεξεργασίας και δύο επίπεδων ιεραρχίας μνήμης όπως φαίνονται και στο Σχήμα 2.1 .



Σχήμα 2.1 : Το Μοντέλο Εξωτερικής Μνήμης

Σε οποιαδήποτε χρονική στιγμή, η επεξεργασία των δεδομένων είναι εφικτή μόνο αν τα εν λόγω δεδομένα βρίσκονται ήδη στην εσωτερική μνήμη. Σε αντίθετη περίπτωση τα δεδομένα αυτά θα πρέπει να μεταφερθούν από την εξωτερική στην εσωτερική μνήμη για να είναι σε θέση να τύχουν περεταίρω επεξεργασίας. Αν και εξ ορισμού η εσωτερική μνήμη είναι γρηγορότερη εντούτοις έχει περιορισμένο μέγεθος. Η μεταφορά δεδομένων ανάμεσα στην εσωτερική και την εξωτερική μνήμη είναι εφικτή μόνο υπό μορφή μπλοκ (block) δεδομένων. Η ανάγνωση ή η εγγραφή ενός μπλοκ από και προς την εξωτερική μνήμη αφορά πάντοτε εγγραφές οι οποίες καταλαμβάνουν συνεχόμενες θέσεις σε αυτή. Εμείς θα δανειστούμε την ορολογία *Σελίδα (Page)* από τον τομέα των βάσεων δεδομένων όπου οι έννοιες μπλοκ και σελίδα ταυτίζονται. Το μέγεθος κάθε σελίδας μετράται σε bytes και εξαρτάται απόλυτα από το *Σύστημα Αρχείων (file system)* που χρησιμοποιείται από το λειτουργικό. Για παράδειγμα στα Linux όπου γίνεται χρήση του ext2 ή του ext3 συστήματος αρχείων το block size είναι 4096 bytes = 4KB.

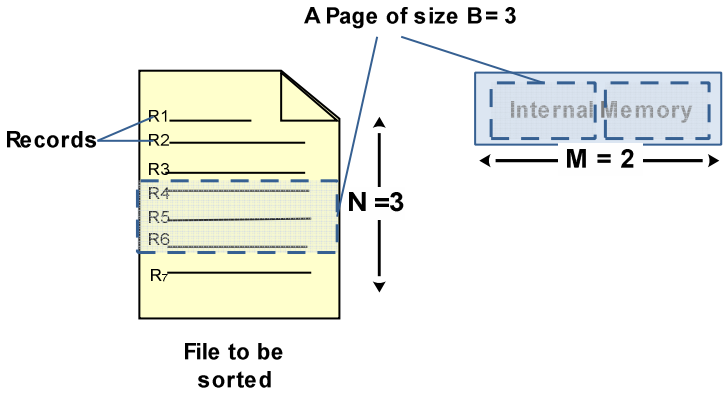
Στο σημείο αυτό να αναφερθούμε στους συμβολισμού που θα χρησιμοποιούμε για το υπόλοιπο αυτού του εγγράφου. Θα χρησιμοποιούμε λοιπόν τον συμβολισμό L_x για να καθορίζουμε το μέγεθος σε bytes του αντικειμένου x. Πιο συγκεκριμένα θα αναφερόμαστε στο *Μέγεθος μιας Σελίδας* ως L_p , στο *Συνολικό Μέγεθος του Αρχείου* ως L_f, στο *Μέγεθος μιας Εγγραφής του Αρχείου* ως L_r και στο *Μέγεθος της Εσωτερικής Μνήμης* ως L_m. Ακόμη θα γίνεται χρήση της σταθεράς B που θα καθορίζει το *Μέγιστο Αριθμό των Εγγραφών που μπορούν να μεταφερθούν ανά πάσα στιγμή μέσα σε μια σελίδα* και της σταθεράς N που θα καθορίζει τον *Συνολικό Αριθμό των Σελίδων* που περιέχονται στο προς ταξινόμηση αρχείο. Επίσης θα χρησιμοποιούμε τον συμβολισμό M ως την σταθερά που θα καθορίζει τον *Μέγιστο Αριθμό των Σελίδων που μπορούν να βρίσκονται ανά πάσα στιγμή μέσα στην εσωτερική μνήμη*. Στο σημείο αυτό θα πρέπει επίσης να γίνει μια τετριμμένη παραδοχή ότι $1 \leq B$ και $1 \leq M < N$.

Το παράδειγμα που ακολουθεί (Σχήμα 2.3) υποθέτει ότι έχουμε ένα αρχείο προς ταξινόμηση, που περιέχει 7 εγγραφές $R_{1-7.}$ Υποθέτουμε επίσης ότι: L_r = 40 bytes, L_p = 128 bytes και L_{mm} = 250 bytes. Από τα πιο πάνω, προκύπτει ότι:

$B = \lfloor L_p / L_r \rfloor = \lfloor 128/40 \rfloor = 3$ έγραφες ανά σελίδα,

$N = \lceil total\ \#\ of\ records / B \rceil = \lceil 7/3 \rceil = 3$ σελίδες

και $M = \lfloor L_f / L_p \rfloor = \lfloor 250/128 \rfloor = 2$ σελίδες.



Σχήμα 2.2 : Βασικές Έννοιες Μοντέλου Εξωτερικής μνήμης

Στο μοντέλο μοντελοποιείται και η παραλληλία μέσω μιας σταθεράς P που δηλώνει τον αριθμό των σελίδων που μπορούν να μεταφέρονται ταυτόχρονα ανά πάσα στιγμή στο σύστημα. Εμείς κάνουμε την παραδοχή ότι P = 1. Τέλος, σύμφωνα με το μοντέλο το

κόστος μεταφοράς των σελίδων (εγγραφή και ανάγνωση στην εξωτερική μνήμη) ορίζεται ως κόστος I/O και πλέον η απόδοση ενός αλγόριθμου μπορεί να χαρακτηριστεί βάσει του I/Os κόστους που προκαλεί. Θα αναφερόμαστε στο ολικό I/O κόστος ενός προγράμματος με το συμβολισμό $C_p^{I/O}$, όπου p το αντίστοιχο πρόγραμμα.

2.2 Η Μνήμη Flash

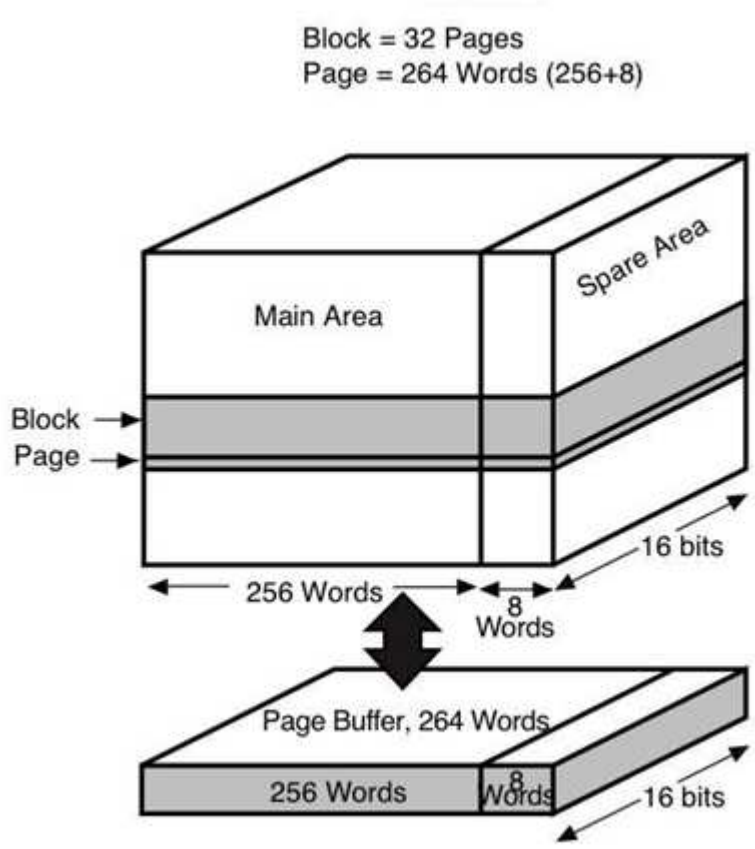
Υπάρχουν δύο διαφορετικοί τύποι μνήμης flash, η NOR flash και NAND flash, η ονομασία των οποίων οφείλεται στο είδος της λογικής πύλης που χρησιμοποιείται στις εσωτερικές διασυνδέσεις των τρανζίστορ-ψηφίων για την κάθε περίπτωση. Η NOR flash αποτελεί παλαιότερου τύπου μνήμης flash και προσφέρει γρήγορες προσβάσεις στην μνήμη (τυχαία προσπέλαση σε οποιοδήποτε μέρος της μνήμης) αλλά είναι ιδιαίτερα αργή στις λειτουργίες της εγγραφής και διαγραφής δεδομένων. Για το λόγο αυτό η NOR flash χρησιμοποιείται κυρίως για αποθήκευση κώδικα (π.χ., για το BIOS). Η NAND flash αποτελεί την νεότερη γενεά μνήμης flash και χαρακτηρίζεται από μεγαλύτερη ανθεκτικότητα, μεγαλύτερη χωρητικότητα και χαμηλότερο κόστος. Επίσης η NAND flash παρουσιάζει χαμηλότερους χρόνους ανάγνωσης από την NOR flash (λόγω κυρίως του I/O συστήματος διπροσωπίας που χρησιμοποιεί) αλλά είναι ταχύτερη όσον αφορά τις λειτουργίες της διαγραφής και της εγγραφής δεδομένων. Πρόσφατα ανακοινώθηκε από την SAMSUNG η δημιουργία ενός νέου υβριδικού τύπου μνήμης flash στην οποία δόθηκε το όνομα SAMSUNG OneNAND [21] και η οποία συνδυάζει τα θετικά στοιχεία κάθε τύπου προσφέροντας με αυτό τον τρόπο την ταχύτητα της NOR flash στις λειτουργίες της ανάγνωσης δεδομένων και την ταχύτητα της NAND flash στις λειτουργίες της αποθηκεύσεις των δεδομένων (εγγραφής και διαγραφής).

Οι μνήμες flash παρουσιάζουν κάποιες σημαντικές διαφοροποιήσεις από τον παραδοσιακό μαγνητικό δίσκο [8]. Οι βασικότερες είναι οι εξής δύο :

1. **Δεν υπάρχουν κινητά μέρη:** Σε αντίθεση λοιπόν με τον μαγνητικό δίσκο, οι μνήμες flash δεν διαθέτουν μηχανικά μέρη (π.χ., κεφαλή του δίσκου) και έτσι δεν υποφέρουν από *καθυστερήσεις αναζήτησης (seek time)* ή *περιστροφής (rotation delay)* όπως οι μαγνητικοί δίσκοι.

2. **Μη συμμετρικό κόστος εγγραφής και ανάγνωσης δεδομένων**: Αν και στον μαγνητικό δίσκο το κόστος εγγραφής και ανάγνωσης δεδομένων είναι το ίδιο, στην μνήμη flash μια ανάγνωση είναι κατά α φορές πιο γρήγορή απ' ότι μια αντίστοιχη εγγραφή, όπου α μια σταθερά που εξαρτάται από τη συσκευή. (π.χ., στις NAND flash συνήθως χρειαζόμαστε περίπου 60 μs για ανάγνωση σε σχέση με περίπου 800 μs για εγγραφή).

Εμείς θα επικεντρωθούμε στην NAND flash αφού η μνήμη αυτή βρίσκει τις περισσότερες εφαρμογές στον σύγχρονο κόσμο. Σε γενικές γραμμές κάθε μνήμη NAND flash οργανώνεται σε σελίδες, όπου το τυπικό μέγεθος κάθε σελίδας είναι τα 256 ή 512 bytes. Ένας ορισμένος αριθμός σελίδων (συνήθως 16 ή 32 σελίδες) συνθέτει ένα μπλοκ μνήμης με τυπικό μέγεθος 4KB-64KB. Τέλος η συνολική μνήμη απαρτίζεται από ένα αριθμό τμημάτων (π.χ., 512 τμήματα).



Σχήμα 2.3 : Η οργάνωση της NAND Flash

Η οργάνωση της μνήμη NAND flash με τον τρόπο που αναφέραμε οδηγεί σε κάποιους περιορισμούς [33]. Οι κυριότεροι περιορισμοί είναι οι εξής :

1. **Ανάγνωσης** : Η ανάγνωση των δεδομένων από την μνήμη flash μπορεί να γίνει σε διάφορα μεγέθη μνήμης από 1 byte μέχρι και ένα ολόκληρο block (συνήθως 4KB-64KB)
2. **Διαγραφής** : Η διαγραφή δεδομένων από την μνήμη flash μπορεί να γίνει μόνο υπό μορφή ενός block (συνήθως 4KB-64KB)
3. **Εγγραφής** : Η εγγραφή δεδομένων στην μνήμη flash μπορεί να γίνει μόνο υπό μορφή μιας σελίδας (συνήθως 256 ή 512), αφού όμως έχει προηγηθεί η διαγραφή της σελίδας αυτής πράγμα που συνεπάγεται και την διαγραφή ολόκληρου του block που την περιέχει (συνήθως 4KB-64KB)
4. **Επανεγγραφής**: Κάθε σελίδα μπορεί να ξαναεγγραφεί ένα περιορισμένο αριθμό φορών (συνήθως 10000 – 100000 φορές).

2.3 Επιπλέον Παραδοχές

Δεδομένου του γεγονότος ότι το Μοντέλο Εξωτερικής Μνήμης που αναφέραμε στην παράγραφο 2.1 αφορά μαγνητικούς δίσκους, θα πρέπει να αναθεωρήσουμε μια βασική αρχή του. Στο εν λόγο μοντέλο το κόστος ανάγνωσης και εγγραφής ενός block από και προς το δίσκο είναι συμμετρικό και προκαλεί το ίδιο I/O κόστος (=1 I/O). Σύμφωνα με τα όσα αναφέρθηκαν στην παράγραφο 2.2 το κόστος ανάγνωσης και εγγραφής ενός block στην μνήμη flash δεν είναι συμμετρικό όπως συμβαίνει στον δίσκο. Έτσι θα πρέπει να διαφοροποιηθεί ο τρόπος υπολογισμού του I/O κόστους για τις πράξεις της ανάγνωσης και της εγράφης προς την εξωτερική μνήμη που υποθέτουμε πλέον ότι είναι μνήμη flash. Έτσι θα συμβολίζουμε λοιπόν με C_R το I/O κόστος για μια ανάγνωση από την μνήμη flash και με C_W το I/O κόστος για μια εγγραφή στην μνήμη flash. Από την στιγμή που μια εγγραφή είναι κατά α φορές πιο χρονοβόρα από ότι μια ανάγνωση στην μνήμη flash,όπου α μια σταθερά που εξαρτάται από τη συσκευή, τότε μπορούμε να κάνουμε την επιπλέον παραδοχή ότι $C_w = \alpha C_R$. Να αναφέρουμε επίσης ότι στην βιβλιογραφία έχει υπολογιστεί ότι στην NAND η σταθερά αυτή είναι ≈ 10.

2.4 Πίνακας Συμβολισμών και Παραδοχών

Ακολουθεί ένας πίνακας συμβόλων και παραδοχών που συνοψίζει όλα όσα αναφέρθηκαν και επεξηγήθηκαν στα δύο πρώτα κεφάλαια.

Συμβολισμός	Περιγραφή
N	Ο συνολικός αριθμός των σελίδων που περιέχονται στο προς ταξινόμηση αρχείο.
M	Ο μέγιστος αριθμός των σελίδων που μπορούν να βρίσκονται ανά πάσα στιγμή μέσα στην εσωτερική μνήμη.
B	Ο μέγιστος αριθμός των εγγραφών που μπορούν να μεταφερθούν ανά πάσα στιγμή μέσα σε μια σελίδα
P	Ο μέγιστος αριθμός των σελίδων που μπορούν να μεταφέρονται ταυτόχρονα στο σύστημα.
L_r	Το μέγεθος μιας εγγραφής του αρχείου σε bytes
L_p	Το μέγεθος της σελίδας του συστήματος σε bytes
L_f	Το μέγεθος του προς ταξινόμηση αρχείου σε bytes
L_m	Το μέγεθος της κύριας μνήμης σε bytes
$C_p^{I/O}$	Συνολικό I/O κόστος για το πρόγραμμα p.
C_R	I/O κόστος ανάγνωσης μια σελίδας από τη μνήμη flash
C_w	I/O κόστος εγγραφής μια σελίδας προς τη μνήμη flash

Παραδοχές
1. $1 \leq B$ και $1 \leq M < N$
2. $P = 1$
3. $C_w = \alpha C_R$, όπου α μια σταθερά που εξαρτάται από τη συσκευή

Πίνακας 2.1 : Πίνακας Σύμβολism;vn , Εννοιών και Παραδοχών

Στα κεφάλαια που ακολουθούν θα γίνεται απλή χρήση των πιο πάνω συμβολισμών και παραδοχών όπως αυτά ορίστηκαν και επεξηγήθηκαν σε αυτό το κεφάλαιο χωρίς επιπλέον επεξηγήσεις. Ο αναγνώστης προτρέπεται να επιστρέφει στο πίνακα αυτό για υπενθύμιση της σημασίας των όρων και συμβολισμών.

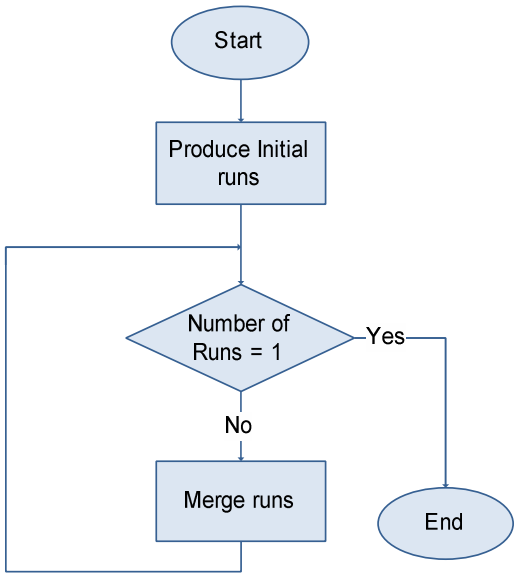
Κεφάλαιο 3

Αλγόριθμοι Εξωτερικής Ταξινόμησης

Στο κεφάλαιο αυτό θα μελετήσουμε τον όρο εξωτερική ταξινόμηση και θα δώσουμε τα βασικά χαρακτηριστικά ενός τυπικού αλγόριθμου εξωτερικής ταξινόμησης όπως ορίστηκαν από τον Donald Knuth [13]. Στη συνέχεια θα παρουσιάσουμε τους τρεις βασικούς αλγόριθμους εξωτερικής ταξινόμησης που θα μελετήσουμε, τον External-Merge-Sort [13], τον Replacement-Selection-Sort [24] και τον αλγόριθμο FAST [20].

3.1 Εισαγωγή στην Εξωτερική Ταξινόμηση

Το πρόβλημα της *Εξωτερικής Ταξινόμησης (External Sorting)* αναφέρεται σε περιπτώσεις όπου τα δεδομένα που περιέχονται στο προς ταξινόμηση αρχείο δεν χωρούν όλα μαζί στην *Εσωτερική (Internal)* μνήμη [13]. Αυτό έχει ως αποτέλεσμα να είναι αδύνατη η ταξινόμηση με τη χρήση κάποιου από τους κλασσικούς αλγόριθμους εσωτερικής ταξινόμησης (π.χ., merge sort, quick sort, bucket sort, κτλ), αφού οι αλγόριθμοι αυτοί κάνουν άμεσα ή έμμεσα την "αφελή" παραδοχή ότι τα προς ταξινόμηση δεδομένα βρίσκονται ολόκληρα στην εσωτερική μνήμη κατά τη διαδικασία της ταξινόμησης. Σε αντίθεση λοιπόν με αυτούς τους αλγόριθμους, οι αλγόριθμοι εξωτερικής ταξινόμησης θεωρούν ότι μόνο ένα μέρος των προς ταξινόμηση δεδομένων μπορεί να βρίσκεται ανά πάσα χρονική στιγμή στην εσωτερική μνήμη, ανάλογα του μεγέθους της και όχι όλα τα δεδομένα ταυτόχρονα. Οι αλγόριθμοι αυτοί ανήκουν στην γενικότερη κατηγορία των αλγορίθμων εξωτερικής μνήμης, αλγόριθμοι που κάνουν χρήση της εξωτερικής μνήμης σαν χώρο αποθήκευσης ενδιάμεσων αποτελεσμάτων κατά την επεξεργασία των δεδομένων και προσπαθούν να απαλείψουν το κόστος που συνεπάγεται η χρήση των καναλιών εισόδου και εξόδου για μεταφορά δεδομένων.

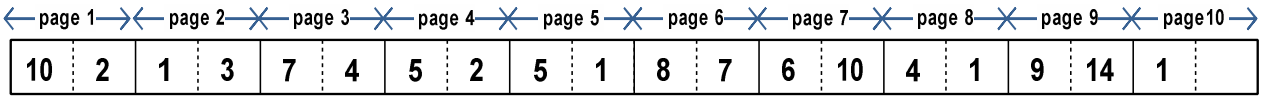


Σχήμα 3.1 : Ένας τυπικός αλγόριθμος εξωτερικής ταξινόμησης που κάνει χρήση της λογικής της συνένωσης

Η συντριπτική πλειοψηφία των υφιστάμενων αλγορίθμων εξωτερικής ταξινόμησης ακολουθούν μια κοινή στρατηγική, η οποία στηρίζεται στην λογική της *Συνένωσης*

(Merging). Ένας τυπικός αλγόριθμος εξωτερικής ταξινόμησης αποτελείται από δύο φάσεις, την φάση της εσωτερικής ταξινόμησης ή Φάση Παραγωγής των Αρχικών Runs και την Φάση της συνένωσης. Κατά την Φάση της Παραγωγής των Αρχικών Runs, ένα μέρος των προς ταξινόμηση εγγραφών ικανό να χωρέσει στην εσωτερική μνήμη μεταφέρεται σε αυτή και ταξινομείται. Η παραγόμενη ακολουθία ταξινομημένων εγγραφών γράφεται πίσω στην εξωτερική μνήμη σε μορφή προσωρινού αρχείου. Έτσι διαδοχικά ολόκληρο το προς ταξινόμηση αρχείο θα μπει στην εσωτερική μνήμη και θα ταξινομηθεί σε μορφή μικρότερων κομματιών με αποτέλεσμα την παραγωγή ενός αριθμού από ταξινομημένα προσωρινά "υποαρχεία". Θα αναφερόμαστε σε αυτά τα αρχεία ως *runs*. Κατά τη δεύτερη φάση του αλγόριθμου, γίνεται μια επαναληπτική σταδιακή συνένωση των ήδη παραγόμενων runs σε runs μεγαλύτερου μεγέθους, μέχρι να φτάσουμε σε ένα και μόνο παραγόμενο run το οποίο θα αποτελεί και το ταξινομημένο πλέον αρχείο. Επίσης θα ονομάζουμε ως ένα *Πέρασμα (Pass)* την διαδικασία ανάγνωσης όλων των προς ταξινόμηση δεδομένων στην εσωτερική μνήμη, την επεξεργασία τους μέσα σε αυτή και τελικά την εγγραφή τους σε μορφή runs στην εξωτερική.

Στις παραγράφους που ακολουθούν θα παρουσιάσουμε τρεις βασικούς αλγόριθμους εξωτερικής ταξινόμησης. Η περιγραφή του κάθε αλγόριθμου θα γίνει σε δύο επίπεδα. Το πρώτο επικεντρώνεται σε μια λεκτική περιγραφή των χαρακτηριστικών και του τρόπου λειτουργίας του κάθε αλγόριθμου και το δεύτερο αφορά την σχηματική αναπαράσταση της εκτέλεσης του αλγόριθμου μέσα από ένα κοινό παράδειγμα. Στην ουσία θα δείξουμε πώς ο κάθε αλγόριθμος θα αντιμετωπίσει ένα υποθετικό σενάριο που θα ορίσουμε. Σύμφωνα λοιπόν με το εν λόγω σενάριο υποθέτουμε ότι υπάρχει ένα προς ταξινόμηση αρχείο 19 εγγραφών αποθηκευμένο στην εξωτερική μνήμη τύπου NAND Flash, το οποίο θέλουμε να ταξινομήσουμε σε αύξουσα σειρά και τόσο το σχήμα όσο και ο τύπος δεδομένων των χαρακτηριστικών κάθε εγγραφής του αρχείου είναι γνωστά εκ των προτέρων. Υποθέτουμε επίσης ότι η ακολουθία αριθμών που παρουσιάζουμε πιο κάτω (Σχήμα 3.2) αποτελεί το κλειδί βάση του οποίου θα γίνει η ταξινόμηση. Να αναφέρουμε ότι για λόγους οικονομίας χώρου τα υπόλοιπα δεδομένα παραλείπονται από το εν λόγω σχήμα.



Σχήμα 3.2 : Το προς ταξινόμηση αρχείο

Θεωρούμε επίσης ότι οι εγγραφές έχουν *σταθερό μέγεθος (fixed length)* L_r = 200 Bytes συμπεριλαμβανομένου και των 4 bytes του κλειδιού της ταξινόμησης. Ας θεωρήσουμε επίσης ότι στο σενάριο που μελετάμε L_p = 512 Bytes και ότι L_m = 2000 Bytes. Με αυτά τα δεδομένα κάθε σελίδα μπορεί να περιέχει μέχρι και $\lfloor 512/200 \rfloor$ = 2 εγγραφές, άρα B=2 εγγραφές ανά σελίδα. Επίσης στην διαθέσιμη μνήμη χωρούν $\lfloor 2000/512 \rfloor$ = 3 σελίδες, άρα M=3 και τέλος N= $\lceil 19/2 \rceil$ = 10 σελίδες.

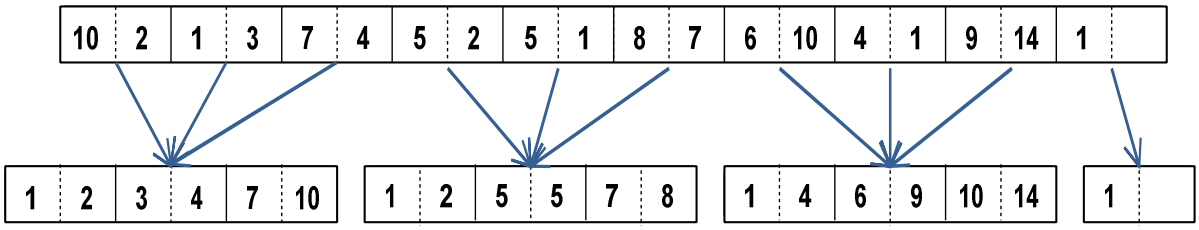
3.2 External-Merge-Sort

Ο αλγόριθμος αυτός αποτελεί τον βασικότερο αλγόριθμο εξωτερικής ταξινόμησης πάνω σε μαγνητικούς δίσκους και στηρίζεται στην λογική της συνένωσης που αναφέραμε πιο πάνω. Κατ' επέκταση αποτελείται από δύο φάσεις: την φάση της παραγωγής των αρχικών runs και την φάση της συνένωσης. Σε γενικές γραμμές ο εν λόγω αλγόριθμος χαρακτηρίζεται από απλότητα στην υλοποίηση, ενώ ένα άλλο χαρακτηριστικό του είναι ότι τα παραγόμενα runs σε κάθε pass έχουν όλα ίδιο μέγεθος.

3.2.1 External-Merge-Sort: Φάση Παραγωγής των Αρχικών Runs

Κατά τη διάρκεια της πρώτης φάσης του αλγορίθμου ένα μπλοκ από M συνεχόμενες σελίδες μεταφέρεται από την εξωτερική στην εσωτερική μνήμη, ταξινομείται με τη χρήση ενός αλγόριθμου εσωτερικής ταξινόμησης (π.χ quick sort κτλ), και στη συνέχεια γράφεται πίσω στην εξωτερική μνήμη σε μορφή ενός run. Επαναλαμβάνουμε την ίδια διαδικασία μέχρι να μην υπάρχει άλλη εγγραφή στο αρχείο. Να αναφέρουμε επίσης ότι ο αριθμός των παραγόμενων runs μετά το τέλος της πρώτη φάση του αλγόριθμου είναι $\lceil N/M \rceil$, και ότι όλα τα παραγόμενα runs έχουν μέγεθος M σελίδων (το τελευταίο μπορεί να έχει μικρότερο μέγεθος ανάλογα με τον αριθμό των εγγραφών του αρχικού αρχείου). Στο Σχήμα 3.3 παρουσιάζουμε τον τρόπο με τον οποίο θα εκτελείτο η πρώτη φάση του αλγόριθμου στο υποθετικό σενάριο που αναφέραμε πιο πάνω.

Οι 10 σελίδες του αρχικού αρχείου θα μεταφερθούν στην εσωτερική μνήμη ανά τριάδες (αφού M = 3) εκτός από την τελευταία που θα μπει μόνη στην εσωτερική μνήμη. Εκεί θα ταξινομηθούν και στη συνέχεια θα γραφτούν και πάλι στην flash.

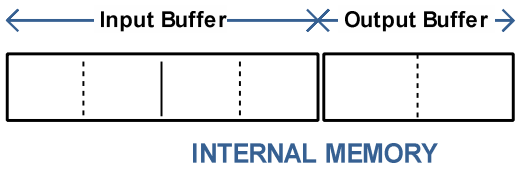


Σχήμα 3.3: External-Merge-Sort (φάση παραγωγής των runs)

Τα παραγόμενα runs θα είναι 4 και θα έχουν μέγεθος 3 (=M) σελίδων έκτος από το τελευταίο που θα έχει μέγεθος μιας σελίδας.

3.2.2 External-Merge-Sort: Φάση Συνένωσης

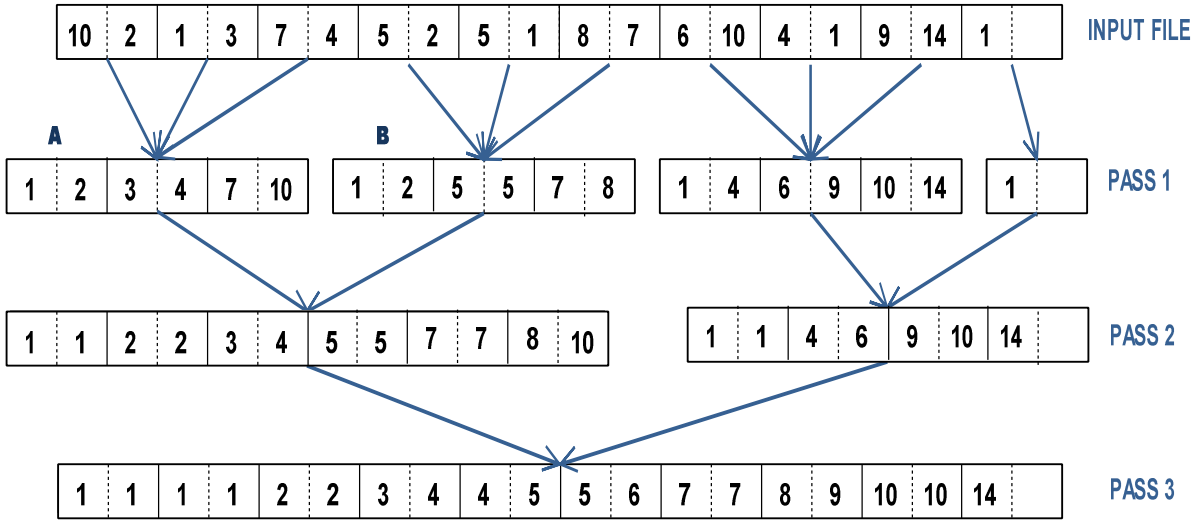
Μετά την παραγωγή των αρχικών runs ξεκινά η δεύτερη φάση του αλγόριθμου η φάση της συνένωσης. Κατά τη φάση αυτή η διαθέσιμη εσωτερική μνήμη των M σελίδων διασπάτε σε Input Buffer μεγέθους M-1 σελίδων και Output Buffer μεγέθους μιας σελίδας. Στο σενάριο που περιγράφουμε αυτό θα είχε ως αποτέλεσμα την διάσπαση της εσωτερικής μνήμης σε ένα Input Buffer μεγέθους 2 σελίδων και ένα Output Buffer μεγέθους μιας σελίδας όπως φαίνεται και στο Σχήμα 3.4.



Σχήμα 3.4 : External-Merge-Sort - Η διάσπαση της εσωτερικής μνήμης σε Input και Output buffers (φάση συνένωσης)

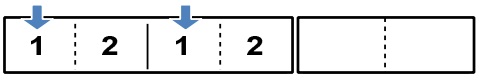
Η φάση της συνένωσης είναι μια αναδρομική διαδικασία όπου η είσοδος της είναι τα παραγόμενα runs από το προηγούμενο pass. Η διαδικασία τελειώνει όταν μετά από την

συνένωση ο αριθμός των παραγόμενων runs είναι 1 (δηλαδή το αρχείο έχει ταξινομηθεί). Έτσι ο συνολικός αριθμός των passes κατά την φάση αυτή είναι $\lceil \log_{M-1}(\frac{N}{M}) \rceil$ και συνολικά χρειαζόμαστε $\lceil \log_{M-1}(\frac{N}{M}) \rceil + 1$ passes και για τις δύο φάσεις του αλγόριθμου. Στο σενάριο που περιγράφουμε, θα χρειαστεί να γίνουν συνολικά 7 συνενώσεις, ενώ η όλη διαδικασία θα διαρκέσει 3 passes.



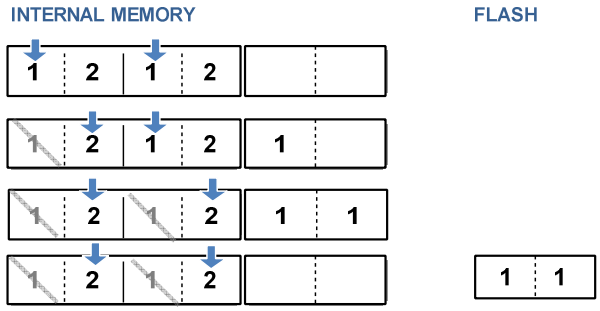
Σχήμα 3.5: External-Merge-Sort - Παράδειγμα εκτέλεσης

Σε κάθε pass, η συνένωση γίνεται σε ομάδες των M-1 runs μέχρι να μην υπάρχουν άλλα runs για συνένωση. Κατά την διαδικασία της συνένωσης, αρχικά διαβάζουμε μια σελίδα από κάθε run και την τοποθετούμε σε μια από τις M-1 διαθέσιμες σελίδες στο Input Buffer. Υπάρχουν επίσης M-1 δείκτες που καθορίζουν την επόμενη υποψήφια προς ταξινόμηση εγγραφή για κάθε σελίδα του Input Buffer και που αρχικά είναι η πρώτη εγγραφή. Για να καταλάβουμε καλύτερα την διαδικασία θα δείξουμε την συνένωση των A και B runs του Σχήματος 3.5. Αρχικά μεταφέρουμε από κάθε run την πρώτη του σελίδα στην εσωτερική μνήμη και η εσωτερική μνήμη έχει την ακόλουθη μορφή.



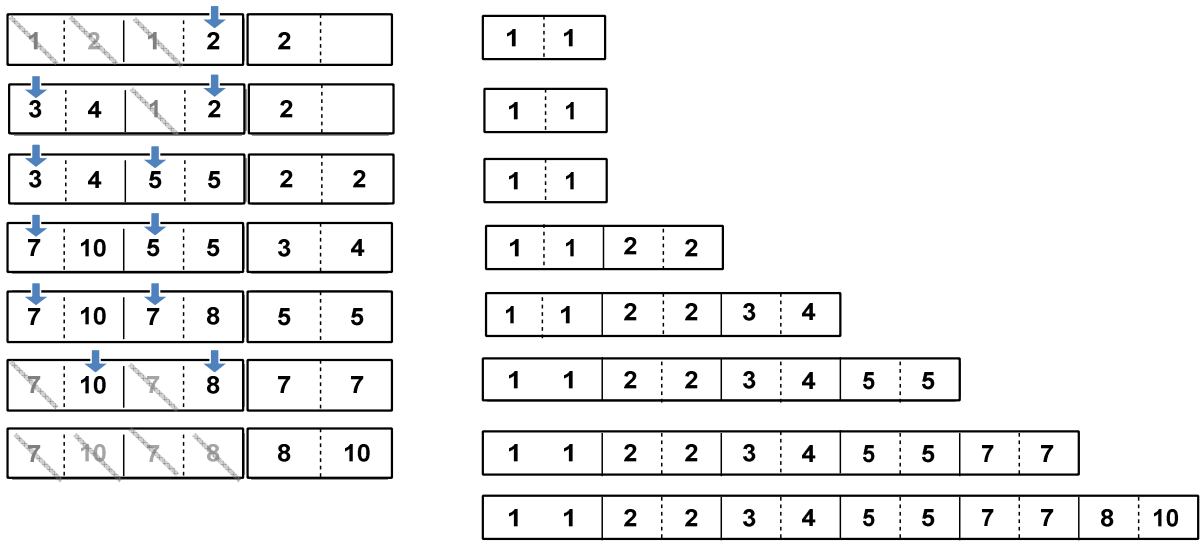
Σχήμα 3.6: External-Merge-Sort - Αρχική κατάσταση εσωτερικής μνήμης κατά την φάση της συνένωσης

Στην συνέχεια επιλέγουμε την εγγραφή με τη μικρότερη τιμή στο κλειδί της ταξινόμησης από τις υποψήφιες προς ταξινόμηση εγγραφές και την προσθέτουμε στη σελίδα του Output Buffer αυξάνοντας ταυτόχρονα και τον αντίστοιχο δείκτη στην επόμενη εγγραφή. Όταν η σελίδα του Output Buffer γεμίσει τότε γράφεται στην μνήμη και μια νέα κενή σελίδα δημιουργείται.



