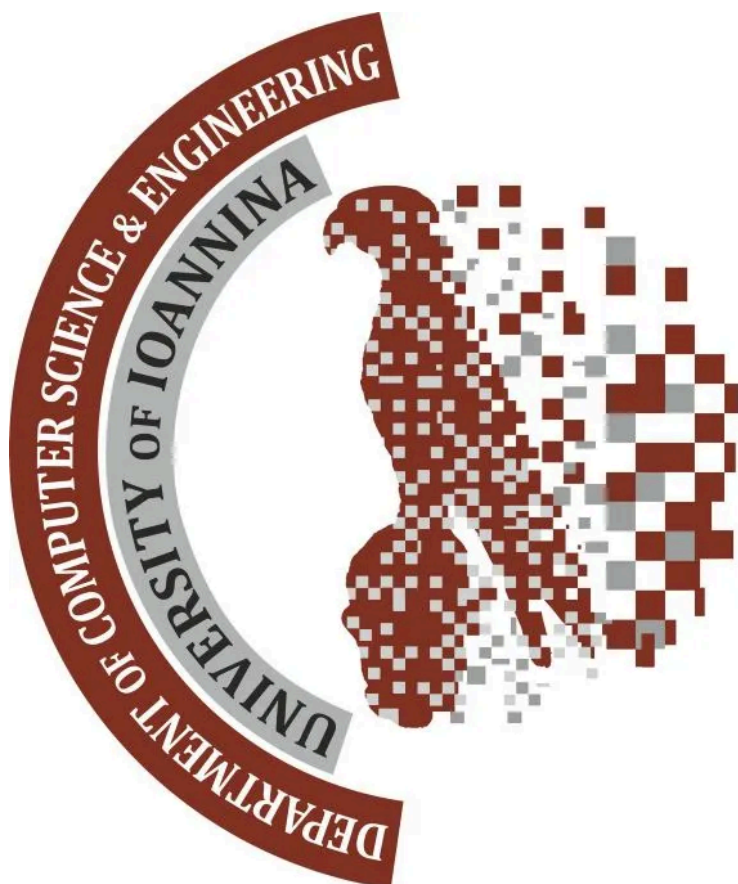


Πανεπιστήμιο Ιωαννίνων
Τμήμα Μηχανικών Η/Υ και Πληροφορικής

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ 1Η

ΥΛΟΠΟΙΗΣΗ ΠΟΛΥΝΗΜΑΤΙΚΗΣ ΛΕΙΤΟΥΡΓΙΑΣ ΣΕ ΜΗΧΑΝΗ ΑΠΟΘΗΚΕΥΣΗΣ ΔΕΔΟΜΕΝΩΝ



ΟΜΑΔΑ:

Σολδάτου Χριστίνα Ολυμπία, Α.Μ. 4001

Τσιτσιμίκλη Αικατερίνη, Α.Μ. 4821

Περιεχόμενα

❖ Εισαγωγή.....	3
❖ Εκτέλεση Προγράμματος.....	3
❖ Παραδοτέο - Εξήγηση	4
❖ Μελέτη της Μηχανής Αποθήκευσης	5
❖ 1ο βήμα: Μια καθολική κλειδαριά	7
❖ 2ο βήμα: Πολλαπλοί αναγνώστες και ένας γραφέας σε ολόκληρη τη βάση..	9
❖ <u>Ανάλυση κώδικα bench.c</u>	9
❖ <u>Ανάλυση κώδικα bench.h</u>	17
❖ <u>Ανάλυση κώδικα kiwi.c</u>	18
❖ <u>Ανάλυση κώδικα db.c</u>	23
❖ <u>Ανάλυση κώδικα db.h</u>	29
❖ Ειδοποίηση / Αφύπνιση Νήματος.....	30
❖ 3ο βήμα: Προηγμένη Λύση.....	31
❖ 4ο βήμα: Πειραματική Επαλήθευση.....	32
❖ 5ο βήμα: Στατιστικά Απόδοσης.....	33
❖ Writes:	34
❖ Reads:	37
❖ Readwrites 50%-50%:.....	40
❖ Readwrites 40%-60%:.....	43
❖ Readwrites 30%-70%:.....	46

□ Εισαγωγή

Στο πλαίσιο του μαθήματος MYY601 "Λειτουργικά Συστήματα", ανατέθηκε η πρώτη εργαστηριακή άσκηση με θέμα την υλοποίηση πολυνηματικής λειτουργίας σε μια μηχανή αποθήκευσης δεδομένων. Ο σκοπός της άσκησης είναι η ενσωμάτωση των λειτουργιών add και get σε ένα πολυνηματικό περιβάλλον, με στόχο την αποτελεσματική και ταυτόχρονη εκτέλεση εργασιών από πολλαπλά νήματα. Η μηχανή αποθήκευσης βασίζεται στην δομή δεδομένων του δέντρου log-structured merge (LSM-tree), και η υλοποίηση απαιτεί σημαντική κατανόηση της αρχιτεκτονικής της, καθώς και των μηχανισμών συγχρονισμού που προσφέρει η βιβλιοθήκη Pthreads του Linux. Ακολουθεί η αναλυτική εξέταση των τεχνικών υλοποίησης και τα αποτελέσματα της εφαρμογής των πολυνηματικών λειτουργιών στην μηχανή αποθήκευσης Kiwi.

□ Εκτέλεση Προγράμματος

Για την εκτέλεση του προγράμματος που αναπτύχθηκε στο πλαίσιο της εργασίας, αρχικά ανοίγουμε το τερματικό στον κατάλογο `~/kiwi/kiwi-source` του πηγαίου κώδικα.

Αφού ανοίξει το τερματικό, εκτελούμε την εντολή `make clean`.

```
myy601@myy601lab1:~/kiwi/kiwi-source$ make clean
```

Έπειτα, εκτελούμε την εντολή `make all`.

```
myy601@myy601lab1:~/kiwi/kiwi-source$ make all
```

Αφού ολοκληρωθεί η διαδικασία, μεταβαίνουμε στον κατάλογο `bench`, χρησιμοποιώντας την εντολή `cd bench`.

```
myy601@myy601lab1:~/kiwi/kiwi-source$ cd bench
```

Έπειτα, αναλόγως με το τι θέλουμε να κάνουμε, εκτελούμε και μία εντολή από τις παρακάτω:

Για εγγραφή δεδομένων (write):

Για την εγγραφή 100000 δεδομένων στην μηχανή αποθήκευσης, εκτελούμε την εντολή `./kiwi-bench write 100000`. Αμέσως μετά, στην επόμενη γραμμή θα εμφανιστεί το μήνυμα "Give number of threads:" για να δώσουμε το πλήθος των νημάτων που επιθυμούμε να δημιουργηθούν.

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench write 100000
Give number of threads: 
```

Εικόνα 1

Εάν επιλεγεί μόνο ένα νήμα, η διαδικασία δεν θα εκτελεστεί πολυνηματικά και επομένως θα επιστρέψει τα αποτελέσματα που θα αναμέναμε από την αρχική υλοποίηση της βάσης. Επιλέγοντας περισσότερα νήματα, τότε το πλήθος των δεδομένων που δώσαμε στην αρχή θα διαιρεθεί με το πλήθος των νημάτων, άρα τα δεδομένα θα κατανεμηθούν αναλογικά και έτσι θα ανατεθούν περισσότερες λειτουργίες σε κάθε νήμα.

Για ανάγνωση δεδομένων (read):

Αντίστοιχα, για την ανάγνωση 100000 δεδομένων, εκτελούμε την εντολή **./kiwi-bench read 100000**. Μετά θα μας ζητηθεί πάλι να καθορίσουμε τον αριθμό των νημάτων.

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench read 100000
Give number of threads: 
```

Εικόνα 2

Για εγγραφή και ανάγνωση δεδομένων (readwrite):

Εάν θέλουμε να εκτελέσουμε και τις δύο λειτουργίες, εγγραφή και αναζήτηση 100000 στοιχείων, χρησιμοποιούμε την εντολή **./kiwi-bench readwrite 100000 50 50**, όπου τα 50 και 50 αντιπροσωπεύουν τα ποσοστά επί τοις εκατό των στοιχείων που θα αναζητηθούν και θα εγγραφούν αντίστοιχα. Ακολούθως, μας ζητείται να καθορίσουμε τον αριθμό των νημάτων για την αναζήτηση και την εγγραφή. Ενδεικτικά βάζουμε 20 νήματα.

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 100000 50 50
Give number of threads for reading: 20
Give number of threads for writing: 20
```

Εικόνα 3

☐ Παραδοτέο - Εξήγηση

Στο παραδοτέο μας περιέχεται:

- η περιγραφή της άσκησης και του κώδικά μας στο Report.pdf και
- ένας φάκελος kiwi-source.zip, στον οποίο, όπως μας ζητήθηκε και από την εκφώνηση της άσκησης, περιλαμβάνονται μόνο τα αρχεία που έχουμε αλλάξει. (bench.c, bench.h, kiwi.c, db.c, db.h)

Μας ζητείται επίσης να προσδιορίσουμε το μονοπάτι στο οποίο βρίσκονται τα αρχεία αυτά στον πηγαίο κώδικα που μας δίνεται:

```
/home/myy601/kiwi/kiwi-source/bench/bench.c
/home/myy601/kiwi/kiwi-source/bench/bench.h
/home/myy601/kiwi/kiwi-source/bench/kiwi.c
/home/myy601/kiwi/kiwi-source/engine/db.c
/home/myy601/kiwi/kiwi-source/engine/db.h
```

□ Μελέτη της Μηχανής Αποθήκευσης

Η μηχανή αποθήκευσης Kiwi βασίζεται στην αρχιτεκτονική του δέντρου log-structured merge (LSM-tree), επιλογή που αποσκοπεί στην αποδοτική διαχείριση και ανάκτηση μεγάλων όγκων δεδομένων. Η Kiwi υποστηρίζει τις βασικές λειτουργίες add και get, αποθηκεύοντας και ανακτώντας ζεύγη κλειδιού-τιμής. Η δομή αυτή επιτρέπει την ταχύτατη αναζήτηση και ταξινόμηση των δεδομένων, είτε αυτά βρίσκονται στη μνήμη (memtable) είτε στον σκληρό δίσκο (sst), μέσω της οργάνωσης των αρχείων σε πολλαπλά επίπεδα.

Memtable

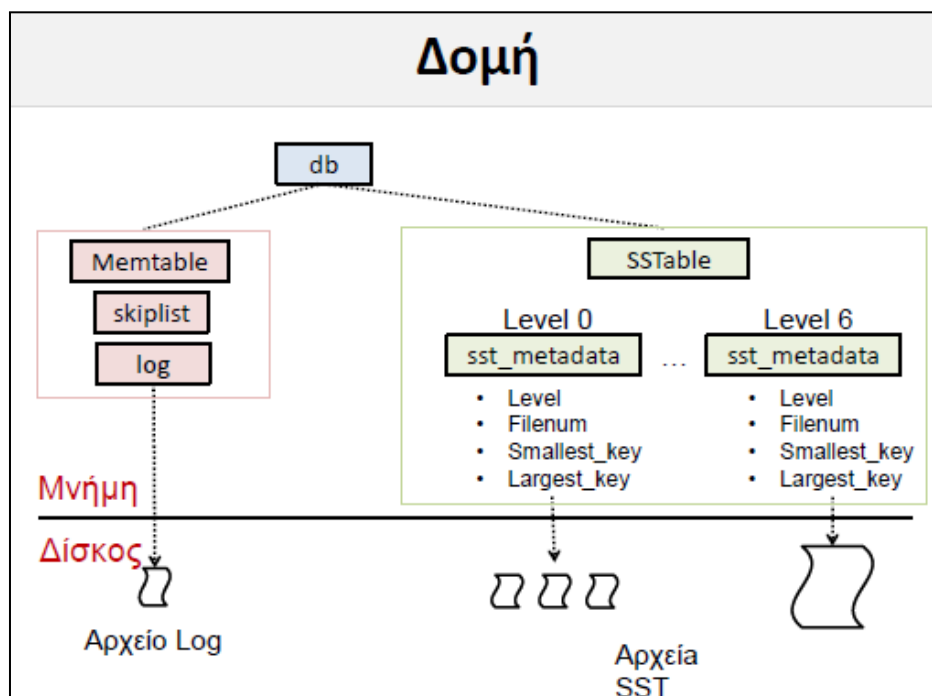
Πρόκειται για μια διαταγμένη δομή αποθήκευσης στη μνήμη, υλοποιημένη μέσω μιας skip list. Αναλαμβάνει να διαχειρίζεται τις εντολές ADD, δηλαδή την προσθήκη δεδομένων. Διαθέτει επίσης ένα αρχείο καταγραφής (log) για την προσωρινή αποθήκευση των νέων δεδομένων στο σκληρό δίσκο, επιτρέποντας την εύκολη επαναφορά τους σε περίπτωση σφάλματος. Όταν γίνεται μια αναζήτηση με την εντολή GET, το σύστημα τσεκάρει πρώτα το

Memtable για το κλειδί. Αν δεν το βρει εκεί, συνεχίζει την αναζήτηση στο Sorted String Table, ξεκινώντας από το επίπεδο 0.

Sorted String Table (SST)

Αυτή είναι μια ταξινομημένη δομή αποθήκευσης στον σκληρό δίσκο. Έχει πολλά επίπεδα, από το 0 έως το 6, με κάθε επίπεδο να περιέχει έναν αριθμό αρχείων αποθηκευμένων στον δίσκο. Τα αρχεία αυτά περιέχουν ένα εύρος κλειδιών και τις αντίστοιχες τιμές τους.

(Υλικό από την Θεωρία)



Η αποδοτικότητα της Kiwi ενισχύεται από τη δυνατότητα γρήγορης ανάκτησης δεδομένων σε περίπτωση σφάλματος, χάρη στην παράλληλη αποθήκευση δεδομένων στο memtable και στον δίσκο. Καθώς το memtable γεμίζει, τα δεδομένα μεταφέρονται στον δίσκο, απελευθερώνοντας χώρο για νέα δεδομένα. Η διαδικασία συγχώνευσης και συμπύκνωσης των αρχείων συμβάλλει στην διατήρηση της υψηλής απόδοσης της μηχανής.

Η εργασία εστιάζει στην ανάπτυξη μιας πολυνηματικής υλοποίησης για τις λειτουργίες add και get, επιδιώκοντας την παράλληλη εκτέλεση τους χωρίς παρεμβάσεις. Η διαχείριση των νημάτων πραγματοποιείται μέσω της κύριας συνάρτησης (**main**) στο αρχείο bench.c, όπου οι χρήστες μπορούν να επιλέξουν την επιθυμητή λειτουργία και να ορίσουν το πλήθος των εγγραφών ή αναγνώσεων. Η εγγραφή δεδομένων υλοποιείται μέσω της συνάρτησης `_write_test()`, η οποία ανοίγει τη βάση, προσθέτει δεδομένα στο memtable και κλείνει τη βάση δεδομένων μετά την ολοκλήρωση των εργασιών. Για την ανάγνωση, η `_read_test()` εκτελεί αντίστοιχες λειτουργίες, διασφαλίζοντας την άμεση πρόσβαση στα δεδομένα είτε από το memtable είτε από τον δίσκο.

Η προσέγγιση αυτή επιτρέπει την αποτελεσματική και συνεπή διαχείριση της αποθήκευσης και ανάκτησης δεδομένων στη μηχανή Kiwi, προσφέροντας μια σταθερή και αξιόπιστη λύση για εφαρμογές με έντονες απαιτήσεις δεδομένων.

Άσκηση

Thread-Safe Engine

- Υποστήριξη ταυτόχρονων λειτουργιών PUT και GET
- 1ο βήμα: μια καθολική κλειδαριά
- 2ο βήμα: πολλαπλοί αναγνώστες και ένας γραφέας σε ολόκληρη τη βάση

Threaded Benchmark

- Δημιουργία πολλαπλών νημάτων με χρήση παραμέτρου από τη γραμμή εντολών και δίκαιος διαμοιρασμός των εργασιών στα νήματα
- Εισαγωγή νέας λειτουργίας readwrite με την οποία εκτελούνται παράλληλα puts και gets με βάση ένα ποσοστό που δίνεται ως παράμετρο στη γραμμή εντολών. Αν πχ το ποσοστό είναι 50 τότε έχουμε 50% PUTs και 50% GETs.
- Σωστή ενημέρωση μετρήσεων απόδοσης και εκτύπωσή τους στην οθόνη

□ 1ο βήμα: Μια καθολική κλειδαριά

Για την εκπλήρωση του πρώτου βήματος της εργασίας μας, πρέπει να αναλύσουμε την τρέχουσα υλοποίηση της μηχανής αποθήκευσης δεδομένων Kiwi, εστιάζοντας στην πολυνηματική διαχείριση και συγχρονισμό των εντολών add και get. Παρακάτω παρουσιάζεται μια σύνοψη της διερεύνησης:

Περιγραφή Υλοποίησης και Συγχρονισμού

❖ Υπάρχοντα Νήματα:

Η υλοποίηση περιλαμβάνει δύο βασικά νήματα:

1. Ένα νήμα που διαχειρίζεται τις εντολές add ή get, εκτελώντας ακολουθίες λειτουργιών που εισάγουν ή ανακτούν δεδομένα.

- **Add:** Η λειτουργία add διαχειρίζεται την εισαγωγή δεδομένων στο Memtable και στο log. Αν το log του Memtable έχει γεμίσει, τότε το παλιό Memtable δε δέχεται νέες εγγραφές, μαρκάρεται για συγχώνευση, και δημιουργείται νέο Memtable που θα συνεχίζει να λαμβάνει τα νέα δεδομένα. Η συγχώνευση (merge) πραγματοποιείται ασύγχρονα από βοηθητικό νήμα. Όταν το Memtable χρειάζεται σύμπτυξη (compaction), το νήμα εκτελεί την σύμπτυξη και συγχώνευση με τα αρχεία SST στο δίσκο.
- **Get:** Η λειτουργία get αναζητά δεδομένα πρώτα στο memtable και, αν δεν βρεθούν, συνεχίζει την αναζήτηση στα αρχεία SST, ξεκινώντας από το επίπεδο 0 και συνεχίζοντας σε υψηλότερα επίπεδα ανάλογα με την ανάγκη.

2. **Μηχανής Αποθήκευσης:** Ένα νήμα που είναι υπεύθυνο για τη σύμπτυξη αρχείων στο δίσκο όταν ικανοποιούνται ορισμένες προϋποθέσεις (π.χ., όταν το memtable γεμίσει και χρειάζεται να μετακινηθεί στο δίσκο).

❖ Λειτουργίες Συγχρονισμού:

Οι λειτουργίες συγχρονισμού επιτυγχάνονται μέσω:

1. Mutexes (Αμοιβαίος Αποκλεισμός): Χρησιμοποιούνται για να διασφαλίσουν ότι μόνο ένα νήμα μπορεί να εκτελέσει κρίσιμες κομμάτια κώδικα ταυτόχρονα, προλαμβάνοντας τις αναγνωστικές ή εγγραφικές συγκρούσεις.
2. Μεταβλητές Συνθήκης: Χρησιμοποιούνται σε συνδυασμό με mutexes για να περιμένει ένα νήμα μέχρι ένα ορισμένο συμβάν να συμβεί (π.χ., το memtable να γίνει διαθέσιμο για εγγραφή).

Επισκόπηση Λειτουργίας Εγγραφής (Write)

Η διαδικασία εγγραφής στην μηχανή Kiwi ακολουθεί βασικά βήματα όπου η λειτουργία `db_add` παίζει κεντρικό ρόλο. Αυτή η λειτουργία ελέγχει αν χρειάζεται να γίνει σύμπτυξη του memtable με τα αρχεία SST (`sst_merge`) πριν εισάγει τα νέα δεδομένα. Αν το memtable δεν έχει γεμίσει, τα δεδομένα προστίθενται απευθείας μέσω της `memtable_add`. Η διαδικασία εγγραφής συνοδεύεται από τη δημιουργία ή ενημέρωση ενός αρχείου log για την ασφαλή ανάκτηση δεδομένων σε περίπτωση σφάλματος.

Επισκόπηση Λειτουργίας Ανάγνωσης (Read)

Για την ανάγνωση, η `db_get` αναλαμβάνει τον έλεγχο της ύπαρξης των αναζητούμενων δεδομένων αρχικά στο memtable μέσω της `memtable_get` και εν συνεχεία, αν δεν βρεθούν, στα αρχεία SST μέσω της `sst_get`.

Δημιουργία και Διαχείριση Νημάτων

Η απόφαση να μετακινήσουμε τη δουλειά της δημιουργίας και διαχείρισης των νημάτων στο αρχείο `bench.c` γίνεται για να μπορούμε να έχουμε καλύτερο έλεγχο στο πώς τρέχουν τα προγράμματα και να διασφαλίσουμε ότι η διεπαφή χρήστη (μέσω της `main`) είναι απλή και οργανωμένη.

- Η λειτουργία `_write_test` και `_read_test` στο `kiwi.c` παρέχουν τη βασική λογική για την εκτέλεση των εντολών `write` και `read` αντίστοιχα, ενώ η `main` στο `bench.c` διαχειρίζεται την παραμετροποίηση και την αρχικοποίηση των νημάτων βάσει των επιλογών του χρήστη.
- Για την υποστήριξη της λειτουργίας `readwrite`, τα νήματα δημιουργούνται με τέτοιο τρόπο ώστε να επιτρέπουν την ταυτόχρονη εκτέλεση εντολών `add` και `get`, διατηρώντας την αποδοτικότητα των δεδομένων.

□ 2ο βήμα: Πολλαπλοί αναγνώστες και ένας γραφέας σε ολόκληρη τη βάση

Ανάλυση κώδικα bench.c

Οι αλλαγές που κάναμε στο αρχείο είναι οι ακόλουθες:

- **Πολυνηματισμός:** Ο νέος κώδικας bench.c εισάγει πολυνηματισμό με `pthread_t`, `pthread_create`, και `pthread_join` (π.χ., γραμμές που αρχίζουν από την `create_threads` και `join_thread`).
- **Δυναμική διαχείριση μνήμης:** Ο νέος κώδικας χρησιμοποιεί `malloc` για να διαχειριστεί δυναμικά τη μνήμη των νημάτων (π.χ., γραμμές που αφορούν `pthread_t* thread_id = (pthread_t*) malloc(threads * sizeof(pthread_t));`).
- **Επικαιροποίηση λειτουργικότητας και λογικής:** Η νέα έκδοση του κώδικα επεκτείνει τις λειτουργίες που υποστηρίζει, εισάγοντας λειτουργίες ανάγνωσης/εγγραφής μέσω διαφορετικών λογικών και παραμέτρων (π.χ., υποστήριξη `readwrite`, διαχείριση ποσοστών στις γραμμές από `if (strcmp(argv[1], "readwrite") == 0) {...}`).
- **Ανάλυση εισόδων χρήστη με διαφορετικές παραμέτρους:** Ο νέος κώδικας χειρίζεται πιο περίπλοκες εισόδους από τον χρήστη, προσαρμόζοντας τις λειτουργίες ανάλογα (π.χ., διαχείριση `argc` και `argv` στην αρχή της `main`).
- **Διαχείριση δομών και αναφορά σε συναρτήσεις:** Εισαγωγή δομών για τη διαχείριση των ορισμάτων που περνούν στις λειτουργίες των νημάτων (π.χ., ορισμός και χρήση της δομής `struct orisma`).

Ακολουθεί η αναλυτική εξήγηση του κώδικα παρακάτω:

Κεντρική Ιδέα

Το πρωταρχικό έργο του bench.c είναι να εκτιμήσει την αποδοτικότητα μιας βάσης δεδομένων όταν λειτουργεί υπό συνθήκες έντονης χρήσης. Αυτό επιτυγχάνεται:

- Αναλύοντας τα ορίσματα της γραμμής εντολών για τον προσδιορισμό του τρόπου λειτουργίας και άλλων παραμέτρων.
- Δημιουργώντας δυναμικά πολλαπλά νήματα για την παράλληλη εκτέλεση των καθορισμένων λειτουργιών της βάσης δεδομένων.
- Υπολογίζοντας και αναφέροντας τις μετρικές επιδόσεων με βάση τις λειτουργίες που ολοκληρώνονται.

- ❖ Στον κώδικα μας, όπως είναι λογικό, οι συναρτήσεις λειτουργικότητας `_random_key(char *key, int length)`, `_print_header(int count)`, `_print_environment()` παραμένουν αναλλοίωτες.

- ❖ `anoigma_db(DB **db)`

Ανοίγει μια σύνδεση με τη βάση δεδομένων.

Πώς λειτουργεί: Καλεί την `db_open(DATAS)` με το προκαθορισμένο όνομα της βάσης δεδομένων (DATAS) για την αρχικοποίηση της σύνδεσης με τη βάση δεδομένων και την εκχωρεί στον παρεχόμενο δείκτη DB.