Σχήμα 3.7(α) : External-Merge-Sort - Φάση συνένωσης

Αν σε μια σελίδα δεν υπάρχουν άλλες εγγραφές τότε διαβάζεται από το αντίστοιχο run η αμέσως επόμενη αν υπάρχει σελίδα και ο δείκτης αρχικοποιείται στην πρώτη εγγραφή της σελίδας αυτής Η διαδικασία επαναλαμβάνεται μέχρι να μην υπάρχουν άλλες σελίδες σε κανένα από τα M-1 runs.



Σχήμα 3.7(β) : External-Merge-Sort - Φάση συνένωσης

3.2.3 External-Merge-Sort: I/O Κόστος σε Flash

Το συνολικό I/O κόστος του αλγόριθμου είναι το συνολικός κόστος των αναγνώσεων και των εγγραφών του από και προς την μνήμη flash.

$$C_{ExternalMergeSort}^{I/O} = total\ \#\ of\ \ reads \cdot C_R + total\ \#\ of\ \ writes \cdot C_W$$

Τόσο ο αριθμός των αναγνώσεων και όσο και των εγγραφών για κάθε σελίδα είναι ίσος με τον αριθμό των passes που εκτελεί ο αλγόριθμος, αφού σε κάθε pass διαβάζονται όλες οι σελίδες από την flash (ολόκληρο το αρχείου κατά την πρώτη φάση και σε μορφή runs κατά τη δεύτερη) και ξαναγράφονται όλες στη flash σε μορφή runs. Άρα κάθε σελίδα διαβάζεται και γράφεται $\left\lceil \log_{M-1}\left(\frac{N}{M}\right)\right\rceil + 1$ φορές. Από την στιγμή που υπάρχουν N σελίδες προς ταξινόμηση, τότε # of reads = # of writes = $N \cdot \left(\left\lceil \log_{M-1}(N/M) + 1\right\rceil\right)$. Έτσι έχουμε την ακόλουθη ανάλυση για τη μνήμη Flash.

$$C_{ExternalMergeSort}^{I/O} = N \cdot total\ \#\ of\ \ passes \cdot C_R + N \cdot total\ \#\ of\ \ passes \cdot C_W$$

$$C_{ExternalMergeSort}^{I/O} = N \cdot total\ \#\ of\ \ passes \cdot \left(C_R + C_W\right)$$

$$C_{ExternalMergeSort}^{I/O} = N \cdot \left(\left\lceil \log_{M-1}\left(\frac{N}{M}\right)\right\rceil + 1\right) \cdot \left(C_R + C_W\right)$$

$$C_{ExternalMergeSort}^{I/O} = N \cdot \left(\left\lceil \log_{M-1}\left(\frac{N}{M}\right)\right\rceil + 1\right) \cdot \left(C_R + a \cdot C_R\right)$$

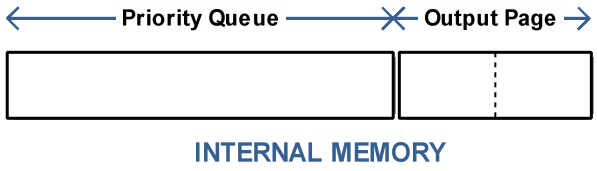
$$C_{ExternalMergeSort}^{I/O} = N \cdot \left(\left\lceil \log_{M-1}\left(\frac{N}{M}\right)\right\rceil + 1\right) \cdot \left((1+a) \cdot C_R\right)$$

3.3 Replacement-Selection-Sort

Αποτελεί μια προσπάθεια βελτιστοποίησης της απόδοσης του External-Merge-Sort με την παραγωγή μεγαλύτερου μεγέθους runs κατά την αρχική φάση. Στην ουσία ακολουθείται μια νέα τεχνική κατά την παραγωγή των αρχικών runs αλλά η φάση της συνένωσης είναι η ίδια με αυτή του External-Merge-Sort. Σύμφωνα λοιπόν με τον Knuth στο [13] η χρήση της εν λόγω τεχνικής και όταν το M >> B, οδηγεί στην παράγωγή κατά προσέγγιση διπλάσιου μεγέθους (= 2M) runs από αυτά που παράγονται κατά την αρχική φάση του αλγόριθμου External-Merge-Sort. Να αναφέρουμε επίσης ότι υπάρχουν δύο πιθανές υλοποιήσεις. Η πρώτη αφορά τη χρήση ενός *Δέντρου Επιλογής (Selection Tree)* και η δεύτερη την χρήση μιας *Ουράς Προτεραιότητας (Priority Queue)*. Εμείς θα παρουσιάσουμε την υλοποίηση με ουρά προτεραιότητας όπως περιγράφεται στο [4].

3.3.1 Replacement-Selection-Sort: Φάση Παραγωγής των Αρχικών Runs

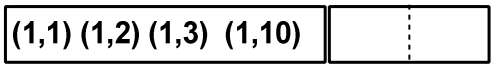
Αρχικά η εσωτερική μνήμη χωρίζεται σε δύο περιοχές. Η πρώτη αποτελεί την περιοχή που χρησιμοποιείται για τη διατήρηση της ουράς προτεραιότητας και η δεύτερη αφορά μια σελίδα η οποία χρησιμοποιείται σαν Output Buffer. Ο αλγόριθμος κάνει επίσης χρήση μιας μεταβλητής έστω KEY_MAX η οποία κρατά τη μεγίστη τιμή που έχει γραφτεί μέχρι στιγμής στην εξωτερική μνήμη για το τρέχον run και μιας δεύτερης μεταβλητής έστω CURRENT_RUN η οποία κρατά τον αριθμό του τρέχοντος run. Η χρήση της πρώτης μεταβλητής θα δώσει την δυνατότητα απόφασης κατά πόσο μια εγγραφή θα ανήκει στο τρέχον run ή θα ανήκει στο επόμενο run ενώ η δεύτερη μεταβλητή είναι αυτή που καθορίζει σε πιο run βρισκόμαστε.



Σχήμα 3.8: : Replacement-Selection-Sort - Η διάσπαση της εσωτερικής μνήμης σε Priority Queue και Output Buffer

Οι εγγραφές αποθηκεύονται στην ουρά προτεραιότητας σε μορφή (Run Number, Record), όπου το Run Number αναφέρεται στον αριθμό του run στο οποίο θα ανήκει

αυτή η εγγραφή και το Record στην ίδια την εγγραφή. Η σειρά προτεραιότητας καθορίζεται αρχικά από το Run Number (major key) και στη συνέχεια από την τιμή του κλειδιού της ταξινόμησης για την συγκεκριμένη εγγραφή (minor key). Σύμφωνα με τον αλγόριθμο αρχικά μεταφέρονται m εγγραφές από το αρχικό αρχείο στην εσωτερική μνήμη και στη συνέχεια δημιουργούμε μια ουρά προτεραιότητας πάνω σε αυτές, βάσει αυτών που αναφέρθηκαν πιο πάνω. Να σημειώσουμε ότι το Run Number θα είναι 1 για τις m αρχικές εγγραφές, αφού σίγουρα θα ανήκουν όλες στο αρχικό run. Επιστρέφοντας στο σενάριο που παρακολουθούμε αρχικά θα γίνει μια μεταφορά 4 εγγραφών από την εξωτερική στην εσωτερική μνήμη και στην συνέχεια θα δημιουργούσαμε την ουρά προτεραιότητας που θα είχε την ακόλουθη μορφή



Σχήμα 3.9 : Replacement-Selection-Sort - Η αρχική κατάσταση του Priority Queue

Θυμίζουμε ότι το αρχείο εισόδου ήταν το ακόλουθο:

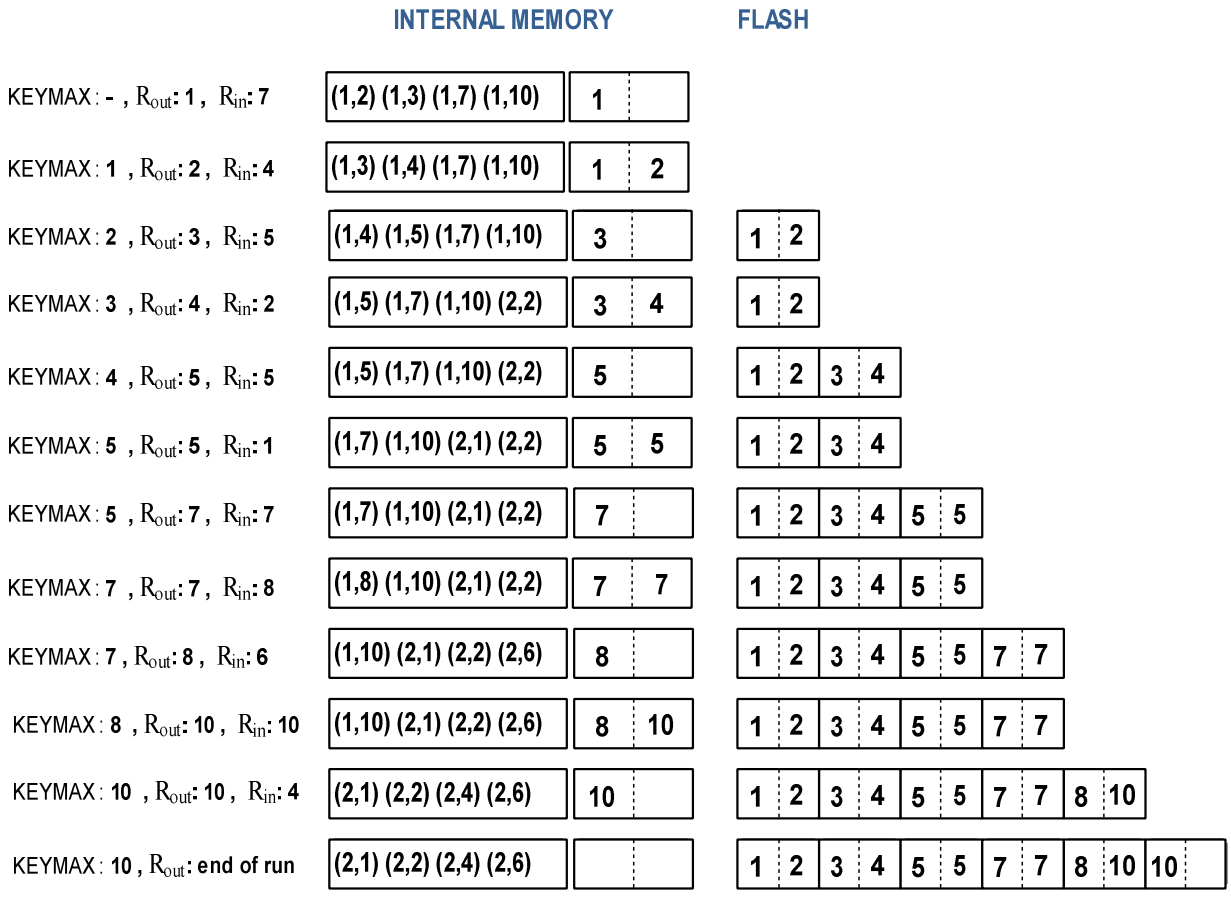
10	2	1	3	7	4	5	2	5	1	8	7	6	10	4	1	9	14	1	

Σχήμα 3.10 : Το προς ταξινόμηση αρχείο

Η όλη διαδικασία προχωρά με μια ακολουθία από εξαγωγές και εισαγωγές εγγραφών στη ουρά προτεραιότητας, μέχρι να μην υπάρχουν άλλες εγγραφές στο αρχικό αρχείο. Κάθε εξαγωγή ακολουθείται από μια εισαγωγή έτσι ώστε να καλυφτεί η κενή θέση στην ουρά. Όταν μια εγγραφή έστω R_{out} εξάγεται από την ουρά, συγκρίνουμε το αντίστοιχο Run Number για αυτή την εγγραφή με την μεταβλητή CURRENT_RUN. Αν ισούνται τότε προσθέτουμε την εγγραφή στη σελίδα του Output Buffer. Σε αντίθετη περίπτωση θα έχουμε το Run Number να ισούται με το CURRENT_RUN + 1 που πρακτικά σημαίνει ότι δεν υπάρχουν άλλες εγγραφές που να ανήκουν στο τρέχον run (θυμίζουμε ότι η προτεραιότητα αφορά αρχικά το Run Number). Στην περίπτωση αυτή, η σελίδα του Output Buffer γράφεται στην μνήμη σαν σελίδα του τρέχοντος run. Στη συνέχεια αυξάνουμε τον αριθμό του CURRENT_RUN, δημιουργούμε μια νέα κενή σελίδα στο Output Buffer και προσθέτουμε την εγγραφή R_{out} σε αυτή. Για κάθε νέα εισαγωγή εγγραφής στο Output Buffer εξισώνουμε την μεταβλητή KEY_MAX με το κλειδί της ταξινόμησης της εγγραφής αυτής. Σε οποιαδήποτε χρονική στιγμή, η σελίδα του Output Buffer γεμίσει τότε γράφεται στην μνήμη σαν σελίδα του τρέχοντος run και

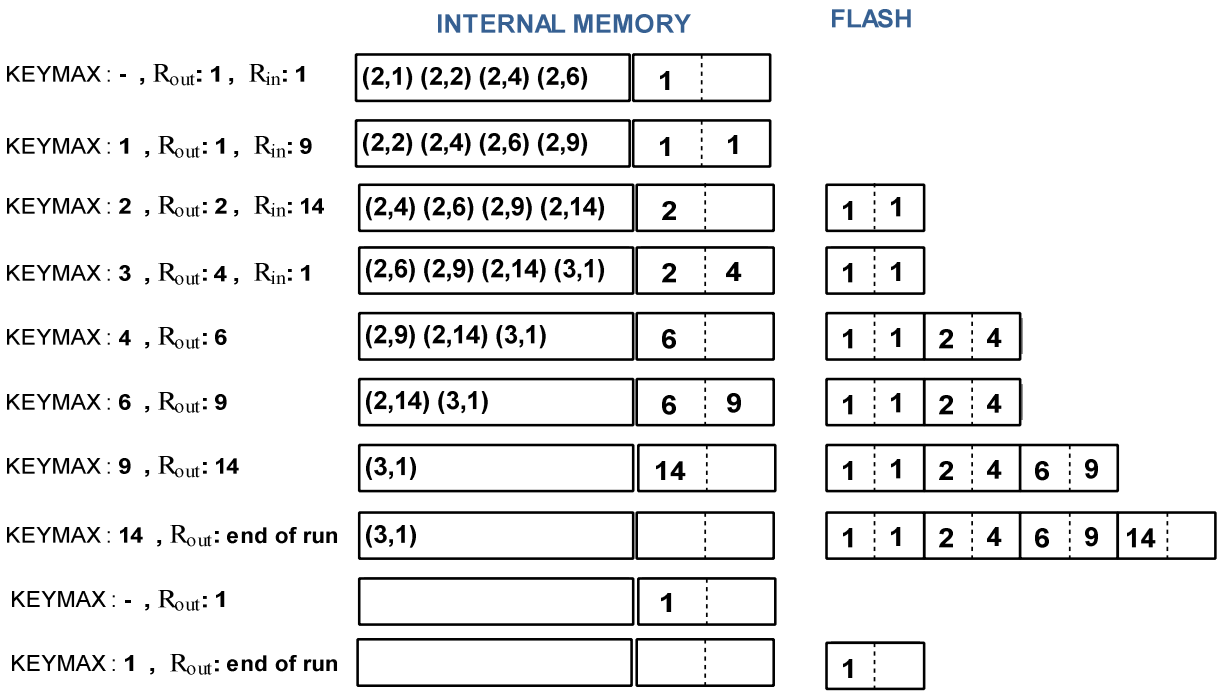
μια νέα κενή σελίδα δημιουργείται. Όταν μια εγγραφή έστω R_{in} θα πρέπει να εισαχθεί στην ουρά, τότε ελέγχουμε την τιμή του κλειδιού ταξινόμησης με την μεταβλητή KEY_MAX. Αν είναι μεγαλύτερη ή ίση με αυτή τότε ορίζουμε το αντίστοιχο Run Number ίσο με το CURRENT_RUN, ενώ αν είναι μικρότερη το Run Number ορίζεται ίσο με το CURRENT_RUN +1.

Ακολούθως παρουσιάζουμε τον τρόπο εκτέλεσης του αλγόριθμου κατά την φάση της παραγωγής των runs για το υποθετικό σεναρίου που παρακολουθούμε. Στο σχήμα 3.10 παρουσιάζουμε την δημιουργία του πρώτου run και στο σχήμα 3.11 την δημιουργία του δευτέρου και τρίτου run.



Σχήμα 3.11: Replacement-Selection-Sort - Παραγωγή πρώτου run

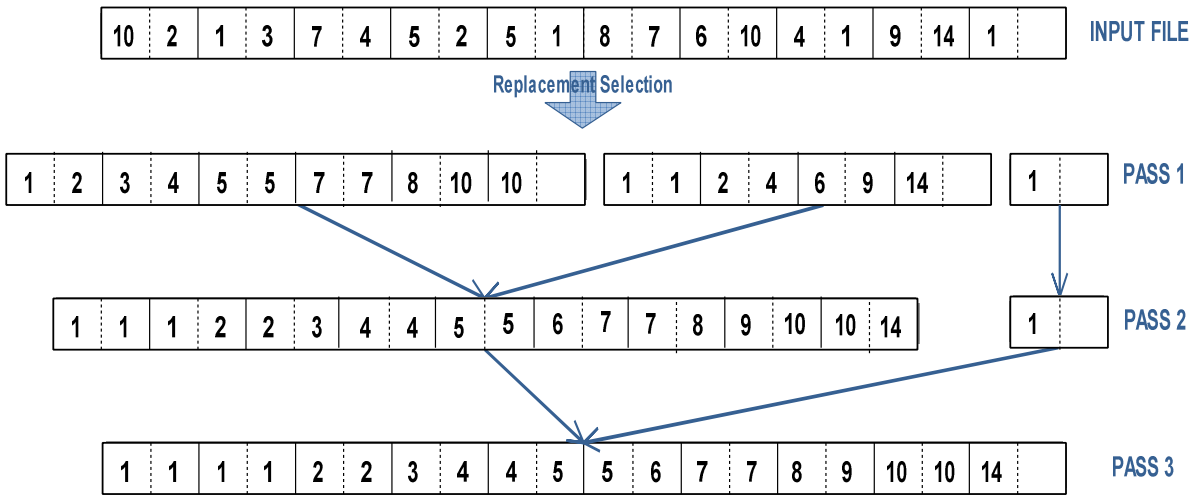
Παρατηρούμε ότι το πρώτο run που δημιουργείται έχει μεγέθους 6 σελίδων πράγμα που σημαίνει ότι είναι διπλάσιο (=2M) από τα runs που παράγονται στην πρώτη φάση του External-Merge-Sort.



Σχήμα 3.12: Replacement-Selection-Sort - Παραγωγή δευτέρου και τρίτου run

3.3.2 Replacement-Selection-Sort: Φάση Συνένωσης

Όπως έχει ήδη αναφερθεί, η φάση της συνένωσης θα είναι η ίδια με αυτή του External-Merge-Sort, μόνο που τα προς συνένωση run έχουν διαφορετικά μεγέθη. Επίσης, όπως αναφέρθηκε και πιο πάνω η χρήση της εν λόγω τεχνικής έχει ως αποτέλεσμα τα παραγόμενα runs να έχουν κατά προσέγγιση μέγεθος 2M. Αν και τα παραγόμενα runs είναι στην πλειοψηφία τους μεγαλύτερα υπάρχει η περίπτωση κάποιες από τις τελευταίες σελίδες των παραγόμενων runs να μην είναι πλήρως γεμάτες. Στο σενάριο που παρακολουθούμε οι τελευταίες σελίδες κάθε παραγόμενου run στο πρώτο pass είναι μισογεμάτες. Αυτό έχει ως αποτέλεσμα την αύξηση των προς ανάγνωση αλλά και προς εγγραφή σελίδων και σίγουρα αυτό είναι ένα επιπλέον I/O κόστος. Τέλος στο σενάριο που παρακολουθούμε η όλη διαδικασία θα διαρκέσει 3 passes όπως φαίνεται και στο ακόλουθο σχήμα.



Σχήμα 3.13: Replacement-Selection-Sort - Παράδειγμα εκτέλεσης

3.3.3 Replacement-Selection-Sort: I/O Κόστος σε Flash

Όπως και στον External-Merge-Sort έτσι και στον Replacement-Selection-Sort, το συνολικό I/O κόστος του αλγόριθμου είναι το συνολικός κόστος των αναγνώσεων και των εγγραφών του από και προς την μνήμη flash. Και πάλι ο αριθμός των αναγνώσεων και των εγγραφών για κάθε σελίδα είναι ίσος με τον αριθμό των passes αλλά το μέγεθος των αρχικών runs είναι κατά προσέγγιση 2M. Από τα πιο πάνω προκύπτει η ακόλουθη ανάλυση για τη μνήμη Flash.

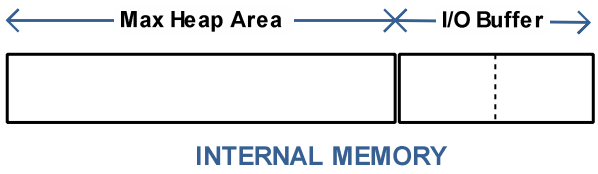
$$C^{I/O}_{\text{Repl. Selection}} = N \cdot \#\text{ of } passes \cdot \left(C_R + C_W\right)$$

$$C^{I/O}_{\text{Repl. Selection}} = N \cdot \left(\left\lceil \log_{M-1}\left(\frac{N}{2M}\right)+1\right\rceil\right) \cdot \left(C_R + C_W\right)$$

$$C^{I/O}_{\text{Repl. Selection}} = N \cdot \left(\left\lceil \log_{M-1}\left(\frac{N}{2M}\right)+1\right\rceil\right) \cdot \left((1+a)\cdot C_R\right)$$

3.4 FAST: Flash-Aware External Sorting Algorithm

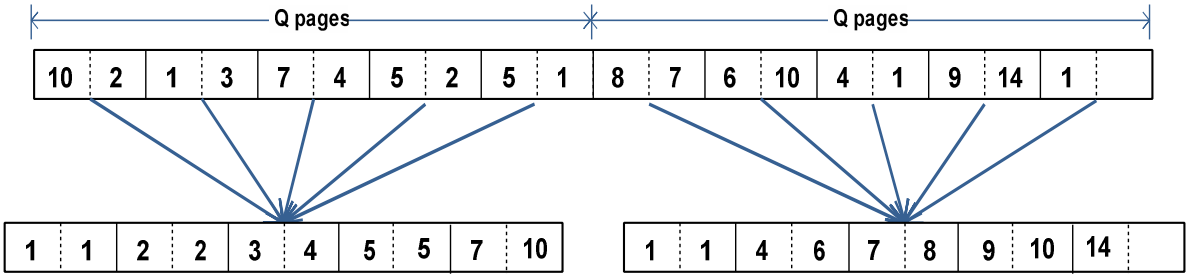
Αποτελεί ένα ακόμα αλγόριθμο συνένωσης, ο οποίος προτάθηκε από τους H. Park και K. Shim [20] και επικεντρώνεται στην εξωτερική ταξινόμηση αρχείων σε συσκευές που κάνουν χρήση της μνήμης flash σαν εξωτερική μνήμη. Σύμφωνα λοιπόν με τους H. Park και K. Shim ο αριθμός των passes εξαρτάται από δυο παράγοντες: το μέγεθος των runs και το μέγεθος του αρχείου εισόδου. Αναπόφευκτα το δεύτερο είναι αμετάβλητο αλλά το πρώτο μπορεί να μεταβληθεί. Έτσι ο αλγόριθμος που πρότειναν παράγει μεγαλύτερου μεγέθους runs τόσο κατά τη φάση της παραγωγής των αρχικών runs, όσο και κατά τη φάση της συνένωσης. Πιο συγκεκριμένα τα παραγόμενα runs της πρώτης φάσης του αλγόριθμου έχουν όλα μέγεθος Q, όπου $M \leq Q \leq N$, και η συνένωση γίνεται σε ομάδες των Q runs. Να θυμίσουμε ότι οι αντίστοιχες τιμές στον External-Merge-Sort ήταν M και M-1. Για να είναι αυτό κατορθωτό, ο αλγόριθμος χωρίζει την διαθέσιμη εσωτερική μνήμη σε δύο περιοχές. Η πρώτη αποτελεί την περιοχή που χρησιμοποιείται για τη διατήρηση ενός σωρού μεγίστων και για την ταξινόμηση των δεδομένων και η δεύτερη αφορά μια σελίδα η οποία χρησιμοποιείται σαν Input και Output Buffer αναλόγως της περιστάσεως.



Σχήμα 3.13: : FAST - Η διάσπαση της Εσωτερικής Μνήμης σε Max Heap Area και Output Buffer

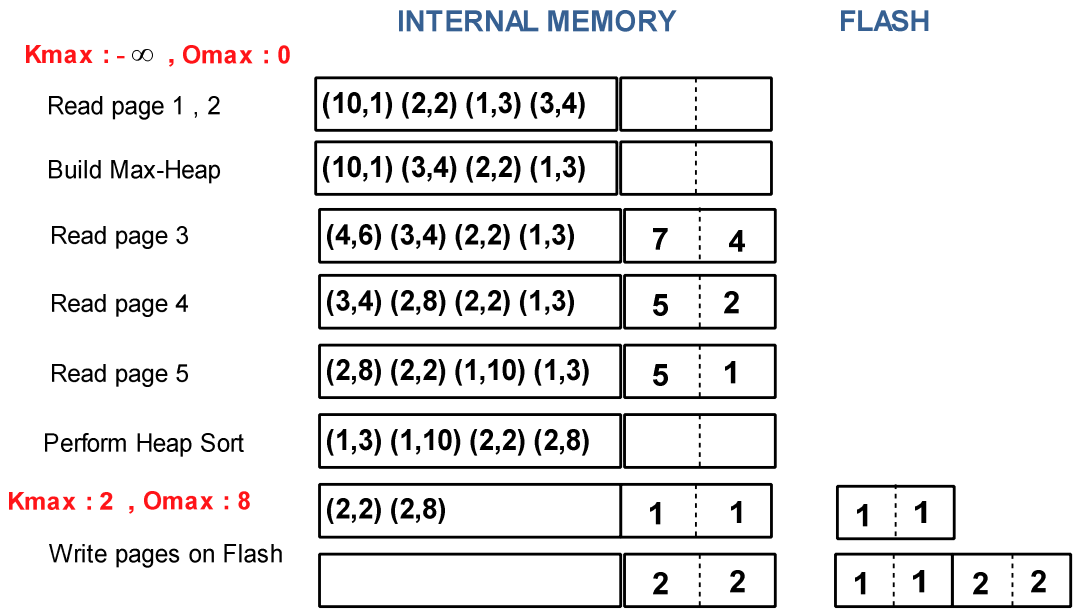
3.4.1 *FAST: Φάση Παραγωγής των Αρχικών Runs*

Κατά την φάση αυτή το αρχείο εισόδου θα χωριστεί σε $\lceil N/Q \rceil$ blocks Q συνεχόμενων σελίδων. Κάθε block θα ταξινομείται ξεχωριστά και θα παραχθούν συνολικά $\lceil N/Q \rceil$ runs μεγέθους Q (το τελευταίο μπορεί να έχει και μικρότερο μέγεθος). Αν θεωρήσουμε λοιπόν ότι Q = 5 για το σενάριο που παρακολουθούμε, το αρχείο εισόδου θα διασπαστεί σε 2 blocks όπου το κάθε ένα περιέχει 5 σελίδες κι έτσι τελικά θα παραχθούν 2 runs μεγέθους 5 σελίδων.



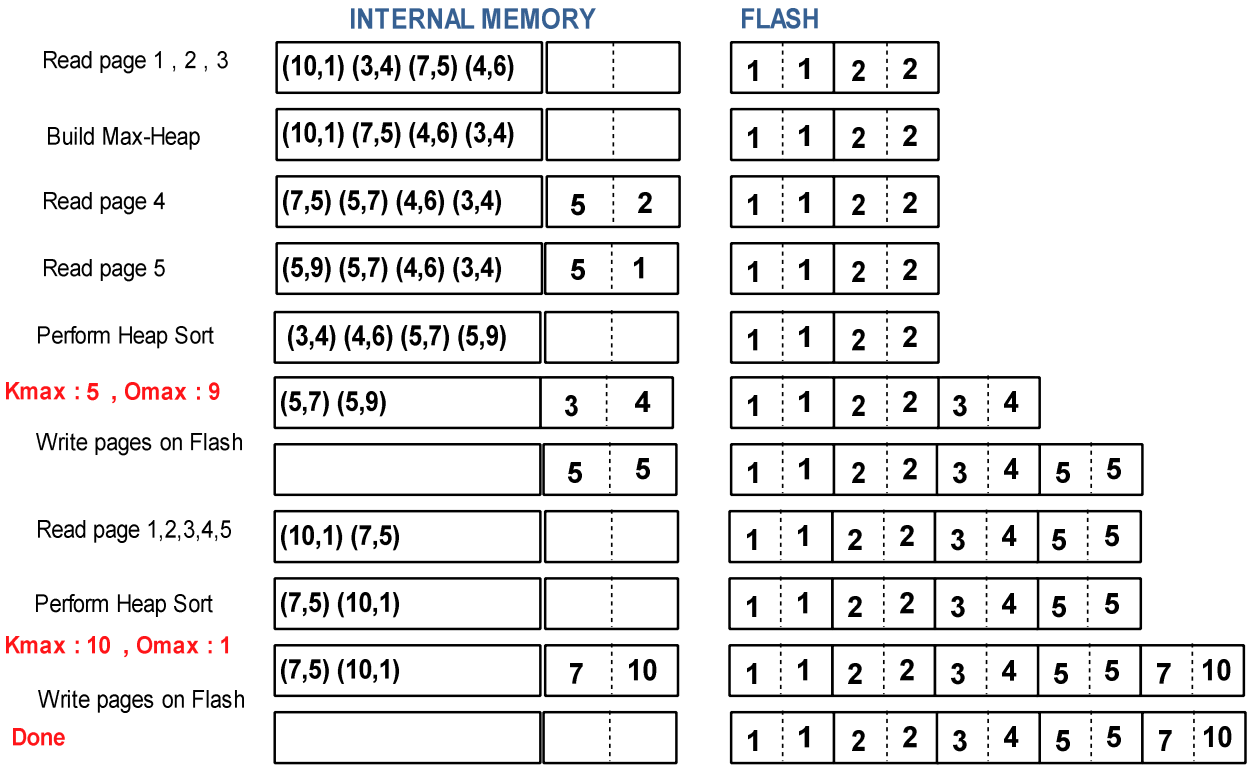
Σχήμα 3.14: FAST - Φάση Παραγωγής Αρχικών Runs

Σύμφωνα λοιπόν με τον αλγόριθμο, διαβάζουμε τις Q συνεχόμενες σελίδες $\lceil Q/M \rceil$ φορές. Σε κάθε επανάληψη, βρίσκουμε τις μικρότερες τιμές που μπορούν να χωρέσουν στην εξωτερική μνήμη αποφεύγοντας κάθε φορά τιμές που έχουμε ήδη γραφτεί σε αυτή από προηγούμενες επαναλήψεις. Για το λόγο αυτό είναι απαραίτητη η χρήση 2 επιπλέον μεταβλητών της Kmax και της Omax, όπως τις ονόμασαν οι H. Park και K. Shim και ενός σωρού μεγίστων. Η Kmax αποθηκεύει την μέγιστη τιμή του κλειδιού ταξινόμησης από τις εγγραφές που είναι ήδη γραμμένες στην εξωτερική μνήμη και η Omax αποθηκεύει την θέση της εν λόγω εγγραφής στο αρχείο εισόδου. Οι εγγραφές αποθηκεύονται στο σωρό σε μορφή (Record, Record Position), όπου το Record αναφέρεται στην ίδια την εγγραφή και το Record Position αναφέρεται στη θέση της εγγραφής στο αρχείο εισόδου. Η σειρά προτεραιότητας καθορίζεται αρχικά από την τιμή του κλειδιού της ταξινόμησης για την συγκεκριμένη εγγραφή (major key) και στη συνέχεια από την Record Position (minor key). Πάμε τώρα να δούμε πώς αυτό θα δουλέψει στο σενάριο που παρακολουθούμε. Θα δείξουμε λοιπόν πως θα ταξινομηθούν οι 5 (=Q) πρώτες σελίδες του αρχείου εισόδου. Αρχικά θέτουμε τις Kmax, Omax ίσες με $-\infty$ και 0 αντίστοιχα. Οσο η περιοχή του σωρού δεν έχει γεμίσει μεταφέρουμε μια προς μια στο I/O Buffer τις σελίδες και αντιγράφουμε στην περιοχή του σωρού όλες τις εγγραφές που είναι μεγαλύτερες από την Kmax, ή αν είναι ίσες με την Kmax αυτές που βρίσκονται σε μεγαλύτερη θέση από την Omax. Όταν η περιοχή του σωρού έχει γεμίσει τότε δημιουργούμε έναν σωρό μεγίστων (αυτό αποτελεί και το πρώτο βήμα του heap sort). Τώρα στην κορυφή βρίσκεται η μεγαλύτερη τιμή από αυτές που περιέχονται στο σωρό.



Σχήμα 3.15(α): FAST – Διαδικασία Παραγωγής Αρχικών Runs

Ακολούθως μεταφέρουμε τις υπόλοιπες σελίδες μια προς μια στο I/O Buffer και για κάθε εγγραφή από κάθε σελίδα ελέγχουμε αν μπορεί να συμπεριληφθεί στο τρέχον run. Για να γίνει αυτό θα πρέπει να είναι είτε μεγαλύτερη από την Kmax, ή αν είναι ίση με την Kmax να βρίσκεται σε μεγαλύτερη θέση από την Omax. Αν η εγγραφή περάσει αυτό τον έλεγχο, ελέγχουμε αν είναι μικρότερη από την τιμή που βρίσκεται στην κορυφή του σωρού. Αν είναι εισάγεται στο σωρό, αλλιώς προχωρούμε στην επόμενη. Όταν διαβαστούν και οι Q σελίδες τότε εκτελούμε και το δεύτερο βήμα του heap sort και ταξινομημένα πλέον δεδομένα γράφονται στην μνήμη σε μορφή σελίδων. Επίσης εξισώνουμε το Kmax με την τιμή της μεγαλύτερης εγγραφής και το Omax με την τιμή του αντίστοιχου Record Position. Επαναλαμβάνουμε το ίδιο μέχρι να μην υπάρχουν άλλες έγγραφες που να μπορούν να μπουν στην περιοχή του σωρού κατά το πρώτο βήμα.