```
//anoigei th vash dedomenwn
void anoigma_db(DB **db) {
    *db = db_open(DATAS);
}
```

Εικόνα 4

- ❖ `orismata_apo_xrhsth(int argc, char** argv, long int* count)`

Αναλύει τα ορίσματα της γραμμής εντολών για να καθορίσει τη διαμόρφωση του `benchmark`, όπως τον τρόπο λειτουργίας και τον αριθμό των εισόδων.

Πώς λειτουργεί: Ελέγχει αν παρέχονται τα ελάχιστα απαιτούμενα ορίσματα και εξάγει τον αριθμό των καταχωρήσεων από τα ορίσματα. Εάν δεν παρέχονται επαρκή ορίσματα, εκτυπώνει πληροφορίες χρήσης και τερματίζει.

```
//pairnei ta orismata apo to xrhsth kai kanei elegxo
void orismata_apo_xrhsth(int argc, char** argv, long int* count) {
    if (argc < 3) {
        fprintf(stderr, "Usage: db-bench <write | read> <count> or db-bench <readwrite> <count>");
        exit(1);
    }
    *count = atoi(argv[2]);
}
```

Εικόνα 5

- ❖ `create_threads(pthread_t* thread_id, int threads, void* (*synarthsh)(void*), struct orisma* arg)`

Δημιουργεί έναν καθορισμένο αριθμό νημάτων για την εκτέλεση μιας συγκεκριμένης συνάρτησης.

Πώς λειτουργεί: Διατρέχει σε βρόχους όλα τα νήματα, χρησιμοποιώντας το `pthread_create` για να δημιουργήσει κάθε νήμα, περνώντας έναν δείκτη συνάρτησης και παραμέτρους για την εκτέλεση του νήματος.

```
//dhmiourgei nhmata
void create_threads(pthread_t* thread_id, int threads, void* (*synarthsh)(void*), struct orisma* arg) {
    for(int i = 0; i < threads; i++) {
        pthread_create(&thread_id[i], NULL, synarthsh, (void*)arg);
    }
}
```

Εικόνα 6

❖ join_thread(pthread_t* thread_id, int threads)

Περιμένει να ολοκληρώσουν τις εργασίες τους όλα τα νήματα που έχουν δημιουργηθεί.

Πώς λειτουργεί: Διατρέχει σε βρόχους όλα τα νήματα και καλεί το `pthread_join` σε κάθε ένα από αυτά, διασφαλίζοντας ότι το κύριο νήμα περιμένει να τελειώσουν όλα τα νήματα εργασίας.

```
//perimenei na oloklhrwthoun ola ta nhmata
void join_thread(pthread_t* thread_id, int threads) {

    for(int i = 0; i < threads; i++) {
        pthread_join(thread_id[i], NULL);
    }
}
```

Εικόνα 7

❖ main(int argc, char** argv)

Η συνάρτηση `main` αποτελεί τον πυρήνα για την εφαρμογή των διαδικασιών benchmarking που περιγράφονται στο αρχείο `bench.c`. Οργανώνει και διαχειρίζεται μια σειρά από ενέργειες, αρχίζοντας από την επεξεργασία των εισερχόμενων παραμέτρων, προχωρώντας στην πραγματοποίηση των βασικών λειτουργιών ανάγνωσης και εγγραφής της βάσης δεδομένων μέσω πολλαπλών παράλληλων νημάτων και καταλήγοντας στη συλλογή και παρουσίαση των τελικών αποτελεσμάτων. Αναλυτικά:

- Αρχικοποίηση μεταβλητών: Αρχικοποιεί έναν δείκτη βάσης δεδομένων `db` σε `NULL` και μια μεταβλητή `long int count` για να αποθηκεύσει τον αριθμό των λειτουργιών που πρόκειται να εκτελεστούν.

```
DB* db = NULL;
long int count;
```

Εικόνα 8

- Συλλογή arguments από χρήστη: Καλεί τη συνάρτηση `orismata_apo_xrhsth` για να αναλύσει τα ορίσματα της γραμμής εντολών που παρέχει ο χρήστης. Εξαγάγει τον αριθμό των λειτουργιών και ορίζει τον τρόπο λειτουργίας.

```
orismata_apo_xrhsth(argc, argv, &count);
```

Εικόνα 9

- Ορισμοί μεταβλητών για το χειρισμό νημάτων: Δηλώνει έναν ακέραιο αριθμό threads για την αποθήκευση του αριθμού των νημάτων και έναν δείκτη thread_id τύπου pthread_t για την αποθήκευση των αναγνωριστικών νημάτων.

```
int threads = 0; //nimata
pthread_t* thread_id;
```

Εικόνα 10

- Δομή για τη διαβίβαση επιχειρημάτων σε νήματα: Δηλώνει μια δομή arg τύπου orisma για να περνάει ορίσματα στα νήματα για λειτουργίες ανάγνωσης/εγγραφής.

```
struct orisma arg;
```

Εικόνα 11

- Αρχικοποίηση γεννήτριας τυχαίων αριθμών: Αρχικοποιεί τη γεννήτρια τυχαίων αριθμών με την τρέχουσα ώρα ως αρχικό σημείο. Με αυτόν τον τρόπο εξασφαλίζει ότι η ακολουθία των τυχαίων αριθμών θα είναι διαφορετική κάθε φορά που θα εκτελείται η εντολή, εφόσον η ώρα αλλάζει συνεχώς.

```
srand(time(NULL));
```

Εικόνα 12

- Επιλογή λειτουργίας ανάγνωσης ή εγγραφής: Ελέγχει αν ο τρόπος λειτουργίας που έχει καθοριστεί από τον χρήστη είναι είτε ανάγνωση, είτε εγγραφή.

```
//leitourgies read kai write
if (strcmp(argv[1], "write") == 0 || strcmp(argv[1], "read") == 0) {
```

Εικόνα 13

- Εισαγωγή αριθμού νήματος και κατανομή μνήμης για αναγνωριστικά νήματος: Ζητάει από το χρήστη τον αριθμό των νημάτων, τον διαβάζει στη μεταβλητή threads και κατανέμει μνήμη για τα αναγνωριστικά νημάτων ανάλογα με malloc.

```
printf("Give number of threads: ");
scanf("%d", &threads);

thread_id = (pthread_t*) malloc(threads * sizeof(pthread_t));
```

Εικόνα 14

- Εκτύπωση επικεφαλίδας και πληροφοριών περιβάλλοντος: Καλεί την `_print_header` για την εκτύπωση των ρυθμίσεων του benchmark και την `_print_environment` για την εκτύπωση του περιβάλλοντος του συστήματος.

```
_print_header(count);  
_print_environment();
```

Εικόνα 15

- Αρχικοποίηση βάσης δεδομένων: Καλεί την `ανοigma_db` για να ανοίξει τη σύνδεση βάσης δεδομένων.

```
ανοigma_db(&db);
```

Εικόνα 16

- Δημιουργία νήματος για δοκιμή εγγραφής ή ανάγνωσης: Ανάλογα με τον τρόπο λειτουργίας, δημιουργεί νήματα για την εκτέλεση δοκιμών εγγραφής ή ανάγνωσης, χρησιμοποιώντας το `create_threads`. Περιμένει να τελειώσουν όλα τα νήματα καλώντας το `join_thread` και στη συνέχεια εκτυπώνει στατιστικά στοιχεία για τις πράξεις που εκτελέστηκαν. Κλείνει τη σύνδεση με τη βάση δεδομένων χρησιμοποιώντας το `db_close`.

```
if (strcmp(argv[1], "write") == 0) {  
    create_threads(thread_id, threads, _write_test, &arg);  
    join_thread(thread_id, threads);  
  
    db_close(db);  
  
    print_statistics_write(count); //ektypwsh statistikwn leitourgias write  
}  
else if (strcmp(argv[1], "read") == 0) {  
    create_threads(thread_id, threads, _read_test, &arg);  
    join_thread(thread_id, threads);  
  
    db_close(db);  
  
    print_statistics_read(count); //ektypwsh statistikwn leitourgias read  
}  
}
```

Εικόνα 17

- Επιλογή λειτουργίας ανάγνωσης και εγγραφής: Ελέγχει αν ο τρόπος λειτουργίας που έχει καθοριστεί από τον χρήστη είναι μικτός, δηλαδή και ανάγνωση και εγγραφή.

```
//leitourgia readwrite  
else if (strcmp(argv[1], "readwrite") == 0) {  
    |
```

Εικόνα 18

- Αρχικοποίηση μεταβλητών: Αρχικοποιεί μια ακέραια μεταβλητή `r` στο 0, η οποία χρησιμοποιείται σαν σημαία για τυχαία δημιουργία κλειδιού. Έπειτα μετατρέπει το τέταρτο όρισμα της γραμμής εντολών (`argv[3]`) σε ακέραιο και το αναθέτει στην `pososto_read` και παρόμοια με την προηγούμενη γραμμή, μετατρέπει το πέμπτο όρισμα της γραμμής εντολών (`argv[4]`) σε ακέραιο αριθμό και το αναθέτει στην `pososto_write`. Οι δύο αυτές μεταβλητές αντιπροσωπεύουν το ποσοστό των δεδομένων προς ανάγνωση και εγγραφή αντίστοιχα.

```
int r = 0;
int pososto_read = atoi(argv[3]);
int pososto_write = atoi(argv[4]);
```

Εικόνα 19

- Εισαγωγή αριθμού νήματος και κατανομή μνήμης για αναγνωριστικά νήματος: Αρχικοποιεί ακέραιες μεταβλητές για κάθε νήμα που αντιστοιχεί σε λειτουργία ανάγνωσης και εγγραφής αντίστοιχα και ζητάει από το χρήστη τον αριθμό των νημάτων για κάθε μία λειτουργία ξεχωριστά. Έπειτα τον διαβάζει στη μεταβλητή `threads_read` και `threads_write` αντίστοιχα. Καλεί την `_print_header` και την `_print_environment`. Δηλώνει δείκτες `read_thread_id` και `wite_thread_id` τύπου `pthread_t` για την αποθήκευση των αναγνωριστικών νημάτων για τα νήματα ανάγνωσης και εγγραφής, αντίστοιχα. Ελέγχει αν ο αριθμός των ορισμάτων της γραμμής εντολών (`argc`) είναι ίσος με 4. Αν ναι, θέτει το `r` σε 1. Καλεί τη συνάρτηση `anoigma_db` για να ανοίξει τη σύνδεση βάσης δεδομένων.

```
int threads_read=0; //nimata gia reading
printf("Give number of threads for reading: ");
scanf("%d", &threads_read);

int threads_write=0; //nimata gia writing
printf("Give number of threads for writing: ");
scanf("%d", &threads_write);

_print_header(count);
_print_environment();

pthread_t *read_thread_id, *write_thread_id;

if (argc == 4)
|   r = 1;

anoigma_db(&db);
```

Εικόνα 20

- Δομή για τη διαβίβαση επιχειρημάτων σε νήματα: Οι επόμενες γραμμές δηλώνουν και αρχικοποιούν τις δομές struct orisma read_arg και struct orisma write_arg, οι οποίες χρησιμοποιούνται για να περάσουν τα ορίσματα στις δοκιμαστικές συναρτήσεις ανάγνωσης και εγγραφής. Αυτές οι δομές αρχικοποιούνται με r και db. Η μνήμη για την αποθήκευση των αναγνωριστικών νημάτων κατανέμεται δυναμικά με βάση τον αριθμό των νημάτων ανάγνωσης και εγγραφής με malloc. Τα ορίσματα για τις λειτουργίες ανάγνωσης και εγγραφής (read_arg.count και write_arg.count) υπολογίζονται με το count πολλαπλασιασμένο με τα καθορισμένα ποσοστά (pososto_read, pososto_write) και τη διαίρεση με τον αριθμό των νημάτων.

```
//dhmiourgia struct gia orismata tou read_test kai tou write_test
struct orisma read_arg;
read_arg.r = r;
read_arg.db = db;
struct orisma write_arg;
write_arg.r = r;
write_arg.db = db;

read_thread_id = (pthread_t*) malloc(threads_read * sizeof(pthread_t));
write_thread_id = (pthread_t*) malloc(threads_write * sizeof(pthread_t));

//arxikopoihsh orismatwn gia leitourgies read-write
read_arg.count = (count * pososto_read) / (100 * threads_read);
write_arg.count = (count * pososto_write) / (100 * threads_write);
```

Εικόνα 21

- Δημιουργία νήματος για δοκιμή εγγραφής και ανάγνωσης: Το επόμενο μέρος δημιουργεί τα νήματα για τις λειτουργίες ανάγνωσης και εγγραφής, χρησιμοποιώντας τη συνάρτηση create_threads. Στη συνέχεια χρησιμοποιείται η join_thread, διασφαλίζοντας ότι το κύριο νήμα περιμένει να ολοκληρωθούν όλες οι λειτουργίες ανάγνωσης και εγγραφής. Η μνήμη για τα αναγνωριστικά νημάτων απελευθερώνεται.

```
//dhmiourgia threads gia read-write
create_threads(read_thread_id, threads_read, _read_test, &read_arg);
create_threads(write_thread_id, threads_write, _write_test, &write_arg);

//wait mexri ola ta nhmata read-write na oloklhrwsoun ta tasks tous
join_thread(read_thread_id, threads_read);
join_thread(write_thread_id, threads_write);

free(read_thread_id);
free(write_thread_id);
```

Εικόνα 22

- Υπολογισμός και εκτύπωση στατιστικών: Υπολογίζει το συνολικό αριθμό των στοιχείων προς ανάγνωση και εγγραφή πολλαπλασιάζοντας το συνολικό αριθμό στοιχείων (count) με το ποσοστό που καθορίζεται για την ανάγνωση (pososto_read) και την εγγραφή (pososto_write) και στη συνέχεια διαιρεί το καθένα με το 100 για να μετατρέψει το ποσοστό σε πραγματικό αριθμό στοιχείων. Έπειτα, η σύνδεση με τη βάση δεδομένων κλείνει με την εντολή db_close(db). Τέλος, καλούνται οι συναρτήσεις για την εκτύπωση στατιστικών στοιχείων για τις λειτουργίες ανάγνωσης και εγγραφής με τα υπολογισμένα σύνολα.