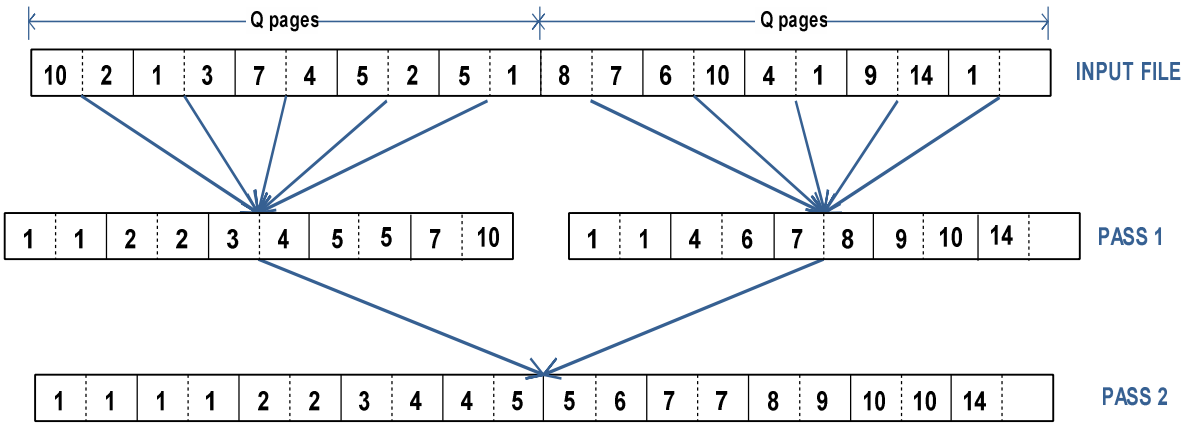


Σχήμα 3.15(β): FAST - Διαδικασία Παραγωγής Αρχικών Runs

3.4.2 *FAST: Φάση Συνένωσης*

Η φάση της συνένωσης γίνεται πάνω σε Q runs και το παραγόμενο run έχει μέγεθος Q x το μέγεθος των αρχικών runs. Η συνένωση επιτυγχάνεται με την χρήση των μεταβλητών Kmax, Omax, και του σωρού μεγίστων όπως ορίστηκαν και πιο πάνω, αλλά και με την χρήση δυο επιπλέον δομών - δύο πινάκων ακέραιων μεγέθους Q. Οι πίνακες αυτοί, L και B όπως τους ονόμασαν οι H. Park και K. Shim, κρατούν πληροφορίες τόσο για την επόμενη προς ανάγνωση σελίδα σε κάθε run αλλά και για τη μεγαλύτερη μέχρι στιγμής εγγραφή που διαβάστηκε από το αντίστοιχο run. Έτσι B[i] και L[i] αναφέρονται στην επόμενη προς ανάγνωση σελίδα για το i-οστό run και στο μεγαλύτερο κλειδί το οποίο διαβάστηκε τελευταίο από το i-οστό run , αντίστοιχα.

Έτσι ο συνολικός αριθμός των passes κατά την φάση αυτή είναι $\lceil \log_Q(N/Q) \rceil$ και συνολικά $\lceil \log_Q(N/Q) \rceil + 1$ χρειαζόμαστε passes και για τις δύο φάσεις του αλγόριθμου. Στο σενάριο που παρακολουθούμε, θα χρειαστεί να γίνουν συνολικά 3 συνενώσεις, ενώ η όλη διαδικασία θα διαρκέσει μόνο 2 passes.



Σχήμα 3.16: Παράδειγμα εκτέλεσης του FAST

3.4.3 *FAST: I/O Κόστος Αλγόριθμου σε Flash*

Σύμφωνα με τους H. Park και K. Shim το I/O κόστος του αλγόριθμου σε μνήμη Flash υπολογίζεται ως εξής:

(α) το I/O κόστος της πρώτης φάσης του αλγόριθμου είναι:

$$C_{rgen}^{I/O} = \left\lceil \frac{N}{Q} \right\rceil \left\lceil \frac{Q}{M_1} \right\rceil \cdot Q \cdot C_R + N \cdot C_W = \left\lceil \frac{N}{Q} \right\rceil \left\lceil \frac{Q}{M_1} \right\rceil \cdot Q \cdot C_R + N \cdot a \cdot C_R$$

$$C_{rgen}^{I/O} = \left(\left\lceil \frac{N}{Q} \right\rceil \left\lceil \frac{Q}{M_1} \right\rceil \cdot Q + N \cdot a \right) \cdot C_R$$

, όπου

$$M_1 = \left\lfloor \left\lfloor \frac{L_p (M-1) - 2 \cdot L_n}{L_r + L_n} \right\rfloor \Big/ \left\lfloor \frac{L_p}{L_r} \right\rfloor \right\rfloor$$

και L_n αποτελεί το μέγεθος σε bytes ενός ακέραιου αριθμού

και (β) το I/O κόστος της δεύτερης φάσης του αλγόριθμου είναι:

$$C_{merge}^{I/O} = \sum_{k=2}^{\lceil \log_Q N \rceil} \left(\left\lceil \frac{N}{Q^k} \right\rceil \cdot \left(\left\lceil \frac{Q^k}{M_2} \right\rceil \cdot Q + Q^k \right) \cdot C_R + N \cdot C_W \right)$$

$$C_{merge}^{I/O} = \sum_{k=2}^{\lceil \log_Q N \rceil} \left(\left\lceil \frac{N}{Q^k} \right\rceil \cdot \left(\left\lceil \frac{Q^k}{M_2} \right\rceil \cdot Q + Q^k \right) \cdot C_R + N \cdot a \cdot C_R \right)$$

όπου

$$M_2 = \left\lfloor \left\lfloor \frac{L_p(M-1) - L_n(2Q+2)}{L_r + L_n} \right\rfloor \Big/ \left\lfloor \frac{L_p}{L_r} \right\rfloor \right\rfloor$$

και L_n αποτελεί το μέγεθος σε bytes ενός ακέραιου αριθμού

Άρα συνολικά έχουμε

$$C_{FAST}^{I/O} = C_{rgen}^{I/O} + C_{merge}^{I/O}$$

$$C_{FAST}^{I/O} = \left(\left\lceil \frac{N}{Q} \right\rceil \left\lceil \frac{Q}{M_1} \right\rceil \cdot Q + N \cdot a \right) \cdot C_R + \left(\sum_{k=2}^{\lceil \log_Q N \rceil} \left(\left\lceil \frac{N}{Q^k} \right\rceil \cdot \left(\left\lceil \frac{Q^k}{M_2} \right\rceil \cdot Q + Q^k \right) \cdot C_R + N \cdot a \cdot C_R \right) \right)$$

3.5 Συζήτηση

Εύκολα μπορεί κάποιος να διακρίνει ότι και οι τρείς αλγόριθμοι ακολουθούν μια κοινή στρατηγική. Στηρίζονται λοιπόν στην λογική της συνένωσης που επιβάλει δύο φάσεις στον αλγόριθμο. Η πρώτη διαρκεί ένα και μόνο pass ενώ η διάρκεια της δεύτερης εξαρτάται από το μέγεθος του αρχείου και της διαθέσιμης εσωτερικής μνήμης καθώς επίσης και από το μέγεθος των παραγόμενων runs στο προηγούμενο pass. Αναπόφευκτα το μέγεθος του αρχείου και της διαθέσιμης μνήμης είναι σταθερά. Η όλη προσπάθεια λοιπόν που γίνεται για βελτίωση της απόδοσης επικεντρώνεται στην παραγωγή μεγαλύτερων runs κατά τη διάρκεια της πρώτης φάσης (στον Replacement-Selection-Sort) ή και των δύο φάσεων (στον FAST).

Όσον αφορά τον External-Merge-Sort, σε γενικές γραμμές χαρακτηρίζεται από απλότητα στην υλοποίηση, ενώ τα παραγόμενα runs σε κάθε pass έχουν όλα το ίδιο μέγεθος το οποίο εξαρτάται από την μεταβλητή M. Ο Replacement-Selection-Sort από την άλλη παράγει κατά προσέγγιση διπλάσιου μεγέθους runs κατά την πρώτη φάση με στόχο την μείωση του συνολικού αριθμού των passes και κατ' επέκταση και του I/O κόστους που προκαλείται. Όμως, ναι μεν τα παραγόμενα runs είναι μεγαλύτερα αλλά λόγω του γεγονότος ότι τα δεν έχουν σταθερό μέγεθος υπάρχει η περίπτωση κάποιες από τις τελευταίες σελίδες των παραγόμενων runs να μην είναι πλήρως γεμάτες. Για

παράδειγμα στο σενάριο που παρουσιάσαμε (Σχήμα 3.13) οι τελευταίες σελίδες κάθε παραγόμενου run στο πρώτο pass είναι μισογεμάτες. Αυτό έχει ως αποτέλεσμα των αύξηση των προς ανάγνωση αλλά και προς εγγραφή σελίδων σε κάθε pass και σίγουρα αυτό είναι ένα επιπλέον I/O κόστος. Όμως στο ίδιο παράδειγμα είδαμε ότι ο αριθμός των συνενώσεων μειώνεται αισθητά σε σχέση με τον External-Merge-Sort, πράγμα που συνεισφέρει θετικά στην απόδοση του αλγόριθμου. Τέλος ο FAST εισάγει μια νέα μεταβλητή την Q, όπου $M \leq Q \leq N$, η οποία καθορίζει το μέγεθος των αρχικών παραγόμενων runs και επηρεάζει το μέγεθος των runs σε κάθε pass. Στην ουσία τα παραγόμενα runs σε κάθε φάση του αλγόριθμου έχουν σταθερό μέγεθος και εξαρτώνται από την Q. Η εύρεση όμως του κατάλληλου Q δεν αποτελεί εύκολη υπόθεση και ο καθορισμός μιας γενικής φόρμουλας που να το κάνει αυτό αποφεύγεται ακόμα και από τους ίδιους τους δημιουργούς του αλγόριθμου.

Κεφάλαιο 4

XSORT: Εξωτερική Ταξινόμηση με Ιστογράμματα

Όπως είδαμε και στο προηγούμενο κεφάλαιο, οι υπάρχοντες αλγόριθμοι εξωτερικής ταξινόμησης ακολουθούν μια κοινή στρατηγική. Στην ουσία στηρίζονται στην λογική της συνένωσης, η οποία αποτελείται από δύο φάσεις: τη φάση της παραγωγής των αρχικών runs και τη φάση της συνένωσης. Στο κεφάλαιο αυτό θα παρουσιάσουμε μια νέα τεχνική που προσπαθεί να ξεφύγει από την λογική αυτή, προτείνοντας τη χρήση ιστογραμμάτων που θα βοηθήσουν στην εκτέλεση της ταξινόμησης σε μόνο δύο passes και θα απαλλάξει τον αλγόριθμο από την πολυπλοκότητα των πολλών συγκρίσεων.

4.1 Η Βασική Ιδέα

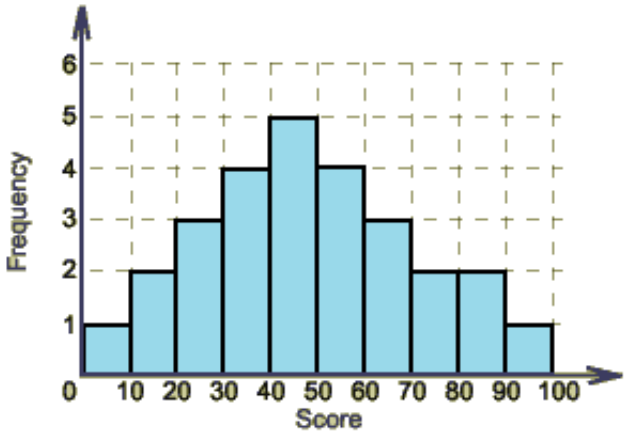
Η νέα τεχνική που θα παρουσιάσουμε προτείνει την χρήση ιστογραμμάτων και την ταξινόμηση των δεδομένων σε buckets. Ο όρος ιστόγραμμα προέρχεται από την Στατιστική και απεικονίζει γραφικά μια κατανομή συχνοτήτων των δεδομένων ενός συνόλου. Σε ένα ιστόγραμμα τα διαστήματα ονομάζονται κλάσεις και ο αντίστοιχος αριθμός που παρουσιάζει τον αριθμό των εμφανίσεων σε κάθε κλάση ονομάζεται *Κάδος (Bin ή Bucket).*

Αν θεωρήσουμε ότι έχουμε τα αποτελέσματα των εξετάσεων μιας ομάδας φοιτητών με πιθανή βαθμολόγηση από το 0 - 100, τότε μπορούμε να χωρίσουμε το διάστημα αυτό σε 10 κλάσεις, όπου κάθε μια καλύπτει διάστημα 10 πιθανών βαθμολογιών. Ο πίνακας συχνοτήτων θα είχε την ακόλουθη μορφή.

Κλάση	0 - 9	10 - 19	20 - 29	30 - 39	40 - 49	50 - 59	60 - 69	70 - 79	80 - 89	90 - 99
Συχνότητα	1	2	3	4	5	4	3	2	2	1

Πίνακας 4.1: Πίνακας συχνοτήτων βαθμολογίας μαθητών

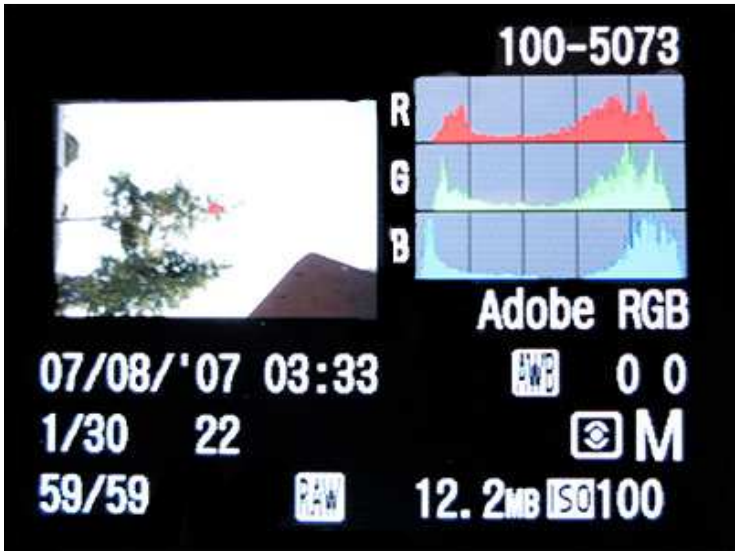
Και το αντίστοιχο ιστόγραμμα θα είχε αυτή τη μορφή, όπου στον άξονα των x εμφανίζονται οι κλάσεις και στον άξονα των y το περιεχόμενο του αντίστοιχου bucket.



Σχήμα 4.1: Ιστόγραμμα βαθμολογίας μαθητών

Η χρήση ιστογραμμάτων δεν είναι κάτι καινούργιο στο χώρο των υπολογιστών. Στο τομέα των βάσεων δεδομένων τα ιστογράμματα βρίσκουν ευρεία χρήση κυρίως σε θέματα Optimization (π.χ., Query Optimization) και Database Management. [11].

Επίσης τα ιστογράμματα χρησιμοποιούνται για την επεξεργασία εικόνων και παίζουν καθοριστικό ρόλο στον τρόπο λειτουργίας της ψηφιακής φωτογραφικής μηχανή. Το ιστόγραμμα μιας εικόνας είναι η γραφική παράσταση που απεικονίζει την αριθμητική κατανομή των διαφόρων χρωμάτων στην εικόνα και περιέχει σημαντικές πληροφορίες για την ποιότητά της, έτσι ώστε σε εφαρμογές επεξεργασίας εικόνας να μπορούμε αυτόματα να αντιληφθούμε κακοφωτισμένες, υπερφορτισμένες και με χαμηλό contrast εικόνες [7].



Σχήμα 4.2: RGB ιστογράμματα σε ψηφιακές βιντεοκάμερες

Και στις δύο αυτές περιπτώσεις το ιστόγραμμα χρησιμοποιείται σαν φορέας χρήσιμων μεταπληροφοριών που κάνουν την λήψη μιας απόφασης ευκολότερη και ταχύτερη. Το ερώτημα που μπαίνει εδώ είναι αν θα μπορούσαν τέτοιου είδος μεταπληροφορίες να βοηθήσουν σε μια αποδοτικότερη ταξινόμηση. Επίσης η απόδοση κάποιων από τους αλγόριθμους ταξινόμησης εξαρτάται από την κατανομή που έχουν τα προς ταξινόμηση δεδομένα, άρα εύλογα υποψιαζόμαστε ότι κατά κάποιον τρόπο υπάρχει μια συσχέτιση των δύο. Το ιστόγραμμα παρουσιάζει αυτή την κατανομή και ίσως αυτό να μπορούσε να βοηθήσει στην ταξινόμηση.

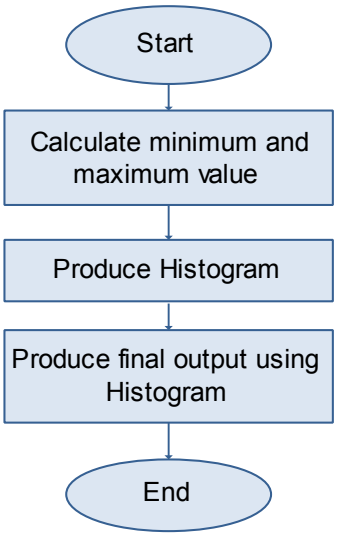
Επιστρέφοντας στην προτεινόμενη ιδέα, αρχικά μετατρέπουμε την διαθέσιμη εσωτερική μνήμη σε ένα πίνακα ακέραιων προκαθορισμένου μεγέθους που θα αποτελούν τα buckets του ιστογράμματος. Κάθε ένα από τα διαθέσιμα buckets θα αντιστοιχίζεται με μια διακριτή τιμή στο πεδίο τιμών του κλειδιού της ταξινόμησης

(δηλαδή σε μια κλάση), αρχίζοντας από την μικρότερη και προχωρώντας προς την μεγαλύτερη τιμή. Αν για παράδειγμα το πεδίο τιμών του κλειδιού ταξινόμησης για ένα αρχείο είναι [1,20], τότε στο bucket 0 αντιστοιχούμε την τιμή 1, στο bucket 1 την τιμή 2, στο bucket 2 την τιμή 3 κ.ο.κ. Για να αποφανθούμε το πεδίο τιμών του κλειδιού ταξινόμησης μπορούμε να διατρέξουμε μια φορά ολόκληρο το αρχείο (δηλ file scan) και να βρούμε την μικρότερη και μεγαλύτερη τιμή που μπορεί να πάρει το εν λόγω χαρακτηριστικό. Το κάθε bucket θα αποτελεί ένα μετρητή που θα καθορίζει τον αριθμό των εμφανίσεων της διακριτής τιμή που του αντιστοιχεί στο προς ταξινόμηση αρχείο. Έτσι διατρέχοντας μια ακόμα φορά ολόκληρο το αρχείο και αυξάνοντας την τιμή του αντίστοιχου bucket σε κάθε εμφάνιση της αντίστοιχης διακριτής τιμής, μπορούμε να δημιουργήσουμε ένα ιστόγραμμα.

Το ιστόγραμμα που κατασκευάσαμε, περιέχει σημαντικές μεταπληροφορίες που θα μας βοηθήσουν στην ταξινόμηση του αρχείου. Αυτό που εννοούμε εδώ είναι ότι βάση του ιστογράμματος είμαστε πλέον σε θέση να δώσουμε μια αφηρημένη περιγραφή του τελικού αποτελέσματος. Αν για παράδειγμα η ανάθεση των buckets είναι αυτή που αναφέραμε στο προηγούμενο παράδειγμα και το bucket[0] = 5, το bucket[1] = 0, και το bucket[2] = 20 κτλ, τότε ξέρουμε ότι στο τελικό αποτέλεσμα υπάρχουν αρχικά 5 1 ακολουθούμενοι από 20 3 κτλ. Αν το ζητούμενο είναι η ταξινόμηση μόνο του κλειδιού τότε αυτό μπορεί να γίνει εύκολα, διατρέχοντας το ιστόγραμμα και γράφοντας στο αρχείο εξόδου την διακριτή τιμή που αντιστοιχεί στο bucket i bucket[i] φορές. Έτσι με δύο file scans (κόστους 2 N αναγνώσεων σελίδων) και N εγγραφές σελίδων έχουμε δημιουργήσει το τελικό αποτέλεσμα. Όμως τις πλείστες των περιπτώσεων η ταξινόμηση αφορά ολόκληρη την πληροφορία και όχι μόνο το κλειδί. Έτσι θα πρέπει να βρούμε τον τρόπο να μεταφέρουμε και την υπόλοιπη πληροφορία στο ταξινομημένο αρχείο.

Η αρχική προσέγγιση ήταν να διατρέχουμε το αρχείο αναζητώντας τις bucket[i] εμφανίσεις της τιμής που αντιστοιχεί σε κάθε bucket i του ιστογράμματος. Η κάθε αναζήτηση θα σταματούσε με τη συμπλήρωση του απαιτούμενου αριθμού εμφανίσεων. Αν και η χρήση του ιστογράμματος στις πλείστες των περιπτώσεων μας βοηθά να απορρίψουμε μια μεγάλη ομάδα σελίδων όταν βέβαια έχουμε συμπληρώσει τον απαιτούμενο αριθμό εμφανίσεων, εντούτοις στην χειρότερη περίπτωση θα πρέπει κάθε

φορά να διατρέξουμε ολόκληρο το αρχείο αφού η τιμή που αναζητούμε μπορεί να βρίσκεται σε οποιαδήποτε σελίδα του προς ταξινόμηση αρχείου.



Σχήμα 4.3: Η αρχική ιδέα

Όπως έχουμε αναφέρει και σε προηγούμενο κεφάλαιο, παρατηρείται μια ασυμμετρία στο κόστος ανάγνωσης και εγγραφής μιας σελίδας ($C_W = \alpha\ C_R$) στην μνήμη flash. Αν και η όλη διαδικασία περιορίζει στο ελάχιστο τις χρονοβόρες έγγραφες (μόνο N εγγραφές), προκαλεί όμως υπερβολικά μεγάλο αριθμό αναγνώσεων πράγμα που εκτοξεύει το I/O κόστος. Τα πράγματα είναι ακόμα χειρότερα στις περιπτώσεις όπου τόσο το μέγεθος του ιστογράμματος όσο και του αρχείου είναι πολύ μεγάλα. Ναι μεν θα πρέπει να μειώσουμε τις εγγραφές στην μνήμη flash αλλά θα πρέπει αυτό να μην αντιστοιχεί σε υπερβολική αύξηση των αναγνώσεων που μπορεί να προκαλέσει άκρως αντίθετα αποτελέσματα. Χρειαζόμαστε λοιπόν ένα μηχανισμό του θα μας επιτρέπει να αποφεύγουμε αχρείαστες αναγνώσεις σελίδων, απορρίπτοντας εύκολα κάποιες σελίδες του αρχείου κατά την αναζήτηση της επόμενης τιμής.

Αν τα προς ταξινόμηση δεδομένα ήταν χωρισμένο σε ταξινομημένες ομάδες τότε θα ήμασταν σε θέση κρατώντας κάποιες επιπλέων πληροφορίες για κάθε ομάδα να υπολογίσουμε σε ποια από τις σελίδες της ομάδας πιθανόν να υπάρχει η τιμή που αναζητούμε. Αν για παράδειγμα ένα αρχείο 1000 σελίδων ήταν χωρισμένο σε 10 ταξινομημένες ομάδες και εμείς αναζητούσαμε το μοναδικό 1 που υπάρχει σε αυτό δεν θα χρειαζόταν να διαβάσουμε και τις 1000 σελίδες του αρχείου αλλά το μέγιστο 10 από

αυτές για να το βρούμε. Θα ήταν ίσως καλύτερα κατά την αρχική φάση του αλγόριθμου, μαζί με τον εύρεση της μέγιστης και ελάχιστης τιμής να ταξινομούσαμε τα δεδομένα σε runs, όπως γινόταν και στους προηγούμενους αλγόριθμους. Έτσι κάθε run θα αποτελούσε μια ομάδα ταξινομημένων δεδομένων. Οι μικρότερες τιμές θα εμφανίζονται στις πρώτες σελίδες ενώ οι μεγαλύτερες στις τελικές σελίδες του run.

4.2 Ο Αλγόριθμος Εξωτερικής Ταξινόμησης XSORT

Ο αλγόριθμος που προτείνουμε έχει το όνομα XSORT, κάνει χρήση της πιο πάνω λογικής και αποτελείται από τρεις φάσεις. Στόχος του αλγόριθμου είναι να κρατήσει σταθερό τον αριθμό των χρονοβόρων εγγραφών στη μνήμη flash, εκτελώντας όσο το δυνατότερο λιγότερες λιγότερο ακριβές αναγνώσεις. Στην ουσία η όλη διαδικασία θα διαρκεί 2 passes συν κάποια επιπλέον file scans για την παραγωγή και διαχείριση του ιστογράμματος.

Κατά την πρώτη φάση (*Φάση της Παραγωγής των Αρχικών Runs και Υπολογισμού της Μεγίστης και Ελάχιστης Τιμής του Ιστογράμματος*) δημιουργείται ένας αριθμός από runs και ταυτόχρονα βρίσκουμε την μέγιστη και ελάχιστη τιμή του κλειδιού ταξινόμησης. Αν ο αριθμός των παραγόμενων runs είναι ίσος με 1, τότε έχουμε τελειώσει με την ταξινόμηση, αλλιώς προχωρούσε στην εκτέλεση και των επόμενων φάσεων. Κατά τη δεύτερη φάση (*Φάση της Παραγωγής του Ιστογράμματος*) παράγεται ένα ιστογράμματα βάσει της μέγιστης και ελάχιστης τιμής που υπολογίστηκε στην προηγούμενη φάση. Τέλος στην τρίτη και τελευταία φάση (*Φάση της Παραγωγής του Τελικού Αποτελέσματος*) παράγουμε το ταξινομημένο αρχείο, κάνοντας χρήση του παραγόμενου ιστογράμματος από τη δεύτερη φάση του αλγόριθμου και μιας τεχνικής που διαχειρίζεται τα παραγόμενα από την πρώτη φάση του αλγόριθμου runs χωρίζοντας τα σε νεκρά και ενεργά με σκοπό την αποφυγή αχρειαστων επιπλέον αναγνώσεων. Στο Σχήμα 4.4 παρουσιάζουμε την γενική μορφή του αλγόριθμου XSORT σε ψευδοκώδικα.

Algorithm 1 XSORT

Input: The file to be sorted
Output: The sorted file

```
1:  procedure XSORT(file)
2:      Set system globals  (N,M .. )
3:      Runs = GenInitialRunsAndDefHistogramRage ( file, &min, &max)
4:      if (Runs > 0)
5:          histogamFile = CreateHistogram( file, min, max)
6:          ProduceFinalOuptput(file, histogamFile)
7:      end if
8:  end procedure
```

Σχήμα 4.4: Ο αλγόριθμος XSORT

Σύμφωνα με τον πιο πάνω ψευδοκώδικα, η είσοδος του αλγόριθμου είναι το προς ταξινόμηση αρχείο και η έξοδός του το ταξινομημένο αρχείο. αρχικά αρχικοποιούνται όλες οι παράμετροι του συστήματος (N,M κτλ) και στη συνέχεια εκτελούνται οι τρείς φάσεις του. Στην παρούσα εργασία θα παρουσιάσουμε δύο παραλλαγές του XSORT: τον XSORT-RSEL και τον XSORT-FAST, όπως θα δούμε στη επόμενη παράγραφο.

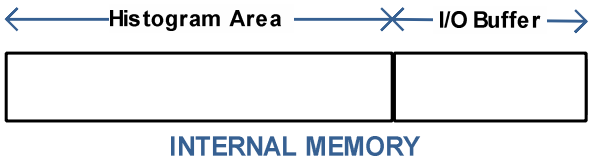
4.2.1 *XSORT: Φάση της Παραγωγής των Αρχικών Runs και Υπολογισμού της Μεγίστης και Ελάχιστης Τιμής του Ιστογράμματος*

Όπως αναφέραμε η πρώτη φάση του αλγόριθμου είναι ο καθορισμός του πεδίου τιμών (η εύρεση δηλαδή της μέγιστης και ελάχιστης τιμής που μπορεί να πάρει το κλειδί της ταξινόμησης) και η δημιουργία των αρχικών runs. Καλό θα ήταν τα παραγόμενα runs να είχαν όσο το δυνατό μεγαλύτερο μέγεθος γιατί κάτι τέτοιο θα βοηθούσε στην τελευταία φάση του αλγόριθμου. Θα μπορούσαμε λοιπόν να χρησιμοποιήσουμε την λογική παραγωγής των αρχικών runs των υπαρχόντων αλγόριθμων δημιουργώντας στην ουσία ένα υβριδικό αλγόριθμο. Θα υπάρχουν δύο παραλλαγές του XSORT: ο XSORT-RSEL που αφορά τον προτεινόμενο αλγόριθμο που κάνει χρήση της λογικής του αλγόριθμου Replacement-Selection-Sort για την δημιουργία των αρχικών runs ως και τον XSORT-FAST που κάνει χρήση της λογικής του αλγόριθμου FAST για την δημιουργία των αρχικών runs. Έτσι σε κάθε περίπτωση υλοποιείται η πρώτη φάση του αντίστοιχου αλγόριθμου, όπως παρουσιάστηκαν στο κεφάλαιο 3 με μια επιπλέον

προσθήκη της εύρεσης της ελάχιστης και της μέγιστης τιμής του κλειδιού της ταξινόμησης κατά τη διάρκεια της δημιουργίας των runs.

4.2.2 *XSORT: Φάση της Παραγωγής του Ιστογράμματος*

Κατά την φάση της Παραγωγής των Ιστογραμμάτων, η διαθέσιμη μνήμη χωρίζεται σε δύο περιοχές. Η πρώτη αφορά την περιοχή όπου θα είναι το ιστόγραμμα και η δεύτερη είναι η περιοχή του Input και Output Buffer που θα χρησιμοποιείται αρχικά για ανάγνωση σελίδων του αρχείου εισόδου και ακολούθως για τη δημιουργία σελίδων ιστογράμματος και αποθήκευσή τους στην εξωτερική μνήμη.



Σχήμα 4.5: : Η διάσπαση της εσωτερικής μνήμης σε περιοχή ιστογράμματος, Input Buffer και Output Buffer αρχικά για ανάγνωση σελίδων του αρχείου και ακολούθως για δημιουργία σελίδων ιστογράμματος

Μια σελίδα ιστογράμματος αποτελείται από εγγραφές του τύπο (id, freq) όπου το id είναι η τιμή και το freq είναι οι φορές εμφάνισης της συγκεκριμένης τιμής στο αρχείο εισόδου. Υπάρχει επίσης μια συνάρτηση f(x) η οποία αντιστοιχίζει μια τιμή από το αρχείο εισόδου με το bucket του ιστογράμματος στο οποίο αντιστοιχεί. Στο Σχήμα 4.6 που ακολουθεί παρουσιάζουμε τον ψευδοκώδικα της φάσης αυτής.

Algorithm 2 CreateHistogram

Input: The file to be sorted, minimum histogram value, maximum histogram value
Output: A file pointer to Histogram file

```
1:  procedure CreateHistogram( file, min, max)
2:      // allocate Histogram space
3:      Histogram[((M-1) * PageSize) / sizeof(int)]
4:      // produce histogram range by range
5:      Hmin = 0
6:      while (Hmin < (max-min)+1 )
7:              If (Hmin + ((M-1) * PageSize) / sizeof(int) < (max-min)+1) then
```

```
 8:              Hmax = Hmin + ((M-1) * PageSize) / sizeof(int)
 9:          else
10:              Hmax = (max-min)+1
11:          end if
12:          allocate an Input Buffer Page
13:          // utilize histogram values
14:          Histogram[0…((M-1) * PageSize) / sizeof(int) -1 ] = 0
15:          // do a file scan and produce a histogram for the current range
16:          for currentPage = 0 to N do
17:              Read  next page from file into Input Buffer
18:              for each record in page do
19:                  If (record value ≥ Hmin and record value ≤ Hmax ) then
20:                      Histogram[f(record value )-f(Hmin)] ++
21:                  end if
22:              end for
23:          end for
24:          delete Input Buffer Page
25:          allocate a Histogram Output Buffer Page
26:          // write histogram pages in memory
27:          for i = 0 to ((M-1) * PageSize) / sizeof(int) do
28:              if (Histogram Output Buffer Page is full ) then
29:                  Write Histogram Output Buffer Page at the end of Histogram_File
30:                  reallocate a new Histogram Output Buffer Page
31:              end if
32:              if (Histogram[i] != 0 ) then
33:                  put ( i + Hmin , Histogram[i]) pair into Histogram Output Buffer Page
34:              end if
35:          end for
36:          if (Histogram Output Buffer Page is not empty ) then
37:              Write Histogram Output Buffer Page at the end of Histogram_File
38:          end if
39:          delete Histogram Output Buffer Page
40:          Hmin = Hmax
41:      end while
42:      return Histogram_File
43:  end procedure
```

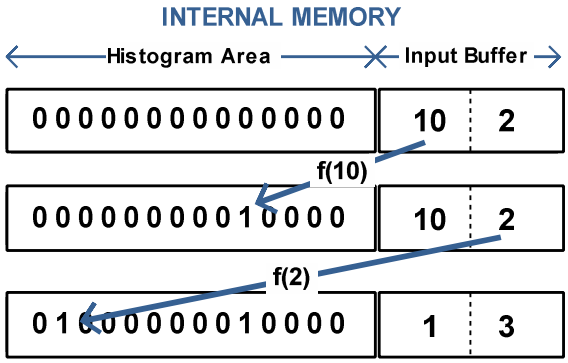
Σχήμα 4.6: Ψευδοκώδικας της διαδικασίας CreateHistogram

Ας δούμε όμως ένα παράδειγμα εκτέλεσης. Επιστρέφοντας στο παράδειγμα που ακολουθούσαμε στο κεφάλαιο 3 θυμίζουμε ότι το αρχείο εισόδου ήταν το ακόλουθο:

10	2	1	3	7	4	5	2	5	1	8	7	6	10	4	1	9	14	1	

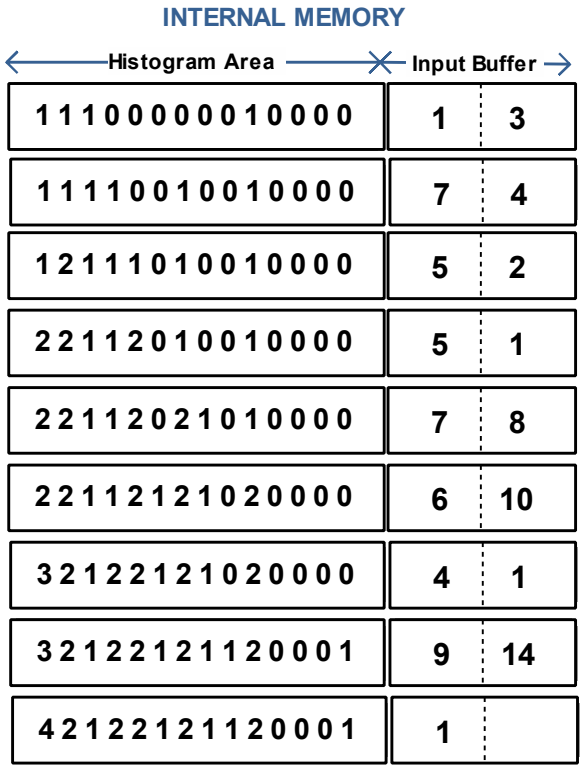
Σχήμα 4.7 : Το προς ταξινόμηση αρχείο

Αρχικά χωρίζουμε την εσωτερική μνήμη σε μια περιοχή όπου θα είναι ο πίνακας με τις πληροφορίες του ιστογράμματος αρχικοποιώντας με 0 όλους τους *Κάδους (Bucket)* του και στην περιοχή του Input Buffer, η οποία θα χρησιμοποιείται για ανάγνωση σελίδων του αρχείου εισόδου (γραμμές 3, 12 και 14 του ψευδοκώδικα). Στην συνέχεια προχωράμε με τη δημιουργία του ιστογράμματος (γραμμές 16-23 του ψευδοκώδικα). Έτσι αρχικά έχουμε την ανάγνωση της πρώτης σελίδας του αρχείου εισόδου στο Input Buffer και για κάθε εγγραφή της αυξάνουμε την τιμή του αντίστοιχου bucket στο ιστόγραμμα όπως πιο κάτω.



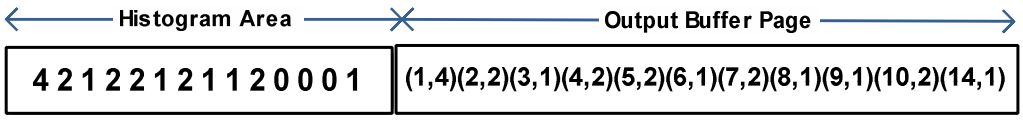
Σχήμα 4.8(α) : Η αντιστοίχιση μιας τιμής αρχείου και ενός bucket ιστογράμματος

Συνεχίζουμε να διαβάζουμε σελίδα - σελίδα από το αρχείο μέχρι να μην υπάρχουν άλλες προς ανάγνωση σελίδες σε αυτό. Ταυτόχρονα αυξάνουμε τις αντίστοιχες τιμές και στο τέλος έχουμε το ιστόγραμμα.