```
//ypologismos statistikwn read kai write
long int synoliko_count_read = count * pososto_read / 100;
long int synoliko_count_write = count * pososto_write / 100;

db_close(db);

//ektypwsh statistikwn leitourgias read kai write
print_statistics_read(synoliko_count_read);
print_statistics_write(synoliko_count_write);
```

Εικόνα 23

- Χειρισμός σφάλματος άκυρης λειτουργίας: Εάν καθοριστεί ένας μη έγκυρος τρόπος λειτουργίας, εκτυπώνει ένα μήνυμα σφάλματος και τερματίζει το πρόγραμμα.

```
}
else {
    fprintf(stderr, "Invalid operation specified. Usage: db-bench <write | read> <count>");
    exit(1);
}
```

Εικόνα 24

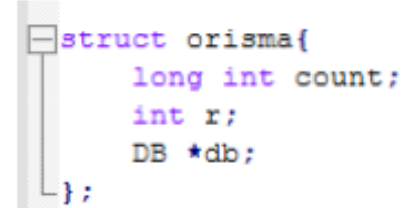
Ανάλυση κώδικα bench.h

Προστέθηκαν στο νέο bench.h

1. Δήλωση της δομής struct orisma:

Προστέθηκε μια δομή με το όνομα orisma, η οποία περιέχει τα εξής πεδία:

- `long int count`; Αντιπροσωπεύει τον αριθμό των λειτουργιών που θα εκτελεστούν στη δοκιμή.
- `int r`; Σημαία για να υποδείξει την τυχαία δημιουργία κλειδιού.
- `DB *db`; Ένας δείκτης σε ένα αντικείμενο της βάσης δεδομένων. Δηλώνει ότι οι λειτουργίες δοκιμών θα εκτελούνται σε μια συγκεκριμένη οντότητα βάσης δεδομένων.



```
struct orisma{  
    long int count;  
    int r;  
    DB *db;  
};
```

2. Δήλωση των συναρτήσεων print_statistics_write και print_statistics_read:

Αυτές οι συναρτήσεις προορίζονται για την εκτύπωση των αποτελεσμάτων των δοκιμών εγγραφής και ανάγνωσης αντίστοιχα.

```
void print_statistics_write(long int count);  
void print_statistics_read(long int count);
```

Εικόνα 26

Ανάλυση κώδικα kiwi.c

Οι αλλαγές που κάναμε στο αρχείο είναι οι ακόλουθες:

- **Πολυνηματισμός και Mutex:** Ο νέος κώδικας εισάγει mutex για συγχρονισμό σε αντίθεση με τον δεύτερο.
- **Κλήση συναρτήσεων:** Ο νέος κώδικας χρησιμοποιεί δομές για τις παραμέτρους στις συναρτήσεις `_write_test` και `_read_test`.
- **Μεταβλητή found:** Στον νέο κώδικα, το `found` είναι καθολική μεταβλητή.

Ακολουθεί η αναλυτική εξήγηση του κώδικα παρακάτω:

Κεντρική Ιδέα

Το `kiwi.c` έχει σχεδιαστεί ειδικά για τη συγκριτική αξιολόγηση της απόδοσης της βάσης δεδομένων με την ταυτόχρονη εκτέλεση πολλαπλών λειτουργιών ανάγνωσης και εγγραφής, προσομοιώνοντας πρότυπα χρήσης στον πραγματικό κόσμο, όπου πολλοί χρήστες αλληλεπιδρούν με τη βάση δεδομένων ταυτόχρονα. Αναλυτικά:

❖ Αρχικοποίηση Mutex

Κάθε **mutex** προστατεύει ένα συγκεκριμένο κομμάτι των κοινών δεδομένων.

- **write_cost_mutex** είναι για το συγχρονισμό των ενημερώσεων του συνολικού χρόνου που δαπανάται για τις λειτουργίες εγγραφής (`total_write_cost`),
- **read_cost_mutex** για το συνολικό χρόνο για τις λειτουργίες ανάγνωσης (`total_read_cost`)
- **found_mutex** για τον αριθμό των επιτυχώς εντοπισμένων καταχωρήσεων στις λειτουργίες ανάγνωσης (`found`).

Πώς λειτουργούν: Ένα νήμα, πριν ενημερώσει την τιμή αυτών των κοινών μεταβλητών, κλειδώνει το αντίστοιχο **mutex**. Αυτό το κλείδωμα διασφαλίζει ότι κανένα άλλο νήμα δεν μπορεί να έχει ταυτόχρονη πρόσβαση στη μεταβλητή, αποτρέποντας τις συγκρούσεις δεδομένων. Μετά την ενημέρωση της τιμής, το νήμα απελευθερώνει (ξεκλειδώνει) το mutex, επιτρέποντας σε άλλους να έχουν πρόσβαση στη μεταβλητή.

```
//arikopoihsh mutex gia ton ypologismo tou kostous write kai read kai gia to found
pthread_mutex_t write_cost_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t read_cost_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t found_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Εικόνα 27

- ❖ Ορισμός κοινόχρηστων μεταβλητών: Αυτές οι μεταβλητές αποθηκεύουν το συνολικό κόστος (χρόνος σε sec) των εργασιών ανάγνωσης και εγγραφής, καθώς και το πλήθος των επιτυχώς εντοπισμένων κλειδιών από τις δοκιμές ανάγνωσης.

```
//synoliko kostos read kai write
double total_read_cost = 0.0;
double total_write_cost = 0.0;
int found = 0; //metavlhth gia thn katametrhsh tw'n keys pou exoun vrethei
```

Εικόνα 28

- ❖ print_statistics_write(long int count)

Εκτυπώνει τα στατιστικά στοιχεία που σχετίζονται με τις λειτουργίες εγγραφής. Λαμβάνει μία μόνο παράμετρο, την `count`, η οποία είναι τύπου `long int` και αντιπροσωπεύει το συνολικό αριθμό των λειτουργιών εγγραφής που εκτελέστηκαν. Στη συνέχεια, εκτυπώνει την απόκριση, τη ρυθμαπόδοση και το κόστος (χρόνο).

Πώς λειτουργεί: Διαμορφώνει και εκτυπώνει τις υπολογισμένες μετρήσεις με βάση το `total_write_cost` και τον αριθμό των λειτουργιών εγγραφής που εκτελέστηκαν.

```
//ektypwsh statistikwn gia to write
void print_statistics_write(long int count){

    printf(LINE);
    printf("|Random-Write    (done:%ld): %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec);\n",
        count, (double)(total_write_cost / count)
        , (double)(count / total_write_cost)
        , total_write_cost);

    return;
}
```

Εικόνα 29

- ❖ print_statistics_read(long int count)

Εκτυπώνει τα στατιστικά στοιχεία για τις λειτουργίες ανάγνωσης, παρόμοια με την print_statistics_write, δηλαδή την απόκριση, τη ρυθμαπόδοση και το κόστος (χρόνο) αλλά περιλαμβάνει επίσης τον αριθμό των επιτυχημένων εύρεσεων κλειδιών (**found**).

Πώς λειτουργεί: Παρόμοια με την print_statistics_write, μορφοποιεί και εκτυπώνει τις μετρήσεις των λειτουργιών ανάγνωσης, χρησιμοποιώντας το `total_read_cost`, τον αριθμό των λειτουργιών ανάγνωσης και τον αριθμό των ευρεθέντων κλειδιών.

```
//ektypwsh statistikwn gia to read
void print_statistics_read(long int count){

    printf(LINE);
    printf("|Random-Read      (done:%ld, found:%d): %.6f sec/op; %.1f reads /sec(estimated); cost:%.3f(sec)\n",
        count, found,
        (double)(total_read_cost / count),
        (double)(count / total_read_cost),
        total_read_cost);

    return;
}
```

Εικόνα 30

❖ metrhsh_start()

Καταγράφει την ώρα έναρξης μιας λειτουργίας συγκριτικής αξιολόγησης.

Πώς λειτουργεί: Χρησιμοποιεί την `get_ustime_sec()`, μια συνάρτηση που επιστρέφει την τρέχουσα ώρα σε μικροδευτερόλεπτα για να καταγράψει την ώρα έναρξης.

```
//ypologizei to start
long long metrhsh_start() {
    return get_ustime_sec();
}
```

Εικόνα 31

❖ ypologismos_koustos(long long start)

Υπολογίζει τη διάρκεια μιας λειτουργίας συγκριτικής αξιολόγησης, δηλαδή το κόστος.

Πώς λειτουργεί:

- Λαμβάνει ως είσοδο έναν χρόνο έναρξης (start),
- Καταγράφει τον τρέχοντα χρόνο (και πάλι σε μικροδευτερόλεπτα),
- Υπολογίζει τη διαφορά και επιστρέφει τον χρόνο που παρήλθε σε δευτερόλεπτα

```
//ypologizei to kostos(cost)
double ypologismos_koustos(long long start) {
    long long end = get_ustime_sec();
    return (double)(end - start);
}
```

Εικόνα 32

❖ update_total_cost(pthread_mutex_t *mutex, double *total_cost, double cost) {

Ενημερώνει με ασφάλεια το συνολικό κόστος (χρόνο) για ένα σύνολο λειτουργιών (είτε ανάγνωσης είτε εγγραφής) σε όλα τα νήματα.

Πώς λειτουργεί: Χρησιμοποιεί ένα **mutex** για να εξασφαλίσει ασφαλείς για τα νήματα ενημερώσεις της κοινής μεταβλητής `total_cost`. Αυτό αποτρέπει συνθήκες ανταγωνισμού όταν πολλά νήματα προσπαθούν να ενημερώσουν το κόστος ταυτόχρονα.

```
//ypologismos synolikou koustos (total_cost)
void update_total_cost(pthread_mutex_t *mutex, double *total_cost, double cost) {
    pthread_mutex_lock(mutex);
    *total_cost += cost;
    pthread_mutex_unlock(mutex);
}
```

Εικόνα 33

❖ write_test(struct parametroi *write_pars)

Εκτελεί μια σειρά από λειτουργίες εγγραφής στη βάση δεδομένων και μετρά το χρόνο που απαιτείται για την ολοκλήρωση αυτών των λειτουργιών.

Πώς λειτουργεί:

- Δημιουργεί μια σειρά από κλειδιά (και αντίστοιχες τιμές) είτε τυχαία (αν έχει οριστεί το r) είτε διαδοχικά.
- Εισάγει αυτά τα ζεύγη κλειδιών-τιμών στη βάση δεδομένων χρησιμοποιώντας την **db_add()**.
- Μετρά το συνολικό χρόνο που απαιτείται για τις λειτουργίες εγγραφής και ενημερώνει το συνολικό συνολικό κόστος εγγραφής (total_write_cost) χρησιμοποιώντας τη συνάρτηση update_total_cost.

```
//dokimh gia to write
void *_write_test(struct orisma *write_arg){

    int i;
    double cost;
    long long start = metrhsh_start();
    Variant sk, sv;

    long int count= write_arg->count;
    int r = write_arg->r;
    DB* db = write_arg->db;

    char key[KSIZE + 1];
    char val[VSIZE + 1];
    char sbuf[1024];

    memset(key, 0, KSIZE + 1);
    memset(val, 0, VSIZE + 1);
    memset(sbuf, 0, 1024);
```

```
for (i = 0; i < count; i++) {
    if (r)
        _random_key(key, KSIZE);
    else
        snprintf(key, KSIZE, "key-%d", i);
    fprintf(stderr, "%d adding %s\n", i, key);
    snprintf(val, VSIZE, "val-%d", i);

    sk.length = KSIZE;
    sk.mem = key;
    sv.length = VSIZE;
    sv.mem = val;

    db_add(db, &sk, &sv);
    if ((i % 10000) == 0) {
        fprintf(stderr, "random write finished %d ops%30s\r",
                i,
                "");
        fflush(stderr);
    }
}

cost = ypologismos_koustous(start);
update_total_cost(&write_cost_mutex, &total_write_cost, cost);

return NULL;
```

Εικόνα 34

❖ read_test(struct parametroi *read_pars)

Εκτελεί μια σειρά από λειτουργίες ανάγνωσης από τη βάση δεδομένων και μετρά το χρόνο που απαιτείται για την ολοκλήρωση αυτών των λειτουργιών, παρακολουθώντας επίσης τον αριθμό των επιτυχημένων ευρημάτων.

Πώς λειτουργεί:

- Δημιουργεί μια σειρά από κλειδιά είτε τυχαία είτε διαδοχικά.
- Προσπαθεί να ανακτήσει την τιμή κάθε κλειδιού από τη βάση δεδομένων με την **db_get()**.
- Αυξάνει έναν global μετρητή **found** για κάθε επιτυχημένη εύρεση.
- Μετράει το συνολικό χρόνο που χρειάστηκε για τις λειτουργίες ανάγνωσης και ενημερώνει το **total_read_cost**.

```
//dokimh gia to read
void *_read_test(struct orisma *read_arg){

    int i;
    int ret;
    double cost;
    long long start = metrhsh_start();
    Variant sk;
    Variant sv;
    char key[KSIZE + 1];

    long int count= read_arg->count;
    int r = read_arg->r;
    DB* db= read_arg->db;

    for (i = 0; i < count; i++) {
        memset(key, 0, KSIZE + 1);

        if (r)
            _random_key(key, KSIZE);
        else
            snprintf(key, KSIZE, "key-%d", i);
        fprintf(stderr, "%d searching %s\n", i, key);
        sk.length = KSIZE;
        sk.mem = key;
        ret = db_get(db, &sk, &sv);
        if (ret) {
            pthread_mutex_lock(&found_mutex);
            found++;
            pthread_mutex_unlock(&found_mutex);
        } else {
            INFO("not found key#%s",
                sk.mem);
        }

        if ((i % 10000) == 0) {
            fprintf(stderr, "random read finished %d ops%30s\r",
                i,
                "");
            fflush(stderr);
        }
    }

    cost = ypologismos_kostous(start);
    update_total_cost(&read_cost_mutex, &total_read_cost, cost);

    return NULL;
}
```

Εικόνα 35

Ανάλυση κώδικα db.c

Οι αλλαγές που κάναμε στο αρχείο είναι οι ακόλουθες:

- **Εισαγωγή συγχρονισμού:** Προστέθηκαν mutex και condition variables για τον συγχρονισμό προσβάσεων στις λειτουργίες db_add και db_get.
- **Αλλαγές στις λειτουργίες db_add και db_get:** Βελτιώσεις για την υποστήριξη πολλαπλών αναγνωστών και ενός συγγραφέα.
- **Διαφορετική διαχείριση Memtable και SSTables**

Ακολουθεί η αναλυτική εξήγηση του κώδικα παρακάτω:

Κεντρική Ιδέα

Το αρχείο db.c περιέχει τη βασική λειτουργικότητα που απαιτείται για την αλληλεπίδραση με την προσαρμοσμένη βάση δεδομένων, συμπεριλαμβανομένου του ανοίγματος και κλεισίματος συνδέσεων βάσης δεδομένων, της προσθήκης, ανάκτησης και αφαίρεσης καταχωρήσεων και της επανάληψης των περιεχομένων της βάσης δεδομένων.

❖ Αρχικοποίηση Mutex

Η διαχείριση ταυτόχρονων προσβάσεων σε μια βάση δεδομένων είναι κρίσιμη για την αποφυγή αλλοίωσης δεδομένων και συνθηκών ανταγωνισμού, καθώς και για την αποδοτική ταυτόχρονη πρόσβαση. Η ελεγχόμενη πρόσβαση προστατεύει τη συνέπεια των δεδομένων από τις ταυτόχρονες εγγραφές και αναγνώσεις, αποτρέπει σφάλματα που προκύπτουν από παράλληλες λειτουργίες και βελτιστοποιεί την ταυτόχρονη χρήση για αυξημένη αποδοτικότητα.

Πώς λειτουργούν:

- `pthread_mutex_t add_get_mutex;` και `pthread_mutex_t reader_mutex;`:
Αυτά τα mutexes χρησιμοποιούνται για να εξασφαλίζουν αποκλειστική πρόσβαση σε κρίσιμα τμήματα κώδικα ή δεδομένα, προλαμβάνοντας την ταυτόχρονη εκτέλεση λειτουργιών που μπορεί να προκαλέσουν ασυνέπειες. Ο `add_get_mutex` συγχρονίζει τις λειτουργίες εγγραφής και ανάγνωσης, διασφαλίζοντας την ατομικότητα των εγγραφών και αποφεύγοντας ασυνεπείς αναγνώσεις. Το `reader_mutex` προστατεύει την μεταβλητή που καταγράφει τον αριθμό των ενεργών αναγνωστών, διασφαλίζοντας την ακρίβεια της καταμέτρησης αυτής.
- `pthread_cond_t readers_exist;` και `pthread_cond_t writer_exist;`:
Πρόκειται για μεταβλητές κατάστασης που χρησιμοποιούνται σε συνδυασμό με mutexes για να μπλοκάρουν και να αφυπνίζουν νήματα υπό ορισμένες συνθήκες. Η μεταβλητή συνθήκης `readers_exist` μπορεί να χρησιμοποιηθεί για να σηματοδοτήσει τις αναμενόμενες λειτουργίες εγγραφής ότι οι αναγνώστες έχουν τελειώσει και είναι ασφαλές να συνεχίσουμε με την εγγραφή.

Αντίστροφα, η `writer_exist` σηματοδοτεί σε αναμονή λειτουργίες ανάγνωσης ότι ο συγγραφέας έχει τελειώσει και είναι ασφαλές να διαβάσουμε. Αυτοί οι μηχανισμοί διασφαλίζουν ότι οι συγγραφείς δεν τροποποιούν τα δεδομένα που διαβάζονται και ότι οι αναγνώστες βλέπουν μια συνεπή άποψη των δεδομένων.

➤ `int reader;` και `int writer;`:

Αυτές οι ακέραιες μεταβλητές παρακολουθούν τον αριθμό των ενεργών αναγνωστών και το αν ένας συγγραφέας είναι ενεργός αυτή τη στιγμή. Είναι κρίσιμες για την υλοποίηση του μοτίβου κλειδώματος αναγνώστη-συγγραφέα, όπου το σύστημα επιτρέπει σε πολλούς αναγνώστες να διαβάζουν ταυτόχρονα, αλλά απαιτεί αποκλειστική πρόσβαση για τη γραφή.

```
//arxikopoihsh mutex gia ton sygxronismo tw'n add kai get
pthread_mutex_t add_get_mutex = PTHREAD_MUTEX_INITIALIZER;
//mutex gia ari8mo anagnwstwn
pthread_mutex_t reader_mutex = PTHREAD_MUTEX_INITIALIZER;
//metablhth gia to an yparxoyn anagnwstes
pthread_cond_t readers_exist = PTHREAD_COND_INITIALIZER;
//metablhth gia to an uparxei enas syggrafeas
pthread_cond_t writer_exist = PTHREAD_COND_INITIALIZER;

//arithmos anagnwstwn kai suggrafewn
int reader = 0;
int writer = 0;
```

Εικόνα 36

❖ `int db_add(DB* self, Variant* key, Variant* value)`

Η συνάρτηση `db_add` φροντίζει για την ασφαλή εισαγωγή δεδομένων στη βάση, συγχρονίζοντας τις ταυτόχρονες λειτουργίες εγγραφής και ανάγνωσης, για να αποφεύγονται διενέξεις δεδομένων και να διατηρείται η ακεραιότητα των δεδομένων. Επιπλέον, πραγματοποιεί προληπτικούς ελέγχους για την ανάγκη συμπίεσης του πίνακα μνήμης, βελτιστοποιώντας έτσι την αποδοτικότητα της αποθήκευσης και της ανάκτησης δεδομένων.

```

//prosthiki enos stoixeiou sth db
int db_add(DB* self, Variant* key, Variant* value)
{
    //eksasfalish apokleistikhhs prosvashs gia tous syggrafeis
    pthread_mutex_lock(&add_get_mutex);
    while (reader > 0) { // wait an yparxoun anagnwstes
        pthread_cond_wait(&readers_exist, &add_get_mutex);
    }
    writer++; //o syggrafeas einai energos

    if (memtable_needs_compaction(self->memtable)) {
        INFO("Starting compaction of the memtable after %d insertions and %d deletions",
            self->memtable->add_count, self->memtable->del_count);
        sst_merge(self->sst, self->memtable);
        memtable_reset(self->memtable);
    }

    int var_1 = memtable_add(self->memtable, key, value);

    writer--; //o syggrafeas exei teleiwsei
    if (writer == 0) { //an den yparxoun alloi syggrafeis, dwse shma stous anagnwstes pou perimenoun
        pthread_cond_broadcast(&writer_exist);
    }
    pthread_mutex_unlock(&add_get_mutex);
    return var_1;
}

```

Εικόνα 37

Πώς λειτουργεί:

1. Απόκτηση κλειδώματος:

pthread_mutex_lock(&add_get_mutex);: Η συνάρτηση ξεκινά με το κλείδωμα του mutex **add_get_mutex**, εξασφαλίζοντας την αποκλειστική πρόσβαση στο μπλοκ κώδικα που ακολουθεί. Αυτό εμποδίζει άλλες λειτουργίες εγγραφής να προχωρήσουν μέχρι να ολοκληρωθεί η τρέχουσα εγγραφή, αλλά στοχεύει ειδικά στο να μπλοκάρει τις εγγραφές όταν οι αναγνώστες είναι ενεργοί.

```
pthread_mutex_lock(&add_get_mutex);
```

Εικόνα 38

2. Περίμενε να τελειώσουν οι αναγνώστες:

while (reader > 0) { pthread_cond_wait(&readers_exist, &add_get_mutex); }: Εάν υπάρχουν ενεργοί αναγνώστες ($reader > 0$), η συνάρτηση περιμένει. Χρησιμοποιεί τη μεταβλητή συνθήκης **readers_exist** για να διακόψει το τρέχον νήμα μέχρι να τελειώσουν όλοι οι αναγνώστες. Η αναμονή πραγματοποιείται στο mutex **add_get_mutex**, το οποίο αποδεσμεύεται προσωρινά κατά τη διάρκεια της αναμονής για να επιτραπεί στους αναγνώστες να ολοκληρώσουν τις λειτουργίες τους, και στη συνέχεια αποκτάται ξανά μόλις λάβει σήμα για να συνεχίσει.

```

while (reader > 0) { // wait an yparxoun anagnwstes
    pthread_cond_wait(&readers_exist, &add_get_mutex);
}

```

Εικόνα 39

3. Δείκτης λειτουργίας εγγραφής:

writer++;: Αυξάνει τον μετρητή **writer** για να δείξει ότι μια λειτουργία εγγραφής βρίσκεται σε εξέλιξη. Αυτό είναι ζωτικής σημασίας για τις λειτουργίες ανάγνωσης, ώστε να ελέγχουν και να περιμένουν αν γίνεται εγγραφή.

```
writer++; //o syggrafeas einai energos
```

Εικόνα 40

4. Έλεγχος συμπίκνωσης:

Ελέγχει αν ο πίνακας μνήμης χρειάζεται συμπίεση βάσει ορισμένων κριτηρίων (όπως το μέγεθος ή ο αριθμός των πράξεων) με την εντολή **memtable_needs_compaction(self->memtable)**. Εάν απαιτείται συμπίεση, εκτελεί τη συμπίεση συγχωνεύοντας τον πίνακα μνήμης στους SSTables (μια μορφή αποθήκευσης στο δίσκο) και στη συνέχεια επαναφέρει τον πίνακα μνήμης για περαιτέρω λειτουργίες εγγραφής.

```
if (memtable_needs_compaction(self->memtable)) {  
    INFO("Starting compaction of the memtable after %d insertions and %d deletions",  
        self->memtable->add_count, self->memtable->del_count);  
    sst_merge(self->sst, self->memtable);  
    memtable_reset(self->memtable);  
}
```

Εικόνα 41

5. Εισαγωγή:

int var_1 = memtable_add(self->memtable, key, value);: Προσθέτει το ζεύγος κλειδί-τιμή στον πίνακα μνήμης. Αυτή η λειτουργία είναι η βασική λειτουργικότητα της συνάρτησης, αποθηκεύοντας τα παρεχόμενα δεδομένα στη δομή μνήμης της βάσης δεδομένων.

```
int var_1 = memtable_add(self->memtable, key, value);
```

Εικόνα 42

6. Ολοκλήρωση εγγραφής:

Μετά τη λειτουργία εισαγωγής, μειώνει τον μετρητή **writer** (**writer--;**) για να δηλώσει ότι η τρέχουσα λειτουργία εγγραφής έχει ολοκληρωθεί. Εάν αυτή η εγγραφή ήταν η μοναδική (**writer == 0**), σηματοδοτεί σε τυχόν αναμενόμενες πράξεις ανάγνωσης ότι μπορούν να προχωρήσουν με μετάδοση στη μεταβλητή συνθήκης **writer_exist**.

```
writer--; //o syggrafeas exei teleiwsei
```

Εικόνα 43

7. Απελευθέρωση κλειδώματος:

pthread_mutex_unlock(&add_get_mutex);: Απελευθερώνει το mutex που κλειδώθηκε στην αρχή της συνάρτησης, επιτρέποντας σε άλλες λειτουργίες εγγραφής να προχωρήσουν.

```
if (writer == 0) { //an den yparxoun alloi syggrafeis, dwse shma stous anagnwstes pou perimenoun  
    pthread_cond_broadcast(&writer_exist);  
}  
pthread_mutex_unlock(&add_get_mutex);
```

Εικόνα 44

Note: Πληροφορίες για **pthread_cond_broadcast** θα βρείτε αναλυτικά στη σελίδα 30.

8. Κατάσταση επιστροφής:

Επιστρέφει την κατάσταση της λειτουργίας προσθήκης του πίνακα μνήμης (`var_1`), η οποία δείχνει αν το ζεύγος κλειδιού-τιμής εισήχθη επιτυχώς.

```
return var_1;
```

Εικόνα 45

❖ `int db_get(DB* self, Variant* key, Variant* value)`

Η συνάρτηση `db_get` από τη βάση δεδομένων διευκολύνει την ασφαλή και ταυτόχρονη ανάκτηση τιμών με βάση συγκεκριμένα κλειδιά σε ένα πολυνηματικό περιβάλλον, επικεντρώνοντας στην αποφυγή συγκρούσεων με λειτουργίες εγγραφής και υποστηρίζοντας ταυτόχρονες αναγνώσεις χωρίς αλληλεπιδράσεις. Αλληλεπιδρά με τη μνήμη (`memtable`) και δομές αποθήκευσης σε δίσκο (`SSTable`) για την απόδοσή της.

```
//anagnwsh stoxeiou apo th bash dedomenwn
int db_get(DB* self, Variant* key, Variant* value)
{
    int var_2 = 0;

    pthread_mutex_lock(&reader_mutex);
    while (writer > 0) { //wait an o syggrafeas einai energos
        pthread_cond_wait(&writer_exist, &reader_mutex);
    }
    reader++; //o anagnwsths einai energos
    pthread_mutex_unlock(&reader_mutex);

    if (memtable_get(self->memtable->list, key, value) == 1) {
        var_2 = 1; //vrethike sth memtable
    } else {
        //an den exei vrethei sth memtable prospathise na to vreis sto sst
        var_2 = sst_get(self->sst, key, value); //an vrethei sto sst tha ginei 1, alliws 0
    }
    pthread_mutex_lock(&reader_mutex);
    reader--; //o anagnwsths teleiwse

    if (reader == 0) { //an einai o teleutaios anagnwsths, dwse shma stous syggrafeis pou perimenoun
        pthread_cond_signal(&readers_exist);
    }
    pthread_mutex_unlock(&reader_mutex);

    return var_2;
}
```

Εικόνα 46

Πώς λειτουργεί:

1. Λήψη Κλειδώματος για Αναγνώστες:

pthread_mutex_lock(&reader_mutex);: Κλειδώνει ένα mutex αφιερωμένο στη διαχείριση των μετρήσεων αναγνωστών. Αυτό εξασφαλίζει ότι η λειτουργία αύξησης του αριθμού των ενεργών αναγνωστών γίνεται με ασφάλεια, χωρίς συγκρούσεις.

```
pthread_mutex_lock(&reader_mutex);
```

Εικόνα 47

2. Αναμονή για Ενεργούς Συγγραφείς:

while (writer > 0) { pthread_cond_wait(&writer_exist, &reader_mutex); }: Αν υπάρχει ενεργός συγγραφέας ($writer > 0$), η συνάρτηση περιμένει. Αυτό γίνεται για να διασφαλιστεί ότι η λειτουργία ανάγνωσης δεν θα προχωρήσει ενώ μια λειτουργία εγγραφής που θα μπορούσε να τροποποιήσει τα δεδομένα που διαβάζονται βρίσκεται σε εξέλιξη. Η μεταβλητή συνθήκης **writer_exist** χρησιμοποιείται για την αναμονή και ενημερώνεται από έναν συγγραφέα μόλις τελειώσει, στο σημείο αυτό η λειτουργία ανάγνωσης μπορεί να συνεχίσει.

```
while (writer > 0) { //wait an o syggrafeas einai energos  
    pthread_cond_wait(&writer_exist, &reader_mutex);  
}
```

Εικόνα 48

3. Αύξηση Μέτρησης Αναγνωστών:

- **reader++;**: Αυξάνει τον αριθμό των ενεργών αναγνωστών. Αυτό δηλώνει σε πιθανούς συγγραφείς ότι μια λειτουργία ανάγνωσης βρίσκεται σε εξέλιξη.
- **pthread_mutex_unlock(&reader_mutex);**: Ξεκλειδώνει το **mutex** αναγνωστών μετά από ασφαλή αύξηση της μέτρησης αναγνωστών.

```
reader++; //o anagnwsths einai energos  
pthread_mutex_unlock(&reader_mutex);
```

Εικόνα 49

4. Προσπάθεια Ανάκτησης της Τιμής:

- Πρώτον, προσπαθεί να βρει το κλειδί στο memtable χρησιμοποιώντας **memtable_get(self->memtable->list, key, value)**. Αν το κλειδί βρεθεί ($= 1$), η **var_2** ορίζεται σε 1, δηλώνοντας επιτυχία.
- Αν το κλειδί δεν βρεθεί στο memtable, στη συνέχεια ελέγχει το SSTable χρησιμοποιώντας **sst_get(self->sst, key, value)**. Η επιστρεφόμενη τιμή από το **sst_get** ανατίθεται στο **returned_value**, δηλώνοντας εάν το κλειδί βρέθηκε στο SSTable.

```
if (memtable_get(self->memtable->list, key, value) == 1) {  
    var_2 = 1; //vrethike sth memtable  
} else {  
    //an den exei vrethei sth memtable prospathise na to vreis sto sst  
    var_2 = sst_get(self->sst, key, value); //an vrethei sto sst tha ginei 1, allwos 0  
}
```

Εικόνα 50

5. Μείωση Μέτρησης Αναγνωστών και Ειδοποίηση Αναμονής Συγγραφέων:

- Μετά την προσπάθεια ανάκτησης, κλειδώνει ξανά το `reader_mutex` για να μειώσει ασφαλώς τη μέτρηση των αναγνωστών. Έπειτα μειώνει τη μέτρηση των ενεργών αναγνωστών.
- Αν αυτή η λειτουργία ήταν ο τελευταίος ενεργός αναγνώστης (`reader == 0`), ειδοποιεί τυχόν αναμένοντες συγγραφείς μέσω της μεταβλητής συνθήκης `readers_exist`. Αυτό δηλώνει στους συγγραφείς ότι είναι ασφαλές να προχωρήσουν με την εγγραφή εφόσον δεν υπάρχουν ενεργοί αναγνώστες πλέον. Τέλος, ξεκλειδώνει το `reader_mutex`.

```
pthread_mutex_lock(&reader_mutex);
reader--; //o anagnwsths teleiwsse

if (reader == 0) { //an einai o teleutaios anagnwsths, dwse shma stous syggrafeis pou perimenoun
    pthread_cond_signal(&readers_exist);
}
pthread_mutex_unlock(&reader_mutex);

return var_2;
```

Εικόνα 51

Ανάλυση κώδικα db.h

Προστέθηκαν στο νέο bench.h

1. Αρχικοποίηση `int readers_num` και `int writer_num`:

Χρησιμοποιούνται για να παρακολουθούν τον αριθμό των νημάτων ανάγνωσης και εγγραφής που διαβάζουν και γράφουν αντίστοιχα αυτή τη στιγμή από τη βάση δεδομένων. Αυτός ο αριθμός είναι χρήσιμος για τη διαχείριση της πρόσβασης, διασφαλίζοντας ότι οι λειτουργίες ανάγνωσης δεν παρεμβαίνουν στις λειτουργίες εγγραφής και αντίστροφα.

2. `pthread_mutex_t no_readers_writer;`:

Δηλώνεται μια μεταβλητή mutex με το όνομα `no_readers_writer`, ο οποίος μπορεί να χρησιμοποιηθεί για να διασφαλιστεί ότι είτε κανένας αναγνώστης δεν διαβάζει είτε κανένας συγγραφέας δεν γράφει σε οποιαδήποτε δεδομένη στιγμή.

3. `pthread_cond_t readers_exist;` `pthread_cond_t writer_exist;`:

Οι μεταβλητές αυτές χρησιμοποιούνται για να σηματοδοτούν αλληλέγγυα μεταξύ των νημάτων αναγνωστών και συγγραφέων: η πρώτη ειδοποιεί τους συγγραφείς ότι όλοι οι αναγνώστες έχουν τελειώσει και είναι ασφαλές να αρχίσουν να γράφουν, ενώ η δεύτερη ειδοποιεί τους αναγνώστες ότι ο συγγραφέας έχει τελειώσει και είναι ασφαλές να αρχίσουν πάλι να διαβάζουν.

```
typedef struct _db {
    // char basedir[MAX_FILENAME];
    char basedir[MAX_FILENAME+1];
    SST* sst;
    MemTable* memtable;
    int readers_num;
    int writer_num;
    pthread_mutex_t no_readers_writer;
    pthread_cond_t readers_exist;
    pthread_cond_t writer_exist;
} DB;
```


Ειδοποίηση / Αφύπνιση Νήματος

Η χρήση των `pthread_cond_signal()` και `pthread_cond_broadcast()`, όπως βλέπουμε και στην θεωρία, στο πλαίσιο των πολυνηματικών προγραμμάτων είναι κρίσιμη για τον συγχρονισμό των νημάτων (`threads`) και τη διαχείριση της πρόσβασης σε κοινόχρηστους πόρους.

❖ `pthread_cond_signal()`

Λειτουργία: Ειδοποιεί ένα από τα πολλά νήματα που μπορεί να περιμένουν σε μια μεταβλητή συνθήκης. Αυτό σημαίνει ότι αν πολλά νήματα έχουν καλέσει την `pthread_cond_wait()` για την ίδια μεταβλητή συνθήκης, μόνο ένα από αυτά θα "ξυπνήσει" και θα προχωρήσει.

Εφαρμογή στον κώδικα: Στο τμήμα όπου μειώνεται ο μετρητής των αναγνωστών (`reader--`), και αν δεν υπάρχουν πλέον αναγνώστες, χρησιμοποιείται η `pthread_cond_signal()` για να ειδοποιηθεί ένα από τα πιθανώς αναμενόμενα νήματα εγγραφής ότι μπορεί πλέον να προχωρήσει.

❖ `pthread_cond_broadcast()`

Λειτουργία: Ειδοποιεί όλα τα νήματα που περιμένουν σε κάποια μεταβλητή συνθήκης. Αυτό είναι χρήσιμο όταν πολλαπλά νήματα πρέπει να ενημερωθούν για μια αλλαγή κατάστασης και πρέπει όλα μαζί να προχωρήσουν.

Εφαρμογή στον κώδικα: Χρησιμοποιείται μετά την ολοκλήρωση μιας λειτουργίας εγγραφής (`writer--`) για να ειδοποιηθεί όλα τα νήματα ανάγνωσης που μπορεί να περιμένουν, ότι οι λειτουργίες εγγραφής έχουν ολοκληρωθεί και μπορούν να προχωρήσουν.

Ειδοποίηση / Αφύπνιση Νήματος

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Με την `pthread_cond_signal()` ένα νήμα ειδοποιεί κάποιο άλλο νήμα που περιμένει σε κάποια μεταβλητή συνθήκης:

- μπορεί να έχουν μπλοκάρει στη συγκεκριμένη μεταβλητή συνθήκης πολλά νήματα (καλώντας `pthread_cond_wait()`). Μόνο ένα από αυτά θα ξεμπλοκάρει και θα συνεχίσει.

Με την `pthread_cond_broadcast()` ένα νήμα ειδοποιεί όλα τα νήματα που περιμένουν σε κάποια μεταβλητή συνθήκης:

- Όλα τα μπλοκαρισμένα νήματα από αυτά θα ξεμπλοκάρουν και θα συνεχίσουν

Το νήμα που θα καλέσει `signal/broadcast` έχει κλειδώσει ένα ανάλογο `mutex` → πρέπει επίσης να ελευθερώσει το `mutex` ώστε να συνεχιστεί η εκτέλεση της `pthread_cond_wait()` για τα άλλα νήματα.

ΜΥΥ601: Λειτουργικά Συστήματα, Πανεπιστήμιο Ιωαννίνων 48

Εικόνα 52

□ 3ο βήμα: Προηγμένη Λύση

Δυστυχώς, τη προηγμένη λύση για την εφαρμογή συγχρονισμού αναγνωστών-γραφέων στην εκτέλεση των λειτουργιών add και get, δεν καταφέραμε να την υλοποιήσουμε.

Μπορέσαμε μόνο να αναβαθμίσουμε συγκεκριμένα αρχεία, όπως τα bench.c, bench.h, kiwi.c, db.c, και db.h, όπως έχουμε αναφέρει νωρίτερα.