Σχήμα 4.8(β) : Η δημιουργία του ιστογράμματος

Με το τέλος αυτής της διαδικασίας η περιοχή του Input Buffer μετατρέπεται σε περιοχή του Output Buffer που χρησιμοποιείται για τη δημιουργία σελίδων ιστογράμματος και αποθήκευσή τους στην εξωτερική μνήμη (γραμμές 24-38 του ψευδοκώδικα). Έτσι αρχικά μετατρέπουμε το ιστόγραμμα σε σελίδες ιστογράμματος όπου κάθε εγγραφή είναι του τύπο (id, freq) με το id να είναι η τιμή και το freq να είναι οι φορές εμφάνισης της συγκεκριμένης τιμής στο αρχείο εισόδου όπως αναφέρθηκε νωρίτερα ενώ η κάθε γεμάτη σελίδα αποθηκεύεται στην μνήμη flash. Να σημειώσουμε ότι τιμές που παρουσιάζονται 0 φορές (στο παράδειγμα οι 11, 12, 13) δεν θα εγγραφούν στην σελίδα του ιστογράμματος, και αυτό γιατί η εμφάνιση 0 φορών μιας τιμής στο αρχείο είναι ένδειξη απουσίας της συγκεκριμένης τιμής από αυτό. Στο Σχήμα 4.8γ που ακολουθεί φαίνεται αυτή η μετατροπή του ιστογράμματος σε σελίδα ιστογράμματος.

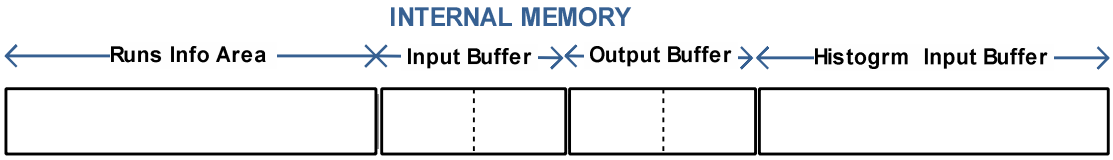


Σχήμα 4.8(γ) : Η μετατροπή του ιστογράμματος σε σελίδα ιστογράμματος για εγγραφή στη εξωτ. μνήμη

Παρατηρείστε ότι στο ξευδοκώδικα που παρουσιάζουμε στο Σχήμα 4.6 η όλη διαδικασία που περιγράφηκε πιο πάνω βρίσκεται σε ένα βρόγχο. Αυτό συμβαίνει γιατί προσπαθούμε να καλύψουμε την περίπτωση όπου το πεδίο ορισμού άρα κατ' επέκταση το μέγεθος του ιστογράμματος ξεπερνά την διαθέσιμη μνήμη. Αυτό μπορεί να επιλυθεί με τη διάσπαση του πεδίου ορισμού σε μικρότερα πεδία ικανά να χωρέσουν στην κύρια μνήμη και εκ νέου αναγνώσεις ολόκληρου του αρχείου και γενικά εκ νέου εκτέλεση της διαδικασίας σε άλλο πεδίο ορισμού. Για να μην χαθεί η πληροφορία σε κάθε επανάληψη γράφουμε τα αποτελέσματα σε μορφή σελίδων ιστογράμματος στην μνήμη flash όπως είδαμε να συμβαίνει και πιο πάνω. Βέβαια αυτό προϋποθέτει επιπλέον κόστος το οποίο είναι μικρό δεδομένου ότι δεν θα υπάρχουν πολλές τέτοιες σελίδες για εγγραφή.

4.2.3 *XSORT: Φάση της Παραγωγής του Τελικού Αποτελέσματος*

Στην Φάση της Παραγωγής του Τελικού Αποτελέσματος η διαθέσιμη μνήμη χωρίζεται σε τέσσερεις περιοχές. Η πρώτη αφορά την περιοχή του Input Buffer που χρησιμοποιείται για ανάγνωση σελίδων του αρχείου εισόδου, η δεύτερη αφορά το Output Buffer που χρησιμοποιείται για δημιουργία σελίδων του τελικού αρχείου, η τρίτη ένα δεύτερο Input Buffer που χρησιμοποιείται για ανάγνωση σελίδων του ιστογράμματος και τέλος η τέταρτη κρατά πληροφορίες σχετικά την σελίδα μέχρι την οποία έχει διαβαστεί κάθε run. Στην ουσία αποτελεί ένα πίνακα ακεραίων μεγέθους όσος και ο αριθμός των runs όπου κρατούμε τον αριθμό της σελίδας μέχρι την οποία διαβάσαμε για κάθε run.



Σχήμα 4.9: : Η διάσπαση της εσωτερικής μνήμης σε περιοχή πίνακα πληροφοριών των runs , Input Buffer για ανάγνωση σελίδων του αρχείου και Output Buffer για δημιουργία σελίδων του τελικού αρχείου και , Histogram Input Buffer για ανάγνωση σελίδων ιστογράμματος

Διαβάζουμε, λοιπόν μια προς μία κάθε σελίδα του αρχείου ιστογράμματος στο Histogram Input Buffer και αναζητούμε για κάθε εγράφη της (id, freq) να βρούμε την τιμή id freq φορές σε όλα τα runs. Στον πίνακα πληροφοριών για τα runs υπάρχει η

σελίδα η οποία έχει διαβαστεί μέχρι στιγμής. Κάθε κελί του πίνακα αρχικοποιείται με 0 για να δείχνει στην αρχική σελίδα. Κάθε run μπορεί να είναι είτε *ενεργό (active)* ή *νεκρό (dead)*. Ενεργό σημαίνει ότι υπάρχει έστω και μια εγγραφή στο run η οποία δεν έχει ταξινομηθεί άρα θα υπάρχει κάποιος αριθμός σελίδας στην αντίστοιχη θέση του πίνακα για το συγκεκριμένο run ενώ νεκρό σημαίνει ότι όλες οι εγγραφές του run έχουν ήδη ταξινομηθεί και η τιμή αυτή θα είναι -1. Τα νεκρά runs δεν θα συμπεριλαμβάνονται στην αναζήτηση που γίνεται και δεν θα διαβαστούν. Με αυτό τον μηχανισμό δημιουργούμε νεκρά και ενεργά σημεία στο σύνολο των προς ταξινόμηση δεδομένων και είμαστε σε θέση να απορρίπτουμε ομάδες σελίδων μεγέθους όσο και το νεκρό run αποφεύγοντας αχρείαστες επιπλέον αναγνώσεις. Με άλλα λόγια αν μια εγγραφή έχει ήδη γραφτεί στην μνήμη αυτή δεν θα ξαναδιαβαστεί στις επόμενες αναζητήσεις γιατί θα θεωρείται νεκρό σημείο. Στο σχήμα 4.10 που ακολουθεί παραθέτουμε τον ψευδοκώδικα της φάσης αυτής.

Algorithm 3 ProduceFinalOutput

Input: toatal number of Runs ,Histogram file pointer
Output: The sorted file

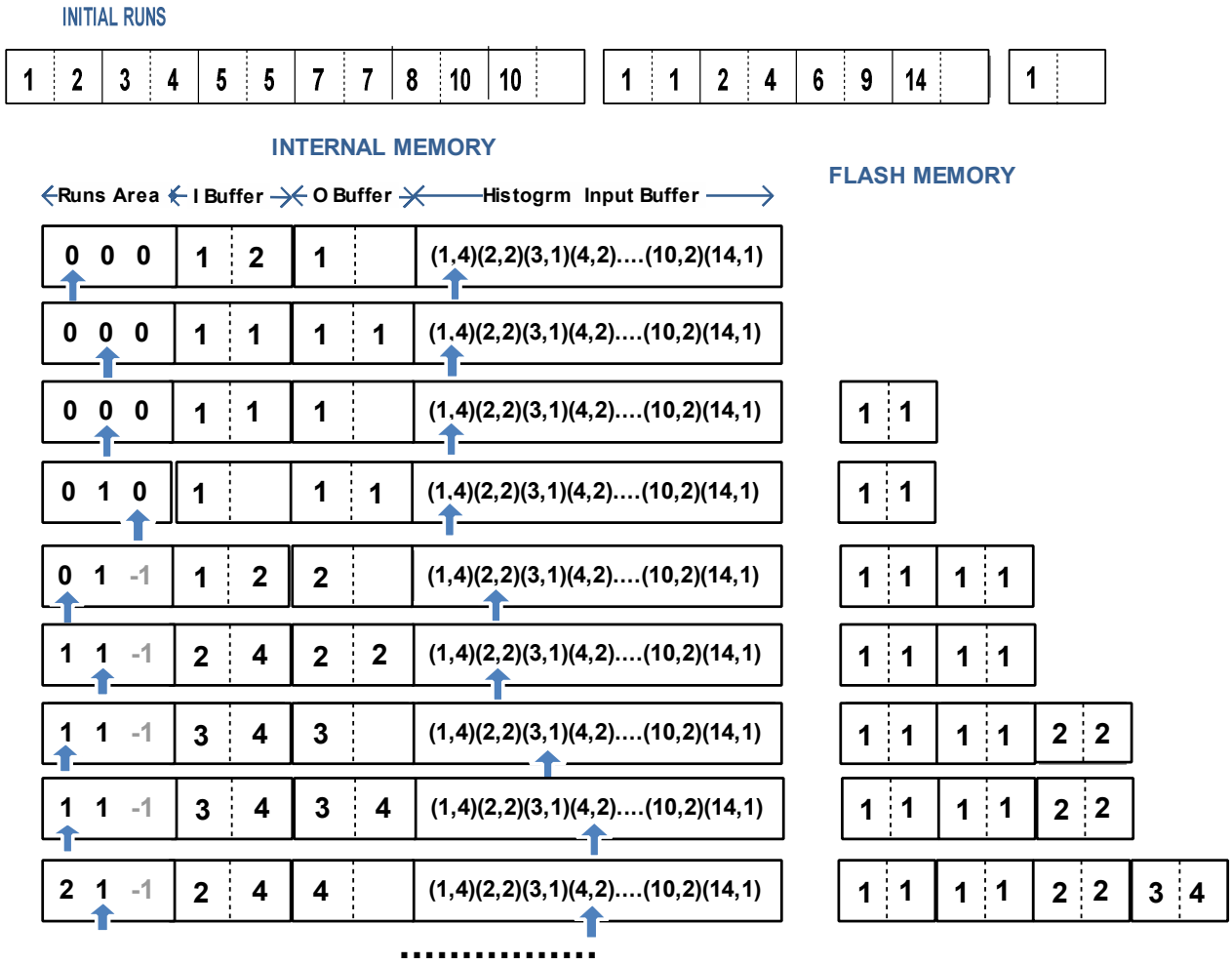
```
1:  procedure ProduceFinalOutput( totalRuns, histogam_File)
2:      // allocate structures
3:      allocate an Input Buffer Page
4:      allocate a Histogram Input Buffer Page
5:      allocate an Output Buffer Page
6:      RunsInfo[0...totalRuns] = 0
7:      while (true )
8:          // if there is no other page in histogram file yoou are done
9:          if (no other page in histogam_File ) then
10:             break
11:         end if
12:         // else read next histogram page
13:         Read  next page from histogam_File into Histogram Input Buffer Page
14:         // search in each run for the record.id value
15:         for each record in Histogram Input Buffer Page do
16:             counter = 0
17:             for currentRun = 0 to totalRuns do
18:                 // if current run is a dead run ignore it
19:                 if (RunsInfo[currentRun]==-1) then
20:                     continue
21:                 end if
22:                 // else next pages  and search for mathes
```

```
23:             done = false
24:             while (!done)
25:                 if (no page in Run[currentRun] file ) then
26:                     RunsInfo[currentRun] = -1
27:                     break
28:                 end if
29:                 Read RunsInfo[currentRun] page from Run[currentRun] file into
                    Input Buffer Page
30:                 for each record.value in  Input Buffer Page <= Histogram Input Buffer
                    Page record.id do
31:                     if (record.value  == Histogram Input Buffer Page record.id ) then
32:                         counter++
33:                         insert Input Buffer Page record in Output Buffer Page
34:                     end if
35:                     if (Output Buffer Page is full ) then
36:                         Write Output Buffer Page at the end of Sorted_File
37:                         reallocate a new Output Buffer Page
38:                     end if
39:                 end for
40:                 if (page last record was inserted into  Output Buffer) then
41:                     RunsInfo[currentRun]++
42:                 else
43:                     done = true
44:                 end if
45:             end while
46:             // if you have found the expected number of id value continue with the
                net value in histogram
47:             if (counter==Histogram Input Buffer Page record.freq ) then
48:                 break
49:             end if
50:         end for
51:     end for
52: end while
53: if (Output Buffer Page is not empty ) then
54:     Write Output Buffer Page at the end of Sorted_File
55: end if
56: delete Histogram Output Buffer Page
57: end procedure
```

Σχήμα 4.10: Ψευδοκώδικας της διαδικασίας ProduceFileOutput

Ας επιστρέψουμε στο παράδειγμα που ακολουθούμε και ας θεωρήσουμε ότι έχουμε τα παραγόμενα runs από τον Replacement-Selection-Sort και το αρχείο όπου είναι αποθηκευμένο το ιστόγραμμα που παρήγαμε κατά τη δεύτερη φάση του αλγόριθμου.

Θα παρουσιάσουμε στην συνέχεια ένα μέρος της εκτέλεσης της τρίτης φάσης του αλγόριθμου πάνω σε αυτά τα δεδομένα.

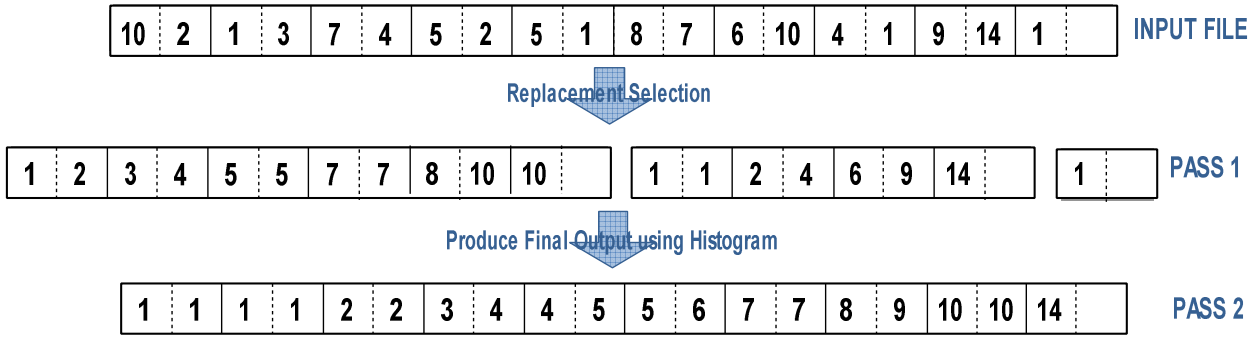


Σχήμα 4.11 : Το τρίτο και τελευταίο μέρος του αλγόριθμου XSORT-RSEL

4.3 Παραδείγματα Χρήσης XSORT

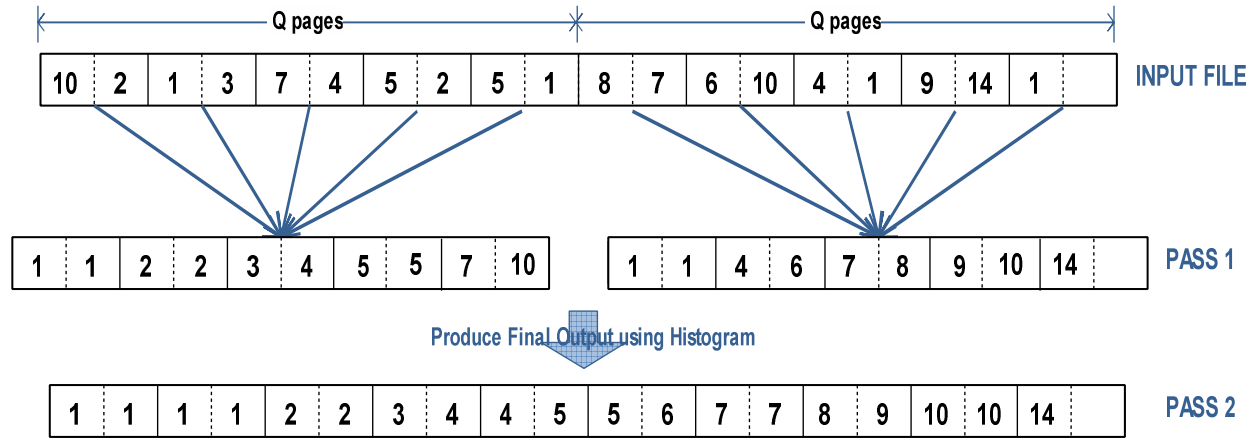
Γενικά η εκτέλεση των δύο παραλλαγών του αλγόριθμου φαίνονται πιο κάτω:

XSORT-RSEL



Σχήμα 4.11 : Εκτέλεση του XSORT-RSEL

XSORT-FAST



Σχήμα 4.12 : Εκτέλεση του XSORT-FAST

Παρατηρήστε ότι και στις δύο περιπτώσεις η εκτέλεση καλύπτει μόνο 2 passes συν κάποια (ανάλογα με το εύρος του ιστογράμματος) file scans για τη δημιουργία του ιστογράμματος. Αυτό πρακτικά σημαίνει σταθερό αριθμό (2N) εγγραφών συν τις απαραίτητες αναγνώσεις οι οποίες είναι σίγουρα περισσότερες από 3N (χρειαζόμαστε τουλάχιστον 3 αναγνώσεις όλων των σελίδων).

4.4 Ανάλυση Κόστους Αλγορίθμου XSORT σε Flash

Γενικά το I/O κόστος του αλγόριθμου XSORT καθορίζεται από το συνολικό I/O κόστος των τριών φάσεων που έχει. Με άλλα λόγια

$$C^{I/O}_{XSORT} = C^{I/O}_{firstpass} + C^{I/O}_{histgen} + C^{I/O}_{finalpass}$$

Το κόστος της πρώτης φάσης διαφέρει για τις δύο παραλλαγές και είναι το ίδιο με το αντίστοιχο κόστος που έχει η πρώτη φάση του Replacement-Selection-Sort και FAST, για XSORT-RSEL και XSORT-FAST αντίστοιχα, όπως αυτό ορίστηκε στο κεφάλαιο 3 στα υποκεφάλαια 3.2.3 και 3.3.3 αντίστοιχα.

Το κόστος της $2^{ης}$ φάσης είναι ο αριθμός των φορών που πρέπει να εκτελεστεί το file scan για τη δημιουργία του ιστογράμματος συν το κόστος εγγραφής των σελίδων του ιστογράμματος στην εξωτερική μνήμη.

$$C^{I/O}_{histgen} = \left\lceil Hrange * L_{integer} \Big/ \left(L_m - L_p\right) \right\rceil \cdot FilescanCost + NumbHistPages \cdot C_W$$

$$C^{I/O}_{histgen} = \left\lceil Hrange * L_{integer} \Big/ \left(L_m - L_p\right) \right\rceil \cdot N \cdot C_R + NumbHistPages \cdot C_W$$

Όπου Hrange το μέγεθος του ιστογράμματος (μέγιστη – ελάχιστη τιμή) και $1 \leq \text{NumbHp} \leq \left\lceil Hrange * L_{integer} \Big/ \left(L_m - L_p\right) \right\rceil$

Το κόστος της $3^{ης}$ φάσης είναι το κόστος ανάγνωσης από την εξωτερική μνήμη των σελίδων του ιστογράμματος συν το κόστος εύρεσης της τιμής που ψάχνουμε κάθε φορά συν το κόστος εγγραφής όλων των σελίδων μια φορά στο τελικό αρχείο.

$$C^{I/O}_{finalpass} = NumbHistPages \cdot C_R + b \cdot N \cdot C_R + N \cdot C_W$$

Όπου $1 \leq b \leq B$.

Συνοψίζοντας τα πιο πάνω θα έχουμε τα πιο κάτω κόστη για μνήμες Flash:

XSORT-RSEL:

$$C^{I/O}_{XSORT-RSEL} = N \cdot C_R + \left\lceil Hrange * L_{integer} \Big/ \left(L_m - L_p\right) \right\rceil \cdot N \cdot C_R$$
$$+ NumbHistPages \cdot C_W + NumbHistPages \cdot C_R + b \cdot N \cdot C_R + N \cdot C_W$$

XSORT-FAST:

$$C^{I/O}_{XSORT-FAST} = \left\lceil \frac{N}{Q} \right\rceil \left\lceil \frac{Q}{M_1} \right\rceil \cdot Q \cdot C_R + N \cdot C + \left\lceil Hrange * L_{integer} \Big/ \left(L_m - L_p\right) \right\rceil \cdot N \cdot C_R$$
$$+ NumbHp \cdot C_W + NumbHp \cdot C_R + b \cdot N \cdot C_R + N \cdot C_W$$

4.5 Συζήτηση

Η παρουσίαση του αλγόριθμου που έγινε πιο πάνω αφορά περιπτώσεις όπου το κλειδί της ταξινόμησης είναι ακέραιος αριθμός και μπορούμε να διαχωρίσουμε εύκολα τις διακριτές τιμές του. Στην περίπτωση όμως που το κλειδί της ταξινόμησης είναι δεκαδικός αριθμός ή κείμενο θα πρέπει να βρεθεί η καταλληλότερη συνάρτηση f(x) που θα αντιστοιχίζει την κάθε τιμή στο πεδίο ορισμού του κλειδιού της ταξινόμησης με μια διακριτή τιμή. Στην υλοποίηση που κάναμε για διαχείριση δεκαδικών αριθμών χρησιμοποιήσαμε την λογική του να πολλαπλασιάζουμε με 10^x τον δεκαδικό αριθμό, όπου x η επιθυμητή δεκαδική ακρίβεια, και να αποκόπτουμε το υπόλοιπο μέρος του αριθμού έτσι ώστε να παίρνουμε μια διακριτή τιμή. Βέβαια αναλόγως της δεκαδικής ακρίβειας που θα χρησιμοποιηθεί, το μέγεθος του ιστογράμματος μεγαλώνει και ίσως να μην χωράει ολόκληρο στην κύρια μνήμη κατά τη διάρκεια της δημιουργίας του με αποτέλεσμα επιπλέον filescans για τη δημιουργία του. Τέλος όσον αφορά τον τύπο κειμένου, θα πρέπει να χρησιμοποιηθεί μια κατάλληλη συνάρτηση κατακερματισμού.

Γενικά το μεγαλύτερο πλεονέκτημα του προτεινόμενου αυτού αλγόριθμου είναι ότι ξεφύγει από λογική της συνένωσης. Η λογική αυτή, επιβάλει στον αλγόριθμο δύο

φάσεις μια εκ των οποίων είναι η φάση της συνένωσης η οποία μπορεί να διαρκέσει από 1 μέχρι και δεκάδες passes. Κάθε pass συνεπάγεται N αναγνώσεις και N εγγραφές και έτσι πολλές επαναλήψεις στη φάση αυτή προκαλούν και την αύξηση του χρόνου εκτέλεσης του αλγόριθμου. Σε αντίθεση με αυτή τη λογική ο XSORT υποθέτει ότι κάθε ταξινόμηση μπορεί να γίνει σε μόνο δύο passes, πράγμα που πρακτικά σημαίνει ότι κρατάμε σταθερό τον αριθμό των χρονοβόρων εγγραφών στην μνήμη flash (=2N). Επίσης η χρήση του ιστογράμματος σε συνδυασμό με την τεχνική του ενεργού και του νεκρού run καθορίζει ότι ο αριθμός των επιπλέον αναγνώσεων θα είναι όσο το δυνατών μικρότερος. Τέλος οι υπόλοιποι αλγόριθμοι προσπαθούν κατά τη διάρκεια της συνένωσης, να βρίσκουν το μικρότερο αντικείμενο από μια ομάδα υποψηφίων προς ταξινόμηση δεδομένων, πράγμα που επιβάλλει αρκετές συγκρίσεις των αντικειμένων.
Αντίθετα ο XSORT αποφεύγει τις πολλές συγκρούσεις, αφού για κάθε υποψήφιο αντικείμενο απλά ελέγχουμε κατά πόσο αυτό ισούται με την τιμή που ψάχνουμε χωρίς να γίνουν επιπλέον συγκρίσεις. Αυτό έχει θετική επίδραση στην μείωση του CPU time delay.

4.6 Συγκριτικός Πίνακας Αλγορίθμων

Σε αυτή την ενότητα θα παρουσιάσο υμε ένα πίνακα ο οποίος συγκρίνει όλους του αλγόριθμους εξωτερικής ταξινόμησης.

	External-Merge-Sort	Replacement-Selection-Sort	FAST	XSORT -RSEL	XSORT -FAST
Μέγεθος αρχικών runs	M	Κατά προσέγγιση 2M	Q	Κατά προσέγγιση 2M	Q
Αρχική Φαση	N εγγραφές και N αναγνώσεις	N εγγραφές και N αναγνώσεις	N εγγραφές και Q/M ·Q αναγνώσεις	N εγγραφές και N αναγνώσεις	N εγγραφές και Q/M ·Q αναγνώσεις
Φάση Συνένωσης αρχικών runs	Συνένωση M-1 runs κάθε φορά (N εγγρ. και N αναγν. σε κάθε pass)	Συνένωση M-1 runs κάθε φορά (N εγγρ. και N αναγν. σε κάθε pass)	Συνένωση Q runs κάθε φορά (N εγγρ. και Q/M ·Q αναγν. σε κάθε pass)	Δεν Ισχύει	Δεν Ισχύει
Φάση Παραγωγής Ιστογράμματος	Δεν Ισχύει	Δεν Ισχύει	Δεν Ισχύει	Μερικά file scans και X* εγγραφές	Μερικά file scans και X* εγγραφές
Φάση Παραγωγής Τελικού Αποτελέσματος	Δεν Ισχύει	Δεν Ισχύει	Δεν Ισχύει	N εγγραφές και X* αναγνώσεις	N εγγραφές και X* αναγνώσεις
Συνολικός αριθμός Περασμάτων	$\lceil \log_{M-1}\left(\frac{N}{M}\right)\rceil+1$	$\lceil \log_{M-1}\left(\frac{N}{2M}\right)\rceil+1$	$\lceil \log_{Q}\left(\frac{N}{Q}\right)\rceil+1$	2	2
I/O κόστος	$N\cdot\left(\left\lceil \log_{M-1}\left(\frac{N}{M}\right)+1\right\rceil\right)\cdot\left((1+a)\cdot C_R\right)$	$N\cdot\left(\left\lceil \log_{M-1}\left(\frac{N}{2M}\right)+1\right\rceil\right)\cdot\left((1+a)\cdot C_R\right)$	$\left(\left\lceil\frac{N}{Q}\right\rceil\left\lceil\frac{Q}{M_1}\right\rceil\cdot Q+N\cdot a\right)\cdot C_R+\sum_{k=2}^{\lceil\log_Q N\rceil}\left(\left\lceil\frac{N}{Q^k}\right\rceil\cdot\left\lceil\frac{Q^k}{M_2}\right\rceil\cdot Q+Q^k\right)\cdot C_R+N\cdot a\cdot C_R$	$C^{I/O}_{XSORT-RSEL}=N\cdot C_R+\left\lceil Hrange*L_{integer}/(L_m-L_p)\right\rceil\cdot N\cdot C_R+NumbHistPages\cdot C_W+NumbHistPages\cdot C_R+b\cdot N\cdot C_R+N\cdot C_W$	$C^{I/O}_{XSORT-FAST}=\left\lceil\frac{N}{Q}\right\rceil\left\lceil\frac{Q}{M_1}\right\rceil\cdot Q\cdot C_R+N\cdot C+\left\lceil Hrange*L_{integer}/(L_m-L_p)\right\rceil\cdot N\cdot C_R+NumbHistPages\cdot C_W+NumbHistPages\cdot C_R+b\cdot N\cdot C_R+N\cdot C_W$

Πίνακας 3.1: Συγκριτικά στοιχεία αλγόριθμων εξωτερικής ταξινόμησης. * όπου X (αριθμών παραγόμενων σελίδων ιστογράμματος) ≥ N + NumbHp

Κεφάλαιο 5

Πειραματική Μεθοδολογία

Με σκοπό να διερευνήσουμε την απόδοση του προτεινόμενου αλγόριθμου XSORT έναντι των υπολοίπων αλγόριθμων, εκτελέσαμε ένα αριθμό πειραμάτων πάνω σε τρεις συσκευές οι οποίες κάνουν χρήση της μνήμης Flash σαν μνήμη αποθήκευσης. Στο κεφάλαιο αυτό θα περιγράψουμε λεπτομερώς την πειραματική διαδικασία που ακολουθήθηκε, τα τεχνικά χαρακτηριστικά των συσκευών και τα data sets που χρησιμοποιήθηκαν.

5.1 Πειραματική Υποδομή

Σε αυτή την παράγραφο θα παρουσιάσουμε τις συσκευές και τα δεδομένα που χρησιμοποιήθηκαν για την εκτέλεση των πειραμάτων.

5.1.1 *Οι Συσκευές που Χρησιμοποιήθηκαν*

Η μνήμη Flash βρίσκει μια πληθώρα εφαρμογών. Εμείς επιλέξαμε να δοκιμάσουμε την απόδοση των αλγορίθμων μας πάνω σε τρεις βασικές εφαρμογές της (σε USB Key, SD cards και SSD) τα χαρακτηριστικά των οποίων φαίνονται στον πίνακας που ακολουθεί.

SanDisk® Cruzer® Micro USB Flash Drive 4 GB [22]	
	General • **Storage Capacity:** 4 GB • **Width:** 20.6 mm • **Depth:** 7.9 mm • **Height:** 57.2 mm • **Compatibility:** Non-specific • **Product Type:** USB flash drive Memory • **Features:** U3 Smart capable • **Interface Type:** Hi-Speed US
HTC HERO [9] **with an SanDisk microSD™** [23] **2GB**	
	SanDisk MicroSD™ • **Memory Card Capacity:** 2GB • **Memory Card Model:** MicroSD • High speed data transfer rates • Compatible with all microSD / TransFlash mobile devices HTC HERO • **Processor:** Qualcomm® MSM7200A™, 528 MHz • **Operating System:** Android™ • **Memory:** ROM: 512 MB RAM: 288 MB
Kingston SSDNow V Series SNV425-S2/128GB 2.5" 128GB SATA II Internal Solid State Drive (SSD) [12]	
	Specification • **Capacity:** 128GB • **Storage Temperatures:** -40C to 85C • **Operating temperatures:** 0C to 70C • **Vibration Operating:** 2.17G (7800Hz) • **Vibration Non-Operation:** 20G (202000Hz) • **Sequential Speed:** 200MB/sec. read; 160MB/sec. write • **PCMARK HDD 2005 Score+:** 20,177 • **Power Specs:** 128GBActive: 5.2W (TYP)Sleep: 0.7W (TYP) 64GB Active: 5.2W (TYP)Sleep: 0.7W (TYP) • **Life expectancy:** 1 million hours mean time before failure

Πίνακας 5.1: Οι συσκευές που χρησιμοποιήθηκαν

5.1.2 *Περιγραφή των Datasets*

Χρησιμοποιήθηκαν δύο Datasets. Το πρώτο το ονομάσαμε ATMOSPHERIC (Washington State Climate Dataset) και αφορά δεδομένα του πραγματικού κόσμου ενώ το δεύτερο το ονομάσαμε Customers Dataset και αφορά δεδομένα παραγόμενα από τον generator DBGEN που χρησιμοποιείται για την παραγωγή πινάκων βάσης δεδομένων για το TPC-H benchmark. Στην συνέχεια θα παρουσιάσουμε τα εν λόγω Datastes.

5.1.2.1 ATMOSHERIC (Washington State Climate Dataset)

Πρόκειται για ένα σύνολο περιοδικών περιβαλλοντικών μετρήσεων όπως αυτές καταγράφηκαν από το Τμήμα Περιβαλλοντικών Σπουδών του Πανεπιστημίου της Ουάσιγκτον (Department of Atmospheric Sciences , University of Washington) [28] . Οι εν λόγω μετρήσεις καλύπτουν χρονικό διάστημα 6 χρόνων (από τις 12:09:00 1999-07-30 μέχρι τις 10:20:00 2005-02-27), ενώ κάθε εγράφη (record) αποτελεί ένα στιγμιότυπο ενός λεπτού και παρουσιάζει πληροφορίες όπως η θερμοκρασία του αέρα, η υγρασία, η ατμοσφαιρική πίεση και η ταχύτητα του ανέμου. Πιο συγκεκριμένα η κάθε εγράφη περιέχει 10 χαρακτηριστικά και έχει την ακόλουθη μορφή .

ATMOSPHERIC (Washington State Climate Dataset) Record Format		
Description	**type**	**Size (in bytes)**
1. Java_time (seconds since 1-1-1970)	string	15
2. GMT Date	string	20
3 . Julian Date)	double	8
4. Temperature (F)	float	4
5. Dewpoint Temperature (F)	float	4
6. Relative_humidity(%)	float	4
7. Wind_speed (knots)	float	4
8. Peak_wind_speed (knots)	float	4
9. Pressure (millibars)	float	4
10. Cumulative Rain (0.01 in. increments)	double	8
	Record size (L_r)	75 bytes

Πίνακας 5.2: Μορφή των εγγραφών τουWashington State Climate Dataset

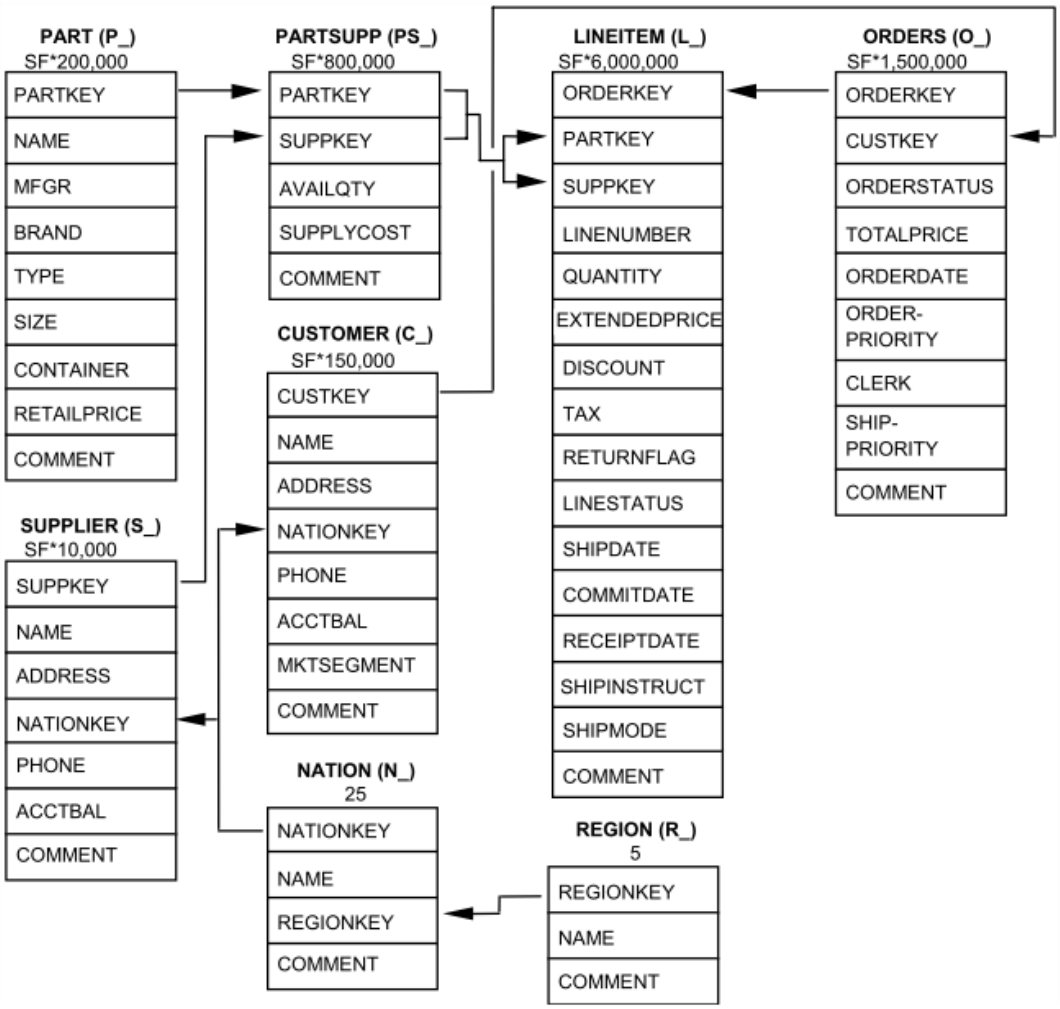
Στον πιο πάνω πίνακα παρουσιάζεται επίσης ο τύπος του κάθε χαρακτηριστικού της εγγραφής καθώς και το συνολικό μέγεθός της (record size) που είναι 75 bytes. Το αρχείο που χρησιμοποιήθηκε στα πειράματα που έγιναν είχε μέγεθος 214MB και περιείχε 2 899 087 εγγραφές . Επίσης, στα εν λόγω πειράματα η ταξινόμηση έγινε κατά αύξουσα σειρά με βάση το χαρακτηριστικό Temperature.