- Χρησιμοποιήσαμε τεχνικές αμοιβαίου αποκλεισμού για να επιτρέψουμε σε πολλαπλά νήματα να εκτελούνται ταυτόχρονα χωρίς να παρεμβαίνουν το ένα στη λειτουργία του άλλου.
- Αποφασίσαμε να μην επιτρέπουμε σε πολλούς χρήστες που γράφουν δεδομένα να δραστηριοποιούνται ταυτόχρονα, καθώς αυτό θα μπορούσε να οδηγήσει σε ανακριβή καταχωρήσεις.
- Επιτρέψαμε την ταυτόχρονη ανάγνωση από πολλούς χρήστες, καθώς αυτό δεν προκαλεί προβλήματα,
- Απαγορεύσαμε την ταυτόχρονη ανάγνωση και εγγραφή δεδομένων, γιατί μπορεί να μην είναι δυνατή η ακριβής ανάκτηση των δεδομένων.

Για το τρίτο βήμα της εργασίας, θα εστιάζαμε στην υλοποίηση του συγχρονισμού αναγνωστών-εγγραφέων στην εκτέλεση των λειτουργιών add και get της μηχανής αποθήκευσης, χρησιμοποιώντας τις pthreads. Αυτή η προσέγγιση στοχεύει στην αποδοτική διαχείριση παράλληλων προσβάσεων στη βάση δεδομένων, επιτρέποντας πολλαπλές αναγνώσεις ταυτόχρονα, ενώ εξασφαλίζει ότι οι εγγραφές εκτελούνται αποκλειστικά.

Για το συγχρονισμό αναγνωστών-εγγραφέων, θα χρησιμοποιούσαμε τα παρακάτω βήματα:

1. **Εισαγωγή Μηχανισμού Κλειδώματος:** Θα ξεκινούσαμε δημιουργώντας μια δομή κλειδώματος για αναγνώστες και συγγραφείς. Αυτή η δομή θα περιέχει μεταβλητές για την καταμέτρηση των ενεργών αναγνωστών, ένα mutex για τον συγχρονισμό της πρόσβασης στην δομή και μια μεταβλητή συνθήκης για τους συγγραφείς.
2. **Υλοποίηση Λογικής Αναγνώστη:**
 - Κατά την έναρξη μιας ανάγνωσης, θα αυξάνουμε τον αριθμό των ενεργών αναγνωστών.
 - Αν είναι ο πρώτος αναγνώστης, θα πρέπει να αποκτήσει ένα κλείδωμα που εμποδίζει την είσοδο νέων συγγραφέων.
 - Μετά την ολοκλήρωση της ανάγνωσης, θα μειώσουμε τον αριθμό των ενεργών αναγνωστών.
 - Αν ήταν ο τελευταίος αναγνώστης, θα απελευθερώσει το κλείδωμα, επιτρέποντας στους συγγραφείς να προχωρήσουν.

3. Υλοποίηση Λογικής Συγγραφέα:

- Πριν από την έναρξη μιας εγγραφής, ο συγγραφέας θα πρέπει να περιμένει μέχρι να μην υπάρχουν ενεργοί αναγνώστες.
- Αυτό επιτυγχάνεται χρησιμοποιώντας την μεταβλητή συνθήκης για να περιμένει μέχρι όλοι οι ενεργοί αναγνώστες να έχουν ολοκληρώσει.
- Μετά την ολοκλήρωση της εγγραφής, ο συγγραφέας θα ειδοποιήσει τους αναμένοντες αναγνώστες και συγγραφείς ότι η διαδικασία έχει ολοκληρωθεί και μπορούν να προχωρήσουν.

4. Ενσωμάτωση στις Λειτουργίες db_add και db_get:

Η παραπάνω λογική θα ενσωματωθεί στις λειτουργίες db_add και db_get. Θα χρησιμοποιήσουμε τις δομές κλειδώματος που δημιουργήσαμε για να ελέγξουμε την πρόσβαση στην μηχανή αποθήκευσης.

5. Πειραματική Επαλήθευση και Δοκιμές:

Τέλος, θα πραγματοποιήσουμε σειρά από πειραματικές δοκιμές για να επαληθεύσουμε την σωστή λειτουργία της υλοποίησής μας. Θα δοκιμάσουμε σενάρια με μεγάλο αριθμό ταυτόχρονων αναγνώστων και μεμονωμένους συγγραφείς, καθώς και σενάρια με εναλλασσόμενες αναγνώσεις και εγγραφές για να εξασφαλίσουμε την ορθή λειτουργία του συστήματος.

□ 4ο βήμα: Πειραματική Επαλήθευση

Στο τέταρτο βήμα, καλούμαστε να προσθέσουμε νέα ορίσματα στη γραμμή εντολών του αρχείου bench.c, ώστε να μπορούμε να λαμβάνουμε μια επιπλέον είσοδο από τον χρήστη. Αυτή η είσοδος θα καθορίζει το ποσοστό των εντολών add και get. Στις επόμενες εικόνες μπορούμε να δούμε την σωστή λειτουργία του readwrite, όπως αναφέραμε στη θεωρητική εξήγηση παραπάνω.
(Σελ. 9 - Ανάλυση bench.c)

Για 100.000 λειτουργίες, με ποσοστά 50% - 50% , για 20 read και write threads:

```
myy601@myy601lab1:~/kiwi/kiwi-source$ cd bench
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 100000 50 50
Give number of threads for reading: 20
Give number of threads for writing: 20

[1779] 06 Apr 17:03:41.185 : log.c:46 Removing old log file testab/s1/12.log
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+
|Random-Read      (done:50000, found:42947): 0.003200 sec/op; 312.5 reads /sec(estimated); cost:160.
000(sec)
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+
|Random-Write     (done:50000): 0.003340 sec/op; 299.4 writes/sec(estimated); cost:167.000(sec);
```

Εικόνα 53

Για 100.000 λειτουργίες, με ποσοστά 10% - 90% , για 20 read και write threads:

```
myy601@myy601lab1:~/kiwi/kiwi-source$ cd bench
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 100000 10 90
Give number of threads for reading: 20
Give number of threads for writing: 20

+-----+-----+-----+-----+-----+
+
|Random-Read   (done:10000, found:9593): 0.002000 sec/op; 500.0 reads /sec(estimated); cost:20.000(sec)
+-----+-----+-----+-----+-----+
+
|Random-Write   (done:90000): 0.001678 sec/op; 596.0 writes/sec(estimated); cost:151.000(sec);
myy601@myy601lab1:~/kiwi/kiwi-source/bench$
```

Εικόνα 54

Για 100.000 λειτουργίες, με ποσοστά 0% - 100% , για 20 read και write threads:

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 100000 0 100
Give number of threads for reading: 20
Give number of threads for writing: 20

+-----+-----+-----+-----+-----+
+
|Random-Read   (done:0, found:0): -nan sec/op; -nan reads /sec(estimated); cost:0.000(sec)
+-----+-----+-----+-----+-----+
+
|Random-Write   (done:100000): 0.001510 sec/op; 662.3 writes/sec(estimated); cost:151.000(sec);
myy601@myy601lab1:~/kiwi/kiwi-source/bench$
```

Εικόνα 55

□ 5ο βήμα: Στατιστικά Απόδοσης

Στο πέμπτο και τελευταίο βήμα της άσκησης, θα συλλέξουμε και θα παρουσιάσουμε στατιστικά στοιχεία που αφορούν την απόδοση των λειτουργιών **add** και **get**, επικεντρωμένοι στον χρόνο απόκρισης και στον ρυθμό απόδοσης. Ο **χρόνος απόκρισης** αντιπροσωπεύει το διάστημα που απαιτείται για την εκτέλεση των διαφόρων λειτουργιών από το πρόγραμμα υπό τη συνθήκη παράλληλης εκτέλεσης από πολλαπλά νήματα, ενώ ο **ρυθμός απόδοσης** δείχνει τον αριθμό των εργασιών που ολοκληρώνονται ανά μονάδα χρόνου. Για την ακριβή μέτρηση και συγκρίσιμη απεικόνιση των δεδομένων αυτών, θα πρέπει να χρησιμοποιήσουμε απαραίτητες τεχνικές συγχρονισμού που εξασφαλίζουν τον αμοιβαίο αποκλεισμό μεταξύ των νημάτων που ενημερώνουν τις μετρήσεις. Τα αποτελέσματα αυτά θα παρουσιαστούν με τη μορφή γραφικών παραστάσεων, επιτρέποντας μια άμεση και οπτική κατανόηση των αποδόσεων ως προς τον χρόνο και τον ρυθμό των λειτουργιών.

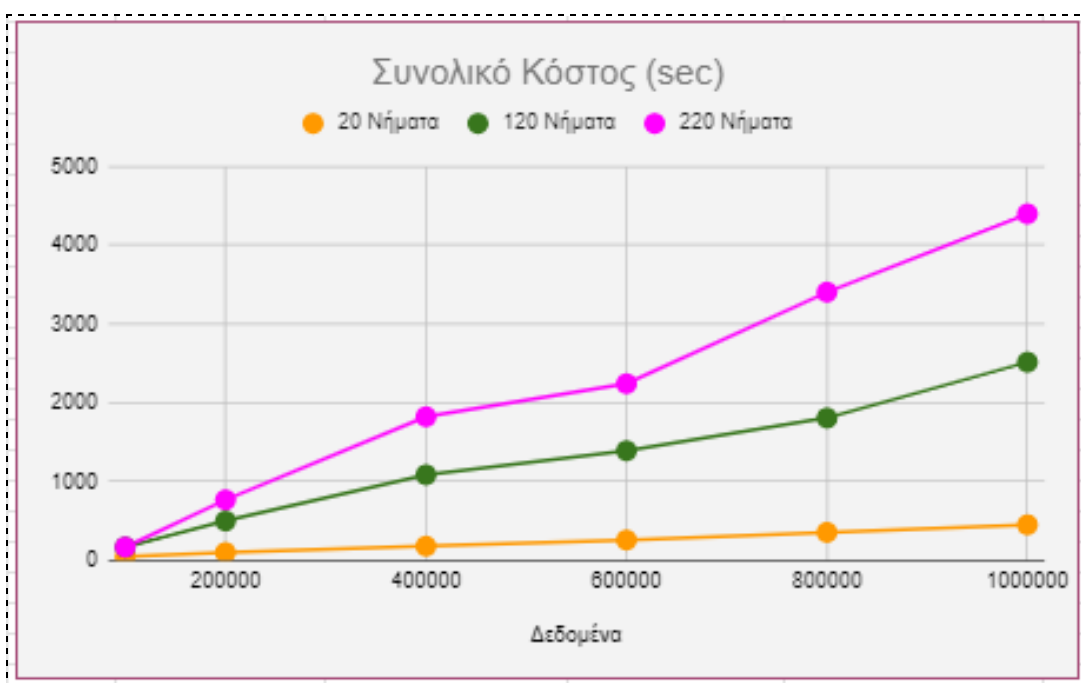
***Note:** Πρέπει να λάβουμε υπόψη ότι οι ίδιες εντολές μπορεί να παράγουν διαφορετικά αποτελέσματα, εξαιτίας της απροβλεπτικότητας που εισάγουν τα νήματα στην εκτέλεση.

❖ Writes:

Συνολικό Κόστος (sec)

	Συνολικό κόστος (sec)		
	Νήματα		
Δεδομένα	20 Νήματα	120 Νήματα	220 Νήματα
100000	44	168	168
200000	99	500	768
400000	180	1086	1823
600000	258	1391	2243
800000	355	1807	3405
1000000	451	2516	4399

Εικόνα 56



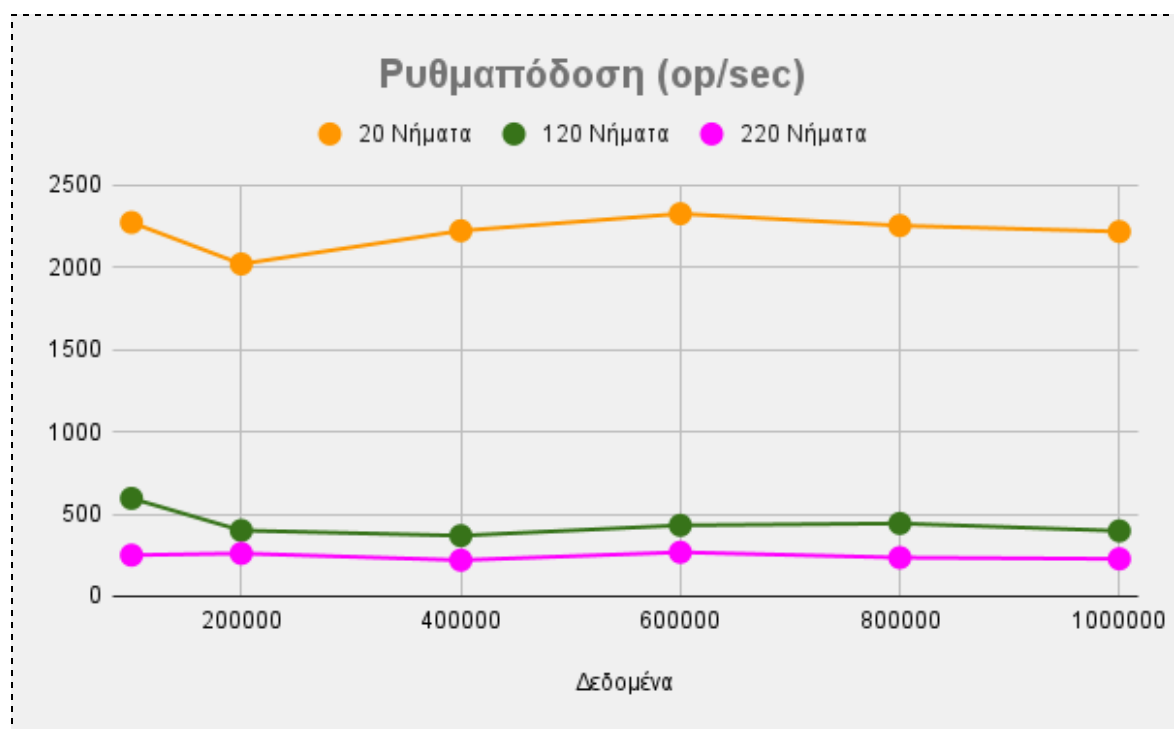
Εικόνα 57

Από το γράφημα της εικόνας 57 είναι φανερό ότι το συνολικό κόστος των διεργασιών write αυξάνεται με την προσθήκη περισσότερων νημάτων στο σύστημα αποθήκευσης. Παρόλο που η χρήση 20 νημάτων είναι η λιγότερο αποδοτική σε σχέση με τα 120 και τα 220 νήματα, η αύξηση του αριθμού των νημάτων από 120 σε 220 δεν επιφέρει σημαντική μείωση στον χρόνο. Ενδιαφέρον παρουσιάζει η ανάλογη αύξηση του κόστους με την προσθήκη νημάτων, δείχνοντας ότι παρά την βελτιωμένη απόδοση, το κόστος μπορεί να αυξάνει υπερβολικά. Η βελτιστοποίηση του αριθμού των νημάτων είναι επομένως κρίσιμη για την επίτευξη οικονομικών και αποδοτικών λειτουργιών εγγραφής.

Ρυθμαπόδοση (op/sec)

Δεδομένα	Ρυθμαπόδοση (op/sec)		
	Νήματα		
	20 Νήματα	120 Νήματα	220 Νήματα
100000	2272,7	595,2	250
200000	2020,2	400	260,4
400000	2222,2	368,3	219,4
600000	2325,6	431,3	267,5
800000	2253,5	442,7	234,9
1000000	2217,3	397,5	227,3

Εικόνα 58



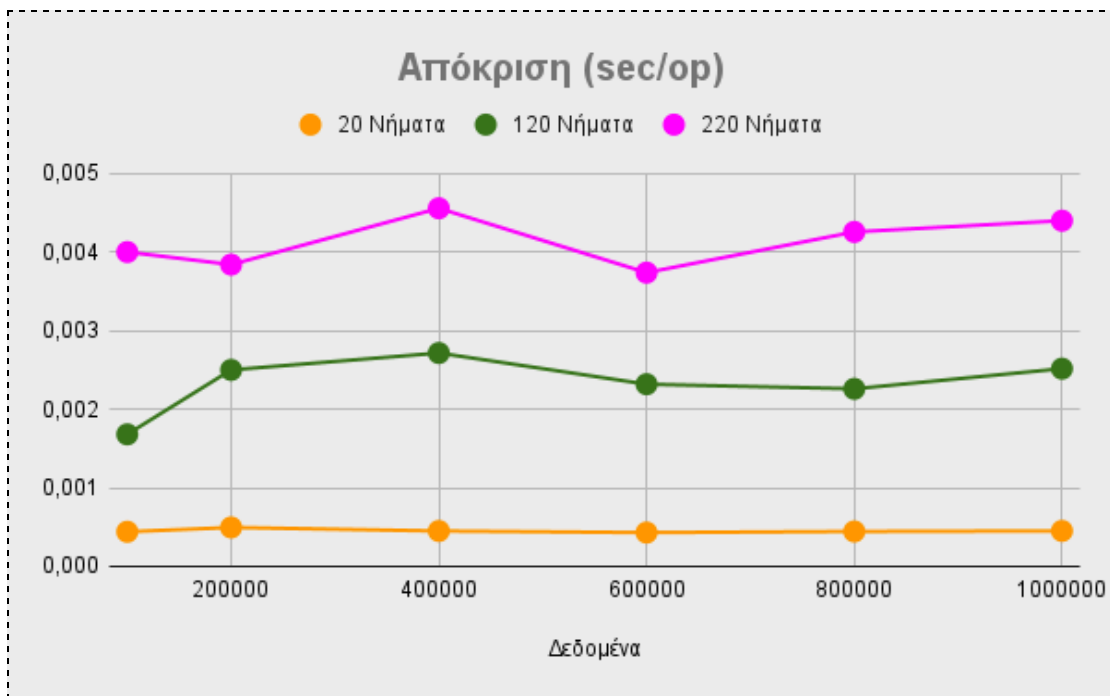
Εικόνα 59

Τα δεδομένα από το διάγραμμα της εικόνας 59 δείχνουν ότι με 20 νήματα, η ρυθμαπόδοση είναι σταθερή και υψηλή, ενώ με 120 και 220 νήματα η απόδοση παραμένει σταθερά χαμηλή. Η χρήση πολλών νημάτων φαίνεται να μην αυξάνει την απόδοση ανάλογα, ένδειξη ότι η διαχείριση των νημάτων και οι διαθέσιμοι πόροι παίζουν κρίσιμο ρόλο στην επίτευξη της μέγιστης ρυθμαπόδοσης.

Απόκριση (sec/op)

	Απόκριση (sec/op)		
	Νήματα		
Δεδομένα	20 Νήματα	120 Νήματα	220 Νήματα
100000	0,00044	0,00168	0,004
200000	0,000495	0,0025	0,00384
400000	0,00045	0,002715	0,004557
600000	0,00043	0,002318	0,003738
800000	0,000444	0,002259	0,004256
1000000	0,000451	0,002516	0,004399

Εικόνα 60



Εικόνα 61

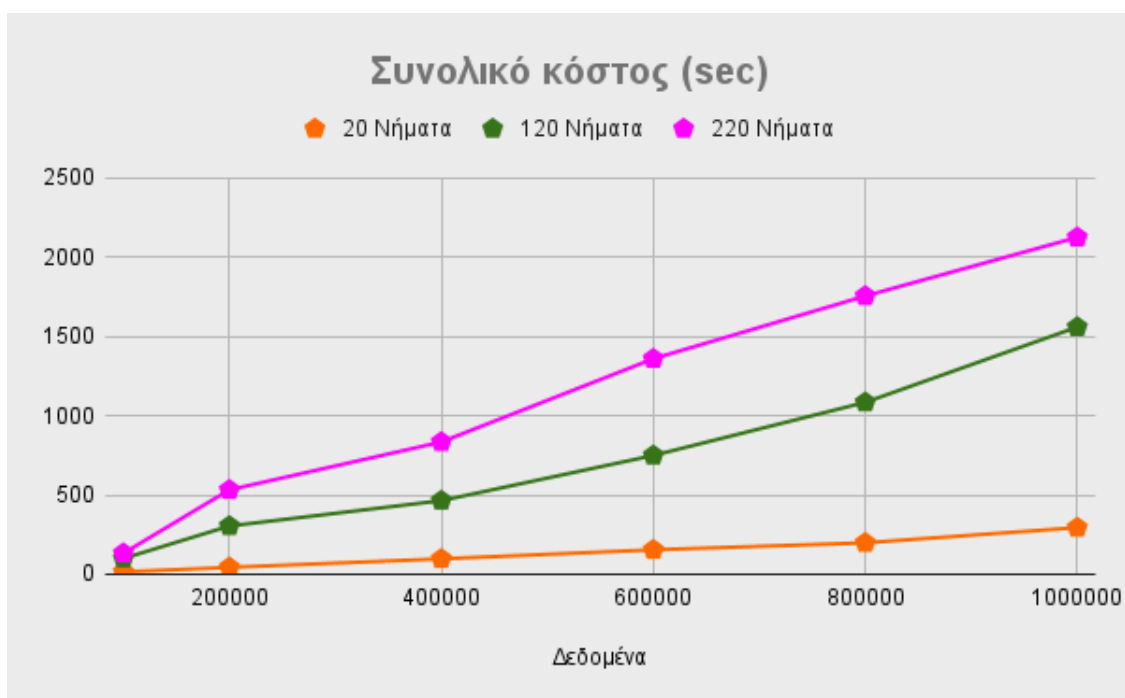
Στο διάγραμμα απόκρισης του συστήματος για λειτουργίες write της εικόνας 61, παρατηρούμε σχεδόν σταθερό χρόνο ανά εγγραφή για κάθε συγκεκριμένο αριθμό νημάτων, αντανακλώντας την προσδοκώμενη συμπεριφορά του συστήματος. Ωστόσο, η απόκριση αυξάνεται καθώς αυξάνονται τα νήματα, δείχνοντας ότι το μεγαλύτερο πλήθος νημάτων μπορεί να επηρεάσει αρνητικά την αποδοτικότητα, πιθανώς λόγω αυξημένου ανταγωνισμού.

❖ Reads:

Συνολικό Κόστος (sec)

	Συνολικό κόστος (sec)		
	Νήματα		
Δεδομένα	20 Νήματα	120 Νήματα	220 Νήματα
100000	15	99	132
200000	43	304	532
400000	97	464	834
600000	154	749	1359
800000	198	1084	1755
1000000	294	1559	2125

Εικόνα 62



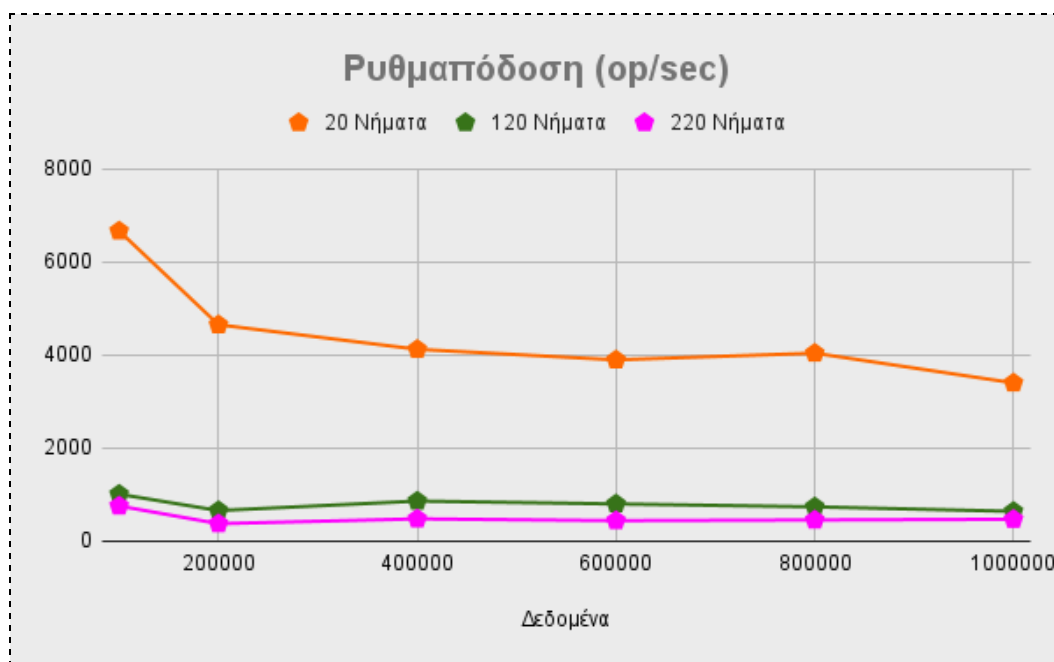
Εικόνα 63

Κατά τη διαδικασία εγγραφής, η τοποθέτηση ενός αυξανόμενου αριθμού νημάτων στο σύστημα αποθήκευσης, διατηρώντας σταθερό τον όγκο των δεδομένων, επιταχύνει σημαντικά την ολοκλήρωση των επερχόμενων αναγνώσεων χάρη στην παραλληλία των διεργασιών. Ωστόσο, αυτό συνοδεύεται από μια αναλογική αύξηση στο συνολικό κόστος, στο οποίο συμπεριλαμβάνεται η διάρκεια κάθε νήματος, παρόμοια με τη διαδικασία των εγγραφών. Από τη σύγκριση των συνολικών κοστών ανά διάγραμμα, καταλήγουμε στο συμπέρασμα ότι, διατηρώντας ίδιο όγκο δεδομένων και αριθμό νημάτων σε κάθε διαδικασία, το συνολικό κόστος των εγγραφών υπερβαίνει σημαντικά εκείνο των αναγνώσεων. Αυτό είναι αναμενόμενο, καθώς οι αναγνώσεις πραγματοποιούνται σε παράλληλο μοντέλο, ενώ οι εγγραφές όχι.

Ρυθμαπόδοση (op/sec)

Δεδομένα	Ρυθμαπόδοση (op/sec)		
	Νήματα		
	20 Νήματα	120 Νήματα	220 Νήματα
100000	6666,7	1010,1	757,6
200000	4651,2	657,9	375,9
400000	4123,7	862,1	479,6
600000	3896,1	801,1	441,5
800000	4040,4	738	455,8
1000000	3401,4	641,4	470,6

Εικόνα 64



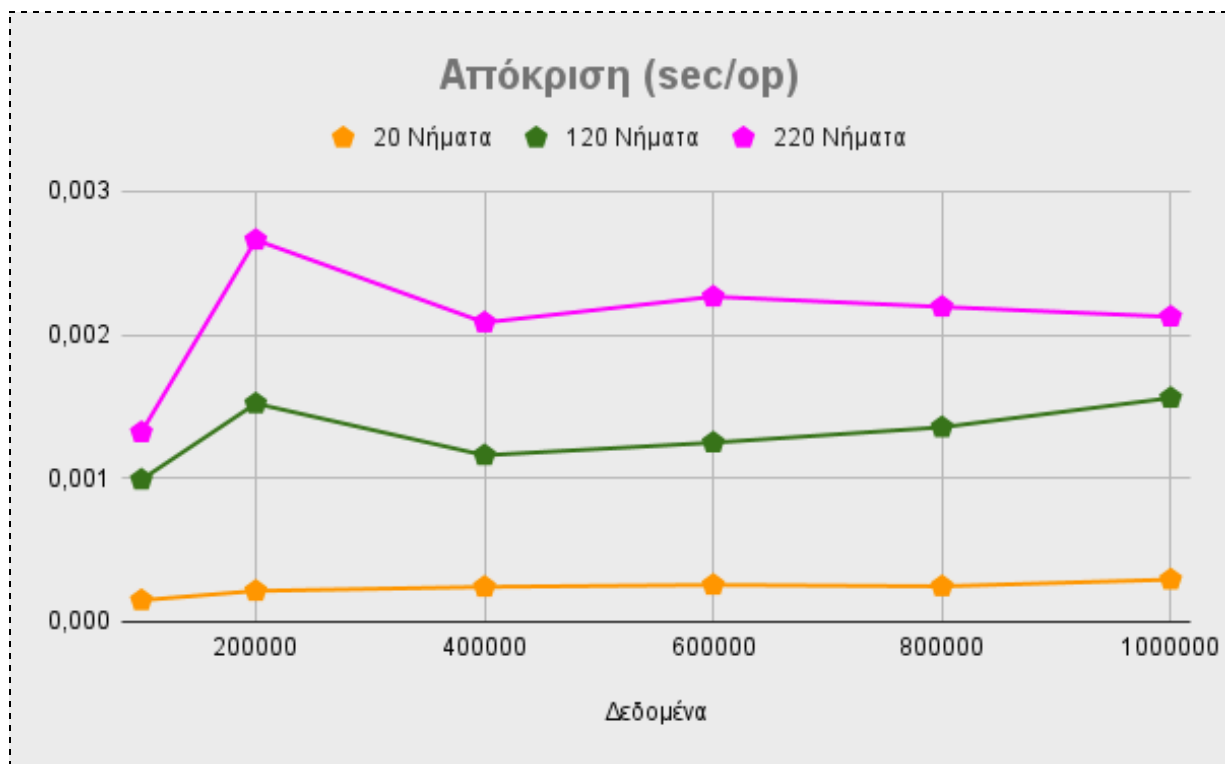
Εικόνα 65

Αυτό το γράφημα εμφανίζει την απόδοση μιας λειτουργίας read, μετρούμενη σε λειτουργίες ανά δευτερόλεπτο, για 20, 120 και 220 νήματα. Είναι προφανές ότι για 120 και 220 νήματα, η απόδοση παραμένει σχετικά σταθερή, υποδεικνύοντας ομαλές και συνεπείς λειτουργίες μηχανής αποθήκευσης. Επιπλέον, καθώς αυξάνεται ο αριθμός των νημάτων, η απόδοση μειώνεται. Αυτό είναι αναμενόμενο, διότι με σταθερό όγκο δεδομένων και αυξανόμενο κόστος ανά νήμα, η αναλογία της απόδοσης προς τον αριθμό των νημάτων είναι πιθανό να μειωθεί.

Απόκριση (sec/op)

	Απόκριση (sec/op)		
	Νήματα		
Δεδομένα	20 Νήματα	120 Νήματα	220 Νήματα
100000	0,00015	0,00099	0,00132
200000	0,000215	0,00152	0,00266
400000	0,000243	0,00116	0,002085
600000	0,000257	0,001248	0,002265
800000	0,000247	0,001355	0,002194
1000000	0,000294	0,001559	0,002125

Εικόνα 66



Εικόνα 67

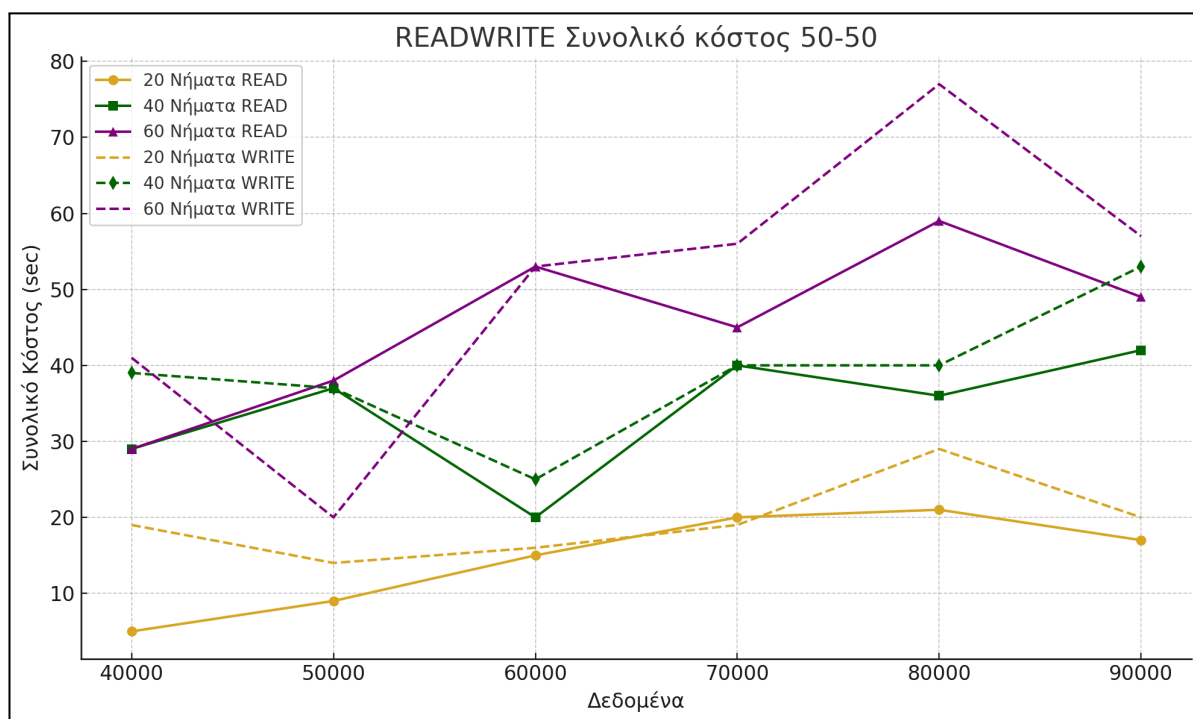
Το παραπάνω γράφημα απεικονίζει το χρόνο απόκρισης του συστήματος για μια λειτουργία read. Η προσδοκία για ένα καλά ρυθμισμένο σύστημα είναι ότι ο χρόνος απόκρισης θα πρέπει να είναι σχετικά σταθερός, ανεξάρτητα από τον όγκο των δεδομένων που υποβάλλονται σε επεξεργασία. Αυτό επιβεβαιώνεται γενικά από το γράφημα, το οποίο δείχνει μια σχεδόν σταθερή γραμμή για κάθε αριθμό νημάτων. Ειδικότερα, το γράφημα δείχνει ότι για σταθερό όγκο δεδομένων, καθώς αυξάνεται ο αριθμός των νημάτων, ο χρόνος απόκρισης αυξάνεται αντίστοιχα. Αυτό μπορεί να αποδοθεί στο συνολικό κόστος (total_cost) των λειτουργιών που αυξάνεται με τον αριθμό των νημάτων, καθώς το σύστημα πρέπει να διαχειριστεί περισσότερες ταυτόχρονες διεργασίες.