5.1.2.2 Customers Dataset (TPC-H benchmark)

Γενικά ο TPC (Transaction Processing Performance Council) [26] είναι ένας μη κερδοσκοπικός οργανισμός που ιδρύθηκε το 1988 και καθορίζει ένα πλαίσιο αξιολόγησης συστημάτων μέσα από μια σειρά *Μετρήσεων Επιδόσεων (benchmarks)* επεξεργασίας *δοσοληψιών* (*transaction processing)* και *βάσεων δεδομένων* (databases). Τα benchmark του TCP βρίσκουν ευρεία αποδοχή από την παγκόσμια βιομηχανία και αποτελούν σημείο αναφοράς για την επίδοση υπολογιστικών συστημάτων.

Το TPC Benchmark™H (TPC-H) μοντελοποιεί ένα σύστημα λήψης αποφάσεων το οποίο επεξεργάζεται ένα μεγάλο όγκο δεδομένων, εκτελώντας επερωτήσεις με μεγάλο βαθμό πολυπλοκότητας και απαντώντας σε ερωτήματα καθοριστικής σημασίας για την επιχείρηση. Στην ουσία προσομοιώνει το φόρτο εργασίας μια τυπικής εταιρεία μέσα από την εκτέλεση μιας σειράς ad-hoc επερωτήσεων και ταυτόχρονων τροποποιήσεων δεδομένων της βάσης.

Το DBGEN είναι ένα πρόγραμμα δημιουργίας μιας βάσης δεδομένων που θα χρησιμοποιηθεί για την εκτέλεση των επερωτήσεων του TPC-H. Είναι γραμμένο σε ANSI 'C' για λόγους portability και έχει χρησιμοποιηθεί με επιτυχία σε διάφορα συστήματα. Η εκτέλεση του προγράμματος χωρίς παραμέτρους θα δώσει 8 αρχεία κειμένου όπου το κάθε ένα θα αντιπροσωπεύει ένα από τους πίνακες του σχήματος της βάσης όπως φαίνεται στο Σχήμα 5.1. Ο χρήστης είναι σε θέση να επιλέξει τον πίνακα που θέλει να παράγει και να καθορίσει το μέγεθός του μέσω command-line παραμέτρων. Εμείς στα πειράματα που κάναμε χρησιμοποιήσαμε τα δεδομένα του πίνακα customer.



Σχήμα 5.1: Το σχήμα της βάσης του TPC-H [26]

Η μορφή της κάθε εγγραφής του πίνακα Customer είναι η ακόλουθη.

Customer Dataset Record Format		
Description	**type**	**Size (in bytes)**
1. CUSTKEY	integer	4
2. NAME	string	19
3. ADDRESS	string	25
4. NATIONKEY	integer	4
5. PHONE	string	16
6. ACCTBAL	float	4
7. MKTSEGMENT	string	11
8. COMMENT	string	103
	Record size (L_r)	186 bytes

Πίνακας 5.3: Μορφή των εγγραφών του Customer Dataset

Στον πιο πάνω πίνακα παρουσιάζεται επίσης ο τύπος του κάθε χαρακτηριστικού της εγγραφής καθώς και το συνολικό μέγεθός της (record size) που είναι 186 bytes. Όσον αφορά το μέγεθος του αρχείου, δημιουργήσαμε διάφορα στιγμιότυπα του πίνακα customer (δηλ αρχεία διαφόρων μεγεθών) και ορίσαμε σαν το κλειδί ταξινόμησης το χαρακτηριστικό NATION_KEY, το οποίο έχει πεδίο ορισμού από το 0 μέχρι και το 24..

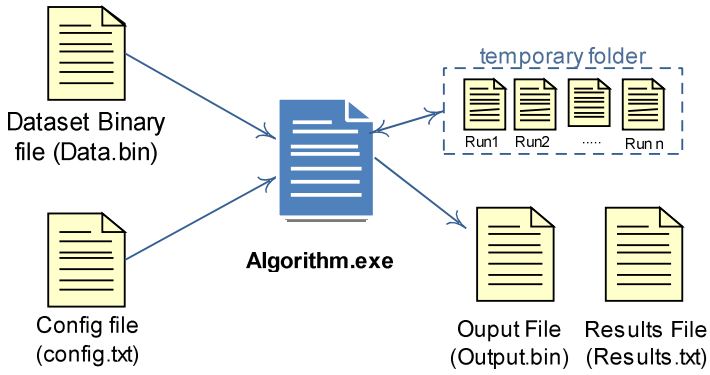
5.2 Πειραματική Διαδικασία

Αρχικά γράψαμε τον κώδικα για τους πέντε αλγόριθμους. (External-Merge-Sort, Replacement-Selection-Sort, FAST και τις δύο παραλλαγές του XSORT) και στην συνέχεια ελέγξαμε την ορθότητα και την δυνατότητα εκτέλεσής του και στις τρεις συσκευές που θα χρησιμοποιούσαμε. Να αναφέρουμε επίσης ότι χρειάστηκε και κάποια προεργασία πάνω στα δεδομένο έτσι ώστε να μπορούν να χρησιμοποιηθούν ως είσοδος στα προγράμματα που γράψαμε.

Η υλοποίηση όλων των αλγόριθμων και των σχετικών βιβλιοθηκών που αναφέρονται εδώ έγινε σε γλώσσα προγραμματισμού C. Ο λόγος που επιλέξαμε την συγκεκριμένη γλώσσα προγραμματισμού ήταν το γεγονότος ότι αυτή προσφέρει καλύτερη διαχείρισης της μνήμης σε σχέση με άλλες υψηλότερου επιπέδου γλώσσες που αποκρύπτουν πολλές λεπτομέρειες στα θέματα αυτά. Από την στιγμή μάλιστα που θέλουμε να μετρήσουμε απόδοση επιβάλλεται η χρήση μιας γλώσσας προγραμματισμού χαμηλού επιπέδου όπως είναι η C.

Όσον αφορά την υλοποίηση των αλγόριθμων, αρχικά υλοποιήσαμε κάποιες βιβλιοθήκες που περιέχουν κοινές για όλους του αλγόριθμους μεταβλητές (π.χ., N, B κ.τ.λ), δομές δεδομένων (π.χ., τη δομή μιας σελίδας ή μιας εγγραφής) καθώς επίσης και κάποιες κοινές μεθόδους όπως για παράδειγμα την ανάγνωση μιας σελίδας από ένα αρχείο, την εγγραφή μιας σελίδας από την εσωτερική μνήμη σε ένα αρχείο στη εξωτερική κτλ. Η υλοποίηση του κάθε αλγόριθμου έγινε σε διαφορετικό αρχείο το οποίο φέρει το όνομα του αλγόριθμου στον όποιο αντιστοιχεί (ExternalMergeSort.c, XSORT-FAST.c κτλ). Κατά την εκτέλεση του ο κάθε αλγόριθμος αρχικοποιεί όλες τις μεταβλητές του βάσει ενός config αρχείου που του δίδεται σαν command line είσοδος. Στην συνέχεια διαβάζει τα προς ταξινόμηση δεδομένα από το δυαδικό αρχείο, η

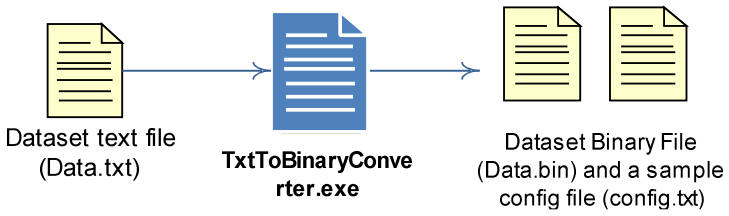
διεύθυνση του οποίου καθορίζεται στο config αρχείο και ακολούθως προχωρεί στη εκτέλεση της ταξινόμησης κάνοντας χρήση ενός προσωρινού φακέλου (που επίσης καθορίζεται στο config αρχείο) για αποθήκευση των ενδιάμεσων runs.



Σχήμα 5.2: Η εκτέλεση ενός αλγόριθμου

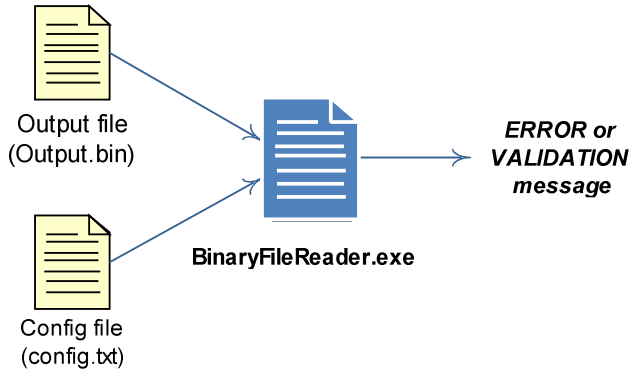
Το τελικό αποτέλεσμα σε κάθε περίπτωση είναι ένα δυαδικό αρχείο (το ταξινομημένο αρχείο) και ένα ascii αρχείο που περιέχει κάποια στατιστικά για τον αλγόριθμο όπως για παράδειγμα τον χρόνο εκτέλεσης του, το συνολικό αριθμό των αναγνώσεων και των εγγραφών κτλ. Να αναφέρουμε επίσης ότι χρησιμοποιήσαμε την βιβλιοθήκη <time.h> για να μετρήσουμε το χρόνο εκτέλεσης του κάθε αλγόριθμου. Η συγκεκριμένη βιβλιοθήκη κρίθηκε η καταλληλότερη, αφού βρίσκει ευρεία χρήση σε πειράματα μετρήσεων απόδοσης αλγορίθμων.

Όσον αφορά τώρα την *προεργασία (preprocessing)* πάνω στα δεδομένα δημιουργήσαμε ένα πρόγραμμα το οποίο μετατρέπει ένα ascii αρχείο σε δυαδικό αρχείο και παράγει το αντίστοιχο confing αρχείο που θα χρησιμοποιηθεί ως είσοδος στο αλγόριθμο και το οποίο μπορεί να μεταβληθεί και να αναπαραχθεί από τον χρηστή. Η αρχική σκέψη ήταν να χρησιμοποιηθεί η XML για τη σύνταξη του confing αρχείου αλλά θα ήταν ιδιαίτερα δύσκολη η επεξεργασία του στο περιβάλλον του android που χρησιμοποιείται από την κινητή συσκευή, λόγω της μη ύπαρξης αντίστοιχων βιβλιοθηκών επεξεργασίας XML όπως αυτές που υπάρχουν για Linux. Έτσι για λόγους portability προτιμήθηκε το συγκεκριμένο αρχείο να έχει μορφή αρχείου κείμενου.



Σχήμα 5.3: Η μετατροπή ενός ascii αρχείου σε δυαδικό

Τέλος όσον αφορά τον έλεγχο του τελικού αποτελέσματος δημιουργήσαμε ένα πρόγραμμα το οποίο παίρνει ως είσοδο το τελικό αρχείο και το αρχείο παραμέτρων το οποίο δόθηκε στο πρόγραμμα που το παρήγαγε και ελέγχει κατά πόσο έγινε ορθά η ταξινόμηση εμφανίζοντας το αντίστοιχο μήνυμα αποτυχίας ή επιτυχίας.



Σχήμα 5.4: Ο έλεγχος του αποτελέσματος

Στην συνέχεια ορίσαμε τα διάφορα datasets που θα χρησιμοποιούσαμε, όπως αυτά αναφέρονται στην παράγραφο 5.1 και τα αρχεία παραμέτρων (config files) των πειραμάτων, δημιουργήσαμε τα απαιτούμενα scripts και εκτελέσαμε τα πειράματα. Όσον αφορά τα πειράματα που εκτελέσαμε χρησιμοποιήθηκαν δύο διαφορετικά datasets, όπως αυτά περιγράφηκαν σε προηγούμενη ενότητα του κεφαλαίου αυτού. Κάθε πείραμα αποτελείτο από ένα σύνολο διαφορετικών benchmarks όπου μεταβάλλαμε μόνο την παράμετρο που καθορίζει την διαθέσιμη κύρια μνήμη του συστήματος. Περισσότερες πληροφορίες για τα πειράματα που εκτελέσαμε βρίσκονται στον πίνακα 5.4.

Χαρακτηριστικά	USBKEYS		HTC-HERO	SSD
	ATMOSPHERIC DATASET	CUSTOMER DATASET	ATMOSPHERIC DATASET	CUSTOMER DATASET
Μέγ. Αρχείου (Lf)	214 MB	800 MB	214 MB	12 GB
Συν. Αριθ. εγγραφών	2,899,086	4,400,000	2,899,086	66,011,000
Μέγ. σελίδας (Lp)	4 KB	4 KB	16 KB	4 KB
Μέγ. εγγραφής (Lr)	75 bytes	186 bytes	75 bytes	186 bytes
# σελ. στην Κ. Μν. (B)	54	22	218	22
# σελ. στο αρχείο (N)	5,3687	200,000	5,3687	3,000,500
αριθμός benchmarks	9	8	9	7
Κλειδί ταξινόμησης	Temperature	Nation_Key	Temperature	Nation_Key
Τύπος κλειδιού ταξ.	Δεκαδικός	Ακέραιος	Δεκαδικός	Ακέραιος

Πίνακας 5.4: Η πειραματική διαδικασία

Κεφάλαιο 6

Πειραματικά Αποτελέσματα

Στο Κεφάλαιο αυτό θα παρουσιάσουμε τα αποτελέσματα από τις πειραματικές μετρήσεις που έγιναν στη βάση της πειραματικής διαδικασίας που ορίσθηκε στο προηγούμενο κεφάλαιο. Αρχικά θα δώσουμε μια γενική ιδέα των αποτελεσμάτων, ενώ στην συνέχεια θα προχωρήσουμε σε μια πιο αναλυτική παρουσίασή τους συγκρίνοντας τους πέντε αλγόριθμους σε τρία βασικά χαρακτηριστικά το μέσο όρο εκτέλεσης του κάθε αλγόριθμού, το συνολικό αριθμό των passes και των IOs που προκαλεί. Θα γίνει επίσης προσπάθεια επικυρώσεις και της θεωρητικής ανάλυσης που προηγήθηκε.

6.1 Εισαγωγή

Η ανάλυση θα γίνει σε τρία επίπεδα. Το πρώτο αφορά το μέσο όρο της συνολικής εκτέλεσης του αλγόριθμου (συνυπολογίζοντας και το CPU time), το δεύτερο τον αριθμό των passes που χρειάζεται ο αλγόριθμος για την παραγωγή του τελικού αποτελέσματος και το τρίτο τον αριθμό των εγγραφών και αναγνώσεων που εκτελούνται στην μνήμη flash κατά την εκτέλεση του αλγόριθμου.

Γενικά παρατηρείται μια υπεροχή των δύο προτεινόμενων αλγόριθμων XSORT σε όλα τα επίπεδα που αναφέραμε πιο πάνω και σε όλες τις συσκευές στις οποίες δοκιμάστηκαν. Πιο συγκεκριμένα οι δύο αλγόριθμοι παρουσιάζουν χρόνους εκτέλεσης μέχρι και 2 φορές καλύτερους από αυτούς των υπολοίπων αναλόγως της συσκευής και του dataset που χρησιμοποιείται. Όσον αφορά τους δύο παρουσιάζουν παραπλήσιους χρόνους εκτέλεσης με τον XSORT-FAST να παρουσιάζει καλύτερους χρόνους. Γενικά οι XSORT αλγόριθμοι εκτελούν σταθερό αριθμό passes (=2) και σταθερό αριθμό εγγραφών στην μνήμη flash (=2N). Εκεί που υπερτερεί ο XSORT-FAST έναντι του XSORT-RSEL είναι στον αριθμό των επιμέρους αναγνώσεων πράγμα που του δίνει και μικρότερους χρόνους εκτελέσεις. Επόμενος καλύτερος αλγόριθμος είναι ο FAST αφού γενικά παρουσιάζει καλύτερους χρόνους εκτέλεσης από τους άλλους δύο πράγμα που οφείλεται στα λιγότερα passes που εκτελεί και κατ' επέκταση στις λιγότερες εγγραφές στην εξωτερική μνήμη (flash). Εξαίρεση αποτελεί η περίπτωση των πειραμάτων πάνω στην κινητή συσκευή όπου FAST παρουσιάζεται χειρότερος από όλους. Τέλος οι άλλοι δύο άλλοι αλγόριθμοι (External-Merge-Sort, Replacement-Selection-Sort) παρουσιάζουν κοντινούς χρόνους εκτελέσεις αλλά στις πλείστες των περιπτώσεων ο Replacement-Selection-Sort παρουσιάζεται καλύτερός από τον External-Merge-Sort. Επίσης ο αριθμός των απαιτούμενων passes για τους δύο αυτούς αλγόριθμους είναι ο ίδιος με δύο μόνο εξαιρέσεις όπου ο Replacement-Selection-Sort παρουσιάζει μικρότερο αριθμό (ένα λιγότερο pass) και αισθητά καλύτερο χρόνο εκτέλεσης. Από αυτό το σημείο και για το υπόλοιπο του κεφαλαίου θα αναφερόμαστε στον External-Merge-Sort ως EXMS, στον Replacement-Selection-Sort ως RSEL, στον XSORT-RSL ως XS-RSEL και στον XSORT-FAST ως XS-FAST

6.2 Παρουσίαση και ανάλυση αποτελεσμάτων

Στις 4 υποενότητες (6.3-9) παρουσιάζονται οι αντίστοιχες πειραματικές μετρήσεις για κάθε ένα από τα 4 πειράματα που έγιναν όπως αυτά ορίζονται στον πίνακα 5.4 του προηγούμενο κεφαλαίου. Σε κάθε υποενότητα, η ανάλυση των αποτελεσμάτων γίνεται στη βάση των τριών παραμέτρων που αναφέρθηκαν στην προηγούμενη ενότητα μέσα από τρεις γραφικές παραστάσεις. Οι τρεις αυτές γραφικές παραστάσεις είναι:

- ***Μέσος Χρόνος Εκτέλεσης Αλγορίθμων:*** Η εν λόγω γραφική παράσταση παρουσιάζει το μέσο χρόνο εκτέλεσης κάθε ενός από τους πέντε αλγόριθμους που μελετάμε για διαφορετικά μεγέθη κύριας μνήμης του συστήματος. Στον οριζόντιο άξονα παρουσιάζεται η μεταβλητή Μ, η οποία αναφέρεται στον αριθμό των σελίδων που μπορούν να χωρέσουν στην κύρια μνήμη ανά πάσα χρονική στιγμή και στον κάθετο άξονα παρουσιάζεται ο αντίστοιχος μέσος χρόνος εκτέλεσης του κάθε αλγόριθμου σε δευτερόλεπτα. Να αναφέρουμε επίσης ότι με τον όρο συνολικός χρόνος εκτέλεσης εννοούμε το χρόνο που απαιτείται από την αρχή της εκτέλεσης του αλγόριθμου μέχρι την παραγωγή του τελικού αποτελέσματος, που στην ουσία συνεπάγεται την εγγραφή και της τελευταίας πλειάδας στο ταξινομημένου πλέον αρχείο. Με άλλα λόγια έχουμε μια συνάθροιση του I/O κόστους και του CPU time για κάθε αλγόριθμο.

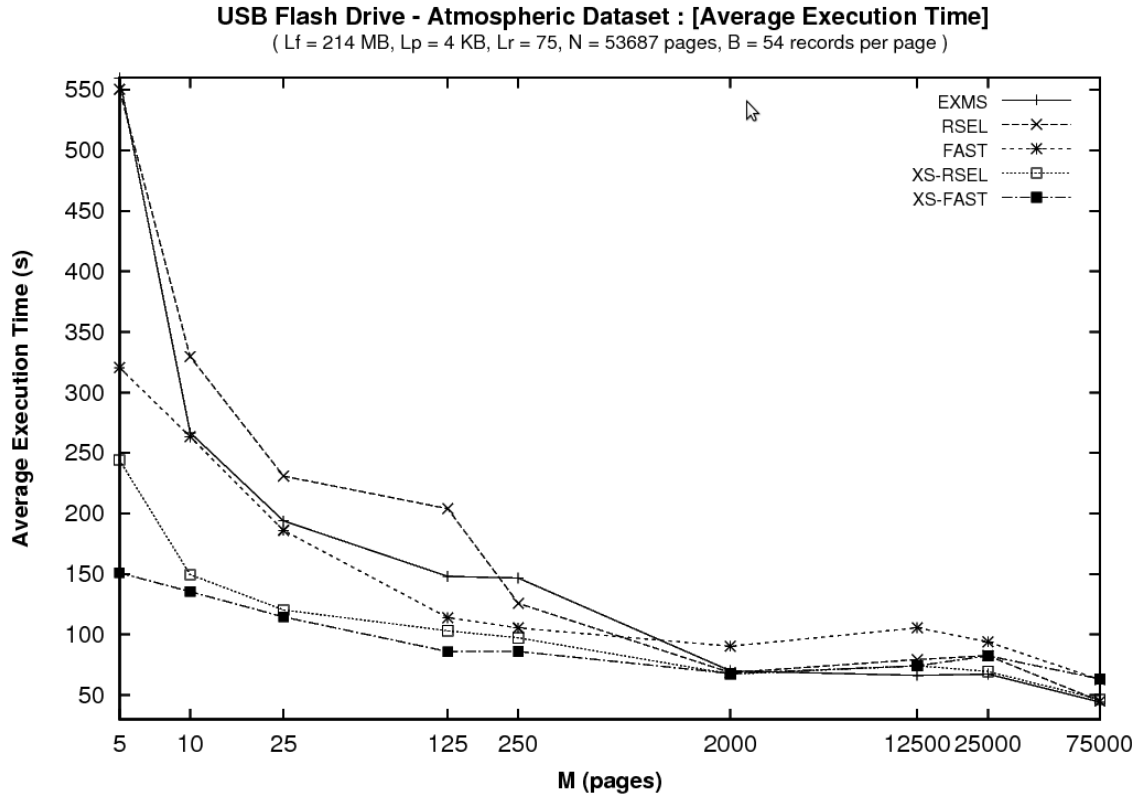
- ***Συνολικός Αριθμός Passes:*** Η εν λόγω γραφική παράσταση παρουσιάζει τον συνολικό αριθμό των passes που χρειάζεται κάθε ένας από τους πέντε αλγόριθμους για την ολοκλήρωση της εκτέλεσής του σε διαφορετικά μεγέθη κύριας μνήμης του συστήματος. Ο συνολικός αριθμός των passes εμφανίζεται στον κάθετο άξονα της γραφικής παράστασης ενώ η μεταβλητή Μ παρουσιάζεται στον οριζόντιο άξονα. Η παρουσίαση γίνεται σε μορφή σύνθετου ιστογράμματος όπου σε κάθε διαφορετικό Ν αντιστοιχούν πέντε ράβδοι , μία για κάθε αλγόριθμο.

- ***Συνολικός αριθμός I/O:*** Η εν λόγω γραφική παράσταση παρουσιάζει τον συνολικό αριθμό των I/O πράξεων (αναγνώσεων και εγγραφών προς την εξωτερική μνήμη) που προκαλεί ο κάθε ένας από τους πέντε αλγόριθμους κατά

τη διάρκεια της εκτέλεσής του για διαφορετικά μεγέθη της κύριας μνήμης του συστήματος. Ο συνολικός αριθμός των I/O εμφανίζεται στον κάθετο άξονα της γραφικής παράστασης ενώ η μεταβλητή M παρουσιάζεται στον οριζόντιο άξονα. Η παρουσίαση γίνεται και πάλι σε μορφή σύνθετου ιστογράμματος όπου σε κάθε διαφορετικό M αντιστοιχούν πέντε ράβδοι, μία για κάθε αλγόριθμο. Όπως αναφέραμε και πιο πάνω ο συνολικός αριθμός των I/Os πράξεων για κάθε αλγόριθμο αποτελεί το άθροισμα όλων των αναγνώσεων και όλων των εγγραφών που προκαλούνται από τον εν λόγω αλγόριθμο. Από τη στιγμή που οι δύο αυτές πράξεις δεν είναι συμμετρικές καλό θα ήταν να τις διαχωρίσουμε. Έτσι η κάθε ράβδος θα χωρίζεται σε δύο μέρη, όπου το σκούρο κομμάτι της θα παρουσιάζει τις χρονοβόρες εγγραφές στη μνήμη flash, ενώ το ανοιχτόχρωμο κομμάτι θα παρουσιάζει τον αριθμό των αναγνώσεων της μνήμης flash.

Να αναφέρουμε στο σημείο αυτό ότι στο Παράρτημα Β βρίσκονται όλα αποτελέσματα των πειραμάτων που έγιναν. Η παρουσίαση γίνεται σε μορφή πίνακα αρχικά βάση του πειράματος και στη συνέχεια βάση του αλγόριθμου που χρησιμοποιείται κάθε φορά. Να σημειώσουμε ότι για τον FAST και XSORT-FAST έγινε μια πληθώρα μετρήσεων που αφορούσαν την μεταβολή της μεταβλητής Q για κάθε διαφορετικό M. Στις γραφικές παραστάσεις που παρουσιάζεται μόνο η καλύτερη από αυτές τις μετρήσεις. Να αναφέρουμε επίσης ότι στους πίνακες του Παραρτήματος Β υπάρχουν και κάποιες επιπλέον πληροφορίες όπως για παράδειγμα ο χρόνος εκτέλεσης της κάθε φάσης του κάθε αλγόριθμου ξεχωριστά κτλ.

6.3 USB KEY – ATMOSPHERIC DATASET



Σχήμα 6.1 : USB Flash Drive – Atmospheric Dataset : Μέσος χρόνος εκτέλεσης

Σύμφωνα με την γραφική παράσταση του μέσου χρόνου εκτέλεσης των αλγορίθμων που μελετάμε για την περίπτωση αυτή (Σχήμα 6.1) παρατηρούμε ότι οι δύο υλοποιήσεις του XSORT υπερνικούν τους υπόλοιπους αλγόριθμους για όλα τα διαφορετικά μεγέθη κύριας μνήμης, με καλύτερο εκ των δύο τον XS-FAST. Ρίχνοντας μια ματιά στους Πίνακες 6.1 και 6.2 που ακολουθούν και παρουσιάζουν το χρόνο εκτέλεσης των δύο φάσεων ξεχωριστά για κάθε αλγόριθμο, παρατηρούμε ότι οι αλγόριθμοι που κάνουν χρήση της λογικής της συνένωσης υποφέρουν σε θέμα χρόνου στη φάση της συνένωσης όταν κυρίως το μέγεθος της διαθέσιμης κύριας μνήμης είναι σχετικά μικρό. Συγκριτικά παρατηρούμε ότι για τους EXMS, RSEL η φάση αυτή καλύπτει περισσότερο από το 70% του συνολικού χρόνου εκτέλεσης των αλγόριθμων και για τον FAST καλύπτει περίπου το 65% του συνολικού χρόνου εκτέλεσης. Να σημειώσουμε ότι για τους XSORT αλγόριθμους συνυπολογίζουμε μαζί τον χρόνο εκτέλεσης της $2^{ης}$ και $3^{ης}$ φάσης σαν $2^{η}$ φάση για να μπορούμε να κάνουμε καλύτερα τη σύγκριση. Σε αυτούς του αλγόριθμους η $2^{η}$ και $3^{η}$ φάση μαζί καλύπτουν ποσοστό κάτω του 45% του συνολικού χρόνου εκτέλεσης. Δεδομένου λοιπόν ότι κατά την πρώτη φάση όλοι οι

αλγόριθμοι παρουσιάζουν περίπου ίδιους χρόνους εκτέλεσης, αυτό που δημιουργεί τη διαφορά στους συνολικούς χρόνους μεταξύ τους είναι ο χρόνος που χρειάζεται για την εκτέλεση των υπόλοιπων φάσεων. Παρατηρούμε επίσης ότι η διαφορά στο χρόνο εκτέλεσης των XSORT αλγόριθμων από τους υπόλοιπους είναι μεγαλύτερη όταν η διαθέσιμη κύρια μνήμη έχει μικρό μέγεθος ενώ σε μεγαλύτερα μεγέθη μνήμης η διαφορά από τους υπόλοιπους μειώνεται αισθητά και τελικά συμπίπτει.

USB Flash Drive – ATMOSPHERIC DATASET (PHASE 1(s))					
M	EXMS	RSEL	FAST	XS-RSEL	XS-FAST
5	159.1175	191.07075	116.39825	132.828	118.4098
10	83.31425	158.21425	117.225	96.39625	106.1715
25	88.191	133.78975	102.019	84.33575	88.0665
125	57.8425	88.1015	67.28275	74.46775	58.9805
250	43.79675	64.3215	54.3255	70.602	59.35075
2000	39.72475	41.14775	44.7485	39.96875	42.558
12500	41.67575	54.2525	52.19775	46.78775	52.34625
25000	43.85075	49.808	93.9105	43.14575	57.05875
75000	44.20025	44.94975	62.64275	46.44725	63.291

Πίνακας 6.1 : USB Flash Drive – Atmospheric Dataset : Χρόνος εκτέλεσης πρώτης φάσης

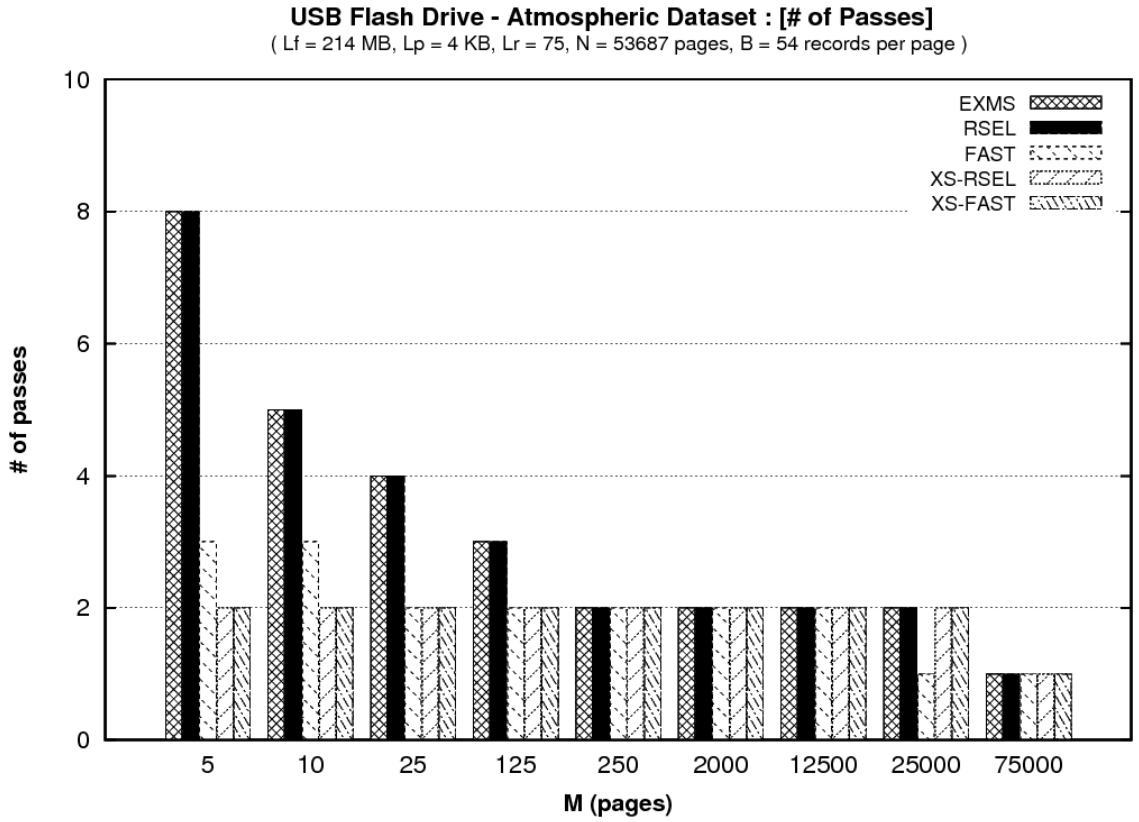
USB Flash Drive – ATMOSPHERIC DATASET (PHASE 2(s))					
M	EXMS	RSEL	FAST	XS-RSEL	XS-FAST
5	400.41975	359.267	204.226	111.285	32.558
10	183.482	171.3355	146.23575	53.12075	29.31725
25	105.80675	97.158	84.01675	35.985	26.48975
125	90.15575	115.9395	46.65875	28.67475	26.91075
250	102.90875	61.4815	51.1555	26.65225	26.6855
2000	30.304	27.33575	45.636	27.32575	25.2515
12500	24.59875	25.1325	53.45875	27.56475	25.788
25000	23.4995	32.75225	0	26.314	25.34425
75000	0	0	0	0	0

Πίνακας 6.2 : USB Flash Drive – Atmospheric Dataset : Χρόνος εκτέλεσης δεύτερης φάσης

Αυτό συμβαίνει γιατί σε μικρότερου μεγέθους διαθέσιμη κύρια μνήμη οι άλλοι τρείς αλγόριθμοι θα χρειαστούν, περισσότερες επαναλήψεις στην φάση της συνένωσης και κατ' επέκταση και περισσότερα passes για την παραγωγή του τελικού αποτελέσματος σε αντίθεση με τα μόνο 2 passes που εκτελούν οι αλγόριθμοι XSORT συν κάποιες επιπλέον αναγνώσεις του αρχείου. Περισσότερα passes όπως θα δούμε και στην συνέχεια συνεπάγονται και περισσότερες εγγραφές στη μνήμη flash οι οποίες είναι χρονοβόρες και προσθέτουν στον συνολικό χρόνο εκτέλεσης του αλγόριθμου. Από την

άλλη όταν η διαθέσιμη μνήμη είναι σχετικά μεγάλη τα passes των υπόλοιπων αλγόριθμων μειώνονται και αντίστοιχα μειώνονται και οι χρονοβόρες εγγραφές στην μνήμη flash. Η μείωση αυτή θα έχει ως αποτέλεσμα και την αισθητή μείωση του συνολικού χρόνου εκτέλεσης των εν λόγω αλγορίθμων.

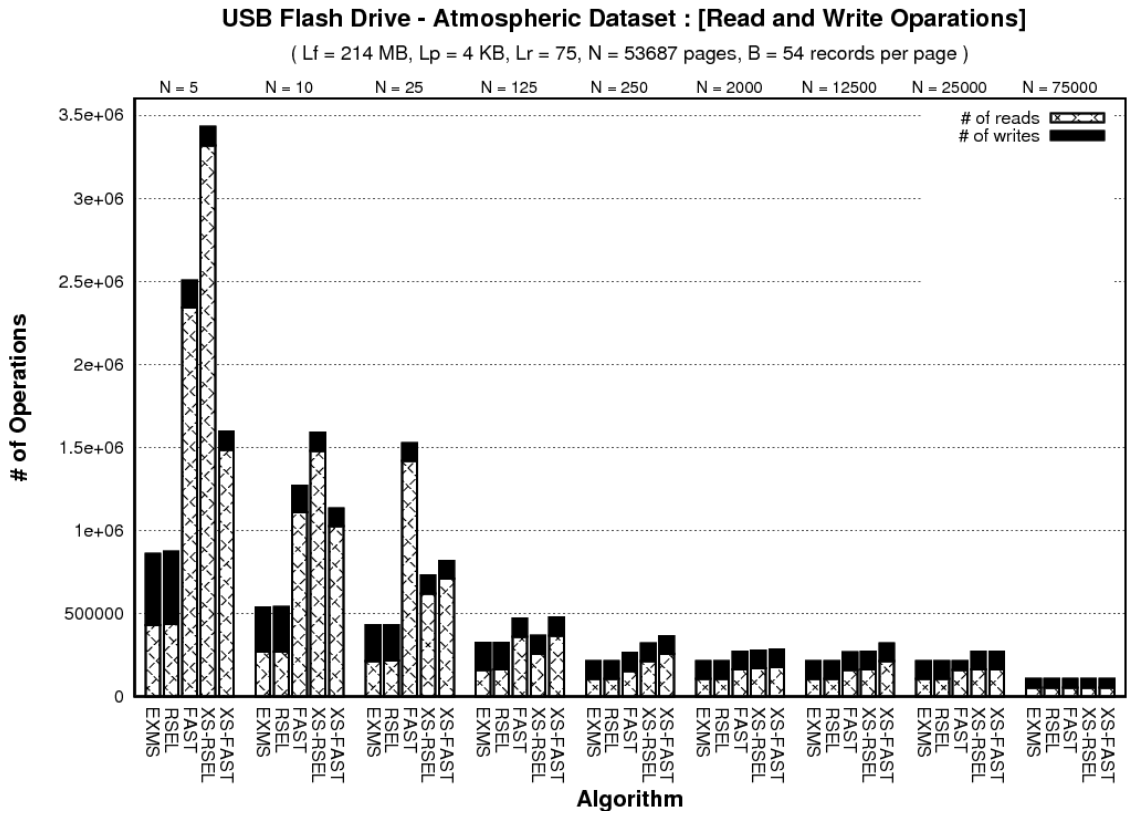
Όσον αφορά τον αριθμό των passes που παρουσιάζονται στην γραφική παράσταση του Σχήματος 6.2, παρατηρούμε ότι οι δύο αλγόριθμοι EXMS και RSEL χρειάζονται περισσότερα passes ενώ οι XSORT αλγόριθμοι έχουν πάντοτε σταθερό αριθμό passes (=2). Επίσης παρατηρούμε ότι ο αλγόριθμος FAST παρουσιάζει σχετικά μικρό αριθμό passes. Έτσι επιβεβαιώνονται πλήρως τα όσα αναφέρθηκαν προηγουμένως αφού βλέπουμε ότι μικρότερου μεγέθους κύρια μνήμη προκαλεί περισσότερα passes στους EXMS, RSEL και FAST ενώ ο αριθμός των passes στους XSROT αλγόριθμου παραμένει πάντοτε σταθερός και ίσος με 2.



Σχήμα 6.2 : USB Flash Drive – Atmospheric Dataset : Συνολικός αριθμός των passes

Συσχετίζοντας τις γραφικές παραστάσεις που παρουσιάζουν τον αριθμό των passes (Σχήμα 6.2) και τον αριθμό των I/O (Σχήμα 6.3), παρατηρούμε ότι ο αριθμός των

passes επηρεάζει και τον αριθμό των εγγραφών στην μνήμη flash. Πιο συγκεκριμένα ανεξάρτητος αλγόριθμου ο αριθμός των εγγραφών στην εξωτερική μνήμη είναι ανάλογος με τον αριθμό των passes (2 passes συνεπάγονται και 2N εγγραφές , 3 passes 3N έγγραφες κ.ο.κ). Ακόμη παρατηρούμε ότι ο αριθμός των αναγνώσεων και των εγγράφων στον EXMS είναι ακριβώς ο ίδιος για όλα τα test cases, στον RSEL είναι σχεδόν ο ίδιος ενώ στους άλλους τρείς αλγόριθμους οι αναγνώσεις είναι πολύ περισσότερες απ' ότι οι εγγραφές. Αυτό ήταν αναμενόμενο και οφείλεται στον τρόπο λειτουργίας του κάθε αλγόριθμου όπως περιγράφηκε στα κεφάλαια 3 και 4. Για παράδειγμα η λογική που εφαρμόζουν οι τρεις XS-FAST, XS-RSEL και FAST προσπαθεί να αντικαταστήσει τις χρονοβόρες εγγραφές με όσο το δυνατό λιγότερες αναγνώσεις (είτε με την παραγωγή μεγαλύτερου μεγέθους runs όπως ο FAST, είτε με τη χρήση ιστογραμμάτων όπως οι XSORT αλγόριθμοι).



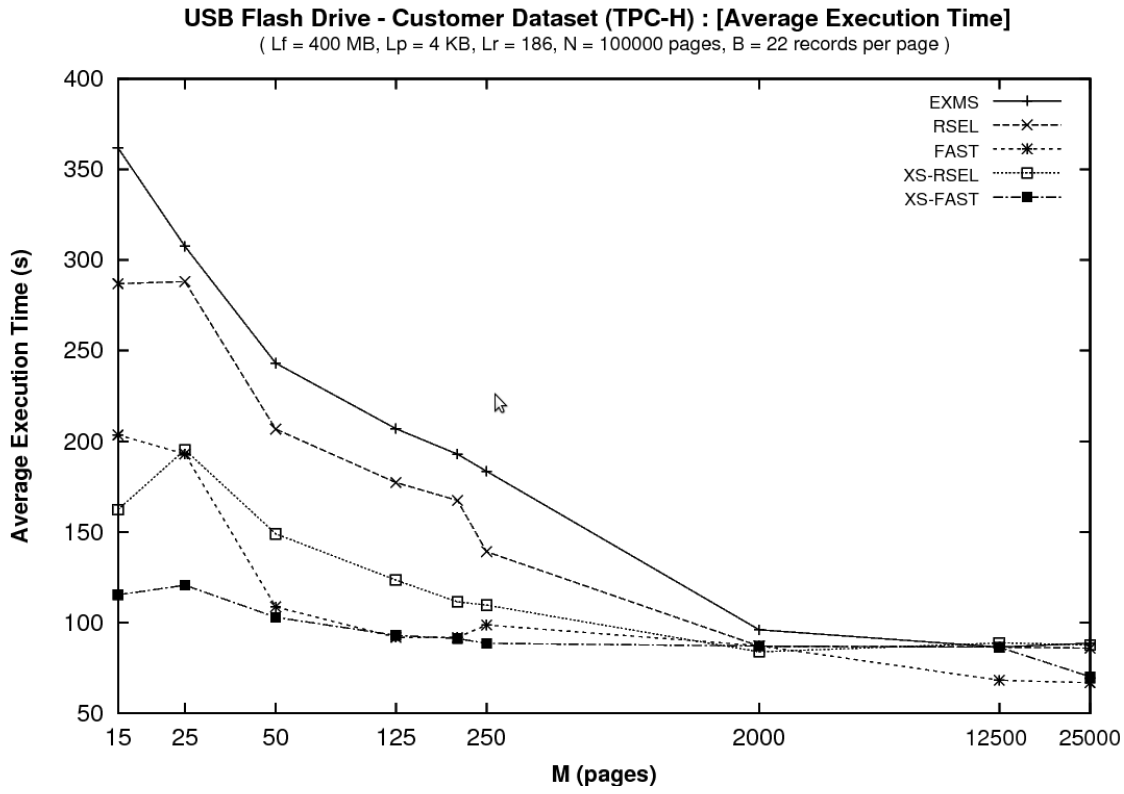
Σχήμα 6.3 : USB Flash Drive – Atmospheric Dataset : Συνολικός αριθμός I/O

Τέλος, παρατηρούμε ότι για M=75000 σελίδες (= 300MB) όλοι οι αλγόριθμοι παρουσιάζουν μια κοινή συμπεριφορά που αφορά ίδιο αριθμό passes (=1) και ίδιο αριθμό I/O (N εγγραφές και N αναγνώσεις). Επίσης παρατηρούμε ότι υπάρχει μια σύγκλιση προς ίδιους χρόνους εκτέλεσης. Αυτό οφείλεται στο γεγονός ότι στην

περίπτωση αυτή τα δεδομένα χωρούν ολόκληρα στην διαθέσιμη εσωτερική μνήμη και έτσι όλοι οι αλγόριθμοι κάνουν μόνο ένα pass για να παράγουν το τελικό αποτέλεσμα πράγμα που συνεπάγεται ότι ο συνολικός χρόνος εκτέλεσης για κάθε αλγόριθμο είναι στην ουσία ο χρόνος εκτέλεσης της πρώτης φάσης εκτέλεσής του.

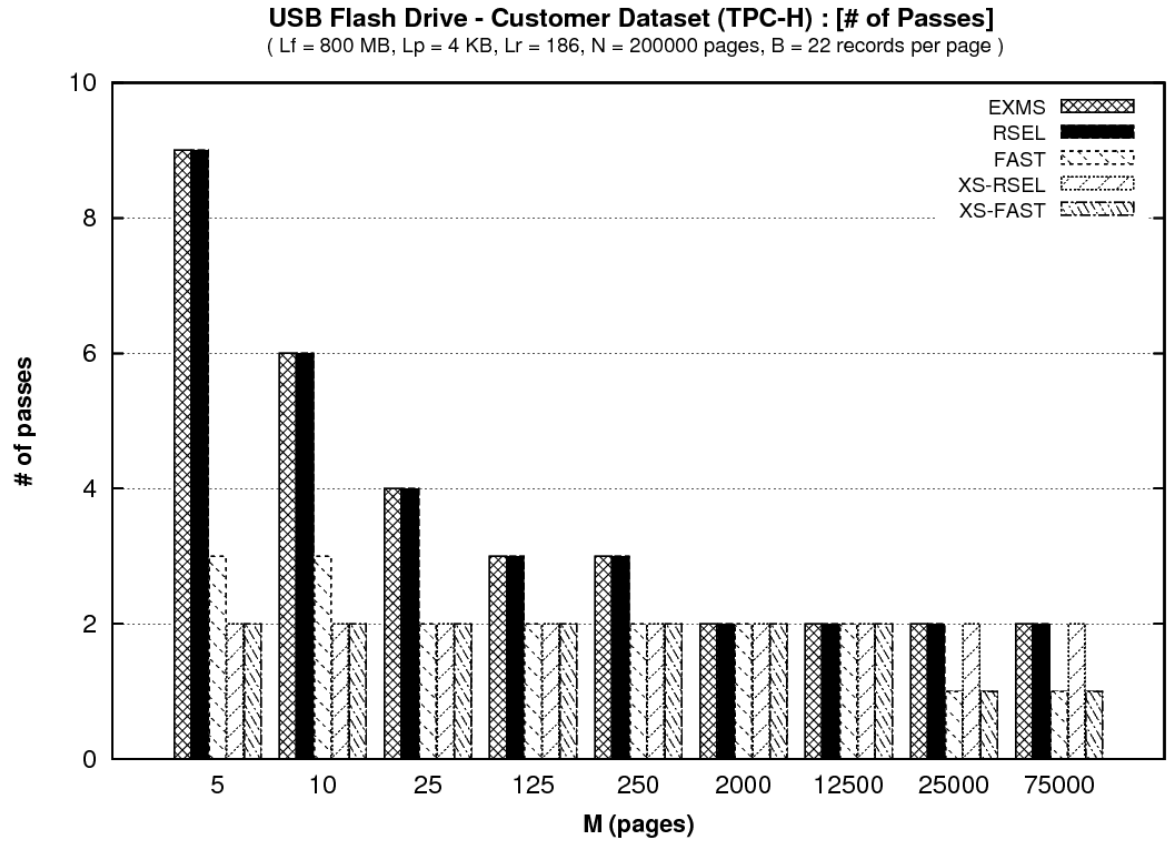
6.4 USB KEY – CUSTOMER DATASET

Σε αυτό το dataset τα προς ταξινόμηση δεδομένα είναι περισσότερα και η ταξινόμηση γίνεται σε κλειδί τύπου ακεραίου.

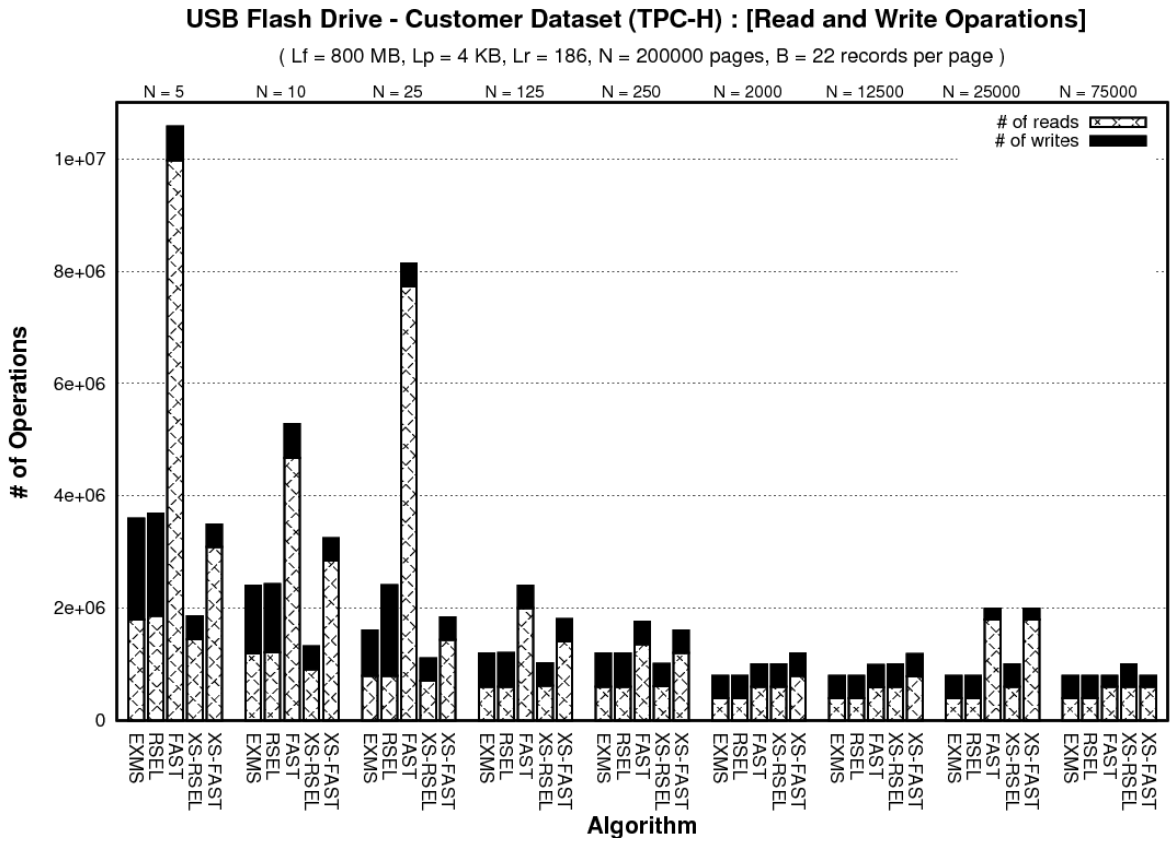


Σχήμα 6.4 : USB Flash Drive – Customer Dataset : Μέσος χρόνος εκτέλεσης

Κι εδώ επιβεβαιώνεται η υπεροχή του XS-FAST , ακολουθούμενος αυτή τη φορά από τον FAST, που τελικά αφήνει τρίτο τον XS-RSEL. Ακολουθεί ο RSEL και χειρότερος παρουσιάζεται ο EXMS. Επίσης παρατηρούμε ότι από στο διάστημα M = 2000-12500 (=8MB – 50MB) όλοι οι αλγόριθμοι παρουσιάζουν παραπλήσιους χρόνους εκτέλεσης και αυτό οφείλεται όπως θα δούμε και παρακάτω στο γεγονός ότι εκτελούν ίδιο αριθμό passes και κατ' επέκταση έχουν όλοι περίπου ίδιο IO κόστος.



Σχήμα 6.5 : USB Flash Drive – Customer Dataset: Συνολικός αριθμός των passes

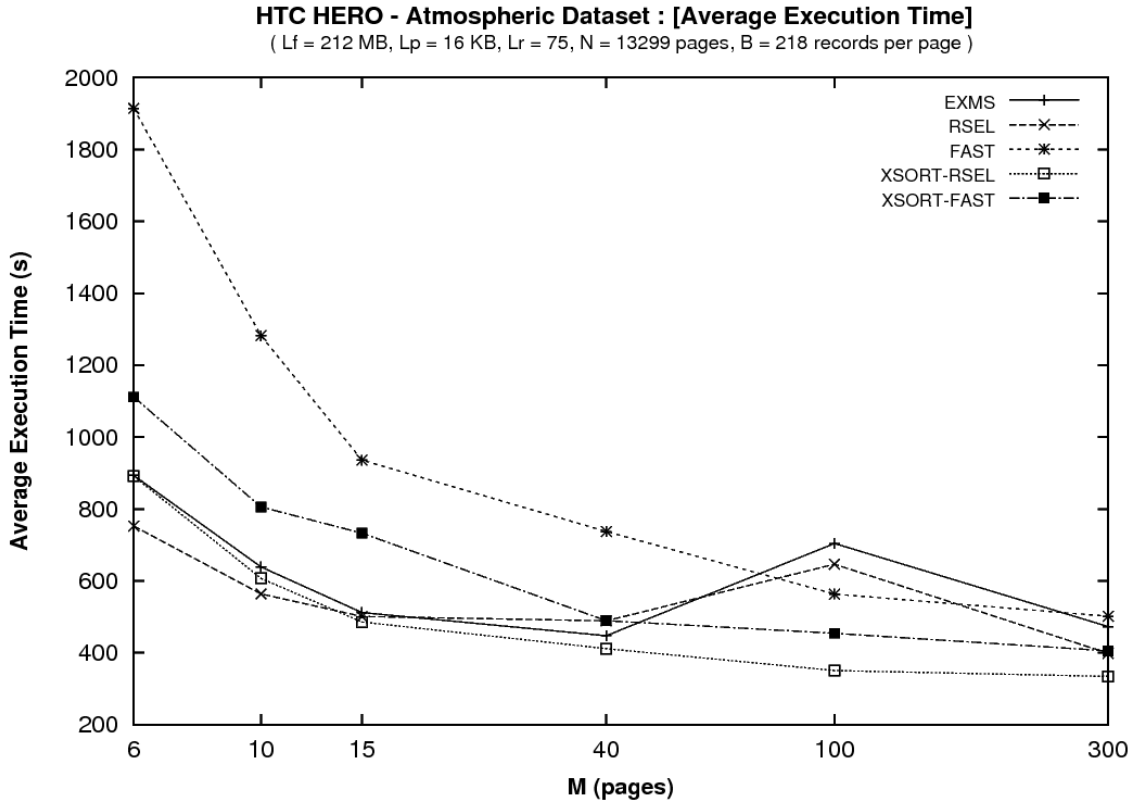


Σχήμα 6.6 : USB Flash Drive – Customer Dataset: Συνολικός αριθμός I/O

Βλέπουμε επίσης, ότι στο διάστημα M=25000-75000 (100MB-300MB) οι XS-FAST και FAST έχουν αισθητά μικρότερο χρόνο εκτέλεσης από του υπόλοιπους, παρατήρηση που εξηγείται αν δούμε τις γραφικές παραστάσεις του αριθμό των passes και του συνολικού αριθμού των IOs που χρειάζεται να κάνει κάθε αλγόριθμος για την ολοκλήρωση της εκτέλεσής του (Σχήμα 6.5 και 6.6 αντίστοιχα). Παρατηρούμε λοιπόν ότι στο διάστημα αυτό οι XS- FAST και FAST εκτελούν μόνο ένα pass. Αυτό οφείλεται στον τρόπο εκτέλεσης της πρώτης φάσεις του FAST που ταξινομεί το αρχείο με ένα pass. Στην ουσία σε αυτό το σημείο ο XS-FAST "εκμεταλλεύεται" κατά κάποιο τρόπο την λογική εκτέλεσης της πρώτης φάσης του αλγόριθμου FAST.

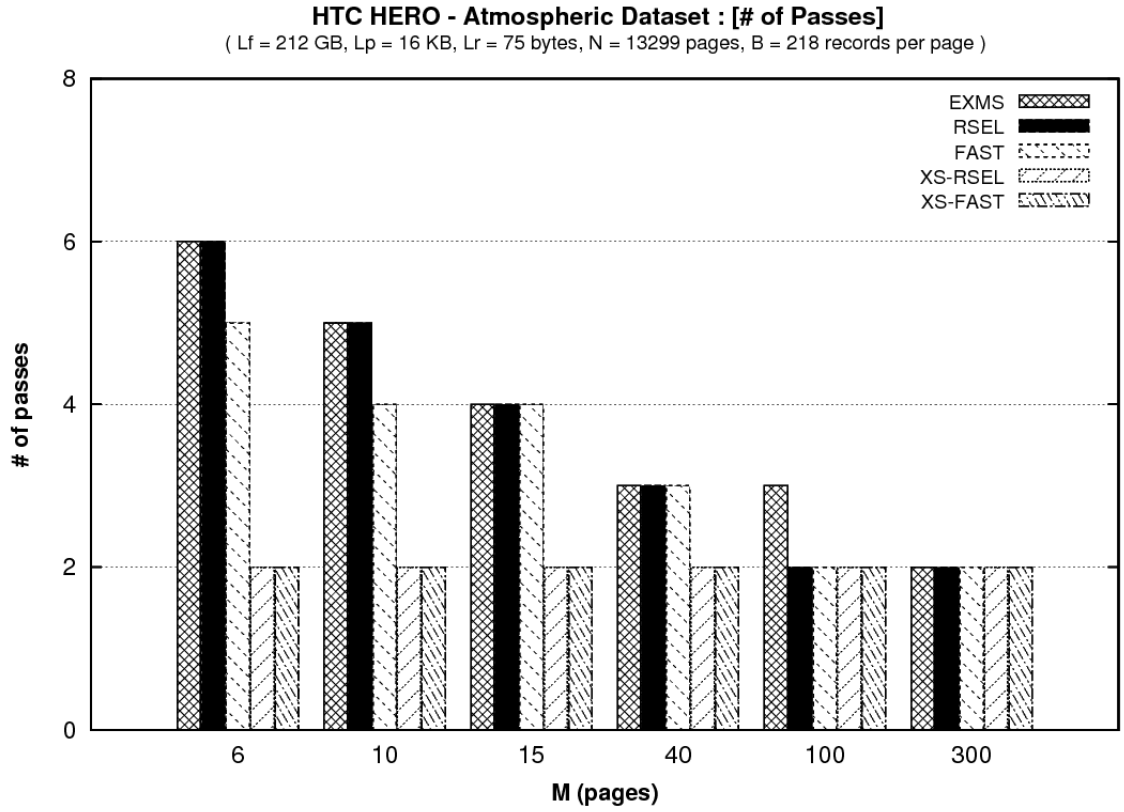
6.5 HTC HERO – ATMOSPHERIC DATASET

Σε αυτό το πείραμα τα αποτελέσματα ήταν διαφορετικά σε σχέση με τα υπόλοιπα στο σύνολό τους πειράματα.



Σχήμα 6.7 : HTC HERO – Atmospheric Dataset : Μέσος χρόνος εκτέλεσης

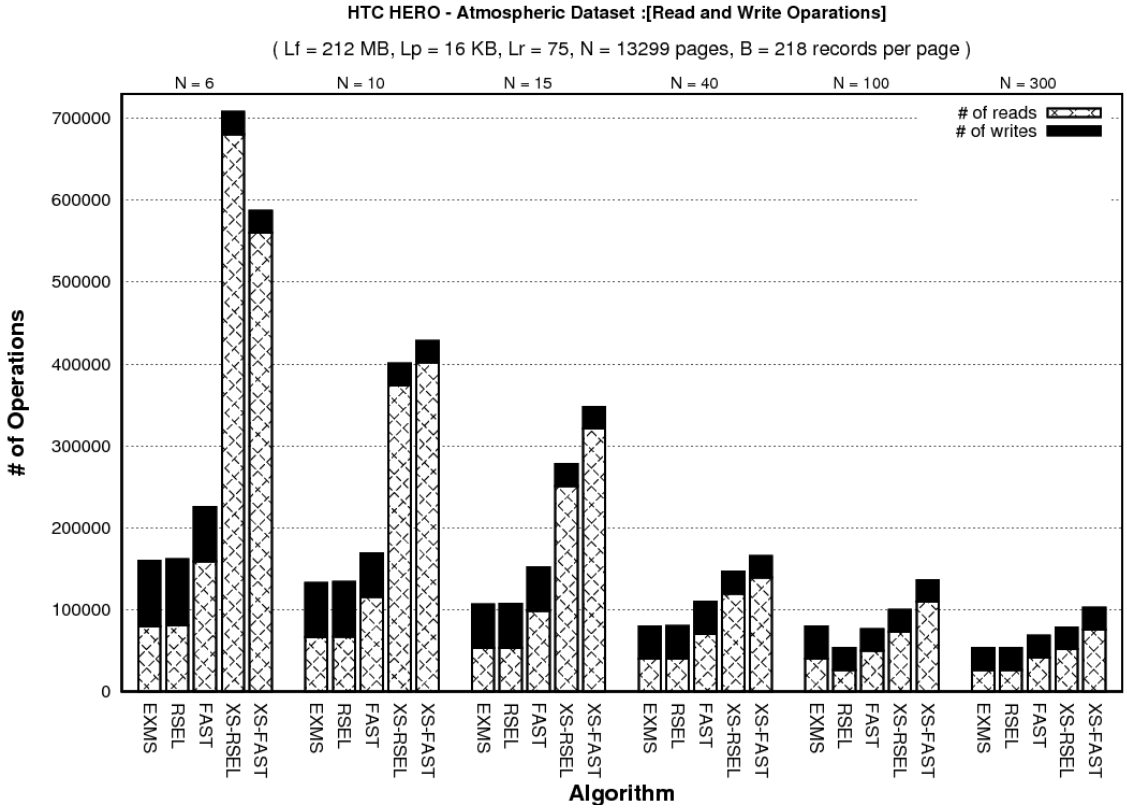
Αν και ο αλγόριθμος XS-RSEL παρουσιάζει τον καλύτερο χρόνο εκτέλεσης όπως φαίνεται και στην γραφική παράσταση του Σχήματος 6.7 εντούτοις η διαφορά από τους RSEL και EXMS που ακολουθούν είναι πολύ μικρή, ενώ οι αλγόριθμοι FAST και XS-FAST παρουσιάζουν χρόνους χειρότερους ακόμη και από τον EXMS. Πιο συγκεκριμένα χειρότερος παρουσιάζεται ο FAST ενώ ακολουθεί ο XS-FAST. Από την άλλη οι αλγόριθμοι RSEL και EXMS παρουσιάζουν μια αστάθεια στους χρόνους εκτέλεσης τους με αποκορύφωμα την απότομη αύξηση του χρόνου εκτέλεσης τους για M=100 (=1,6MB) όπου ο χρόνος εκτέλεσης εκτοξεύεται στα 1000 δευτερόλεπτα. Αν προσπαθήσουμε να βρούμε την αιτία σε θέματα που αφορούν των αριθμό των passes και το I/O κόστος οι αλγόριθμοι RSEL και EXMS όπως ήταν αναμενόμενο είναι πράγματι αυτοί που παρουσιάζουν τα περισσότερα passes και κατ' επέκταση τις περισσότερες έγγραφες στη μνήμη flash όπως βλέπουμε και στις αντίστοιχες γραφικές παραστάσεις (Σχήμα 6.8 και 6.9).



Σχήμα 6.8 : HTC HERO – Atmospheric Dataset: Συνολικός αριθμός των passes

Να θυμίσουμε επίσης ότι βάση του μοντέλου συστήματος που ορίσαμε εξαρχής, C_w = α C_R. Αυτό πρακτικά σημαίνει ότι μπορώ αν αντικαταστήσω κάθε εγγραφή ενός

αλγόριθμου με α αριθμό αναγνώσεων και θα έχω ακριβώς το ίδιο I/O κόστος. Αν το α είναι μικρό τότε μικραίνει και ο αριθμός των αναγνώσεων που αντιστοιχούν σε μια εγγραφή. Θα μπορούσαμε να πούμε ότι σε αυτή την περίπτωση, το α είναι αρκετά μικρό και οι πολλές εγγραφές εμφανίζουν ένα επιπλέον κόστος. Κάτι τέτοιο δεν φαίνεται όμως να ισχύει γιατί αυτό θα επηρέαζε και το χρόνο του XS-RSEL, ο οποίος όπως φαίνεται και στην πιο κάτω γραφική παρουσιάζει αρκετά μεγάλο αριθμό αναγνώσεων, ιδιαίτερα για μικρότερες μνήμης.



Σχήμα 6.9 : HTC HERO – Atmospheric Dataset: Συνολικός αριθμός I/O

Μια άλλη σκέψη είναι ότι σε αυτή τη συσκευή ίσως να υπάρχει μεγάλο κόστος στη χρήση του επεξεργαστή που επηρεάζει το συνολικό χρόνο εκτέλεσης του αλγόριθμου. Αυτό μπορεί να ισχύει αν λάβουμε υπόψη μας ότι στην συσκευή τρέχει και ένας μεγάλος αριθμός εφαρμογών του κινητού τηλεφώνου που συναγωνίζονται με την εκτέλεση του. Γενικά ο πολύπλοκος μηχανισμός του FAST με τις πολλές συγκρίσεις που εκτελεί και στις δύο φάσεις του, πιθανών να επιβραδύνει την εκτέλεση. Αναζητώντας τα βαθύτερα αίτια αυτής της συμπεριφοράς χωρίσαμε τον χρόνο εκτέλεσης σε δύο φάσης για κάθε αλγόριθμο. Στους αλγόριθμους που κάνουν χρήση της λογικής της συνένωσης η δύο ατές φάσης είναι προφανείς. Όσον αφορά τώρα τους

XSORT αλγόριθμους θεωρούμε ότι η πρώτη φάση είναι η φάση της παραγωγής των αρχικών runs και του υπολογισμού του πεδίου ορισμού του ιστογράμματος ενώ η δεύτερη φάση αποτελεί την φάση της παραγωγής του ιστογράμματος καθώς επίσης και την τελική φάση της δημιουργίας του ταξινομημένου αποτελέσματος. Στην συνέχεια παρουσιάζουμε δύο πίνακες που συνοψίζουν τον χρόνο εκτέλεσης της πρώτης (Πίνακας 6.1) και της δεύτερης (Πίνακας 6.2) φάσης του αλγόριθμου.

HTC – ATMOSPHERIC DATASET (PHASE 1(s))					
M	EXMS	RSEL	FAST	XS-RSEL	XS-FAST
6	337.156	283.672	423.015	194.2462	435.003
10	259.961	226.212	208.0605	182.2202	267.588
15	161.0394	184.7352	190.908	177.4046	263.754
40	181.531	232.174	215.866	222.9722	248.605
100	193.084	249.157	268.8795	213.459	243.102
300	195.3548	257.7638	239.526	224.8726	246.408

Πίνακας 6.1 : HTC HERO – Atmospheric Dataset: Χρόνος εκτέλεσης πρώτης φάσης

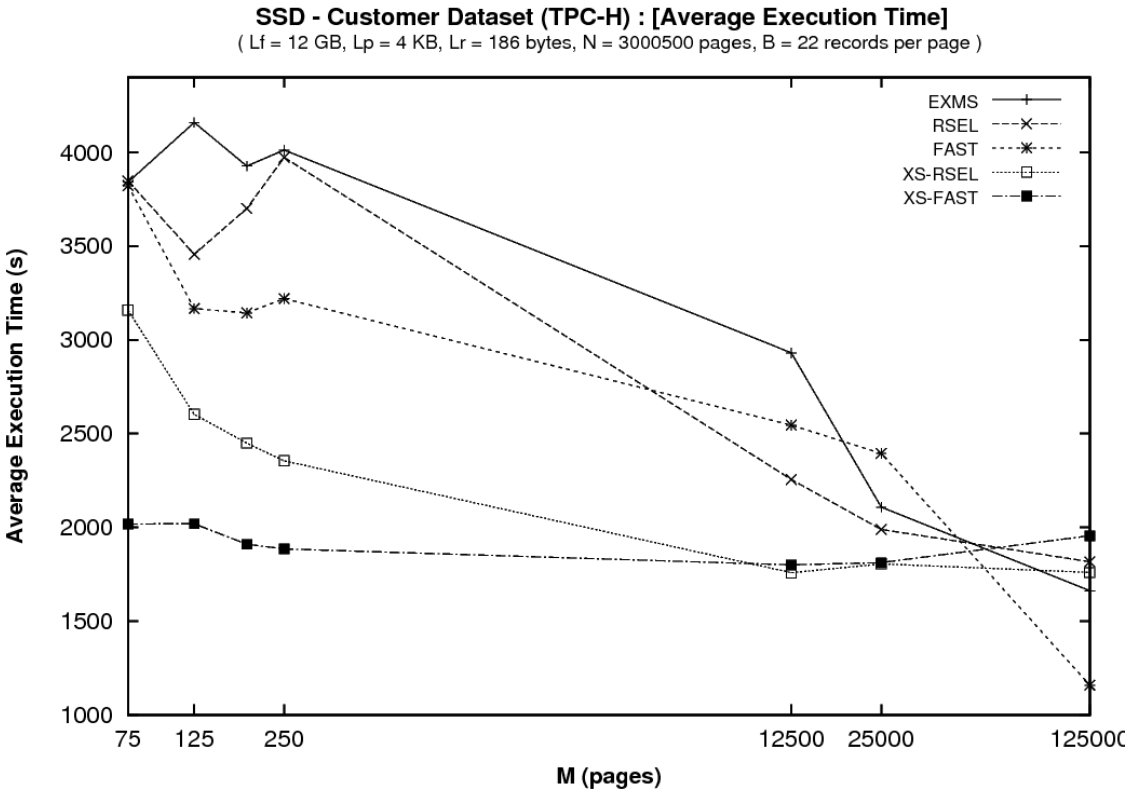
HTC – ATMOSPHERIC DATASET (PHASE 2(s))					
M	EXMS	RSEL	FAST	XS-RSEL	XS-FAST
6	529.063	443.412	1491.2	697.2264	676.44
10	378.847	337.2	1073.7	424.6182	538.02
15	260.618	249.4166	745.1	307.9218	469.17
40	265.534	256.358	521.31	188.4438	240.32
100	530.572	397.031	294.16	137.0402	211.05
300	251.187	127.9502	261.62	108.59	158.53

Πίνακας 6.2 : HTC HERO – Atmospheric Dataset: Χρόνος εκτέλεσης δεύτερης φάσης

Από τα πιο πάνω φαίνεται ότι η διαδικασία της παραγωγής των αρχικών runs (πρώτη φάση) για τον FAST και XS-FAST διαρκεί πολύ περισσότερο χρόνο από τη η αντίστοιχη διαδικασία στους άλλους αλγόριθμους. Επίσης η δεύτερη φάση του FAST παρουσιάζει τους χειρότερους χρόνους από όλους τους αλγόριθμους.

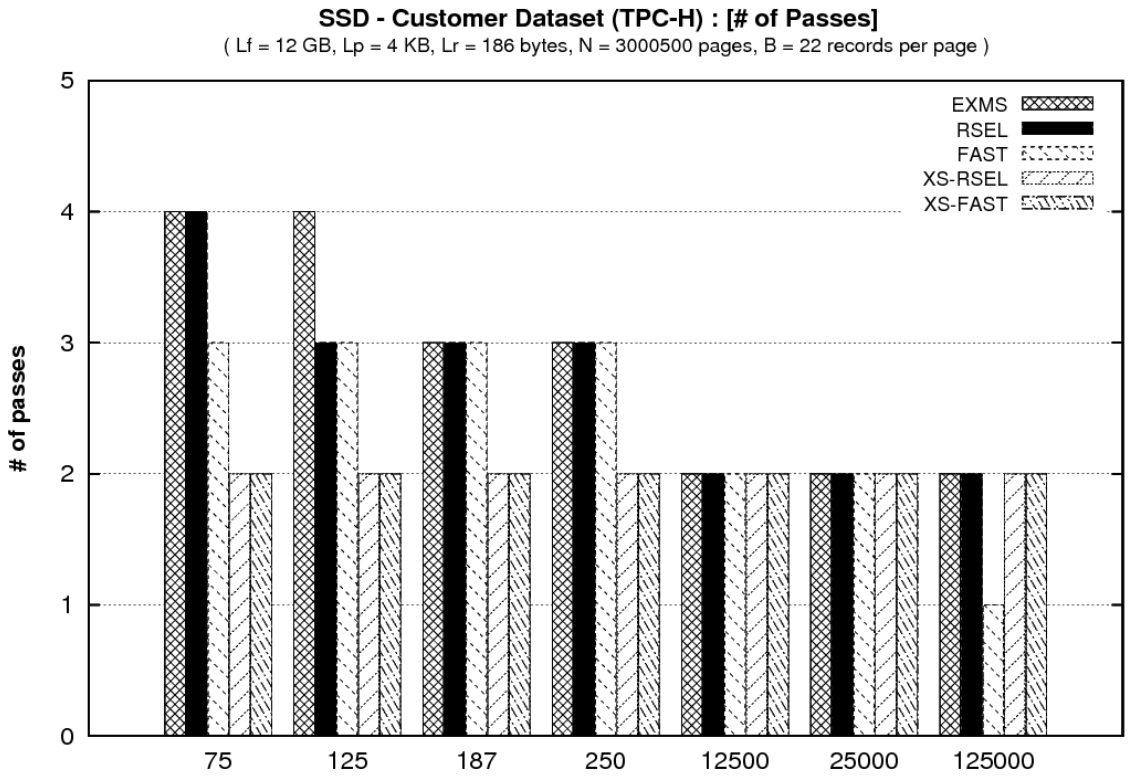
6.6 SSD – CUSTOMER DATASET

Σε αυτό το πείραμα έχουμε να κάνουμε με ένα τεράστιο όγκο δεδομένων (= 12 GB) και τη χρήση ενός SSD.