❖ Readwrites 50%-50%:

Συνολικό Κόστος (sec)

	Συνολικό κόστος READ (sec)			Συνολικό κόστος WRITE (sec)		
	Νήματα			Νήματα		
Δεδομένα	20 Νήματα	40 Νήματα	60 Νήματα	20 Νήματα	40 Νήματα	60 Νήματα
40000	5	29	29	19	39	41
50000	9	37	38	14	37	20
60000	15	20	53	16	25	53
70000	20	40	45	19	40	56
80000	21	36	59	29	40	77
90000	17	42	49	20	53	57

Εικόνα 68



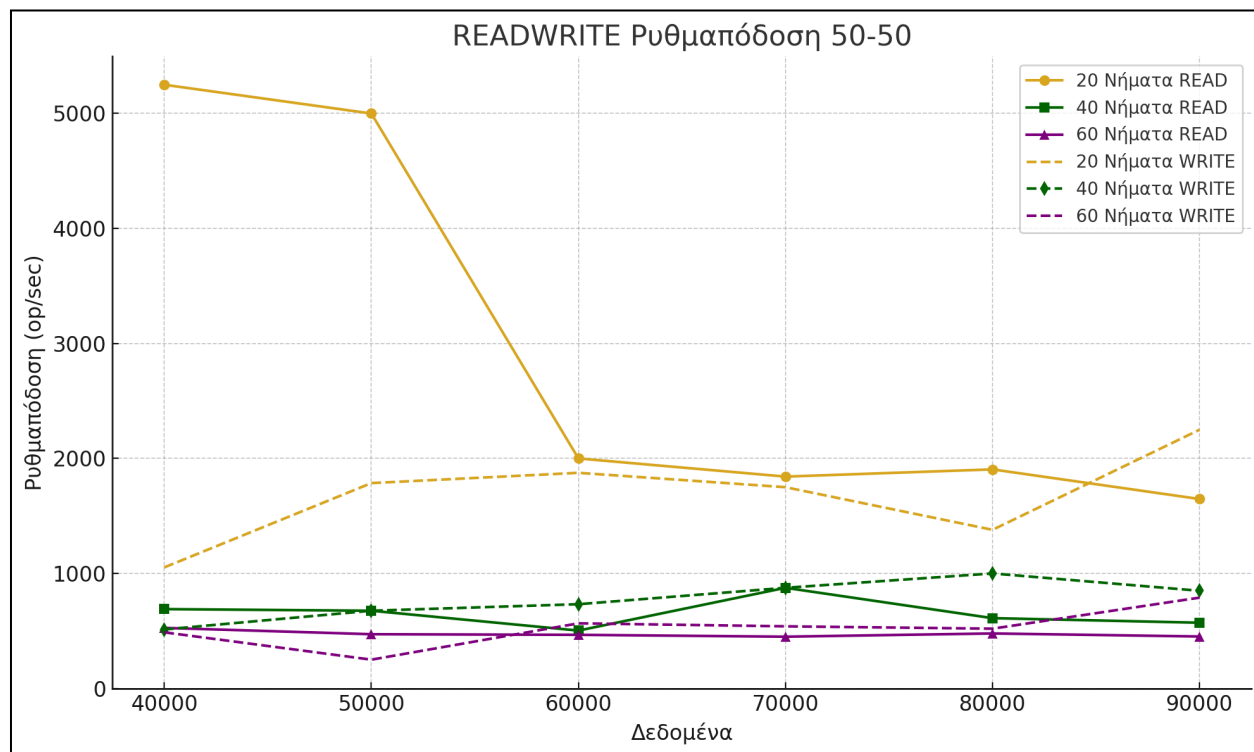
Εικόνα 69

Το παραπάνω γράφημα απεικονίζει το συνολικό κόστος σε δευτερόλεπτα για λειτουργίες read και write, για ποσοστό 50% read και 50% write. Σε διάφορους αριθμούς νημάτων (20, 40 και 60) το γράφημα υποδεικνύει μια γενική αύξηση του συνολικού κόστους καθώς αυξάνεται ο όγκος των δεδομένων, γεγονός που είναι ένα προβλέψιμο αποτέλεσμα, καθώς περισσότερα δεδομένα απαιτούν συνήθως περισσότερο χρόνο για την επεξεργασία τους. Αυτό που ξεχωρίζει στην ανάλυση είναι ότι για οποιοδήποτε δεδομένο αριθμό νημάτων, το κόστος των λειτουργιών εγγραφής ξεπερνά σταθερά αυτό των λειτουργιών ανάγνωσης. Αυτό είναι κατανοητό, δεδομένου ότι οι λειτουργίες εγγραφής είναι πιο απαιτητικές σε πόρους, καθώς συχνά απαιτούν πρόσθετα βήματα.

Ρυθμαπόδοση (op/sec)

	Ρυθμαπόδοση READ (op/sec)			Ρυθμαπόδοση WRITE (op/sec)		
	Νήματα			Νήματα		
Δεδομένα	20 Νήματα	40 Νήματα	60 Νήματα	20 Νήματα	40 Νήματα	60 Νήματα
40000	5250	689,7	526,3	1052,6	512,8	487,8
50000	5000	675,7	471,4	1785,7	675,7	250
60000	2000	503	466	1875	732,4	566
70000	1842,1	875	450,0	1750	875	540
80000	1904,8	611,1	478	1379,3	1000	519,5
90000	1647,1	571,4	451,7	2250	849,1	789,5

Εικόνα 70



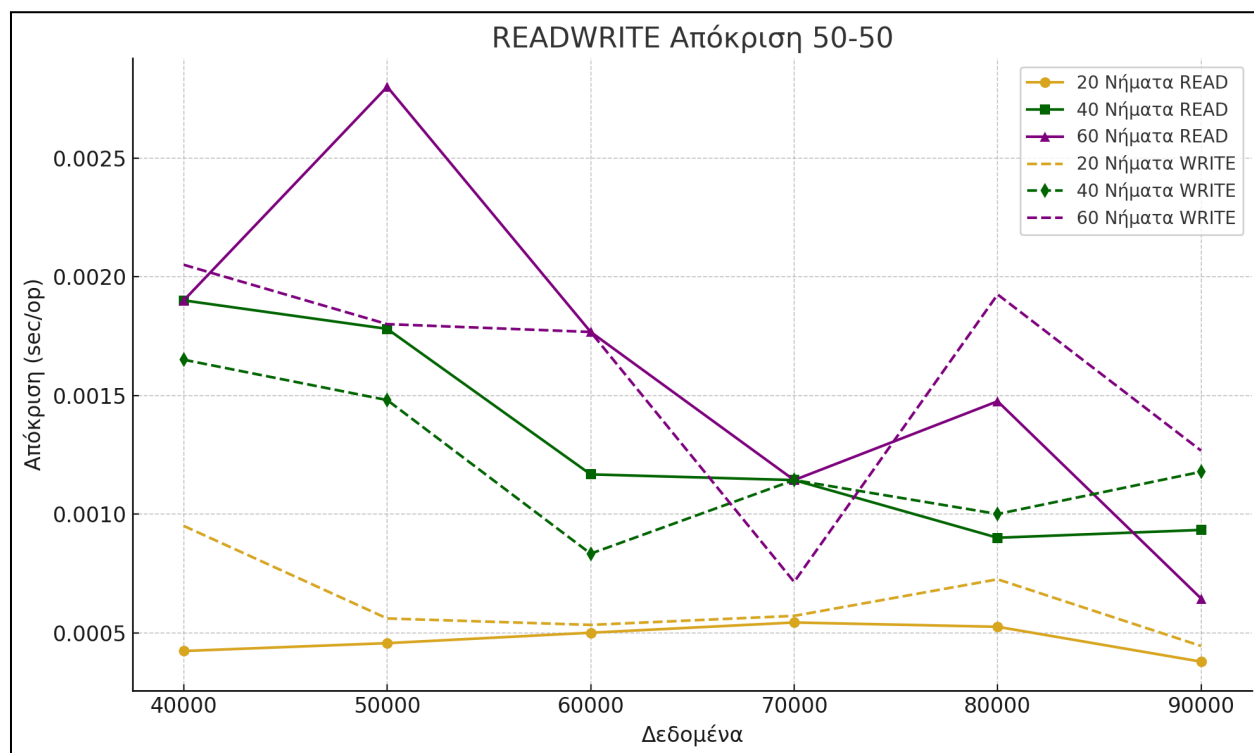
Εικόνα 71

Το παραπάνω γράφημα απεικονίζει τη ρυθμαπόδοση των λειτουργιών read και write του συστήματος αποθήκευσης, για ποσοστό 50% read και 50% write. Παρατηρείται ότι, για 40 και 60 νήματα, η απόδοση τόσο για τις λειτουργίες read, όσο και για τις λειτουργίες write παραμένει σχετικά σταθερή, γεγονός που μας οδηγεί στο συμπέρασμα ότι η μηχανή αποθήκευσης λειτουργεί ομαλά υπό αυτές τις συνθήκες. Ωστόσο, με 20 νήματα, παρατηρούμε μια απότομη πτώση, ειδικά στην απόδοση read στο όριο των 50.000, η οποία μπορεί να υποδηλώνει περιορισμούς στους πόρους του συστήματος ή μια λειτουργική συμφόρηση σε αυτόν τον χαμηλότερο αριθμό νημάτων. Επιπλέον, καθώς αυξάνεται ο αριθμός των νημάτων, παρατηρείται μια τάση μείωσης της απόδοσης, η οποία είναι ένα λογικό αποτέλεσμα.

Απόκριση (sec/op)

	Απόκριση READ (sec/op)			Απόκριση WRITE (sec/op)		
	Νήματα			Νήματα		
Δεδομένα	20 Νήματα	40 Νήματα	60 Νήματα	20 Νήματα	40 Νήματα	60 Νήματα
40000	0,000423	0,0019	0,0019	0,00095	0,00165	0,00205
50000	0,000456	0,00178	0,00028	0,00056	0,00148	0,0018
60000	0,0005	0,001167	0,001767	0,000533	0,000833	0,001767
70000	0,000543	0,001143	0,000114	0,000571	0,001143	0,000714
80000	0,000525	0,0009	0,001475	0,000725	0,001	0,001925
90000	0,000378	0,000933	0,000644	0,000444	0,001178	0,001267

Εικόνα 72



Εικόνα 73

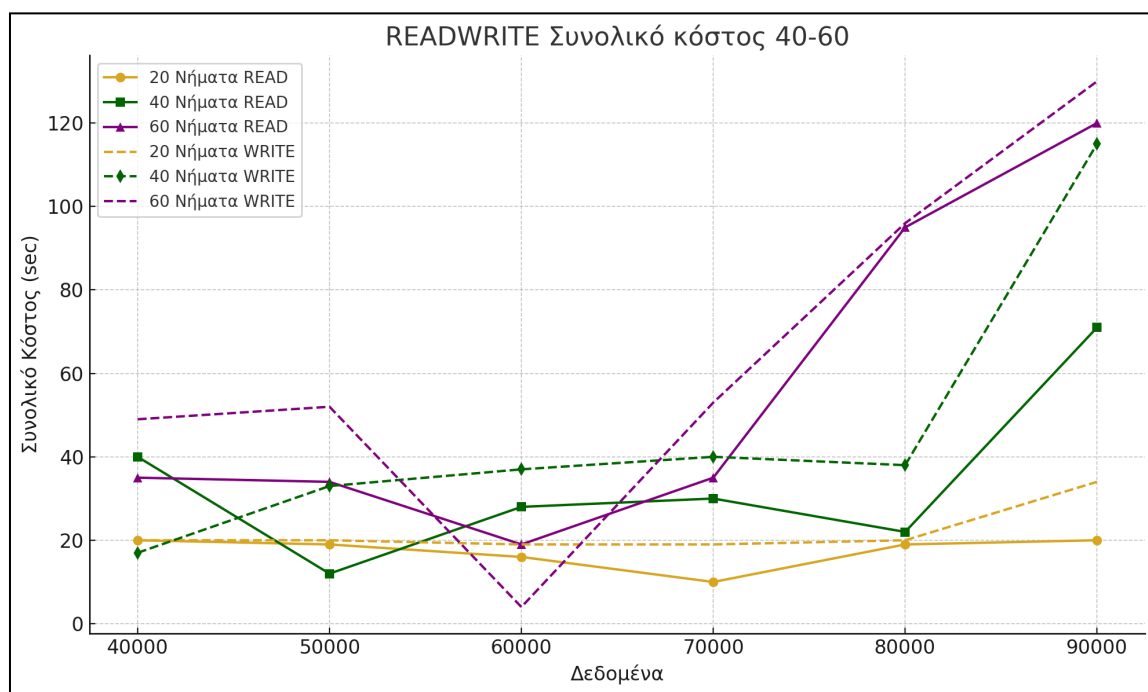
Το παραπάνω γράφημα απεικονίζει την απόκριση του συστήματος των λειτουργιών read και write του συστήματος αποθήκευσης, για ποσοστό 50% read και 50% write. Τυπικά, ο στόχος είναι η καθυστέρηση (κόστος) των εργασιών ανάγνωσης-εγγραφής να είναι σταθερή και να μην επηρεάζεται από την ποσότητα των δεδομένων που εισάγονται. Ωστόσο, καθώς αυξάνονται τα νήματα, αυξάνεται και η καθυστέρηση, επειδή η μηχανή αποθήκευσης αντιδρά ταχύτερα καθώς οι λειτουργίες κατανέμονται σε περισσότερα νήματα. Αυτό οδηγεί σε αύξηση του χρόνου απόκρισης, η οποία φαίνεται στο γράφημα όπου οι λειτουργίες write τοποθετούνται ψηλότερα από τις λειτουργίες read. Ο λόγος είναι ότι το συνολικό κόστος των λειτουργιών write είναι μεγαλύτερο από αυτό των λειτουργιών read, διογκώνοντας έτσι την αναλογία του χρόνου απόκρισης.

❖ Readwrites 40%-60%:

Συνολικό Κόστος (sec)

	Συνολικό κόστος READ (sec)			Συνολικό κόστος WRITE (sec)		
	Νημάτα			Νημάτα		
Δεδομένα	20 Νημάτα	40 Νημάτα	60 Νημάτα	20 Νημάτα	40 Νημάτα	60 Νημάτα
40000	20	40	35	20	17	49
50000	19	12	34	20	33	52
60000	16	28	19	19	37	4
70000	10	30	35	19	40	53
80000	19	22	75	20	38	87
90000	20	51	95	34	75	96

Εικόνα 74



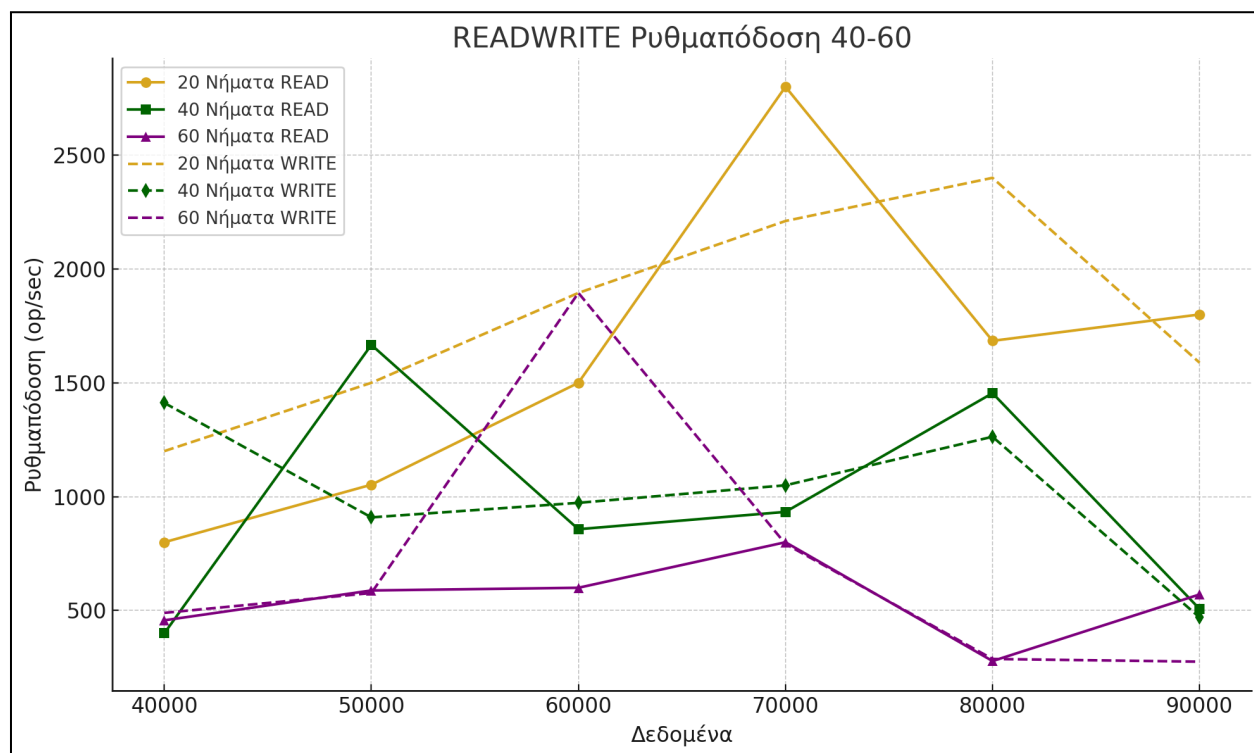
Εικόνα 75

Το παραπάνω γράφημα απεικονίζει το συνολικό κόστος σε δευτερόλεπτα για λειτουργίες read και write, για ποσοστό 40% read και 60% write. Με το συγκεκριμένο ποσοστό αναμένουμε μεγαλύτερη συχνότητα writes σε σύγκριση με reads. Παρατηρώντας έναν σταθερό αριθμό νημάτων, υπάρχει μια ανοδική τάση στις τιμές του συνολικού κόστους, η οποία είναι προβλέψιμη, διότι όσο αυξάνονται τα δεδομένα, αυξάνεται και ο χρόνος που χρειάζονται για να εκτελεστούν. Επιπλέον, είναι σημαντικό να σημειωθεί ότι, για σταθερό αριθμό νημάτων, οι writes είναι σταθερά ψηλότερες στο γράφημα σε σύγκριση με τις reads. Αυτό οφείλεται κυρίως στο ότι τα writes χρειάζονται περισσότερο χρόνο από τα reads, αλλά ένας σημαντικότερος λόγος είναι το ποσοστό τους. Τέλος, όταν συγκρίνουμε τα νήματα, όσο περισσότερα νήματα χρησιμοποιούμε στο σύστημά μας, τόσο υψηλότερο θα είναι το συνολικό κόστος του συστήματος ($\text{total_write_cost} + \text{total_read_cost}$), όπως φαίνεται από τις αύξουσες γραμμές στο γράφημα που αντιστοιχούν σε αυξανόμενο αριθμό νημάτων

Ρυθμαπόδοση (op/sec)

	Ρυθμαπόδοση READ (op/sec)				Ρυθμαπόδοση WRITE (op/sec)		
	Νήματα						
Δεδομένα	20 Νήματα	40 Νήματα	60 Νήματα	20 Νήματα	40 Νήματα	60 Νήματα	
40000	800	400	457,1	1200	1411,8	489,8	
50000	1052,6	1666,7	588,2	1500	909,1	576,9	
60000	1500	857,1	600	1894,7	973	1894,7	
70000	2800	933,3	800	2210,5	1050	792,5	
80000	1684,2	1454,5	278,3	2400	1263,2	287,4	
90000	1800	507	570,7	1588,2	469,6	275,5	

Εικόνα 76



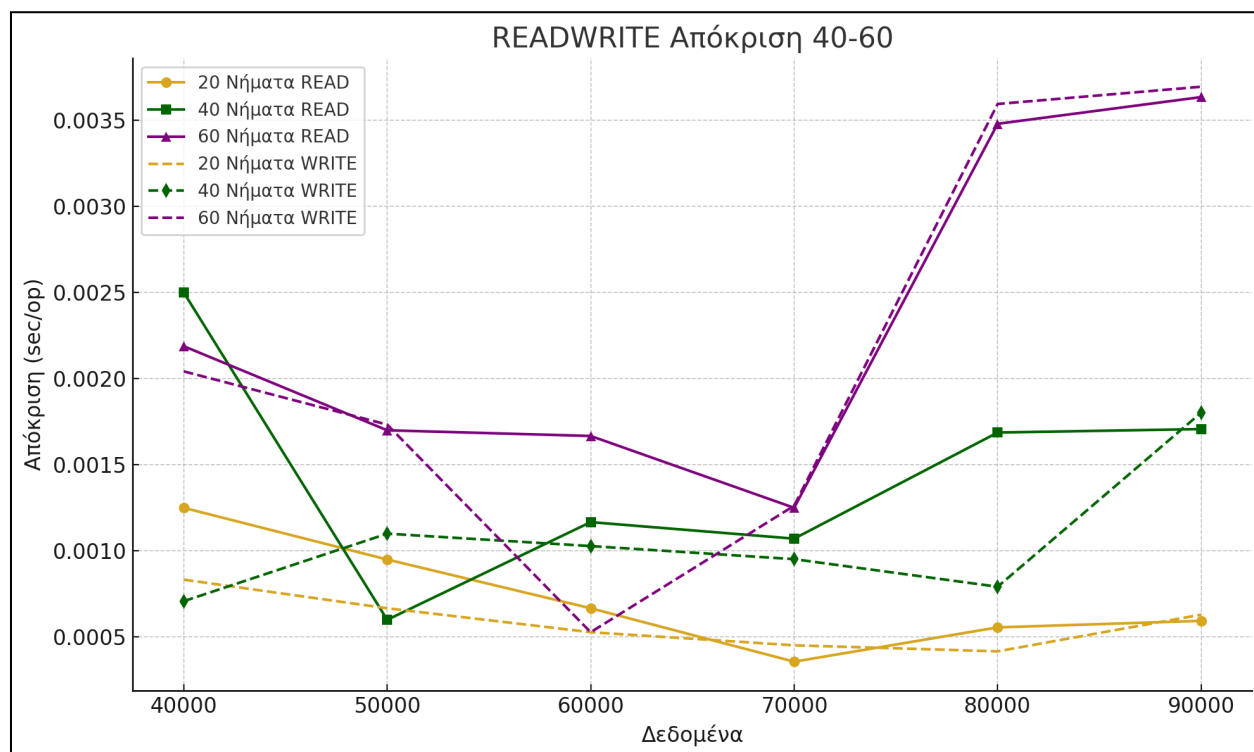
Εικόνα 77

Στο παραπάνω γράφημα παρουσιάζονται οι μετρήσεις απόδοσης, συγκεκριμένα οι λειτουργίες ανά δευτερόλεπτο, για λειτουργίες read και write, με ποσοστό 40% read και 60% write. Για τις ομάδες νημάτων 40 και 60, η απόδοση για τις λειτουργίες read και write παραμένει σχετικά σταθερή, γεγονός που υποδηλώνει ότι η απόδοση της μηχανής αποθήκευσης εκτελείται με σχετική ομαλότητα. Ωστόσο, στο επίπεδο των 20 νημάτων, παρατηρούμε αστάθεια, η οποία ενδεχομένως αποδίδεται σε διάφορους πόρους του συστήματος. Επιπλέον, είναι εμφανής μια συνολική μείωση των επιδόσεων καθώς αυξάνεται ο αριθμός των νημάτων, αυτό είναι λογικό, αφού με σταθερό όγκο δεδομένων και αυξημένη επιβάρυνση ανά νήμα, αναμένεται ο λόγος της ρυθμαπόδοσης να μειώνεται για τους αντίστοιχους writers και readers. Τέλος, για κάθε ομάδα νημάτων που έχουμε εφαρμόσει, οι λειτουργίες write παρουσιάζουν σταθερά χαμηλότερη απόδοση από τις λειτουργίες read στη γραφική αναπαράσταση, ένα λογικό αποτέλεσμα, δεδομένου ότι το συνολικό κόστος των λειτουργιών write είναι συνήθως υψηλότερο τόσο σε χρόνο όσο και σε αναλογία σε σύγκριση με τις λειτουργίες read.