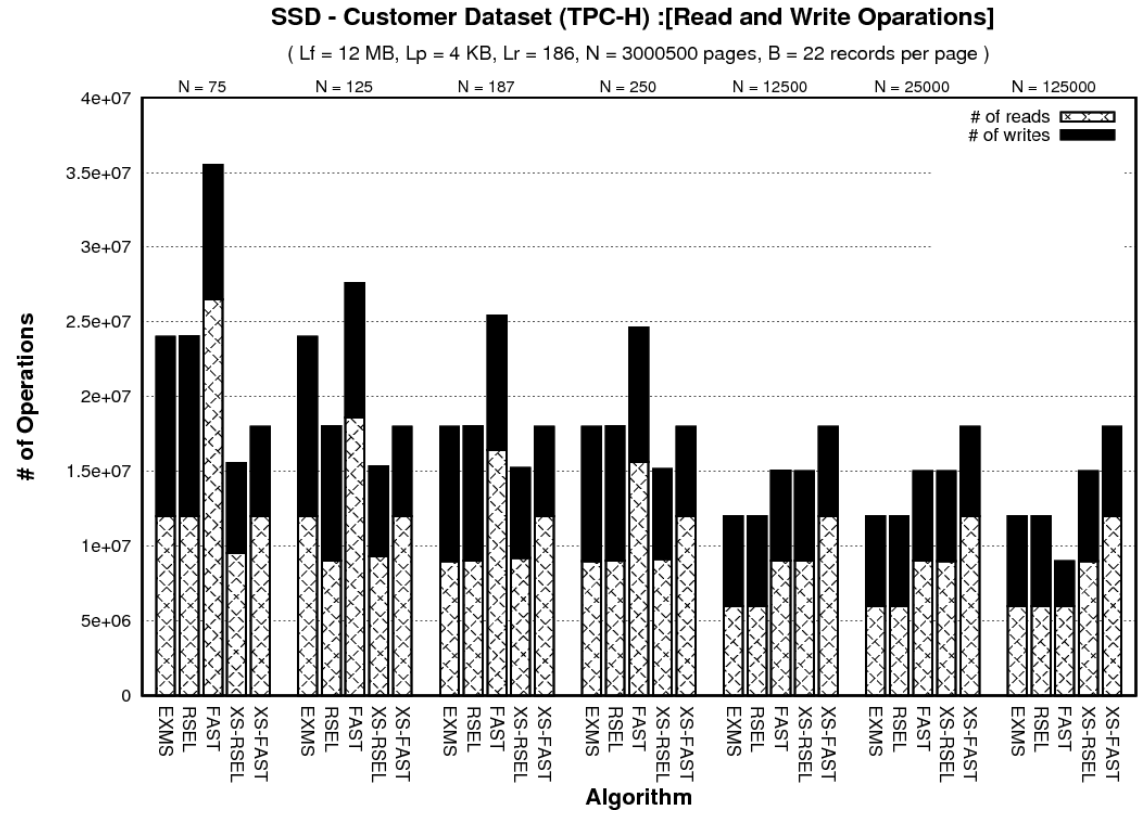


Σχήμα 6.10 : SSD – Customer Dataset : Μέσος χρόνος εκτέλεσης

Σύμφωνα λοιπόν με την γραφική παράσταση του μέσου χρόνου εκτέλεσης των αλγορίθμων (Σχήμα 6.10) βλέπουμε και πάλι την υπεροχή των δύο παραλλαγών του XSORT σε σχέση με τους υπόλοιπους αλγόριθμους, με τον XS-FAST να παρουσιάζει και πάλι καλύτερα αποτελέσματα από τον XS-RSEL. Επόμενος καλύτερος ο FAST, ακολουθεί ο RSEL ενώ χειρότερος παρουσιάζεται ο EXMS. Σε γενικές γραμμές, οι XSORT αλγόριθμοι παρουσιάζονται περίπου 2 φορές πιο γρήγοροι απ' ότι οι EXMS και RSEL και 1,5 φορά από ότι ο FAST. Φαίνεται πως ο μεγάλος όγκος δεδομένων βοήθησε στην ανάδειξη της διαφοράς μεταξύ της λογικής χρήσεως ιστογραμμάτων και της κλασσικής λογικής της συνένωση.



Σχήμα 6.11 : SSD – Customer Dataset: Συνολικός αριθμός των passes



Σχήμα 6.12 : SSD – Customer Dataset: Συνολικός αριθμός I/O (αναγνώσεων και εγγραφών)

Επίσης και σε αυτό το πείραμα, επιβεβαιώνεται και πάλι η σύνδεση του αριθμού των passes με την αύξηση των εγγραφών και κατ’ επέκταση και του χρόνου εκτέλεσης του αλγόριθμου.

Κεφάλαιο 7

Συμπεράσματα και Μελλοντικές Επεκτάσεις

7.1 Συμπεράσματα

Σύμφωνα λοιπόν με τη μελέτη που έχουμε κάνει βρήκαμε τόσο θεωρητικά όσο και πειραματικά ότι ο αριθμός των passes ανεξαρτήτως αλγόριθμου αυξάνει και τον αριθμό των I/Os που εκτελεί ο κάθε αλγόριθμος. Στην ουσία για κάθε pass χρειάζονται N αναγνώσεις και N εγγραφές αφού ένα pass προϋποθέτει την ανάγνωση ολόκληρων των δεδομένων σε οποιαδήποτε σειρά από την εξωτερική μνήμη (μνήμη flash), την επεξεργασία τους στην KME και τελικά την εγγραφή τους σε μορφή run(s) στην εξωτερική μνήμη. Λόγω της ασυμμετρίας που παρατηρείται ανάμεσα στις λειτουργίες της ανάγνωσης και της εγγραφής στην μνήμη flash, μας ενδιαφέρει περισσότερο να μειώσουμε τις χρονοβόρες εγγραφές παρά τις αναγνώσεις στη μνήμη flash που παρουσιάζουν λιγότερο κόστος.

Αν η μείωση των passes μπορεί να γίνει με ένα επιπλέον κόστος αναγνώσεων αυτό θα είχε αποτέλεσμα και την μείωση των χρονοβόρων εγγραφών, αντικαθιστώντας τις χρονοβόρες εγγραφές με όσο το δυνατό λιγότερες αναγνώσεις. Την λογική αυτή ακολουθούν οι τρεις αλγόριθμοι XSORT-FAST, XSORT-RSEL και FAST με τη χρήση δύο διαφορετικών τεχνικών. Ο πρώτοι κάνουν χρήση ιστογραμμάτων και εκτελούν

μόνο δύο passes ενώ ο δεύτερος παράγει μεγαλύτερου μεγέθους runs αποσκοπώντας στην μείωση των passes. Τα πειράματα που έγιναν έδειξαν ότι πράγματι η λογική αυτή δουλεύει κυρίως σε περιπτώσεις όπου η εσωτερική μνήμη είναι πολύ μικρή και οι άλλοι δύο αλγόριθμοι θα χρειαστούν περισσότερα passes για την εκτέλεση της ταξινόμησης. Εκεί που υστερεί ο FAST είναι ότι αν και εκτελεί λιγότερα passes δεν ξεφεύγει από την λογική της συνένωσης που επιβάλλει αναδρομικές συνενώσεις των παραγόμενων σε κάθε pass runs και δεν εγγυάται σταθερό ή μικρό αριθμό passes σε όλα τα σενάρια.

Τέλος οι πειραματικές μετρήσεις που έγιναν έδειξαν επίσης ότι η χρήση ιστογραμμάτων και της τεχνικής που χωρίζει το προς ταξινόμηση αρχείο σε ενεργά και νεκρά σημεία έχει οδηγήσει στο να κρατήσουμε σταθερό των αριθμό των εγγραφών και να μειώσουμε όσον το δυνατό τις επιπλέον αναγνώσεις, πράγμα που οδηγεί τον αλγόριθμο σε μικρότερους χρόνους εκτέλεσης. Γενικά, τα πειράματα έδειξαν πως η προτεινόμενη λογική παρουσιάζει από 35-50% καλύτερη απόδοση από όλους τους υπόλοιπους αλγόριθμους και επίσης ότι μπορεί να δουλέψει αποδοτικά ακόμα κι αν ο όγκος δεδομένων είναι αρκετά μεγάλος (πειράματα σε Solid-State δίσκο με 12GB προς ταξινόμηση δεδομένα) .

Η παρούσα διπλωματική μπορεί να αποτελέσει το έναυσμα ότι θα πρέπει να ξεφύγουμε από την λογική της συνένωσης όσον αφορά το πρόβλημα της εξωτερικής ταξινόμησης σε μνήμες Flash και να ψάξουμε τεχνικές που να κρατούν σταθερό των αριθμό των εγγραφών και εκτελούν όσον το δυνατό λιγότερο αριθμό επιπλέον αναγνώσεων.

7.2 Μελλοντικές Επεκτάσεις

Αν και η λογική φαίνεται να δουλεύει αποδοτικά, εντούτοις βρίσκεται σε πολύ αρχικό στάδιο. Επίσης για να είμαστε και ακριβοδίκαιοι τα χαρακτηριστικά που επιλεγήκαν ως κλειδιά στα πειράματα που έγιναν είχαν μικρό πεδίο τιμών, πράγμα που σημαίνει ότι ήταν εφικτή η δημιουργία των ιστογραμμάτων σε ένα ή δύο file scans. Θα πρέπει ο αλγόριθμος να δοκιμαστεί και σε μεγαλύτερου πεδίου ορισμού κλειδιά ταξινόμησης Από την άλλη συνήθως τα χαρακτηριστικά παρουσιάζουν κατανομές δεδομένων όπως αυτές που επιλέγηκαν στα πειράματα. Να θυμίσουμε επίσης ότι το ένα dataset ήταν πραγματικό και το άλλο αποτελούσε μέρος ενός κοινά αποδεκτού benchmarktou TPC.

Όσον αφορά τώρα την μελλοντική δουλεία που πρέπει να γίνει, αυτή αφορά τον καθορισμό του τρόπου με τον οποίο θα γίνεται η ταξινόμηση και βάση ενός κλειδιού τύπου κειμένου καθώς επίσης και να εξηγηθούν οι λόγοι για την συμπεριφορά των αλγόριθμων στη συσκευή. Τέλος θα πρέπει να μελετηθεί και η ένταξη της παραλληλίας και τεχνικών όπως το Double buffering στον XSORT αλγόριθμο πράγμα που αναμένεται να ανεβάσει την απόδοσή του.

Βιβλιογραφία

[1] **A. Aggarwal, and J. S. Vitter**. "The input/output complexity of sorting and related problems." *Communications of the ACM*, 1988: 1116–1127.

[2] **P. Andreou, O. Spanos, D. Zeinalipour, G. Samaras, and P. K. Chrysanthis**. "FSort: External Sorting on Flash-based Sensor Devices." *DMSN '09*, 2009.

[3] **A. Birrell, M. Isard, C. Thacker, and T. Wobber**. "A design for high-performance flash disks." *ACM Operating Systems Review*, 2007: 88–93.

[4] **T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein**. "More on Sorting Algorithms." Chap. 5 in *Introduction to Algorithms*, 167-187. MIT Press and McGraw-Hill, 2001.

[5] **Y. Diao, D. Ganesan, G. Mathur, and P. Shenoy**. "Rethinking Data Management for Storage-centric Sensor Networks." *CIDR*, 2007.

[6] **A. Goldberg, and R. Werneck**. "Computing point-to-point shortest paths from external memory." *7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, 2005, SIAM ed.

[7] **R. C. Gonzalez, and R. E. Woods**. "Image Enhancement in the Spatial Domain." In *Digital Image Processing*. New Jersey: Prentice Hall, 2008.

[8] **G. Graefe**, "The five-minute rule twenty years later, and how flash memory changes the rules,." *3rd international workshop on Data management on new hardware*, June 2007.

[9] **HTC**. http://www.htc.com/www/product/hero/overview.html.

[10] **In-Stat**. 2010. http://www.in-stat.com.

[11] **Y. Ioannidis**, "Query optimization." *ACM Computing Surveys (CSUR)*, March 1996: 121-123.

[12] **Kingston**. http://www.kingston.com/ukroot/ssd/v_series.asp.

[13] **D. E. Knuth**, *Sorting and Searching.* Vol. 3, in *The Art of Computer Programming*. Addison Wesley, 1998.

[14] **I. Koltsidas, and S. D. Viglas**. "Flashing Up the Storage Layer." *Proceedings of the VLDB Endowment*, August 2008.

[15] **S. Lee, and B. Moon**. "Design of flash-based DBMS: an in-page logging approach." *SIGMOD Conference*, 2007, ACM ed.: 55–66.

[16] **S. Lee, , B. Moon, and C. Park**. *Advances in Flash Memory SSD Technology for Enterprise Database Applications.* Providence, Rhode Island, U.S.A: SIGMOD'09, 2009.

[17] **G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy**. "Capsule: An Energy-Optimized Object Storage System for Memory-Constrained Sensor Devices." *4th international conference on Embedded networked sensor systems*, 2006, November ed.

[18] **D. Myers, and S. Madden**. "On the use of NAND flash disks in high-performance relational databases." 2007.

[19] **S. Nath, and A. Kansal**. "FlashDB: Dynamic Self-tuning Database for NAND Flash." *6th international conference on Information processing in sensor networks*, April 2007.

[20] **H. Park, and K. Shim**. "FAST: Flash-aware external sorting for mobile database systems." *Journal of Systems and Software*, 2009: 1298-1312.

[21] **Samsung**. http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_OneNAND.html.

[22] **Sandisk**. http://www.sandisk.com/products/computing-products/sandisk-micro-usb-flash-drive.

[23] **Sandisk**. http://www.sandisk.com/business-solutions/flash-memory-cards/sandisk-microsd-and-microsdhc-cards.

[24] **H. H. Seward**, *Master's thesis.* MIT Digital Computer Laboratory Report R-232, 1954.

[25] **SQLite**. http://www.sqlite.org/.

[26] **Transaction Processing Performance Council**. 2008. http://www.tpc.org.

[27] **D. Tsirogiannis**, S. Harizopoulos, and M. A. Shah. *Query Processing Techniques for Solid State Drives.* Providence, Rhode Island, USA: SIGMOD'09, 2009.

[28] **University of Washington, Seattle**. http://www-k12.atmos.washington.edu/k12/grayskies/.

[29] **J. S. Vitter**, *Algorithms and Data Structures for External Memory.* Hanover, MA: now Publishers, 2008.

[30] **Wikipedia, the free encyclopedia**. *Computer data storage.* http://en.wikipedia.org/wiki/Computer_data_storage.

[31] **C. H. Wu, L. P. Chang, and T. W. Kuo**. "An efficient R-tree implementa tion over flash-memory storage systems." *Proceedings of the eleventh ACM international symposium on Advances in geographic information systems*, 2003, ACM Press ed.: 17–24.

[32] **C. H. Wu, L. P. Chang, and T. W. Kuo**. "An efficient B-tree layer for flash-memory storage systems." *RTCSA, New Orleans*, February 2003.

[33] **T. Yoshii**, C. Black, and S. Chahal. *Solid-State Drives in the Enterprise: A Proof of Concept.* Intel Information Technology, Computer Manufacturing Data Storage, 2009.

[34] **D. Zeinalipour**, S. Lin, V. Kalogeraki, D. Gunopoulos, and W. Najjar. "MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices." *Usenix FAST*, 2005: 31-44.

Παράρτημα Α

Οδηγός Εγκατάστασης/Χρήσης Εφαρμογών C/C++ σε Περιβάλλον Android mOS

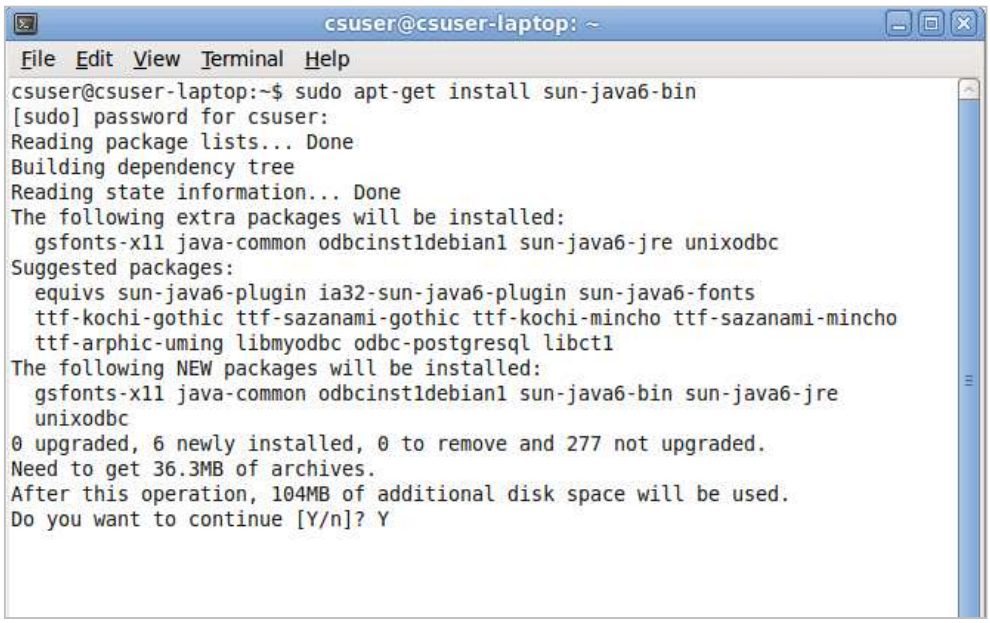
Στο Παράτημα αυτό θα περιγράψουμε τον τρόπο με τον οποίο μπορούμε να μεταγλωττίσουμε, να εγκαταστήσουμε και να εκτελέσουμε τις δικές μας εφαρμογές γραμμένες σε γλώσσα προγραμματισμού C/C++ σε περιβάλλον Android mOS. Γενικά για να είναι αυτό κατορθωτό θα πρέπει να εγκαταστήσουμε στον υπολογιστή μας δύο συλλογές εργαλείων που βοηθούν στην ανάπτυξη και τον έλεγχο εφαρμογών για Android περιβάλλοντα. Οι δύο αυτές συλλογές είναι το Android-SDK και το Android-NDK τις οποίες μπορούμε να προμηθευτούμε από την επίσημη ιστοσελίδα των Android Developers με URL: http://developer.android.com/index.html. Στην σελίδα αυτή υπάρχει μια τεράστια συλλογή πληροφοριών για ανάπτυξη εφαρμογών σε Android συστήματα αλλά παρόλα αυτά το κομμάτι που αφορά εφαρμογές γραμμένες σε γλώσσα προγραμματισμού C/C++ είναι ανεπαρκής. Σκοπός μας είναι να δώσουμε μια καθαρή περιγραφή για το πώς θα μπορούσε κάποιος να εκτελέσει τέτοιου είδους εφαρμογές στο Android mOS ξεκινώντας από την απαραίτητη προεργασία και φτάνοντας μέχρι και την τελική εκτέλεση ενός παραδείγματος που θα παρουσιάσουμε. Να αναφέρουμε ότι η διαδικασία που θα περιγραφεί ακολουθήθηκε στα πειράματα που έγιναν σε κινητή συσκευή HTC-Hero και αφορά λειτουργικό Ubuntu 9.10.

Γ.1 Προεργασία (Εγκατάσταση Java και Eclipse)

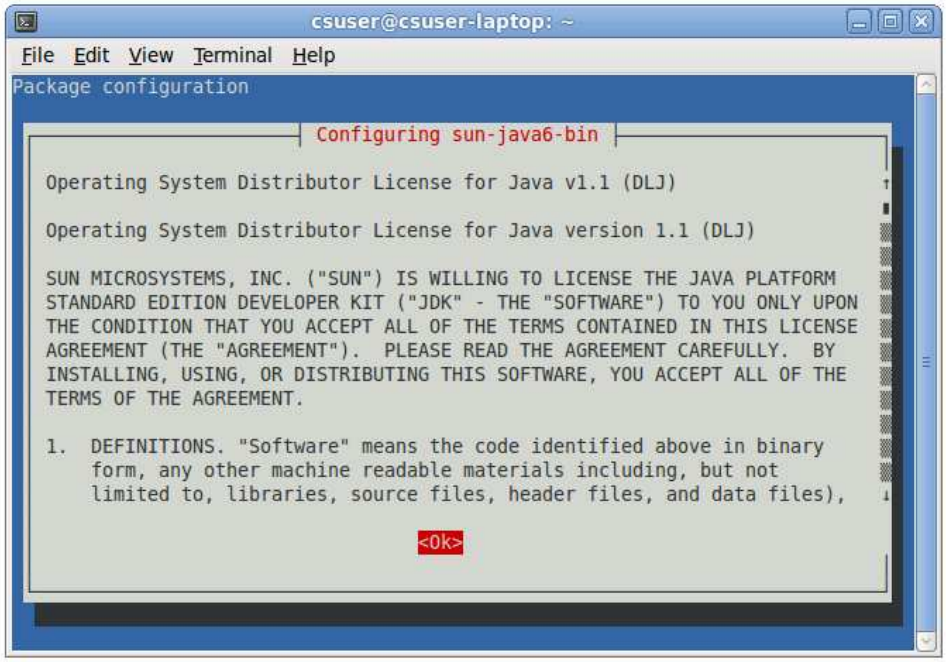
Αρχικά θα πρέπει να εγκατασταθούν στον υπολογιστή μας η έκδοση 6 της Java και μια από τις εκδόσεις του Eclipse 3.3, 3.4 ή 3.5 απαραίτητα για την εγκατάσταση του Android-SDK. Όσον αφορά την Java, πληκτρολογούμε σε ένα τερματικό (Applications > Accessories > Terminal) την ακόλουθη εντολή:

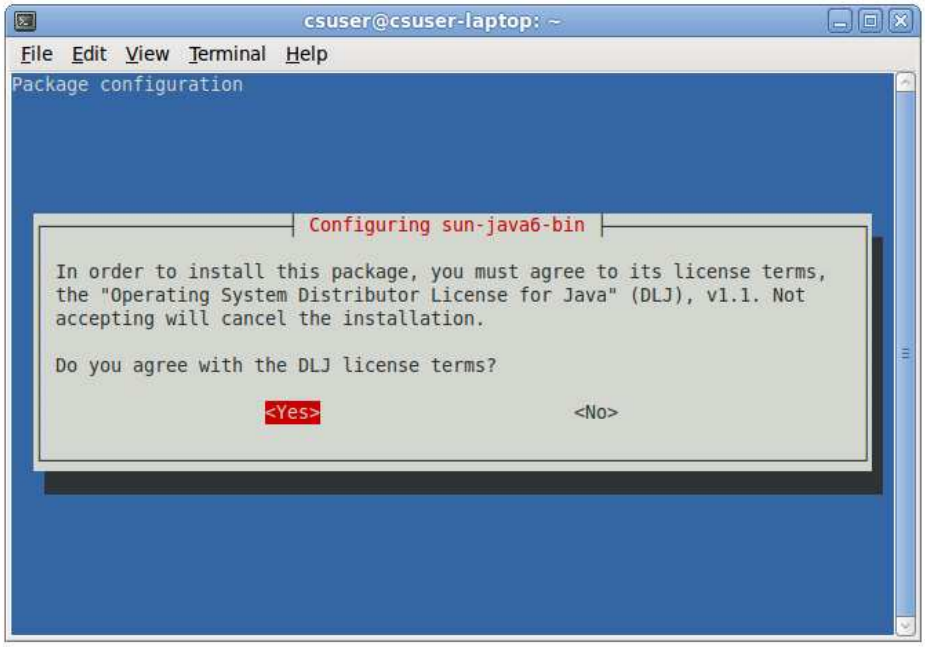
```
sudo apt-get install sun-java6-bin
```

Η εντολή θα έχει ως αποτέλεσμα την έναρξη της διαδικασίας εγκατάστασης της Java. Πληκτρολογούμε Y στην πρώτη ερώτηση που μας γίνεται, έτσι ώστε να συνεχίσει η διαδικασία .

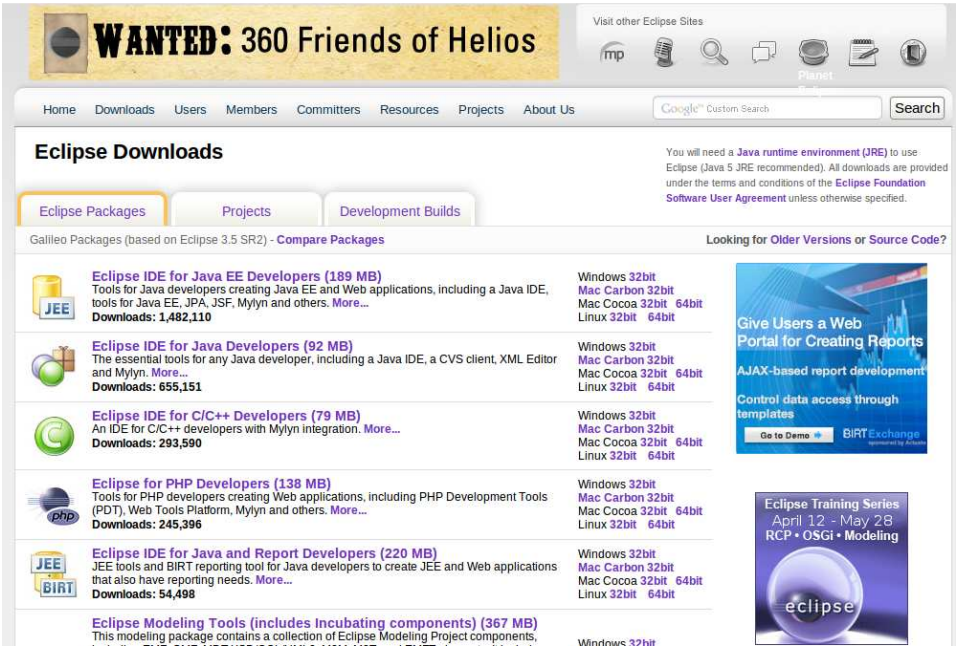


Όταν φτάσουμε στο σημείο όπου μας ζητείτε να αποδεχτούμε τους όρους άδειας χρήσης (License) του προγράμματος πατάμε OK αρχικά και μετά Yes. Για να είναι εφικτή η επιλογή μέσα στο τερματικό μετακινηθείτε στο μενού χρησιμοποιώντας τα βελάκια του πληκτρολογίου.



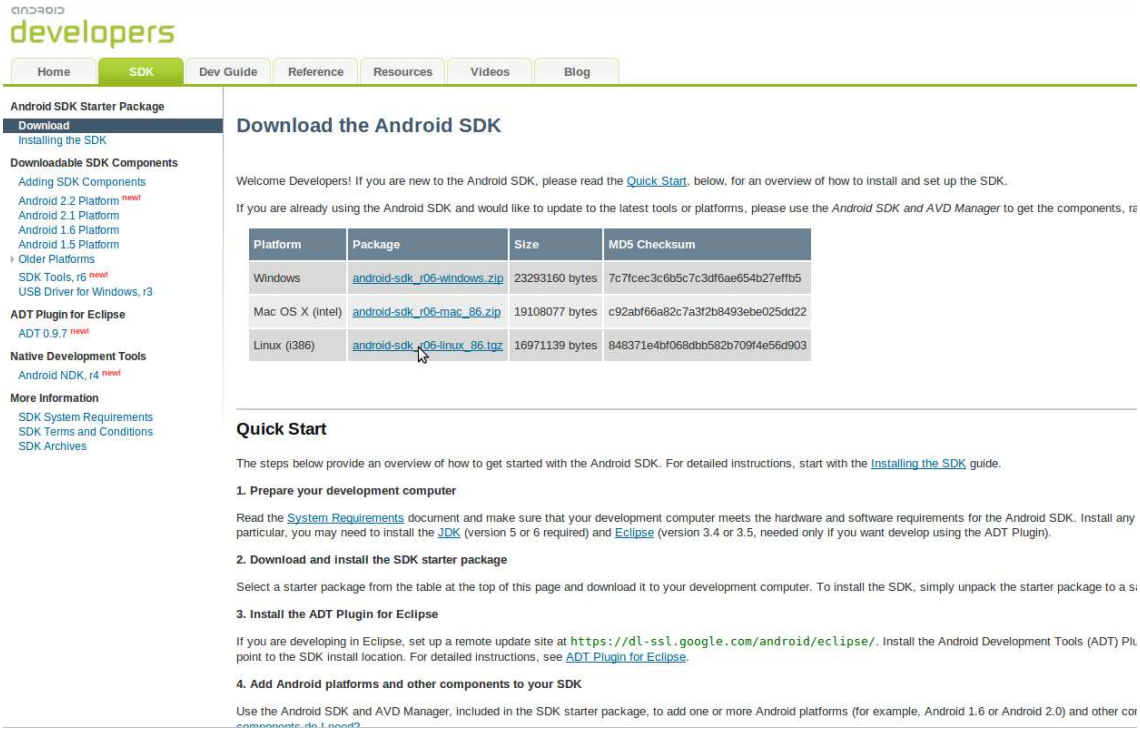


Εδώ έχουμε τελειώσει με την Java. Θα πρέπει τώρα να εγκαταστήσουμε και το Eclipse. Για να γίνει αυτό θα πρέπει να το κατεβάσουμε από την διεύθυνση και να το εγκαταστήσουμε στον υπολογιστή μας. http://www.eclipse.org/downloads/. Επιλέγουμε μια από τις εκδόσεις Java, Java EE ή RCP και την κατεβάζουμε. Αποσυμπιέζουμε το αρχείο που κατεβάσαμε σε ένα φάκελο της αρεσκείας μας και έχουμε τελειώσει με την εγκατάσταση του Eclipse.

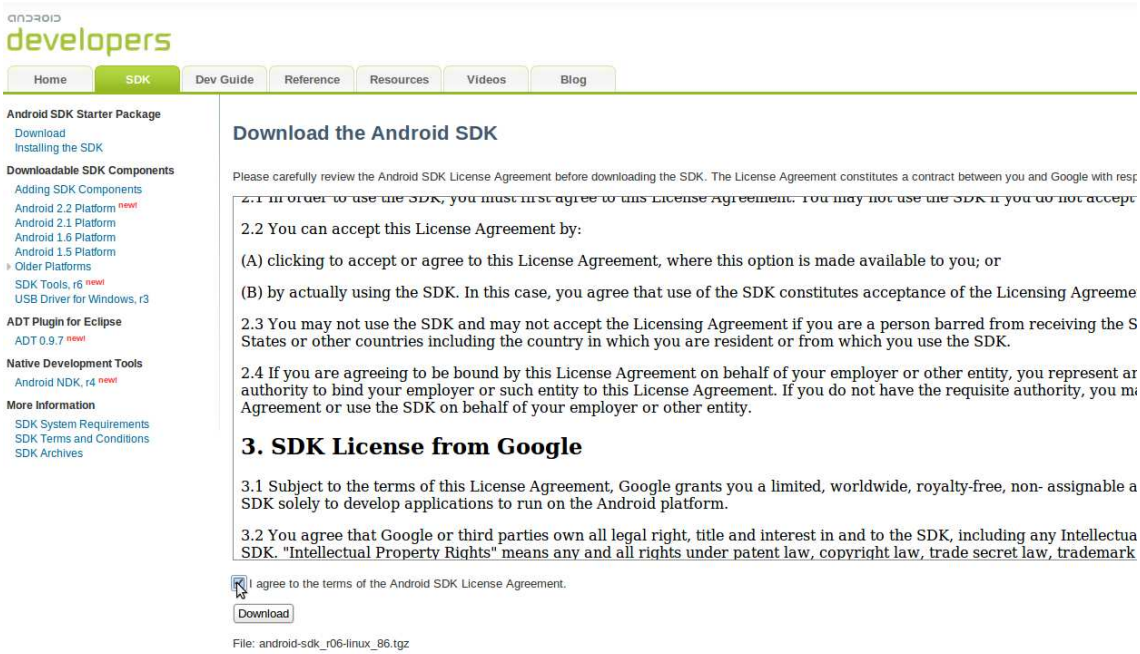


Γ.2 Εγκατάσταση Android-SDK

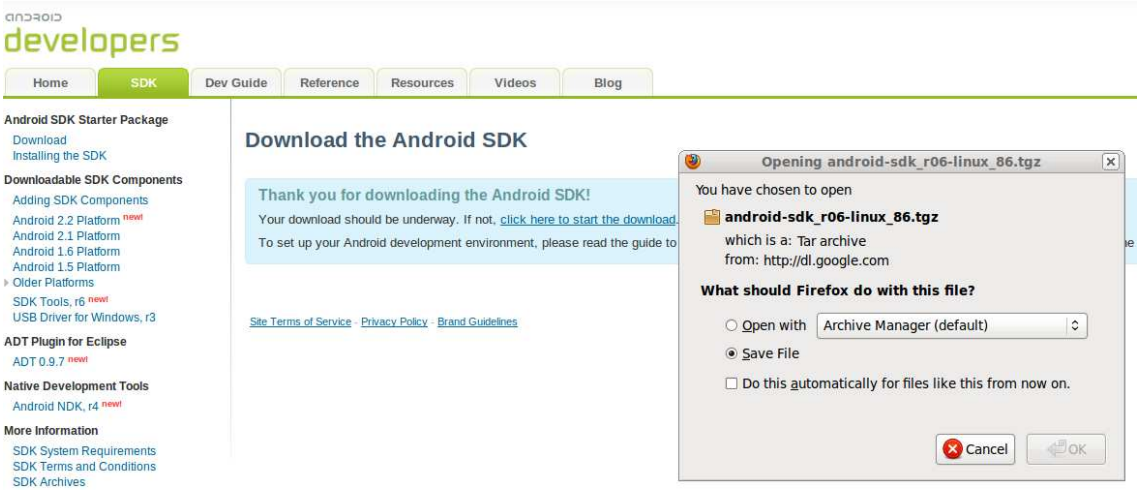
Στην συνέχεια θα πρέπει να προχωρήσουμε με την εγκατάσταση του Android SDK. Προωθηθείτε στην ιστοσελίδα http://code.google.com/android/download.html και από εκεί επιλέξτε την έκδοση για το Linux.



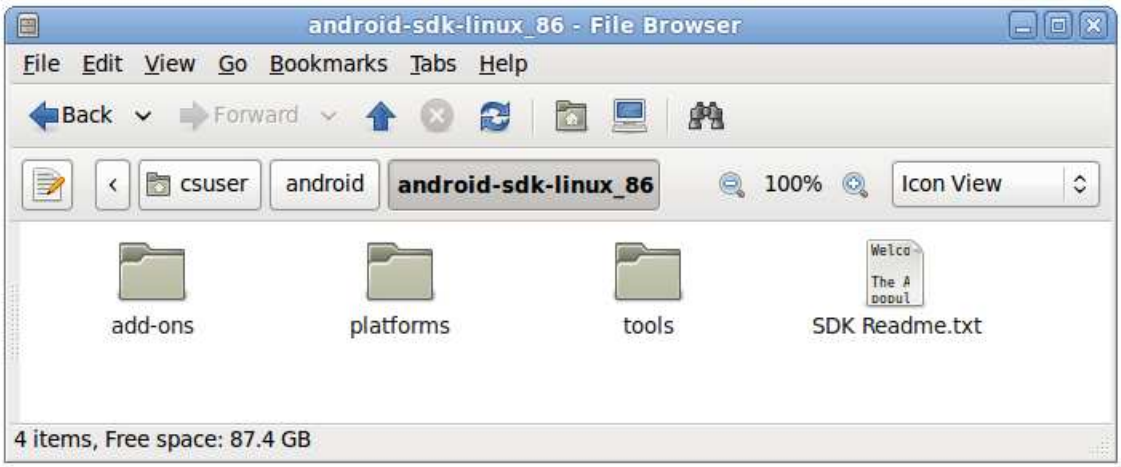
Θα εμφανιστεί μια οθόνη με τους όρους άδειας χρήσης (License) του προγράμματος τους οποίους αποδεχόμαστε πατώντας αρχικά στο κουτί όπου αναγράφετε σε αυτή την αποδοχή “I agree to the terms of the Android SDK License Agreement” και στη συνέχεια πατώντας το κουμπί download.



Τέλος κατεβάζουμε το συμπιεσμένο αρχείο στον υπολογιστή μας και το αποσυμπιέζουμε σε ένα φάκελο της αρεσκείας μας π.χ., στο </home/homedir/android/ >. Στο παράδειγμα εμείς θα το αποθηκεύσουμε στο /home/csuser/android/.



Έτσι η μορφή του φάκελου /home/csuser/android/android-sdk-linux_86/, θα είναι η εξής:



Από αυτά μας ενδιαφέρει περισσότερο είναι να έχουμε πρόσβαση στο φάκελος tools στον οποίο βρίσκεται ένα ολοκληρωμένο πακέτο εργαλείων που θα μας βοηθήσουν στη δημιουργία και τον έλεγχο των εφαρμογών που θα γράφουμε. Θα ήταν πρακτικότερο να συμπεριλάβουμε στο PATH του συστήματος μια αναφορά προς τον φάκελο αυτό, πράγμα που μπορεί να γίνει με τον εξής τρόπο:

Πληκτρολογούμε αρχικά μέσα σε ένα τερματικό (Applications > Accessories > Terminal) την εντολή

```
gedit ~/.bashrc
```

και προσθέτουμε στο αρχείο που θα ανοίξει την ακόλουθη γραμμή:

```
[…]
export PATH=${PATH}:/home/csuser/android/android-sdk-linux_86/tools/
[…]
```

Αποθηκεύουμε τις αλλαγές στο αρχείο και επιστρέφοντας στο τερματικό πληκτρολογούμε την εντολή

```
sudo gedit /etc/udev/rules.d/51-android.rules
```

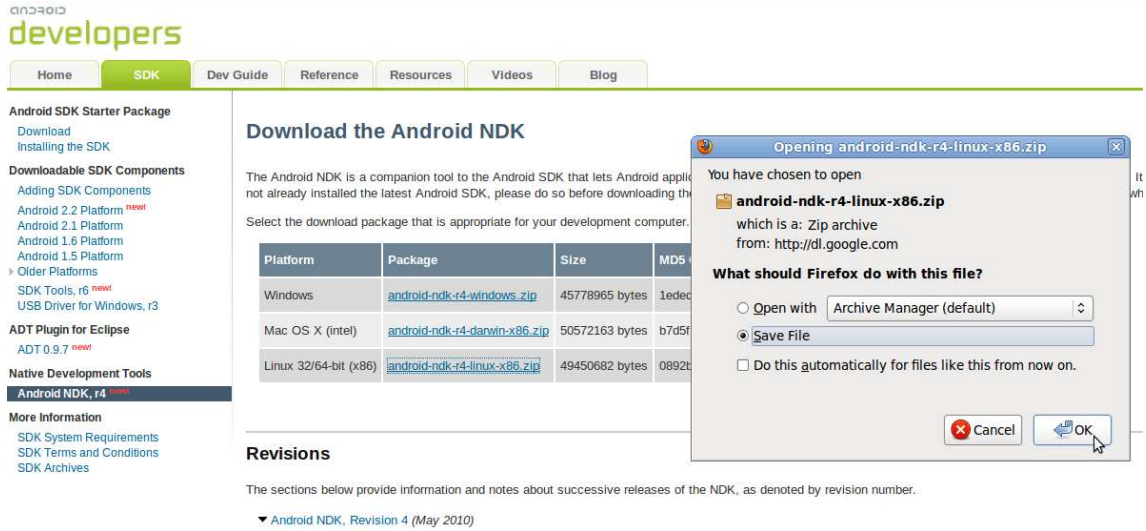
και προσθέτουμε στο αρχείο που θα ανοίξει την ακόλουθη γραμμή:

```
[…]
SUBSYSTEM=="usb", SYSFS{idVendor}=="0bb4", MODE="0666"
[…]
```

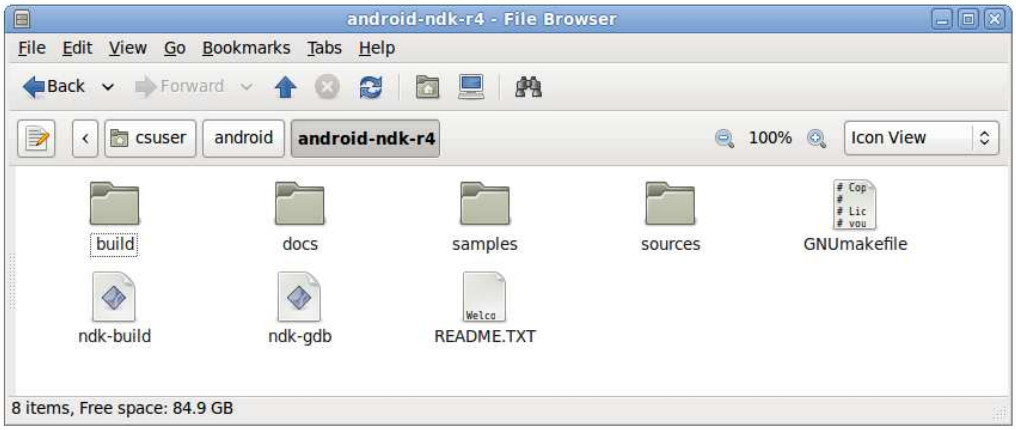
Αποθηκεύουμε και πάλι τις αλλαγές στο αρχείο και επανεκκινούμε τον υπολογιστή. Τώρα είμαστε πλέον σε θέση να καλούμε τα εργαλεία του SDK/tools από οπουδήποτε και να έχουμε πρόσβαση μέσω USB σε μια συσκευή HTC.

Γ.3. Εγκατάσταση Android-NDK

Προωθηθείτε στην ιστοσελίδα http://developer.android.com/sdk/ndk/index.html και από εκεί επιλέξτε την έκδοση για το Linux. Κατεβάζουμε το συμπιεσμένο αρχείο στον υπολογιστή μας και το αποσυμπιέζουμε σε ένα φάκελο της αρεσκείας μας π.χ., στο </home/homedir/android/>.



Με τον τρόπο αυτό έχουμε εγκαταστήσει και το Android-NDK. Η μορφή του φάκελου /home/csuser/android/android-ndk-r4/ (=$NDK) θα είναι η εξής:



Γ.4 Μεταγλώττιση Εφαρμογής

Θεωρούμε λοιπόν ότι θέλουμε να εγκαταστήσουμε και να εκτελέσουμε ένα πρόγραμμα γραμμένο σε γλώσσα C. Για χάριν παραδείγματος θεωρούμε ότι έχουμε το πρόγραμμα MyHelloAndroid.c. Το εν λόγω πρόγραμμα απλά τυπώνει στην οθόνη "`Hello Android From MyHelloAndroid.c`" και έχει την ακόλουθη δομή:

```
#include <stdlib.h>
#include <stdio.h>

void main(){

 printf("\nHello Android\nFrom MyHelloAndroid.c\n");

}
```

Θα πρέπει πρώτα να μεταγλωττίσουμε το πρόγραμμα αυτό με τη χρήση του Android-NDK. Για να το πετύχουμε κάτι τέτοιο, προωθούμαστε στον φάκελο όπου έχουμε εγκαταστήσει το ndk ($NDK) και στη συνέχεια στο φάκελο samples. Δημιουργούμε ένα νέο φάκελο στον οποίο θα δώσουμε το όνομα της εφαρμογής π.χ., MyHelloAndroid. Μέσα σε αυτόν δημιουργούμε ένα xml αρχείο AndroidManifest.xml βάση αυτών που αναφέρονται στην ιστοσελίδα του NDK. Στο παράδειγμα αυτό θα έχει την εξής μορφή:

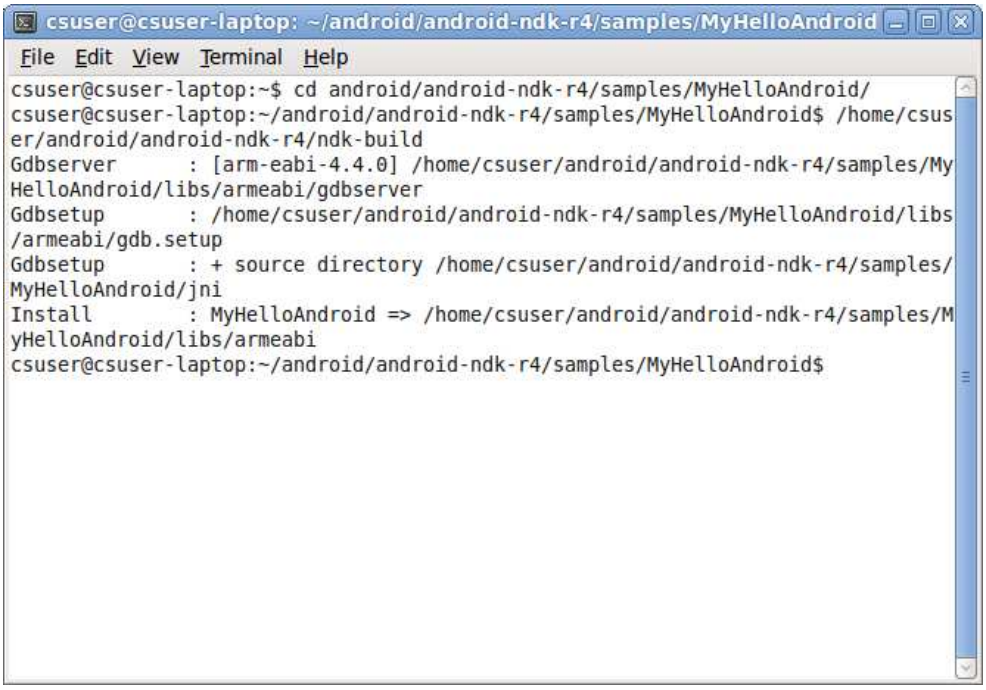
```
<?xml version="1.0" encoding="utf-8" ?>
       - <manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.example.myhello" android:versionCode="1" android:versionName="1.0">
         <uses-sdk android:minSdkVersion="3" />
       - <application android:label="@string/app_name" android:debuggable="true">
          - <activity android:name=".HelloJni" android:label="@string/app_name">
              - <intent-filter>
             <action android:name="android.intent.action.MAIN" />
              <category android:name="android.intent.category.LAUNCHER" />
                 </intent-filter>
            </activity>
     </application>
  </manifest>
```

Στην συνέχεια θα πρέπει να καθορίσουμε τον κώδικα που θέλουμε να μεταγλωττίσουμε. Αυτό πρέπει να γίνει μέσω ενός αρχείου android.mk το οποίο θα πρέπει να τοποθετηθεί κάτω από το φάκελο $NDK/<application name>/jni/, όπου <application name> ο φάκελο της αντίστοιχης εφαρμογής που δημιουργήσαμε στο προηγούμενο βήμα. Περισσότερες πληροφορίες για τη σύνταξη του εν λόγω αρχείο υπάρχουν στο αρχείο $NDK/docs/ANDROID-MK.TXT. Στο παράδειγμα αυτό έχει την εξής μορφή:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := MyHelloAndroid
LOCAL_SRC_FILES := MyHelloAndroid.c

include $(BUILD_EXECUTABLE)
```

Αξίζει να σημειωθεί ότι στο LOCAL_SRC_FILES μπορούμε να θέσουμε ανάφερες σε ένα ή περισσότερα αρχεία γραμμένα σε c ή c++ που συνδέονται μεταξύ τους χωρίς να ανησυχούμε πως θα γίνει η σύνδεση και η μεταγλώττιση. Η τελευταία γραμμή (include $(BUILD_EXECUTABLE)) καθορίζει ότι επιθυμούμε την παραγωγή ενός εκτελέσιμου αρχείο σε περιβάλλον Android.

Για να παράγουμε το εκτελέσιμο αυτό αρχείο, το οποίο μπορεί να εγκατασταθεί και να εκτελεστεί κατευθείαν σε περιβάλλοντα Android, θα πρέπει να μεταφερθούμε στο φάκελο της εφαρμογής που δημιουργήσαμε νωρίτερα και να εκτελέσουμε την εντολή `$SDK/ndk-build`.



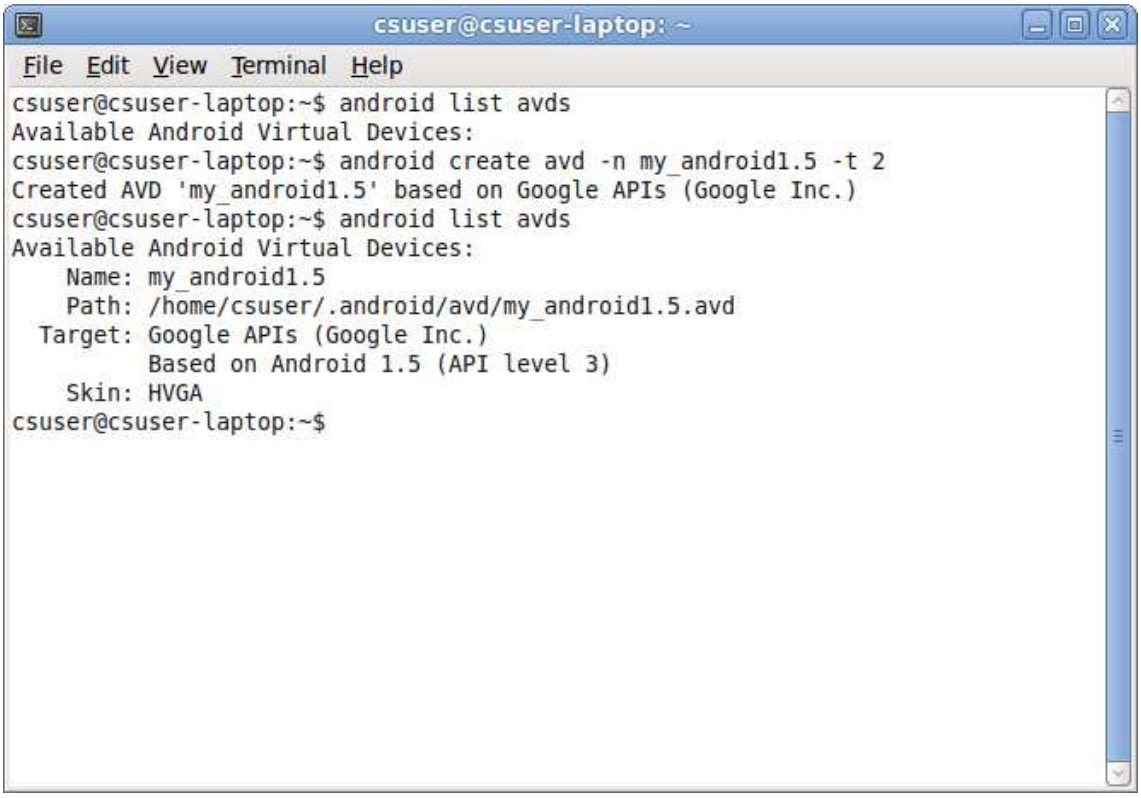
Αυτό θα έχει ως αποτέλεσμα να ξεκινήσει η μεταγλώττιση και με το τέλος της, το παραγόμενο εκτελέσιμο αρχείο θα βρίσκεται στο φάκελο $NDK/sample/<application name>/bin/ndk/local/armeabi.

Γ.5 Εκτέλεση Εφαρμογής σε Περιβάλλον Android

Αφού έχουμε το εκτελέσιμο αρχείο θα πρέπει αυτό να μεταφερθεί στη συσκευή πάνω στην οποία θέλουμε να το εκτελέσουμε. Αυτό μπορεί να γίνει είτε σε μια εικονική συσκευή που θα δημιουργήσουμε και θα προσομοιώνει την λειτουργία του περιβάλλοντος του Android είτε σε μια πραγματική συσκευή με τη χρήση των εργαλείων του Android-SDK .

Αρχικά να πούμε ότι η εντολή `android list avds`, δίνει μια λίστα με όλες τις εικονικές συσκευές τις οποίες μπορούμε να χρησιμοποιήσουμε. Όσον αφορά τη δημιουργία μιας εικονικής συσκευής πληκτρολογούμε

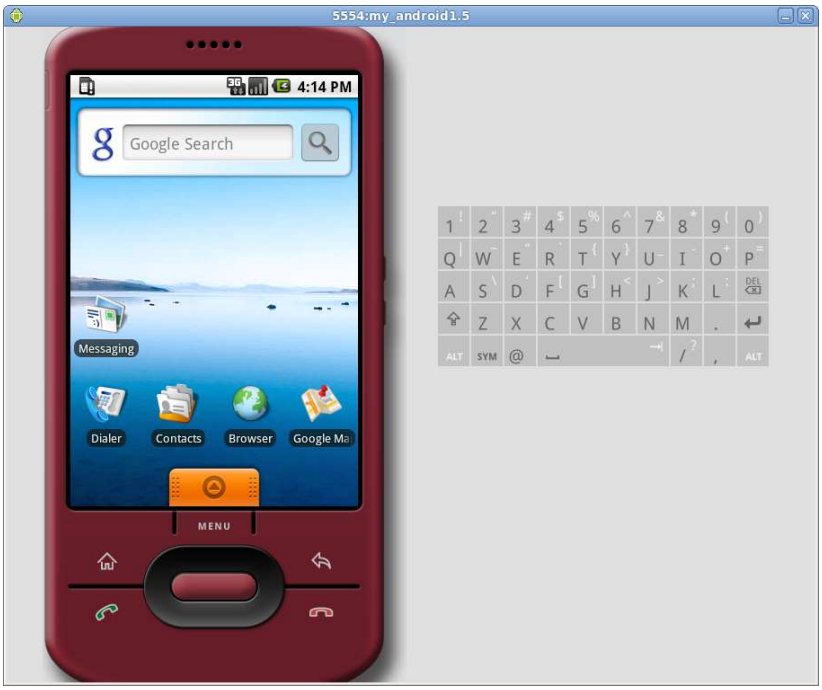
```
android create avd -n <avd_name> -t <targetID>
```



Στην συνέχεια για να μπορούμε να δουλέψουμε πάνω σε μια εικονική συσκευή θα πρέπει να την εκκινήσουμε. Αυτό γίνεται με την εντολή

```
emulator –avd <avd_name>
```

Με την εκτέλεση της εντολής αυτής ένα νέο παράθυρο εμφανίζεται. Θα χρειαστεί ένα μικρό χρονικό διάστημα για να αρχικοποιηθεί η εικονική συσκευή. Στο διάστημα αυτό η συσκευή θα εμφανίζεται σαν offline. Όταν η συσκευή θα είναι έτοιμη θα παρουσιάζει την ακόλουθη μορφή



Όσον αφορά τώρα μια πραγματική συσκευή το μόνο που έχουμε να κάνουμε είναι να συνδέουμε το USB καλώδιο σε ένα από τα διαθέσιμα USB ports. Μην επιλέξετε το USB connected από τη συσκευή γιατί θα χαθεί η επικοινωνία με την sdcard. Η εντολή `adb devices` μας δίνει όλες τις ενεργές συσκευές του συστήματος και την κατάστασή τους.

```
csuser@csuser-laptop:~$ adb devices
List of devices attached
HT99VL900078    device
emulator-5554   offline

csuser@csuser-laptop:~$ adb devices
List of devices attached
HT99VL900078    device
emulator-5554   device
```

Στην πιο πάνω εικόνα υπάρχουν 2 συσκευές η πρώτη (HT99VL900078) είναι η πραγματική συσκευή και η δεύτερη (emulator-554) η εικονική συσκευή που δημιουργήσαμε και ενεργοποιήσαμε στο προηγούμενο παράδειγμα.

Με τη χρήση της εντολή `adb -s <serial number> <command>` μπορούμε να εκτελέσουμε οποιαδήποτε εντολή υποστηρίζει το εργαλείο adb σε οποιαδήποτε ενεργή συσκευή βάζοντας απλά το αντίστοιχο serial number π.χ., emulator-554. Η `adb <command>` δηλαδή η adb χωρίς την παράμετρο –s αναφέρεται στην πραγματική συσκευή. Για τις υποστηριζόμενες εντολές κοιτάξτε το αντίστοιχου section στην επίσημη ιστοσελίδα των Android Developers. Από αυτές μας ενδιαφέρουν ιδιαίτερα η `adb pull <remote> <local>` που αφορά την μεταφορά δεδομένων από τη συσκευή στον υπολογιστή, η `adb push <local> <remote>` που αφορά την μεταφορά δεδομένων από τον υπολογιστή σ τη συσκευή και η `adb shell` που δημιουργεί ένα remote terminal όπου μπορούμε να εκτελούμε command-line εντολές στη συσκευή μας.

Για να εκτελέσουμε λοιπόν την εφαρμογή μας στη συσκευή θα πρέπει αρχικά να μεταφέρουμε το εκτελέσιμο στο data/local/ φάκελο της συσκευής μας με τη χρήση της `adb push` και στη συνέχεια μέσω του shell να το εκτελέσουμε, όπως δείχνει η ακόλουθη εικόνα.

Το ίδιο μπορεί να γίνει και με τη χρήση μιας εικονικής συσκευής και της εντολής `adb -s <serial number> <command>`. Εναλλακτικά μπορούμε επίσης να κατεβάσουμε μέσω του Android Market στη συσκευή μας την εφαρμογή Connect Bot, το οποίο

προσομοιώνει ένα terminal σε Linux και να εκτελέσουμε από εκεί την εφαρμογή μας αφού πρώτα την μεταφέρουμε στο φάκελο data/local/ με τη χρήση της `adb push`.

Παράρτημα Β

Παρουσίαση Πειραματικών Αποτελεσμάτων

Σε αυτό το παράρτημα θα παρουσιάσουμε τα αποτελέσματα που είχαμε μετά το τέλος της πειραματικής διαδικασίας. Να σημειώσουμε ότι για τον FAST και XSORT-FAST έγινε μια πληθώρα μετρήσεων που αφορούσαν την μεταβολή της μεταβλητής Q για κάθε διαφορετικό M. Στους πιο κάτω πίνακες παρουσιάζεται μόνο η καλύτερη από αυτές τις μετρήσεις. Επίσης στους XSORT αλγόριθμους η φάση 1 είναι η φάση της παραγωγής των αρχικών runs και του υπολογισμού του πεδίου ορισμού του ιστογράμματος ενώ η φάση 2 αποτελεί την φάση της παραγωγής του ιστογράμματος καθώς επίσης και την τελική φάση της δημιουργίας του τελικού αποτελέσματος.

USB KEY – ATMOSPHERIC DATASET

M	Lm	Q	Phase1 (s)	Phase2(s)	Total (s)	reads	writes	Passes
External-Merge-Sort (EXMS)								
5	20480	N/A	159.1175	400.41975	559.538	429496	429496	8
10	40960	N/A	83.31425	183.482	266.796	268435	268435	5
25	102400	N/A	88.191	105.80675	193.998	214748	214748	4
125	512000	N/A	57.8425	90.15575	147.999	161061	161061	3
250	1024000	N/A	43.79675	102.90875	146.706	107374	107374	2
2000	8192000	N/A	39.72475	30.304	70.0298	107374	107374	2
12500	51200000	N/A	41.67575	24.59875	66.2748	107374	107374	2
25000	102400000	N/A	43.85075	23.4995	67.3513	107374	107374	2
75000	307200000	N/A	44.20025	0	44.2025	53687	53687	1
Replacement-Selection-Sort (RSEL)								
5	20480	N/A	191.07075	359.267	550.348	436558	436558	8
10	40960	N/A	158.21425	171.3355	329.55	270522	270522	5
25	102400	N/A	133.78975	97.158	230.948	215366	215366	4

125	512000	N/A	88.1015	115.9395	204.042	161166	161166	3
250	1024000	N/A	64.3215	61.4815	125.804	107426	107426	2
2000	8192000	N/A	41.14775	27.33575	68.4845	107381	107381	2
12500	51200000	N/A	54.2525	25.1325	79.3855	107375	107375	2
25000	102400000	N/A	49.808	32.75225	82.5618	107375	107375	2
75000	307200000	N/A	44.94975	0	44.9518	53687	53687	1
FAST								
5	20480	80	116.39825	204.226	320.624	2342975	161061	3
10	40960	80	117.225	146.23575	263.46	1107153	161061	3
25	102400	500	102.019	84.01675	186.035	1419864	107374	2
125	512000	600	67.28275	46.65875	113.941	360369	107374	2
250	1024000	249	54.3255	51.1555	105.482	153784	107374	2
2000	8192000	3000	44.7485	45.636	90.3853	161526	107374	2
12500	51200000	13000	52.19775	53.45875	105.657	159393	107374	2
25000	102400000	55000	93.9105	0	93.9108	161061	53687	1
75000	307200000	55000	62.64275	0	62.6565	53687	53687	1
XSORT-RSEL (XS-RSEL)								
5	20480	N/A	132.828	111.285	244.1128	3316591	112897	2
10	40960	N/A	96.39625	53.12075	149.5165	1479716	109257	2
25	102400	N/A	84.33575	35.985	120.3213	620192	107965	2
125	512000	N/A	74.46775	28.67475	103.1423	257327	107480	2
250	1024000	N/A	70.602	26.65225	97.25425	212260	107428	2
2000	8192000	N/A	39.96875	27.32575	67.295	168568	107383	2
12500	51200000	N/A	46.78775	27.56475	74.35675	162885	107377	2
25000	102400000	N/A	43.14575	26.314	69.46525	162149	107377	2
75000	307200000	N/A	46.44725	0	46.4485	53687	53687	1
XSORT-FAST (XS-FAST)								

5	20480	70	118.4098	32.558	150.9673	1486902	107376	2
10	40960	100	106.1715	29.31725	135.4885	1025879	107376	2
25	102400	200	88.0665	26.48975	114.556	707619	107376	2
125	512000	150	58.9805	26.91075	85.89175	367191	107376	2
250	1024000	249	59.35075	26.6855	86.03625	257132	107376	2
2000	8192000	1999	42.558	25.2515	67.81	175623	107376	2
12500	51200000	12500	52.34625	25.788	78.135	214240	107376	2
25000	102400000	24999	57.05875	25.34425	82.4035	162869	107376	2
75000	307200000	55000	63.291	0	63.3055	53687	53687	1

USB KEY – CUSTOMER DATASET

M	Lm	Q	Phase1 (s)	Phase2(s)	Total (s)	reads	writes	Passes
External-Merge-Sort (EXMS)								
5	20480	N/A	1092.68275	1876.003	2968.7	1800000	1800000	9
10	40960	N/A	427.70575	784.0183	1211.7	1200000	1200000	6
25	102400	N/A	257.4855	369.6393	627.12	800000	800000	4
125	512000	N/A	275.344	241.2088	516.55	600000	600000	3
250	1024000	N/A	205.841	241.0035	446.84	600000	600000	3
2000	8192000	N/A	130.42925	107.7545	238.18	400000	400000	2
12500	51200000	N/A	135.948	102.137	238.09	400000	400000	2
25000	102400000	N/A	141.1375	103.9853	245.12	400000	400000	2
75000	307200000	N/A	134.26	107.2023	241.46	400000	400000	2
Replacement-Selection-Sort (RSEL)								
5	20480	N/A	645.61	1439.493	2085.1025	1864783	1820668	9
10	40960	N/A	311.33825	681.4048	992.743	1220939	1206980	6
25	102400	N/A	173.3265	326.411	499.73775	806787	802256	4
125	512000	N/A	196.23275	218.9098	415.143	601222	600406	3
250	1024000	N/A	165.65375	233.027	398.68125	600609	600205	3
2000	8192000	N/A	129.434	103.2588	232.69575	400079	400026	2
12500	51200000	N/A	135.5065	107.3125	242.81925	400015	400006	2
25000	102400000	N/A	128.39025	106.0008	234.4065	400009	400004	2

75000	307200000	N/A	134.4165	96.0875	230.50725	400003	400001	2
FAST								
5	20480	80	232.795	415.23	648.0285	9973468	600000	3
10	40960	80	205.0515	320.85	525.8985	4677924	600000	3
25	102400	500	209.5955	229	438.5955	7733200	400000	2
125	512000	800	125.3745	128.54	253.913	2003000	400000	2
250	1024000	1000	115.3855	124.25	239.6325	1360600	400000	2
2000	8192000	3000	101.3875	117.55	218.9338	606700	400000	2
12500	51200000	13000	130.6918	125.98	256.6713	595256	400000	2
25000	102400000	200010	167.8903	0	167.8908	1800000	200000	1
75000	307200000	200010	140.748	0	140.778	600000	200000	1
XSORT-RSEL (XS-RSEL)								
5	20480	N/A	N/A	N/A	1538.4282	1447011	415140	2
10	40960	N/A	N/A	N/A	635.7344	918185	406179	2
25	102400	N/A	N/A	N/A	428.672	710722	402165	2
125	512000	N/A	N/A	N/A	298.483	620702	400403	2
250	1024000	N/A	N/A	N/A	274.941	610282	400205	2
2000	8192000	N/A	N/A	N/A	236.7422	601292	400027	2
12500	51200000	N/A	N/A	N/A	246.227	600221	400007	2
25000	102400000	N/A	N/A	N/A	239.6038	600121	400005	2
75000	307200000	N/A	N/A	N/A	245.33	600050	400002	2
XSORT-FAST (XS-FAST)								
5	20480	50	N/A	N/A	387.5933	3095999	400001	2
10	40960	100	N/A	N/A	352.2928	2847999	400001	2
25	102400	115	N/A	N/A	285.8485	1441700	400001	2
125	512000	500	N/A	N/A	248.014	1409601	400001	2
250	1024000	800	N/A	N/A	237.2065	1206000	400001	2
2000	8192000	3150	N/A	N/A	224.7378	799986	400001	2
12500	51200000	12750	N/A	N/A	250.4525	791634	400001	2
25000	102400000	200010	N/A	N/A	167.8908	1800000	200000	1
75000	307200000	200010	N/A	N/A	144.0325	600000	200000	1

HTC – ATMOSPHERIC DATASET								
M	**Lm**	**Q**	**Phase1 (s)**	**Phase2(s)**	**Total (s)**	**reads**	**writes**	**Passes**
External-Merge-Sort (EXMS)								

6	98304	N/A	337.156	529.063	866.221	79794	79794	6
10	163840	N/A	259.961	378.847	638.807	66495	66495	5
15	245760	N/A	161.0394	260.618	421.6568	53196	53196	4
40	655360	N/A	181.531	265.534	447.065	39897	39897	3
100	1638400	N/A	193.084	530.572	723.657	39897	39897	3
300	4915200	N/A	195.3548	251.187	446.5428	26598	26598	2
400	6553600	N/A	201.495	203.174	404.67	26598	26598	2
1000	16384000	N/A	209.716	98.525	308.243	26598	26598	2
3125	51200000	N/A	240.662	71.723	312.398	26598	26598	2
Replacement-Selection-Sort (RSEL)								
6	98304	N/A	283.672	443.412	727.08	80847	80847	6
10	163840	N/A	226.212	337.2	563.41	66975	66975	5
15	245760	N/A	184.7352	249.4166	434.15	53480	53480	4
40	655360	N/A	232.174	256.358	488.53	39979	39979	3
100	1638400	N/A	249.157	397.031	646.19	26635	26635	2
300	4915200	N/A	257.7638	127.9502	385.71	26612	26612	2
400	6553600	N/A	260.039	106.266	366.31	26607	26607	2
1000	16384000	N/A	283.708	71.415	355.13	26600	26600	2
3125	51200000	N/A	285.647	70.83	356.49	26599	26599	2
FAST								
6	98304	10	423.015	1491.2	1914.231	158970	66495	5
10	163840	15	208.0605	1073.7	1281.807	115703	53196	4
15	245760	15	190.908	745.1	936.004	98651	53196	4
40	655360	45	215.866	521.31	737.18	70068	39897	3
100	1638400	180	268.8795	294.16	563.042	49794	26598	2
300	4915200	300	239.526	261.62	501.149	41772	26598	2
400	6553600	400	242.604	267.45	510.054	40915	26598	2
1000	16384000	1010	261.792	277.63	539.426	39908	26598	2
3125	51200000	3125	298.21	311.77	609.99	39117	26598	2
XSORT-RSEL (XS-RSEL)								
6	98304	N/A	194.2462	697.2264	891.47	680397	27447	2
10	163840	N/A	182.2202	424.6182	606.84	373667	27030	2
15	245760	N/A	177.4046	307.9218	485.33	251007	26863	2
40	655360	N/A	222.9722	188.4438	411.42	119597	26680	2
100	1638400	N/A	213.459	137.0402	350.5	73081	26636	2
300	4915200	N/A	224.8726	108.59	333.46	51883	26613	2
400	6553600	N/A	225.9924	94.3372	320.33	49168	26608	2
1000	16384000	N/A	239.427	83.8712	323.3	44353	26601	2
3125	51200000	N/A	240.8138	93.3158	334.14	41719	26600	2

XSORT-FAST (XS-FAST)								
6	98304	10	435.003	676.44	1111.445	560462	26599	2
10	163840	15	267.588	538.02	805.609	402030	26599	2
15	245760	20	263.754	469.17	732.919	321060	26599	2
40	655360	70	248.605	240.32	488.921	139169	26599	2
100	1638400	110	243.102	211.05	454.154	109532	26599	2
300	4915200	300	246.408	158.53	404.937	75889	26599	2
400	6553600	400	245.17	139.74	384.919	70872	26599	2
1000	16384000	1000	263.657	97.428	361.087	61090	26599	2
3125	51200000	3125	307.291	101.78	409.08	55519	26599	2

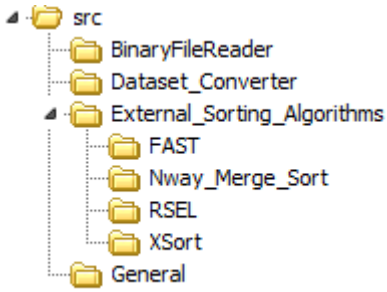
SSD – CUSTOMER DATASET								
M	**Lm**	**Q**	**Phase1 (s)**	**Phase2(s)**	**Total (s)**	**reads**	**writes**	**Passes**
External-Merge-Sort (EXMS)								
75	307200	N/A	991.409333	2849.314	3840.723667	12002000	12002000	4
125	512000	N/A	830.9095	3328.662	4159.5715	12002000	12002000	4
187.5	768000	N/A	1027.026	2900.138	3927.164333	9001500	9001500	3
250	1024000	N/A	872.503333	3140.076	4012.579667	9001500	9001500	3
12500	51200000	N/A	707.221667	2223.331	2930.553667	6001000	6001000	2
25000	102400000	N/A	717.974333	1390.061	2108.036667	6001000	6001000	2
125000	512000000	N/A	748.894333	912.7063	1661.602667	6001000	6001000	2
Replacement-Selection-Sort (RSEL)								
75	307200	N/A	928.983333	2918.917	3847.900667	12012319	12012319	4
125	512000	N/A	915.649333	2540.562	3456.212	9007557	9007557	3
187.5	768000	N/A	835.078	2866.245	3701.323	9005515	9005515	3
250	1024000	N/A	862.526333	3110.721	3973.248	9004554	9004554	3
12500	51200000	N/A	869.430333	1385.981	2255.412	6001064	6001064	2
25000	102400000	N/A	855.863333	1132.894	1988.758	6001031	6001031	2
125000	512000000	N/A	945.741333	870.626	1816.369	6001021	6001007	2
FAST								
75	307200	150	1184.647	2638.639	3823.285	26506330	9001500	3
125	512000	175	881.793	2284.91	3166.702	18590196	9001500	3
187.5	768000	190	1014.371	2129.743	3144.113	16416870	9001500	3
250	1024000	255	1022.142	2198.02	3220.162	15631079	9001500	3

12500	51200000	17500	1129.769	1415.328	2545.096	9034780	6001000	2
25000	102400000	30000	965.78	1428.671	2394.451	9013120	6001000	2
125000	512000000	125000	1156.572	1.155	1157.729	6000500	3000500	2
XSORT-RSEL (XS-RSEL)								
75	307200	N/A	941.244333	2217.419	3158.663333	9524603	6011182	2
125	512000	N/A	899.809	1703.495	2603.304333	9311951	6007002	2
187.5	768000	N/A	814.352667	1635.014	2449.367	9207932	6004993	2
250	1024000	N/A	850.31	1504.493	2354.803	9155533	6004043	2
12500	51200000	N/A	861.687667	896.5503	1758.241667	9004576	6001065	2
25000	102400000	N/A	876.743333	928.592	1805.342	9003044	6001032	2
125000	512000000	N/A	883.135333	876.0547	1759.222667	9001823	6001008	2
XSORT-FAST (XS-FAST)								
75	307200	80	914.033	1104.518	2018.552	12001981	6001001	2
125	512000	175	1105.987	913.357	2019.343	12002001	6001001	2
187.5	768000	350	990.586	919.831	1910.416	12002001	6001001	2
250	1024000	300	985.461	900.057	1885.519	12001801	6001001	2
12500	51200000	12500	952.133	848.487	1800.621	12001501	6001001	2
25000	102400000	25000	1021.183	791.388	1812.573	12001501	6001001	2
125000	512000000	125000	1198.452	757.449	1955.904	12001501	6001001	2

Παράρτημα Γ

Πηγαίος Κώδικας

Ακολουθεί ο πηγαίος κώδικας των XSORT αλγοριθμων αλλά γενικά ο πηγαίος κώδικας όλων των αλγόριθμων βρίσκεται στο CD που συνοδεύει την παρούσα διπλωματική κάτω από τον φάκελο src που έχει την εξής μορφή:



Στο φάκελο

- **BinaryFileReader:** Βρίσκεται η υλοποίηση του προγράμματος που κάνει τον έλεγχο αν το παραγόμενο αποτέλεσμα είναι ορθό.
- **Dataset_Converter:** Βρίσκεται η υλοποίηση του προγράμματος που κάνει την προεργασία στα δεδομένα των dataset δημιουργώντας τα κατάλληλα αρχεία.
- **General:** Βρίσκονται όλες οι βοηθητικές κοινές βιβλιοθήκες που γράψαμε.
- **External_Sorting_Algorithms/FAST:** Βρίσκεται η υλοποίηση που κάναμε για τον FAST
- **External_Sorting_Algorithms/Nway_Merge_Sort:** Βρίσκεται η υλοποίηση που κάναμε για τον External-Merge-Sort
- **External_Sorting_Algorithms/RSEL:** Βρίσκεται η υλοποίηση που κάναμε για τον Replacement-Selection-Sort
- **External_Sorting_Algorithms/XSORT:** Βρίσκεται η υλοποίηση που κάναμε για τον XSORT.

Παράρτημα Δ

Ψηφιακός Δίσκος