Απόκριση (sec/op)

	Απόκριση READ (sec/op)			Απόκριση WRITE (sec/op)		
	Νήματα			Νήματα		
Δεδομένα	20 Νήματα	40 Νήματα	60 Νήματα	20 Νήματα	40 Νήματα	60 Νήματα
40000	0,00125	0,0025	0,002188	0,000833	0,000708	0,002042
50000	0,00095	0,0006	0,0017	0,000667	0,0011	0,001733
60000	0,000667	0,001167	0,000167	0,000528	0,001028	0,000528
70000	0,000357	0,001071	0,00125	0,000452	0,000952	0,001262
80000	0,000556	0,001687	0,003479	0,000417	0,001792	0,003594
90000	0,000594	0,001707	0,00363	0,00063	0,001802	0,003694

Εικόνα 78



Εικόνα 79

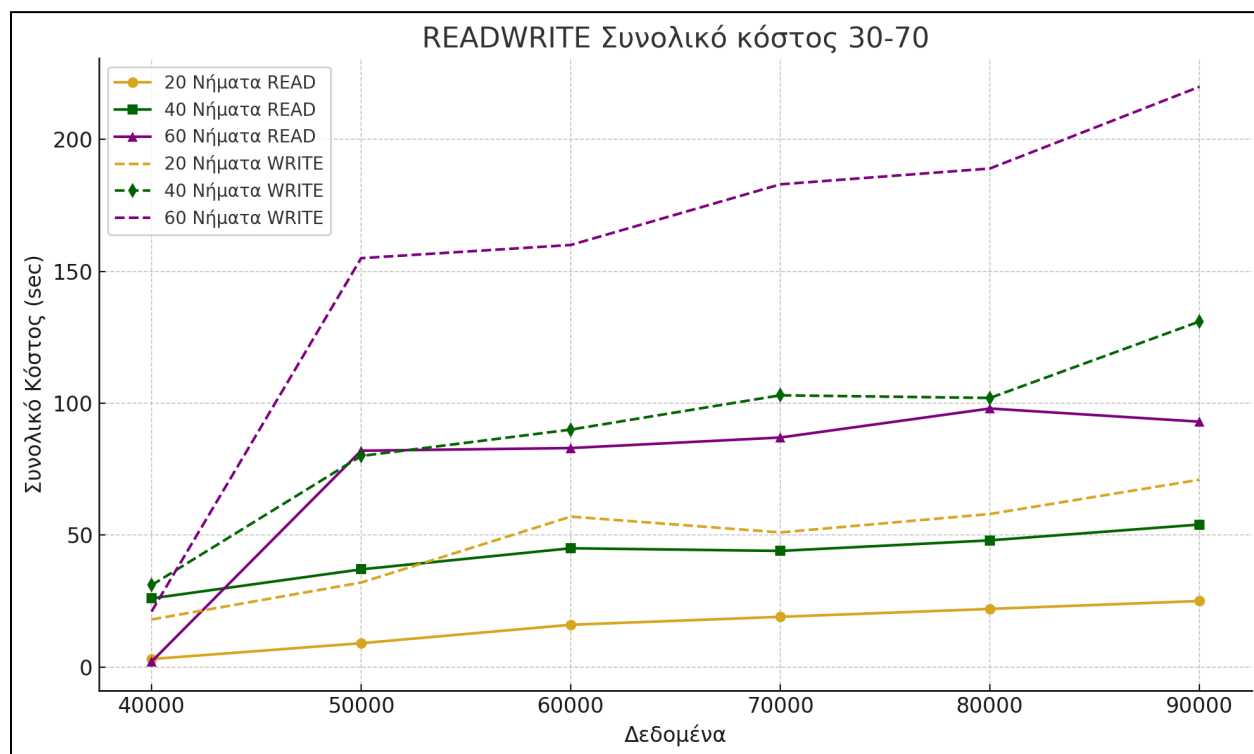
Στο παραπάνω γράφημα μετράμε την απόκριση του συστήματος μέσω λειτουργιών read και write, με μια διαμόρφωση 40% reads και 60% writes. Σε όλο το φάσμα, η αύξηση των νημάτων οδηγεί σε αύξηση της καθυστέρησης, υποδεικνύοντας ότι ενώ η μηχανή αποθήκευσης μπορεί να χειριστεί περισσότερες λειτουργίες ταυτόχρονα, υπάρχει συμβιβασμός όσον αφορά τους χρόνους ολοκλήρωσης των επιμέρους λειτουργιών. Συγκεκριμένα, με 20 νήματα, υπάρχει συνέπεια στην καθυστέρηση read, ωστόσο η καθυστέρηση write εμφανίζει ένα πιο ακανόνιστο μοτίβο, το οποίο θα μπορούσε να σχετίζεται με την πολυπλοκότητα των λειτουργιών write. Καθώς ο αριθμός των νημάτων κλιμακώνεται σε 40 και 60, η καθυστέρηση για τις reads παρουσιάζει φθίνουσα τάση, ενώ η καθυστέρηση write αυξάνεται σημαντικά στις 80.000 λειτουργίες, πιθανότατα λόγω του υψηλότερου φόρτου εργασίας που επιβάλλει η διαμόρφωση write 60%. Αυτή η εκτίναξη της καθυστέρησης write τονίζει το γεγονός ότι οι λειτουργίες write έχουν γενικά υψηλότερο κόστος από τις reads, το οποίο επιτείνεται όταν αντιπροσωπεύουν μεγαλύτερο ποσοστό των συνολικών λειτουργιών.

❖ Readwrites 30%-70%:

Συνολικό Κόστος (sec)

	Συνολικό κόστος READ (sec)			Συνολικό κόστος WRITE (sec)		
	Νήματα			Νήματα		
Δεδομένα	20 Νήματα	40 Νήματα	60 Νήματα	20 Νήματα	40 Νήματα	60 Νήματα
40000	3	26	2	18	31	21
50000	9	37	82	32	80	155
60000	16	45	83	57	90	160
70000	19	44	87	51	103	183
80000	22	48	98	58	102	189
90000	25	54	93	71	131	220

Εικόνα 80



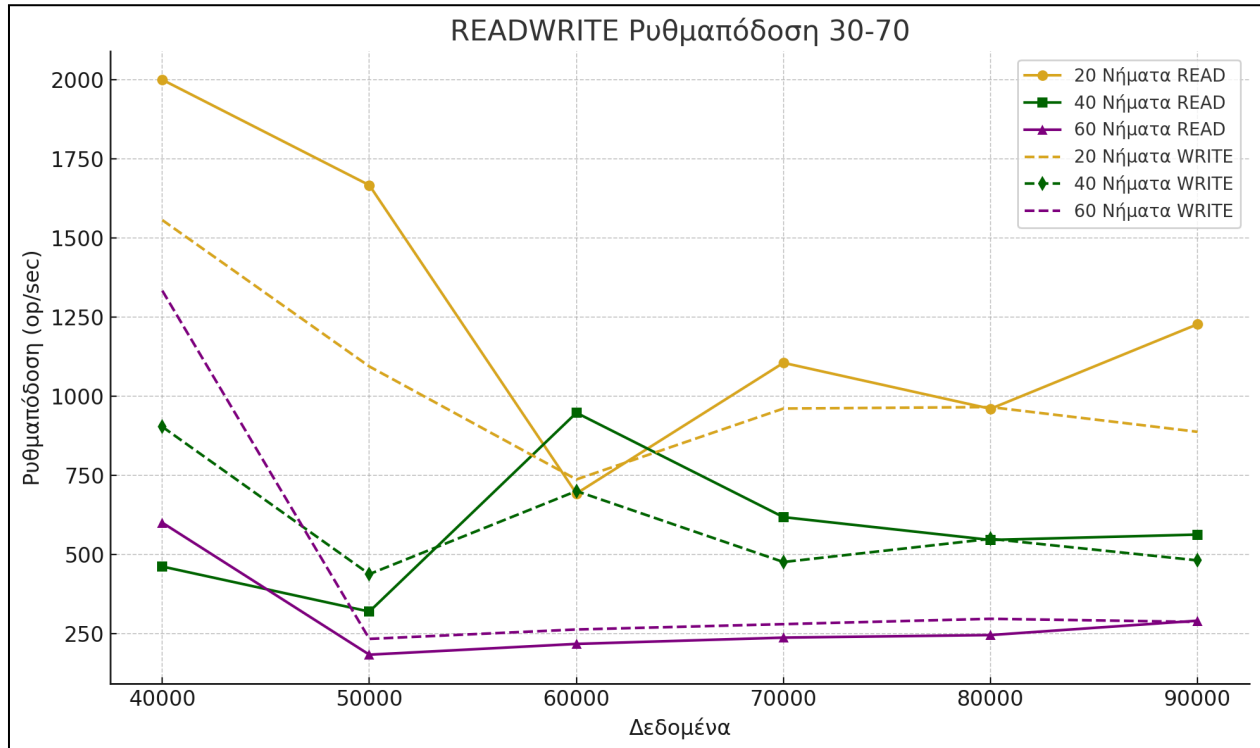
Εικόνα 81

Το παραπάνω γράφημα απεικονίζει το συνολικό κόστος σε δευτερόλεπτα για λειτουργίες read και write, για ποσοστό 30% read και 70% write. Μια τέτοια κατανομή μας οδηγεί στο να αναμένονται πολύ περισσότερες λειτουργίες write παρά read. Συγκεκριμένα, παρατηρούμε μια τάση αύξησης του συνολικού κόστους σε όλους τους αριθμούς νημάτων, η οποία είναι λογική λόγω του αυξανόμενου όγκου δεδομένων. Καθώς αυξάνονται τα δεδομένα, αυξάνεται και ο χρόνος για την επεξεργασία τους. Ειδικότερα, για σταθερό αριθμό νημάτων, το γράφημα τοποθετεί τις λειτουργίες write πάνω από τις λειτουργίες read. Αυτό προκύπτει από το γεγονός ότι η WRITE καταναλώνει περισσότερο χρόνο από την READ, αλλά ταυτόχρονα έχει πολύ μεγαλύτερο ποσοστό στο μείγμα λειτουργιών. Επιπλέον, όταν συγκρίνουμε ένα συνολικό κόστος χρήσης διαφόρων νημάτων σε διαφορετικά χρονικά σημεία, παρατηρούμε ότι το συνολικό κόστος ($\text{total_write_cost} + \text{total_read_cost}$) αυξάνεται επίσης πολύ.

Ρυθμαπόδοση (op/sec)

Δεδομένα	Ρυθμαπόδοση READ (op/sec)			Ρυθμαπόδοση WRITE (op/sec)		
	Νήματα			Νήματα		
	20 Νήματα	40 Νήματα	60 Νήματα	20 Νήματα	40 Νήματα	60 Νήματα
40000	4000	461,5	600	1555,6	903,2	1333,3
50000	1666,7	319,1	182,9	1093,8	437,5	225,8
60000	692,3	947,4	216,9	736,8	700	262,5
70000	1105,3	617,6	236,8	960,8	475,7	279,4
80000	960	545,5	244,9	965,5	549	296,3
90000	1227,3	562,5	290,3	887,3	480,9	286,4

Εικόνα 82



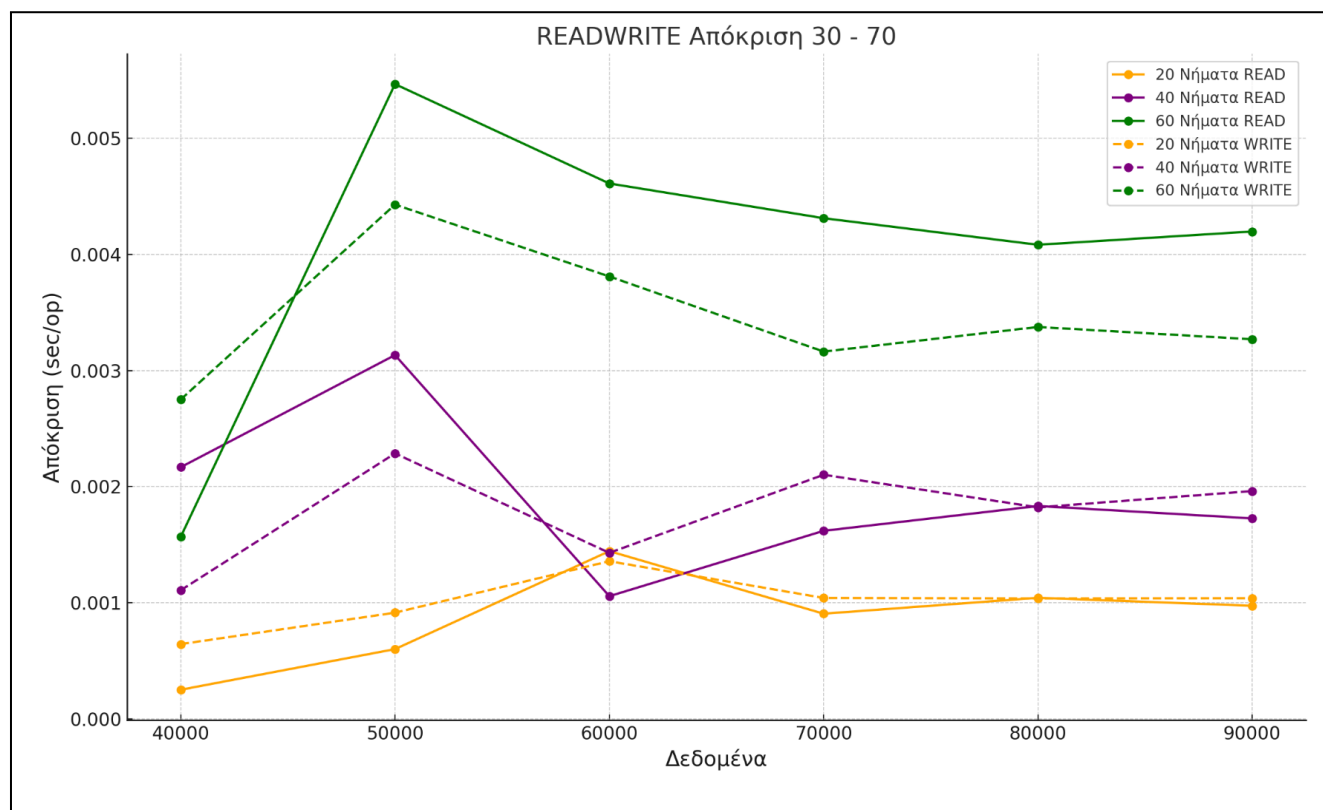
Εικόνα 83

Στο παραπάνω γράφημα παρουσιάζονται οι μετρήσεις απόδοσης, συγκεκριμένα οι λειτουργίες ανά δευτερόλεπτο, για λειτουργίες read και write, με ποσοστό 30% read και 70% write. Παρατηρούμε ότι η απόδοση, τόσο στις λειτουργίες read όσο και στις λειτουργίες write, διατηρεί μια σχετική σταθερότητα όταν τα σύνολα νημάτων είναι 40 και 60. Αυτό υποδηλώνει ότι η απόδοση της μηχανής αποθήκευσης στις λειτουργίες read και write υπό αυτές τις συνθήκες νημάτων θα είναι σχετικά ομαλή. Όμως όταν τα νήματα είναι 20, η διακύμανση είναι αξιοσημείωτη και μπορεί να αποδοθεί στην καταπόνηση άλλων πόρων του συστήματος, όπως η ταχύτητα του επεξεργαστή, η ποσότητα της μνήμης RAM ή η ταχύτητα του σκληρού δίσκου. Επιπλέον, καθώς αυξάνεται ο αριθμός των νημάτων, είναι εμφανής η μείωση της απόδοσης. Τέλος, η read παρουσιάζει υψηλότερα ποσοστά απόδοσης από την write, ανεξάρτητα από την ομάδα νημάτων που υλοποιείται. Αυτό είναι αναμενόμενο, αφού οι λειτουργίες WRITE απαιτούν γενικά περισσότερο χρόνο για να εκτελεστούν και εκτός αυτού, η WRITE είναι επίσης ένα ποσοστό του ποσοστού των λειτουργιών.

Απόκριση (sec/op)

Δεδομένα	Απόκριση READ (sec/op)			Απόκριση WRITE (sec/op)		
	Νήματα			Νήματα		
	20 Νήματα	40 Νήματα	60 Νήματα	20 Νήματα	40 Νήματα	60 Νήματα
40000	0,00025	0,002167	0,001567	0,000643	0,001107	0,00275
50000	0,0006	0,003133	0,005467	0,000914	0,002286	0,004429
60000	0,001444	0,001056	0,004611	0,001357	0,001429	0,00381
70000	0,000905	0,001619	0,004312	0,001041	0,002102	0,003163
80000	0,001042	0,001833	0,004083	0,001036	0,001821	0,003375
90000	0,000815	0,001778	0,003444	0,001127	0,002079	0,003492

Εικόνα 84



Εικόνα 85

Στο παραπάνω γράφημα παρουσιάζονται οι χρόνοι απόκρισης του συστήματος για μια λειτουργία ανάγνωσης-εγγραφής που έχει ρυθμιστεί σε 30% reads και 70% writes. Είναι προφανές ότι οι λειτουργίες write έχουν υψηλότερη καθυστέρηση από τις λειτουργίες read σταθερά σε όλους τους αριθμούς νημάτων. Αυτό οφείλεται στο γεγονός ότι ενώ η μία μπορεί να εκτελείται, η άλλη περιμένει στην ουρά. Επιπλέον, οι λειτουργίες write χρειάζονται αναγκαστικά περισσότερο χρόνο για να ολοκληρωθούν, γεγονός που επιδεινώνεται στη συνέχεια από το ότι αποτελούν την πλειοψηφία στο ποσοστό λειτουργιών. Η καθυστέρηση για τις λειτουργίες read και write αυξάνεται όσο αυξάνονται τα νήματα, γεγονός που υποδηλώνει ότι, ενώ το σύστημα αποθήκευσης μπορεί να διαχειριστεί μεγαλύτερη ποσότητα ταυτόχρονων εργασιών, αυτό γίνεται εις βάρος της ταχύτητας των μεμονωμένων λειτουργιών. Είναι ενδιαφέρον το γεγονός ότι βλέπουμε μια τεράστια αιχμή στην καθυστέρηση write σε

υψηλότερους αριθμούς για τη διαμόρφωση 60 νημάτων, γεγονός που δείχνει μια συμφόρηση για όταν βρίσκονται σε εξέλιξη πολλαπλές λειτουργίες write. Η γενική τάση που προκύπτει είναι ότι καθώς ο φόρτος εργασίας αυξάνεται, ειδικά για τις πλειοψηφικές write, η καθυστέρηση αυξάνεται.